C#
Kompendium

Unser Online-Tipp
für noch mehr Wissen ...



... aktuelles Fachwissen rund
um die Uhr – zum Probelesen,
Downloaden oder auch auf Papier.

www.InformIT.de

Arne Schäpers
Rudolf Huttary
Dieter Bremes

C#

Windows- und Web-Programmierung mit Visual Studio .NET

eBook
Die nicht autorisierte Weitergabe dieses eBooks an Dritte ist eine Verletzung des Urheberrechts!

KOMPENDIUM
Einführung
Arbeitsbuch
Nachschlagewerk

Markt+Technik Verlag

Bibliographische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:
Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1
05 04 03 02

ISBN 3-8272-6015-9

© 2002 by Markt+Technik Verlag,
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz H. Rauner, Gmund
Lektorat: Erik Franz, efranz@pearson.de
Korrektur: Brigitte Hamerski
Herstellung: Elisabeth Egger, eegger@pearson.de
Satz: Michael und Silke Maier, Ingolstadt
Druck und Verarbeitung: Bercker, Kevelaer
Printed in Germany

Inhaltsübersicht

	Vorwort	21
	Der Aufbau dieses Buchs	23
Teil 1	**Annäherung an .NET**	27
Kapitel 1	**Einführendes zur Philosophie von .NET**	29
Kapitel 2	**Das .NET-Puzzle aus der Vogelperspektive**	35
Kapitel 3	**Erste Schritte mit Visual Studio .NET**	49
Teil 2	**Die Sprache C#**	79
Kapitel 4	**Warum eine neue Sprache?**	81
Kapitel 5	**Grundlegende Konzepte**	95
Kapitel 6	**Anweisungen und Ausführungskontrolle**	123
Kapitel 7	**Datentypen**	163
Kapitel 8	**Objekte, Klassen, Vererbung**	241
Kapitel 9	**Ausnahmebehandlung**	339
Kapitel 10	**Ereignisbasierte Programmierung**	355
Kapitel 11	**Einsatz der .NET-Basisklassen**	361
Teil 3	**Windows-Anwendungen**	407
Kapitel 12	**Einführung in die Windows-Programmierung**	409
Kapitel 13	**Formulare**	439
Kapitel 14	**Grafik**	459
Kapitel 15	**Steuerelemente**	539
Kapitel 16	**Steuerelemente selbst implementieren**	633
Kapitel 17	**Dialogfelder**	673

Kapitel 18 **Zwischenablage und Drag&Drop** 725

Teil 4 **Verteilte Programme und Webanwendungen** 745
Kapitel 19 **XML** 747
Kapitel 20 **Einführung in verteilte Programme mit .NET** 809
Kapitel 21 **ASP.NET allgemein** 869
Kapitel 22 **Webanwendungen** 881
Kapitel 23 **Webdienste** 941

Teil 5 **Weiterführendes** 997
Kapitel 24 **Multithreading** 999
Kapitel 25 **.NET und COM** 1041
Kapitel 26 **Legacy Code und Windows API** 1069

Anhang
A **Erfahrungen mit der Installation** 1107
B **Einsatz des Debuggers** 1115
C **Internet-Protokolle** 1125
D **Was ist auf der CD-ROM?** 1139
Stichwortverzeichnis 1153

Inhaltsverzeichnis

Vorwort 21

Der Aufbau dieses Buchs 23

Teil 1 Annäherung an .NET 27

Kapitel 1 Einführendes zur Philosophie von .NET 29

1.1 Die Idee von Microsoft 29

Mainstream-Technologie als »Freeware« 30

Antwort auf Java 30

Fundament als Idee 31

.NET-Framework als Bündel 32

Kapitel 2 Das .NET-Puzzle aus der Vogelperspektive 35

2.1 Common Type System (CTS) 35

CTS vs. COM als Objektmodell 36

Struktur des .NET-Objektmodells 37

CLS 38

.NET-Klassenhierarchie als gemeinsame Typbibliothek 40

2.2 Common Language Runtime 41

Gemeinsame Typverwaltung 42

Codeverwaltung 46

Kapitel 3 Erste Schritte mit Visual Studio .NET 49

3.1 Windows-Anwendungen 49

Programmgerüst und Dateistruktur 50

Entwurf der ersten Windows-Anwendung 52

3.2 Konsolenanwendungen 56

Kommandozeilenparameter 59

3.3	**Webdienste**	61
	Implementation eines Dienstes	63
	Ein Webdienst-Client	65
	Aufruf des Webdienstes ServerTime	67
	Aufruf des Webdienstes ServerPassword	68
	Die ausführbare Datei	70
3.4	**Fehlersuche**	70
	Haltepunkte	70
3.5	**Konfigurationen**	74
	Debug und Release	74
	Aktionen beim Start von VS.NET	76
Teil 2	**Die Sprache C#**	79
Kapitel 4	**Warum eine neue Sprache?**	81
4.1	**Die J++-Affäre**	81
4.2	**Höchstrichterlich zur »Schichtung« verurteilt**	82
4.3	**Entwicklung aus Sicht von Microsoft**	83
	Verteilte Anwendungen	84
	Objektmodell ohne Sprache?	86
	Typsysteme unter Windows	87
Kapitel 5	**Grundlegende Konzepte**	95
5.1	**Starke Typisierung**	95
	Namensräume	96
	Modifizierer	101
5.2	**Benennungskonventionen**	104
	Bezeichnerwahl	105
	Schreibweise von Bezeichnern	110
	Reihenfolge bei der Strukturierung	111
5.3	**Integrierte vs. benutzerdefinierte Datentypen**	111

5.4 Wert- vs. Verweistypen . . . 112
Abstrakte Datentypen . . . 112
Gibt es denn Zeiger in C#? . . . 113
Heap und Stack . . . 114
Werttypen . . . 115
Verweistypen . . . 116
5.5 Alle Datentypen sind Klassen . . . 116
Ausführungsgeschwindigkeit . . . 117
Der Modifizierer sealed . . . 117
5.6 Strikte Typisierung . . . 118
.NET-Basisklassen, eine üppige Grundausstattung . . . 118
C# hat ein Heimspiel . . . 119
Typorganisation . . . 119
5.7 Zeiger ade . . . 120
Mit Automatikgetriebe . . . 120

Kapitel 6 Anweisungen und Ausführungskontrolle . . . 123
6.1 Sprachebenen . . . 123
6.2 Anweisungsfolgen und Threads . . . 124
Strukturierung von Anweisungsfolgen . . . 125
6.3 Kommentare . . . 127
Zeilenkommentare . . . 128
Blockkommentare . . . 128
Dokumentationskommentare . . . 131
6.4 Arten von Anweisungen . . . 142
Überlaufkontrolle – checked und unchecked . . . 142
6.5 Kontrollstrukturen . . . 143
Bedingte Verzweigung . . . 144
Schleifen . . . 147
Unbedingte Verzweigung, Sprünge . . . 155
Codebeispiel – Primzahlen aufzählen . . . 157

Kapitel 7 Datentypen 163
7.1 Integrierte Datentypen 163
Einfache Werttypen 165
Einfache Verweistypen 172
7.2 Komplexe Datentypen 182
Komplexe Werttypen 182
Komplexe Verweistypen 198
Typumwandlung 222
7.3 Variablen 228
Syntax 228
Geltungsbereich 231
Lebensdauer 232
Literale, Konstanten und readonly-Werte 236

Kapitel 8 Objekte, Klassen, Vererbung 241
8.1 Objekte vs. Klassen 242
Objektbegriff 242
Terminologie – Objekt, Objektvariable, Instanz 243
8.2 Klasse 244
Statischer vs. dynamischer Bereich 244
Syntax 245
Instanziierung – new-Operator und Konstruktor 251
8.3 Methoden 257
Allgemeines über Methoden 258
Überladen von Methoden 268
Überschreiben von Methoden 272
Virtuelle Methoden 275
Statische Methoden 277
Instanzmethoden 281
Konstruktoren 281
Destruktoren 286
Eigenschaften 295
Operatoren 304

8.4 Vererbung 315
Arten der Vererbung 316
8.5 Schnittstellen 322
COM-Schnittstellen 322
C#-Schnittstellen 322
Syntax 323
Umgang mit .NET-Schnittstellen 326
8.6 Abstrakte Klassen 331
Abstrakte Klassen und Schnittstellen in C++ 331
Abstrakte Klassen und Schnittstellen in C# 331
Schnittstellen vs. abstrakte Klassen 335

Kapitel 9 Ausnahmebehandlung 339
9.1 Der Mechanismus 340
Sprachübergreifende Fehlerbehandlung 341
Typsicherheit 341
Das Prinzip 341
Ausnahmeobjekte 344
9.2 Syntax 345
try 345
throw 345
catch 346
finally 347
9.3 .NET-Ausnahmeklassen 349
Codebeispiel – Ausnahmen und Ausnahmeklassen 351
9.4 Ausnahmen richtig eingesetzt 354

Kapitel 10 Ereignisbasierte Programmierung 355
10.1 Ereignisorientiertes Programmdesign 355
10.2 Codebeispiel – Monitor 356
Übung 359

Kapitel 11 Einsatz der .NET-Basisklassen . . . 361
11.1 Strings . . . 361
System.String . . . 363
StringBuilder . . . 372
11.2 Collection-Klassen . . . 375
Schnittstellengrundlage von Auflistungsklassen . . . 376
Eigene Auflistungsklassen . . . 377
11.3 Dateien . . . 381
Dateisystem . . . 381
Dateizugriff, Streams . . . 390

Teil 3 Windows-Anwendungen . . . 407

Kapitel 12 Einführung in die Windows-Programmierung . . . 409
12.1 Das Codegerüst stellt sich vor . . . 412
Codegerüst anlegen, kompilieren und starten . . . 412
Codegerüst mit dem Designer erweitern . . . 416
Maus . . . 418
Timer . . . 427
Tastatur . . . 430

Kapitel 13 Formulare . . . 439
13.1 Abstammung und Erbmasse . . . 439
Formularobjekte . . . 442
Bildlaufleisten des Formulars verwenden . . . 455

Kapitel 14 Grafik . . . 459
14.1 GDI+ . . . 459
14.2 Das Prinzip – Zeichnen auf Anforderung . . . 460
OnPaint() vs. Paint-Behandlung . . . 461
14.3 Für das Zeichnen benötigte Strukturen und Klassen . . . 463
Koordinatensysteme . . . 463
Positionen, Abmessungen, Rechtecke . . . 468

Farben – die Color-Struktur 484
Schreibzeug – Stift und Pinsel 494

14.4 Zeichnen 509
Grafikkontext – Graphics-Klasse 509
Textausgabe – DrawString() 527

Kapitel 15 Steuerelemente 539

15.1 Allgemeines 540
Tipps und Tricks beim Formularentwurf 541
Eigenschaften 546
Methoden 551
Ereignisse 553
Koordinatensystem für Abmessungen und Position 555
Generieren und Einbinden von Steuerelementen zur Laufzeit 556

15.2 Wichtige Steuerelemente der Toolbox 563
Button, CheckBox, RadioButton 563
Label und TextBox 568
ListBox und ComboBox 571
Analoganzeigen: ScrollBar, ProgressBar, TrackBar 578
Panel, GroupBox, Splitter 585
TreeView 588
Menüs 600
Bildliste (ImageList) 613
Symbolleiste – ToolBar 618
Statusleiste 624
Codebeispiel – Symbol- und Statusleiste 626

Kapitel 16 Steuerelemente selbst implementieren 633

16.1 Wohin mit dem Quelltext für die Steuerelementklasse? 633
Als neues Element zum bestehenden Projekt hinzufügen 634
Steuerelementbibliothek 635

16.2 Visuelles Steuerelementdesign unter .NET 635
Codeansicht, Entwurfsansicht, Laufzeitansicht 636

16.3 Implementierung eines eigenen Benutzersteuerelements . . . 638
Zum Eingewöhnen . . . 640
Codebeispiel – Knob . . . 642
16.4 Bestehende Steuerelemente modifizieren . . . 656
Überlistung des Designers . . . 657
Codebeispiel – MemTextBox . . . 658
16.5 Eigene Steuerelemente verfügbar machen . . . 665
Als neues Element in das Projekt kopieren . . . 665
Steuerelemente in Steuerelementbibliotheken zusammenfassen . . . 665
16.6 Eigene Benutzersteuerelemente weiter ableiten . . . 669
Ausgangssituation . . . 670

Kapitel 17 Dialogfelder . . . 673
17.1 Modale Dialoge . . . 674
MessageBox.Show() . . . 674
Standarddialoge . . . 676
Eigene modale Dialoge . . . 692
17.2 Nicht-modale Dialoge . . . 698
Codebeispiel – Suchen/Ersetzen-Dialog . . . 699
17.3 Gestaltung . . . 703
TabControl-Steuerelement . . . 704
17.4 Formularvererbung . . . 708
Spezialisierung der Basisklasse . . . 709
Modifikation der Basisklasse . . . 709
Mögliche Implementierungsstrategien . . . 710
Codebeispiel – Eigenschaftsdialog mit Vererbung . . . 713

Kapitel 18 Zwischenablage und Drag&Drop . . . 725
18.1 Zwischenablage . . . 725
Steuerelemente mit eigener Zwischenablagenfunktionalität . . . 725
Die Zwischenablage direkt ansprechen . . . 727

18.2 Drag&Drop ... 736
Ablauf der Operation ... 736
Zusammenfassung ... 742

Teil 4 Verteilte Programme und Webanwendungen ... 745

Kapitel 19 XML ... 747
19.1 Einführung in XML ... 747
Was ist XML? ... 747
XML, XHTML, XSL, etc.: Wie hängt was zusammen? ... 748
XML-Syntax ... 750
19.2 XSLT und XPath einsetzen ... 756
Codebeispiel – Die erste Transformation ... 757
Codebeispiel – Vorlagen einsetzen ... 760
Benannte Vorlagen, Parameter und Variablen einsetzen ... 762
Sortieren und Nummerieren ... 765
Bedingungen und Prädikate einsetzen ... 769
Elemente und Attribute einfügen ... 773
Gruppieren und Zwischensummen durch Rekursion bilden ... 775
19.3 XML-Klassen in .NET ... 783
XML erzeugen und transformieren ... 784
Validieren von XML ... 801

Kapitel 20 Einführung in verteilte Programme mit .NET ... 809
20.1 Beweggründe für die Schichtenaufteilung ... 809
Zweischichtsysteme ... 810
Dreischichtsysteme ... 811
n-Schichtsysteme ... 812
20.2 Technologien für Mehrschichtsysteme unter .NET ... 813
Technologien der Datenbankschicht ... 813
Technologien der Mittelschicht ... 813
Technologien zur Client-Kommunikation ... 814

20.3 Internet-Programmierung mit .NET 814
IE-Hosting 815
DNS-Abfragen mit der DNS-Klasse 820
Whois-Abfragen mit der TcpClient-Klasse 822
Mails senden mit den Klassen SmtpMail und MailMessage 826
Mails empfangen mit der TcpClient-Klasse 833
Ein Screen Scraper mit den Klassen WebClient und Regex 841
Eine automatisierte Auto-Suche mit der WebClient-Klasse 845
Eine asynchrone Auto-Suche mit den Klassen WebClient und der WebRequest 850
Austauschbare Protokolle mit den Klassen WebRequest und WebResponse 863

Kapitel 21 ASP.NET allgemein 869
21.1 Wie kommt die Endung .NET an ASP? 869
21.2 Bearbeitung einer HTTP-Anfrage in ASP.NET 871
Funktionsweise der HTTP-Pipeline 872
Objekte der HTTP-Pipeline 873
Konfiguration der HTTP-Pipeline 874
21.3 Konfiguration der IIS 877
Dateinamenerweiterung registrieren 877
Virtuelle Verzeichnisse anlegen und Anwendungen anmelden 878

Kapitel 22 Webanwendungen 881
22.1 Das Web ohne Web Forms 881
Grundlagen von HTML 881
HTML in praktischen Beispielen 884
22.2 Web Forms-Seiten mit HTML-Serversteuerelementen 899
22.3 Web Forms-Seiten mit Webserver-Steuerelementen 910
22.4 Web Forms-Seiten und Datenbindung 915
Web Forms-Seiten und Serversteuerelemente an Daten binden 916
Spezielle Webserver-Steuerelemente für Ausgaben in Listenform 917
Das DataGrid-Steuerelement 919
Das Repeater-Steuerelement 924
Das DataList-Steuerelement 927

Kapitel 23 Webdienste ... 941
23.1 Akronyme, Akronyme 942
23.2 Codebeispiel – Mini-Webdienst ... 943
Der Server ... 943
Der Client ... 944
Debugging von Webdiensten ... 952
23.3 Codebeispiel – Asynchrone Methoden und Timeouts ... 955
23.4 Einfaches Zustand halten, Caching und mehrere Webdienste in einem Projekt ... 961
Codebeispiel – Zustand halten ... 961
Caching ... 965
Mehrere Webdienste in einem Projekt implementieren ... 968
23.5 Fortgeschrittene Techniken für das Caching ... 971
23.6 Webdienste erweitern ... 981
SOAP-Erweiterungen verwenden ... 981
SOAP-Header verwenden ... 984
Codebeispiel – SOAP-Header und SOAP-Erweiterungen ... 985

Teil 5 Weiterführendes ... 997

Kapitel 24 Multithreading ... 999
24.1 Einfache Beispiele ... 999
Grundvoraussetzungen ... 999
Die Klasse Thread im Überblick ... 1002
Codebeispiel – Threads ohne eigene Klasse ... 1003
24.2 Synchronisation für Variablenzugriffe ... 1004
lock ... 1005
Deadlocks ... 1007
Wettrennen (Race) ... 1008
24.3 Steuerelemente ... 1009
24.4 Synchrone und asynchrone Rückrufe ... 1010
Delegaten ... 1011
Synchronisation über Steuerelemente ... 1012

Synchronisation auf neuen Wegen 1013
Asynchron ausgeführte Methoden 1014
24.5 Interaktion mit Threads 1016
Die Beispielklasse Recognizer 1017
Entkoppelung 1019
24.6 Codebeispiel – interaktive Threads 1020
Der Programmablauf 1029
24.7 Weitere Synchronisationsobjekte 1030
Codebeispiel –Threads mit Work-Liste 1030
Events 1034
Semaphoren 1037

Kapitel 25 .NET und COM 1041
25.1 Die COM-Technologie im Überblick 1041
Konzepte für wiederverwendbaren Code 1041
COM-Metadaten 1043
Die Grenzen von COM 1044
25.2 .NET-Hüllklassen für COM 1047
Codebeispiel – Anlegen von RCWs 1047
Der Einsatz 1050
Fehlerprüfungen 1051
ActiveX-Steuerelemente 1053
Untersuchungen mit ILDASM 1058
COM ohne Stellvertreterklassen und RCWs 1059
25.3 .NET-Komponenten in COM 1060
Voraussetzungen 1060
Codebeispiel – .NET-DLL als COM-Komponente 1061
Grenzen 1067

Kapitel 26 Legacy Code und Windows API 1069
26.1 DLL-Aufrufe 1069
Deklaration externer Routinen 1070
Ladezeitpunkte und Fehlerprüfung 1074

Codebeispiel – Ladezeitpunkt einer DLL 1074
Fehlerprüfungen 1076
26.2 Marshalling 1078
MarshalAs und UnmanagedType 1078
Strings und Referenzparameter 1079
Strukturtypen als Parameter 1081
Marshalling direkter Funktionsergebnisse 1089
26.3 Rückrufe 1090
26.4 Codebeispiel – Herunterfahren von Windows 1093
26.5 Nicht gesicherter Code 1097
Voraussetzungen 1098
Syntax und Geltungsbereiche 1099
Möglichkeiten und Grenzen 1101

Anhang 1105

Anhang A Erfahrungen mit der Installation 1107
A.1 Das Entwicklungssystem 1107
Internetanbindung und Service Packs 1108
Zeitbedarf und Umfang der Installation 1109
Konflikte mit anderen Programmen 1109
Windows Component Update 1111
A.2 Die Clientseite: .NET Framework 1112

Anhang B Einsatz des Debuggers 1115
B.1 Grundfunktionen des Debuggers 1115
B.2 Fehlersuche in DLLs 1120
Fehlersuche mit Delphi 1120
Fehlersuche mit MSVC 1123

Anhang C Internet-Protokolle 1125
C.1 HTTP 1125
C.2 TCP 1128

C.3 **IP** 1130
C.4 **Der Weg durchs Netz** 1132
C.5 **Der Protokollstapel** 1136
Der TCP/IP-Stapel 1137
Der Protokollstapel in .NET 1138

Anhang D **Was ist auf der CD-ROM?** 1139
D.1 **.NET Framework SDK** 1139
D.2 **Übersicht über die Beispielprojekte** 1140
Installation 1140
Teil 2 – Die Sprache C# 1141
Teil 3 – Windows-Anwendungen 1144
Teil 4 – Verteilte Programme und Web-Anwendungen 1148
Teil 5 – Weiterführendes 1150

Stichwortverzeichnis 1153

Vorwort

Oh Gott, die Microsoft-Marketingabteilung hat wieder mal was Neues erfunden.

Das dürfte der erste Gedanke Vieler gewesen sein – angesichts der ersten Pressemitteilungen aus Redmond: Hätte man dort die Jubelvokabeln herausgestrichen, wäre schließlich nur noch der Briefkopf übrig geblieben. Eine neue Programmiersprache also, mit einem DLL-Hintertürchen wie bei Visual Basic, damit man auch »richtige« Programme schreiben kann, eingebettet in ein nebulöses Laufzeitsystem, das mit Zwischencode hantiert, aber – um Gottes Willen – ja kein Java-Klon sei soll. Dazu natürlich eine völlig neue Philosophie, die alles bisher Dagewesene ... Und wer nicht erst seit letzter Woche programmiert, hat spätestens an diesem Punkt den eingangs zitierten Satz als Stoßgebet zum Himmel geschickt.

Was tatsächlich draus geworden ist: Ein Quantensprung in der Größenordnung des Übergangs von DOS zu Windows – nicht auf der Benutzer-, sondern endlich, endlich einmal wieder auf der Programmiererseite.

Wie dieser Quantensprung im Einzelnen aussieht, erfahren Sie in diesem Buch. Die folgenden rund 1100 Seiten verstehen sich weder als Wiederholung der von Microsoft gelieferten – vorerst noch recht spärlichen – Online-Dokumentation, noch wenden sie sich an Leute, die seit den frühen Beta-Tests von Visual Studio .NET nur noch in C# träumen: Die Schwerpunkte liegen eindeutig in der Vermittlung des notwendigen theoretischen und praktischen Hintergrundwissens für eigene Entwicklungen in den wichtigsten Bereichen und Technologien, für die .NET entwickelt wurde:

- Windows-Anwendungen
- Webanwendungen
- Objektorientierte Programmierung, Komponentendesign
- Effizientes Programmdesign auf der Basis der erstaunlich intelligenten Werkzeuge der Entwicklungsumgebung Visual Studio .NET.

Nicht nur der Gesamtaufbau des Buchs an sich, sondern auch die Themen der einzelnen Kapitel sind in sich so strukturiert, dass sie vom Einfachen beginnend ins Komplizierte führen. Fortgeschrittene EinsteigerInnen ebenso wie langjährige ProgrammiererInnen finden so gleichermaßen in jedem der fünf Buchteile Inhalte, die für ihre Perspektive geschrieben sind. Blicke über den Tellerrand lohnen sich natürlich für beide Gruppen – sei es, um einen

Eindruck von der faszinierenden Tiefe und Vielfalt der in .NET integrierten Technologien zu erhalten oder sei es »nur« zur Einordnung auf das gar nicht so triviale Grundlagenwissen für die schnelle Migration.

Eine ähnliche Bandbreite hat auch das umfangreiche, dem Buch auf CD beiliegende Beispielmaterial. Es gestattet das interaktive Studium der dargestellten Programmiertechniken und eignet sich auch hervorragend als Testplattform für weiterführende eigene Experimente in die jeweilige Richtung.

Die Voraussetzungen, die Sie mitbringen müssen, sind schnell skizziert:

- Grundlegende Kenntnisse in irgendeiner Programmiersprache – C#, Visual Basic, C, C++, Delphi, Java – und etwas praktische Erfahrung in der Programmierung damit.
- Visual Studio .NET auf Ihrer Festplatte und ein vernünftiges System mit Windows XP oder Windows 2000. Eine englischsprachige Version dieses Produkts können Sie auch als 60-Tage-Trialware gegen Entrichtung einer Schutzgebühr von 10 $ direkt bei Microsoft bestellen. Näheres hierzu unter `http://msdn.microsoft.com/vstudio/productinfo/trial.asp`
- Interesse an der rein objektorientierten Programmierung mit einer äußerst schlanken und ballastfreien Programmiersprache, einer erstaunlich stabilen Entwicklungsumgebung für eine überzeugende Plattform, die ohne Zweifel in kürzester Zeit wie ein Lauffeuer den gesamten Erdball erobert haben wird.

Erfahrungen in der Programmierung mit der Win32-API sind übrigens nicht notwendig; sie wird von der .NET-Klassenhierarchie vollständig verkapselt und schimmert eigentlich nur noch konzeptuell an vielen Stellen durch. Das ist einer der wirklich interessanten Punkte an .NET und C#. Ein radikal neuer Anfang ohne den für Microsoft-Produkte gewohnten bitteren Beigeschmack tonnenschwerer Altlasten.

In diesem Sinne wünschen wir Ihnen viel Erfolg bei der Eroberung der schönen neuen Welt. Sie werden es nicht bereuen. Wir zumindest können Ihnen versichern, dass unsere eigene anfängliche Skepsis nach kurzer Zeit schon in helle Begeisterung für das Produkt umgeschlagen ist.

München, August 2002

Die Autoren

P.S.: Sollten Sie als Leser Kritik, Lob oder gar Verbesserungsvorschläge für kommende Auflagen haben: Ihre E-mail an `Rudolf.Huttary@pearson.de` wird direkt an die Autoren weitergeleitet.

Der Aufbau dieses Buchs

Die fünf Teile dieses Buchs führen zwar vom Grundlegenden zum Komplexen, sowohl in sich als auch von ihrer Anordnung her. Dennoch bauen sie nicht direkt aufeinander auf. Wer sich durch Multithreading oder verteilte Anwendungen kämpft, ohne die Syntax von C# zu beherrschen und zuvor wenigstens etwas mit der Windows-Programmierung und der .NET-Klassenhierarchie geflirtet zu haben, dem gebührt natürlich trotzdem eine Tapferkeitsmedaille (die, wie die meisten Ehrungen dieser Art, postum verliehen wird).

Teil 1 – Annäherung an .NET

Dieser Teil ist bewusst kurz gehalten, und selbst für ganz besonders Eilige auf jeden Fall Pflicht, zumal wenn noch keine echte Berührung mit .NET stattfand. Sie finden darin: Was sich Microsoft bei .NET gedacht hat, woraus .NET besteht, wie man mit Visual Studio .NET einfache Anwendungen (Konsolen- Windows-, Web-Anwendungen) erstellt und wie sich die grundlegende Fehlersuche in dieser Entwicklungsumgebung gestaltet. Der Rest dieses Buchs baut auf diese in den ersten 60 Seiten vorgestellten Grundlagen und Techniken.

Teil 2 – Die Sprache C#

Geht nach einem Abriss der in diesem Fall zum Verständnis wirklich förderlichen Vorgeschichte in der gebotenen Gründlichkeit auf die Sprache C# als solche ein: Es geht um Syntax, Semantik, Datentypen, Kontrollstrukturen, Klassen, Objekte, Methoden, Schnittstellen, Delegaten ebenso wie um Typsicherheit, objektorientierte Programmierung, Ausnahmebehandlung und natürlich Vererbungsmechanismen. Auch wenn die Beispielebene im Wesentlichen bei der Konsole bleibt, werden alle Konzepte auch und vor allem mit Blick auf die Windows- und Web-Programmierung vorgestellt. Dies gilt insbesondere für die am Ende des Teils diskutierten, zentralen Klassen der .NET-Klassenhierarchie: String- und Sammelklassen sowie Klassen für Dateioperationen usw.

Der typische Leser wird dieses kommentierte Grundgesetz der Sprache erst einmal überfliegen, später immer wieder darauf zurückkommen, und bei jedem dieser Rückgriffe ein wenig mehr darüber staunen, wie viel an Überlegung – fast möchte man sagen: Genie – in jedem Detail des verblüffend stimmigen Gesamtkonzepts steckt. Beispielhaft sei hier angeführt, dass `int` in C# eine Klasse ist, was man erst einmal als netten Formalismus abtun wird. Wie weitreichend die Folgen sind, was für eine noch nie da gewesene

Geschlossenheit sich dadurch ergibt, *kann* auch mit allen Wassern gewaschenen ProgrammiererInnen – gerade denen – von vornherein gar nicht klar sein.

Dass es hier nicht ums Herunterbeten der einzelnen Sprachelemente geht, zeigen nicht zuletzt die 28 Beispielprogramme für die zentralen Themen dieses Buchteils.

Teil 3 – Windows-Anwendungen

Beginnt mit der Erläuterung des Codegerüsts das Visual Studio .NET aktiv im Designer verwaltet (»Vorlagenmakros« wie bei Microsoft C/C++ haben ausgedient), geht dann auf die grundlegende Interaktion mit dem Benutzer (Maus, Tastatur sowie Timer) ein, beschäftigt sich als Nächstes ausführlich mit Formularen nebst ihrer ausgesprochen umfangreichen Erbmasse und schließlich mit dem, was ein Programm dem Benutzer zu sagen hat: Zeichenoperationen über GDI+. Weiter geht es mit einer Referenz für die eingebauten Steuerelemente von Windows, der Implementation und Ableitung *eigener* Steuerelemente, Dialogfeld-Klassen und deren Ableitungen und zu guter Letzt dem Datentransfer zwischen Anwendungen über Drag&Drop und die Zwischenablage.

Der Leser findet hier nahezu alles, was er für die Entwicklung und Programmierung von Windows-Anwendungen wissen muss – untermauert mit mehr als 40 ausgewachsenen teils sogar recht komplexen Beispielprogrammen. Bei anderen Programmiersprachen und -umgebungen wären die Kapitel über selbstdefinierte Steuerelemente und Drag&Drop bereits etwas für Spezialisten, mit C# gehören sie zum Alltagsgeschäft.

Teil 4 – Verteilte Programme und Web-Anwendungen

Hat mit reinen Windows-Anwendungen nur noch insoweit zu tun, als Formulare auch hier eines der möglichen Ein- und Ausgabemedien darstellen. Nach einer Einführung in XML (die – unvermeidlich – erst einmal eine Schneise in den Dschungel der Internet-Akronyme und -Protokolle von HTML bis XSL schlagen muss) geht es erst einmal um die von .NET für XML zur Verfügung gestellten Klassen. Als Nächstes stehen einfache Client-/Server-Systeme und dann Mehrschichtsysteme auf dem Programm, bevor dieser Buchteil dann auf die Programmierung mit dem Internet Explorer als Host, Mail und verteilte Datenbanksysteme mit ASP.NET eingeht.

Web-Formulare, die Server- und Clientseite von Webdiensten, beiderseitiges Caching und Zustandshaltung wären bei anderen Programmiersprachen – falls man das dort überhaupt hinkriegt – wiederum Themen für einige wenige Gurus, und die Kommunikation über Firewalls hinweg mit SOAP erst recht. Wie rund 25 ausgewachsene Beispielprogramme zeigen, lassen sich mit C# und der .NET-Laufzeitumgebung nicht nur Clients, sondern auch Server ausgesprochen geradlinig implementieren.

Teil 5 – Weiterführendes

Beschäftigt sich mit Multithreading, für das C# – wer hätte es gedacht – einige verblüffend einfache Lösungen und neue Konzepte in petto hat und wirft dann einen Blick über den Zaun: COM- und ActiveX-Komponenten lassen sich geradlinig in .NET-Anwendungen integrieren, in umgekehrter Richtung geht es mit ein wenig Sorgfalt auch – was die Migration natürlich sehr erleichtert, weil man Existierendes stückweise ersetzen und nicht auf einen Schlag umbauen muss. Auf- und Rückrufe der Windows-API sowie anderer DLLs – und vor allem, was darüber nicht in der Online-Dokumentation steht –, runden diesen Buchteil ab, der ebenfalls reichlich mit Praxisbeispielen bestückt ist.

Der Leser findet hier Antworten auf spezielle Fragen, die sich automatisch ergeben, wenn er vor die Aufgabe gestellt ist, bestehende Software mit .NET-Technologie zu bestücken und umgekehrt. Die gut 15 Beispielprogramme exerzieren eine Reihe in der Online-Dokumentation – wenn überhaupt – nur recht spärlich beschriebener Techniken vor, die dabei ein Rolle spielen.

Anhang

Der Anhang ist kein Nachschlagewerk, eine Liste der Klassen fehlt ihm genauso wie die früher unvermeidliche ASCII-Tabelle. Dort geht es vielmehr um Dinge, die thematisch nicht so recht in die Buchteile gepasst haben – wie Details zu Internetprotokollen (TCP, IP, HTTP) und ihre Untersuchung im Netzwerk-Monitor, die Fehlersuche bei paralleler Entwicklung mit verschiedenen Sprachen (Delphi, MSVC, C#) und die Installation von .NET sowohl auf der Entwicklungsmaschine als auch auf den jeweiligen Zielsystemen. Und außerdem findet sich hier eine kompakte Übersicht über das umfangreiche Beispielmaterial auf der Begleit-CD und den Umgang damit.

Teil 1 Annäherung an .NET

Kapitel 1: Einführendes zur Philosophie von .NET 29

Kapitel 2: Das .NET-Puzzle aus der Vogelperspektive 35

Kapitel 3: Erste Schritte mit Visual Studio .NET 49

Dieser Teil ist bewusst kurz gehalten, wird das Herz ganz besonders Eiliger erfreuen, stellt aber für alle Pflicht dar – vor allem für die LeserInnen, die noch keine echte Berührung mit .NET hatten.

Die folgenden drei Kapitel beschreiben, was sich Microsoft bei .NET gedacht hat, woraus .NET besteht, wie man mit Visual Studio .NET einfache Anwendungen (Konsolen- Windows-, Web-Anwendungen) erstellt, und wie sich die grundlegende Fehlersuche in dieser Entwicklungsumgebung gestaltet. Der Rest dieses Buchs setzt diese ersten 60 Seiten Grundlagen und Techniken voraus.

1 Einführendes zur Philosophie von .NET

Mit .NET hat Microsoft die Synthese im klassischen Hegelschen Sinne mal wieder geschafft (und sogar im ersten Anlauf): Gegensätzliches fügt sich auf einer höheren Ebene – der von .NET – zu einem ungemein stimmigen Gesamtbild zusammen. Das ist wohl der Eindruck, dem sich kaum ein Entwickler nach der ersten halbwegs intensiven Berührung mit .NET als Technologie erwehren kann.

Die letzte Synthese dieser Art war bekanntlich die Win32-API, die als Vehikel für den Übergang in die 32-Bit-Technologie diente und seither als gemeinsamer Mantel für die sich zunehmend differenzierende Familie der Windows-Betriebssystem firmiert. Die Win32-API spielt weiterhin eine wichtige Rolle, obwohl der Blick der einzelnen Anwendung darauf von .NET völlig verdeckt wird.

Wer bisher Anwendungen auf der Microsoft-Schiene entwickelte, war trotz der – letztlich doch recht schwachen – Abstraktion durch die Win32-API den Besonderheiten des jeweiligen Betriebssystems ausgesetzt. Schlimmer noch wog allerdings, dass auch das Betriebssystem umgekehrt dem Ansturm der Anwendungen ausgesetzt war, die ihm nicht immer gerade freundlich gesinnt waren oder sich zumindest nicht immer so verhielten, wie es sich die Microsoft-Entwickler ausgedacht hatten. Da .NET viel Wert auf Ausführungssicherheit und Codesicherheit legt, scheinen diese Aspekte ausgemerzt.

Dieses Kapitel skizziert kurz die zentralen Ideen, die hinter .NET stecken.

1.1 Die Idee von Microsoft

Beim Blick auf das bisherige Lebenswerk von Bill Gates muss selbst der schärfste Konkurrent – oder gerade der – ebenso wie unsereins vor Hochachtung erblassen. Von Neid erst gar nicht zu sprechen. Wie aus der einfachen Idee, so etwas wie ein Betriebssystem für etwas »zu groß geratene Taschenrechner« zusammenzuzimmern – die Idee war noch nicht mal von ihm, er war nur im richtigen Moment der richtige Mann am richtigen Ort, sie anzupacken – ein alles verschlingender Gigant wurde, ist beispiellos für die Technikgeschichte der Neuzeit. Fraglos ist es nicht die Idee gewesen, die das Leben aller auf diesem Planeten so verändert hat, sondern ihre konsequente Verfechtung, ihre akribische Ausdeklination und ihre stete Erneuerung.

.NET stellt die nun bisher neueste und – beinahe auf den ersten Blick erkennbar – großartigste Fassung der Microsoftschen Idee dar, deren Auswirkungen auf die Informationstechnologie gar nicht groß genug eingeschätzt werden können. Der Technologieschub, den .NET mit sich bringt, ist wohl wieder nur das *vorläufige* Ende der Fahnenstange, auch wenn es zum jetzigen Zeitpunkt sichtlich schwer fällt, über noch Großartigeres nachzudenken. Ein Brocken wie .NET will von der Welt erst mal verdaut sein – mit all seinen Auswüchsen und Aspekten.

1.1.1 Mainstream-Technologie als »Freeware«

Gebrauchsfertige Technologie ist nur eine Seite des wirtschaftlichen Erfolgs – und um ehrlich zu sein: An deren Spitze oder auch nur besonders innovativ war Microsoft vor .NET hier noch nie so recht. Masse ist die andere. Die wahre Profession des Konzerns war schon immer ein ausschließlich auf den Mainstream ausgerichtetes Monumentalmarketing, das mehr Fingerspitzengefühl im Aufgreifen »neuer« Technologien und Ideen zeigte, das aus Fehlern schneller lernte und das nicht zuletzt auch die Kunst des Wartens besser beherrschte als alle anderen auf dem Markt.

Welcher andere Konzern hat denn bisher jahrelang sehenden Auges akzeptiert, wie seine Produkte tonnenweise unter der Hand kopiert und weitergegeben wurden? Und, Hand aufs Herz: Bei wie vielen Microsoft-Produkten, mit denen *Sie* in Ihrer bisherigen Karriere als PC-AnwenderIn und ProgrammiererIn gearbeitet haben, haben Sie oder Ihre Firma es mit dem Lizenzabkommen mit Microsoft eher kavaliersmäßig gehalten?

Bill Gates dafür als Märtyrer hinzustellen, hat angesichts des doch nicht unbeachtlichen Vermögens, das er sich mit seinem offensichtlich aufgehenden Kalkül erwerben konnte, allerdings eher erheiternde Effekte – zumal, wenn das Gegenüber nicht gerade ein eingenordeter Microsoftianer ist. Trotzdem hat der Spruch schon seine Richtigkeit: Wenn ein Freeware-Konzept jemals richtig funktioniert hat, dann war es die Produktpalette von Microsoft.

Ob das die Grenze vom Humor zum Galgenhumor überschreitet, war stets die Frage – die spätestens seit Windows Me, Windows XP und .NET beantwortet ist: .NET mit seinem Sicherheitsmodell und den darauf aufsetzenden inhärenten Lizenzierungsmechanismen schützt nicht nur Anwendungen und Benutzer, sondern eben auch Microsoft. Die Daumenschrauben sind angesetzt, und es bleibt lediglich offen, wann man es in Redmond für opportun hält, sie anzuziehen …

1.1.2 Antwort auf Java

Mit Aufkommen des Internets wurden Aspekte wie Portabilität und Kompatibilität auf der einen Seite und Sicherheit auf der anderen Seite immer

wichtigere Kriterien für den Erfolg von Anwendungen. Lebendiger Ausdruck dieser Entwicklung war die ungemein stürmische Rezeption von Java als Programmiersprache für Inter- und Intranet-Anwendungen.

Dieser von Sun Systems in Gang gesetzte Zug war zweifellos ein Schritt in eine völlig neue Richtung, der erstmals eine Programmiersprache mit eingebautem mehrschichtigem Sicherheitskonzept ins Gespräch brachte. Die Durchsetzung des Java-Sicherheitskonzepts erfolgt in der so genannten *Sandbox*, auch *virtuelle Java-Maschine* genannt, die eine künstliche, aber vollständige Umgebung für ein Programm darstellt, aus der es (zumindest theoretisch) nicht »ausbrechen« kann.

Die Antwort von Microsoft darauf ließ – untypisch für die Branche – Jahre auf sich warten, fiel dafür umso nachhaltiger aus. Da die Skepsis gegenüber den Machenschaften des Konzerns unter anderem nach dem Browser-Krieg gegen Netscape groß war und sich Microsoft auch in der Java-Gemeinde mit seiner Lizenzentwicklung J++ alles andere als Freunde gemacht hatte, wurden C# und die dahinter stehende .NET-Technologie anfänglich nur zu gerne als billiger Klon der Java-Idee abgetan.

Erst mit Verbreitung der ersten Betas wurde der Welt – wie seinerzeit bei Windows 95 – klar, dass .NET mehr als nur die mit eigenen Anpassungen verwässerte Weiterentwicklung einer – wieder einmal – woanders abgeschauten Technologie war. Obwohl Java und C# auf den ersten Blick keine allzu großen Unterschiede aufweisen, ist dies eher auf die gemeinsame Abstammung von C++ zurückzuführen denn auf ein Plagiat. Der wesentliche Unterschied beider Sprachen gegenüber C++ ist der Verzicht auf ein Zeigerkonzept bzw. die direkte Abbildung des Zeigerkonzepts auf das Typsystem und eine damit einhergehende automatische Freispeicherverwaltung durch eine Laufzeitumgebung – beides für Visual Basic seit dem Übergang zur Objektorientierung längst Stand der (Microsoft-)Technik.

1.1.3 Fundament als Idee

Dass sich in eine Laufzeitumgebung aber noch einiges mehr an Funktionalität als nur eine Speicherverwaltung packen lässt, nämlich auch ein Sicherheitsmodell, scheint wohl der innovative Kern der Java-Idee gewesen zu sein.

Diese Idee hat Microsoft zweifelsohne für .NET aufgegriffen und um einiges weiter ausgereizt – allerdings unter einer völlig anderen Perspektive: Die Java-Sandbox ist Eins zu Eins auf die Sprache Java und ihre Bedürfnisse zugeschnitten und bleibt in ihrer Anpassung an die einzelne Plattform (Betriebssystem) minimalistisch, um auf möglichst vielen Plattformen problemlos einsetzbar zu sein. Das Gegenstück von Microsoft ist ein geradezu monumentales *Fundament*, das, wenngleich vom Prinzip her ebenfalls platt-

formunabhängig, vorerst »nur« auf der Plattform Windows verfügbar ist. Dies wird wohl auch noch eine Weile so bleiben, auch wenn bereits Spekulationen über Portierungen für Linux und MAC-OS in Umlauf sind.

Dass eine Portabilität von .NET für Microsoft als Betriebssystemhersteller mit dem höchsten Marktanteil nicht gerade an oberster Stelle stehen kann, ist nur allzu verständlich. Es mag sogar generell verwundern, warum Microsoft mit .NET überhaupt eine so vollständige Trennung zwischen Win32-API und Anwendungen vorgenommen hat. Wer die Internet Explorer-Kontroverse mitverfolgt hat, wird wissen, dass die Konzernpolitik bisher doch eher in die andere Richtung gewiesen hat.

Zweifellos ist eine Idee aber umso besser, je vielschichtiger sie einen Beitrag zur Lösung anstehender Probleme leistet. Als kompakte Schnittstelle zwischen der Win32-API und der Anwendungssoftware gibt .NET auch einen klaren Schnitt für den eigenen Strukturwandel des Konzerns vor, der ihm durch kartellamtliche Einmischung gerichtlich aufoktroyiert wurde: Die Trennung in einen Konzern für Betriebssystemsoftware und einen für Anwendungssoftware. Auch wenn die Trennung aktuell noch nicht vollzogen ist, so ist sie doch absehbar.

1.1.4 .NET-Framework als Bündel

Als eigene Schicht über dem Betriebssystem lässt sich eine Technologie wie .NET nicht mehr mit Metaphern wie *sandbox* oder gar *nutshell* umschreiben. Sie wird zum »Betriebssystem über dem Betriebssystem« und begründet eine eigene Welt, in die Microsoft nahezu alles hineingepackt hat, was aufseiten der Anwendungsprogrammierung schon lange auf der Wunschliste stand – und noch mehr. Neben den bereits angesprochenen Aufgaben der .NET-Schicht hat sich Microsoft für die Ausgestaltung von .NET gerade in punkto Kompatibilität sehr ehrgeizige Ziele gesetzt. Um diese Ziele zu erreichen, ist gleich ein ganzes Bündel ineinander greifender Maßnahmen und Technologien erforderlich, die sich wie ein Puzzle zu einem höheren Ganzen, dem .NET-Framework, zusammenfügen.

Gemeinsame Plattform für viele Sprachen

Das .NET-Framework übernimmt die Rolle der gemeinsamen Infrastruktur für alle wichtigen Entwicklungssprachen. Momentan gehören C#, C++ mit verwalteten Erweiterungen, Visual Basic .NET, JScript .NET, ASP.NET auf Initiative von Microsoft und bald auch Borlands .NET-Version von Delphi zum Club der .NET-Sprachen und kommen so in den Genuss der Infrastruktur. Obwohl auch ein Java. NET eine echte Bereicherung dieses Angebots darstellen würde, wird Sun diesen Weg voraussichtlich wohl nicht gehen.

Damit eine Sprache als .NET-Sprache auftreten kann, muss sie Federn lassen. Im Falle von C++ kommt neben den verwalteten Erweiterungen also auch ein Satz von Regeln dazu, die den Gebrauch von Zeigerdatentypen regeln und insbesondere Zeigerarithmetik ausschließen. Der Weg von Visual Basic 6 zu Visual Basic .NET ist eine Metamorphose fast bis zur Unkenntlichkeit, die Sprache wirkt im neuen Gewand jedoch geschlossener als je zuvor. Bei Delphi bleibt noch abzuwarten, wie stark sich die Sprache letztlich verändern wird. Einzig C#, das extra für .NET entwickelt wurde, spiegelt den Geist von .NET Eins zu Eins wieder.

Intermediate Language als gemeinsame Zwischensprache

Zielcode der einzelsprachlichen .NET-Compiler ist nicht wie bisher Maschinencode, sondern IL: Ein zwischensprachlicher Code, der von einem JIT-Compiler des .NET-Framework bei Bedarf in Maschinencode übersetzt wird. Wie der Bytecode von Java ist der Zwischencode recht maschinennah, aber kompakt, und ermöglicht ein Konzept für Codesicherheit. Während Java hier mit Prüfsummen und einem Interpreter arbeitet, benutzt .NET bei Bedarf ein Signierverfahren und bringt grundsätzlich nativen Maschinencode zur Ausführung.

Übergeordnetes Objektmodell

Das .NET-Framework ist eine vollständig objektorientierte Umgebung, in der alle Einzelsprachen ein und demselben Objektmodell unterworfen sind. Es schafft die notwendigen strukturellen Voraussetzungen für das Miteinander (Kommunikation), aber auch für das Ineinander (Vererbung) verschiedenensprachlichen Codes. Anders als das Component Object Model – besser bekannt als COM –, das nur die Schnittstellenvererbung kennt, sieht das Objektmodell von .NET auch die Implementierungsvererbung vor – als *Einfachvererbung* wie bei Delphi, nicht als Mehrfachvererbung wie bei C++.

Jeder Datentyp ist Klasse

Die Objektorientierung von .NET geht so weit, dass *jeder* Datentyp als Klasse aufgefasst wird – zumindest formal. Das betrifft insbesondere auch die integrierten Datentypen der .NET-Sprachen: Integer, Boolean usw. werden als Klassen behandelt. Dabei unterscheidet das Modell strikt zwischen zwei Arten von Datentypen: Werttypen und Verweistypen.

Typsicherheit

Oberstes Gebot des Objektmodells von .NET ist die Typsicherheit. Das .NET-Framework sorgt mit einer Reihe von Maßnahmen dafür, dass sich ein Programm schlicht nicht daneben benehmen kann. Eine der Maßnahmen setzt direkt an der Spezifikation der einzelnen .NET-Sprache an: Sie verbannt das Konzept des Zeigers und der Adresse sowie die damit zusam-

menhängenden Mechanismen aus der Sprache. Implizit gibt es das Konzept natürlich weiterhin, es wird aber vollständig auf das Typsystem abgebildet.

Der »Preis« dafür ist eine für die Anwendungsprogrammierung höchst angenehme automatische Speicherverwaltung durch die .NET-Laufzeitumgebung. Kurzum, die Speichervergabe, -organisation und -freigabe bis hinunter zur Ebene des einzelnen Objekts unterliegt vollständig der Kontrolle der Infrastruktur. Das einzig Gewöhnungsbedürftige daran: Die Freigabe findet nicht (wie bisher gewohnt) synchron, sondern asynchron statt.

Neben der statischen Typüberprüfung durch den Compiler erfolgt eine dynamische Typüberprüfung zur Laufzeit auf der Basis von Laufzeittypinformationen. Typumwandlungen sind zum einen über die Vererbungszusammenhänge des Objektmodells möglich, zum anderen können sie auch auf der Basis einzelner Klassen – und mit den regulären Mitteln der jeweiligen .NET-Sprache – definiert werden. (Das beispielsweise in C/C++ erlaubte Umwandeln von Zeigertypen in beliebige andere Zeigertypen hat also keine Entsprechung, weil den .NET-Sprachen dafür das Mittel bewusst fehlt.) Passen zwei Werte vom Typ her nicht zusammen, fällt dies entweder dem Compiler oder der Laufzeitumgebung auf. Im einen Fall kommt es zu Fehlermeldungen, im anderen Fall zu Ausnahmen, für die das .NET-Framework einen sehr effizienten (natürlich gleichfalls objektorientierten) Mechanismus bereitstellt. Einen dritten Fall gibt es definitiv nicht.

Ausführungssicherheit

Die Typsicherheit ist nur *ein* Baustein des gesamten .NET-Sicherheitskonzepts. Sie ist eine syntaktische Garantie, das heißt, sie stellt sicher, dass die Logik eines Programms zu jedem Zeitpunkt wohldefiniert ist. Der »Absturz« eines Programms ist unter .NET ein geregelter Vorgang, der in jedem Fall auf eine nicht behandelte Ausnahme (oder eine Endlosschleife) zurückzuführen ist – beides semantische Phänomene. Mithin kann es also nicht mehr – wie in der bisherigen Anwendungsprogrammierung – zu Abstürzen aufgrund undefinierter Zustände kommen, die gegebenenfalls andere Anwendungen oder gar das gesamte System mit in den Abgrund reißen.

Die anderen Bausteine des .NET-Sicherheitskonzepts laufen auf das für Windows NT entwickelte richtlinienbasierte Sicherheitsmodell hinaus. Der Code wird zum einen aufgrund formaler Kriterien wie Sicherheitszonen und expliziter Vertrauenseinstufungen (rollenbasierte Sicherheit), zum anderen aufgrund von Sicherheitsmerkmalen wie Signierung klassifiziert. Diese Klassifizierung definiert die Umgebung, die der Code zu »sehen« bekommt – mithin die Rechte für den Zugriff auf Systemressourcen der verschiedensten Art.

2 Das .NET-Puzzle aus der Vogelperspektive

Als Rahmentechnologie umfasst .NET eine Vielzahl einzelner Komponenten: Aspekte, Modelle, Spezifikationen, Mechanismen und letztlich Codekomponenten. Wie die Teile eines Puzzles fügen sie sich auf höherer Ebene zu einem ebenso stimmigen wie vielschichtigen Gesamtkonzept zusammen.

Dieses Kapitel stellt die wichtigsten Komponenten einzeln vor und beleuchtet ihr Zusammenspiel.

2.1 Common Type System (CTS)

Die konkrete Fassung des .NET-Objektmodells ist das so genannte Common Type System (CTS). Als Spezifikation, der alle .NET-Sprachen unterworfen sind, bezieht sich das CTS auf die gemeinsame Ebene der Intermediate Language (IL). In dieser maschinennahen Zwischensprache liegen alle Datentypen vor, die eine .NET-Anwendung bzw. -Codekomponente einbindet, insbesondere die integrierten Datentypen der jeweils verwendeten .NET-Sprache. Das CTS definiert die so genannten .NET-Basisklassen als Grundlage für die Implementierung der integrierten Datentypen aller .NET-konformen Sprachen (sowie der Sprachen selbst) und gibt die Regeln für deren Zusammenspiel vor. Hierzu gehört insbesondere der im Namensbereich `System` versammelte üppige Fundus an elementaren und systemnahen Typen, wie `Int32` oder `UInt16`, aus denen .NET-Sprachen ihre integrierten Typen im Allgemeinen durch einfache Umbenennung – also ohne weitere Indirektion – rekrutieren. Damit stehen den .NET-Sprachen potenziell nicht nur dieselben Datentypen zur Verfügung, sondern sie verwenden tatsächlich sogar dieselben Implementierungen der Datentypen.

Umgekehrt wird jeglicher einzelsprachlicher .NET-Quellcode in die IL übersetzt und damit wiederum Teil des CTS, weshalb das CTS tatsächlich den Dreh- und Angelpunkt allen .NET-Codes darstellt.

Mit dieser Positionierung macht das CTS nicht nur klare Vorgaben, was der Compiler einer .NET-Sprache vorfindet und was er zu liefern hat, sondern legt auch – mit Blick auf den Vererbungsmechanismus des Objektmodells – das Zusammenspiel der Datentypen fest. (Tatsächlich sind die .NET-Compiler im Allgemeinen sogar selbst in einer der .NET-Sprachen – in C# –

implementiert.) Dummerweise wirken die Strukturvorgaben mitunter bis in die Spezifikationen der Einzelsprachen zurück.

Das Umkrempeln der Sprachspezifikationen stellt wohl den größte Kraftakt dar, den Microsoft für .NET vollbringen musste – und den wohl auch nur ein solcher Marktgigant zu Wege bringen konnte. Dass das Ergebnis nicht in jedem Fall zufriedenstellend ist, zeigt der Fall C++. Diese Sprache wurde durch die „verwalteten Erweiterungen" noch weiter überfrachtet. Wer sich auf die verwalteten Erweiterungen einlässt, fängt besser gleich mit C# an. Als ballastfreie Sprache, die Microsoft eigens als Implementationssprache für .NET aus der Taufe gehoben hat, transportiert C# den Geist von .NET natürlich am reinsten und am besten. Tatsächlich ist C# weitgehend aus demselben Holz geschnitzt wie C++, was die Migration zu einem echten Heimspiel geraten lässt.

2.1.1 CTS vs. COM als Objektmodell

Wer von VB oder Delphi her kommt, dem wird es auf den ersten Blick wahrscheinlich gar nicht auffallen, wie üppig C# mit vordefinierten Datentypen ausgestattet ist. C/C++-Programmierer, die bisher weder eine anständige Lösung für Strings noch ein Standardmittel für den Umgang mit Zeit- und Datumswerten (das Y2K-Problem lässt grüßen) zur Verfügung hatten, wenn sie nicht bereit waren, sich auf die STL, ATL oder die MFC einzulassen, sind hier erheblich weniger verwöhnt. Sie haben trotz der wohl größeren Umgewöhnung allen Grund, drei Kreuze zu machen, dass es nun so etwas wie das CTS gibt, dem alle .NET-Sprachen unterschiedslos unterworfen sind, und das die Grundlage für den Export sowie die sprachübergreifende gemeinsame Nutzung von Datentypen unter .NET darstellt.

Da jede Programmiersprache ihre eigene Entwicklungsgeschichte hat und die darin verwirklichten Konzepte letztlich ein Produkt evolutionärer Veränderungen sind, ist es nicht allzu verwunderlich, dass die Implementierungen der Datentypen und der Verarbeitungsmechanismen von Sprache zu Sprache höchst unterschiedlich ausgefallen sind. Mit zunehmender Vernetzung der Computersysteme und wachsender Komplexität der Anwendungen und Lösungen wurde allerdings der Ruf nach sprachübergreifendem Datenaustausch immer lauter. Und genau dem standen die Unterschiede auf der Ebene der integrierten Datentypen im Weg.

Von textbasierten Datenaustauschformaten einmal abgesehen, war das Objektmodell COM der erste ernsthafte Versuch von Microsoft, so etwas wie übergeordnete Datentypen auf die Beine zu stellen, die ein Miteinander verschiedensprachlich implementierter Komponenten ermöglichen. Zur Erzeugung von COM-Objekten ist es erforderlich, die vom COM-Standard vorgeschriebenen sprachunabhängigen Datentypen mit inhärenten Mitteln der jeweiligen Sprache zu synthetisieren – was sich durch geeignete Typbib-

liotheken zwar problemlos bewerkstelligen lässt, aber prinzipbedingt einen gewissen Überbau mit Laufzeitverlusten nach sich zieht.

Das Objektmodell der .NET-Technologie geht mit dem CTS einen anderen, wesentlich effizienteren Weg: Allen .NET-Sprachen stehen prinzipiell dieselben Datentypen zur Verfügung, weshalb der Datenaustausch von Komponente zu Komponente und auch die Vererbung auf Basis ein und derselben Implementierung erfolgt. Aus Sicht des Typsystems und der IL existieren de facto keine Unterschiede zwischen den Einzelsprachen.

2.1.2 Struktur des .NET-Objektmodells

Wie für Objektmodelle mit Vererbung üblich, schreibt das CTS auch für das .NET-Objektmodell eine hierarchische Organisation der Datentypen vor, die in einem allgemeinsten Datentyp ihren Ausgangspunkt hat, nämlich `System.Object`. Alle anderen Datentypen sind direkt oder indirekt Ableitungen dieses Datentyps (Abbildung 2.1) und können vorteilhafterweise damit auch gehandhabt werden (Stichworte: Virtualität, Boxing).

Werttypen, Verweistypen

Auf der Ebene der Einzelsprachen unterscheidet das CTS in nie dagewesener Klarheit zwischen WERTTYPEN und VERWEISTYPEN. Wie die Namensgebung bereits ausdrückt, repräsentieren Werttypen echte Werte, Verweistypen hingegen lediglich Verweise auf andere Größen. Werttypen, deren Werte auf das von C her bekannte Strukturkonzept zurückgehen, bilden die Grundlage für die Implementierung der Verweistypen. Und hinter den Verweistypen verbirgt sich das altbekannte Konzept der Zeigertypen.

Delegaten, Arrays, Klassen

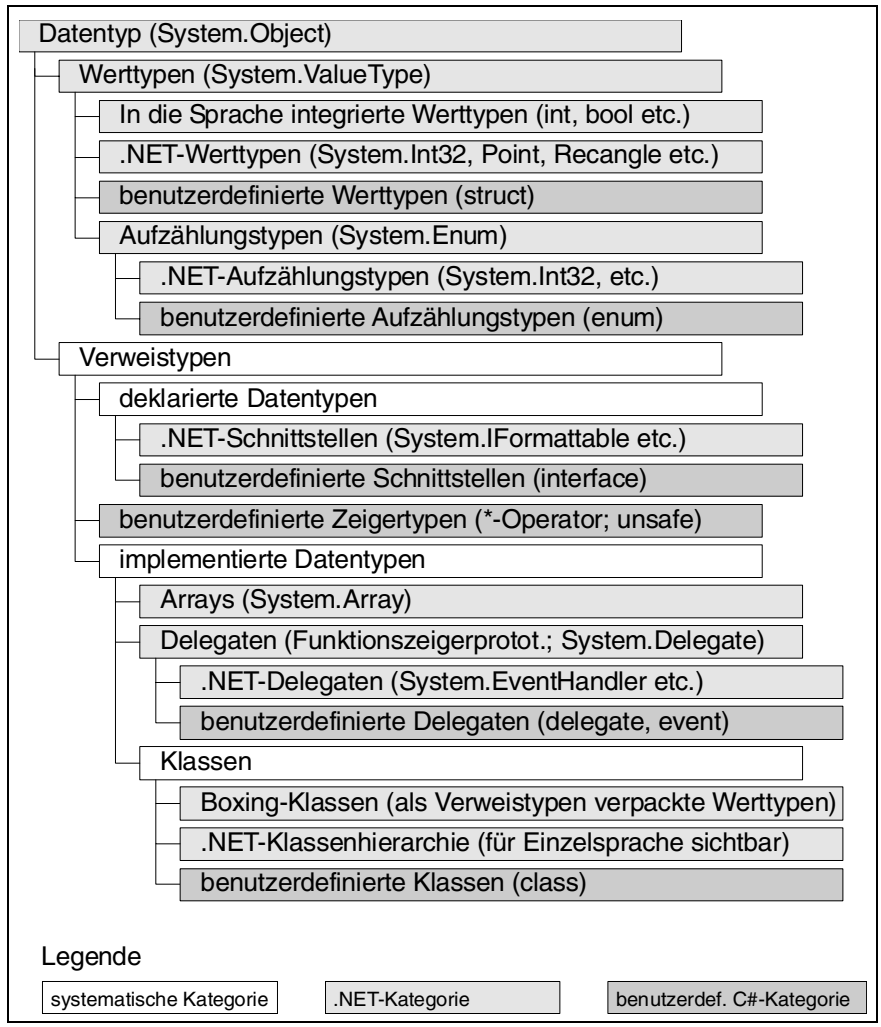
Für den Zweig der Verweistypen unterscheidet das Modell weiter zwischen Schnittstellentypen, die Einsprungtabellen (Stichwort *vtables*) für Routinen samt Prototypen deklarieren und Datentypen mit konkreter Implementation: Arrays, Delegaten und Klassen. Für Delegaten und Arrays sieht das CTS eine feste Implementation vor, auf die die einzelnen .NET-Sprachen direkt aufsetzen. DELEGATEN sind das .NET-Gewand für das altbekannte Konzept der Funktionszeigertypen.

Naturgemäß bilden die Klassen die größte Menge an Datentypen – sowohl der vordefinierten als auch der benutzerdefinierten. Als Brücke zwischen Strukturen und Objekten spezifiziert das CTS einen Mechanismus, der Werttypen als Verweistypen verpackt (Boxing) und wieder entpackt (Unboxing).

Für alle Datentypen ist ein Mechanismus namens TYPREFLEXION verfügbar, der es ermöglicht, Informationen über die Art, die Ausstattung und den

Aufbau der einzelnen Datentypen als solche einzuholen. (Interessantweise findet hier ein Kategoriesprung statt, denn die Datentypen müssen dafür faktisch auf Objekte abgebildet werden, die natürlich ihrerseits wieder einen Datentyp tragen.)

Abbildung 2.1: Das CTS im Überblick



2.1.3 CLS

Einzelsprachliche Komponenten sollten auch auf der Basis komplexer Datentypen miteinander interagieren können. Nun finden sich bekanntlich nicht in allen Sprachen die gleichen integrierten Datentypen. Beispielsweise kennt VB keine vorzeichenlosen Ganzzahltypen, während C/C++ und auch C# alle verfügbaren Ganzzahltypen (mit 1, 2, 4 oder 8 Byte) sowohl mit als auch ohne Vorzeichen als integrierte Datentypen anbieten. Die hierarchisch organisierte

und logisch in verschiedene Namensräume aufgeteilte Menge der .NET-Basisklassen ist allerdings reichhaltig genug, um die Anforderungen aller .NET-Sprachen hinsichtlich ihrer integrierten Datentypen abzudecken. Tatsächlich ist es damit allein aber noch nicht getan. Als sprachübergreifende Einrichtung spannen die .NET-Datentypen zwar eine Struktur für die sprachübergreifende Verwendung von Daten und Code auf, jedoch läuft dies für jedes Sprachenpaar auf eine andere Teilmenge hinaus.

Eine Art kleinster gemeinsamer Nenner für alle Sprachen und damit für die Programmierung sprachübergreifender Komponenten ist die Common Language Specification (CLS). Sie schreibt eine Untermenge an elementaren Datentypen vor, die eine Sprache mindestens unterstützen muss, um beispielsweise auch mit der .NET-Klassenhierarchie in vollem Umfang zu arbeiten. Eine Beschränkung auf diese Untermenge bei der Programmierung von Codekomponenten stellt sicher, dass diese von jeder CLS-konformen Sprache aus ansprech- und verwendbar sind.

Tabelle 2.1 gibt einen Überblick über den Regelsatz; Abbildung 2.2 verdeutlicht die Rolle der CLS als Filter für der Programmierung sprachübergreifender Codekomponenten. Aus Sicht einer Einzelsprache besteht natürlich keine Verpflichtung, die CLS zu beachten. Die .NET-Klassenhierarchie ist weitgehend (was auch heißt: nicht hundertprozentig) CLS-konform.

Tabelle 2.1: Regelsatz der CLS

CLS-Regel	Auswirkung für C#
Bezeichner werden nicht nach Groß- und Kleinschreibung unterschieden.	Für öffentliche (`public`) oder geschützte (`protected`) Elemente dürfen keine Bezeichner verwendet werden, die sich nur von der Groß-/Kleinschreibung her unterscheiden.
Nur eine eingeschränkte Auswahl der .NET-Basistypen ist CLS-konform.	Die integrierten Datentypen `sbyte`, `ushort`, `uint`, `ulong` dürfen nicht als Parametertypen und Rückgabetypen für öffentliche oder geschützte Eigenschaften, Methoden und Operatoren verwendet werden.
Es sind keine variablen Parameterlisten erlaubt.	Keine. C# erlaubt zwar über das Schlüsselwort `params` die Deklaration einer variablen Parameteranzahl für Methoden, bildet diese aber auf Arrays mit fester Elementanzahl ab, weshalb dies keine Verletzung der Regel darstellt.
Globale Variablen und Funktionen sind nicht erlaubt.	Keine. Diese Regel ist in der Sprachspezifikation von C# enthalten.

Tabelle 2.1: Regelsatz der CLS (Forts.)

CLS-Regel	Auswirkung für C#
Zeigertypen sind nicht erlaubt.	Öffentliche oder geschützte Elemente bzw. deren Rückgabe- und Parametertypen dürfen nicht als `unsafe` bzw. explizit als Zeigertypen deklariert werden.
Alle Ausnahmen müssen von der Basisklasse `System.Exception` abgeleitet sein.	Keine. Diese Regel ist Teil der Sprachspezifikation von C#.

Anwendungsübergreifender Datenaustausch

Damit eine .NET-Komponente generell Daten von anderssprachlichem Code verarbeiten kann, muss ihre öffentliche Schnittstelle (als `public` deklarierte Datenfelder, Eigenschaften und Methoden) auf die CLS-konformen Datentypen beschränkt sein.

Gemischtsprachliche Wiederverwendung von Code

CLS-konformer Code ist vollständig objektorientiert und steht über den Mechanismus der Vererbung grundsätzlich allen Einzelsprachen in Form von Klassen zur Verfügung. Das .NET-Objektmodell sieht neben der reinen Schnittstellenvererbung (vgl. COM) auch die Implementierungsvererbung (vgl. C++, Delphi) vor, was gemischtsprachliche Ableitungen für benutzerdefinierte Datentypen nicht nur möglich, sondern auch zu einer völlig normalen Angelegenheit macht. Das beste Beispiel dafür ist die .NET-Klassenhierarchie selbst, die in C# geschrieben ist, aber allen .NET-Sprachen zur Verfügung steht.

Damit eine .NET-Komponente als Basisklasse für die sprachübergreifende Vererbung nutzbar ist, muss ihre öffentliche (`public`) und geschützte (`protected`) Schnittstelle auf die CLS-konformen Datentypen beschränkt sein.

2.1.4 .NET-Klassenhierarchie als gemeinsame Typbibliothek

Die .NET-Klassenhierarchie ist eine umfangreiche Sammlung von Bibliotheksklassen, die allen .NET-Sprachen gleichermaßen zur Verfügung stehen. Aufbauend auf die grundlegenden .NET-Basisklassen, aus denen die .NET-Sprachen ihre integrierten Datentypen zwangsweise rekrutieren müssen, präsentiert sich die .NET-Klassenhierarchie als überaus luxuriös ausgestattete Umgebung. Diese Umgebung definiert zum einen die gesamte Sicht einer .NET-Anwendung auf die jeweilige Plattform sowie auf das .NET-Framework selbst; zum anderen ist sie aufgrund der im Objektmodell begründeten Implementierungsvererbung eine höchst effiziente Ausgangsposition für die Implementation eigener komplexer Datentypen.

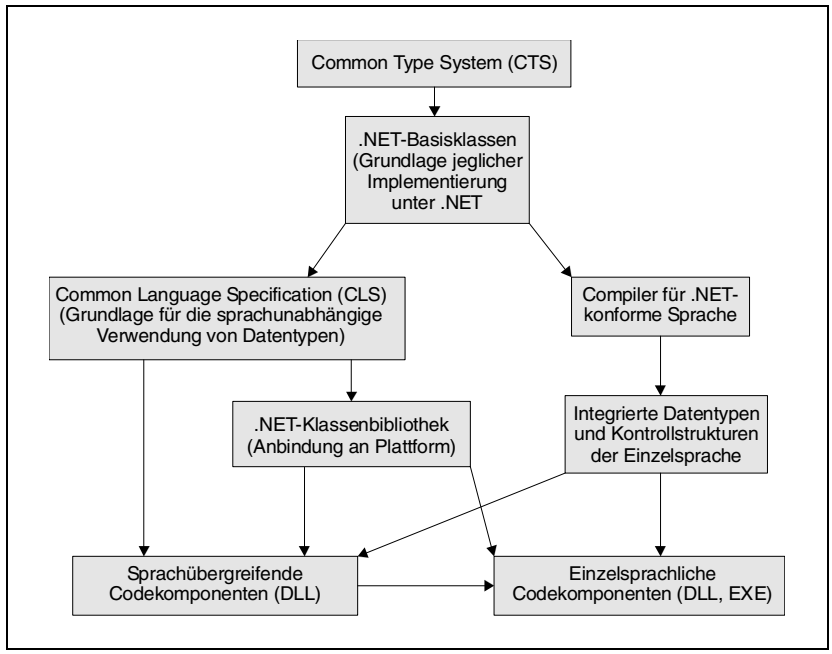


Abbildung 2.2: Rolle der CLS für sprachübergreifende Komponenten

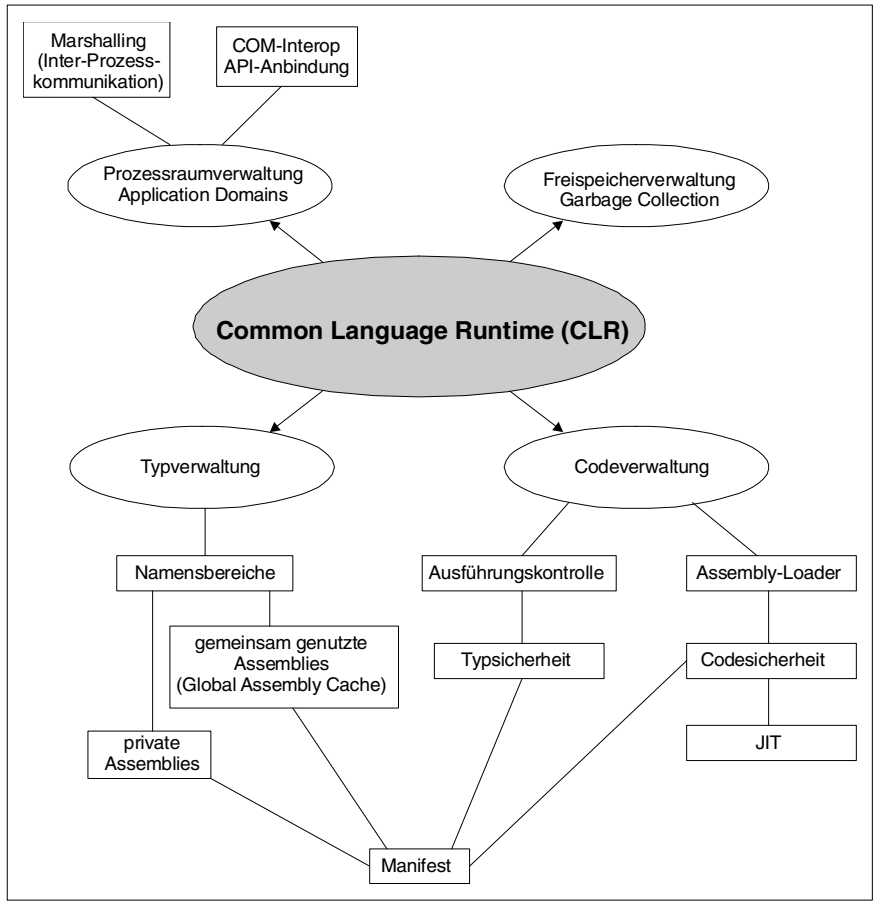
Für die Anbindung an die Plattform enthält die .NET-Klassenhierarchie unter anderem eine Unmenge von Hüllklassen (»Wrapper«) für alle möglichen plattformspezifischen Dienste (die – auf die Plattform Windows bezogen – den Zugriff auf die Win32-API vermitteln). Der überwiegende Teil der in der .NET-Klassenhierarchie definierten Klassen steht auch für eigene Ableitungen zur Verfügung und bildet so zusammen mit den in die Sprache integrierten Datentypen (deren Implementierung gleichfalls auf die .NET-Klassenhierarchie fußt) die Grundlage jeglicher benutzerdefinierter Datentypen.

Wie die .NET-Klassenhierarchie im Einzelnen ausgestaltet ist und in die praktische Programmierung eingeht, ist übergeordnetes Thema dieses Buchs.

2.2 Common Language Runtime

Der Begriff Common Language Runtime (*CLR)* steht für die Laufzeitumgebung des .NET-Framework. Zum Aufgabenbereich der CLR gehören eine Reihe von Verwaltungsaufgaben, die im Zusammenhang mit der Ausführung von .NET-Anwendungen stehen. Darunter fallen: Typ-, Versions-, Speicher-, Prozessraum- und Sicherheits-Management. Abbildung 2.3 versinnbildlicht die Zusammenhänge.

Abbildung 2.3:
Aufgaben der CLR



2.2.1 Gemeinsame Typverwaltung

Die Typverwaltung ist für COM eine relativ schwerfällige und recht unflexible Angelegenheit. COM-Objekte müssen grundsätzlich unter einem GUID registriert werden, der den Zusammenhang zwischen dem ausführbaren Code (*.dll*) und der Typinformation (*.tlb*) herstellt. Das heißt, dass die für den Aufruf eines COM-Objekts notwendige Information auf mindestens drei verschiedene Dateien verteilt ist: die Windows-Registrierung, die DLL und die Typbibliothek. Das hat verschiedene Nachteile: Erstens muss ein COM-Objekt grundsätzlich registriert werden, damit die Typverwaltung von seiner Existenz erfährt. Dies erfordert natürlich gewisse Rechte aufseiten des Anwenders. Zweitens kann immer nur eine Version eines COM-Objekts installiert sein, da nur ein GUID zur Verfügung steht. Neuere Versionen eines COM-Objekts müssen die ältere überschreiben, sodass die Installation einer Anwendung nicht selten erhebliche Seitenwirkungen auf andere bereits installierte Anwendungen hat (Stichwort DLL-Hölle). Drit-

tens ist die Verteilung der relevanten Informationen auf verschiedene Stellen im System eine filigrane Angelegenheit. Es reicht eine Pfadänderung, der Austausch eines Datenträgers oder das Wegfallen einer Freigabe, um die Systemkonsistenz zu beeinträchtigen oder gar nachhaltig zu zerstören.

Die .NET-Typverwaltung vermeidet diese Schwächen weitgehend: Die Codedateien der .NET-Typen enthalten neben dem Code auch alle Typinformationen und sind damit selbstbeschreibend. Die Windows-Registrierung bleibt außen vor (was wohl einerseits auf das Konto der plattformübergreifenden Ausrichtung von .NET geht, andererseits das Resultat bitterer Erfahrung der vergangenen Jahre sein dürfte). .NET arbeitet stattdessen mit einer ausgeklügelten Typauflösungsstrategie, die ein buntes Nebeneinander aus privaten und gemeinsam genutzten Codekomponenten in verschiedenen Versionen gestattet. Die gemeinsam genutzten Codekomponenten werden direkt von .NET in einem Pool verwaltet.

Namensräume

Die Typverwaltung der CLR operiert auf der Basis von NAMENSRÄUMEN – einem strukturierten Benennungsschema für Datentypen, das vom tatsächlichen Speicherort und damit von der Version eines .NET-Datentyps abstrahiert. Die Zuordnung zu einem Namensraum ist rein logischer Natur, mit Abstammung im Sinne von Vererbung hat sie nichts zu tun. Die TYPAUFLÖSUNG erfolgt zur Laufzeit und fällt in den Aufgabenbereich der .NET-Typverwaltung. Sie schafft die Zuordnung zwischen dem einzelnen über den Namensraum qualifizierten Typbezeichner und der physischen Datei, in der die Implementation des Datentyps steckt. Um zu verstehen, wie das vor sich geht, ist ein Blick auf die Codeorganisation unter .NET erforderlich.

Assemblies

Die folgende Beschreibung ist auf die Sprache C# zugeschnitten; das Prinzip ist für die anderen .NET-Sprachen aber dasselbe.

C#-Quellcode ist eine Ansammlung von Klassendefinitionen, die einzeln oder zu mehreren in *.cs*-Dateien gespeichert sind. Ziel der Klassendefinitionen ist die Zusammenstellung einer Klassenbibliothek oder eines ausführbaren Programms. Auf der Ebene des Quellcodes gibt es zwischen beiden nur einen einzigen Unterschied: Für ein Programm ist es erforderlich, dass eine – und nur eine – der Klassen eine Methode namens `Main()` definiert.

Eine KOMPILIERUNGSEINHEIT (in Visual Studio .NET als *Projekt* bezeichnet) kann aus beliebig vielen *.cs*-Dateien sowie gegebenenfalls zusätzlich einzubindenden Ressource-Dateien bestehen. Der C#-Compiler übersetzt die Kompilierungseinheit in die IL und fasst den Code zur so genannten ASSEMBLY zusammen, die eine eigenständige Codekomponente darstellt. Obwohl eine Assembly je nach gewünschtem Ausgabetyp vom Compiler die

Dateierweiterung *.exe* oder *.dll* erhält, findet sich dort (abgesehen von einer recht kurzen Initialisierungsroutine zum Aufruf der Laufzeitumgebung) kein Maschinencode – der Versuch, eine als *.exe*-Datei vorliegende Assembly auf einem System zu starten, auf dem das .NET-Framework nicht installiert ist, endet mit einer Fehlermeldung. Grundsätzlich kann jede Kompilierungseinheit zu einer Klassenbibliothek (*.dll*) übersetzt werden. Für ein startfähiges Programm (*.exe*) ist wie gesagt eine `Main()`-Methode Pflicht.

Eine Assembly enthält im Wesentlichen drei Abschnitte: Einen *Codeabschnitt* mit dem prozeduralen Code der Klassen, einen *Ressourcenabschnitt* mit den eingebundenen Ressourcen und das so genannte MANIFEST mit den Metadaten. Die METADATEN umfassen alle Typinformationen über die in der Assembly enthaltenen Datentypen sowie Informationen über die Assembly selbst – darunter eine vierteilige Versionsinformation sowie optional eine Herstellerinformation, eine Produktinformation, ein Copyrightvermerk, eine Kulturinformation und je nach Art der Assembly auch Sicherheitsmerkmale (Schlüssel, digitale Signatur).

Die Bindung einer Assembly zu anderen Assemblies wird durch *Verweise* auf Namensbereiche ausgedrückt und erst zur Laufzeit – und dann auch nur bei Bedarf – hergestellt. Die Verweise können statischer oder dynamischer Natur sein. Statische Verweise sind bereits zur Übersetzungszeit bekannt und werden als Teil der Metadaten gespeichert. Dynamische Verweise werden hingegen erst zur Laufzeit generiert und bedürfen einer expliziten Ladeanforderung, die ihrerseits die Typauflösung veranlasst. Dynamische Verweise lassen sich auch als partielle Verweise formulieren, womit die Typauflösung noch weiter in den Vordergrund rückt.

In Anlehnung an das klassische DLL-Konzept von Windows erlaubt .NET sowohl private als auch gemeinsam genutzte Assemblies.

Versionsinformation

In einer Assembly sind zwei unterschiedliche Versionsinformationen gespeichert. Neben der bereits erwähnten vierteiligen Versionsnummer, die sich aus Hauptversion, Nebenversion, Buildnummer und Revision zusammensetzt, steht eine textbasierte Versionsinformation im Angebot. Sie spielt allerdings keine Rolle für die Versionsverwaltung und ist rein informeller Natur.

Private Assemblies

Private Assemblies suchen die Typverwaltung im Verzeichnis sowie in den Unterverzeichnissen der jeweiligen Anwendung (*.exe*). Sie sind nur für den Gebrauch durch die Anwendung bestimmt. Da die Zugehörigkeit rein durch das Dateisystem vermittelt wird, ist eine Registrierung nicht erforderlich. Zwei Anwendungen können eine private Assembly nur dann gemeinsam

nutzen, wenn sie entweder aus demselben Verzeichnis heraus gestartet werden oder physische Kopien verwenden. Eine private Assembly kann einen so genannten *starken Namen* besitzen, muss aber nicht.

Starke Namen

Ein STARKER NAME setzt sich aus der Identität (Textname) der Assembly, der Versionsinformation und den Kulturdaten (sofern vorhanden) – sowie einem öffentlichen Schlüssel und einer digitalen Signatur zusammen. Er wird mithilfe eines zum öffentlichen Schlüssel passenden privaten Schlüssels aus den in der Assembly enthaltenen Informationen generiert und stellt – vergleichbar mit einem GUID – eine weltweit eindeutige Benennung dar. (Zum .NET-Framework gehört ein Werkzeug für die Erzeugung von Schlüsseln.)

Gemeinsam genutzte Assemblies

Gemeinsam genutzte Assemblies fasst die .NET-Typverwaltung – vergleichbar mit einer Registrierung – logisch im so genannten GLOBAL ASSEMBLY CACHE (GAC), auch schlicht *globaler Cache* genannt, zusammen. Die darin versammelten Datentypen stehen allen .NET-Anwendungen gleichermaßen zur Verfügung. Um potenziellen Namenskonflikten zwischen Assemblies verschiedener Hersteller vorzubeugen, enthält der GAC ausschließlich Assemblies mit starken Namen. (Zur GAC-Installation (und Deinstallation) einer Assembly stellt .NET gleichfalls Werkzeuge bereit.)

Typauflösung

Die Typauflösung ist an sich ein komplexer Vorgang, bei dem viele Indirektionen möglich sind. Im Normalfall läuft er so ab: Die Typverwaltung sucht den Datentyp zuerst in den privaten Assemblies der Anwendung und dann erst im GAC. Findet sich der Datentyp in einer privaten Assembly, wird er ohne weitere Prüfung der Versionsinformation verwendet. Findet er sich nur im GAC, entscheidet die Versionsinformation der jeweiligen Assembly. Damit eine Assembly für die Auflösung eines Verweises überhaupt in Betracht gezogen wird, muss im Normalfall die Haupt- und Nebenversion passen. Enthalten mehrere Assemblies im GAC eine Implementation des Datentyps, entscheiden Buildnummer und Revision darüber, welche das Rennen macht.

Das geschilderte Standardverhalten lässt sich allerdings auch noch an verschiedenen Stellen (und Ebenen) durch Richtlinien- und Konfigurationsdateien modifizieren.

2.2.2 Codeverwaltung

Die Codeverwaltung der CLR läuft unbemerkt im Hintergrund ab und entzieht sich weitgehend der Kontrolle des Programmierers. Nach der Typauflösung tritt der Assembly-Loader in Aktion, der seinerseits Maßnahmen zur Prüfung der Codesicherheit entlang der Sicherheitsrichtlinie des .NET-Framework sowie – auf Anforderung durch den Code selbst – zusätzliche Sicherheitsüberprüfungen auf den Plan ruft. Das Sicherheitsmodell von .NET ist recht ausgefeilt. Es bezieht die Sicherheitseinrichtungen der Plattform ein und ergänzt diese durch eigene Sicherheitskonzepte wie Sicherheitszonen, einem Modell für Vertrauenswürdigkeit, signierte Dateien usw.

JIT-Kompilierung

Wie schon ausgeführt, liegen Assemblies als IL-Code vor. Zur Ausführung des Codes ist ein weiterer Übersetzungsschritt erforderlich – die so genannte just-in-time-Kompilierung durch einen JIT-Compiler. Als fester Bestandteil der CLR obliegt es diesem Compiler, den IL-Code in plattformspezifischen Code umzusetzen – immer vorausgesetzt, der Code wurde als vertrauenswürdig eingestuft. Hierfür gibt es grob zwei Strategien: JIT-Kompilierung unmittelbar bei Aufruf und JIT-Kompilierung bei Installation. Die erste Strategie ist für private Assemblies vorgesehen und bedeutet zwar eine gewisse Verzögerung, die laut Microsoft jedoch kaum ins Gewicht fällt und zusätzlich durch einen internen Cache abgefedert wird, der den übersetzten Code eine Zeit lang vorhält. Übersetzt wird in diesem Fall nur der Code, der tatsächlich benötigt wird – auf der Ebene einzelner Methoden. Ist eine schon übersetzte Methode nicht mehr im Cache, muss sie für den nächsten Aufruf erneut übersetzt werden.

Die zweite Strategie bleibt gemeinsam genutzten Assemblies vorbehalten und sieht einen Aufruf des JIT-Compilers bei Installation einer Assembly im GAC vor. Die Assembly wird dabei en bloc übersetzt.

Ausführung

Die Ausführung von .NET-Code geschieht wie üblich im Kontext eines Prozesses. Der Prozess stellt der Anwendung einen eigenen 4 GB-Adressraum zur Verfügung, der gegen Zugriffe durch andere Prozesse abgeschirmt ist. Diese Abschirmung ist aber nicht nur ein Segen, sondern auch ein Fluch – vor allem dann, wenn zwei Anwendungen miteinander kommunizieren wollen: Ein Marshalling, das Prozessgrenzen überwinden muss, ist grundsätzlich laufzeitintensiv.

Application Domains

In Form der Application Domains bietet das .NET-Framework als verwaltete Umgebung eine Lösung für dieses Dilemma an, die ihr Debüt mit COM+ hinter sich gebracht hat. Das Konzept der Application Domains sieht vor,

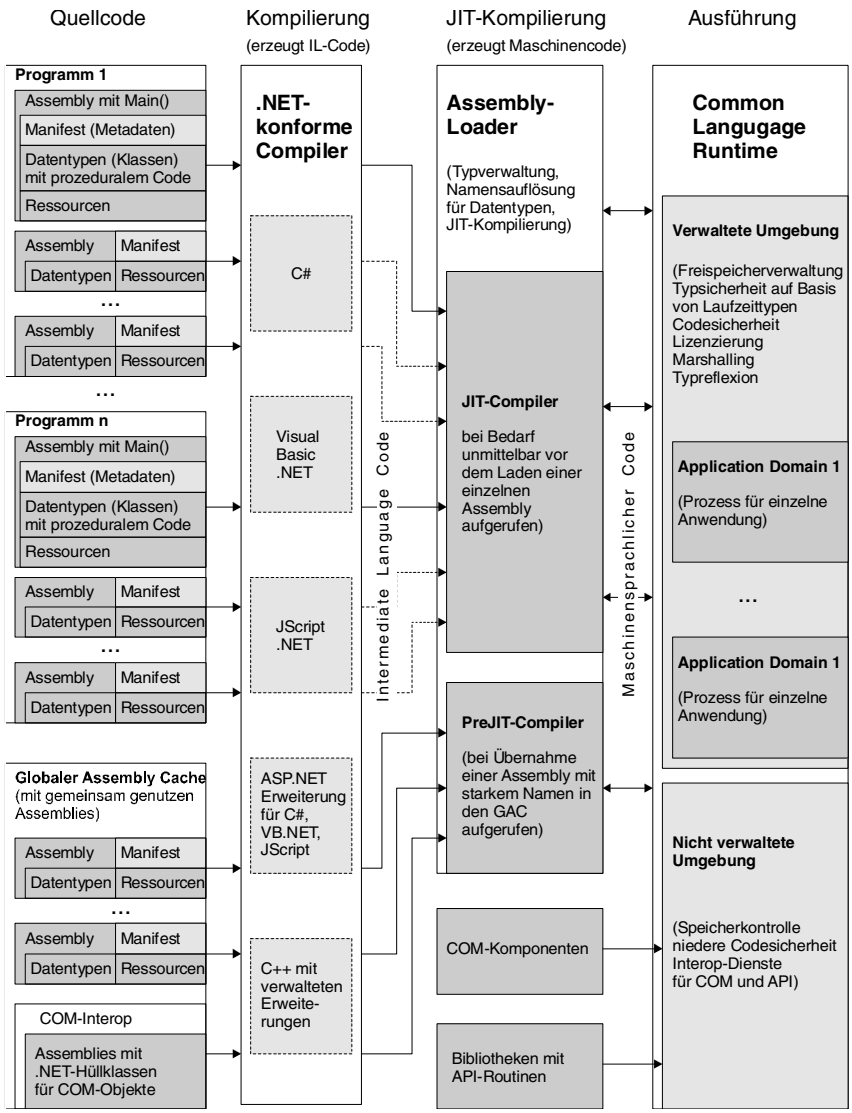


Abbildung 2.4: Das Ausführungsmodell von NET

dass sich mehrere Anwendungen einen einzigen Prozessraum teilen und darin als Threads ausgeführt werden. Die Kommunikation zwischen den Threads erfordert zwar Synchronisationsmaßnahmen, doch das Marshalling von Thread zu Thread ist erheblich schneller als von Prozess zu Prozess. Voraussetzung für die gemeinsame Ausführung in einer Application Domain ist allerdings, dass sich der beteiligte Code »anständig« benimmt – er muss daher auf jeden Fall typsicher sein und darf keine unsicheren Bestandteile haben. (C# fordert, dass unsicherer Code explizit als solcher mit dem Schlüsselwort `unsafe` deklariert werden muss; Code dieser Art lässt sich nur mit dem zusätzlichen Compilerschalter */unsafe* in IL übersetzen.)

Speicherverwaltung und Garbage Collector

Als verwaltete Umgebung für Sprachen ohne explizites Zeigerkonzept muss die CLR eine Speicherverwaltung für den Heap bereitstellen, die den Speicher auf der Ebene einzelner Objekte zuteilt und effizient organisiert. Für die Freigabe zugeteilten Speichers (und die dabei anfallenden Umschichtungen) ist der so genannte GARBAGE COLLECTOR (GC) – zu Deutsch „Freispeicherverwaltung" – zuständig. Er stellt einen asynchronen – das heißt, in einem eigenen System-Thread ausgeführten – Mechanismus dar, der in regelmäßigen Zeitabständen alle Belegungen für inzwischen aus der Sicht des Programms zerstörte Objekte freigibt und dabei entstehende Lücken durch Umschichtung schließt. Das Prinzip des Mechanismus ist verblüffend einfach. Der GC geht einfach den Verweisen aller Variablen im aktuellen Geltungsbereich (das sind die Variablen auf dem Stack) in einer Art Tiefensuche nach und findet so die Objekte heraus, die noch gebraucht werden. Objekte, zu denen kein Verweis mehr existiert, werden danach auch „physisch" zerstört.



Die Stringverwaltung benutzt einen eigenen Stringpool, dessen Pflege nach anderen Regeln geschieht als für ordinäre Heap-Objekte.

3 Erste Schritte mit Visual Studio .NET

Visual Studio .NET ist – wie sollte man es von einem Microsoft-Produkt auch anders erwarten – in jeder Hinsicht überwältigend: Eine integrierte Entwicklungsumgebung, die Editor, Formulardesigner, Compiler, Online-Dokumentation (MSDN), Debugger, Profiler und Klassenbrowser miteinander vereint, für eine Vielzahl unterschiedlicher Projekte gut ist – und für eine Vielzahl unterschiedlicher Sprachen. Wie bei jeder neu eingeführten (und aller Abstraktion zum Trotz auf der Entwicklerseite reichlich systemnahen) Technik sind die Anforderungen an den Computer vergleichsweise hoch – nicht nur, was den freien Platz auf der Platte (2,4 GByte) für eine Komplettinstallation, die Leistungsfähigkeit des Prozessors und die Größe des Hauptspeichers, sondern auch, was den Aktualitätsgrad des Systems betrifft. Details dazu finden Sie in Anhang A, »Erfahrungen mit der Installation«, der die Installationsprozedur nebst den darin enthaltenen kleinen Fallstricken beschreibt.

In diesem Kapitel geht es weder um eine Liste der verfügbaren Compileroptionen und Sprachen noch um die Feinheiten der diversen Assistenten, die sich in diesem Entwicklungssystem tummeln. Vielmehr soll es ein erstes Gefühl dafür vermitteln, wie man einfachere Projekte mit Visual Studio .NET zu Platte bringt.

3.1 Windows-Anwendungen

Allen Erweiterungen zum Trotz unterscheidet sich Visual Studio .NET von seinen Vorgängern an der Oberfläche hauptsächlich in einem Punkt: Es wird dem »Visual« in seinem Namen gerecht, weil man hier – ähnlich wie bei VB und Delphi – visuelle Komponenten in Formulare einsetzen und über Methoden wie `OnClick()` sozusagen miteinander verdrahten kann. (Mit Microsofts C++ ist dasselbe nicht nur in den älteren Ausgaben von Visual Studio ein recht mühsamer Prozess, der seine Ursprünge im Ressourcen-Editor von Windows 3.0 auch 12 Jahre später nicht verleugnen kann.)

Im Prinzip wäre es denkbar, ein neues Projekt im Programmierstil der 70er Jahre anzufangen, also mit einer leeren Textdatei. VS.NET verfügt aber über eine Reihe von Assistenten, die von Ihnen einige grundlegende Dinge wissen wollen – beispielsweise, ob bei dem Projekt eine fensterbasierte Windows-Anwendung herauskommen soll, ein Web Service oder ein Konsolen-

programm –, und die dann ein entsprechendes Rahmengerüst anlegen. Bitte beachten Sie, dass diese Assistenten mehr oder weniger komplexe Makros sind und deshalb Einbahnstraßen darstellen: Wer ein als ASP angelegtes Projekt später als Windows-Fenster haben will, könnte sich theoretisch zwar an einen Umbau machen – praktisch ist dann aber ein neues Projekt entsprechenden Typs (und das Kopieren einzelner Programmabschnitte) wesentlich sinnvoller.

VS.NET zeigt beim Start auf der *Startseite* eine Liste der zuletzt bearbeiteten Projekte (die direkt nach der Installation natürlich leer ist). Ein Klick auf NEUES PROJEKT ruft den entsprechenden Assistenten auf den Plan. Wie in Abbildung 3.1 zu sehen, will VS.NET Projektdateien standardmäßig im Ordner »Eigene Dateien\Visual Studio-Projekte« unterbringen, was sich über DURCHSUCHEN quasi-permanent ändern lässt: Visual Studio verwendet das an dieser Stelle ausgewählte Basisverzeichnis im Weiteren so lange als Vorgabe, bis Sie erneut DURCHSUCHEN benutzen. Und wo es gerade um Änderungen geht: Auf der Startseite, über die Sie übrigens auch nach einem Wechseln des Basisverzeichnisses an die bereits existierenden Projekte herankommen, findet sich ein Wahlpunkt MEIN PROFIL. Mit ihm legen Sie fest, was Sie beim nächsten Start von VS.NET zu sehen bekommen wollen: die Projektliste, das zuletzt geladene Projekt, ein leeres Editorfenster usw. (Der Königsweg für das Einstellen dieser und weiterer Optionen führt über den Dialog OPTIONEN/UMGEBUNG, aufzurufen über das Menü EXTRAS.)

3.1.1 Programmgerüst und Dateistruktur

Fürs erste Ausprobieren reichen die Standardvorgaben. Wählen Sie also im linken Fenster *Visual C#-Projekte*, im rechten Fenster *Windows-Anwendung*, belassen Sie den Projektnamen auf *WindowsApplication1* und klicken Sie auf OK.

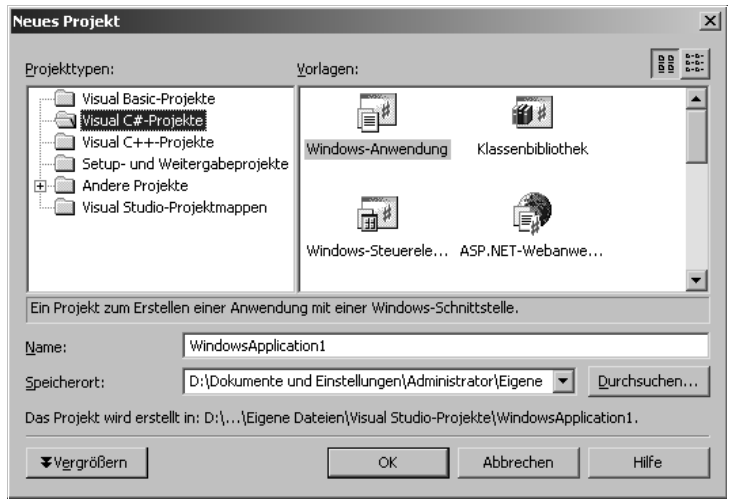


Abbildung 3.1: Der Assistent für neue Projekte nach der Auswahl von *Visual C#-Projekte*

VS.NET erzeugt daraufhin einen Ordner WINDOWSAPPLICATION1 (in *Eigene Dateien\Visual Studio-Projekte*), und hinterlegt darin einige Quelldateien mit dem Codegerüst des neuen Projekts sowie diverse weitere Unterordner. Abbildung 3.2 zeigt eine Exploreransicht der Dateistruktur. (Der Ordner VSMACROS enthält ebendies – Makros für Visual Studio – und ist nicht projektspezifisch; er wurde bei der Installation von Visual Studio .NET angelegt.)

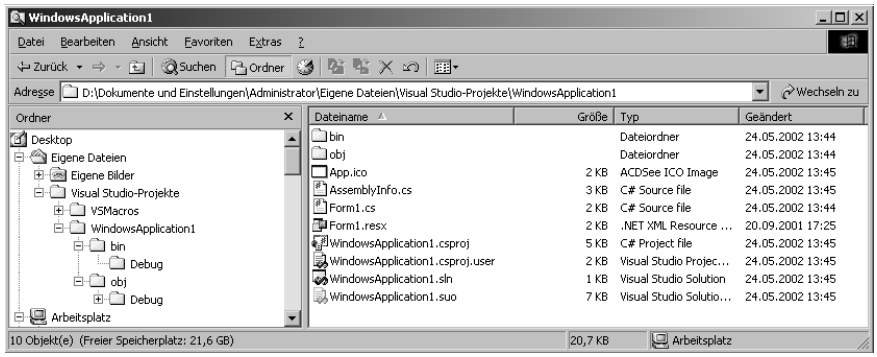


Abbildung 3.2: Die für eine Windows-Anwendung angelegte Verzeichnisstruktur

Im Folgenden ein kurzer Überblick über die einzelnen Dateien:

- *App.ico* – das Icon des Projekts, also das Symbol, mit dem die Anwendung später im Explorer erscheint. Diese Datei kann auch mit einem externen Programm beliebig verändert werden.
- *AssemblyInfo.cs* – das Grundgerüst für ein Assembly, mit dem sich unter anderem das Projekt digital signieren lässt. Die Datei lässt sich mit einem Texteditor (oder eben dem Editor von Visual Studio .NET) bearbeiten und enthält selbst recht ausführliche weitere Informationen.
- *Form1.cs* – das Grundgerüst der Windows-Anwendung (in C#). Den Inhalt dieser Datei bekommen Sie später im Editor zu sehen.
- *Form1.resx* – eine XML-Datei, die die Ressourcen des Formulars bzw. die ihm zugeordneten Komponenten, für Belange des .NET-Laufzeitumgebung beschreiben. Diese Datei wird vom Designer automatisch gepflegt.
- *WindowsApplication1.csproj* – die Konfiguration für das Projekt (Debug und Release, Referenzen auf die .NET-Umgebung). Einen Teil der in dieser Datei niedergelegten Information bekommen Sie später in den Eigenschaftsseiten des Projekts zu sehen.
- *WindowsApplication1.csproj.user* – weitere Konfigurationseinstellungen, hauptsächlich für den Debugger, ebenfalls über die Eigenschaftsseiten des Projekts zu erreichen.
- *WindowsApplication1.sln* – beschreibt, wie die Elemente der vorliegenden Projektmappe (Projekte, Projektelemente, Konfigurationen, Ver-

weise usw.) miteinander zu einer »Lösung« (Solution) verwoben sind. Diese Information ist für Ablauf des Link-Vorgangs wichtig, also den Zusammenbau der Assembly.

- *WindowsApplication1.suo* – beschreibt den aktuellen Zustand der Projektmappe, geöffnete Fenster und deren Arrangement, die Aufgabenliste sowie die sonstigen benutzerspezifische Einstellungen, die VS.NET benötigt, um eine geschlossene Projektmappe bei nächsten Öffnen wieder in den zuletzt vorhandenen Zustand zu versetzen.

Wenn Sie das Projekt später einmal als eigenständige Anwendung ausführen wollen, also nicht innerhalb von VS.NET, finden Sie die EXE-Datei im Unterordner *bin\Debug* bzw. bei einer Compilierung im Release-Modus im Unterordner *bin\Release*.

3.1.2 Entwurf der ersten Windows-Anwendung

VS.NET zeigt nach all diesen Vorbereitungen ein leeres Formular in der Ansicht des Designers und befindet sich im visuell orientierten Entwurfsmodus. Die wesentlichen Aufgaben der drei umliegenden Fenster (vgl. Abbildung 3.3) sind:

- PROJEKTMAPPEN-EXPLORER – dient zur Navigation und zum Auffinden der einzelnen Elemente in der Projektmappe. Über das per Rechtsklick erreichbare Kontextmenü wechseln Sie beispielsweise bei Form1.cs zwischen dem visuellen Entwurfsmodus (DESIGNER ANZEIGEN) und dem Texteditor (CODE ANZEIGEN) mit dem C#-Quelltext des Formulars.
- KLASSENANSICHT – findet sich im gleichen Fenster wie der Projektmappen-Explorer gelegen und gibt den internen Zusammenhang aller in der Projektmappe befindlichen Projekte bis in die Ebene der einzelnen Klassenelemente wieder.
- EIGENSCHAFTEN – stellt die Eigenschaften (auch Ereignisse und Eigenschaftsseiten) des im Designer oder Projektmappen-Explorer ausgewählten Elements dar und ermöglicht es, sie zu manipulieren.
- AUFGABENLISTE – enthält die Fehlermeldungen und Warnungen des jeweils letzten Compilerlaufs sowie die im Quelltext verstreuten Tokenkommentare (wie `// TODO ...`) als Hyperlinks, die direkt (per Doppelklick) an die jeweilige Stelle im Quelltext führen.

Eine Schaltfläche in das Formular einfügen

Am linken Rand des Fensters findet sich eine Leiste, über die weitere Hilfsfenster erreichbar sind. Um in das leere Formular einige Windows-Steuerelemente einzusetzen, benötigen Sie das TOOLBOX-Fenster, das über das Menü ANSICHT/TOOLBOX, den Tastenbefehl Strg+Alt+X oder – falls vorhan-

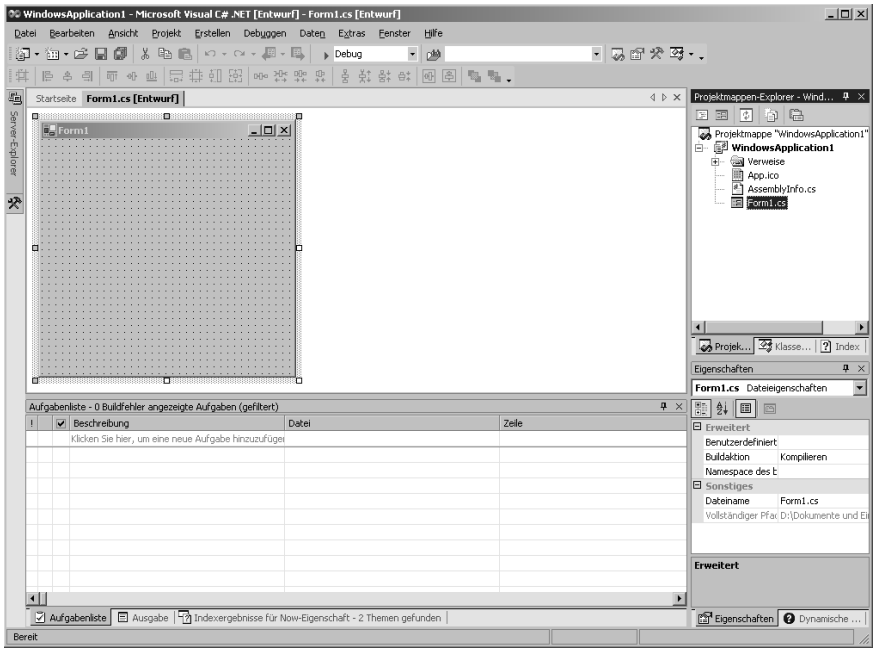


Abbildung 3.3: Formular für eine frisch generierte Windows-Anwendung

den – durch einen schlichten Klick auf den Reiter in der Leiste einzublenden ist. Die Pin-Schaltfläche in der Titelleiste fixiert das Fenster und bzw. verbannt es wieder in die Leiste. Zum Einfügen einer Schaltfläche klicken Sie in der Toolbox auf *Windows Forms*, dann auf *Button*, und schließlich im Formularentwurf auf die Stelle, an der die Schaltfläche im Fenster erscheinen soll (Abbildung 3.4).



Wenn die Schaltfläche lediglich als schwarzes Klötzchen (oder gleich gar nicht) erscheint, haben Sie die Maus beim Klicken bewegt, also gezogen – und das interpretiert der Editor als Größeneinstellung des aktuellen Elements. Werfen Sie gegebenenfalls einen Blick ins Fenster EIGENSCHAFTEN*: Wenn dort* `button1` *steht, können Sie die Größe der Schaltfläche auch über die Eigenschaft* `Size` *per numerischer Eingabe Ihren Wünschen anpassen.*

Der Schaltfläche eine Aktion zuordnen

Um dieser Schaltfläche eine Aktion zuzuordnen, doppelklicken Sie darauf oder (allgemeiner) klicken im EIGENSCHAFTEN-Fenster auf das Blitzsymbol, suchen in der daraufhin erscheinenden Liste für Ereignisse den Eintrag `Click` und doppelklicken in der rechten (im Moment noch leeren) Spalte neben diesem Eintrag. VS.NET fügt daraufhin das Codegerüst einer Methode namens `button1_Click()` in den Quelltext ein, schaltet in den Texteditor um und setzt die Schreibmarke gleich an die richtige Stelle, damit Sie die Aktion sofort implementieren können (Abbildung 3.5).

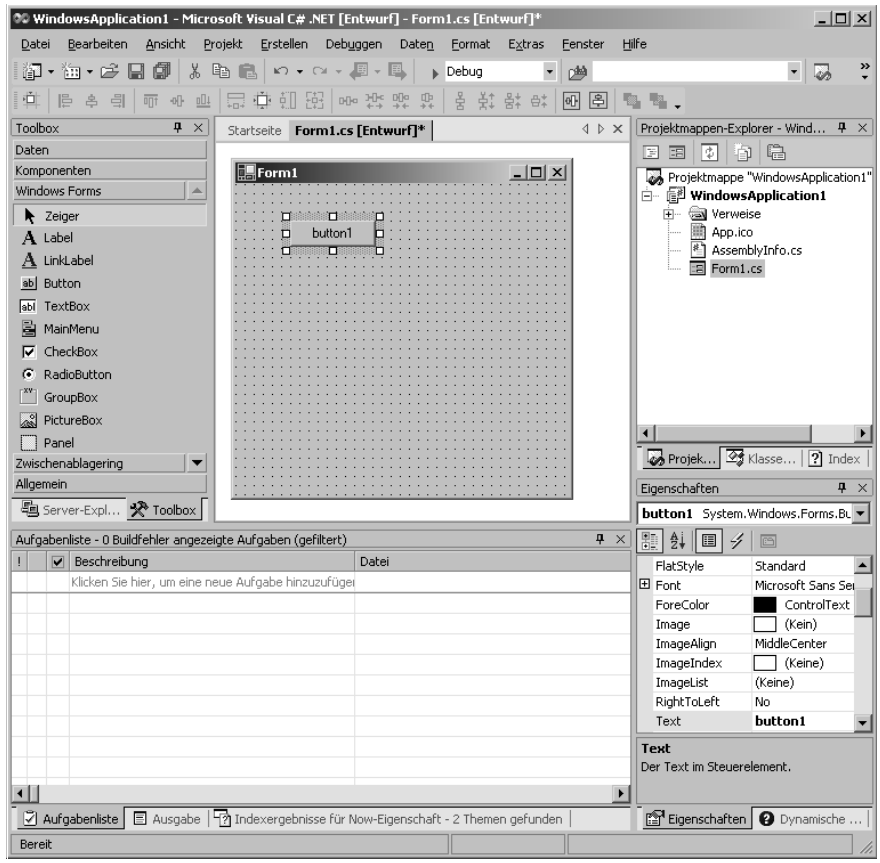


Abbildung 3.4: Das Formular mit einer eingefügten Schaltfläche, deren Eigenschaften über das gleichnamige Fenster veränderbar sind

Der Designer beschränkt sich (nicht nur bei derartigen Ereignissen) auf das Erzeugen des notwendigen Drumherums. Diese Methode ergänzen Sie nun um eine »echte« Aktion, nämlich die Ausgabe eines Meldungsfensters per `MessageBox`:

```
private void button1_Click(object sender, System.EventArgs e)
{
  MessageBox.Show ("Klick!", "Die erste Demo-Anwendung");
}
```

Anwendung übersetzen und starten

Der Menübefehl DEBUGGEN/STARTEN – ersatzweise F5 – weist VS.NET an, die Quelldateien dieses Jahrhundertwerks zu speichern, in Zwischencode zu übersetzen, mit den Laufzeitbibliotheken zu einer EXE-Datei zu binden und schließlich, wenn alles gut gegangen ist, auszuführen. (Wer von Microsoft Visual C/C++ her kommt, darf sich darüber freuen, dass die Rückfrage »Dateien x.c und y.cpp wurden verändert. Neu übersetzen?« endlich in den Bytehimmel gewandert ist.) Das Ergebnis zeigt Abbildung 3.6.

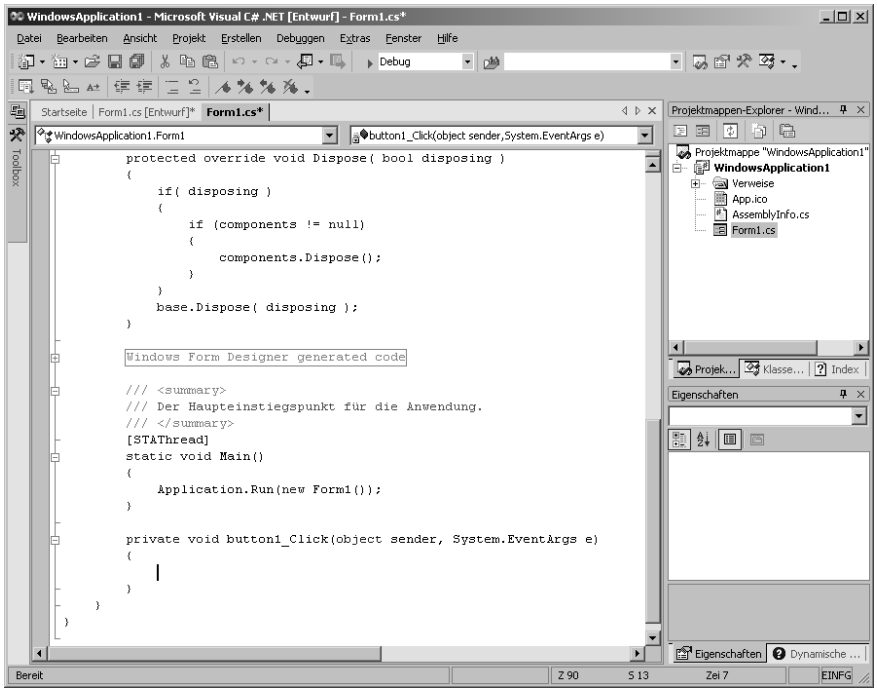


Abbildung 3.5: Eine vom Designer erzeugte Methode für Klicks auf *button1*

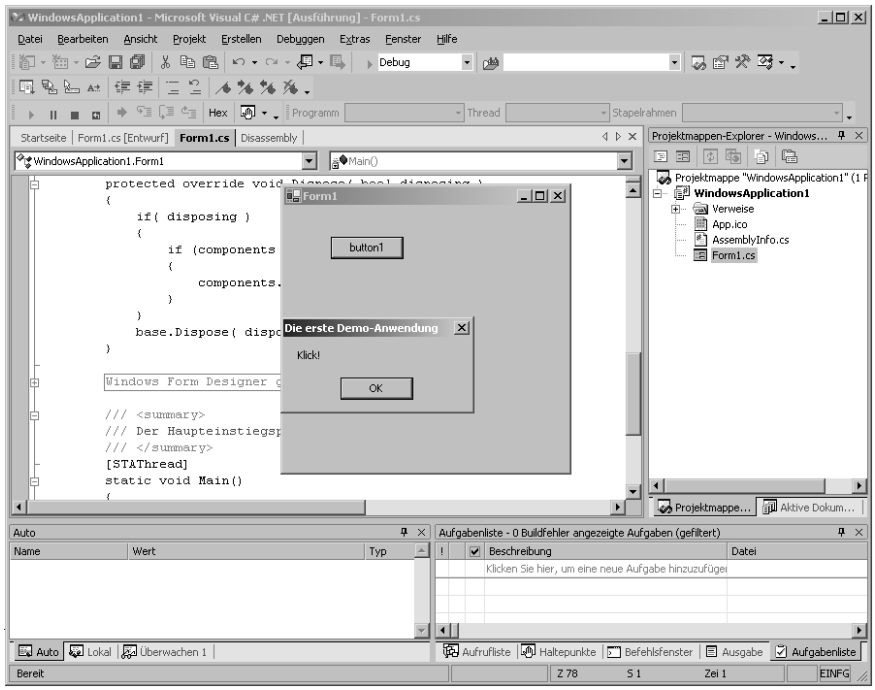


Abbildung 3.6: Das erste Programm, innerhalb von VS.NET ausgeführt

Das vom Windows Explorer aus ausführbare Programm, also die EXE-Datei *WindowsApplication1.exe*, finden Sie wie bereits erwähnt in *Eigene Dateien\Visual Studio-Projekte\WindowsApplication1\bin\Debug*.

Da .NET praktisch alle Elemente der grafischen Benutzeroberfläche von Windows als Komponenten verkapselt, wobei ihre Daten als Eigenschaften verfügbar sind, und Visual Studio .NET die Definition von Methoden zur Behandlung von Ereignissen per Klick möglich macht, geht das Erstellen von Windows-konformen Oberflächen hier genauso flüssig von der Hand wie bei VB 6 und Delphi. (Wer an Visual C/C++ gewöhnt ist, wird sich nach einer kurzen Einarbeitung höchstens fragen, wieso sein Hauscompiler dasselbe immer noch so entsetzlich umständlich macht.)

Kleiner Ausblick

Das folgende Fragment demonstriert die – zumindest an der Oberfläche – ausgesprochen simple »Verdrahtung« zweier Steuerelemente. Auf einen Klick in einem Listenfeld hin setzt es die Beschriftung einer Schaltfläche mit dem im Listenfeld ausgewählten Text:

```
private void listBox1_SelectedIndexChanged(object sender,
   System.EventArgs e)
{
    button1.Text = listBox1.Items[listBox1.SelectedIndex].ToString();
}
```

Wenn Sie eine Viertelstunde Zeit haben, dann können Sie das in diesem Abschnitt vorgestellte Projekt ja einmal entsprechend erweitern, oder – selbstverständlich nur um des Lerneffekts willen – dafür sorgen, dass `button1` auf Mausklicks hin seine Position im Formular so ändert, dass der Benutzer daneben getroffen hat. (Wer `button1_Click` entsprechend ausbauen will, sollte sich die Methoden `MousePosition` und `PointToClient` näher ansehen – deren Aufruf man sich nach der Zuordnung einer Methode `button1_MouseDown` allerdings sparen kann, weil bei dieser Methode die Mauszeigerposition als Parameter mitgeliefert wird. Die aktuelle Größe des Formulars lässt sich über `this.ClientSize` abfragen, der Rest sollte sich durch Studium der Methode `InitializeComponent` in der Datei *Form1.cs* erschließen – dort wird das Setzen der Position von `button1` vorexerziert.)

3.2 Konsolenanwendungen

Auch wenn sicher niemand mehr versuchen wird, eine Textverarbeitung für die Konsole (die in älteren Windows-Versionen noch weniger vornehm »DOS-Box« hieß) zu schreiben: In C# ist diese Art von Anwendungen nicht nur für das schnelle Ausprobieren syntaktischer Konstrukte und Sprachfeatures gut. Tatsächlich eignet sich eine Konsolenanwendung aber nicht zuletzt aufgrund der kürzeren Kompilierungszeit und des einfacheren Code-

gerüsts wunderbar als abschließende, praktische Antwort auf Syntax- oder Semantikfragen (und die Autoren geben gerne zu, dass fast alle Syntaxbeispiele in diesem Buch in einem Projekt *ConsoleApplicationxx* eingetippt worden sind).

Angelegt werden Konsolenprogramme in VS.NET mit denselben Schritten wie Windows-Anwendungen – nur dass Sie im Dialogfeld NEUES PROJEKT ein wenig weiter nach unten rollen müssen (siehe Abbildung 3.7). Die von VS.NET für Konsolenprogramme angelegte Ordnerstruktur ist dieselbe wie für Windows-Anwendungen, die ausführbare Datei findet sich hinterher im Unterordner *bin\debug* bzw. *bin\Release*.

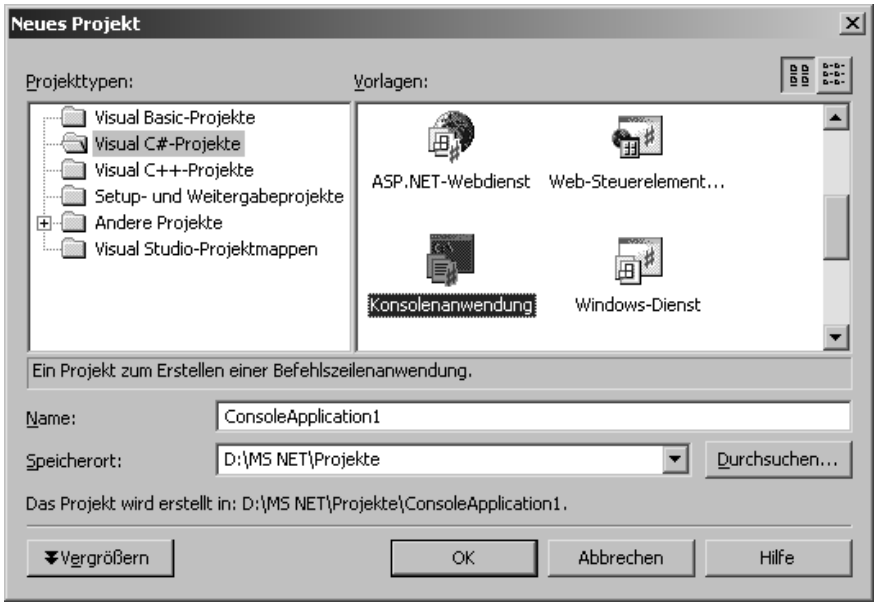


Abbildung 3.7: Anlegen eines Konsolenprogramms

Der vom Assistenten erzeugte Code ist allerdings um einiges kürzer und kann hier ohne weiteres komplett wiedergegeben werden.

```
using System;

namespace ConsoleApplication1
{
  /// <summary>
  /// Zusammendfassende Beschreibung für Class1.
  /// </summary>
  class Class1
  {
    /// <summary>
    /// Der Haupteinstiegspunkt für die Anwendung.
    /// </summary>
    [STAThread]
```

```
    static void Main(string[] args)
    {
      //
      // TODO: Fügen Sie hier Code hinzu, um die Anwendung zu starten
      //
    }
  }
}
```

Um aus diesem Gerüst das übliche »Hello, world«-Programm zu machen, ersetzen Sie den »TODO«-Kommentar im Rumpf der Methode `Main` durch:

```
Console.WriteLine("Hallo, Welt!");
Console.ReadLine();
```

Mit `WriteLine()` lässt sich noch einiges mehr anstellen als nur vorgefertigte Stringparameter auszugeben. Tatsächlich ist diese Methode für Alles und Jedes überladen – insgesamt 19 Mal. In ihrer für den praktischen Einsatz wichtigsten Variante ermöglicht die Methode die formatierte Ausgabe beliebig vieler Parameter beliebigen Datentyps, die durch einen im ersten Parameter zu übergebenden Formatstring beschrieben werden (C-Programmierer werden sich hier an `printf()` erinnert fühlen.) Die Syntax für den Formatstring erlaubt Platzhalter für den zweiten bis n-ten Parameter der Methode, wobei der zweite Parameter durch {0}, der dritte durch {1} und so weiter symbolisiert wird. Zusätzliche, durch Komma oder Doppelpunkt abzutrennende Formatsymbole in den geschweiften Klammern ermöglichen Formatierungen wie eine bestimmte Anzahl von Dezimalstellen nachdem Komma, das Auffüllen mit Leerzeichen bis zu einer bestimmten Breite etc. (Die Online-Hilfe bringt hier unter dem Stichwort »Formatieren von Zahlen« sowie zur Methode `String.Format()` eine Reihe aussagekräftiger Beispiele.)

Ersatzweise könnte man die Ausgabe des Programms also auch so zusammenschustern:

```
string Greeting = "Hallo";
Console.WriteLine("{0},{1,5}!", Greeting, "Welt");
```

Beachten Sie, dass das Leerzeichen in der Ausgabe hier über die Längenangabe 5 für den dritten Parameter der Methode zustandekommt.

Der `ReadLine()`-Aufruf wartet auf eine Eingabe beliebiger Länge, die mit einem [↵] abgeschlossen sein muss, verwirft die eingelesenen Zeichen sofort wieder und hat hier nur einen einzigen Zweck: Das mittels [F5] im Debugger gestartete Programm so lange aufzuhalten, bis der Benutzer die vorangehenden Ausgaben bewundert hat. Ärgerlicherweise öffnet VS.NET nämlich für Konsolenprogramme ein eigenes Konsolenfenster, lässt das Programm ablaufen und schließt dieses Konsolenfenster sofort nach Ende des Programms wieder. Erfolgt der Start der Anwendung hingegen mit dem

Menübefehl DEBUGGEN/STARTEN OHNE DEBUGGER (bzw. Strg+F5), ist `ReadLine()` nicht erforderlich, weil die Laufzeitumgebung dann von sich aus eine entsprechende Abfrage einsetzt, wie in Abbildung 3.8 zu sehen.



Abbildung 3.8: Ausgabe des ersten Konsolenprogramms

3.2.1 Kommandozeilenparameter

Konsolenanwendungen werden üblicherweise mit Kommandozeilen-Parametern gesteuert, die dann ihrerseits als Array `args` der Methode `Main` verfügbar sind. Das Festlegen solcher Kommandozeilen-Parameter für Testläufe innerhalb der Entwicklungsumgebung ist in VS.NET leider derartig gut versteckt, dass wir nicht darum herum kommen, dieses einführende Kapitel mit einer detaillierten Anleitung zu versehen:

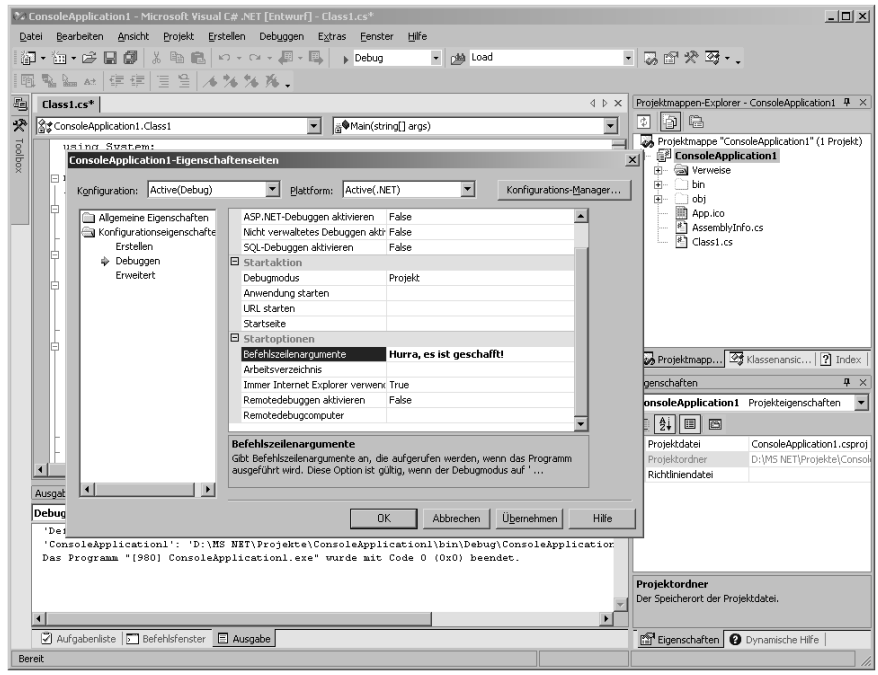
1. Wählen Sie im PROJEKTMAPPEN-EXPLORER den Projektnamen der Anwendung (hier: *ConsoleApplication1*) aus.
2. Ein Rechtsklick auf dem Projektnamen bringt ein Kontextmenü zum Vorschein, in dem Sie den Befehl EIGENSCHAFTEN wählen. Alternative Wege sind ANSICHT/EIGENSCHAFTSSEITEN, der Tastenbefehl ⇧+F4 und ein Klick auf das rechte der vier Symbole im Fenster EIGENSCHAFTEN.
3. Wählen Sie im daraufhin erscheinenden Dialogfeld CONSOLEAPPLICATION1-EIGENSCHAFTSSEITEN im linken Fenster den Ordner *Konfigurationseigenschaften* und dort den Unterpunkt *Debuggen*.
4. Wenn Sie nun das rechte Fenster dieses Dialogfeldes bis zum Punkt *Startoptionen* rollen, kommt schlussendlich ein Feld *Befehlszeilenargumente* in Sicht (siehe Abbildung 3.9).

Es wäre sicher nett, wenn sich diese Einstellung auch direkt über das Fenster EIGENSCHAFTEN erreichen ließe. Wie Abbildung 3.9 sozusagen nebenbei zeigt, ist dieses Fenster in der aktuellen Version von VS.NET sowieso nur recht spärlich besetzt.

Der folgende Codeauszug zeigt der Vollständigkeit halber den programmgesteuerten Zugriff auf solche Parameter:

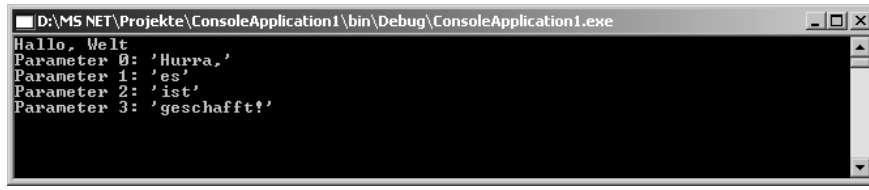
```
for (int x = 0; x < args.Length; x++)
  Console.WriteLine("Parameter {0}: '{1}'", x, args[x]);
```

Abbildung 3.9: Fast schon ein easter egg – Definition von Kommandozeilenparametern in VS.NET



In Abbildung 3.10 finden Sie schließlich das Ergebnis. Wie dort zu sehen, liefert die .NET-Umgebung im Element mit dem Index 0 schlicht den ersten Kommandozeilenparameter – und tanzt damit aus der Reihe der meisten anderen Programmiersprachen, die ursprünglich einmal DOS zur Grundlage hatten: Dort ist das Element 0 von *args* (bei Delphi: *ParamStr*) für den Namen der ausführbaren Datei reserviert und deshalb immer besetzt. In einer C#-Anwendung kann `args` dagegen ohne weiteres 0 Elemente enthalten.

Abbildung 3.10: Ausgewertete Kommandozeilenparameter





Windows-Anwendungen bekommen ebenfalls Kommandozeilenparameter zu sehen, soweit sie beim Aufruf angegeben werden – etwa über eine Verknüpfung auf dem Desktop. Wer diese Möglichkeit in einer Windows-Anwendung vorsehen will, muss in VS.NET exakt dieselben Einstellungen vornehmen wie für ein Konsolenprogramm. Des Weiteren ist ein Nachtrag der Parameterdeklaration `string[] args` *für die Methode* `Main()` *erforderlich: Der Assistent vereinbart diese Methode leider in der parameterlosen Variante.*

3.3 Webdienste

Die zusammen mit .NET eingeführten Webdienste stellen einen höchst eleganten Weg zur losen Koppelung von Systemen dar, die – im Gegensatz zu Microsofts DCOM – auch übers Internet und damit über die Grenzen von Firewalls hinweg funktioniert, weil der Datentransfer auf bekannt »harmlosen« Protokollen wie HTML, XML und SOAP aufbaut. Was man damit anfangen kann, lässt sich am einfachsten an einem Beispiel verdeutlichen: Stellen Sie sich eine Webseite vor, die für eine Suche nach Artikeln die ersten zehn Fundstellen in einer Liste darstellt. Mit herkömmlichen Techniken wird diese Seite serverseitig per CGI oder ASP generiert, und enthält eine Schaltfläche, deren Funktion man nur so umschreiben kann: »Anweisung an den Server: Komplette HTML-Seite erneut per CGI oder ASP generieren, Suchlauf in der Datenbank erneut ausführen, diesmal die ersten 10 Ergebnisse verwerfen und das zweite Zehnerpack an Fundstellen einbauen. Danach alles erneut übertragen.« Auch wenn diese Beschreibung unterschlägt, dass sich Tausende von Entwicklern den Kopf darüber zerbrochen haben, wie man diese katastrophale Verschwendung an Rechenzeit und Datenaufkommen einigermaßen in den Griff bekommt, und dabei natürlich nicht ganz ohne Erfolg geblieben sind: Die »Interaktivität« von Webseiten basiert nun einmal weitgehend darauf, dass der Server auf Eingaben und Mausklicks hin eine neue HTML-Seite generiert.

Wie praktisch wäre es doch, in einer HTML-Seite eine visuelle Komponente unterzubringen, die die Abfrage des nächsten darzustellenden Elements über den schlichten Aufruf einer serverseitigen Funktion erledigt, wobei Parameter und die zu erwartenden Ergebnisse in der Syntax von C# formuliert werden – mithin in derselben Weise wie bei einem Methodenaufruf. Nun, genau so sieht ein Webdienst für den Programmierer aus – und zwar sowohl auf der Client- als auch auf der Serverseite. Mehr noch: Webdienste lassen sich sowohl von (im Internet Explorer ablaufenden ASP.NET-Anwendungen) als auch von schlichten (.Net-konformen) Windows-Anwendungen aus gleichermaßen nutzen.

Angelegt werden Projekte für Webdienste in VS.NET mit denselben Schritten wie Windows-Anwendungen und Konsolenprogramme, wobei hier allerdings bereits im Dialogfeld NEUES PROJEKT der erste Unterschied auffallen sollte (vgl. Abbildung 3.11): Die Standardvorgabe für den Speicherort ist ein Unterverzeichnis von `http://localhost` – und wie nunmehr unschwer zu erraten, setzen Projekte dieser Art einen laufenden Internet Information Server auf dem Entwicklungssystem voraus. (Dessen Standardkonfiguration sorgt wiederum dafür, dass dieser URL zu `c:\inetpub\wwwroot` aufgelöst wird.)

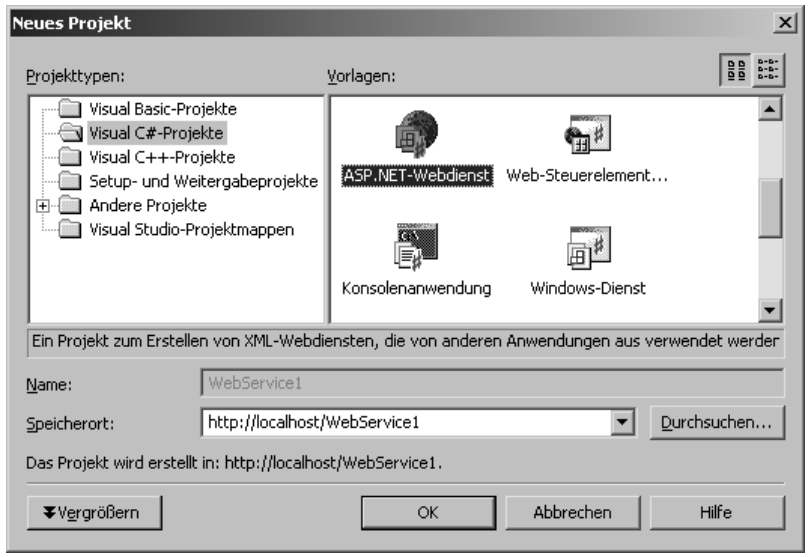


Abbildung 3.11: Anlegen eines Projekts für einen Webdienst

INFO

Die Version des Internet Information Server, die sich auf der Installations-CD von Windows 2000 findet, hat eine Reihe von Sicherheitslücken, die von Viren wie Nimda ausgenutzt werden. Eine entsprechende Aktualisierung per Windows-Update ist deshalb auch dann ein unbedingtes Muss, wenn Sie eine Firewall vorgeschaltet haben. Wer auf Nummer Sicher gehen will, führt dieses Windows-Update direkt nach der Installation aus, und schaltet den IIS vor dem Herstellen der Internetverbindung erst einmal über START/PROGRAMME/VERWALTUNG/DIENSTE ab. (Ärgerlicherweise ist dieser Tipp nicht nur etwas für ausgesprochene Paranoiker: Im April 2002 war eine Unzahl von Systemen weltweit von diesem Virus befallen, und einen der Autoren hat es wenige Minuten nach einer Neuinstallation wg. Nimda ein zweites Mal erwischt, nämlich während des allfälligen Windows-Updates.)

INFO

Im Rahmen der Installation legt VS.NET ein lokales Benutzerkonto mit dem Namen VS Developers an, das Schreibrechte für wwwroot hat. Über dieses Konto (und das http-Protokoll) werden sämtliche Operationen mit Webdienst-Projektdateien abgewickelt.

Für das Anlegen von Projekten für einen neuen Webdienst nimmt sich VS.NET etwas mehr Zeit. Ein Grund dafür dürfte sein, dass die Dateizugriffe ausschließlich per http geschehen, ein zweiter, dass dabei eine komplette Baumstruktur entsteht (deren Ordner allerdings überwiegend leer sind). Das Ergebnis einer erfolgreichen Kompilierung ist eine (ISAPI-)DLL im Ordner *\localhost\bin*; separateVerzeichnisse für Debug- und Release-Versionen gibt es hier nicht.

3.3.1 Implementation eines Dienstes

Für einen neu angelegten Webdienst zeigt VS.NET erst einmal (wie bei einer Windows-Anwendung) den Designer, über den Sie nun aus der Toolbox oder dem Server-Explorer Komponenten in das Projekt einsetzen könnten. Visuelle Komponenten wie Schaltflächen sind zwar zulässig, haben aber nicht viel Sinn, da der Webdienst auf dem Server ausgeführt wird. Tatsächlich geht es hier in erster Linie um Komponenten für den Zugriff auf Datenbanken und damit um ein Thema, das den Rahmen dieses Einführungskapitels sprengen würde.

Nach dem Umschalten auf die Codeansicht zeigt VS.NET ein (im Moment noch auskommentiertes) Beispiel für den einfachsten aller Dienste: eine parameterlose Methode, die einen fixen String zurückgibt (vgl. Abbildung 3.12).

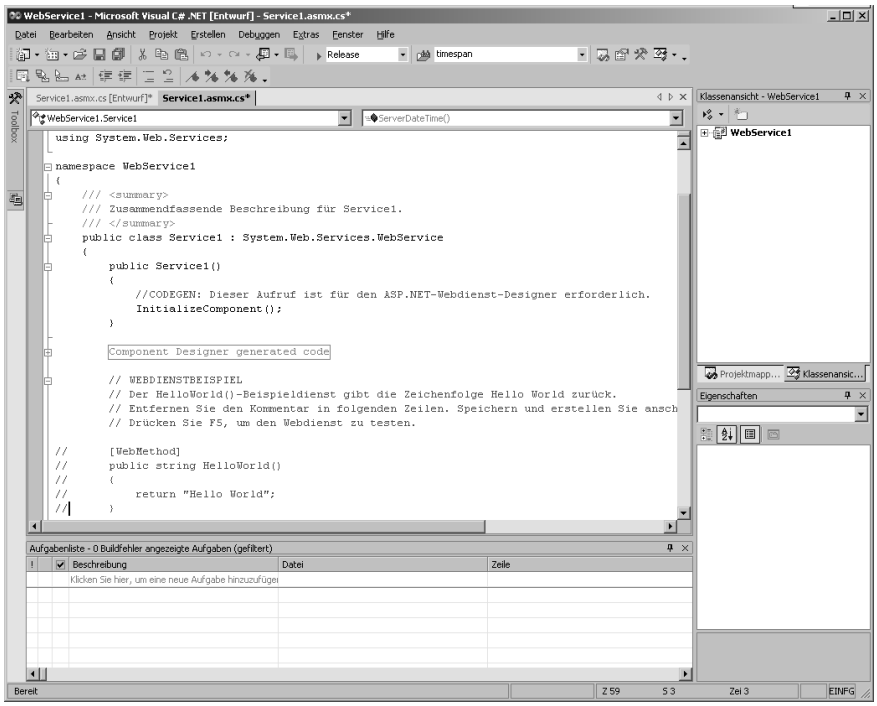


Abbildung 3.12: Der Assistent gibt einen Webdienst als Beispiel in auskommentierter Form vor.

Zur Demonstration entfernen Sie die Kommentarzeichen (alle fünf – also auch das vor `[WebMethod]`) und ergänzen den Quelltext um zwei weitere Methoden:

```
[WebMethod]
public string HelloWorld()
{
  return "Hello World";
}
```

```
[WebMethod]
public string ServerDateTime()
{
  return "Aktuelle Serverzeit ist: " + System.DateTime.Now.ToString();
}

[WebMethod]
public bool ServerPassword(string Username, string PW)
{
  return (Username == "Fritz") && (PW == "geheim");
}
```

Tatsächlich ist das bereits alles, was Sie tun müssen – und zwar nicht nur zur Definition dreier Webdienste, sondern auch für die notwendigen Tests: Bei der Compilierung (am einfachsten: per F5) erzeugt VS.NET selbstständig einen Client in Form einfacher HTML-Seiten, der den Aufruf dieser drei Funktionen ausführt. Dass `ServerPassword` zwei Strings als Parameter erwartet, bekommt VS.NET ebenfalls mit: Wie in Abbildung 3.13 zu sehen, enthält die HTML-Seite entsprechende Eingabefelder.

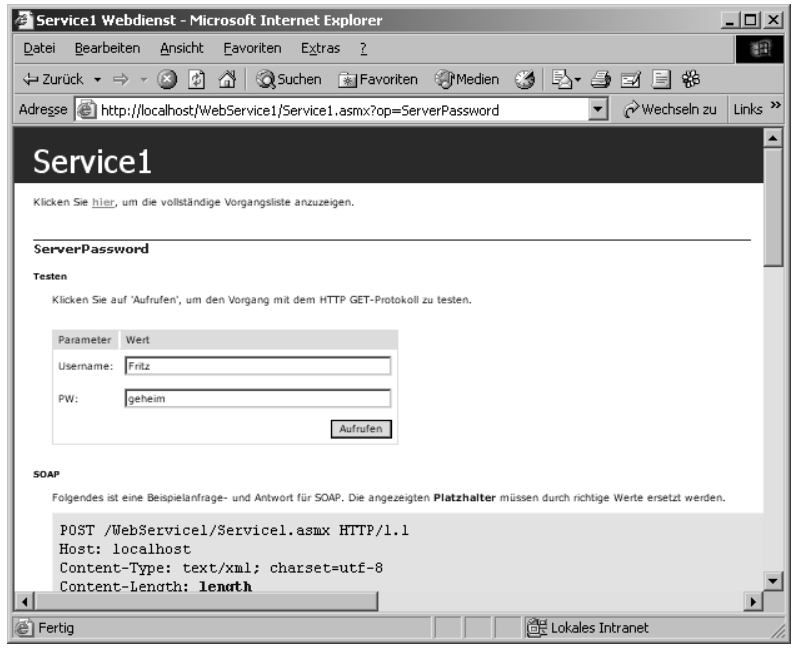


Abbildung 3.13: Von VS.NET für einen Webdienste generierte HTML-Seiten mit Eingabefeldern für die Parameter

Abbildung 3.14 zeigt das Ergebnis des Funktionsaufrufs in XML. Die automatisch generierte Testumgebung beschränkt sich auf die Darstellung dieses Ergebnisses in einem eigenen Fenster des Internet Explorers und ein schlichtes `http GET`. Die für den Aufruf des jeweiligen Diensts erzeugte Seite enthält ein weitgehend ausgearbeitetes Quelltext-Beispiel sowohl für http als auch für SOAP.



Abbildung 3.14: Das Ergebnis der Abfrage

3.3.2 Ein Webdienst-Client

Der in diesem Abschnitt vorgestellte ASP.NET-Client basiert auf WebForms und ist zwar vom Frontend einer e-Commerce-Website ähnlich weit entfernt wie die zuvor besprochenen Webdienste, demonstriert aber das Prinzip recht anschaulich: Er erlaubt die Abfrage der Serverzeit – also den Aufruf des Webdienstes `ServerDateTime` – erst nach einer korrekten Anmeldung. (Wer jetzt stutzt, weil `ServerDateTime` wie zuvor demonstriert voraussetzungslos aufgerufen werden kann, mithin die »Sicherheit« hier clientseitig implementiert ist, hat nicht nur den vorangehenden Abschnitt gut verstanden, sondern auch recht – aber schließlich geht es um eine Demonstration.)

Zum Anlegen eines ASP.NET-Clients rufen Sie den Assistenten mit DATEI/NEU/PROJEKT auf den Plan und wählen im rechten Fenster ASP.NET-Webanwendung. Wie in Abbildung 3.15 zu sehen, schlägt VS.NET auch hier `http://localhost` als Speicherort vor. Belassen Sie es bei den vorgeschlagenen Namen.

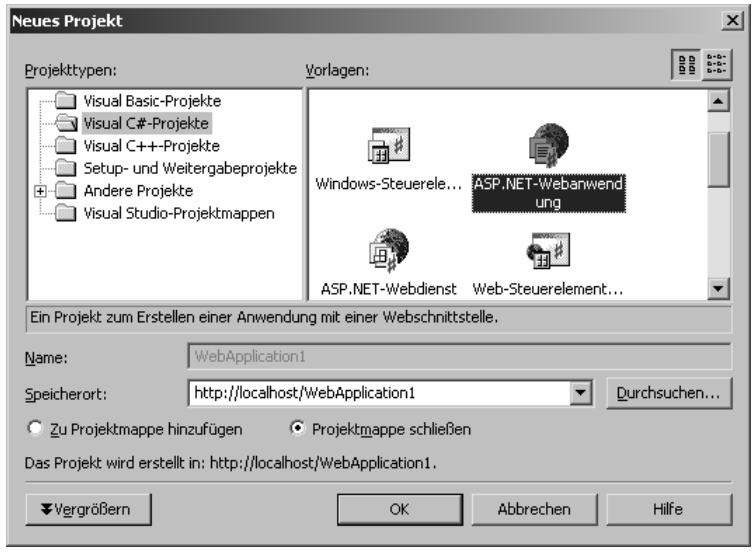


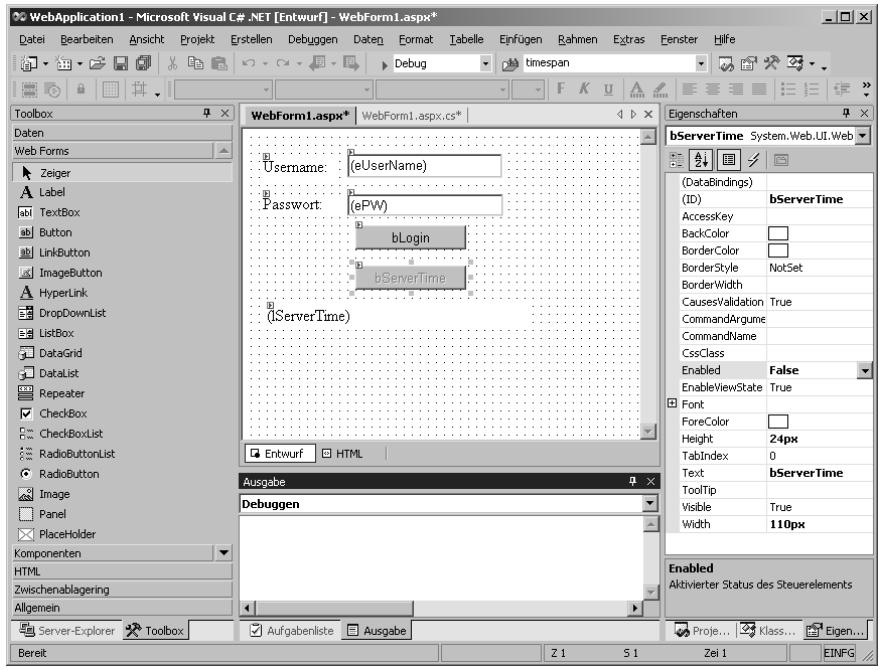
Abbildung 3.15: ASP.NET-Clients werden ebenfalls in http://localhost gespeichert.

Nachdem der Assistent seine Arbeit getan hat (was auch hier wieder einen Moment dauern kann), zeigt VS.NET ein leeres Webformular. Aktivieren Sie die Toolbox (Strg+Alt+X), wählen Sie dort WEBFORMS und schmücken Sie das Formular mit insgesamt fünf Komponenten: Zwei Labels (IDs: lUsername, lPW), zwei Texteingabefeldern (eUsername, ePW), einem dritten Label (lServerTime) und schließlich zwei Schaltflächen (bLogin, bServerTime). In Abbildung 3.16 haben wir für die Eingabefelder und die Schaltfläche die Komponentennamen zusätzlich als Eigenschaft *Text* eingetragen, um die Zusammenhänge deutlich zu machen. Wie dort zu sehen, ist die Eigenschaft `bServerTime.Enabled` auf `false` gesetzt: diese Schaltfläche soll erst verwendbar sein, wenn sich der Benutzer angemeldet hat.



Wer an Visual Basic oder Delphi gewöhnt ist, wird mit dem Einsetzen einer Handvoll Standardkomponenten in ein Formular wohl kaum Schwierigkeiten haben. Für leidgeplagte C/C++-Programmier gilt: Wer die Windows-Beispielanwendung überblättert hat, möge einige Seiten zurückgehen. Das Einsetzen visueller Komponenten und die Zuordnung von Methoden zur Ereignisbehandlung ist dort detailliert beschrieben.

Abbildung 3.16: Entwurf des Client-Formulars





Beim Design von Windows-Anwendungen gibt VS.NET Standardgrößen für Labels, Schaltflächen usw. vor, weshalb man dort mit einem einfachen Klick zum Einsetzen solcher Elemente auskommt. Bei WebForms ist dagegen für all diese Elemente das Aufziehen eines Rahmens verlangt. Mit einer Ausrichtung am Raster (fünftes Symbol in der zweiten Zeile) tut man sich leichter bei der Größenbestimmung.

3.3.3 Aufruf des Webdienstes ServerTime

Welche Aktion der Schaltfläche `bServerTime` zuzuordnen ist, dürfte sich bereits aus ihrem Namen ergeben: Der Aufruf des Webdienstes und das Einsetzen des Ergebnisses in die Eigenschaft *Text* des Labels `lServerTime`. Wobei natürlich die Frage zu klären bleibt, wie man an den Webdienst herankommt.

Die erste Teilantwort darauf ist der Menübefehl PROJEKT/WEBVERWEIS HINZUFÜGEN, der das gleichnamige Dialogfeld auf den Plan ruft, in dem Sie nun – eine aktive Internetverbindung vorausgesetzt – weltweit nach aktiven Webdiensten suchen können. Da es im Moment aber um einen Dienst geht, der ausschließlich lokal auf Ihrem System verfügbar ist, müssen Sie den URL selbst einsetzen: `http://localhost/webservice1/service1.asmx`. Wie in Abbildung 3.17 zu sehen, zeigt das Dialogfeld danach die Auswahlseite für die im vorangehenden Abschnitt erstellten Webdienste an. Schließen Sie das Dialogfeld mit einem Klick auf VERWEIS HINZUFÜGEN.

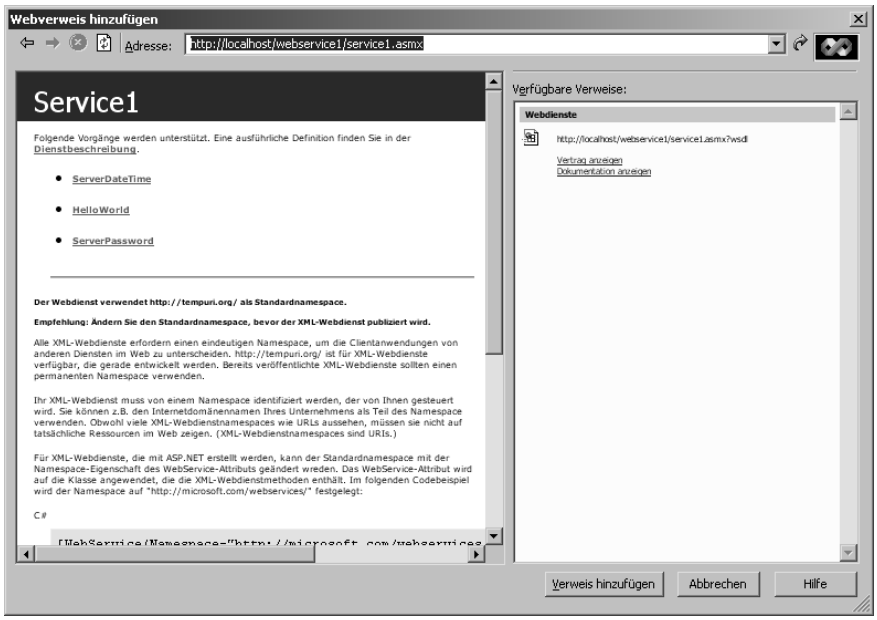


Abbildung 3.17: Über die direkte Eingabe der lokalen URL sind die selbst erstellten Webdienste erreichbar und lassen sich dem Projekt als Verweis hinzufügen.

In der Projektmappe sollten Sie nun einen zusätzlichen Teilbaum *Webverweise/localhost* sehen, der zwei Beschreibungsdateien (service1.disco und service1.wsdl) enthält. Abbildung 3.18 zeigt die Veränderungen im Klassensystem der Anwendung: Der Assistent hat eine Klasse `Service1` erzeugt, die eine Hüllfunktion übernimmt: Ihre Methoden, zu denen auch `ServerDateTime` und `ServerPassword` gehören, erzeugen Serveraufrufe im SOAP-Format, lesen die per XML zurückgegebenen Ergebnisse und geben ihrerseits Daten im gewohnten Format von C# zurück.

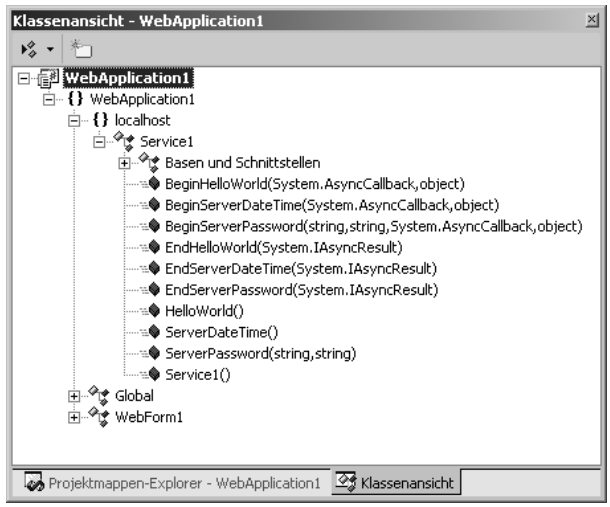


Abbildung 3.18:: Die von VS.NET generierte Hüllklasse Service1 mit ihren Methoden

Klicks auf bServerTime

Was nach diesen Vorbereitungen noch zu tun bleibt, ist größtenteils Verdrahtungsarbeit. Beim Klick auf `bServerTime` soll der Webdienst `ServerTime` aufgerufen und der zurückgegebene String über das Label `lServerTime` dargestellt werden.

Wählen Sie also `bServerTime` im Formular aus und klicken Sie im Fenster EIGENSCHAFTEN auf das Blitzsymbol. Ein Doppelklick in der rechten Spalte neben dem Ereignis *Click* erzeugt die Methode `bServerTime_Click`, deren Rumpf Sie folgendermaßen ausfüllen:

```
private void bServerTime_Click(object sender, System.EventArgs e)
{
  // Instanz der Hüllklasse anlegen
  localhost.Service1 ws = new localhost.Service1();

  // Aufruf eines Webdiensts
  lServerTime.Text = ws.ServerDateTime;
}
```

Wer will, kann `bServerTime.Enabled` einmal kurz auf `true` zurücksetzen und das Programm danach per F5 ausprobieren.

3.3.4 Aufruf des Webdienstes ServerPassword

Der Aufruf dieses Dienstes läuft syntaktisch nach demselben Strickmuster wie der Aufruf von `ServerTime`. Die Eigenschaft `Enabled` der beiden Schaltflächen wird abhängig vom Funktionsergebnis gesetzt. Verpassen Sie `bLogin` also nach demselben Muster wie `bServerTime` eine Methode `Click`, deren Rumpf dann folgendermaßen ausgefüllt wird:

```
private void bLogin_Click(object sender, System.EventArgs e)
{ bool LoggedIn;

  // Instanz der Hüllklasse anlegen
  localhost.Service1 ws = new localhost.Service1();

  // Aufruf von ServerLogin, mit Übergabe der
  // Werte von eUserName und ePW als Parameter
  LoggedIn = ws.ServerPassword(eUserName.Text, ePW.Text);

  bLogin.Enabled = !LoggedIn;
  bServerTime.Enabled = LoggedIn;
}
```

Wie zu sehen, stellt sich auch der Webdienst `Serverpassword` dem Programm als normale Klasse mit Methoden dar. Abbildung 3.19 zeigt schließlich das Ergebnis.



Abbildung 3.19: Das fertige Webformular nach einem gelungenen Login und der Abfrage der Serverzeit

Natürlich ließe sich hier noch einiges tun: Das Passwort sollte sinnvollerweise nur über Sternchen dargestellt werden, die Eingabefelder könnten über eine (gemeinsame Methode) `TextChanged` dafür sorgen, dass `bLogin` nur dann wählbar ist, wenn beide auch wirklich Text enthalten und so weiter.

So einfach dieses Beispiel aber auch ist: Wenn Sie die letzten knapp 10 Seiten praktisch am Computer nachvollzogen haben, sind Sie nun stolzer Besitzer eines Server/Client-Systems, das nicht nur theoretisch selbst dann funktioniert, wenn zwischen den beteiligten Computern ein ganzer Kontinent liegt. Den dafür notwendigen Programmieraufwand kennen Sie inzwischen – und genau jetzt sollte der Moment gekommen sein, wo sich DCOM- und CORBA-Programmierer einen Hochachtungsschluck auf die neuen Herrlichkeiten genehmigen (oder verzweifelt überlegen, wie lange man sie noch zwingen wird, mit ihren alten Techniken zu arbeiten).

3.3.5 Die ausführbare Datei

... hört in diesem Fall auf den Namen *webform1.aspx* und befindet sich – solange Sie es bei den Standardvorgaben belassen haben – im Ordner *wwwroot\WebApplication1*. Wenn Ihr Entwicklungssystem über ein LAN mit anderen Systemen verbunden ist, sollten Sie die Seite einmal von einem der anderen Systeme aus über den Internet Explorer abrufen. Als erster Teil des URL lässt sich gegebenenfalls auch die IP-Adresse des Entwicklungssystems verwenden – beispielsweise:

```
http://192.168.0.3/webapplication1/webform1.aspx
```

3.4 Fehlersuche

Was sich heutzutage integrierte Entwicklungsumgebung nennen darf, muss nicht nur Editor, Compiler, Linker und Online-Dokumentation (vulgo: Hilfestellung) nahtlos integrieren, sondern auch einen Debugger, der idealerweise sowohl auf Quelltext- als auch auf Maschinenebene funktioniert. VS.NET genügt diesen Forderungen in vorbildlicher Weise, wobei die Ebene des Maschinencodes hier *nicht* den Symbolen und Befehlen der Intermediate Language entspricht, wie es eigentlich zu erwarten wäre, sondern echten, vom JIT-Compiler erzeugten Prozessorbefehlen.

3.4.1 Haltepunkte

Das Setzen von Haltepunkten ist auch in einem laufenden Programm möglich, sofern dieses Programm aus VS.NET heraus und mit dem Befehl DEBUGGEN/STARTEN – ersatzweise F5 – gestartet worden ist.

Um einen Haltepunkt an beliebiger Stelle eines Quelltexts einzufügen, setzen Sie entweder den Cursor auf die jeweilige Zeile und verwenden die Taste F9 oder den Befehl HALTEPUNKT EINFÜGEN aus dem per *Rechtsklick* aufzurufenden Kontextmenü. Alternativ tut es auch ein *Linksklick* in den linken Fensterrahmen auf der Höhe der jeweiligen Zeile (zu klicken ist in diesem Fall exakt dort, wo in Abbildung 3.20 der Kreis erscheint). Mit einem Linksklick in diesen Kreis bzw. der Taste F9 wird man den Haltepunkt übrigens auch wieder los.



Linksklicks vor einer Quelltextzeile sind an sich der schnellste Weg zum Setzen (und Entfernen) von Haltepunkten. Unerfreulicherweise bietet die aktuelle Version von VS.NET nur über Umwege die Möglichkeit zur Festlegung von Eigenschaften eines auf diese Weise gesetzten Haltepunkts. Wenn Sie die Unterbrechung des Programmlaufs von Bedingungen abhängig machen wollen, müssen Sie den Befehl NEUER HALTEPUNKT... des Kontextmenüs oder Strg+B benutzen.

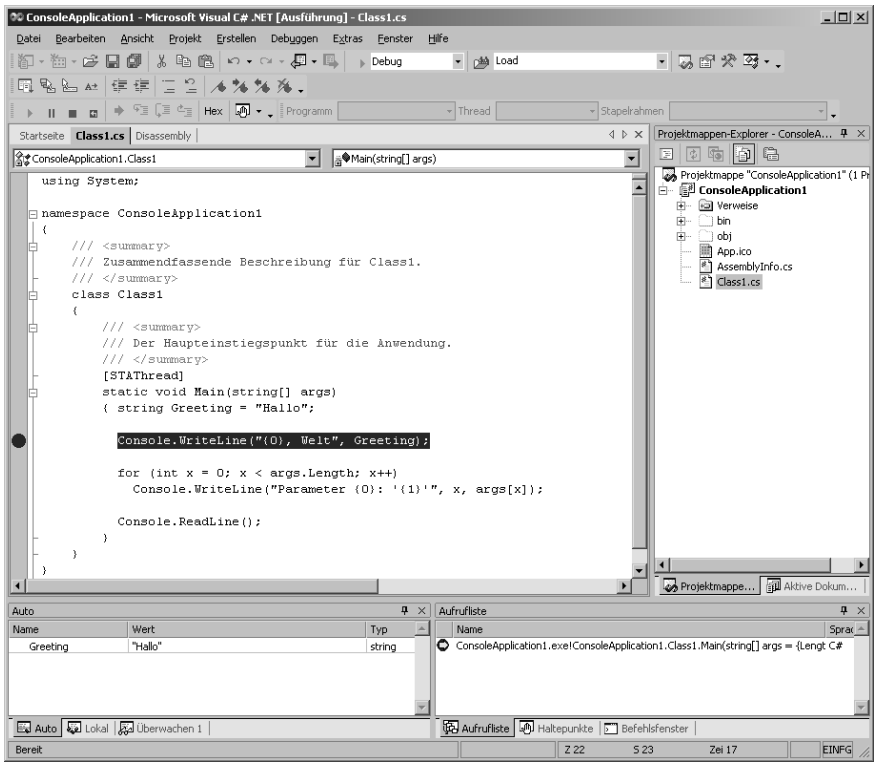


Abbildung 3.20: VS.NET beim Erreichen eines zuvor gesetzten Haltepunkts

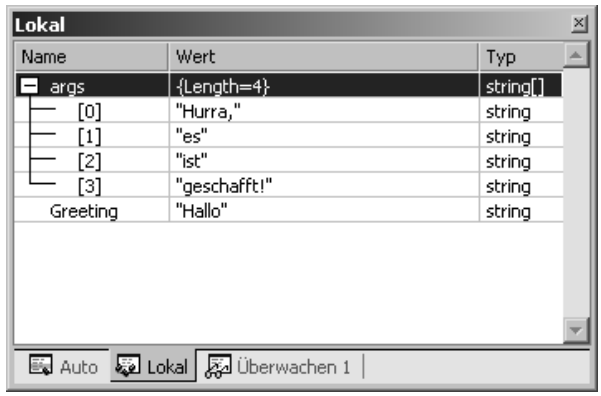
Nachdem VS.NET das Programm an einem Haltepunkt unterbrochen hat, bekommen Sie in einem Fenster namens AUFRUFLISTE die Hierarchie der Methodenaufrufe zu sehen, die zu diesem Haltepunkt geführt haben. (Im gegebenen Beispiel enthält diese Hierarchie gerade einmal ein Element, nämlich die Methode `Main` – vgl. Abbildung 3.20.)

Das – offensichtlich erst nach einer Unterbrechung zu erreichende – Fenster HALTEPUNKTE listet die einzelnen Haltepunkte auf und erlaubt per Kontextmenü die Veränderung ihrer Eigenschaften, beispielsweise durch eine (in C# formulierte) Abbruchbedingung oder einen Wiederholungszähler.

Untersuchen von Variablen

Im Fenster AUTO zeigt VS.NET automatisch (daher der Name) die Werte der Variablen an, die in der aktuellen Anweisung erscheinen. Wenn Sie sämtliche lokalen Variablen der aktuellen Methode im Blick haben wollen, klicken Sie auf den Reiter LOKAL. Strukturierte Variablen werden in beiden Fenstern mit einem vorangestellten Pluszeichen angezeigt; ein Klick darauf stellt die einzelnen Elemente (oder Felder, je nach Variablentyp) dar – vgl. Abbildung 3.21.

Abbildung 3.21: Lokale Variablen nach Erreichen eines Haltepunktes



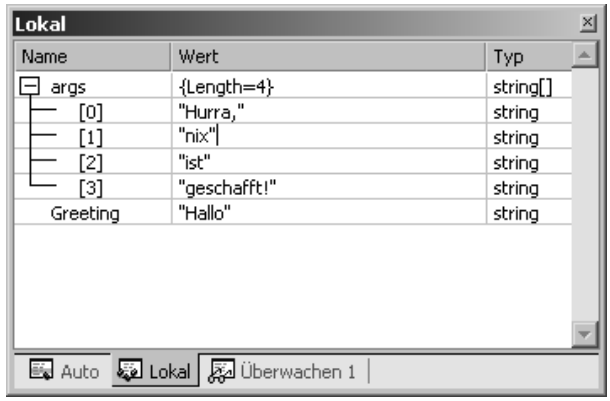


Wenn Sie eine Variable permanent im Blick behalten wollen, verwenden Sie das Fenster ÜBERWACHEN. Dessen Bedienung lässt zwar erst einmal stutzen, weil man im Kontextmenü (wie auch in den beiden anderen Fenstern) einen Befehl NEU vergeblich sucht. Des Rätsels Lösung: Klicken Sie einfach ins linke Feld der ersten Zeile, und tragen Sie den gewünschten Variablennamen ein.

Ändern von Variablen

Das willkürliche Ändern eines Variablenwerts zu Testzwecken funktioniert in allen drei Fenstern (AUTO, LOKAL und ÜBERWACHEN) auf dieselbe, nach kurzer Eingewöhnungszeit doch recht intuitive Weise: Klicken Sie in der Spalte *Wert* an die gewünschte Stelle, und tragen Sie den neuen Wert ein. Bei Strings ist in diesem Fall darauf zu achten, dass die Anführungszeichen erhalten bleiben – vgl. Abbildung 3.22.

Abbildung 3.22: Manuelle Änderung des Wertes einer Variablen



Um auf die Schnelle einen einzelnen, im Fenster AUTO nicht angezeigten Variablenwert zu untersuchen oder zu verändern, kann man schließlich die SCHNELLÜBERWACHUNG benutzen, die über das Menü DEBUGGEN bzw. den Tastenbefehl Alt+Strg+Q erreichbar ist.

Fortsetzen des Programms

Zur Fortsetzung eines per Haltepunkt unterbrochenen Programms bietet VS.NET mehrere Funktionen, die ihrerseits wiederum auf zwei oder gar drei Wegen erreichbar sind:

- *Einzelschritt* – führt die jeweils nächste Anweisung aus. Wenn es dabei um den Aufruf einer Methode geht, deren Quelltext verfügbar ist, wird lediglich die erste Anweisung dieser Methode ausgeführt. Erreichbar über DEBUGGEN/EINZELSCHRITT, das Symbol »Einzelschritt« in der Menüleiste und den Tastenbefehl F11.
- *Prozedurschritt* – unterscheidet sich von Einzelschritt nur in einem Punkt: Methodenaufrufe, die in der jeweils nächsten Anweisung enthalten sind, werden en bloc ausgeführt. Erreichbar über DEBUGGEN/PROZEDURSCHRITT, das Symbol »Prozedurschritt« in der Menüleiste und den Tastenbefehl F10.
- *Ausführen bis Rücksprung* – führt die aktuelle Methode bis zu einem beliebigen `return` en bloc aus und hält das Programm dann an. Erreichbar über DEBUGGEN/AUSFÜHREN BIS RÜCKSPRUNG, das Symbol »Ausführen bis Rücksprung« in der Menüleiste und den Tastenbefehl ⇧+F11. (Delphi-Programmierer, die gerne ein halbes Dutzend *Exit*-Anweisungen verstreuen, vermissen ein Äquivalent dieses Befehls immer noch. Nach unserer Meinung ja zu Recht, weil Methoden mit einer Vielzahl von »Ausgängen« unübersichtlich sind – aber hier kann C# seine Verwandtschaft zu C/C++ eben doch nicht verleugnen.)
- *Weiter* – setzt die Ausführung bis zum nächsten Haltepunkt oder dem regulären Ende des Programms fort. Erreichbar über DEBUGGEN/WEITER, das Starttastensymbol in der Menüleiste und den Tastenbefehl F5 (der während einer Debuggersitzung also nicht die Bedeutung »Programm abbrechen und neu starten« hat, wie man vermuten könnte).
- *Debuggen beenden* – bricht die Ausführung ab und wirft das Programm aus dem Hauptspeicher (was mit der Freispeicherverwaltung von .NET auch spurlos klappt – bei unverwaltetem Code, der Systemressourcen belegt, muss man sich bei diesem Befehl dagegen auf die Verwaltung von Windows verlassen). Erreichbar über DEBUGGEN/DEBUGGEN BEENDEN, das Stoptastensymbol in der Menüleiste und den Tastenbefehl ⇧+F5.



Mit dem Pausetastensymbol in der Menüleiste bzw. Strg + Pause *lässt sich der Lauf eines (unter Kontrolle des Debuggers gestarteten) Programms sozusagen ungezielt unterbrechen – beispielsweise, wenn das Programm in eine Endlosschleife geraten ist.*

Dank des verwalteten Codes haben Sie bei C# eine hohe Chance, nach einer derartigen Unterbrechung tatsächlich die fehlerhafte Stelle im Quelltext angezeigt zu bekommen. (Bei Delphi und Microsoft C/C++ landet man mit solchen Stopversuchen leider meist in irgendeiner Routine des Systemkerns.)

3.5 Konfigurationen

Wie wohl auch nicht anders zu erwarten, verfügt VS.NET über eine Vielzahl an Einstellmöglichkeiten, bleibt aber bei allen Sprachen – der neu entwickelten, schlanken Laufzeitumgebung .NET sei Dank – zumindest in der aktuellen Version noch weit bescheidener als Microsoft C/C++ (von dem ja nicht nur Spötter behaupten, es verfüge garantiert auch über einen Kommandozeilenschalter, mit dem man den Prozessor rückwärts laufen lassen kann). Die folgenden Abschnitte verstehen sich dennoch weiß Gott nicht als Referenz, sondern beschränken sich bewusst auf die Dinge, die man bei der Einarbeitung in VS.NET über kurz oder lang sucht.

3.5.1 Debug und Release

Die Voreinstellung für neue Projekte ist sinnvollerweise der Debug-Modus: Der Compiler erzeugt zusätzliche Informationen für den Debugger, die ausführbare Datei wird im Unterordner *bin\Debug* abgelegt. Im Release-Modus fehlen die zusätzlichen Informationen, die ausführbare Datei landet im Unterordner *bin\Release* des Projekts.

Das Umschalten zwischen den beiden Modi geschieht *ausschließlich* über den Menübefehl ERSTELLEN/KONFIGURATIONS-MANAGER (siehe Abbildung 3.23), mit dem sich bei Bedarf auch weitere, beliebig benannte Konfigurationen wie beispielsweise Spezialversionen eines Programms für den firmeninternen Bedarf anlegen lassen.

Für die Festlegung, was im einen und was im anderen Modus geschieht, ist nicht der Konfigurations-Manager zuständig – das läuft über die Eigenschaften des Projekts, die über das Kontextmenü des Projektnamens im Projektmappen-Explorer erreichbar sind. Dort festgelegte *Allgemeine Eigenschaften* gelten für jede Konfiguration (weshalb das Listenfenster für die Konfiguration auch deaktiviert ist, solange Sie sich in diesen Eigenschaften bewegen). Nach Auswahl der Unterabteilung *Konfigurationseigenschaften* ist das Listenfenster zur Auswahl der Konfiguration aktiviert (siehe Abbildung 3.24).

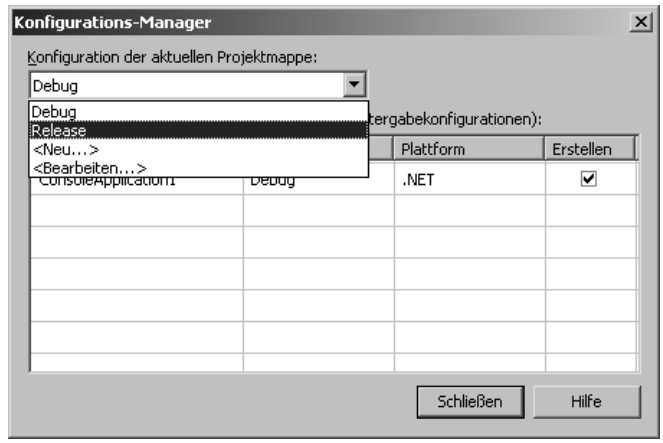


Abbildung 3.23: Umschalten zwischen Debug und Release über den Konfigurations-Manager

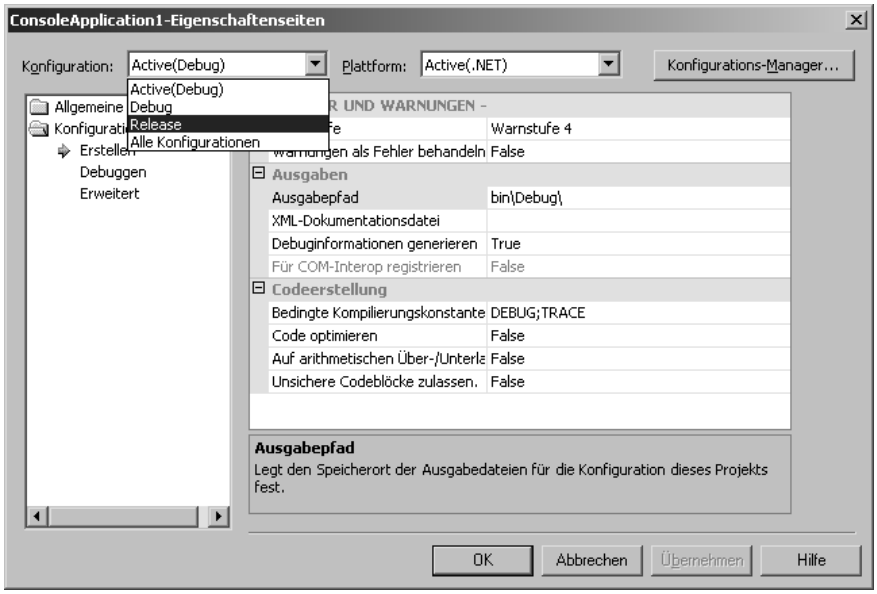


Abbildung 3.24: Festlegen der Einstellungen für die Konfigurationen über die Projekteigenschaften

Bei der ersten Compilierung nach der Umschaltung in den Release-Modus legt VS.NET zwei weitere Unterordner für das Projekt an: *bin\Release* und *obj\Release*. In *bin\Release* wird schließlich die ausführbare EXE-Datei gespeichert.

Wie durch den Zusatz »Active« im Listenfenster auch angedeutet, geht es hier nicht um das Umschalten zwischen Konfiguration, sondern um das Festlegen ihrer Eigenschaften. Für die Auswahl der aktiven Konfiguration ist wie gesagt ausschließlich der Konfigurations-Manager zuständig, der konsequent in den Eigenschaftsseiten auch noch einmal eine eigene Schaltfläche spendiert bekommen hat.



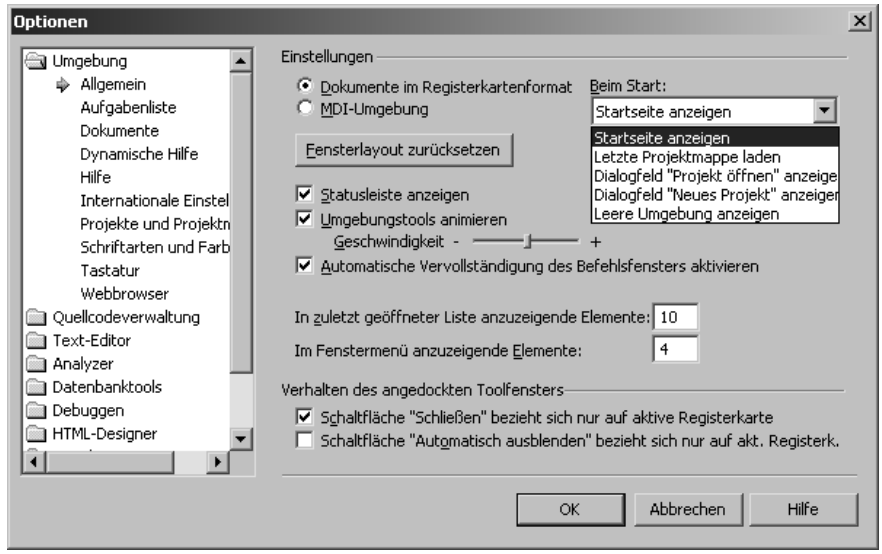
:-)
TIPP

Wenn Sie Eigenschaften festlegen wollen, die für beide Compilierungsmodi gleichermaßen gelten, wählen Sie vorher in den Projekteigenschaften Alle Konfigurationen (vgl. Abbildung 3.24). Ansonsten behandelt VS.NET sämtliche Angaben – wie beispielsweise Kommandozeilenparameter – für die Debug- und Release-Konfiguration als individuell.

3.5.2 Aktionen beim Start von VS.NET

Der weitaus größte Teil der Einstellmöglichkeiten für VS.NET selbst verbirgt sich in einem Dialogfeld, das über den Menübefehl EXTRAS/OPTIONEN zu erreichen ist (siehe Abbildung 3.25).

Abbildung 3.25: Die Startoptionen im Dialogfeld *Extras/Optionen*



Für das Studium dieses Dialogfelds sollte man sich auf jeden Fall einen Moment Zeit nehmen: Über die Abteilung *Text-Editor* lässt sich beispielsweise festlegen, ob, wie heftig, und mit was (nämlich Leerzeichen oder Tabs) die automatische Einrückung zuschlagen soll.

Eine Option zur automatischen Speicherung aller bearbeiteten Dateien vor dem Start eines neu erstellten Programms aus VS.NET heraus werden Sie übrigens vergeblich suchen – einfach deshalb, weil die Entwicklungsumgebung grundsätzlich alles Veränderte in Sicherheit bringt, bevor sie irgendein anderes Programm startet. (In die Kategorie »anderes Programm« fallen hier auch die über das Menü EXTRAS zu erreichenden Hilfsprogramme wie Spy++ oder ILDASM.)

Und schließlich gibt es auch noch einen Wermutstropfen im Zusammenhang mit Dateien: Sicherungskopien von Quelltexten (á la *Form1.cs.bak* oder *Form1.~cs*) haben die Entwickler von Visual Studio leider schon immer für überflüssig gehalten.

Teil 2 Die Sprache C#

Kapitel 4: Warum eine neue Sprache? 81

Kapitel 5: Grundlegende Konzepte 95

Kapitel 6: Anweisungen und Ausführungskontrolle 123

Kapitel 7: Datentypen 163

Kapitel 8: Objekte, Klassen, Vererbung 241

Kapitel 9: Ausnahmebehandlung 339

Kapitel 10: Ereignisbasierte Programmierung 355

Kapitel 11: Einsatz der .NET-Basisklassen 361

Dieser Teil ist der Konzeption der Sprache C# gewidmet. Er zeichnet in seinen verschiedenen Kapiteln ein Bild der Sprache selbst, dem von ihr transportierten Objektbegriff und in Teilen auch von der Landschaft, die sie umgibt.

Obwohl es eines der Ziele des Teils ist, die Sprache und die ihr zugrunde liegenden Ideen möglichst verständlich und umfassend vorzustellen, beugen sich die einzelnen Kapitel weder den Erfordernissen einer Einführung in die Sprache C# noch dem Vollständigkeitsanspruch einer C#-Sprachreferenz. Sie sind vielmehr bemüht, das Neue der Sprache kontrastiv zu anderen Sprachen – insbesondere natürlich C/C++ – herauszuarbeiten. Das heißt, der hinreichend versierte Programmierer findet die Konzepte der Sprache nicht nur erläutert, sondern auch in ihrer Andersartigkeit vor dem Hintergrund dessen, was er schon kennt, beschrieben.

4 Warum eine neue Sprache?

Als Microsoft 1999 das Zauberwort C# wie eine Katze aus dem Sack ließ, war bereits nach kurzer Zeit klar, dass diese Sprache mehr als nur eine Antwort auf Java sein würde und ein ähnlicher Hype wie seinerzeit bei Java zu erwarten war – wenngleich noch niemand genau sagen konnte, wie diese Antwort im Detail aussehen und welche Konsequenzen sie für den Einzelnen haben würde.

Während Java nämlich als Programmiersprache mit internetorientierter Sicherheitsphilosophie und Sandboxkonzept technologisch im Wesentlichen nur für sich selbst steht, präsentiert sich C# als Galionsfigur einer allumfassenden Technologie, deren Zielsetzung darin besteht, die gesamte Produktlinie jenseits der Betriebssysteme von Microsoft auf neue Beine zu stellen.

Mit .NET hat Microsoft zweifelsohne das schwerste Geschütz in seiner bewegten Firmengeschichte aufgefahren. Tragweite und Sogwirkung der Gesamtkonzeption sind so enorm, dass .NET nicht nur die Windows-basierte Informationstechnologie in kürzester Zeit völlig revolutionieren wird, es schafft dem Konzern auch die Ausgangsposition für den erneuten Ansturm auf das Internet und den allfälligen eigenen Strukturwandel.

C# wurde »als Kind der neuen Zeit« in eine blühende Landschaft hineingeboren und darf keinesfalls als isoliertes Sprachprodukt begriffen werden. Dieses Kapitel erörtert verschiedene Hintergründe, die für die Entstehung der Sprache C# wegweisend waren.

4.1 Die J++-Affäre

Als Sun Systems in den 90er Jahren aus der Not eine Tugend machte und der Internet-Gemeinde mit Java eine neue, plattformübergreifende Programmiersprache für das Anwendungsdesign im Intra- und Internet präsentierte, hatte Microsoft dem eigentlich nichts entgegenzusetzen. Man sprang als Lizenznehmer von Sun auf den bereits fahrenden Zug auf und versuchte, den Lauf der Dinge zumindest noch mit einem eigenen Sprachprodukt namens J++ einigermaßen mitzugestalten – sozusagen aus der ersten Reihe heraus und über das Eigengewicht, nachdem der Platz auf der Kanzel schon vergeben war. Allerdings stand das Produkt von Anfang an unter keinem guten Stern, und

rückblickend wird man J++ eher als Bauernopfer bezeichnen, denn als ernstgemeinten Versuch, der Welt ein »Java für Windows« zu bescheren.

Zunächst einmal bot J++ Microsoft zweierlei: Erstens die Möglichkeit, seine eigene Gemeinde zu hätscheln und zweitens eine Spielwiese zum Sammeln von Erfahrungen mit einer Technologie, die an sich nicht das ureigenste Geschäft eines Betriebssystemgiganten darstellt – nämlich Plattformunabhängigkeit.

Lerne mit den Waffen des Gegners zu kämpfen und hintertreibe sie, um ihnen ihre Durchschlagskraft zu nehmen.

Microsoft verstand es bestens, diese beiden Schienen sowohl technisch als auch politisch zu nutzen: Mit jeder Revision weichte Microsoft den von Sun vorgegebenen Java-Standard ein Stückchen weiter auf, um J++ besser an bestehende Microsoft-Technologien, allem voran an die eigene OLE-Technolgie ActiveX, anzupassen. Das hintertrieb die beiden Grundgedanken von Java: das auf der Sandbox beruhende Sicherheitskonzept und die Plattformunabhängigkeit. J++ wurde zunächst hinter vorgehaltener Hand, dann aber zunehmend unverhohlener als Störmanöver von Microsoft bezeichnet, als Ballon, der die Java-Gemeinde aufspalten und den Grundgedanken von Java verwässern sollte. Sun reagierte darauf, indem es an eine Verlängerung des Lizenzvertrags mit Microsoft die Bedingung knüpfte, dass ein Rückbau von J++ an den Sun-Standard zu erfolgen hätte. Anstatt darauf einzugehen, gab Microsoft sein sichtlich schmaler werdendes Trittbrett auf und konzentrierte sich darauf, sein eigenes Süppchen zu kochen.

Damit hatte J++ seine Schuldigkeit getan. Der Java-Sandkasten war zu klein geworden. Der Konzern wusste inzwischen, worum es ging und wie er den hinter Java steckenden Kerngedanken für sich und seine Bedürfnisse vereinnahmen konnte.

4.2 Höchstrichterlich zur »Schichtung« verurteilt

Hinzu kam, dass der Konzern aufgrund seiner Wettbewerbspraktiken kartellamtlich unter Beschuss geraten war und nach zähen gerichtlichen Auseinandersetzungen zu einer Aufteilung verurteilt wurde. Der Urteilsspruch des Richters sah eine organisatorische Auftrennung des Konzerns in eine Firma für Betriebssystemsoftware und eine andere für Anwendungssoftware vor. Auch wenn Microsoft die faktische Zerschlagung durch juristische Kniffe und einen opportunen Farbwechsel im politischen Klima der USA bisher noch abwenden konnte, hat das Management auf das sich bereits lange im Vorfeld abzeichnende Urteil doch umgehend reagiert und sich auf seine Auswirkungen eingestellt.

Das .NET-Framework trennt die Gefilde der Anwendungs-Software tatsächlich sauber von der Betriebssystem-Software ab – mit dem äußerst prak-

tischen »Nebeneffekt«, dass das Framework als Sandbox für Anwendungs-Software fungieren kann und gleichzeitig (insbesondere auch aus juristischer Sicht) Plattformunabhängigkeit gegeben ist. Unter diesem Licht betrachtet, spiegelt die .NET-Konzeption die Bereitschaft des Konzerns wider, die durch das Urteil auferlegten strukturellen Veränderungen umzusetzen. Sollte es eines Tages doch noch zur Spaltung kommen, kann das .NET-Framework im Firmenteil für Anwendungs-Software als plattformunabhängige Betriebssystemabstraktion fungieren, weshalb der Unterbau für die nahtlose Integration von Anwendungs-Software auch dann problemlos verfügbar bleibt.

Auf jeden Fall kann nun niemand mehr den Vorwurf erheben, Microsoft würde seine Dominanz im Bereich Betriebssysteme wettbewerbswidrig bei Anwendungen in die Waagschale werfen, um die Verbreitung bestimmter Technologien zu steuern. Es steht jedem frei, sich auf .NET einzulassen oder nicht, die Betriebssystemprodukte von Microsoft zwingen ihn zu nichts.

.NET extrapoliert die hinter Java steckende Sandbox-Idee auf nahezu den gesamten Anwendungs-Software-Bereich von Microsoft. Inwieweit .NET allerdings bald zum »betriebssystemunabhängigen« Betriebssystem avanciert, ist eine Frage der Zeit – und der Akzeptanz durch die Entwicklergemeinde. Das kartellrechtliche Problem ist wohl noch lange nicht ausgestanden.

4.3 Entwicklung aus Sicht von Microsoft

Als Microsoft sich dafür entschied, die Anwendungsentwicklung von der Ebene der Win32-API abzulösen und auf ein neues Fundament zu stellen, das erstens den gesamten überflüssig gewordenen historischen Ballast über Bord warf und zweitens eine natürliche Unterstützung für Komponentenarchitekturen und verteilte Anwendungen bot, stellte sich natürlich auch die Frage, welche Programmiersprache fortan im Zentrum der Anwendungsentwicklung stehen sollte.

Bis Windows 3.x war es im Wesentlichen die Sprache C, die als Vehikel für die ernsthafte Anwendungsentwicklung benutzt wurde. Mit Aufkommen der OLE-Technologie rückte C++ als hinreichend maschinennahe Sprache mit objektorientierter Philosophie zunehmend in den Vordergrund. Ursprünglich eher als »me too«-Produkt schnell auf den Markt geworfen, um Konkurrenten wie Zortech und Borland im Hype um C++ Paroli zu bieten, entwickelte sich die MFC-Klassenbibliothek (Microsoft Foundation Classes) spätestens beim Übergang zur 32-Bit-Technologie als die am weitesten verbreitete Basis für die Anwendungsentwicklung in der Windows-Welt. Die Win32-API ist allerdings nach wie vor C reinsten Wassers – schon allein deshalb, weil sich sonst überhaupt keine andere Programmiersprache als C++ für die Anwendungsentwicklung verwenden ließe.

4.3.1 Verteilte Anwendungen

Der von C++ transportierte Objektbegriff ist als »Bottom-up«-Konzept in der Sprache verankert und somit primär auf die Sprache selbst und die Ebene der einzelnen Anwendung beschränkt. Für eine Komponentenarchitektur, mit der sich verteilte Anwendungen realisieren ließen, braucht es mehr: ein »Top-Down«-Konzept.

Während andere Betriebssysteme wie beispielsweise NeXT oder BeOS, aber auch – man höre und staune – das 20 Jahre alte Unix von Anfang an komponentenbasiert angelegt waren, übernahm die Microsoft-Plattform die entsprechenden Konzepte nur schrittweise mit der gewohnten Behäbigkeit. Die ersten zaghaften Ansätze für eine Komponentenarchitektur entstanden aus dem Bedürfnis heraus, die einzelnen Office-Anwendungen in eine Office-Suite zu integrieren. Die Office-Anwendungen sollten ihre Dokumente wechselseitig einbetten und Datenaustausch betreiben können. Der Standard dafür hieß OLE 1.0, kam unter Windows 3.1 auf und hatte mit einer Komponentenarchitektur an sich noch wenig zu tun.

So eng die Grenzen von OLE 1.0 auch gesteckt waren, die Technologie an sich wurde ein voller Erfolg und erwies sich als wegweisend für den so genannten *dokumentenzentrierten Ansatz*, bei dem nicht mehr die einzelne Anwendung, sondern das Dokument im Mittelpunkt des Benutzerinteresses stand. Das Regelwerk für diese Technologie war der OLE 2.0-Standard, der neben der neuen 32-Bit-Technologie eine der wichtigsten Säulen für den Erfolg von Windows 95 bildete.

COM – sprachunabhängige Objekte als Top-Down-Konzept

OLE 2.0 war mehr als nur der Nachfolger einer Einzeltechnologie. Der Standard legte den Grundstein für ein ganzes Bündel (von Microsoft später unter dem Namen ActiveX zusammengefasster) Technologien, die ihren gemeinsamen Nenner darin hatten, dass sie sich um ein und denselben Objektbegriff gruppierten. Den Kern von OLE 2.0 bildet das Component Object Model, kurz COM. Als sprachübergreifendes »Top-Down«-Objektkonzept benennt es eine Reihe von Mechanismen und Standards, die das Zusammenspiel mehrerer voneinander unabhängig implementierter Codekomponenten regeln – der so genannten COM-Objekte. COM umfasst:

- Einen Verwaltungsmechanismus, der das systemweite Auffinden von COM-Objekten ermöglicht (Voraussetzung dafür ist die zentrale Windows-Registrierung)
- Einen Schnittstellenstandard, der regelt, in welcher Form ein COM-Objekt seine Dienste zur Verfügung stellt.
- Einen Lade- und Bindemechanismus, der es Anwendungen (Komponenten) ermöglicht, COM-Objekte zur Laufzeit aufzurufen (auszuführen) und deren Dienste zu nutzen.

- Einen Mechanismus für den Austausch binärer Daten (Objekte) mit COM-Objekten, der auch prozess- oder auch systemübergreifend (DCOM) funktioniert
- Einen *Referenzzählungsmechanismus*, der darüber Aufschluss gibt, wann ein Objekt nicht mehr benötigt wird und abgebaut werden kann.
- Einen Mechanismus für die Fehlerbehandlung im Zusammenhang mit COM-Objekten.

Die Integration von COM bedeutete für die Plattform Windows einen gewaltigen Schritt nach vorn. Endlich war eine Infrastruktur vorhanden, wie sie für die Realisierung verteilter Anwendungen mit Komponentenarchitektur erforderlich war.

Kein Wunder also, dass COM bereits nach recht kurzer Zeit die gesamte Anwendungslandschaft für die Plattform Windows durchsetzte.

.NET – sprachunabhängige Objekte als Bottom-Up-Konzept

So elegant »Top-Down«-Ansätze aus der konzeptionellen Perspektive sind, so unbeholfen werden sie, wenn es darum geht, sie tatsächlich auszudeklinieren. Als Standard, der nicht aus dem konkreten Umfeld einer Sprache heraus erwachsen ist, benötigt COM einen gewaltigen Apparat als Unterbau – grundlegende Datentypen und Typbibliotheken für den typsicheren Datenaustausch, eine Typverwaltung auf Systemebene und eine Art Laufzeitsystem mit Aufrufmechanismus, Bindemechanismus und Speicherverwaltung.

Auf der anderen Seite hat natürlich auch jede der Einzelsprachen, mit denen man COM-Objekte implementieren kann, wiederum ihren eigenen Apparat, weshalb ein großer Teil der für die Initialisierung und den Unterhalt von COM-Objekten erforderlichen Laufzeit schlicht für die wechselseitige Anpassung der heterogenen Strukturen und Mechanismen anfällt: die der Einzelsprache auf der einen und die von COM auf der anderen Seite. Ärgerlicherweise ist das bei Huckepackarchitekturen – oder vornehmer ausgedrückt: geschichteten Architekturen – dieses Umfangs prinzipbedingt.

Für komplexe Objekte, wie (wechselseitig) eingebettete Dokumente, die ja wie gesagt den Ausgangspunkt dieser Technologie darstellten, schlug dieser »Mehraufwand für doppelte Haushaltsführung« nicht allzu sehr zu Buche. Der Wunsch, den Objektgedanken von COM auch auf einfachere, besser verwobene und weniger selbstständige Gebilde zu übertragen, scheiterte dagegen bereits an einfachen Kosten/Nutzen-Abwägungen.

Kurzum, an filigranere Objektarchitekturen war mit diesem Ansatz nicht zu denken. Es musste etwas her, dem der Objektgedanke von Grund auf in die Wiege gelegt war. Metaphorisch gesagt, ein Ansatz, der die Objekte bereits

»vorsprachlich« und nicht erst »nachsprachlich« verheiratete und tatsächlich auf eine Art Umkehrung der Ursächlichkeit hinauslief: die neue Implementierung aller »zum Club gehörigen« Sprachen auf Basis eines gemeinsamen, selbstverständlich objektorientierten Typsystems und der gesamten damit zusammenhängenden Infrastruktur. Das – und nichts anderes – ist der Grundgedanke des Typsystems von .NET.

Die Idee ist ebenso genial wie verrückt, erfordert sie doch eine Revolution auf breitester Front, die alle wichtigen Entwicklersprachen mit einem Streich unter ein und denselben Hut bringen soll, aller Standards, Spezifikationen und zu erwartenden Widerstände zum Trotz. Es steht außer Frage, dass nur ein Gigant wie Microsoft es sich überhaupt leisten konnte, ein derart ehrgeiziges Projekt in Angriff zu nehmen – und generalstabsmäßig bis ins letzte Detail durchzuziehen.

Hut ab, das Ergebnis kann sich sehen lassen.

4.3.2 Objektmodell ohne Sprache?

Bei soviel Ruf nach Sprachunabhängigkeit stellt sich umgekehrt die Frage, inwieweit es nicht doch sinnvoll ist, ein Objektmodell zu 100 Prozent auf eine bestimmte Programmiersprache abzustimmen.

Von der Theorie her lautet die Antwort natürlich »Ja«. Wer dabei jedoch auf C++ schielt, muss herbe Enttäuschungen hinnehmen. Die Objektmodelle *gegebener* Sprachen (Java einmal beiseite genommen) eignen sich allesamt nicht als Grundlage für system- und sprachübergreifende Komponentenarchitekturen. Ergo: Will der Prophet nicht zum Berge, muss der Berg wohl zum Propheten. Microsoft hat den Spieß einfach umgekehrt und die Sprachspezifikation an das Objektmodell angepasst – für COM etwas weniger rigoros als für .NET.

Als Windows-gerechte Weiterentwicklung der nur schwach standardisierten Sprache BASIC erhielt Visual Basic ab der Version 4.0 einen objektorientierten Sprachkern mit der Möglichkeit zur OLE-Automatisierung – ein skriptorientierter Auswuchs der an sich allgemeineren COM-Spezifikation. In etwas abgewandelter Form als VBA (Visual Basic for Applications) avancierte der Sprachkern dann bald zur gemeinsamen Skriptsprache für die Schlachtschiffe Word, Excel und Access der Microsoft Office-Suite. Damit wurde VBA zur Sprache, der COM als Objektmodell gewissermaßen »auf den Leib geschrieben« war. VBA selbst ist allerdings weniger eine sprachspezifische COM-Implementierung, sondern vielmehr eine Art Laufzeitsystem, das eine ganze Palette an Anwendungssoftware verbrüdert, indem es ihnen die Automatisierung (hierzu gleich noch mehr) ermöglicht. Man denke hier insbesondere an die Makrosprachen der Microsoft Office-Anwendungen Word, Excel und Access sowie an den Windows Scripting Host.

Damit stellt VBA nichts anderes als die Vorstufe zu .NET dar, wenngleich der Ansatz an sich »demokratischer« ist und auch noch Platz für Alternativen lässt. Im Vergleich dazu ist .NET ein Konzept nach der Maxime: »Friss oder stirb«.

Die Sprache der Sprachen für .NET ist C#. Die Antwort könnte aber auch VB.NET oder Delphi.NET lauten, denn Microsoft hat .NET als offenen Standard konzipiert – als Umgebung für Sprachcompiler beliebiger Art (die sich freilich den Regeln des CTS beugen müssen).

Als speziell auf .NET zugeschnittene Sprache ist C# allerdings frei von Altlasten und Kunstgriffen, die das Bild trüben könnten und bietet somit auch einen ungefilterten Blick auf das unter .NET verwendete Objektmodell.

4.3.3 Typsysteme unter Windows

Bevor es im nächsten Kapitel um die konkreten Datentypen der Sprache C# geht, sollte noch kurz beleuchtet werden, wie sich der Begriff des Typsystems unter Windows im Lauf der Zeit entwickelt hat.

Sprachspezifische Bibliotheken

Die ursprüngliche Praxis in der Windows-Programmierung sah so aus, dass Typsysteme oder auch nur zusätzliche Datentypen in Bibliotheken zusammengefasst waren, die jeweils für sich kompiliert und zur Linkzeit als statische (*.lib*) oder zur Laufzeit als dynamische Bibliothek (*.dll*) an den Programmcode gebunden wurden. Um etwa in C/C++ einen Bibliothekstyp zu verwenden, muss man die zur Bibliothek gehörige Headerdatei (*.h* bzw. *.hpp*) mit den Typinformationen einbinden und die *.lib*-Datei zum Linkzeit bzw. die *.dll*-Datei zur Laufzeit bereitstellen. Nach diesem Schema funktioniert beispielsweise auch die MFC (Microsoft Foundation Classes Library), eine recht umfangreiche Klassenbibliothek für C++, mit dem Microsoft die Windows-Programmierung auf objektorientierte Beine stellen wollte.

Bei sprachspezifischen Bibliotheken ergibt sich eine Gruppierung der Typen also rein aus der physischen Gruppierung in Codedateien sowie über Ableitungszusammenhänge.

Trotz der Verlagerung auf die objektorientierte Programmierung mit C++ fristen Objekte ihre Existenz zunächst einmal auf der Ebene (und im Allgemeinen auch im Prozessraum) der Einzelanwendung. Der anwendungsübergreifende Datenaustausch geschieht auf Grundlage von Streams, wobei die entsprechenden Mechanismen von der Windows-API als Dienste bereitgestellt werden.

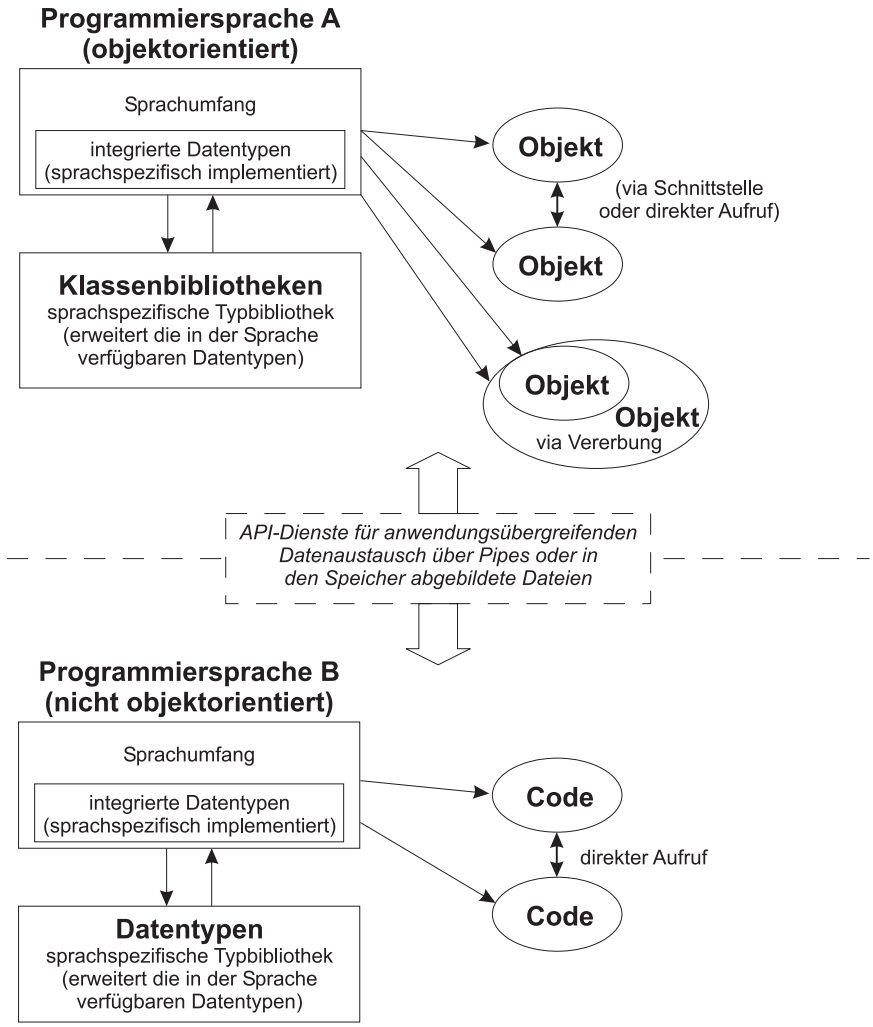


Abbildung 4.1: Datentypen als sprachspezifische Bibliotheken bspw. MFC

COM-Objekte

Mit Einführung des COM als übergeordnetes Objektmodell für OLE 2.0 kam so etwas wie die Zentralisierung eines zunächst sprach- und später auch systemübergreifenden Typsystems ins Spiel. Grundlage dafür ist ein im Betriebssystem verankertes Subsystem, COM bzw. DCOM, das im Zusammenspiel mit der Windows-Registrierung Mechanismen zum Auffinden sowie zum (gegebenenfalls prozess- oder systemübergreifenden) Aufrufen und Kommunizieren mit COM-Objekten bereitstellt. Die Interaktion zwischen COM-Objekt und Programm erfolgt auf der Basis von Schnittstellen und einer Handvoll speziell für COM standardisierter Datentypen – Werttypen, um in der Terminologie von C# zu bleiben.

Der COM-Standard verlangt, dass ein COM-Objekt an einem festgelegten Einsprungpunkt eine IUnknown-Schnittstelle bereitstellen muss, die es Clients ermöglicht, den Einsprungpunkt für jede weitere Schnittstelle des Objekts anzufordern und den obligatorischen Verweiszähler des Objekts zu aktualisieren. Das Wissen darüber, welche Schnittstellen ein Objekt implementiert und welche Methoden zu einer Schnittstelle gehören, findet ein entsprechend ausgerüsteter Compiler in einer Typbibliothek (.tlb), die speziell für das Objekt bereitgestellt werden muss und über die die Registrierung des Objekts aufzufinden ist. Nachdem all dies bereits zur Übersetzungszeit passieren kann, handelt es sich hier um FRÜHE BINDUNG.

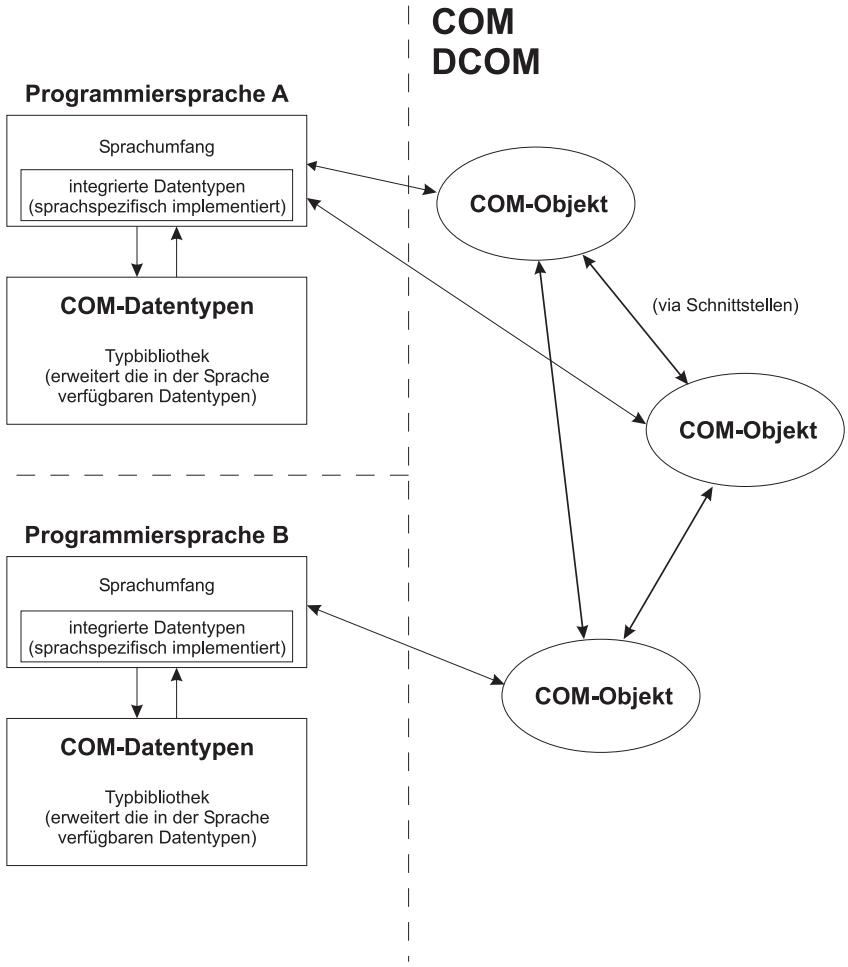


Abbildung 4.2: COM – ein Modell für sprachunabhängige Datentypen

OLE-Automatisierung

Um COM-Objekte auch ohne Typbibliothek verwendbar zu machen und einen standardisierten Datenaustausch zu ermöglichen, führte Microsoft

einen Automatisierungsstandard ein, der auch eine späte Bindung zur Laufzeit ermöglichte. Vom Kern her fordert dieser Standard für ein Automatisierungsobjekt zusätzlich die Implementierung einer `IDispatch`-Schnittstelle, über die ein Client gegebenenfalls auch erst zur Laufzeit Informationen über die Schnittstellen bzw. die darin offen gelegten Eigenschaften und Methoden des Objekts in Erfahrung bringen kann.

COM+

Nachdem sich DCOM – die systemübergreifende Variante von COM – als Umgebung für verteilte Anwendungen in der Praxis in verschiedener Hinsicht (darunter gravierende Gründe wie: Sicherheit und Effizienz) als problematisch erwiesen hatte, stellte Microsoft mit COM+ eine Erweiterung des COM-Standards vor. COM+ bringt unter anderem eine verbesserte Laufzeitumgebung mit einem verfeinerten Sicherheitsmodell und gleich ein ganzes Bündel von Diensten für die Laufzeit- und Speicherplatzoptimierung von COM-Objekten mit sich.

Die Spezifikation von COM+ sieht Mechanismen vor, die dem COM+-Subsystem eine rigidere Verwaltung der Objekte ermöglichen, beispielsweise durch eine Ausführung im selben Prozessraum oder und durch ein Caching fertig initialisierter (statusloser) COM-Objekte.

Typmanagement unter .NET

Die bis dato vorgestellten Typsysteme sind also entweder sprachspezifisch oder aber auf hochkomplexe *COM-Objekte* gemünzt, die erhebliche Anforderungen an die programmtechnische Umsetzung und die Verbreitung über das Netzwerk stellen. Es bleibt eine Lücke: Für sprach- und systemübergreifende »Wald- und Wiesentypen« sind die COM-Mechanismen viel zu laufzeitintensiv und der COM-Standard ein viel zu enges Korsett. Außerdem erweist sich die Beschränkung der Vererbung auf die reine Schnittstellenvererbung vom objektorientierten Standpunkt her als hochgradig unbefriedigend, zumal der Ersatzmechanismus der DELEGATION für das Laufzeitverhalten verteilter Anwendungen alles andere als förderlich ist.

CTS als gemeinsamer Nenner

Bei der Konzeption des Typsystems *für* .NET ist Microsoft mit dem CTS einen von Grund auf anderen Weg gegangen. Das CTS ist kein im Nachhinein aufgesetztes abstraktes Typsystem für bestehende Sprachen, sondern gemeinsamer Nenner und Ausgangspunkt für die erneute Implementierung dieser Sprachen. Bei so viel Gleichmacherei ist es nicht verwunderlich, dass Microsoft im Endeffekt bei jeder Sprache hier und da Änderungen in der Spezifikation vornehmen musste, um die für .NET gesetzten Ziele zu erreichen.

C++ erhielt eine *verwaltete Erweiterung*, die an sich eine Einschränkung (also eine Teilmenge) der Sprache auf die von .NET erwarteten und geduldeten Mechanismen darstellt.

VB wurde so stark umgekrempelt, dass selbst (oder gerade) eingefleischten VB6-Programmierern der Umstieg auf VB.NET alles andere als leicht fällt.

C# wurde als neue Sprache eingeführt, deren Spezifikation speziell auf die Bedürfnisse von .NET zugeschnitten ist. Da es (wie an anderer Stelle schon diskutiert) kein J++.NET oder gar Java.NET geben konnte und C++ ein viel zu mächtiges Gebilde ist, um – seiner Zeigerarithmetik, Mehrfachvererbung und Templates beraubt – als Hofsprache für .NET zur firmieren, musste eine neue Sprache her, die all das enthielt, was .NET ausmachte, aber auch nicht viel mehr. Abbildung 4.2 verdeutlicht, dass C# dennoch ein wenig über den Tellerrand des CTS blickt: Wenn Not am Mann ist, kann diese Sprache nämlich auch mit Zeigern etwas anfangen.

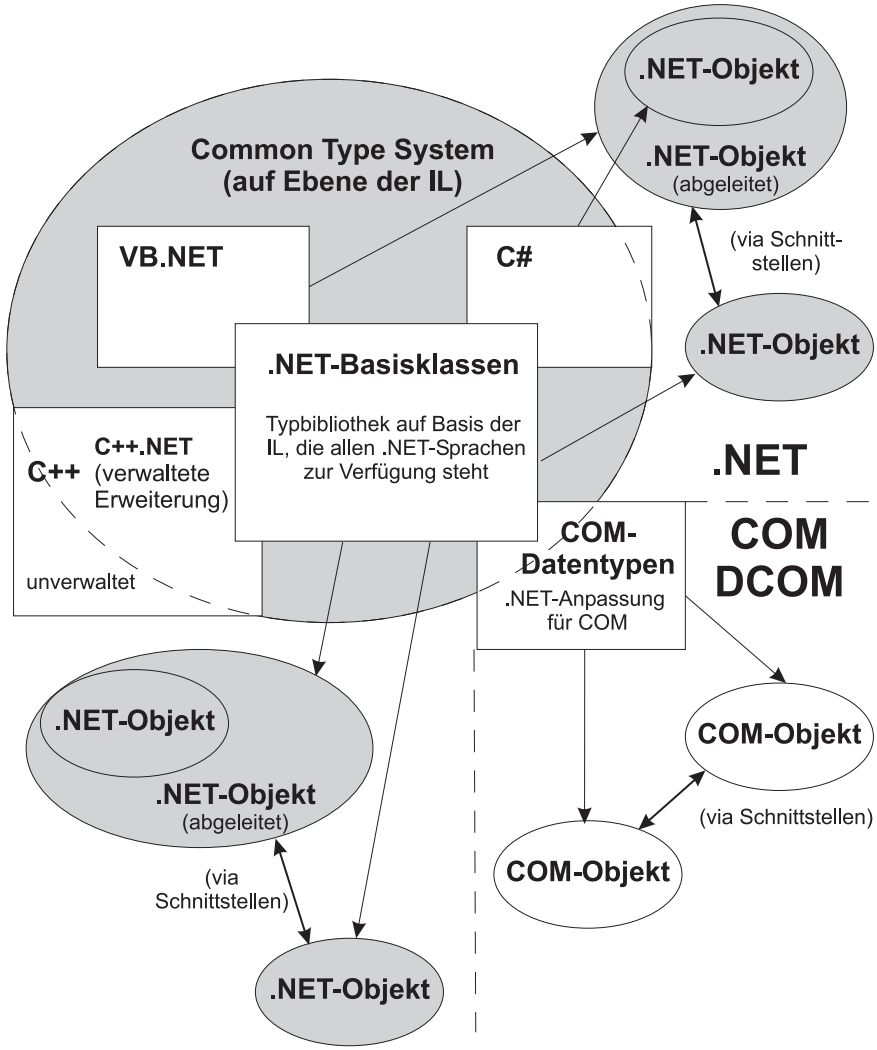
Das Ergebnis kann sich in der Tat sehen lassen: Die Typsysteme der .NET-Sprachen sind damit strukturell äquivalent und auf der Ebene des CLS (Common Language System) sogar identisch.

Die abgeschlossene Welt des .NET Framework bietet im Gegenzug eine Reihe von Annehmlichkeiten in Form von Systemdiensten, die das Erlernen einer neuen Sprache (C#), das Einstellen auf eine veränderte Implementierung (VB.NET) oder die Beschränkung auf eine Untermenge (VC++) mehr als aufwiegen:

- Die Vererbung auf Codebasis und auf Schnittstellenbasis – insbesondere stellt .NET eine, die Win32-API vollständig ersetzende Systemschnittstelle in Form einer Klassenbibliothek bereit.
- Eine automatische Freispeicherverwaltung.
- Eine kontextbezogene Typverwaltung – jede Anwendung erhält ihre eigene Sicht auf die .NET-Klassenhierarchie.
- Eine Versionsverwaltung – jede Anwendung behält ihre (historische) Sicht auf gemeinsam genutzte Komponenten, selbst wenn neue Versionen dieser Komponenten auf dem System installiert werden.
- Eine Durchsetzung von Sicherheitsrichtlinien – das Sicherheitskonzept von .NET arbeitet mit einem Verfahren, das die Benennung gleich mit einer Zertifizierung auf kryptologischer Basis verbindet.

Diese Dienste sollen insbesondere auch sicherstellen, dass es nicht mehr zu ungewollten (bzw. gewollten, feindseligen) Wechselwirkungen zwischen verschiedenen Anwendungen kommen kann.

Abbildung 4.3: Datentypen für COM-Objekte



Ausweg aus der »DLL-Hölle«

Probleme der genannten Art (Stichwort: Korsett) sind für die bisherige Windows-Praxis geradezu symptomatisch. Sie äußern für den Benutzer mit den seltsamsten Kapriolen, die meist erst einige Zeit nach Installation einer neuen Anwendung und einem damit einhergehenden Update von gemeinsam genutzten Bibliotheksmodulen auffallen. Das Hinterlistige an der Problematik ist es wohl, was ihr zu dem unschönen Namen »DLL-Hölle« verhalf. Eine Anwendung kann »besten Wissens und Gewissens« implementiert und getestet sein und trotzdem Seiteneffekte auf andere Anwendungen haben, wenn sie eine neuere Version einer gemeinsam genutzten Bibliothek installiert. Beispielsweise, wenn eine andere Anwendung auf eine bestimmte, womöglich sogar fehlerhaft implementierte Version einer Bibliothek angewiesen ist.

Es versteht sich, dass ein durchschnittlicher Benutzer mit der Frage, ob er die neue Version einer bestimmten DLL installieren oder die alte beibehalten will, eher überfordert ist. Dennoch war dieser Dialog bislang die einzige Möglichkeit, einen gewissen Einfluss auf die Installation neuer Programmen zu nehmen, wenngleich er mehr der Frage an den Verurteilten nach der gewünschten Todesart gleichkommt. Entscheidet man sich für die neue Version einer Bibliothek, läuft man Gefahr, dass eine bereits installierte Anwendung den Dienst verweigert; entscheidet man sich für die alte, ist fraglich, inwieweit die neue Anwendung damit zurecht kommt. Anders gesagt: Der Dialog ist hauptsächlich ein Hinweis dafür, dass nach der Installation potenziell Ausfälle zu erwarten sind.

Vordergründige Ursachen, soweit sie sich überhaupt vom Benutzer ausmachen lassen, sind dann meist bei der Anwendungsentwicklung unbemerkt gebliebene Abweichungen von Schnittstellenspezifikationen und sonstige Programmierfehler, die bei der für die Installation verantwortlichen Anwendung nicht zum Tragen kommen, sondern perfiderweise eben bei Anwendungen, die unfreiwillig vom Update der Bibliotheksmodule betroffen sind.

Hintergründig werden solche Fehler durch die Unvollkommenheit und Undifferenziertheit des für COM benutzten Typverwaltungsmechanismus begünstigt. Die global eindeutige Benennung von COM-Objekten mit so genannten GUIDs (Globally Unique Identifier) verhindert ironischerweise die problemfreie Koexistenz unterschiedlicher Versionen einer ansonsten gleichen Bibliothek.

Aber auch DLLs, die keine COM-Objekte implementieren, machen Schwierigkeiten, welche daher rühren, dass der Lademechanismus von Windows Dateien dieser Art in einer bestimmten Reihenfolge auf dem System sucht. Ist eine DLL bereits geladen, muss die anfordernde Anwendung eben mit der Version vorlieb nehmen, die sich im Speicher befindet. Ansonsten gilt, dass die DLL zuerst im Anwendungsverzeichnis gesucht wird; falls die Suche dort erfolglos bleibt, geht es mit dem aktuellen Verzeichnis und schließlich dem System-Verzeichnis weiter. Die Zugangssoftware für T-Online ist nach wie vor ein prominentes Beispiel für eine Anwendung, deren Integration in die Landschaft anderer Windows-Programme einem Benutzer erhebliche Bauchschmerzen bereiten kann.

Assembly Pools

Unter .NET spielt die physische Organisation der Datentypen in Dateien – genauer: Assemblies – und Verzeichnisse nur noch insofern eine Rolle, als sie in dieser Verpackung die Prüfung der Versions- und Sicherheitsrichtlinien durch .NET passieren. Allen Anwendungen gemeinsam zur Verfügung stehende Assemblies werden in einem zu .NET gehörigen Systemverzeichnis, dem so genannten Global Assembly Cache (ein Ordner namens *Assem-*

bly) vorgehalten, während anwendungsspezifische Assemblies im Verzeichnisbaum der jeweiligen Anwendung zu finden sind.

Eine Anwendung merkt von dieser Trennung überhaupt nichts, sofern sie ihr nicht über den .NET-Reflection-Mechanismus nachspürt. Sie »sieht« immer nur Datentypen in Namensräumen (hierzu gleich noch mehr), deren Implementierung ihr das .NET-Framework unter Beachtung der Versions- und Sicherheitsrichtlinien zuteilt.

Vom Prinzip her war das natürlich immer schon so unter Windows. Neu ist, dass mit .NET nun in beiden Pools verschiedene Versionen einer Assembly (also verschiedene Implementierungen eines Datentyps) nebeneinander existieren können. Das ausgefeilte Versionsmanagement von .NET sorgt dafür, dass eine Anwendung mit der am besten passenden Assembly arbeitet.

Dabei ist es Anwendungen freigestellt, eigene Datentypen in eigenen Namensräumen zu organisieren oder auch in die Binnenstruktur der von `.NET` vorgegebenen Namensräume einzugliedern – ohne Auswirkung auf andere Anwendungen, versteht sich.

5 Grundlegende Konzepte

»Sag mir, woher Du kommst, und ich sage Dir, wer Du bist.« Dass C# seine Abstammung von C/C++ nicht verleugnet, verrät schon der Name. Immerhin, die Sprache ist runder als alles bisher Dagewesene, aller Kanten des Zeichens # zum Trotz. Wer nicht gerade zu den polyglotten Programmierern gehört, dem wird es vielleicht gar nicht so auffallen, dass C# eine ganze Reihe von Errungenschaften der verschiedensten Sprachen – besonders prominent neben C++: Visual Basic, Delphi und Java – adaptiert und zu einem geschlossenen Konzept bündelt. Zugegeben, viele dieser Errungenschaften sind gar nicht so sehr Teil der Sprache selbst, sondern der Philosophie von .NET, und C# in Form der .NET-Basisklassen in die Wiege gelegt.

Dieses Kapitel nähert sich der Sprache gleichermaßen von ganz außen und von ganz innen. Es erörtert zunächst die Konzeption der Datentypen, insbesondere die Unterscheidung zwischen Werttypen und Verweistypen, und stellt dann den prozeduralen Kern der Sprache, das Anweisungskonzept und die verschiedenen Kontrollstrukturen vor.

5.1 Starke Typisierung

Als »Behältnisse« für die Speicherung, die Manipulation und den Transport von Daten sind Datentypen das A und O jeglicher Programmierung. Obwohl es Programmiersprachen gibt, die versuchen, mit möglichst wenigen Datentypen auszukommen oder zumindest den damit zusammenhängenden Ballast vom Programmierer weitgehend fernzuhalten (das Variant-Konzept von VB weist beispielsweise in diese Richtung), hat sich die so genannte starke Typisierung in den letzten Jahren mehr und mehr als Standard für die business-orientierte Programmierung herauskristallisiert.

Es ist daher kein Wunder, wenn die Typisierung unter .NET mit C# als Vorzeigesprache ein bisher noch nicht gekanntes Maß erreicht – und zudem auch noch Blüten treibt, die absolut wegweisend für die künftige Entwicklung verteilter Anwendungen sein dürften.

5.1.1 Namensräume

Unter .NET gehört jeder Datentyp einem Namensraum an. Ein Namensraum ist ein Mittel, Datentypen unter einem gemeinsamen Namen zu organisieren und ansprechbar zu machen. Da dieses Mittel rein sprachlicher Natur ist, spielt es keine Rolle, in welchem Zusammenhang die zusammengefassten Datentypen zueinander stehen. Insbesondere ist also kein Ableitungszusammenhang gefordert, wie man zunächst vermuten könnte.

Das von .NET verwendete Namensraumkonzept erinnert stark an die mit Windows 2000 eingeführten Active Directories. Beide Ansätze etablieren ein Benennungsschema in Form einer zusätzlichen Abstraktionsebene, das losgelöst von allen physischen Gegebenheiten eine rein logische Organisation von Elementen erlaubt – im einen Fall Datentypen, im anderen Dateien.

Als allgemeine Benennungskonvention für .NET-Datentypen bieten Namensräume vier entscheidende Vorteile:

- Ausschaltung von Namenskollisionen – bei Wahl eines Typbezeichners steht keine Kollision mit externen Symbolen mehr zu befürchten, die erst zur Link-Zeit oder – schlimmer – zur Laufzeit auffallen.
- Hierarchische Namensorganisation durch Verschachtelung – mehrere (unterschiedliche) Datentypen mit gleichem Typnamen können koexistieren, wenn sie in unterschiedlichen Zweigen eines Namensraums untergebracht sind.
- Typen in unterschiedlichen Namensräumen dürfen gleiche Namen erhalten – der Bezeichner des Namensraums qualifiziert den Typbezeichner.
- Flexibilität und Erweiterbarkeit – ein Namensraum kann jederzeit umgeordnet werden und darf beliebig viele Bezeichner und untergeordnete Namensräume enthalten.

Syntax

Die Syntax für Namensräume lautet:

```
namespace Raum[.Unterraum ...]
{
  [class-Klassendefinitionen]
  [struct-Klassendefinitionen]
  [interface-Schnittstellendeklarationen]
  [enum-Vereinbarungen für Aufzählungstypen]
  [delegate-Vereinbarungen für Methodenzeiger]
}
```

Variablen- und Konstantendefinitionen sind auf der Ebene des Namensraums offensichtlich nicht erlaubt. Tatsächlich ist der richtige Ort dafür – auch für Konstanten (!) – die Klassendefinition.

Hier ein Beispiel dafür, wie die Definition in einem Namensraum aussehen kann.

```
namespace MyNamespace
{
  class MyRefClass
  {
    // Code für Definition der Klasse
  }
...
}

namespace MyOtherNamespace
{
  ...
}

namespace MyNameSpace
{
  struct MyStruct
  {
    // Code für Definition der Struktur
  }
...
}
```

Augenscheinlich macht es keinen Unterschied, ob der in der `namespace`-Anweisung bezeichnete Namensraum schon besteht oder neu eingeführt wird. Die beiden Klassen `MyRefClass` und `MyStruct` gehören demselben Namensraum an.

Um auf einen Typ in einem bestimmten Namensraum zu referieren, muss man den Typbezeichner mit dem Namensraumbezeichner *qualifizieren*, das heißt, diesen vor den Typbezeichner stellen, abgetrennt durch einen Punkt:

```
MyNamespace.MyStruct myStruct;
```

Zur Abkürzung besteht die Möglichkeit, über eine `using`-Direktive alle Bezeichner eines bestimmten Namensraums kollektiv zu qualifizieren. Im Weiteren kommt man dann für die Bezeichner aus dem angegebenen Namensraum ohne Präfix aus:

```
using MyNamespace;
...
MyStruct myStruct;
MyNamespace.MyStruct myStruct1; // gleichfalls erlaubt
```

Falls ein und derselbe Typbezeichner in zwei Namensbereichen definiert ist und für beide Namensbereiche eine `using`-Direktive verwendet wird, müssen die Typbezeichner explizit qualifiziert werden, da der Compiler sonst keine eindeutige Auswahl treffen kann (und sich sinnvollerweise beschwert).

Bezeichnerwahl

Innerhalb eines C#-Namensraums müssen alle Bezeichner eindeutig sein. Insbesondere dürfen auch Typbezeichner (nicht jedoch Schlüsselwörter der Sprache) als Variablenbezeichner verwendet werden (was in manchen Programmiersprachen nicht gestattet ist), selbst wenn erstere im selben Namensraum definiert sind.

```
using System.Collections;
namespace test3
{
  public class Form1 : System.Windows.Forms.Form
  {
    private int ArrayList;     // ArrayList ist hier ein Bezeichner
    private ArrayList al;      // ArrayList ist eine .NET-Klasse
...
```

Namensraumbezeichner dürfen ohne weiteres als Typ- oder Variablenbezeichner verwendet werden. In den .NET-Basisklassen findet man viele Beispiele dafür, obwohl eine solche Mehrfachverwendung von Bezeichnern irreführend sein kann – und von Visual Studio auch zuweilen sogar unter Ausgabe recht kryptischer Fehlermeldungen missinterpretiert wird, wenn unterschiedliche Assemblies im Spiel sind.



Obwohl `using`*-Direktiven genau dort zu finden sind, wo in C/C++-Dateien include-Anweisung stehen, haben sie mit diesen nichts gemein. Eine* `using`*-Direktive ist keine Referenz auf eine Bibliothek oder Datei und fügt auch keinen irgendwie gearteten externen Code in den Quelltext ein! Sie ist ein Hinweis für den Compiler, dass er Bezeichner, die ohne Qualifizierung notiert sind, gegebenenfalls in diesem Namenraum finden kann. (Delphi-Programmierer, die nun beifällig nicken und an die dort möglichen uses-Klauseln für Units denken, müssen sich mit dem Spruch bescheiden, dass fast getroffen eben auch vorbei ist. Was sich mit* `using` *auf der Sprachebene vergleichen lässt, ist nicht etwa* `uses`*, sondern die Pascal-Anweisung* `with`*.)*

Verschachtelung von Namensräumen

Namensräume lassen sich durch Unterordnung weiterer Namensräume nach Belieben strukturieren.

```
namespace MyNameSpace
{
  namespace MySubNameSpace
  {
    namespace MySubSubNameSpace
    {
      struct MyStruct
      { ... }
    }
..}
}
```

Eine andere Schreibweise für exakt dieselbe Definition wäre:

```
namespace MyNameSpace.MySubNameSpace.MySubSubNameSpace
{
  struct MyStruct
  { ... }
}
```

Auch hier gilt, dass die aufgeführten Namensräume – unter Umständen also die gesamte Hierarchie – bei Bedarf angelegt werden, also nicht unbedingt schon vorher existieren müssen. Um eine Variable mit dem Typ `MyStruct` zu vereinbaren, schreibt man

```
MyNameSpace.MySubNameSpace.MySubSubNameSpace.MyStruct myStruct;
```

bzw. mit Verwendung oder `using`-Direktive:

```
using MyNameSpace.MySubNameSpace.MySubSubNameSpace;
...
MyStruct myStruct;
```

Mit einer Qualifizierung nach dem folgenden Muster kann der Compiler hingegen nichts anfangen.

```
using MyNameSpace.MySubNameSpace;
...
MySubSubNameSpace.MyStruct myStruct;   // Fehlermeldung des Compilers
```

Dies zeigt, dass der C#-Compiler strukturierte Namensraumbezeichner als solche nicht auflöst (sie also nicht ihrerseits als Elemente eines Namensraums betrachtet) und auch der Namensraumstruktur von sich aus keine weitere Bedeutung beimisst.

Umbenennung von Namensräumen

Mehrfach verschachtelte Namensräume sind oft schlecht zu notieren. Auch kann es erforderlich sein, einen Namensraum im Nachhinein noch umzubenennen, wenn der Code bereits steht. Für diesen Fall gibt es die `using`-Anweisung (im Gegensatz zur Direktive). Sie vereinbart einen Alias für einen bestehenden Namensraum:

```
using NRaum = Namensraum.UnterNamensraum.UnterUnterNamensraum;
```



Obwohl die Umbenennung von Namensräumen an sich problemlos ist, sofern sie konsequent vorgenommen wird, kann es nach einer Umbenennung zu unerwarteten Problemen im Zusammenhang mit dem Einlesen von Ressourcen (Bitmaps, Symbole, Cursor-Formen) kommen. In einer Assembly gespeicherte Ressourcen ordnet der Compiler dem so genannten Standardnamensraum zu, der über die Eigenschaft `Standardnamespace` *auf der Eigenschaftsseite* ALLGEMEIN *des Projekts angepasst werden kann. Beim*

Anlegen des Codegerüsts setzt der Assistent den Wert dieser Eigenschaft auf den automatisch generierten Namensraum. Umbenennungen im Codegerüst schlagen aber nicht auf die Eigenschaft `Standardnamespace` *durch. Da zum Einbinden von Ressourcen ein beliebiger im Standardnamensraum definierter Datentyp anzugeben ist, müssen Sie den Wert dieser Eigenschaft gegebenenfalls an den neuen Namensraum anpassen, andernfalls erhalten Sie Laufzeitfehler beim Einlesen der Ressourcen.*

Regeln für Namensräume

Es gilt:

Typvereinbarungen finden immer in einem Namensraum statt. Typvereinbarungen, die keinem `namespace`-Block angehören, gehören implizit dem für jedes Projekt individuell existierenden bezeichnerlosen Namensraum an – und müssen darin gleichfalls eindeutig sein.

Die Wahl des Namensraums für einen Typbezeichner ist dem Programmierer grundsätzlich freigestellt. Die Gruppierung von Typbezeichnern in Namensräume ist eine schlichte Benennung und induziert eine rein logische Ordnung jenseits der physischen Repräsentation. Für Compiler und .NET macht es absolut keinen Unterschied, in welchem Namensraum ein Typ definiert wird. Insbesondere bringt eine tiefere Verschachtelung keine Laufzeitnachteile mit sich.

Namensräume können nach Belieben verschachtelt werden, wobei übergeordnete Namensräume auch unbesetzt bleiben dürfen.

Eine `using`-Direktive bezieht untergeordnete Namensräume nicht mit ein (keine Teilqualifizierung wie bei VBA oder der with-Anweisung von Delphi).

Steht eine `using`-Direktive außerhalb von Namensbereichsblöcken, gilt sie für den gesamten folgenden Quelltext; steht sie innerhalb eines Namensbereichsblocks (üblicherweise am Anfang), gilt sie nur für diesen Block:

```
using NamensraumX;  // gilt für den gesamten folgenden Quelltext

namespace NamensraumY
{
  using NamensraumZ;  // nur innerhalb dieses Blocks
  struct MeineWertKlasse
  {
  // ...
  }
}
```

5.1.2 Modifizierer

C# kennt eine bunte Palette von Modifizierern für die Vereinbarung von Datentypen und Methoden, deren Aufgabe – grob gesprochen – darin besteht, die Sichtbarkeit, Rolle und Zugriffsart einer Größe (Typdefinition, Methode, Datenfeld etc.) genauer zu spezifizieren.

Die prominentesten MODIFIZIERER sind die so genannten Zugriffsmodifizierer `public`, `private`, `protected` und `internal`. Sie regeln, in welchem Kontext ein Datenfeld, eine Methode oder ein Datentyp sichtbar ist.

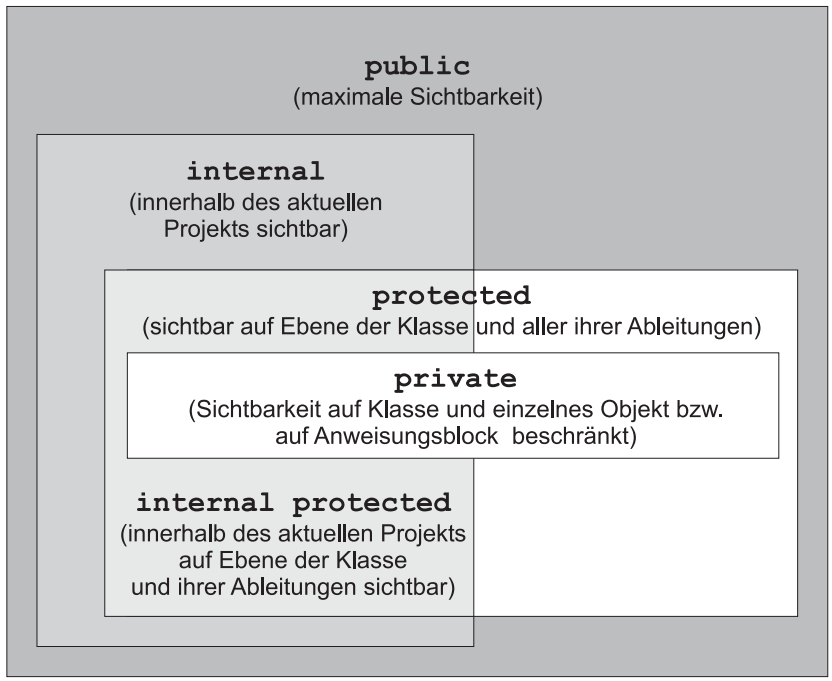


Abbildung 5.1: Wirkung der Zugriffsmodifizierer

Tabelle 5.1 gibt einen Überblick über alle Modifizierer von C# und ihre Bedeutung. Diese Tabelle zeigt auch, für welche Elemente und in welchem Kontext sie angewendet werden dürfen.

Tabelle 5.1: Überblick über die in C# verfügbaren Modifizierer und ihre Anwendbarkeit

Modifizierer	class-Klasse	struct-Klasse	Datenfeld	Methode	Wirkung
`abstract`	nicht mit sealed kombinierbar	–	–	möglich; nur innerhalb abstrakter Klasse	*Klasse:* Ist nur als Basisklasse verwend- und nicht instanziierbar; *Methode:* Muss von abgeleiteter Klasse überladen werden, wenn diese nicht ihrerseits abstrakt sein soll.
`const`	–	–	möglich; nicht mit `static` kombinierbar		Modifiziert Datenfeld zur Konstanten. Initialisierungsvereinbarung erforderlich (im Konstruktor ist keine Initialisierung nicht mehr möglich) und der Wert ist danach unveränderlich.
`event`	–	–	–	–	Deklariert ein Ereignis
`extern`	–	–	–	möglich; nicht mit `abstract` kombinierbar	Methode ist statisch und wird durch ein anderes Modul bereitgestellt.
`internal`	möglich (Standard)	möglich (Standard)	möglich	möglich	Zugriff auf aktuelles Projekt beschränkt
`internal protected`	möglich; wenn Vereinbarung verschachtelt	möglich; wenn Vereinbarung verschachtelt	möglich	möglich	Zugriff auf aktuelles Objekt und die jeweilige Klasse plus Ableitungen beschränkt

Modifizierer	class-Klasse	struct-Klasse	Datenfeld	Methode	Wirkung
`new`	möglich; wenn Vereinbarung verschachtelt	–	möglich	möglich	Gleichnamiges Element der Basisklasse wird überschrieben
`override`	–	–	–	möglich; nicht mit `static` oder `new` kombinierbar	Ersetzt Basisklassenmethode mit gleicher Signatur, die als `virtual`, als `abstract` oder ihrerseits als `override` vereinbart ist.
`private`	möglich; wenn Vereinbarung verschachtelt	möglich; wenn Vereinbarung verschachtelt	möglich (Standard)	möglich (Standard)	Zugriff nur innerhalb der enthaltenden Klasse
`protected`	möglich; wenn Vereinbarung verschachtelt	möglich; wenn Vereinbarung verschachtelt	möglich	möglich	Zugriff auf das Element auf die definierende Klasse und deren Ableitungen beschränkt
`public`	möglich	möglich	möglich	möglich	Uneingeschränkter Zugriff
`readonly`	–	–	möglich	–	Element darf nicht als Linkswert verwendet werden, außer bei Initialisierungsvereinbarung und in Konstruktor.

Tabelle 5.1: Überblick über die in C# verfügbaren Modifizierer und ihre Anwendbarkei (Forts.)

Tabelle 5.1: Überblick über die in C# verfügbaren Modifizierer und ihre Anwendbarkei (Forts.)

Modifizierer	class-Klasse	struct-Klasse	Datenfeld	Methode	Wirkung
`sealed`	nicht mit `abstract` kombinierbar	– (implizit immer)	–	–	Nicht als Basisklasse für die weitere Ableitung verwendbar
`static`	–	–	möglich	möglich	Element ist statisch (der statischen Instanz zugeordnet)
`unsafe`	möglich	möglich	möglich	möglich	Code unterliegt Einschränkungen hinsichtlich der Sicherheit. Erfordert das Setzen der Compileroption `/unsafe`.
`virtual`	–	–	–	möglich, nicht mit `static` kombinierbar	Methode ist virtuelles Element der Klasse

Spezifischere Informationen über die einzelnen Modifizierer finden Sie insbesondere in den folgenden beiden Kapiteln, die Klassen und Methoden ausführlicher vorstellen.

5.2 Benennungskonventionen

Um die Lesbarkeit von Programmen zu verbessern und die Kommunikation zwischen gemeinsam arbeitenden Programmierern zu vereinfachen, existieren für jede Programmiersprache und -umgebung bestimmte Konventionen, die weder Teil der Sprache noch der Spezifikation sind. Es handelt sich hier um pragmatische Richtlinien für die Benennung von Codegrößen (Konstanten, Parametern, Datentypen, Namensbereichen, Modulen etc.), die die Arbeit im Team vereinheitlichen, den Austausch und die Pflege von Quellcode vereinfachen, die Produktivität erhöhen und gleichzeitig die Fehlerquote verringern sollen.

5.2.1 Bezeichnerwahl

Was die Zusammensetzung eines Bezeichners betrifft, hält sich C# an die »üblichen Regeln«, die von anderen Programmiersprachen her schon bekannt sein dürften – falls nicht, belehrt einen im Zweifelsfall der Compiler über entsprechende Fehlermeldungen. Hier ein kurzer Überblick:

- Obermenge der für C#-Bezeichner zulässigen Zeichen ist die Menge der Unicode-Zeichen. Bei der Wahl der Codeseiten unterliegt man vom Prinzip her zwar keinen Einschränkungen, hierzulande dürfte es sich aber empfehlen, sich auf den deutschen (mit Umlauten) oder gar nur auf den englischen Buchstaben- und Ziffernsatz plus Unterstrich zu beschränken. Bezeichner wie `\u1234\u1242k\u0041` und `®ÆÁ_` sind also zulässig, Freunde oder auch nur Freude machen wird man sich damit aber langfristig nicht.
- Zwischen Groß- und Kleinbuchstaben wird (wie in C/C++ und Java, aber im Gegensatz zu VB und Delphi) unterschieden: `Zähler` und `zähler` stellen für C# also zwei unterschiedliche Bezeichner dar.
- Schlüsselwörter von C# *dürfen* nicht, Schlüsselwörter von VB.NET *sollten* nicht als Bezeichner verwendet werden. Da erstere vom VS.NET-Editor erkannt und farblich unterlegt werden und natürlich auch der Compiler ein wachsames Auge darauf hat, sollten hier keine Zweifelsfälle entstehen. Schlüsselwörter von VB.NET gehen hingegen »glatt« durch und bereiten erst Probleme, wenn die Komponente nach VB.NET importiert werden soll. Ihre Verwendung verstößt also gegen die Regeln des CLS. Tabelle 5.3 gibt einen Überblick über die für VB.NET reservierten Bezeichner.

  Verwendet man Code aus anderen Programmiersprachen, kann es vorkommen, dass in diesem Code Namen erscheinen, die in C# Schlüsselwörter darstellen. Diesen Namen stellt man im C#-Code das @-Zeichen voran.
- Schließlich muss der Name in seinem Gültigkeitsbereich eindeutig sein. Sie dürfen also nicht in einem Namensraum zwei Klassen mit gleichem Namen oder in einer Klasse zwei Methoden gleichen Namens definieren.

Ansonsten sollte der Name nicht zu lang, aber dennoch aussagekräftig sein, damit man am Namen die Verwendung des Elements ablesen kann. Für Hilfsvariablen, die keine besonderen Daten speichern, gibt es aber meist keine sinnvollen Namen. Solche Variablen heißen dann `n`, `m`, `x`, `y` oder `tmp`. Hier sollten Sie aber darauf achten, dass die Zeichen `l`, `1`, `i`, `j` und `t` sowie `n` und `m` optisch oft schwierig zu unterscheiden sind.

Tabelle 5.2: Für C# reservierte Schlüsselwörter

C#-Schlüsselwort	C#-Schlüsselwort	C#-Schlüsselwort	C#-Schlüsselwort
abstract	enum	long	stackalloc
As	event	namespace	static
Base	explicit	New	string
Bool	extern	null	struct
break	false	object	switch
Byte	finally	operator	this
Case	fixed	Out	throw
catch	float	override	true
Char	for	params	try
checked	foreach	private	typeof
class	goto	protected	uint
const	if	public	ulong
continue	implicit	readonly	unchecked
decimal	in	Ref	unsafe
default	int	return	ushort
delegate	interface	sbyte	using
Do	internal	sealed	virtual
double	is	short	void
else	lock	sizeof	while



Wenn importierte Bezeichner mit C#-Schlüsselworten kollidieren, ist in C# der Ofen noch nicht aus. In diesem Fall haben Sie die Möglichkeit, dem Bezeichner das Zeichen @ voranzustellen. C# nimmt den Bezeichner dann »wörtlich« und interpretiert ihn nicht als Schlüsselwort:

```
bool fixed = false;   // Fehlermeldung
bool @fixed = false;  // nicht empfehlenswert, aber okay
```

Tabelle 5.3: Für VB.NET reservierte Schlüsselwörter

VB-Bezeichner	VB-Bezeichner	VB-Bezeichner	VB-Bezeichner
Abs	Add	AddHandler	AddressOf
Alias	And	Ansi	AppActivate
Append	As	Asc	Assembly
Atan	Auto	Beep	Binary

Tabelle 5.3: Für VB.NET reservierte Schlüsselwörter (Forts.)

VB-Bezeichner	VB-Bezeichner	VB-Bezeichner	VB-Bezeichner
BitAnd	BitNot	BitOr	BitXor
Boolean	ByRef	Byte	ByVal
Call	Case	Catch	CBool
Cbyte	CDate	CdbI	Cdec
ChDir	ChDrive	Choose	Chr
Cint	Class	Clear	CLng
Close	Collection	Command	Compare
Const	Cos	CreateObject	CShort
CSng	CStr	CurDir	Date
DateAdd	DateDiff	DatePart	DateSerial
DateValue	Day	DDB	Decimal
Declare	Default	Delegate	DeleteSetting
Dim	Dir	Do	Double
Each	Else	ElseIf	Empty
End	Enum	EOF	Erase
Err	Error	Event	Exit
Ex	Explicit	ExternalSource	False
FileAttr	FileCopy	FileDateTime	FileLen
Filter	Finally	Fix	For
Format	FreeFile	Friend	Function
FV	Get	GetAllSettings	GetAttr
GetException	GetObject	GetSetting	GetType
GoTo	Handles	Hex	Hour
If	IIf	Implements	Imports
In	Inherits	Input	InStr
Int	Integer	Interface	IPmt
IRR	Is	IsArray	IsDate
IsDbNull	IsNumeric	Item	Kill
LCase	Left	Lib	Line
Loc	Local	Lock	LOF

Tabelle 5.3: Für VB.NET reservierte Schlüsselwörter (Forts.)

VB-Bezeichner	VB-Bezeichner	VB-Bezeichner	VB-Bezeichner
Log	Long	Loop	LTrim
Me	Mid	Minute	MIRR
MkDir	Module	Month	MustInherit
MustOverride	MyBase	MyClass	Namespace
New	Next	Not	Nothing
NotInheritable	NotOverridable	Now	NPer
NPV	Null	Object	Oct
Off	On	Open	Option
Optional	Or	Overloads	Overridable
Overrides	ParamArray	Pmt	PPmt
Preserve	Print	Private	Property
Public	Put	PV	QBColor
Raise	RaiseEvent	Randomize	Rate
Read	ReadOnly	ReDim	Remove
RemoveHandler	Rename	Replace	Reset
Resume	Return	RGB	Right
RmDir	Rnd	RTrim	SaveSettings
Second	Seek	Select	SetAttr
SetException	Shared	Shell	Short
Sign	Sin	Single	SLN
Space	Spc	Split	Sqrt
Static	Step	Stop	Str
StrComp	StrConv	Strict	String
Structure	Sub	Switch	SYD
SyncLock	Tab	Tan	Text
Then	Throw	TimeOfDay	Timer
TimeSerial	TimeValue	To	Today
Trim	Try	TypeName	TypeOf
UBound	Ucase	Unicode	Unlock

VB-Bezeichner	VB-Bezeichner	VB-Bezeichner	VB-Bezeichner
Until	Val	Weekday	While
Width	With	WithEvents	Write
WriteOnly	Xor	Year	

Tabelle 5.3: Für VB.NET reservierte Schlüsselwörter (Forts.)

Ungarische Notation

In der klassischen Windows-Programmierung ist die nach Microsofts langjährigen Chefprogrammierer Charles Simonyi benannte *ungarische Notation* recht weit verbreitet. Ihr liegt der Gedanke zugrunde, den Datentyp einer Variable durch Wahl eines »sprechenden« Bezeichnerpräfixes nach außen hin kenntlich zu machen. So findet man in VB6 für String-Bezeichner zumeist das Präfix »s« oder »str«, für Integervariablen das Präfix »i« oder »int« etc. In C++ haben sich über die Windows-SDKs Präfixe wie »psz« oder »lpsz« (für Variablen des Typs `char *`) eingebürgert, die dem Leser des Quelltextes das Nachschlagen des Variablentyps, zumindest für die verbreiteten Datentypen, weitgehend ersparen.

Für die .NET-Programmierung rät Microsoft aber von der ungarischen Notation ab, da die Kontextsensitivität von VS.NET so weit gediehen ist, dass der Programmierer die Art und den Datentyp einer Variable jederzeit in Erfahrung bringen kann, wenn er nur den Mauszeiger auf die fragliche Variable setzt (vgl. Abbildung 5.2). Gleiches gilt für die Überladungen und Parameter von Methoden. Der Editor von VS.NET zeigt die Signaturen als Tooltip an (Abbildung 5.3).

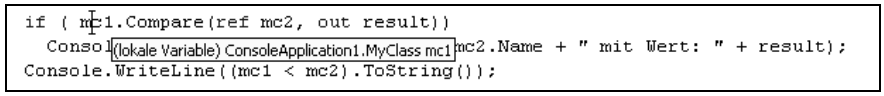


Abbildung 5.2: Kontextsensitive Datentypanzeige in VS.NET

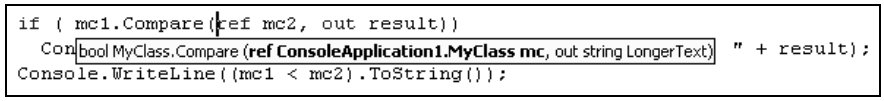


Abbildung 5.3: Kontextsensitive Signaturanzeige in VS.NET

Eigene Benennungssysteme

In der Tat leisten Benennungssysteme wichtige Schützenhilfe, wenn es um das Ausdenken von Namen und Bezeichnern geht. Wer die Wahl hat, hat bekanntlich auch die Qual, und da hilft es schon, wenn man sich dabei an formalen Kriterien orientieren kann.

Während sich die Benennungskonventionen für die meisten Sprachen schlicht über den Austausch von Programmierer zu Programmierer im Lauf

der Zeit herauskristalliert haben, hat Microsoft .NET (wie seinerzeit für die SDKs) die Richtlinien für die Entwicklung gleich mit auf den Weg gegeben. Diese Richtlinien entspringen einerseits natürlich dem Willen, die Arbeit im eigenen Konzern bestmöglich zu koordinieren, andererseits sind sie aber auch Ausdruck langjähriger kollektiver Erfahrung im Umgang mit objektorientierter Programmierung, die von Entwicklern aus aller Welt in Newsgroups und Entwicklerforen zusammengetragen wurden, um schließlich von Microsoft für .NET zu adaptiert zu werden.

Damit wir uns recht verstehen: Obwohl Sie diesen Richtlinien nach Möglichkeit folgen sollten, um die (ggf. auch spätere) Zusammenarbeit mit anderen Entwicklern von Anfang an auf ein gesundes Fundament zu stellen, besteht weder ein Zwang noch sind sie immer der Weisheit letzter Schluss. (Tatsächlich werden Sie feststellen, dass sich auch die Beispiele dieses Buches nicht immer sklavisch an die Regeln halten.) Wenn Sie zu der Auffassung gelangen, dass Sie für Ihren Anwendungsbereich besser bedient sind, indem Sie die eine oder andere Richtlinie nicht beherzigen oder ihr gar zuwider handeln, dann nur zu. Auch besteht kein Grund, unternehmensweit eingeführte Standards sofort über Bord zu werfen und blind dem Ruf von Microsoft zu folgen. Solange Ihr Vorgehen in dem anvisierten Rahmen konsistent bleibt und Ihrer Arbeitsweise entgegen kommt, werden Sie auch unter .NET bestens zurecht kommen. Stellen Sie sich dann aber darauf ein, dass Sie Ihre abweichenden Richtlinien nach außen hin kommunizieren und vor allem dokumentieren müssen.

5.2.2 Schreibweise von Bezeichnern

Wenn Sie von der C/C++-Programmierung her kommen, werden Sie es vielleicht noch gewohnt sein, zusammengesetzte Bezeichner mit Unterstrichen zu strukturieren. Für C# im Besonderen und .NET im Allgemeinen empfiehlt Microsoft nun die von Pascal her kommende Schreibweise, bei der jedes (semantisch) unabhängige Teilwort mit einem Großbuchstaben beginnt. Das Ergebnis ist intuitiv und gut lesbar. Ein Beispiel:

```
const int MaxBufferSize = 1024;
string KundeAnschriftStrasse;
```

Obwohl C# zwei Bezeichner bereits unterscheidet, wenn auch nur ein Zeichen von der Klein- und Großschreibung her abweicht, ist es mit Blick auf die Codekompatibilität zu anderen Sprachen nicht zu empfehlen, diese Unterscheidung für `public`-Elemente auszureizen, da nicht alle .NET-Sprachen Groß-/Kleinschreibung unterscheiden. Während C# die folgende Vereinbarung klaglos akzeptiert, wäre eine Verwendung der C#-Klasse `DocumentDescriptor` von VB.NET oder Delphi.NET aus beispielsweise nicht möglich:

```
namespace Documents
{
  public enum documentDescriptor {Gif, Jpg, Bmp, Tif, Wmf};
  public class DocumentDescriptor
  {
  ...
  }
}
```

`public`-Bezeichner sollten also in jedem Fall über die Schreibweise hinaus im jeweiligen Namensraum eindeutig sein. Es ist in der Tat weit verbreitet, Objekt und Klasse nur über die Schreibweise zu unterscheiden, etwa das Objekt mit kleinem Anfangsbuchstaben und die Klasse mit großen Anfangsbuchstaben zu bezeichnen. Von dieser Praxis müssen Sie nicht abweichen, solange nur die Klasse als `public` vereinbart ist, das Objekt jedoch nicht.

5.2.3 Reihenfolge bei der Strukturierung

Bei der Bezeichnerstrukturierung unterscheidet man zwischen der *VerbObjekt*- und der *ObjektVerb*-Reihenfolge, je nachdem, ob zuerst das Verb oder zuerst das (gegebenenfalls zusammengesetzte) Objekt genannt wird. Sie haben die Wahl zwischen `ActivateCurrentClient()`, `ClientCurrentActivate()` und `CurrentClientActivate()`, wobei Sie sich bevorzugt zwischen der ersten und zweiten Reihenfolge entscheiden sollten. Nicht in Betracht kommt `ClientActivateCurrent()`. Hier steht das Verb in der Mitte, sodass der Bezug an sich nicht ganz klar wird.

Vermeiden Sie bei der Wahl der einzelnen Bezeichnerteile Redundanzen. Anstatt `CustomDatabaseOpenDatabase()` reicht also `CustomDatabaseOpen()` bzw. `OpenCustomDB()`.

Wählen Sie das Strukturierungssystem, das Ihnen am besten geeignet erscheint, und bleiben Sie dann konsequent. Mischungen verschlechtern die Lesbarkeit und Verständlichkeit Ihres Codes und verschleiern die Struktur wiederverwendbarer Komponenten.

5.3 Integrierte vs. benutzerdefinierte Datentypen

Traditionell unterscheidet man bei einer Programmiersprache zwischen den integrierten und den benutzerdefinierten (vom Programmierer festgelegten) Datentypen. Die INTEGRIERTEN DATENTYPEN sind – bildhaft ausgedrückt – die Aussteuer der Sprache. Ihre Repräsentationen und – damit zusammenhängend – ihre Wertebereiche sowie ihre grundlegenden Operationen sind fest vorgegeben. Damit sind sie Grundlage jeglicher Manipulationen und Interpretationen von Daten, die in dieser Sprache formuliert werden. Die BENUTZERDEFINIERTEN DATENTYPEN, oft besser als KOMPLEXE DATENTYPEN

bezeichnet, lassen sich, wie der Name schon sagt, mit den Mitteln der Sprache frei definieren. Sie sind sozusagen der Stoff, aus dem Bibliotheken gemacht werden. Ihre Repräsentation erfolgt letztlich auf der Basis der integrierten Datentypen, ebenso ihre Manipulation. Für die Implementierung – also die Interpretation der Repräsentation und die Ausstattung mit Operationen –, stehen der gesamte Sprachumfang und die Gesamtheit aller bereits definierten komplexen Datentypen zur Verfügung.

Wie alle .NET-Sprachen bezieht auch C# seine integrierten Datentypen durch einfache Umbenennung aus den .NET-Basisklassen und stellt darüber hinaus verschiedene sprachliche Mittel bereit, die eine komfortable Implementierung benutzerdefinierter Datentypen nach den Regeln der objektorientierten Programmierung ermöglichen.

5.4 Wert- vs. Verweistypen

Eine andere Einteilung der Datentypen ist die in Wert- und Verweistypen. *Werttypen* repräsentieren einen konkreten Wert, etwa eine Zahl oder eine Bitfolge, *Verweistypen* die Adresse eines Werttyps oder eines weiteren Verweistyps.

Werttypen als solche sind unkompliziert, und ihre Handhabung ist mit keinerlei Risiken verbunden. Anders die Verweistypen, für die sich wahre Abgründe an konzeptuellem Ballast und Sicherheitsüberlegungen auftun.

Sprachen wie C/C++ und Pascal/Delphi erlauben es dem Programmierer, Zeigertypen zu vereinbaren, um Adressen als solche zu manipulieren und somit vom Prinzip her jede beliebige Speicherzelle in dem für die jeweilige Anwendung sichtbaren (virtuellen) Adressraum anzusprechen. Die Typisierung bietet dem Compiler zwar grundsätzlich die Möglichkeit, die konsistente Anwendung von Zeigern durchzusetzen – da die Sprachen aber die Mittel der Adressarithmetik und der expliziten Typumwandlung bereitstellen, steht die Hintertür für alle möglichen Fehler und natürlich auch Unsinn weit offen. Für die gewöhnliche Anwendungsentwicklung birgt diese Form der maschinenahen Programmierung mit umgehbaren »Fingerzeigen« des Compilers ein erhebliches Potenzial an Fallstricken und Sicherheitsrisiken: Die Sprachkonzeption selbst konterkariert hier also den Anspruch des Compilers, möglichst fehlerfreien und sicheren Code zu erzeugen.

5.4.1 Abstrakte Datentypen

Mit dem Siegeszug der objektorientierten Programmierung rückte der so genannte ABSTRAKTE DATENTYP mit seiner verkapselten Implementierung in den Mittelpunkt des Interesses – zuerst für die Anwendungsentwicklung, bald aber auch für die Sprachentwicklung. Auf die Sprachebene transfor-

miert bietet dieses Konzept insbesondere die Möglichkeit, die unseligen Zeiger mit den rein formalen Mitteln des Typsystems so zu verpacken, dass sie für den Programmierer nicht mehr sichtbar und damit auch seiner Willkür nicht mehr ausgeliefert sind.

Mit rein formalen Verweistypen anstelle von Zeigern auszukommen, bedeutet für eine Sprache insbesondere, dass sie

- eine äußerst solide Grundlage für die Umsetzung von Sicherheitskonzepten bietet
- und auf ein Laufzeitsystem mit automatischer Freispeicherverwaltung angewiesen ist.

Anders gesagt, der Verweistyp resultiert aus dem Bedürfnis, das Zeigerkonzept aus dem Sprachumfang generell zu verbannen – zugunsten eines Sicherheitskonzepts, das allerdings nur im Doppelpack mit einer geeignet ausgestatteten Laufzeitumgebung zu erhalten ist.

Ein großer Teil des Erfolgs von Java und VB lässt sich darauf zurückführen, dass diese Sprachen ohne explizites Zeigerkonzept auskommen. Bei VB war es die Sprachtradition, die nach einer zeigerfreien Lösung verlangte, bei Java die ausgeprägte Sicherheitsphilosophie. .NET stellt ein sprachunabhängiges verteiltes Objektmodell mit ausgefeiltem Sicherheitskonzept vor. Zeigerarithmetik in der Hand des Programmierers hat darin keinen Platz mehr. Delphi bis einschließlich Version 6 präsentiert sich sozusagen als Zwischenlösung: Objekte werden bei diesem Compiler über (versteckte) Zeiger angesprochen, eine Freispeicherverwaltung gibt es dort aber nicht. Konsequente Folge: Während Arithmetikfehler bei der Objektadressierung äußerst selten vorkommen, erzeugen Zugriffsversuche auf bereits abgebaute Objekte nur im Glücksfall Schutzverletzungen – wenn nicht, kann man unter Umständen tagelang nach dem Fehler suchen.

Bleibt die Frage, wie die Zusammenarbeit mit Code aussieht, der in Sprachen mit konventioneller Zeigerarithmetik geschrieben wurde? Um ohne Einschränkung von der .NET-Architektur zu profitieren, müssen Sprachen mit explizitem Zeigerkonzept wie C/C++ und Pascal/Delphi eine Reihe von syntaktischen Einschränkungen hinnehmen, deren Einhaltung der Compiler durchsetzt.

5.4.2 Gibt es denn Zeiger in C#?

Als .NET-Sprache Nummer Eins kommt C# bestens ohne Zeigerkonzept zurecht. Die Syntax ist so ausgelegt, dass der Programmierer auch ohne mit Zeigern zu hantieren alles formulieren kann, was für die Programmierung unter .NET erforderlich ist. Sämtliche Verweise und Zeigeroperationen ver-

stecken sich hinter Typnamen, impliziten Typvereinbarungen (durch den Compiler) und syntaktischen Konstrukten wie dem Typumwandlungsoperator.

Das soll aber nicht heißen, dass es in C# kein Zeigerkonzept gäbe. Als C/C++-Derivat kennt C# durchaus explizite Zeiger, und man kann sie auch benutzen – beispielsweise, wenn es darum geht, ein Stück Code auf maximale Ausführungsgeschwindigkeit zu trimmen, die Kompatibilität mit älterem Code aufrecht zu erhalten, die Windows-API direkt anzusprechen oder dem Benutzer schlicht konkrete Adressen für irgendwelche Datenstrukturen zu nennen.

Codeblocks, die explizit mit Zeigern arbeiten, müssen mit dem Modifizierer `unsafe` kennzeichnet werden. Ihr Programmcode verliert dadurch eine Reihe von Privilegien und Bequemlichkeiten gegenüber originärem .NET-Code – beispielsweise ist hier die Freispeicherverwaltung von .NET außer Kraft.

5.4.3 Heap und Stack

Auch aus anderem Grund ist die Unterscheidung zwischen Wert- und Verweistyp naheliegend. Aus dem Blickwinkel der lokalen Variablen (= Stackrahmen) einer Methode ist der Speicherort für Werttypen traditionell der STACK, während Verweistypen auf Daten referieren, die auf dem HEAP untergebracht sind (also in einem dem Programm auf Anforderung vom Betriebssystem zugeteilten Speicherbereich). Auf dem Heap gespeicherte Werte sind generell darauf angewiesen, dass es Verweise (oder Verweisketten) auf sie gibt, die ihrerseits vom Stack ausgehen und den eigentlichen (direkten) Wert eines Verweistyps darstellen. Abbildung 5.4 verdeutlicht diesen Zusammenhang.



Es versteht sich, dass auch auf dem Heap gespeicherte Werte (Instanzen von Klassen) sowohl Werte von Werttypen als auch von Verweistypen sein können. Das trifft generell für alle Datenfelder einer `class`*-Instanz zu. Die Verweiskette nimmt aber grundsätzlich ihren Ausgangspunkt im Stack – und hier im Stackrahmen der Methode* `Main()`.

Mit .NET hat dieser an sich uralte Dualismus nun endlich auch in konzeptueller Hinsicht einen deutlich sichtbaren Niederschlag gefunden – nicht zuletzt deshalb, weil sich der Programmierer aufgrund der im .NET-Framework vorhandenen Freispeicherverwaltung nun nicht mehr um die Verwaltung des Heap kümmern muss (bzw. darf). Kurz gesagt:

- Variablen mit Werttyp *stehen für* einen Wert (im Stack)
- Variablen mit Verweistyp *verweisen auf* einen Wert (im Heap).

Der Unterschied zwischen einem Wert- und einem Verweistyp manifestiert sich im Allgemeinen in der Wirkungsweise der Zuweisungsoperation »=«. Bei einem Werttyp erzeugt die Zuweisungsoperation eine physische Kopie des Wertes, bei einem Verweistyp hingegen nur einen weiteren Verweis auf ein und denselben Speicherbereich.

Keine Regel ohne Ausnahme! Auch C# bzw. .NET hat seine Brüche, so beispielsweise, wenn es um die Frage geht, ob Strings nun als Wert- oder Verweistyp gelten. Um es vorweg zu nehmen: Sie sind ein Verweistyp, werden aber wie ein Werttyp gehandhabt und verhalten sich auch so.

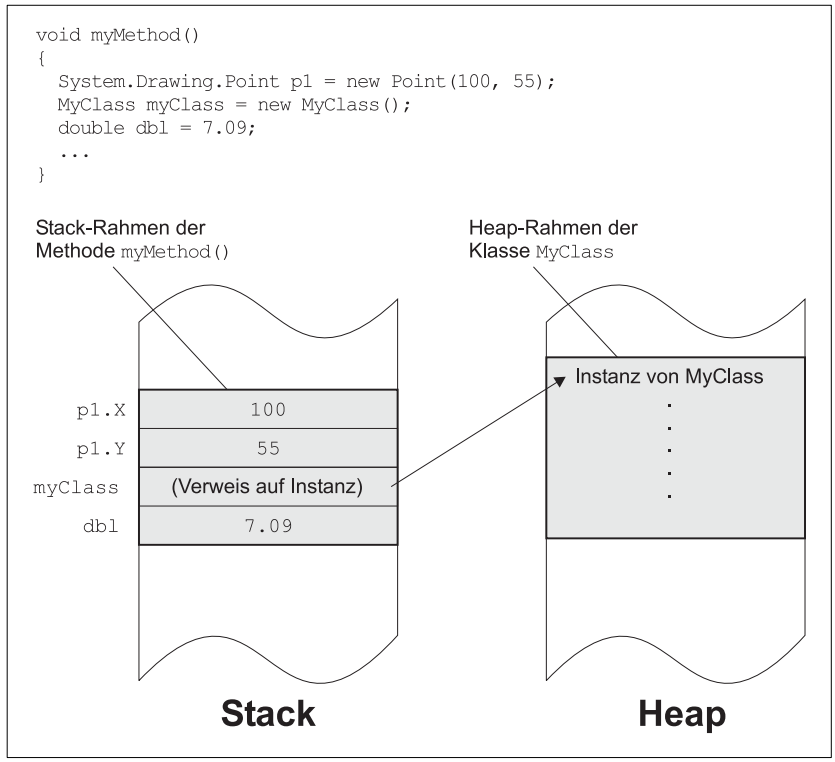


Abbildung 5.4: Speicherort für Verweistypen und Werttypen

5.4.4 Werttypen

Die etwas unglücklich klingende neudeutsche Wortschöpfung »Werttyp« hat sich als allzu wörtliche Übersetzung von *value type* über die MSDN-Hilfe eingebürgert. Ein WERTTYP repräsentiert einen Wert unmittelbar, ohne jegliche Indirektion. Der Wert hat eine feste, bei der Deklaration bereits bekannte Länge, und sein Speicherbereich wird im Zuge der Vereinbarung fest auf dem Stack belegt. Für lokale Variablen von Methoden gilt: C# initialisiert Werttypen von sich aus *nicht* mit Standardwerten, daher müssen sie vor ihrem ersten Gebrauch als RECHTSWERT explizit initialisiert

werden – durch Zuweisung, per Konstruktor (bei `struct`-Klassen) oder als `out`-Parameter einer Methode. Datenfelder von Klassen initialisiert der Compiler hingegen automatisch entweder mit 0 oder `null` – er erzeugt aber eine Warnung, falls er weder bei Vereinbarung noch im Konstruktor eine explizite Initialisierung vorfindet.

Sprachintern verfügt C# über dreizehn integrierte Werttypen und ermöglicht selbstverständlich auch – mit `enum` und `struct` – eigene Definitionen. Darüber hinaus finden sich in den verschiedenen Namensbereichen der .NET-Basisklassen die Implementierungen zahlloser von Microsoft vordefinierter Werttypen, die im Allgemeinen als `struct`-Klassen gehalten und von der abstrakten Klasse `ValueType` abgeleitet sind – für Windows-Programmierung zentrale Datentypen sind beispielsweise die zum Namensbereich `System.Drawing` gehörenden Typen `Rectangle`, `Size`, `Point RectangleF`, `SizeF` und `PointF`.

5.4.5 Verweistypen

Verweistypen repräsentieren einen Verweis auf einen Wert (im Allgemeinen ein Objekt), der auf dem Heap untergebracht ist. Damit ist der Wert eines Verweistyps genau genommen nichts weiter als eine Adresse, die als solche jedoch in der Sprache C# weder direkt in Erscheinung tritt noch explizit manipulierbar ist – zumindest, solange kein UNSICHERER CODE im Spiel ist. Dies ist der Preis für die Teilnahme an der verwalteten Umgebung. Der Speicherort für die Adresse ist natürlich – sofern nicht weitere Indirektionen im Spiel sind – der Stack.

Mehrere Verweise auf ein und denselben Wert sind möglich. Ein auf dem Heap gelegener Wert existiert nur so lange, wie Verweise darauf existieren. Verwaiste Werte werden früher oder später von der Freispeicherverwaltung eliminiert. Der genaue Zeitpunkt, zu dem dies passiert, ist nicht festgelegt, da diese Aufräumarbeiten in den Aufgabenbereich des .NET-Frameworks fallen und asynchron geschehen (nämlich über einen eigenen im Hintergrund mit niedriger Priorität laufenden Thread).

5.5 Alle Datentypen sind Klassen

C# versteht sich als vollständig objektorientierte Sprache. Somit sind alle Datentypen – insbesondere auch die integrierten – Klassen im eigentlichen Sinne der OOP. Wenn Sie von einer klassischen Programmiersprache her kommen, werden Sie diese Aussage sicherlich nicht ohne Stirnrunzeln zur Kenntnis genommen haben. Zu Recht, denn bisher hatte noch jede objektorientierte prozedurale Sprache, sei es nun C++, Delphi oder Java einen »harten« Kern einfacher (»primitiver«) Datentypen, die *keine* Klassen im Sinne der objektorientierten Programmierung waren. Diese Datentypen

besaßen weder Eigenschaften noch Methoden, kamen selbstredend auch nicht als Basisklassen für eigene Ableitungen in Frage, und die Sprache selbst stellte die notwendigen Standardoperationen sowie Operatoren bereit.

5.5.1 Ausführungsgeschwindigkeit

Das hatte natürlich seinen Grund: die Ausführungsgeschwindigkeit. Klassisch implementierte Datentypen sind nun einmal leichtfüßiger als Objekte mit ihrem Ballast an virtuellen Methodentabellen, Konstruktoren usw. Objektorientierung hat eben ihren Preis: Sie verschlingt Laufzeit und Speicherplatz, also Ressourcen, mit denen ein Compiler gerade bei den grundlegenden Datentypen äußerst penibel haushalten sollte.

Von diesem Argument kann sich auch ein .NET-Framework nicht freimachen – CTS hin oder her. Möge der Formalismus seine Blüten treiben: »Schein« und »Sein« waren noch nie dasselbe. Aber seien wir ehrlich: Von der Auffassung her bereitet es in der Tat keine Schwierigkeiten, selbst den primitivsten Datentyp als Klasse und dessen Wert als Objekt zu betrachten. Warum also nicht gleich alle Datentypen einer Sprache zu Klassen erklären und formal gleich behandeln? Das gibt nach außen hin ein homogenes Bild. Und nach innen hin kann der Compiler ja darauf getrimmt werden, die einen Klassen als »echte« Klassen und die anderen als klassisch-harte Datentypen zu implementieren. Im Endeffekt ist das schließlich eine Frage der Typbibliotheken.

5.5.2 Der Modifizierer sealed

Genau so halten es C# bzw. das .NET-Framework auch: Die elementaren Basistypen des CTS sind »hart« implementiert und stehen deshalb auch nicht als Basisklassen für eigene Ableitungen zur Verfügung – wohl aber haben sie Methoden und sind ihrerseits Ableitungen von `object` bzw. `System.ValueType`. Formal wird dieser Umstand durch den neuen Modifizierer `sealed` ausgedrückt. Eine als `sealed` vereinbarte Klasse ist vom Mechanismus der Vererbung ausgeschlossen, kann selbst aber Ableitung einer gewöhnlichen Klasse sein. So lautet die Vereinbarung des integrierten Datentyps `string` alias `System.String` beispielsweise:

```
public sealed class String : IComparable, ICloneable, IConvertible,
                             IEnumerable
```

Mit diesem Kunstgriff ist die Illusion perfekt. Selbst ein so altehrwürdiger Datentyp wie `int` ist damit eine Klasse (genauer: eine `struct`-Klasse), wie der Editor von VS.NET eindrucksvoll offenbart (vgl. Abbildung 5.5). Vier der sechs Methoden sind formal von der Mutter aller Klassen, `object`, ererbt, die anderen beiden, `CompareTo()` und `GetTypeCode()`, von der Basisklasse aller `struct`-Klassen bzw. Werttypen: `System.ValueType`.

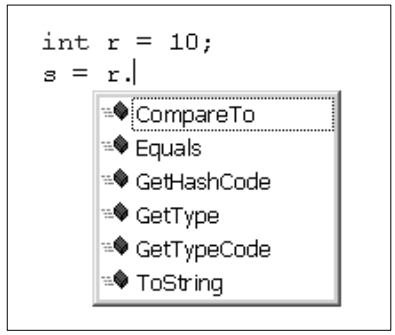


Abbildung 5.5: Visual Studio .NET enttarnt einen int-Wert als Objekt.

5.6 Strikte Typisierung

Als Abkömmling von C/C++ ist C# die strikte Typisierung nicht nur in die Wiege gelegt, die Sprache ist hier zuweilen sogar päpstlicher als der Papst. So verweigert der C#-Compiler beispielsweise implizite Zuweisungen zwischen Aufzählungstypen (vgl. `enum`) und Ganzzahltypen, wenn keine explizite Typumwandlung angegeben ist, und lehnt Typumwandlungen für den logischen Datentyp `bool` generell ab.

Wie nicht anderes zu erwarten, nimmt es der C#-Compiler auch mit der Typsicherheit äußerst genau und gibt sich Mühe, fehlerträchtigen Konstrukten, die in C/C++ völlig legitim waren, über eine strengere Syntax zu Leibe zu rücken. Zweifelhaften Ruhm hat hier beispielsweise die Verwechslungsmöglichkeit zwischen dem Zuweisungsoperator »=« und dem Vergleichsoperator »==« erlangt. Diese in der laxen C/C++-Syntax begründete Falle führt insbesondere im Zusammenhang mit Bedingungsausdrücken zu schwer aufzufindenden Fehlern. Nicht zuletzt aus diesem Grund behandelt C# den Datentyp `bool` äußerst restriktiv und schreibt vor, dass der Bedingungsausdruck in `if`-Konstrukten und bei Abbruchkriterien von Schleifen ein `bool`-Wert sein muss.

5.6.1 .NET-Basisklassen, eine üppige Grundausstattung

Strikte Typisierung, Typsicherheit und Typvielfalt gehen Hand in Hand. Traditionelle Programmiersprachen stellen jeweils ihren eigenen Fundus an vordefinierten Datentypen und Typbibliotheken zur Verfügung – und sorgen so ungewollt dafür, dass der Zugriff auf sprachunabhängige Objektstandards (wie COM) nicht nur laufzeitintensiv, sondern auch von der Implementierung her erheblichen Einschränkungen unterworfen ist. Beispielsweise bleiben Vererbungsmechanismen auf abstrakte Schnittstellendeklarationen beschränkt, wenn mehr als eine traditionelle Programmiersprache im Spiel ist – an eine Vererbung von Basisklassencode im Sinne der klassischen objektorientierten Programmierung ist hier nicht zu denken.

Genau dieses Problem packt .NET an der Wurzel. Die konzeptuelle Überlegenheit dieser Technologie resultiert unter anderem daraus, dass das so genannte Common Type System (CTS) die notwendige Infrastruktur für ein weitgehend sprachunabhängiges Zusammenspiel von Datentypen auf objektorientierter Basis schafft, auf die alle .NET-Sprachen aufbauen müssen. Eine Einschränkung des CTS, das Common Language System (CLS), gibt darüber hinaus einen Kompatibilitätsstandard für die sprachübergreifende Nutzung von Datentypen vor. Das heißt im Einzelnen:

- In anderen Programmiersprachen implementierte Datentypen können so benutzt werden, als wären sie in der eigenen Sprache geschrieben – und umgekehrt. Insbesondere steht die Implementierung solcher Datentypen auch für eigene Ableitungen zur Verfügung, weshalb diese Datentypen echte Basisklassen im Sinne der objektorientierten Programmierung sind.
- Aus Sicht einer Sprache macht es keinen Unterschied mehr, ob ein Datentyp in der gleichen Sprache oder einer anderen Sprache implementiert ist. Aus Sicht des Programmierers spielt es ebenfalls keine Rolle, weil die Laufzeitunterschiede tatsächlich marginal sind.
- .NET kann allen Sprachen ein und dieselbe umfangreiche Basisklassen-Bibliothek als »Erbe« zur Verfügung stellen, in dem die grundlegenden sowie auch eine Unmenge komplexer Datentypen sprachunabhängig vordefiniert sind.

5.6.2 C# hat ein Heimspiel

Gegenüber C/C++ ist C# als Derivat dieser Sprachen mit einer erweiterten Menge an integrierten Datentypen ausgestattet (besonders hervorzuheben: der Datentyp `string`). Zudem ist der wahrlich als opulent zu bezeichnende Typfundus der .NET-Basisklassen für die native .NET-Sprache C# vom Prinzip her ein echtes Heimspiel, da die .NET-Klassenhierarchie größtenteils in C# implementiert wurde.

Kurzum, der C#-Programmierer findet von Anfang an eine schier endlose Fülle an Datentypen vor, die nahtlos in die Sprache integriert sind. Dieses Typsystem kann er nach Belieben aufstocken, um an passender Stelle eigene Datentypen einzureihen, die dann wiederum allen anderen .NET-Sprachen zugute kommen können.

5.6.3 Typorganisation

Es versteht sich, dass eine so enorme Menge an Typen nach einer verbesserten Organisationsform und einem zentralisierten Typmanagement verlangt. Mit dem bereits erläuterten Konzept der Namensräume bietet .NET ein ausgefeiltes *Typmanagement*, das dem Programmierer den bequemen Zugriff

auf die Klassen der .NET-Klassenbibliothek, sonstige gemeinsam genutzte .NET-Klassen (unabhängig von der zur Implementierung verwendeten Sprache) sowie seine selbst definierten Klassen nach ein und demselben Schema gestattet.

Mit anderen Worten: Aus Sicht der Anwendung finden sich alle Datentypen zu einer einzigen übergeordneten .NET-Klassenhierarchie zusammen.

5.7 Zeiger ade

Basic kommt seit jeher ohne Zeigerkonzept aus und wurde unter anderem wohl auch deshalb nie so richtig ernst genommen. Es ist daher nicht verwunderlich, dass VB-Programmierer, die sich nicht gerade die API-Programmierung mit VB als akademische Disziplin ausgesucht haben, mit Zeigern eher wenig bis gar nichts am Hut haben und daher wohl auch keine rechte Vorstellung mit diesem Konzept verbinden. Auf der anderen Seite bezogen die typischen Compiler-Sprachen – Pascal/Delphi, C/C++ – einen großen Teil ihres Selbstverständnisses (und zugegeben auch ihres Potenzials) aus der Möglichkeit, mit Adressen zu jonglieren und die entsprechende Arithmetik dafür bereitzustellen.

C# versteht sich (ebenso wie nun auch VB.NET) als ernstzunehmende Compiler-Sprache und verzichtet trotzdem auf ein Zeigerkonzept. Wie kommt das?

Im Zeitalter der verteilten Anwendungen, der komponentenorientierten Programmierung, der Viren und des allgemeinen Sicherheitsbedürfnisses einer zunehmend vom Computer abhängigen Informationsgesellschaft setzt sich die Auffassung immer weiter durch, dass Zeiger und Adressen die Programmierung nicht nur unnötig komplizieren, sondern auch wirksame Schutzmechanismen gegen unliebsame Übergriffe von Code in fremde Gefilde erfordern – seien diese Übergriffe nun beabsichtigt oder nicht. Die virtuelle Adressverwaltung und das verdrängende (»präemptive«) Multitasking sind zwar echte Errungenschaften in punkto Ausführungssicherheit, der zusätzliche Overhead muss aber teuer mit Ausführungszeit bezahlt werden, zumal, wenn es um den Datenaustausch zwischen Programmen und Komponenten geht, dem Lebenselixier verteilter Anwendungen.

5.7.1 Mit Automatikgetriebe

Eines der großen Hindernisse ist also der Austausch von Daten über Prozessgrenzen hinweg. Andererseits bietet nur die Prozessgrenze wirklich Sicherheit, so lange die Adressierung unter der Regie des Programmierers geschieht und dieser so schalten und walten kann, wie er will.

Um beides zu haben, keine Prozessgrenzen *und* Sicherheit, bleibt nur die Möglichkeit, dem Programmierer das Mittel der Adressierung zu entziehen und alle damit verbundenen Aufgaben einer automatischen Verwaltung zu übertragen (darunter: Prüfung der Einhaltung von Arraygrenzen, Freispeicherverwaltung, Typsicherheit etc.). Eine funktionierende Automatik dieser Art vorausgesetzt, kann die Prozessgrenze fallen: Objekte lassen sich, auch wenn sie aus verschiedenen Anwendungen stammen, dann in einer gemeinsamen Umgebung (das heißt: im gleichen Prozess) behandeln. Als Folge wird der Datenaustausch zwischen den Anwendungen enorm beschleunigt.

Dass dies funktionieren kann, hat Sun mit seinem Sandbox-Konzept in Form der virtuellen Java-Maschine vorgemacht. Umgekehrt war COM+ ein Schritt in diese Richtung, ohne die Einschränkung einer Sandbox, aber mit umso strengeren Regeln bei der Implementierung.

Mit .NET holt Microsoft nun die gesamte Welt in die Sandbox oder bildet, wenn man so will, die gesamte Welt in der Sandbox nach – und gewinnt damit den Kampf gegen den Zeiger auf breitester Front. Das .NET-Laufzeitsystem bietet ein Code-Management für Objekte, das die komponentenorientierte Programmierung nicht nur einfacher und erheblich sicherer macht, sondern auch ein buntes Miteinander verschiedensprachlicher Module auf binärkompatibler Basis ermöglicht, das selbst vor der Implementierungsvererbung nicht Halt macht.

6 Anweisungen und Ausführungskontrolle

Ähnlich wie sich ein Text aus einer Reihe von Sätzen zusammensetzt, setzen sich Programme aus Anweisungen zusammen. Der Begriff der Anweisung in einer Programmiersprache erinnert also entfernt an den Satzbegriff natürlicher Sprachen. Vergleiche dieser Art können zwar zum prinzipiellen Verständnis maschineller Sprachen etwas betragen, sie allzu sehr strapazieren zu wollen, führt aber nicht sehr weit.

6.1 Sprachebenen

Anders als bei natürlichen Sprachen spielen bei Computersprachen Rekursion, Iteration und verschiedene Sprachebenen eine bedeutende Rolle. Eine wichtige Eigenschaft von Computersprachen ist beispielsweise die Möglichkeit, Anweisungsfolgen in Prozeduren zusammenzufassen, also komplexe Anweisungen aus einfachen Anweisungen aufzubauen. Umgekehrt zerfällt jede Anweisung in eine mehr oder weniger komplexe Folge elementarer Anweisungen, wobei die Definition dessen, was elementar ist, eine reine Frage der Sprachebene ist. Als unterste Sprachebene gilt die plattformspezifische Maschinensprache, deren Befehlsfolgen der jeweilige Prozessor unmittelbar verarbeiten kann.

Einer der wesentlichen Gedanken des hinter der .NET-Technologie steckenden Sandbox-Konzepts ist die Einführung einer gemeinsamen Sprachebene, auf der für alle .NET-Sprachen Codekompatibilität herrscht. Die Hochsprache C# ist deshalb – wie alle anderen .NET-Sprachen auch – auf Basis einer Zwischensprache namens IL (Intermediate Language) definiert. Diese wiederum hat ein maschinensprachliches Äquivalent auf der jeweiligen Hardwareplattform. (.NET ist zur Zeit zwar ausschließlich auf der Plattform Intel in Verbindung mit Windows verfügbar, Borland hat aber Anfang Mai 2002 bereits etwas von einer Linux-Version verlauten lassen). Mit anderen Worten: Der C#-Compiler übersetzt C#-Code in IL-Code, und ein JIT-Compiler (Just-in-Time-Compiler) den IL-Code kurz vor der Ausführung in den so genannten *nativen Code*, also in eine Folge von plattformspezifischen Maschinenbefehlen, die der jeweilige Prozessor im Allgemeinen direkt verarbeiten kann (je nach Architektur kann hier sogar noch eine weitere Sprachebene ins Spiel kommen, man denke etwa an den Hardware Abstraction Layer (HAL) von Windows NT oder an einen Windows-Emulator für den Macintosh).

6.2 Anweisungsfolgen und Threads

Eine einzelne Anweisung macht allein noch kein Programm aus, auf die Abfolge vieler Anweisungen kommt es an. Natürlich kann man sich ein Programm als schlichten Aufruf der »Anweisung« `Main()` vorstellen, die ja, wie an anderer Stelle ausgeführt, den definierten Einsprungpunkt jedes C#-Programms darstellt. Diese Betrachtung gibt allerdings wenig Aufschluss über das Innenleben des Programms. Komplexe Anweisungen bzw. Aufrufe wie `Main()` zerfallen über den Vorgang der Substitution in Abfolgen einfacherer Anweisungen, diese wiederum in Abfolgen noch einfacherer Anweisungen, usw. Auf der untersten Substitutionsebene finden sich schließlich nur noch primitive Anweisungen der jeweiligen Programmiersprache (die ihrerseits im Rahmen der Sprachimplementierung noch weiter zerfallen, zunächst in die IL und am Ende – wie gesagt – in Maschinenanweisungen).

Allem Multiprocessing, -tasking und -threading moderner Computerarchitekturen und Betriebssysteme zum Trotz orientiert sich das Ablaufmodell heutiger Computer nach wie vor an der von Neumann-Architektur, bei der eine CPU eine Befehlskette sequenziell abarbeitet. Wenn mehrere CPUs vorhanden sind, gibt es eben mehrere Befehlsketten – und komplexe Logiken sowie Verwaltungsmechanismen, die diese miteinander in Einklang bringen. Das Ablaufmodell wird dadurch zwar etwas komplexer, wirklich grundlegende Änderungen, die aus der Beschränktheit der von Neumann-Architektur hinausführen würden, gibt es jedoch nicht.

Rein formal definieren sich in einer Programmiersprache geschriebene Programme als geordnete Mengen primitiver Anweisungen, deren Abarbeitung in einem oder mehreren Ausführungspfaden, so genannten THREADS, erfolgt. Wie der Begriff »Pfad« bereits nahe legt, besteht ein Thread aus einer endlichen, strikt geordneten Folge von Anweisungen, die einen definierten Anfang sowie ein definiertes Ende hat (von Endlosschleifen einmal abgesehen, die es faktisch auch gar nicht gibt, weil jeder Computer einmal ausgeschaltet wird – spätestens, wenn man ihn ausrangiert). Diese Art der Abarbeitung von Anweisungen wird auch als *synchrone* Ausführung bezeichnet. Mit anderen Worten: Alle Anweisungen in einem Thread werden synchron und in eindeutiger Reihenfolge ausgeführt. Im Gegensatz dazu spricht man von *asynchroner* Ausführung wenn zwei Anweisungen (ohne besondere Synchronisationsmaßnahmen) in unterschiedlichen Threads ausgeführt werden

Es wäre verfrüht, Threads an dieser Stelle bereits weiter zu vertiefen. Vielmehr soll der Begriff des Ausführungspfads die hinter prozeduralen Programmiersprachen steckende Philosophie der *Anweisungsfolge* und deren deterministische Abarbeitung deutlich machen.

6.2.1 Strukturierung von Anweisungsfolgen

Als prozedurale Programmiersprache stellt C# verschiedene syntaktische Mittel zur Gestaltung und Organisation von Anweisungsfolgen bereit und orientiert sich dabei, wie zu erwarten, weitgehend an der Syntax von C/C++.

Zunächst einmal fordert C#, dass eine Anweisung grundsätzlich durch ein Semikolon abgeschlossen wird. Eine Anweisungsfolge bestehend aus den Anweisungen *Anweisung1* bis *AnweisungN* schreibt sich demnach so:

```
Anweisung1;
Anweisung2;
...
AnweisungN;
```

Kommaoperator

Der von C/C++ her bekannte Kommaoperator zur sequenziellen Auswertung von Anweisungen wird von C# nur in bestimmtem Kontext unterstützt, nämlich:

- bei der Variablendefinition

```
int a, b = 10, c;
```

- im Zusammenhang mit `for`-Kontrollstrukuren

```
for(i=0, j=9; i<j; i+=2, j++)
    Anweisungsblock
```

Die Auswertungsreihenfolge des Kommaoperators geht von links nach rechts. Die Schleife

```
for (int x = 0, y = x; x <= 1; y = x, ++x)
  Console.WriteLine (y);
```

liefert daher auch im zweiten Durchlauf die Ausgabe "0", während

```
for (int x = 0, y = x; x <= 1; x++, y = x)
  Console.WriteLine (y);
```

trotz Postfixnotation des ++-Operators beim zweiten Durchlauf die Ausgabe "1" erzeugt.

Anweisungsblöcke

Anweisungsfolgen lassen sich durch geschweifte Klammern zu Anweisungsblöcken zusammenfassen. Derartige Blöcke *müssen* verwendet werden bei

- Namensraumzuordnungen

```
namespace MyNamespace
{
  Klassendefinitionen
}
```

- Klassendefinitionen

```
class MyClass
{
  Datentypdefinitionen
  Datenfelddefinitionen
  Methodendefinitionen
}
```

- Methodendefinitionen

```
Modifizierer Rückgabetyp MyMethod([Parameterliste])
{
  Anweisungsfolge
}
```

- Kontrollstrukturen wie `switch`, `try`, `checked`, `fixed`, `unsafe` usw.

```
checked  // Überlaufkontrolle aktivieren
{
  Anweisungsfolge
}
```

- Kontrollstrukturen, wie `for`, `while`, `if`, wenn die Anweisungsfolge mehr als eine Anweisung enthält

```
if (BoolescherAusdruck)
{
  AnweisungsfolgeWennTrue
}
else
{
  AnweisungsfolgeWennFalse
}
```

Zu guter Letzt *können* Anweisungsblöcke auch ohne Kontrollausdruck zur Gestaltung der Sichtbarkeit von Variablen innerhalb von Methodendefinitionen notiert werden:

```
...
{
  int i = 10;
  string s;
...
}    // Ab hier sind i und s nicht mehr bekannt
...
{
  long i = 20;     // ... und können neu vereinbart werden
  string s;
```

```
...
}    // Ab hier sind i und s nicht mehr bekannt
...
```

Variablen, die in einem Anweisungsblock vereinbart werden, sind nur in diesem Block sichtbar. Auf diese Weise ist es also möglich, ein und denselben Variablenbezeichner in einer Methodendefinition mehrfach zu vereinbaren, sogar für völlig verschiedene Datentypen. Allerdings sollte man sich in diesem Zusammenhang nicht zu der Annahme hinreißen lassen, dass der C#-Compiler solchen untergeordneten Anweisungsblöcken einen eigenen Stack-Rahmen spendiert – was beispielsweise vorteilhaft wäre, um den Stack-Verbrauch bei rekursiven Aufrufen zu minimieren oder Ressourcen potenziell früher freizugeben. (Um dies zu erreichen, muss der entsprechende Code in eigenständigen Methoden zusammenfasst werden, denn nur Methoden haben einen eigenen Stack-Rahmen.)

Man mag sich nun fragen, inwieweit es überhaupt Vorteile bringt, untergeordnete Anweisungsblöcke zu verwenden, wenn kein Kontrollausdruck dies erfordert. Außer der zusätzlichen Freiheit bei der Bezeichnerwahl, einer zusätzlichen Strukturierung des Codes und einer Begrenzung der Sichtbarkeit von Bezeichnern ergeben sich tatsächlich keine Vorteile.

6.3 Kommentare

Kommentare sind keine Anweisungen im eigentlichen Sinne. Sie gehen nicht in den Programmcode ein, sondern dienen rein der Dokumentation des Quelltexts. Der C#-Compiler unterscheidet drei Arten von Kommentaren:

- mit `//` eingeleitete *Zeilenkommentare*
- mit `/*...*/` geklammerte *Blockkommentare*
- mit `///` eingeleitete *Dokumentationskommentare*

Die ersten beiden Kommentararten sind für C/C++- und Delphi-Programmierer gute Bekannte. Dokumentationskommentare stellen hingegen ein Novum dar. Sie erlauben es, XML-Tags in Quelltexten unterzubringen, um den Code vor Ort zu dokumentieren. Der enorme Vorteil dabei ist, dass VS.NET Dokumentationskommentare tatsächlich umgehend für die während der Quelltexteingabe angezeigten Prototyp- und Parameterbeschreibungen auswertet, womit auch für eigene Datentypen, deren Elemente und Parameter Kurzbeschreibungen verfügbar werden.

6.3.1 Zeilenkommentare

Der Zeilenkommentar ist die häufigste Kommentarart, die in C#-Programmen zu finden ist – wobei die Häufigkeit natürlich vom jeweiligen Programmierstil abhängt. In gut dokumentierten Programmen wird man fast in jeder Zeile oder zumindest in allen wichtigen Zeilen einen Zeilenkommentar finden. Ein Zeilenkommentar kann an jeder beliebigen Stelle einer Zeile beginnen und erstreckt sich dann grundsätzlich bis zum Ende dieser Zeile. Kommentare, die nicht bis zum Zeilenende gehen, lassen sich nur als Blockkommentare formulieren.

```
Index = -1;  // ab hier wird alles bis zum Zeilenende ignoriert
if (Index == NOINDEX /* -1 */) // zulässig
if (Index == NOINDEX //-1)   // Fehler: schließende Klammer als Kommentar
```

Aufgabenliste

In Form der *Aufgabenliste* bietet Visual Studio .NET ein sehr praktisches Werkzeug an, das speziell darauf zugeschnitten ist, die noch ausstehenden Arbeiten nach verschiedenen Kriterien zusammenzufassen. Ein nur allzu bekannter Teil dieser Aufgabenliste ist die Aufstellung der Compilerfehler und -warnungen. Ein weniger bekannter Teil sind die TOKENKOMMENTARE, die der Programmierer nach eigenem Gutdünken unter Angabe eines Tokens unmittelbar nach dem Kommentarsymbol in Kommentarform in den Quelltext einfügen kann. Tokenkommentare erscheinen (wie Compilerfehler) in der Aufgabenliste als Hyperlinks und erlauben den direkten Sprung in den Quelltext per Doppelklick. Abbildung 6.1 zeigt, wie die Anzeige von Tokenkommentaren eingeschaltet wird, Abbildung 6.3, wie die Aufgabenliste aussieht.

Durch die Verwendung von `TODO`-, `HACK`- oder `UNDONE`-Kommentaren erhält der Programmierer einen nach Aufgabenarten oder anderen Kriterien sortierbaren Überblick über die noch ausstehenden Arbeiten und »losen Enden«. Falls diese Kategorien nicht ausreichen, lassen sich auch eigene Token über den Dialog EXTRAS/OPTIONEN/AUFGABENLISTE frei definieren und mit der gewünschten Priorität versehen. (vgl. Abbildung 6.2).



Tokenkommentare lassen sich auch auf der Basis von Blockkommentaren formulieren.

6.3.2 Blockkommentare

Der Blockkommentar geht auf die Sprache C zurück. Er beginnt mit der Zeichenfolge `/*` und endet mit der (umgekehrten) Zeichenfolge `*/`. Ein Blockkommentar kann nicht um einen anderen Blockkommentar herum gesetzt werden – Verschachtelung ist also nicht erlaubt –, um einen Zeilenkommentar herum hingegen schon.

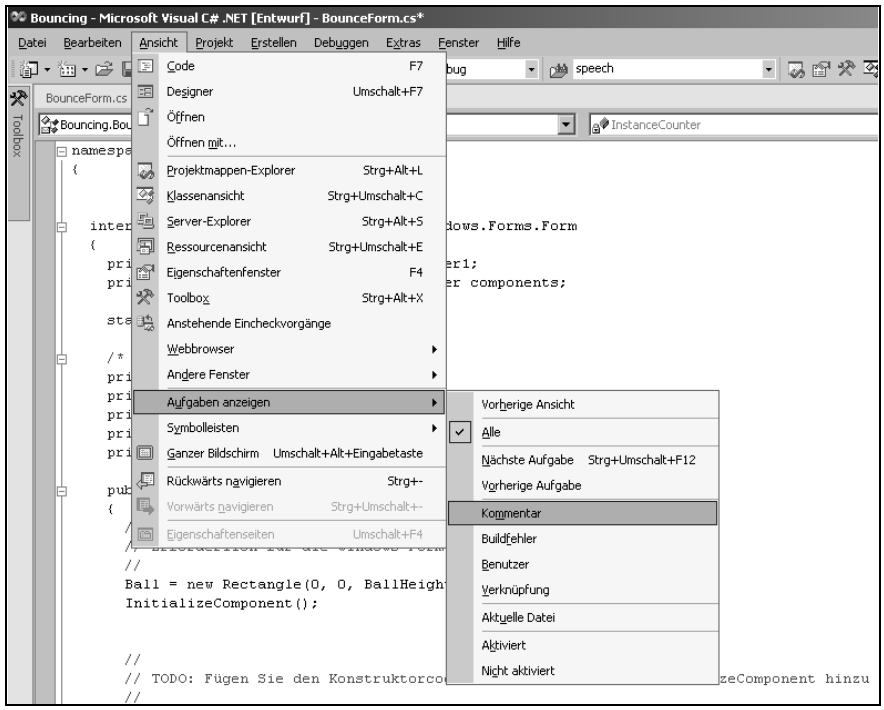


Abbildung 6.1: Anzeige der Tokenkommentare in der Aufgabenliste aktivieren

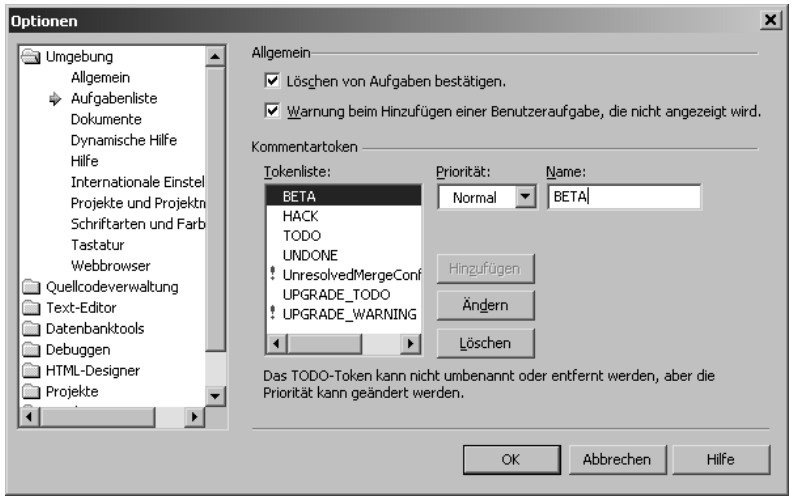


Abbildung 6.2: Definition eines eigenen Kommentartokens über EXTRAS/OPTIONEN

Das folgende Codebeispiel zeigt den Gebrauch den Blockkommentars:

```
// Es folgt ein typischer Blockkommentar, der einen Codeteil
// samt Zeilenkommentaren ausklammert ...
/*
    Ball.X+=BallSpeedX;
    Ball.Y+=BallSpeedY;
```

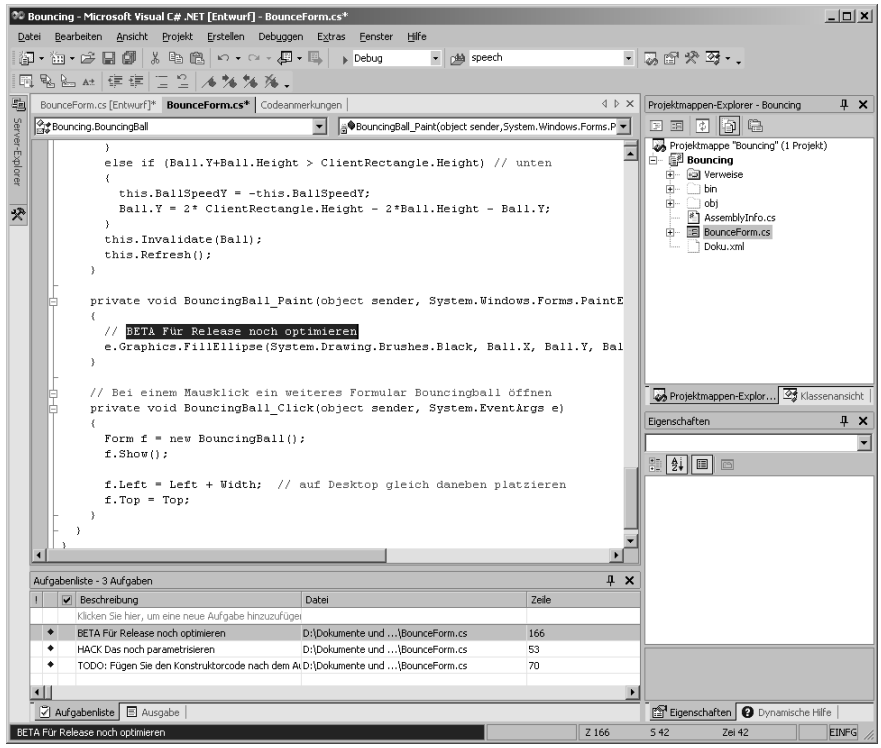


Abbildung 6.3: In der Aufgabenliste (unten links) werden Kommentartoken samt Text als Hyperlink angezeigt.

```
// Korrektur bei Bandenkollision
if (Ball.X < 0) // links
{
  this.BallSpeedX = -this.BallSpeedX;
  Ball.X = - Ball.X;
}
else if (Ball.X + Ball.Width  > ClientRectangle.Width) // rechts
{
  this.BallSpeedX = -this.BallSpeedX;
  Ball.X = 2* ClientRectangle.Width - 2*Ball.Width - Ball.X;
}  */    // Hier ist der Blockkommentar zuende!
```

In C# wird diese Kommentarart eigentlich kaum noch für die Quelltextdokumentation verwendet – dafür nimmt man nun bevorzugt den einfacheren Zeilenkommentar –, sondern hauptsächlich zum nachträglichen Ausklammern zusätzlicher Prüfungen und dem zeitweiligen »Auskommentieren« bestimmter Codeteile für Testläufe. Obwohl C# auch eine (gegenüber dem C/C++-Präprozessor stark eingeschränkte) Syntax für die bedingte Kompilierung unterstützt (`#define`, `#if`, `#else`, `#endif`), ist die Fehlersuche mit Blockkommentaren nach wie vor ein probates Mittel.

Das folgende Codebeispiel demonstriert einen einfachen, aber nichtsdestotrotz häufigen Fehler bei der gemischten Verwendung beider Kommentararten:

```
/* Ball.X += BallSpeedX;
   Ball.Y += BallSpeedY; // Position aktualisieren */
```

Tatsächlich ignoriert der Compiler nach der Zeichenfolge // *alles* bis zum Ende der jeweiligen Zeile – also auch den Abschluss des Blockkommentars. Da der Editor Kommentaren eine andere Farbe gibt als dem restlichen Quelltext, fallen Fehler dieser Art aber recht schnell ins Auge.

6.3.3 Dokumentationskommentare

Während C# mit den Block- und Zeilenkommentaren seiner Abstammung von C/C++ Rechnung trägt, geht die Sprache mit den Dokumentationskommentaren neue Wege. Hinter der kommentarbasierten Dokumentation steht die Idee, die Software-Dokumentation auf XML-Basis zu stellen und »vor Ort«, also in unmittelbarer Umgebung des Quellcodes, in die Quelltextdatei zu integrieren, um sie während der Programmierung als Referenzinformation anzuzeigen. Der Compiler kann dann zusätzlich die Aufgabe übernehmen, eine separate Dokumentationsdatei zusammenzustellen, die sich als (immer aktuelle) Eingabedatei für die weitergehende Dokumentation oder als Informationsquelle für externe Werkzeuge, Hilfesysteme usw. nutzen lässt.

Vorteile

Die physische »Nähe« der Dokumentationskommentare zum Quellcode bietet natürlich den Vorteil, dass Änderungen des Codes bzw. der Spezifikation von Methoden oder Klassen ihren Weg in die Dokumentation wesentlich direkter finden können, als das bei getrennter Pflege von Quellcode und Dokumentation der Fall wäre. Die Dokumentation auf der Basis von Dokumentationskommentaren dürfte also von der Tendenz her zu einer besseren Synchronisation und Durchdringung von Dokumentation und Quellcode führen. Das offensichtlichste Beispiel dafür sind die vom VS.NET-Texteditor eingeblendeten ToolTip-Fensterchen, die zu jedem Element eine Syntaxhilfe mit Schnellreferenz präsentieren und dem Programmierer damit erheblich unter die Arme greifen (vgl. die Abbildungen 6.9 und 6.10, Seite 136).

Nachteile

Umgekehrt bürdet sie dem Programmierer zusätzliche Arbeit auf, die ihn von der eigentlichen Programmierarbeit ablenkt. Zudem sind Dokumentationskommentare keine Garantie für eine korrekte Dokumentation, da große Teile der Dokumentationsstruktur schlicht vom Programmierer selbst angelegt und natürlich auch gepflegt werden müssen, was im Zuge der Fehlersuche gerne unter den Tisch fällt oder schlicht in Vergessenheit gerät.

Handhabung

Damit der Compiler Dokumentationskommentare extrahiert, deren Syntax prüft (wobei Fehler gegebenenfalls in Form von Warnungen moniert werden) und zu einer XML-Datei kompiliert, reicht es, in Visual Studio .NET die Eigenschaft XML-DOKUMENTATIONSDATEI auf der Eigenschaftsseite des jeweiligen Projekts auf den gewünschten Ausgabedateinamen zu setzen (vgl. Abbildung 6.4). Bei einem Kommandozeilenaufruf verwenden Sie dazu den Compilerschalter `/doc`, gefolgt vom Namen der Ausgabedatei, also zum Beispiel `/doc:XMLAusgabe.xml`.

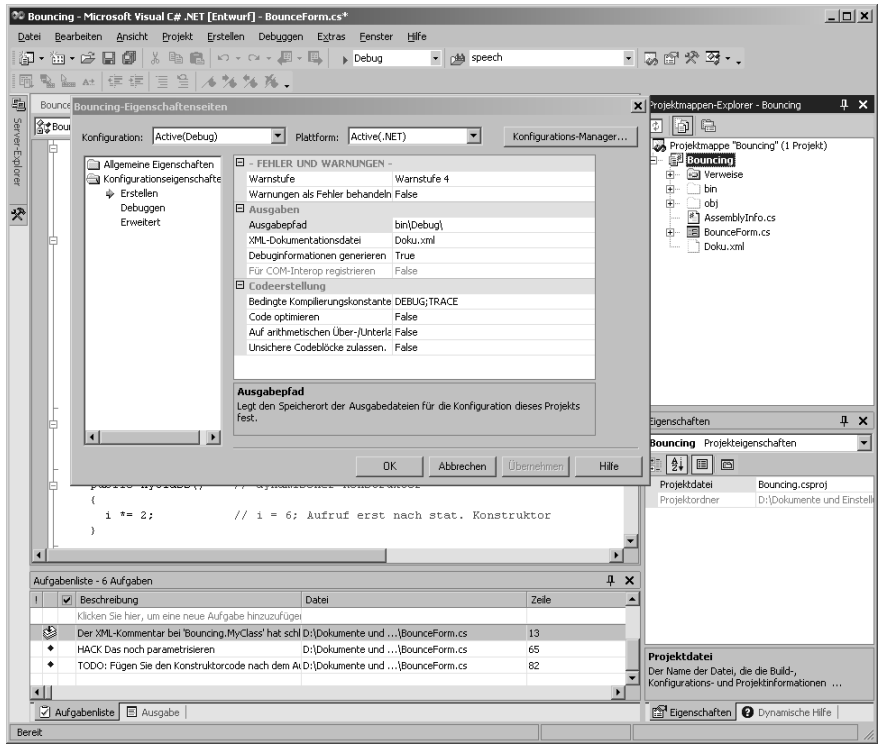


Abbildung 6.4: Der Compiler gibt eine XML-Datei aus, wenn Sie der Eigenschaft XML-DOKUMENTATIONSDATEI einen Dateinamen zuweisen.

Der Editor von Visual Studio .NET erkennt Dokumentationskommentare als solche und erlaubt es, sie im Rahmen der üblichen Ansichtsstruktur blockweise ein- und auszublenden. Bei der Eingabe von Dokumentationskommentaren blendet der Editor, auf den jeweiligen Kontext zugeschnitten, eine Liste mit den jeweils zulässigen XML-Tags ein (vgl. Abbildung 6.5).

Kommentarwebseiten erstellen

Wie Abbildung 6.6 zeigt, gibt die Anzeige einer nackten XML-Datei im Internet Explorer noch nicht viel her. Für ein visuell ansprechenderes und funktionelleres Layout müssen Sie ein Stylesheet-Dokument in Form einer *.css*-Datei unterlegen. Auch wenn Sie sich die Mühe machen können, diese

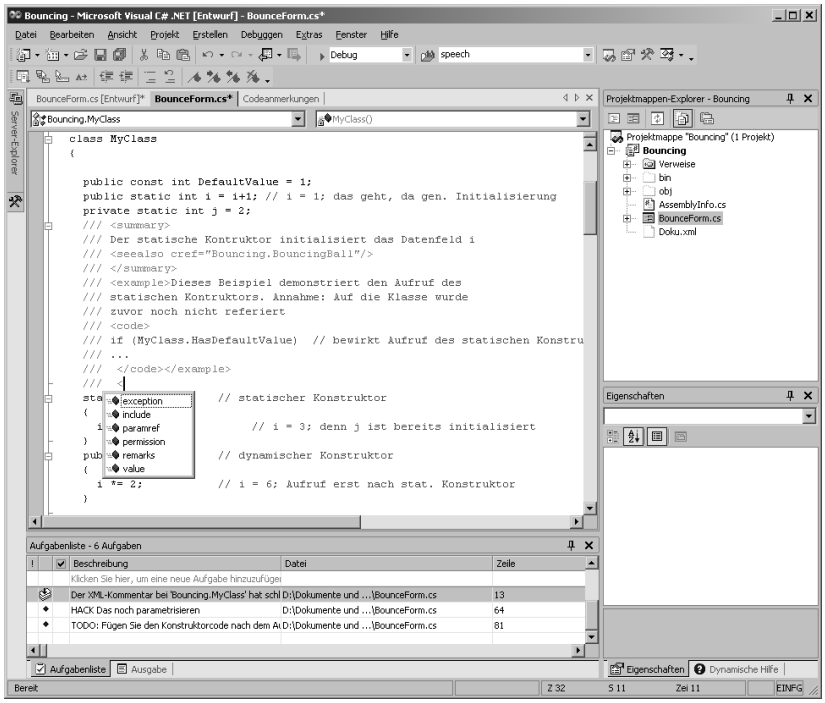


Abbildung 6.5: Visual Studio .NET bietet auch für Dokumentationskommentare eine kontextsensitive Überwachung der XML-Syntax.

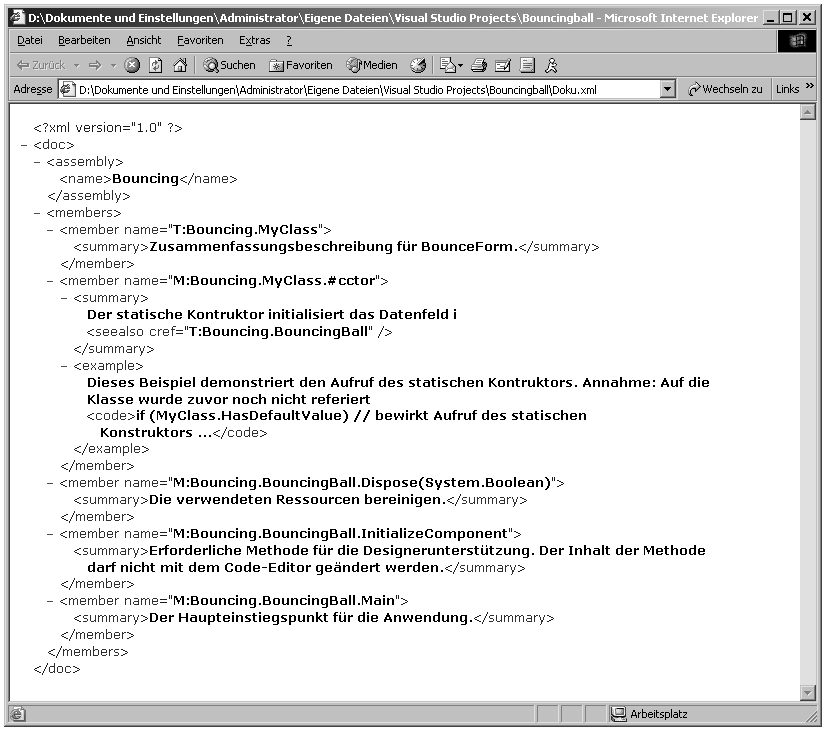


Abbildung 6.6: Anzeige einer XML-Datei mit Dokumentationskommentaren im Internet Explorer

selbst zu entwerfen, führt der schnellste Weg zu einer visuell ansprechenden Dokumentation auf Basis der Dokumentationskommentare über den Menübefehl EXTRAS/ERSTELLEN VON KOMMENTARWEBSEITEN. In dem erscheinenden gleichnamigen Dialogfeld stellen Sie ein, für welche Projekte der aktuellen Projektmappe Sie Kommentarwebseiten erstellen wollen und geben einen Pfad vor (Abbildung 6.7).

Abbildung 6.7: Der Dialog ERSTELLEN VON KOMMENTARWEBSEITEN



Einen Mausklick später erhalten Sie bereits das fertige Ergebnis in Form zahlreicher automatisch benannter *.htm*-Dokumente und einem verbindenden Stylesheet namens *commentreport.css*. Abbildung 6.8 zeigt die durchaus ansprechende Ansicht einer auf diese Weise erstellten Kommentarwebseite, die dem Projekt *Benutzersteuerelemente* aus der Codebasis für Teil 3 dieses Buchs entstammt.

XML-Tags für Dokumentationskommentare

Die folgenden Unterabschnitte stellen die verschiedenen XML-Tags mit den dafür vorgesehenen untergeordneten Tags vor, die für C#-Dokumentationskommentare vordefiniert sind. Weitergehende Beispiele finden sich in der Hilfe zu Visual Studio .NET.

Summary

Das <*summary*>-Tag wird dazu verwendet, um eine Klasse oder ein Klassenelement (Datenfeld, Methode, Datentyp) zu beschreiben. Der Texteditor von VS.NET bindet <*summary*>-Informationen umgehend in sein ToolTip-System ein, das den Programmierer während des Tippens kontextsensitiv mit Informationen (Prototyp- und Parameterbeschreibungen) zu den einzelnen Elementen versorgt.

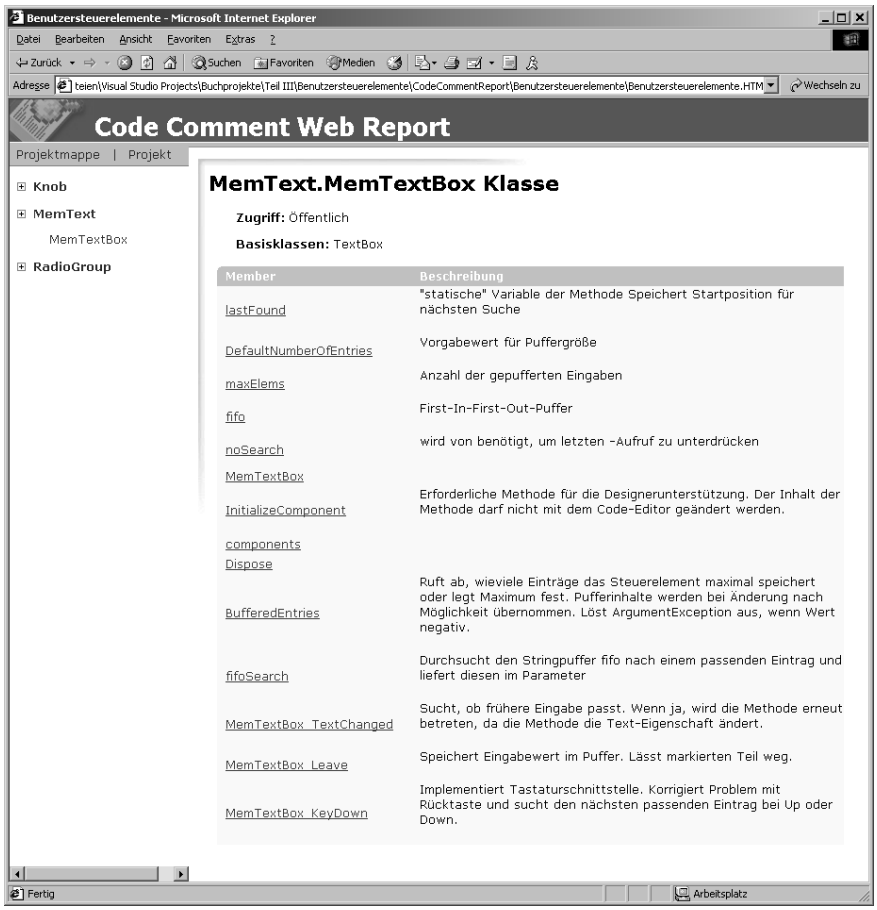


Abbildung 6.8: Kommentarwebseite aus dem Projekt *Benutzersteuerelemente*

```
/// <summary>
///   <c> ... </c>
///   <code> ... </code>
///   <list ...> ... </list>
///   <see ... />
///   <seealso ... />
/// </summary>
```

Wann immer Sie die Zeichenfolge `///` in einer eigenen Zeile vor einer Klassendefinition oder einer Klassenelementdefinition eingeben, ergänzt der Editor ein darauf zugeschnittenes Dokumentationskommentargerüst. Bei Eingabe der drei Zeichen vor einer Methode mit Parametern ergänzt der Editor beispielsweise:

```
/// <summary>
///
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
```

```
private void timer1_Tick(object sender, System.EventArgs e)
{
...
```

Diese Struktur können Sie dann mit dem notwendigen Text ausschmücken:

```
/// <summary>
/// Behandlungsroutine für das von von dem Steuerelement timer1 generierte
/// Tick-Ereignis. Schaltet zum nächsten Bild weiter.
/// </summary>
/// <param name="sender">
/// Absender des Ereignisses</param>
/// <param name="e">
/// Objekt, welches das Ereignis näher beschreibt
/// </param>
private void timer1_Tick(object sender, System.EventArgs e)
{ ... }
```

VS.NET wertet diese Information auf zweierlei Art aus: Erstens greift der Editor *sofort* darauf zurück und baut sie in die ToolTip-Hilfe für das betreffende Element ein. Die Abbildungen 6.9 und 6.10 zeigen das Resultat. Zweitens stellt der Compiler alle Dokumentationskommentare eines Projekts optional als XML-Datei zusammen. Den XML-Code aus dem Codebeispiel stellt der Internet Explorer wie in Bild 6.11 dar.

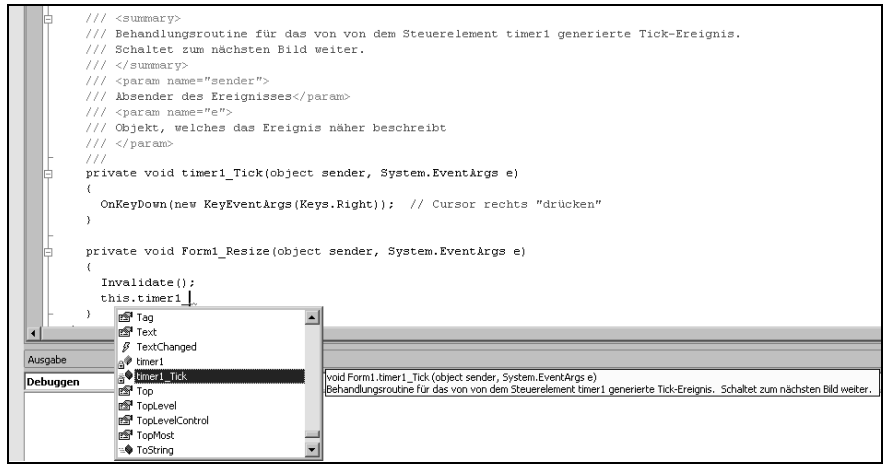


Abbildung 6.9: Der Editor greift *summary*-Informationen sofort auf, um sie als ToolTip anzuzeigen.

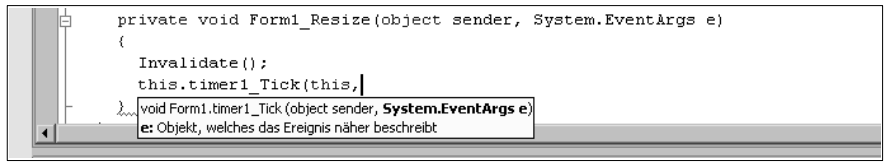


Abbildung 6.10: Der Editor zeigt *param*-Informationen da an, wo sie gebraucht werden.

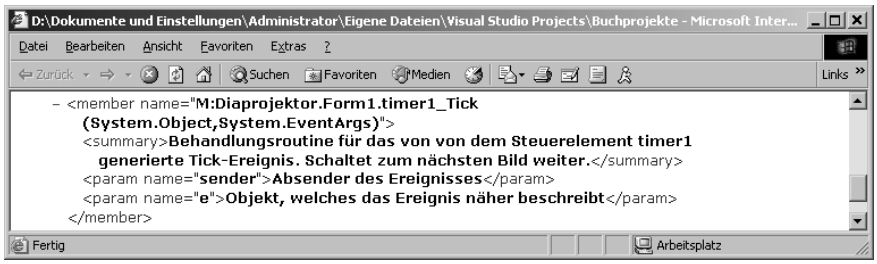


Abbildung 6.11: Beispiel einer XML-Datei, die C#-Compiler aus Dokumentationskommentaren erzeugt hat.

Example

Das <*example*>-Tag wird dazu verwendet, den im untergeordneten <*code*>-Tag aufgeführten Beispielcode für die Verwendung einer Klasse oder eines Klassenelements näher zu beschreiben.

```
/// <example>
///   <c> ... </c>
///   <code> ... </code>
///   <list ...> ... </list>
///   <see ... />
///   <seealso ... />
/// </example>
```

Exception

Das <*exception*>-Tag wird dazu verwendet, um eine Ausnahme zu beschreiben, die eine Klasse auslösen kann. Der Text sollte vor allem erläutern, unter welchen Umständen die jeweilige Ausnahme zu Stande kommt:

```
/// <exception cref = "AusnahmeKlasse">
///   <c> ... </c>
///   <code> ... </code>
///   <list ...> ... </list>
///   <see ... />
///   <seealso ... />
/// </exception>
```

Remarks

Das <*remarks*>-Tag wird dazu verwendet, zusätzliche Bemerkungen oder Besonderheiten einer Klasse oder eines Klassenelements zu formulieren:

```
/// <remarks>
///   <c> ... </c>
///   <code> ... </code>
///   <list ...> ... </list>
///   <see ... />
///   <seealso ... />
/// </remarks>
```

Param

Das <*param*>-Tag wird dazu verwendet, um einen Parameter einer Methode näher zu beschreiben. Der Editor von Visual Studio .NET bietet dieses Tag kontextbezogen für jeden Parameter an, den die im nachfolgenden Code definierte Methode erwartet. Bei Einfügen eines neuen Dokumentationskommentars vor eine Methode ergänzt er automatisch für jeden Parameter eine <*param*>-Struktur. (Ein Beispiel für dieses Tag findet sich im Abschnitt »Summary«.)

```
/// <param name="ParameterBezeichner">
///   <c> ... </c>
///   <code> ... </code>
///   <list ...> ... </list>
///   <see ... />
///   <seealso ... />
/// </remarks>
```

Paramref

Das <*paramref*>-Tag wird dazu verwendet, um auf den Bezeichner eines (nicht zwingend) an anderer Stelle per <*param*>-Tag beschriebenen Parameters zu referieren. Die Referenz wird als solche markiert und erscheint in einer eigenen Zeile.

```
/// <paramref name="ParameterBezeichner"/>
```

Value

Das <*value*>-Tag wird dazu verwendet, den für einen Parameter gültigen Wertebereich näher zu beschreiben.

```
/// <value>
///   <c> ... </c>
///   <code> ... </code>
///   <list ...> ... </list>
///   <see ... />
///   <seealso ... />
/// </value>
```

Include

Das <*include*>-Tag ermöglicht einen Verweis auf eine andere XML-Datei mit Dokumentationskommentaren, welche eingebunden werden soll:

```
/// <include file='AndereXMLDatei' path='TagPfad[@name="id"]'/>
```

C

Das <*c*>-Tag zeichnet den eingeschlossenen Text als Codeelement aus. (Ein Beispiel für dieses Tag findet sich im Abschnitt »Summary«.) Untergeord-

nete <*code*>-Tags sind dafür gedacht, mehrzeiligen Code in Codebeispielen zu notieren.

Listing 6.1: see, seealso

```
/// <c>
///   <code> ... </code>
///   <list ...> ... </list>
///   <see ... />
///   <seealso ... />
/// </c>
/// <code>
///   <c> ... </c>
///   <see ... />
///   <seealso ... />
///   <list ...> ... </list>
/// </code>
```

Die Tags <*see*> und <*seealso*> sind dafür gedacht, Querverweise in die Dokumentation einzufügen. Das <*see*>-Tag notiert einen Verweis auf ein Klassenelement (Member) der aktuellen Umgebung, das an anderer Stelle näher beschrieben ist. Der Compiler prüft in diesem Fall selbstständig, ob das referierte Element vorhanden ist.

Listing 6.2: list, listheader, item, term

```
/// <see cref = "VerweisAufMember"/>
/// <seealso cref = "VerweisAufMember"/>
```

Die Tags <*list*>, <*listheader*>, <*item*> und <*term*> ermöglichen die Definition einer Auflistung oder Tabelle.

Das <*list*>-Tag begrenzt die Auflistungsstruktur und definiert die Art der Auflistung. Möglich sind Aufzählungspunkte, nummerierte Aufzählungen und Tabellen.

```
/// <list type = "bullet" | "number" | "table">
///   <c> ... </c>
///   <code> ... </code>
///   <description> ... </description >
///   <item> ... </item>
///   <listheader> ... </listheader>
///   <para> ... </para>
///   <see ... />
///   <seealso ... />
///   <term> ... </term>
/// </list>
```

Das <*listheader*>-Tag zeichnet eine Tabellenüberschrift aus:

```
/// <listheader>
///   <c> ... </c>
///   <code> ... </code>
///   <description> ... </description >
///   <item> ... </item>
///   <para> ... </para>
```

```
///   <see ... />
///   <seealso ... />
///   <term> ... </term>
/// </listheader>
```

Das <*item*>-Tag zeichnet eine Tabellen- bzw. Auflistungszeile aus:

```
/// <item>
///   <c> ... </c>
///   <code> ... </code>
///   <listheader> ... </listheader>
///   <para> ... </para>
///   <see ... />
///   <seealso ... />
///   <term> ... </term>
/// </item>
```

Das <*term*>-Tag zeichnet einen Begriff aus, der im Allgemeinen in einer darauffolgenden <*description*>-Spalte erläutert wird:

```
/// <term>
///   <c> ... </c>
///   <code> ... </code>
///   <item> ... </item>
///   <see ... />
///   <seealso ... />
/// </term>
```

Das <*description*>-Tag zeichnet eine Begriffserläuterung für eine im Allgemeinen vorausgegangene <*term*>-Spalte aus:

```
/// <description>
///   <c> ... </c>
///   <code> ... </code>
///   <item> ... </item>
///   <listheader> ... </listheader>
///   <para> ... </para>
///   <see ... />
///   <seealso ... />
///   <term> ... </term>
/// </description>
```

Hier ein prototypisches Beispiel für die Auszeichnung einer zweispaltigen Tabelle mit Überschriftzeile und zwei Elementzeilen:

```
/// <list type="table">
///   <listheader>
///     <term>Beschriftung für Begriffe</term>
///     <description>Beschriftung für Erläuterung</description>
///   </listheader>
///   <item>
///     <term>erster Begriff</term>
///     <description>Erläuterung für den ersten Begriff</description>
```

```
///   </item>
///   <item>
///     <term>zweiter Begriff</term>
///     <description>Erläuterung für den zweiten Begriff</description>
///   </item>
/// </list>
```

Para

Das *<para>*-Tag ermöglicht eine Strukturierung des Textes in Absätze.

```
/// <para>
///   <c> ... </c>
///   <code> ... </code>
///   <item> ... </item>
///   <see ... />
///   <seealso ... />
///   <term> ... </term>
/// </para >
```

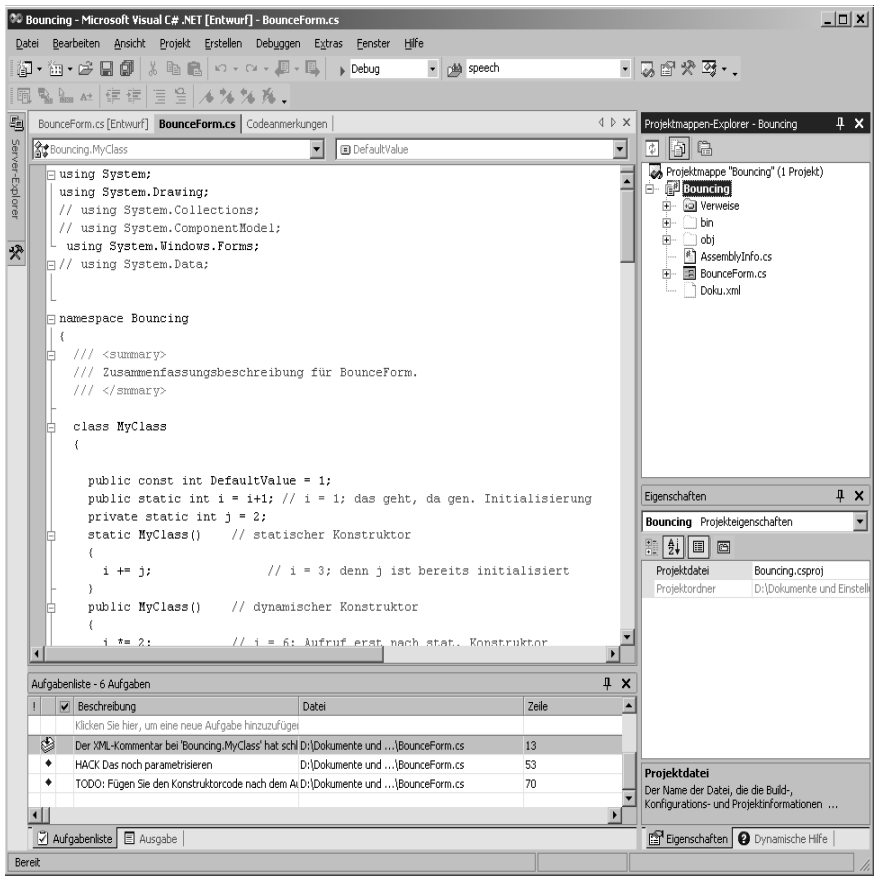


Abbildung 6.12: Der Compiler gibt Warnungen aus, wenn die Syntax von Dokumentationskommentaren nicht stimmt.

6.4 Arten von Anweisungen

Der C#-Compiler unterscheidet eine Fülle verschiedener Anweisungen.

- Vereinbarung, gegebenenfalls mit Initialisierung
- (kombinierte) Zuweisung
- Methodenaufruf
- wertverändernde Ausdrücke – `++` (Inkrement) und `--` (Dekrement)
- Sprünge – `break`, `continue`, `goto`, `return`
- Kontrollstrukturen
  - bedingte Verzweigungen – `if` und `switch`-Strukturen
  - Schleifen – for, foreach, do, while
  - Ausnahmebehandlungsstrukturen
  - Überlaufkontrolle – `checked`-/`unchecked`-Strukturen
  - `fixed`-Struktur
  - `lock`-Struktur

6.4.1 Überlaufkontrolle – checked und unchecked

Die Schlüsselwörter `checked` und `unchecked` werden oft zu den Operatoren gezählt, sind aber im eigentlichen Sinne keine. Sie wirken vielmehr auf der Ebene von Anweisungsblöcken und bestimmen, ob der Compiler Code zur Prüfung von Wertüberschreitungen (`checked`) einsetzt oder nicht (`unchecked`). Standardmäßig fügt der Compiler keinen Prüfcode ein, weshalb arithmetische Überläufe unbemerkt bleiben. Das Standardverhalten des Compilers ist aber davon abhängig, ob der Compilerschalter */checked* gesetzt ist oder nicht. Ist er nicht gesetzt, ermöglicht es der Operator `checked`, die Prüfung für einen geklammerten Codeblock gezielt einzuschalten. Und umgekehrt: Bei gesetztem Compilerschalter */checked* ermöglicht es der Operator `unchecked`, die Prüfung für einen geklammerten Codeblock explizit auszuschalten.

Abbildung 6.13 zeigt die Wirkung des `checked`-Operators auf Code, der über die Stränge schlägt.



In VS.NET wird der Compilerschalter /checked auf der Eigenschaftsseite des Projekts über die Eigenschaft KONFIGURATIONSEIGENSCHAFTEN/ERSTELLEN/AUF ARITHMETISCHEN ÜBER-/UNTERLAUF ÜBERPRÜFEN *gesetzt (vgl. Abbildung 6.14).*

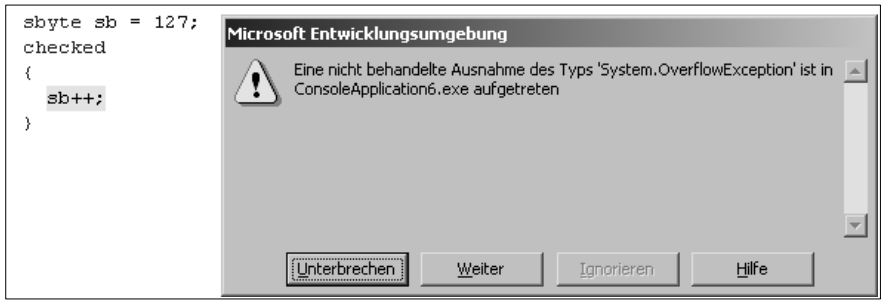


Abbildung 6.13: Wirkung des `checked`-Operators

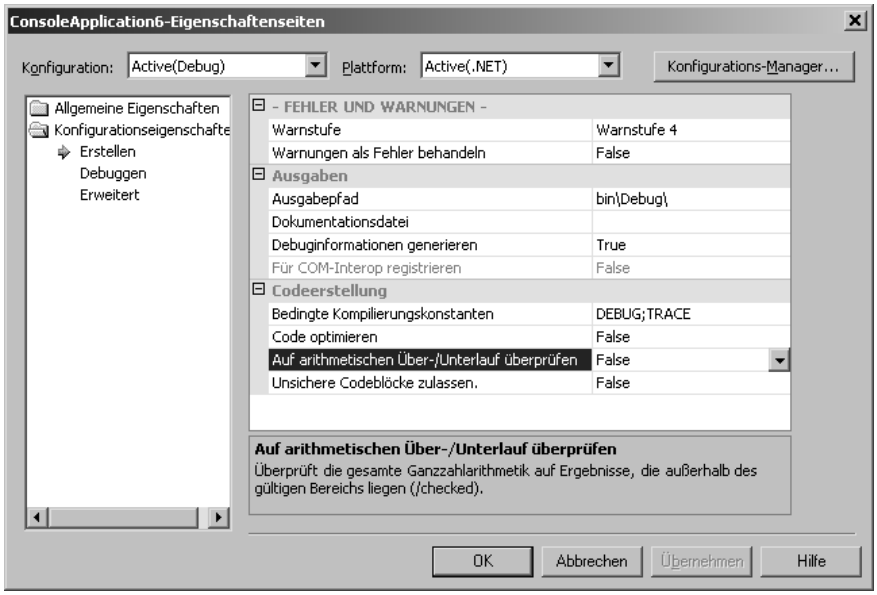


Abbildung 6.14: Compilerschalter werden über den Eigenschaftsseiten-Dialog gesetzt (gezeigt sind die Standardeinstellungen für neue Projekte).

6.5 Kontrollstrukturen

Es gibt eine Kategorie von Anweisungen, auf die keine prozedurale Programmiersprache verzichten kann: *Kontrollstrukturen*. Sie sind das Inventar, das einen großen Teil der Ausdruckskraft einer Programmiersprache ausmacht. Jede Art von Programmlogik und Ausführungskontrolle, die in irgendeiner Form mit Entscheidungen zu tun hat, lässt sich nur auf der Basis von Kontrollstrukturen realisieren.

Obwohl C# im Bereich Kontrollstrukturen seine Abstammung von C/C++ weder verleugnen kann noch will, wartet die Sprache doch mit einigen kleinen, aber recht wirkungsvollen Unterschieden auf, die den Code letztlich sicherer und weniger fehleranfällig machen.

6.5.1 Bedingte Verzweigung

Die bedingte Verzweigung gibt es als einfache `if...else`-Verzweigung und als komplexe `switch...case`-Mehrfachverzweigung.

if

Die C#-Syntax der `if`-Anweisung deckt sich weitgehend mit der von C/C++:

```
if (booleschерAusdruck)
{
  Anweisungsblock   // ausgeführt, wenn booleschерAusdruck "true" ist
}
[else
{
  Anweisungsblock   // ausgeführt, wenn booleschерAusdruck "false" ist
}]
```

Der `else`-Zweig ist optional. Allerdings ist die `if`-Anweisung von C# in der Praxis um einiges rigider als die von C/C++ oder Visual Basic – tatsächlich entspricht sie der von Pascal/Delphi: C# duldet ausschließlich echte Werte des Typs `bool` im Bedingungsausdruck. Und dies aus gutem Grund: In C/C++ ist es »legitim«, statt einem ==-Vergleich eine Zuweisung in den Bedingungsausdruck zu setzen, was sich gerade für »mehrsprachige« Programmierer immer wieder als Fallstrick erweist und schwer auffindbare Fehler hervorbringt. Während der C/C++-Compiler Formulierungen wie

```
...
if (i = 10)    // würde vom C#-Compiler als Syntaxfehler moniert
  cout << "Text wird unabhängig vom Wert von i immer ausgegeben";
else
  cout << "Text wird nie ausgegeben";
...           // i hat nun den Wert 10
```

lediglich mit einer Warnung bedenkt, meldet der C#-Compiler in solchen Situationen einen Syntaxfehler, da keine implizite (und auch keine explizite) Typumwandlung in den Typ `bool` definiert ist. In der Tat war diese beliebte Fehlerquelle – oder auch nur Einladung zur laxen Programmierung – in C/C++ Anlass dafür, in C# einen strikt isolierten logischen Datentyp `bool` (bzw. `System.Bool` in den .NET-Basisklassen) einzuführen.

Ein Bedingungsausdruck ist dann vom Typ `bool`, wenn er auf eine der folgenden Arten gebildet wird:

- eine boolesche Variable oder eine Methode mit dem Rückgabewert `bool`
- einen relationalen Operator <, >, <=, >= und vergleichbare Operanden
- einen (Un-)Gleichheitsoperator ==, != und vergleichbare Operanden
- einen booleschen Ausdruck (boolesche Operanden und boolesche Operatoren)



Wie bei C/C++ können die geschweiften Klammern um einen Anweisungsblock entfallen, wenn dieser nur aus einer einzigen Anweisung besteht. Kurzformen für `else if`*-Zweige, wie man sie in vielen Programmiersprachen findet, gibt es in C# nicht – und Delphi-Programmierer werden sich daran gewöhnen müssen, dass auch eine einzelne Anweisung nach einem* `if` *den Abschluss mit einem Semikolon verlangt.*

Es möglich, in den Anweisungsblöcken einer `if`*-Struktur Variablen zu vereinbaren, wenn gewährleistet ist, dass deren Bezeichner im übergeordneten Anweisungsblock (im Allgemeinen: der jeweiligen Methode) noch nicht bekannt sind. Die Sichtbarkeit solcher Variablen bleibt dann auf den jeweiligen Anweisungsblock beschränkt.*

switch

Während bei der `if`-Struktur ein zweiwertiger Bedingungsausdruck des Typs `bool` darüber entscheidet, welcher von zwei möglichen Zweigen zur Ausführung kommt, lassen sich mit einer `switch`-Struktur beliebig viele Fälle unterscheiden.

Die C#-Syntax der `switch`-Anweisung deckt sich gleichfalls weitgehend mit der von C/C++:

```
switch (Steuerausdruck)
{
  case KonstanterAusdruck:
    [AnweisungsblockMitBreak]
 [case KonstanterAusdruck:
    [AnweisungsblockMitBreak]
  ...]
 [default:
   [Anweisungsblock]]
 ]
}
```

Als Datentyp für den *Steuerausdruck* sind alle Ganzzahltypen wie `sbyte`, `uint` und `long`, `char` und – besonders hervorzuheben – `string` sowie natürlich alle Aufzählungstypen erlaubt. Darüber hinaus kommen auch Datentypen in Frage, für die eine implizite Typumwandlung in einen der genannten Typen definiert ist.

Die Logik der `switch`-Anweisung erschließt sich gewissermaßen von selbst. Die Ausführung vergleicht den Wert des Steuerausdrucks mit den Werten der konstanten Ausdrücke der `case`-Zweige. Damit dieser Vergleich eindeutige Ergebnisse liefert, müssen die `case`-Zweige mit unterschiedlichen Werten assoziiert sein. Eine besondere Ordnung der `case`-Zweige ist vom Prinzip her nicht erforderlich, kann aber bei der Zusammenfassung mehrerer Fälle *(fall through)* Code einsparen. Passt keiner der `case`-Zweige, kommt – sofern vorhanden – der `default`-Zweig zur Ausführung.

```
public enum documentDescriptor {Gif, Jpg, Bmp, Tif};

public string DocTypeSelect(string filename, documentDescriptor DocDesc)
{
  string fn;
  switch(DocDesc)
  {
    case documentDescriptor.Gif:
      fn = filename+".Gif";
      break;
    case documentDescriptor.Jpg:
      fn = filename+".Jpg";
      break;
    case documentDescriptor.Bmp:
      fn = filename+".Bmp";
      break;
    case documentDescriptor.Tif:
      return filename+".Tif";         // return! break nicht erforderlich
    default:                          // Kommt nie vor, sollte i.A. aber
      fn = filename;                  // dennoch implementiert werden
      break;
  }
  return fn;
}
```



Der Vollständigkeit halber sei erwähnt, dass man die Methode einfacher auch so hätte implementieren können:

```
public string DocTypeSelect1(string filename, documentDescriptor DocDesc)
{
  return filename + "." + DocDesc;    // implizit ToString()
}
```

Für C# gilt, dass *jeder* `case`-Zweig – auch der `default`-Zweig – sofern er Anweisungen enthält, per `break`-Anweisung oder wahlweise auch per `return` abgeschlossen werden muss.

Ein *fall through* ist nur für Fälle möglich, die keinen Code enthalten und geht immer soweit, bis ein Fall kommt, der Code enthält.

```
public enum Tage {Mo, Di, Mi, Do, Fr, Sa, So};
...
Tage MeinGeburtsTag;
...
switch(MeinGeburtsTag)
{
  case Tage.Mo:   // fall through bis zum Fall Tage.Fr
  case Tage.Di:
  case Tage.Mi:
  case Tage.Do:
  case Tage.Fr:
    result = "Werktag";
    break;
```

```
  case Tage.Sa:
    result = "Samstag";
    break;
  default:
    result = "Sonntag";
    break;
}
```

In C/C++ dürfen `case`-Zweige hingegen optional per `break` getrennt werden, eine Verpflichtung dazu besteht nicht. Das ermöglicht zwar komplexere *fall through*-Logiken, der allzu freie Gebrauch der `break`-Anweisung in der `switch`-Syntax birgt hier aber einiges mehr an Fehlerpotenzial als an praktisch wirklich nutzbarer Ausdruckskraft.

In seltenen Fällen wird man aber auch bei der Programmierung mit C# zur Vermeidung überflüssigen Codes den Wunsch verspüren, einen *fall through* über mehrere nicht leere Fälle auszudehnen. In dieser Situation erlaubt es die C#-Syntax, anstelle der `break`-Anweisung eine `goto`-Anweisung zu einem anderen Fall (oder zu `default`) zu setzen:

```
switch(MeinGeburtsTag)
{
  case Tage.Mo:
    Irgendeine.Anweisung();
    goto case Tage.Di;
  case Tage.Di:
  case Tage.Mi:
  case Tage.Do:
  case Tage.Fr:
    result = "Werktag";
    break;
  case Tage.Sa:
    result = "Samstag";
    break;
  default:
    result = "Sonntag";
}
```



Im Gegensatz zu Delphi überprüft C# bei Aufzählungstypen allerdings nicht, ob sämtliche möglichen Werte durch `case`*-Zweige bzw.* `default` *abgedeckt sind. Trotzdem – oder gerade deshalb – sollte man es sich zur Gewohnheit machen, jedem* `switch`*- einen* `default`*-Zweig zu verpassen, der gegebenenfalls dann eine Fehlermeldung ausgibt.*

6.5.2 Schleifen

Zum Thema Schleifen hat C# ausnahmsweise nicht viel Neues beizutragen. Wer von C/C++ oder Delphi kommt, dem mag die in der Tradition von Visual Basic stehende `foreach`-Schleife ungewohnt vorkommen, gut. Mancher wird es vielleicht auch als Einschränkung empfinden, das Abbruchkri-

terium »sauber« als booleschen Ausdruck formulieren zu müssen und nicht einfach auf die implizite Typumwandlung von Ganzzahltypen durch den Compiler bauen zu können – ein Umstand, der aber auf die Emanzipation des Datentyps `bool` zurückzuführen und nicht als Besonderheit des Schleifenkonzepts zu sehen ist. Das ist aber auch schon alles; brandneue Konzepte gibt es nicht.



Schleifen lassen sich erforderlichenfalls durch den Befehl `break` *(Fortsetzung der Ausführung bei der ersten Anweisung nach dem Schleifenkörper) explizit abbrechen. Der Befehl* `continue` *überspringt den Rest des Schleifenrumpfes und setzt die Ausführung im Schleifenkopf fort. Er bricht also nur den aktuellen Schleifendurchlauf ab, lässt die Schleife aber intakt. Zum Verlassen verschachtelter Schleifen eignet sich der Befehl* `goto` *marke, wobei marke eine »sichtbare« – das heißt: nicht in einem nebengeordneten Anweisungsblock befindliche – Sprungmarke sein muss.*

for

Die `for`-Struktur von C# deckt sich nahezu völlig mit der von C/C++:

```
for([Initialisierung]; [Schleifenkriterium]; [Aktualisierung])
{
  Anweisungsblock
}
```

Die Kontrollstruktur an sich ist dreiteilig und besteht aus:

- Einem optionalen *Initialisierungsteil*, der die Vereinbarung einer oder mehrerer Schleifenvariablen (gleichen Typs) ermöglicht.
- Einem optionalen *Schleifenkriterium*, das als Ausdruck vom Typ `bool` zu formulieren ist. Die Überprüfung des Kriteriums erfolgt vor Eintritt und vor jedem Wiedereintritt in die Schleife. Nur wenn sich bei der Auswertung des Ausdrucks `true` ergibt, wird der Anweisungsblock im Schleifenkörper abgearbeitet.
- Einem *Aktualisierungsteil*, der eine oder – für C# eine echte Ausnahme: durch Kommas getrennt – mehrere Aktualisierungsanweisungen enthalten kann. Die Abarbeitung erfolgt in Leserichtung, also von links nach rechts.

Enthält der optionale *Anweisungsblock* mehr als eine Anweisung, ist er in geschweiften Klammern zu notieren. Man kann eine `for`-Schleife vom Prinzip her so notieren, dass der Anweisungsblock leer bleibt und alle Anweisungen des Schleifenkörpers durch Kommas getrennt im Aktualisierungsteil stehen:

```
for(long a = 1, b = 2, c = 0; a < 20; a++, b += a, c = a+b)
       ; // nichts
```

Allerdings gilt das allgemein als schlechter Programmierstil. Man beachte, dass die syntaktisch zwar korrekte, aber etwas exotisch anmutende `for`-Anweisung außer ihrem Laufzeitverbrauch keine nach außen hin verwertbare oder auch nur feststellbare Wirkung hat – der Sichtbarkeitsbereich der Variablen endet am Ende des Schleifenkörpers, mithin also am Semikolon.

Eine `for`-Anweisung sollte nach Möglichkeit so formuliert sein, dass sie eine definierte Schleifenvariable besitzt, die in das Schleifenkriterium eingeht und im Aktualisierungsteil aktualisiert wird. Die Arbeit mit mehr als einer Schleifenvariablen ist in der Praxis kaum erforderlich. Variablen, die Werte aus einer `for`-Schleife heraustragen sollen, müssen außerhalb der Kontrollstruktur vereinbart werden. Für den vorgestellten Beispielcode wäre dies im Falle der Variablen `c` ratsam:

```
long c = 0;
for(long a = 1, b = 2; a < 20; a++)
  {
    b += (a + 1);
    c = (a + 1 ) + b;
  }
```

Durch diese Umstellung, gerät allerdings die Auswertungsreihenfolge durcheinander: In der ersten Formulierung findet die Aktualisierung von `a` vor der von `b` und `c` statt – und im zweiten Fall erst danach.

Ist das Schleifenkriterium implizit, empfiehlt sich im Allgemeinen die Verwendung einer `while`- oder `do`-Schleife. Die Aktualisierung einer gegebenenfalls zusätzlich vorhandenen Zählvariable kann dann im Schleifenkörper erfolgen.

while

Die `while`-Schleife ist eine abweisende Schleife. Die Kontrollstruktur prüft *vor* Eintritt in den Schleifenkörper, ob das *Schleifenkriterium*, welches als Ausdruck vom Typ `bool` formuliert sein muss, erfüllt ist. Falls die Auswertung des Schleifenkriteriums `false` ergibt, bricht die Schleife ab, ohne den Anweisungsblock (erneut) auszuführen.

```
while ( Schleifenkriterium )
{
  Anweisungsblock
}
```

Der folgende Codeauszug stellt die Arbeitsroutine der Windows-Anwendung *TextReader* vor, die als Beispiel im Abschnitt »string vs. StringBuilder« zu finden ist. Das Schleifenkriterium verlässt sich darauf, dass die Methode `ReadLine()` eines `StreamReader`-Objekts den Wert `null` zurückliefert, wenn das Ende des Streams (hier: Dateiende) erreicht ist.

```
// Gibt Inhalt des StreamReader-Objekts sr zeilenweise in Textfeld aus
public void ShowTextFileString(StreamReader sr)
{
  textBox1.Text=null;
  string Zeile;
  while(null != (Zeile = sr.ReadLine()))  // zeilenweise bis Dateiende
    textBox1.Text += (Zeile + "\r\n");
}
```

do … loop

Im Gegensatz zur `while`-Schleife prüft die `do`-Kontrollstruktur das *Schleifenkriterium* erst *nach* dem Durchlaufen des Schleifenkörpers. Sie ist damit eine nicht abweisende Schleife, weil der Schleifenkörper mindestens einmal zur Ausführung kommt. Ist der Wert des Schleifenkriteriums `true`, kommt der Anweisungsblock im Schleifenkörper erneut zur Ausführung, andernfalls bricht die Schleife ab.

```
do
{
  Anweisungsblock
}
while (Schleifenkriterium);
```

Die Leseschleife für die Windows-Anwendung *TextReader* (vgl. Seite 149) ließe sich auch als `do`-Schleife statt als `while`-Schleife formulieren – allerdings nur, weil die Operation `+ null` für Strings definiert ist und nicht etwa eine Ausnahme auslöst!

```
// Gibt Inhalt des StreamReader-Objekts sr zeilenweise in Textfeld aus
public void ShowTextFileString(StreamReader sr)
{
  textBox1.Text=null;
  string Zeile;
  do
  {
    Zeile = sr.ReadLine();
    // im letzten Durchlauf hat Zeile den Wert null, es wird
    // also nur noch ein Zeilenvorschub angehängt
    textBox1.Text += (Zeile + "\r\n");
  }
  while(null != Zeile);
}
```

Ein komplexeres Beispiel mit verschachtelten Schleifen und Sprunganweisungen findet sich im Abschnitt »Codebeispiel – Primzahlen aufzählen« auf Seite 157.

foreach

Mit der `foreach`-Schleife übernimmt C# ein Konzept, das in Visual Basic schon lange zur Grundausstattung der Sprache gehört. Iterationsmuster dieser Art ermöglichen es, Arrayobjekte oder Instanzen selbstdefinierter oder

(beispielsweise als .NET-Basisklasse) vordefinierter Auflistungen elementweise aufzuzählen bzw. zu durchlaufen. Die allgemeine Form der `foreach`-Schleife ist:

```
foreach (ElementTyp ElementVar in AufzählungsObjekt)
{
  Anweisungsblock
}
```

Ähnlich wie bei der `for`-Schleife wird bei der `foreach`-Schleife eine Laufvariable *ElementVar* vereinbart, die als Datentyp den Elementtyp des aufzuzählenden Objekts haben muss. Im Falle eines Arrays ist das der Elementtyp der aufzuzählenden Dimension, im Falle einer Auflistung der Basistyp der in der Auflistung gespeicherten Elemente.

Während sich die `for`-Anweisung nur zum expliziten Aufzählen von Objekten wie Arrays eignet, deren Implementierung Methoden für die Indizierung – beispielsweise einen Indexer – und das Abfragen der Elementanzahl bereitstellt, bietet die `foreach`-Schleife eine erheblich höhere Abstraktion, wenngleich sie im Grund genommen natürlich die gleichen oder zumindest ähnliche Forderungen an die Implementierung des Objekts stellt.

Die Implementierung, die C# Arrays (vgl. die abstrakte Basisklasse `System.Array`) zugrunde legt, erfüllt alle Voraussetzungen für den einfachen und unproblematischen Einsatz von `foreach`. Der folgende Code zerlegt einen Satz in einzelne Wörter, speichert sie in einem `string`-Array und zählt das Array dann für die Ausgabe an der Konsole auf:

```
string [] Words = "Das sind verschiedene Wörter".Split(" ".ToCharArray());
foreach (string s in Words)
  System.Console.WriteLine(s);
```

Auch für andere vordefinierte Aufzählungsklassen ist die Anwendung der `foreach`-Schleife kein großer Akt. Abbildung 6.15 zeigt den relevanten Code und die Konsolenausgabe (im Ausgabefenster) des einfachen Windows-Projekts *foreach-Demo*, das nichts weiter tut, als auf einen Mausklick hin die Einträge eines auf dem Formular befindlichen Listenfeld-Steuerelements über die Konsole auszugeben.

Ein gewisser Nachteil bei der Verwendung der `foreach`-Struktur ist die fehlende *Typsicherheit*. Der Compiler kann nicht in jedem Fall prüfen, ob der deklarierte Elementtyp mit dem tatsächlichen Elementtyp übereinstimmt. Abbildung 6.16 zeigt einen Fall, wo er das nicht kann – und den resultierenden Ausnahmefehler.

Das wirft natürlich die Frage auf, wie ein Objekt bzw. die dahinterstehende Klasse beschaffen sein muss, damit es in einer `foreach`-Schleife aufgezählt werden kann. Dafür gibt es einen sprachinternen und einen sprachübergrei-

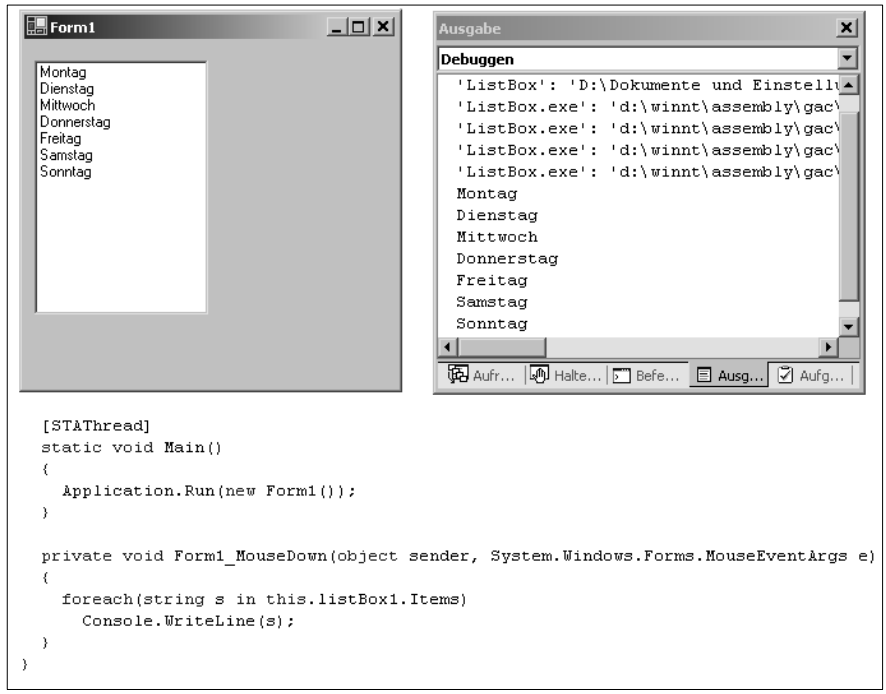


Abbildung 6.15: Einträge eines Listbox-Steuerelements aufzählen

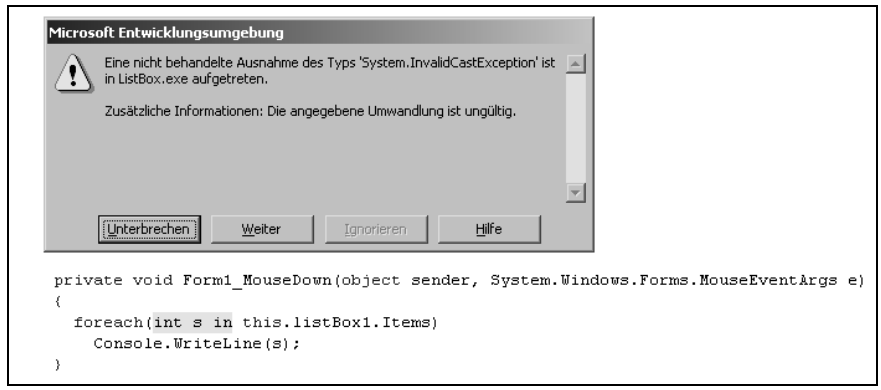


Abbildung 6.16: Auflistungen bieten nicht in jedem Fall Typsicherheit – und ein unpassender Elementtyp fällt dann erst zur Laufzeit auf.

fenden Ansatz. Um es gleich vorwegzunehmen: Der sprachinterne Ansatz bietet Typsicherheit, der sprachübergreifende nicht.

Zuerst ein Blick auf den *sprachinternen Ansatz*: Damit die Objekte einer Klasse sprachintern in einer `foreach`-Struktur aufgezählt werden können, muss die Klasse eine Methode `GetEnumerator()` mit der folgenden Signatur implementieren:

```
// liefert ein Objekt, das die Methoden MoveNext,
// Current und Reset bereitstellt
public EnumClass GetEnumerator() ...
```

Damit aber nicht genug. Wie bereits im Kommentar angedeutet, muss das von dieser Methode gelieferte Objekt seinerseits drei Methoden mit den folgenden Signaturen bereitstellen:

```
// Bereitet den Zugriff auf nächstes Element vor.
// Achtung: Diese Methode wird auch vor dem Zugriff
// auf das erste Element verwendet!
public bool MoveNext() ...
// Eigenschaft, liefert nächstes Element
public ElementTyp Current{get;} ...
// Aufzählung von vorn beginnen
public void Reset() ...
```

Nachdem der Rückgabetyp der Methode `Current()` beim sprachinternen Ansatz im Allgemeinen exakt an den jeweiligen Elementtyp angepasst werden kann, kann der Compiler die Typsicherheit von `foreach`-Konstrukten gewährleisten, was beim sprachübergreifenden Ansatz nicht der Fall ist.

In der Praxis ist es natürlich auch möglich, dass die fragliche Klasse alle vier Methoden selbst bereitstellt und `GetEnumerator()` schlicht `this` zurückgibt. Der Code ist in diesem Fall aber nicht mehr *reentrant*, und bei verschachtelten `foreach`-Schleifen ebenso wie in Multithreading-Umgebungen sind hier natürlich Probleme und seltsame Effekte zu erwarten (hierzu gleich noch mehr).

Angesichts dieser Ausstattung ist klar, wie der Compiler die `foreach`-Kontrollstruktur implementiert: Er ruft einmalig `GetEnumerator()` des aufzuzählenden Objekts auf, und dann die Methode `Reset()` des von `GetEnumerator()` bereitgestellten Aufzählobjekts. Für den Elementzugriff ruft er zuerst `MoveNext()` und weist dann der Laufvariablen den Wert der Eigenschaft `GetCurrent` – jeweils des Aufzählobjekts – zu.

Der folgende Code veranschaulicht die Logik einer `foreach`-Schleife unter Verwendung einer `for`-Schleife. (Er geht davon aus, dass die Klasse `MyEnumClass` des aufzuzählenden Objekts alle vier Methoden implementiert.)

```
MyEnumClass MyEnumObject = new ... ;

{ // foreach(ElemType et in MyEnumObject)
  // {
  //   Anweisungsfolge
  // }
  MyEnumClass mec = MyEnumObject.GetEnumerator();
  mec.Reset();
  mec.MoveNext();
  for(ElemType et = mec.Current; mec.MoveNext(); et = mec.Current)
  {
    Anweisungsfolge
  }
}
```

Der *sprachübergreifende Ansatz* sieht vom Prinzip her genauso aus, bedient sich aber des etwas formelleren Mittels der Schnittstellenvererbung, um einem gegebenenfalls anderssprachlich implementierten Clientobjekt die Sicherheit zu geben, dass die entsprechenden Methoden mit der gewünschten Signatur bereitgestellt werden. Eine Klasse, die von einer Schnittstelle abgeleitet ist, geht bekanntlich einen Vertrag ein (den der Compiler durchsetzt), die von der Schnittstelle »geforderten« Elemente zu implementieren.

Damit die Objekte einer Klasse sprachübergreifend über den `foreach`-Mechanismus aufgezählt werden können, muss die Klasse von der Schnittstelle `IEnumerable` abgeleitet sein. Die Definition dieser Schnittstelle sieht so aus:

```
interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Wie man sieht, liefert die darin deklarierte Methode `GetEnumerator()` einen `IEnumerator`-Schnittstellenzeiger zurück – bzw. einen Verweis auf ein Objekt, das die Schnittstelle `IEnumerator` unterstützt. Wie zu erwarten, lautet die Definition dieser Schnittstelle:

```
interface IEnumerator
{
  object Current { get; }
  bool MoveNext();
  void Reset();
}
```

Um den Aspekt der Typsicherheit noch einmal aufzugreifen: Die `Current`-Eigenschaft trägt den Typ `object`, weshalb für die Zuweisung ihres Wertes an die Laufvariable eine Typumwandlung erforderlich ist, die erst zur Laufzeit geschieht – der Compiler kann hier also keine Typsicherheit gewährleisten!

Im Prinzip spricht nichts dagegen, dass eine Klasse beide Schnittstellen implementiert:

```
class MyEnumClass : IEnumerable, IEnumerator
{
...
}
```

Allerdings vergibt man sich damit wieder die Möglichkeit, ein und dasselbe Objekt in einem bestehenden Kontext mehrfach aufzählen zu können – beispielsweise rekursiv, wie im nachfolgenden Beispielprogramm *PrimNumbers*.

Eine saubere schnittstellenbasierte Implementierung verkapselt die Implementierung der `IEnumerator`-Schnittstelle in einer eingebetteten Klasse und liefert mit jedem Aufruf von `GetEnumerator()` ein neues Objekt dieser Klasse:

```
class MyEnumClass : IEnumerable
{
...
  public IEnumerator GetEnumerator()
  {
    return new MyEnumerator(this);
  }
  private class MyEnumerator : IEnumerator  // eingebettete Klasse
  {
    private MyEnumClass collection;
    private int index = -1;
  ...
    public MyEnumerator(MyEnumClass mec)  // Konstruktor
    {
      collection = mec;
    }
    /// TODO: Implementierung der Schnittstellenelemente mit
    ///       Aufzählungslogik
  }
}
```

Nach alldem dürfte klar sein, dass es ein hinreichendes Kriterium für die Aufzählbarkeit einer Klasse in einer `foreach`-Schleife ist, wenn sie die Schnittstelle `IEnumerable` implementiert. Im Namensraum `System.Collections` finden sich eine Reihe von Schnittstellendefinitionen wie `ICollection`, `IDictionary` und `IList`, die ihrerseits von `IEnumerable` abgeleitet sind. Damit versteht sich, dass auch Klassen, die eine dieser Schnittstellen in ihrer Vererbungsliste haben, in einer `foreach`-Schleife aufzählbar sind. Wie zu erwarten, gilt das für die meisten im Namensraum `System.Collections` definierten Klassen, aber auch für viele in anderen Namensräumen definierte Klassen (besonders prominent: `System.Array`), sowie natürlich für alle Klassen, die diese Klassen wiederum als Basisklassen haben.

Ein praxisnahes Beispiel mit der Implementierung einer aufzählbaren Klasse namens *PrimNumbers* finden Sie im übernächsten Abschnitt »Codebeispiel – Primzahlen aufzählen« auf Seite 157.

6.5.3 Unbedingte Verzweigung, Sprünge

Die unbedingte Verzweigung bzw. der explizite Sprung ist eines der Konzepte, die in Hochsprachen zwar schon lange verpönt, aber dennoch nicht totzukriegen sind. Für die objektorientierte Programmierung verbieten sich generell Sprünge, die aus dem Geltungsbereich der jeweiligen Methode hinausführen, da der Kontext hier eine wichtige Rolle spielt. Während C++ mit den Anweisungen *setjmp()* und *longjmp()* noch einen Notnagel für den Fall

der Fälle bereitstellte, der das Abspeichern eines Ausführungskontextes und den späteren Rücksprung in diesen Kontext von einem beliebigen Punkt des Programms aus ermöglichte, muss sich der C#-Programmierer in solchen Situationen allein auf den Fehlerbehandlungsmechanismus beschränken – er ist damit aber bestens für alle erdenklichen Situationen gerüstet.

Tabelle 6.1 stellt die fünf von C# bereitgestellten Sprunganweisungen, ihren Einsatzbereich und ihre Wirkung vor.

Tabelle 6.1: Die unbedingten Sprunganweisungen von C#

Anweisung	Anwendbar in	Rolle und Wirkung
`break`	`switch`, `for` `foreach`, `while`, `do`	Unbedingter, sofortiger Abbruch einer Schleife (bei verschachtelten Schleifen: nur aktuelle Schleife); Muss gesetzt werden, um das Ende eines `case`-Zweiges in einer `switch`-Struktur zu kennzeichnen.
`continue`	`for`, `foreach`, `while`, `do`	Unbedingter Sprung zur Prüfung des Schleifenkriteriums für den nächsten Schleifendurchlauf. (Beendet den aktuellen Schleifendurchlauf mit sofortiger Wirkung.)
`goto` *Marke*	überall, mit Sprungziel im Geltungsbereich der Methode	Unbedingter Sprung zu einer durch eine Sprungmarke benannten Anweisung innerhalb der aktuellen Methode
`return` [*Wert*]	überall	Beendet die Methode. Falls der Rückgabetyp einer Methode nicht `void` ist, muss (in jedem Codepfad) ein Funktionswert mit dem geforderten Datentyp per `return`-Anweisung an den Aufrufer zurückgegeben werden.
`throw` *Ausn*	überall	generiert eine Ausnahme, die innerhalb einer `try/catch`-Struktur behandelt werden kann. Unbehandelte (das heißt: von der aktuellen Methode nicht in einer `try/catch`-Struktur behandelte) Ausnahmen werden automatisch an die jeweils aufrufende Methode weitergereicht. In letzter Instanz ist dies das .NET-Laufzeitsystem, welches das Programm unterbricht, eine Fehlermeldung ausgibt und den Start eines Debuggers anbietet.

Das im folgenden Abschnitt vorgestellte Codebeispiel stellt die ersten vier Sprunganweisungen aus Tabelle 6.1 im praktischen Einsatz vor.

6.5.4 Codebeispiel – Primzahlen aufzählen

Das folgende etwas komplexere Codebeispiel, eine vollständige Konsolenanwendung namens *PrimNumbers*, demonstriert das Zusammenspiel verschiedener Schleifen, bedingter Verzweigungen und unbedingter Sprünge. Das Programm zeigt nicht nur die vollständige Implementierung einer aufzählbaren Klasse entlang des sprachübergreifenden Ansatzes, es stellt auch eine ungewöhnliche (weil rein prozedurale) Repräsentation für endliche Primzahlenfolgen vor.

Primzahlalgorithmen

Die effizientesten bekannten Primzahlalgorithmen arbeiten nach dem so genannten »Sieb des Eratosthenes« (vgl. beispielsweise: `http://www.mathe-prisma.uni-wuppertal.de/Module/Primz/SiebE/SiebErat.htm`), einem Abzählverfahren, bei dem beginnend mit der Primzahl 2 alle Primzahlvielfachen bis zu einer gegebenen Zahl *N* gestrichen werden. Die auf die jeweils vorige Primzahl folgende, noch nicht gestrichene Zahl ist wiederum eine Primzahl, deren Vielfache erneut zu streichen sind. Diese »Siebung« wird fortgesetzt, bis alle Primzahlen kleiner *N* gefunden sind.

Die Klasse `PrimNumbers` des Beispielprogramms implementiert einen anderen, wie gesagt, recht eigenwilligen Algorithmus, der zwar bei weitem nicht so effizient ist, aber Einblick in das »Wesen des Aufzählens« gibt. Ein Objekt dieser Klasse »repräsentiert« eine bei Konstruktion anzugebende feste Anzahl von Primzahlen. Obwohl man eine Repräsentationsstruktur für diese Zahlen vergebens suchen wird, ist das Objekt in der Lage, die Primzahlfolge von 2 an in der jeweiligen Länge aufzuzählen. Dazu prüft es die ungeraden Zahlen größer 2 in aufsteigender Folge daraufhin, ob sie eine Primzahl sind. Da eine Zahl dann eine Primzahl ist, wenn sie keinen Teiler hat, und ein Teiler immer kleiner gleich der Wurzel der Zahl sein muss, reicht es für die Primzahlprüfung, nur die Primzahlen bis zur Wurzel der Zahl als Teiler zu betrachten. Die Implementierung ermittelt diese Primzahlen rekursiv, indem sie das Objekt jeweils erneut aufzählt – aber eben nicht mehr ganz so weit, sodass die Rekursion sicher abbricht. Man beachte, dass in jeder Rekursionsstufe ein neues `IEnumerator`-Objekt vereinbart wird. Dafür sorgt der Compiler mit seinem impliziten Aufruf von `GetEnumerator()`. Andernfalls wäre die verschachtelte (reentrante) Aufzählung nicht möglich.

Indexer

Sozusagen der Vollständigkeit halber stellt die Klasse auch einen Indexer bereit, der die *i*-te Primzahl liefert. Die Implementierung des Indexers sieht so aus, dass sich die Methode ein Objekt derselben Klasse mit Folgenlänge *i* vereinbart, dieses aufzählt und die letzte Primzahl als Funktionswert zurückgibt. (Mehr über den Begriff des Indexers finden Sie im Abschnitt 8.3 (Seite 257))

Der Code

Hier der vollständige Code der Klasse `PrimNumbers` sowie etwas Testcode, der ihre prinzipielle Funktionstauglichkeit demonstriert (zur Ausgabe vgl. Abbildung 6.17):

```
using System;
using System.Collections;

namespace PrimNumbers
{
  class PrimNumbers : IEnumerable
  {
    int MaxIndex;    // Objekt repräsentiert alle Primzahlen
                     // bis zur MaxIndex-ten Primzahl

    public PrimNumbers(int Max)   // Konstruktor
    {
      MaxIndex = Max;
    }

    // Deklaration eines Indexers für den Array-orientierten
    // Zugriff auf das Objekt
    public long this [int index]
    {
      get
      {
        // Indexgrenzen testen, Gültigkeit erzwingen!
        if (index <= 0 || index > MaxIndex)
          return 0;
        else
        {
          // Alle Primzahlen bis Index aufzählen
          // und letzte zurückgeben
          PrimNumbers Prims = new PrimNumbers(index);
          long ret = 0;
          foreach(long r in Prims)
            ret = r;
          return ret;
        }
      }
    }

    // Implementierung der Schnittstelle IEnumerable
    public IEnumerator GetEnumerator()
    {
      return new PrimEnum(this);
    }
```

```
// Definition der eingebetteten Auflistungsklasse
private class PrimEnum : IEnumerator
{
  private int nIndex = -1;          // Index der Aufzählung
  private PrimNumbers collection;
  private long LastPrimFound = 2;   // Zuletzt gefundene Primzahl

  public PrimEnum(PrimNumbers p)
  {
    collection = p;
  }

  // Nächstes Element
  public bool MoveNext()
  {
    nIndex++;
    return(nIndex < collection.MaxIndex);
  }

  public object Current
  {
    get
    {
      switch (nIndex)
      {
        case 0:
          return LastPrimFound = 2; break;
        case 1:
          return LastPrimFound = 3; break;
        default:
          do
          {
            LastPrimFound+=2;
            // Teste alle Primzahlen bis zur Wurzel, ob sie Teiler sind
            // Hier geht es in die Rekursion!
            foreach (long p in collection)
            {
              if (LastPrimFound % p == 0)
              {
                goto Continue;  // Zahl lässt sich teilen
                // äußere Schleifen forsetzen
              }
              else if (p * p > LastPrimFound)
                break;   // weitere Teiler zu probieren, wäre sinnlos
            }
            // Primzahl ist gefunden, äußere Schleife abbrechen
            break;

          Continue: // Marke muss vor einem Befehl stehen!
               continue;
          }
          while(true);     // Endlosschleife

          return LastPrimFound;
      }  // switch
    }
  }
```

```
      public void Reset()
      {
        nIndex = -1;
      }
    }

    [STAThread]
    static void Main(string[] args)
    {
      PrimNumbers Prims = new PrimNumbers (300);

      // Alle Primzahlen aufzählen
      foreach (long p in Prims)
        System.Console.Write("{0,5}", p);
      System.Console.WriteLine("\n\r");

      // Einzelne Primzahlen abfragen, Indexer testen
      PrimNumbers Prims1 = new PrimNumbers (1000);
      System.Console.WriteLine(Prims1[500]);
      System.Console.WriteLine(Prims1[1000]);
    }
  }
}
```

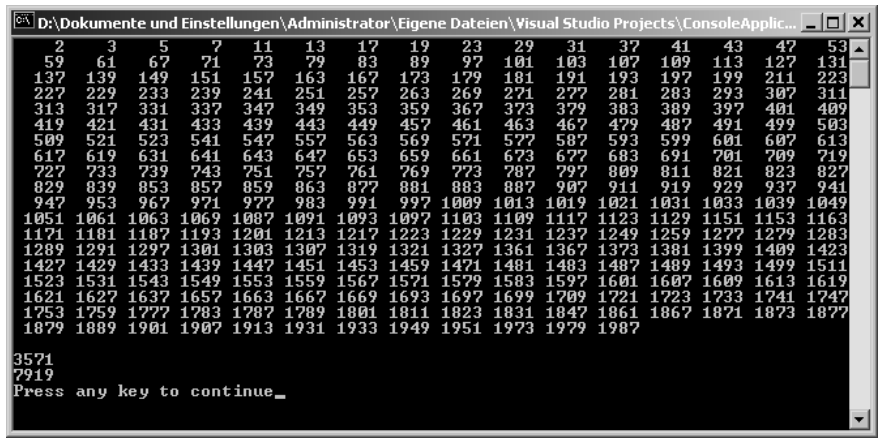


Abbildung 6.17: Das Programm gibt die ersten 300 Primzahlen sowie die 500ste und die 1000te Primzahl aus.

Interessanterweise bedient sich der Code hier selbst einer `foreach`-Kontrollstruktur, für die er letztlich die Implementierung liefert – Münchhausen lässt grüßen. Die Logik der beiden verschachtelten Schleifen ist etwas vertrackt: Sobald ein Teiler gefunden ist, bricht die innere `foreach`-Schleife ab und setzt die äußere Schleife fort, die den nächsten Primzahlkandidaten generiert und testet. Die `continue`-Anweisung bedient in diesem Fall nur die Forderung, dass auf eine Sprungmarke eine Anweisung folgen muss; Von der Logik her könnte sie entfallen, da die Schleife an dieser Stelle ohnehin fortgesetzt würde. Passt ein Teiler nicht, gibt das Wurzelkriterium darüber

Auskunft, ob es überhaupt Sinn macht, weitere Teiler zu probieren. Falls ja, geht die verschachtelte Aufzählung weiter, andernfalls bricht `break` die äußere Schleife und damit den Test des Primzahlkandidaten ab: Eine neue Primzahl ist gefunden und wird als `get`-Wert der `Current`-Eigenschaft zurückgegeben. Das `PrimEnum`-Objekt speichert sie zudem in einem privaten Datenfeld als Ausgangspunkt für die nächste Primzahl in der Aufzählung.

7 Datentypen

Alleiniger Sinn und Zweck einer Programmiersprache ist die Datenverarbeitung im eigentlichen Sinne des Wortes. Prozedurale Programmiersprachen unterscheiden für diesen Zweck einen wohldefinierten Satz an grundlegenden Datentypen, die auch als *integrierte Datentypen* der Sprache bezeichnet werden. Jeder dieser Datentypen ist die Kombination einer elementaren Repräsentation und Interpretation von Werten aus einem spezifischen Wertebereich. Außerdem definiert ein Datentyp einen Satz von Operationen über diesen Werten, die es ermöglichen, Werte des Datentyps miteinander oder mit Werten anderer Datentypen zu verknüpfen (beispielsweise Addition) oder miteinander in Bezug zu setzen (beispielsweise Vergleich).

Höhere Programmiersprachen wie C# stellen darüber hinaus sprachliche Mittel bereit, um integrierte Datentypen zu aggregieren, auf diese Weise neue, so genannte *komplexe Datentypen* zu generieren und deren Operationen zu definieren.

Damit aber nicht genug. Um Werte, gleich welchen Datentyps, vernünftig organisieren, manipulieren und transportieren zu können, benötigt eine Programmiersprache ein ausgefeiltes Benennungssystem, auch *Variablenkonzept* genannt.

Die integrierten Datentypen der Sprache C# lassen sich, wie im vorangehenden Kapitel diskutiert, in *Werttypen* und *Verweistypen* unterteilen. Dieses Kapitel stellt diese Datentypen einzeln vor und erörtert die dahinter stehenden Konzepte und Besonderheiten speziell mit Blick auf ihren Neuheitswert.

7.1 Integrierte Datentypen

C# stellt eine reiche Auswahl an integrierten Datentypen zur Verfügung – fünfzehn an der Zahl, davon dreizehn Wert- und zwei Verweistypen. Obwohl die Implementierungen für all diese Datentypen Teil der .NET-Basisklassen sind und damit eigentlich vom .NET-Framework gestellt werden, merkt man als Programmierer nichts davon, da die Sprache Aliasnamen in Form von Schlüsselwörtern dafür bereitstellt. Die Abbildung der sprachinternen Datentypen ist eine reine Umbenennung, weshalb es in der Tat völlig gleich ist, ob man bei Vereinbarung eines einfachen Datentyps einen C#-Typbezeichner angibt, also:

```
long lWert;
```

oder gleich den entsprechenden Bibliothekstyp:

```
System.Int64 lWert;
```

Beide Formulierungen sind identisch und führen zum gleichen Code.



Ausnahmen bestätigen die Regel: Der Datentyp `void`*, der in C# nur als Rückgabetyp für Methoden auftreten kann, stimmt nicht mit* `System.Void` *überein – er ist sprachspezifisch implementiert.*

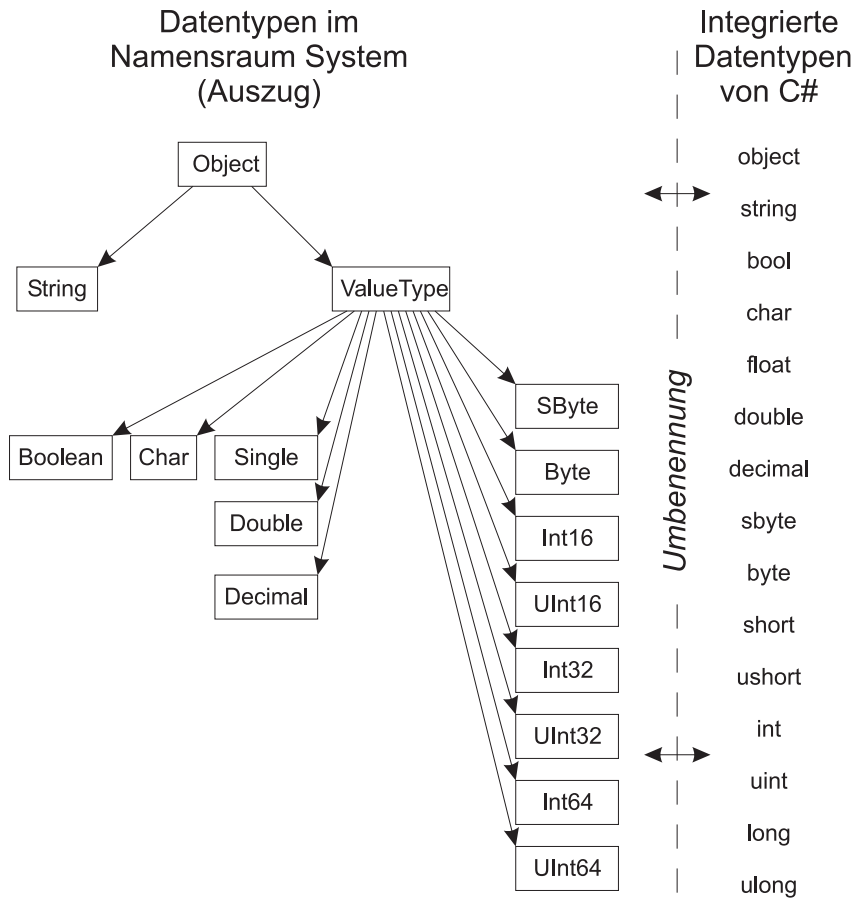


Abbildung 7.1: C# bezieht seine integrierten Datentypen aus dem Namensraum `System` der .NET-Basisklassen.

7.1.1 Einfache Werttypen

Was die lokalen Variablen einer Methode angeht, ist der Speicherort für Werttypen der Stack (vgl. aber Boxing/Unboxing). Der Speicher für einen Werttyp wird bei Ausführung der Variablendefinition belegt und am Ende des Geltungsbereichs der jeweiligen Variablen wieder freigegeben.

Vereinbarung und Initialisierung

Die C#-Syntax für die Vereinbarung von Werttypen ist C/C++ entlehnt: Hier wie dort besteht die Möglichkeit, Deklaration und Initialisierung en bloc vorzunehmen oder in zwei getrennte Anweisungen zu packen:

```
float fWert1;
fWert = 10.1;
float fWert2 = 100;
```

Eine Vorbelegung mit einem typspezifischen Standardwert (etwa: 0 für numerische Werte), wie Sie es vielleicht von Visual Basic her gewohnt sind, findet für lokale Variablen *nicht* statt. Der Compiler meldet deshalb einen Fehler, wenn er eine Variable in einem Rechtswert vorfindet, die zuvor nicht initialisiert wurde.

```
float f;
float g;
  g = Wurzel(f);   // f ist hier noch nicht initialisiert
```

Die Fehlermeldung lautet:

```
... Verwendung von möglicherweise nicht zugewiesenen lokalen Variablen 'f'
```

Vereinfachend kennt C# auch die kaskadierte Vereinbarung für mehrere Variablen ein und desselben Typs, wobei die Abarbeitung von links nach rechts geschieht und auch hier wahlweise eine sofortige Initialisierung möglich ist:

```
int iWert = 10;
long lWert = iWert, m, n = lWert;
```

Diese Art der Vereinbarung ist C/C++ entlehnt. Beachten Sie, dass die gleiche Schreibweise in VB6 eine völlig andere Bedeutung hat (Variablen ohne vorangestellten Typbezeichner werden implizit mit dem Vorgabetyp – im Allgemeinen `Variant` – vereinbart). Ein genauer Blick auf das Fragment gibt weiterhin einen Hinweis darauf, dass der C#-Compiler implizite Typkonvertierungen unterstützt, solange gewährleistet bleibt, dass der Wertebereich des Linkswertes den des Rechtswerts umfasst und somit ein Überlauf zur Laufzeit ausgeschlossen ist:

```
uint uWert = 10;
long lWert = uWert;   // das funktioniert
float fWert = uWert;  // das auch
int iWert = uWert;    // das toleriert der Compiler nicht
```

Typumwandlung

Typumwandlungen, die rein formal auf eine Wertbeschneidung hinauslaufen, wie etwa von `uint` nach `int` oder von `double` nach `byte`, müssen explizit mittels *cast-Operator* notiert werden: Dazu stellt man dem Rechtswert den gewünschten Ergebnistyp in Klammern voran:

```
double dWert = 10.7;
byte bWert = (byte) dWert; // Nachkommastellen werden abgeschnitten
Console.WriteLine(bWert);
```



Zusätzlich zum cast-Operator bieten einzelne Datentypen oft auch statische Methoden für die Typumwandlung in anderen Datentypen an. Darüber hinaus gibt es noch die .NET-Klasse `Convert`*, die darauf spezialisiert ist, Typumwandlungen für Werttypen zu handhaben und zahlreiche statische Methoden mit einer Unmenge von Überladungen bereitstellt, sodass letztlich immer eine Umwandlung von einem Werttyp in einen anderen Werttyp möglich ist.*

Ganzzahltypen – sbyte, byte, short, ushort, int, uint, long, ulong

C# unterscheidet acht verschiedene Ganzzahltypen, deren Repräsentation von 8 bis 64 Bit reicht.

Tabelle 7.1: Ganzzahltypen und ihre Wertebereiche

C#-Typ	CTS-Name	Wertebereich
`sbyte`	`System.SByte`	-128 bis 127
`byte`	`System.Byte`	0 bis 255
`short`	`System.Int16`	-32.768 bis 32.767
`ushort`	`System.UInt16`	0 bis 65.535
`int`	`System.Int32`	-2.147.483.648 bis 2.147.483.647
`uint`	`System.UInt32`	0 bis 4294967295
`long`	`System.Int64`	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
`ulong`	`System.UInt64`	0 bis 18.446.744.073.709.551.615

Man beachte, dass C# damit eine auf 64-Bit-Prozessoren ausgerichtete – und somit zukunftsweisende – Repräsentation für die Datentypen `short`, `int` und `long` zugrunde legt. Außerdem »dekliniert« die Sprache die Typen ein-

zeln aus, wie in den .NET-Basisklassen angeboten. Das von C/C++ her bekannte Schlüsselwort `unsigned` existiert somit nicht.

Implizite Typkonvertierungen funktionieren nur in einer Richtung, vom vorzeichenlosen zum vorzeichenbehafteten Wert, und auch nur dann, wenn der vorzeichenbehaftete Wert »um eine Nummer größer« ist.

Ganzzahlliterale lassen sich dezimal oder hexadezimal notieren. Um Zweideutigkeiten (insbesondere bei Funktionsaufrufen) zu vermeiden, kann man ein Literal durch Anhängen des Buchstabens `U` bzw. `u` als vorzeichenlosen `int`-Wert, durch Nachstellen von `L` bzw. `l` als `long`-Wert sowie durch Anhängen von `UL` bzw. `ul` als vorzeichenlosen `long`-Wert qualifizieren. Obwohl der Compiler mit seiner impliziten Typkonvertierung im Allgemeinen richtig liegt, sollte man diese Buchstaben der besseren Lesbarkeit halber notieren, und zwar möglichst in Großschreibung, da der Buchstabe `l` leicht mit der Ziffer `1` zu verwechseln ist. Hier einige Beispiele:

```
int i = 1010;           // dezimale Notation
uint ui = 1010u;        // dezimale Notation, qualifiziert
long l1 = -1;           // dezimale Notation
long l = -1L;           // dezimale Notation, qualifiziert
byte b = 0xff;          // hexadezimale Notation
// short j = 0xffff;    // Fehler: Wertebereich überschritten
short j = -0x1;         // das geht
ushort us = (ushort) j; // hier wird 65535 zugewiesen(!)
ulong ul = 1UL;         // dezimal und qualifiziert
```

Gleitkommatypen – float, double

C# kennt (wie für Programmiersprachen üblich) zwei Gleitkommatypen: `float` mit 32-Bit-Repräsentation und `double` mit 64-Bit-Repräsentation. Entsprechend unterscheiden sich ihre Genauigkeiten und Wertbereiche.

C#-Typ	CTS-Name	Wertebereich
`float`	`System.Single`	7 signifikante Dezimalstellen $\pm 1{,}5 \cdot 10^{-45}$ bis $\pm 3{,}4 \cdot 10^{38}$
`double`	`System.Double`	15 signifikante Dezimalstellen $\pm 5{,}0 \cdot 10^{-324}$ bis $\pm 1{,}7 \cdot 10^{308}$

Tabelle 7.2: Gleitkommatypen und ihre Wertebereiche

Für die Notation von Gleitkommaliteralen gibt es folgende Regeln:

- Gleitkommaliterale werden im amerikanischen Format notiert (Punkt trennt Vor- und Nachkommaanteil)
- Gleitkommaliterale mit einfacher Genauigkeit (`float`) müssen durch Anhängen des Buchstabens `F` bzw. `f` als solche gekennzeichnet werden: `10.3F`. Fehlt er, interpretiert der Compiler das Literal als `double`.

- Gleitkommaliterale mit doppelter Genauigkeit (`double`) können durch Anhängen des Buchstaben `D` bzw. `d` als solche gekennzeichnet werden: `.3D`. Fehlt er, interpretiert der Compiler das Literal allerdings auch standardmäßig als `double`.
- Ganzzahlliterale konvertiert der Compiler implizit und ggf. durch Rundung in das jeweilige Gleitkommaformat.
- Exponentenanteile werden durch E bzw. e abgetrennt: `10.0E10`. Sie können auch negativ sein: `10.0E-10`.
- Sofern in einem Ausdruck auch nur ein Gleitkommawert erscheint, wandelt der C#-Compiler implizit alle Operanden in Gleitkommawerte um und wendet Gleitkommaarithmetik an.
- Kommen in einem Ausdruck Gleitkommawerte einfacher und doppelter Genauigkeit vor, wandelt der C#-Compiler implizit alle Operanden in `double`-Werte um.



Der IEEE-Gleitkommatyp Extended mit einer Genauigkeit von 80 Bit, der von Intels mathematischen Coprozessoren intern eingesetzt wird, fehlt in C#. Ob das ein Zugeständnis an C/C++ oder an die postulierte Plattformunabhängigkeit ist, muss offen bleiben.

Dezimaler Typ – decimal

Der Datentyp `decimal` ist für C/C++-Programmierer ein Novum, für VB-Programmierer »alter Schnee«, und Delphi-Programmierern als `Comp` (wenn dort auch »nur« mit 80 Bit) bekannt. Da er im Gegensatz zu `float` und `double` keine Rundungsfehler im Zusammenhang mit der Multiplikation und der Division verursacht, eignet er sich insbesondere für finanzmathematische Aufgabenstellungen, wo es auf exakte Berechnungen mit maximaler Präzision (bis zu 28 Dezimalstellen) ankommt. Abbildung 7.3 entnehmen Sie den beachtlichen Wertebereich, der sich durch die 128-Bit-Repräsentation ergibt.

Tabelle 7.3: Der Datentyp `decimal`

C#-Typ	CTS-Name	Wertebereich
`decimal`	`System.Decimal`	28 signifikante Dezimalstellen $\pm 1{,}0 \cdot 10^{-28}$ bis $\pm 7{,}9 \cdot 10^{-28}$

`decimal`-Literale müssen unter Anhängen des Buchstabens `M` bzw. `m` notiert werden, sofern sie Nachkommastellen haben. Fehlt er, interpretiert der Compiler einen Wert mit Nachkommastellen standardmäßig als `double` und meldet einen Fehler. Ganzzahlwerte konvertiert der Compiler hingegen implizit.

```
decimal d0 = 10M;       // qualifizierte Notation
decimal d1 = 10L;       // Compiler konvertiert implizit
decimal d2 = 0xffff;    // Compiler konvertiert implizit
```

```
// decimal d3 = 10.0  // Fehler: Compiler interpretiert ein double-Literal!
decimal d4 = 10.000000000000000001m; // das ist korrekt
```

Boolescher Typ – bool

Den logischen Datentyp `bool` behandelt der C#-Compiler nicht nur als eigenständigen Typ – was mit Blick auf andere Programmiersprachen keine Selbstverständlichkeit darstellt –, er schottet ihn auch recht rigide gegen alle Typumwandlungsversuche impliziter wie expliziter Natur ab. Der Grund dafür ist, dass die zu laxe Handhabung dieses Datentyps in anderen Programmiersprachen erstens eine nicht zu unterschätzende Fehlerquelle (C/C++-Entwickler seien an dieser Stelle herzlich gegrüßt) und zweitens eine ewige Verlockung für ebenso unsaubere wie schlecht lesbare Formulierungen (dieser Gruß geht auch an die Adresse der VB-Entwickler) darstellt.

Tabelle 7.4: Der Datentyp `bool`

C#-Typ	CTS-Name	Wertebereich
`bool`	`System.Boolean`	`true` und `false`

Als literale Werte für den Datentyp `bool` sind nur die beiden reservierten Schlüsselwörter `true` und `false` vorgesehen – Zahlenwerte definitiv nicht. Der Datentyp `bool` spielt eine große Rolle bei der Ablaufsteuerung und erscheint vornehmlich in Kontrollstrukturen wie `if`, `for`, `while` etc.:

```
bool b = x == y;        // Initialisierung mit logischem Ausdruck
if (b)
   Anweisung();
```

Das folgende Fragment zeigt ein Fehlermuster, das in C/C++ ein Quell vielen Unheils war, in C# jedoch aus syntaktischen Gründen vom Compiler moniert wird – eben weil für `bool` keine Typumwandlung definiert ist.

```
int x = 1;
// if (x = 0)            // diesen Fehler schließt C# syntaktisch aus!
if (x == 0)
    Anweisung();
```

Zeichen – char

Auch der Datentyp `char`, in C/C++ noch nichts anderes als eine Umbenennung für das vorzeichenlose Byte, hat sich unter C# zu einem eigenständigen Datentyp gemausert – wenngleich nicht ganz so strikt wie `bool`, da der Compiler die Typumwandlung zumindest in Richtung `ushort` implizit betreibt und expliziten Typumwandlungen mit anderen Ganzzahltypen nichts in den Weg legt.

```
ushort us = 65;
char c = (char) us;   ' explizit
us = c;               ' implizit
```

Wie von Java vorgelegt, repräsentiert C# einzelne Zeichen (und damit auch Strings) ausschließlich im Unicode, also mit 16 Bit. Obwohl der ASCII-Code damit vom Prinzip her ausgedient hat, findet er sich – dieses Zugeständnis mussten die Unicode-Entwickler der heiligen Kuh »Kompatibilität« machen – noch in den unteren 8 Bit der Repräsentation. Die oberen 8 Bit kodieren den jeweiligen Zeichensatz und sind beim reinen ASCII-Code schlicht 0. Diesen Zusammenhang sollte man kennen, wenn man beispielsweise an ältere Dateiformate gerät, die Strings in 8 Bit-Darstellung enthalten.



Für die Umwandlung zwischen verschiedenen Textrepräsentationen steht Ihnen die (abstrakte) .NET-Klasse `System.Text.Encoding` *mit ihrer statischen* `Convert()`*-Methode zur Verfügung. Die Ableitungen* `ASCIIEncoding`*,* `UnicodeEncoding`*,* `UTF7Encoding` *und* `UTF8Encoding` *dieser Klasse ermöglichen die Repräsentation einer bestimmten Codierung als Objekt und spielen speziell im Zusammenhang mit der Web-orientierten Programmierung eine gewisse Rolle.*

Tabelle 7.5: Der Datentyp `char`

C#-Typ	CTS-Name	Wertebereich
`char`	`System.Char`	Alle Unicode-Zeichen; `'\u0000'` bis `'\uffff'`

Der C#-Compiler akzeptiert drei verschiedene Schreibweisen für `char`-Literale nach dem Muster der folgenden Codezeilen:

```
char c1 = 'A'        // Zeichen in Hochkommata
char c2 = '\x41'     // Hexwert in Hochkommata
char c3 = '\u0041'   // Unicode-Wert in Hochkommata
```

Die beiden Präfixschreibweisen unterscheiden sich insofern, als nach dem Präfix `\u` immer eine vierstellige Hexadezimalzahl stehen muss, während die Stellenanzahl nach dem Präfix `\x` zwischen eins und vier sein kann. Beachten Sie, dass `char`-Literale in einfache Hochkommata eingeschlossen werden, Strings jedoch in Anführungszeichen.

Von C/C++ sind eine Reihe von Escape-Sequenzen für eine Reihe häufig verwendeter Sonderzeichen bekannt, die C# ohne Änderung übernimmt. Abbildung 7.6 gibt einen Überblick.

Tabelle 7.6: Escape-Sequenzen für Zeichenkonstanten

Escape-Sequenz	Unicode-Notation	Zeichen
`\'`	`'\u0027'`	Hochkomma
`\"`	`'\u0022'`	Anführungszeichen
`\\`	`'\u005C'`	Umgekehrter Schrägstrich
`\0`	`'\u0000'`	Null (Unicode-Wert 0)

Escape-Sequenz	Unicode-Notation	Zeichen
\a	'\u0007'	Piepton (Bell; Alert)
\b	'\u0008'	Rückschritt (Backspace)
\f	'\u000C'	Seitenvorschub (Formfeed)
\n	'\u000A'	Zeilenvorschub (Linefeed)
\r	'\u000D'	Wagenrücklauf (Carriage Return)
\t	'\u0009'	Horizontaler Tabulator
\v	'\u000B'	Vertikaler Tabulator

Tabelle 7.6: Escape-Sequenzen für Zeichenkonstanten (Forts.)

Auch wenn C# einen im Vergleich zu C/C++ sehr eleganten Umgang mit Strings ermöglicht, gibt es Situationen, in denen der Datentyp `char` die bessere Wahl ist. Der folgende Auszug aus dem Beispielprojekt *Hex-Wandlung* zeigt eine Routine, die einen `char`-Wert in hexadezimaler Schreibweise als String zurückliefert:

```
// Liefert Unicode-Repräsentation eines Unicode-Zeichens
static string ToHexString(char c)
{
  char [] HexDigits = "0123456789ABCDEF".ToCharArray();
  string s = "";
  for (int i = 3 ; i>=0; i--)
    s += HexDigits[c >> i * 4 & 0xf];
  return "\\u" + s;
}
```

Der etwas trickreiche Code, der sich als Schnittmuster für die Umrechnung in alle möglichen Notationen anbietet (wie wäre es mit der von C# nicht mehr unterstützten oktalen oder binären Schreibweise?), nutzt hier nicht nur die Verwandtschaft zwischen `char[]` und `string`, sondern bedient sich auch der impliziten Typumwandlung nach `ushort`.

Leichter ans Ziel gelangt man natürlich mit den inhärenten Mitteln der Methode `ToString()`:

```
static string hex(char c)
{
  return "\\u" + ((ushort)c).ToString("x4");
}
```

Vom C#-Compiler hier eine implizite Umwandlung nach `ushort` zu erwarten, wäre vermessen, zumal natürlich auch `char` eine Methode namens `ToString()` besitzt.

7.1.2 Einfache Verweistypen

C# kennt zwei vordefinierte Verweistypen: `object` und `string`. Ersterer ist die Mutter aller Objekte: Von diesem Typ sind alle anderen .NET-Datentypen direkt oder indirekt abgeleitet. Letzterer benimmt sich ganz und gar nicht wie ein Verweistyp, wird aber als solcher behandelt.

Zeichenketten – string

In der C/C++-Tradition zählt der Datentyp `string` in seiner expliziten Implementierung als `char[]` nicht nur ganz klar zu den komplexen Typen, er zählt auch zu den Hauptärgernissen des Programmierers – nicht erst, seit es Unicode gibt. Sprachen mit einem dedizierten Laufzeitsystem wie Basic haben es da schon immer einfacher gehabt, denn für eine verkapselte Implementierung ist nun einmal eine im Hintergrund wirkende Freispeicherverwaltung unverzichtbar. Dass dies auch mit einem C-Derivat ohne weiteres machbar ist, hat Java eindrucksvoll demonstriert. Mit dem .NET-Framework als sprachübergreifende Laufzeitumgebung können nun alle Sprachen mit .NET-Anbindung von einem zu den .NET-Basisklassen gehörigen Stringdatentyp profitieren. Allen voran natürlich auch C#, das sich nicht mit Fragen der Abwärtskompatibilität herumschlagen muss, sondern gewissermaßen aus dem Vollen schöpfen und es sich sogar leisten kann, einen Datentyp wie `string` in die Phalanx der elementaren Datentypen einzureihen.

Allerdings hat `string` etwas von einem geschenkten Gaul, und dem schaut man bekanntlich ja eher nicht (so genau) ins Maul. Als Umbenennung von `System.String` ist `string` direkter Abkömmling von `System.Object` und nicht von `System.ValueType`. Wäre da nicht die höchst offizielle Deklaration der Klasse `System.String`:

```
public sealed class String : IComparable, ICloneable, IConvertible, IEnumerable
```

könnte man in der Tat vermuten, dass `string` als `struct`-Klasse und nicht als `class`-Klasse implementiert ist.

Damit ist die Klasse `string` vom Prinzip her erst einmal ein Verweistyp, wenngleich sich die Implementierung bemüht, im praktischen Umgang den Eindruck zu erwecken, ein String sei nichts anderes als eine Ganzzahl oder ein Zeichen – also kurz gesagt, ein Werttyp:

- Die Klasse steht nicht als Basisklasse für eigene Ableitungen zur Verfügung.
- Stringzuweisungen erzeugen im Allgemeinen echte Wertkopien.
- `string`-Parameter werden standardmäßig »by value« übergeben.

➡ Die Vereinbarung und Initialisierung von Stringvariablen hat dieselbe Syntax wie für Werttypen (vgl. `byte`, `int` etc.). Daneben bietet `string` aber auch verschiedene Konstruktoren an, sodass auch eine Initialisierung mit `new`-Operator erfolgen kann, wie für Verweistypen üblich.

Mit dieser Phänomenologie lässt sich `string` konzeptuell als Werttyp begreifen, auch wenn seine Abstammung dies nicht direkt widerspiegelt: `string` ist also eine Bruderklasse von `System.ValueType` – und keine Ableitung.

C#-Typ	CTS-Name	Wertebereich
`string`	`System.String`	`null`; `""`; Folgen von Unicode-Zeichen, deren Länge noch durch einen `int`-Wert beschreibbar ist.

Tabelle 7.7: Der Datentyp `string`

Die Ausstattung der Klasse `string` ist beachtlich. Selbst VB-Programmierer, die in dieser Hinsicht einiges gewohnt sind, dürften über die Fülle der Methoden staunen, von denen viele zudem noch in mehreren überladenen Varianten existieren (vgl. Abbildung 7.2).

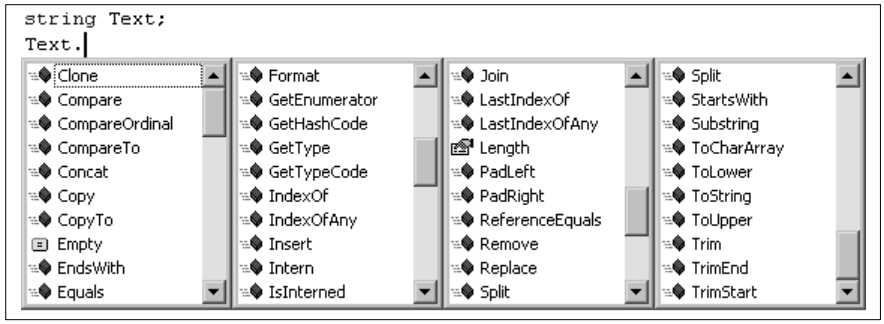


Abbildung 7.2: Die Methoden von `System.String`, wie von VS.NET interaktiv präsentiert (Bildmontage)

Ein großer Teil dieser Methoden hat eine lange Tradition im Bereich der Stringbibliotheken. Einige, wie `StartsWith()`, `EndsWith()` oder `CopyTo()`, sind hingegen echte Debütanten und demonstrieren, wie sehr sich .NET um Komfort bemüht. Referenzinformationen und die Bedeutung der Parameter finden Sie in der Online-Dokumentation. Die jeweiligen Aufgaben und Rollen der Parameter erschließen sich allerdings in den meisten Fällen bereits, indem man einen genaueren Blick auf die Parameterliste des von Visual Studio .NET eingeblendeten Funktionsprototypen wirft. Die Namen der Parameter sind meist sprechend genug. (Eine tabellarische Charakterisierung der einzelnen Methoden finden Sie in Kapitel 11, »Einsatz der .Net-basisklassen«, auf Seite 361ff)

Etwas verwundern dürfte das Vorhandensein von `ToString()`. Diese Methode zählt wie bei jeder Klasse zum Erbe der Basisklasse `object` und liefert für `string`

nichts weiter als den Wert an sich. Außer der überladenen Variante, die vom Prinzip her die Tür zu anderen Repräsentationen offen hält, dürfte diese Methode jenseits der Typreflexion und der generischen Anwendung in Auflistungen mit gemischten Elementtypen keine sinnvolle Anwendung haben.

In C# werden Stringliterale wie üblich in doppelten Anführungszeichen geschrieben:

```
string Text = "Das ist ein Text";
```

Für die Notation von Sonderzeichen (umgekehrter Schrägstrich, CR, LF, etc.) gelten generell die Regeln des Datentyps `char` – mit dem Unterschied, dass bei Strings auch einzelne Zeichen in doppelten Anführungszeichen zu notieren sind:

```
string CR = "\r";
string Pfad = "c:\\ ";     // der Backslash wird häufig falsch notiert!
string URL = "http://198.162.2.1:88";   // Kommentar erst ab hier!
string fn = "c:\\\u0045igene Dateien\\test.dat";  // mit Unicode-Notation
```

Um einen String aus mehreren Stringliteralen zusammenzusetzen, verlangt C# den +-Operator:

```
string URL = "http://" +
             "198.162.2.1:88";
```

Ohne ein Pluszeichen gestattet es der C#-Compiler nicht, Stringliterale auf mehrere Quellcodezeilen zu verteilen. Auch das Hintereinandersetzen von Stringliteralen ohne Operator à la C/C++ oder Delphi akzeptiert er nicht.

Für besondere Fälle ist in C# ein Präfixzeichen für Stringliterale in Form des Klammeraffen @ definiert. Es hat die Bedeutung, dass alle Zeichen mit Ausnahme des doppelten Anführungszeichens innerhalb der Zeichenkette als literale Zeichen behandelt werden. Das gilt auch für Zeilenendezeichen im Quelltext, wenn ein Stringliteral über mehrere Zeilen hinweg notiert ist. Es gibt somit keine Möglichkeit, innerhalb solcher Stringliterale die Unicode-Notation zu verwenden oder ein doppeltes Anführungszeichen zu notieren. Hier ein Beispiel:

```
string s = @" Der Compiler versteht zwar CR's, jedoch die Sequenz \u0041
           nicht als Unicode-Notation!";
System.Console.WriteLine(s);
```

Die Konsolenausgabe lautet somit:

```
Der Compiler versteht zwar CR's, jedoch die Sequenz \u0041
           nicht als Unicode-Notation!
```

Man beachte, dass führende Leerzeichen hier gleichfalls ihren Weg in den String finden.

string vs. StringBuilder

Es ist nicht alles Gold was glänzt. Die an sich sehr komfortabel implementierte Klasse `string` hat einen Nachteil: Ihre Effizienz lässt zu wünschen übrig, wenn es um die exzessivere Manipulation (Verketten, Anschneiden usw.) von Strings geht. Das liegt daran, dass ein `string`-Objekt als solches unveränderlich ist und Zuweisungen, die im Zusammenhang mit einer Stringoperation stehen, im Allgemeinen *echte* Wertkopien generieren. Der Hintergrund ist folgender: Das .NET-Laufzeitsystem verwaltet speziell für Strings einen internen Pool, in dem nur eindeutige Stringwerte gespeichert werden. Ist ein als Rechtswert berechneter Stringwert darin bereits vorhanden, erzeugt die Zuweisung schlicht eine weitere Referenz auf den bereits bestehenden Wert, andernfalls wird der Wert in den Pool aufgenommen, und die Zuweisung erzeugt eine echte Wertkopie. (Ein Codebeispiel, das dies deutlich macht, finden Sie im Abschnitt »System.String«, Seite 363) In der Praxis heißt dies aber, dass bei nahezu jeder Stringoperation letztlich ein neues `string`-Objekt generiert und mit einer Kopie des Rechtswertes instanziiert werden muss, es sei denn, der Wert ist schon bekannt.

Bei der kombinierten +=-Zuweisung müssen somit zwei Werte in einen dritten kopiert werden, der (falls noch nicht bekannt) seinerseits als neuer Wert im Pool landet, während der alte Wert (falls nicht anderweitig noch benutzt) verworfen werden muss. Ein `string`-Objekt mit einer Länge von 200.000 Zeichen für Zeichen aufzubauen, gerät damit schnell zum Staatsakt, wie das folgende Codebeispiel unerfreulich eindrucksvoll vor Augen führt – nämlich mit einer Laufzeit im Bereich von einigen Minuten:

```
string s = "";
Console.WriteLine("Beginn: {0}", System.DateTime.Now);
for (int x = 0; x < 200000; x++)
  s += " ";
Console.WriteLine("Ende  : {0}", System.DateTime.Now);
```

Es ist daher nicht verwunderlich, dass unter den .NET-Basisklassen in Form der Klasse `System.Text.StringBuilder` eine zweite Stringklasse zu finden ist, die insbesondere auf die Stringmanipulation hin optimiert ist und deren Operationen nicht nur Veränderungen auf der Ebene der Repräsentation bewirken, sondern auch wesentlich schneller über die Bühne gehen. Wie Bild 7.3 zeigt, wird ein `StringBuilder`-Objekt für gewöhnlich (es stehen sechs Konstruktoren zur Auswahl) unter Angabe eines Kapazitätswertes konstruiert, der den (anfänglichen) Speicherbedarf des Objekts einigermaßen gut vorgibt. Je besser dieser Wert ist, desto besser ist das Verhältnis von Laufzeit zu Speichernutzung. (Allzu viele Gedanken um den »richtigen« Kapazitätswert muss man sich allerdings nicht machen: Das zuvor angeführte

Horrorbeispiel ist mit einem `StringBuilder`-Objekt auch dann in Sekundenbruchteilen beendet, wenn das Objekt eine Anfangskapazität von 16 hat.) Der Kapazitätswert kann nachträglich (manuell oder automatisch) vergrößert werden, falls dies erforderlich ist. Das `StringBuilder`-Äquivalent der `string`-Operation += ist die `Append`-Methode. Sie versucht, das `StringBuilder`-Objekt zunächst im bestehenden Puffer zu »verlängern«. Falls der Puffer nicht ausreicht, vergrößert sie (automatisch) dessen Kapazität und »verlängert« dann erst den String. (Wer sich einmal mit der STL bzw. ATL für C++ beschäftigt hat, wird `StringBuilder` als Containerklasse wiedererkennen.)

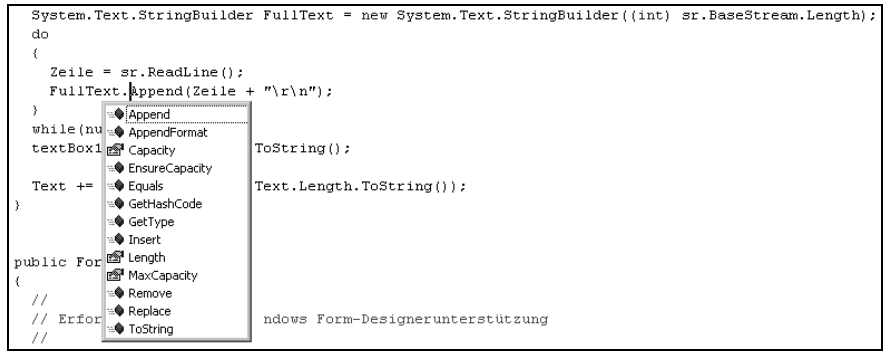


Abbildung 7.3: Angebot eines Objekts der Klasse `StringBuilder`

Codebeispiel – TextReader

Das folgende Beispielprogramm, die Windows-Anwendung *TextReader*, liest eine Textdatei zeilenweise ein, um sie in einem auf dem Formular befindlichen Multiline-Textfeld auszugeben. Ein Mausklick in den Clientbereich des Formulars öffnet den Standarddialog DATEI ÖFFNEN zur Auswahl der Textdatei. Der Zugriff auf die Datei erfolgt über ein `StreamReader`-Objekt.

Zum Vergleich der beiden Stringklassen enthält das Programm zwei Leseroutinen, von denen die eine, `ShowTextFileString()`, konventionell mit dem Datentyp `string` und die andere, `ShowTextFileStringBuilder()`, mit einem `StringBuilder`-Objekt hantiert. Je länger die zu lesende Textdatei, desto auffälliger wird der Laufzeitunterschied – wobei allerdings zu beachten ist, dass die Länge der `Text`-Eigenschaft von Textfeldern unter Windows 98 und ME auf 64 KByte beschränkt ist. Die `string`-Version benötigt für eine Textdatei mit 120 KByte auf einem 1 GHz-System unter W2K bereits ein paar Minuten. Das »Umschalten« zwischen den beiden Versionen geschieht schlicht durch Auskommentieren des jeweiligen Aufrufs in der Methode `Form1_MouseDown()`:

```
using System;
using System.IO;
using System.Windows.Forms;
```

```
public class Form1 : System.Windows.Forms.Form
{
  private System.Windows.Forms.TextBox textBox1;  // Textfeld für Ausgabe
  private System.Windows.Forms.OpenFileDialog openFileDialog1;
  ...
  // Öffnet Textdatei und ordnet diese dem Stream sr zu
  public bool OpenTextFile(out StreamReader sr)
  {
    openFileDialog1.Filter = "Textdateien (*.txt)|*.txt";
    try
    {
      openFileDialog1.ShowDialog();
      sr = new StreamReader(openFileDialog1.OpenFile());
      return true;
    }
    catch
    {
      sr = null;
      return false;
    }
  }

  // Gibt Inhalt des StreamReader-Objekts sr zeilenweise in Textfeld aus
  public void ShowTextFileString(StreamReader sr)
  {
    textBox1.Text=null;
    string Zeile;
    while(null != (Zeile = sr.ReadLine()))  // zeilenweise bis Dateiende
      textBox1.Text += (Zeile + "\r\n");
  }

  // Gibt Inhalt des StreamReader-Objekts sr zeilenweise in Textfeld aus
  public void ShowTextFileStringBuilder(StreamReader sr)
  {
    textBox1.Text=null;
    string Zeile;
    System.Text.StringBuilder FullText =
         new System.Text.StringBuilder(((int) sr.BaseStream.Length));
    while(null != (Zeile = sr.ReadLine()))  // zeilenweise bis Dateiende
      FullText.Append(Zeile + "\r\n");
    textBox1.Text = FullText.ToString();

    Text += ("  " + textBox1.Text.Length.ToString());
  }

...

  [STAThread]
  static void Main()
  {
    Application.Run(new Form1());
  }
```

```
  // Textdatei öffnen und anzeigen
  private void textBox1_MouseDown(object sender,
     System.Windows.Forms.MouseEventArgs e)
  {
    StreamReader sr;
    if (OpenTextFile(out sr))
      ShowTextFileStringBuilder(sr);
//     ShowTextFileString(sr);
  }
  // Passt Multiline-Textfeld an
  public void Form1_Resize(object sender, System.EventArgs e)
  {
    textBox1.Size = ClientSize;
  }
  private void Form1_Load(object sender, System.EventArgs e)
  {
    // Multiline-Textfeld in Clientbereich einpassen
    textBox1.Left = textBox1.Top = 0;
    textBox1.Size = ClientSize;
    textBox1.Multiline = true;
    textBox1.ScrollBars = System.Windows.Forms.ScrollBars.Vertical;
    textBox1_MouseDown(null, null);
  }
}
```

Das Programmgerüst wurde per Assistent erzeugt; hier sind nur die bearbeiteten Codeteile abgedruckt. Der Initialisierungscode in `Form1_Load()` passt das Textfeld in den Clientbereich ein, setzt die notwenigen Attribute und sorgt dafür, dass bei einem Klick eine der beiden Dateileseroutinen aufgerufen wird. Bild 7.4 zeigt eine Ausgabe des Programms nach Auswahl einer Textdatei und einem darauffolgenden Klick in das Formular.

Der allgemeine Verweistyp – object

In objektorientierten Sprachen ist es üblich, dass sich unter den integrierten Datentypen einer findet, der als Basisklasse oder zumindest als eine Art Wurzel für alle anderen (Objekt-)Datentypen fungiert. C# hält sich nicht nur an diese Konvention, sondern erweist sich sogar als äußerst vorbildlich. Die Sprache stellt mit dem Datentyp `object` (alias `System.Object`) wirklich einen perfekten Allroundtyp bereit, dem Werte beliebigen Typs zugewiesen werden können. Der C#-Compiler unterstützt dafür sogar die implizite Typumwandlung (wegen der Vererbungsbeziehung).

Für *Verweistypen* ist das im Allgemeinen keine große Sache, da die entsprechende Objektvariable bekanntlich ja ohnehin nur die Adresse eines auf dem Heap gelegenen Wertes beherbergt, und die lässt sich ja einfach kopieren. Zuweisungen in der umgekehrten Richtung laufen nach dem gleichen Schema, bedürfen aber einer expliziten Typumwandlung in den »richtigen Typ«, wobei hier alle Typen erlaubt sind, die auf der Ableitungslinie (Ahnenreihe) zwischen `object` und dem Ausgangstyp gelegen sind.

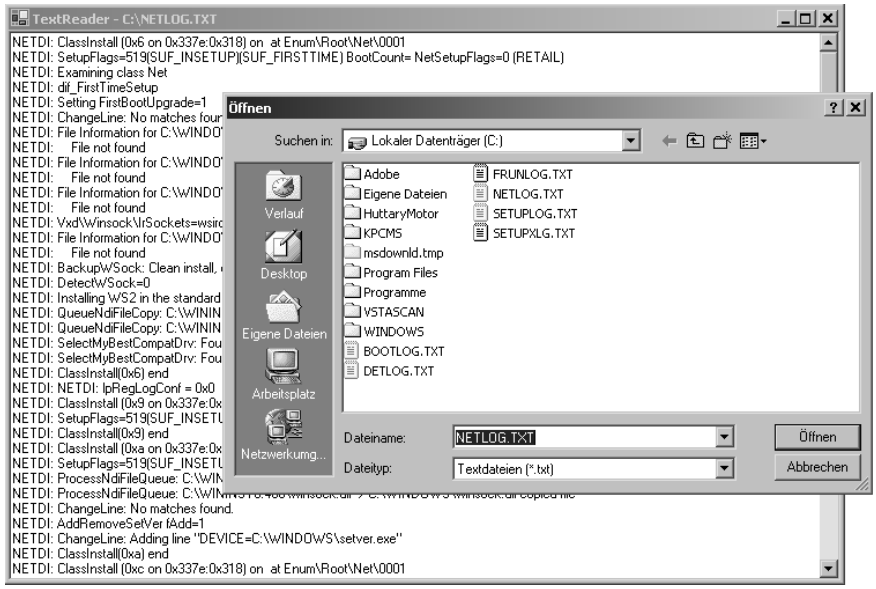


Abbildung 7.4: Das Programm TextReader gibt den Inhalt einer Textdatei in einem Textfeld aus. Ein Mausklick in das Formular aktiviert den Standarddialog ÖFFNEN.

So interessant das klingt, für die praktische Programmierung würde dieses Hin und Her kaum Vorteile bringen, wäre da nicht die Geschichte mit der *Polymorphie* und den virtuellen Methoden, die die objektorientierte Programmierung erst zu dem gemacht haben, was sie heute ist. Die Situation ist mit einem kleinen Codebeispiel schnell skizziert: Die Klasse `object` stellt eine Implementierung der Methode `ToString()` bereit und vererbt diese als virtuelle Methode. Bei Aufruf dieser Methode liefert diese den Typnamen des jeweiligen Objekts(!). Also:

```
class MyClass     // von object abstammend!
{
}
...

// irgendwo in einer Methode ...
MyClass my = new MyClass();
object o = my;              // implizite Typumwandlung
string s1 = my.ToString();  // s1 = "MyClass"; ererbte Variante
string s2 = o.ToString();   // s1 = "MyClass"; ererbte Variante(!)
```

»Überraschenderweise« liefert dieser Aufruf auch »aus der Sicht« von `object` dasselbe Ergebnis, und nicht `"object"`, wie man vielleicht im ersten Moment erwarten würde. `System.Object.ToString()` führt seinerseits einen `GetType()`-Aufruf durch, der den Typ des jeweiligen Objekts liefert (und sinnvollerweise nicht überschrieben werden kann).

Ähnlich sieht die Sache aus, wenn eine Klasse ihr Erbe »ausschlägt« und eine eigene Variante von `ToString()` implementiert:

```
public class Vektor2D
{
  public double x, y;
  public Vektor2D(double new_x, double new_y)
  {
    x = new_x;
    y = new_y;
  }
  public double Abs()
  {
    return System.Math.Sqrt(x*x+y*y);
  }
  public override string ToString()
  {
    return "("+x.ToString()+", "+y.ToString()+")";
  }
}
...
// irgendwo in einer Methode ...
Vektor2D v2D = new Vektor2D(10, 20);
object o = v2D;                // implizite Typumwandlung
string s1 = v2D.ToString();    // s1 = (10, 20);  wie erwartet!
string s2 = o.ToString();      // s2 = (10, 20);  ToString() ist virtuell!
Vektor2D v2d = (Vektor2D) o;   // hier ist eine Typumwandlung notwendig
```

Hier kommt in beiden Fällen die `ToString()`-Variante von `Vektor2D` zum Aufruf und nicht, wie man vielleicht annehmen könnte, im zweiten Fall die Variante von `object`. Der Trick bei den virtuellen Methoden ist, dass immer die Methode des tatsächlich dahinter stehenden Objekts zum Aufruf kommt, auch wenn die Objektvariable den Typ einer Basisklasse trägt.

Diese Aussage gilt in der Tat nur für Methoden, die als `virtual` bzw. `override` vereinbart sind. Das folgende, etwas komplexere Beispiel aus dem Beispielprojekt *ToStringInherit* zeigt, dass dieses Verhalten nicht immer vorausgesetzt werden kann. Betrachten wir die Implementierung einer Klasse `Vektor3D`, die auf die Implementierung von `Vektor2D` aufsetzt:

```
public class Vektor3D: Vektor2D
{
  public double z;
  public Vektor3D(double x, double y, double new_z): base(x, y)
  {
    z = new_z;
  }
  public new double Abs()
  {
    return System.Math.Sqrt(base.x * base.x + base.y * base.y + z*z);
  }
  public override string ToString()
  {
    return "("+x.ToString()+", "+y.ToString()+", "+z.ToString()+")";
  }
}
```

```
...
// irgendwo in einer Methode ...
Vektor3D v3D = new Vektor3D(0, 3, 4);
Vektor2D v2D = v3D;                    // Basisklassenvariable!
object o = v3D;
string s1 = v3D.Abs().ToString();  // s1 = " 5"; x, y, z gehen ein
string s2 = v2D.Abs().ToString();  // s2 = " 3"; nur x und y gehen ein!
// string s3 = o.Abs().ToString(); // Das geht überhaupt nicht!
```

Da `Abs()` keine virtuelle Methode ist, ruft jede Variable die zu ihrem Vereinbarungstyp gehörige Variante auf.

Bei Werttypen eröffnet sich ein ganz anderes Problemfeld: Bekanntlich repräsentiert ein Werttyp ein konkretes Bytefeld auf dem Stack, während `object` wie gesagt nur die Adresse für ein Objekt speichert. Trifft der Compiler nun auf eine Zuweisung der Art:

```
object o = 10;
```

findet ein so genanntes BOXING statt. Das heißt, die implizite Typkonvertierung sorgt dafür, dass die Speicherung des Rechtswerts (hier also des Wertes 10) in Form eines primitiven Objekts auf dem Heap erfolgt und die Adresse dieses Gebildes schließlich in der Objektvariable `o` landet. Die umgekehrte Operation ist das UNBOXING. Hier wird eine Objektvariable (hier des Typs `object`) per expliziter Typumwandlung in einen Werttyp verwandelt und dann zugewiesen:

```
object o = 10;
int i = (int) o;
```

Auch gilt grundsätzlich, dass der Ausgangtyp (Typ vor dem Boxing) mit dem Zieltyp (Typ nach dem Unboxing) verträglich sein muss. Typverengung fällt auch dann *nicht* in die Kategorie »verträglich«, wenn eine entsprechende explizite Typumwandlung unterstützt wird:

```
object o = 100L;
byte b = (byte) o;     // Laufzeitfehler!
```

Um dieses an sich sinnvolle Konstrukt zu retten, muss man etwas mehr Aufwand bei der Typumwandlung betreiben:

```
byte b = (byte) (long) o;     // mühsam nährt sich das Eichhörnchen ...
```

Völlig außer Diskussion ist die Typüberschreitung:

```
object o = 10;
string s = (string) o; // Laufzeitfehler
```

Code wie dieser führt in jedem Fall zu einem Laufzeitfehler (was ja auch kein Wunder ist).

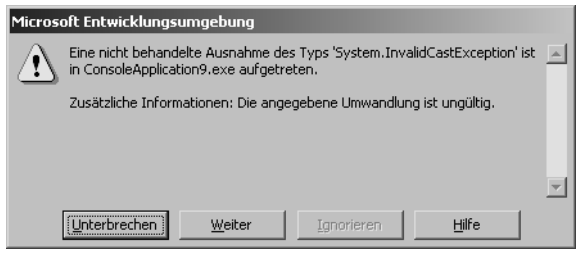


Abbildung 7.5: Boxing- und Unboxing-Typen sind nicht verträglich.

7.2 Komplexe Datentypen

Zu den komplexen oder auch benutzerdefinierten Datentypen einer Programmiersprache zählen alle Typen, die nicht zu den integrierten Datentypen gerechnet werden. Kennzeichnend für komplexe Datentypen ist, dass sie durch Umbenennung (Aufzählungstypen, Delegaten) oder durch Aggregatbildung (Strukturen, Klassen, Arrays, Schnittstellen) aus einfacheren und damit letztlich aus den integrierten Datentypen hervorgehen.

Analog zu den integrierten Datentypen wird auch bei den komplexen Datentypen zwischen Wert- und Verweistypen unterschieden.

7.2.1 Komplexe Werttypen

C# stellt die sprachlichen Mittel für zwei unterschiedliche Arten von komplexen Werttypen bereit. Bei der einen Art handelt es sich um Aufzählungstypen, die im Wesentlichen die Definition von Konstantensammlungen und endlichen Mengen ermöglichen. Die andere Art sind so genannte `struct`-Klassen, die einerseits die klassischen Strukturen umfassen (also nackte Aggregate von Werttypen), andererseits aber auch Strukturen mit prozeduralem Anhang im Sinne der objektorientierten Programmierung, wie sie beispielsweise von C++ her bekannt sind.

Aufzählungstypen – enum

AUFZÄHLUNGSTYPEN, neudeutsch auch ENUMERATIONEN genannt, sind aus der Programmierung schon lange nicht mehr wegzudenken. Ob sie unbedingt zu den komplexen Datentypen zu rechnen sind, darüber kann man natürlich streiten. Schließlich sind Aufzählungstypen vom Prinzip her nichts anderes als Umbenennungen eines Ganzzahltyps – bei C# also Umbenennungen von `int`.

Mit C# sind Aufzählungstypen zu vollwertigen Datentypen avanciert. Sie eignen sich bestens zur Modellierung von Wertebereichen (bzw. allgemeiner: Mengen) mit einer festen Anzahl von Elementen. Damit sind sie ein probates Mittel für

- *die Benennung von diskreten Zuständen* – beispielsweise für die unterschiedlichen Betriebsmodi einer Komponente im Zusammenhang mit Zustandsvariablen (Stichwort: endlicher Automat)
- *die Benennung von Kategorien* – beispielsweise zur Dimensionsbenennung in mehrdimensionalen Arrays
- *die Benennung von Fällen* – zur Fallbenennung in symbolischen Fallunterscheidungen (`switch`)
- *die Vereinbarung von Konstanten*
- *Bitvektoren*, die aus einzelnen Bitgruppen aufgebaut sind.

Die Syntax für die Vereinbarung eines gewöhnlichen Aufzählungstyps lautet:

```
[Attribute] [Modifizierer] enum MeinAufzählungstyp [:Basistyp]
{
    Element1 [= Ganzzahlwert] [,
    ...]
}
```

Die Syntax erlaubt es, jedem einzelnen Element explizit eine Ganzzahl als Wert zuzuordnen, die auch negativ sein darf. Zudem ist es legitim, mehreren Elementen den gleichen Wert zuzuordnen. Erfolgt eine solche Zuordnung nicht, erhält das erste Element standardmäßig den Wert 0 und nachfolgende Elemente jeweils einen um 1 höheren Wert. Die implizite Wertzuweisung orientiert sich immer an der jeweils zuletzt angegebenen, expliziten Wertzuweisung.

Um einen Aufzählungstyp als Bitvektor zu definieren, dessen Wertebereich explizit als Oder-Kombination der Bitmuster der aufgezählten Elementwerte definiert ist, ist das `[Flags]`-Attribut erforderlich (etwas mehr über Attribute finden Sie ab Seite 245). Dieses Attribut wirkt sich auf die Wertdarstellung der `ToString()`-Methode aus. Standardmäßig liefert die Methode nur für solche Aufzählwerte einen Konstantenbezeichner, für die eine Konstantendefinition vorliegt, und ansonsten schlicht eine Dezimalzahl. Mit dem `[Flags]`-Attribut hingegen drückt sie den Wert eines Bitvektors nach Möglichkeit als Kombination der Bezeichner der in der Aufzählung für die einzelnen Bitfelder vereinbarten Konstantenwerte aus:

```
enum MyEnum {Eins = 1, Vier = 4};
[Flags] enum MyBitVektor {Eins = 1, Vier = 4};
...
Console.Write(MyEnum.Eins.ToString());       // Ausgabe: "Eins"
Console.Write(MyEnum.Vier);                   // Ausgabe: "Vier"
Console.Write(MyEnum.Eins | MyEnum.Vier);     // Ausgabe: "5"
Console.Write(MyBitVektor.Eins |
              MyBitVektor.Vier);              // Ausgabe: "Eins, Vier"
```

Ein prominentes Beispiel einer `[Flags]`-Aufzählklasse ist `System.Windows.Forms.Keys`. Sie definiert die Konstanten für jede Taste der Tastatur und ermöglicht die Beschreibung des Tastaturzustands.

Aufzählungstypen als vollwertige Datentypen

Das Besondere an den Aufzählungstypen von C# ist, dass sie *vollwertige Datentypen* sind, denen standardmäßig der Datentyp `int` zugrunde liegt, sofern nicht ein anderer Ganzzahltyp – beispielsweise `long` – als *Basistyp* vereinbart ist. Im Einzelnen heißt das:

- Aufzählungstypen dürfen wie jeder andere Datentyp in Datenfeld-, Variablen- und Methodendeklarationen verwendet werden.
- Der Compiler überwacht die Typkonformität von Aufzählungstypen ebenso streng wie die eines jeden anderen Datentyps. (VB6- und C/C++-Programmierer sind hier erheblich mehr Toleranz gewohnt.)
- Für Aufzählungstypen sind die bitweisen Operatoren `|`, `&`, `^` und `~` definiert. Die Aufzählungselemente lassen sich also zu Bitvektoren und Bitmasken kombinieren.
- Es gibt keine implizite Typumwandlung zwischen Aufzählungstypen und Ganzzahltypen – auch nicht mit `int`. Explizite Typumwandlungen mit dem Klammeroperator sind hingegen jederzeit möglich. *Achtung*: Weder der Compiler noch das Laufzeitsystem wachen darüber, dass der deklarierte Wertebereich eines Aufzählungstyps bei der Wertzuweisung an eine Variable eingehalten wird.
- Der Sichtbarkeitsbereich eines Aufzählungstyps wird durch den Modifizierer (`private`, `public` etc.) bestimmt. Damit ist es beispielsweise möglich, eine Konstantensammlung über eine `public`-Deklaration zu exportieren und somit auch für andere Klassen, Module und Namensbereiche sichtbar zu machen. (C/C++-Programmierer werden hier aufatmen, weil das dort notwendige Einbinden von Konstantensammlungen über Header-Dateien schon immer eine lästige und höchst fehlerträchtige Angelegenheit war.)
- Der Wertebereich für die Initialisierung von Elementen ist auf den Wertebereich des Datentyps `int` (bzw. *Basistyp*) beschränkt.
- Die `ToString()`-Methode eines `enum`-Werts liefert standardmäßig den Konstantenbezeichner bzw. eine Kombination aller elementaren Konstantenbezeichner eines Bitvektors als String. Dieses wirklich praktische Feature ist ein Resultat des inhärenten Typinformationsmechanismus von .NET und stellt ein echtes Novum gegenüber den Programmiersprachen VB, Delphi und C++ dar.

Aufzählungstypen aufzählen

In manchen Fällen kann es notwendig sein, die durch einen Aufzählungstyp definierten Konstanten aufzuzählen. Dies ist grundsätzlich möglich, wenn man weiß, dass C# `enum`-Typen auf die .NET-Basisklasse `System.Enum` zurückführt und diese eine Methode namens `GetValues()` für die Typreflexion unterstützt:

```
// alle Elemente der Aufzählung KnownColor ermitteln und
// in Array packen
Color[] colors;
Array  arr = System.Enum.GetValues(typeof(KnownColor));
colors = new Color[arr.Length];
for(int i = 0; i< arr.Length;i++)
{
  colors[i] = Color.FromKnownColor((KnownColor)arr.GetValue(i));
}
```

Codebeispiel – ConsoleEnumTest

Das Konsolenprogramm *ConsoleEnumTest* demonstriert den Umgang mit Aufzählungstypen in verschiedenen Situationen und unter verschiedenen Aspekten.

```
using System;
namespace ConsoleEnumTest
{
  class Class1
  {
    public enum MeinAufzählungsTyp
    {
      Rudi,             // = 0 (implizit)
      Arne,             // = 1 (implizit)
      Dieter = 13,
      Peter = 13,
      Walter            // = 14 (implizit)
    }
    static void Main(string[] args)
    {
      MeinAufzählungsTyp arne = MeinAufzählungsTyp.Arne;
      Class1 cl = new Class1();
      cl.Fallunterscheidung(MeinAufzählungsTyp.Rudi);  // Literal
      cl.Fallunterscheidung(arne);                     // Variable
      cl.Fallunterscheidung((MeinAufzählungsTyp) 13);  // Peter (Typumw.)
      // der nächste Fall lässt sich nicht ausdrücken
      Console.WriteLine("Das vierte Element ist " +
                        MeinAufzählungsTyp.Peter.ToString());
      cl.Fallunterscheidung(MeinAufzählungsTyp.Peter + 1);  // Walter!
    }
```

```
    private void Fallunterscheidung(MeinAufzählungsTyp azt)
    {
      switch (azt)
      {
        case MeinAufzählungsTyp.Rudi:
          Console.Write ("Erstes Element: ");
          break;
        case MeinAufzählungsTyp.Arne:
          Console.Write ("Zweites Element: ");
          break;
        case MeinAufzählungsTyp.Dieter:
          Console.Write ("Drittes Element: ");
          break;
    /*    case MeinAufzählungsTyp.Peter:  // Fehler, da Element doppelt !!
          Console.Write ("Viertes Element: ");
          break;
    */
        case MeinAufzählungsTyp.Walter:
          Console.Write ("Fünftes Element: ");
          break;
      }
      Console.Write(azt.ToString());
      Console.WriteLine(" hat den Wert " + ((int)azt).ToString());
    }
  }
}
```

Strukturen – struct-Datentypen

Das Schlüsselwort `struct` hat eine lange Tradition in der C-Programmierung. Es ermöglicht die Zusammenstellung komplexer benutzerdefinierter Datenstrukturen aus einfacheren Datenstrukturen und integrierten Datentypen. In diesem Sinne lässt sich jede Datenstruktur als Komposition integrierter Datentypen auffassen.

Die Syntax für die Vereinbarung eines Strukturtyps lautet:

```
[Modifizierer] struct MeinStrukturtyp [: Schnittstelle [, ...]]
{
  [[Modifizierer] const Datentyp KonstantenBez1 = Initwert [, ...];]
  ...]
  [[Modifizierer] Datentyp DatenfeldBez [, ...];]
   ...]
  [[Modifizierer] Datentyp MethodenBez([Parameterliste])
   { [[Anweisung;] ... ] }
  ...]
}
```

Die folgende Vereinbarung zeigt, wie man drei `double`-Werte zu einem eigenen Strukturtyp namens `Vektor3D` zusammenstellt.

```
public struct Vektor3D
{
  public double x, y, z;
}
...
Vektor3D v3D;      // Vektorstruktur anlegen
v3D.x = 1.945;
v2D.y = 1.4;
...
```

Dieser Datentyp kann nun seinerseits in weiteren `struct`-Vereinbarungen erscheinen:

```
public struct Line3D
{
  public Vektor3D startp;
  public Vektor3D endp;
}
```

Werttypen, deren Werte ja direkt auf dem Stack gehalten werden, bieten verschiedene Geschwindigkeitsvorteile gegenüber Verweistypen, die mit Belegungen auf dem Heap verbunden sind: Erstens kostet weder die Belegung noch die Freigabe auf dem Stack nennenswert Zeit, da der dafür nötige Verwaltungsaufwand bereits während der Kompilierung erbracht wird und sich zur Laufzeit schlicht auf vorberechnete Manipulationen des Stackzeigers beschränkt. Zweitens erfordert die byteweise Kopie eines auf dem Stack gelegenen Wertes nichts weiter als einen schnellen Blocktransfer unter Berücksichtigung der Längen von Ausgangs- und Zieltyp. (`Clone()`-Methoden von Verweistypen dauern erheblich länger.)

struct-Klassen

Neben der schlichten Typaggregation bietet C# nach dem Vorbild von C++ die Möglichkeit, `struct`-Datentypen Code in Form von Methoden zuzuordnen. Vom Ergebnis her erhält man damit nichts anderes als Klassen im Sinne der objektorientierten Programmierung. Im Gegensatz zu C++ sind bei C# allerdings die Fronten zwischen `class` und `struct` klar gezogen: `struct`-Klassen sind Werttypen, `class`-Klassen Verweistypen. (Nicht zuletzt auch aus diesem formalen Grund zählt `System.String` zu den Verweistypen.)

Hier ein einfaches Beispiel für eine `struct`-Klasse mit einer Methode:

```
public struct Vektor3D
{
  public double x, y, z;  // Inline-Initialisierung ist nicht möglich!
  public double Abs()
  {
    return System.Math.Sqrt( x*x + y*y + z*z);
  }
}
```

Der Umgang mit diesem Typ sieht genauso aus wie mit einem Werttyp, wobei der Compiler streng darüber wacht, dass alle Datenfelder der `struct`-Klasse vor der ersten Verwendung als Rechtswert initialisiert wurden.

```
Vector3D v3D1;
Vector3D v3D2;
v3D1.x = 10;
v3D1.y = 20;
v3D1.z = 30;
string s = "Der Betrag ist: " + v3D.Abs().ToString();  // 49.94...
v3D2 = v3D1;      // bitweise Kopie!

// Boxing/Unboxing
object o = v3D1
Vektor v3D2 = (Vektor) o;
```

Die Zuweisung bei `struct`-Werten bewirkt, dass der Linkswert eine bitweise Kopie des Rechtswerts erhält und die beiden Werte im weiteren Verlauf unabhängig sind. Auch das Boxing/Unboxing funktioniert nach dem für Werttypen gewohnten Schema.

Initialisierung und Konstruktor

Wie üblich, initialisiert der Compiler zwar Datenfelder von Objekten (`class`-Objekten), die einen `struct`-Typ tragen, automatisch mit 0. Lokale Variablen müssen hingegen explizit und vollständig initialisiert werden, bevor ein Einsatz als Rechtswert möglich ist:

```
struct MyStruct
{
  public int i;
  public int j;
}

class MyClass
{
  public MyStruct ms;      // automatisch mit 0 initialisiert
}
...
MyStruct ms1;
ms1.i = ms1.j = 0;      // erforderlich
MyStruct ms2 = ms1;     // Initialisierungsvereinbarung
MyClass mc = new MyClass();
MyStruct ms3 = mc.ms;   // mc.ms wurde bei Konstruktion initialisiert
```

Da eine Inline-Initialisierung von Datenfeldern nach dem Muster von Arrays nicht vorgesehen ist, bleiben drei Möglichkeiten für die Initialisierung:

- Zuweisung eines Wertes des gleichen Typs bei Vereinbarung oder zu einem späteren Zeitpunkt,

- elementweise, sofern die Datenfelder `public` sind,
- per Konstruktoraufruf.

`struct`-Instanzen erben einen parameterlosen Konstruktor, der nicht überschrieben werden kann und der die Datenfelder auf den Vorgabewert 0 setzt. Dieser Konstruktor, der `struct`-Datenfelder von `class`-Klassen automatisch ausführt, kann zur Initialisierung einer lokalen `struct`-Variablen explizit aufgerufen werden, um einen (unbenannten) initialisierten Wert des jeweiligen Typs zu erhalten und diesen zuzuweisen:

```
MyStruct ms4 = new MyStruct();
```

Der parameterlose Konstruktor kann aber nach Belieben durch eigene Konstruktoren mit mindestens einem Parameter überladen werden. Die Definition eines parameterlosen statischen Konstruktors zur Initialisierung statischer Datenfelder der Struktur ist ebenfalls zulässig.

Für die folgende Vektorstruktur ist ein eigener Konstruktor definiert, ein Operator sowie zwei *überladene* Varianten der Methode `Abs()`:

```
public struct Vektor
{
  double x, y, z;    // verkapselt!
  public Vektor(double new_x, double new_y, double new_z)
  {
    x = new_x;
    y = new_y;
    z = new_z;
  }
  static public Vektor operator + (Vektor v1, Vektor v2)  // Addition
  {
    return new Vektor(v1.x + v2.x, v1.y + v2.y, v1.z + v2.z);
  }
  public double Abs()
  {
    return System.Math.Sqrt( x*x + y*y + z*z);
  }
  static public double Abs(double x, double y, double z)
  {
    return System.Math.Sqrt( x*x + y*y + z*z);
  }
}
...
// Verwendung der Definition in irgendeiner Methode ...

// Initialisierung per Konstruktor
Vektor v1 = new Vektor(20, 0, 0);   // bitweise Kopie!
Vektor v2 = new Vektor(10, 0, 0);   // bitweise Kopie!

// Addition und Aufruf der Instanzenmethode Abs()
string s1 = (v1+v2).Abs().ToString();  // s1 = "30";
```

```
// new Vektor liefert wirklich einen fertigen struct-Wert
v1 = v2 + new Vektor();             // das geht!

// Aufruf der statischen Methode Abs
double d = Vektor.Abs(1, 1, 1);   // 1.73...

// Der Operator == ist für Vektor nicht vordefiniert/überladen
// if (v1 == v2)      // würde eine Fehlermeldung des Compilers erzeugen
```

Man beachte, dass die Initialisierung per Konstruktor nichts weiter als die bitweise Kopie von einem (unbenannten) Rechtswert in einen Linkswert ist. Da der C#-Compiler Datenfelder standardmäßig als `private` vereinbart, führt in diesem Beispiel kein Weg am Aufruf des Konstruktors vorbei.

class- vs. struct-Klassen

Das vorangehende Codebeispiel dürfte ansatzweise gezeigt haben, dass mit `struct`-Vereinbarungen durchaus ernstzunehmende Klassenimplementierungen zu verwirklichen sind, deren Vorteil vor allem in der Ausführungsgeschwindigkeit zu sehen ist. Anders als `class`-Objekte eignen sich `struct`-Objekte auch für die »By value«-Parameterübergabe in Funktionsaufrufen (Übergabe als echte Wertkopie). Wie das `Vektor`-Beispiel zeigt, werden sie daher häufig auch einfach als Zusammenfassung mehrerer Wert-Parameter benutzt

Im Vergleich zu `class`-Klassen unterliegen `struct`-Klassen allerdings gewissen Einschränkungen – insbesondere, was die Vererbung betrifft:

- `struct`-Klassen sind grundsätzlich implizit von `System.ValueType` abgeleitet – einer Klasse die ihrerseits direkt von `System.Object` abstammt.
- C# kennt keine Implementierungsvererbung für `struct`-Klassen. Sie können also weder selbst als Basisklasse verwendet noch von einer anderen Klasse (gleich ob `class`- oder `struct`-Klasse) abgeleitet werden. Gegen die Angabe von Schnittstellenklassen im Vererbungsteil spricht hingegen nichts. Eine Schnittstellenimplementierung auf der Basis von `struct`-Klassen ist also möglich.
- Es ist nicht möglich, virtuelle Methoden in einer `struct`-Klasse zu vereinbaren. Dieses Konzept hat auch nur im Zusammenhang mit Implementierungsvererbung einen Sinn.
- Der Standardkonstruktor einer `struct`-Klasse kann nicht überschrieben werden.
- Als Datenfelder für `struct`-Klassen sind Werttypen und Verweistypen zulässig, nicht aber Arrays. In `class`-Klassen sind auch Arrays erlaubt.
- Die für `class`-Klassen erlaubte Initialisierungsvereinbarung für Instanzfelder ist für `struct`-Klassen nicht möglich. Sie müssen demnach per

Konstruktor oder `public`-Zugriff initialisiert werden. Für statische Datenfelder ist die Initialisierungsvereinbarung hingegen in beiden Fällen möglich.

- Wird ein `struct`-Objekt einer Methode übergeben, erhält die Methode standardmäßig – wie für Werttypen üblich – eine bitweise Kopie des Objekts als Parameter (VB6-Programmierer aufgepasst). Bei umfangreichen Objekten kann dies eine erhebliche Belastung des Stacks und damit eine Vergeudung von Laufzeit bedeuten. Einen vernünftigen Kompromiss stellt in diesem Fall die Vereinbarung als `ref`-Parameter (mehr darüber im folgenden Kapitel) dar. Die Methode sollte dann aber keine (bleibenden) Änderungen am Objekt vornehmen, da diese aufruferseitig als Seiteneffekte sichtbar werden. Für diese Parameter wäre ein *const*-Modifizierer schön, den es aber in C# so nicht gibt.

- Wie für Werttypen üblich, findet für `struct`-Objekte ein Boxing statt, wenn sie einem Verweistyp zugewiesen werden. Das ist beispielsweise der Fall, wenn eine Speicherung in einer Collection-Klasse wie `ArrayList` erfolgt. Umgekehrt ist dann natürlich auch ein Unboxing erforderlich.

  Für `struct`-Objekte gibt es, wie für Werttypen generell, keinen Destruktor. Auch ist kein `Dispose()`-Mechanismus verfügbar.

Standardoperatoren

Für die einfachen Datentypen ist eine bunte Vielfalt von Standardoperatoren definiert, die weitgehend unverändert von C/C++ übernommen wurden. Vier davon stehen nur für `unsafe`-Code zur Verfügung, da sie das Typmanagement von .NET umgehen. Tabelle 7.8 gibt einen Überblick über die Operatoren; Tabelle 7.9 zeigt die Prioritäten, die die Operatoren untereinander haben.

In C# ist es möglich, die meisten Operatoren auch für `class`- und `struct`-Typen zu überladen und so mit einer für den Typ passenden Bedeutung zu versehen. Details zum Überladen von Operatoren finden sich im Abschnitt »Methoden« ab Seite 257.

Tabelle 7.8: Überblick über die C#-Standardoperatoren

Operatorkategorie	Operatoren
Arithmetisch	`+`, `-`, `*`, `/`, `%`
Boolesch (Not, And, Or)	`!`, `&&`, `||`, `true`, `false`
Bitweise logisch (And, Or, Xor, Not)	`&`, `|`, `^`, `~`
Stringverkettung	`+`
Inkrement, Dekrement	`++`, `--`
Verschieben bitweise	`<<`, `>>`

Tabelle 7.8: Überblick über die C#-Standardoperatoren (Forts.)

Operatorkategorie	Operatoren
Vergleich (relational)	==, !=, <, >, <=, >=
Zuweisung, kombinierte Zuweisung	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
Zugriff auf Element	.
Indizierung (Arrays und Indexer)	[]
Typumwandlung	(*ZielDatentyp*)
Bedingte Auswertung	*LogAusdruck* ? *ThenWert* : *ElseWert*
Delegat hinzufügen/entfernen	+, -
Instanziierung	new
Typinformationen	is, typeof
Länge des Datentyps (unsafe)	sizeof
Überlaufkontrolle	checked, unchecked
Dereferenzierung ?	[]
Dereferenzierung und Adresse (unsafe)	*, ->, &

Tabelle 7.9: Operatorprioritäten absteigend geordnet

Operatorart (nach Priorität geordnet)	Operatoren
Primäroperatoren (höchste Priorität)	., (), [], *x*++, *x*--, new, typeof, checked, unchecked
Unäre Präfix-Operatoren	+, -, !, ~, ++*x*, --*x*, (*Datentyp*)
Multiplikation, Division, Modulo	*, /, %
Addition, Subtraktion	+, -
Bitweises Verschieben	<<, >>
Vergleich (Wert- und Typ-)	<, >, <=, >=, is, as
Gleichheitsoperatoren	==, !=
Bitweises And	&
Bitweises Xor	^
Bitweises Or	|
Logisches And	&&
Bitweises Or	||

Operatorart (nach Priorität geordnet)	Operatoren
Bedingungsoperator	? :
Zuweisung und kombinierte Zuweisung (niedrigste Priorität)	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Tabelle 7.9: Operatorprioritäten absteigend geordnet (Forts.)

Die folgenden Abschnitte diskutieren Besonderheiten und Ungewöhnliches im Umgang mit verschiedenen Operatoren, ohne sie im Einzelnen extra vorzustellen.

Arithmetische Operatoren

Über die arithmetischen Operatoren +, -, *, / gibt es nicht viel zu sagen. Sie entsprechen den vier Grundrechenarten und sind für alle numerischen Datentypen definiert.

Einzig der Modulo-Operator % wird im täglichen Leben nicht so oft benutzt. Er steht für die ganzzahlige Restwertdivision, bei der der Divisor so oft vom Dividenden abgezogen wird, bis ein Rest entsteht, der kleiner als der Divisor ist. Ist der Dividend ganzzahlig durch den Divisor teilbar, bleibt also 0 als Rest. Der Vollständigkeit halber ist % auch für die Gleitkommatypen `float`, `double` und `decimal` definiert, obwohl eine Anwendung dieses Operators für diesen Zahlenbereich an sich eher ungewöhnlich ist.

```
Console.WriteLine(10 % 6);                  // Ausgabe 4
Console.WriteLine(10 % -6);                 // Ausgabe 4
Console.WriteLine(-10 % 6);                 // Ausgabe -4
Console.WriteLine(-10 % -6);                // Ausgabe -4
Console.WriteLine(10.2 % 6.4);              // Ausgabe 3,8
Console.WriteLine(-10.2 % -6.4);            // Ausgabe -3,8
```

Logische Operatoren

Im Gegensatz zu anderen Programmiersprachen (rühmliche Ausnahme: Delphi) können die logischen Operatoren !, && und || in C# nur mit booleschen Operanden aufgerufen werden. Da es noch nicht einmal eine explizite Typumwandlung zwischen dem Datentyp `bool` und numerischen Datentypen gibt (von den entsprechende Methoden der Klasse `Convert` einmal abgesehen), führt hier an der sauberen Programmierung kein Weg vorbei.

```
int i = 4
bool b = !(i < 10 && i > 3)          // b = false
```

Wer allerdings in altgewohnter C-Manier weiterprogrammieren will, der kann auf die bitweisen logischen Operatoren zurückgreifen. Sie sind für alle ganzzahligen Datentypen (also nicht für `float` und `double`) und auch für `bool` (!)

definiert. Damit steht implizit auch die Xor-Operation in Form von ^ als primärer logischer Operator zur Verfügung.

```
int i = 4
bool b = i < 10 ^ i > 3        // b = false, Xor-Operation
i = 7 ^ 8                      // i = 15 , bitweises Xor
i = ~15                        // i = -16, bitweises Not, Einerkomplement
i = 17 & 7                     // i = 1,   bitweises Und
i = 17 | 15                    // i = 31   bitweises Oder
i = 0x11 | 0xF                 // i = 0x1F = 31, bitweises Oder
```

Auf Gleichheit testen

Wer nicht von C/C++ her kommt, für den wird der Vergleichsoperator == etwas gewöhnungsbedürftig sein. Während in C/C++ die Möglichkeit der Verwechslung von == und = in logischen Ausdrükken noch für allerlei Unsinn und Kummer bei der Fehlersuche geführt hat, ist ein solches Durcheinander in C# nicht mehr möglich.

Die Codezeile

```
if (a = 10) TuWas();  // in C/C++ immer wahr, in C# syntaktisch falsch
```

liefert in C/C++ zwar eine Warnung, lässt sich aber anstandslos kompilieren. Von der Semantik her ist sie aber Unsinn, da in der Klammer eine Zuweisung stattfindet und das Ergebnis 10 als Wert ungleich 0 implizit in den Wahrheitswert `true` umgewandelt wird. In C# erkennt der Compiler hier einen Fehler, da es in dieser Sprache keine implizite Konvertierung zwischen dem Datentyp `bool` und Integerdatentypen mehr gibt. Korrekt ist also:

```
if (a == 10) TuWas();  // C# syntaktisch korrekt
```

Synonym zum Operator == kann auch die von `object` an jeden Datentyp vererbte Methode `Equals()` für den Test auf Gleichheit verwendet werden. Die Implementierung stimmt mit der von == überein, einzig die Aufrufart ist unterschiedlich.

Bitweise Operationen

Als Mitglied der C-Familie hat C# selbstverständlich auch die maschinennahen Operationen für die bitweise Manipulation von Integerwerten im Repertoire. Auf Gleitkommawerte, Strings oder den logischen Datentyp sind diese Operationen verständlicherweise nicht anwendbar. Beachten Sie, dass es auch zu diesen Operationen die entsprechenden kombinierten Zuweisungen (wie |=) gibt.

Das folgende Codebeispiel zeigt, wie ein ARGB-Farbwert aus einzelnen Komponenten zusammengesetzt wird. Die expliziten Typumwandlungen sind lästig, hängen im vorliegenden Fall aber damit zusammen, dass die Methode der CLS-Konformität halber einen `int`-Wert erwartet.

```
byte Red = 34;
byte Green = 192;
byte Blue = 147;
byte Alpha = 128;
uint argb = (uint) (Alpha << 24 | Green << 16 | Red << 8 | Blue);
System.Drawing.Color c = System.Drawing.Color.FromArgb((int)argb);
argb |= 0xff000000;    // nicht transparent
System.Drawing.Color c1 = System.Drawing.Color.FromArgb((int)argb);
```

Inkrement und Dekrement

Für Zähloperationen, die den Wert einer Variablen bei jedem Schritt erhöhen bzw. herabsetzen, haben die unären Operatoren `++` und `--` eine lange Tradition. In C/C++ und C# stehen sie als Postfix- (nachgestellt) und als Präfix-Operatoren (vorangestellt) zur Verfügung, deren Semantik sich leicht unterscheidet. Bei Präfix-Notation wird die Operation sofort ausgeführt, der neue Wert der Variable geht in den Ausdruck mit ein. Bei Postfix-Notation behält die Variable vorerst ihren alten Wert und nimmt den neuen Wert erst nach Auswertung des Ausdrucks an – mithin geht hier also der alte Wert in den Ausdruck ein.

```
int a = 10;
int b = a++;     // b = 10, a = 11 (Postinkrement)
int c = ++a;     // a = 12, c = 12 (Präinkrement)
```

Kombinierte Zuweisung

Die Operatoren für die kombinierte Zuweisung ermöglichen einen etwas kompakteren Code, wenn es um die Manipulation einzelner Werte geht. Es versteht sich, dass sie nur auf Linkswerte (also initialisierte Variablen) angewendet werden dürfen. Die beiden folgenden Codezeilen sind äquivalent:

```
a += b;
a = a + b;
```

Ein anderes Beispiel:

```
for (int k = 1; k < 0xFFFF ; k *= 2 )
  ...;
```

is-Operator

Der `is`-Operator definiert einen Vergleich zwischen einem Wert und einem Datentyp. Er liefert also eine Aussage darüber, ob eine Variable einem bestimmten Datentyp (oder einer Basisklasse davon) angehört. Sein Ergebnis ist ein Wert vom Typ `bool`.

```
int i = 10;
if (i is object) ... // das ist wahr, denn object
                     // ist Basisklasse jeder Klasse
```

as-Operator

Der `as`-Operator ist ein Typumwandlungsoperator für Verweistypen. Er versucht eine Typumwandlung des ersten Operanden in den als zweiten Operanden angegebenen Zieltyp. Gelingt die Typumwandlung nicht, liefert der Operator den Wert `null`. Damit hat der Ausdruck

```
MyObj as MyTyp
```

die Bedeutung

```
MyObj is MyTyp ? (MyTyp)MyObj : null
```

sizeof-Operator

Der `sizeof`-Operator ist ein Relikt der Sprache C und kann in C# nur für `unsafe`-Code verwendet werden. Er liefert die Anzahl der Bytes, die für die Repräsentation eines einfachen integrierten Datentyps erforderlich ist. Die Größe verwalteter Datentypen – die von .NET-Klassen wie `System.String`, aber auch die eigener `class`-Klassen – kann nicht mit dem `sizeof`-Operator bestimmt werden. Die Größe von `unsafe struct`-Klassen kann hingegen bedingt ermittelt werden, solange sicher gestellt ist, dass die Datenfelder der Klasse nur einfache und keine verwalteten Datentypen haben.

```
unsafe struct TestStruct
{
  int i, j;
  long x, y;
  ...           // Definition der Methoden der Klasse
}

...
unsafe
{
  Console.WriteLine(sizeof(TestStruct));    // Ausgabe: 24
}
```



Zur Kompilierung von `unsafe`-Code muss der Compilerschalter /unsafe gesetzt sein. In VS.NET ist dieser Schalter auf der Eigenschaftsseite des Projekts über die Eigenschaft KONFIGURATIONSEIGENSCHAFTEN/ERSTELLEN/UNSICHERE CODEBLÖCKE ZULASSEN erreichbar (vgl. Abbildung 6.14).

typeof-Operator

Der `typeof`-Operator ist ein unärer Operator. Er gibt als Ergebnis ein `System.Type`-Objekt zurück, das den im Operanden genannten Typ näher beschreibt. Er liefert dasselbe Ergebnis, wie die von der Basisklasse `object` an jeden Datentyp vererbte Methode `GetType()` und spielt eine wichtige Rolle für die Typreflexion über das `Type`-Objekt, die zu den stark fortschrittlichen Konzepten der .NET-Programmierung gehört. Für die gewöhnliche Win-

dows-Programmierung ist die Anwendung dieses Operators kaum erforderlich. Typisch ist seine Anwendung aber im Zusammenhang mit dem Auslesen von Ressourcen – Bitmaps, Cursorformen und Symbole – aus dem Ressource-Abschnitt der ausführbaren Datei:

```
notifyIcon.Icon = new Icon(typeof(MyDerivedDialog), "MYICON.ICO");
```

Die entsprechenden Überladungen der Konstruktoren erwarten im ersten Parameter irgendeinen Datentyp aus dem Standardnamensbereich der Assembly, in der die Ressource gespeichert ist.

Ein Beispiel aus dem Bereich der Typreflexion wäre die Aufzählung der Member einer Klasse. Die folgende aus dem in Teil 3 ausführlicher diskutierten Beispielprojekt *Farben* stammende Methode zählt wahlweise (nur) die `Color`-Eigenschaften der Klasse `Color` oder der Klasse `SystemColors` auf und hinterlegt sie einem Array.

```
Color[] colors
private void setColorArray(Type t)
{
  // alle Elemente der Klasse ermitteln und in Array packen
  ArrayList colorsX = new ArrayList();
  System.Reflection.PropertyInfo[] colorPIs = t.GetProperties();
  for(int i = 0; i < colorPIs.Length; i++)
    if (colorPIs[i].GetGetMethod().ReturnType == typeof(Color))
      colorsX.Add(colorPIs[i].GetValue(null,null));
  colorsX.CopyTo(colors = new Color[colorsX.Count],0);
}
```

Der Aufruf der Methode lautet entsprechend:

```
setColorArray(typeof(Color));
```

bzw.

```
setColorArray(typeof(SystemColors));
```

Bedingte Auswertung

Der aus C übernommene *ternäre* (dreiwertige) Operator `? :` liefert in Abhängigkeit vom Wahrheitswert von *`LogischerAusdruck`* entweder *`WahrWert`* oder *`FalschWert`* als resultierenden Wert:

```
Linkswert = LogischerAusdruck ? WahrWert : FalschWert;
```

Obwohl der Operator den zweifelhaften Ruf hat, kryptischem Programmierstil Vorschub zu leisten, hat er dennoch seinen Weg in die Sprache C# gefunden. Das liegt wohl daran, dass dieser Operator durchaus auch Vorteile zu bieten hat, die sich vor allem in kompakterem und in vielen Fällen auch schnellerem Code äußern. Vom Prinzip her wäre das, was der Operator für

die bedingte Auswertung leistet, natürlich auch per `if`-Konstrukt realisierbar, doch die `if`-Struktur ist in zeitkritischen Routinen oft zu schwerfällig und erfordert das Zwischenspeichern von Teilausdrücken in zusätzlichen Variablen.

```
public int Signum(long i)
{
  return i < 0 ? -1 : 1;
}
```

7.2.2 Komplexe Verweistypen

Die wirklich fortgeschrittenen Konzepte der Sprache C# stecken in den komplexen Verweistypen. An erster Stelle sind hier natürlich die `class`-Vereinbarungen zu nennen, die nahezu den gesamten Code von C#-Programmen stellen (ein kleiner Teil findet sich auch in `struct`-Vereinbarungen). Weitere komplexe Verweistypen sind Schnittstellen, Delegaten (als Klassen verpackte Funktionszeiger) und Arrays.

Klassen – class-Datentypen

Das Mittel der Wahl, um eigene Datentypen – generell: eigenen Code – zu implementieren, ist in C# die `class`-Klasse. Als »große Schwester« der `struct`-Klasse verkörpert sie das mächtigste Konzept der Sprache: die objektorientierte Programmierung mit Implementierungsvererbung. Aus diesem Grund ist generell die `class`-Klasse gemeint, wenn allgemein nur von Klasse gesprochen wird.

Da sich in C# letztlich alles um Klassen und Objekte dreht, ziemt es sich, dieses komplexe Thema mit einem eigenen Kapitel zu bedenken. Kapitel 8 »Objekte, Klassen, Vererbung« stellt Klassen und die damit unmittelbar zusammenhängenden Konzepte (Konstruktoren, Methoden, Eigenschaften, Operatoren, Vererbung usw.) in der gebührenden Breite vor; Kapitel 9, »Ausnahmebehandlung«, beleuchtet beim Entwurf von Klassen auftauchende Fragen.

Schnittstellen – interface

Eine *Schnittstelle* ist eine verbindliche Spezifikation für eine Reihe von Diensten, die rein formal durch eine Sammlung geeigneter Funktionsprototypen charakterisiert sind. Damit ist die Schnittstelle zwar ein eigenständiger Datentyp – es ist jederzeit möglich, Variablen mit dem Datentyp einer Schnittstelle zu vereinbaren –, instanziierbar ist sie aber nur mittelbar, nämlich über ein Objekt einer Klasse, das die Schnittstelle *implementiert*. Als Teil des Vererbungs- und Spezifikationskonzepts für Klassen sind Schnittstellen somit eng an das Klassenkonzept geknüpft und werden daher mit diesen zusammen im Kapitel »Objekte, Klassen, Vererbung« ausführlicher vorgestellt.

Ein für die praktische Programmierung mit Schnittstellen typisches Beispiel, das den Einsatz der vordefinierten Schnittstellenklassen `IEnumerable` und `IEnumator` veranschaulicht, findet sich im Abschnitt »Codebeispiel – Primzahlen aufzählen« ab Seite 157 .

Delegaten

Waren etwa in C/C++ gewöhnliche Zeiger und selbst Zeiger auf Zeiger noch nach relativ kurzer Einarbeitungszeit beherrschbar, gehörte der Umgang mit Funktionszeigern dort ganz klar zu den höheren Weihen der Programmierung – und zu den Konzepten, wo Schludrigkeit am schlimmsten bestraft wurde. C# kapselt Funktionszeiger als Objekte, so genannte *Delegaten* und entzaubert damit dieses für die ereignisbasierte Programmierung unter Windows so wichtige Konzept: Esoterisch daran bleibt nur noch der Terminus.

C# versteht sich als *typsichere* Sprache. Als solche kann sie es sich nicht leisten, Schludrigkeiten, die zu unerlaubten Systemzuständen führen, überhaupt erst möglich zu machen. Die Lösung folgt dem allgemeinen Strickmuster für die Umschiffung expliziter Zeiger: C# unterwirft den Umgang mit Funktionszeigern einem spezifisch auf die Beschreibung von Funktionsprototypen ausgelegten Typsystem, das dem Compiler auf rein formaler Ebene die vollständige Kontrolle über den Einsatz von Funktionszeigern ermöglicht und die für verwalteten Code notwendige Sicherheit gewährleistet.

Das Typsystem macht aus Funktionszeigern abstrakte Größen, die letztlich wie Objekte gehandhabt werden. Wie die Betrachtung der integrierten Datentypen ergeben hat, lässt sich in C# jede Größe als Objekt begreifen, warum also nicht auch Funktionsaufrufe? Klassen beschreiben Objekte, Delegaten Funktionsaufrufe. Ein Objekt verkörpert einen Wert, ein Funktionsaufruf gleichermaßen – schließlich kann man anstelle eines Rechtswertes jederzeit einen Funktionsaufruf verwenden –, mit dem Unterschied, dass der Funktionsaufruf gegebenenfalls zusätzlich noch Argumente verarbeitet. Die formale Beschreibung der Argumente ist damit die zusätzliche Aufgabe des für Delegaten zuständigen Typsystems. Der Rest ist eine reine Frage der Interpretation.

Kurzum, es macht Sinn, Delegaten als gewöhnliche Klassen zu begreifen, deren Instanzen Funktionsaufrufe von Methoden anderer Klassen oder Objekte repräsentieren. Diese an sich nicht ungewöhnliche Sichtweise wird nur durch den C#-Compiler etwas verschleiert, der eine spezifische Syntax für Delegaten unterstützt. Ein Blick hinter die Kulissen deckt es auf: Der C#-Compiler vereinbart Delegaten in der Tat als Ableitungen der .NET-Basisklasse `System.Delegate` bzw. `System.MulticastDelegate` und implementiert die mit den Delegaten verbundene Syntax mit den Mitteln dieser Klassen.



Am besten stellen Sie sich einen Delegaten als Klasse vor, die einen Funktionsprototypen vor allem »benennt« und deren Instanzen Funktionszeiger verkapseln.

Syntax

Da es in C# keine einfachen Zeiger mehr gibt, muss jede Größe einen echten Datentyp haben. Um eine Funktionsvariable vereinbaren zu können, muss daher eine Typvereinbarung für den Prototyp bestehen. Das Muster dafür ist:

```
[Modifizierer] delegate Rückgabetyp MyFunction ([Parameterliste])
```

Diese Deklaration sieht nicht anders aus als der Kopf einer Instanzmethode in einer Klasse, mit dem Unterschied, dass das Schlüsselwort `delegate` nach dem optionalen Modifizierer erscheint und die Deklaration wie alle anderen Typdeklarationen auch auf der Ebene des Namensraums (also nicht ausschließlich innerhalb einer Klasse) möglich ist. Was die Zugriffsmodifizierer betrifft, so gelten die gleichen Regeln wie bei der Klassendefinition: Auf der Ebene des Namensraums sind `internal` und `public` erlaubt, innerhalb einer Klassenvereinbarung zusätzlich auch `protected` und `private`.

Hier ein kleines Beispiel. Die Typvereinbarung für einen Delegaten, der drei `bool`-Parameter erwartet und einen zurückliefert, sieht beispielsweise so aus:

```
delegate bool TernaryOperation (bool b1, bool b2, bool b3);
```

Angenommen, eine Klasse `Class1` vereinbart nun eine Methode, deren Prototyp mit `TernaryOperation` verträglich ist, dann kann man diese Methode einer Instanz von `TernaryOperation` als Wert zuordnen:

```
class Class1
{
 ...
  public static bool TernaryOp1 (bool b1, bool b2, bool b3)
  { ... }
  public bool TernaryOp2 (bool b1, bool b2, bool b3)
  { ... }
}
...
TernaryOperation MyTernary1 = new TernaryOperation (Class1.TernaryOp1);
Class1 cl = new Class1();
TernaryOperation MyTernary2 = new TernaryOperation (cl.TernaryOp2);
```

Die Instanz selbst lässt sich wie ein gewöhnliches Objekt benutzen, das heißt von Funktionsvariable zu Funktionsvariable zuweisen und auch an Methoden übergeben, die einen Parameter des Typs `TernaryOperation` erwarten. Außerdem sind für Delegaten die Operatoren + und – definiert (hierzu aber gleich noch mehr).

Für den Aufruf schreibt man einfach den Bezeichner der Delegatinstanz, in Klammern dahinter die vom Prototyp geforderten Parameter. Im Gegensatz zu Methoden müssen hier also weder ein Objekt- noch ein Klassenbezeichner notiert werden:

```
bool res1 = MyTernary1(true, false, true);
bool res2 = MyTernary2(true, false, true);
```

Funktionszeiger auch für Instanzmethoden

Wer von C/C++ her kommt, wird sich vielleicht fragen, ob der Formalismus bei C# hier nicht nur einfach seine Blüten treibt. Warum also kann man anstatt der »lästigen« Instanziierung mit dem `new`-Operator nicht einfach schreiben:

```
TernaryOperation MyTernary1 = Class1.TernaryOp1; // Syntaxfehler
```

Mit Blick auf statische Methoden wäre diese Lösung für C# vielleicht noch vertretbar, sofern man darauf vertrauen kann, dass die statische Instanz einer Klasse während des gesamten Programmlaufs erhalten bleibt (was per se nicht unbedingt gesichert ist). Ähnlich wie bei der Instanziierung eines `struct`-Objekts könnte der Compiler hier eine Typüberprüfung vornehmen und die Typsicherheit wäre wohl gewährleistet, solange der Zugriff auf die Klasse möglich ist. Soweit deckt sich das Szenario mit der Situation in C/C++, wo Funktionszeiger ausschließlich für statische Funktionen zulässig sind.

C# geht mit dem Delegatkonzept aber noch einen Schritt weiter, indem die Sprache auch Instanzmethoden als Werte für Delegatinstanzen zulässt, wie das genannte Beispiel ohne viel Aufhebens demonstriert. Bei Verwendung der vorgeschlagenen »vereinfachten« Schreibweise

```
TernaryOperation MyTernary2 = cl.TernaryOp2;     // Syntaxfehler
```

ergäbe sich natürlich sofort das Problem, dass das Objekt `cl` sein Leben vor `MyTernary2` aushauchen könnte und der Funktionszeiger spätestens nach dem nächsten Auftritt der Freispeicherverwaltung ohne Kontext dastünde. Die von der C#-Syntax geforderte explizite Instanziierung eines Delegaten bringt hingegen ein neues »Funktionszeigerobjekt« hervor, das seinerseits eine Referenz auf das Objekt `cl` hält, und nicht nur auf eine seiner Methoden. Solange diese Objektreferenz besteht, lässt die Freispeicherverwaltung das Objekt `cl` ungeschoren, weshalb der Funktionszeiger, um den es eigentlich geht, seinen Kontext und somit seine Gültigkeit behält.

Verwendung

An Einsatzmöglichkeiten für Delegaten mangelt es nicht. Der wichtigste Einsatzbereich im Zusammenhang mit der Windows-Programmierung

dürfte fraglos in der Ereignisbehandlung liegen. Ein Objekt erklärt sich für die Behandlung eines Ereignisses zuständig, indem es eine (oder gegebenenfalls auch mehrere) seiner Methoden als Rückruffunktion registriert. Weil dieser Bereich so wichtig ist, stellt C# in Form des Schüsselworts `event` sogar ein syntaktisches Mittel für die vereinfachte Registrierung von Ereignisbehandlungsroutinen bereit (mehr dazu im Abschnitt »Ereignisse« auf Seite 208).

Weitere Einsatzfelder eröffnen sich im Zusammenhang mit der Implementierung von Klassenbibliotheken mit generischer Orientierung – man extrapoliere hier etwa den im Zusammenhang mit Funktionszeigern gerne als »typisches Beispiel« vorgestellten Sortieralgorithmus, dem eine Vergleichsoperation als Parameter übergeben wird.

Ein mindestens genauso breites Einsatzfeld für Delegaten ist der gesamte Bereich der asynchronen Programmierung – als Stichworte hierzu mögen an dieser Stelle Multithreading, Aufgabenplanung (Scheduling) und Lastverteilung (Load Balancing) reichen.

Codebeispiel – Ausgabe einer Logiktabelle für ternäre Logikfunktionen

Das folgende Listing des Projekts *DelegateDemo* zeigt ein kleines als Konsolenanwendung ausgelegtes Testprogramm, das den Einsatz einer generischen Methode `LogicTableTernary()` für die Ausgabe einer Wahrheitstabelle für ternäre (=dreiwertige) Logikoperationen demonstriert.

```
using System;

namespace DelegateDemo
{
  using System;
  // Delegatdeklaration
  delegate bool TernaryOperation (bool b1, bool b2, bool b3);

  public class MyClass
  {
    public static bool TernaryOp1 (bool b1, bool b2, bool b3)
    {
      return b1 & b2 ^ b3;
    }

    public bool TernaryOp2 (bool b1, bool b2, bool b3)
    {
      return b1 & b2 & b3;
    }

    public bool TernaryOp3 (bool b1, bool b2, bool b3)
    {
      return b1 | b2 & b3;
    }
  }
```

```
  public class MainClass
  {
    static public void Main()
    {
      // Instanziierung und Zuordnung einer statischen Methode
      TernaryOperation MyDelegate =
        new TernaryOperation (MyClass.TernaryOp1);
      bool test = MyDelegate(true, true, true);  // Einfacher Aufruf
      LogicTableTernary(MyDelegate);        // Übergabe als Parameter

      MyClass mc = new MyClass();   // Instanziierung eines Objekts
      // Zuordnung einer Instanzmethode
      MyDelegate = new TernaryOperation (mc.TernaryOp2);
      LogicTableTernary(MyDelegate);
      MyDelegate = new TernaryOperation (mc.TernaryOp3);
      LogicTableTernary(MyDelegate);
    }

    // Ausgabe einer Logiktabelle für Ternaryäre logische Operationen
    static void LogicTableTernary(TernaryOperation t)
    {
      Console.WriteLine("Logiktabelle");
      Console.WriteLine("Op1   Op2   Op2   | Rückgabe");
      Console.WriteLine("------------------|----");
      Console.WriteLine("True  True  True  | " + t(true, true, true));
      Console.WriteLine("True  True  False | " + t(true, true, false));
      Console.WriteLine("True  False True  | " + t(true, false, true));
      Console.WriteLine("True  False False | " + t(true, false, false));
      Console.WriteLine("False True  True  | " + t(false, true, true));
      Console.WriteLine("False True  False | " + t(false, true, false));
      Console.WriteLine("False False True  | " + t(false, false, true));
      Console.WriteLine("False False False | " + t(false, false, false));
      Console.WriteLine();
    }
  }
}
```

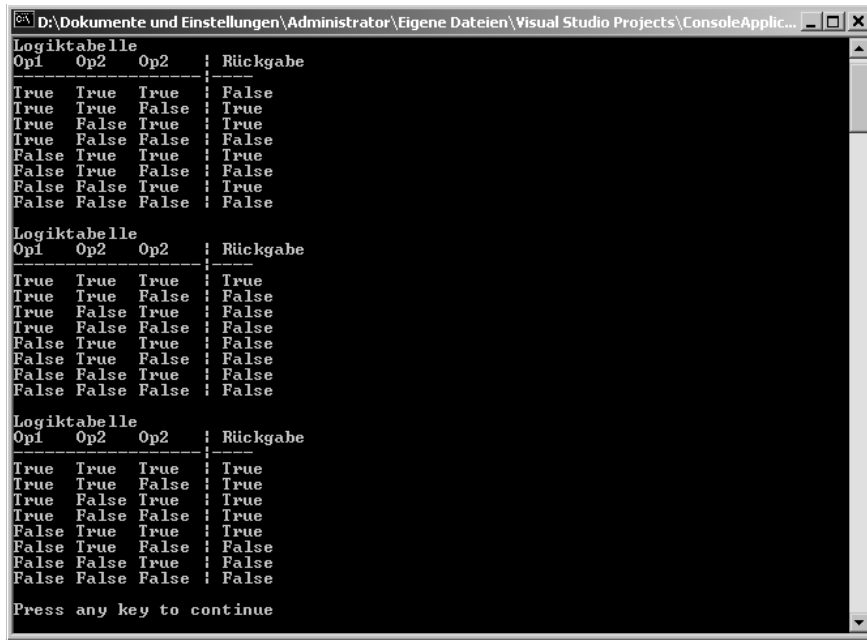
Abbildung 7.6 zeigt die Ausgabe des Programms.

Verkettung von Delegaten zu Multicast-Delegaten

In der bisherigen Betrachtung war ein Delegat nichts weiter als ein formaler Rahmen für einen einzelnen Funktionsaufruf. Im Zusammenhang mit Ereignissen kann es jedoch erforderlich sein, mehr als eine Behandlungsroutine pro Ereignis zu registrieren – etwa, wenn in der GUI-Programmierung mehrere Objekte auf ein und dasselbe Ereignis reagieren sollen.

Behandlungsroutinen für Ereignisse müssen keinen Funktionswert zurückliefern, denn es gibt keinen Aufrufer im eigentlichen Sinne, der mit dem Wert etwas anfangen könnte. Natürlich könnte der »Zustellmechanismus« einen solchen fordern, er tut es aber nicht – mit gutem Grund: Gleichartige Methodenaufrufe, die `void` zurückliefern, lassen sich auf einfache Weise verketten. Sie werden in einer Liste zusammengefasst und der Reihe nach unter

Abbildung 7.6: Ausgabe der Logiktabelle für die übergebene ternäre Logikoperation



Übergabe jeweils derselben Parameter aufgerufen. Genau diese Logik ist es, die bei der Ereignisbehandlung benötigt wird. Was liegt also näher, als das Delegatenkonzept so auszuweiten, dass eine Delegatinstanz auch mit einer Liste von Funktionsaufrufen zurechtkommt und Operationen bereitstellt, die es ermöglichen, Funktionsaufrufe in die Liste einzufügen bzw. herauszunehmen. In diesem Fall spricht man von *Multicast-Delegaten*.

Das .NET-Framework stellt diese Funktionalität in der von der abstrakten Klasse `System.Delegate` abgeleiteten .NET-Basisklasse `System.MultiCastDelegate` für alle Delegaten bereit. Und die Implementierung der Operatoren + und – (bzw. += und -=) macht die Verwaltung der Aufruflisten zu einer wahrlich intuitiven Angelegenheit.

Ausgehend von einer Vereinbarung wie:

```
delegate void MyMultiCast(string s);
```

lassen sich Instanzen nach Belieben zu Aufruflisten zusammenfügen

```
MyMultiCast CallingList = new MyMultiCast(p.InstanceMethod);
MyMultiCast d = new MyMultiCast(MainClassStringMethod);
CallingList += d;
```

oder einzelne Instanzen wieder aus einer bestehenden Liste entfernen

```
CallingList -= d;
```

Sollte `d` zuvor nicht in `AufrufListe` eingefügt worden sein, ist das kein Unglück. In diesem Fall passiert schlicht nichts – die Operation ist »tolerant« implementiert.

Wie zu erwarten, lassen sich auch mehrelementige Aufruflisten in eine Aufrufliste einfügen, solange nur der Prototyp stimmt:

```
CallingList += CallingList + CallingList;
```



Wundern Sie sich nicht, dass der C#-Compiler für Multicast-Delegaten (selbst für solche, die für die Ereignissignalisierung verwendet werden) auch Prototypen zulässt, die andere Typen als `void` *zurückgeben. Der Sender erhält in diesem Fall den Funktionswert, der zuletzt (via +=) in der Aufrufliste registrierten Delegatinstanz.*

Zur Abarbeitung einer bestehenden Aufrufliste wird die Delegatinstanz wie gehabt schlicht unter Angabe der erforderlichen Parameter als gewöhnlicher Funktionsaufruf notiert:

```
Aufrufliste("Das ist der Parameter");
```

Die Aufrufe werden nun in der Einfügereihefolge abgearbeitet und erhalten alle ein und denselben Parameter.

Dabei stellt sich natürlich die Frage, was passiert, wenn ein `ref`- oder `out`-Parameter dabei ist:

```
delegate void MyMultiCastRef(ref int i);
```

Angenommen in `MyClass` sind folgende Methoden vereinbart:

```
public static void Increment(ref int i)
{
  ++i;
}

public static void MultiplyByTwo(ref int i)
{
  i *= 2;
}
```

Packt man diese beiden Methoden nun in die Aufrufliste einer Delegatinstanz, dann spielt es für das Ergebnis in der Tat eine Rolle, in welcher Reihenfolge die Liste aufgebaut wurde.

```
MyMultiCastRef CallingList;
MyMultiCastRef Inc = new MyMultiCastRef(MyClass.Increment);
MyMultiCastRef Mult = new MyMultiCastRef(MyClass.MultiplyByTwo);
Liste = Mult + Inc;
int i = 2;
```

```
CallingList (ref i);    // i = 5
int j = 2;
CallingList = Inc + Mult;
CallingList (ref j);    // j = 6
```

Wie man sich leicht überlegt, wirkt sich im Falle eines `out`-Parameters hingegen nur der letzte Aufruf in der Aufrufliste aus.

Codebeispiel – MultiCastDelegateDemo

Das kleine Beispielprogramm *MultiCastDelegateDemo* demonstriert den Umgang mit Delegaten für mehrfache Funktionsaufrufe noch einmal in einem komplexeren Zusammenhang:

```
namespace MultiCastDelegateDemo
{
  delegate void MyMultiCastRef(ref int i);
  delegate void MyMultiCast(string s, ref int i);
  public class MyClass
  {
    public void InstanceMethod(string s, ref int i)
    {
      i++;
      Console.WriteLine("Aufruf der Instanzmethode. " + s + i);
    }

    static public void StaticMethod(string s, ref int i)
    {
      i++;
      Console.WriteLine("Aufruf der statischen Methode. " + s + i);
    }
  }

  public class MainClass
  {
    static public void Main()
    {
      MyClass mc = new MyClass();

      // Delegatinstanz vereinbaren und drei Funktionsaufrufe verketten
      MyMultiCast CallingList  = new MyMultiCast(mc.InstanceMethod);
      MyMultiCast d = new MyMultiCast(MainClassStaticMethod);
      d += d;   // Aufrufliste d wird ergänzt
      CallingList  += d; // d wird ein zweites Mal eingefügt ...
      CallingList  += new MyMultiCast(MyClass.StaticMethod);
      // Direkter Aufruf
      int i = 10;
      CallingList (" Parameter des direkten Aufrufs", ref i);

      Console.WriteLine("**********************");
      CallingList  -= d;    // Aufrufliste d aus der Liste entfernen
      // Indirekter Aufruf via InvokeMultiCast
      InvokeMultiCast(CallingList, " Parameter des indirekten Aufrufs", i);
```

```
      MyMultiCastRef Liste;
      MyMultiCastRef Inc = new MyMultiCastRef(Inkrement);
      MyMultiCastRef Mult = new MyMultiCastRef(MultiplyByTwo);
      Liste = Mult + Inc;
      int ii = 2;
      Liste (ref ii);    // i = 5
      int j = 2;
      Liste = Inc + Mult;
      Liste (ref j);    // j = 6
    }

    static void MainClassStaticMethod(string s, ref int i)
    {

      Console.WriteLine("Aufruf von MainClassStringMethod. " + s + i);

    }

    static void InvokeMultiCast(MyMultiCast m, string Argument, int i)
    {
      m(Argument, ref i);
    }

    public static void Inkrement(ref int i)
    {
      ++i;
    }

    public static void MultiplyByTwo(ref int i)
    {
      i *= 2;
    }
  }
}
```

Dieses Beispiel demonstriert an sich recht schön, wie die Listenimplementierung für Delegaten arbeitet. Man beachte, dass der Code die Delegatinstanz `d` mit einer Kopie ihrer selbst verdoppelt, als Element in die `AufrufListe` einfügt und später wieder entfernt. Abbildung 7.7 zeigt die Ausgaben des Programms.

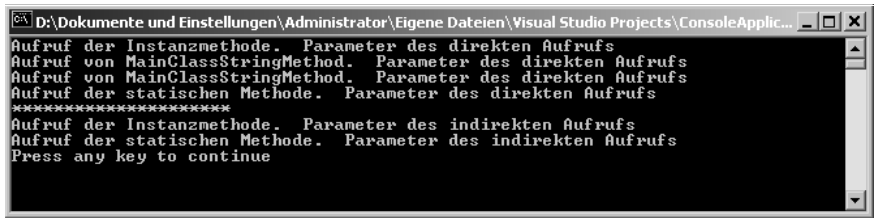


Abbildung 7.7: Ausführung eines mit mehreren Aufrufen initialisierten Multicast-Delegaten. Die Ausgabe zeigt, wie Aufruflisten ihrerseits in Aufruflisten eingefügt und wieder daraus entfernt werden können.

Die Abarbeitung einer Aufrufliste erfolgt also elementweise in der Reihenfolge des Einfügens der Elemente. Jedes Element kann seinerseits eine Liste sein, die jeweils komplett abgearbeitet wird, bevor das nächste Element auf gleicher Ebene an der Reihe ist.

Ereignisse

Der Mechanismus »Multicast-Delegat« ist so allgemein, dass er sich als »Transportschicht« für beliebige ereignisbasierte Benachrichtigungsmodelle eignet – darunter besonders prominent: das Nachrichtenmodell von Windows.

Obwohl C# mit dem Zusatz `event` deklarierte Datentypen als eigenständige Verweistyp-Kategorie behandelt, geht es dabei vom Prinzip her um nichts weiter als um »veredelte« Multicast-Delegaten, deren höchst offizieller Zweck die Signalisierung von Ereignissen ist. Auch wenn die Implementierung des Mechanismus zur Signalisierung von Ereignissen hinter den Kulissen etwas komplexer aussieht – Ereignisse werden im Allgemeinen nicht synchron behandelt, sondern in einer Warteschlange organisiert und ihre Verarbeitung unterliegt einer gewissen Planung – so lässt er sich in seiner Wirkungsweise doch am besten verstehen, wenn man dabei Multicast-Delegaten vor Augen hat. Schließlich hat die Ereignissignalisierung für das Konzept der Multicast-Delegaten ja Pate gestanden.

Mechanismus der Ereignissignalisierung

Bei der Ereignissignalisierung kommt es darauf an, ein oder mehrere zur Übersetzungszeit üblicherweise noch nicht bekannte bzw. existierende Empfängerobjekte über eine spezifische Zustandsänderung zu informieren – entweder des Systems oder eines Senderobjekts, das gegebenenfalls auf von anderen Objekten stammende Ereignisse reagiert. Ein Beispiel dafür sind bestimmte Maus- oder Tastaturaktionen des Benutzers, ein anderes die Reaktion eines Senderobjekts auf irgendeine Anforderung eines Clients.

Das Signalisierungsschema ist höchst einfach: Ein Empfängerobjekt, das sich von einem Senderobjekt über das Auftreten eines bestimmten Ereignisses informieren lassen will, registriert eine Behandlungsroutine in Form einer Delegatinstanz beim Senderobjekt. Das Senderobjekt stellt dazu ein eigens für diesen Zweck als `event` deklariertes Datenfeld bereit. Hierbei handelt es sich um eine noch nicht instanziierte, also mit `null` initialisierte Delegatvariable – eines passenden Delegattyps, versteht sich! Um das Ereignis schließlich zu signalisieren, ruft das Senderobjekt die durch dieses Datenfeld referenzierte Aufrufliste wie einen Funktionsaufruf auf – vorausgesetzt, diese Liste ist nicht leer (`null`). Die hinter dem Multicast-Delegaten stehende Logik sorgt dafür, dass die registrierten Delegatinstanzen der Reihe nach (aber nicht unbedingt synchron) abgearbeitet werden.

Die Codeebene des Senderobjekts für das genannte Schema sieht so aus:

1. *Festlegung eines Multicast-Delegaten* – die Senderklasse legt sich für jeden Ereignistyp, den sie signalisieren will, auf einen Delegaten fest (Polymorphismus ist möglich), dessen Instanzen das Ereignis repräsentieren – bzw. die für das Ereignis relevanten Daten transportieren. Diese Klasse kann entweder in einem beliebigen Namensraum (beispielsweise als .NET-Basisklasse) vordefiniert sein oder von der Senderklasse selbst exportiert werden. Die in der Parameterliste genannten Parameter übertragen Werte vom Sender- zum Empfängerobjekt, die das Ereignis näher beschreiben. Ein Beispiel für die Syntax:

   ```
   public delegate void MyTimeoutHandler ([Parameterliste]);
   ```

   In »ausgefüllter« Form also beispielsweise:

   ```
   public delegate void Timeout (object sender, System.EventArgs e);
   ```

2. *Vereinbarung einer Klasse zur Beschreibung des Ereignisses – dieser Schritt ist optional, die durch den Delegaten beschriebene Parameterliste kann auch leer sein bzw. mit anderweitig definierten Datentypen auskommen. Für die Ereignissignalisierung im Stil von Windows enthält die Parameterliste einen Verweis auf das Senderobjekt und einen Verweis auf ein Objekt, welches das Ereignis an sich beschreibt. Die hinter diesem Objekt stehende Klasse ist dann (direkt oder indirekt) von* `System.EventArgs` *abgeleitet:*

   ```
   public class MyTimoutEventArgs : System.EventArgs
   {
     public System.DateTime TimeoutStamp;
     public int RetrialsUntilTimeout;
     public MyTimoutEventArgs(int retrials)
     {
         TimeoutStamp = DateTime.Now;
         RetrialsUntilTimeout = retrials;
     }
   }
   ```

   Der unter 1. genannte Delegat wäre mit diesem Parametertyp demnach so zu vereinbaren:

   ```
   public delegate void Timeout (object sender, MyTimoutEventArgs e);
   ```

3. *Bereitstellung eines* `event`*-Datenfelds* – die Senderklasse vereinbart ein (wahlweise statisches oder dynamisches) Multicast-Delegat-Datenfeld unter Angabe des `event`-Modifizierers:

   ```
   [static] [Modifizierer] event MyMulticastDelegate MyEventField;
   ```

   Ein Datenfeld mit dem unter 2. genannten Delegaten wäre demnach beispielsweise wie folgt zu vereinbaren:

```
class SenderClass
{
  public event Timeout MyTimeOut;
  ...
}
```

4. *Signalisierung des Ereignisses* – das Senderobjekt prüft, ob Empfängerobjekte Behandlungsroutinen (Delegatinstanzen) für das Ereignis registriert haben und ruft dann gegebenenfalls das Multicast-Delegat-Datenfeld unter Bereitstellung entsprechender Parameterwerte auf.

```
private virtual void OnTimeOut(MyTimeOutEventArgs e)
{
  if (MyTimeOut != null)
    MyTimeOut(this, e);
}
```

Dieser Aufruf ist vom Prinzip her in jeder beliebigen Methode des Objekts möglich – auch in statischen Methoden. Um der Vererbung Rechnung zu tragen, wird hierfür im Allgemeinen aber eine spezifische `virtual`-Methode bereitgestellt, deren alleinige Aufgabe die Ereignissignalisierung ist. Zudem besteht die Konvention, dass der Bezeichner einer solchen Methode das Präfix `On` trägt.

Abgeleitete Klassen können diese Methode dann aus zwei Gründen überschreiben:

1. Um die Ereignissignalisierung des Basisklassenobjekts selbst zu gestalten. Die abgeleitete Klasse würde dann den Code der Basis abwandeln und auf die eigenen Bedürfnisse zuschneiden:

```
private override void OnTimeOut(MyTimeOutEventArgs e)
{
    if (MyTimeout != null)   // MyTimeout ist ererbt
     MyTimeOut(this, e);    // Aufruf mit anderem Absender
}
```

2. Zweitens, um sich »aus erster Hand« (und synchron) über Ereignisse, die das Basisklassenobjekt signalisiert, informieren zu lassen, ohne dafür eine eigene Routine per Delegatinstanz beim Basisklassenobjekt zu registrieren. Also beispielsweise, um selbst ein Ereignis zu signalisieren, das gegebenenfalls wiederum einen von `MyTimeOutEventArgs` abgeleiteten Parametertyp einsetzt, der das Ereignis aus Sicht der abgeleiteten Klasse besser beschreibt. In diesem Fall darf die überschreibende Methode nicht vergessen, die Basisklassenvariante aufzurufen, damit die gegebenenfalls beim Basisklassenobjekt registrierten Behandlungsroutinen abgearbeitet werden:

```
private override void OnTimeOut(MyTimeOutEventArgs e)
{
     base.OnTimeOut(ref e);
     HandleOnTimeOut(e);
}
```

Die Codeebene eines Empfängerobjekts für das genannte Schema sieht so aus:

1. *Vereinbarung einer geeigneten Behandlungsroutine* – um vom Senderobjekt über ein Ereignis informiert werden zu können, muss das Empfängerobjekt eine Methode bereitstellen, die als Wert für das Delegatfeld des Senderobjekts zulässig ist. Die Signatur dieser Methode ist also so zu wählen, wie es der für das Ereignis deklarierte Delegat vorschreibt. Um im Beispiel zu bleiben, also etwa so:

   ```
   class TargetClass
   {
     SenderClass Sender;
     ...   // Instanziierung des oder Anbindung an Senderobjekt
     private void Sender_MyTimeOut(object sender, MyTimeOutEventArgs e)
           ...
   }
   ```

   In Einklang mit der von Visual Studio .NET vorgeschlagenen Benennungskonvention für die Ereignisbehandlung empfiehlt es sich, Behandlungsroutinen des Empfängerobjekts mit dem Namen des Senderobjekts, gefolgt von einem Unterstrich und dann dem Ereignisnamen zu benennen – wie es das Beispiel zeigt.

2. *Registrieren der Behandlungsroutine beim Senderobjekt* – das Empfängerobjekt generiert eine Delegatinstanz, initialisiert diese mit der für das Ereignis vorgesehenen Behandlungsroutine und fügt diese in die Aufrufliste des vom Senderobjekt veröffentlichten Multicast-Delegat-Datenfelds ein.

   ```
   Sender.MyTimeOut += new TimeOut(Sender_MyTimeOut);
   ```

   Das wär's. Signalisiert das Senderobjekt nun das Ereignis durch Aufruf der Methode `OnTimeOut()`, kommen (früher oder später) die Methoden `Sender_MyTimeOut()` der Empfängerobjekte zum Aufruf.

Arrays

In C/C++ sind Arrays schon immer ein delikates Thema und Quell vielen Unheils gewesen. Der Griff über die Arraygrenzen hinaus, die allzu nahe Verwandtschaft mit dem Zeigerkonzept, die aufwändige Freispeicherverwaltung und die notwendige Sorgfalt bei der Konstruktion und Destruktion von Arrayinhalten ließen den Umgang mit Arrays nicht selten zum Drahtseilakt geraten. Steter Speicherschwund durch im Heap verlorene (»verwaiste«) Objekte war gewissermaßen das Markenzeichen von Programmen für Windows 3.1 – mit der bekannten und belächelten Notwendigkeit, das System nach einigen Stunden Arbeit neu starten zu müssen. Eine Anwendung war umso besser programmiert, je weniger Speicherschwund sie produzierte – und Schwund gab es überall. Windows 9x traf zwar Vorkehrungen, den von einer Anwendung angeforderten Speicher spätestens bei Beendigung der Anwendung wieder zu recyceln, das änderte aber

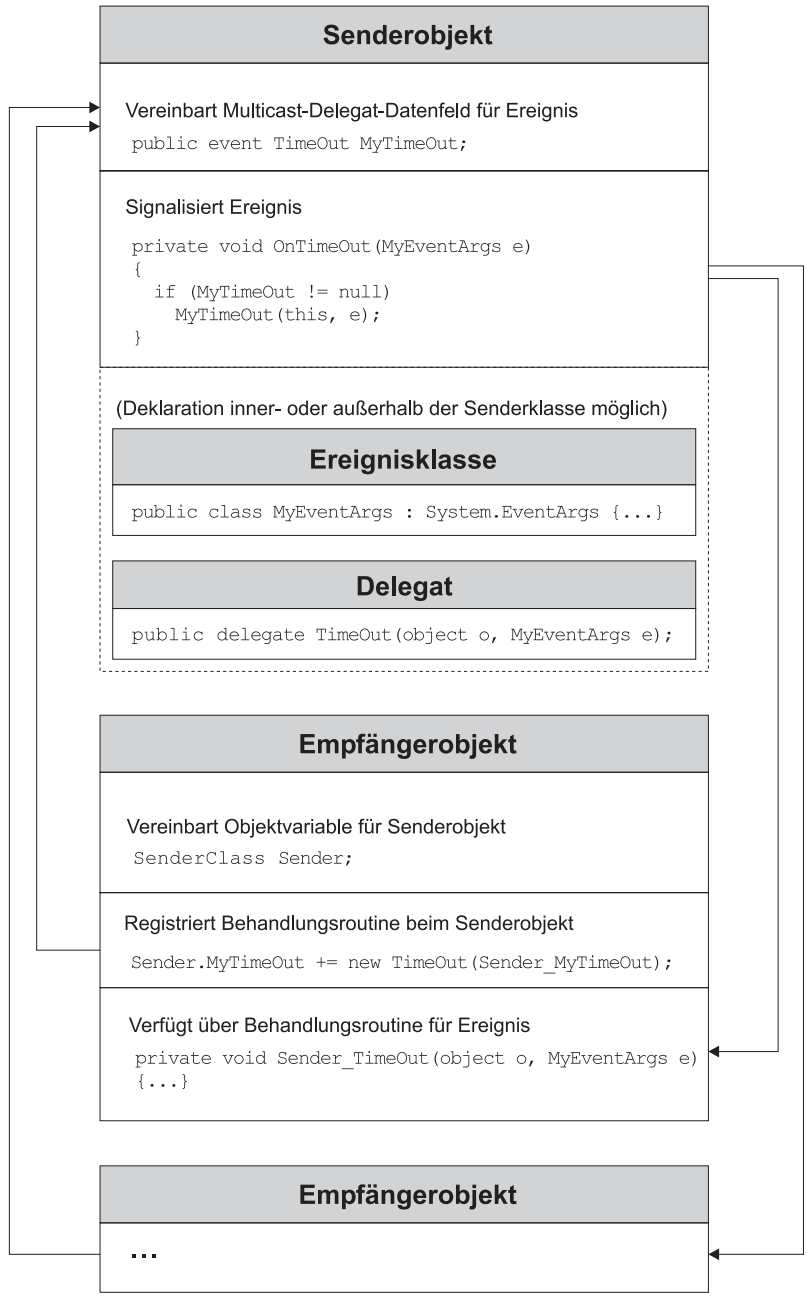


Abbildung 7.8: Ereignismechanismus

nicht viel daran, dass die Anwendungen nach wie vor ihre Undichtigkeiten hatten – nur sie fielen eben nicht mehr so auf, weil das verdrängende Multitasking und die weitgehend neu geschriebene Speicherverwaltung das Betriebssystem an sich erheblich robuster machen als die Vorgängerversion.

Für den Einsatz in der verwalteten Umgebung des .NET-Laufzeitsystems steht hingegen in erster Linie die Sicherheit des Codes im Vordergrund, weshalb man für C# den Weg ging, Arrays nach dem Vorbild der Containerklassen der STL/ATL auf der Basis eines eigenständigen Datentyps zu implementieren. Die Sprache führt ihr Arraykonzept daher auf die abstrakte Klasse `System.Array` zurück, die ihrerseits verschiedene Schnittstellen implementiert:

```
public abstract class Array : ICloneable, IList, ICollection, IEnumerable
```

In C# sind Arrays echte Objekte mit Methoden und Eigenschaften, die auf unerlaubte Zugriffsversuche (nicht zuletzt: Überschreitung der Arraygrenzen) mit Ausnahmen reagieren. Die automatische Freispeicherverwaltung der .NET-Laufzeitumgebung trägt das ihre zur Robustheit der Array-Implementierung von C# bei – und erweist sich hier geradezu als segensreiche Einrichtung, die den Umgang mit Arrays zu einer ähnlich einfachen Angelegenheit wie bei Visual Basic macht.

Dynamische Arrays?

Im Gegensatz zu Visual Basic und Delphi unterstützt C# keine dynamischen Arrays. Ein Array wird einmal dimensioniert, und das war es dann. Nachträgliche Verkleinerungen oder Vergrößerungen lassen sich nur durch (mehr oder weniger schikanöses) Kopieren der Elemente in ein entsprechend dimensioniertes neues Array bewerkstelligen.

Trotzdem müssen Sie bei C# nicht auf dynamische Arrays verzichten. In der .NET-Basisklasse `ArrayList` findet sich alles, was das Herz begehrt. (Mehr über die Arbeit mit dieser Klasse in Kapitel 11, »Einsatz der .NET-Basisklassen«.

Syntax

Die für den Umgang mit Arrays geltende Syntax orientiert sich zwar weitgehend am Üblichen, ist aber bei mehrdimensionalen Arrays etwas eigenwillig ausgefallen.

- *Vereinbarung* – Arrays werden als Objektvariable eines Elementtyps, gefolgt von einem leeren eckigen Klammerpaar vereinbart:

  ```
  int[] myIntArray;
  ```

- *Instanziierung* – Arrays werden wie Objekte unter Verwendung des `new`-Operators angelegt. Dabei muss die Elementanzahl angegeben werden. Die Anweisung

  ```
  myIntArray = new int[10];
  ```

ordnet `MyIntArray` ein Arrayobjekt mit 10 Elementen zu, wobei die Indizierung grundsätzlich mit 0 beginnt und hier mit 9 endet.

- *Initialisierung* – die Initialisierung kann erstens elementweise unter Verwendung des Elementzugriffs passieren, zweitens unterstützt C# (wie von C/C++ her gewohnt) die Initialisierung per {}-Wertliste. Der Compiler ermittelt in diesem Fall die Elementanzahl automatisch.

```
myIntArray = new int[] {2, 3, 5, 7, 11, 13, 17, 19};
```

- *Elementzugriff* – die einfachste Form des Elementzugriffs ist die Verwendung des Indexers bzw. der Indexschreibweise. Alternativ lassen sich auch die Methoden `GetValue()` und `SetValue()` verwenden, wenn nicht mehr als drei Arraydimensionen vorhanden sind.

```
int i = myIntArray[7];
```

- *Zuweisung* – die Zuweisung zwischen zwei Arrayvariablen (nicht: Arrayelementen) ist keine Wertzuweisung, es wird also im Unterschied zur Stringzuweisung *keine* Kopie des Arrayinhalts generiert, sondern nur ein weiterer Verweis darauf:

```
int[] myIntArray1 = myIntArray;     // Weitere Referenz auf das Array
```

  Um eine identische Wertkopie eines Arrays zu erzeugen, steht die generische `Clone()`-Methode zur Verfügung, die allerdings generell eine explizite Typumwandlung in den jeweiligen Zieltyp erfordert. Partielle Wertkopien (Indexbereich) sind mit der statischen Methode `Array.Copy()` oder der Instanzmethode `CopyTo()` möglich.

- *Übergabe als Parameter* – Arrays sind Verweistypen. Die Übergabe eines Arrays als »by value«-Parameter erzeugt daher keine Kopie des Arrays, sondern – wie für Objekte üblich – nur der Arrayvariablen, mithin des Verweises auf das Array. Das Äquivalent einer echten Wertübergabe lässt sich erzielen, indem das Array vor Übergabe per `Clone()` dupliziert wird:

```
MyMethod(int [] arr) { ... };
...
MyMethod(myIntArray);           // By-Val-Aufruf erzeugt keine Kopie
MyMethod( (int[]) myIntArray.Clone());   // Echte Wertkopie
```

In den folgenden Abschnitten geht es erst einmal um eindimensionale Arrays.

Eindimensionale Arrays

Die Syntax für die Vereinbarung und Instanziierung eines eindimensionalen Arrays sieht so aus:

```
ElemTyp[] ArrayBezeichner;
ArrayBezeichner = new ElemTyp[ElemAnzahl];
```

beide Schritte lassen sich wie gewohnt auch kombinieren:

```
ElemTyp[] ArrayBezeichner = new ElemTyp[ElemAnzahl];
```

Als *ElemTyp* kann jeder Datentyp vereinbart werden – auch Arraytypen; zu Multiarrays aber gleich noch mehr. Für *ElemAnzahl* erwartet der Compiler einen Wert vom Typ `int`, gestattet freilich aber auch Werte, die er implizit in diesen Typ konvertieren kann:

```
byte AnzWochentage = 7;
string[] WochenTage = new string[AnzWochentage];
```

C#-Arrays sind grundsätzlich nullbasiert: Der niedrigste zulässige Index für den Elementzugriff ist 0, der höchste im genannten Beispiel `AnzWochentage - 1`, also 6.

Die elementweise Initialisierung von Arrays kann eine lästige Angelegenheit sein. C# unterstützt daher nach dem Vorbild von C/C++ und Delphi auch die Wertlisteninitialisierung. Statt

```
WochenTage[0] = "Sonntag";
WochenTage[0] = "Dienstag";
...
WochenTage[6] = "Samstag";
```

kann man daher auch kurz schreiben.

```
string [] WochenTage = {"Sonntag", "Montag", "Dienstag", "Mittwoch",
                        "Donnerstag", "Freitag", "Samstag"};
```

Zur Initialisierung einer bereits vereinbarten Arrayvariablen per Wertliste muss man den Rechtswert hingegen »ausschreiben« und explizit typisieren:

```
WochenTage = new string[] {"Sonntag", "Montag", "Dienstag", "Mittwoch",
                           "Donnerstag", "Freitag", "Samstag"};
```

Das ist klarer, zumal so auch polymorphe Initialisierungen wie die folgende möglich sind:

```
object[] oArr = new string[] {"Sonntag", "Montag", "Dienstag", "Mittwoch",
                              "Donnerstag", "Freitag", "Samstag"};
```

Es ist des weiteren zulässig (wenngleich nur selten sinnvoll), für den Rechtswert eine Elementanzahl anzugeben. Die Angabe muss ein konstanter Wert sein und mit der Anzahl der Elemente in der Wertliste übereinstimmen:

```
WochenTage = new string[7] {"Sonntag", "Montag", "Dienstag", "Mittwoch",
                            "Donnerstag", "Freitag", "Samstag"};
```

Als Elemente der Wertliste sind auch initialisierte Variablen (Wert- und Verweistypen) sowie Ausdrücke zulässig:

```
string s = "tag";
string die = "Dienstag";
WochenTage = new string[] {"Sonn" + s, "Mon" + s, die, "Mittwoch",
                            "Donnerstag", "Freitag", "Samstag"};
```

Zum Erweitern eines bestehenden Arrays kopiert man das Array in ein neues entsprechender Größe. Das geht je nach Bedarf elementweise oder *en bloc*. Um das `WochenTage`-Array beispielsweise in eines mit 9 Elementen ab Index 2 zu kopieren, schreibt man:

```
string [] Tage;
WochenTage.CopyTo(Tage = new string[WochenTage.Length + 2], 2);
```

Die Methode `CopyTo()` stammt hier also vom »kleineren« Arrayobjekt. Reicht das im ersten Parameter angegebene Array von der Größe her nicht aus oder ist der Startindex im zweiten Parameter so gesetzt, dass die Arraygrenzen überschritten werden, kommt es – wie immer bei der Überschreitung von Arraygrenzen – zu einem Laufzeitfehler. Abbildung 7.9 zeigt diese Situation, wenn als Startindex 3 gewählt wird.

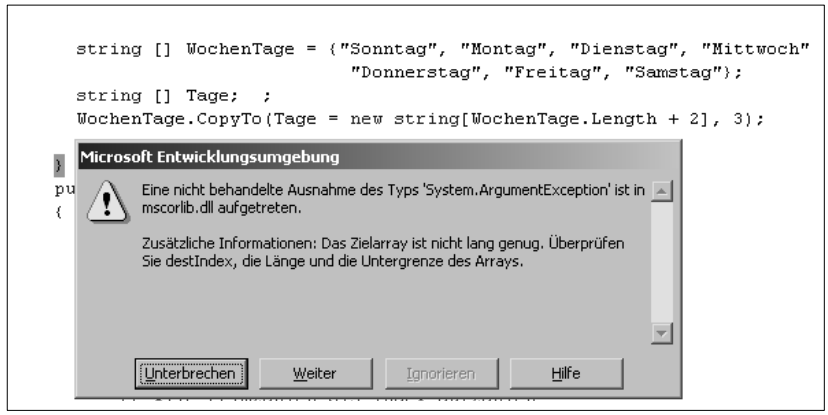


Abbildung 7.9: Ist das Zielarray beim Kopiervorgang zu kurz, kommt es zu einer `System.ArgumentException`

Laufzeitfehler dieser Art lassen sich mit den üblichen Mitteln der Ausnahmebehandlung abfangen, wenn der Zugriff in ein `try...catch`-Konstrukt gepackt wird. Mehr dazu im Kapitel 9, »Ausnahmebehandlung«.

```
try
{
  WochenTage.CopyTo(Tage = new string[WochenTage.Length + 2], 3);
}
```

```
catch(ArgumentException e)
{
  Console.WriteLine(e.Message);  // "Zusätzliche Informationen: Das Ziel...
}
```

Mehrdimensionale Arrays

Die Implementierung des Arraykonzepts über Objekte bringt einen zusätzlichen Freiheitsgrad für mehrdimensionale Arrays. C# unterstützt neben den üblichen mehrdimensionalen Arrays auch *Multiarrays*: Verschachtelte Arrays, deren Elemente wiederum Arrays sind.

Für ein *mehrdimensionales Array* werden die Indexbereiche der einzelnen Dimensionen per Kommaliste vereinbart:

```
int [,] Arr2D = new int [2, 2];
Arr2D[0,0] = 1;
Arr2D[0,1] = 2;
Arr2D[1,0] = 3;
Arr2D[1,1] = 4;
```

Natürlich ist auch hier die Wertlisteninitialisierung zulässig. Die Anweisung

```
int[,,] Arr3D = { {{0,1},{2,3}}, {{4,5},{6,7}}, {{8,9},{10,11}}};
```

vereinbart ein Array mit drei Dimensionen und dem »quaderförmigen« Indexbereich [3, 2, 2]. Die Zuordnung der Initialisierungswerte sieht so aus:

```
Console.WriteLine("{0} {1}", Arr3D[0,0,0], Arr3D[0,0,1]); // 0 1
Console.WriteLine("{0} {1}", Arr3D[0,1,0], Arr3D[0,1,1]); // 2 3
Console.WriteLine("{0} {1}", Arr3D[1,0,0], Arr3D[1,0,1]); // 4 5
...
```

Die Eigenschaft `Rank` spiegelt die Anzahl der Dimensionen bzw. den *Rang* eines Arrays wider. Unter Angabe der jeweiligen Dimension liefern die Methoden `GetLowerBound()` und `GetUpperBound()` den kleinsten und größten zulässigen Indexwert für den Elementzugriff; `GetLength()` ermittelt die Anzahl der Elemente. `GetLowerBound()` liefert für konventionelle C#-Arrays immer 0, da diese – wie schon gesagt – grundsätzlich nullbasiert sind. Damit ist der Wert von `GetLength()` immer um Eins größer als der von `GetUpperBound()`. Der folgende, nicht gerade elegant wirkende Codeauszug verwendet diese Methoden und demonstriert eine Möglichkeit, die Elemente eines zweidimensionalen Arrays in *einer* Schleife auszugeben:

```
int[,] Arr3D = { {0,1}, {2,3}, {4,5},{6,7}};
int upper = Arr3D.GetLength(0) * Arr3D.GetLength(1);
for (int i = 0; i < upper; i++)
  Console.WriteLine(Arr3D[i / (Arr3D.GetUpperBound(1)+1),
                          i % (Arr3D.GetUpperBound(1) +1)]);
```

Multiarrays

Bei *Multiarrays* handelt es sich um Arrays, deren Elemente gewöhnliche (gegebenenfalls auch mehrdimensionale) Arrays sein können oder wiederum Multiarrays mit einem um 1 kleineren Rang. (Multiarrays mit Rang 1 sind somit gewöhnliche Arrays.) Anders als bei mehrdimensionalen Arrays sind die Indexbereiche bei Multiarrays nicht von vornherein festgelegt und können sich auch im Programmverlauf ändern.

Zur Vereinbarung eines Multiarrays mit einem bestimmten Rang notiert man so viele Klammerpaare, wie es der Rang erfordert. Ein Multiarray mit Rang 4 könnte beispielsweise wie folgt vereinbart und initialisiert werden:

```
int [][][][] M = {new int[3][][], new int[8][][], new int[6][][]};
M [0][0] = new int [3][];
M [0][1] = new int [7][];
M [0][0][0] = new int [6];
M [0][0][2] = M [0][0][0];  // Verweis auf bereits bestehende Instanz!
M [0][0][0][0] = 10;  // Wertzuweisung
// M [0,0,0,0] = 10;  // diese Notation ist in C# nicht erlaubt
```

Der Code demonstriert nicht nur, wie die Wertlisteninitialisierung für ein Multiarray aussieht, sondern auch, dass eine vollständige Initialisierung (Besetzung) des Multiarrays nicht erforderlich ist, um mit ihm zu arbeiten. Abbildung 7.10 zeigt die Verhältnisse grafisch. Die zusätzliche Flexibilität hat natürlich ihren Preis. Der Umgang (die Initialisierung, aber auch der sichere Elementzugriff) mit Multiarrays erfordert einiges an zusätzlichem Codier- und Verwaltungsaufwand, und nicht zu vergessen: Auch ein gewisses Verständnis für die unterschiedlichen Schichten eines solchen Arrays.

Ein Multiarray mit Rang 4 ist also zunächst einmal ein Array von Arrayvariablen, die ihrerseits wieder Multiarrays mit Rang 3 sind. Die Elemente dieser Multiarrays sind wiederum Multiarrays mit Rang 2 und erst deren Elemente sind gewöhnliche eindimensionale Arrays, die den vereinbarten Elementtyp haben. Um einen Wert (Instanz) des vereinbarten Elementtyps zuzuweisen, sind also vier Indizes in je eigenen Klammerpaaren zu notieren. Um einem Multiarray mit Rang 4 ein Multiarray mit Rang 2 einzugliedern, sind dagegen exakt zwei Indizes (in je eigenen Klammerpaaren) erforderlich. Da in dem betrachteten Fall einer Objektvariable ein Wert zugewiesen wird, kann der Wert eine neue oder auch eine bereits bestehende Instanz des jeweiligen Arraytyps sein. Mithin sind auch mehrfache Verweise auf ein und dasselbe Elementarray möglich, wie der Beispielcode zeigt.

AnhängerInnen der akademischen Disziplin sei gesagt, dass die C#-Syntax vom Prinzip her auch mehrdimensionale Multiarrays bzw. Multiarrays mit mehrdimensionalen Elementen zulässt. Bei der Verwaltung solcher Datenstrukturen dürfte der Spaß aber bald aufhören:

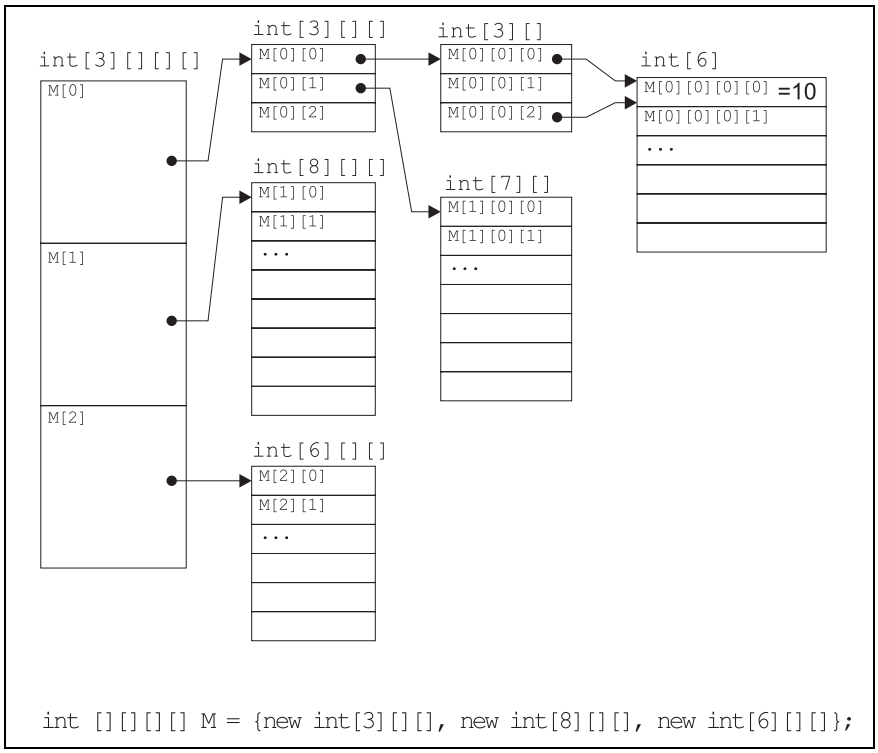


Abbildung 7.10: Vierdimensionales Multiarray

```
int [][,,] MultiArr1x3 = {new int[3,2,2], new int[8,3,1], new int[6,6,2]};
int [,][,] MultiArr2x2 = {{new int[3,2], new int[8,3]},
                          {new int[6,6], new int[3,4]}};
```

Zuweisung und Parameterübergabe

C#-Arrays sind echte Verweistypen. Arrayvariablen sind damit Objektvariablen, denen Instanzen zugeordnet sind, und die Zuweisung ist keine Wertzuweisung, sondern die Zuweisung eines Verweises auf eine bestehende Instanz. Es ist also jederzeit möglich, dass zwei oder mehr Arrayvariablen auf ein und dasselbe Array verweisen. Dies geschieht beispielsweise automatisch, wenn ein Array als Wertparameter an eine Methode übergeben wird. In diesem Fall erhält die jeweilige Methode eine Kopie der Arrayvariablen und nicht des Arrays.

Zuweilen kann es aber auch erforderlich sein, ein Array zu kopieren. Die eingangs erwähnte Methode `CopyTo()` eignet sich nur für eindimensionale Arrays und fordert ein bereits instanziiertes Array hinreichender Größe als Zielobjekt. Um allgemeine Arrays, also auch mehrdimensionale Arrays, Multiarrays und mehrdimensionale Multiarrays zu kopieren, ist die Methode `Clone()` das geeignete Mittel. Da sie aber den Ergebnistyp `object` liefert, ist eine explizite Typumwandlung in den ursprünglichen Arraytyp erforderlich. Die folgende

Zeile erzeugt eine Wertkopie des im vorangehenden Abschnitt diskutierten mehrdimensionalen Multiarrays `MultiArr2x2`:

```
int [,][,] MultiArr2x3Copy =  (int [,][,]) MultiArr2x2.Clone();
```

Ein Destruktoraufruf für die Entsorgung einer nicht mehr benötigten Arrayinstanz erübrigt sich erfreulicherweise. Das erledigt die automatische Freispeicherverwaltung des .NET-Laufzeitsystems, sobald die Geltungsbereiche aller auf die Instanz verweisenden Objektvariablen erloschen sind.

Sortieren, Durchsuchen und teilweises Löschen von Arrays

Eine häufige Aufgabe im Zusammenhang mit Arrays ist das Sortieren und Durchsuchen. Wie zu erwarten, finden sich Standardimplementierungen dieser Funktionalitäten in der umfangreichen .NET-Klassenbibliothek, und zwar in Form statischer Methoden der Klasse `System.Array`. Die gleich achtfach überladene Methode `Sort()` eignet sich zum vollständigen oder teilweisen Sortieren eindimensionaler Arrays mit oder ohne Schlüsselarray; die in vier Varianten angebotene Methode `BinarySearch()` implementiert eine binäre Suche über einem eindimensionalen Array. Voraussetzung für die Anwendung dieser Methode ist, dass das Array zuvor sortiert wurde bzw. ein sortierter Index in Form eines Schlüsselarrays angegeben wird.

Abbildung 7.11 zeigt die prinzipielle Anwendung dieser Methoden und ihre Wirkung in einem Überwachungsfenster des Debuggers.

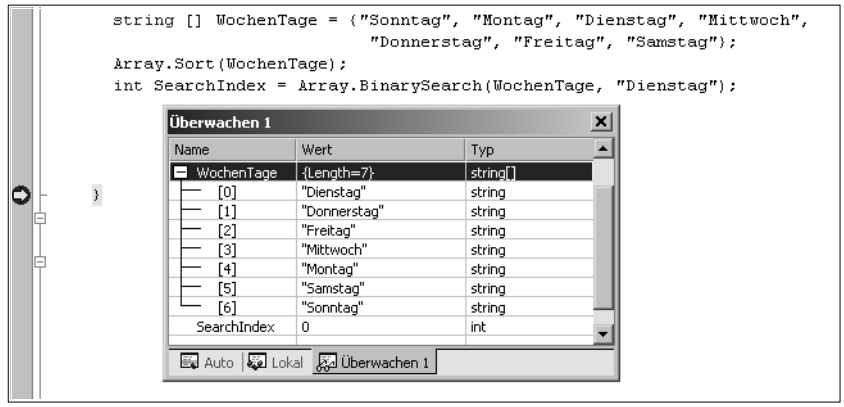


Abbildung 7.11: Das Überwachungsfenster zeigt den Arrayinhalt und das Ergebnis der binären Suche nach Erreichen eines zuvor gesetzten Haltepunkts.

Sowohl `Sort()` als auch `BinarySearch()` bieten darüber hinaus die Möglichkeit, eine individuelle Sortierordnung vorzugeben. Erforderlich dafür ist ein Objekt mit `IComparer`-Schnittstelle. Abbildung 7.12 zeigt den Code einer Routine, die zwei Arrays mit gleichem Inhalt generiert und auf zwei unterschiedliche Weisen sortiert. Im Überwachungsfenster ist das Resultat zu sehen. Man beachte, dass das Array `SortWithIgnoreCase` nicht nur verkehrt

herum, sondern auch ohne Beachtung der Groß-/Kleinschreibung sortiert wird (vgl.: e, E, D, d usw.).

```
public class MyComparer : IComparer
{
  // Absteigend, ohne Unterscheidung der Groß/Kleinschreibung sortieren
  public int Compare(object s1, object s2)
  {
    return -String.Compare((string) s1, (string) s2, true);
  }
}

public void ArraySortExample()
{
  string [] SortStandard = "b,e,d,a,c,E,B,A,D,C".Split(',');
  string [] SortWithIgnoreReverse = (string []) SortStandard.Clone();

  Array.Sort(SortStandard);
  Array.Sort(SortWithIgnoreReverse, new MyComparer());
}
```

Überwachen 1

Name	Wert	Typ
SortStandard	{Length=10}	string[]
[0]	"a"	string
[1]	"A"	string
[2]	"b"	string
[3]	"B"	string
[4]	"c"	string
[5]	"C"	string
[6]	"d"	string
[7]	"D"	string
[8]	"e"	string
[9]	"E"	string
SortWithIgnoreReverse	{Length=10}	string[]
[0]	"e"	string
[1]	"E"	string
[2]	"D"	string
[3]	"d"	string
[4]	"C"	string
[5]	"c"	string
[6]	"b"	string
[7]	"B"	string
[8]	"a"	string
[9]	"A"	string

Abbildung 7.12: Sortieren mit unterschiedlichen Sortierordnungen

Um Ressourcen zu sparen, empfiehlt es sich, umfangreiche Arrayinhalte möglichst umgehend zu löschen, sobald sie nicht mehr benötigt werden. Am einfachsten erledigt ein Aufruf der statischen Methode `Array.Clear()` diese Aufgabe. Er setzt Elemente mit einem Werttyp auf 0 und Elemente mit einem Verweistyp (und darum geht es) auf `null`. Den Elementen zugeordnete Objektinstanzen werden somit freigegeben. Hier ein Beispiel für die Anwendung der Methode:

```
string [] SortStandard = "b,e,d,a,c,E,B,A,D,C".Split(',');
Array.Clear(SortStandard, 2, 4);  // löscht 4 Elemente ab Index 2
```

7.2.3 Typumwandlung

Der Nachteil eines strikten Typsystems ist, dass sich des öfteren die Situation ergibt, Werte verschiedener Typen und Repräsentationen ineinander überführen zu müssen – speziell, wenn Typen stark miteinander verwandt sind, wie etwa die Ganzzahltypen, die sich ja nur im Wertebereich unterscheiden. Wie jede typisierende Sprache sieht auch C# für diesen Fall Schleichwege vor. Anders als bei C/C++ oder Delphi, wo die Sprache unter anderem explizite Typumwandlungen von jedem Zeigertyp in jeden Zeigertyp gestattet, seien sie nun sinnvoll oder nicht, lässt C# eine Typumwandlung zwischen zwei Datentypen aber nur dann zu, wenn dafür ein entsprechender Typumwandlungsoperator explizit definiert ist.

Um einen Wert von einem Ausgangsdatentyp in einen Zieldatentyp zu überführen, sind sechs Fälle zu unterscheiden:

1. *Implizite Typumwandlung* – für den Ausgangsdatentyp ist ein impliziter Typumwandlungsoperator definiert, der diesen in den Zieldatentyp überführt. Der Compiler kann von diesem Operator automatisch Gebrauch machen, um Typunterschiede aufzulösen.

   ```
   float f = 10;    // implizite Typumwandlung int/float
   double d = f;    // implizite Typumwandlung float/double
   ```

2. *Explizite Typumwandlung* – für den Ausgangsdatentyp ist ein Klammeroperator definiert, der diesen in den Zieldatentyp überführt. Klammeroperatoren müssen unter Angabe des gewünschten Zieldatentyps vor den betroffenen Wert gestellt – also explizit notiert – werden.

   ```
   decimal dec = 10M;
   double d = (double) dec;  // explizite Typumwandlung decimal/double
   ```

3. *Typumwandlung über einen dritten Typ* – für den Ausgangstyp ist eine implizite oder explizite Typumwandlung in einen dritten Typ definiert und von dem dritten Typ wiederum eine implizite oder explizite Typumwandlung in den Zieldatentyp. Typumwandlungen dieser Art müssen explizit notiert werden.

4. *Konversionsmethode* – für den Ausgangsdatentyp oder den Zieldatentyp ist eine – gewöhnlich statische – Methode (zuweilen auch: Instanzmethode) definiert, die gegebenenfalls unter Berücksichtigung zusätzlicher Formatinformationen oder Interpretationsmodi einen Wert des Zieldatentyps zurückliefert. Ein gutes Beispiel dafür ist die Methode `ToString()`, die jeder Datentyp von der Basisklasse `object` erbt und die eine standardmäßige String-Repräsentation des jeweiligen Wertes liefert. Für Umwandlungen zwischen Werttypen definiert die `Convert`-Klasse eine Fülle statischer Methoden.

5. *Typumwandlung bei Konstruktion* – vielfach bieten Klassen auch mehrfach überladene Konstruktoren an, die verschiedene Typumwandlungen im Zuge der Konstruktion anbieten. Die Implementierung des jeweiligen Konstruktors verwendet dann meist intern eine der vorgenannten Möglichkeiten für die Typumwandlung. Ein Beispiel dafür wäre der Konstruktor der Klasse `Point`, der einen `Size`-Parameter erwartet.
6. *Keine Typumwandlung möglich* – es ist weder eine direkte noch eine indirekte Typumwandlung zwischen dem Ausgangsdatentyp und dem Zieldatentyp definiert.

Syntax

Damit die explizite oder implizite Typumwandlung von einer Klasse *`Source`* in eine Klasse *`Target`* möglich wird, muss die Klasse *`Source`* einen Typumwandlungsoperator entlang der folgenden Syntax definieren:

```
class Source
{
  [Modifizierer] static {explicit | implicit} operator Target (Source s)
  {
    ... // Konstruktion und Initialisierung von TargetValue
    return TargetValue
  }
```

Implizite Typumwandlung

Die implizite Typumwandlung ist die eleganteste Form, da der Programmierer davon kaum etwas bemerkt. Die integrierten Datentypen und ihre Vererbungsmechanismen definieren eine Reihe impliziter Typumwandlungen, die Abbildung 7.13 veranschaulicht.

In eigenen Klassen werden implizite Typumwandlungen als statische Operatoren mit dem Zieldatentyp als Bezeichner und dem Zusatz `implict` vereinbart. Ein konkretes Codebeispiel findet sich im nächsten Abschnitt.

Eine implizite Typumwandlung kann auch explizit notiert werden. Implizite Typumwandlungen werden für gewöhnlich nur dann definiert, wenn sich ein Datentyp verlustfrei in den Zieldatentyp überführen lässt. Eine Typumwandlung in umgekehrter Richtung ist dann meist verlustbehaftet und – wenn überhaupt – nur explizit möglich.

Explizite Typumwandlung

Explizite Typumwandlungen werden dann definiert, wenn es vom Prinzip her Sinn macht, den Ausgangsdatentyp in den Zieldatentyp zu überführen, der Wertebereich des Ausgangsdatentyps jedoch nur teilweise oder unter Inkaufnahme von Genauigkeitsverlusten bzw. Repräsentationsverlusten in

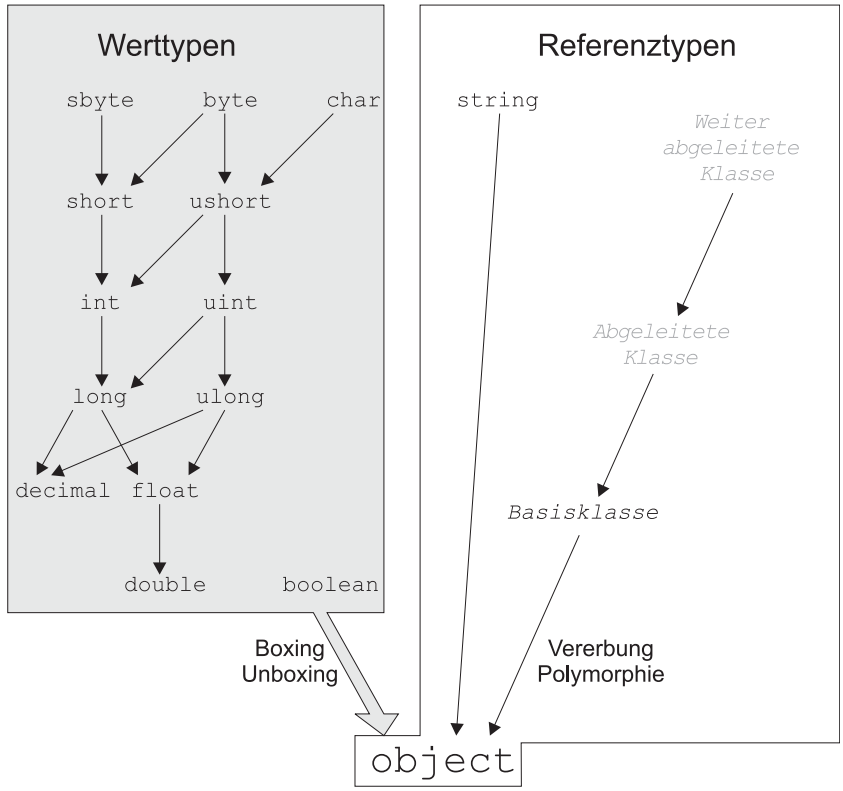


Abbildung 7.13: Typumwandlung zwischen den integrierten Datentypen. In Pfeilrichtung ist eine implizite, gegen die Pfeilrichtung eine explizite Typumwandlung möglich. Typumwandlungen zwischen `object` und Werttypen erfordern Boxing/Unboxing; Typumwandlungen zwischen Verweistypen hingegen einen Vererbungszusammenhang.

den Zieldatentyp überführbar ist. Die andere Richtung ist dann häufig (jedoch nicht immer) problemlos, weshalb explizite Typumwandlungen häufig die Gegenrichtung impliziter Typumwandlungen darstellen.

In eigenen Klassen werden explizite Typumwandlungen auf dieselbe Weise wie implizite vereinbart – wobei statt `implicit` das Schlüsselwort `explict` angegeben wird.

Als Alternative oder Ergänzung zu einem expliziten Typumwandlungsoperator bietet sich die vom Konzept her allgemeinere Konvertierungsmethode an. Konvertierungsmethoden bieten den Vorteil, dass sie beliebige Parameter verarbeiten können, was grundsätzlich die Möglichkeit verschiedener Interpretationen für die Typumwandlung eröffnet.

Codebeispiel – Typumwandlung

Der folgende, sich rein auf die Typumwandlung konzentrierende Codeauszug aus dem Beispielprojekt *Typumwandlung* zeigt die Definition zweier Klassen, deren Objekte in der einen Richtung implizit und in der anderen Richtung explizit typkompatibel sind:

```
class LongBased
{
  private long Counter;
  public LongBased(int i)
  {
    Counter = i;
  }

  // explizite Typumwandlung von LongBased nach IntBased
  public static explicit operator IntBased (LongBased l)
  {
    return new IntBased((int)l.Counter);
  }
}

class IntBased
{
  private int Counter;
  public IntBased(int i)
  {
    Counter = i;
  }

  // implizite Typumwandlung von IntBased nach LongBased
  public static implicit operator LongBased (IntBased i)
  {
    return new LongBased(i.Counter);
  }
}
...
LongBased longB = new LongBased(10);
IntBased intB = (IntBased) longB;      // explizite Typumwandlung
longB = intB;                          // implizite Typumwandlung
```

Typumwandlung bei Vererbung

Ist ein Datentyp `B` von einem Datentyp `A` abgeleitet, definiert der Vererbungsmechanismus eine implizite Typumwandlung von `B` nach `A` und eine explizite Typumwandlung von `A` nach `B`. Die explizite Typumwandlung ist nur dann erfolgreich, wenn das entsprechende Objekt zuvor in den betreffenden Basistyp gewandelt wurde – andernfalls kommt es zu einem Laufzeitfehler. Das folgende Codebeispiel demonstriert das Problem:

```
class Base {}

class Derived : Base {}
...
public void TestCast()
{
  Base b = new Derived();   // Implizite Typumwandlung
  Derived d = (Derived) b;  // Explizite Typumwandlung nur deshalb möglich,
                            // weil zuvor implizite stattgefunden hat
  Base b1 = new Base();
  Derived d1 = (Derived) b1;  // Laufzeitfehler
}
```

Wenn ein Vererbungszusammenhang zwischen zwei Objekten besteht, werden explizite Typumwandlungen in Richtung der abgeleiteten Klasse zwar vom Compiler akzeptiert – es stellt sich aber erst zur Laufzeit heraus, ob sie legitim sind. Bei der expliziten Typumwandlung entlang der Vererbungslinie spielt die in der Instanz gespeicherte Typinformation eine wichtige Rolle. Diese Typinformation bleibt bei der impliziten Typumwandlung gegen die Vererbungslinie natürlich erhalten. Eine Typumwandlung entlang der Vererbungslinie ist daher nur dann legitim, wenn das entsprechende Objekt tatsächlich den gewünschten Zieltyp bzw. einen davon abgeleiteten Typ hat.

Aus genau diesem Grund lassen sich über den Vererbungsmechanismus auch keine Typumwandlungen »erschleichen«, die als solche nicht definiert sind. Abbildung 7.14 zeigt den prototypischen Fall.

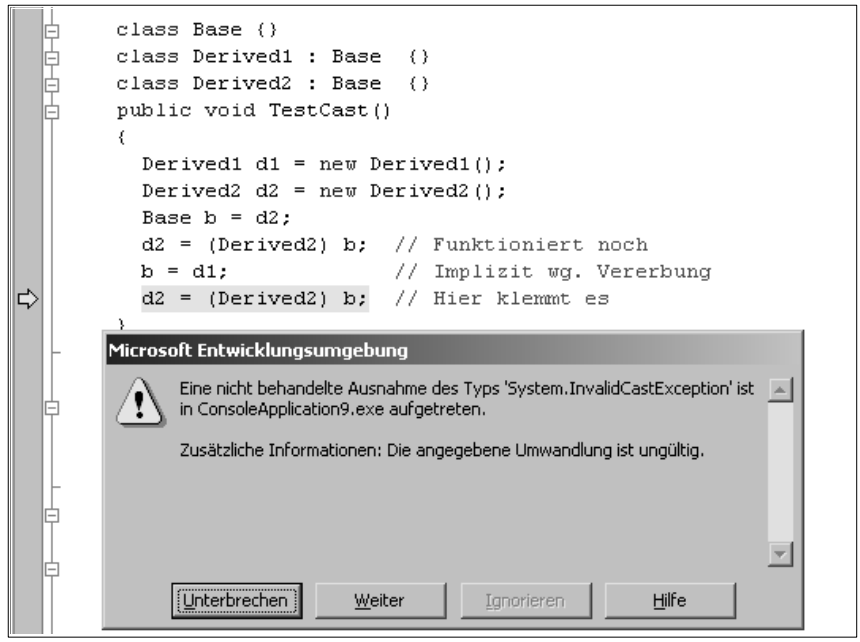


Abbildung 7.14: Die »Typumwandlung um die Ecke« per Vererbungsmechanismus scheitert.

Boxing und Unboxing

Wie Abbildung 7.13 zeigt, ist `object` als »Mutter« aller Klassen ein dankbarer Zieldatentyp für die implizite Typumwandlung aller möglichen Datentypen, gleich, ob Wert- oder Verweistyp. Was die Umwandlung von Verweistypen nach `object` und wieder zurück betrifft, so gibt es dem, was im vorangegangen Abschnitt »Typumwandlung bei Vererbung« zu lesen ist, nichts hinzuzufügen.

Bei Werttypen, deren Instanzen ja bekanntlich als lokale Variablen »nackt« auf dem Stack liegen können, liegt der Fall etwas anders. Diese Werte führen an sich keine Typinformation mit und lassen sich auch nicht so ohne

weiteres als Verweistyp behandeln oder auf dem Heap speichern. Nachdem Werttypen aber gleichfalls via `System.ValueType` von `object` abstammen (so will es die .NET-Klassenhierarchie) und von diesen Datentypen eine beachtliche Mitgift in die Wiege gelegt bekommen – man denke dabei etwa an Methoden wie `ToString()` oder `GetType()` –, muss ein Mechanismus her, der zwischen den beiden Erscheinungsformen vermittelt. Denn: Instanzmethoden – gleich, ob ererbt oder nur für den jeweiligen Datentyp definiert (vgl. `struct`-Instanz) – können nur über das Gerüst einer Verweistypinstanz aufgerufen werden, das die erforderliche Methodentabelle bereitstellt.

Je nach Richtung der Verwandlung heißt dieser Mechanismus Boxing oder Unboxing.

Boxing

Boxing »verpackt« einen Werttyp in einen passenden Verweistyp. Der Vorgang entspricht damit der impliziten Typumwandlung in den jeweiligen Verweistyp, wobei die automatisch generierte Verweistypinstanz ein Datenfeld mit dem Werttyp enthält, das bei Konstruktion mit einer Kopie des Wertes initialisiert wird:

```
struct MyStruct
{
  public float a, b, c, d;
}
...
MyStruct ms = new MyStruct();
object o = new MyStruct();    // Boxing
System.ValueType vt = new MyStruct(); // Boxing
System.Collections.ArrayList al = new System.Collections.ArrayList();
al.Add(o);          // als object-Instanz
al.Add(ms);         // Boxing als MyStruct
al.Add(ms.a);       // Boxing als float
al.Add(10.5);       // Boxing als double
al.Add(vt);         // Boxing als MyStruct
```

Die gänzlich auf dem Heap gespeicherte Verweistypinstanz ist erstens völlig unabhängig vom ursprünglichen Wert (der ja auf dem Stack liegen kann) und trägt zweitens eine Laufzeittypinformation, die es ermöglicht, den Wert des Datenfelds jederzeit per Unboxing wieder einer Variable bzw. einem Datenfeld mit dem ursprünglichen Datentyp zuzuweisen.

Unboxing

Unboxing »entpackt« einen zuvor als Verweistyp verpackten Werttyp und entspricht damit der expliziten Typumwandlung vom Verweistyp in den ursprünglichen Datentyp. Dabei ist die Angabe eines Typumwandlungsoperators obligatorisch. Beachten Sie, dass Unboxing grundsätzlich *nur* in den ursprünglichen Datentyp möglich ist – andernfalls ergibt sich ein Laufzeitfehler. Für Umwandlungen in (implizit oder explizit) kompatible Daten-

typen müssen Sie einen weiteren Typumwandlungsoperator notieren, wie die folgende Codesequenz in Fortführung des vorigen Beispiels zeigt:

```
ms = (MyStruct) al[4];
ms.a = (float) (double) al[3];  // wurde als double eingelagert!
ms.b = (float) al[2];
MyStruct ms1 = (MyStruct) al[1];
MyStruct ms2 = (MyStruct) al[0];  // object-Element raus, dann Unboxing
```

7.3 Variablen

Der symbolische Umgang mit Daten ist die Grundlage der hochsprachlichen Programmierung schlechthin. Variablen stellt man sich am besten als Behältnisse für Werte mit einem bestimmten Datentyp vor, welche diese Werte speichern, benennen und Manipulationen zugänglich machen.

7.3.1 Syntax

Die Syntax für die Variablenvereinbarung lautet:

```
[Modifizierer] Datentyp Bezeichner [= InitAusdruck [, Bezeichner [= ...]]];
```

Dabei gilt die folgende Regel: Eine Variable gehört *immer* dem Kontext einer Klasse an – die Vereinbarung von Variablen außerhalb einer Klassendefinition, also auf Namensbereichsebene – ist in C# nicht vorgesehen.

Modifizierer

Als Modifizierer sind die Zugriffsmodifizierer `private`, `public`, `internal`, `protected` und `protected internal` mit der üblichen Bedeutung zulässig (vgl. Abbildung 7.15).

Variablen, die auf Klassenebene ohne den Modifizierer `static` vereinbart werden, heißen *Instanzfelder* oder *Objektfelder*. Findet die Vereinbarung nicht auf Klassenebene, sondern innerhalb einer Methode statt, spricht man von *lokalen Variablen*. Für lokale Variablen kann kein Zugriffsmodifizierer angegeben werden, weil sie grundsätzlich nur innerhalb der jeweiligen Methode sichtbar sind.



Die von Delphi her bekannte Möglichkeit zur Definition global gültiger, aber nur innerhalb einer Methode sichtbarer Variablen (`const Typ = Wert;`) bietet C# nicht. (Interessanterweise setzen derartige Konstrukte ab der Delphi-Version 6 einen Compilerschalter voraus, der standardmäßig nicht aktiviert ist – bei Borland scheint man also in die selbe Richtung gedacht zu haben.)

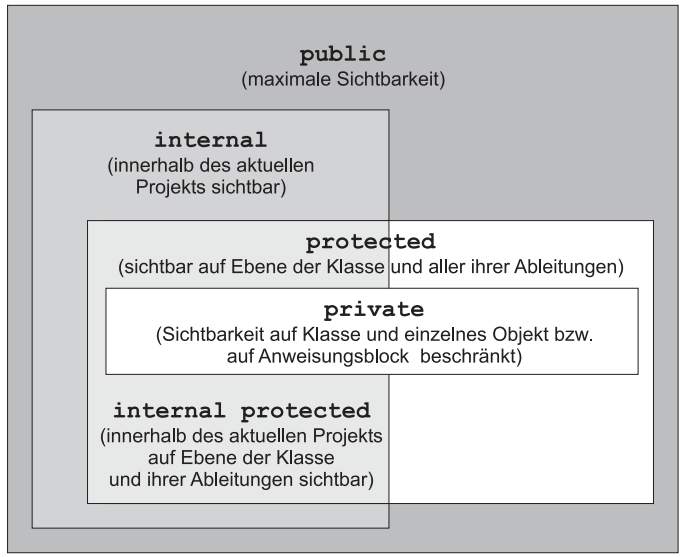


Abbildung 7.15: Wirkungsweise der Zugriffsmodifizierer

Bei nicht angegebenem Zugriffsmodifizierer wird eine Variable implizit als `private` vereinbart:

```
int MyVar = 10;       // implizit private
```

Eine Variable kann optional im Rahmen ihrer Deklaration mit einem Ausdruck initialisiert werden. Falls das nicht geschieht, setzt der Compiler durch, dass die Variable vor ihrem ersten Gebrauch als Rechtswert eine Initialisierung per Zuweisung oder als `out`-Parameter erfährt.

Bei Vereinbarung mehrerer Variablen innerhalb der gleichen Anweisung kann die sofortige Initialisierung für jede Variable optional erfolgen, wobei aber alle Variablen denselben Datentyp und dieselben Modifizierer erhalten (VB6-Programmierer aufgepasst). Es ist also insbesondere nicht zulässig zu schreiben:

```
// private double MyPrivVal = 10.1, public MyPublVal;   // Fehler
```

Statische Datenfelder

Durch Angabe des `static`-Modifizierers wird ein Datenfeld zum *statischen Datenfeld* oder kurz *Klassenfeld* der jeweiligen Klasse und somit zum Bestandteil der statischen Instanz. Im Gegensatz zu C/C++ (und Delphi) kennt C# für Methoden *keine* statischen Variablen. Informationen, die zwischen zwei Aufrufen einer Methode erhalten bleiben sollen, müssen somit immer auf Ebene der Instanz (oder der Klasse) verwaltet werden, denn Datenfelder haben aus Sicht einer Methode globalen Status. Beachten Sie, dass statische Methoden nur auf statische Datenfelder zugreifen können, während Instanzmethoden beide Arten von Datenfeldern zur Verfügung stehen.

Initialisierung und Wertzuweisung

Statische Datenfelder und Instanzfelder werden implizit mit dem Wert 0 (Werttypen) bzw. `null` (Verweistypen) initialisiert. Für den Datentyp `string` erfolgt eine Initialisierung mit dem Wert `""`. Eine abweichende Initialisierung ist bei Vereinbarung sowie per Konstruktor möglich.

Statische Datenfelder dürfen bei Vereinbarung nur mit konstanten Werten oder Ausdrücken initialisiert werden. Hierzu sind neben `const`-Werten auch Werte bzw. Ausdrücke zu rechnen, die aus anderen statischen Datenfeldern gebildet sind. (Kriterium ist, dass alle beteiligten Werte zur Übersetzungszeit bekannt sein müssen). Die Initialisierung mit Werten, die erst zur Laufzeit bekannt sind, kann durch Bereitstellung eines – grundsätzlich parameterlosen – statischen Konstruktors erfolgen (vgl. Seite 282).

Die Initialisierungsvereinbarung von Instanzfeldern hat vom Prinzip her dasselbe Muster: Erlaubt sind Ausdrücke, die aus Konstanten, statischen Datenfeldern und anderen Instanzfeldern gebildet sind. Allerdings lassen sich Instanzfelder so – indirekt über statische Datenfelder – auch mit Werten initialisieren, die erst zur Laufzeit bekannt sind.

```
class MyClass
{
  static System.Collections.ArrayList al; // null
  System.Collections.ArrayList al1 = al;  // Initialis. mit Laufzeitwert
  static IntBased()                 // statischer Konstruktor
  {
    al = new System.Collections.ArrayList(); // Instanziierung
  }
  ...
}
```

Für die Initialisierung eines Instanzfelds per Konstruktor gibt es hingegen keine Einschränkungen. Mehr dazu finden Sie im Abschnitt »Instanziierung – new-Operator und Konstruktor« ab Seite 251.

Sonstige Modifizierer

Die Modifizierer `const` und `readonly` versehen die Variable mit einem Schreibschutz und verleihen ihr – faktisch oder quasi – den Status einer Konstanten (vgl. hierzu den Abschnitt »Literale, Konstanten und readonly-Werte«).

Der Modifizierer `new` spielt bei der Vererbung eine Rolle. Er wird angegeben, um ein gleichnamiges Datenfeld (allgemeiner: Member) der Basisklasse zu überschreiben und erneut mit anderer Bedeutung – im Allgemeinen mit einem neuen Datentyp oder auch nur mit einem anderen Initialisierungswert – zu vereinbaren:

```
class Basisklasse
{
  public string DerName = "Das ist die Basisklasse";
}

class AbgeleiteteKlasse: Basisklasse
{
  public new string DerName = "Das ist die abgeleitete Klasse";
}
class WeiterAbgeleiteteKlasse: AbgeleiteteKlasse
{
  public new int DerName = 10;
}
```

Das (grobe, weil nur für Methoden vorgesehene) Delphi-Äquivalent von `new` *in diesem Kontext ist* `reintroduce`.



Bezeichner

Für die Bezeichnerwahl von Variablen gelten die allgemeinen Benennungsregeln von C#. Weiter einschränkend empfiehlt es sich, die von Microsoft für die .NET-Programmierung vorgeschlagene Benennungskonvention zu beachten. (Beide Regelsätze sind im Abschnitt 5.2 »Benennungskonventionen«, ab Seite 104, weiter ausgeführt.)

7.3.2 Geltungsbereich

Um die Eindeutigkeit der Benennung zu gewährleisten, muss der Bezeichner einer Variablen in ihrem jeweiligen Geltungsbereich eindeutig sein. Das wirft natürlich generell erst einmal die Frage auf, was man sich unter einem Geltungsbereich vorzustellen hat und welche praktische Dimension er hat.

Der *Geltungsbereich* einer Variablen ist der Bereich des Codes, in dem die Variable bekannt ist, die einen Wert zugewiesen bekommen kann (man denke hier auch an die Umbenennung bei `ref`- und `out`-Parametern von Methoden) oder einen Wert verkörpert.

Klassen- und Objektebene

Auf Klassen- bzw. Objektebene wird der Geltungsbereich von Datenfeldern über den Zugriffsmodifizierer geregelt. Liegt eine `private`-Vereinbarung vor, ist das Datenfeld nur für den Code der Klasse selbst sichtbar. Es wird weder vererbt, noch ist es für den Besitzer eines Objekts der Klasse (bzw. im Falle einer statischen Vereinbarung als Teil der statischen Instanz) sichtbar. Bei einer Vereinbarung mit `protected`, `internal` oder `internal public` ist das Datenfeld gleichfalls nur im jeweiligen Geltungsbereich sichtbar, was zu interessanten Effekten im Zusammenhang mit der Vererbung führen kann. Beachten Sie dazu das folgende Codebeispiel:

```
class Base
{
  protected string Nae = "Das ist die Klasse Base";
}

class Derived: Base
{
  private new int Nae = 20;   // Feld wird überschrieben
}

class FurtherDerived : Derived
{
  public FurtherDerived()
  {
    Nae += " ergänzt von der Klasse FurtherDerived";  // von Base vererbt
  }
}
```



Die Sicht auf ein ererbtes Datenfeld kann auf Klassen- oder Objektebene dadurch verdeckt sein, dass eine Überschreibung des Datenfelds mit einer eigenen Variante vorliegt. Um den Zugriff auf das ererbte Datenfeld aufrecht zu erhalten, besteht aber immer die Möglichkeit, den Bezeichner mit dem Schlüsselwort `base` *zu qualifizieren:*

```
class Base
{
  protected string Nae = "Das ist die Basisklasse";
}

class Derived: Base
{
  private new int Nae = 20;   // Feld wird überschrieben
  public Derived()
  {
    Nae = base.Nae.Length;   // Zugriff auf Datenfeld der Basisklasse
  }
}
```

7.3.3 Lebensdauer

Die Lebensdauer von Variablen und Datenfeldern hängt eng mit ihrem Geltungsbereich zusammen. Lokale Variablen von Methoden, deren Werte ja auf dem Stack gespeichert sind, existieren so lange, wie der Stack-Rahmen des für die Vereinbarung verantwortlichen Anweisungsblocks besteht. Objekte existieren hingegen im Allgemeinen etwas länger als die Objektvariablen, die darauf verweisen. Die .NET-Freispeicherverwaltung vernichtet nur solche Objekte, auf die keine einzige Objektvariable mehr verweist, arbeitet aber asynchron und wird aus der Sicht des Programms also in unregelmäßigen und nicht exakt vorher bestimmbaren Zeitabständen aktiv.

Datenfelder – auf Klassenebene vereinbarte Variablen

Bei der Lebensdauer von Datenfeldern muss zwischen statischen Datenfeldern und Instanzfeldern unterschieden werden. In beiden Fällen gilt aber: Ein Datenfeld existiert so lange, wie die Instanz existiert, der es zugeordnet ist.

Statische Datenfelder

Statische Datenfelder werden im Gefolge der statischen Instanz der Klasse angelegt und initialisiert, sobald der erste Zugriff auf die statische Instanz oder die erste Instanziierung eines Objekts der Klasse stattfindet. Mit anderen Worten, sie sind immer verfügbar und existieren während des gesamten Programmlaufs.

Instanzfelder

Instanzfelder werden zusammen mit neuen Objekten der Klasse angelegt und initialisiert, wobei jedes Objekt seinen eigenen Satz erhält. Die Lebenszeit von Instanzfeldern erlischt mit Zerstörung des Objekts.

Lokale Variablen

Interessanter wird die Betrachtung der Lebensdauer bei den lokalen Variablen. Da lokale Variablen auf der Ebene eines Anweisungsblocks in einer Methode vereinbart werden, beginnt ihr Dasein, wenn die Ausführungskontrolle auf ihre Vereinbarung trifft, und endet, wenn der Block abgearbeitet ist. In C# spricht nichts dagegen, lokale Variablen auch mitten im Code einer Methode zu vereinbaren (viele Programmiersprachen wie C, VB und Delphi/Pascal fordern, dass dies zu Beginn eines Anweisungsblocks zu geschehen hat), wobei der Compiler auch in diesem Fall erzwingt, dass eine rechtzeitige Initialisierung vor dem ersten Einsatz als Rechtswert stattfindet.

```
... // Anweisungsblock in einer Methode
int i = 0;
MyClass mc = new MyClass();
while (i++ < 10)
  mc.DoOutput(i);
int j = i;             // wird vom Compiler nicht moniert
...
```

In Kontrollstrukturen

Allerdings gibt es einige generelle Einschränkungen im Zusammenhang mit Kontrollstrukturen bzw. Anweisungsblöcken von Kontrollstrukuren zu beachten: Die Vereinbarung einer lokalen Variablen in einem Anweisungsblock ist nur gestattet, wenn derselbe Variablenbezeichner nicht auch auf übergeordneter Ebene verwendet wird. Die Verwendung in anderen Anweisungsblöcken auf gleicher Ebene ist hingegen gestattet.

So ist es ein Fehler zu schreiben:

```
int o;
int i = 1;
while (i++ < 10)
{
  int o = i;      // wird vom Compiler moniert, da o schon bekannt ist
 ...
}
```

und obwohl die folgende »Variante« gar nicht schlecht aussieht

```
int o;
int i = 1;
while (i++ < 10)
  int otto = i;   // wird vom Compiler nicht akzeptiert
```

gibt es noch ein weiteres Problem – das allerdings in Praxis kaum vorkommen wird: Damit die vereinbarte Variable `otto` keine Probleme wegen Mehrfachvereinbarung macht, sind geschweifte Klammern erforderlich. Nur so bleibt der Geltungsbereich auf den Anweisungsblock der Kontrollstruktur beschränkt. Mit anderen Worten: `otto` ist außerhalb des Anweisungsblocks unbekannt – und sichtlich überflüssig:

```
int o;
int i = 1;
while (i++ < 10)
{
  int otto += i;   // wird vom Compiler akzeptiert
}
```

Möglich, ja sogar ausdrücklich empfohlen, ist die Vereinbarung von Zählvariablen im Initialisierungszweig von `for`-Schleifen bzw. von Iterationsvariablen in `foreach`-Schleifen:

```
...
for(int i = 0; i < 10; i++)
  mc.DoSomething(i);
for(int i = 100; i < 110; i++)
  mc.DoSomething(i);
...
```

Der Geltungsbereich der lokalen Variablen bleibt hier auf den Anweisungsblock der Kontrollstruktur beschränkt.

In geklammerten Anweisungsblöcken

In Anweisungsblöcken, beispielsweise in `checked`-, `unchecked`- oder `using`-Anweisungsblöcken, aber auch in einfach geklammerten Blöcken lassen sich jederzeit lokale Variablen vereinbaren, deren Geltungsbereich an der schließenden Klammer erlischt.

```
unchecked
{
  int i = 10;
...
}
checked
{
  int i = 15;
...
}
{
  int i = 5;
...
}
```

In allen genannten Fällen ist es dabei allerdings wichtig, dass keiner der in einem untergeordneten Anweisungsblock vereinbarten Variablenbezeichner auch in einem übergeordneten Anweisungsblock vereinbart wird (weder vorher noch nachher). Alle in diese Richtung zielenden Initiativen scheitern daher schlicht am C#-Compiler. Sowohl so:

```
int i = 1;
{
  int i = 20;                // Fehler: i ist nicht eindeutig
}
```

als auch so:

```
int i = 10;
for(int i = 0; i<10; i++)    // Fehler: i ist nicht eindeutig
  mc.DoSomething(i);
```

als auch so:

```
for(int i = 0; i<10; i++)
  mc.DoSomething(i);
int i = 10;                   // Fehler: i ist nicht eindeutig
```

Und das, obwohl `i` nach Beendigung des Anweisungsblocks nicht mehr bekannt ist:

```
for(int i = 0; i < 10; i++);
  mc.DoSomething(i);
int j = i;                    // Fehler: i ist nicht mehr bekannt
```

Verdecken auf Klassenebene vereinbarter Variablen

Während das versehentliche Verdecken von Variablen über verschachtelte Funktionsdefinitionen nicht nur für Pascal-Derivate, sondern auch bei C/C++ immer ein großes Thema war, ist für C# nur in einem einzigen Fall eine Kollision dieser Art möglich: Nachdem auf Klassenebene vereinbarte Bezeichner grundsätzlich auch innerhalb von Methoden als *globale* Variab-

len bekannt sind, kann die Vereinbarung einer lokalen Variablen die Verdeckung eines gleichnamigen Datenfelds bewirken. Um in diesem Fall weiterhin auf das Datenfeld zugreifen zu können, ist eine entsprechende Qualifizierung des Bezeichners erforderlich: für statische Felder mit dem Klassennamen, für Instanzfelder mit `this`:

```
class MyClass2
{
  int i;
  static int j;
  static MyClass2()
  {
    int j = 10;
    MyClass2.j = j;
  }
  MyClass2(int i)
  {
    this.i = i;
  }
}
```



Vermeiden Sie einfach Doppelbenennungen, die zum Verdecken von Variablen führen können. Das reduziert die Fehleranfälligkeit und verbessert die Lesbarkeit Ihres Codes.

7.3.4 Literale, Konstanten und readonly-Werte

Wenn es um Konstanten geht, findet man in der Fachliteratur selten eine saubere Trennung zwischen den so genannten *literalen Konstanten* und den *Wertkonstanten*. Das liegt wohl daran, dass diese beiden Konzepte in der C-Tradition nur ein Lauf des Präprozessors trennt und sie dort deshalb faktisch dasselbe sind. In der Pascal-Tradition wurde hier schon immer sauberer unterschieden (oder nachträglich korrigiert: Wie bereits erwähnt, sind Zuweisungen an über *const* definierte Variablen seit Delphi 6 nur noch nach Setzen einer standardmäßig abgeschalteten Compileroption möglich.)

Literale Konstante

Eine *literale Konstante* oder schlicht *Literal* ist ein im Quellcode notierter Wert, für den keine symbolische Benennung (Bezeichner) vereinbart ist. Literale Konstanten sind nur als Rechtswerte verwendbar. Für die Notation des Wertes sind – je nach anvisiertem Datentyp – sprachspezifische Regeln und Formatfestlegungen zu beachten. Mit anderen Worten: Ein Literal ist eine Zeichenfolge im Quelltext, die der Compiler als unbenannten Wert eines bestimmten Datentyps interpretiert.

Wertkonstante

Eine *Wertkonstante* ist ein Bezeichner, dem eine literale Konstante als Wert zugeordnet ist und der diese symbolisch verkörpert. Die Zuordnung darf nur einmal bei Vereinbarung erfolgen und ist ab dann unveränderlich. Im weiteren Programmverlauf darf eine Wertkonstante nur noch als Rechtswert verwendet werden.

Syntax

Konstantenvereinbarungen haben in C# die folgende Syntax:

```
[Modifizierer] const IntegrDatentyp KonstBezeichner = ConstAusdruck;
```

Dabei gelten folgende Regeln:

- Ohne Angabe eines Zugriffsmodifizierers wird die Konstante implizit als `private` vereinbart.
- Eine Konstante muss bei Deklaration initialisiert werden. (Eine Initialisierung im Konstruktor ist nicht möglich.) Als Initialisierungswerte sind nur konstante Ausdrücke zulässig, das heißt: literale Konstanten, andere Wertkonstanten sowie daraus unter Verwendung der Standardoperatoren gebildete Ausdrücke.
- Jenseits von Zugriffsmodifizierern ist nur der `new`-Modifizierer zulässig. Er ermöglicht es, die Definition (also Wert *und* Datentyp) einer ererbten Wertkonstanten in einer abgeleiteten Klasse zu ändern.
- Als Datentypen stehen nur integrierte Datentypen (`int`, `uint`, `byte` ... `string` usw.) zur Verfügung.
- Konstanten sind implizit statische Elemente und als solche über den Klassennamen (statische Instanz) anzusprechen. Eine explizite Vereinbarung mit `static` ist nicht zulässig.

Ein Beispiel:

```
MyClass
{
  public const uint MyConst = 10;
  public const double MyConst1 = MyConst * 0.7;  // konstanter Ausdruck
  private const string MyConst2 = "Hallo";
}

MySecondClass : MyClass                // abgeleitete Klasse
{
  public new const uint MyConst = 12;  // überschreibt Basisklassenelement
...
}
```

Einzelheiten darüber, in welchem Format literale Konstanten für einen gegebenen integrierten Datentyp zu notieren sind, finden Sie bei Vorstellung der einzelnen Datentypen. Da die Initialisierung einer Konstanten bereits zum Übersetzungszeitpunkt stattfindet, darf der Initialisierungsausdruck weder Methodenaufrufe noch Variablen enthalten, da deren Werte erst zur Laufzeit verfügbar sind.

Anwendung

Der Vorteil der Wertkonstante liegt darin, dass ihr Wert zentral an einer Stelle definiert wird (und dort auch für den nächsten Compilerlauf geändert werden kann). Deshalb sollte man also immer mit Wertkonstanten arbeiten und literale Konstanten nur in Ausnahmefällen oder experimentellem Code einsetzen. (Die allgemein akzeptierte Ausnahme von dieser Regel sind Strings in Rückmeldungen eines Programms – und das auch nur so lange, wie man sie nicht in eine andere Sprache übersetzen muss.) Der Export konstanter Werte ist in C# ausschließlich in Form von `const`-Wertkonstanten möglich.

Ansonsten sind Wertkonstanten und literale Konstanten von der Anwendung her wechselseitig austauschbar. Sie dürfen überall da eingesetzt werden, wo ein Rechtswert erwartet wird, also in Zuweisungen, als Index, als *by-value*-Parameter in Methodenaufrufen usw. Nicht zulässig sind sie hingegen für `ref`- und `out`-Parameter, weil dort ein Linkswert erwartet wird.

```
double d = Math.Sqrt(MySecondClass.MyConst);
```



Speziell für Fallunterscheidungen bietet sich die Möglichkeit an, umfangreichere Konstantensammlungen als `enum`*-Datentyp zu vereinbaren.*

Vielleicht werden Sie sich wundern, diesen Abschnitt *nach* dem Abschnitt »Variablen« zu finden. Das hat natürlich seinen Grund: Aus dem Blickwinkel von C# ist es angemessener, Konstanten als Variablen »mit Schreibschutz« aufzufassen, denn als benannte Literale. Wenn Sie von C/C++ her kommen, werden Sie es vielleicht noch gewohnt sein, Konstanten über die Präprozessoranweisung `#define` zu definieren, wiewohl das Schlüsselwort `const` auch in C++ existiert (und auch von dort entlehnt ist). Mit Blick auf die komponentenorientierte Programmierung sollte es dort auch bevorzugt verwendet werden, da hier mit Konstanten auf Ebene des Präprozessors allein nicht auszukommen ist. In C++ hat das Schlüsselwort `const` zudem noch andere Aufgaben im Bereich der Parameterdeklaration für Funktionen, die für C# nicht übernommen wurden. `#define`-Anweisungen findet man – Gott sei Dank – in C# nur noch im Zusammenhang mit bedingter Compilierung.

Readonly-Werte

Der Modifizierer `readonly` tut, was sein Name verspricht: Er versieht ein Datenfeld mit einem Schreibschutz und bewirkt damit vom Effekt her dasselbe wie eine `const`-Vereinbarung. Faktisch besteht natürlich ein Unterschied – warum sollte man auch zwei Schlüsselworte für ein und dasselbe Konzept haben. Während die Initialisierung von `const`-Vereinbarungen bereits zur Übersetzungszeit geschieht und aus diesem Grund nur Ausdrücke enthalten kann, die ihrerseits konstant und zur Übersetzungszeit auch berechenbar sind, unterliegen die Initialisierungsausdrücke für `readonly`-Vereinbarungen vom Prinzip her keinerlei Beschränkungen: Sie werden erst zur Laufzeit berechnet. Die Versiegelung eines `readonly`-Datenfeldes erfolgt unmittelbar nach Konstruktion. Damit kann die Initialisierung entweder gleich bei Vereinbarung oder aber im Konstruktor geschehen – und dort beispielsweise auch Methodenaufrufe beinhalten.

Die Versiegelung bewirkt, dass `readonly`-Datenfelder in gewöhnlichen Methoden nur als Rechtswerte, nicht jedoch als Linkswert oder als `ref`- bzw. `out`-Parameter verwendbar sind.

```
class MyClass
{
  readonly static int j = 20 ;
  static MyClass()
  {
    MyRandomClass moc = MyRandomClass();
    moc.InitRandom(out j)     // das geht nur im Konstruktor!;
  }
}
```

8 Objekte, Klassen, Vererbung

Dieses Kapitel widmet sich in aller Ausführlichkeit dem mit dem Schlüsselwort `class` verbundenen Klassen- und Objektbegriff von C#. Schwerpunkt ist der strukturelle Aufbau und die formale Organisation der Daten und des Codes. Sie finden in diesem Kapitel den gesamten Hintergrund, den Sie für die praktische Programmierung mit der .NET-Klassenhierarchie und eigenen Klassen benötigen.

Was die Repräsentation der Daten in Datenfeldern betrifft, so verlässt sich die Darstellung weitgehend auf das vorangehende Kapitel, das die integrierten Datentypen, die Konstanten und das Variablenkonzept – also die Grundlagen für komplexe Datentypen – erörtert. Der Klassenbegriff aus diesem Kapitel rundet diese Konzepte wiederum »nach oben hin« ab.

Ausführlicher widmet sich das Kapitel den Methoden, also der prozeduralen Dimension einer Klasse. Es diskutiert die verschiedenen Ausprägungen einer Methode – als unspezialisierte Methode, als Konstruktor, als Eigenschaft, als Operator und als Destruktor – und natürlich auch die für ein Verständnis des Methodenkonzepts zentralen Mechanismen des Überladens und Überschreibens. (Obwohl nicht Thema dieses Kapitels, so sei an dieser Stelle angemerkt, dass sich der Methodenbegriff der `class`-Klasse nahezu Eins zu Eins auf die `struct`-Klasse übertragen lässt, wenn man die Vererbung und die damit verbundenen Konzepte wie virtuelle Methoden außer Acht lässt.)

Zweites großes Thema des Kapitels ist die Vererbung. Der Text zeigt einerseits die an den verschiedenen Stellen in der Sprachsyntax zu findenden Haken und Ösen des Vererbungsmechanismus auf und diskutiert andererseits in geschlossenerer Darstellung die verschiedenen Facetten und Möglichkeiten, die das Vererbungskonzept von C# in sich birgt. Ein wichtige Rolle spielen in diesem Zusammenhang auch abstrakte Klassen und Schnittstellen, von denen die .NET-Klassenbibliothek regen Gebrauch macht. Ein Verständnis dieser Konzepte ist damit nicht nur für das abstrakte Anwendungsdesign förderlich.

8.1 Objekte vs. Klassen

Wenn etwas aus der heutigen Programmierwelt nicht mehr wegzudenken ist, dann sind das Klassen und Objekte. Während die klassische imperative Programmierung mit ihrem so genannten »anwendungszentrierten Ansatz« noch die strikte Trennung zwischen Daten und Code predigte und weitgehend monolithische Anwendungen hervorbrachte, die über oft hochgradig komplexen Datenstrukturen operierten, hat sich die objektorientierte Programmierung geradezu die gegenteilige Ansicht zu eigen gemacht: Sie geht davon aus, dass die Repräsentation und die Manipulation von Daten eng zusammengehören und nichts weiter als unterschiedliche Aspekte des gleichen Dings sind – nämlich des Objekts.

In C# programmieren, ist objektorientierte Programmierung par excellence: Jeder Datentyp ist hier eine Klasse und jeder Wert ein Objekt – vom einfachen Wahrheitswert bis zur gesamten Anwendung ein einziges, durchgehendes Konzept.

8.1.1 Objektbegriff

Wer von C++ oder von Delphi her kommt, bei dem wird der Objektbegriff von C# (bzw. .NET) auf gut bestellten Boden fallen. Visual Basic-Programmierer haben es hingegen schwerer, sie betreten stellenweise echtes Neuland, aller Objektorientierung ihrer Sprache zum Trotz. Obwohl Visual Basic auch bereits einige Versionen vor VB.NET mit dem Label »Object Basic« verkauft wurde und darin Klassen, Objekte und auch Vererbung in Form der Schnittstellenvererbung zunehmend eine Rolle spielten, war der Klassenbegriff bis einschließlich Version 6 ganz klar »COM-lastig«. Für eine Sprache, die speziell als Entwicklungswerkzeug für die OLE-Automatisierung auf der Basis des COM geschneidert wurde, ist das natürlich weder ein Wunder noch ein Makel.

Vererbungsarten prägen den Objektbegriff

Schnittstellenvererbung ist keine echte Vererbung im Sinne der traditionellen objektorientierten Programmierung, wo die Klasse Implementierung in Form von echtem Code erbt, auf die sie aufsetzen kann, sondern eher eine Art Pflichtenheft, wo das Erbe zur Erbringung von Implementierung verpflichtet und dabei die Form der Darbringung – eben die Schnittstelle – dieser Implementierung vorschreibt.

Die im Objektmodell verankerte (im Rahmen des CLS sogar sprachübergreifende) *Implementierungsvererbung*, bei der eine Klasse tatsächlich bestehenden und gegebenenfalls sogar von einer anderen .NET-Sprache stammenden Code wiederverwenden kann, war eine der großen Herausforderungen und übergeordnetes Ziel für die Konzeption von .NET.

Auch wenn der VB-Programmierer nun – gleich, ob er nach C# oder VB.NET umsteigt – nicht nur umdenken, sondern auch einiges hinzulernen muss: Der Gegenwert stimmt; es lohnt sich. Tatsächlich ist der Möglichkeitshorizont, der sich auftut, geradezu überwältigend.

8.1.2 Terminologie – Objekt, Objektvariable, Instanz

Wo in der imperativen Programmierung eine Variable eines bestimmten Datentyps vereinbart wird und einen Wert dieses Datentyps zugeordnet bekommt, so vereinbart man in der objektorientierten Programmierung eine *Objektvariable* für eine bestimmten Klasse und ordnet dieser eine *Instanz* bzw. ein *Objekt* dieser *Klasse* zu. Und wo die imperative Programmierung mit einer losen Sammlung von Funktionen und Prozeduren arbeitet, die nur implizit – nämlich über die Datentypen der Ein-/Ausgabeparameter und des Funktionswerts – auf einen oder mehrere Datentypen zugeschnitten sind, sorgt der Klassenbegriff bei der objektorientierten Programmierung für eine feste Zuordnung zwischen einer konkreten Datenstruktur (*Datenfelder* des Objekts) und den zugehörigen Operationen (*Methoden* des Objekts). Die Operation eines Objekts hat demnach einen Ich-Bezug; das hat eine gewöhnliche Funktion oder Prozedur nicht.

In vielen Büchern, die zum Thema objektorientierte Programmierung etwas beitragen, täuscht der etwas laxe Sprachgebrauch über die tatsächlichen Beziehungen zwischen den Begriffen Objektvariable, Objekt, Instanz und Klasse hinweg. Häufig werden erstere sogar als Synonym für letztere benutzt, was von der Sache her eine falsche Auffassung vermittelt und viel zur Mystifizierung der objektorientierten Programmierung beigetragen hat. Selten waren die Begriffe aber so klar auszumachen wie in C#.



Bei der objektorientierten Programmierung ist es wichtig, zwischen der Klasse als Datentypdefinition einerseits und den Werten als physischen Vertretern dieser Definition andererseits zu unterscheiden. Die Werte heißen Objekte oder Instanzen, wobei der Begriff »Instanz« noch etwas weiter reicht als der Begriff »Objekt« (hierzu gleich noch mehr) – ein Umstand, der in der einschlägigen Literatur selten zum Ausdruck gebracht wird. Anders gesagt, Objekte oder Instanzen sind konkrete Vertreter einer Klasse, die eine Identität und somit eine physische Existenz (auf dem Heap) haben. In der Tat hat auch die Klasse eine Instanz – und somit eine physische Existenz –, die den Datentyp und das Gemeinsame aller Objekte repräsentiert. Hierzu gleich noch mehr im Zusammenhang mit dem statischen Vereinbarungsteil.

Objektvariablen

Wie für Verweistypen üblich, werden Objekte an geeignete Variablen – *Objektvariablen* – gebunden. Eine Objektvariable speichert nichts weiter als

einen Verweis auf das Objekt, und es ist möglich, dass beliebig viele Objektvariablen auf ein und dasselbe Objekt referieren. Umgekehrt können Zustandsänderungen eines Objekts von mehreren Objektvariablen reflektiert werden:

```
class MyClass
{
  public int i = 1;
}

MyClass mf1 = new MyClass();
MyClass mf2 = mf1;
mf2.i = 2;
if (mf1.i == mf2.i)   // immer true!
...
```

8.2 Klasse

Das Klassenkonzept von C# gibt den unter .NET herrschenden Klassenbegriff Eins zu Eins wieder – und erinnert dabei mitunter stark an die ATL bzw. STL.

8.2.1 Statischer vs. dynamischer Bereich

Konzeptuell unterteilt sich eine Klasse in einen statischen Bereich und einen dynamischen Bereich. Der *statische Bereich* ist die Ausstattung der Laufzeitinstanz der Klasse als solche und bildet den so genannten *Klassenkontext*. Er lässt sich mit einer Bibliothek vergleichen. Diese Instanz, von der es je Klasse zur Laufzeit immer nur eine geben kann, wird in diesem Buch als *statische Instanz* bezeichnet. Sie kann (maximal) einen parameterlosen Konstruktor, sowie Konstanten, Datentypen, Datenfelder, Methoden und Eigenschaften umfassen, in deren Deklaration der Modifizierer `static` erscheint. Die statische Instanz verkörpert eine Umgebung, die bei dem ersten Zugriff auf die Klasse *einmalig* (durch den statischen Konstruktor) initialisiert wird und dann allen Objekten der Klasse (in ihrem öffentlichen Teil auch externen Objekten) zur gemeinsamen Nutzung zur Verfügung steht.

Im Gegensatz dazu bildet der *dynamische Bereich* die individuelle Ausstattung eines jeden einzelnen Objekts – den so genannten *Objekt-* oder *Instanzkontext*. An Elementen umfasst er: Konstruktoren (in beliebig vielen Überladungen), Datenfelder, Methoden und Eigenschaften. Dieser Kontext wird für jedes einzelne Objekt individuell eingerichtet, initialisiert und wieder abgebaut.

8.2.2 Syntax

Die Syntax für die Vereinbarung einer Klasse lautet:

```
 [Attribute]
[Modifizierer] class MeineKlasse [: {Basisklasse | Schnittstelle[, ...]}]
{
  //statische Elemente
  [Datentypen]
  [Konstanten]
  [statische Datenfelder]
  [statischer Konstruktor]
  [statische Methoden]
  [Operatoren]
  //dynamische Elemente
  [Instanzfelder]
  [Konstruktoren]
  [Instanzmethoden]
  [Destruktor]
}
```

Die in eckigen Klammern notierten Elemente können, aber müssen nicht vorhanden sein. Die einfachste Klassendefinition lautet somit

```
class MyClass { }
```

Vereinbarungen dieser Art werden oft als Vorausdeklarationen getroffen, um den Datentyp »schon einmal« formal benutzen und bestimmte Codeteile kompilieren zu können. Die Implementierung selbst kann dann zu einem späteren Zeitpunkt nachgetragen werden.

Attribute

Attribute werden durch eckige Klammern begrenzt, was die verwendete Notation hier nicht wiedergibt, und gehören zu den stark fortgeschrittenen Konzepten der Sprache. Sie stellen eine übergeordnete Deklarationsebene dar, die eine weitere Spezifikation des Codes – sowohl auf der Ebene einzelner Datentypen (`class`, `struct`, `enum` und `delegate`) als auch auf der Ebene der einzelnen Elemente von `class`- oder `struct`-Datentypen ermöglicht. Diese Metaebene ist selbst wieder klassenbasiert, wobei hier Codeeinheiten als Instanzen begriffen werden und somit Methoden, Eigenschaften usw. aufweisen. Das Typsystem für diese Metaebene ist gleichfalls in den .NET-Basisklassen vorbereitet und lässt sich in C# erweitern. Attribute sind somit in Form von Attributklassen freidefinierbar. (Gemeinsames Kennzeichen dieser Klassen ist, dass ihre Bezeichner auf »Attribute« enden.) In der .NET-Klassenhierarchie finden sich in verschiedenen Namensbereichen eine Fülle vordefinierter Attribute (gemeinsame Basisklasse ist `System.Attribute`), die den verschiedensten Zwecken dienen – angefangen von der einfachen Beschreibung des Codes und seines Ausführungsmodells bis hin zu seiner Einordnung im ausgefeilten Sicherheitsmodell.

Häufig verwendete Attribute sind `[Flags]` oder `[Description]`, die dieses Buch allerdings nicht einem größerem theoretischen Rahmen erläutert, sondern jeweils nur an Ort und Stelle und nach Bedarf.

Ein ebenso prominentes (wie in den meisten Fällen überflüssiges) Attribut ist der Zusatz `[STAThread]` vor der Methode `Main()`, der in jedem automatisch erzeugten Codegerüst zu finden ist:

```
[STAThread]
static void Main() { ... }
```

Dieses Attribut, das nur für die Methode `Main()` einen Sinn macht, kennzeichnet das Ausführungsmodell der gesamten Anwendung als *Singlethreaded Apartment* mit Blick auf das Zusammenspiel mit COM-Objekten.

Modifizierer

Modifizierer regeln die Sichtbarkeit und Vererbungseigenschaften der Klasse.

Zugriffsmodifizierer

Auf oberster Ebene, das heißt, auf Ebene des jeweiligen Namensbereichs, darf eine Klasse nur als `public` (maximaler Sichtbarkeitsbereich) oder `internal` (Sichtbarkeit auf Projektebene beschränkt) vereinbart werden. Fehlt der Zugriffsmodifizierer, vereinbart der Compiler implizit den Sichtbarkeitsbereich `internal`.

Die `private`- oder `protected`-Vereinbarung einer Klasse ist nur im Zusammenhang mit verschachtelten Datentypvereinbarungen (Vereinbarung einer Klasse *in* einer Klasse) möglich. In diesem Fall vereinbart der C#-Compiler standardmäßig den Sichtbarkeitsbereich `private`, wenn kein Modifizierer angegeben ist.

Vererbungsmodifizierer

Modifizierer für die Vererbungseigenschaften sind `sealed` und `abstract`, die sich gegenseitig ausschließen. `sealed` verbietet die weitere Ableitung der Klasse, `abstract` deklariert die Klasse als abstrakte Klasse, die als solche selbst nicht instanziiert werden kann.

Klassenelemente

Die elementaren Bestandteile einer Klasse, jenseits der *Konstanten-* und *Typvereinbarungen* werden auch *Mitglieder* genannt, wobei dieser Begriff eine nicht gerade gelungene Eindeutschung der amerikanischen Bezeichnung *Member* darstellt und zusammen mit grässlichen Wortschöpfungen wie »Mitgliedsfunktionen« für Methoden und »Mitgliedsvariablen« oder »Mitgliedsfelder« für Instanzfelder seinen Weg in die Dokumentation zur objektorientierten Programmierung gefunden hat.

Der Körper einer `class`-Vereinbarung besteht ausschließlich aus Elementvereinbarungen, die an sich in beliebiger Reihenfolge getroffen werden können. In der Praxis empfiehlt sich jedoch die Einhaltung einer Reihenfolge, bei der zumindest die Elementarten gruppiert sind. Besser noch ist es, der Lesegewohnheit für prozedurale Sprachen entgegenzukommen und in etwa folgende Abfolge anzustreben:

1. Konstantenvereinbarungen
2. Datenfeldvereinbarungen
3. Datentypvereinbarungen
4. Konstruktoren, Destruktor
5. Operatoren, Eigenschaften und sonstige Methoden

Die Sichtbarkeit der Klassenelemente wird gleichfalls durch Zugriffsmodifizierer geregelt, die vor die Klassenvereinbarung gesetzt werden. Ist kein solcher Modifizierer angegeben, vereinbart der C#-Compiler das Klassenelement standardmäßig als `private`.

Zugriffsmodifizierer sind nicht zulässig für statische Konstruktoren und für Schnittstellenmethoden, wenn der Bezeichner mit dem Schnittstellenbezeichner qualifiziert ist. Der Compiler vereinbart diese Elemente generell als `public`.

Die folgenden Abschnitte charakterisieren die Klassenelemente nur kurz. Im Anschluss werden Konstruktoren, Destruktoren, Operatoren und sonstige Methoden ausführlicher vorgestellt.

Konstanten

```
[Attribute]
[Modizifierer] const DatenTyp Bezeichner = KonstanterAusdruck;
```

Eine Klassendefinition kann beliebige `const`-Vereinbarungen zur Bereitstellung von Wertkonstanten enthalten. Da `const`-Konstanten mit einem konstanten Wert bzw. einem zur Übersetzungszeit auswertbaren Ausdruck initialisiert werden müssen, macht die Vereinbarung anderer Datentypen als der integrierten Datentypen wenig Sinn – hier würde als Initialisierungswert nur der generische Wert `null` in Frage kommen. Damit sind insbesondere auch Arraytypen nicht zulässig.

```
class MyClass
{
  public const int Fraction = 4;
  public const double MyConst = System.Math.PI / Fraction;
  ...
```

`const`-Konstanten gehören ausschließlich zur Ausstattung der statischen Instanz. Das Muster für den externen Zugriff auf eine Klassenkonstante sieht demnach so aus:

```
double sin45 = Math.Sin(MyClass.MyConst);
```

`readonly`-Konstanten zählen an sich zu den Datenfeldern. Sie dürfen jeden beliebigen Datentyp tragen und müssen im zugehörigen Konstruktor (je nach Vereinbarung im statischen oder Instanzkonstruktor) initialisiert werden.

Sinn der folgenden Klassenvereinbarung ist der Export von String-Konstanten. Die Klasse stellt alle Konstanten über die statische Instanz bereit:

```
class MyErrors
{
  public const string Underflow =
    "Operationen wurde wegen Unterlauf abgebrochen";
  public const string Overflow =
    "Operationen wurde wegen Überlauf abgebrochen";
  public const string Done =
    "Operation erfolgreich abgeschlossen";
  public const string InitFail =
    "Initialisierung für Operationen nicht möglich";
  public static readonly string CompName;
  MyErrors() {}       // privater Konstruktor!!!
  static MyErrors()   //
  {
    CompName = SystemInformation.ComputerName;
  }
}
```



Die Klasse ist weder instanziierbar noch als Basisklasse verwendbar, da der Konstruktor (implizit) als `private` *vereinbart ist. Wenn Vererbung erwünscht ist, muss man den Konstruktor als* `public` *vereinbaren; soll Vererbung erzwungen (und/oder eine direkte Instanziierung verhindert) werden, ist die Klasse als* `abstract` *zu deklarieren.*

`readonly`-Konstanten ohne `static`-Vereinbarung sind hingegen »personalisierte« Konstanten, die zur Ausstattung einzelner Objektinstanzen beitragen und im dynamischen Konstruktor initialisiert werden:

```
class MyErrors
{
...
  public readonly string CompName;
  public MyErrors()
  {
    CompName = SystemInformation.ComputerName;
  }
}
...
this.Text = (new MyErrors()).CompName;
```

Instanzfelder und statische Datenfelder

```
[static] Modifizierer Datentyp Bezeichner [= KonstanterAusdruck];
```

Die Vereinbarung eines Datenfelds ist im Grunde genommen nichts anderes als die Definition einer Variablen: Es wird ein Bezeichner mit einem bestimmten Datentyp vereinbart, wobei grundsätzlich jeder Datentyp zulässig ist (vgl. hierzu Kapitel 26, »Datentypen«).

Standardmäßig initialisiert der Compiler Datenfelder mit 0 bzw. `null` (bzw. `""` für den Typ `string`). Die Initialisierungssyntax für Datenfelder unterliegt der Einschränkung, dass nur konstante Ausdrücke (also zur Übersetzungszeit Berechenbares) als Initialisierungswerte zulässig sind. Die Initialisierung eines Datenfelds mit einem Wert, der erst zur Laufzeit bekannt ist, fällt in den Aufgabenbereich des Konstruktors. (Ausnahme: Die Initialisierungssyntax erlaubt es, Instanzfelder mit den Werten statischer Datenfelder zu initialisieren.)

Sofern ein Datenfeld nicht als `private` vereinbart ist, wird es an abgeleitete Klassen vererbt, kann dort aber durch ein gleichnamiges Datenfeld beliebigen Typs überschrieben (besser: verdeckt) werden. Zur Kennzeichnung der Überschreibung sollten Sie den `new`-Modifizierer verwenden, dessen Fehlen der Compiler ansonsten aber nur mit einer Warnung bedenkt. Der Zugriff von der abgeleiteten Klasse auf das vererbte Datenfeld ist aber nach wie vor möglich, wenn eine Qualifizierung des Bezeichners mit `base` erfolgt.

```
class MyBase
{
  protected int i = 20;
}
class MyDerived : MyBase
{
  public new static int i = 20;   // öffentliches Instanzfeld
                                  // verdeckt MyBase-Feld
  public MyDerived()
  {
    base.i = 30;       // Zugriff auf ererbtes Feld via base-Schlüsselwort
  }
}
```

Das optionale Schlüsselwort `static` entscheidet darüber, ob ein Datenfeld der statischen Instanz oder den dynamischen Instanzen (Objekten) der Klasse zugeordnet ist. Aus Sicht der Vereinbarung bestehen zwischen statischen und dynamischen Datenfeldern nur zwei Unterschiede:

1. Statische Datenfelder werden mit dem Zusatz `static` vereinbart. Alle anderen Felder sind Instanzfelder.
2. Für die Initialisierung statischer Datenfelder ist der statische Konstruktor zuständig.

Was den Gebrauch anbetrifft, so ergeben sich gleichfalls zwei wesentliche Unterschiede:

1. Für den qualifizierten Zugriff auf statische Datenfelder muss der Klassenname als Qualifizierer angegeben werden: `MyClass.myField`. Für den qualifizierten Zugriff auf Instanzfelder muss ein Objekt zur Qualifizierung angegeben werden – entweder eine (initialisierte) Objektvariable `myObject.myField` oder ein anonymes Objekt `new MyClass().myField`.

```
class MyClass
{
  static int StaticElem = 10;
  int InstanceElem = 10;
}
...
int staticElem = MyClass.StaticElem;
int instanceElem = mc.InstanceElem;
```

2. Statische Datenfelder existieren genau einmal. Sie werden zusammen mit der statischen Instanz der Klasse einmalig initialisiert und damit für alle Instanzen der Klasse sowohl sichtbar als auch modifizierbar. (Über die Sichtbarkeit »von außen« entscheidet der Zugriffsmodifizierer.) Demgegenüber verfügt jede Instanz über einen eigenen Satz an Instanzfeldern, die zusammen mit der Instanz angelegt, initialisiert und auch wieder vernichtet werden.



Eine Vereinbarung von Datenfeldern mit dem Zusatz `virtual` *oder* `abstract` *ist nicht erlaubt. Verwenden Sie statt dessen Eigenschaften.*

Für C/C++- und Delphi-Programmierer: Da C# keine statischen Variablen für Methoden kennt, muss die entsprechende Information »ersatzweise« in `private`*-Datenfeldern gespeichert werden – für statische Methoden in statischen Datenfeldern, für Instanzmethoden in Instanzfeldern. Daraus folgt, dass diese Variablen für alle Methoden der Klasse sichtbar sind.*

Für Java-Programmierer: In C# ist der Zugriff auf statische Datenfelder über Instanzvariablen nicht möglich.

Im Gegensatz zu den lokalen Variablen einer Methode werden Datenfelder als Teil der zur jeweiligen Instanz gehörigen Information auf dem Heap gespeichert und automatisch vom Compiler vorinitialisiert.

Datentypen

Eine Klasse kann selbst Datentypen definieren – für den privaten, aber auch den öffentlichen Gebrauch. Erlaubt sind beliebige `class`-, `struct`-, `delegate`- und `enum`-Definitionen, mit der Möglichkeit für die weitere Verschachtelung bei `class`- und `struct`-Definitionen.

Obwohl es vom Prinzip her möglich ist, eine innere Klasse von einer äußeren Klasse abzuleiten (der Compiler schreitet hier zumindest nicht ein), ist von einem solch rekursiven Design dringend abzuraten, da es unweigerlich zu Stack-Überläufen führt:

```
class Base
{
  protected Derived d;
  public Base()
  {
    d = new Derived();              // Stacküberlauf!
  }
  public class Derived : Base  {}  // möglich, aber problematisch
}
```

Statische Methoden, Operatoren, Konstruktoren, Instanzmethoden, Destruktor

Detaillierte Informationen über die prozeduralen Klassenelemente, ihre Vereinbarung und ihren Gebrauch finden Sie im Abschnitt »Methoden« ab Seite 257.

8.2.3 Instanziierung – new-Operator und Konstruktor

Wie gelangt man zu einem Wert eines als Klasse vereinbarten Datentyps? Dazu bedarf es der *Instanziierung* der Klasse unter Zuhilfenahme des `new`-Operators. Dieser für jede Klasse verfügbare generische Operator belegt den für das Objekt erforderlichen Speicherplatz auf dem Heap. Da ein Objekt aber seinerseits weitere Objekte umfassen und ansprechen kann, muss es zudem noch *konstruiert* werden und kann dabei gegebenenfalls auch gleich noch initialisiert werden. Diese Konstruktion ist Aufgabe des so genannten *Konstruktors*, einer Methode, deren Bezeichner grundsätzlich der Klassenname ist. (Detaillierte Informationen zu Konstruktoren finden Sie im Abschnitt »Konstruktoren« ab Seite 281.)

Syntaktisch sieht die Instanziierung also so aus:

```
new MyClass();
```

Im Verein liefern die beiden eine zunächst noch anonyme Referenz auf das neue Objekt, die im Allgemeinen sofort einer bereits bestehenden oder in derselben Anweisung neu vereinbarten Objektvariablen zugewiesen wird, damit das Objekt im weiteren Programmverlauf ansprechbar bleibt:

```
MyClass myObject;
myObject = new MyClass();
```

bzw.

```
MyClass myObject = new MyClass();
```

Beachten Sie aber, dass es durchaus Fälle gibt, in denen man bereits mit der anonymen Referenz auskommen kann, beispielsweise wenn die Referenz auf das neue Objekt als Parameter in einem Funktionsaufruf übergeben wird oder das Objekt so eigenständig ist, dass sein Besitzer nur einen einzigen Aufruf vornehmen muss.:

```
new MyForm().Show()     // bringt Formular auf den Bildschirm
```

Entsprechender Code findet sich übrigens auch in jedem per Assistent erzeugten Programmgerüst für eine Windows-Anwendung:

```
internal class MyForm : System.Windows.Forms.Form
{
 ...
  static void Main()
  {
    Application.Run(new MyForm());
  }
}
```

Standardkonstruktor

Der parameterlose Konstruktor wird auch als *Standardkonstruktor* bezeichnet. Eine generische Variante dieser Methode (genau genommen sind es zwei: eine statische und eine dynamische) zählt zur Grundausstattung einer jeden Klasse und wird ihr über den Mechanismus der Vererbung zuteil (hierzu gleich noch mehr).

Statischer vs. dynamischer Vereinbarungsteil

Die Definition einer Klasse unterteilt sich wie gesagt in einen statischen und einen dynamischen Teil. Der statische Teil definiert die Ausstattung der Klasse als solche (statische Instanz), vergleichbar mit einer Bibliothek. Mit anderen Worten: Um mit den statischen Bestandteilen einer Klasse zu arbeiten, braucht zuvor kein konkretes Objekt der Klasse erzeugt werden. Es ist klar, dass zu dieser Ausstattung alle formalen Vereinbarungen, also Konstanten, Aufzählungstypen und sonstige Typen gehören, sofern diese nicht einen Modifizierer tragen, der sie nach außen hin unsichtbar macht. Weiterhin gehören auch alle als `static` vereinbarten Elemente (Methoden und Datenfelder) hinzu. Die folgende einfache Klasse enthält zwei statische Elemente und eine Konstante:

```
class MyClass
{
  public const int DefaultValue = 1;
  public static int i = DefaultValue;
  public static bool HasDefaultValue()
  {
     return i == DefaultValue;
  }
}
```

Der Editor von VS.NET macht diese statischen Elemente sichtbar. Wie in Abbildung 8.1 zu sehen, erbt jede Klasse auch die statischen Elemente ihrer Basisklasse – im Falle von `object` als Basisklasse also die statischen Methoden `Equals` und `ReferenceEquals`.

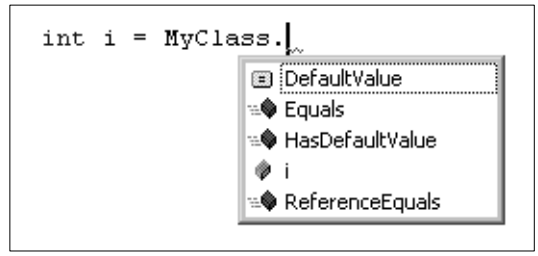


Abbildung 8.1: Der Editor von VS.NET zeigt die Elemente der statischen Instanz an.

Statische Elemente lassen sich außerhalb der Klasse nur über die statische Instanz ansprechen. In der Ausstattung von Objekten der Klasse sind sie nicht zu finden – womit nicht gesagt sein soll, dass Instanzmethoden der Klasse »von innen her« keinen Zugriff darauf hätten.



Initialisierung statischer Elemente

C# nimmt auch bei statischen Elementen standardmäßig eine Initialisierung mit dem Wert 0 bzw. `null` vor. Darüber hinaus muss es natürlich auch noch die Möglichkeit geben, statische Elemente gezielt mit eigenen Werten zu initialisieren. Da C# die Definition eines (parameterlosen) statischen Konstruktors erlaubt, verläuft die Initialisierung statischer Elemente in drei Stufen:

1. *Initialisierung durch generischen statischen Standardkonstruktor* – dieser Konstruktor initialisiert die statischen Elemente auf Ebene der integrierten Datentypen mit dem Wert 0 bzw. `null`.
2. *Initialisierungssyntax* – das statische Element erhält seinen Initialisierungswert optional durch eine Zuweisung bei Vereinbarung.
3. *Initialisierung im statischen Konstruktor* – das statische Element erhält seinen Wert während der Ausführung des für die Klasse definierten statischen Standardkonstruktors. Dieser Konstruktor kommt ein einziges Mal – entweder beim ersten (anonymen) Zugriff auf ein statisches Mitglied der Klasse oder unmittelbar vor Konstruktion des allerersten Objekts dieser Klasse, also noch vor Ausführung des jeweiligen dynamischen Konstruktors, zum Aufruf.

Das folgende Codefragment verdeutlicht, wie die Initialisierung der statischen Elemente abläuft:

```
class MyClass
{
  public static int i = i+1;  // i = 1; das geht, da gen. Initialisierung
  private static int j = 2;
  static MyClass()            // statischer Konstruktor
  {
    i += j;                   // i = 3; denn j ist bereits initialisiert
  }
  public MyClass()            // dynamischer Konstruktor
  {
    i *= 2;                   // i = 6; Aufruf erst nach stat. Konstruktor
  }
}
...
MyClass mc = new MyClass();   // erster Zugriff auf MyClass
int k = MyClass.i;    // k = 6;
```



Wenn Sie eine Klasse vereinbaren, die ausschließlich aus statischen Elementen besteht, erhalten Sie so etwas wie eine Bibliothek, wie Sie sie vielleicht von der prozeduralen Programmierung her kennen. Um zu verhindern, dass ein Client (völlig nutzlose) Objekte dieser Klasse anlegt – der parameterlose Konstruktor dafür ist ja generisch vorhanden –, gibt es zwei Möglichkeiten: Die bessere ist, Sie vereinbaren die Klasse als abstrakte Klasse. Sie ist dann nicht instanziierbar, kann aber jederzeit als Basisklasse für andere Klassen verwendet werden. Die schlechtere ist der traditionelle Weg, einen parameterlosen `private`*-Konstruktor zu vereinbarten, der den generischen überschreibt:*

```
class MyClass
{
  // TODO statische Vereinbarungen
  private MyClass() {}
}
```

Mit dieser Vereinbarung schreitet der Compiler gegen den Aufruf `new MyClass()` *ein. Die Klasse steht in diesem Fall auch nicht mehr als Basisklasse zur Verfügung.*

Codebeispiel – Instanzzähler

Das Beispielprojekt *Bouncing*, eine Windows-Anwendung, ist eine kleine Ballanimation, bei der der gesamte Code in einer einzigen Formularklasse namens `BouncingBall` zusammengefasst ist. Eine `Timer`-Komponente `timer1`, die alle 50 Millisekunden ein `Tick`-Ereignis generiert, haucht der Animation Leben ein. Die Behandlungsroutine `timer1_Tick()` aktualisiert die Positionsbeschreibung des Balls unter Berücksichtigung von Bandenreflexionen und veranlasst dann, dass der Bereich um den Ball neu gezeichnet wird.

Das Formular reagiert auf Mausklicks, indem es weitere Instanzen seiner selbst erzeugt und die dabei aufgehenden Formulare nebeneinander anordnet. Jede Instanz kann ein Nachfolgerformular generieren und hält einen

Verweis darauf in einem Instanzfeld. Die Logik ist so ausgelegt, dass jeweils zuerst alle untergeordneten Fenster geschlossen werden, wenn auf ein älteres Formular erneut geklickt wird.

Die Instanzzählung basiert auf einem statischen Datenfeld `InstanceCounter`, dessen Wert bei jeder Initialisierung eines neuen Formulars aktualisiert und über die `Text`-Eigenschaft in der Titelleiste des Fenster sichtbar gemacht wird. Um alle Formulare zu schließen, reicht es, das erste Formular zu schließen.

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Bouncing
{
  /// <summary>
  /// Bouncingball: einfache Ballanimation, bei der ein Ball
  /// an Banden reflektiert wird. Mausklick öffnet weitere Instanzen
  /// </summary>
  public class BouncingBall : System.Windows.Forms.Form  // Formularklasse
  {
    private static int InstanceCounter;
    private BouncingBall NextForm; // für untergeordnetes Formular
    private Rectangle Ball;
    private int BallSpeedX = 5;
    private int BallSpeedY = 5;
    private const int BallHeight = 30;
    private const int BallWidth = 30;

    private System.Windows.Forms.Timer timer1;
    private System.ComponentModel.IContainer components;

    /// <summary>
    /// Konstruktor
    /// </summary>
    public BouncingBall()
    {
      InitializeComponent();
      Ball = new Rectangle(0, 0, BallHeight, BallWidth);
    }

    #region Windows Form Designer generated code
    /// <summary>
    /// Erforderliche Methode für die Designerunterstützung.
    /// Der Inhalt der Methode darf nicht mit dem Code-Editor geändert
    /// werden.
    /// </summary>
    private void InitializeComponent()
    {
      this.components = new System.ComponentModel.Container();
      this.timer1 = new System.Windows.Forms.Timer(this.components);
      //
      // timer1
      //
      this.timer1.Tick += new System.EventHandler(this.timer1_Tick);
```

```
    //
    // BouncingBall
    //
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.ClientSize = new System.Drawing.Size(292, 272);
    this.Name = "BouncingBall";
    this.Text = "BouncingBall";
    this.Click += new System.EventHandler(this.BouncingBall_Click);
    this.Closing += new System.ComponentModel.CancelEventHandler(
                    this.BouncingBall_Closing);
    this.Load += new System.EventHandler(this.BounceForm_Load);
    this.Paint += new System.Windows.Forms.PaintEventHandler(
                  this.BouncingBall_Paint);
  }
  #endregion
  /// <summary>
  /// Der Haupteinstiegspunkt für die Anwendung.
  /// </summary>
  [STAThread]
  static void Main()
  {
    Application.Run(new BouncingBall());
  }
  private void BounceForm_Load(object sender, System.EventArgs e)
  {
    InstanceCounter+=1;
    Text= "BouncingBall " + InstanceCounter.ToString();
    timer1.Interval = 50;
    timer1.Enabled = true;
  }
  private void timer1_Tick(object sender, System.EventArgs e)
  {
    Invalidate(Ball);  // alte Position neu zeichnen
    Ball.X += BallSpeedX;
    Ball.Y += BallSpeedY;
    // Korrektur bei Bandenkollision
    if (Ball.X < 0) // links
    {
      BallSpeedX = -BallSpeedX;
      Ball.X = - Ball.X;
    }
    else if (Ball.X + Ball.Width > ClientRectangle.Right) // rechts
    {
      BallSpeedX = -BallSpeedX;
      Ball.X = 2 * (ClientRectangle.Right - Ball.Width) - Ball.X;
    }
    if (Ball.Y < 0)           // oben
    {
      BallSpeedY = -BallSpeedY;
      Ball.Y = - Ball.Y;
    }
```

```
      else if (Ball.Y + Ball.Height > ClientRectangle.Bottom) // unten
      {
        BallSpeedY = -BallSpeedY;
        Ball.Y = 2 * (ClientRectangle.Bottom - Ball.Height) - Ball.Y;
      }
      Invalidate(Ball);  // neue Position neu zeichnen
    }
      private void BouncingBall_Paint(object sender,
      System.Windows.Forms.PaintEventArgs e)
    {
      e.Graphics.FillEllipse(System.Drawing.Brushes.Black, Ball);
    }
    // Bei Mausklick ein weiteres Formular Bouncingball öffnen
    private void BouncingBall_Click(object sender, System.EventArgs e)
    {
      if (NextForm != null)          // untergeordnete F. ggf. schließen
        NextForm.Close();
      NextForm = new BouncingBall(); // Verweis auf untergeordnetes F.
      NextForm.Show();
      NextForm.Left = Left + Width;  // gleich daneben platzieren
      NextForm.Top = Top;
    }
    // untergeordnete Formulare schließen.
    private void BouncingBall_Closing(object sender,
      System.ComponentModel.CancelEventArgs e)
    {
      if (NextForm != null)
        NextForm.Close();
    }
  }
}
```

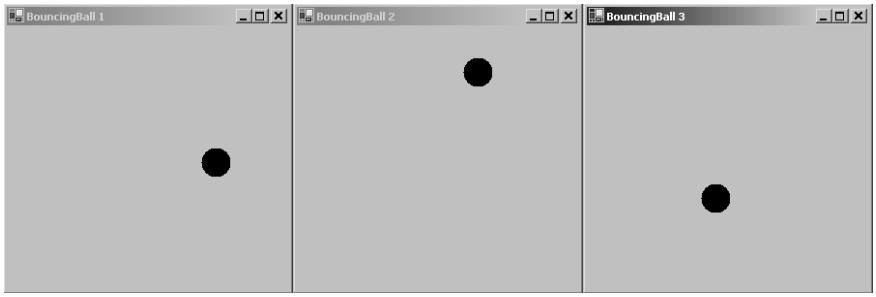


Abbildung 8.2: Drei Instanzen des BouncingBall-Formulars

8.3 Methoden

Als Effekt der konsequenten Ausrichtung auf die objektorientierte Programmierung steckt der gesamte prozedurale Code eines C#-Programms im Klassendesign. C# macht Schluss mit herrenlosem Code in Form reiner Funktionsbibliotheken und setzt auf das allgemeinere Konzept der Objektbibliotheken.

Anders als C/C++ und Delphi kennt C# also keine globalen Variablen und Funktionen mehr. In C# hat jede Funktion (und Variable) grundsätzlich einen Klassen- oder Objektbezug, wobei dieser Bezug nicht nur eine Schlüsselrolle für die Auflösung des Bezeichners (im Sinne einer Qualifizierung) spielt, er ist (generischer) Teil ihrer Spezifikation und liefert die Kontextinformation, die für Abarbeitung der Funktion erforderlich und charakteristisch ist. Man bezeichnet solche Funktionen daher – wie in der objektorientierten Programmierung üblich – als *Methoden*. Begrifflich etwas schärfer ist die Aufteilung in:

- *Statische Methoden*
- *Instanzmethoden (Objektmethoden)*
- *Konstruktoren*
- *Destruktoren*
- *Eigenschaften*
- *Operatoren*

Bei allen diesen Formen handelt es sich um spezielle Ausprägungen von Funktionen, die nicht nur von der Syntax her gewisse Unterschiede aufweisen, sondern in ihren Aufgabenbereichen auch semantisch streng voneinander abgegrenzt sind. Einzig Operatoren lassen sich als Sonderfall statischer Methoden mit eigener Schreibweise auffassen – und durch solche ersetzen.

Eines der wirklich großen Konzepte, die C++ nachhaltig zur objektorientierten Programmierung (aber nicht nur dazu) beigetragen hat, sind die Mechanismen des Überladens (*overloading*) und Überschreibens (*overriding*) von Methoden. Beide haben etwas mit der Mehrfachverwendung ein und desselben Methodenbezeichners zu tun und stellen damit nicht nur den Dreh- und Angelpunkt für die Organisation prozeduralen Programmierens dar, sondern verleihen der Sprache auch einen großen Teil ihrer Eleganz. C# übernimmt dieses Konzept ohne wesentliche Änderungen.

8.3.1 Allgemeines über Methoden

Rein formal gesehen sind Methodenvereinbarungen nichts weiter als gewöhnliche Funktionsvereinbarungen, wie sie in jeder ernst zu nehmenden Programmiersprache möglich sind – von einigen Feinheiten einmal abgesehen, die eine objektorientierte Sprache, und speziell C#, naturgemäß mitbringt. Was eine Methode von einer gewöhnlichen Funktion (die es in C# so nicht mehr gibt) unterscheidet, ist der Bezug, den sie zu ihrer Klasse bzw. zu den Objekten ihrer Klasse hat. Eine Methode ist grundsätzlich als Ausstattung einer Instanz zu sehen: Im Falle einer statischen Methode als Ausstattung der *statischen Instanz* der Klasse und ansonsten als Ausstattung eines

konkreten Objekts der Klasse. Umgekehrt stellt diese Instanz den Kontext für die Methode und liefert so etwas wie einen »Ich-Bezug«, der eine feste Perspektive auf die Daten der Klasse bzw. des jeweiligen Objekts mit sich bringt.

Syntax

Die allgemeine Syntax für Methoden lautet:

```
[Attribute]
[static] [Modifizierer] [Rückgabetyp] Bezeichner[([Parameterliste])]
{
  [Anweisungsfolge]
}
```

Dabei gilt:

```
Parameterliste = Parameter [[, Parameter] ...]
```

mit

```
Parameter = [{ref, out}] Datentyp Variable
```

Man beachte, dass die einzelnen hier optional angeführten Bestandteile je nach Art der Methode entweder wegfallen oder obligatorisch sein können.

Attribute spielen im Allgemeinen nur für die stark fortgeschrittene Programmierung ein Rolle und werden hier nicht abgehandelt (vgl. aber Seite 245).

Die *Modifizierer* regeln die Sichtbarkeit (Zugriffsmodifizierer) und die Art der Methode. Wichtig im Zusammenhang mit Methoden sind die Vererbungsmodifizierer `virtual`, `override` und `new`. Der Modifizierer `virtual` ermöglicht die Deklaration einer neuen, nicht bereits ererbten virtuellen Methode, `override` das Überschreiben einer ererbten virtuellen Methode und `new` das Überschreiben einer ererbten nicht-virtuellen Methode. (Die entsprechenden Konzepte werden im Verlauf der Diskussion noch eigens vorgestellt.)

Der *Rückgabetyp* spezifiziert den Datentyp des Funktionswerts, wobei hier jeder bekannte Datentyp in Frage kommt. Wird ein Typ ungleich `void` spezifiziert, erwartet der Compiler im Anweisungsblock der Methode eine `return`-Anweisung, die einen Wert dieses Typs an den Aufrufer zurückliefert: Im Falle eines Werttyps erhält der Aufrufer eine Kopie des zurückzugebenden Werts, im Falle eines Verweistyps schlicht eine Kopie der Referenz. Für Konstruktoren wird kein Rückgabetyp spezifiziert.

Die *Parameterliste* benennt die Aufrufparameter der Methode. Für sie gelten je nach Art der Methode verschiedene Einschränkungen:

- *Eigenschaften* werden grundsätzlich ohne Parameterliste vereinbart. (Der für den `set`-Accessor benötigte Parameter wird über das Schlüsselwort `value` zur Verfügung gestellt.)
- Bei *unären Operatoren* muss die Parameterliste exakt einen *Call-by-value*-Parameter (hierzu gleich noch mehr) spezifizieren, dessen Datentyp die eigene Klasse sein muss.
- Bei *binären Operatoren* muss die Parameterliste zwei *Call-by-value*-Parameter spezifizieren, wobei einer der Parameter die eigene Klasse als Datentyp tragen muss.
- *Destruktoren* werden mit leerer Parameterliste vereinbart.
- Bei sonstigen Methoden und Konstruktoren kann die Parameterliste beliebig viele Parameter benennen, aber auch leer bleiben.

Der *Bezeichner* bildet zusammen mit den in der Parameterliste genannten Datentypen die *Signatur* einer Methode. Diese spielt eine große Rolle für die Mechanismen des Überladens und des Überschreibens.

Das optionale Schlüsselwort `static` liefert die Unterscheidung zwischen statischen Methoden und Instanzmethoden. Nur letztere bereichern die Ausstattung des einzelnen Objekts, erstere gehören zur statischen Instanz der Klasse und unterliegen so gewissen Einschränkungen hinsichtlich ihres Aufrufs, ihres Kontexts und der Vererbung. Der folgende Codeauszug demonstriert den formalen Unterschied zwischen einer statischen Methode und einer Instanzmethode:

```
class MyClass
{
  public static void StatischeMethode()
  {
   ...
  }
  public int InstanzMethode(int Param)
  {
   ...
     return FunktionswertFürParameter;
  }
}
...
MyClass mc = new MyClass();     // neues Objekt instanziieren
int i = mc.InstanzMethode(28);  // Instanzmethode aufrufen
MyClass.StatischeMethode();     // anonymer Aufruf der statischen Methode
```

Parameterübergabe

Da die Parameterübergabe an Methoden zu den zentralen Aufgaben bei der Programmierung gehört und ihre Rolle oft genug – mit dem Preis eines schlechten Laufzeitverhaltens – unterschätzt wird, ist ein Verständnis dessen, was dabei passiert, auch für die C#-Programmierung besonders wich-

tig. Im Gegensatz zu C/C++ fordert C# vom Programmierer ein deutlicheres »Bekenntnis«, welche Aufgabe ein Parameter in der Parameterliste einer Methode hat. Der Ertrag kann sich aber sehen lassen: Gegenüber C/C++ schafft es C#, einen großen Teil potenzieller Laufzeitfehler, wie sie beispielsweise auf fehlende Initialisierung zurückzuführen sind, noch zur Übersetzungszeit zu erkennen und auszuschalten. Des weiteren ist die vom C#-Compiler durchgesetzte Typsicherheit eine gute Versicherung für das reibungslose Zusammenspiel zwischen dem erwarteten Parameter und dem konkret dargebotenen Wert. Das muntere »Zurechtbiegen« von `void`-Zeigern und expliziten Typumwandlungen in Struktur- oder Objektzeiger, wie sie bei der API-Programmierung mit C/C++ oder Delphi zur Tagesordnung gehören, haben endgültig ausgedient: Bei C# sind nur Typumwandlung zulässig, für die ein expliziter oder impliziter Typumwandlungsoperator definiert wurde. Das verbessert natürlich die Robustheit des Codes zur Laufzeit enorm. Womit aber nicht gesagt sein soll, dass man in C# mit Typumwandlungen keinen Unsinn anstellen könnte. Laufzeitfehler durch missglückte explizite Typumwandlungen gehören zu den schwer behandelbaren Ausnahmen, da Reparaturen bei Vorliegen eines falschen Datentyps kaum möglich sind.

Aufrufmechanismen – call-by-value vs. call-by-reference

An konkreten Aufrufmechanismen unterscheidet C# – wie alle anderen Programmiersprachen auch – zwischen *Call-by-value*- und *Call-by-reference*-Aufrufen. Bekanntlich erhält eine Methode im ersten Fall eine Kopie des Wertes und im zweiten Fall einen Verweis auf den übergebenen Wert.

Die Schlüsselworte ref und out

Da in C# Parameter standardmäßig als *Call-by-value*-Parameter übergeben werden, muss ein *Call-by-reference*-Parameter immer als solcher gekennzeichnet sein. Dazu ist sowohl bei der Parameterdeklaration als auch beim Funktionsaufruf eines der Schlüsselworte `ref` oder `out` voranzustellen – je nachdem, ob der Parameter als zu initialisierender Ein-/Ausgangsparameter (`ref`) oder als (nicht zu initialisierender) reiner Ausgangsparameter (`out`) fungiert.

Wer von Visual Basic her kommt und an die Schlüsselwörter *ByVal* und *ByRef* gewöhnt ist, dürfte hier zwar keine Verständnisschwierigkeiten haben, er muss sich jedoch damit abfinden, dass das Standardverhalten in C# (und übrigens nun auch in VB.NET) genau anders herum spezifiziert ist: VB bis zur Version 6 vereinbart einen Parameter automatisch als *Call-by-reference*-Wert, C# hingegen als *Call-by-value*-Wert.

Vom Effekt her sind `ref`- und `out`-Parameter nichts weiter als Umbenennungen der vom Aufrufer übergebenen Variablen. Ausdrücke, Konstanten oder Literale dürfen hier nicht angegeben werden, erwartet wird ein so genannter

Linkswert, also eine Größe, die bei einer Zuweisung links stehen kann. Zudem prüft der C#-Compiler `ref`-Parameter auf vorherige Initialisierung und stellt sicher, dass einem `out`-Parameter in der Methode ein Wert zugewiesen wird.

Neu bei C# ist, dass die Schlüsselworte `ref` und `out` auch beim Aufruf der Methode anzugeben sind. Das verbessert die Lesbarkeit des Quellcodes enorm und ruft die Rollen der Parameter ins Gedächtnis zurück.

Ein Beispiel:

```
class MyClass
{
  ...
  public void MyMethod(int i1, ref int i2, out int i3)
  {
    i1 *= 2;
    i2 *= 2;
    i3 = i1 * i2;
  }
}
...
int InParam = 10;
int InOutParam = 20;
int OutParam;
MyClass mc = new MyClass();
mc.MyMethod(InParam, ref InOutParam, out OutParam);
Console.WriteLine("InParam ={0} InOutParam = {1} OutParam = {2}",
     InParam, InOutParam, OutParam);
```

Die Konsolenausgabe für diesen Code belegt das geschilderte Verhalten (vgl. Abbildung 8.3). Änderungen des ersten Parameters schlagen nicht bis zum Aufrufer durch, Änderungen des zweiten und dritten schon.



Abbildung 8.3: Konsolenausgabe für Testcode

Das bisher Diskutierte dürfte für Werttypen (gleich, ob einfach oder komplex) keinerlei Verständnisprobleme aufwerfen. Schwieriger wird es, die gleiche Sicht für Verweistypen aufrecht zu erhalten, zumal Verweistypen als eigentlichen Wert bekanntlich ja einen Verweis repräsentieren.

Call-by-value-Übergabe und Verweistypen

Die Preisfrage lautet nun: Wie sieht eine *Call-by-value*-Übergabe für einen Verweistyp aus? Oder um es als Gretchenfrage zu formulieren: Wie verhält es sich mit dem Wert – wird der nun kopiert oder nicht? Wer nun danach geht, dass `string` ja ein Verweistyp ist und die Geschichte schlicht ausprobiert, dürfte zu falschen Schlüssen kommen, denn `string` ist kein typischer Verweistyp.



Obwohl `string`-Werte an sich als Wertkopie übergeben werden, muss der Stringwert selbst dabei im Allgemeinen nicht kopiert werden (vgl. Seite 363ff). Mit anderen Worten: Call-by-value-Übergaben von Strings sind vom Prinzip ebenso effizient wie Call-by-ref-Übergaben. Der folgende Codeauszug zeigt dies:

```
string s = new String(' ', 200000);
Console.WriteLine(DateTime.Now);
for (int i = 0; i < 100000000; i++)
  sd(s);
Console.WriteLine(DateTime.Now);
for (int i = 0; i < 100000000; i++)
  sd1(10);
Console.WriteLine(DateTime.Now);
...
static void sd(string s)
{
  string w = s;
}
  static void sd1(int s)
{
  int w = s;
}
```

Die Ausgabe lautet:

```
10.07.2002 14:26:16
10.07.2002 14:26:18
10.07.2002 14:26:20
```

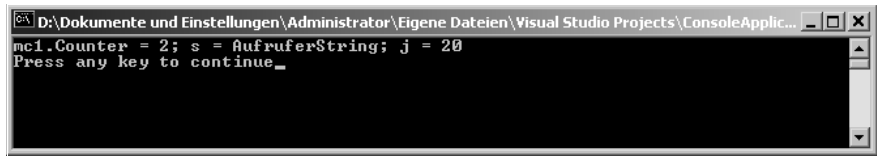
Was bei einer *Call-by-value*-Übergabe tatsächlich passiert, ist eine Zuweisungsoperation. Ist diese für den jeweiligen Datentyp als Kopieroperation implementiert (wie bei den Werttypen, die von `System.ValueType` abstammen, oder eben auch bei `string`), erhält die Methode eine neue Instanz des Parameters – und andernfalls nur eine neue Objektvariable, die auf das zu übergebende Objekt verweist. Damit hat ein *Call-by-value*-Parameter, der einen Verweistyp trägt, exakt dieselbe Wirkung wie ein `ref`-Parameter, der einen Werttyp trägt:

```
class MyClass
{
  public int counter = 1;
  public void myMethod(MyClass mc, string testString, ref int i)
  {
    mc.counter ++;
    testString = "In der Methode geändert";
    i = 2 * i;
  }
}
...
MyClass mc1 = new MyClass();    // mc1.counter wird hier auf 1 gesetzt
int j = 10;
string s = "AufruferString";

mc1.MyMethod(mc1, s, ref j);

Console.WriteLine("mc1.counter = {0}; s = {1}; j = {2}",
      mc1.counter, s, j);
```

Abbildung 8.4: Ergebnis des Beispielcodes



Besonders effektiv ist dieser Code nicht. Er zeigt aber zwei Dinge recht schön: Erstens, dass ein Objekt problemlos Referenzen auf sich selbst verarbeiten kann (unbemerkt) und zweitens eben das, was es zu zeigen gab: Das Objekt und auch der `int`-Wert wurden von der Methode manipuliert.

Beachten Sie, dass der Wert der als Eingangsparameter verwendeten Objektvariable beim Aufruf tatsächlich kopiert wird. Die Kopie ist allerdings nichts anderes als eine weitere Referenz auf dasselbe Objekt. Somit kann die Methode das Objekt jederzeit – sichtbar für den Aufrufer – manipulieren. Dass sich das Objekt im diskutierten Beispiel selbst manipuliert, bleibt unbemerkt bzw. spielt keine besondere Rolle – man könnte hier auch ein beliebiges anderes Objekt der Klasse `MyClass` übergeben. Von dem String hingegen erhält die Methode eine echte Kopie, sodass Veränderungen, die sie daran vornimmt, vom Aufrufer unbemerkt bleiben. Abbildung 8.4 zeigt die Auswirkung des Aufrufs aus Sicht des Aufrufers.

Call-by-reference-Übergabe und Verweistypen

Soweit dürfte der Fall also klar sein. Wie aber sieht die *Call-by-reference*-Übergabe für Parameter aus, die einen Verweistyp tragen? Wie zu erwarten, bildet `string` auch hier wieder die Ausnahme, sodass die Frage besser so gestellt werden sollte: Was passiert, wenn eine gewöhnliche Objektvariable als `ref`- oder `out`-Parameter übergeben wird? Ausgehend davon, dass der Compiler in diesem Fall schlicht eine Umbenennung der Variable vornimmt, ist die Antwort naheliegend: Die Methode kann den Wert der Objektvari-

able an sich ändern, ihr also – für den Aufrufer sichtbar – ein (anderes) Objekt zuweisen. Zudem erlaubt es die Objektvariable selbstverständlich auch, auf das jeweilige Objekt zuzugreifen, sodass ein *Call-by-reference*-Parameter auch die Möglichkeiten des *Call-by-value*-Parameters abdeckt – zumindest, was `ref`-Parameter angeht, `out`-Parameter erfordern ja zunächst einmal eine Wertzuweisung innerhalb der Methode.



Eine Call-by-reference-Übergabe für Verweistypen ist nur dann sinnvoll, wenn es die Spezifikation der jeweiligen Methode erfordert, einer als Parameter übergebenen Objektvariable ein anderes Objekt zuzuweisen. Ansonsten ist sie weder erforderlich noch bietet sie irgendwelche Laufzeitvorteile.

Bei umfangreicheren Werttypen (im Allgemeinen jedoch nicht bei `string`*-Werten) lassen sich durch Verwendung der Call-by-reference-Übergabe hingegen Laufzeitverbesserungen erzielen.*

Codebeispiel – Compare-Methode mit ref-Parameter

Das folgende kleine Konsolenprogramm verwendet eine etwas ungewöhnliche – weil objektgebundene – Implementierung der Methode `Compare`. Die Methode erwartet eine Objektvariable als `ref`-Parameter und weist dieser das »größere« Objekt zu. Man beachte, dass hierfür die `this`-Referenz verwendet wird. Zudem gibt es einen `out`-Parameter des Typs `string`, der den (in der Klasse als `protected` vereinbarten) Vergleichsstring preisgibt.

```
using System;
namespace ConsoleApplication1
{
  class MyClass
  {
    public string Name;
    protected string Text;
    public MyClass() {}
    public MyClass(string NewName, string NewText)
    {
      Name = NewName;
      Text = NewText;
    }
    // Vergleichsoperation, liefert "größeres" Objekt in ref-param
    // und größeren String in out-param. Ergebnis ist false,
    // wenn Text-Element beider Objekte nicht initialisiert wurde
    public bool Compare(ref MyClass mc, out string LongerText)
    {
      LongerText = mc.Text;              // geht, weil Text protected
      if (Text == null && mc.Text == null) return false;
      if (String.Compare(Text, mc.Text) > 0)
      {
        LongerText = Text;
        mc = this;
      }
      return true;
    }
```

```
    public static bool operator < (MyClass mc1, MyClass mc2)
    {
      if (String.Compare(mc1.Text,mc2.Text) < 0)
        return true;
      else
        return false;
    }

    public static bool operator > (MyClass mc1, MyClass mc2)
    {
      return !(mc1 < mc2);
    }
  }

  class KlassenVergleich
  {
    static void Main(string[] args)
    {
      MyClass mc1 = new MyClass("mc1", "Text erstes Objekt");
      MyClass mc2 = new MyClass("mc2", "Text zweites Objekt");
      string result;
      if ( mc1.Compare(ref mc2, out result))
        Console.WriteLine("Groesser ist " + mc2.Name + " mit Wert: " +
                          result);
      Console.WriteLine((mc1 < mc2).ToString());
    }
  }
}
```

Methoden mit variabler Parameteranzahl – params

Nicht immer steht von vornherein fest, mit wie vielen Parametern eine Methode aufgerufen wird. Ein gutes Beispiel dafür ist die Methode `Console.WriteLine()`. Sie existiert nicht nur in zahlreichen Überladungen, sondern kann speziell in der Variante, bei der der erste Parameter ein Formatstring mit Argumentplatzhaltern ist, faktisch mit beliebig vielen Parametern aufgerufen werden. So umfasst der Aufruf

```
Console.WriteLine("{14}{13}{12}{11}{10}{9}{8}{7}{6}{5}{4}{3}{2}{1}{0}",
                  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14);
```

beispielsweise 16 Argumente, welche die Methode klaglos entgegennimmt und in die folgende Ausgabe umsetzt:

```
14131211109876543210
```

Um nun nicht für jede Parameteranzahl eine Überladung bereitstellen zu müssen, unterstützt C# eine spezielle Syntax, die es einer Methode ermöglicht, ihre Parameterliste als Array entgegenzunehmen, obwohl aufrufseitig einzelne Werte notiert sind.

Syntax

Formal sieht die Syntax dafür so aus:

```
[Modifizierer] RückgabeTyp Bezeichner([Paramliste,] params Typ[] ArgList)
```

Wie üblich wird die Parameterliste in Klammern notiert. Um ab einem bestimmten Parameter eine variable Anzahl weiterer Parameter zuzulassen, muss der letzte Parameter als Array eines bestimmten Typs (notfalls `object`) vereinbart und mit dem Schlüsselwort `params` zur Parameterliste erklärt werden. Die Methode kann das Array dann elementweise auswerten. Die Array-Variable hat den Wert `null`, wenn die Parameterliste leer ist.

Codebeispiel

Abbildung 8.5 zeigt den Code und die Ausgabe eines kleinen Konsolenprogramms, das eine Methode mit einer variablen Anzahl von `int`-Parametern definiert und aufruft.

```
class Class1
{
  static void VariableParameters(params int[] IntArray)
  {
    if (IntArray != null)     // parameterloser Aufruf
      foreach (int i in IntArray)
        Console.Write(" {0}", i);
    Console.WriteLine();
  }
  [STAThread]
  static void Main(string[] args)
  {
    Console.WriteLine("Parameterloser Aufruf");
    VariableParameters();
    Console.WriteLine("Aufruf mit 2 Parametern");
    VariableParameters(1, 2);
    Console.WriteLine("Aufruf mit 7 Parametern");
    VariableParameters(1, 2, 3, 4, 5, 6, 7);
  }
}
```

```
D:\Dokumente und Einstellungen\Administrator\...
Parameterloser Aufruf

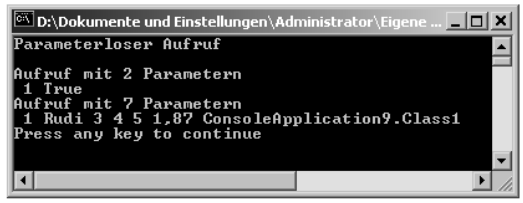
Aufruf mit 2 Parametern
 1 2
Aufruf mit 7 Parametern
 1 2 3 4 5 6 7
Press any key to continue
```

Abbildung 8.5: Methode, die eine variable Anzahl von int-Parametern verarbeitet

Nicht immer jedoch kann davon ausgegangen werden, dass alle Parameter einer variablen Parameterliste den gleichen Datentyp tragen. Um auch unterschiedliche Datentypen zuzulassen, benötigt das `params`-Array einen Elementtyp, der als Basisklasse für alle zu erwartenden Datentypen fungieren kann. Als Standardwahl bietet sich hier `object` an, obwohl natürlich auch andere Basisklassen, beispielsweise abstrakte Klassen wie `ValueType` oder auch Schnittstellen in Frage kommen. Abbildung 8.6 zeigt eine modifizierte Variante des Beispielprogramms, die mit Parametern beliebigen Typs zurecht kommt.

Abbildung 8.6: Methode, die eine variable Anzahl von Parametern beliebigen Typs verarbeitet

```
class Class1
{
  static void VariableParameters(params object[] IntArray)
  {
    if (IntArray != null)     // parameterloser Aufruf
      foreach (object i in IntArray)
        Console.Write(" {0}", i);
      Console.WriteLine();
  }
  [STAThread]
  static void Main(string[] args)
  {
    Console.WriteLine("Parameterloser Aufruf");
    VariableParameters();
    Console.WriteLine("Aufruf mit 2 Parametern");
    VariableParameters(1, true);
    Console.WriteLine("Aufruf mit 7 Parametern");
    VariableParameters(1, "Rudi", 3, 4, 5, 1.87, new Class1());
  }
}
```



8.3.2 Überladen von Methoden

Wer das Konzept des Überladens nicht kennt, weil er etwa von Visual Basic oder von einer stark typisierenden Programmiersprache wie C her kommt, den dürfte es überraschen, dass beispielsweise die Methode `Console.WriteLine()` einmal nur einen Parameter und ein andermal zwei, drei oder noch mehr Parameter entgegennimmt. Darüber hinaus schluckt sie auch so ziemlich »alles«, was man ihr vorsetzt, sei es ein String, eine Ganzzahl oder eine Gleitkommazahl. Ein Beispiel:

```
int i = 4;
decimal dec = 15.3;
double dbl = 3.1415;
Console.Write("Der Wert von i ist ");  // string-Parameter
Console.WriteLine(i);                 // int-Parameter
Console.WriteLine("Der Wert von dec ist {0} und der von dbl ist {1}",
  dec, dbl);                          // drei Parameter
```

Vom Prinzip her müsste hier eine Verletzung des Typsystems oder der Syntax vorliegen, wenn nicht irgendwelche geheimnisvollen Mechanismen am Werk sind, die eine plausible Erklärung liefern. Natürlich nutzt `Console.WriteLine()` intern die für alle Objekte verfügbare Methode `ToString()`, aber entgegennehmen tut sie echte Werte der Typen `int`, `long`, `bool`, `string` usw. Wie geht das?

Hierbei gilt es erst einmal, zwei Dinge auseinander zu halten: Dass eine Methode wie `Console.WriteLine()` eine beliebige Anzahl von Parametern ent-

gegennehmen kann, ist eine Geschichte, dass sie generell mit verschiedenen Parametertypen und mit verschiedenen Parameteranzahlen zurechtkommt, eine andere. Die erste Geschichte hängt mit dem Schlüsselwort `params` zusammen und wurde bereits im Zusammenhang mit der Parameterübergabe im vorangehenden Abschnitt geklärt: Variable Parameterzahlen sind ein vergleichsweise einfaches syntaktisches Konzept, das auf eine (implizite) Array-Initialisierung durch den Compiler hinausläuft und eben speziell darauf ausgelegt ist, Methoden mit einer beliebigen Parameteranzahl zu gestatten. Die zweite Geschichte ist das Ergebnis des Überlademechanismus, einem wirklich mächtigen Konzept, das die Namensauflösung verfeinert und die Benennungsproblematik gerade für generische Implementierungen, wie sie eine Methode wie `WriteLine()` verkörpert, entschärft.

Überlademechanismus

Der Überlademechanismus ermöglicht es, dass unterschiedliche Varianten einer Methode unter demselben Bezeichner *nebeneinander* existieren können.

```
private int myMethod () {...}                                  // Variante 1
protected void myMethod (int i) {...}                          // Variante 2
public long myMethod (long l) {...}                            // Variante 3
public virtual string myMethod (string s, long i) {...}   // Variante 4
```

Die Unterscheidung von Methoden und ihrer Varianten geschieht also nicht auf Ebene der Bezeichner, sondern auf Basis der *Signaturen*. Unter der Signatur einer Methode versteht man die Kombination aus dem Methodenbezeichner und den Parametertypen. Für die Variante 4 der eben vorgestellten Methode `myMethod()` ergibt sich die Signatur somit aus der Information »myMethod«, »string« und »int«. Nach diesem Prinzip können in der Tat beliebig viele Varianten einer Methode definiert werden. Jede einzelne Variante verkörpert letztlich eine eigenständige Methode – unabhängig von den anderen. Der Rückgabetyp (oder gar der Modifizierer) spielt hingegen keine Rolle, obwohl er, wie der Codeauszug zeigt, durchaus unterschiedlich sein darf – aber nicht muss.

Aufruf überladener Methoden

Zur *Namensauflösung* bei überladenen Methoden ermittelt der Compiler anhand der für den einzelnen Aufruf vorgefundenen Typkonstellation, welche der für den jeweiligen Methodenbezeichner im Angebot stehenden Signaturen am besten passt. Das Rennen macht dabei jeweils die Methode, deren Signatur entweder vollkommen übereinstimmt oder deren Signatur mit geringstem Aufwand adaptierbar ist. Mit anderen Worten: Falls die benötigte Signatur nicht verfügbar ist, probiert der Compiler, ob durch eine implizite Typumwandlung der Parameter, beispielsweise von `int` zu `long`, etwas zu holen ist.

```
myMethod("Das ist Text", 10);  // Variante 4 (zweiter Parameter wird
                               // implizit nach long umgewandelt)
myMethod(10);                  // Variante 2
myMethod(10L);                 // Variante 3
long l = myMethod(10)          // Fehler! keine impl. Umwandlung für V 3
long l = myMethod(10L)         // Variante 3
```

Falls der Compiler mehrere gleich gute Möglichkeiten für die Adaption der Signatur findet – oder überhaupt keine –, meldet er einen Fehler.



Versehentliches Überschreiben von Basisklassenmethoden

Ohne an dieser Stelle bereits genauer auf den Vererbungsmechanismus eingehen zu wollen, sei angemerkt, dass eine Klasse jederzeit auch Überladungen für Methoden bereitstellen kann, die sie von einer Basisklasse ererbt hat. Wichtig ist dabei nur, dass die Signaturen der überladenen Methoden auch mit Blick auf die Basisklasse eindeutig sind. Während es bei C++ so ist, dass die Methoden der abgeleiteten Klasse bei übereinstimmenden Signaturen die jeweiligen Basisklassenmethoden schlicht »verdecken" bzw. überschreiben, muss dies in C# zumindest bei nicht-virtuellen Methoden eigens durch den Modifizierer `new` *angezeigt werden. Dies sorgt für eine bessere Trennung zwischen Überladung und Überschreibung:*

```
class MyBase
{
  private int myMethod () {...}                              // Variante 1
  protected int myMethod (int i) {...}                       // Variante 2
}
class MyDerived: MyBase
{
  private new int myMethod () {...}           // Variante 1, überschrieben
  public void myMethod (long l) {...}                        // Variante 3
  public virtual string myMethod (string s, int i) {...}  // Variante 4
}
```

Wann ist Überladung sinnvoll?

Wie bereits ausgeführt, haftet dem Überlademechanismus grundsätzlich nichts Magisches an. Er implementiert ein Konzept, das kontextuelle Information für die Namensauflösung bei Methodenaufrufen mit einbezieht, nicht mehr und nicht weniger. Anstatt zwei Methoden zu überladen, kann man sie jederzeit auch schlicht unterschiedlich benennen. Andersherum geht es jedoch nur, wenn die Signatur bei gleicher Benennung unterschiedlich bleibt.

So viel zur formalen Seite. Die praktische Dimension der Überladung sieht natürlich anders aus. Die Regel lautet: Zwei Methoden sollten nur dann gleich benannt – also überladen – werden, wenn sie konzeptuell hinreichend äquivalent sind. »Konzeptuell hinreichend äquivalent« heißt in diesem Zusammenhang, dass sie von der Idee her die gleiche Operation implemen-

tieren, nur eben für verschiedene Datentypen oder eine verschiedene Anzahl von Datentypen.

Damit sind die beiden Hauptanwendungsbereiche für die Überladung klar erkennbar:

- generische Implementierung von Funktionalitäten
- optionale Parameter

Der erste Anwendungsbereich dürfte nach Betrachtung der Methode `Console.WriteLine()` im vorigen Abschnitt bereits deutlich abgesteckt sein: Es geht um die datentypübergreifende Implementierung.

Der zweite Anwendungsbereich weist in eine andere Richtung. Er bietet beispielsweise einen brauchbaren Ausweg für ein konzeptuelles Manko, das C# gegenüber C++, VB6 und Delphi hat: C# kennt keine optionalen Parameter. Um optionale Parameter mit C# zu verwirklichen, stellt man verschiedene Hüllen für ein und dieselbe Methode als Überladungen bereit. Der folgende Codeauszug zeigt das Prinzip:

```
private const int ParamDefault = 10;
public int myMethodWithOptParm (int dimension, int Opt)
{
  ...
}
public int myMethodWithOptParm (int dimension)
{
   return myMethodWithOptParm (int dimension, ParamDefault);
}
```

Mit ihren 19 Varianten ist die Methode `Console.WriteLine()` ein recht eindrucksvolles Beispiel für beide Anwendungsbereiche der Überladung und es lohnt sich auf jeden Fall, mal einen genaueren Blick darauf zu werfen.



Achten Sie bei der Benennung und Anordnung der Parameter auf ein leicht zu durchschauendes System und halten Sie dieses System für alle Überladungen ein. Die gleiche Anordnung ist Voraussetzung für die generische Anwendung, die einheitliche Benennung erleichtert das intuitive Rollenverständnis gerade im Zusammenhang mit der Implementierung optionaler Parameter. Obwohl überladene Methoden unterschiedliche Rückgabetypen aufweisen dürfen, was gerade bei generischer Zielsetzung hilfreich sein kann, sollten sie der Einheitlichkeit halber bevorzugt auf denselben Rückgabetyp getrimmt werden.

8.3.3 Überschreiben von Methoden

Obwohl das *Überschreiben* – zuweilen wörtlich auch treffender als »Überreiten« übersetzt – von Methoden an sich ein völlig anderer Mechanismus als das Überladen ist, werden diese beiden Konzepte häufig genug durcheinander geworfen oder gar verwechselt. Der Grund dafür ist wohl die etwas unglückliche Wortschöpfung für die beiden Begriffe – sowohl im Englischen (*override* und *overload*) als auch im Deutschen.

Natürlich haben beide Begriffe, wie eingangs schon festgestellt, etwas mit der Mehrfachverwendung von Methodenbezeichnern zu tun. Der Unterschied besteht aber darin, dass

- das *Überladen* (*overload*) ein Nebeneinander mehrerer Methoden mit gleichem Bezeichner ermöglicht (auch über die Grenzen der Vererbung hinweg), wobei C# (im Gegensatz zu Delphi) keine syntaktische Kennzeichnung durch ein eigenes Schlüsselwort fordert.
- das *Überschreiben* (*override*) etwas mit dem Verdecken von Methoden mit gleichem Bezeichner ausschließlich in Verbindung mit der Vererbung zu tun hat und C# (gerade deshalb) eine syntaktische Markierung durch ein Schlüsselwort fordert.

Verstehen Sie das also nicht falsch: Überladung und Vererbung sind voll verträglich, es besteht aber kein Voraussetzungszusammenhang. Überschreiben ist hingegen nur möglich (und nötig), weil Vererbung im Spiel ist.

Codebeispiel – Überschreiben von Methoden

Am besten lässt sich das Überschreiben anhand eines Codebeispiels demonstrieren:

```
class MyBaseClass
{
  public string NonVirtualMethod()
  {
    return "NonVirtualMethod der Basisklasse";
  }
  public virtual string VirtualMethod()
  {
    return "VirtualMethod der Basisklasse";
  }
}

class MyClass : MyBaseClass
{
  public new string NonVirtualMethod()
  {
    return "NonVirtualMethod von MyClass";
  }
```

```
  public override string VirtualMethod()
  {
    return "VirtualMethod von MyClass";
  }
}

class Class1
{
  static void Main(string[] args)
  {
    MyClass mc = new MyClass();
    MyBaseClass mbc1 = new MyBaseClass();
    MyBaseClass mbc2 = mc;

    Console.WriteLine(mc.NonVirtualMethod());
    Console.WriteLine(mbc1.NonVirtualMethod());
    Console.WriteLine(mbc2.NonVirtualMethod());
    Console.WriteLine(mc.VirtualMethod());
    Console.WriteLine(mbc1.VirtualMethod());
    Console.WriteLine(mbc2.VirtualMethod());
  }
}
```

Das in dem Code vorgestellte Szenario ist denkbar einfach und prototypisch. Neben der Klasse `Class1`, die die Methode `Main()` bereitstellt und für die Bildschirmausgabe verantwortlich ist, gibt es zwei Klassen: die abgeleitete Klasse `MyClass` und die Basisklasse `MyBaseClass`. Beide Klassen definieren zwei `public`-Instanzmethoden, jeweils eine virtuelle und eine nicht-virtuelle, mit demselben Bezeichner. `MyClass` erbt zwar die Methoden von `MyBaseClass`, überschreibt diese aber durch eigene Varianten. Man beachte, dass die syntaktischen Erfordernisse für die beiden Methodenarten unterschiedlich sind.

Zum Überschreiben gewöhnlicher Methoden ist in C# (im Gegensatz zu C++) das Schlüsselwort `new` erforderlich und zum Überschreiben von `virtual`-Methoden das Schlüsselwort `override`:

```
public new string NonVirtualMethod() {...}
public override string VirtualMethod() {...}
```

Die unterschiedliche Syntax wäre nicht gerechtfertigt, wenn es nicht auch einen semantischen Unterschied gäbe (mehr dazu im Abschnitt »Polymorphe Implementierung«, Seite 319). Abbildung 8.7 zeigt die Ausgabe des Programms und enthüllt insbesondere die Wirkung der `override`-Vereinbarung auf eine Objektvariable des Typs `MyBaseClass`, der ein Objekt der abgeleiteten Klasse zugewiesen wurde.

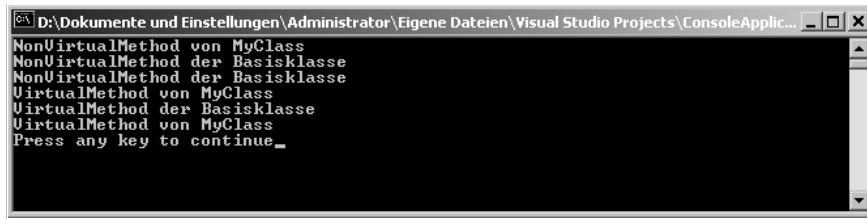


Abbildung 8.7: Ausgabe des Beispielcodes. Die letzte Zeile enthüllt das Wesen der virtuellen Methoden.



Auch das Überschreiben statischer Methoden ist möglich. Da es keine virtuellen statischen Methoden gibt, spielt hier nur das Schlüsselwort `new` *eine Rolle:*

```
class MyBaseClass
{
  public static string NonVirtualMethod() { ... }
}

class MyClass : MyBaseClass
{
  public static new string NonVirtualMethod() { ... }
}
```

Methoden werden nur verdeckt, nicht überschrieben

Wie schon angesprochen, ist der Begriff »Überschreiben« nicht gerade sehr glücklich gewählt – »Verdecken« wäre treffender gewesen. Das wird spätestens dann klar, wenn die Frage auftaucht, ob die Basisklassenvariante einer überschriebenen Methode in der abgeleiteten Klasse nicht doch noch irgendwie ansprechbar ist, und was bei weiterer Ableitung mit der `private`-Überschreibung einer Basisklasse passiert.

Die erste Frage ist schnell beantwortet: Im Gegensatz zu C++ verfügt C# mit dem Schlüsselwort `base` – als Pendant zu `this` – über einen Qualifizierer, der es ermöglicht, das Basisklassenobjekt generisch zu benennen. Überschriebene Basisklassenmethoden lassen sich damit jederzeit wieder »ausgraben«.

```
public new string NonVirtualMethod()
{
  return "NonVirtualMethod von MyClass" + " " + base.NonVirtualMethod();
}

public override string VirtualMethod()
{
  return "VirtualMethod von MyClass" + " " + base.VirtualMethod();
}
```



Das Delphi-Gegenstück zu `base` *ist das Schlüsselwort* `inherited`.

Angesichts dieses Ergebnisses dürfte denn auch nicht verwundern, dass eine einmal nur als `private` überschriebene Basisklassenmethode bei weiterer Ableitung wieder auftaucht:

```
class MyBaseClass
{
  public string NonVirtualMethod()
  {
    return "NonVirtualMethod der Basisklasse";
  }
}

class MyClass : MyBaseClass
{
  private new string NonVirtualMethod()
  {
    return "NonVirtualMethod von MyClass";
  }
}
class MyYoungestClass : MyBaseClass
{ }

class Class1
{
  static void Main(string[] args)
  {
    MyYoungestClass myc = new MyYoungestClass();
    Console.WriteLine (myc.NonVirtualMethod());
  }
}
```



Abbildung 8.8: Ausgabe des Beispielcodes

8.3.4 Virtuelle Methoden

Die `virtual`-Deklaration begründet eine eigenständige Deklarationsart für Methoden, die das Tor zur Welt der so genannten POLYMORPHEN IMPLEMENTIERUNG aufstößt. Damit bildet sie gleichzeitig das Rückgrat für die Konzepte »Schnittstelle« und »abstrakte Klasse«, die sich von gewöhnlichen Klassen darin unterscheiden, dass sie virtuelle Methoden auch ohne Implementierung deklarieren dürfen (abstrakte Klasse) bzw. müssen (Schnittstelle). Unter *polymorpher Implementierung* versteht man die Anwendung einer besonderen Vererbungsart (virtuelle Vererbung), die es ermöglicht, dass eine Objektvariable die Methoden und Eigenschaften des jeweils tatsächlich gebundenen Objekts anspricht – obwohl sie »nur« den Datentyp einer Basisklasse trägt.



Zum Verständnis mag es beitragen, wenn Sie sich vorstellen, dass die Vererbung virtueller Elemente eine Möglichkeit eröffnet, Code der abgeleiteten Klasse gegen die Vererbungsrichtung auf der Ebene der Basisklasse anzusprechen.

Die `virtual`-Deklaration einer Methode in einer Basis (Klasse oder Schnittstelle) begründet eine *Überschreibungslinie* für diese Methode. Abgeleitete Klassen können – und sollen – diese Methode überschreiben, um sie auf ihre eigenen Gegebenheiten anzupassen bzw. auszudifferenzieren. In diesem Fall erfordert die Deklaration (in Abweichung zu C++) anstelle von `virtual` das Schüsselwort `override` – bei ansonsten gleichem Prototyp und auch gleichem Zugriffsmodifizierer:

```
class Base
{
  public virtual void Virtual()
  {
    Console.WriteLine("Variante der Basis");
  }
}
class Derived : Base
{
  public override void Virtual()
  {
    Console.WriteLine("Variante der Klasse A ");
  }
}
```

Der folgende Testcode demonstriert die Wirkung der `virtual`-Deklaration:

```
Base b;
b = new Base();
b.Virtual();                  // Aufruf der Variante von Base
b = new Derived();
b.Virtual();                  // Aufruf der Variante von Derived
```



Der Modifizierer `sealed` ermöglicht es, der Überschreibungslinie einer virtuellen Methode auf der Ebene der aktuellen Klasse ein Ende zu setzen. Abgeleitete Klassen erben diese Methode dann nur noch, können sie aber nicht mehr überschreiben.

```
class MoreDerived : Derived
{
  sealed public override void Virtual()    // nicht mehr überschreibbar
  {
    Console.WriteLine("Variante der Klasse MoreDerived");
  }
}
```

8.3.5 Statische Methoden

Die am wenigsten spezialisierte Form der Methode ist die statische Methode. Sie deckt sich weitgehend mit dem Begriff der allgemeinen Funktion aus der klassischen Programmierung. Während die (in C# nicht mehr vorhandene) allgemeine Funktion grundsätzlich alle auf Modulebene angesiedelten Größen (Konstanten, Variablen, Funktionen, Datentypen) »sieht«, beschränkt sich das Blickfeld einer statischen Methode auf die statischen Elemente bzw. die statische Instanz der jeweiligen Klasse. Selbst wenn man die Klasse als Modul begreift, was sich ja anbietet, »sieht« die statische Methode erheblich weniger als die klassische Funktion, denn der gesamte mit Objekten verbundene Apparat (Instanzmethoden, Instanzfelder) bleibt ihr verborgen.

Charakteristisch für die statische Methode ist also, dass sie zur anonymen Ausstattung der Klasse gehört und Teil ihrer statischen Instanz ist. Im Einzelnen heißt das:

- Eine statische Methode sieht alle Elemente der statischen Instanz: Konstanten, Datentypen, statische Datenfelder, statische Methoden, statische Eigenschaften.
- Eine statische Methode hat *keine* Sicht auf konkrete Instanzen (Objekte) der Klasse und deren Elemente.
- Eine statische Methode ist umgekehrt aber für Instanzmethoden sichtbar und kann von diesen jederzeit und ohne Einschränkung aufgerufen werden.
- Für den Aufruf einer statischen Methode durch einen Client ist eine Qualifizierung mit dem Klassennamen erforderlich. Ein Aufruf über eine Objektvariable ist – im Gegensatz zu Java – nicht möglich.

Sieht man davon ab, dass sich statische Methoden nicht mit dem Zusatz `virtual` bzw. `override` vereinbaren lassen und das Überschreiben also gewissen Einschränkungen unterliegt, da virtuelle Methoden für die statische Instanz einer Klasse schlicht keinen Sinn ergeben würden, unterscheidet sich die statische Methode von der Syntax her nur durch das Schlüsselwort `static` von der dynamischen Methode. Vom Überladen her bestehen keine Unterschiede. Alle weiteren Unterschiede sind semantischer Natur. So hat in einer statischen Methode das Schlüsselwort `this` nichts zu suchen, da die statische Instanz kein Objekt im eigentlichen Sinne ist.

Syntax

```
[Modifizierer] static Rückgabetyp MethodenBez ([Paramliste])
{
  [Anweisungsblock]
}
```

Fehlt ein Zugriffsmodifizierer, vereinbart der Compiler die Methode standardmäßig als `private`.

Die Methode Main()

Eine besondere Rolle kommt der statischen Methode mit dem Namen `Main()` zu. Wie bereits mehrfach ausgeführt, stellt Sie den definierten Einsprungpunkt eines Programmmoduls (Projekts) dar und damit die oberste Prozedurebene eines C#-Programms. Ein Projekt ist nur dann *startfähig*, wenn darin eine (und nur eine) Methode `Main()` definiert ist und wenn es einen startfähigen Ausgabetyp besitzt. Die Einstellung des Ausgabetyps wird über die Eigenschaftsseite des Projekts vorgenommen, wobei hier die Wahl zwischen *Windows-Anwendung*, *Konsolenanwendung* und *Klassenbibliothek* besteht. Enthält ein Projekt keine Methode `Main()`, kann es nur zu einer Klassenbibliothek kompiliert werden, deren Code nicht startfähig und somit darauf angewiesen ist, von einem startfähigen Projekt aus angesprochen zu werden.

Für die Vereinbarung der Methode `Main()` gibt es einen gewissen Spielraum:

- Als Rückgabetyp kann `void` und `int` vereinbart werden.
- In der Praxis spielt es keine Rolle, mit welchem Zugriffsmodifizierer `Main()` vereinbart wird. Selbst bei einer Vereinbarung mit `private` geht die Startfähigkeit nicht verloren.
- `Main()` kann entweder parameterlos oder mit einem Stringarray als Parameter vereinbart werden. Das Stringarray liefert die Aufrufparameter in der Kommandozeile des Programms.

Das folgende einfache Programm gibt seine Aufrufparameter an der Konsole aus (vgl. auch Kapitel 3):

```
class Class1
{
  static void Main(string[] args)
  {
    foreach (string s in args)
    {
      Console.WriteLine(s);
    }
  }
}
```

Im Gegensatz zu bestimmten C-Versionen gibt der erste Aufrufparameter (also `args[0]`) nicht den Namen der ausführbaren Datei samt Pfad wieder, sondern das erste Kommandozeilenargument.

Um bei dem obigen Code überhaupt etwas zu sehen zu bekommen, lässt sich dem Programm für den Start aus VS.NET heraus eine Kommandozeile vordefinieren – die wohlgemerkt nur auch für diesen Fall gültig ist. Abbildung 8.9 zeigt den Eigenschaftsdialog des zugehörigen Projekts, Abbildung 8.10 die Ausgabe des Programms.

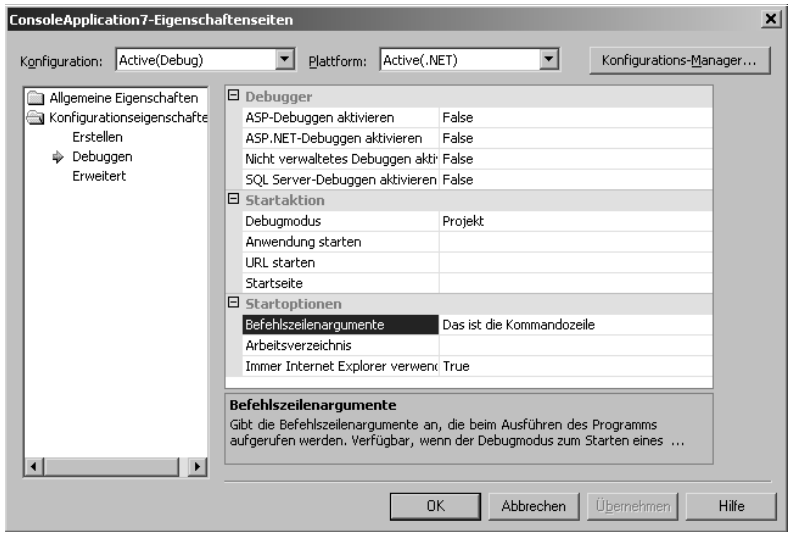


Abbildung 8.9: Angabe einer Kommandozeile für den Start der Anwendung aus VS.NET heraus



Abbildung 8.10: Ausgabe der Kommandozeilenargumente

Ist Main() wirklich der Einsprungpunkt?

Wenngleich `Main()` eine gewisse Eigenständigkeit besitzt, darf man nicht aus den Augen verlieren, dass die Methode aus syntaktischen Gründen immer einer Klasse angehören muss, deren statische Instanz noch vor dem Aufruf von `Main()` initialisiert wird. Das aber wiederum bedeutet, dass ein gegebenenfalls vorhandener statischer Konstruktor noch vor `Main()` zum Aufruf kommt, was die obige Aussage relativiert. Der eigentliche Einsprungpunkt ist also tatsächlich der statische Konstruktor der Klasse, die ihrerseits die Methode `Main()` definiert. Für die Praxis hat dies aber kaum Bedeutung, da der Aufgabenbereich des statischen Konstruktors (vgl. Abschnitt 8.3.7 »Konstruktoren«, Seite 281) nicht über die Initialisierung des statischen Datenfelder hinausgehen sollte. Dennoch ist folgende Code an sich legitim:

```
class Class1
{
  static Class1()
  {
    Console.WriteLine("im statischen Konstruktor");
  }
  static void Main(string[] args)
  {
    Console.WriteLine("in der Methode Main");
  }
}
```

und produziert die zu erwartende Ausgabe:



Abbildung 8.11: Der statische Konstruktor wird noch vor `Main` ausgeführt.

Damit der Einsprungpunkt definiert bleibt, darf in einem Projekt immer nur eine Klasse über die Methode `Main()` verfügen. Sind in einer Projektmappe mehrere Projekte versammelt, die eine Klasse mit einer `Main()`-Methode enthalten, muss das gewünschte Startprojekt auf der Eigenschaftsseite STARTPROJEKT der Projektmappe im Listenfeld EINZELNES STARTPROJEKT festgelegt werden (vgl. Abbildung 8.12). Die Option MEHRERE STARTPROJEKTE ermöglicht es, mehrere startfähige Projekte parallel – das heißt: in verschiedenen Prozessen – zu starten.

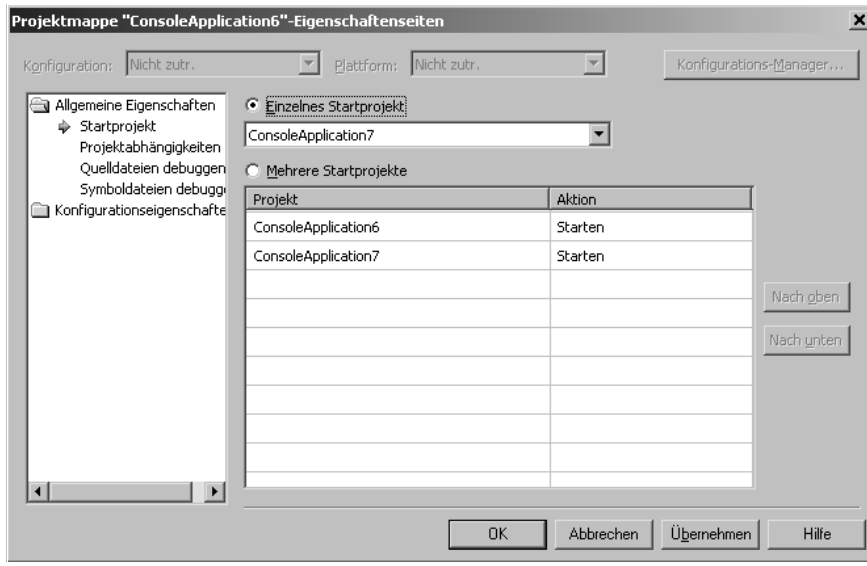


Abbildung 8.12: Festlegung eines Startprojekts

8.3.6 Instanzmethoden

Die *Instanzmethode*, oft auch *Objektmethode* genannt, ist das Gegenstück der statischen Methode auf Ebene der Objekte. Syntaktisch gesehen unterscheiden sich die beiden Methodenarten kaum: Ohne das Schlüsselwort `static` wird die statische Methode schlicht zur Instanzmethode. Semantisch gibt es hingegen gravierende Unterschiede.

Charakteristisch für die Instanzmethode ist, dass sie immer einen Ich-Bezug zu einem konkreten Objekt hat. Im Einzelnen heißt das:

- Eine Instanzmethode »sieht« *alle* in der Klassendefinition getroffenen Definitionen: Konstanten, Datentypen, statische und nicht-statische Methoden und Datenfelder.
- Eine Instanzmethode »sieht« immer nur die Datenfelder des eigenen Objekts, nicht die anderer Objekte. Die statischen Datenfelder hingegen werden von Instanzmethoden gemeinsam benutzt.
- Eine Referenz auf das eigene Objekt kann formal durch das Schlüsselwort `this` ausgedrückt werden. Wenn der Basisklassenanteil des Objekts gemeint ist, steht das Schlüsselwort `base` zur Verfügung. Eine Qualifizierung mit `this` oder `base` ist aber nur in speziellen Fällen wirklich notwendig, da alle Elemente des Objekts (und der Klasse) zum Kontext der Instanzmethode gehören und ohne weitere Qualifizierung ansprechbar sind.
- Eine Instanzmethode kann nur mit Bezug zu einem Objekt aufgerufen werden oder anders gesagt: Man benötigt ein Objekt, um eine seiner Instanzmethoden aufrufen zu können.
- Im Gegensatz zu statischen Methoden lassen sich Instanzmethoden bei Verwendung der Schüsselworte `virtual` bzw. `override` auch als virtuelle Methoden vereinbaren (mehr über das dahinter steckende, ausgesprochen mächtige Konzept finden Sie im Abschnitt »Überschreiben von Methoden«, Seite 272).

Syntax

```
[Modifizierer] Rückgabetyp Methode ([Paramliste])
{
  [Anweisungsblock]
}
```

8.3.7 Konstruktoren

Konstruktoren obliegt die Aufgabe, die Datenfelder der Klasse sowie der Instanzen zu initialisieren. Wie der Name schon sagt, spielen sie eine zentrale Rolle bei der Konstruktion der Klasse und ihrer Objekte. Das Kennzei-

chen eines Konstruktors ist erstens, dass er immer den Bezeichner der Klasse trägt, und zweitens, dass man für ihn keinen Rückgabewert deklarieren muss bzw. kann. Zudem kann er explizit nur in Verbindung mit dem `new`-Operator verwendet werden.

Konstruktoren lassen sich sowohl für `class`- als auch für `struct`-Klassen vereinbaren und sind dafür gedacht, die Klasse und ihre Objekte vor dem ersten Gebrauch in einen definierten Initialzustand zu versetzen.

Standardmäßig erbt jede Klasse je einen parameterlosen statischen und dynamischen Standardkonstruktor, deren Aufruf der Compiler automatisch veranlasst, wenn kein anderer Konstruktor vereinbart ist.

Das von C++ her bekannte Gegenstück des Konstruktors, der *Destruktor*, ist in C# zwar nach wie vor vorhanden, spielt hier aber eine untergeordnete Rolle, da sein Aufgabenbereich aufgrund der von der .NET-Infrastruktur durchgeführten automatischen Freispeicherverwaltung erheblich kleiner geworden ist. Auch hat sich seine Semantik verändert. Die explizite Bereitstellung eines Destruktors ist daher nur noch in seltenen Fällen – etwa bei Verwendung unsicheren Codes – wirklich notwendig. (Mehr darüber im Abschnitt »Destruktoren«, Seite 286.)

Syntax für struct-Klassen

Die Syntax für Instanzkonstruktoren von `struct`-Klassen lautet:

```
[Modifizierer] StructName (Param[, Param ...]) [: this.(Param[, Param ...]]
{
  [Anweisungsblock]
}
```

In `struct`-Klassen darf kein parameterloser Instanzkonstruktor überladen werden. Ein solcher wird vererbt und sorgt dafür, dass alle Datenfelder einer `struct`-Instanz mit der nötigen Bitbreite auf den Wert 0 bzw. `null` gesetzt werden.

Der optionale Doppelpunktzusatz ermöglicht es, den Aufruf auf einen anderen innerhalb derselben Klasse definierten Konstruktor zurückzuführen, wobei dessen Aufruf grundsätzlich vor Eintritt in den eigenen Anweisungsblock erfolgt und mit `this` notiert wird (delegierte Konstruktion).

Statischer Konstruktor

Für die Initialisierung von `static`-Datenelementen kann ein parameterloser statischer Konstruktor definiert werden:

```
[Modifizierer] static StructName ()
{
  [Anweisungsblock;]
}
```

Dieser Konstruktor kommt implizit und nur ein einziges Mal zum Aufruf, wenn die Strukturklasse das erste Mal angesprochen wird, sei es im Zuge einer Instanzvereinbarung oder bei Aufruf einer statischen Methode. Nicht empfehlenswert ist die Vereinbarung eines parameterlosen Konstruktors mit leerem Anweisungsblock, da dieser die Objektinitialisierung unnötig verlangsamt.

Der folgende Code stellt die wichtigsten Konstruktor-Szenarien für `struct`-Klassen vor:

```
struct MyStruct
{
  static int StaticField;
  public int InstanceField1;
  public int InstanceField2;
  static MyStruct() // statischer Konstruktor
  {
    StaticField = 12;
  }
  public MyStruct(int i)
  {
    InstanceField1 = 0;   // alle Felder müssen einen Wert bekommen!
    InstanceField2 = i;
  }
  public MyStruct(int i, int k) : this(i)
  {
    InstanceField = k;
  }
}
...
MyStruct te = new MyStruct(2, 2);
```

Der Debugger enthüllt, dass die letzte Codezeile zum Aufruf aller drei Konstruktoren führt – zuerst der statische (nur wenn die Klasse an dieser Stelle das erste Mal angesprochen wird), dann der einparametrige und schließlich der zweiparametrige.

Die gleichfalls zulässige Vereinbarung

```
MyStruct te;
```

würde hingegen den statischen sowie unsichtbar den ererbten parameterlosen Instanzkonstruktor auf den Plan rufen, mit dem Effekt, dass das statische Feld den Wert 12 erhält, die Instanzfelder hingegen den Wert 0.

Syntax für class-Klassen

Die Syntax für `class`-Klassen lautet:

```
[Modifizierer] KlassenName ([Paramliste]) [: this([Paramliste]
{
  [Anweisungsblock;]
}
```

bzw.

```
[Modifizierer] KlassenName ([Paramliste]) [: base([Paramliste]
{
  [Anweisungsblock;]
}
```

Bei `class`-Klassen sieht die Syntax für die Konstruktorvereinbarung vom Prinzip her also genauso aus wie bei `struct`-Klassen, im Detail gibt es jedoch einige Unterschiede. So ist die Definition eines parameterlosen Instanzkonstruktors für `class`-Klassen erlaubt – und in der Tat auch sehr häufig zu finden.

Zudem kommt die Vererbung ins Spiel, sodass zusätzlich auch noch die Konstruktoren der Basisklasse für den delegierten Teil der Konstruktion im Angebot stehen – die Qualifikation findet dann mit `base` statt. Auch hier kommt der delegierte Konstruktor in jedem Fall *vor* Ausführung des eigenen Anweisungsblocks zum Aufruf, sodass der Konstruktor selbst gewissermaßen »das letzte Wort« hat (sofern er nicht seinerseits von einem anderen Konstruktor vorgeschoben wird).



Der parameterlose Basisklassenkonstruktor muss übrigens nicht eigens angegeben werden, da er grundsätzlich auch implizit (je nach Ausstattung der Basisklasse in der überschriebenen oder generischen Fassung) zum Aufruf kommt, sofern keine andere Variante des Basisklassenkonstruktors explizit angegeben wird. Aus der Zeile:

```
public MyClass() {...}
```

macht der Compiler also von sich aus:

```
public MyClass() : base() {...}
```

Statischer Konstruktor

Für die Initialisierung von `static`-Datenelementen besteht in `class`-Klassen die Möglichkeit, einen parameterlosen statischen Konstruktor zu definieren:

```
[Modifizierer] static KlassenName()
{
  [Anweisungsblock;]
}
```

Dieser Konstruktor kommt implizit und nur ein einziges Mal während des Programmlaufs zum Aufruf und zwar, wenn die Klasse das erste Mal angesprochen wird, sei es im Zuge einer Instanzvereinbarung oder bei Aufruf eines statischen Elements.

Aufruf des Konstruktors

Die oft vertretene Position, ein Konstruktor gebe deshalb keinen Funktionswert zurück, weil keiner deklariert ist, ist nur halb richtig. Da Konstruktoren nur im Zusammenspiel mit dem `new`-Operator explizit aufgerufen werden können, erübrigt sich aber eine Diskussion darüber – die Geschehnisse selbst werden nämlich von der Sprache schlicht verkapselt.

Fakt ist, dass der Ausdruck

```
new MyClass();
```

eine neue (zunächst unbenannte) Instanz der `class`-Klasse `MyClass` generiert und deren Adresse verkörpert. Im Allgemeinen wird man diese Adresse in einer Objektvariablen geeigneten Typs festhalten, um das Objekt weiterhin ansprechen zu können. Also:

```
MyClass mc = new MyClass();
```

Möglich ist aber auch die Übergabe einer unbenannten neuen Instanz an eine Methode, die ein Objekt entsprechenden Typs erwartet. Ein prominentes Beispiel dafür ist die folgende Anweisung, die in jedem mit der Vorlage »Windows-Anwendung« generierten Programmgerüst zu finden ist:

```
Application.Run(new MyClass());
```

In seltenen Fällen wird man eine unbenannte Instanz generieren, nur um kurz eine ihrer Methoden zu verwenden oder eine Eigenschaft abzufragen:

```
Type BasisTyp = (new MyClass()).GetType().BaseType;
```

Überladene Konstruktoren aufrufen

Da sich für eine Klasse beliebig viele Konstruktoren definieren (überladen) lassen, ist auch eine Konstruktion mit spezifischer Initialisierung möglich. In diesem Fall werden die Initialisierungswerte als Parameter übergeben, und es kommt der Konstruktor zum Aufruf, dessen Signatur (Parameterliste) passt. Also etwa:

```
MyClass mc = new MyClass( ref i, 10 * AnderesObjekt.Datenfeld1, out j);
```

Auch `struct`-Objekte, die ja Werttypen verkörpern, lassen sich per Konstruktor in Verbindung mit dem `new`-Operator instanziieren.

```
MyStruct ms = new MyStruct(1, 2, 3, 4);
```

Bei `struct`-Klassen gilt allerdings die Einschränkung, dass der parameterlose Konstruktor – zusätzlich zu dem, dass er nicht überschreibbar ist – auch nicht explizit notiert werden kann. Er kommt bei Verwendung der Standardsyntax für Werttypen implizit zum Aufruf:

```
MyStruct ms;   // involviert Aufruf des generischen Standardkonstruktors
```

Regeln für die Konzeption von Konstruktoren

Nachdem das Objekt bei Ausführung des Konstruktors noch nicht fertig ist (Kunststück: der Konstruktur ist ja noch an der Arbeit), gelten für den Anweisungsblock eines Konstruktors besondere Regeln.

- Ein Konstruktor sollte sich auf die Initialisierung der Datenfelder beschränken. Weitergehende Operationen fallen in den Aufgabenbereich von Methoden.
- Ein Konstruktor darf zur Vermeidung redundanten Codes einen zweiten Konstruktor der Klasse (`this`) oder Basisklasse (`base`) aufrufen, vorausgesetzt, er kann die dafür notwendigen Parameter bereitstellen. Der eigene Anweisungsblock wird erst nach Ausführung dieses zweiten Konstruktors ausgeführt.
- Da dem Aufruf eines selbst definierten Konstruktors grundsätzlich die Ausführung des generischen Konstruktors vorausgeht, sind alle Datenfelder auf den Wert 0 bzw. `null` vorinitialisiert. Der Konstruktor kann daher problemlos Methoden aufrufen, die mit diesen Datenfeldern arbeiten, auch wenn er sie selbst noch nicht initialisiert hat.

8.3.8 Destruktoren

Wie bereits im vorigen Abschnitt angeklungen, stellt der Destruktor so etwas wie ein Gegenstück des Konstruktors dar. Während die Rollen von Konstruktor und Destruktor in C++ und Delphi noch weitgehend paritätisch verteilt sind – der Konstruktor kümmert sich um den Aufbau und die Initialisierung des Objekts, er fordert also die dafür nötigen Ressourcen an, und der Destruktor betreibt den Abbau des Objekts sowie die Freigabe der Ressourcen – bietet sich in C# ein ganz anderes Bild: Da die automatische Freispeicherverwaltung der .NET-Laufzeitumgebung einen automatisierten Abbau aller .NET-Objekte, auf die keine Referenzen mehr existieren, betreibt, ist der Destruktor in C# im eigentlichen Sinne des Begriffs arbeitslos geworden.



Solange Sie also mit .NET-Klassen arbeiten, können Sie sich umgekehrt darauf verlassen, dass alle »Geister, die Sie riefen«, früher oder später sicher wieder verschwinden – und zwar völlig ohne Ihr Zutun.

In der Praxis kann aber doch nicht auf ihn verzichtet werden. Dass es den Destruktor in C# noch gibt, liegt daran, dass es trotz alledem noch eine Reihe von Situationen gibt, in denen ein Destruktor tatsächlich noch gebraucht wird – oder auch nur »von Nutzen ist«. Eine davon schafft offensichtlich die Programmierung mit `unsafe`-Code. Eine andere ergibt sich im Zusammenhang mit der Belegung logischer Ressourcen – geöffnete Kommunikationsverbindungen zu Dateien, Datenbanken usw. Ganz arbeitslos ist der Konstruktor also auch in C# nicht.

Generell gesagt, werden Sie einen Destruktor bereitstellen müssen, wann immer Komponenten eines anderen Objektmodells, beispielsweise COM-Objekte oder externe Ressourcen im Spiel sind, die nicht (oder nicht vollständig) von der .NET-Laufzeitumgebung verwaltetet werden.

Wer schon einmal größere Projekte in C++ programmiert hat und weiß, welchen Aufwand es bedeutet, Zombie-Objekten und Speicherverlusten aufgrund vergessener Freigaben nachspüren zu müssen, wird angesichts der Verhältnisse bei C# zu den drei Kreuzen noch ein viertes oder gar ein fünftes schlagen. In der Tat gibt es für C++ schon lange Überlegungen, eine automatische Freispeicherverwaltung einzuführen, doch die bisherigen Vorstöße in diese Richtung haben ihren Weg in den Standard noch nicht gefunden. Die .NET-Offensive »C++ mit verwalteten Erweiterungen« könnte allerdings die Vorhut für eine solche Entwicklung spielen.

Asynchrone Destruktion

Einer der grundlegenden Unterschiede zwischen der Destruktion von .NET-Objekten und der Destruktion von C++- Delphi- oder VB-Objekten ist der Zeitpunkt. Bisher sah die objektorientierte Programmierung einen *synchronen* (also: deterministischen) Aufruf des Destruktors vor, und zwar *bei* Erlöschen des Geltungsbereichs des Objekts. In der verwalteten Umgebung von .NET hingegen geschieht die Destruktion hingegen asynchron zu irgendeinem nicht allzu fernen, aber unbestimmten Zeitpunkt *nach* Erlöschen des Geltungsbereichs des Objekts.

Aus semantischer Sicht kann dies eine gravierende Auswirkung auf die Programmlogik haben, denn:

- Es nicht klar, in welcher Reihenfolge .NET-Objekte zerstört werden.
- Es nicht klar, zu welchem Zeitpunkt .NET-Objekte zerstört werden. Man kann aber im Allgemeinen davon ausgehen, *dass* die Freispeicherverwaltung in hinreichend regelmäßigen Intervallen tätig wird.
- Es besteht noch nicht mal die Garantie, *dass* ein .NET-Objekt überhaupt zu irgendeinem Zeitpunkt vor dem Programmende abgebaut wird. Beispielsweise kann der Destruktor eines Objekts in eine Endlos-

schleife geraten oder aufgrund irgendwelcher Sperren eine unbestimmte Zeit in Wartestellung verharren, sodass der für die Objektzerstörung zuständige Thread seine Arbeit nicht mehr fortsetzen kann.

Vom Prinzip her wäre es natürlich denkbar, von einem C#-Programm aus die .NET-Freispeicherverwaltung explizit anzustoßen. Das würde in der Tat den Zeitpunkt der Destruktion etwas besser greifbar machen. Rein formal ist dafür auch nichts weiter als ein Aufruf der statischen Methode `System.GC.Collect()` erforderlich. Mit Blick auf die Laufzeiterfordernisse ist von einem solchen Aufruf aber eher abzuraten und erinnert wohl mehr an die Geschichte mit »den Kanonen und den Spatzen« (oder an die API-Funktion `GlobalCompact()` aus den Zeiten von Windows 3.x) als an eine elegante Lösung. Der Aufruf sollte daher als letztes Mittel angesehen werden, selten benutzte, aber extrem wertvolle Ressourcen etwas schneller freizugeben, als es ohnehin passieren würde. Ein Beispiel wäre etwa, dass ein Objekt von derselben Ressource wie ein anderes Objekt Gebrauch machen will, und die Gewissheit benötigt, dass die Destruktion des anderen Objekts zum gegebenen Zeitpunkt wirklich erfolgt ist.

Was die explizite Freigabe von Ressourcen betrifft, so führen in C# drei an sich recht unterschiedliche, aber letztlich wieder ineinander verwobene Wege zum gleichen Ziel:

- C#-Destruktor – eine explizit als Destruktor gekennzeichnete Methode, die der C#-Compiler sprachintern in einen adäquaten Aufruf der `Finalize()`-Methode verwandelt. Einen solchen Destruktor müssen Sie in jedem Fall implementieren, wenn Sie unverwaltete Aufrufe (Plattformaufrufe) und DLLs einsetzen.
- `Dispose()`-Methode – eine gewöhnliche Methode mit konventionalisierter Bedeutung, die die wahlfreie, explizite Freigabe belegter Ressourcen ermöglicht, dabei aber eine gewisse Unterstützung durch die Sprachsyntax genießt.
- `Close()`-Methode – eine gewöhnliche Methode mit konventionalisierter Bedeutung, die einem Client des Objekts die wahlfreie, explizite und synchrone Freigabe belegter Ressourcen ermöglicht.

Syntax – Bereitstellung eines Finalize()-Destruktors

Die `Finalize()`-Methode hat eine gewisse Tradition in der objektorientierten Programmierung – und dürfte Ihnen vielleicht auch schon von VB6 her bekannt sein. Sie verkörpert den optionalen Destruktor für die objektorientierte Programmierung in verwalteten Umgebungen, in denen die Destruktion an sich automatisch erfolgt. .NET ist eine solche Umgebung, und so ist es kein Wunder, dass diese Methode von vielen .NET-Basisklassen implementiert wird, die etwas mit systemweiten Ressourcen (beispielsweise mit Handles) zu tun haben.

Um so überraschender dürfte es sein, dass eine Methode mit diesem Bezeichner in C# weder auftaucht und noch implementiert werden kann. Das liegt daran, dass C# die Destruktorsyntax von C++ adaptiert hat und der Compiler sich vorbehält, die `Finalize()`-Methode implizit selbst zu vereinbaren. Die Destruktorvereinbarung sieht so aus:

```
~MyClass()
{
 ...
}
```

Die vorangestellte Tilde symbolisiert die »Negierung« des Standardkonstruktors. Der Destruktor hat keine Parameter, keinen Rückgabetyp und keine Modifizierer. Insbesondere besteht also auch nicht die Möglichkeit, einen statischen Destruktor zu vereinbaren, der für die statische Instanz der Klasse zuständig ist.

Der Compiler generiert aus der Destruktorsyntax implizit die folgende Vereinbarung:

```
protected override void Finalize()
{
  try
  {
    // der Ccmpiler fügt den Code von ~MyClass()hier ein!
  }
  finally
  {
    base.Finalize();
  }
}
```

Bestünde in C# die Möglichkeit, eine `Finalize()`-Methode zu vereinbaren, müsste der Compiler den expliziten Aufruf der Methode unterbinden und den Aufruf der Basisklassenvariante erzwingen oder implizit hinzufügen. Die für C# geltende Syntax ist hier auf jeden Fall aussagekräftiger.

Mit anderen Worten: Der Destruktor steht (wie üblich) für den expliziten Aufruf *nicht* zur Verfügung! Die .NET-Freispeicherverwaltung ruft ihn intern automatisch für alle verwaisten Objekte auf, wenn sie feststellt, dass keine Objektvariable mehr eine Referenz darauf enthält.

Die Implementierung eines Destruktors garantiert, dass die für ein Objekt zusätzlich notwendigen Aufräumarbeiten irgendwann passieren, sobald auf das Objekt nicht mehr referiert wird. Wann das aber sein wird, darauf hat der Programmierer, wie gesagt aufgrund der Asychronizität, keinen Einfluss. Außerdem ist es auch nicht einsichtig, dass die Freigabe einer Ressource unbedingt auch eine Zerstörung des Objekts, das sie angefordert hat, implizieren muss – zumal, wenn der Auf- und Abbau des Objekts laufzeitintensiv ist und ein Objekt-Caching (vgl. COM+) Vorteile bringt.

Kurzum: Wenn es um die Freigabe knapper oder sehr begehrter Ressourcen geht – etwa um Datensatzsperren oder Kommunikationsverbindungen – reicht es im Allgemeinen nicht, die Hände in den Schoß zu legen und sich auf das Einsetzen der Freispeicherverwaltung zu verlassen. Hier ist ein Mechanismus gefragt, den der Client des Objekts wahlfrei von sich aus aktivieren kann. Die Lösung bringt die Methoden `Dispose()` und `Close()` ins Spiel.

Dispose() und Close()

Um es gleich vorweg zu nehmen, `Dispose()` und `Close()` sind gewöhnliche Methoden, deren Semantik mit Blick auf das große .NET-Ganze schlicht konventionalisiert ist. Vordergründig betrachtet, könnten beide Methoden also beliebige Aufgaben verrichten, ohne dass einem der Compiler oder das Laufzeitsystem irgendwie in die Quere kommen würden. Hintergründig betrachtet, fällt den beiden Methoden aber aufgrund einer unter .NET bestehenden Konvention eine festgelegte Rolle im Zusammenhang mit der Ressourcenfreigabe zu, wobei die Sprache zumindest für `Dispose()` optional noch eine gewisse syntaktische Unterstützung bereithält – dazu aber gleich noch mehr.

```
public void Dispose()
{
  // aktive, aber finale Freigabe der Ressourcen
}
```

und

```
public void Close()
{
  // aktive Freigabe ggf. erneut belegter Ressourcen
}
```

Die Form beider Methoden an sich ist gleich und es reicht, nur eine zu implementieren – wiewohl es auch Situationen gibt, in denen man besser beide implementiert. Der Bezeichner `Dispose()` suggeriert mehr die Bedeutung einer finalen Entsorgung, wenn das Objekt ausgedient hat, während `Close()` traditionell die Bedeutung einer Freigabe unabhängig vom Verlust der jeweiligen Instanz hat und meist das Gegenstück zu einer `Open()`-Methode bildet.

Das hinter diesen Methoden steckende Kalkül ist einfach: Um eine explizite Freigabe wichtiger Ressourcen zu erreichen, stellt man schlicht eine der beiden oder beide Methoden bereit, die der Client dann nach Gutdünken aufrufen kann, um eine Freigabe zu erzwingen, wenn er es für opportun erachtet.

Dem Client eine solche Methode optional anzubieten, heißt aber noch lange nicht, dass dieser sie auch benutzt. Da ein Objekt den Client nicht zwingen

kann, eine bestimmte Methode aufzurufen, muss die Implementierung der Klasse sicherstellen, dass die Freigabe auch erfolgt, wenn der Client »schlechte Manieren« hat, oder eben keinen Wert auf eine zügige Freigabe legt und auf den Destruktor vertraut:

```
public void Dispose()
{
  FreeIt();                  // deterministische Freigabe der Ressourcen
  GC.SuppressFinalize(this); // Finalize kann entfallen
}

private void FreeIt()
{
  ...
}

~MyBaseClass()
{
  FreeIt();                  // nondeterministische Freigabe der Ressourcen
}
```

Da `Dispose()` den Destruktoraufruf im Allgemeinen obsolet macht, sollte die Methode durch einen Aufruf der statischen Methode `GC.SuppressFinalize(this)` dafür Sorge tragen, dass die Freispeicherverwaltung auf die Ausführung des Destruktors verzichtet.

Die Schnittstelle IDisposable

Zweifellos kann ein allzu eifriger Aufruf von `Dispose()` auch ins Auge gehen, denn nichts und niemand hält den Client davon ab, `Dispose()` zu einem Zeitpunkt aufzurufen, an dem die Ressource eigentlich noch benötigt wird.

Nicht zuletzt aus diesem Grund, aber auch, um eine bessere Analogie zu dem Destruktorkonzept von C++ bereitzustellen, sieht C# eine gewisse Unterstützung für den automatischen Aufruf von `Dispose()` vor. Grundlage dafür ist:

- Das Schlüsselwort `using`, das hier als Anweisung auftritt und nicht als Direktive wie im Zusammenhang mit der allgemeinen Qualifizierung von Namensbereichen.
- Eine Ableitung der jeweiligen Klasse von der Schnittstelle `IDisposable`.
- Eine `public`-Implementierung der Methode `Dispose()`.

Wie an anderer Stelle erläutert, geht eine von einer Schnittstelle abgeleitete Klasse die Verpflichtung ein, eine Implementierung für alle in der Schnittstellenbeschreibung genannten Methoden bereitzustellen. Im Falle der Schnittstelle `IDisposable` benennt die Schnittstellenbeschreibung eine einzige Methode, die – wie könnte es anders sein – den Bezeichner `Dispose()` trägt und den folgenden Prototyp hat:

```
public void Dispose();
```

Bezogen auf eine Klasse namens `MyClass` würde das Ganze also etwa so aussehen:

```
class MyClass : IDisposable
{
  public void Dispose()
  {
    ...             // Ressourcen deterministisch freigeben
  }
...
}
```

Nun kommt der spannendere Teil. Nach dieser Vorarbeit ist der Compiler in der Lage, für einen automatischen Aufruf von `Dispose()` zu sorgen. Die Syntax dafür erinnert an `try/catch/finally`-Blöcke für die Ausnahmebehandlung und ist auch in der Tat nichts anderes, mit dem Unterschied, dass es sich hier eben um einen `using`-Block handelt:

```
...
using (MyClass mc = new MyClass())
{
  // Einsatz des MyClass-Objekt
} // hier endet der Geltungsbereich von mc, der Compiler ruft mc.Dispose()
```

In dieser Syntax ist kein expliziter `Dispose()`-Aufruf erforderlich, da der Compiler diesen implizit am Ende des `using`-Blocks einsetzt.

Die syntaktische Unterstützung für `Dispose()` mag zwar auf den ersten Blick beeindrucken, in der Praxis ist sie aber nichts weiter als reine Augenwischerei. Mit der zusätzlichen Klammerebene bedeutet sie weder eine wirkliche Erleichterung noch verhindert sie, dass der Client von sich aus `Dispose()` aufruft oder das Objekt auf anderem Weg (über eine andere Objektvariable) den `using`-block »überlebt«. Ein deterministischer Destruktor, wie man ihn von C++ her kennt, resultiert aus *dieser* Syntax noch lange nicht.

Codebeispiel – Datenbankverbindung auf- und abbauen

Das folgende Beispielprojekt *DisposeDemo* zeigt das Zusammenspiel zwischen `Open()`, `Close()`, `Dispose()` und einem Destruktor für eine Klasse namens `MyDatabaseConnection`. Die Klasse stellt Methoden für den Aufbau, die Statusabfrage und das explizite Schließen einer Datenbankverbindung bereit, ohne jedoch faktisch eine Verbindung mit einer Datenbank herzustellen. »Vergisst« der Client, die zuvor geöffnete Datenbankverbindung zu schließen, sorgt der Destruktor der Klasse dafür, dass die Verbindung automatisch abgebaut wird. Eine von der Implementierung gepflegte Statusvariable ermöglicht es, die Geschehnisse über die Methode `PrintStatus` an der Konsole gut zu verfolgen. Hier zunächst der Quellcode der Klasse `MyDatabaseConnection`:

```
class MyDatabaseConnection : IDisposable
{
  private string Status;
  private bool ConnectionDisposed = false;
  private string ConnectString;
  private int DBHandle = 0;

  public MyDatabaseConnection(string DBConnect)
  {
    ConnectString = DBConnect;
    Status = "Neues Objekt, Connect-String ist: " + DBConnect;
  }

  public bool Open()
  {
    if (DBHandle > 0)
    {
      Status = "Open gescheitert: DB-Verbindung besteht schon";
      return false;
    }
    if (ConnectionDisposed)
    {
      Status = "Open gescheitert: DB-Verbindung nicht mehr zu öffnen";
      return false;
    }
    DBHandle = OpenDB(ConnectString);
    if (DBHandle <= 0)
    {
      Status = "Open gescheitert: DB konnte nicht geöffnet werden";
      return false;
    }
    Status = "Open ist passiert, DB-Verbindung geöffnet";
    return true;
  }

  private int OpenDB(string Connect)
  {
    return 1;  // Hier wird eigentlich die Verbindung aufgebaut
  }

  public void Close()
  {
    Status = "Close ist passiert";
    DBHandle = 0;
    // hier eigentlich die Verbindung abgebaut
  }

  public void Dispose()
  {
    ConnectionDisposed = true;
    Close();
    GC.SuppressFinalize (this);
    Status = "Dispose ist passiert";
  }
```

```
  public void PrintStatus()
  {
    Console.WriteLine(Status);
  }

  ~MyDatabaseConnection()
  {
    Close();
    Status = "Destruktor wird ausgeführt";
    PrintStatus();
  }
}
```

Der einparametrige Konstruktor versorgt das neue Objekt mit der notwendigen Verbindungsinformation, sodass es die Verbindung umgehend mit `Open()` herstellen kann. `Close()` ermöglicht es, die Verbindung jederzeit zu schließen, um sie zu einem späteren Zeitpunkt gegebenenfalls erneut zu öffnen. Ist hingegen einmal ein `Dispose()`-Aufruf passiert, lässt sich die Verbindung nicht mehr eröffnen. Sollte der Client den `Dispose()`-Aufruf unterlassen – und auch nur dann –, kommt der Destruktor bei Verwaisung des Objekts zum Aufruf. In der bestehenden Implementierung gibt er von sich aus die entsprechende Statusinformation aus, da der Client auf das Objekt zu diesem Zeitpunkt ja keinen Zugriff mehr hat.

Der folgende `Main()`-Code testet die Klasse. Er zeigt erstens die `using`-Syntax (nebst ihren Grenzen) und lotet zweitens das Verhalten eines `MyDatabaseConnection`-Objekts in irregulären Situationen aus.

```
class Class1
{
  static void Main(string[] args)
  {
    MyDatabaseConnection MyDBC1;
    // Demo der using-Syntax
    using (MyDatabaseConnection MyDBC =
      new MyDatabaseConnection("Erster Verbindungsstring"))
    {
      MyDBC.PrintStatus();
      MyDBC.Open();
      MyDBC.PrintStatus();
      MyDBC.Close();
      MyDBC.PrintStatus();
      MyDBC.Open();
      MyDBC.PrintStatus();
      MyDBC.Open();               // Fehler zu erwarten
      MyDBC.PrintStatus();
      MyDBC1 = MyDBC;
    }    // hier kommt Dispose() ... aber MyDBC1 verschleppt das Objekt
    MyDBC1.PrintStatus();
    MyDBC1.Open();              // Fehler zu erwarten
    MyDBC1.PrintStatus();
```

```
    // neues Objekt anlegen
    MyDBC1 = new MyDatabaseConnection("Zweiter Verbindungsstring");
    MyDBC1.PrintStatus();
    MyDBC1.Open();
    MyDBC1.PrintStatus();
  }
}
```

Abbildung 8.13 zeigt die Ausgabe des Programms. Man beachte, dass die Objektvariable `MyDBC1` hier bewusst eine Referenz auf das Objekt des `using`-Blocks jenseits der Blockgrenzen verschleppt und die Grenzen der syntaktischen `Dispose()`-Unterstützung durch C# aufzeigt. Auch kommt der Destruktor nur für das zweite Objekt zum Aufruf, da der Testcode keinen `Dispose()`-Aufruf dafür enthält.

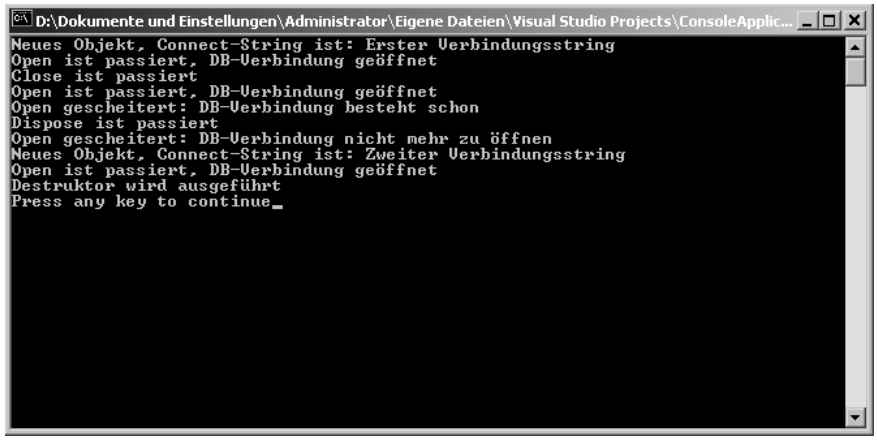


Abbildung 8.13: Ausgabe des Beispielprogramms

8.3.9 Eigenschaften

Bei der vierten speziellen Erscheinungsform einer Methode handelt es sich um die EIGENSCHAFTEN. Man versteht darunter eine zweiteilige Methode, die den kontrollierten schreibenden und lesenden Zugriff auf einen Wert vermittelt und über die eine Instanz ihren Zustand reflektiert bzw. manipulierbar macht. Der Wert kann als Datenfeld in der Klasse verankert sein, sich aber auch schlicht durch Berechnung ergeben – wie das bereits vorgestellte Codebeispiel »Primzahlen« demonstriert (vgl .ab Seite 157).

Wer von VB herkommt, dem werden Eigenschaften in Form von Property Get- und Property Set-Methoden geläufig sein; Delphi-Programmierern ist das Konzept ebenfalls bestens vertraut. Die syntaktischen Mittel, die C# für die Implementierung von Eigenschaften bereitstellt, sind allerdings um einiges intuitiver und gleichzeitig übersichtlicher.



Um einzelne Datenfelder einer Instanz nach außen hin sichtbar zu machen, besteht grundsätzlich die Möglichkeit einer `internal` oder `public`-Vereinbarung. So vereinbarte Datenfelder zählen an sich also bereits zu den Eigenschaften. Der Client der Instanz kann in diesem Fall allerdings ohne Einschränkung mit der Eigenschaft verfahren und den Wert nach Belieben und unter Ausschöpfung des vollen Wertebereichs für das Datenfeld modifizieren. Das ist nicht in jedem Fall erwünscht. Ein Schreibschutz lässt sich vom Prinzip her zwar über eine `readonly`-Vereinbarung des jeweiligen Datenfelds realisieren, er hat jedoch den »klitzekleinen« Nachteil, dass auch der Code der Klasse selbst – außerhalb des Konstruktors – dann nicht mehr schreibend auf das Datenfeld zugreifen kann.

Anforderungen für Eigenschaften

Das Konzept der Eigenschaften setzt die Idee um, dass die einzelne Instanz eine genaue Kontrolle darüber hat, was mit einem Datenfeld passiert. Im Einzelnen bedeutet das:

- *Einhaltung des Gültigkeitsbereichs* – die für ein Datenfeld zulässigen Werte decken sich oft nicht mit dem vollen Wertebereich des zugrunde liegenden Datentyps. Die Instanz sollte Verletzungen des Gültigkeitsbereichs beim Setzen eines Datenfelds erkennen und darauf reagieren können.
- *Wahrung der Integrität* – die Werte einzelner Datenfelder können auch voneinander abhängig sein. Damit die Instanz nicht in Gefahr gerät, einen ungültigen Zustand anzunehmen, sollten Wertänderungen eines Datenfelds auch auf andere Datenfelder durchschlagen können.
- *Durchsetzung von Schreibsperren* – ein Datenfeld sollte sich temporär oder ständig gegen Wertänderungen durch den Client schützen lassen, ohne dass der Code der Klasse von einer solchen Schreibsperre betroffen ist.
- *Durchsetzung von Lesesperren* – ein Datenfeld sollte sich temporär oder ständig gegen einen Lesezugriff durch den Client schützen lassen, ohne dass der Code der Klasse von einer solchen Lesesperre betroffen ist.

Dieser Anforderungskatalog macht deutlich, dass gar kein Weg daran vorbei führt, Eigenschaften anders als in Form von Methoden zu implementieren. Allerdings wäre es wünschenswert, Eigenschaften, hinter denen Methoden stecken, wie Datenfelder notieren zu können. Das heißt, sowohl der Wertzugriff als auch die Wertzuweisung sollte ohne die schwerfällige Methodensyntax möglich sein – und genau diesen Weg geht C# auch.

Syntax

Syntaktisch zerfällt die Definition einer Eigenschaft in zwei Komponenten, von denen mindestens eine vorhanden sein muss: in einen `get`-Accessor und einen `set`-Accessor:

```
[Modifizierer] Datentyp EigenschaftName
{
  [get
   {
     [Anweisungsblock;]
      return Rückgabetyp
   }]
  [set
   {
     [Anweisungsblock;]
   }]
}
```

Der `get`-Accessor kommt immer dann zum Aufruf, wenn der Wert der Eigenschaft abgefragt wird. Er hat die Pflicht, den Wert der Eigenschaft als Funktionswert zurückzugeben. Für Wertänderungen hingegen ist der `set`-Accessor zuständig.

Innerhalb des `set`*-Accessors ist der neue Wert für die Eigenschaft über das Schlüsselwort* `value` *verfügbar.*



Fehlt der `set`-Accessor, kann der Client den Wert der Eigenschaft zwar abfragen, nicht jedoch ändern; fehlt der `get`-Accessor, kann er ihren Wert zwar setzen, nicht jedoch abfragen.

Gebrauch

Es versteht sich, dass eine Eigenschaft nicht unbedingt ein Datenfeld als »alter ego« benötigt. Die Methode kann den Wert auch aus anderen Datenfeldern errechnen oder auf beliebige andere Weise ermitteln. Und das Setzen einer Eigenschaft kann auch direkt in irgendeine Aktion umgemünzt werden. In den meisten Fällen werden hinter einer Eigenschaft aber ein oder mehrere `private`-Datenfelder stehen, um ihren Wert zu speichern und wieder abrufbar zu machen. Eben weil die Datenfelder privat sind und so auch bei sprachübergreifendem Gebrauch der Klasse keine Probleme zu erwarten sind, ist es eine häufige, nicht jedoch zwingende Praxis, Datenfelder und Eigenschaften nur über die Groß- und Kleinschreibung zu unterscheiden.

Codebeispiel – Eigenschaften

Das folgende Codebeispiel zeigt eine einfache Klasse mit vier Eigenschaften, deren Objekte einen Preis mit prozentualem Mehrwertsteueranteil repräsen-

tieren. Geboten sind »Erhalt der Integrität« und »Gültigkeitsprüfung«: Neue Objekte müssen mit einem Mehrwertsteuerfaktor initialisiert werden, der zwischen 0 und 1 liegt. Eine Änderung der `Brutto`-Eigenschaft ändert bei unverändertem Steueranteil auch die Nettoeigenschaft – und umgekehrt. Eine Änderung des Steuerfaktors führt zu einer Änderung der `Brutto`-Eigenschaft bei unveränderter `Netto`-Eigenschaft.

Alle Eigenschaften sind nur für einen positiven Wertebereich definiert. Bei Verletzung einer Gültigkeitsregel reagiert das Objekt mit einer Ausnahme – genau diese Verhalten demonstriert der Code auch, wie Abbildung 8.14 zeigt.

```
class Preis
{
  private decimal mehrwertsteuer;    // als Faktor
  private decimal netto;
  private decimal brutto;
  public Preis(decimal mwst)
  {
    Mehrwertsteuer = mwst;           // Eigenschaft!
  }
  public Preis(double mwst)
  {
    Mehrwertsteuer = (decimal) mwst; // Eigenschaft!
  }

  public decimal Brutto
  {
    set
    {
      if (value < 0)                 // nicht negativ
        throw new Exception("Brutto darf nicht negativ sein");
      else
      {
        brutto = value;
        netto = value / (1 + mehrwertsteuer)  ;
      }
    }
    get
    {
      return brutto;
    }
  }

  public decimal Netto
  {
    set
    {
      if (value < 0)
        throw new Exception("Netto darf nicht negativ sein");
      else
      {
        netto = value;
```

```
        brutto = (1 + mehrwertsteuer) * value ;
      }
    }
    get
    {
      return netto;
    }
  }

  public decimal Mehrwertsteuer
  {
    set
    {
      if (value < 0 || value > 1)     // zwischen 0 und 100%
        throw new Exception(
          "Mehrwertsteuer muss größer 0 und kleiner 1 sein");
      else
      {
        mehrwertsteuer = value;
        brutto = netto * (1 + value) ;  // Brutto neu berechnen
      }
    }
    get
    {
      return mehrwertsteuer;
    }
  }
}

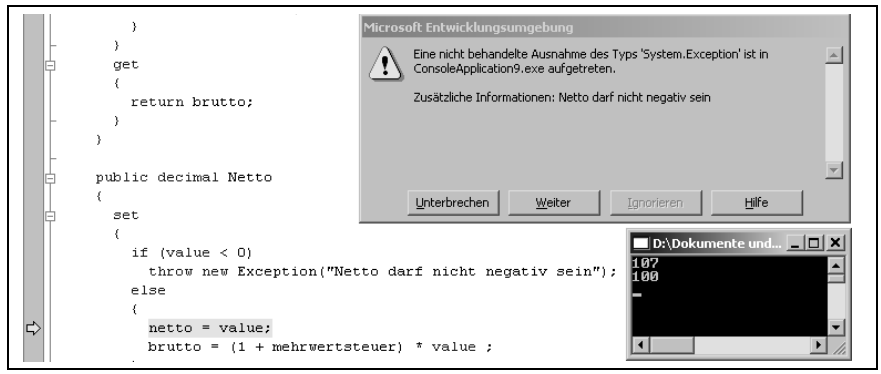
class Class1
{
  [STAThread]
  static void Main(string[] args)
  {
    Preis p1 = new Preis(0.07M);
    Preis p2 = new Preis(0.16);
    p1.Netto = 100;
    Console.WriteLine(p1.Brutto);
    p2.Brutto = 116;
    Console.WriteLine(p2.Netto);
    p1.Netto = -100;     // Fehler, generiert Ausnahme
  }
}
```

Indexer – indizierter Zugriff auf Eigenschaften und Datenfelder

Bei einem Indexer handelt es sich um eine spezialisierte Instanzmethode, die einen indexorientierten Zugriff auf Eigenschaften und Datenfelder eines Objekts implementiert.

Ähnlich wie Eigenschaften zählen Indexer nicht zu den traditionellen Elementen der objektorientierten Programmierung, sondern stellen mehr oder weniger einen syntaktischen Kunstgriff dar, um die Notation für den Zugriff auf verschiedene von einem Objekt repräsentierte Werte (im Allge-

Abbildung 8.14: Ein Lauf des Beispielprogramms



meinen: öffentliche Eigenschaften und Datenfelder) zu vereinfachen und an die Array-Schreibweise anzugleichen.

Der Zugriff auf die Eigenschaften eines mit einem Indexer ausgestatteten Objekts `MyObj` sieht vom Prinzip her so aus:

```
MyObj[i] = NeuerWert;
AndererWert = MyObj[j];
```

Das Objekt wird wie ein Array notiert, und der angegebene Elementindex wählt die dem Indexwert zugeordnete Eigenschaft aus.

Syntax und Verwendung

```
[Modifizierer] Rückgabetyp this[Indizierertyp IndexVar]
{
  [get
   {
     [Anweisungsblock;]
      return Rückgabetyp
   }]
  [set
   {
     [Anweisungsblock;]
   }]

}
```

Der Bezeichner für den Indexer ist festgelegt: Er lautet `this` – was angesichts der Notation für den Zugriff nicht sehr überraschend ist. Da `this` ja immer auf die aktuelle Instanz verweist, lassen sich Indexer nur für Objekte zur Verfügung stellen. Für statische Eigenschaften unterstützt C# (zur Zeit noch) keinen eigenen Indexer. Beachten Sie, dass das eckige Klammerpaar unmittelbar nach `this` Teil der Syntax ist – lesen Sie den geklammerten Ausdruck also nicht als »optional«. Was die `get`- und `set`-Accessoren betrifft, so gilt dasselbe wie für gewöhnliche Eigenschaften.

Dabei ist es der Implementierung absolut freigestellt, welche Datenfelder und Eigenschaften sie in welcher Reihenfolge zurückliefert oder setzt – und gleich, ob statisch oder nicht. Für den `Indizierertyp` gelten von der Syntax her keine Einschränkungen. Im Regelfall wird man hier aber mit Blick auf die anstehende Fallunterscheidung einen Ganzzahltyp oder einen Aufzählungstyp (`enum`) vereinbaren. Eine `ref`- oder `out`-Deklaration für `IndexVar` ist verständlicherweise nicht erlaubt.

Codebeispiel – ein Indexer für die Klasse Preis

Für die Beispielklasse `Preis` aus dem vorangegangen Abschnitt sind drei Eigenschaften des Typs `decimal` definiert. Der folgende Code aus dem Beispielprojekt *Indexer* erweitert die Klasse um einen Indexer für diese Eigenschaften:

```
public decimal this[int i]
{
  get
  {
    switch (i)
    {
      case 0:
        return Mehrwertsteuer;
      case 1:
        return Netto;
      case 2:
        return Brutto;
      default:
        throw new IndexOutOfRangeException ("Index ungültig");
    }
  }
  set
  {
    switch (i)
    {
      case 0:
        Mehrwertsteuer = value; break;
      case 1:
        Netto = value; break;
      case 2:
        Brutto = value; break;
      default:
        throw new IndexOutOfRangeException ("Index ungültig");
    }
  }
}
```

Der Testcode in der `Main`-Methode lässt sich nun wie folgt umformulieren:

```
Preis p1 = new Preis(0.07M);
Preis p2 = new Preis(0.16);
p1[1] = 100;
```

```
Console.WriteLine(p1[2]);
p2[2] = 116;
Console.WriteLine(p2[1]);
```

Ein ungewöhnlicheres Beispiel für einen Indexer, der eine variable Anzahl von implizit definierten Eigenschaften zurückliefert, finden Sie im Abschnitt »Codebeispiel – Primzahlen aufzählen«, Seite 157.

Arrays als Eigenschaften

Dass eine Klasse auch Arrays als Datenfelder vereinbaren kann, versteht sich beinahe von selbst. Was aber, wenn der Wunsch besteht, solche Datenfelder als Eigenschaften zu verpacken? Sieht C# auch syntaktische Mittel für Indizierung von `get`- und `set`-Accessoren sowie `value`-Werten vor? Wohlgemerkt: Es geht darum, dass die Eigenschaft den Zugriff auf die Elemente des Arrays vermittelt – nicht den Zugriff auf das Arrayobjekt. So gestellt, ist die Antwort auf die Frage ein klares *Nein*.

Eine Aggregierung von Eigenschaften zu Arrays ist in C# nicht nur nicht vorgesehen, sie würde auch den Gedanken der objektorientierten Programmierung hintertreiben. Schließlich sind C#-Arrays echte Objekte – nicht nur (wie in C/C++ oder – mit Einschränkungen – in Delphi) schlichte Ansammlungen von Elementen ohne »Eigenintelligenz«, für die letztlich nur der Compiler die Instanz ist, die den Zugriff vermittelt. Kurzum: Es ist Aufgabe des Array-Objekts, den Zugriff auf seine Elemente zu gestalten und zu überwachen.

Der Indexer ermöglicht es einem Objekt, nach außen hin als Array aufzutreten und stellt für C# tatsächlich das Bindeglied dar, das Konzept der Eigenschaft mit dem des Arrays zu verheiraten. Mit dieser Überlegung bahnt sich auch eine Lösung an, um in C# indizierbare Eigenschaften zu erhalten, wenn man einen Indexer zur Verfügung hat. In der Tat fördert ein Blick auf die .NET-Klassenhierarchie eine Fülle von Beispielen für Klassen mit indizierbaren Eigenschaften zutage – etwa die Klasse `ImageList`, deren Instanzen eine indizierbare `Images`-Eigenschaft mit dem Datentyp `ImageCollection` besitzen.

Die Lösung setzt demnach nicht an der Syntax für Eigenschaften, sondern am Datentyp an. Dieser regelt den Elementzugriff, wobei der Indexer die notwendigen `set`- und `get`-Accessoren beisteuert.

Nachdem die Klasse `Preis` im vorangegangenen Beispiel einen Indexer erhalten hat, kann sie als Elementtyp für eine indizierbare Eigenschaft fungieren. Das etwas modifizierte Beispielprojekt *Indexer1* fügt dem Arrangement eine weitere Klasse `WithIndexProperty` hinzu, die eine Eigenschaft mit diesem Typ definiert, sowie etwas Testcode zur Demonstration, dass die Geschichte auch funktioniert:

```
class WithIndexProperty
{
  public Preis preis;
  public WithIndexProperty()
  {
    preis = new Preis(0.15);
  }
}
...
WithIndexProperty wip = new WithIndexProperty();
wip.Preis[2] = 100.45M;    // Brutto setzen
Console.WriteLine("Netto ist: {0:.##}", wip.Preis[1]);  // 87,35
```



Abbildung 8.15: Ausgabe des Programms *Indexer1*

Beachten Sie, dass im vorliegenden Fall auch das Datenfeld indizierbar ist und dieselbe Zugriffskontrolle wie die Eigenschaft bietet.

Eigenschaften, deren Datentyp wie bei einem Datenfeld sofort durch ein eckiges Klammerpaar verrät, dass es sich um ein Array handelt, lassen sich so allerdings nicht realisieren. Das eckige Klammerpaar im Datentyp kennzeichnet in C# aber die Implementation auf der Basis der .NET-Klasse `System.Array` und steht somit für eine feste Semantik, von der eigene Implementierungen aber abweichen werden – sonst wären sie schließlich unnötig.

Vererbung

Aus Sicht der Vererbung sind Indexer nichts anderes als gewöhnliche Methoden. Um den Indexer einer Basisklasse zu überschreiben, ist das Schüsselwort `new` erforderlich; liegt in der Basisklasse eine `virtual`-Vereinbarung vor, das Schlüsselwort `override`. Ein Indexer, der auch Eigenschaften der Basisklasse aufzählt, kann dafür jederzeit den Indexer des Basisklassenobjekts aufrufen:

```
...
  case 3:
  case 4:
  case 5:
  case 6:
    return base[i];
...
```

Indexer in Schnittstellendefinitionen

Nachdem ein Indexer vom Prinzip her nichts anderes als eine gewöhnliche Methode oder Eigenschaft ist, darf er auch als Element einer Schnittstelle definiert werden. Die Syntax dafür deckt sich, bis auf den Parameter in eckigen Klammern, mit der für Eigenschaften:

```
interface MyInterface
{
  ...
  decimal this [int] i {get; set;}
}
```

Eine Klasse, in deren Vererbungsliste diese Schnittstelle erscheint, muss demnach einen Indexer mit der geforderten Signatur implementieren.

8.3.10 Operatoren

Operatoren haben ihren Weg recht früh in die Programmiersprachen gefunden. Sprachen wie Basic, Fortran, Pascal und C unterstützen Operatoren seit jeher als implizites Sprachkonzept, haben jedoch nie großes Aufheben darum gemacht. Möglichkeiten, eigene Operatoren zu definieren, wie sie die Programmiersprache Algol68 schon fast zur akademischen Disziplin hochstilisierte, wurden lange Zeit als unwichtig angesehen und kamen erst mit der objektorientierten Programmierung wieder auf den Wunschzettel – unter anderem wohl, weil die damit verbundenen Konzepte nicht nur schwer zu definieren, sondern eben auch nicht gerade trivial in der Implementierung waren. Eine Grundvoraussetzung bildete beispielsweise das allgemeinere Konzept des Überladens von Funktionen, das erst mit C++ seinen Weg in die allgemeinen Sprachdefinitionen fand und den Weg für die saubere Implementierung abstrakter Datentypen ebnete. Es ist daher nicht verwunderlich, dass C# als in der Tradition von C/C++ stehende Sprache nicht nur über eine reichhaltige Ausstattung an Operatoren verfügt (die Standardoperatoren von C# werden im Abschnitt »Standardoperatoren« ab Seite 191 vorgestellt), sondern auch deren Überladung für benutzerdefinierte Datentypen gestattet.

Operatoren sind spezialisierte Methoden

Vom Prinzip her sind Operatoren relativ stark spezialisierte Methoden mit ein oder zwei Parametern (selten mehr), so genannten *Operanden*. Die Besonderheit von Operatoren besteht darin, dass sie eine besonders intuitive, an die mathematische Schreibweise angelehnte Notation mitbringen und die Lesbarkeit von Ausdrücken stark verbessern.

In der Tat sind wir es gewohnt, Ausdrücke der folgenden Art zu schreiben und zu lesen:

```
int a = b * c + v % 8 / 3;
```

Wäre jede der darin genannten Operationen als Methode (etwa der Klasse `MyClass`) zu schreiben, würde der Ausdruck schon fast monströse Gestalt annehmen – und wäre kaum noch vernünftig lesbar:

```
int a = MyClass.Add(MyClass.Mult(b, c), MyClass.Div(MyClass.Mod(v, 8), 3));
```

Charakteristika

Eine wichtige Eigenschaft von Operatoren ist, dass sie ohne Klammern notiert werden können. Die Charakterisierung eines Operators umfasst daher neben dem Bezeichner und den Datentypen für die Operanden und dem Rückgabewert noch drei weitere Eigenschaften:

- Wertigkeit
- Position
- Priorität

Da in C# keine neuen Operatoren definiert werden können, sondern »nur« die Überladung vordefinierter Operatoren möglich ist, sind diese Eigenschaften inhärent festgelegt und lassen sich nicht frei definieren.

Die *Wertigkeit* macht eine Aussage über die Anzahl der Operanden. Zwar findet sich in der Menge der Standardoperatoren von C# auch ein dreiwertiger Operator (der Operator für die bedingte Auswertung `_?_:_`), da dieser jedoch nicht überladen werden kann, spielt sich die Operatordefinition in C# rein auf der Ebene der unären (ein Operand) und der binären (zwei Operanden) Operatoren ab.

Die *Position* macht eine Aussage darüber, wo ein Operator bezogen auf seine Operanden notiert wird. Für unäre und binäre Operatoren lassen sich dabei drei Positionen unterscheiden.

1. *Infixnotation* – in C# werden binäre Operatoren ausschließlich auf diese Art notiert: *`Operand1 Operator Operand2`*
2. *Präfixnotation* – in C# werden unäre Operatoren in der Regel vorangestellt. Das Schema lautet: *`Operator Operand`*.
3. *Postfixnotation* – es gibt in C# zwei Operatoren, `++` und `--`, für die auch die nachgestellte Notation nach dem Schema *`Operand Operator`* definiert ist.

Die *Priorität* eines Operators spielt eine wichtige Rolle für die Reihenfolge, in der ein Ausdruck interpretiert wird. Der oben genannte Ausdruck wird bekanntlich nach der Regel »Punkt geht vor Strich« ausgewertet, sodass sich die folgende hier durch Klammerung verdeutlichte Auswertungsreihenfolge ergibt.

```
int a = ((b * c) + ((v % 8) / 3));
```

Tabelle 8.1 gibt noch einmal einen Überblick für die in C# verfügbaren Standardoperatoren und ihre Hierarchie. Tabelle 8.2 zeigt, welche davon überladbar sind.

Tabelle 8.1: Operatorprioritäten absteigend geordnet

Operatorart (Priorität absteigend)	Operatoren
Primäroperatoren	., (), [], *x*++, *x*--, new, typeof, checked, unchecked
Unäre Präfix-Operatoren	+, -, !, ~, ++*x*, --*x*, (*DatenTyp*)
Multiplikation, Division, Modulo	*, /, %
Addition, Subtraktion	+, -
Bitweises Verschieben	<<, >>
Vergleich (Wert- und Typ-)	<, >, <=, >=, is, as
Gleichheitsoperatoren	==, !=
Bitweises And	&
Bitweises Xor	^
Bitweises Or	|
Logisches And	&&
Bitweises Or	||
Bedingte Auswertung	? :
Zuweisung und kombinierte Zuweisung	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Einschränkungen für das Überladen von Operatoren

So praktisch der Umgang mit Operatoren ist, wer sich einmal mit ihrer Definition näher beschäftigt hat, der weiß, welche Freiheiten dieses Konzept birgt und ahnt vielleicht auch, welche Komplexität diese Freiheiten für den Compilerbau bedeuten. Um es also noch einmal deutlich zu sagen: C# erlegt dem Programmierer beim Überladen von Operatoren relativ starke Einschränkungen auf und bleibt dabei noch ein gutes Stück hinter C++ zurück – von Algol68 ganz zu schweigen.

Hier ein Überblick über die Einschränkungen:

- Operatoren müssen statisch vereinbart werden.
- Operatoren müssen Überladungen von Standardoperatoren sein.

- Nicht alle Standardoperatoren lassen sich überladen. Insbesondere der Zuweisungsoperator kann nicht überladen werden. Tabelle 8.2 listet die überladbaren Operatoren auf. Obwohl `[]` und `()` nicht zu den überladbaren Operatoren zählen, sind sie quasi-überladbar: Die Bedeutung von `[]` wird durch Definition eines Indexers festgelegt, die Bedeutung des Typumwandlungsoperators `()` durch `implict`- oder `explict`-Überladungen der jeweiligen Datentyp-Operatoren.
- Binäre Standardoperatoren können nur als binäre Operatoren überladen werden, unäre Standardoperatoren nur als unäre.
- Operatoren müssen einen Wert zurückgeben. Ihr Rückgabewert darf also nicht als `void` vereinbart sein.
- Überladene Operatoren behalten die Notation (Infix, Präfix, Postfix) der gleichnamigen Standardoperatoren bei.
- Überladene Operatoren behalten die Prioritäten der gleichnamigen Standardoperatoren bei.
- Bei binären Operatoren muss mindestens ein Operand, bei unären Operatoren der Operand und der Rückgabetyp den Datentyp der Klasse tragen.
- Als Operanden dürfen keine `ref`- oder `out`-Parameter vereinbart werden.
- Die Überladung der Vergleichsoperatoren muss immer paarweise komplementär erfolgen.
- Die Überladung der kombinierten Operatoren (+=, *= usw.) ergibt sich implizit aus der Überladung des Grundoperators (+, * usw.).
- Die Überladung der logischen Operatoren `&&` und `||` ergibt sich implizit aus der Überladung der unären Operatoren `true` und `false` sowie der binären Operatoren `|` und `&` (vgl. Beispiel).

Tabelle 8.2: In C# überladbare Operatoren

Operator	Art	Einschränkungen/Bemerkungen
`+, - *, /, %`	binäre arithmetische Operatoren	Infixnotation; Ein Operand muss die eigene Klasse als Datentyp tragen.
`+, -`	unäre arithmetische Operatoren	Präfixnotation; Operand und Rückgabewert müssen die eigene Klasse als Datentyp tragen.
`++, --`	unäre arithmetische Operatoren	Prä- oder Postfixnotation; Operand und Rückgabewert müssen die eigene Klasse als Datentyp tragen.

Tabelle 8.2: In C# überladbare Operatoren (Forts.)

Operator	Art	Einschränkungen/Bemerkungen
`&`, `|`, `^`, `<<`, `>>`	binäre bitweise Operatoren	Infixnotation; Mindestens ein Operand muss den Datentyp der Klasse tragen.
`!`, `~`	unäre bitweise Operatoren	Präfixnotation; Operand und Rückgabewert müssen die eigene Klasse als Datentyp tragen.
`true`, `false`	unärer logischer Operator	Als solche nicht notierbar, Aufruf nur implizit im Zusammenhang mit der Auswertung anderer implizit definierter Operatoren, Operand muss die eigene Klasse als Datentyp tragen.
`==`, `!=`, `>=`, `<=`, `<`, `>`	binäre Vergleichsoperatoren	Infixnotation; Mindestens ein Operand muss die eigene Klasse als Datentyp tragen. Komplementäre Operatoren müssen paarweise definiert werden.

Syntax

Die Syntax für die Überladung eines binären Operators für eine Klasse *MyClass* lautet:

```
[Modifizierer] static Class1 operator BinärOp (Class2 Op1, Class3 Op2)
{
  Anweisungsblock;
}
```

Hierbei ist gefordert, dass *Class2* und/oder *Class3* mit der Klasse *MyClass* identisch sind, in der der Operator vereinbart wird.

Die Syntax für die Überladung eines unären arithmetischen oder bitweisen Operators für eine Klasse *MyClass* lautet:

```
[Modifizierer] static MyClass operator UnärArithBitOp (MyClass Op)
{
  Anweisungsblock;
}
```

Die Syntax für die Überladung der unären logischen Operatoren `true` und `false` für eine Klasse *MyClass* lautet:

```
[Modifizierer] static bool operator UnärLogOp (MyClass Op)
{
  Anweisungsblock;
}
```

Typumwandlungsoperatoren

Die Syntax für die Überladung eines expliziten oder impliziten Typumwandlungsoperators lautet:

```
public static {implicit | explicit} operator Class1(Class2 Op)
{
  Anweisungsblock;
}
```

Hierbei ist gefordert, dass *Class1* und/oder *Class2* mit der Klasse identisch sind, die den Operator vereinbart. Im Allgemeinen wird dies *Class1* sein.

Typumwandlungsoperatoren zählen gleichfalls zu den unären Operatoren, haben aber eine Sonderstellung, da sie augenscheinlich auch die gleiche Signatur – bei unterschiedlichem Ergebnistyp – aufweisen können. Damit dies syntaktisch ins Bild passt, kann man sich die Typumwandlung so vorstellen, dass hier eine Überladung des Klammer-Operators (Cast-Operator) *samt* Typbezeichner stattfindet.

Ein Typumwandlungsoperator muss entweder als `implicit` oder als `explicit` vereinbart werden.

Hier ein Beispiel für einen Operator, der die explizite Typumwandlung eines Vectors in einen Wert des Typs `double` definiert. Der Operator liefert als Ergebnis die Länge (Betrag) des Vektors:

```
class Point3D
{
 ...
  public static explicit operator double(Point3D p)
  {
    return Math.Sqrt(p.X * p.X + p.Y * p.Y + p.Z * p.Z);
  }
}
```

Ein so definierter Operator kann nun wie jeder andere Typumwandlungsoperator notiert werden:

```
Point3D P3D = new Point3D(3, 4, 0);
double Abs = (double) P3D;      // Abs = 5
```

Die `implicit`-Vereinbarung ist um einiges komplexer. Zum einen umfasst sie die `explicit`-Vereinbarung. Das heißt: Ein für die implizite Typumwandlung vereinbarter Operator kann jederzeit auch für die explizite Typumwandlung nach dem eben gezeigten Muster verwendet werden. Das aber nur nebenbei. In erster Linie gibt sie dem Compiler das Recht, die Typumwandlungsoperation unaufgefordert und nach eigenem Gutdünken einzusetzen, um Ausdrücke sinnvoll aufzulösen.

Wann und wie sich der Compiler für implizite Typumwandlungsoperationen entscheidet, ist in einem Regelsatz festgelegt, der allerdings nicht unbedingt als intuitiv zu bezeichnen ist. Würde der Compiler bei der Auflösung eines Operators kaskadierte Typumwandlungen oder auch nur Umwandlungen anwenden, bei der die eigene Klasse nicht Ausgangs- oder Zieltyp ist, käme er schnell in Teufels Küche. Um sicher zu sein, dass der Compiler die gewünschte Interpretation eines Ausdrucks übernimmt, sollte man in Zweifelsfällen besser explizite Typumwandlungen notieren oder zumindest Klammern setzen.

Für die Anwendung der impliziten Typumwandlung im Zusammenhang mit der Auflösung von Operatoren gelten folgende Regeln:

- Zur Auflösung von unären Operatoren wendet der Compiler keine implizite Typumwandlung auf den Operanden an.
- Bei binären Operatoren greift der Compiler nur dann zum Mittel der impliziten Typumwandlung, wenn *ein* Operand in einem Datentyp vorliegt, für den keine passende Überladung des Operators definiert ist und eine Auflösung nach Anwendung der impliziten Typumwandlung in die *eigene* Klasse möglich ist.
- Die implizite Typumwandlung eines Operanden geschieht nach Möglichkeit in Richtung des komplexeren Datentyps. Falls dafür keine implizite Typumwandlung definiert ist, nimmt der Compiler die andere Richtung.

Das folgende Codebeispiel zeigt eine Klasse `Vektor` mit je einem impliziten Typumwandlungsoperator für die Umwandlung nach und von `double`.

```
class Vektor
{
  public double X, Y, Z;
  public Vektor(double x, double y, double z)
  {
    X = x;
    Y = y;
    Z = z;
  }
  public static implicit operator Vektor(double d)
  {
    Trace.Msg("double nach Vektor");
    return new Vektor(d, 0, 0);
  }
  public static implicit operator double(Vektor p)
  {
    Trace.Msg("Vektor nach double");
    return Math.Sqrt(p.X * p.X + p.Y * p.Y + p.Z * p.Z);
  }
}
```

Betrachtet man jetzt den Aufruf

```
double d = 1;
Vektor v = new Vektor(1, 1, 1);
double d1 = d + v;   // Vektor nach double, double.+
Vektor v1 = v + d;   // Vektor nach double, double.+, double nach Vektor
```

Dann ist klar, dass die Klasse `Vektor` keinen `operator +` überlädt. Die Frage lautet nun, was passiert, wenn ein solcher definiert wird. Also:

```
class Vektor
{
 ...
  public static Vektor operator +(Vektor v1, Vektor v2)
  {
    Trace.Msg("Vektor.+");
    return new Vektor(v1.X + v2.X, v1.Y + v2.Y, v1.Z + v2.Z);
  }
}
```

Der gleiche Aufruf führt nun zu einem völlig anderen Ergebnis:

```
double d = 1;
Vektor v = new Vektor(1, 1, 1);
double d1 = d + v;   // double nach Vektor, Vektor.+, Vektor nach double
Vektor v1 = v + d;   // double nach Vektor, operator +
```

Codebeispiel – Implementierung einer Klasse Point3D

Das folgende, etwas umfangreichere Beispielprojekt *OperatorExample* zeigt die Implementierung verschiedener Operatoren für eine Vektorklasse namens `Point3D`. Um die Auswertungsreihenfolge von Ausdrücken, in denen diese Operatoren vorkommen, verfolgbar zu machen, bedient sich der Code der in der Klasse `Trace` der durch ein `Conditional`-Attribut bedingt vereinbarten Methode `Msg`. Die Ablaufverfolgung wird durch Definition des Symbols `TRACEMODE` eingeschaltet.

```
#define TRACEMODE   // Ablaufverfolgung aktivieren
using System;
using System.Diagnostics;

namespace OperatorExample
{
  class Trace
  {
    [Conditional ("TRACEMODE")]
    public static void Msg(string msg)
    {
      Console.WriteLine(msg);
    }
  }
```

```
class Point3D
{
  protected double X, Y, Z;

  //Initialisierungskonstruktor
  public Point3D(double x, double y, double z)
  {
    X = x;
    Y = y;
    Z = z;
  }

  // Vektorvorzeichen: -Point3D
  public static Point3D operator -(Point3D p)  // Unär
  {

    Trace.Msg("Vorzeichen -");
    return new Point3D(-p.X, -p.Y, -p.Z);
  }

  // Vektoraddition: Point3D + Point3D
  public static Point3D operator +(Point3D p1, Point3D p2)
  {
    Console.WriteLine("+");
    return new Point3D(p1.X + p2.X, p1.Y + p2.Y, p1.Z + p2.Z);
  }

  // Vektorsubtraktion: Point3D - Point3D
  public static Point3D operator -(Point3D p1, Point3D p2)
  {
    Trace.Msg("Minus -");
    return -p2 + p1;
  }

  // Scalares Produkt: double * Point3D
  public static double operator *(double d, Point3D p)
  {
    Trace.Msg("Skalares Produkt *");
    return new Point3D(d * p.X, d * p.Y, d * p.Z);
  }

  // Scalares Produkt: Point3D * double
  public static double operator *(Point3D p, double d)
  {
    Trace.Msg("Skalares Produkt *");
    return new Point3D(d * p.X, d * p.Y, d * p.Z);
  }

  // Scalarprodukt: Point3D * Point3D
  public static double operator *(Point3D p1, Point3D p2)
  {

    Trace.Msg("Skalarprodukt *");
    return p1.X * p2.X + p1.Y * p2.Y + p1.Z * p2.Z;
  }
```

```
    //Vektorprodukt: Point3D % Point3D
    public static Point3D operator %(Point3D p1, Point3D p2)
    {
      Trace.Msg("Vektorprodukt %");
      return new Point3D(p1.Y*p2.Z - p1.Z*p2.Y,
          p1.Z * p2.X - p1.X*p2.Z, p1.X * p2.Y - p1.Y*p2.X);
    }

    //Vergleich: Point3D < Point3D
    public static bool operator <(Point3D p1, Point3D p2)
    {
      Trace.Msg("Vergleich <");
      return (double) p1 < (double) p2;   // implizit Betrag !!!
    }

    //Vergleich: Point3D > Point3D
    public static bool operator >(Point3D p1, Point3D p2)
    {

      Trace.Msg("Vergleich <");
      return p2 < p1;
    }


    public override string ToString()
    {
      return "(" + X + "," + Y + "," + Z + ")";
    }

    // Vektorbetrag: (double) Point3D
    public static implicit operator double(Point3D p)
    {
      Trace.Msg("Vektorbetrag (double)");
      return Math.Sqrt(p.X * p.X + p.Y * p.Y + p.Z * p.Z);
    }
  }

  class TestProgramm
  {
    private static void Main()
    {
      double p ;
      Point3D p1 = new Point3D(1, 0, 1);
      Point3D p2 = new Point3D(0, 1, 0);
      Console.WriteLine("p1 = " + p1);      // implizit ToString()
      Console.WriteLine("p2 = " + p2);      // implizit ToString()
      Console.WriteLine("Auswertungsreihenfolge von p2 + p1 - p2");
      Console.WriteLine("p2 + p1 - p2 = " + (p2 - p1 + p2));
      Console.WriteLine();

      Console.WriteLine("Auswertungsreihenfolge von p2 - p1 % p2");
      Console.WriteLine("p2 - p1 % p2 = " + (p2 - p1 % p2));
      Console.WriteLine();

      Console.WriteLine("Auswertungsreihenfolge von p2 - p1 * p2");
```

```
      // implizit: Betrag !!
      Console.WriteLine("p2 - p1 * p2 = " + (p2 - p1 * p2));
      Console.WriteLine();

      Console.WriteLine("Vektorprodukt p1 % p2 = " + (p1 % p2));
      Console.WriteLine();
      p1 = -p1;            // Vorzeichen
      Console.WriteLine("Ändere Vorzeichen p1 = -p1: p1 = " + p1);
      Console.WriteLine();
      Console.WriteLine("|p1| = " + (double) p1);
      Console.WriteLine();

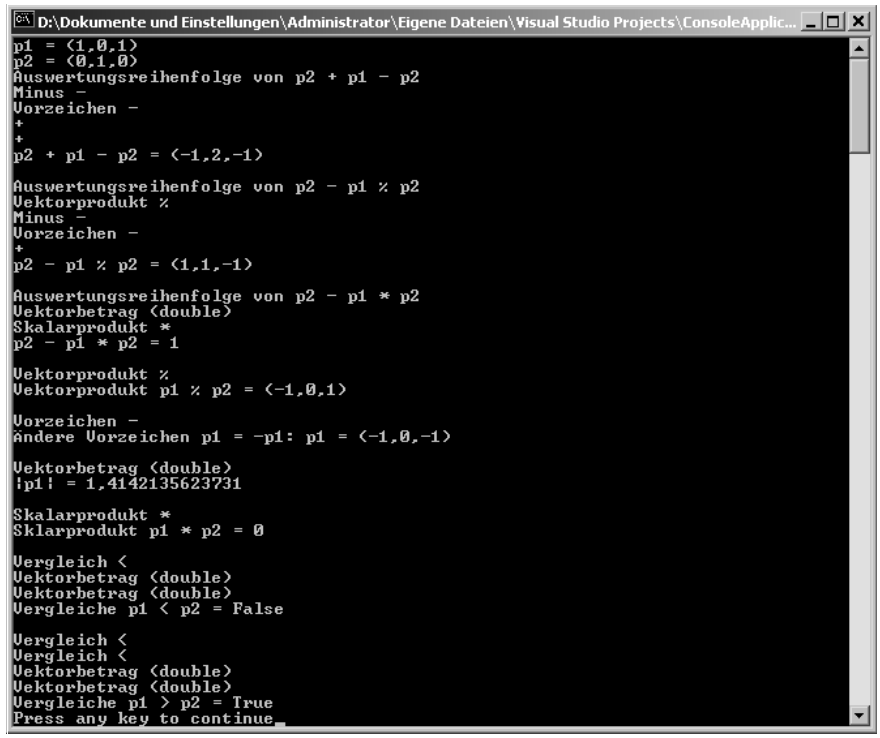
      Console.WriteLine("Sklarprodukt p1 * p2 = " + p1 * p2);
      Console.WriteLine();

      Console.WriteLine("Vergleiche p1 < p2 = " + (p1 < p2));
      Console.WriteLine();

      Console.WriteLine("Vergleiche p1 > p2 = " + (p1 > p2));
    }
  }
}
```

Abbildung 8.16 zeigt die Ausgabe des Programms.

Abbildung 8.16: Ausgabe des Testprogramms



Der Code bedarf natürlich einer gewissen Erläuterung. Während die unäre Operation `operator -()` und die binäre Operation `operator +()` völlig geradlinig implementiert sind, wurde die binäre Operation `operator -` auf Basis dieser beiden Operatoren implementiert. Von dem etwas schlechteren Laufzeitverhalten einmal abgesehen (für den zusätzlichen Aufruf muss ein weiteres Objekt konstruiert werden) ist es völlig legitim, für die Implementierung eines Operators auf andere Operatoren zurückzugreifen, so lange dabei keine Endlosrekursion entsteht. Der Code für die Implementierung der Vergleichsoperatoren > und < weist eine ähnliche Struktur auf.

Auch das *Skalarprodukt* ist als `operator *` geradlinig implementiert. Man beachte aber, dass die Operation den Datentyp `double` zurückliefert, während die überladenen Varianten des *skalaren Produkts* ein Objekt der Klasse selbst liefern. Um auch noch das *Vektorprodukt* als Operator unterzubringen, muss die Implementierung auf einen anderen Operator ausweichen, da die Operanden des Vektorprodukts vom Typ her mit denen des Skalarprodukts übereinstimmen. Die Klasse implementiert das Vektorprodukt somit kurzerhand via `operator %`. Die Semantik der Implementierung kann wohl nicht mehr als sinngemäße Erweiterung der Modulooperation bezeichnet werden, sodass der Operator in der Klasse `Point3D` eine völlig neue Bedeutung erhält. Die Designentscheidung hätte natürlich jederzeit auch so ausfallen können, dass `operator *` das Vektorprodukt und `operator %` das Skalarprodukt bereitstellen.

Die Betragsfunktion ist als implizite Typumwandlung in den Datentyp `double` implementiert. Dieses aus mathematischer Sicht an sich sinnvolle Design könnte allerdings zu unbemerkten Flüchtigkeitsfehlern bei der Programmierung führen, sodass an dieser Stelle durchaus die Überlegung angebracht wäre, ob man hier der Sicherheit halber nicht besser doch eine `explicit`-Vereinbarung wählen sollte – oder gleich eine Methode.

8.4 Vererbung

Die Vererbung ist nicht nur das komplexeste Konzept aus der Welt der objektorientierten Programmierung, sie ist zweifellos auch das faszinierendste. Als eine der zentralen Säulen – wenn nicht gar als *die* zentrale Säule – hat sie einen guten Teil zum Siegeszug der objektorientierten Programmierung beigetragen. Weitere wichtige Säulen sind:

- *Überladen von Methoden* – der Überlademechanismus ist an sich nicht an die Vererbung gebunden, fügt sich aber nahtlos in das Konzept ein und bereichert es.
- *Überschreiben von Klassenelementen* – dieser Mechanismus macht *nur* im Zusammenspiel mit der Vererbung Sinn. Er löst zum einen das Problem der Doppelbenennung von Klassenelementen über die Verer-

bungsgrenzen hinweg und beschert der Vererbung zum anderen das Konzept der VIRTUALITÄT – bzw. den Schlüssel zur POLYMORPHIE. (Beide Begriffe werden im Verlauf der Diskussion noch genauer ausgeleuchtet.)

Vererbung steht für die Wiederverwendung von Code. Vererbung ist allerdings nicht gleich Vererbung. Tatsächlich kocht hier jede objektorientierte Programmiersprache ihr eigenes Süppchen und trägt das ihre zum »großen Ganzen« bei. Von höhersprachlichen Veteranen der objektorientierten Programmierung wie LISP und Smalltalk einmal abgesehen, die jenseits akademischer Sphären kaum praktische Bedeutung erlangt haben, war es die Sprache C++, die die Vererbung – oder das, was heute darunter verstanden wird – am deutlichsten geprägt hat. In der Sprache C++ ist alles angelegt, was in der objektorientierten Programmierung Rang und Namen hat. *Schnittstellenvererbung*, *Implementierungsvererbung, Mehrfachvererbung* und *Polymorphismus*. An sich bietet C++ damit die idealen Voraussetzungen für die komponentenorientierte Programmierung, wäre da nicht ein kleiner Schönheitsfehler: Für C++ wurde nie ein binärer Standard verabschiedet, der das Miteinander von Schnittstellen, Basisklassen, Klassen und Objekten regeln würde. (Bei Java wurde dieser Fehler nicht noch einmal gemacht.) Das hatte die unangenehme Folge, dass sich noch nicht einmal die gängigen C++-Compiler – am prominentesten Borland C++ und Visual C++ – gegenseitig verstanden. Was nützten daher die prächtigsten Klassenbibliotheken, wenn noch nicht einmal innersprachlich so etwas wie Wiederverwendbarkeit gegeben war.

Obwohl die Implementierungsvererbung und ihre Vorteile längst bekannt waren, sollte die einfachere Schnittstellenvererbung in Form des COM als sprachunabhängige »Huckepacklösung« erst einmal das Rennen machen. Der Objektbegriff von .NET, den C# Eins zu Eins widerspiegelt, stellt die zweite Stufe und gleichzeitig den zweiten Anlauf der praktischen Umsetzung dar: die einfach vererbende Implementierungsvererbung. Die dritte Stufe, die mehrfach vererbende sprachunabhängige Implementierungsvererbung, steht noch aus.

8.4.1 Arten der Vererbung

Wie bereits in der Einleitung angeklungen, unterscheidet man eine Reihe von Spielarten für die Vererbung. Das Grundprinzip der Vererbung ist, dass eine Klasse, die so genannte *Ableitung*, auf der Grundlage einer *Basis* definiert wird. (Es ist auch möglich, dass eine Schnittstelle auf der Basis anderer Schnittstellen definiert wird, dieser Fall ist aber seltener und wird in Abschnitt 8.5.2 diskutiert.) Die Basis, entweder eine Schnittstelle oder eine Klasse, wirkt dabei wie eine Schablone und vererbt der Ableitung ihre Elemente. Sie kann ihrerseits wiederum von einer Basis abgeleitet sein. Eine Klasse kann Basisklasse beliebig vieler abgeleiteter Klassen sein, sodass die

Vererbung eine Hierarchie impliziert. Die Spitze dieser Hierarchie ist im Objektmodell vordefiniert – bei C# in Form der Klasse `object`.

Jede C#-Klasse erbt die Ausstattung *einer* Basisklasse. Ist keine Basisklasse angegeben, fungiert standardmäßig `System.Object` als Basisklasse. Darüber hinaus kann eine Klasse optional eine oder mehrere Schnittstellen implementieren, die dann im Vererbungsteil zusätzlich aufgelistet werden:

```
{[abstract] class | interface} Ableitung [: Klasse [, Schnittstelle, ...]]
{ ... }
```

Zur ererbten Grundausstattung einer Klasse zählen alle Elemente der Basisklasse – Datentypen, Konstanten, Datenfelder und Methoden (auch die Konstruktoren). Elemente, die nicht als `private` vereinbart sind, können von der abgeleiteten Klasse so benutzt werden, als seien sie Teil ihrer eigenen Definition. Zudem dürfen sie durch gleichnamige Klassenelemente ersetzt werden – ein Vorgang, der als ÜBERSCHREIBEN bezeichnet wird. Die Sprache unterscheidet hier zwischen der *polymorphen* und der *nicht polymorphen* Implementierung.

Abbildung 8.17 zeigt, wie die Vererbung aus der Sicht von C# organisiert ist.

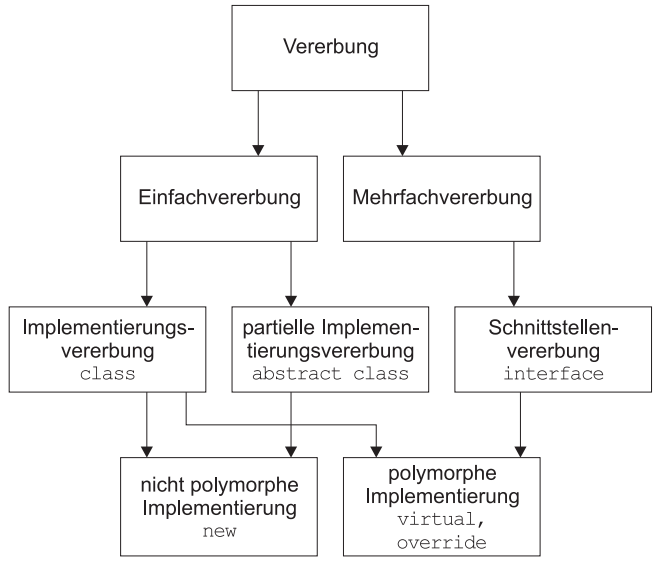


Abbildung 8.17: Vererbungszusammenhänge für C#

Einfachvererbung

Bei der EINFACHVERERBUNG kann die abgeleitete Klasse maximal von einer Basisklasse abgeleitet sein. Die Implementierungsvererbung von C# (bzw. .NET) ist auf Einfachvererbung beschränkt.

Mehrfachvererbung

Bei der MEHRFACHVERERBUNG wird eine Klasse von mehreren Basisklassen und/oder Schnittstellen abgeleitet (vgl. C++). C# erlaubt die Mehrfachvererbung nur bei Schnittstellenvererbung. Eine C#-Klasse kann von maximal einer Basisklasse die Implementierung erben, zusätzlich aber von beliebig vielen Schnittstellen abgeleitet sein.

Schnittstellenvererbung

Für die Schnittstellenvererbung muss die Basis als Schnittstelle (`interface`) definiert sein. »Vererbt« wird in diesem Fall nur eine Struktur bestehend aus einer Menge von Methodendeklarationen. Die abgeleitete Klasse muss für jede dieser Methoden eine Definition – mithin also eine Implementierung – bereitstellen. Da sie keinerlei Code, sondern nur eine Funktionszeigertabelle für die Einsprungadressen der Methoden erbt, ist die mehrfache Schnittstellenvererbung völlig unproblematisch und wird von C# folglich in Ergänzung zur einfach vererbenden Implementierungsvererbung unterstützt.

```
interface IInterface { ... }        // Schnittstellendeklaration

// MyClass2 muss die Elemente der Schnittstelle IInterface implementieren.
class MyClass2 : IInterface { ... }
```

Implementierungsvererbung

Bei der IMPLEMENTIERUNGSVERERBUNG muss die Basisklasse als `class`-Klasse definiert sein, `struct`-Klassen stehen dafür nicht zur Verfügung. Die abgeleitete Klasse erbt dann alle nicht als `private` vereinbarten Klassenelemente der Basisklasse samt der zugehörigen Codeanteile. Während C++ die Implementierungsvererbung gleichfalls als Mehrfachvererbung zulässt, beschränkt sich C# hier auf die Einfachvererbung. Sie darf aber mit der Schnittstellenvererbung kombiniert werden.

Der folgende Codeauszug zeigt die reine und die gemischte Implementierungsvererbung. Die beiden abgeleiteten Klassen erben jeweils den Code der Eigenschaft.

```
class BaseClass                     // Klassendefinition
{
  protected int Counter
  {
    get { ... }
    set { ... }
  }
}

interface IInterface1 { ... }       // Schnittstellendeklaration

interface IInterface2 { ... }       // Schnittstellendeklaration
```

```
// MyClass1 erbt Counter samt Implementierung
class MyClass1 : Baseclass { ... }

// MyClass2 erbt Counter samt Implementierung, muss aber seinerseits
// die Elemente beider Schnittstellen implementieren.
class MyClass2 : Baseclass, IInterface1, IInterface2 { ... }
```

Partielle Implementierungsvererbung

Diese Vererbungsart hat das Ziel, eine bessere Strukturierung von Klassenhierarchien zu ermöglichen. Sie ist vom Prinzip her eine Chimäre zwischen der Implementierungsvererbung und der Schnittstellenvererbung, sollte aber als Sonderfall der Implementierungsvererbung begriffen werden – nicht zuletzt, weil C# dafür die Einfachvererbung vorschreibt. Die Basisklasse ist in diesem Fall eine `abstract`-Klasse. Sie kann ähnlich wie eine Schnittstelle `abstract`-Methoden deklarieren, für die dann die abgeleitete Klasse die Implementierung bereitstellen muss. Zudem hat sie aber im Gegensatz zur Schnittstelle auch die Möglichkeit, wie eine gewöhnliche Klasse Konstanten, Datentypen sowie (statische und nicht statische) Datenfelder und Methoden zu definieren und weiterzuvererben.

```
abstract class MyAbstractBase
{
  public int DataField;              // Datenfeld wird vererbt
  public abstract int Method() {}    // Muss von Ableitung definiert werden
  public int Method(int i)           // Implementierung wird vererbt
  {
    return
  }
}
```

Weitere Details zum Thema abstrakte Klassen finden sich im Abschnitt »Abstrakte Klassen«, ab Seite 331.

Polymorphe Implementierung

Die Möglichkeit der polymorphen Implementierung zählt zu den absoluten Highlights der objektorientierten Programmierung und verleiht insbesondere der formularbasierten Windows-Programmierung unter .NET ihr charakteristisches Gesicht.

Lexikalisch bedeutet der Begriff Polymorphie, dass ein und dasselbe »Ding« in verschiedener Gestalt auftreten kann. Auf die objektorientierte Programmierung gemünzt, ermöglicht die polymorphe Implementierung, dass eine Objektvariable, die den Typ einer Basis trägt, Objekte davon abgeleiteter Klassen so handhaben kann, als trage sie deren Typ. Es ist daher möglich, einer solchen Variable Objekte unterschiedlicher Klassen zuzuordnen – solange diese von der Basis abstammen – und auch deren eigene Methoden aufzurufen, soweit sie eigene Varianten der so genannten

virtuellen Methoden der Basisklasse implementieren. Und das sozusagen von der Basisklasse aus.

Das folgende Lehrbuchbeispiel skizziert die Verhältnisse für eine Basisklasse `Base` und drei davon abgeleitete Klassen `A`, `B`, `C`.

```
class A
{
  public virtual void VirtM()
  {
    Console.WriteLine("Variante der Klasse A");
  }
}
class B : A
{
  public override void VirtM()
  {
    Console.WriteLine("Variante der Klasse B");
  }
}
class C : A
{
  public override void VirtM()
  {
    Console.WriteLine("Variante der Klasse C");
  }
}
class D: A
{
}
```

Die Basisklasse definiert eine virtuelle Methode, die zwei der abgeleiteten Klassen mit je individuellen Varianten überschreiben. Eine mit dem Typ der Basisklasse vereinbarte Objektvariable demonstriert den Effekt:

```
A b;                    // Objektvariable hat Typ der Basisklasse
b = new A();
b. VirtM ();            // Ausgabe: "Variante der Klasse A"
b = new B();
b. VirtM ();            // Ausgabe: "Variante der Klasse B"
b = new C();
b. VirtM ();            // Ausgabe: "Variante der Klasse C"
b = new D();
b.Virtual();            // Ausgabe: "Variante der Klasse A"
```

Das Großartige daran ist, dass der – aus Sicht der Objektvariablen – gleiche Aufruf zu drei unterschiedlichen Ergebnissen führt. Einzig die Klasse `D`, die keine eigene Variante von `VirtM()` definiert, tanzt aus der Reihe: Sie erbt die Variante der Basisklasse `A`. Abbildung 8.18 verdeutlicht den Zusammenhang grafisch.

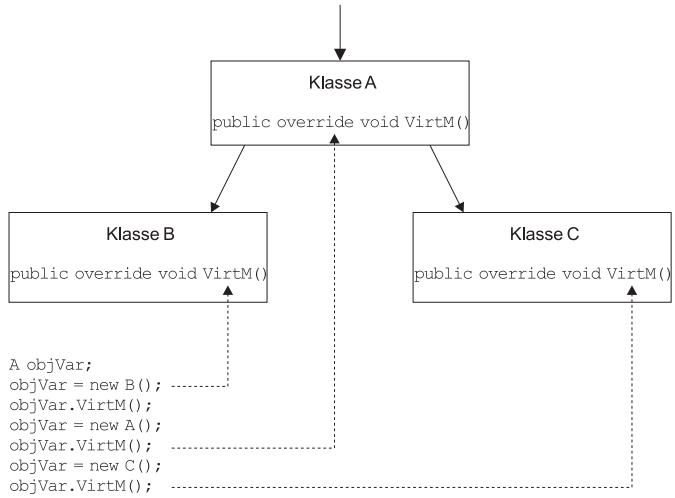


Abbildung 8.18: Polymorphe Implementierung heißt, dass eine Objektvariable immer die virtuelle Methode des tatsächlich gebundenen Objekts aufruft.

Das ganze Spiel geht natürlich über beliebig viele Vererbungsstufen hinweg, wobei auch Schnittstellen und/oder abstrakte Klassen als Basis auftreten können. Die im Zusammenhang mit Schnittstellen mögliche Mehrfachvererbung tut der Sache keinen Abbruch, da die Vererbungslinie von einer bestimmten Basis zur Klasse des jeweils gebundenen Objekts durch weitere Basen nicht beeinflusst wird. Abbildung 8.19 schematisiert die polymorphe Implementierung für eine mehrstufige Vererbungslinie.

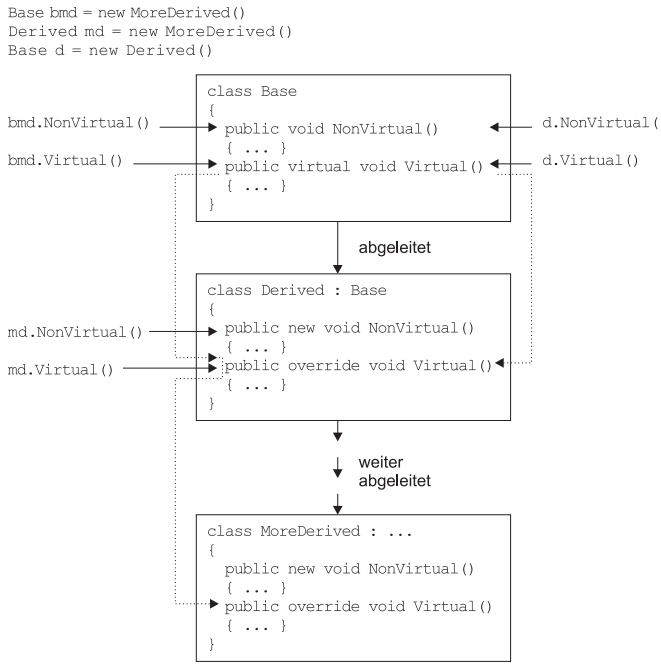


Abbildung 8.19: Polymorphismus entlang einer Vererbungslinie

8.5 Schnittstellen

Dem Konzept der Schnittstellen haftet etwas Geheimnisvolles an, nicht zuletzt aufgrund seiner begrifflichen Assoziation mit COM, dessen (auch heute nur einem kleinen Kreis erlesener Spezialisten zugänglicher) Kernmechanismus bekanntlich auf Schnittstellen beruht.

8.5.1 COM-Schnittstellen

Wie bereits an anderer Stelle ausgeführt, stellt COM das erste wirklich tragfähige Modell für die komponentenorientierte Programmierung unter Windows dar, das (mit geringen Erweiterungen zu DCOM) auch für systemübergreifende Lösungen geeignet ist, ohne auf eine spezielle Programmiersprache gemünzt zu sein. Der Kerngedanke des COM besteht darin, dass ein COM-Objekt Schnittstellen (darunter auch ganz spezielle wie `IUnknown` und `IDispatch`) implementiert und seine Dienste über diese Schnittstellen sprachunabhängig sowohl offen legt, als auch kommuniziert.

Charakteristisch für COM-Schnittstellen ist, dass sie

- auf spezifisch für COM definierte Datentypen beschränkt sind,
- mit so genannten GUIDs, global eindeutigen Bezeichnern, assoziiert sind,
- auf eine zentrale Registrierung angewiesen sind,
- nach ihrer Veröffentlichung ein für alle mal unveränderlich sind,
- eine definierte (und gleichfalls unveränderliche) Repräsentation als Tabelle mit Funktionszeiger haben.

Mehr über das Thema COM-Schnittstellen und deren Einbindung in .NET-Anwendungen finden Sie in Teil 5.

8.5.2 C#-Schnittstellen

Das Schnittstellenkonzept von C# deckt sich nur insofern mit dem von COM, als sich C#-Schnittstellen durch Beachtung bestimmter Einschränkungen und zusätzlicher Formalismen in COM-Schnittstellen verwandeln lassen (vgl. Teil 5). Prinzipiell sind Schnittstellen in C# um einiges allgemeiner gehalten und an die Gegebenheiten der Sprache bzw. des CTS angepasst.

In der in der Tradition von C/C++ stehenden objektorientierten Programmierung werden Schnittstellen als *abstrakte* Klassen deklariert, die nichts weiter als abstrakte Methodendeklarationen enthalten und keinerlei Implementierung bereitstellen – reine »Papiertiger« also, wenn man so will.

In C# muss man hingegen zwischen abstrakten Klassen und Schnittstellen unterscheiden, da diese beiden Begriffe hier für unterschiedliche Konzepte stehen.

- *Abstrakte Klassen* werden in C# mit dem Modifizierer `abstract` vereinbart. Sie können – im Gegensatz zu Schnittstellenklassen – auch Teilimplementierungen enthalten und sind, wie andere C#-Klassen, auf die Einfachvererbung beschränkt. (Detailliertere Informationen über abstrakte Klassen finden Sie im Abschnitt »Abstrakte Klassen«, Seite 331)
- *Schnittstellenklassen* werden in C# mit dem Modifizierer `interface` gekennzeichnet und sind reine Deklarationen – also formale Gebilde ohne Implementierung. Für Schnittstellenklassen besteht generell die Möglichkeit der Mehrfachvererbung.

Mit Schnittstellenklassen (aber auch abstrakten Klassen) allein lässt sich noch nicht viel anfangen. Ihre Bestimmung ist es, Basisklassen für andere Klassen zu sein, die sie *implementieren* (oder im Falle von abstrakten Klassen: komplettieren).

8.5.3 Syntax

Die Syntax für die Vereinbarung einer C#-Schnittstelle lautet:

```
 [Modifizierer] interface IMyInterface [:Interface1 [, ...]}]
{
  [dynamische Methodenprototypen]
}
```

Eine Schnittstellenklasse kann von einer oder mehreren anderen Schnittstellenklassen abgeleitet sein, muss aber nicht. Findet die Vereinbarung auf der Ebene des Namensbereiches statt, sind nur die Modifizierer `public` und `internal` erlaubt (und sinnvoll). Bei Vereinbarung innerhalb einer Klasse sind zusätzlich auch die Modifizierer `private`, `protected` sowie `new` zulässig. Die Elemente der Schnittstellenklasse sind Prototypen für Instanzmethoden und dürfen keine Zugriffsmodifizierer erhalten – der Compiler vereinbart sie implizit als `public virtual`. Statische Elemente sind nicht erlaubt, ebensowenig wie verschachtelte Deklarationen.

Für C# gilt die Konvention, dass die Bezeichner von Schnittstellenklassen mit dem Zeichen »I« beginnen. (Der Compiler setzt diese Konvention nicht durch – er akzeptiert auch an dieser Stelle beliebige Bezeichner.)

Eine Klasse, die eine Schnittstelle implementiert, erfüllt eine Art Vertrag: Sie muss alle im Rahmen der Schnittstelle spezifizierten Dienste als Methoden mit dem geforderten Prototyp (spezifikationsgerecht) implementieren.



Codebeispiel zum Gebrauch der Syntax

Das folgende bewusst einfach gehaltene Beispielprojekt *Schnittstellen* demonstriert die formale Seite des Umgangs mit Schnittstellenklassen. Der Code definiert drei Schnittstellenklassen `MyInterface1`, `IMyInterface2` und `IMyInterface`, wobei die ersten beiden als Basis in der Vererbungsliste der dritten aufgeführt sind. Zudem definiert er eine Klasse `MyClass`, die alle drei (!) Schnittstellen implementiert.

```
interface MyInterface1       // Präfix I ist nicht unbedingt erforderlich
{
  string Name {get;}         // Eigenschaft mit Lesezugriff
  string NameRW {get; set;}  // Eigenschaft mit Vollzugriff
}
interface IMyInterface2
{
  string Name1(bool b);       // Methode
}
interface IMyInterface : MyInterface1, IMyInterface2
{
  new string Name();          // Überschreibung
}
public class MyClass : IMyInterface  // Klasse implementiert Schnittstellen
{
  public virtual string Name()   // IMyInterface.Name
  {
    return "IMyInterface.Name";
  }
  public virtual string Name1(bool b)
  {
    return "IMyInterface2.Name1";
  }
  string MyInterface1.Name  // lässt sich nicht anders schreiben ...
    // muss trotz Überschreibung implementiert werden(!)
  {
    get
    {
      return "MyInterface1.Name";
    }
  }
  public virtual string NameRW   // MyInterface1.NameRW
  {
    get
    {
      return "MyInterface1.NameRW";
    }
    set                          // nur der Form halber
    {
    }
  }
}
```

Als erstes wäre hier zu bemerken, dass `MyClass` keine andere Wahl hat, als alle drei Schnittstellenklassen zu implementieren, obwohl die Klasse selbst nur die Schnittstellenklasse `IMyInterface` als Basis benennt. In der Praxis wird es nur selten wirklich erforderlich sein, Schnittstellenklassen per Vererbung zu bündeln. Da eine Klasse beliebig viele Schnittstellenklassen implementieren darf (aber nur die Implementierung von einer abstrakten oder konkreten Klasse erben kann), wäre die folgende Schreibweise zu der genannten äquivalent – unabhängig davon, in welchem Verhältnis die Schnittstellenklassen zueinander stehen:

```
public class MyClass : IMyInterface, MyInterface1, IMyInterface2
...
```

Der Modifizierer `new` in der Deklaration von `IMyInterface` ist obligatorisch (der Compiler meldet sonst eine Warnung, wenn eine Schnittstelle einen Bezeichner »überschreibt«. Man beachte, dass die Schnittstellenvererbung keine Überschreibung im üblichen Sinne kennt. Überschriebene Methoden müssen in allen Varianten implementiert werden, da kennt der Compiler keine Gnade. Die Qualifizierung einer im Rahmen der Schnittstellenvererbung überschriebenen Methode erfordert, wie sollte es auch anders sein, das Voranstellen des Schnittstellenbezeichners:

```
string MyInterface1.Name  // lässt sich nicht anders schreiben ...
 // muss trotz Überschreibung implementiert werden(!)
```

Ein genauer Blick auf diese Vereinbarung enthüllt aber noch eine weitere Besonderheit: Es fehlen die ansonsten für die Implementierung von Schnittstellenmethoden erforderlichen Modifizierer `public virtual`. Demnach lässt sich eine Schnittstellenmethode in der implementierenden Klasse auf zwei äquivalente Arten schreiben:

```
public virtual string NameRW { ... }
```

und alternativ eben:

```
string MyInterface1.NameRW { ... }
```

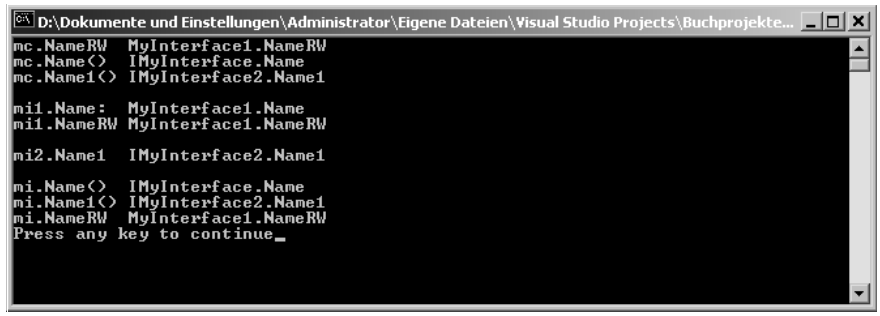
Die erste Form empfiehlt sich, wenn eine Klasse nur eine Schnittstelle implementiert oder zumindest die Bezeichner eindeutig sind. Die zweite Form sollte hingegen verwendet werden, wenn eine Klasse mehrere Schnittstellen implementiert.

Testcode

Die folgende `Main()`-Funktion testet die Klasse und ihre Schnittstellen (Abbildung 8.20):

```
static void Main(string[] args)
{
  MyClass mc = new MyClass(); // Objekt
  Console.WriteLine("mc.NameRW  {0}", mc.NameRW);
  Console.WriteLine("mc.Name()  {0}", mc.Name());
  Console.WriteLine("mc.Name1() {0}", mc.Name1(true));
  Console.WriteLine();
  MyInterface1 mi1 = mc;                  // Schnittstelleninstanz
  IMyInterface2 mi2 = mc;                 // Schnittstelleninstanz
  IMyInterface mi = mc;                   // Schnittstelleninstanz
  Console.WriteLine("mi1.Name:  {0}", mi1.Name);
  Console.WriteLine("mi1.NameRW {0}", mi1.NameRW);
  Console.WriteLine();
  Console.WriteLine("mi2.Name1  {0}", mi2.Name1(true));
  Console.WriteLine();
  Console.WriteLine("mi.Name()  {0}", mi.Name());
  Console.WriteLine("mi.Name1() {0}", mi.Name1(true));
  Console.WriteLine("mi.NameRW  {0}", mi.NameRW);
}
```

Abbildung 8.20: Ausgabe des Programms *Schnittstellen*



8.5.4 Umgang mit .NET-Schnittstellen

In der umfangreichen .NET-Klassenbibliothek sind eine ganze Reihe von Schnittstellen vordefiniert, die in der Vererbungsliste der verschiedensten Klassen erscheinen. Der Namensraum `System.Collections` ist beispielsweise eine wahre Fundgrube für Schnittstellen, die die Implementierungen von Listen, Arrays und einer Fülle weiterer Auflistungstypen strukturieren.

Die darin auch zu findenden Schnittstellen `IEnumerable` und `IEnumerator` geben ein prägnantes Beispiel für typische C#-Schnittstellen und deren verzwicktes Zusammenspiel ab. `IEnumerable` definiert nur eine Methode:

```
interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Die Methode `IEnumerable` liefert eine Instanz für die zweite Schnittstelle, `IEnumerator`, welche eine Eigenschaft und zwei Methoden definiert:

```
interface IEnumerator
{
  object Current {get; }   // Nur-Lesen-Eigenschaft
  bool MoveNext();
  void Reset();
}
```

Eine Klasse, die aufzählbar sein soll, implementiert die `IEnumerable`-Schnittstelle:

```
class EnumeratedObject : IEnumerable
```

Somit kann sich jeder Client, der mit dieser Klasse hantiert, blindlings darauf verlassen, dass ihre Objekte:

1. die Methode `GetEnumerator()` besitzen
2. `GetEnumerator()` ein Aufzählobjekt liefert, das wiederum die von der Schnittstelle `IEnumerator` geforderte Mindestausstattung besitzt.

Im Falle von C# verlässt sich übrigens auch der Compiler darauf: Für Klassen, die diese Schnittstelle implementieren, sind `foreach`-Iterationen zulässig.

Die Praxis sieht dann im Allgemeinen so aus: Ein Client vereinbart eine Objektvariable für die Schnittstelle und weist dieser ein Objekt einer Klasse zu, das die Schnittstelle implementiert. Die Schnittstellenvariable ermöglicht dann den Aufruf aller in der Schnittstelle spezifizierten Methoden nach folgendem Muster:

```
// Instanzierung von EnumeratedObject
System.Collections.IEnumerator Coll = new EnumObject(EnumeratedObject);
Coll.MoveNext();
Object Elem = Coll.Current;
```

Dem Datentyp `EnumObject` fällt hier die nicht ganz einfache Rolle der Aufzählklasse zu. Das folgende Codebeispiel geht auf diese Rolle näher ein.

Implementierung einer Aufzählklasse

Wenn man versucht, die Schnittstelle `IEnumerator` als Aufzählklasse zu implementieren, ergeben sich drei konkrete Fragen, auf die die Spezifikation der Schnittstelle an sich keine Antwort gibt:

1. Welches Objekt wird aufgezählt?
2. Wie sieht der Elementzugriff auf das Objekt aus?
3. Wie viele Elemente gibt es überhaupt?

Alle drei Fragen laufen darauf hinaus, dass die Implementierung der Aufzählklasse entweder konkret etwas über die aufzuzählenden Objekte wissen oder zumindest zusätzliche Vorannahmen machen muss.

»Konkret« ist immer einfacher. Für gewöhnlich werden Implementierungen dieser Schnittstelle daher spezifisch auf eine bestimmte Klasse (oder Basisklasse), die aufzählbar sein soll, zugeschnitten und sind meist sogar Bestandteil ihrer Klassendefinition – im Allgemeinen als verschachtelte Klasse. Die Implementierung erzwingt in diesem Fall über den Parametertyp des Konstruktors, welchen Datentyp das aufzuzählende Objekt haben soll, und hantiert ansonsten mit der (bekannten) spezifischen Ausstattung dieses Datentyps. Ein praxisnahes Beispiel für einen solchen Ansatz findet sich im Abschnitt »Codebeispiel – Primzahlen aufzählen« auf Seite 157

Codebeispiel – die eigene Schnittstelle IGeneralEnumerable

»Zusätzliche Vorannahmen« riechen hingegen nach einer weiteren Schnittstelle, die genau das Gewünschte von der Implementierung einfordert. Das Konsolenprogramm *EnumeratorBeispiel* demonstriert diesen Weg. Es definiert:

- Eine neue Schnittstelle `IGeneralEnumerable` auf der Basis von `IEnumerable`, die zusätzlich zu `GetEnumerator()` eine Methode `Item()` für den Elementzugriff und eine Eigenschaft `ItemCount` für Elementanzahl fordert.
- Eine einfache Klasse `MyGeneralEnumerator`, die `IGeneralEnumerable` implementiert.
- Eine Aufzählklasse `MyGeneralEnumerator`, die `IEnumerator` implementiert und beliebige Objekte mit der Schnittstelle `IGeneralEnumerable` aufzählen kann.

Der in der Methode `Main()` versammelte Testcode zeigt, wie die Aufzählung in diesem Szenario vonstatten geht und testet auch die mehrfache Aufzählung anhand einer `foreach`-Iteration.

```
// EnumeratorBeispiel demonstriert die Definition einer eigenen
// Schnittstelle. Die Objekte einer Klasse, die diese Schnittstelle
// implementiert, sind über die unspezifische Aufzählklasse
// MyGeneralEnumerator unabhängig vom zugrundegelegten Elementtyp
// auflistbar

using System;
namespace EnumeratorBeispiel
{
  // Schnittstelle fordet neben GetEnumerator() zusätzlich den Elementzugiff
  // und die Elementanzahl
  interface IGeneralEnumerable : System.Collections.IEnumerable
  {
    object Item(int i);     // Sollte IndexOutRange-Ausnahme generieren
    int ItemCount{ get; }   // Sollte Wert >= 0 liefern
  }
```

```
// Klasse implementiert die Schnittstelle IGeneralEnumerable
class MyEnumerable: IGeneralEnumerable
{
  public int[] Elems = {1,2,3,4};
  public virtual System.Collections.IEnumerator GetEnumerator()
  {
    return new MyGeneralEnumerator (this);
  }
  public virtual object Item(int i)
  {
    if(i < 0 || i > Elems.Length - 1)
      throw new IndexOutOfRangeException (
        "Element mit Index " + i + " nicht bekannt");
    return Elems[i];
  }
  public virtual int ItemCount
  {
    get
    {
      return Elems.Length;
    }
  }
}

// Unspezifische Aufzählklasse
// Achtung: Implementierung ist nicht threadsicher
class MyGeneralEnumerator : System.Collections.IEnumerator
{
  private const int InitIndex = -1;
  private bool Ready = false;            // Letztes Element aufgezählt?
  private int Index = InitIndex;         // Index der Aufzählung
  private IGeneralEnumerable collection; // aufzuzählendes Objekt
  public MyGeneralEnumerator (IGeneralEnumerable coll)
  {
    collection = coll;
  }

  // Liefert aktuelles Element
  public virtual object Current
  {
    get
    {
      if (Index == InitIndex)
        throw new System.ComponentModel.InvalidEnumArgumentException(
          "MoveNext nach Reset erforderlich");
      else if (Ready)
        throw new System.ComponentModel.InvalidEnumArgumentException(
          "Keine weiteren Elemente");
      // Hier ist eine Ausnahme zu befürchten, wenn das Element
      // inzwischen nicht mehr existiert
      return collection.Item(Index);
    }
  }
```

```
    // Nächstes Element anvisieren
    public virtual bool MoveNext()
    {
      if (Index < collection.ItemCount - 1 )
      {
        Index ++;
        return true;
      }
      else
      {
        Ready  = true;
        return false;
      }
    }

    // Auflistung erneut beginnen
    public virtual void Reset()
    {
      Index = InitIndex;
      Ready = false;
    }
  }
  class Class1
  {
    [STAThread]
    static void Main(string[] args)
    {
      // Schnittstellenzeiger auf aufzuzählendes Objekt
      IGeneralEnumerable GEable = new MyEnumerable();
      // Schnittstellenzeiger auf Allgemeines Aufzählerobjekt
      System.Collections.IEnumerator GEor = new
        MyGeneralEnumerator(GEable);
      GEor.MoveNext();  // Erstes Element
      GEor.MoveNext();  // Zweites Element
      Console.WriteLine("Zweites Element ist {0}", GEor.Current);
      // Zweite Aufzählung des gleichen Objekts dazwischenschieben
      foreach (object o in GEable)
        Console.Write(o);
      Console.WriteLine();
      // Erste Aufzählung fortsetzen
      GEor.MoveNext();
      Console.WriteLine("Drittes Element ist {0}", GEor.Current);
      // Resetvorgang
      GEor.Reset();
      GEor.MoveNext();
      Console.WriteLine("Erstes Element ist {0}", GEor.Current);
      MyEnumerable mj = new MyEnumerable();
      int i = (int) mj.Item (5);
    }
  }
}
```

8.6 Abstrakte Klassen

Um im Jargon zu bleiben: C# hat das Konzept der abstrakten Klasse von Delphi geerbt. Delphi steht seinerseits in der Tradition von C++, lässt aber im Gegensatz zu C++ Mehrfachvererbung nur bei Schnittstellen zu und beschränkt sich bei der Implementierungsvererbung generell auf die Einfachvererbung. C# folgt diesem Design.

8.6.1 Abstrakte Klassen und Schnittstellen in C++

In C++ ist das Konzept der abstrakten Klasse eher implizit angelegt: Eine Klasse ist dann eine abstrakte Klasse, wenn sie mindestens eine virtuelle Methode deklariert und nicht implementiert:

```
class MyAbstractCPPClass    /* C++-Klasse  */
{
  ...
  virtual void MyAbstractMethod(void) = 0;
};
```

Es ist nicht möglich, von einer abstrakten Klasse wie dieser Instanzen zu generieren, darauf achtet der Compiler. Eine abstrakte Klasse lässt sich aber als Basis für andere Klassen verwenden, die ihre Ausstattung erben, aber ihrerseits zu abstrakten Klassen werden, wenn sie keine Implementierung für noch nicht definierte virtuelle Methoden bereitstellen. Instanziierbar sind letztlich aber nur Klassen, bei denen alle virtuellen Methoden eine Definition (also Code) besitzen.

Schnittstellen werden in C++ gleichfalls implizit vereinbart. Eine abstrakte Klasse wird dann zur Schnittstelle, wenn sie ausschließlich virtuelle Methoden deklariert, ohne diese zu implementieren.

8.6.2 Abstrakte Klassen und Schnittstellen in C#

In C# (und Delphi) finden sich diese Konzepte gleichfalls – jedoch wesentlich expliziter. Die Sprache stellt ein Schlüsselwort dafür bereit. Da es mit Blick auf die Schnittstellenvererbung sinnvoll ist, Mehrfachvererbung zu erlauben, wird die Schnittstelle zum eigenständigen Konzept, das von der abstrakten Klasse abgetrennt ist:

- Die Deklaration einer Schnittstellenklasse erfolgt explizit mit dem Schlüsselwort `interface`. Eine Schnittstelle darf ausschließlich Methodenprototypen deklarieren, die der Compiler implizit als `public virtual` vereinbart.
- Abstrakte Klassen und ihre abstrakten Bestandteile werden explizit mit dem Schlüsselwort `abstract` deklariert. Eine C#-Klasse *muss* als `abstract`

deklariert werden, wenn sie auch nur ein `abstract`-Element vereinbart. `abstract`-Elemente sind hier implizit immer als `virtual` vereinbart – eine explizite `virtual`-Deklaration verbietet sich. Konsequent erfordern Implementierungen von abstrakten Elementen in abgeleiteten Klassen den `override`-Modifizierer. Auch dürfte sich verstehen, dass der Zugriffsmodifizierer `private` (also die Standardvorgabe des Compilers) für `abstract`-Elemente nicht zulässig ist.

(Delphi verlangt – und kennt – keine `abstract`-Deklaration von Klassen, fordert aber für abstrakte Methoden eine Festlegung auf *virtual* oder *dynamic*. Abstrakte Eigenschaften werden dort über abstrakte *read*- und *write*-Methoden definiert.)

Syntax

Die Syntax für die Vereinbarung einer abstrakten Klasse deckt sich weitgehend mit der einer gewöhnlichen Klasse.

```
[Modizifizier] abstract class MyClass [: BasisKlasse[, IInterface, ...]]
{
  [abstrakte Elemente]
  [nichtabstrakte Klassenelemente]
}
```

Als *abstrakte Elemente* kommen nur Elemente in Frage, die als `virtual` deklariert werden dürfen, also Instanzmethoden und -eigenschaften. Ihre Vereinbarung erfordert gleichfalls den Modifizierer `abstract`. Im Gegensatz zu C++ akzeptiert der C#-Compiler keine rein formalen Parameter – Bezeichner sind also nicht nur erforderlich, sie sollten auch »sprechend« genug gewählt werden, damit sie die Funktion des Parameters näher beschreiben. (Der Editor von VS.NET zeigt die Prototypen als Kontextinfo an, sowie eine Erläuterung, wenn Dokumentationskommentare vorhanden sind.)

Bei Instanzmethoden erwartet der Compiler ein Semikolon nach dem Prototyp, bei Eigenschaften hingegen nicht.

Codebeispiel – selbstdefinierte abstrakte Klasse

Der folgende Code aus dem Beispielprojekt *AbstrakteKlassen* vereinbart eine abstrakte Klasse mit zwei abstrakten und drei nicht-abstrakten Elementen:

```
public abstract class MyAbstractClass
{
  // Abstrakte Elemente
  public abstract int MyValue{ get; set; }      // abstrakte Eigenschaft
  public abstract string MyAbstractVirtual();   // abstrakte Methode
  // Nicht-Abstrakte Elemente
  public static int InstanceCounter;            // statisches Feld
  public string Name;                           // Instanzfeld
```

```
  public MyAbstractClass()
  {
    Name = "MyAbstractClass" + InstanceCounter;
  }
  public virtual string MyVirtual()             // Instanzmethode
  {
    return "MyAbstractClass.MyVirtual()";
  }
}
```

Eine Ableitung (und Implementierung) dieser abstrakten Klasse könnte beispielsweise so aussehen:

```
class MyDerivedClass : MyAbstractClass
{
  public new string Name;
  public MyDerivedClass ()
  {
    InstanceCounter++;
    Name = "MyDerivedClass" + InstanceCounter;  // nicht polymorph
  }
  // Implementiert die ererbte abstrakte Eigenschaft
  public override int MyValue
  {
    get
    {
      return InstanceCounter;                   // ererbtes Datenfeld
    }
    set
    {
    }
  }
  // Implementiert die ererbte abstrakte Methode
  public override string MyAbstractVirtual()
  {
    return "MyDerivedClass.MyAbstractVirtual()";
  }
  // Überschreibt die ererbte, bereits implementierte virtuelle Methode
  public override string MyVirtual()            // polymorph
  {
    return "MyDerivedClass.MyVirtual()";
  }
}
```

`MyDerivedClass` erfüllt ihren Pflichtteil, indem sie die beiden abstrakten Elemente implementiert. In der Kür pflegt sie den ererbten Instanzzähler `InstanceCounter`, überschreibt das Datenfeld `Name` und zudem die von der abstrakten Basisklasse bereits implementierte Methode `MyVirtual()`. Der folgende Testcode arbeitet mit beiden Klassen:

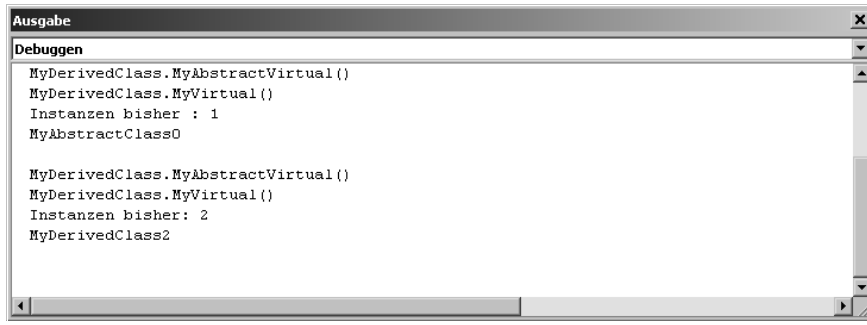
```
public void TestAbstract()
{
  MyAbstractClass mac = new MyDerivedClass();
```

```
Console.WriteLine(mac.MyAbstractVirtual());
Console.WriteLine(mac.MyVirtual());
Console.WriteLine("Instanzen bisher : {0}",
  MyAbstractClass.InstanceCounter);
Console.WriteLine(mac.Name);
Console.WriteLine();

MyDerivedClass mdc = new MyDerivedClass();
Console.WriteLine(mdc.MyAbstractVirtual());
Console.WriteLine(mdc.MyVirtual());
Console.WriteLine("Instanzen bisher: {0}", mdc.MyValue);
Console.WriteLine(mdc.Name);
```

Abbildung 8.21 zeigt die Ausgaben der Windows-Anwendung im Ausgabefenster des Debuggers.

Abbildung 8.21: Ausgabe des Testcodes



Die weitgehend polymorphe Implementierung bedingt, dass die Objektvariablen `mac` und `mdc` nahezu die gleiche Ausgabe generieren. Einziger Unterschied: `mac` verwendet das Instanzfeld `Name` der abstrakten Klasse, `mdc` die überschriebene Variante der abgeleiteten Klasse. In diesem Zusammenhang sollte ein weiteres interessantes Detail nicht unerwähnt bleiben: Die Konstruktoren beider Klassen personalisieren ihr Datenfeld `Name` durch Anhängen des Wertes von `InstanceCounter`. Da der Konstruktor der Basisklasse immer vor dem der abgeleiteten Klasse zum Aufruf kommt, enthält dieses Feld bei Konstruktion des Basisklassenanteils noch den Wert 0 – es wäre also sinnvoller, `InstanceCounter` über den Konstruktor der Basisklasse zu pflegen.



Eine C#-Klasse kann auch als `abstract` *vereinbart werden, wenn sie keine abstrakten Elemente definiert. In diesem Fall verhindert der Compiler schlicht, dass Instanzen der Klasse ins Leben gerufen werden, was beispielsweise für Klassen mit ausschließlich statischen Anteilen sinnvoll ist. Die (bisher übliche) Strategie, den Konstruktor in dem Fall als* `private` *zu vereinbaren, hätte zwar den gleichen Effekt, würde aber die weitere Ableitung der Klasse wirksam verhindern. (Wenn nur dies gewünscht ist, steht in C# das Schlüsselwort* `sealed` *zur Verfügung, das aus offensichtlichem Grund nicht zusammen mit* `abstract` *verwendet werden darf.)*

Für abgeleitete Klassen gilt die genannte Beschränkung nicht, auch wenn sie keinerlei Code beisteuern (und die Basisklasse also nur umbenennen), sofern sie nicht ihrerseits als `abstract` *vereinbart sind:*

```
abstract class MyAbstractClass
{
  public static int Counter = 10;
}
class MyDerived : MyAbstractClass{} // Keine eigene Implementierung
...
MyDerived md = new MyDerived();     // funktioniert problemlos
// MyAbstractClass ma = new MyAbstractClass(); // Fehler, abstrakt
int mi = MyAbstractClass.Counter;   // kein Problem
```

Implementierungszwang



Verglichen mit Delphi # ist der C#-Compiler etwas konsequenter und lässt dem Programmierer weniger Freiheit bei der Implementierung abgeleiteter Klassen. Während es der Delphi-Compiler bei einer Warnung belässt, wenn eine abgeleitete Klasse ein »nicht benutztes« abstraktes Element nicht implementiert, und also einen Laufzeitfehler (eine AbstractError-Ausnahme) in Kauf nimmt, falls das Element etwa über den Mechanismus der Typreflexion bzw. im Rahmen des Polymorphismus doch angesprochen wird, wertet der C#-Compiler »Nachlässigkeiten« dieser Art als Syntaxfehler – die Bemühung nach Typsicherheit lässt grüßen. Es kann also nötig sein, den Code einer C#-Klasse mit Dummyimplementierungen aufzublähen, die nichts weiter tun, als auf möglichst einfache Weise den erwarteten Rückgabewert zu generieren und gegebenenfalls auch Ausnahmen generieren, wenn sie wider Erwarten doch einmal zum Zuge kommen sollten.

8.6.3 Schnittstellen vs. abstrakte Klassen

Der Apfel fällt bekanntlich nicht weit vom Stamm. Es ist daher nicht verwunderlich, dass die Konzepte »Schnittstelle« und »abstrakte Klasse« große Ähnlichkeiten aufweisen.

Kombination aus konkreter Klasse und Schnittstelle?

Selbstredend ist eine abstrakte Klasse, die verschiedene Klassenelemente implementiert und nur einen Teil der Implementierung an die nächste Generation delegiert, mehr als nur eine Schnittstelle. Es macht also durchaus Sinn, eine abstrakte Klasse als Kombination aus einer nicht-abstrakten Klasse und einer Schnittstellendefinition zu betrachten.

Die Auffassung, *jede* abstrakte Klasse ließe sich in eine nicht-abstrakte Klasse und eine Schnittstelle auftrennen, ist allerdings falsch. Der Code einer abstrakten Klasse hat nämlich die Möglichkeit, die abstrakten Elemente der Klasse so zu verwenden, als seien sie implementiert – auf Kredit

sozusagen. Die nicht-abstrakte Klasse hätte hingegen entweder keine Kenntnis von diesen Elementen, oder müsste sie selbst implementieren, wenn sie ihrerseits als Ableitung der Schnittstelle aufträte.

Abstrakte Klasse als Ersatz für Schnittstelle?

Eine abstrakte Klasse, die ausschließlich abstrakte Elemente definiert, unterscheidet sich aus der Sicht einer abgeleiteten Klasse formal nur in einem – aber doch recht wesentlichen – Punkt von einer Schnittstelle: Sie unterliegt der Implementierungsvererbung, weshalb sich eine weitere Basisklasse (gleich, ob abstrakt oder nicht) in der Vererbungsliste verbietet. Wären dieselben abstrakten Elemente als Schnittstelle organisiert, bliebe diese Möglichkeit erstens erhalten und zweitens könnten beliebig viele weitere Schnittstellen in der Vererbungsliste erscheinen. Mit anderen Worten: Eine abstrakte Klasse muss auf Gedeih und Verderb die gesamte Ausstattung (abstrakte Elemente und Implementierung) definieren, die eine potenzielle Ableitung benötigt. Für weitere abstrakte Elemente bleibt dann nur noch der Weg der Schnittstellenvererbung.

Designrichtlinien

Angesichts der doch recht augenfälligen Überschneidung der beiden Konzepte stellt sich in der Tat die Frage, wann eine Funktionalität besser als Schnittstelle und wann sie besser als abstrakte Klasse zu kapseln ist.

Für eine Schnittstelle sollte man sich entscheiden, wenn es um die Definition scharf abgegrenzter Funktionalitäten oder auch nur Funktionsmerkmale geht, wie Aufzählbarkeit, Sortierbarkeit, Kopierbarkeit, Komprimierbarkeit, deren Formalisierung relativ unspezifisch für viele auch recht unterschiedliche Klassen von Interesse ist. Ein Paradebeispiel dafür wäre die Schnittstelle `ICloneable`. Sie fordert, dass abgeleitete Klassen die Methode `Clone()` implementieren und darüber eine Wertkopie ihrer selbst zurückgeben.

Umgekehrt begründen abstrakte Klassen im Allgemeinen thematisch eng verwandte Hierarchien von Klassen, die sich auf einen gemeinsamen Codekern stützen und diesen dann gegebenenfalls durch Überschreibung ererbter Klassenelemente individuell ausgestalten. Ein gutes Beispiel hierfür gibt die abstrakte Klasse `CommonDialog` ab, von der die bekannten Standarddialoge `ColorDialog`, `FontDialog`, `PrintDialog`, `PageSetupDialog` abstammen und über die gleichfalls abstrakte Ableitung `FileDialog` auch `FileOpenDialog` und `SaveDialog`.

Eine Schnittstelle ist wie schon gesagt, ein Vertrag, der (nach Veröffentlichung) nicht mehr kündbar ist, weshalb sich nachträgliche Änderungen an einer bereits veröffentlichten Schnittstelle verbieten. Mit anderen Worten, es gibt keinen Versionsbegriff für Schnittstellen. Statt eine geänderte oder erweiterte Version einer Schnittstelle herauszugeben, muss man hier eine neue Schnittstelle nachschieben, die zwar ihrerseits von der alten Schnittstelle abgeleitet sein kann, vom Effekt her aber dennoch als eigenständige und unabhängige Schnittstelle zu betrachten ist.

Abstrakte Klassen unterliegen dagegen der Versionskontrolle von .NET und lassen sich daher, mit unterschiedlichen Versionsnummern versehen, in verschiedenen Varianten unters Volk bringen. Die problemlose Koexistenz der Varianten auf ein und demselben System bewahrt die Integrität bestehender Anwendungen.

9 Ausnahmebehandlung

Seit jeher ist eine der unschöneren Seiten der Programmierung der Umgang mit Situationen und Bedingungen, in denen ein Programm leider nicht so kann wie es will. Die Gründe dafür sind ebenso zahllos wie in den meisten Fällen unvorhersehbar:

- Hausgemachte Fehler – solche Fehler lassen sich durch ein verbessertes Programmdesign ausmerzen.
  - *Typverletzungen* – ungültige Typumwandlungen, die erst durch die Laufzeittypprüfung aufgedeckt werden (beispielsweise beim Unboxing).
  - *Wertebereichsüberschreitung* – Operationen, deren Ergebnis im Zieldatentyp nicht mehr darstellbar sind (zum Beispiel die Multiplikation zweier `int`-Werte und Zuweisung zu einer `int`-Variablen).
  - *Indexfehler* – falsch berechnete Indizes oder nicht ausreichend dimensionierte Arrays.
- Systemseitige Fehler – erscheinen aus heiterem Himmel, lassen sich nicht vorhersehen und sind im Allgemeinen auch schlecht behandelbar.
  - *Veränderte Systembedingungen* – eine andere Anwendung (beispielsweise ein Installationsprogramm oder die Systemsteuerung) hat die Systemkonfiguration (z.B. Bildschirmauflösung, Komponenten) seit dem letzten Start oder zur Laufzeit Programms verändert.
  - *knappe Ressourcen* – Speicherplatzmangel, das System kann benötigte Ressourcen wie Arbeitsspeicher, Fenster- oder Dateiobjekte (bzw. Handles dafür) aufgrund wie auch immer gearteter Knappheiten nicht mehr bereitstellen.
- I/O-Fehler – dies ist die häufigste Fehlerursache. Fehler dieser Art lassen sich gut behandeln und bereiten im Allgemeinen keine ernsthaften Schwierigkeiten.
  - *Benennungsprobleme* – Namenskonflikte (unzulässige Bezeichner, fehlende Eindeutigkeit, nicht existierende Pfade), volle Datenträger, nicht auffindbare Ressourcen.

- *Technische Ausfälle* – ausgefallene Netzwerkverbindungen, defekte Datenträger oder I/O-Geräte, Überschreitungen von Zeitlimits für Kommunikationsanforderungen et cetera, korrumpierte Daten etwa aufgrund von Virenbefall.
- *Zugriffsverletzungen* – Verstoß gegen Sicherheitszonen, eingeschränkte Zugriffsrechte aufgrund von Systemrichtlinien oder gemeinsamer Nutzung.

Neben Fehlern, die aus einer mangelhaften Programmlogik herrühren und durch ein konsequentes Software-Design sowie praxisnahe Testzyklen zumindest vom Prinzip her eliminierbar sind, gibt es viele Fehlerbedingungen, die ein Programm weder vermeiden noch vorhersehen kann. Aber: Es kann sich darauf einrichten, dass solche Fehler passieren, und für diesen Fall verschiedene Vorkehrungen treffen, um

- *den Fehler aussagekräftig zu melden* – Meldungen dieser Art können ein Feedback für die Fehlersuche sein oder Aufschluss über Problemschwerpunkte für die weitere Entwicklung geben.
- *den Fehler zu umschiffen* – beispielsweise durch Aufruf eines Suchdialogs, wenn eine Ressource an der erwarteten Stelle nicht gefunden wurde.
- *den geordneten Rückzug anzutreten* – den Programmzustand soweit konservieren, dass auf jeden Fall bereits eingegebene Daten nicht verloren sind.

Dieses Kapitel diskutiert die Ausnahmenbehandlung unter drei Aspekten. Mit Blick auf den Mechanismus an sich als sprachübergreifende Einrichtung von .NET, mit dem Schwerpunkt auf der Codeseite und die in C# dafür vorhandenen Sprachkonstrukte, und schließlich konzentriert auf die in der .NET-Klassenhierarchie vordefinierten Ausnahmeklassen und deren benutzerdefinierte Erweiterung.

9.1 Der Mechanismus

Die beiden schlagendsten Argumente für eine Entwicklung unter .NET sind:

- die Möglichkeit der sprachübergreifenden Verwendung von Komponenten
- Typsicherheit durch eine verwaltete Umgebung.

Beide Argumente haben unmittelbar mit dem Fehlerbehandlungsmechanismus von .NET zu tun.

9.1.1 Sprachübergreifende Fehlerbehandlung

Für den sprachübergreifenden Einsatz einer Komponente – der Anspruch von .NET geht hier ja bis zur völligen Transparenz – muss auch und vor allem ein sprachübergreifender Fehlerbehandlungsmechanismus existieren. Das heißt: Fehlerinformationen müssen sprachübergreifend generier- und verwertbar sein.

Unter .NET ist diese Forderung weit weniger dramatisch, als man vielleicht vermutet. Das sprachübergreifende Typsystem bietet alle Voraussetzungen, um Komponenten und Fehlerinformationen aus dem gleichen Holz zu schnitzen – nämlich über Klassen, Objekte und Vererbung. Verglichen mit anderen Programmiersprachen oder gar COM sind dies geradezu paradiesische Voraussetzungen für eine strukturierte Fehlerbehandlung nach allen Regeln der Kunst. Vor allem Fans der polymorphen Programmierung kommen hier voll auf ihre Kosten.

9.1.2 Typsicherheit

Der Aspekt der Typsicherheit stellt Compiler und Laufzeitsystem gleichermaßen vor eine harte Aufgabe. Es gilt, jegliche Abwege von Programmen bereits im Keim zu ersticken und in geordnete Bahnen zu lenken. Während der Compiler auf strikter Einhaltung des Typsystems beharrt, nimmt das Laufzeitsystem eine Laufzeittypüberprüfung vor und meldet Verstöße gegen das Typsystem sowie alle anderen Unregelmäßigkeiten – wie sollte es anders sein – über den .NET-Fehlermechanismus.

Tatsächlich haben die Entwickler von .NET hier ganze Arbeit geleistet. Programmabstürze, die das gesamte Betriebssystem mit sich reißen oder die Maschine aufhängen, kommen schlicht nicht mehr vor. Selbst die übelsten Patzer pariert das System stoisch, indem es Laufzeitfehler meldet, ohne auch nur im Geringsten außer Tritt zu geraten. Die Erfahrungen der Autoren dieses Buchs sind hier durchwegs positiv und unterstreichen die erstaunliche Stabilität.



9.1.3 Das Prinzip

Ein Fehler in einem Computer ist keineswegs ein ungeordneter oder gar chaotischer Zustand. Er ist das definierte Signal einer Routine, die bei einem Test eine Fehlerbedingung erkannt hat und dies aktiv kundtut, nicht mehr und nicht weniger. Adressaten des Signals sind im Allgemeinen speziell für die Fehlerbehandlung registrierte Routinen, die auf die Bearbeitung der Fehlerbedingung eingerichtet sind – sei es, um sie zu korrigieren, zu ignorieren, den Zustand des Programms darauf einzurichten oder sie schlicht weiterzumelden. Der Weg, den ein solches Signal nimmt, ist klar vorgegeben: Er

führt immer vom Speziellen zum Allgemeinen und von innen nach außen, also von den Tiefen des Systems über die Bibliotheken zum Programm und dann weiter zur Laufzeitumgebung als äußerster Instanz.

Ob die Routine, die den Fehler signalisiert, nun in der Tiefe des Systems, in der .NET-Laufzeitumgebung, einer .NET-Klasse der Klassenhierarchie oder im eigenen Code steckt, macht formell keinen Unterschied – Signal ist Signal. Traditionell wurde dem Signal nichts weiter als eine Nummer zugeordnet, die Aufschluss über die Art des Fehlers geben sollte und sich ihrerseits in einen natürlichsprachlichen Fehlertext auflösen lies. (Dieses Prinzip findet sich in Visual Basic 6 übrigens nach wie vor).

Ausnahmen und strukturierte Fehlerbehandlung

Mit der Klassenkonzeption von C++ hat auch die strukturierte Fehlerbehandlung auf der Basis von Ausnahmen Einzug in die Programmierung gehalten – .NET verstärkt hier nur einen Trend. Die bis dahin verbreiteten, mehr oder weniger hemdsärmeligen Fehlerbehandlungsstrategien (Rückgabe des Fehlercodes als Funktionswert, explizite Abfrage von Fehlerinformationen über gemeinsam genutzte Fehlerstrukturen, Setzen programmeigener Fehlerroutinen und Ausführungskontexte dafür) erhielten endlich ein einheitliches Gesicht und fanden in Form von `try/catch/finally`-Kontrollstrukturen ihren Niederschlag in der Sprache.

Die strukturierte Fehlerbehandlung macht es möglich, Fehler explizit als Ausnahmen zu signalisieren (`throw`) und auf der Basis von Anweisungsblöcken gezielt Fehlerbehandlungsroutinen dafür zu registrieren, die mehr oder weniger spezifisch auf solche Ausnahmen regieren – sie entweder auffangen (`catch`) oder »unbeeindruckt« zur Kenntnis nehmen und vor der Weiterleitung zur nächsten Ebene für Aufräumarbeiten sorgen (`finally`).

Das soll aber nicht heißen, dass beispielsweise die Rückgabe eines Fehlercodes auf der Basis eines Funktionswerts nun gar nicht mehr anzutreffen oder sinnlos wäre. Auch .NET-Methoden machen davon regen Gebrauch, man denke etwa an die `string`-Routine `IndexOf()`, die über den Rückgabewert –1 das »Misslingen« der Operation signalisiert. In der Regel sollten Ausnahmen tatsächlich nur in Ausnahmesituationen eingesetzt werden, die Auswirkungen auf das reguläre Verhalten des Programms haben. Das gilt allein schon deshalb, weil der Mechanismus nicht ohne einen gewissen Überbau an Laufzeit und Speicherbedarf auskommt, der beispielsweise für Sonderwerte von Funktionsaufrufen eher ungerechtfertigt ist.

unchecked vs. checked

Die Grenze, wann die Signalisierung von Ausnahmen sinnvoll ist, und wann nicht, ist hier aber oft schlecht zu ziehen und hängt von verschiedenen Faktoren ab. So geht der C#-Compiler mit Über-/Unterläufen der integrierten

Datentypen beispielsweise sorgloser um als der VB.NET-Compiler. Standardmäßig generiert er keinen Code, der die Einhaltung der Wertebereichsgrenzen für einen arithmetischen Datentyp prüft und gegebenenfalls eine Ausnahme signalisiert. Falls dieses Verhalten generell für den gesamten Code eines Projekts erwünscht ist, lässt sich jedoch ein entsprechender Compilerschalter auf der Eigenschaftenseite des jeweiligen Projekts setzen.

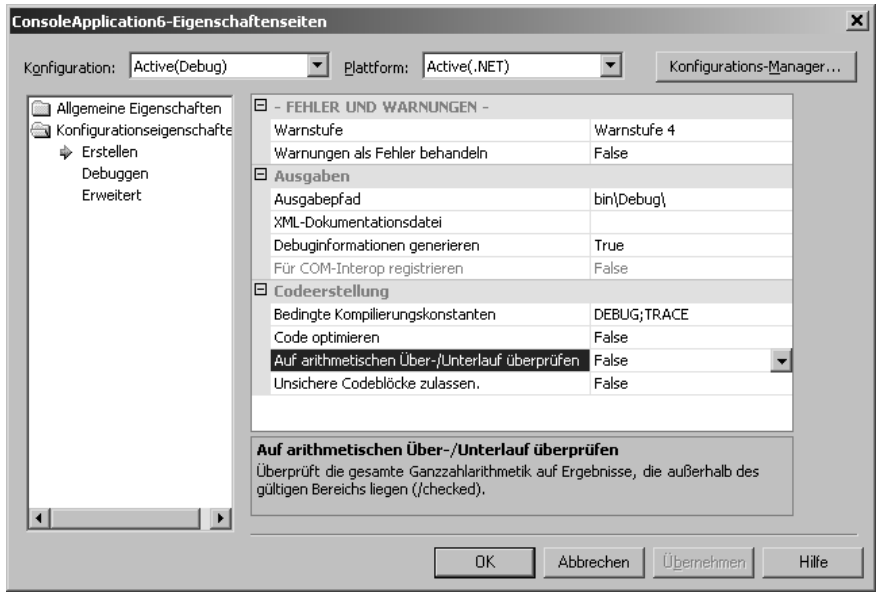


Abbildung 9.1: Compilerschalter für die Signalisierung von Über-/Unterläufen

Um die Prüfung nur für bestimmte Codeblöcke gezielt ein- bzw. auszuschalten, finden sich im Sprachumfang von C# die Kontrollstrukturen `checked` und `unchecked`. Der folgende Codeauszug demonstriert ihre Anwendung.

```
uint i = 0xffffffff;
Console.WriteLine(i);           // 4294967295
Console.WriteLine(i + 1);       // 0, wenn Compilerschalter nicht gesetzt,
                                // ansonsten Ausnahme wegen Überlauf
unchecked
{
  Console.WriteLine(i);         // 4294967295
  Console.WriteLine(i + 1);     // 0
}
checked
{
  Console.WriteLine(i);         // 4294967295
  Console.WriteLine(i + 1);     // Ausnahme wegen Überlauf
}
```

9.1.4 Ausnahmeobjekte

Medium für die Übermittlung der spezifischen Fehlerinformationen für Ausnahmen sind Objekte von Ausnahmeklassen. In der .NET-Klassenhierarchie findet sich eine ganze Dynastie von Ausnahmeklassen, die ihren Ausgangspunkt in der gemeinsamen Basisklasse `Exception` haben. Diese Klasse bietet verschiedene Eigenschaften (und – für die praktisch ausgerichtete Programmierung weniger wichtig – Methoden) als Grundausstattung für die Fehlerbeschreibung an. Tabelle 9.1 gibt einen Überblick.

Tabelle 9.1: Eigenschaften der Klasse `Exception`

Eigenschaft	Beschreibung
`string HelpLink {get; set;}`	Verknüpfung mit Hilfeinformation zu diesem Fehler.
`InnerException` `InnerException {get;}`	Primäres Ausnahmeobjekt, wenn es sich bei der Ausnahme um ein sekundäres Ausnahmeobjekt handelt. Ausnahmeklassen überladen für diesen Zweck eigens Konstruktoren, die es ermöglichen, bei Konstruktion (i.A. innerhalb einer `catch`-Blocks) gegebenenfalls das ursprüngliche Ausnahmeobjekt mitzugeben. Über die `InnerException`-Eigenschaft lässt sich die Kette der Ausnahmeobjekte bis zum ursprünglichen Ausnahmeobjekt zurückverfolgen.
`string Message {get; set;}`	Text, der die Ursache der Ausnahme genauer beschreibt.
`string Source {get; set;}`	Name der Anwendung oder Klasse, in der der Fehler aufgetreten ist.
`string StackTrace {get;}`	Eine Liste aller Methoden, deren Abbruch die Ausnahme seit ihrem ursprünglichen Auftreten erzwungen hat (Aufruf-Stack). Die Information umfasst den vollqualifizierten Methodennamen, den zugehörigen Quelltext (.cs-Datei) und die Zeilennummer des darin enthaltenen Aufrufs.
`string TargetSite {get; }`	.NET-Prototyp der Methode, in der die Ausnahme aufgetreten ist – z.B: `void MyMethod(Int32)`.

Die in der .NET-Klassenhierarchie vordefinierten Ableitungen sind oft nur speziellere Benennungen der Basisklasse oder sie ergänzen deren Ausstattung nach Bedarf um zusätzliche Informationen nach den üblichen Regeln des Klassendesigns. Beispielsweise fügt die Klasse `ArgumentException` eine `ParamName`-Eigenschaft für den Namen des fehlerhaften Aufrufparameters hinzu. Zu den Ableitungen gleich noch mehr. Zuerst aber die sprachliche Seite.

9.2 Syntax

Die Syntax für die Ausnahmebehandlung in C# stammt eins zu eins von C++. Sie sieht einen `try`-Block kombiniert mit einem oder mehreren `catch`-Blöcken sowie einem `finally`-Block vor.

```
try
{
  [Anweisungsfolge]
}
[catch (SpezialisierteAusnahmeKlasse e)
{
  [Anweisungsfolge]
}]
[catch (GleichOderWenigerSpezialisierteAusnahmeKlasse e)
{
  [Anweisungsfolge]
}]
...
[catch
{
  [Anweisungsfolge]
}]
[finally
{
  [Anweisungsfolge]
}]
```

9.2.1 try

Der `try`-Block enthält die regulären Anweisungen des Programms, die möglicherweise zu einer Ausnahme führen – und übernimmt somit die Rolle des Senders der Ausnahme. Stellt er einen Fehler fest, konstruiert er ein Ausnahmeobjekt und bindet verschiedene Informationen an dieses – wie von der Ausnahmeklasse vorgesehen. Im Minimalfall wird dies eine Fehlermeldung in Form eines `string`-Ausdrucks sein. Ein Ausnahmeobjekt kann vom Prinzip her aber auch beliebige Kontexte übermitteln, die für die Fehleranalyse potenziell wertvoll sein könnten.

9.2.2 throw

Für die Signalisierung einer Ausnahme gibt es das Schlüsselwort `throw`:

```
throw Ausnahmeobjekt;
```

Liegt eine `throw`-Anweisung nicht auf der gleichen Prozedurebene wie der `try`-Block, unterbricht sie die Ausführung der aktuellen Methode umgehend und führt einen unbedingten Rücksprung in den Ausführungskontext der Methode aus, die den `try`-Block enthält. Sofern vorhanden, werden dabei

sukzessive auch die Stackrahmen aller im Aufrufstapel dazwischen liegenden Methoden abgebaut und die Aufrufe eliminiert – ohne Rückgabe der Kontrolle, versteht sich. Die Namen aller unterbrochenen Methoden finden sich in der `StackTrace`-Eigenschaft des Ausnahmeobjekts.



`throw` *entspricht direkt dem Delphi-Schlüsselwort* `raise`.

9.2.3 catch

Die immer auf einen `try`-Block folgenden `catch`-Blöcke sind für die Behandlung der Ausnahme zuständig. Dabei gilt, dass auf einen `try`-Block immer unmittelbar ein `catch`- und/oder ein `finally`-Block folgen muss. Gibt es keinen `catch`-Block, muss ein `finally`-Block vorhanden sein und umgekehrt. Es dürfen allerdings auch mehrere `catch`-Blöcke für unterschiedliche Ausnahmeklassen folgen, wobei die (vom Compiler durchgesetzte) Anordnung den Ableitungszusammenhang der Ausnahmeklassen vom Speziellen zum Allgemeinen (`Exception`-Klasse) widerspiegeln muss. Die Reihenfolge für nebengeordnete Ausnahmeklassen ist hingegen nicht vorgeschrieben.

Im Gegensatz zu herkömmlichen Fehlerbehandlungsmechanismen kommt die strukturierte Fehlerbehandlung ohne eine explizite Analyse von Fehlercodes aus. Sie nutzt die Laufzeittypinformation als Auswahlkriterium dafür, welcher `catch`-Block die Ausnahme behandeln soll. Grundsätzlich kommt immer nur *ein* `catch`-Block zum Zuge – und zwar derjenige, der am besten auf das Ausnahmeobjekt passt.

Die allgemeinste Fassung für einen `catch`-Block ist, wenn kein Ausnahmeobjekt vereinbart wird:

```
catch
{ ... }
```



Findet sich kein passender `catch`*-Block, was beispielsweise der Fall ist, wenn das Ausnahmeobjekt weniger spezialisiert ist als die in dem* `catch`*-Block vereinbarte Ausnahmeklasse, kann auf aktueller Ebene keine Ausnahmebehandlung stattfinden. (Falls ein* `finally`*-Block vorhanden ist, wird dieser jedoch auf jeden Fall ausgeführt.) Der Mechanismus sucht dann – immer entlang der Aufrufliste – nach dem nächsten vorgeordneten* `try`*…*`catch`*-Konstrukt mit passendem* `catch`*-Block. In letzter Instanz ist dies die* `try`*-Anweisung des Laufzeitsystems, die die Klasse mit der Methode* `Main()` *für den Programmstart instanziiert hat. Damit gibt es in .NET letztlich keine unbehandelten Ausnahmen.*

Der folgende Codeauszug stammt aus dem in Teil 3 diskutierten Codebeispiel *ListBoxDemo*. Aufgabe der `Load`-Behandlung ist es, `Font`-Objekte für

alle auf dem System installierten Schriften zu erzeugen und in einem Listenfeld zu speichern. Da nicht jede Schrift in dem implizit geforderten Schriftstil »Regular« verfügbar ist, kann es in der `foreach`-Schleife zu Ausnahmen kommen. Die Behandlung ist trivial: Die Schrift wird schlicht ausgelassen.

```
private void FormOwnerDraw_Load(object sender, System.EventArgs e)
 {
   listBox1.DrawMode = DrawMode.OwnerDrawVariable;
   // Alle im System installierten Schriften in die Liste aufnehmen
   foreach (FontFamily ff in FontFamily.GetFamilies(CreateGraphics()))
   {
     try   // nicht alle Fonts unterstützen FontStyle.Regular
     {
       System.Drawing.Font f = new System.Drawing.Font(ff, 20);
       listBox1.Items.Add(f);
       if (f.Name == Font.Name)
         listBox1.SelectedIndex = listBox1.Items.IndexOf(f);
     }
     catch {}
   }
...
```

Ein anderes Beispiel mit nicht so trivialem `catch`-Block wäre:

```
DialogResult res = DialogResult.OK;
do
{
  try
  {
    WriteMyFile();
  }
  catch (Exception exc)
  {
    string msg =
     "Beim Schreiben in die Datei ist folgender Fehler aufgetreten: " +
      exc.Message;
    res = MessageBox.Show(msg, "Fehlermeldung",
            MessageBoxButtons.AbortRetryIgnore, MessageBoxIcon.Question);
  }
} while (res == DialogResult.Retry);
```

Tritt beim Schreiben der Datei mit `WriteMyFile()` ein Fehler auf, kommt der `catch`-Block mit dem darin befindlichen `MessageBox()`-Aufruf zur Ausführung. Je nach Ergebnis des `MessageBox()`-Aufrufs erfolgt ein Wiedereintritt in den Schleifenkörper und eine Wiederholung der Schreiboperation oder ein Abbruch.

9.2.4 finally

Der `finally`-Block enthält Code, im Allgemeinen Aufräumcode, der in jedem Fall ausgeführt wird – gleich, ob

- eine `catch`-Fehlerbehandlung vorhanden ist, oder nicht
- eine `catch`-Fehlerbehandlung notwendig ist (das heißt: eine Ausnahme signalisiert wird), oder nicht
- die `catch`-Fehlerbehandlung ihrerseits via `throw` eine Ausnahme signalisiert, oder nicht.

Dabei gilt die Regel, dass der `finally`-Block unmittelbar *nach* dem `catch`-Block oder, falls kein solcher vorhanden ist, nach Auftreten der Ausnahme ausgeführt wird. Allerdings ist es möglich – und auch übliche Praxis –, von einem `catch`-Block aus per `goto` Marken innerhalb der aktuellen Methode anzuspringen. In diesem Fall gilt folgendes Verhalten: Liegt das Sprungziel *vor* dem `finally`-Block, erfolgt der Sprung ohne Ausführung des `finally`-Blocks (»gehe direkt dorthin, gehe nicht über Los«); liegt das Sprungziel *nach* dem `finally`-Block, kommt zuerst der `finally`-Block zur Ausführung. Sprünge in den `finally`-Block sind nicht erlaubt, da C# Sprünge in Blöcke hinein grundsätzlich verbietet. Der folgende Testcode demonstriert den ersten Fall:

```
string s = "";
startEingabe:
try
{
  Console.WriteLine("Bitte ein 'a' eingeben");
  s = Console.ReadLine();
  if ( s != "a")
    throw new Exception("War das ein 'a'?");
}
catch(Exception e)
{
  Console.WriteLine(e.Message);
  goto startEingabe;

}
finally
{
  Console.WriteLine("Eingabe war: {0}", s);
}
Console.WriteLine("Ende");
```

Abbildung 9.2: Ausgaben des Testcodes



9.3 .NET-Ausnahmeklassen

In der .NET-Klassenhierarchie ist eine bunte Vielfalt von Ausnahmeklassen vordefiniert, die insbesondere für die Fehlersignalisierung gemeinsam genutzter Komponenten genauestens studiert und auch ausgereizt werden sollte. Natürlich spricht auch nichts dagegen, die bestehende Hierarchie durch eigene Ableitungen noch weiter zu verästeln. Eigene Ausnahmeklassen sollten dann als verschachtelte `public`-Klassen der die Ausnahme signalisierenden Komponentenklasse exportiert werden, um den Zusammenhang zu wahren. Das am Ende dieses Abschnitts vorgestellte Programm *AusnahmenDemo* enthält Beispielcode dafür.

In der Regel dürfte das Angebot der .NET-Klassenhierarchie aber mehr als ausreichend sein, solange es nur konsequent genug eingesetzt wird. Tabelle 9.2 stellt die am häufigsten anzutreffenden Ausnahmeklassen vor; Abbildung 9.3 zeigt eine repräsentative Auswahl der von `SystemException` abgeleiteten Ausnahmeklassen im Vererbungszusammenhang.

Tabelle 9.2: Wichtige Ausnahmeklassen der .NET-Klassenhierarchie

Ausnahme	Beschreibung
`Exception`	Basisklasse aller Ausnahmen
`SystemException`	Basisklasse aller zur Laufzeit generierten Ausnahmen
`NullReferenceException`	Signalisiert das unerwartete Auftreten einer nicht instanziierten Objektvariablen
`InvalidOperationException`	Signalisiert, dass die betroffene Operation für das jeweilige Objekt (und den aktuellen Zustand des Objekts) nicht anwendbar ist.
`IndexOutOfRangeException`	Signalisiert eine Bereichsverletzung für einen Arrayindex.
`IOException`	Basisklasse für Ausnahmen, die für Dateizugriffe (allgemeiner: I/O-Operationen) zuständig sind.
`ArgumentException`	Basisklasse für Ausnahmen, die mit Fehlern bei der Parameterübergabe im Zusammenhang stehen.
`ArgumentNullException`	Signalisiert einen unerwarteten (unzulässigen) `null`-Parameter

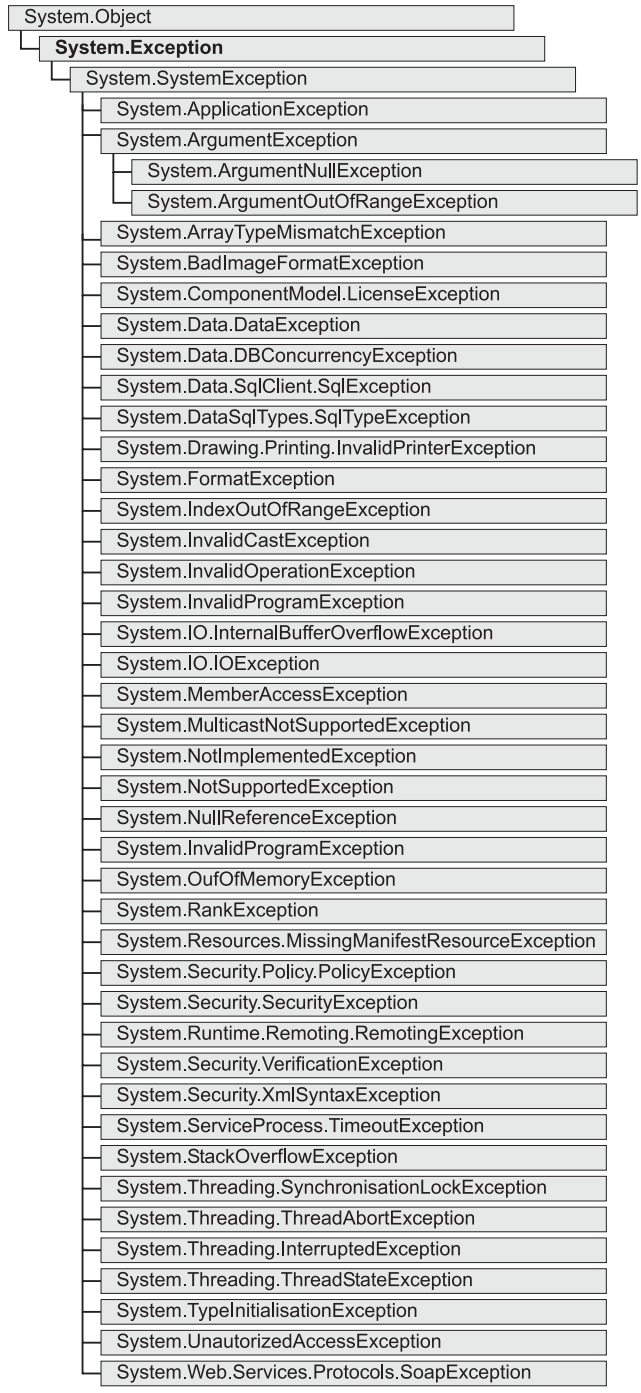


Abbildung 9.3: Überblick über die wichtigen `Exception`-Klassen in der .NET-Klassenhierarchie. Viele der Klassen sind ihrerseits Basisklassen für weitere Ableitungen.

9.3.1 Codebeispiel – Ausnahmen und Ausnahmeklassen

Das Programm *AusnahmenDemo* demonstriert die verschiedenen Techniken, die im Zusammenhang mit Signalisierung und Behandlung von Ausnahmen stehen:

- Signalisierung primärer vs. sekundärer Ausnahmen
- Ausnahmebehandlung mit erneutem Eintritt in den abgebrochenen Anweisungsblock
- Typunterscheidung auf der Basis von `catch` vs. Typabfrage in der Routine
- Auswertung der Aufrufliste und sonstiger Fehlerinformationen
- Vereinbarung und Einsatz eigener Ausnahmenklassen

Bei dem Programm handelt es sich um das Gerippe einer formularlosen Windows-Anwendung, deren Aktivitäten sich auf die Ausgabe von Meldungsdialogen sowie auf Testausgaben in das Debuggerfenster AUSGABE beschränken.

Der Code umfasst zwei verschachtelte `try`-Blöcke, von denen der innere in einer Schleife sitzt und die Methode `WriteMyFile()` aufruft. Anstatt ihrem Namen Ehre zu machen, beschränkt sich diese Methode wahlweise auf die Signalisierung einer `IOException`- oder `Exception`-Ausnahme, je nachdem, ob der Zeilenkommentar entfernt wird oder nicht. Die Behandlung der Ausnahme ruft im ersten Fall einen Meldungsdialog auf den Plan, der die Auswahl erlaubt, die Operation zu wiederholen, sie abzubrechen oder den Fehler zu ignorieren. Bei Abbruch signalisiert der `catch`-Block eine sekundäre Ausnahme, die von der äußeren `try…catch`-Struktur dahingehend behandelt wird, dass eine Auswertung der Fehlerinformationen mit Ausgabe in das Debuggerfenster erfolgt. Der Ablauf des Programms lässt sich am besten im Einzelschrittmodus des Debuggers studieren.

```
public class Form1 : System.Windows.Forms.Form
{
  static void Main()
  {
    try
    {
      TestExceptions();
    }
    catch (Exception e)                    // Fehlerauswertung
    {
      Exception eOrg = e;
      while(e.InnerException != null)
      {
        Console.WriteLine(e.Source + ": " +e.Message);
        e = e.InnerException;
      }
```

```
      Console.WriteLine("Stack:\n=====\n" + eOrg.StackTrace);
    }
  }
  static void WriteMyFile()
  {
    // Das musste ja schiefgehen ...
    throw new System.IO.IOException("Datenträger voll");
    // zweiter Weg ...
    // throw new System.SystemException("Datenträger voll");
  }

  static void TestExceptions()
  {
    DialogResult res = DialogResult.OK;
    do
    {
      try
      {
        WriteMyFile();
      }
      catch (System.IO.IOException exc)  // Dateisystemrelevante Fehler
      {
        string msg =
          "Beim Schreiben in die Datei ist folgender Fehler aufgetreten: "
          + exc.Message;
        res = MessageBox.Show(msg, "Fehlermeldung",
          MessageBoxButtons.AbortRetryIgnore, MessageBoxIcon.Question);
        if (res == DialogResult.Abort) // Abbrechen? Ausnahme signalisieren
          throw new System.IO.IOException(msg, exc);
      }
      catch (SystemException exc)  // andere Fehler werden nicht korrigiert
      {
        string msg = "Nicht korrigierbarer Fehler erkannt : " +
          exc.Message;
        MessageBox.Show(msg, "Fehlermeldung",
          MessageBoxButtons.OK, MessageBoxIcon.Error);
        throw new SystemException(msg, exc);
      }
    } while (res == DialogResult.Retry);
  }
}
```

Die Ausgabe des Programms im Ausgabefenster des Debuggers hat das folgende Muster:

```
AusnahmenDemo: Beim Schreiben in die Datei ist folgender Fehler aufgetreten:
Datenträger voll
Stack:
=====
   at AusnahmenDemo.Form1.TestExceptions() in D:\Dokumente und
Einstellungen\Administrator\Eigene Dateien\Visual Studio
Projects\Buchprojekte\Teil II\AusnahmenDemo\Form1.cs:line 74
   at AusnahmenDemo.Form1.Main() in D:\Dokumente und
Einstellungen\Administrator\Eigene Dateien\Visual Studio
Projects\Buchprojekte\Teil II\AusnahmenDemo\Form1.cs:line 40
Das Programm "[1340] AusnahmenDemo.exe" wurde mit Code 0 (0x0) beendet.
```

Benutzerdefinierte Ausnahmen

Zur Signalisierung von Ausnahmeobjekten mit benutzerdefiniertem Datentyp ist die Definition einer eigenen Ausnahmeklasse erforderlich – vorzugsweise als eingeschachtelter, aber natürlich als `public` gekennzeichneter Datentyp der signalisierenden Klasse. Als Basis muss zumindest die Klasse `Exception` angegeben sein, besser ist jedoch eine weiter spezialisierte Ausnahmeklasse.

Die folgende Definition der Klasse `MyNotSupportedException` ist eine Spezialisierung der Ausnahmeklasse `NotSupportedException`. Die Klasse vereinbart ein zusätzliches `MyContext`-Datenfeld `Context` für zusätzliche Fehlerinformation sowie einen Konstruktor, der einen Parameter dieses Typs entgegennimmt:

```
public class Form1 : System.Windows.Forms.Form
{
  public class MyNotSupportedException : NotSupportedException
  {
    public MyNotSupportedException(string message,
     Exception innerExc, MyContext mc ): base(message, innerExc)
    {
      Context = mc;
    }
  public MyContext Context;
}

public class MyContext
{
  public string Message = "MyContext";
}
```

Vom Umgang her unterscheiden sich benutzerdefinierte Ausnahmeklassen in nichts von .NET-Ausnahmeklassen. Das Muster für die Signalisierung einer Ausnahme dieses Typs wäre also:

```
Context context = new MyContext();
// TODO Initialisierung von context
throw new MyNotSupportedException(msg, exc, context);
```

Und ein `catch`-Block dafür sieht beispielsweise so aus:

```
catch(MyNotSupportedException mne)
{
  Console.WriteLine(mne.Context.Message);
  throw new Exception(mne.Message, mne);   // sekundäre Ausnahme
}
```

Typspezifische Auswertung in allgemeinem catch-Block

Nicht immer ist die Filterwirkung typisierter `catch`-Blöcke ein Segen. Sie hat den Nachteil, dass immer nur ein `catch`-Block zur Ausführung kommt. Eine Möglichkeit, in den Code anderer `catch`-Blöcke zu verzweigen, ist nicht vor-

gesehen. In vielen Fällen läuft aber die Fehlerbehandlung auch für unterschiedliche Fehlertypen weitgehend analog, sodass eine Implementierung mit *Fall-through*-Logik vorteilhafter wäre. Man erreicht dies durch »konventionelle« Auswertung der Laufzeittypinformation:

```
catch (Exception e)  // Fehlerunterscheidung in der Routine
{
  if (e.GetType() == typeof(MyNotSupportedException))
  {
    MyNotSupportedException mse = (MyNotSupportedException) e;
     Console.WriteLine("Fehlerart: MyNotSupportedException, Context: " +
        mse.Context.Message);
    }
...
```

9.4 Ausnahmen richtig eingesetzt

Abschließend noch eine Liste mit Ratschlägen für den wohldosierten Umgang mit Ausnahmen:

- Signalisieren Sie nur dann eine Ausnahme, wenn es die Situation wirklich erfordert, das heißt, wenn ein Funktionswert oder Ausgabeparameter für die gleiche Aufgabe nicht in Frage kommt. Ausnahmen haben im Rahmen des regulären Programmablaufs nichts zu suchen.
- Geben Sie jeder Ausnahme, die Sie signalisieren, einen aussagekräftigen Fehlertext mit.
- Signalisieren Sie immer den Ausnahmetyp, der für die Situation am besten passt. Leiten Sie erforderlichenfalls eigene Ausnahmeklassen ab, wenn Ausnahmeobjekte spezifische Informationen bereitstellen sollen.
- Signalisieren Sie eine Ausnahme des Typs `ArgumentException`, wenn für einen Aufrufparameter ein fehlerhafter Wert übergeben wurde.
- Signalisieren Sie eine Ausnahme des Typs `InvalidOperationException`, wenn die gewünschte Operation aufgrund des aktuellen Objektzustands nicht durchführbar ist.
- Arbeiten Sie mit verketteten Ausnahmen (`InnerException`-Eigenschaft) – dadurch wird es möglich, den Weg einer Ausnahme zurück zu verfolgen.
- Unterlassen Sie es, Ausnahmen des Typs `NullReferenceException` oder `IndexOutOfRangeException` innerhalb von Methoden zu signalisieren.

10 Ereignisbasierte Programmierung

Eine Windows-Anwendung wird vom Betriebssystem über alles und jedes informiert, was sie betrifft. Der Strom dieser Informationen unterteilt sich in einzelne Nachrichten, von denen jede über ein Ereignis berichtet. Wahllos herausgegriffene Beispiele für solche Ereignisse wären etwa: der Start der Anwendung, das Laden des Hauptformulars, die verschiedenen Maus- und Tastaturaktionen des Benutzers, Menükommandos oder, dass ein zuvor verdeckter Teil eines Formulars der Anwendung sichtbar wird (und deshalb neu gezeichnet werden muss).

All diese Nachrichten organisiert das System in einer Nachrichtenwarteschlange und händigt sie der Anwendung eine nach der anderen aus, wobei im Allgemeinen nach dem Fifo-Prinzip (First-in-first-out) verfahren wird. (»Im Allgemeinen« deshalb, weil es auch Nachrichten gibt, die andern vorgezogen werden.) Zu diesem Zweck durchläuft die Anwendung eine Nachrichtenschleife, in der sie die Nachrichten entgegennimmt, nach Art des beschriebenen Ereignisses unterscheidet und zur Bearbeitung an die dafür zuständigen Routinen verteilt. In den Urzeiten der Windows-Programmierung musste man diese Schleife inklusive der Nachrichtenverteilung noch selbst schreiben. Unter .NET ist diese Funktionalität inzwischen standardisiert und tief im Klassendesign vergraben sowie weitgehend verkapselt, sodass der zugehörige Code nach außen hin nicht mehr in Erscheinung tritt.

10.1 Ereignisorientiertes Programmdesign

Diese Art des Programmdesigns erfordert eine etwas andere Denkweise als sie in der traditionellen Programmierung üblich war, wo ein Programm bekanntlich an verschiedenen Punkten aktiv und auf den jeweiligen Zustand bezogen bestimmte Eingaben anforderte, die es für den weiteren Ablauf der Programmlogik benötigte.

Das *ereignisorientierte Programmdesign* sieht so aus, dass ein Programm Routinen bereitstellt, die für die Behandlung der verschiedenen (für das Programm relevanten) Ereignisse zuständig sind und sich dann schlicht darauf verlässt, dass diese Routinen zum Aufruf kommen. Ein fester Ablauf, wann und in welcher Folge welche Ereignisse eintreffen, ist in diesem Ansatz zunächst einmal nicht vorgesehen – obwohl es natürlich Möglichkeiten gibt,

einen solchen Ablauf explizit und zustandsbezogen durch gezieltes Ignorieren von Ereignissen zu erzwingen.



Nach außen hin präsentiert sich ein Windows-Programm als lose Sammlung von Einsprungpunkten, die für den Programmierer unsichtbar mit einer im Verborgenen agierenden Nachrichtenschleife »verdrahtet« sind.

Ganz so unstrukturiert, wie es hier klingt, ist der Ereignisstrom allerdings auch wieder nicht. So wie das System jedem Programm garantiert, dass als erstes seine Methode `Main()` zum Aufruf kommt, garantiert es auch das Eintreffen bestimmter Ereignisse oder Ereignisfolgen zu bestimmten Gelegenheiten, beispielsweise das Ereignis `Load`, wenn ein neues Formular angezeigt wird oder die prinzipielle Ereignisreihenfolge `MouseDown`, `MouseKlick` und `MouseUp`, wenn der Benutzer mit der Maus klickt.

10.2 Codebeispiel – Monitor

Um einen Eindruck vom Geschehen zu vermitteln, stellt dieser Abschnitt eine kleine Konsolenanwendung namens *Monitor* vor, die im kleinen Maßstab simuliert, was bei einer Windows-Anwendung passiert.

Die Funktionalität des Systems ist schnell erklärt. Es gibt drei Klassen: Erstens die Testklasse `Class1`, die die Methode `Main()` stellt und je ein Objekt der anderen beiden Klassen vereinbart. Zweitens die Klasse `MessageLoop`, deren Objekte eine Nachrichtenschleife verkörpern, welche drei unterschiedliche Ereignisse signalisiert: ein `Start`-Ereignis zu Beginn, ein `CharEnter`-Ereignis, jedes Mal, wenn sie ein Zeichen von der Konsole gelesen hat. Und schließlich ein `End`-Ereignis, wenn das Zeichen X gelesen wurde und die Nachrichtenschleife zu Ende ist.

Um Ereignisse überhaupt zu signalisieren zu können, bedarf es passender Ereignisklassen, die die zum Ereignis gehörige Information repräsentieren. Die Nachrichtenschleife benutzt zwei unterschiedliche Ereignisklassen für die drei Ereignisse.

```
// Definition des Start/Stop-Ereignisses
class MonitorEvent : System.EventArgs
{
  public enum Types {Start, End};
  public Types Type;
  public MonitorEvent(Types type)
  {
    Type = type;
  }
}
```

```
// Definition des Tastaturereignisses
class MonitorCharEvent : System.EventArgs
{
  public int Character;
  public MonitorCharEvent(int character)
  {
    Character = character;
  }
}
```

Zur Signalisierung der Ereignisse ist weiterhin die Definition zweier Delegaten vonnöten:

```
// Delegaten für Ereignisroutinen
delegate void MessageFkt(MonitorEvent me);
delegate void MessageFkt1(MonitorCharEvent me);
```

Die Nachrichtenschleife vereinbart die drei Ereignisse und stellt dafür je eine Methode bereit, um die Ereignisse zu triggern. Dies könnte vom Prinzip her auch »vor Ort« (inline) passieren, der Code gibt hier aber die in der Windows-Programmierung übliche Form mit virtuellen Methoden wieder, ebenso das Benennungsschema. Die zentrale Methode der Klasse ist `Run()`. Diese Methode enthält die Schleife und versendet alle Ereignisse an potenzielle Klienten.

```
// Nachrichtenschleife
class MessageLoop
{
  public event MessageFkt Start;
  public event MessageFkt End;
  public event MessageFkt1 CharEnter;

  public void Run()
  {
    int Character;
    OnStart(new MonitorEvent(MonitorEvent.Types.Start)); // Start verkünden
    while(true)                                          // Nachrichtenschleife
    {
      Character = Console.Read();
      if (Character == 'X' )
        break;
      else
        OnCharEnter(new MonitorCharEvent(MonitorEvent.Types.Keyboard,
          Character));
    }
    OnEnd(new MonitorEvent(MonitorEvent.Types.End));  // Ende verkünden
  }
```

```
  // Routinen zum Auslösen der drei Ereignisse
  public virtual void OnStart(MonitorEvent me)
  {
    if (Start != null)
      Start(me);
  }
  public virtual void OnEnd(MonitorEvent me)
  {
    if (End != null)
      End(me);
  }
  public virtual void OnCharEnter(MonitorCharEvent mce)
  {
    if (CharEnter != null)
      CharEnter(mce);
  }
}
```

Die »Gegenstelle« von `MessageLoop` bildet die Klasse `LoopClient`, deren Objekte unter Angabe eines `MessageLoop`-Objekts konstruiert werden. `LoopClient` registriert für jedes der drei Ereignisse eine Behandlungsmethode. Die Aktion dieser Methoden beschränkt sich auf die Ausgabe einer Meldung an der Konsole:

```
class LoopClient
{
  MessageLoop messageLoop1;
  // Verdrahtung der Ereignisse = Registierung von Behandlungsmethoden
  public LoopClient(MessageLoop ml)
  {
    messageLoop1 = ml;      // Instanz der NS. bunkern
    messageLoop1.Start += new MessageFkt(MessageLoop1_Start);
    messageLoop1.End += new MessageFkt(MessageLoop1_End);
    messageLoop1.CharEnter += new MessageFkt1(MessageLoop1_CharEnter);
  }

  public void MessageLoop1_Start(MonitorEvent me)
  {
    Console.WriteLine("Start der Nachrichtenschleife signalisiert");
  }
  public void MessageLoop1_End(MonitorEvent me)
  {
    Console.WriteLine("Beenden der Nachrichtenschleife signalisiert");
  }
  public void MessageLoop1_CharEnter(MonitorCharEvent mce)
  {
    Console.WriteLine("Zeichen gelesen: {0}", (char) mce.Character);
  }
}
```

Das wäre es fast gewesen. Nun fehlt nur noch der Testcode. Er steckt der Übersichtlichkeit halber in einer eigenen Klasse:

```
class Class1
{
  [STAThread]
  static void Main(string[] args)
  {
    MessageLoop ml = new MessageLoop();  // Nachrichtenschleife
    LoopClient lc = new LoopClient(ml);  // Client der Nachrichtenschleife
    ml.Run();                            // Nachrichtenschleife starten
  }
}
```



Abbildung 10.1: Testlauf des Programms *Monitor*

Wenn Sie Schwierigkeiten haben, den Ablauf genau zu verstehen, lohnt es, das Programm mit den Debugger schrittweise zu analysieren. Am `Read()`*-Aufruf angelangt, erwartet das Programm eine Eingabe, die Sie am besten mit einem »X« und dann Return abschließen. (Dass die Eingabe über den Zeilenpuffer läuft und das Programm deshalb ein Return erwartet, ist ein Schönheitsfehler, der der Demonstration des Prinzips aber keinen Abbruch tun sollte.)*

:-) TIPP

10.2.1 Übung

Wenn Sie Lust haben, sich das Programm interaktiv zu erarbeiten, setzen Sie sich doch das Ziel, es dahingehend zu erweitern, dass die Nachrichtenschleife Groß- und Kleinbuchstaben unterscheidet und als unterschiedliche Ereignisse meldet. Kleiner Tipp: Sie können das bestehende Ereignis `MonitorCharEvent` um eine Typinformation erweitern, die der Client auswerten kann, aber nicht muss.

11 Einsatz der .NET-Basisklassen

Die in Teil 2 bisher vorgestellten .NET-Basisklassen hatten mehr oder weniger alle mit der Implementierung der grundlegenden Datentypen sowie der Sprache C# als .NET-Sprache zu tun. Als Laufzeitumgebung, die als eigenständige Schicht zwischen Anwendung und Betriebssystem liegt, ist .NET aber mehr als nur eine reine Sprachumgebung. Um der Forderung nach Typ- und Ausführungssicherheit für verwalteten Code gerecht zu werden, muss .NET tatsächlich eine Verkapselung der gesamten Sicht einer .NET-Anwendung auf seine Umgebung vornehmen. Dazu gehört eine Anbindung an die bestehende COM-Welt, an alle Ein- und Ausgabemechanismen sowie an das jeweilige Betriebssystem selbst. Kurzum, der Programmierer findet in Form der .NET-Klassenhierarchie eine komplett objektorientierte und überwiegend auch homogene Landschaft vor, bei der wirklich alles »ins Bild passt« – der Dateizugriff ebenso wie die eigentliche GUI-orientierte Windows-Programmierung.

In der bestehenden .NET-Klassenhierarchie findet sich vom Prinzip her so ziemlich alles, was für die Programmierung von konsolenbasierten und Windows-orientierten .NET-Anwendungen erforderlich ist. Einzig, wenn es um die Anbindung an bestehenden, weder .NET- noch COM-konformen Code geht, hält sich .NET bedeckt. Obwohl hierfür Mechanismen vorhanden sind, erweisen sich die Wege letztlich jedoch als ziemlich steinig (vgl. Teil 5).

Dieses Kapitel stellt verschiedene wichtige Klassen aus der .NET-Klassenhierarchie vor, die sowohl für die konsolenbasierte wie auch die Windows-orientierte Programmierung eine Rolle spielen, darunter die Klassen für den Umgang mit Strings, Auflistungen, Dateien, Dateisystem und Registrierung. Einen praxisnahen Überblick über die .NET-Klassen, die rein für die Windows-orientierte Programmierung zuständig und als solche im Namensraum `System.Windows.Forms` zusammengefasst sind, finden Sie im Teil 3.

11.1 Strings

Mit der Bereitstellung der vollständig verkapselten und von der weiteren Vererbung ausgeschlossenen Klasse `System.String` schafft .NET den notwendigen Unterbau für den integrierten C#-Datentyp `string`. Die Unterstützung,

die in der .NET-Klassenhierarchie für diesen Datentyp vorhanden ist, kann sich wahrhaft sehen lassen und verwöhnt selbst eingefleischte VB-Programmierer. Obwohl die Klasse `String` vom Auftreten her eine Art Zwischenstellung zwischen den Verweistypen und Werttypen einzunehmen scheint, wird der Datentyp – aller Besonderheiten, die ihn als Werttyp erscheinen lassen, zum Trotz – letztlich doch zu den Verweistypen gezählt.

Gerade wegen dieser Sonderstellung ist nicht nur viel Falsches über diesen Datentyp zu lesen, auch sein Einsatz ist nicht immer von den richtigen Überlegungen geleitet. Generell gilt nämlich die Regel, dass die Manipulation langer Strings wesentlich effizienter mit der Klasse `StringBuilder` als mit der Klasse `String` zu bewerkstelligen ist, während `String` bei den Stringvergleichen und im Komfort besser abschneidet.

Zunächst einmal sollte zwischen dem Stringobjekt als Instanz der Klasse `System.String` und dem zugeordneten *Stringwert* unterschieden werden. Betrachten Sie die beiden folgenden Vereinbarungen:

```
string s1 = null; // entspricht der automatischen Initialisierung
                  // von string-Datenfeldern
string s2 = "";   // leerer String
if (s1 == s2)     // immer false
```

Die Werte von `s1` und `s2` sind tatsächlich unterschiedlich, weil die Klasse den leeren String als regulären Stringwert betrachtet (und wie es sich gehört, auch eine Konstante namens `String.Empty` dafür bereitstellt, die aber nur für Compiler eine Rolle spielt). Mithin findet die automatische Initialisierung von `string`-Datenfeldern mit dem Wert `null` statt, und nicht mit `Empty` – ein Unterschied, der sich wohl ausnutzen lässt, aber auch häufig genug Ursache schwer auffindbarer Fehler ist.

Die .NET-Speicherverwaltung packt Stringwerte nicht bunt gemischt mit anderen Objektwerten auf den Heap, sondern organisiert diese in einem eigenen Pool, der allein Stringwerten vorbehalten ist. Der Pool ist seinerseits natürlich Teil des Heaps. In diesem Pool werden wiederum *nur unterschiedliche* Stringwerte gespeichert. Ein im Pool gespeicherter Stringwert ist unveränderlich und wird durch die Freispeicherverwaltung gelöscht, wenn kein Stringobjekt mehr darauf verweist. Sind die Stringwerte zweier Stringobjekte gleich, verweisen sie tatsächlich auf denselben Wert im Stringpool. Das spart einerseits Speicherplatz und macht andererseits den Stringvergleich auf der Basis von `Equals()` durch die Operatoren == und != sowie ordinale Stringvergleiche (ohne Beachtung einer Sortierordnung) generell sehr effizient.

Die Aussage »die Stringzuweisung ist eine Wertzuweisung und erzeugt eine Wertkopie des Rechtswerts« ist deshalb nur halb richtig. Die Wertkopie ist nicht in jedem Fall echt; faktisch macht dies aber keinen Unterschied, da die

Stringwerte im Pool ja unveränderlich sind. Jegliche Wertmanipulation einer Stringvariablen führt dazu, dass sie auf einen anderen Stringwert im Pool verweist, ohne Seiteneffekt auf andere Stringvariablen, die gegebenenfalls weiterhin auf den alten Wert verweisen.

Mit Blick auf Stringmanipulationen erweist sich diese an sich recht elegante Verwaltung jedoch als ziemlich schwerfällig – zumal, wenn längere Stringwerte im Spiel sind. Das liegt daran, dass jeder neue Stringwert eine »Wertkopie« darstellt und immer nur als Ganzes manipuliert werden kann.

Für Fälle, in denen die Laufzeit von Stringmanipulationen einen wesentlichen Faktor darstellt, enthält die .NET-Klassenhierarchie die Klasse `StringBuilder`. Objekte dieser Klasse reservieren einen Puffer für den Stringwert und manipulieren diesen auf der Basis von Indizes vor Ort – ganz so, wie man es von den Programmiersprachen C/C++ und Delphi her gewohnt ist.

11.1.1 System.String

Die statischen Methoden der Klasse `String` diversifizieren den Funktionsumfang der statischen Operatoren +, == und = etwas. Zugleich geben sie einen gewissen Einblick in das Wesen der String-Verwaltung unter .NET (vgl. insbesondere `IsInterned()`).

Tabelle 11.1: Statische Methoden der Klasse `String`

Statische Methode	Beschreibung
`int Compare()`	Diese in sechs Varianten verfügbare Methode vergleicht zwei Strings. Die einzelnen Überladungen unterscheiden sich darin, dass sie den Vergleich auch für Teile der beiden Strings, ohne Beachtung der Groß/Kleinschreibung, mit Beachtung einer kulturspezifischen Sortierordnung (`CultureInfo`-Klasse) zulassen. Der Rückgabewert ist -1, wenn der Wert des ersten Parameters kleiner, 0, wenn er gleich und 1, wenn er größer als der des zweiten Parameters ist.
`int CompareOrdinal()`	Vergleicht zwei Strings oder Teile davon unter Verwendung der durch die Repräsentation implizierten Sortierordnung. Zum Rückgabewert siehe `Compare()`.
`string Concat()`	Verkettet Strings, die als `string`- oder `object`-Parameter angeben sind. Zwei der neun Varianten erwarten ein `params`-Array, die restlichen deklinieren die Verkettung von 1 bis 4 `object`- sowie 2 bis 4 `string`-Parametern durch.

Tabelle 11.1: Statische Methoden der Klasse `String` (Forts.)

Statische Methode	Beschreibung
`string Copy()`	Liefert eine Kopie des angegebenen `string`-Parameters.
`string Format()`	Formatiert den angegebenen String unter Beachtung von Formatierungsinformationen (vgl. folgender Abschnitt).
`string Intern()`	Das .NET.Laufzeitsystem verwendet einen internen Pool, um mehrfache Referenzen auf ein- und denselben String-Wert durch eine einzige Wertkopie zu bedienen. Wird einer `string`-Variable ein Wert zugewiesen, der bereits in diesem Pool vorkommt, erhält die Variable eine Referenz auf diesen Wert. Die Methode `Intern()` legt den angegebenen Wert erforderlichenfalls im internen Pool an und liefert eine Referenz darauf.
`string IsInterned()`	Wie `Intern()`, mit dem Unterschied, dass die Methode den leeren String zurückliefert, wenn der angegebene Wert im internen Pool nicht vorhanden ist. Damit ermöglicht sie einen Test, ob ein bestimmter Wert im Pool vorhanden ist, ohne den getesteten Wert in den Pool zu schreiben.

Interessant sind die allerdings nur selten verwendeten Methoden `Intern()` und `IsInterned()`. Sie geben darüber Aufschluss, wie Strings in .NET implementiert sind. Betrachten Sie hierzu den folgenden Code aus dem Beispielprojekt *StringOps*:

```
string s = "Rudi";
string s1 = "Ru"+"di";
string s2 = "Rudi";
string s3 = string.IsInterned(s2.Substring(0,4)); // Rudi
string s4 = string.IsInterned(s2.Substring(1,2)); // Leerer String,
                                                   // da nicht im Pool
string s5 = string.Intern(s2.Substring(1,2));     // "ud"
Console.WriteLine("s                          :{0}", s);
Console.WriteLine("s1                         :{0}", s1);
Console.WriteLine("s2                         :{0}", s2);
Console.WriteLine("s3                         :{0}", s3);
Console.WriteLine("s4                         :{0}", s4);
Console.WriteLine("s5                         :{0}", s5);
Console.WriteLine("s == s2                    :{0}", s == s2);
Console.WriteLine("(object)s == (object)s2   :{0}",
  (object)s == (object) s2  );
Console.WriteLine("(object)s1 == (object)s2  :{0}",
  (object)s1 == (object) s2);
Console.WriteLine("(object)s == (object)s5   :{0}",
  (object)s == (object) s5);
```

Die Ausgaben dieses Codes sehen so aus:

```
s                          :Rudi
s1                         :Rudi
s2                         :Rudi
s3                         :Rudi
s4                         :
s5                         :ud
s == s2                    :True
(object)s == (object)s2    :True
(object)s1 == (object)s2   :True
(object)s == (object)s5    :False
```



Es ist prinzipiell nicht möglich, die im String-Pool des Laufzeitsystems gespeicherten Stringwerte zu ändern. Um den Wert einer Stringvariablen zu ändern, muss dieser explizit ein neuer Wert zugewiesen werden. Der alte Wert fällt dann gegebenenfalls der Freispeicherverwaltung zum Opfer, sofern kein anderes Stringobjekt denselben Stringwert aufweist. Somit kann es passieren, dass selbst vermeintlich »kleine« Operationen wie das Entfernen oder Hinzufügen einzelner Zeichen am Ende langer Stringwerte einiges an Laufzeit in Anspruch nehmen. Für die Manipulation längerer Stringwerte empfiehlt es sich, auf die Klasse `StringBuilder` *auszuweichen. Diese Klasse stattet ihre Instanzen mit Operationen aus, die den Objektwert an sich manipulieren. (Ein aussagekräftiges Codebeispiel finden Sie im Abschnitt »string vs. StringBuilder«, Seite 175.)*

Stringobjekte verfügen über einen reichen Satz an Instanzmethoden, denen jedoch gemein ist, dass sie den Wert des Objekts an sich unverändert lassen und das jeweilige Ergebnis als Funktionswert bereitstellen. Tabelle 11.2 gibt einen Überblick.

Tabelle 11.2: Instanzmethoden von Stringobjekten

Methode	Beschreibung
`string Clone()`	Liefert eine Kopie des Objekts mit einer weitere Referenz auf denselben Stringwert. (Da Stringwerte in einem Pool gespeichert werden und nicht manipulierbar sind, entspricht dies faktisch einer Wertkopie.)
`int CompareTo()`	-1, wenn der Wert des Objekts kleiner, 0, wenn er gleich und 1, wenn er größer als der angegebene String ist. In der zweiten Variante nimmt die Methode einen `object`-Parameter entgegen, den sie versucht, per Typumwandlung in einen `string`-Wert zu verwandeln. Misslingt diese Umwandlung, kommt es zu einer Ausnahme.
`void CopyTo()`	Kopiert den Stringwert des Objekts ab dem angegebenen Index in das angegebene `char[]`-Objekt. Weitere Parameter beschreiben den Zielindex und die Anzahl der zu kopierenden Zeichen.

Tabelle 11.2: Instanzmethoden von Stringobjekten (Forts.)

Methode	Beschreibung
`bool EndsWith()`	Liefert `true`, wenn der Stringwert des Objekts mit der angegebenen Zeichenfolge endet.
`CharEnumerator GetEnumerator()`	Liefert ein Objekt, das es erlaubt, den Stringwert zeichenweise aufzuzählen. Damit sind Konstrukte nach dem Muster `foreach(char c in MyString)` möglich.
`int IndexOf()`	Liefert den Startindex des ersten Vorkommens des angegebenen Strings (oder Einzelzeichens) im Stringwert des Objekts. Weitere Parameter ermöglichen es, die Suche auf einen Teil des Stringwerts zu beschränken. Das Ergebnis ist -1, wenn es kein Vorkommen gibt.
`int IndexOfAny()`	Liefert den Startindex des ersten Vorkommens eines beliebigen Zeichens aus dem angegebenen `char[]`-Objekt im Stringwert des Objekts. Das Ergebnis ist -1, wenn es kein Vorkommen gibt.
`string Insert()`	Fügt den angegebenen Stringwert an der angegebenen Position in den Stringwert des Objekts ein und liefert das Ergebnis als neue `string`-Instanz zurück. Das Objekt selbst behält seinen ursprünglichen Wert.
`int LastIndexOf()`	Liefert den Startindex des letzten Vorkommens des angegebenen Strings (oder Einzelzeichens) im Stringwert des Objekts. Weitere Parameter ermöglichen es, die Suche auf einen Teil des Stringwerts zu beschränken. Das Ergebnis ist -1, wenn es kein Vorkommen gibt.
`int LastIndexOfAny()`	Liefert den Startindex des letzten Vorkommens eines beliebigen Zeichens aus dem angegebenen `char[]`-Objekt im Stringwert des Objekts. Das Ergebnis ist -1, wenn es kein Vorkommen gibt.
`string PadLeft()`	Liefert eine rechtsbündige Kopie des Stringwerts am Anfang aufgefüllt mit dem angegebenen Füllzeichen. Der Stringwert wird nicht beschnitten, wenn die Längenangabe zu kurz ist. Das Objekt selbst behält seinen ursprünglichen Wert.
`string PadRight()`	Liefert eine linksbündige Kopie des Stringwerts am Ende aufgefüllt mit dem angegebenen Füllzeichen. Der Stringwert wird nicht beschnitten, wenn die Längenangabe zu kurz ist. Das Objekt selbst behält seinen ursprünglichen Wert.
`string Remove()`	Liefert eine Kopie des Stringwerts, bei der die angegebene Anzahl an Zeichen, ab der angegebenen Position fehlen. Das Objekt selbst behält seinen ursprünglichen Wert.

Methode	Beschreibung
`string Replace()`	Liefert eine Kopie des Stringwerts, bei der alle Vorkommen des angegebenen Suchausdrucks (Stringwert oder Zeichen) durch den angegebenen Ersatzausdruck (Stringwert oder Zeichen) ausgetauscht sind. Das Objekt selbst behält seinen ursprünglichen Wert.
`string[] Split()`	Unterteilt den Stringwert des Objekts unter Beachtung des angegebenen Trennzeichens in Teilstrings und liefert diese als Array zurück.
`bool StartsWith()`	Liefert `true`, wenn der Stringwert des Objekts mit der angegebenen Zeichenfolge beginnt.
`string SubString()`	Liefert einen Teilstring des Stringwerts des Objekts. Das Objekt selbst behält seinen ursprünglichen Wert.
`char[] ToCharArray()`	Liefert eine Kopie des Stringwerts als Zeichenarray.
`string ToLower()`	Liefert eine Kopie des Stringwerts in Kleinschreibung. Das Objekt selbst behält seinen ursprünglichen Wert.
`string ToString()`	Liefert den Stringwert des Objekts. Die einparametrige Variante der Methode ermöglicht die Angabe eines `IFormatProvider`-Objekts, das die kulturspezifische Formatierung des Strings ermöglicht.
`string ToUpper()`	Liefert eine Kopie des Stringwerts in Großschreibung. Das Objekt selbst behält seinen ursprünglichen Wert.
`string Trim()`	Liefert eine Kopie des Stringwerts ohne führende und endende Leerzeichen. Das Objekt selbst behält seinen ursprünglichen Wert.
`string TrimEnd()`	Liefert eine Kopie des Stringwerts ohne endende Leerzeichen. Das Objekt selbst behält seinen ursprünglichen Wert.
`string TrimStart()`	Liefert eine Kopie des Stringwerts ohne führende Leerzeichen. Das Objekt selbst behält seinen ursprünglichen Wert.

Tabelle 11.2: Instanzmethoden von Stringobjekten (Forts.)

Stringformatierung

Die Stringrepräsentation von Datentypen war schon immer ein recht unerfreuliches Kapitel – zumal, wenn es darum ging, bestimmte Formatvorgaben zu beachten. Unter .NET ist dies nicht anders, obwohl die von `object` vererbte und von den einzelnen Datentypen im Allgemeinen überschriebene virtuelle Methode `ToString()` einen weitgehend standardisierten Ansatz für den »Hausgebrauch« bietet. Da sich diese Methode für eigene Datentypen überschrei-

ben lässt, erhält man so eine standardmäßige Stringrepräsentation, die unter anderem auch von der Methode `String.Format()` adaptiert wird.

Was die konkrete Stringformatierung unter Beachtung von Formatvorschriften betrifft, kommt auch .NET weder ohne leidige Formatzeichen noch ohne die bei Missbrauch allfälligen Ausnahmen aus. Der gewählte Ansatz ist allerdings um einiges übersichtlicher als das, was man von anderen Programmsprachen her gewohnt ist.

Die für Stringformatierung allein zuständige Methode `String.Format()` gibt es in zwei Varianten:

```
static string Format(string format, params object[] args);
static string Format(IFormatProvider p, string fmt, params object[] args);
```

Die erste, harmlosere Variante ist vom Prinzip her eine alte Bekannte, denn sie weist das gleiche Aufrufschema wie die zweiparametrige `params`-Variante der Methode `Console.Write()` auf. Tatsächlich delegiert die Implementierung von `Console.Write()` die gesamte Formatierungsarbeit schlicht an die Methode `String.Format()` unter Weiterreichung der Parameter.

Wie von `Console.Write()` her bekannt, interpretiert die Methode in dem String `format` in geschweifte Klammern gesetzte Ausdrücke, die aus einem Argumentplatzhalter für ein Element des Arrays `args` und optionalen Formatierungsinformationen bestehen. Die allgemeine Form eines solchen Ausdrucks ist:

```
{N [, M ][: TypFormat ]}
```

- *N* steht für eine ganze Dezimalzahl und gibt den Index des aus dem Array `args` einzusetzenden Elements an. (Wie für C#-Arrays üblich beginnt die Zählung bei 0.)
- *M* ist optional und steht für eine ganze Dezimalzahl, die die Mindestbreite der resultierenden Zeichenfolge in Zeichen ausdrückt. Ist die Zahl größer als 0, richtet die Methode die Darstellung des zu formatierenden Wertes rechts aus, Werte kleiner 0 bewirken eine linksbündige Ausrichtung.
- *TypFormat* ist eine im Allgemeinen auf den Datentyp des korrespondierenden `args`-Elements gemünzte, datentypspezifische Formatzeichenfolge, die das Darstellungsformat näher beschreibt. Das folgende Codebeispiel demonstriert verschiedene Datenstellungsformate für Datums-/Zeitwerte, Zahlenwerte und Währungsangaben. (Vollständige Informationen über die verfügbaren Zeit/Datumsformate sowie numerischen Formate finden Sie in der Online-Dokumentation.)

Tatsächlich ruft die erste Variante intern die zweite auf und übergibt dieser im ersten Parameter den Wert der Eigenschaft `CultureInfo.CurrentCultureInfo`. Damit nimmt die erste Variante alle Formatierungen grundsätzlich unter Beachtung der jeweils für das System geltenden Kulturinformation vor. (Genauer: Die über diese Eigenschaft abrufbaren Informationen decken sich im Allgemeinen mit den Einstellungen des System-Dialogs SYSTEMSTEUERUNG/LÄNDEREINSTELLUNGEN. Da die Eigenschaft an sich aber auf Thread-Ebene angesiedelt ist, kann ihr Wert für einzelne Threads auch von diesem Vorgabewert abweichen.)

Die zweite Variante erwartet im ersten Parameter ein Objekt, das über die Schnittstelle `IFormatProvider` verfügt. In der .NET-Klassenhierarchie finden sich dafür die Klassen `CultureInfo`, `DateTimeFormatInfo` und `NumberFormatInfo`. Durch Übergabe eines geeignet konstruierten und initialisierten `CultureInfo`-Objekts, das seinerseits (`get`/`set`-)Eigenschaften mit den Typen `DateTimeFormatInfo` und `NumberFormatInfo` enthält, lässt sich die gewünschte Formatierungsart aus einer großen Auswahl der unterschiedlichsten Formatierungsmuster zusammenstellen. Natürlich können im Einzelfall auch nur geeignete initialisierte `DateTimeFormatInfo`- oder `NumberFormatInfo`-Objekte übergeben werden. Hier ein Beispiel:

```
using System.Globalization;
...
CultureInfo cultureInfo = new CultureInfo ("fr-FR");
cultureInfo.DateTimeFormat.TimeSeparator = "&";
cultureInfo.NumberFormat.NumberDecimalDigits = 5;
string t5 = String.Format(cultureInfo,
  "Datum/Zeit: {0:s}, Festkomma: {1,10:f}", DateTime.Now, Math.PI);
Console.WriteLine(t5);
```

Die Ausgabe lautet:

```
Datum/Zeit: 2002-08-09T14:15:28, Festkomma:    3,14159
```

Wenn Sie nur ein bestimmtes Zahlenformat verwenden wollen, beispielsweise die Ausgabe von Tausenderstellen für Ganzzahlen, instanziieren Sie nur ein `NumberFormatInfo`-Objekt und schneiden dieses geeignet zu:

```
NumberFormatInfo nfi = new NumberFormatInfo();
ni.NumberDecimalDigits=0; // Standardwert 2 ist in Systemsteuerung gesetzt
string s = String.Format( nfi, "Dateilänge: {0:n} Byte ", myFile.Length);
Console.WriteLine(s);
```

Die Ausgabe hat dann folgendes Muster:

```
Dateilänge: 10.124.123 Byte
```

Weiterhin besteht die auch Möglichkeit, eigene Formatierungsschemata sowohl für benutzerdefinierte als auch integrierte oder .NET-Datentypen zu verwirklichen. Dazu ist die Implementierung zweier Schnittstellen erforder-

lich, von denen jede die Definition einer Methode verlangt. Wie erwartet, ist dies zunächst die Schnittstelle `IFormatProvider`, die wie folgt definiert ist.

```
interface IFormatProvider
{
  object GetFormat(Type t);
}
```

Eine Klasse, die diese Schnittstelle implementiert, ist schnell zusammengezimmert – allerdings auf Kredit:

```
class MyFormatInfo: IFormatProvider
{
  public object GetFormat(Type t)
  {
    return new MyCurrencyFormat();
  }
}
```

Tatsächlich ist der Datentyp `object` nicht gerade sehr aussagekräftig. Setzt man eine Instanz dieser Klasse aber in einen `String.Format()`-Aufruf ein, enthüllt der Debugger für den Parameter `t` den Datentyp `ICustomFormatter`. Mit anderen Worten, die noch zu implementierende Klasse `MyCurrencyFormat()` muss Ableitung der Schnittstelle `ICustomFormatter` sein. Die Definition dieser Schnittstelle lautet:

```
interface ICustomFormatter
{
  string Format(string format, object arg, IFormatProvider i);
}
```

Die geforderte Methode `Format` wird für *jeden* Argumentplatzhalter aufgerufen und erhält die Formatzeichenfolge *`TypFormat`*, das betroffene Element aus dem Array *`args`* sowie die für ihren Aufruf verantwortliche `IFormatProvider`-Instanz.

Die folgende Implementierung der Schnittstelle durch die Klasse `MyCurrencyFormat` generiert ein eigenes Währungsformat und löst dabei das Problem, dass für die Konsolenausgabe kein _-Zeichen verfügbar ist. (Es wird im ANSI-Code auf das Fragezeichen abgebildet.)

```
class MyCurrencyFormat : ICustomFormatter
{
  public string Format(string format, object arg, IFormatProvider i)
  {
    if (format != null && format.ToLower() == "c")  // Währungsformat
      return String.Format("{0:f} Euro", arg);
    else                          // Rest auf Standardformat abbilden
      return String.Format(("{0:"+format+"}"), arg);
  }
}
```

Das folgende Codebeispiel zeigt unter anderem den praktischen Einsatz der beiden Klassen.

Codebeispiel – Strings formatieren

Das Projekt *StringFormat* demonstriert eine Auswahl von Stringformatierungen für verschiedene Datentypen und Formatierungsausdrücke sowie den Zusammenhang zwischen den Überladungen von `String.Format()` und `Console.WriteLine()`. Die Konsolenausgabe des Programms ist weitgehend selbsterklärend (Abbildung 11.1).

Beachten Sie bei der Analyse der Formatierungszeichenfolgen, dass beim Datumsformat das Formatzeichen `m` für die Minute steht und `M` für den Monat. Weiterhin steht `H` für das übliche 24-Stundenformat und `h` für das englische 12-Stundenformat. Weiterhin spielt auch die Anzahl der Wiederholungen der Formatzeichen eine Rolle. Steht ein Zeichen nur einmal in Folge da, werden einstellige Zahlwerte durch ein führendes Leerzeichen aufgefüllt, ansonsten durch eine 0. Bei `MMM` wird der Monat abgekürzt, bei `MMMM` hingegen ausgeschrieben; `yy` steht für die zweistellige Jahreszahl, `yyy` oder `yyyy` für die vierstellige.

```
// Verschiedene Datumsformate generieren und ausgeben
string t0 = String.Format("{0,-40} {1,26:dd.MM.yy hh:mm:ss}",
  "Kurzes Datum, englische Zeit", DateTime.Now);
string t1 = String.Format(new System.Globalization.CultureInfo("it-IT"),
  "{0,-40} {1,26:dd.MMMM.yy HH:mm:ss}",
  "Langes Datum, italienische Kultur", DateTime.Now);
string t2 = String.Format("{0,-40} {1,26:dd.MMM.yy HH:mm:ss}",
  "Mittleres Datum, deutsche Zeit", DateTime.Now);
string t3 = String.Format("{0,-40} {1,26:dd.MMM.yyyy HH:mm:ss}",
  "Langes Datum, deutsche Zeit", DateTime.Now);
string t4 = String.Format("{0,-40} {1,26:dd.MMMM.yyyy HH:mm:ss}",
  "Vollständiges Datum, deutsche Zeit", DateTime.Now);

Console.WriteLine("Datums/Zeit-Formate linksbündig (26) rechtsbündig (40");
Console.WriteLine("=================================================");
Console.WriteLine("{0}\n{1}\n{2}\n{3}\n{4}\n", t0, t1, t2, t3, t4);

// Verschiedene Zahlenformate ausgeben
Console.WriteLine("Zahlenformate rechtsbündig auf 30 Zeichen");
Console.WriteLine("=====================================");
Console.WriteLine("Hexadezimal                {0,30:x}", 100023);
Console.WriteLine("Festkomma                  {0,30:f}", 100023);
Console.WriteLine("Dezimal                    {0,30:d}", 100023);
Console.WriteLine("Wissenschaftlich           {0,30:e}", 100023);
Console.WriteLine("Währung                    {0,30:c}", 100023);

// Eigenes Format ausgeben
Console.WriteLine("\nEigene Formate rechtsbündig auf 30 Zeichen");
Console.WriteLine("=====================================");
string s = String.Format(new MyFormatInfo(),"Währung (eigenes Format)
{0,30:c}", 100023);
```

```
string s1 = String.Format(new MyFormatInfo(),"Festkomma (eigenes Format)
{0,30:f}", 100023);
Console.WriteLine(s);
Console.WriteLine(s1);
```

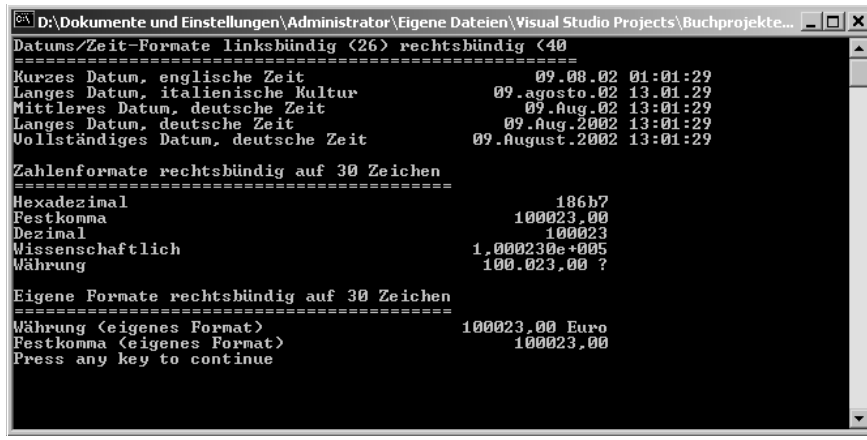


Abbildung 11.1: Ausgabe der Anwendung `StringFormat`. Das Währungssymbol € ist leider nicht im ANSI-Zeichensatz enthalten, deshalb erscheint es in einer Konsolenanwendung regulär als Fragezeichen. Abhilfe schafft ein eigenes Währungsformat.

11.1.2 StringBuilder

Die direkt von `object` abstammende Klasse `System.Text.StringBuilder` implementiert Strings pufferorientiert und als »echte« Verweistypen, was bei der Manipulation langer Strings erhebliche Laufzeitvorteile bringt. Da C# den integrierten Datentyp `string` jedoch auf `System.String` basiert, ist der Umgang mit `StringBuilder` etwas weniger komfortabel – nicht zuletzt auch aufgrund der recht mageren Ausstattung mit Operationen. Zudem taugt der Datentyp nicht für Konstanten und bringt für kurze Stringwerte eher Nachteile. (Ein aussagekräftiges Codebeispiel für lange Strings finden Sie aber im Abschnitt »String vs. StringBuilder«, Seite 175.)

Für die Instanziierung der Klasse `StringBuilder` stehen sechs Konstruktoren bereit

```
public StringBuilder();                 // Initialisiert Objekt mit ""
public StringBuilder(int cap);          // Weist Puffer der Länge cap zu,
                                        // initialisiert Objekt mit ""
public StringBuilder(string s);         // Initialisiert Objekt mit s
public StringBuilder(int cap, int max); // Weist Puffer der Länge cap zu
                                        // und erlaubt eine maximale
                                        // Pufferlänge von max
public StringBuilder(int cap, string s);// Weist Puffer der Länge cap zu,
                                        // initialisiert Objekt mit s
public StringBuilder(string s, int start, int len, int cap);   // Weist
                                        // Puffer der Länge cap zu und
                                        // initialisiert Objekt mit einem
                                        // Teilstring von s
```

Interessant ist der `int`-Parameter `cap`: Er ermöglicht es, eine anfängliche Pufferkapazität für das Objekt vorzugeben. Reicht diese Kapazität für den gegebenenfalls angegebenen Initialisierungsstring oder später für eine Stringoperation nicht mehr aus, verdoppelt das Objekt die Kapazität (auch mehrmals) automatisch und kopiert dann den bereits gespeicherten Stringwert in einen neuen Puffer dieses Umfangs. Ist keine anfängliche Pufferkapazität vorgegeben, verwendet das Objekt sukzessiv die Werte 32, 64, 128 … bis zum Wert der Eigenschaft `MaxCapacity` (unter Windows 2000 und XP: 2 147 483 647). Beträgt die bei Konstruktion angegebene Pufferkapazität 200, erweitert das Objekt den Puffer auf die Längen 400, 800, 1600 usw.

Operationen

Für den zeichenweisen Zugriff auf den Stringwert stellt die Klasse einen Indexer bereit – viel mehr jedoch nicht. Da die Operationen `Append()`, `AppendFormat()`, `Insert()`, `Remove()` und `Replace()` den Stringwert an sich ändern, muss gegebenenfalls die von `object` vererbte Methode `MemberwiseClone()` eingesetzt werden, um Operationen zu verwirklichen, bei denen der Ausgangsstring erhalten bleiben soll.

Die Tabellen 11.3 und 11.4 stellen die Eigenschaften und Methoden der Klasse im Überblick vor.

Tabelle 11.3: Eigenschaften der `StringBuilder`-Klasse

Eigenschaft	Beschreibung
`int Capacity {get; set;}`	Aktuelle Puffergröße – die Zahl der Unicode-Zeichen, die ohne (automatische) Erweiterung des Puffers gespeichert werden können.
`int Length {get; set;}`	Aktuelle Länge des Stringwerts. Wird der Wert Eigenschaft vergrößert, verlängert das Objekt den Stringwert und füllt ihn am Ende mit Null-Zeichen (`'\0'`) auf. Andernfalls wird der Stringwert am Ende abgeschnitten.
`int MaxCapacity {get;}`	Maximale Puffergröße (unter Windows 2000 und XP: 2 147 483 647). Kann bei Konstruktion einer Instanz auf einen kleineren Wert gesetzt werden.

Tabelle 11.4: Methoden einer `StringBuilder`-Instanz

Methode	Beschreibung
`StringBuilder Append()`	Erweitert den Stringwert des Objekts am Ende um eine Stringrepräsentation des angegebenen Werts. (Ruft die Methode `ToString()` des jeweiligen Datentyps auf.)
`StringBuilder AppendFormat()`	Erweitert den Stringwert des Objekts am Ende um eine formatierte Stringrepräsentation des angegebenen Werts. (Ruft die Methode `ToString()` des jeweiligen Datentyps auf.) Detaillierte Informationen zur Stringformatierung finden sich im vorangehenden Abschnitt.
`int EnsureCapacity()`	Stellt sicher, dass das Objekt Stringwerte bis zur angegebenen Länge aufnehmen kann und erweitert dafür gegebenenfalls den bestehenden Stringpuffer auf die geforderte Kapazität. Gibt den (neuen) Wert der Eigenschaft `Capacity` zurück.
`StringBuilder Insert()`	Fügt eine Stringrepräsentation des angegebenen Werts an der angegebenen Position in den Stringwert des Objekts ein. (Ruft dazu die Methode `ToString()` des jeweiligen Datentyps auf).
`StringBuilder Remove()`	Schneidet einen Teilstring (beschrieben durch Startindex und Länge) aus dem Stringwert heraus und verkürzt ihn damit.
`StringBuilder Replace()`	Ersetzt alle Vorkommen des angegebenen Suchausdrucks (Stringwert oder Zeichen) durch den angegebenen Ersatzausdruck (Stringwert oder Zeichen) im Stringwert des Objekts. Wahlweise kann die Operation auch auf ein Teilstück (beschrieben durch Startindex und Länge) des Stringwerts beschränkt werden.

Codebeispiel – Teilstrings

Um einen Teilstring aus einem `StringBuilder`-Objekt zu gewinnen, gibt es verschiedene Möglichkeiten, die je nach Situation Vor- und Nachteile aufweisen können (vgl. Beispielprojekt *StringBuilder*):

1. Die folgende Methode stellt die allgemeinste Fassung der Operation dar. Sie erhält den Ausgangswert und baut den Teilstring zeichenweise in einem neuen `StringBuilder`-Objekt auf:

```
StringBuilder SBPart1(StringBuilder sb, int start, int len)
{
  StringBuilder s = new StringBuilder(len);
  for (int i = start; i < start + len; i++)
    s.Append(sb[i]);
  return s;
}
```

2. Die folgende Methode nimmt den Umweg über `System.String`:

```
StringBuilder SBPart2(StringBuilder sb, int start, int len)
{
  StringBuilder s =
  new StringBuilder(sb.ToString().Substring(start, len));
  return s;
}
```

3. Die folgende Methode nimmt den Verlust des Ausgangswerts in Kauf:

```
StringBuilder SBPart3(StringBuilder sb, int start, int len)
{
  sb.Length = start+len;              // hinten abschneiden
  sb.Remove(0, sb.Length - len);      // vorne abschneiden
  return sb;
}
```

Hier der Testcode für die Methoden:

```
StringBuilder s = new StringBuilder("12345678901234567890123456789O");

Console.WriteLine("{0}\n{1}", s, SBPart1(s, 5, 10));
Console.WriteLine("{0}\n{1}", s, SBPart2(s, 5, 10));
Console.WriteLine("{0}\n{1}", s, SBPart3(s, 5, 10));
```



Abbildung 11.2: Ausgabe des Programms `StringBuilder`

11.2 Collection-Klassen

In der .NET-Klassenhierarchie – speziell im Namensbereich `System.Windows.Forms`, aber auch in anderen Namensbereichen oder als `public`-Datentypen von Klassen – finden sich eine Unmenge von Klassen, deren Bezeichner das Suffix »Collection« trägt. Man bezeichnet diese Klassen als Auflistungsklassen und ihre Objekte schlicht als Auflistungen – prominente Beispiele wären etwa die Klassen `StringCollection`, `ImageList.ImageCollection` und `Form.ControlCollection`.

11.2.1 Schnittstellengrundlage von Auflistungsklassen

Viele, aber bei weitem nicht alle dieser Klassen sind von der abstrakten Basisklasse `CollectionBase` abgeleitet. Was *alle* diese Klassen verbindet und auch den Umgang mit ihren Objekten vereinheitlicht, ist die Tatsache, dass ihre Vererbungslisten die drei Schnittstellenklassen `ICollection`, `IList` und `IEnumerable` als Basis benennen. Die Klassen stellen damit gezwungenermaßen einen gemeinsamen Satz an Methoden bereit, der ihren Einsatz zu einer recht intuitiven Angelegenheit werden lässt.

Die folgenden Definitionen geben einen Überblick über die Eigenschaften und Operationen, die alle Auflistungsklassen unterstützen.

```
interface IEnumerable
{
  IEnumerator GetEnumerator();
}
interface ICollection
{
  int Count {get;}              // Anzahl der Elemente in der Auflistung
  bool IsSynchronized{get;}     // true, wenn der Zugriff auf das Objekt
                                // synchronisiert (= thread-sicher) ist
  object SyncRoot {get;}        // Object, das den Zugriff auf das Objekt
                                // synchronisiert (= thread-sicher macht)
  void CopyTo(Array arr, int index)    // kopiert die Elemente des Objekts
                                // in ein bereitgestelltes Array
}
interface IList
{
  bool IsFixedSize {get;}        // true, wenn die Auflistung eine
                                 // unveränderliche Größe hat
  bool IsReadOnly {get;}         // true, wenn die Auflistung
                                 // schreibgeschützt ist
  object this[int i] {get; set;} // Indexer für den wahlfreien Zugriff auf
                                 // die Elemente der Auflistung
  int Add(object elem);          // Fügt elem in die Auflistung ein,
                                 // Rückgabewert ist der Index
  void Clear();                  // Entfernt alle Elemente aus der
                                 // Auflistung
  bool Contains(object o);       // Liefert true, wenn o Element der
                                 // Auflistung ist
  int IndexOf(object o);         // Liefert den Index von o, wenn o Element
                                 // der Auflistung ist, andernfalls -1
  void Insert(int i, object o);  // Fügt o an der Position i in die
                                 // Auflistung ein
  void Remove(object o);         // Entfernt das o aus der Auflistung
  void RemoveAt(int i);          // Entfernt das Element an der Position i
                                 // aus der Auflistung
}
```

Einzelne Auflistungsklassen können dieser Grundausstattung – je nach Spezialisierung – im Einzelnen noch andere Klassenelemente (Konstanten, Datentypen, Datenfelder, Eigenschaften, Methoden) hinzufügen. So finden sich in vielen Auflistungen die Operationen `AddRange()` und `RemoveRange()`, die es erlauben, gleich mehrere Elementobjekte an einer beliebigen Stelle en bloc einzufügen bzw. daraus zu entfernen.

In Teil 3 finden Sie jede Menge Praxisbeispiele für den Einsatz von .NET-Auflistungsklassen – ein guter Vertreter ist das Beispielprojekt *MenüDemoOwnerDraw*, das mit den drei auf recht unterschiedliche Datentypen spezialisierten Auflistungsklassen `ImageList.ImageCollection`, `StringCollection` und `Menu.MenuItemCollection` arbeitet.

Die allgemeinste Auflistungsklasse ist übrigens `ArrayList`, deren praktischen Einsatz das im Teil 3 vorgestellte Beispielprojekt *Farben* demonstriert. Diese Klasse ist über dem Elementtyp `object` definiert und lässt sich deshalb für beliebige Elementtypen sowie für die Speicherung beliebiger Datentyp-Mischungen einsetzen. Der Preis dafür ist allerdings eine gewisse Entmachtung des Compilers, da der allfällige Typabgleich erst zur Laufzeit und unter Auswertung der Laufzeittypinformationen erfolgen kann. Die nicht selten abenteuerlichen Typumwandlungen sind aber nicht gerade ein Segen für die Ausführungssicherheit und Ursache so mancher schlecht behandelbarer Ausnahme.

Besser ist es auf jeden Fall, Auflistungsklassen einzusetzen, die speziell auf den jeweiligen Elementtyp zugeschnitten sind. Die Implementierung spezifischer Auflistungsklassen eröffnet zudem die Möglichkeit für die typspezifische Optimierung und die Bereitstellung zusätzlicher, ebenfalls rein typspezifischer Operationen.

Der folgenden Abschnitt klärt, welche Möglichkeiten es für die Definition eigener Auflistungsklassen gibt und welchen Aufwand diese im Einzelnen mit sich bringen.

11.2.2 Eigene Auflistungsklassen

Für die Implementierung einer eigenen Auflistungsklasse gibt es drei gangbare Wege, von denen sich zwei recht ähnlich sind:

1. Ableitung der Auflistungsklasse von den drei Schnittstellen `ICollection`, `IList` und `IEnumerable` mit Implementierung der geforderten Methoden und freier Gestaltung der Repräsentations- und Zugriffsmechanismen. Diesen eher ins Grundsätzliche gehenden Weg sollten Sie wählen, wenn Sie eigene Repräsentations- und Zugriffsmechanismen für Ihre Auflistungsklasse implementieren wollen – beispielsweise eine Baumrepräsentation oder Zugriffe, die durch Hash-Tabellen optimiert sind.

2. Ableitung der Auflistungsklasse von der abstrakten Klasse `CollectionBase` mit datentypspezifischer Implementierung der Auflistungsoperationen. Diesen wesentlich einfacheren Weg sollten Sie wählen, wenn Sie eine spezifische Auflistungsklasse für einen bestimmten Datentyp implementieren wollen.
3. Ableitung der Auflistungsklasse von der Klasse `ArrayList` und Überladung der entsprechenden Operationen mit datentypspezifischen Implementierungen der Auflistungsoperationen. Diesen Weg sollten Sie wählen, wenn Sie eine polymorphe Implementierung der Auflistungsklasse für verschiedene Datentypen beabsichtigen.

Codebeispiel – Auflistungsklassen selbst implementieren

Das Beispielprojekt *MyCollections* demonstriert die Implementierung zweier Auflistungsklassen namens `MyCollection` und `MyArrayListCollection` entlang der Wege 2 und 3. Beide Klassen sind auf den eigenen Datentyp `MyClass` spezialisiert. Die auf Polymorphie bedachte Implementierung von `MyArrayListCollection` erlaubt darüber hinaus die Speicherung beliebiger Datentypen, da die Operationen der Basisklasse für den Elementtyp `object` ja vererbt werden.

Der Code

Der Code beider Klassen ist rein auf die Typverarbeitung bedacht und enthält keinerlei überflüssigen Zierat. Die Klasse `MyCollection` beschränkt sich darauf, die Operationen der von `CollectionBase` vererbten `List`-Eigenschaft auf den Datentyp `MyClass` umgemünzt weiterzureichen, weshalb die Klasse auch nur Elemente dieses Datentyps verarbeitet. Die Klasse `MyArrayListCollection` folgt hingegen der (für sie einzig möglichen) Strategie, die entsprechenden Operationen der Basisklasse `ArrayList` für den Datentyp `MyClass` zu überladen und letztlich wieder auf die Operationen der Basisklasse zurückzuführen. Auf diese Weise ergibt sich eine polymorphe Implementierung, die sich sowohl durch weitere Ableitung als auch durch weitere Überladung sinnvoll erweitern lässt.

```
public class MyClass
{
  private string elem;
  public MyClass(string s)
  {
    elem = s;
  }
  public override string ToString()
  {
    return elem;
  }
}
// Implementierung einer Collection-Klasse auf Grundlage von CollectionBase
public class MyCollection: System.Collections.CollectionBase
{
```

```
  public void Insert(int i, MyClass mc) // Element einfügen
  {
    base.List.Insert(i, mc);
  }
  public int Add(MyClass mc)            // Element hinzufügen
  {
    return base.List.Add(mc);
  }
  public int IndexOf(MyClass mc)        // Index von Element
  {
    return base.List.IndexOf(mc);
  }
  public bool Contains(MyClass mc)      // Element vorhanden?
  {
    return base.List.Contains(mc);
  }
  public void Remove(MyClass mc)        // Element entfernen
  {
    base.List.Remove(mc);
  }
  public object this[int i]             // Indexer
  {
    get {return (MyClass) base.List[i];}
    set {base.List[i] = value;}
  }
}
// Implementierung einer Collection-Klasse auf Grundlage von Array
public class MyArrayListCollection: System.Collections.ArrayList
{
  public virtual void Insert(int i, MyClass mc) // Element einfügen
  {
    base.Insert(i, mc);
  }
  public virtual int Add(MyClass mc)            // Element hinzufügen
  {
    return base.Add(mc);
  }
  public virtual int IndexOf(MyClass mc)        // Index von Element
  {
    return base.IndexOf(mc);
  }
  public virtual bool Contains(MyClass mc)      // Element vorhanden?
  {
    return base.Contains(mc);
  }
  public virtual void Remove(MyClass mc)        // Element entfernen
  {
    base.Remove(mc);
  }
  public new MyClass this[int i]             // Indexer
  {
    get {return (MyClass) base[i];}
    set {base[i] = value;}
  }
}
```

Testcode

Der in der Methode `Main()` befindliche Testcode testet beide Auflistungsklassen, indem er zunächst 10 Elemente des Typs `MyClass` einfügt, dann das Element an der Position 7 entfernt und an der Position 5 ein bzw. zwei weitere Elemente einfügt, bevor er den gesamten Inhalt über eine `foreach`-Iteration (!) ausliest und an der Konsole ausgibt. (Beachten Sie, dass die Implementierung, die diese Iteration ermöglicht, vollständig ererbt ist.) Der Testcode für die Auflistungsklasse `MyArrayListCollection` ist polymorph angelegt.

```
static void Main(string[] args)
{
  // Test der Klasse MyCollection
  MyCollection mc = new MyCollection();
  for (int i = 0; i < 10; i++)
    mc.Add(new MyClass("MyCollection: Wert " + i.ToString()));
  mc.RemoveAt(7);
  mc.Insert(5, new MyClass("Später eingefügt"));
  foreach (MyClass m in mc)   // verwendet IEnumerator-Implementierung
    Console.WriteLine(m);     // ruft ToString()

  Console.WriteLine();     // ruft ToString()

  // Test der Klasse MyArrayCollection
  MyArrayListCollection mac = new MyArrayListCollection();
  for (int i = 0; i < 10; i++)
    mac.Add(new MyClass("MyArrayListCollection: Wert " + i.ToString()));
  mac.RemoveAt(7);
  mac.Insert(5, "Später eingefügt und polymorph");
  mac.Insert(5, 27.7); // Polymorphie
  foreach (object m in mac)   // verwendet IEnumerator-Implementierung
    Console.WriteLine(m.ToString());      // ruft ToString()
}
```

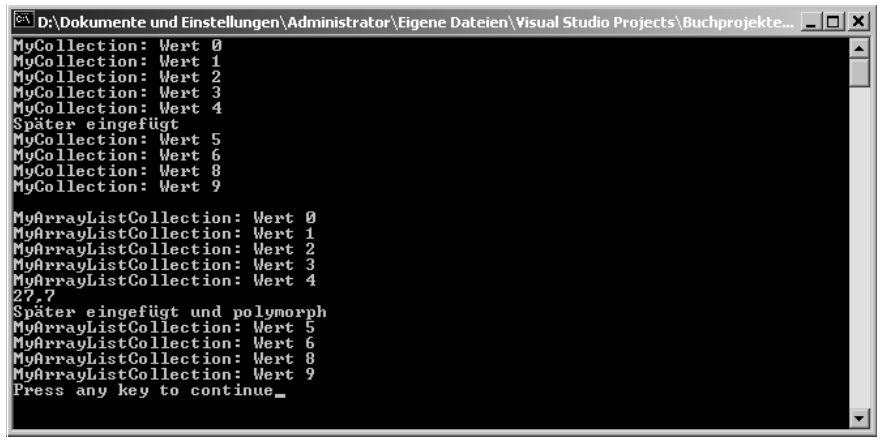


Abbildung 11.3: Ausgabe des Programms MyCollections

11.3 Dateien

Für den Zugriff auf das Dateisystem und die Registrierung sowie für den Umgang mit Dateien finden sich in der .NET-Klassenhierarchie eine Reihe von Klassen, die vom Prinzip her nichts Anderes als mehr oder weniger offensichtliche Anpassungen an bestehende Einrichtungen des Betriebssystems bzw. Funktionen der Win32-API sind. Wer von einer anderen Programmiersprache her kommt, wird in den entsprechenden Klassen alteingesessene Windows-Bibliotheken wiedererkennen, deren Funktionen als Methoden und Eigenschaften verpackt sind.

11.3.1 Dateisystem

Abbildung 11.4 zeigt die Hierarchie der für das Dateisystem vordefinierten .NET-Klassen. Die folgenden Unterabschnitte beschreiben diese Klassen im Detail.

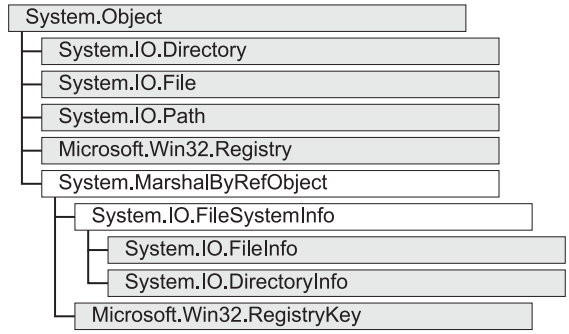


Abbildung 11.4: NET-Klassen für den Zugang und Zugriff auf das Dateisystem

Directory

Die Klasse `Directory` ist eine reine Bibliotheksklasse, die selbst nicht instanziiert werden kann und auch für Ableitungen nicht zur Verfügung steht. Sie verkapselt die wichtigsten verzeichnisbezogenen Operationen des Betriebssystems als statische Methoden. Verschiedene Operationen sind aber auch für Dateipfade definiert. Tabelle 11.5 gibt einen Überblick über die Methoden.

Über das Codebeispiel im folgenden Abschnitt hinausgehend finden Sie in Teil 3 ein auf das `TreeView`-Steuerelement zugeschnittenes Beispielprojekt *TreeViewDemo*, das ausgiebig von der Klasse `Directory` Gebrauch macht und eine interaktive Pfadauswahl unter Darstellung des Verzeichnisbaums ermöglicht.

Tabelle 11.5: Statische Methoden der Klasse `Directory`

statische Methode	Beschreibung
`DirectoryInfo CreateDirectory()`	Erstellt ein neues Verzeichnis anhand des angegebenen Pfads und – sofern erforderlich – alle (noch) nicht existierenden übergeordneten Verzeichnisse. Die Methode generiert verschiedene Ausnahmen – wenn dem Code die Berechtigung fehlt, oder das Verzeichnis aus anderen Gründen nicht erzeugt werden kann.
`void Delete()`	Löscht das angegebene Verzeichnis samt Inhalt. In der zweiparametrigen Variante ermöglicht die Methode auch das Löschen von Verzeichnissen mit allen Unterverzeichnissen.
`bool Exists()`	Liefert `true`, wenn das angegebene Verzeichnis existiert.
`DateTime GetCreationTime()`	Liefert Datum und Zeitpunkt der Erstellung des angegebenen Verzeichnisses.
`string GetCurrentDirectory()`	Liefert den aktuellen Verzeichnispfad der Anwendung.
`string[] GetDirectories()`	Liefert eine Auflistung aller in dem angegebenen Pfad enthaltenden Unterverzeichnisse. In der zweiparametrigen Variante ermöglicht die Methode auch die Spezifikation eines Suchmusters unter Verwendung von Wildcards.
`string GetDirectoryRoot()`	Liefert Datenträger und (wenn vorhanden) das Stammverzeichnis für den angegebenen Datei- oder Verzeichnispfad.
`string[] GetFiles()`	Liefert die (vollständig qualifizierten) Dateipfade der in dem angegebenen Verzeichnispfad enthaltenden Dateien. In der zweiparametrigen Variante ermöglicht die Methode auch die Spezifikation eines Suchmusters unter Verwendung von Wildcards.
`string[] GetFileSystemEntries()`	Liefert die (vollständig qualifizierten) Datei- und Verzeichnispfade der in dem angegebenen Verzeichnispfad enthaltenden Dateien und Unterverzeichnisse. In der zweiparametrigen Variante ermöglicht die Methode auch die Spezifikation eines Suchmusters unter Verwendung von Wildcards.
`DateTime GetLastAccessTime()`	Liefert Datum und Zeitpunkt des letztes Zugriffs auf die angegebene Datei (bzw. Verzeichnis).

Tabelle 11.5: Statische Methoden der Klasse `Directory` (Forts.)

statische Methode	Beschreibung
`DateTime GetLastWriteTime()`	Liefert Datum und Zeitpunkt des letztes Schreibzugriffs auf die angegebene Datei (bzw. Verzeichnis).
`string[] GetLogicalDrives()`	Liefert die Bezeichnungen der dem System bekannten logischen Laufwerke nach dem Muster `"C:\"`.
`string GetParent()`	Liefert das übergeordnete Verzeichnis des angegebenen relativen Datei- oder Verzeichnispfads ab.
`void Move()`	Verschiebt die angegebene Datei oder das angegebene Verzeichnis in das angegebene Zielverzeichnis.
`void SetCreationTime()`	Setzt Datum und Zeitpunkt der Erzeugung der angegebenen Datei (bzw. Verzeichnis) auf den angegebenen `DateTime`-Wert.
`void SetCurrentDirectory()`	Setzt das aktuelle Verzeichnis auf den angegebenen Verzeichnispfad.
`void SetLastAccessTime()`	Setzt Datum und Zeitpunkt des letzten Zugriffs auf die angegebene Datei (bzw. Verzeichnis) auf den angegebenen `DateTime`-Wert.
`void SetLastWriteTime()`	Setzt Datum und Zeitpunkt des letzten Schreibzugriffs auf die angegebene Datei (bzw. Verzeichnis) auf den angegebenen `DateTime`-Wert.

Codebeispiel – Erstellungzeit ändern

Die formularlose Windows-Anwendung *ErstellZeit* ist ein kleines, aber durchaus nützliches Tool, das die Auswahl eines Verzeichnisses gestattet und darin – sowie rekursiv in allen Unterverzeichnissen – die Erstell- und Zugriffszeiten aller Dateien und Verzeichnisse auf die aktuelle Uhrzeit ändert.

Zur Abfrage des Dateipfades verwendet das Programm eine geeignet initialisierte `OpenFileDialog`-Komponente, ignoriert aber den eingegebenen Dateinamen und verwendet nur den Verzeichnispfad. Von diesem Startverzeichnis aus durchläuft es rekursiv alle untergeordneten Verzeichnisse und ändert die Zeiten.

```
using System;
using System.Windows.Forms;
using System.IO;
```

```
namespace Erstellzeit
{
  class ChangeCreateTime
  {
    [STAThread]
    static void Main()
    {
      OpenFileDialog ofd = new OpenFileDialog();
      ofd.ValidateNames = false;
      ofd.FileName = "(alle)";
      if (ofd.ShowDialog() == DialogResult.OK)
        SetDateTime(Path.GetDirectoryName(ofd.FileName));
    }
    static void SetDateTime(string path)  // rekursive Methode
    {
      string[] dirs = Directory.GetDirectories(path);
      if (dirs.Length > 0)            // Gibt es Unterverzeichnisse?
        foreach(string s in dirs)     // Einzeln abarbeiten und
          SetDateTime(s);             // ab in die Rekursion
      string[] files = Directory.GetFileSystemEntries(path);
      foreach(string s in files)      // Zeiten der Dateien und
      {                               //Verzeichnisse ändern
        Directory.SetCreationTime(s, DateTime.Now);
        Directory.SetLastAccessTime(s, DateTime.Now);
        Directory.SetLastWriteTime(s, DateTime.Now);
      }
    }
  }
}
```

File

Wie `Directory` ist auch die Klasse `File` eine reine Bibliotheksklasse, die weder selbst instanziiert werden kann noch für Ableitungen zur Verfügung steht. In Ergänzung zu `Directory` verkapselt sie die wichtigsten dateibezogenen Operationen des Betriebssystems als statische Methoden. Tabelle 11.5 gibt einen Überblick über die Methoden.

Tabelle 11.6: Statische Methoden der Klasse `File`

statische Methode	Beschreibung
`StreamWriter AppendText()`	Öffnet die angegebene Textdatei im Anfügemodus und liefert ein gebrauchsfertiges `StreamWriter`-Objekt dafür.
`void Copy()`	Erzeugt eine Kopie der angegebenen Datei unter dem angegebenen Namen. Die zweiparametrige Variante kann nur neue Dateien erzeugen, die dreiparametrige auch bestehende Dateien überschreiben.

Tabelle 11.6: Statische Methoden der Klasse `File` (Forts.)

statische Methode	Beschreibung
`FileStream Create()`	Öffnet die Datei mit dem angegebenen Namen oder erstellt ein neue und liefert ein `FileStream`-Objekt dafür. Die zweite Variante ermöglicht es, eine von Standardwert abweichende Puffergröße vorzugeben.
`StreamWriter CreateText()`	Öffnet oder erzeugt die angegebene Textdatei im Schreibmodus und liefert ein gebrauchsfertiges `StreamWriter`-Objekt dafür.
`void Delete()`	Löscht die angegebene Datei; erzeugt keine Ausnahme, wenn die Datei nicht existiert.
`bool Exists()`	Liefert `true`, wenn die angegebene Datei existiert.
`FileAttributes GetAttributes()`	Liefert einen Bitvektor, der die Dateiattribute der angegebenen Datei beschreibt.
`DateTime GetCreationTime()`	Liefert Datum und Zeitpunkt der Erstellung der angegebenen Datei.
`DateTime GetLastAccessTime()`	Liefert Datum und Zeitpunkt des letztes Zugriffs auf die angegebene Datei.
`DateTime GetLastWriteTime()`	Liefert Datum und Zeitpunkt des letztes Schreibzugriffs auf die angegebene Datei.
`void Move()`	Verschiebt die angegebene Datei in das angegebene Zielverzeichnis und/oder benennt die Datei um, wenn als Zielpfad ein Dateipfad angegeben ist.
`FileStream Open()`	Öffnet die Datei mit dem angegebenen Namen in dem angegebenen Zugriffsmodus und liefert ein gebrauchsfertiges `FileStream`-Objekt dafür.
`FileStream OpenRead()`	Öffnet die Datei mit dem angegebenen Namen zum Lesen und liefert ein gebrauchsfertiges `FileStream`-Objekt dafür.
`StreamReader OpenText()`	Öffnet angegebene Textdatei zum Lesen und liefert ein gebrauchsfertiges `StreamReader`-Objekt dafür.
`FileStream OpenWrite()`	Öffnet die Datei mit dem angegebenen Namen zum Schreiben und liefert ein gebrauchsfertiges `FileStream`-Objekt dafür.
`void SetAttributes()`	Setzt die Dateiattribute der angegebenen Datei auf den angegebenen `FileAttributes`-Bitvektor.

Tabelle 11.6: Statische Methoden der Klasse `File` (Forts.)

statische Methode	Beschreibung
`void SetCreationTime()`	Setzt Datum und Zeitpunkt der Erzeugung der angegebenen Datei auf den angegebenen `DateTime`-Wert.
`void SetLastAccessTime()`	Setzt Datum und Zeitpunkt des letzten Zugriffs auf die angegebene Datei auf den angegebenen `DateTime`-Wert.
`void SetLastWriteTime()`	Setzt Datum und Zeitpunkt des letzten Schreibzugriffs auf die angegebene Datei auf den angegebenen `DateTime`-Wert.

Path

Bei der Klasse `Path` handelt es sich gleichfalls um eine reine Bibliotheksklasse, deren Elemente allesamt statisch sind. Die Klasse fasst die wichtigsten Informationen und Operationen für den dateisystemunabhängigen und rein formalen Umgang mit Dateinamen und Verzeichnispfaden zusammen. Sie ist weder instanziierbar noch steht sie für eigene Ableitungen zur Verfügung. Die Tabellen 11.7 und 11.8 geben einen Überblick über die Eigenschaften und Methoden der Klasse.

Tabelle 11.7: Eigenschaften der Klasse `Path`

statische Eigenschaft	Beschreibung
`char AltDirectorySeperatorChar`	Alternatives Trennzeichen für Verzeichnisse in Pfaden (Unix)
`char DirectorySeparatorChar`	Trennzeichen für Verzeichnisse in Pfaden (FAT, FAT32, NTFS)
`char[] InvalidPathChars`	In Pfaden nicht zulässige Zeichen
`char PathSeparator`	Trennzeichen für die Notation mehrerer Pfade (beispielsweise für Filter)
`char VolumeSeparatorChar`	Trennzeichen zwischen logischer Laufwerksbezeichnung und Pfadanteil

Tabelle 11.8: Methoden der Klasse `Path`

statische Methode	Beschreibung
`string ChangeExtension()`	Ändert die Dateierweiterung
`string Combine()`	Kombiniert zwei Pfade zu einem. Die Operation ist im Wesentlichen eine String-Verkettung, wobei der im zweiten Parameter angegebene relative Pfad gegebenenfalls unter Ergänzung des Pfadtrennzeichens `DirectorySeparator` an den im ersten Parameter genannten Pfad angehängt wird. Ist der zweite Parameter ein absoluter Pfad oder ein UNC-Pfad, liefert die Methode diesen als Ergebnis.
`string GetDirectoryName()`	Schneidet den Dateinamen ab und liefert den Verzeichnisanteil des angegebenen relativen oder absoluten Dateipfads
`string GetExtension()`	Liefert nur die Erweiterung des angegebenen Dateipfads
`string GetFileName()`	Liefert den Dateinamen des angegebenen Dateipfads samt Erweiterung
`string GetFileNameWithoutExtension()`	Liefert den Dateinamen des angegebenen Dateipfads ohne Erweiterung
`string GetFullPath()`	Liefert den vollständigen Pfad zu dem angegebenen Dateinamen. Der leere String ist als Parameter nicht erlaubt. Relative Pfadangaben werden mit Bezug auf das aktuelle Verzeichnis `Environment.CurrentDirectory` ausgewertet.
`string GetPathRoot()`	Liefert den Wurzelanteil des Pfades, beispielsweise `"C:\"` bei Laufwerkspfad, `"\\Machine\Ressource\"` bei UNC-Pfad oder `""` bei relativem Pfad.
`string GetTempFileName()`	Erzeugt eine temporäre Datei im %Temp%-Verzeichnis des Systems
`string GetTempPath()`	Liefert das %Temp%-Verzeichnis des Systems
`bool HasExtension()`	`true`, wenn der angegebene Dateiname eine Erweiterung enthält
`bool IsPathRooted()`	`true`, wenn der angegebene Pfad absolut ist

Codebeispiel – Path-Demo

Der folgende Code aus dem Projekt *Path-Demo* demonstriert die Wirkung der Eigenschaften und Methoden von `Path`:

```
static void Main(string[] args)
{
  Console.WriteLine("Werte der statischen Path-Eigenschaften");
  Console.WriteLine("=======================================" +
    "==========================");
  Console.WriteLine("Path.AltDirectorySeparatorChar: {0}",
    Path.AltDirectorySeparatorChar);
  Console.WriteLine("Path.DirectorySaperatorChar:    {0}",
    Path.DirectorySeparatorChar);
  Console.WriteLine("Path.InvalidPathChars:          {0}",
    new String(Path.InvalidPathChars));
  Console.WriteLine("Path.AltDirectorySeparatorChar: {0}",
    Path.AltDirectorySeparatorChar);
  Console.WriteLine("Path.PathSeparator:             {0}",
    Path.PathSeparator);
  Console.WriteLine("Path.VolumeSeparatorChar:       {0}",
    Path.VolumeSeparatorChar);
  Console.WriteLine("Path.AltDirectorySeparatorChar: {0}",
    Path.AltDirectorySeparatorChar);

  Console.WriteLine("\nWerte verschiedener Path-Methoden");
  Console.WriteLine("=======================================" +
    "==========================");
  Console.WriteLine("Path.Combine():                 {0}",
    Path.Combine("Rudi\\", "Rudi.jpg" ));
  Console.WriteLine("Path.Combine():                 {0}",
    Path.Combine(@"\\Athlon2000\Rumana", @"Rudi.jpg" ));
  Console.WriteLine("Path.Combine():                 {0}",
    Path.Combine(@"\\Athlon2000\Rumana\", @"Rudi.jpg" ));
  Console.WriteLine("Path.Combine():                 {0}",
    Path.Combine(@"\\Athlon2000\Rumana\", @"\Rudi.jpg" ));
  Console.WriteLine("Path.GetPathRoot():             {0}",
    Path.GetPathRoot(Path.GetFullPath("test.jpg")));
  Console.WriteLine("Path.GetPathRoot():             {0}",
    Path.GetPathRoot(@"\\Athlon2000\Rumana\Sub\Sub\test.jpg"));

  // Temporäre Datei
  string tmpFile = Path.GetTempFileName();
  Console.WriteLine("Path.GetTempFileName(): {0}", tmpFile);
  Console.WriteLine("Path.GetTempPath():     {0}", Path.GetTempPath());
  TestPathMethods(tmpFile);
  TestPathMethods("TestDatei.jpeg");
}

static void TestPathMethods(string s)
{
  Console.WriteLine("\nTest der Path-Methoden für {0}", s);
  Console.WriteLine("=======================================" +
    "==========================");
  Console.WriteLine("GetFullPath(): {0}", Path.GetFullPath(s));
```

```
  Console.WriteLine("GetDirectoryName():   {0}", Path.GetDirectoryName(s));
  Console.WriteLine("GetExtension():       {0}", Path.GetExtension(s));
  Console.WriteLine("Path.GetFileName():   {0}", Path.GetFileName(s));
  Console.WriteLine("GetFileNameWithoutExtension():{0}",
    Path.GetFileNameWithoutExtension(s));
}
```

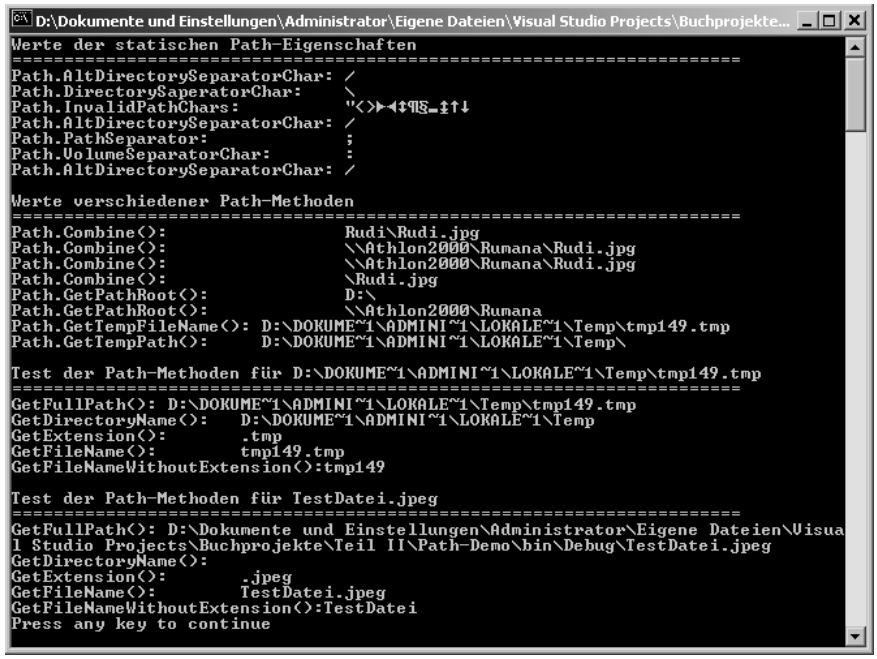


Abbildung 11.5: Ausgaben des Testcodes

FileSystemInfo, FileInfo, DirectoryInfo

Wer schon einmal etwas mit dem Windows Scripting Host oder anderen Scriptsprachen (sowie auch VB) zu tun hatte, trifft mit der `FileSystemInfo`-Klasse auf einen alten Bekannten. Faktisch stellt `FileSystemInfo` einen etwas moderneren Zugang zum Dateisystem auf Basis eines COM-Objekts dar, während die Klassen `Directory`, `File` und `Path` schlichte Hüllklassen für die (älteren) Funktionen der Win32-API sind.

Diese Klassen werden von Visual Studio .NET sowohl bei der Installation als auch im laufenden Betrieb verwendet – und stellen übrigens den Grund dar, wieso VS.NET am Script Blocking von Antivirenprogrammen scheitert (vgl. Anhang A).



Die von der abstrakten Klasse `FileSystemInfo` abstammenden Klassen `FileInfo` und `DirectoryInfo` sind instanziierbar und nehmen die formale Beschreibung eines konkreten Datei- bzw. Verzeichnispfades als Objekt vor. Der Pfad ist bereits bei Konstruktion anzugeben.

Die verschiedenen teilweise von `FileSystemInfo` ererbten Eigenschaften geben Aufschluss über die Umgebung (`Directory`, `DirectoryName`, `FullName`, `Name`, `Parent`) des Objekts, liefern im Wesentlichen aber Informationen (`LastAccessTime`, `CreationTime`, `LastWriteTime`, `Attributes`), die sich alternativ auch über die Bibliotheksklassen `File`, `Directory` und `Path` in Erfahrung bringen lassen. Analoges gilt für die Methoden – die meisten der bereitgestellten Operationen finden in sich in der einen oder anderen Form auch dort in statischer Implementierung.

11.3.2 Dateizugriff, Streams

Dateizugriffe nehmen unter .NET keine Sonderstellung ein. Nach dem Vorbild von C++ erfolgen I/O-Operationen in C# (bzw. .NET) generell auf der Basis von Streams, die ihrerseits eine Abstraktion von Schreib- und Lesevorgängen für beliebige Medien darstellen – seien es Dateien auf physischen Datenträgern, physische Ein-/Ausgabegeräte (beispielsweise eine serielle Schnittstelle), Netzwerkverbindungen mit anderen Computern oder schlichte Speicherblöcke im Arbeitsspeicher.

Abbildung 11.6 zeigt, welche Klassen in der .NET-Klassenhierarchie zur Durchführung stream-orientierter I/O-Operationen vordefiniert sind und stellt auch ihren Vererbungszusammenhang dar. Eine zentrale Stellung nimmt dabei die abstrakte Klasse `Stream` ein, die als Weichensteller für die unterschiedlichen Arten von Datenströmen fungiert und letztlich dafür sorgt, dass die Daten von der Quelle zum Ziel gelangen – sei es gepuffert oder ungepuffert, interpretiert oder roh, verschlüsselt oder unverschlüsselt.

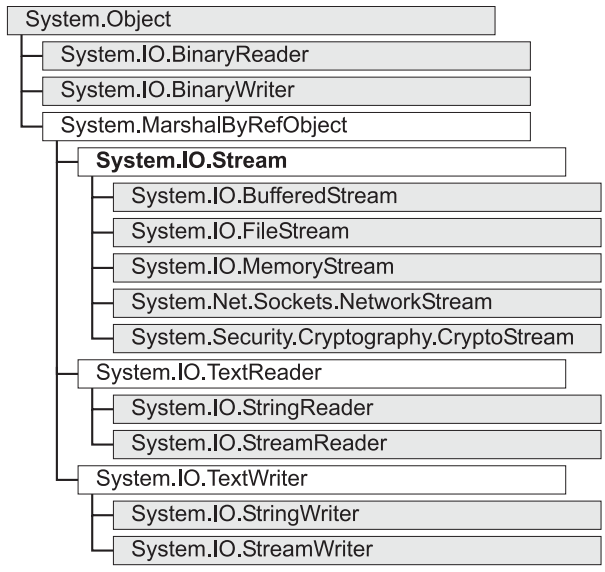


Abbildung 11.6: NET-Klassen für den Dateizugriff

Arten von I/O-Zugriffen

Mit Blick auf die Ausgangs- und Zieldatentypen sind im Wesentlichen drei Arten von I/O-Zugriffen zu unterscheiden:

1. *Rohe Stream-Operationen* – Operationen dieser Art sind schlicht auf der Basis des Datentyps `byte` bzw. `byte[]` definiert. Die Klasse `Stream` vererbt diese Operationen als abstrakte Methoden an die Klassen `BufferedStream`, `FileStream`, `MemoryStream`, `NetworkStream` und `CryptoStream` und zwingt diese, eigene Implementierungen für den Datentransfer mit den spezifischen Medien bereitzustellen, auf die sie spezialisiert sind.
2. *Binär-orientierte Schreib- und Leseoperationen* – Operationen dieser Art sind als Überladungen der rohen Stream-Operationen für die integrierten Datentypen definiert und bieten zudem die Grundlage für die (De-)Serialisierung komplexer und benutzerdefinierter Datentypen. Für die Schreibrichtung ist die Klasse `BinaryWriter` und für die Leserichtung die Klasse `BinaryReader` zuständig. Aufgabe dieser Klassen, deren Instanziierung unter Angabe eines passenden `Stream`-Objekts erfolgt, ist es, die Repräsentationen der integrierten Datentypen in das Rohdatenformat `byte[]` umzusetzen und umgekehrt.
3. *Textorientierte Schreib- und Leseoperationen* – Operationen dieser Art sind auf der Basis der Datentypen `string` und `char` definiert. Sie interpretieren den Datenstrom nicht nur mit Blick auf Sonderzeichen wie Zeilenvorschübe (LF) und Wagenrückläufe (CR), sondern nehmen auch die Umsetzung in die jeweils geltende oder gewünschte Zeichenkodierung vor. Die zuständigen Klassen stammen von den abstrakten Basisklassen `TextReader` und `TextWriter` ab und existieren sowohl als Stream-orientierte Varianten (`StreamReader`, `StreamWriter`) als auch als String-orientierte Varianten (`StringReader` und `StringWriter`). Sie werden daher entweder unter Angabe eines Strings bzw. eines Streams instanziiert; ihre Operationen sind auf die Interpretation und den Transfer einzelner Zeichen sowie Textzeilen ausgerichtet.

Bild 11.7 verdeutlicht die Zusammenhänge zwischen den genannten Klassen und die möglichen Wege des Datenstroms von Datentyp zu Datentyp.

Stream-Operationen

`Stream` ist das Paradebeispiel einer abstrakten Klasse, die den Grundstein für die polymorphe Implementierung einer kleinen Dynastie von Klassen legt und als Portal für diese Klassen fungiert. In fast allen Fällen, in denen eine .NET-Methode oder -Klasse mit stream-basierten Operationen hantiert, wird nichts weiter als ein `Stream`-Objekt erwartet. Damit besteht die Möglichkeit, die Instanz einer beliebigen, von `Stream` abstammenden Klasse anzugeben – gleich, ob .NET-Klasse oder eigene Ableitung.

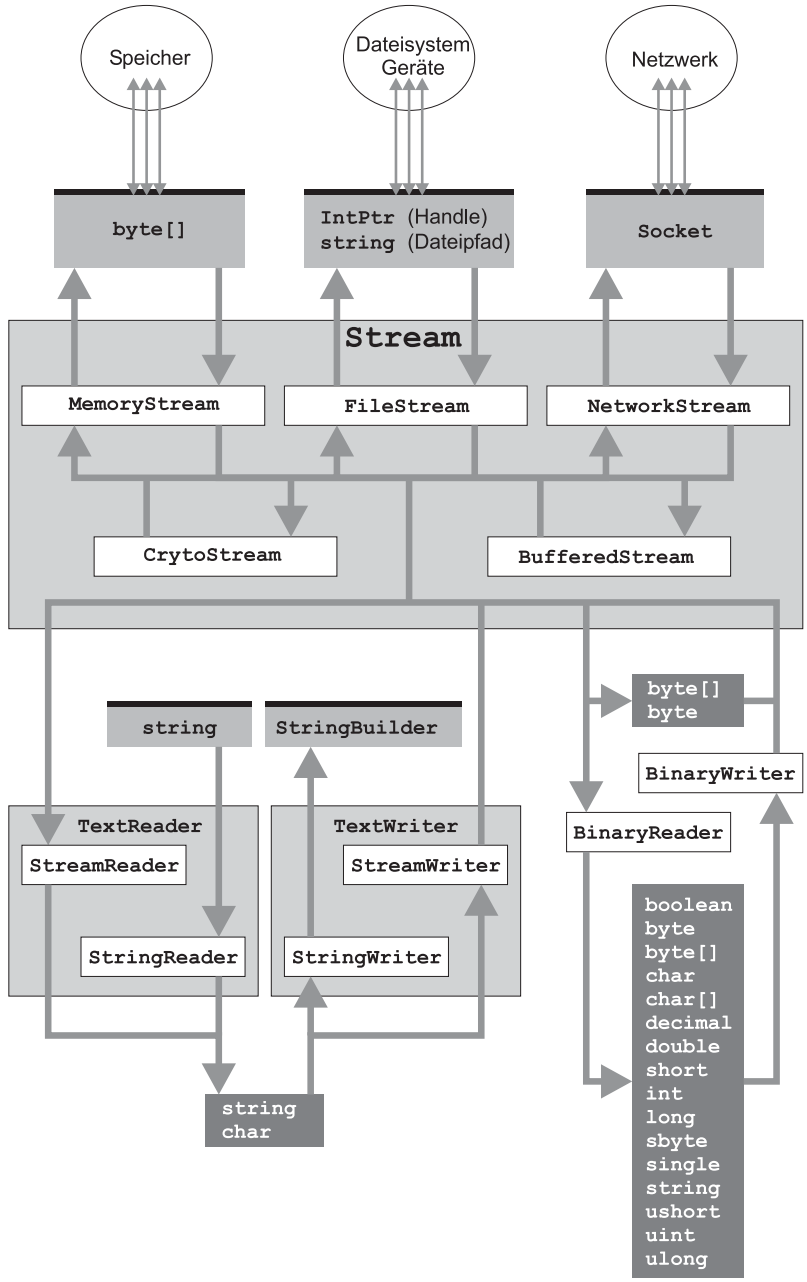


Abbildung 11.7: Die zentrale Stellung der Klasse `Stream` und die möglichen Wege für Datenströme

Wie in Abbildung 11.7 dargestellt, operieren die Klassen `StreamReader`, `StreamWriter`, `BinaryReader`, `BinaryWriter`, aber auch `CryptoStream` und `BufferedStream` über `Stream`-Objekten, wobei `CryptoStream` und `BufferedStream` selbst Ableitungen von `Stream` sind und somit eine Art Filter darstellen. Wie die Namen bereits verraten, ermöglicht ein zwischengeschaltetes `BufferedStream`-Objekt

die Pufferung eines Streams in einem beliebig großen Puffer (Standardwert ist 4096 Byte) und ein `CryptoStream`-Objekt die (De-)Chiffrierung eines Streams auf der Basis einer Chiffrierklasse, die die Schnittstelle `ICryptoTransform` implementiert. Da die Chiffrierung zusammen mit den dahinter steckenden Konzepten Stoff genug für mehrere Bücher gleichen Umfangs wie das vorliegende liefern, wird hier nicht näher darauf eingegangen.

Die Tabellen 11.9 und 11.10 geben einen Überblick über die Eigenschaften und Methoden der Klasse `Stream`. Die Eigenschaften der Klasse sind allesamt abstrakt, wie auch viele der Methoden, weshalb abgeleitete Klasse ihre eigenen Implementierungen dafür bereitstellen müssen. Die restlichen Methoden – darunter auch die für die asynchronen Schreib- und Leseoperationen – sind virtuell; ihre Implementierung wird daher von `Stream` vererbt.

Tabelle 11.9: Eigenschaften, die von abgeleiteten `Stream`-Klassen implementiert werden müssen.

Eigenschaft	Beschreibung
`abstract bool CanRead {get;}`	`true`, wenn der Stream die `ReadXxx`-Operationen unterstützt
`abstract bool CanSeek {get;}`	`true`, wenn der Stream die `Seek`-Operationen unterstützt
`abstract bool CanWrite {get;}`	`true`, wenn der Stream die `WriteXxx`-Operationen unterstützt
`abstract int Length {get; set;}`	Länge des Streams in Byte
`abstract long Position {get; set;}`	Schreib/Leseposition im Stream

Tabelle 11.10: Virtuelle und abstrakte Methoden der Klasse `Stream`

Methode	Beschreibung
`virtual IAsyncResult BeginRead()`	Startet einen asynchronen Lesevorgang für den Stream, der die angegebene Anzahl Bytes in den bereitgestellten Puffer überträgt. Es kann ein Delegat des Typs `AsyncCallback` sowie ein Argument dafür angegeben werden, den die Methode unmittelbar nach Abschluss des Lesevorgangs aufruft. Setzt voraus, dass das jeweilige `Stream`-Objekt asynchrone Lesevorgänge unterstützt und entsprechend konstruiert wurde. Erfordert einen abschließenden `EndRead()`-Aufruf.
`virtual IAsyncResult BeginWrite()`	Wie `BeginRead()`, jedoch als Schreibvorgang, der Daten aus dem Puffer in den Stream überträgt.
`virtual void Close()`	Schließt den Stream und gibt alle dafür angeforderten Ressourcen (Handles, Sockets, Puffer etc.) frei.

Tabelle 11.10: Virtuelle und abstrakte Methoden der Klasse `Stream` (Forts.)

Methode	Beschreibung
`virtual int EndRead()`	Ermöglicht die Resynchronisation asynchroner Lesevorgänge, die mit `BeginRead()` gestartet wurden. Die Methode wird unter Angabe des von dem `BeginRead()`-Aufruf gelieferten `IAsyncResult`-Objekts aufgerufen und blockiert dann die Ausführung, bis die Leseoperation abgeschlossen ist. Danach sorgt sie für die Freigabe der für die Operation belegten Ressourcen und gibt die Anzahl der aus dem Stream gelesenen Bytes zurück.
`virtual int EndWrite()`	Wie `EndRead()`, jedoch für die Resynchronisation asynchroner Schreibvorgänge, die mit BeginWrite() gestartet wurden.
`abstract void Flush()`	Erzwingt das Schreiben aller noch gepufferten Daten in den Stream. (Zeigt nur bei Klassen eine Wirkung, die gepufferte Schreiboperationen implementieren.)
`abstract int Read()`	Überträgt maximal die angegebene Anzahl an Bytes aus dem Stream in das bereitgestellte `byte`-Array. Liefert die Anzahl der tatsächlich gelesenen Bytes und aktualisiert die Eigenschaft `Position`. Blockiert, wenn nicht mindestens ein Byte gelesen wurde, es sei denn, es stehen keine weiteren Bytes mehr aus und das Ende des Streams wurde erreicht. Die Operation wird von abgeleiteten Klassen standardmäßig auf der Basis von `BeginRead()`/`EndRead()` implementiert.
`abstract byte ReadByte()`	Liest ein einzelnes Byte aus dem Stream und liefert dieses als Ergebnis. Aktualisiert die `Position`-Eigenschaft.
`abstract long Seek()`	Setzt die Schreib- bzw. Leseposition im Stream auf einen Wert, den die Methode aus der angegebenen Startposition und einer relativen Distanz dazu errechnet. Die Startposition wird über einen der Werte des Aufzählungstyps `SeekOrigin` ausgedrückt: `Begin`, `Current` oder `End`. Die Methode aktualisiert weiterhin die Eigenschaft `Position` und liefert deren Wert. Alternativ lässt sich die Schreib- bzw. Leseposition auch (absolut) über die `Position`-Eigenschaft setzen.

Methode	Beschreibung
`abstract void SetLength()`	Im Regelfall setzt die Methode die Länge des Streams auf den angegebenen `long`-Wert. Der Stream wird dabei mit undefiniertem Inhalt verlängert oder unter Verlust gegebenenfalls bereits geschriebener oder noch nicht gelesener Daten verkürzt. Diese Methode erzeugt allerdings standardmäßig die Ausnahme `NotSupportedException`, wenn sie für einen Stream aufgerufen wird, der nicht sowohl Schreib- als auch Lesevorgänge unterstützt.
`abstract void Write()`	Überträgt die angegebene Anzahl an Bytes aus dem angegebenen Puffer in den Stream und aktualisiert die `Position`-Eigenschaft. Diese Operation wird von abgeleiteten Klassen standardmäßig auf der Basis von `BeginWrite()`/`EndWrite()` implementiert.
`virtual void WriteByte()`	Überträgt das angegebene Byte in den Stream. Aktualisiert die `Position`-Eigenschaft.

Tabelle 11.10: Virtuelle und abstrakte Methoden der Klasse `Stream` (Forts.)

Indexsequenzieller Zugriff

Wenn Sie direkt mit einem `Stream`-Objekt operieren – und nicht mittelbar über die Klassen `BinaryWriter` und `BinaryReader` bzw. `StreamReader` und `StreamWriter`, bekommen Sie es in erster Linie mit der `Length`-Eigenschaft und den Methoden `Read()` und `Write()` zu tun. Die `Seek()`-Methode und die `Position`-Eigenschaft kommen nur ins Spiel, wenn Sie einen indexsequenziellen Zugriff auf den Stream benötigen.

Obwohl nicht gleich auf den ersten Blick sichtbar, lassen sich indexsequenzielle Zugriffe auch für die Klassen `BinaryWriter` und `BinaryReader` verwirklichen, die selbst keine `Seek()`-Methode oder `Position`-Eigenschaft offen legen – nämlich über die Eigenschaft `BaseStream`, die den vollen Zugriff auf das untergelegte `Stream`-Objekt gestatten.

Für die textorientierten Klassen `StreamReader` und `StreamWriter` gibt es hingegen keine offizielle Möglichkeit für einen indexsequenziellen Zugriff, die über das optionale Öffnen im Append-Modus hinausgehen würde. Das hat natürlich etwas mit der Art des Zugriffs zu tun, der für Textdateien traditionell zeichen- bzw. zeilenorientiert und somit sequenziell ausgerichtet ist. Inoffiziell bleibt vom Prinzip her aber immer noch die Möglichkeit, selbst eine Referenz auf das `Stream`-Objekt zu verwalten (beispielsweise in einer abgeleiteten Klasse) oder die entsprechenden Operationen gleich über die Klassen `BinaryWriter` und `BinaryReader` abzuwickeln.

Codebeispiel – Checksumme berechnen

Die Windows-Anwendung *Checksummen* ist ein Testbett für ein paar Methoden, die jeweils auf unterschiedliche Weise dasselbe Ergebnis berechnen – und dabei verschiedene Arten von Streams und Dateizugriffen vorstellen.

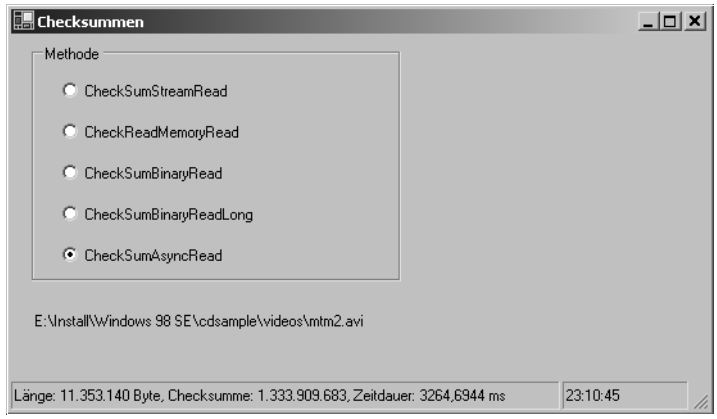


Abbildung 11.8: Formular der Testanwendung *Checksummen*

Wie Abbildung 11.8 zeigt, bietet das Programm fünf Methoden an. Vier davon operieren synchron, die fünfte asynchron. Das Formular der Anwendung ermöglicht den Aufruf der Methoden und gibt die relevanten Größen – darunter auch den Zeitbedarf – in der Statusleiste aus. Der Code für die Auswahl der Methoden und die Statusanzeige:

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Globalization;
using System.IO;
...
private void RadioButton_Click(object sender, EventArgs e)
{
  int index = groupBox1.Controls.IndexOf((Control) sender);
  // Dateinamen abfragen
  OpenFileDialog ofd = new OpenFileDialog();
  if (ofd.ShowDialog() == DialogResult.OK)
  {
    long len = new System.IO.FileInfo(ofd.FileName).Length;
    label1.Text = ofd.FileName;
    statusBarPanel1.Text =
      String.Format("Berechne Checksumme für {0:n0} Bytes", len);
    Application.DoEvents();
    long checkSum = 0;
    DateTime startTime = DateTime.Now;
```

```
    switch(index)
    {
      case 0:
        checkSum = CheckSumStreamRead(ofd.FileName);
        break;
      case 1:
        checkSum = CheckSumMemoryRead(ofd.FileName);
        break;
      case 2:
        checkSum = CheckSumBinaryRead(ofd.FileName);
        break;
      case 3:
        checkSum = CheckSumBinaryReadLong(ofd.FileName);
        break;
      case 4:
        CheckSumAsyncRead(ofd.FileName, startTime);
        return;      // Statusanzeige wird asynchron aktualisiert
    }
    SetStatus (startTime, len, checkSum);
  }
}

void SetStatus (DateTime startTime, long len, long checkSum)
{
  DateTime endTime = DateTime.Now;
  TimeSpan duration = new TimeSpan(endTime.Ticks - startTime.Ticks);
  statusBarPanel1.Text = String.Format(
    "Länge: {0:n0} Byte, Checksumme: {1:n0}, Zeitdauer: {2} ms",
    len, checkSum, duration.TotalMilliseconds);
}
```

Die vier synchronen Methoden erwarten einen Dateipfad als String, lesen die dadurch spezifizierte Datei über ein `FileStream`-Objekt und liefern eine Prüfsumme dafür, die sich durch schlichte Aufsummierung aller Bytes zu einem `long`-Wert ergibt. Die fünfte, asynchron operierende Methode übernimmt selbst die Aktualisierung der Statusanzeige, ist aber sinngemäß genauso definiert.

Drei der Methoden verwenden die Arbeitsroutine `ReadStream()` für die Berechnung. Diese Methode erwartet ein fertiges `Stream`-Objekt und liest dieses zur Berechnung der Prüfsumme byteweise in der angegebene Länge aus:

```
static long ReadStream(Stream inStream, long count)
{
  long checkSum = 0;
  for (long i = 0; i < count; i++)  // Checksumme zeichenweise berechnen
    checkSum +=  inStream.ReadByte();  // Byte-Wert hinzuaddieren
  return checkSum;
}
```

CheckSumStreamRead() – FileStream zeichenweise lesen

Die Methode `CheckSumStreamRead()` nimmt den direktesten (aber nicht unbedingt schnellsten) Weg. Sie öffnet die Datei als `FileStream`-Objekt und reicht dieses direkt an die Methode `ReadStream()` weiter:

```
static long CheckSumStreamRead(string path)
{
  if( !File.Exists(path))
    return -1;
  Stream inStream =
    new FileStream( path, FileMode.Open, FileAccess.Read, FileShare.Read);
//  inStream = new BufferedStream(inStream);   // kann wegfallen
  long res = ReadStream(inStream);
  inStream.Close();             // hier nicht notwendig aber, gute Praxis
  return res ;
}
```

Dass die Methode den Stream vor dem Rücksprung noch schließt, ist zwar gute Programmierpraxis, im vorliegenden Fall aber nicht erforderlich, weil die `Close()`-Methode beim Abbau eines `Stream`-Objeks automatisch ausgeführt wird.

Wahlweise kann der Dateizugriff auch über ein dazwischen geschaltetes `BufferedStream`-Objekt abgewickelt werden. Dies geschieht im Beispielprogramm, wenn die auskommentierte Zeile mitkompiliert wird. Wie Zeitmessungen zeigen, bringt dies im vorliegenden Fall keine echten Vorteile, da das Betriebssystem Dateizugriffe erstens intern ohnehin puffert (der `FileStream`-Konstruktor verwendet einen passenden Vorgabewert für die Überladungen, die keinen Parameter für die Puffergröße erwarten) und der Zugriff durch die Routine zweitens rein sequenziell erfolgt. Eine Pufferung von `FileStream`-Objekten ist im Allgemeinen nur im Zusammenhang mit indexsequenziellen Zugriffen (vgl. `Seek()` und `Position`-Eigenschaft) interessant.

CheckSumMemoryRead() – Checksumme über MemoryStream berechnen

Gerade im Zusammenhang mit zeitaufwändigeren Operationen über gemeinsam genutzte Dateien sollte eine Anwendung darauf achten, die Datei möglichst schnell wieder freizugeben. Die Methode `CheckSumMemoryRead()` liest den Inhalt der Datei en bloc in einen Puffer ein, schließt die Datei sofort und berechnet die Checksumme erst danach – über ein `MemoryStream`-Objekt:

```
static long CheckSumMemoryRead(string path)
{
  if( !File.Exists(path))
    return -1;
  Stream fStream = new FileStream( path, FileMode.Open, FileAccess.Read,
FileShare.Read);
  byte[] buffer = new byte[fStream.Length];
```

```
  fStream.Read(buffer, 0, buffer.Length);     // en bloc in Puffer einlesen
  fStream.Close();                            // Datei freigeben

  // in der Speicher abgebildete Datei verwenden
  return ReadStream(new MemoryStream(buffer));
}
```

Wie die in der Statusleiste ausgegebene Zeitmessung zeigt, ist der Weg, den diese Methode einschlägt – zumindest bei Zugriffen auf lokale Festplatten – erheblich schneller (bei Dateien um 10 MB ca. um den Faktor 5). Allerdings ist hier bei sehr langen Dateien Vorsicht geboten, denn es macht im Allgemeinen keinen Sinn, Puffer für `MemoryStream`-Objekte zu verwenden, die den realen Arbeitsspeicher des Systems zu sehr auslasten oder gar überschreiten.

Beachten Sie weiterhin, dass die angezeigten Zeitwerte nur dann halbwegs aussagekräftig sind, wenn das System echte Festplattenzugriffe ausführen muss und die Datei oder Teile davon nicht wieder aus irgendeinem Cache hervorzaubern kann. Für ernsthaftere Versuchsreihen sollte man also mehrfache Kopien ein- und derselben Datei anfertigen oder zumindest unterschiedliche Dateien gleichen Umfangs einsetzen.

CheckSumBinaryRead() – als Binärdatei lesen

Die Methode `CheckSumBinaryRead()` geht den gar nicht mal so unkonventionellen Weg, das `FileStream`-Objekt mit einem `BinaryReader`-Objekt als `byte`-Array auszulesen und deren Elemente dann aufzusummieren. Grundsätzlich bietet ein `BinaryReader`-Objekt für jeden der in die Sprache integrierten Datentypen sowie auch für verschiedene Arraytypen `ReadXxx()`-Methoden an. Die Methode für das Lesen von `byte`-Arrays ist `ReadBytes()`.

```
static long CheckSumBinaryRead(string path)
{
  BinaryReader br = new BinaryReader(new FileStream(path, FileMode.Open));
  byte[] buffer = br.ReadBytes((int) br.BaseStream.Length);
  long checkSum = 0;
  for (int i = 0; i < buffer.Length; i++)
    checkSum +=  buffer[i];
  return checkSum;
}
```

Im Gegensatz zu den bisher vorgestellten Strategien eignet sich der Ansatz mit dem `BinaryReader`-Objekt auch dafür, Checksummen auf der Basis anderer Datentypen etwa `int` oder `long` oder gar `float` sowie `double` zu berechnen. Der absolute Wert der Checksumme ist dann natürlich anders und das ist auch gut so, denn die Überprüfung einer Datei auf Korruption wird umso sicherer, je mehr unterschiedliche Arten von Checksummen gebildet und überprüft werden.

Die Methode `CheckSumBinaryReadLong()` berechnet die Checksumme, indem sie `long`-Werte (also jeweils 4 Byte auf einmal) aus dem `Stream`-Objekt liest, die Leseposition je Lesevorgang aber nur um eine Byte-Position verschiebt:

```
static long CheckSumBinaryReadLong(string path)
{
  BinaryReader br = new BinaryReader(new FileStream(path, FileMode.Open));
  long checkSum = -1;
  for (long i = 0; i < br.BaseStream.Length -3; i++)
  {
    checkSum += br.ReadInt32();
    br.BaseStream.Seek(-3, SeekOrigin.Current);
  }
  return checkSum;
}
```

Asynchrone Dateioperationen

Laufzeit ist ein kostbares Gut. Sie will nicht nur von Anwendung zu Anwendung gut verteilt, sondern auch innerhalb einer Anwendung gut genutzt sein. Die Ausführungsgeschwindigkeit Stream-basierter Operationen hängt stark von dem jeweiligen Medium ab. Während Zugriffe auf Dateien, die als `MemoryStream`-Objekte in den Arbeitsspeicher abgebildet wurden, noch weitgehend verlustfrei über die Bühne gehen, können dateisystem- und noch schlimmer netzwerkorientierte Operationen oft mit erheblichen Wartezeiten verbunden sein. Damit solche Operationen nicht die Benutzerschnittstelle lahm legen – auch hier gilt die Regel, dass eine Anwendung spätestens nach einer Zehntelsekunde auf Benutzeraktionen reagieren sollte –, kann man aufwändigere Dateizugriffe *asynchron* im Hintergrund erledigen lassen.

Vom Prinzip her heißt dies, dass die Dateioperation in einen Thread ausgelagert wird und sich dann zur Re-Synchronisation per Rückruffunktion meldet, wenn sie abgeschlossen ist. Auch wenn hier definitiv Threads im Spiel sind, besteht keine Notwendigkeit dafür, explizit mit der Klasse `System.Threading.Thread` zu hantieren (vgl. Teil 5). Erfreulicherweise sieht nämlich die Klasse `Stream` selbst einen Mechanismus für die asynchrone Ausführung von Dateioperationen vor, der Arbeitsthreads aus dem Thread-Pool des Systems nutzt und deshalb besonders effektiv implementiert ist.

Die Methoden für die asynchrone Ausführung von I/O-Operationen sind die Paare `BeginRead()`, `EndRead()` und `BeginWrite()`, `EndWrite()`.

Um ein `FileStream`-Objekt auf asynchrone Lesevorgänge einzurichten, muss ein Konstruktor verwendet werden, der es erlaubt, das entsprechende Flag über den letzten Parameter auf `true` zu setzen.

```
FileStream fStream = new FileStream(path, FileMode.Open, FileAccess.Read,
  FileShare.Read, 1, true);   // asynchroner Zugriff möglich
```

Anstoß einer asynchronen Leseoperation

Der Anstoß einer asynchronen Leseoperation erfolgt dann durch Aufruf der Methode `BeginRead()`, die einen Lesepuffer erwartet, eine optionale Rückruffunktion in Form einer Delegatinstanz und – nicht zu vergessen – einen Parameter des Typs `object` für diese Rückruffunktion. Als Ergebnis liefert die Methode ein Objekt des Typs `IAsyncResult`, das gespeichert werden sollte, falls eine spätere Synchronisation der Leseoperation via `EndRead()` erforderlich ist. Wird keine Rückruffunktion angegeben, sieht der `BeginRead()`-Aufruf mit nachfolgender Synchronisation so aus.

```
IAsyncResult iar = fStream.BeginRead(buffer, 0, buffSize, null, null);
  new AsyncCallback(MyCallBack), job);
...  // Anweisungen, die asychron zu Leseoperation ablaufen
int bytesRead = fStream.EndRead(iar);    // Resynchronisation
```

Die Methode `EndRead()` gibt die Kontrolle erst zurück, wenn die Leseoperation abgeschlossen ist. Sie ist so implementiert, dass sie auf die Signalisierung eines von `BeginRead()` gepflegten Synchronisationsobjekts wartet und dann alle für die asynchrone Leseoperation belegten Ressourcen freigibt.

Es versteht sich, dass sich der Aufwand für eine asynchrone I/O-Operation nur lohnt, wenn eine gewisse Datenmenge im Spiel ist. Als Richtwert gibt Microsoft Byteblöcke ab einer Größe von 64 KB an. Wie aber die Zeitmessung der Beispielanwendung zeigt, muss dieser Wert nicht als Schallgrenze angesehen werden. (Die Online-Dokumentation schreibt zu diesem Thema auch, dass Dateioperationen mit Puffern unterhalb dieser Größe synchron ausgeführt werden – was faktisch nicht stimmt.)

Rückrufe – Verarbeitung in mehreren Schüben

Blickt man in die andere Richtung, wird schnell klar, dass es keinen Sinn macht, die Puffergröße grundsätzlich am Umfang der I/O-Operation zu orientieren. Puffer jenseits von etwa 1 MB für I/O-Operationen zu reservieren, wäre nicht nur eine Verschwendung von kostbarem Arbeitsspeicher, man läuft auch Gefahr, dass der Pufferinhalt seinerseits wieder in der Auslagerungsdatei des Systems landet. In der Regel begrenzt man die Puffergröße daher nach oben hin und füllt bzw. verarbeitet den Puffer gegebenenfalls in mehreren Schüben. Mit anderen Worten: Die I/O-Operation wird in mehrere auf die Puffergröße abgestimmte Teiloperationen aufgeteilt.

Da die Ablaufsteuerung dafür offensichtlich nicht in dem Thread liegen kann, der die Operation ursprünglich gestartet hat – sie würde ja eine Synchronisation mit der ausgelagerten I/O-Operation erforderlich machen – führen `BeginXxx()`-Aufrufe nach getaner Arbeit Rückrufe aus, die eine – aus der Sicht von `BeginXxx()` – synchrone Fortsetzung der Operation ermöglichen. Mechanismen dieser Art sind überall dort anzutreffen, wo asynchrone Vorgänge einer gewissen Pflege bedürfen und sei es nur für Statusmeldun-

gen oder den vorzeitigen Abbruch. (Ein prominentes Beispiel dafür ist die Druckerausgabe; vgl. Teil 3). Spätestens an dieser Stelle kommt also die Delegatinstanz ins Spiel, von der zuvor die Rede war.

Die Definition des Delegaten sieht so aus:

```
delegate void AsyncCallback (IAsyncResult ia);
```

Eine Methode `MyCallBack()` mit diesem Prototyp lässt sich somit zur Instanziierung des Delegaten verwenden und an den `BeginRead()`-Aufruf übergeben:

```
fStream.BeginRead(buffer, 0, buffSize, new AsyncCallback(MyCallBack), job);
```

Bei der Variablen `job` handelt es sich um ein Objekt, das `MyCallBack()` mit Parametern versorgt – und zwar über die `AsyncState`-Eigenschaft des übergebenen `IAsyncResult`-Objekts:

```
void MyCallBack(IAsyncResult ia)
{
  AsyncJob m = (AsyncJob) ia.AsyncState;
  ...
```

Welche Informationen das Objekt `job` überbringt und wie die dahinter stehende Klasse `AsyncJob` im Einzelnen aussieht, richtet sich nach den Erfordernissen der jeweiligen Implementierung.

CheckSumAsyncRead() – Checksumme asynchron berechnen und Status anzeigen

Für das diskutierte Codebeispiel sieht die Klasse `AsyncJob` so aus.

```
class AsyncJob       // Klasse beschreibt asynchrone Leseoperation als Job
{
  public const int BuffSize = 0x0100000;  // 1 MB
  public byte[] buffer;          // Zielpuffer für asychrone Leseoperation
  public int UsedBufSize = 0;    // genutzte Länge des Puffers
  public DateTime startTime;     // Startzeit der Operation
  public Stream baseStream;      // nötig für mehrere Pufferinhalte
  public long checkSum = 0;      // aufaddierte Checksumme

  public AsyncJob(byte[] buffer, int UsedBufSize, DateTime startTime,
    Stream baseStream)
  {
    this.buffer = buffer;
    this.UsedBufSize = UsedBufSize ;
    this.startTime = startTime;
    this.baseStream = baseStream;
  }
}
```

Während die Datenfelder `baseStream`, `buffer` und `bufSize` mehr oder weniger Pflicht sind, um die I/O-Operation von `MyCallBack()` aus fortzusetzen, beher-

bergen die Datenfelder `checkSum` und `startTime` anwendungsspezifische Informationen, die für den abschließenden Aufruf der `SetStatus()`-Methode erforderlich sind.

Die asynchrone Variante der schon diskutierten `CheckSumXxx()`-Methoden ist `CheckSumAsyncRead()`. Sie startet den asynchronen Lesevorgang für den geöffneten Stream über einen `BeginRead()`-Aufruf und stellt dafür einen Puffer (begrenzten Umfangs), eine Rückruffunktion und ein fertig initialisiertes `AsyncJob`-Objekt bereit.

Damit wären alle Elemente vorgestellt, die für den asynchronen Lesevorgang eine Rolle spielen – bis auf die Rückrufmethode. Die `CheckSumAsyncRead()`-Methode sieht so aus:

```
void CheckSumAsyncRead(string path, DateTime startTime)
{
  Stream fStream = null;
  try
  {
    fStream = new FileStream(path, FileMode.Open, FileAccess.Read,
      FileShare.Read, 1, true);
  }
  catch (Exception e)
  {
    statusBarPanel1.Text = e.Message;
  }

  // Puffer für asynchrone Leseoperation anlegen
  int buffSize = AsyncJob.BuffSize;
  if (buffSize > fStream.Length)
    buffSize = (int) fStream.Length;
  byte[] buffer = new Byte[buffSize];

  // Parameterobjekt schnüren und asynchrone Leseoperation ausführen
  AsyncJob job = new AsyncJob(buffer, buffSize, startTime, fStream);
  fStream.BeginRead(job.buffer, 0, buffSize,
    new AsyncCallback(MyCallBack), job);
  // weiter geht es mit MyCallBack, bis die Operation fertig ist.
  // ...
}
```

Da der `BeginRead()`-Aufruf die Kontrolle sofort zurückgibt, erfährt die Reaktivität der Benutzerschnittstelle durch die I/O-Operation keinerlei Einbußen. Die Anwendung *CheckSummen* zeigt dies sehr schön. Nicht nur die per `Timer`-Komponente aktualisierte Uhr in der Statusleiste läuft während der I/O-Operation weiter, der Benutzer kann auch das Fenster verschieben oder dessen Größe verändern, und er kann auch eine weitere Datei öffnen, um deren Checksumme zu berechnen – synchron oder asynchron (in letzterem Fall sogar beliebig viele, wobei dann allerdings die Anzeige in der Statusleiste etwas erratisch wirkt).

Sobald der Arbeitsthread `buffSize` Bytes in den Puffer übertragen hat, ruft er über die Delegatinstanz die Methode `MyCallBack()` auf. `MyCallBack()` hält »die Kugel am Rollen« und findet in der `AsyncState`-Eigenschaft der übergebenen `IAsyncResult`-Instanz alles, was dafür notwendig ist. Im Einzelnen hat sie folgende Aufgaben:

- *Berechnung der Checksumme für den aktuellen Pufferinhalt* – dazu speichert die Rückrufmethode die jeweilige Zwischensumme in dem Feld `checkSum` des `AsyncJob`-Objekts, das die `IAsyncResult`-Instanz über die `AsyncState`-Eigenschaft bereitstellt. Die von `BeginRead()` angelegte `IAsyncResult`-Instanz ist nichts anderes als der Kontext der asynchronen I/O-Operation. Da sie erst im Zuge des `EndRead()`-Aufrufs abgebaut wird, hält sie auch das `AsyncJob`-Objekt über die Dauer der Operation am Leben.
- *Steuerung der I/O-Operation* – ist die Operation bei Aufruf der Rückrufmethode noch nicht abgeschlossen, muss diese dafür sorgen, dass weitere `BeginRead()`-Aufrufe stattfinden, bis der gesamte Stream gelesen ist. Ein `BeginRead()`-Aufruf von der Rückrufmethode aus sieht genauso aus, wie der ursprüngliche Aufruf durch `CheckSumAsyncRead()` – und erfordert vom Prinzip her dieselben Vorbereitungen. Als Rückrufroutine setzt die Methode einfach sich selbst ein. Der Puffer und das `AsyncJob`-Objekt lassen sich aber recyceln. Für die abschließende Teiloperation ist eine von der Puffergröße abweichende Transfergröße `UsedBufSize` zu berechnen, da der Puffer dann in der Regel nicht voll wird.
- *EndRead()-Aufruf* – jeder `BeginRead()`-Aufruf muss durch einen `EndRead()`-Aufruf abgeschlossen werden. Da `CheckSumAsyncRead()` die Kontrolle vollständig auf die Rückrufroutine überträgt, muss diese also auch für den `EndRead()`-Aufruf sorgen. Deren `IAsyncResult`-Parameter stimmt übrigens mit dem Objekt überein, das die `BeginRead()`-Methode beim Start der Operation zurückliefert. Dass der Rückrufmechanismus kein Recycling des Parameters vorsieht, lässt sich am Fehlen einer `BeginRead()`-Überladung erkennen, der man diesen Wert irgendwie mitgeben könnte. Somit fällt der `EndRead()`-Aufruf jedes Mal an – und sollte möglichst früh erfolgen, um eine schnelle Freigabe der Ressourcen zu erreichen. Im vorliegenden Fall ist der Aufruf sicher, sobald das `AsyncJob`-Objekt »im Trockenen« ist.
- *Statusaktualisierung* – da die Rückrufmethode während der I/O-Operation von Zeit zu Zeit die Kontrolle erhält, bietet sie sich in idealer Weise als Plattform sowohl für die routinemäßige und abschließende Statusmeldung an. Außerdem kann sie die Operation auch schlicht nicht fortsetzen, wenn ein Abbruch erwünscht ist.

Hier der Code der Methode:

```
private bool ContinueAsync = true;

// Rückrufmethode. Kommt zum Aufruf, wenn der Puffer voll ist.
// Berechnet die Checksumme, aktualisiert die Statusanzeige
// und fordert den nächsten Puffertransfer an, bis der Stream
// gelesen ist
void MyCallBack(IAsyncResult ia)
{
  AsyncJob m = (AsyncJob) ia.AsyncState;
  m.baseStream.EndRead(ia);
  m.checkSum += ReadStream(new MemoryStream(m.buffer), m.UsedBufSize);
  long remaining = m.baseStream.Length - m.baseStream.Position;
  statusBarPanel1.Text = String.Format("Bytes gelesen: {0:n0} von {1:n0}",
    m.baseStream.Position, m.baseStream.Length);

  if (remaining > 0 && ContiuneAsync)  // weiterlesen?
  {
    if (remaining < AsyncJob.BuffSize)
      m.UsedBufSize = (int) remaining;
    m.baseStream.BeginRead(m.buffer, 0, m.UsedBufSize,
      new AsyncCallback(MyCallBack), m);
  }
  else    // fertig, Statusmeldung
    SetStatus(m.startTime, m.baseStream.Length, m.checkSum);
}

private void Form1_Closed(object sender, System.EventArgs e)
{
  ContinueAsync = false;
}
```

Teil 3 Windows-Anwendungen

Kapitel 12: Einführung in die Windows-Programmierung 409
Kapitel 13: Formulare 439
Kapitel 14: Grafik 459
Kapitel 15: Steuerelemente 539
Kapitel 16: Steuerelemente selbst implementieren 633
Kapitel 17: Dialogfelder 673
Kapitel 18: Zwischenablage und Drag&Drop 725

In diesem Teil finden Sie eine praxisnahe Übersicht über das gesamte Feld der formularorientierten Programmierung von Windows-Anwendungen. Die einzelnen Kapitel stellen Entwurfs- und Programmiertechniken vor, die nicht nur Ansatzpunkte und Lösungen für alltägliche Problembereiche der C#-Programmierpraxis bieten, sondern auch kreative Anregungen für das eigene Anwendungsdesign geben sollten. Für nahezu alle Codebeispiele werden Übungen vorgeschlagen, die zu einer interaktiven Erarbeitung der Beispielprojekte und zum vertiefenden Transfer des Themenbereichs einladen.

Themenschwerpunkte bilden einerseits das Formulardesign und die formularbasierte Programmierung im Allgemeinen und andererseits der Einsatz der Steuerelemente aus der Toolbox sowie die Programmierung eigener Steuerelemente und Dialoge. Weiterhin finden Sie in diesem Teil auch vertiefende Programmiertechniken wie die Steuerelement- und Formularvererbung, die Einbindung der Zwischenablage und die Durchführung von Drag&Drop-Operationen.

12 Einführung in die Windows-Programmierung

Um es gleich vorweg zu nehmen: So einfach wie mit C# unter .NET war Windows-Programmierung noch nie. Und das liegt wohlgemerkt nur zum kleineren Teil an der Sprache C# und an .NET selbst. Der größere Teil der Lorbeeren gebührt der integrierten Entwicklungsumgebung Visual Studio .NET. Mit ihr wird Programmierung im Allgemeinen und Windows-Programmierung im Besonderen wahrlich zum Erlebnis – zum schnellen Erfolgserlebnis.

VS.NET verfügt über fünf Einrichtungen, die dem Programmierer seine Tätigkeit so angenehm wie nur möglich machen sollen und ihm situationsgerecht alle relevanten Informationen unauffällig präsentieren:

- *Syntaxprüfung und Codeparsing während der Quelltexteingabe* – der Quelltexteditor moniert Fehler, wie unpaarige Klammern, vergessene Semikolons, unbekannte Bezeichner, Verstöße gegen das Typsystem etc. bereits bei der Eingabe. Das sorgt von Anfang an für eine relativ hohe Güte des Quelltexts und reduziert die Anzahl der erforderlichen Compilerläufe bis zum syntaktisch korrekten Programm.
- *Automatisches Einblenden von Memberlisten und kontextsensitive Eingabehilfe* – das mühsame Nachschlagen, welche Klasse in welchem Namensraum steckt, welches Objekt welche Member hat und wie diese im Einzelnen notiert oder welche Parameter erwartet werden, ist vorbei. Aufgrund des Codeparsings während der Eingabe kann VS.NET auch die Elemente selbstdefinierter Klassen und Namensräume anzeigen (Inline-Hilfe).
- *Prototyp- und Parameterinformationen mit Kurzbeschreibung für jede überladene Variante einer Methode* – VS.NET blendet für alle Elemente kontextsensitiv die Prototyp- und Parameterinformationen als Tooltip-Fenster ein. In den meisten Fällen finden Sie sogar eine Kurzbeschreibung des jeweiligen Elements, wenn entsprechende Dokumentationskommentare mit `<summary>`- und `<param>`-Tags dafür vorhanden sind. Das gilt auch für selbstdefinierte Elemente.
- *Bidirektionales visuelles Design* – Designer und Quelltexteditor arbeiten verschränkt, so dass sich Änderungen im Design sofort im Quelltext niederschlagen und umgekehrt.

- *Farbliche Kennzeichnung von C#-Schlüsselwörtern und Kommentaren* – Fehler wie falsch gesetzte Kommentare oder falsch geschriebene Schlüsselwörter, fallen umgehend ins Auge.

Wenn Sie von einer anderen Programmiersprache wie Visual Basic oder Delphi her kommen, werden Sie sich sofort zuhause fühlen und auch mit den wesentlichen Einrichtungen der integrierten Entwicklungsumgebung in kürzester Zeit vertraut sein. Nur, in Visual Studio ist eben alles ein wenig großartiger, perfekter, informativer und noch besser durchdacht.

Als C/C++-Programmierer hingegen werden Sie es gewohnt sein, für Windows-Anwendungen einiges an Mühsal auf sich zu nehmen – denken Sie nur an die bereits einige hundert Zeilen umfassenden Codegerüste einer gewöhnlichen per Assistent generierten MFC-Anwendung mit Dokument-/Ansicht-Architektur. In Form der Kombination C#, .NET-Klassenbibliothek und Visual Studio .NET wird sich Ihnen eine völlig neue Welt auftun, die nicht nur einfacher und unbelasteter vom zweifellos vorhandenen Überbau ist, sondern endlich den Blick für das Wesentliche freihält. Angefangen vom visuellen Design, das sich nun erstmalig eins-zu-eins im Code wiederfindet und umgekehrt (Schluss mit der »Schattenwirtschaft«) bis hin zu dem völlig ungewohnten Sicherheitsgefühl, dass man »wirklich gut sein muss«, um das System aus dem Tritt zu bringen.

Während man als Benutzer vor noch gar nicht langer Zeit Hunderte von Tastenkürzeln auswendig lernen und einen guten Teil der Programmlogik begriffen haben musste, um mit einem Programm wie Word Perfect oder Word für DOS halbwegs vernünftig umgehen zu können, brachte die Windows-Programmierung eine Reihe neuer Denkweisen und Anforderungen für die Gestaltung von Anwendungen mit sich:

- *Die Bedienung einer Anwendung sollte sich dem Benutzer weitgehend intuitiv erschließen* – Mittel dafür sind: anwendungsübergreifende Einrichtungen wie standardisierte Steuerelemente und Dialoge, Zwischenablage, Drag&Drop und eine weitgehende Konventionalisierung von Bedienabläufen (Mausbelegung, Kontextmenü, Menüaufbau etc.) und der logischen Organisation der grafischen Benutzeroberfläche (Optionsfestlegung über Dialogfelder, Hilfeeinrichtungen und -systematik etc.)
- *Der Benutzer sollte zu jedem Zeitpunkt sehen, was er tut* – Mittel dafür sind: die WYSIWYG-Darstellung (What You See Is What You Get) der bearbeiteten Daten und getroffenen Einstellungen, Status- und Fortschrittsanzeigen.
- *Der Benutzer sollte zu jedem Zeitpunkt sehen, welche Möglichkeiten er hat* – Mittel dafür sind: die strikte Menüführung, Symbolleisten, Werkzeugsammlungen, Schaltflächen mit Quickinfos, Steuerelemente, die ihren Zustand visuell darstellen, usw.

- *Eine Anwendung sollte dem Benutzer zu jedem Zeitpunkt ein Maximum an Flexibilität hinsichtlich der Bedienung bieten* – Mittel dafür sind: Menübefehle samt Tastaturkürzel für die reine Tastaturbedienung in Haupt- und Kontextmenüs, Steuerelemente wie Bildlaufleisten und Symbolschaltflächen, frei programmierbare Schnelltasten für den direkten Zugriff auf Programmbefehle usw.
- *Die Anwendung sollte jederzeit auf den Benutzer reagieren* – Mittel dafür sind: die ereignisbasierte Benutzerschnittstelle (gegebenenfalls Aufruf von `DoEvents()`), Auslagerung von Hintergrundaktivitäten in eigene Threads.

All das werden Sie nur zu gut von anderen Windows-Anwendungen her kennen – und das wird Ihnen die Windows-Programmierung vielleicht als uferloses, wenn nicht gar aussichtsloses Unterfangen erscheinen lassen. Das Wort »uferlos« ist sicher treffend, wenn man es ins Positive wendet und auf das Möglichkeitspotenzial bezieht, zumal die Liste alles andere als vollständig ist. Die Vokabel »aussichtslos« ist hingegen definitiv fehl am Platz. Im Gegenteil: Die Windows-Programmierung mit C# ist nicht nur in ihren Grundzügen schnell erlernt, auch der Weg zur professionellen Anwendung ist weitgehend hürdenlos.

Die meisten der genannten Funktionalitäten bietet die Entwicklungsumgebung bereits als fertige Bausteine an, nämlich als Komponenten der Toolbox. Zudem sorgt sie bereits beim visuellen Design automatisch für eine Integration der Komponenten in den Code und für eine fertige »Verdrahtung« der im Angebot stehenden Aktionen. Als Entwickler müssen Sie nur noch die Entscheidung treffen, welche der Aktionen Ihre Anwendung tatsächlich verwendet, und wie sich diese im Einzelnen auf die Belange Ihres Programms auswirken sollen. Der für die Implementierung anfallende Programmieraufwand hält sich in Grenzen und ist im Allgemeinen speziell auf die einzelne Aktion bezogen. Der jeweilige Code und fügt sich fast wie von selbst als Lösung eines von vielen diskreten Teilproblemen in das Gesamtprogramm ein. Das Programm ist von Beginn an lauffähig und bleibt es auch im Verlauf der Entwicklung, sofern Sie beim schrittweisen Design bleiben, also Aktion für Aktion implementieren und nicht zu viele Baustellen gleichzeitig eröffnen.

Dieses Kapitel stellt das Prinzip »Windows-Programmierung« vor. Dazu diskutiert es – vornehmlich anhand der Benutzerschnittstellen für Tastatur und Maus – die ereignisbasierte Programmierung und leuchtet aus, an welcher Stelle sich beim visuellen Design und bei der Implementierung neuer Aktionen was verändert.

12.1 Das Codegerüst stellt sich vor

Selbst wenn Sie Kapitel 3, »Erste Schritte mit Visual Studio .NET«, übersprungen haben, wird es Ihnen sicher nicht schwer fallen, ein neues Projekt mit dem Typ *Windows-Anwendung* anzulegen und ein paar Steuerelemente aus der TOOLBOX darauf zu platzieren. In diesem Abschnitt werden Sie das Codegerüst genauer kennen lernen und auch die Veränderungen, die eine Bearbeitung des Formulardesigns im Designer auf der Codeseite mit sich bringen.

12.1.1 Codegerüst anlegen, kompilieren und starten

Am besten, Sie vollziehen die folgenden Arbeitsgänge einfach parallel am eigenen System nach, da sie mit jeder Windows-Anwendung im einen oder anderen Zusammenhang anfallen:

1. *Legen Sie ein neues Windows-Projekt an.* Wählen Sie dazu den Projekttyp WINDOWS-ANWENDUNG und geben Sie den Namen `Codegerüst` an. Der schnellste Weg zum Dialog NEUES PROJEKT führt übrigens über die Tastenkombination Strg+N. Nach Auswahl des Projekttyps und Änderung des vorgegebenen Projektnamens (sowie gegebenenfalls des Speicherorts), bestätigen Sie den Dialog mit OK. VS.NET schließt dann ein gegebenenfalls in Bearbeitung befindliches Projekt, generiert das Codegerüst der neuen Anwendung und meldet sich mit dem Designerfenster.

2. *Schalten Sie in die Codeansicht um.* Noch schneller als der Befehl CODE ANZEIGEN im Kontextmenü des Formularentwurfs bzw. der Datei *Form1.cs* im Fenster PROJEKTMAPPEN-EXPLORER ist übrigens der Befehl F7 zum Wechsel vom Formular zum Code. Sie sehen nun das Codegerüst. Für eine Windows-Anwendung ist es recht übersichtlich – und in dieser Form bereits lauffähig.

3. *Kompilieren und starten Sie das Programm versuchshalber.* Um das Programm nur zu kompilieren und auf Syntaxfehler zu testen, benutzen Sie einen der Befehle aus dem Menü ERSTELLEN (alternativ ⇧+Strg+B). Um es zu starten und erforderlichenfalls automatisch zu kompilieren, finden Sie die Befehle STARTEN (F5) und STARTEN OHNE DEBUGGER ⇧+F5 im Menü DEBUGGEN.

4. *Testen Sie die inhärente Funktionalität des Programms kurz aus.* Da das Codegerüst (noch) fehlerfrei ist, sollten Sie nach Start des Programms ein leeres Formular auf dem Bildschirm sehen, das bereits eine voll funktionierende Titelleiste aufweist, also ein Systemmenü und die gewohnten Schaltflächen enthält. Das Formular lässt sich verschieben und in der Größe verändern, es zeichnet sich neu, wenn es verdeckt wird, erscheint in der Taskleiste usw. – wie ein richtiges Windows-Pro-

gramm eben. Beenden Sie schließlich das Programm – beispielsweise über die Tastenkombination Alt+F4.

5. *Durchlaufen Sie das Programm im Einzelschrittmodus des Debuggers.* Der Tastenbefehl F11 startet das Programm gleichfalls im Debugger, ermöglicht es aber, den Programmablauf schrittweise zu verfolgen. (Auch in diesem Fall kompiliert VS.NET das Programm erforderlichenfalls zuerst.) Um das vom Debugger an irgendeiner Stelle angehaltene Programm zwangsweise zu beenden und in den Bearbeitungsmodus des Texteditors zu gelangen, geben Sie den Menübefehl DEBUGGEN/DEBUGGEN BEENDEN oder verwenden ⇧+F5.

Wie erwartet, beginnt die Ausführung mit der statischen Methode `Main()` der Klasse `Form1`. Die erste und einzige Anweisung darin lautet:

```
Application.Run(new Form1());
```

Die statische Methode `Application.Run()` startet eine standardmäßige Nachrichtenschleife für die Anwendung. Die hier benutzte Überladung erwartet als Parameter ein Formularobjekt, das sie als Hauptformular der Anwendung und primären Empfänger aller die GUI (grafische Benutzerschnittstelle) betreffenden Ereignisse in die Nachrichtenschleife einbindet und per `Show()`auch gleich zur Anzeige bringt. Dass ein Formular tatsächlich über eine eigene Nachrichtenschleife verfügt und das Hauptformular deshalb Hauptformular ist, weil es mit der Nachrichtenschleife des Anwendungsobjekts verbunden ist – und diese insbesondere beenden kann –, verrät der Versuch, die obige Zeile durch folgende Zeilen zu ersetzen.

```
new Form1().Show();
Application.Run();
```

Die parameterlose Überladung von `Run()` eignet sich augenscheinlich für den Start von formularlosen Windows-Anwendungen bzw. Threads. Nach außen hin benimmt sich das Programm tatsächlich nicht anders als zuvor. Das Formular reagiert auf Benutzeraktionen und lässt sich wie gewohnt schließen. Tatsächlich bleibt die Nachrichtenschleife des Anwendungsobjekts dabei aber weiterhin aktiv, und Sie müssen den Prozess des Programms explizit über den Debugger beenden bzw. – falls der Start mit Strg+F5 geschah – über den Windows-Taskmanager abschließen (in dessen Liste Sie dann nach *Codegerüst.exe* suchen sollten.)

Ein erneutes F11 führt in die Konstruktion des Formularobjekts, dessen Klasse wie alle Formulare von der Basisklasse `System.Windows.Forms.Form` abgeleitet ist. Der größte Teil des restlichen Codes ist dem Konstruktor der Klasse `Form1()` untergeordnet. Zunächst initialisiert der generische Konstruktor das momentan noch einzige Datenfeld `components`:

```
private System.ComponentModel.Container components = null;
```

Name und Datentyp dieser Objektvariable verraten bereits, dass sie für die Verwaltung von Komponenten vorgesehen ist, die sich dem Formular zuordnen lassen.



.NET-Komponenten sind, leicht vereinfacht gesagt, spezifisch für .NET weiterentwickelte Varianten von COM-Objekten, die auf die Belange und Möglichkeiten der verwalteten Umgebung zugeschnitten sind – das .NET-Gegenstück der ActiveX-Komponente also. .NET sieht Mechanismen vor, die es gestatten, .NET-Komponenten unter COM bzw. COM+ einzusetzen und umgekehrt.

Formulare verwalten Komponenten und Steuerelemente getrennt voneinander, weil Komponenten erstens nicht von der gemeinsamen Basisklasse `System.Windows.Forms.Control` abstammen (sondern von `System.ComponentModel.Component`), weshalb sie sich nicht über die von `Form` vererbte `Controls`-Auflistung verwalten lassen; zweitens, weil Komponenten auch explizit entsorgt werden müssen. Ein kurzer Blick auf die `Dispose()`-Methode im Codegerüst verrät, dass dies auch passiert.

```
protected override void Dispose( bool disposing )
{
  if( disposing )
  {
    if (components != null)
    {
      components.Dispose();
    }
  }
  base.Dispose( disposing );
}
```

Der nächste Debuggerschritt führt in den Konstruktor `Form1()`, der die private Methode `InitializeComponent()` aufruft. Die Namensgebung ist übrigens so zu verstehen, dass das Formular als Komponente initialisiert wird, inklusive aller ihm über den Designer zugeordneten Elemente – und nicht etwa dahingehend, dass hier nur Komponenten initialisiert werden. `InitializeComponent()` entstammt nicht der Basisklasse; die Definition findet sich vielmehr in dem Bereich, den der Editor hinter dem Quelltextknoten `Windows Form Designer generated code` versteckt. Dieser Knoten ist standardmäßig nicht entfaltet, weil der Code darin automatisch vom Designer generiert und auch gepflegt wird. Zudem lädt der Kommentar, der nach Entfaltung des Knotens als erstes ins Auge sticht, nicht unbedingt dazu ein, hier Hand anzulegen – ein Verbot, das in der Praxis aber eher darin erinnert, dass nicht alles so heiß gegessen werden muss, wie es gekocht wird. Dazu aber gleich noch mehr.

Werfen Sie erst einen Blick auf den Codeblock, den eine `#region`-Präprozessordirektive zusammenfasst.

```
#region Windows Form Designer generated code
/// <summary>
/// Erforderliche Methode für die Designerunterstützung.
/// Der Inhalt der Methode darf nicht mit dem Code-Editor geändert werden.
/// </summary>
private void InitializeComponent()
{
  //
  // Form1
  //
  this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
  this.ClientSize = new System.Drawing.Size(292, 272);
  this.Name = "Form1";
  this.Text = "Form1";
}
#endregion
```

Wie zu sehen, erhält das Formularobjekt hier eine Größe, einen Namen und einen Text für die Titelleiste. Schlagen Sie am besten doch gleich einmal über die Stränge: Ändern Sie die Werte für die Eigenschaft `ClientSize` und schalten Sie in den Designer, um zu sehen, wie übel er Ihnen das nimmt. Sie werden überrascht sein: Der Designer liest die Methode und spiegelt die Änderung sofort im Entwurf wider. Umgekehrt spiegelt die Methode Änderungen sofort eins-zu-eins wider, die Sie im Entwurf oder im EIGENSCHAFTEN-Fenster des Formulars vornehmen. Auch das Löschen oder das Hinzufügen von Initialisierungen nimmt der Designer gelassen zur Kenntnis und versucht es im Entwurf umzusetzen. Aber Vorsicht: Obwohl es prinzipiell möglich ist, die Methode – nach der Systematik des Designers – auch manuell zu erweitern, sollten Sie im Allgemeinen davon absehen, da der Designer nur solche Codezeilen in der Methode stehen lässt, die er auch interpretiert. Der richtige Platz für eigenen Initialisierungscode ist nach dem Aufruf der `InitializeComponent()`-Methode im Konstruktor.



Der Designer bildet den Formularentwurf bei jedem Umschalten auf den Quelltextblock unter dem Knoten `Windows Form Designer generated code` *sowie in dem Datenfeld-Vereinbarungsteil ab und umgekehrt. Dies geht so weit, dass Sie den kompletten Quelltext eines Formulars sogar per Zwischenablage in eine andere .cs-Datei importieren können und nach dem Umschalten in den Designer sofort den passenden Entwurf präsentiert bekommen. Der Designer findet also die gesamte Information, der er für die Darstellung des Formularentwurfs benötigt, im Quelltext. Bitmaps und andere, ausschließlich in binärer Form vorliegende Ressourcen werden bei diesem Vorgang allerdings nicht »transportiert«.*

Durchlaufen Sie die Methode weiter mit dem Debugger, bis die Ausführung wieder in der Methode `Main()` angekommen ist. Das Hauptformular ist nun initialisiert, und der `Run()`-Aufruf kann erfolgen. Viel weiter kommen Sie mit dem Debugger im Einzelschrittmodus nun nicht mehr, da der Code des Programmgerüsts erschöpft ist. (Falls Sie auch die Ereignisbehandlung weiter

im Debugger verfolgen wollen, müssen Sie in die entsprechenden Routinen Haltepunkte setzen.) `Run()` eröffnet die Nachrichtenschleife der Anwendung und bringt das Formular über einen internen Aufruf der Methode `Show()` auf den Bildschirm. Die Benutzerschnittstelle der Windows-Anwendung ist damit aktiv und wartet auf Ereignisse. Die Implementierung der gesamten bereits verfügbaren Funktionalität steckt übrigens in der Basisklasse und tritt (vorerst) nicht weiter zu Tage.

12.1.2 Codegerüst mit dem Designer erweitern

Um zu sehen, was passiert, wenn Sie Steuerelemente und Komponenten in den Formularentwurf einfügen, schalten Sie zum Designer und bestücken das Formular mit einer `Timer`-Komponente und einem `Button`-Steuerelement. Die Namensvorgaben können Sie übernehmen oder auch über die `(Name)`-Eigenschaft im EIGENSCHAFTEN-Fenster frei wählen. Der Designer benennt damit die Instanzdatenfelder für den Zugriff auf diese Objekte. Ändern Sie auf alle Fälle die `Text`-Eigenschaft des `Button`-Steuerelements in »Beenden«.

Die Codeansicht offenbart, was der Designer daraus gestrickt hat. Erstens finden sich zu Beginn des `class`-Blocks zwei weitere Datenfelder

```
private System.Windows.Forms.Timer timer1;
private System.Windows.Forms.Button button1;
```

Zweitens ist auch `InitializeComponent()` etwas gewachsen:

```
private void InitializeComponent()
{
  this.components = new System.ComponentModel.Container();
  this.timer1 = new System.Windows.Forms.Timer(this.components);
  this.button1 = new System.Windows.Forms.Button();
  this.SuspendLayout();
  //
  // button1
  //
  this.button1.Location = new System.Drawing.Point(184, 200);
  this.button1.Name = "button1";
  this.button1.Size = new System.Drawing.Size(88, 40);
  this.button1.TabIndex = 0;
  this.button1.Text = "Beenden";
  //
  // Form1
  //
  this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
  this.ClientSize = new System.Drawing.Size(292, 272);
  this.Controls.AddRange(new System.Windows.Forms.Control[] {
    this.button1});
  this.Name = "Form1";
  this.Text = "Form1";
  this.ResumeLayout(false);
}
```

Zu Beginn des Methodenblocks finden sich Instanziierungen für alle drei Datenfelder. Die Instanziierung eines Objekts für das `components`-Datenfeld ist notwendig, weil das `Timer`-Steuerelement als fensterloses Gebilde nicht von `Control` abstammt und über das Containerobjekt für .NET-Komponenten verwaltet wird. Beachten Sie in diesem Zusammenhang den Konstruktoraufruf `Timer()`. Der Rest ist im Wesentlichen Kleinkram bis auf die Aufrufe `SuspendLayout()` und `ResumeLayout()`: Diese beiden Methoden machen es möglich, mehrere Eigenschaften eines Steuerelements oder einer Komponente en bloc zu setzen, ohne sofortige Rückwirkung (in Form von `Layout`- und `Paint`-Ereignissen) auf das Objekt. Der interessanteste Aufruf dabei ist zweifelsohne:

```
this.Controls.AddRange(new System.Windows.Forms.Control[] {
                                                        this.button1});
```

Er zeigt, dass die Verwaltung der Schaltfläche über die ererbte `Controls`-Auflistung geschieht.

Behandlungsmethode für Standardereignis automatisch ergänzen

Nun fehlt zum Verständnis des Codegerüsts eigentlich nur noch das Prinzip, wie der Designer Ereignisbehandlungsmethoden verdrahtet, über die Sie die Aktionen Ihres Programms implementieren.

Dazu schalten Sie in den Designer und doppelklicken auf die Schaltfläche im Formularentwurf. Der Designer verdrahtet daraufhin das Standardereignis `Click` des `Button`-Objekts und generiert eine Behandlungsmethode, in deren Rumpf Sie nun einen `Close()`-Aufruf für das Formular ergänzen:

```
private void button1_Click(object sender, System.EventArgs e)
{
  this.Close();  // manuell eingesetzter Code
}
```

Der zweite Teil der »Verdrahtung« besteht darin, dass der Designer dem von `Form` ererbten `event`-Datenfeld `Click` einen Delegaten des Typs `EventHandler` hinzugefügt hat, der die soeben bearbeitete Methode `button1_Click()` als Behandlungsmethode für dieses Ereignis registriert:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Starten Sie nun das Programm und überprüfen Sie, ob es sich über die Schaltfläche beenden lässt.

Behandlungsmethoden für andere Ereignisse einfügen

Der Designer kennt für jedes Steuerelement (bzw. Komponente) ein Standardereignis, für das er bereits eine Behandlungsmethode einfügt, wenn Sie in der Entwurfsansicht nur darauf doppelklicken. Behandlungsmethoden für andere Ereignisse fügen Sie über das EIGENSCHAFTEN-Fenster ein. Schal-

ten Sie darin auf die Ansicht EREIGNISSE (Blitz-Symbol) und suchen Sie die Zeile für das betreffende Ereignis. Sie haben nun drei Möglichkeiten. Sie können

- auf die Zeile doppelklicken – in diesem Fall generiert der Designer automatisch einen Methodenbezeichner nach dem Benennungsschema *Komponentenname_Ereignis*
- einen eigenen Methodenbezeichner in die Zelle neben dem Ereignisnamen eingeben
- das Kombinationsfeld der Zelle neben dem Ereignisnamen aufklappen und – sofern vorhanden – aus dem Listenfeld eine bereits existierende Behandlungsmethode auswählen.

Der Designer vereinbart mit diesem Bezeichner nun ein zu dem Ereignis passendes Methodengerüst und ergänzt einen Delegaten, der die Methode mit dem Ereignis assoziiert. Sie ist daher sofort »scharf«.

Behandlungsmethoden für Ereignisse entfernen

Wenn Sie Ereignisbehandlungsmethoden löschen oder entfernen wollen, müssen Sie also an zwei Stellen im Quelltext arbeiten. Zum Lösen der »Verdrahtung« gibt es folgende Möglichkeiten:

- Löschen Sie die entsprechende Zeile in `InitializeComponent()` oder kommentieren Sie sie aus. Lassen Sie sich dabei nicht von dem Kommentar vor dieser Methode vor diesem Schritt abschrecken. Wenn Sie nur Änderungen vornehmen, die der Designer versteht, stellt die manuelle Bearbeitung dieser Methode kein Problem dar. Es hat sich auch bewährt, nur die Behandlungsmethoden zu löschen oder umzubenennen und den Compiler die Verdrahtungen suchen zu lassen. Doppelklicks auf die Fehlermeldungen ermöglichen dann das schnelle Auffinden der zu löschenden Zeilen.
- Schalten Sie in die Entwurfsansicht und löschen Sie im EIGENSCHAFTEN-Fenster der betroffenen Komponente den Eintrag für das Ereignis.

12.1.3 Maus

So einfach der Umgang mit der Maus ist, das sich aus ihrem Funktionsumfang ergebende Möglichkeitspotenzial ist gewaltig.

Informationen über die Maus einholen

Damit sich Ihr Code auf die Art und Ausstattung der am System angeschlossenen Maus sowie die in der Systemsteuerung eingestellten Mausparameter einstellen kann, besteht die Möglichkeit, bei der Klasse `SystemInformation`

verschiedene Informationen über die Maus einzuholen. So ist es beispielsweise üblich, dass Programme, die auf eine 3-Tasten-Maus ausgelegt sind, bei angeschlossener 2-Tasten-Maus eine Simulation der mittleren Taste über die beiden Maustasten zulassen. Natürlich sollte sich ein Programm auch darauf einstellen, wenn ein Benutzer mit vertauschter Maustastenbelegung arbeitet.

Tabelle 12.1 gibt einen Überblick über die wichtigsten Eigenschaften der Mauskonfiguration des Systems. Tabelle 12.2 zeigt, wie Sie eine Beschreibung des aktuellen Mauszustands erhalten.

Tabelle 12.1: Systeminformationen über die Maus

`SystemInformation`-Eigenschaft	Bedeutung
`bool MousePresent {get;}`	`true`, wenn eine Maus vorhanden ist
`int MouseButtons {get;}`	Anzahl der Maustasten
`bool MouseButtonsSwapped {get;}`	`true`, wenn die Bedeutung der Maustasten vertauscht ist (für Linkshänder)
`bool MouseWheelPresent {get;}`	`true`, wenn ein Mausrad vorhanden ist
`int MouseWheelScrollLines {get;}`	Anzahl der Zeilen, die ein `MouseWheel`-Ereignis rollt (»Übersetzungsverhältnis« des Mausrads)
`int DoubleClickTime {get;}`	Zeitintervall in Millisekunden, in dem zwei Mausklicks als Doppelklick zu werten sind
`Size DoubleClickSize {get;}`	Pixelbereich, in dem zwei Mausklicks stattfinden müssen, damit noch ein Doppelklick erkannt wird

Tabelle 12.2: Eigenschaften, die den Mauszustand beschreiben

Eigenschaft	Bedeutung
`static Point Control.MousePosition {get; }` oder alternativ `static Point Cursor.Position {get; set; }`	Mausposition in Bildschirmkoordinaten (wobei `Position` im Gegensatz zu `MousePosition` auch das Setzen der Mausposition ermöglicht)
`static MouseButtons Control.MouseButtons{get;}`	Objekt, das den Zustand der Maustasten als `MouseButtons`-Bitvektor beschreibt
`Cursor Cursor {get; set;}`	Form des Mauszeigers

Mauszeigerform

`Control`-Objekte – also Formulare, Steuerelemente etc. – besitzen eine Eigenschaft `Cursor`, die der gleichnamigen Klasse `Cursor` angehört. Dieses Objekt beschreibt die Form, die das System dem Mauszeiger gibt, so lange er sich im Fenster des `Control`-Objekts aufhält. Um die jeweils aktuelle Mauszeigerform in Erfahrung zu bringen oder zu ändern, steht Ihnen die statische Eigenschaft `Cursor.Current` zur Verfügung. Durch Instanziierung der Klasse `Cursor`, beispielsweise unter Angabe einer *.cur*-Datei, lassen sich auch eigene Mauszeigerformen darstellen (wobei animierte Varianten im *.ani*-Format leider nicht erlaubt sind). In den meisten Fällen werden Sie allerdings mit dem reichen Vorrat an vordefinierten Mauszeigerformen auskommen (28 an der Zahl), den die Klasse `System.Windows.Forms.Cursors` bereitstellt. Tabelle 12.3 gibt einen Überblick über verschiedene interessante statische Methoden und Eigenschaften der Klasse `Cursor`.

Tabelle 12.3: Interessante statische Eigenschaften und Methoden der Klasse `Cursor`

Eigenschaft/Methode	Bedeutung
`static Cursor Cursor.Current {get; set;}`	Aktueller Mauszeiger
`static Point Cursor.Point {get; set;}`	Aktuelle Position des Mauszeigers
`static void Cursor.Draw()`	Zeichnet den Mauszeiger auf die Fensterfläche
`static void Cursor.Hide()`	Blendet den Mauszeiger aus, wenn der mit jedem Aufruf herabgesetzte interne Zähler 0 ist.
`static void Cursor.Show()`	Blendet den Mauszeiger ein, wenn der mit jedem Aufruf erhöhte interne Zähler größer 0 ist.

Primäre Mausereignisse

Der größte Teil der Ereignisse, den eine typische Windows-Anwendung im Verlauf ihrer Ausführung zu sehen bekommt, stammt unmittelbar oder mittelbar von der Maus. Tabelle 12.4 listet die vier primären Mausereignisse auf, aus denen letztlich alle anderen mausbasierten Ereignisse abgeleitet sind.

Tabelle 12.4: Die vier primären Mausereignisse

Mausereignis	Tritt auf bei	Typ des Ereignisobjekts
`MouseDown`	Drücken einer Maustaste	`MouseEventArgs`
`MouseUp`	Loslassen einer Maustaste	`MouseEventArgs`
`MouseMove`	Bewegen der Maus	`MouseEventArgs`
`MouseWheel`	Betätigung des Mausrads	`MouseEventArgs`

Die mit diesen Ereignissen assoziierte Information wird einer dafür registrierten Ereignisbehandlungsmethode als `MouseEventArgs`-Parameter übergeben. Das Objekt macht eine Aussage darüber, wo das Ereignis stattfand und welchen Zustand die Maustasten und das Mausrad (nur relative Bewegung) in diesem Moment aufweisen. Tabelle 12.5 gibt einen Überblick über die Eigenschaften dieser Klasse.

Tabelle 12.5: Wichtige Eigenschaften der Klasse `MouseEventArgs`

Eigenschaft	Bedeutung
`MouseButtons Button {get;}`	`enum`-Wert, der den Mauszustand beschreibt
`int Clicks {get;}`	Bei Doppelklick 2, sonst 1 oder 0
`int Delta {get;}`	Relative Veränderung des Mausrads
`int X {get; set;}`	x-Koordinate der Mausposition
`int Y {get; set;}`	y-Koordinate der Mausposition

Die `MouseButtons`-Aufzählung definiert sieben Konstanten, die Sie als Masken verwenden müssen, um die Zustände der Maustasten bzw. des Mausrads aus dem Wert der `Button`-Eigenschaft herauszudestillieren:

```
enum System.Windows.Forms.MouseButtons
{
  None     = 0x00000000,
  Left     = 0x00100000,   // primäre Maustaste (i.d R. links)
  Right    = 0x00200000,   // sekundäre Maustaste
  Middle   = 0x00400000,
  XButton1 = 0x00800000,   // für die IntelliMouse mit fünf Tasten
  XButton2 = 0x01000000
};
```

Wie das konkret geht, entnehmen Sie dem folgenden Codebeispiel.

Codebeispiel – Gummiband

Das kleine Programm *Gummiband* behandelt drei der vier primären Mausereignisse und zeichnet ein »Gummiband« in das Formular, wenn die Maus mit gedrückter linker Taste gezogen wird. Abbildung 12.1 zeigt, wie Sie aus dem Designer heraus eine neue Behandlungsmethode für ein Ereignis in das Codegerüst der Anwendung einfügen.

Die Formularklasse benötigt ein Instanzdatenfeld, um das Gummiband zu repräsentieren. Ein `Rectangle`-Objekt ist hierfür bestens geeignet.

```
private Rectangle RubberBand;   // Gummiband
```

Die Initialisierung von `RubberBand` ist die Aufgabe von `Form1_MouseDown`:

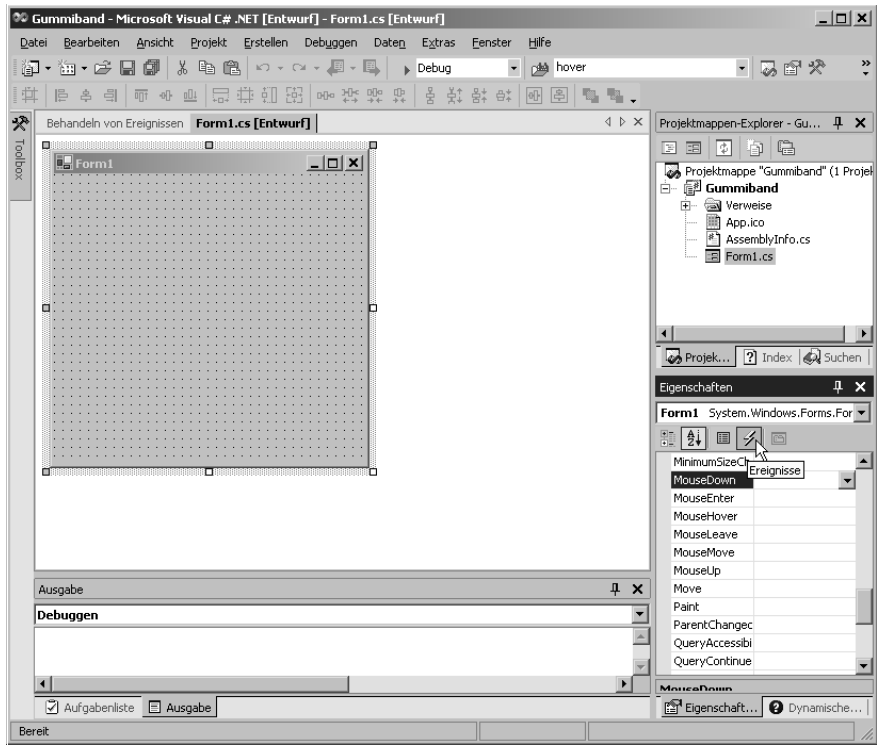


Abbildung 12.1: Behandlungsmethode für Mausereignis in das Codegerüst einfügen

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
  if ( (e.Button & MouseButtons.Left) > 0)  // linke Maustaste?
  {
    this.RubberBand.X = e.X;
    this.RubberBand.Y = e.Y;
  }
}
```

Beachten Sie die Maskierung der `Button`-Eigenschaft mit `MouseButtons.Left`. Schließlich soll die Methode nur bei der linken Maustaste aktiv werden. Freilich hätte man hier auch mit == arbeiten können. Ein Kriterium für »rechte und mittlere Maustaste gedrückt« würde so aussehen:

```
e.Button == (MouseButtons.Middle | MouseButtons.Right)
```

Für die Behandlung der auf `MouseDown` folgenden `MouseMove`-Ereignisse ist die linke obere Ecke des Gummibands nun festgelegt.

Obwohl ein Formularobjekt seine Zeichenoperationen standardmäßig im Rahmen der `Paint`-Behandlung erledigt (hierzu im nächsten Kapitel mehr), ist es durchaus üblich, kleinere Zeichenoperationen für Animationen auch vor Ort in anderen Behandlungsmethoden zu erledigen. Genau dies passiert

in `Form1_MouseMove` auch: Sofern die linke Maustaste gedrückt ist, organisiert sich die Methode ein `Graphics`-Objekt für den Clientbereich des Formulars, überzeichnet zuerst das Gummiband des vorigen Aufrufs mit der Hintergrundfarbe, passt das Gummiband an die neue Mausposition an und zeichnet es schließlich in Vordergrundfarbe.

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
  if ( (e.Button & MouseButtons.Left) > 0)
  {
    Graphics g = this.CreateGraphics();   // Zeichenfläche
    g.DrawRectangle(new Pen(this.BackColor), RubberBand);  // Löschen,
    RubberBand.Width = e.X - RubberBand.X;
    RubberBand.Height= e.Y - RubberBand.Y;
    g.DrawRectangle(new Pen(this.ForeColor), RubberBand);  // neu zeichnen
    Text = RubberBand.ToString();
  }
}
```

Nun fehlt noch die Methode `Form1_MouseUp()`, deren Aufgabe darin besteht, beim Loslassen der linken Maustaste das Gummiband ein letztes Mal zu überzeichnen:

```
private void Form1_MouseUp(object sender, MouseEventArgs e)
{
  if ( (e.Button & MouseButtons.Left) > 0)
  {
    Graphics g = this.CreateGraphics();   // Zeichenfläche
    g.DrawRectangle(new Pen(this.BackColor), RubberBand);
  }
}
```

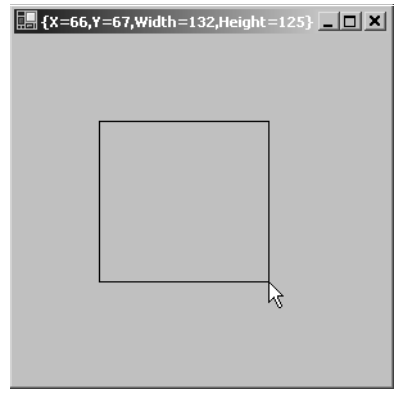


Abbildung 12.2: Die Anwendung *Gummiband* in Aktion

Verbesserungen

Das vorgestellte Gummiband hat verschiedene Nachteile. Erstens überzeichnet es stumpf den Clientbereich des Formulars (löscht dort also Inhalte) und zweitens arbeitet es mit festen Farben, so dass es Kontrastschwierigkeiten

geben kann. Tatsächlich ist `DrawRectangle()` nicht gerade die beste Lösung zum Zeichnen eines Gummibands. Als »anständige« Lösung bietet sich die Methode `ControlPaint.DrawReversibleFrame()` an, die erstens gepunktete oder gestrichelte Linien zeichnet und zweitens dabei das Xor-Verfahren anwendet – der zweite Aufruf mit denselben Parametern macht das Ergebnis des ersten Aufrufs unsichtbar.

Das einzig Gewöhnungsbedürftige an der Methode ist, dass sie Bildschirmkoordinaten und keine Clientkoordinaten erwartet. Dazu können Sie entweder die vom Ereignisobjekt gelieferten Koordinaten über die formulareigene Methode `PointToScreen()` umrechnen oder Sie fragen mit `Control.MousePosition` die Mausposition ab. Das Projekt *Gummiband1* integriert diese Verbesserungen. Hier der relevante Code:

```
private void Form1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{
  if ( (e.Button & MouseButtons.Left) > 0)  // linke Maustaste?
  {
    // RubberBand.Location = PointToScreen(new Point(e.X, e.Y));
    RubberBand.Location = Control.MousePosition;
  }
}

private void Form1_MouseMove(object sender, MouseEventArgs e)
{
  if ( (e.Button & MouseButtons.Left) > 0)  // linke Maustaste?
  {
    if(this.ClientRectangle.Contains(new Point(e.X, e.Y)))
    {
      ControlPaint.DrawReversibleFrame(RubberBand, this.BackColor,
        FrameStyle.Dashed);
      RubberBand.Size =
         (Size) Control.MousePosition - (Size) RubberBand.Location;
      ControlPaint.DrawReversibleFrame(RubberBand, this.BackColor,
        FrameStyle.Dashed);
      Text = RubberBand.ToString();
    }
  }
}

private void Form1_MouseUp(object sender, MouseEventArgs e)
{
  if ( (e.Button & MouseButtons.Left) > 0)  // linke Maustaste?
   {
    ControlPaint.DrawReversibleFrame(RubberBand, this.BackColor,
      FrameStyle.Dashed);
    RubberBand.Size = new Size(0, 0);
  }
}
```

Wahrscheinlich ist es Ihnen aufgefallen: Es ist eine Abfrage hinzugekommen, die prüft, ob die Maus noch im Clientbereich des Formulars ist. Lässt

man diese Abfrage weg, zeichnet das Programm munter über die Grenzen des eigenen Fensters hinweg. Probieren Sie es aus.

Ein anspruchsvolleres Beispiel, das die Gummibandfunktionalität zum Zoomen von Bildausschnitten einsetzt, finden Sie im Abschnitt »Codebeispiel – eine Reise durch die Mandelbrotmenge« ab Seite 516.

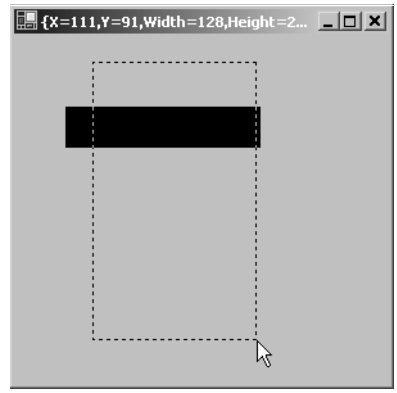


Abbildung 12.3: Die verbesserte Version des Gummibands zeichnet gut sichtbar über bestehende Inhalte hinweg, ohne sie zu verändern.

Übung

Wenn Sie mit der Anwendung ein paar Fingerübungen machen wollen, hier drei Vorschläge:

1. Ändern Sie das Programm dahingehend, dass es nur bei gedrückter rechter *und* linker Maustaste zeichnet
2. Versuchen Sie die gesamte Logik in `Form1_MouseMove()` unterzubringen.
3. Wie kommt das Programm mit einem 50-prozentigen Grauraster als Hintergrund zurecht?

Sekundäre Mausereignisse

Die meisten sekundären Mausereignisse sind von den primären abgeleitet und ergeben sich durch Berücksichtigung zusätzlicher Kontextinformationen, wie zeitliche Abfolge und/oder Zustandskombinationen. So fasst das `Click`-Ereignis beispielsweise `MouseDown` und `MouseUp` zusammen und berichtet von einem schlichten Mausklick, ohne weitere Information über den Mauszustand bekannt zu geben. `MouseEnter` tritt hingegen auf, wenn der Benutzer die Maus in den Fensterbereich des jeweiligen `Control`-Objekts bewegt, und `MouseHover`, wenn der Benutzer mit der Maus ca. eine Sekunde bewegungslos verweilt. Tabelle 12.6 gibt einen Überblick über sekundäre Mausereignisse, die Formulare und (in unterschiedlicher Auswahl) Steuerelemente melden.

Die folgenden aus dem VS.NET-Fenster AUSGABE stammenden Zeilen dokumentieren einen Programmlauf des Beispielprojekts *MausEreignisse*. Das Programm behandelt verschiedene Mausereignisse des Formulars und gibt die relevanten Informationen via `Control.WriteLine()` im Ausgabefenster des Debuggers aus:

```
 1 MouseEnter
 2 MouseMove bei 291,268 Vektor für Tasten/Rad: None
 3 MouseMove bei 287,266 Vektor für Tasten/Rad: None
 4 MouseMove bei 287,265 Vektor für Tasten/Rad: None
 5 MouseHover
 6 MouseDown bei 287,265 Vektor für Tasten/Rad: Left
 7 MouseMove bei 286,265 Vektor für Tasten/Rad: Left
 8 Click
 9 MouseUp bei 286,265 Vektor für Tasten/Rad: Left
10 MouseMove bei 286,265 Vektor für Tasten/Rad: None
   ...
22 MouseDown bei 280,240 Vektor für Tasten/Rad: Left
23 Click
24 MouseUp bei 280,240 Vektor für Tasten/Rad: Left
25 MouseMove bei 280,240 Vektor für Tasten/Rad: None
26 MouseDown bei 280,240 Vektor für Tasten/Rad: Left
27 DoubleClick
28 MouseUp bei 280,240 Vektor für Tasten/Rad: Left
29 MouseMove bei 280,240 Vektor für Tasten/Rad: None
30 MouseMove bei 286,240 Vektor für Tasten/Rad: None
     31 MouseLeave
```

Die Ausgabe offenbart die Abfolge der Mausereignisse und auch das Verhältnis zwischen primären und sekundären Mausereignissen. `Click` trifft nach `MouseDown`, aber noch vor `MouseUp` ein. Und für einen Doppelklick gilt gar die Folge `MouseDown`, `Click`, `MouseUp`, `MouseDown`, `DoubleClick`, `MouseUp`, wobei in beiden Fällen noch `MouseMove`-Ereignisse dazwischenliegen können.

Tabelle 12.6: Wichtige in der Klasse `Control` definierte sekundäre Mausereignisse (die ihrerseits sämtlich Interpretationen primärer Mausereignisse darstellen)

Mausereignis	Tritt auf, wenn	Typ des Ereignisobjekts
`MouseHover`	... die Maus ca. 1 Sek. über dem Steuerelement verweilt (Mouse-held-over). Meist für Anzeige von Quick-Infos verwertet	`EventArgs`
`Click`	... ein Mausklick stattfindet	`EventArgs`
`DoubleClick`	... ein Doppelklick stattfindet	`EventArgs`
`MouseEnter`	... der Mauszeiger den Clientbereich des Steuerelements/Komponente betritt	`EventArgs`
`MouseLeave`	... der Mauszeiger den Clientbereich des Steuerelements/Komponente verlässt	`EventArgs`

Mausereignis	Tritt auf, wenn	Typ des Ereignisobjekts
`DragEnter`	... der Mauszeiger nach Beginn einer Drag&Drop-Operation den Clientbereich des Steuerelements/ Komponente betritt	`DragEventArgs`
`DragOver`	...die Maus während einer Drag&Drop-Operation über den Clientbereich des Steuerelements/ Komponente bewegt wird	`DragEventArgs`
`DragLeave`	... der Mauszeiger nach Beginn einer Drag&Drop-Operation den Clientbereich des Steuerelements/ Komponente verlässt	`EventArgs`
`DragDrop`	... eine Drag&Drop-Operation im Clientbereich des Steuerelements/ Komponente endet	`DragEventArgs`

Tabelle 12.6: Wichtige in der Klasse `Control` definierte sekundäre Mausereignisse (die ihrerseits sämtlich Interpretationen primärer Mausereignisse darstellen) (Forts.)

Übung

Implementieren Sie eine Anwendung, deren Ausgabe dem vorgestellten Muster entspricht, aber auch die Drag&Drop-Ereignisse protokolliert. Setzen Sie dazu die `AllowDrop`-Eigenschaft des Formulars auf `true` und fügen Sie in die Behandlungsmethode für das `DragEnter`-Ereignis die folgende Zeile ein.

```
e.Effect = DragDropEffects.All;
```

Diese Anweisung schafft die Voraussetzung für das Eintreffen des `DragDrop`-Ereignisses (mehr über Drag&Drop im Kapitel 18 »Zwischenablage und Drag&Drop«).

Probieren Sie die Anwendung aus, indem Sie beispielsweise Dateien aus dem Explorer in ihr Fenster ziehen.

12.1.4 Timer

Alles, was mit Zeitverzögerungen und periodischen Abläufen zu tun hat, fällt in den Aufgabenbereich des Timers. Um in Ihren Windows-Programmen eine Zeitsteuerung zu implementieren, stehen Ihnen unter .NET vom Prinzip her zwei Timer-Komponenten zur Verfügung:

1. Der traditionelle Windows-Timer in Form der Klasse `System.Windows.Forms.Timer`. Sie finden die Komponente auf der Seite WINDOWS FORMS der TOOLBOX – unter dem Namen TIMER. Die Zeitbasis dieses Timers ist der System-Tick, der unter Windows 2000 und Windows XP

etwa 100 mal je Sekunde und damit ca. alle 10 ms stattfindet (das unter Windows 9x noch geltende Mindestintervall von 55 ms ist also geschrumpft). Da die Komponente das Zeitintervall über den Datentyp `int` in Millisekunden repräsentiert, sind mit diesem Timer maximale Zeitintervalle bis knapp 25 Tagen möglich.

2. Eine speziell für die Server-Anwendung ausgerichtete Komponente mit erweitertem Funktionsumfang in Form der Klasse `System.Timers.Timer`. Sie finden die Komponente auf der Seite KOMPONENTEN der TOOLBOX –unter dem Namen TIMER. Das minimale Zeitintervall dieses Timers beträgt gleichfalls etwa 10 ms (sind Ihre Hoffungen nun dahin?), da das Intervall als `double` repräsentiert wird, sind aber beliebig lange Intervalle möglich. Außerdem ist die Ausstattung der Klasse etwas reichhaltiger, da sie spezifisch für Multithreading konzipiert ist.

Im Allgemeinen werden Sie mit dem gewöhnlichen Windows-Timer auskommen. Wenn Sie eine Instanz dieser an sich ja fensterlosen Komponente im Designer auf Ihren Formularentwurf ziehen, erscheint diese in einem abgetrennten Fensterbereich unterhalb des Formularentwurfs (vgl. Abbildung 12.4). Tabelle 12.7 gibt einen Überblick über die wichtigsten Eigenschaften und Methoden der Timer-Komponente. Die Bezeichner sind so gewählt, dass der Umgang mit der Komponente völlig intuitiv sein sollte.

Tabelle 12.7: Wichtige Methoden und Eigenschaften der `Timer`-Komponente

`Timer`-Eigenschaft/Methode	Bedeutung
`bool Enabled {get; set;}`	`true`, wenn der Timer aktiviert ist
`int Interval {get; set;}`	Zeitintervall größer 0 in Millisekunden
`void Start()`	Startet den Timer (setzt `Enabled` auf `true`)
`void Stop()`	Stoppt den Timer (setzt `Enabled` auf `false`)
`bool AutoReset {get; set;}`	`true`, wenn das `Elapsed`-Ereignis periodisch generiert werden soll. Diese Eigenschaft ist nur bei `System.Timers.Timer` zu finden. Die Komponente `System.Windows.Forms.Timer` generiert ihr `Tick`-Ereignis grundsätzlich periodisch.

Beide Timer-Komponenten generieren *ein* Ereignis: Die Windows-Timer-Komponente generiert das `Tick`-Ereignis, die andere das `Elapsed`-Ereignis – so dass spätestens hier Verwechslungen ausgeschlossen sein dürften. Während das Tick-Ereignis ein nacktes `EventArgs`-Objekt transportiert, stellt `Elapsed` ein `ElapsedEventArgs`-Objekt bereit, dessen zusätzliches Datenfeld `SignalTime` den genauen Zeitpunkt des Ereignisses in Form eines `DateTime`-Objekts beschreibt.

Codebeispiel – Zeitanzeige

Sie finden in diesem Buch verschiedene Beispiele für den Gebrauch des Windows-Timers. Das Beispielprojekt *Zeitanzeige* erschöpft sich in einer einfachen Anzeige der Systemzeit mit Stunden, Minuten und Sekunden über ein `Label`-Steuerelement. Da sowohl die Timer-Komponente `timer1` als auch das Beschriftungsfeld `label1` über den Designer initialisiert wurden, beschränkt sich der manuell hinzugefügte Code auf den Rumpf der `Tick`-Behandlungsmethode:

```
private void timer1_Tick(object sender, System.EventArgs e)
{
  this.label1.Text = DateTime.Now.ToLongTimeString();
}
```

Abbildung 12.4 zeigt den Formularentwurf, und Abbildung 12.5 die Ausgabe des Programms.

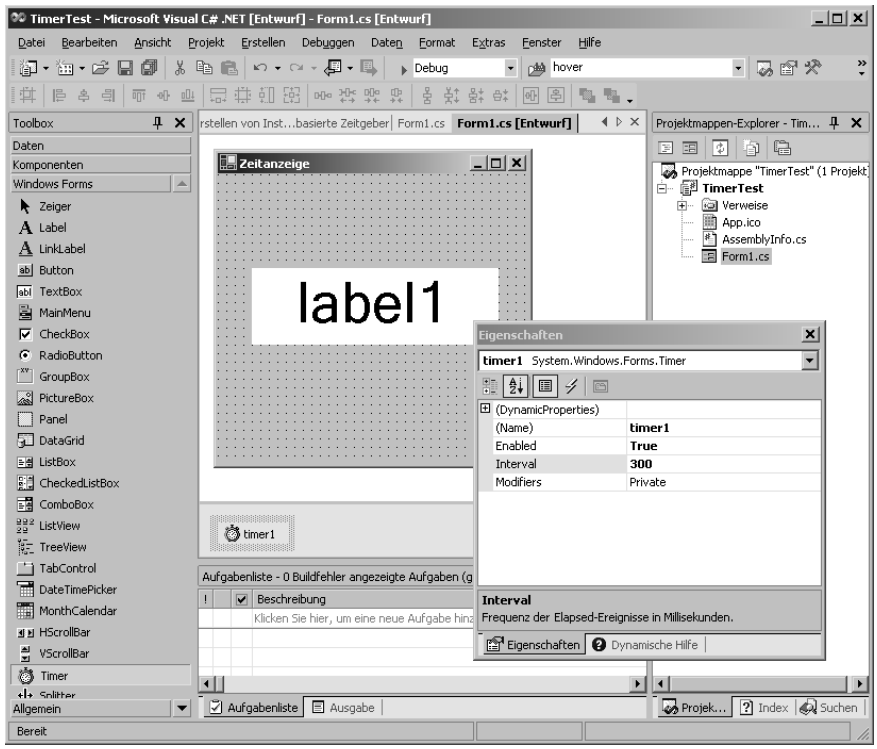


Abbildung 12.4: Timer-Komponente in der Entwurfsansicht des Projekts *Zeitanzeige*

Übung

Implementieren Sie die Anwendung mit der anderen Timer-Komponente und ohne Verwendung der Methode `Now()`.



Abbildung 12.5: Die Zeitanzeige in Aktion

12.1.5 Tastatur

Das zweite zur Standardausstattung eines jeden PCs gehörige Eingabegerät ist die Tastatur, die neben der Texteingabe traditionell auch eine bunte Vielfalt an Steuerfunktionen übernimmt. Es gehört zum guten Ton einer Windows-Anwendung, in den wichtigsten Funktionen (beispielsweise Schließen der Anwendung mit Speichern des Dokuments), wenn nicht gar vollständig per Tastatur bedienbar zu sein. Tatsächlich ist Windows selbst vollständig mauslos bedienbar, was nicht nur für Tastaturfreaks wichtig ist, sondern tatsächlich auch unangenehme Situationen überbrückt, wenn die Maus einmal nicht funktioniert. Eine wirkliche Zeitersparnis sind auch die frei definierbaren Hotkey-Vereinbarungen für den Aufruf von Anwendungen über den Desktop. Dieses praktische Feature macht deutlich, dass nicht jede Tastaturaktion des Benutzers ihren Weg bis zur laufenden Anwendung nehmen muss, sondern unterwegs auch vom System »weggeschnappt« werden kann. Ein Fall, für den dies im Dienste der Anwendung geschieht, ist die im System verankerte standardmäßige Tastatursteuerung per Alt-Taste für Menüs und Formulare mit Steuerelementen.

Eingabefokus

Tastaturereignisse, die das System nicht selbst verwertet, übermittelt es an das jeweils aktive Formular der im Vordergrund laufenden Anwendung. Dieses Formular besitzt den *Eingabefokus*. Sofern das Formular `Control`-Objekte (Steuerelemente oder Komponenten) enthält, die Tastatureingaben verarbeiten können, liegt der Eingabefokus formularintern immer auf einem dieser Objekte, das diesen Zustand im Allgemeinen durch ein Fokusrechteck visualisiert. Tabelle 12.8 gibt einen Überblick über die für den Umgang mit dem Eingabefokus relevanten Eigenschaften und Methoden.

Eigenschaft/Methode	Beschreibung
`static Form Form.ActiveForm {get;}`	Aktives Formular (der Wert ist `null`, wenn der Aufruf in einer momentan im Hintergrund laufenden Anwendung erfolgt, d.h. ein Fenster einer anderen Anwendung im Vordergrund steht)
`void Activate()`	Aktiviert das Formular (von `Form` vererbt)
`bool Focus()`	Setzt den Eingabefokus auf das Steuerelement (von `Control` vererbt)
`bool CanFocus {get;}`	`true`, wenn das Steuerelement den Eingabefokus annehmen kann (von `Control` vererbt)
`bool Focused {get;}`	`true`, wenn das Steuerelement den Eingabefokus besitzt (von `Control` vererbt)
`bool ContainsFocus{get;}`	`true`, wenn das Steuerelement oder ein diesem untergeordnetes Steuerelement den Eingabefokus besitzt (von `Control` vererbt)

Tabelle 12.8: Eigenschaften und Methoden, die mit dem Eingabefokus zu tun haben

Verteilung der Tastaturereignisse auf Formularebene

Nachdem das jedem Formular zugrunde liegende `Form`-Objekt eine eigene Nachrichtenschleife betreibt, übernimmt es auch die weitere Zustellung der Ereignisse an das Steuerelement, das zum jeweiligen Zeitpunkt den formularinternen Eingabefokus besitzt. In der Praxis sieht dies so aus, dass die Nachrichtenschleife die `OnKeyXxx()`-Methoden des Steuerelementobjekts aufruft und damit den für diese Ereignisse installierten Behandlungsmethoden Gelegenheit gibt, das ihre zu tun.



Der eigentliche Einsprungpunkt eines `Control`*-Objekts für die Verarbeitung von Tastaturereignissen ist die virtuelle Methode* `ProcessCmdKey()`*. Sie übernimmt die Reaktion auf eine Reihe von Befehlstasten, beispielsweise die Richtungstasten, denen standardmäßig eine Steuerfunktion zugeordnet ist, und ruft für alles Andere die Methode* `ProcessKeyEventArgs()` *auf.* `ProcessKeyEventArgs()` *veranlasst den Aufruf der* `OnKeyXxx()`*-Methoden, die ihrerseits schließlich* `KeyXxx`*-Ereignisse generieren.*

Um eine Interpretation der Befehlstasten zu verhindern, können Sie die virtuelle Methode `ProcessCmdKey()` *in einer von* `Control` *abgeleiteten Klasse überschreiben und* `ProcessKeyEventArgs()` *auch ohne weitere Filterung aufrufen. Ein Codebeispiel dazu finden Sie im Abschnitt »Die Tastaturschnittstelle« ab Seite 654.*

Tastaturereignisse

Wie Tabelle 12.9 zeigt, gibt es drei unterschiedliche Tastaturereignisse:

- `KeyPress` arbeitet zeichenorientiert und wird nur für Tasten und Tastenkombinationen generiert, denen ein Zeichen zugeordnet ist. Hierzu gehören auch Kombinationen mit den Modifikationstasten Strg und ⇧ sowie Sonderzeichen, die nur über andere Kombinationen beispielsweise über AltGr zur Verfügung stehen. Alt-Kombinationen liefern dagegen im Allgemeinen keine Zeichen.
- `KeyDown` und `KeyUp` treten auf, wann immer der Benutzer eine Taste drückt oder loslässt.

Die Reihenfolge für eine mit einem Zeichen belegte Taste lautet: `KeyDown`, `KeyPress`, `KeyUp`. Tastenkombinationen mit Modifikationstasten – sowie die Modifikationstasten selbst – genieren für jede beteiligte Taste `KeyDown`- und `KeyUp`-Ereignisse.

Tabelle 12.9: Die drei Tastaturereignisse

Tastaturereignis	Tritt auf, wenn	Typ des Ereignisobjekts
`KeyDown`	... Taste niedergedrückt wird	`KeyEventArgs`
`KeyUp`	... Taste gelöst wird	`KeyEventArgs`
`KeyPress`	... Tastendruck erfolgt	`KeyPressEventArgs`

Die mit dem `KeyPress`-Ereignis assoziierte Information wird einer dafür registrierten Behandlungsmethode als `KeyPressEventArgs`-Parameter übergeben und umfasst im Wesentlichen den Code des eingegebenen Zeichens (Tabelle 12.10).

Tabelle 12.10: Wichtige Eigenschaften der Klasse `KeyPressEventArgs`

Eigenschaft	Bedeutung
`bool Handled {get; set;}`	`true`, wenn das Ereignis abschließend behandelt wurde. Ereignisbehandlungsmethoden sollten diese Eigenschaft auswerten, um in Erfahrung zu bringen, ob nicht bereits eine früher aufgerufene Ereignisbehandlungsmethode das Ereignis abschließend behandelt hat. Umgekehrt sollten sie die Eigenschaft in diesem Sinne auch pflegen.
`char KeyChar {get;}`	Eingegebenes Zeichen

Die Klasse `KeyEventArgs` für die anderen beiden Ereignisse ist um einiges umfangreicher und sieht im Wesentlichen eine detaillierte Beschreibung des gesamten Tastaturzustands in Form von `Keys`-Bitvektoren vor (Tabelle 12.11).

Tabelle 12.11: Wichtige Eigenschaften der Klasse `KeyEventArgs`

Eigenschaft	Bedeutung
`bool Alt {get; }`	`true`, wenn Alt gedrückt
`bool Control {get;}`	`true`, wenn Strg gedrückt
`bool Handled {get; set;}`	`true`, wenn das Ereignis abschließend behandelt wurde. Ereignisbehandlungsmethoden sollten diese Eigenschaft auswerten, um in Erfahrung zu bringen, ob nicht bereits eine früher aufgerufene Ereignisbehandlungsmethode das Ereignis abschließend behandelt hat. Umgekehrt sollten sie die Eigenschaft in diesem Sinne auch pflegen.
`Keys KeyCode {get;}`	Reiner Tastencode, ohne Modifizier
`Keys KeyData {get;}`	Vollständige Tastenbeschreibung mit Modifizierern
`int KeyValue {get;}`	`KeyData` als `int`-Wert
`Keys Modifiers {get;}`	Modifizierer ohne Tastencode
`bool Shift {get;}`	`true`, wenn ⇧ gedrückt

Als Aufzählklasse definiert `Keys` eine Unmenge von Konstanten, die eine vollständige Analyse der Eigenschaften `KeyCode`, `KeyData` und `Modifiers` ermöglicht. Einen Überblick finden Sie in der Online-Dokumentation.

Bei `KeyCode` und `KeyData` zeigt sich wieder einmal, dass man bei Microsoft .NET sehr wohl als plattformübergreifenden Ansatz sieht. Hier geht es um abstrakte Bezeichner, die mit den von anderen Programmiersprachen her gewohnten »Scancodes« der Tastatur nichts zu tun haben.



Codebeispiel – Tastaturanalyse

Den Effekt der folgenden `KeyDown`-Behandlung werden Sie vielleicht kennen, wenn Sie schon einmal eine Tastenkombination für einen Menübefehl oder ein Desktop-Symbol vereinbart haben. Das Beispielprojekt *Tastaturanalyse* demonstriert die Behandlung von zwei der drei Ereignisse für ein Textfeld und nutzt dabei ein Feature, das C# »so ganz nebenbei« standardmäßig für Objekte von Aufzählklassen unterstützt, die mit dem `[Flags]`-Attribut vereinbart sind: Die `ToString()`-Methode drückt den Wert eines Bitvektors als Kombination der Bezeichner der in der Aufzählung für die einzelnen Bitfelder vereinbarten Konstantenwerte aus.

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
  const string nl = "\r\n";
  textBox1.Text = nl + "KeyCode: " + e.KeyCode;      // implizit ToString()
  textBox1.Text += nl + "KeyData: " + e.KeyData;     // ...
```

```
  textBox1.Text += nl + "Modifiers: " + e.Modifiers;
  textBox1.Text += nl + "KeyValue: " + e.KeyValue;
}

private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
    e.Handled = true;          // Ausgabe des Zeichens unterdrücken
}
```

Die Methode `textBox1_KeyDown` ist für das `KeyDown`-Ereignis des Textfelds `textBox1` registriert, dessen `Multiline`-Eigenschaft bereits beim Entwurf auf `true` gesetzt wurde. Die Behandlungsmethode für `KeyPress` beschränkt sich darauf, die Ausgabe des Zeichens im Textfeld zu unterdrücken.

Abbildung 12.6 zeigt die Ausgabe des Programms für die Tastenkombinationen ⇧+Strg+Einf, ⇧+A, ⇧+Strg+P und Alt+F12.

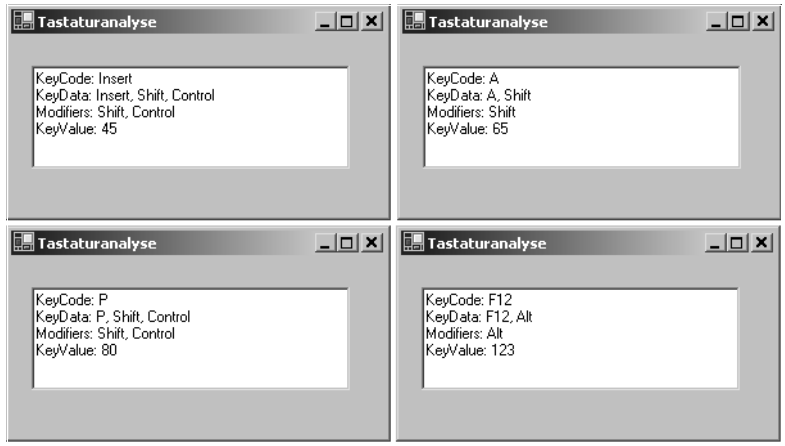


Abbildung 12.6: Ausgabe des Programms Tastaturanalyse für vier unterschiedliche Tastenkombinationen

Probieren Sie das Programm aus, um interaktiv zu erforschen, welchen Konstantennamen die `Keys`-Aufzählung für welche Taste definiert. Tastenkombinationen, die das System abfängt, etwa PrintScreen, bekommen Sie auf diese Weise allerdings nicht zu Gesicht. Außerdem unterscheidet die Darstellung nicht zwischen der rechten und der linken Alt-Taste sowie zwischen Alt+Strg und AltGr.

Die praxisgerechte Implementierung einer Tastaturschnittstelle für eine konkrete Anwendung finden Sie im Codebeispiel *Taschenrechner* (Seite 557).

Vorschau des Formulars

Da das Formularobjekt die Verteilung von Nachrichten übernimmt, hat es die Möglichkeit eine Vorschau zu betreiben und kann dabei wahlweise die weitere Zustellung von Tastaturereignissen an das Steuerelement mit dem Eingabefokus unterdrücken. Um den Vorschaumechanismus zu aktivieren,

müssen Sie die (standardmäßig mit `false` initialisierte) `KeyPreview`-Eigenschaft auf `true` setzen und Behandlungsmethoden für die `KeyXxx`-Ereignisse registrieren, die *vor* der Weitergabe an das jeweilige Steuerelement zum Zuge kommen.

Im Einzelnen sieht der Mechanismus dafür so aus:

1. Ist die `KeyPreview`-Eigenschaft bei Eintreffen eines Tastaturereignisses auf `true` gesetzt, reflektiert das `Form`-Objekt die Ereignisse auf sich selbst. Dazu ruft es seine eigenen `OnKeyXxx()`-Methoden auf. Eine von `Form` abgeleitete Formularklasse kann diese Methoden durch eigene Varianten überschreiben (und darf darin den Aufruf der `base`-Variante nicht vergessen) und/oder wie gewohnt Behandlungsmethoden dafür registrieren, um darauf zu reagieren.

   ```
   public class MyForm : Form
   {
      protected override void OnKeyPress(KeyPressEventArgs e)
      {
        // TODO Erste Verwertung von e auf Formularebene + Signalisierung
        base.OnKeyPress(e);   // Aufruf der Basisklassenvariante
      }
      private void MyForm_KeyPress(object sender, KeyPressEventArgs e)
      {
        // TODO Zweite Verwertung von e auf Formularebene
      }
   ...
   }
   ```

   Falls das Tastaturereignis damit bereits erledigt ist, braucht das Formularobjekt nur die `Handled`-Eigenschaft auf `true` zu setzen. Das unterbindet die weitere Verarbeitung des Ereignisses.

2. Ist die `Handled`-Eigenschaft des Ereignisobjekts nach wie vor `false`, ruft das `Form`-Objekt die `OnKeyXxx()`-Methode des Steuerelements auf, das momentan den Eingabefokus besitzt (exakter: die `OnKeyXxx()`-Methode des zugrunde liegenden `Control`-Objekts. Die virtuelle Methode `OnKeyXxx()` kann durch eine weitere abgeleitete Klasse überschrieben werden und sollte dann nicht vergessen, die `base`-Variante aufzurufen).

3. Die `OnKeyXxx()`-Methode des Steuerelements (bzw. des `Control`-Objekts) leitet die reguläre Behandlung des Ereignisses ein (die ihrerseits gewöhnlich in der Formularklasse definiert ist), indem sie die beim Objekt registrierten Behandlungsmethoden aufruft.

   ```
   public class MyForm : Form
   {
     ...
     private void MyControl_KeyPress(object sender, KeyPressEventArgs e)
     {
   ```

```
        // TODO Dritte Verwertung von e auf Formularebene für Steuerelement
    }
}
```

Abbildung 12.7 zeigt den beschriebenen Ablauf für ein Formularobjekt und ein Steuerelement. Die Methoden `OnKeyXxx()` und `MyControl_KeyXxx()` können implementiert sein, müssen aber nicht.

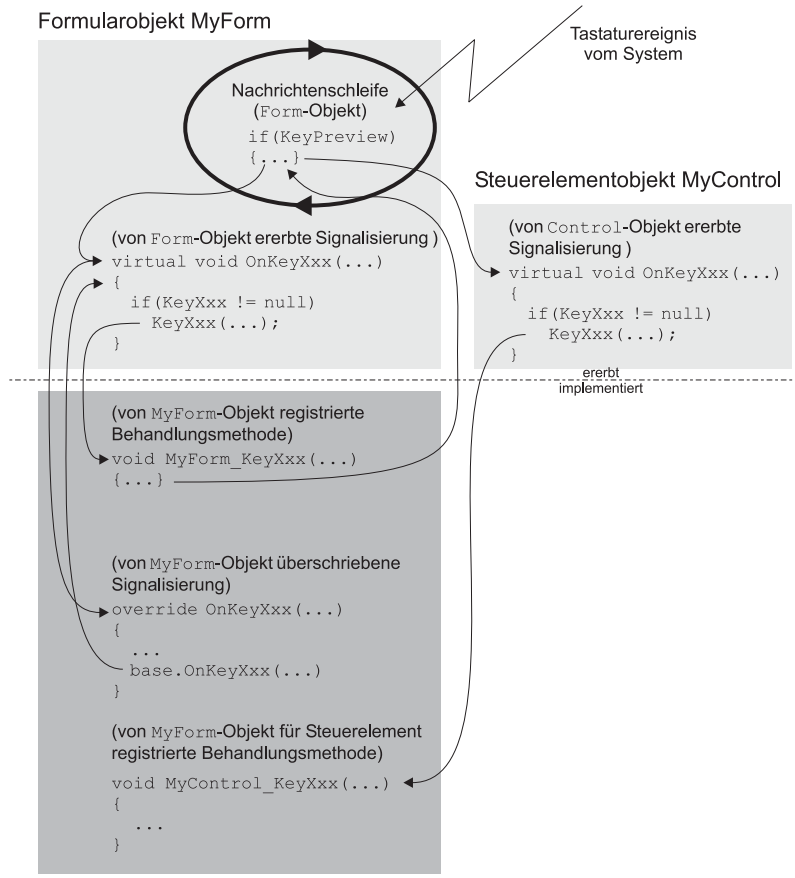


Abbildung 12.7: Verarbeitung eines Tastaturereignisses bei Vorschau durch das Formular

Codebeispiel – Tastaturvorschau

Der folgende Codeauszug entstammt der Beispielanwendung *Tastaturvorschau*, bei der das Formular nur bestimmte `KeyPress`-Ereignisse an seine Steuerelemente freigibt. Herausgefilterte Zeichen erscheinen in der Titelzeile des Formulars.

```
private void InitializeComponent()
{
  ...
  // textBox1
  this.textBox1.KeyPress += new KeyPressEventHandler(
                              this.textBox1_KeyPress);
  // MyForm
  this.KeyPreview = true;
  this.KeyPress += new KeyPressEventHandler(this.MyForm_KeyPress);
  ...
}

private void MyForm_KeyPress(object sender, KeyPressEventArgs e)
{
  switch (e.KeyChar)
  {
    case 'X':
      Close();
      break;
    case '0':
    case '2':
    case '4':
    case '6':
    case '8':
      break;
    default:
      Text += e.KeyChar;
      e.Handled = true;
      break;
  }
}
```



Abbildung 12.8: Die Beispielanwendung *Tastaturvorschau* lässt nur die Zeichen für die geraden Ziffern »0«, »2«, »4«, »6«, »8« passieren.

13 Formulare

Windows-Programmierung unter .NET läuft größtenteils – nach dem Vorbild von Visual Basic – auf die Ausgestaltung von Formularen und Steuerelementen hinaus. Ein guter Teil der eigentlichen Programmierarbeit geschieht unter der Regie des Designers, der es gestattet, die Gestaltung von Formularen, Komponenten und Steuerelementen mit den Mitteln der grafischen Benutzeroberfläche auf visuellem Wege abzuwickeln.

Als Programmierer finden Sie einen übersichtlichen Baukasten mit einer »Grundplatte« – dem Formular- oder Steuerelemententwurf – vor, dessen Elemente – die Steuerelemente und Komponenten – Sie nicht nur per Drag&Drop einfügen und arrangieren, sondern auch mit wenigen Mausklicks wunschgemäß initialisieren und mit einer Codeanbindung versehen können, die sich nahtlos in das bestehende Codegerüst einfügt.

Das vom Designer penibel gepflegte Codegerüst stellt nicht nur eine Ausgangsbasis für die weitere Programmierung dar, es »transportiert« auch eine standardisierte Form des Anwendungsdesigns, an die Sie sich halten sollten.

Das vorliegende Kapitel führt in die Programmierung mit Formularen ein. Seine Abschnitte beleuchten die Abstammung und Erbmasse, die Ereignisprogrammierung, die spezifischen Formularsteuerelemente `MainMenu`, `ToolBar` und `StatusBar` sowie das Heer der anderen Steuerelemente aus der TOOLBOX. Der letzte Abschnitt geht schließlich noch kurz auf die MDI-Programmierung mit mehreren Formularen ein.

13.1 Abstammung und Erbmasse

Ähnlich wie das Fenster einer Konsolenanwendung dient das Hauptformular dem Programm als primäres Ein- und Ausgabemedium für die Interaktion mit dem Benutzer. Als Ausgabemedium zeichnet es in seinen CLIENTBEREICH Text, Grafik und Bitmaps und stellt einen Rahmen für die (eigenständige) Darstellung von Steuerelementen in ihren verschiedenen Zuständen dar. Als Eingabemedium reagiert es auf die Tastatur und die Maus, aber auch auf andere Eingabegeräte, sofern diese angeschlossen und installiert sind. Was den Clientbereich eines Formulars betrifft, so kann letztlich jeder einzelne Bildpunkt Ausgabe- und Eingabeelement sein. Die

Funktionalität der Randelemente (Titelleiste und Rahmen) ist hingegen weitgehend vorgegeben und unterliegt im Allgemeinen nicht der freien Gestaltung.

All das ist jedem bestens bekannt, und dank der konsistenten Oberfläche bekommen selbst Windows-Einsteiger durch kommerzielle Windows-Anwendungen sehr schnell konkrete Vorstellungen von den typischen Formularsteuerelementen Menü, Symbol- und Statusleiste sowie den anderen Steuerelementen aus der TOOLBOX – angefangen vom einfachen Kontrollkästchen bis hin zum komplexen `TreeView`. Für ProgrammiererInnen bleibt die Frage, wie sie sich diese Standardfunktionalität zu Diensten machen und wo sie die notwendigen Strukturen und Bestandteile dafür finden.

Die Antwort auf die Wie-Frage lautet »mittels Vererbung«, die Antwort auf die Wo-Frage »in der .Net-Klassenbibliothek«. Abbildung 13.1 und Abbildung 15.1 (Seite 540) zeigen einen Ausschnitt aus der .NET-Klassenhierarchie, der die Verwandtschaftsbeziehungen zwischen den für Formulare und Steuerelemente relevanten Klassen deutlich macht.

Wann immer Sie neues Codegerüst für eine Windows-Anwendung anlegen, leitet der Assistent für Sie auch eine neue Formularklasse von der Klasse `Form` ab und staffiert diese gleich noch etwas aus – mit einem Konstruktor, einer darin aufgerufenen Methode `InitializeComponent()`, die er im weiteren Entwicklungsprozess selbst pflegt, einer `Dispose()`-Methode, einem Instanzfeld `components` und natürlich mit der statischen Methode `Main()`. Letztere ist der Startpunkt der Anwendung und legt ein Formularobjekt an, welches das Hauptformular der Anwendung verkörpert. All das steckt in dem Einzeiler:

```
Application.Run(new Form1());
```

In der Tat endet diese Anweisung erst, wenn der Benutzer das Hauptformular schließt, was zugleich das Programmende einläutet.



Die Klasse `Form` *ist nicht abstrakt. Sie können jederzeit Instanzen davon generieren und als Grundlage für eigene Dialoge verwenden – die Sie dann aber per Code aufbauen müssen, weil der Designer darauf nicht eingerichtet ist. Wie das funktioniert, gucken Sie sich am besten beim Designer ab. Ein Blick auf das Codegerüst verrät, dass das* `Form`*-Objekt zwar eine* `Controls`*-Auflistung für Steuerelemente besitzt, ihm aber eine* `components`*-Auflistung fehlt (dieses Datenfeld wird ja erst in der Ableitung eingeführt), weshalb es nur mit Steuerelementen bestückt werden kann, die von* `Control` *abstammen. Wer Dialoge sozusagen nach Bedarf anlegen will, muss also einen gewissen Ehrgeiz zum Umgehen des Designers in die Waagschale werfen. Als Trost bleibt, dass man jederzeit auf das reguläre Formulardesign zurückgreifen kann, wenn sich ein »zu Fuß« angelegter Dialog als zu komplex erweisen sollte.*

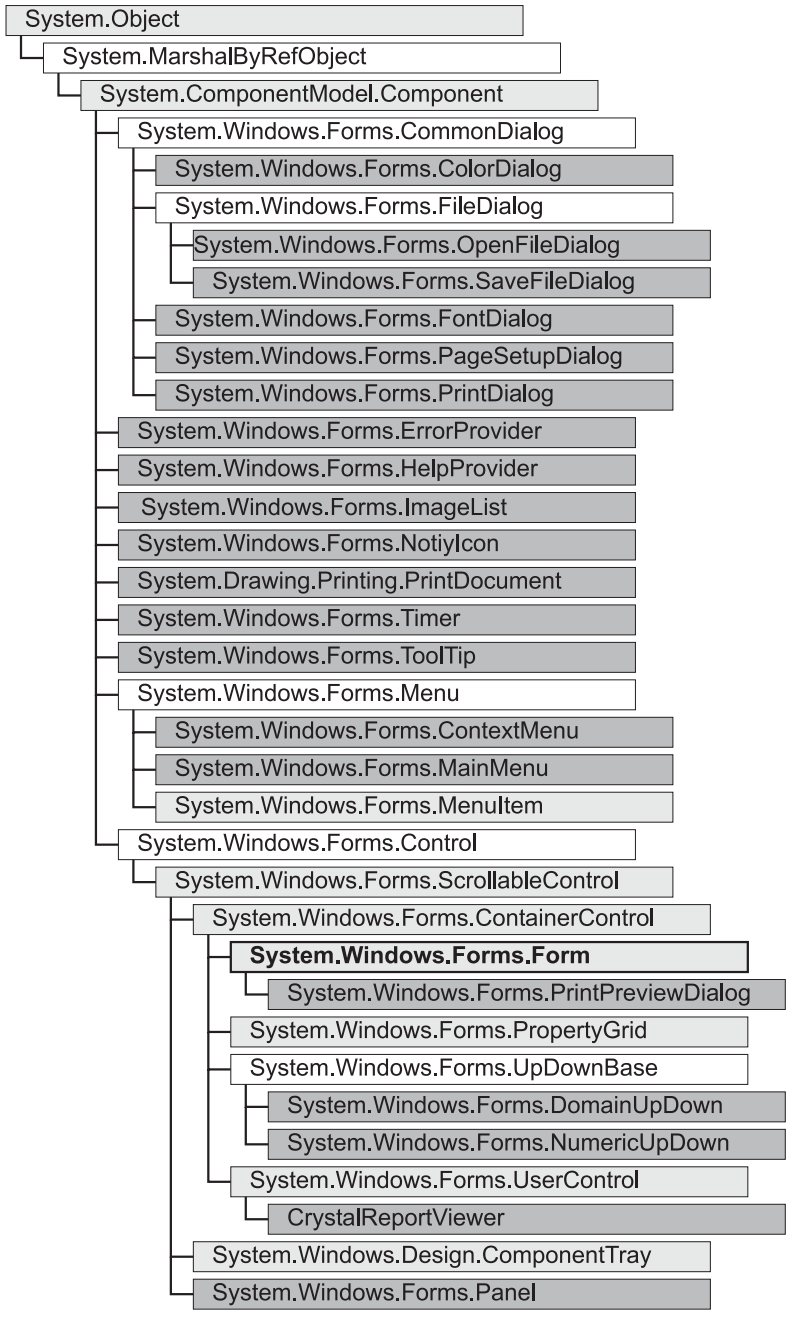


Abbildung 13.1: Die Einordnung der Klasse `Form` und verschiedener Komponenten aus der Toolbox in die Klassenhierarchie, wobei abstrakte Klassen durch einen weißen, in der Toolbox befindliche Komponenten durch einen dunklen Hintergrund gekennzeichnet sind.

13.1.1 Formularobjekte

Wenn Sie ein neues Codegerüst generiert und damit angefangen haben, einzelne Steuerelemente einzufügen und zu verdrahten, werden Sie sich vielleicht fragen, wo denn eigentlich das Formularobjekt selbst bleibt und wie es programmatisch anzupacken ist. Nun, wenn Sie die Formularklasse ausstatten (und genau so ist das Codegerüst ja angelegt), implementieren Sie das Verhalten aller möglichen Objekte, die jemand durch Instanziierung dieser Klasse irgendwann mal ins Leben ruft – und sei es auch nur ein einziges Objekt, wie es für gewöhnlich der Fall ist. Ihre normale Sicht bei der Programmierung ist also die Innensicht der Formularklasse. Statt eine Objektvariable für das konkrete Formularobjekt zu vereinbaren und damit zu arbeiten, greifen Sie über das Schlüsselwort `this` auf die jeweils aktuelle Instanz zu (die im Falle des Hauptformulars in `Main()` generiert wird):

```
this.Text = "Hauptformular";
```



Die explizite Qualifizierung über `this` *macht sofort klar, dass es dabei um eine Eigenschaft oder Methode der Formularklasse geht. Da C# kein Äquivalent zur* `with`*-Anweisung von VB und Delphi kennt, ist sie rein technisch gesehen redundant. (Dass der Bezeichner* `this` *selbst alles andere als überflüssig ist, wird unter anderem in diesem Abschnitt demonstriert.)*

Was nicht ist, kann aber noch werden. So ist es durchaus legitim, gleich mehrere Instanzen einer Formularklasse ins Leben zu rufen und an je eigene Objektvariablen zu binden, beispielsweise an statische Datenfelder der Klasse selbst:

```
static Form f1;
static Form1 f2;
static Form1 f3;

static void Main()
{
  f1 = new Form1();
  f2 = new Form1();
  f3 = new Form1();
  f1.Text = "Erstes Formular";
  f2.Text = "Zweites Formular";
  f3.Text = "Drittes Formular";
  f1.Show();
  f2.Show();
  Application.Run(f3);
}
```

Beachten Sie, dass die Methode `Main()` statisch ist und damit der statischen Instanz der Formularklasse angehört. Sie wird – im Gegensatz zum Konstruktor der Klasse – wirklich nur einmal ausgeführt. Wenn Sie das Codegerüst in dieser Form starten, erhalten Sie drei gleiche Formulare, die sich

getrennt voneinander manipulieren (verschieben, vergrößern usw.) lassen. Einzig beim Schließen tut sich das dritte Formular als Hauptformular hervor und beendet gleich die gesamte Anwendung, während die ersten beiden jeweils nur sich selbst schließen. Nachdem die Formularobjekte an statische Variablen der Klasse gebunden sind, »sehen« sie sich gegenseitig, und es ist ein Leichtes, das beim Schließen des Formulars generierte `Close`-Ereignis so zu behandeln, dass sich die Anwendung von jedem der drei Fenster aus beenden lässt:

```
private void Form1_Closed(object sender, System.EventArgs e)
{
  if (f3 != this)
    f3.Close();
}
```

Grundausstattung

Die Abstammung von `Form`, `ContainerControl` und `Control` (vgl. Abbildung 13.1) beschert einem Formularobjekt ein wahrhaft stattliches Erbe an Eigenschaften, Methoden und Ereignissen. Abbildung 13.2 zeigt des besseren Überblicks halber eine Bildmontage der Elemente, die der VS.NET-Editor für `this` auflistet. Sie finden darunter auch die `protected`-Elemente, die nur in abgeleiteten Klassen verwendbar sind.

Auch wenn jedes der Elemente seine feste Rolle im Gefüge aus .NET und Windows spielt, werden Sie die meisten davon kaum jemals brauchen. Dennoch sollten Sie sich ruhig ein paar Minuten Zeit nehmen, um die Elemente, deren Bezeichner an sich bereits recht vielsagend sind, einmal bewusst zur Kenntnis zu nehmen und vielleicht sogar das eine oder andere anhand der Online-Hilfe zu studieren. Da viele davon über die Basisklasse `Control` vererbt werden, findet sich ein großer Teil der Elemente auch bei den Steuerelementen wieder, besonders wenn Sie eigene Benutzersteuerelemente programmieren wollen (vgl. auch Tabelle 13.1. bis Tabelle 13.3).

Fingerübung – Eigenschaften vom Programmcode aus ändern

Damit Sie ein Gefühl dafür bekommen, wie Sie die Eigenschaften eines Formularobjekts vom Programm aus manipulieren, erweitert der folgende Code unser Beispiel mit den drei Formularobjekten um vier Features:

1. Jedes Formular erhält einen anderen Fensterstil.
2. Jedes Formular erhält eine andere Bitmap als Hintergrundbild.
3. Die drei Formulare werden nebeneinander platziert.
4. Die Formulare kleben aneinander, auch wenn sie verschoben oder in ihrer Größe verändert werden.



Abbildung 13.2: Ausstattung eines Formularobjekts

Punkt 1 ist schnell abgehakt. Beim Studium der `Form`-Elemente ist Ihnen vielleicht auch die Eigenschaft `FormBorderStyle` über den Weg gelaufen. Sie trägt den gleichnamigen `enum`-Typ `FormBorderStyle`, der sieben Stilattribute anbietet. Die Standardvorgabe ist `Sizable`, drei Alternativen dazu sind:

```
f1.FormBorderStyle=FormBorderStyle.Fixed3D;           // feste Größe
f2.FormBorderStyle=FormBorderStyle.None;              // Rahmenlos
f3.FormBorderStyle=FormBorderStyle.SizableToolWindow; // kein Systemmenü
```

`f1` besitzt nun eine feste Größe, von `f2` ist nur der Clientbereich zu sehen und `f3` verfügt nur noch über eine SCHLIEßEN-Schaltfläche in der Titelleiste, kein Systemmenü mehr.

Punkt 2 erfordert nicht viel mehr Mühe. Jedes Formular bekommt sein Bild, das in diesem Fall im Projektverzeichnis gelegen ist.

```
// Hintergrundbilder
f1.BackgroundImage = Image.FromFile("..\\..\\Art Farmer.jpg");
f2.BackgroundImage = Image.FromFile("..\\..\\Klezmorim.jpg");
f3.BackgroundImage = Image.FromFile("..\\..\\Mexican2.jpg");
```

Statt der statischen Methode `Image.FromFile()` hätte man hier auch eine Instanz der Klasse `Bitmap` zuweisen können, da eine implizite Typumwandlung von `Bitmap` nach `Image`, dem Datentyp der `BackgroundImage`-Eigenschaft, definiert ist:

```
f1.BackgroundImage = new Bitmap("..\\..\\Art Farmer.jpg");
```

Punkt 3 wird schon etwas kniffliger. Zunächst fehlt so etwas wie der Bezugspunkt der ganzen Geschichte. Warum nicht die Mitte? Dies lässt sich als Voreinstellung für alle Formularobjekte im Designer bewerkstelligen. Somit muss nur noch das eine nach rechts und das andere nach links, jeweils an den Rand des dritten, mittleren Formulars verschoben werden. Den Code dafür in die Methode `Main()` zu packen, würde zwar funktionieren, mit Blick auf Punkt 4 sollte dafür aber besser eine zentralere Lösung her. Wann immer ein Formular verschoben wird, tritt das `Move`-Ereignis ein, bei Größenänderungen ist es das `Resize`-Ereignis. Das `Move`-Ereignis tritt auch sicher als Folge des `Show()`-Aufrufs ein, so dass für eine Initialisierung zu Beginn gleichfalls gesorgt ist. Die Lösung sieht so aus:

```
private void Form1_Move(object sender, System.EventArgs e)
{
  ArrangeForms(sender);
}

private void Form1_Resize(object sender, System.EventArgs e)
{
  ArrangeForms(sender);
}

private void ArrangeForms(object sender)
{
  if (sender == f3)
  {
    f1.Top = f2.Top = f3.Top;
    f1.Left = f3.Left - f1.Width;
    f2.Left = f3.Left + f3.Width;
  }
}
```

Da beide Behandlungsroutinen fast dasselbe machen, bietet es sich an, die Platzierung in eine eigene Methode auszulagern und diese von beiden Behandlungsmethoden aus aufzurufen. Das vermeidet die unschöne Codewiederholung und konzentriert die Problemlösung auf einen Punkt. Abbildung 13.3 zeigt das Ergebnis. Das linke Formular hat den Stil `Fixed3D`, das mittlere den Stil `FixedToolWindow`. Allein das mittlere Formular, das den Stil

`SizableToolWindow` trägt, erlaubt Größenveränderungen. Es ist in Abbildung 13.3 denn auch etwas schmaler und höher als die anderen beiden. Das rechte Fenster ist »um Rand und Titelleiste« kleiner als das linke, was daher kommt, dass der Designer die Formulargröße standardmäßig über den *Clientbereich* initialisiert:

```
this.ClientSize = new System.Drawing.Size(320, 316);
```

Es ist üblich, bei einem Formular vom Clientbereich aus zu denken (und zu rechnen), da dies der Bereich ist, in dem alle Ausgaben stattfinden und auch die Steuerelemente dargestellt werden. Gleiche Formulargrößen erhalten Sie durch die folgende Initialisierung:

```
this.Size = new System.Drawing.Size(320, 316);
```

Es hat allerdings keinen Sinn, die beiden Zeilen in der Methode `InitializeComponents()` auszutauschen, da der Designer auf seine Version »besteht« und sofort wieder ändert. Ergänzen Sie die Zeile also gegebenenfalls im Konstruktor nach dem Aufruf von `InitializeComponents()`.

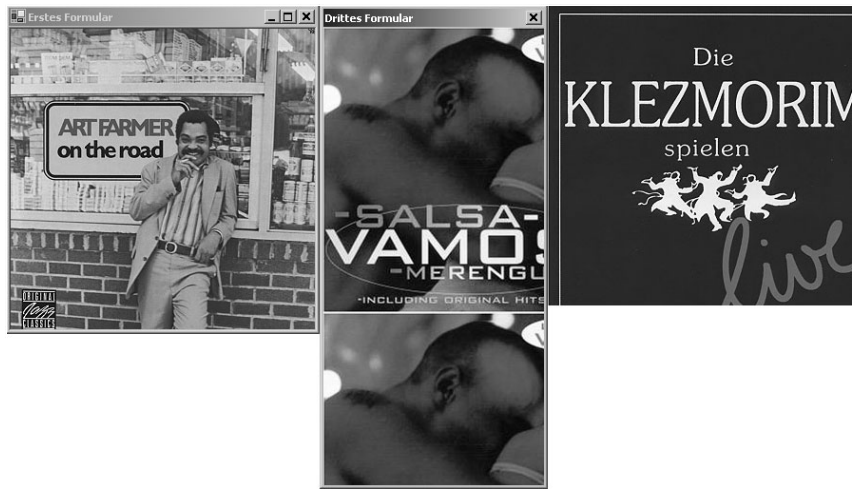


Abbildung 13.3: Die drei Formulare kleben aneinander

Haben Sie beim Ausprobieren bemerkt, dass der Code noch nicht ganz perfekt ist? Tatsächlich kann der Benutzer das linke Fenster mit der Maus verschieben, ohne dass die anderen beiden folgen. Der erste Gedanke wird wohl sein: »Das ist ja trivial, da muss man einfach nur … » – und hält genau so lange, bis man es ausprobiert. Versuchen Sie trotzdem erst einmal in Eigenregie, dem Problem auf die Schliche zu kommen.

Vielleicht hilft Ihnen bei Ihrer Überlegung ja der Gedanke, dass man auch einem konkreten Objekt zur Laufzeit nicht nur beliebig Behandlungsroutinen zuordnen, sondern auch wieder wegnehmen kann – ein Vorgang, der in

der Routine `InitializeComponent()` für `this` und somit für alle Objekte der Klasse geschieht. Das Muster dafür wäre:

```
MyForm1.Click -= new System.EventHandler(this.Form1_Click);
MyForm1.Click += new System.EventHandler(this.Alternate_Click);
```



Der vorgestellte Beispielcode ist als Fingerübung für den Umgang mit mehreren Formularobjekten der derselben Klasse gedacht – nicht als Designgrundlage für Mehrformular-Anwendungen. Solche Anwendungen sehen für gewöhnlich ein Hauptformular vor, dem Instanzen einer oder mehrerer anderer Formularklassen untergeordnet sind. Das Hauptformular startet und verwaltet diese Instanzen. Noch strikter ist die klassische MDI-Anwendung (Multiple Dokument Interface) organisiert, wo die untergeordneten MDI-Fenster auf den Clientbereich des MDI-Hauptformulars beschränkt sind.

Controls-Auflistung und Z-Reihenfolge

Die wohl wichtigste Eigenschaft des `Form`-Objekts ist die (von `ContainerControl` vererbte) `Controls`-Auflistung, die den Typ `ControlCollection` trägt. Über diese Auflistung verwaltet das Formularobjekt seine Steuerelemente. Wenn Sie einen Blick auf das Codegerüst eines Formulars werfen, in das Sie per Designer ein paar Steuerelemente eingefügt haben, finden Sie eine Anweisung nach folgendem Muster:

```
this.Controls.AddRange(new System.Windows.Forms.Control[] {
                                this.checkBox1,
                                this.progressBar1,
                                this.radioButton1,
                                this.radioButton2,
                                this.radioButton3,
                                this.label1,
                                this.textBox1,
                                this.button1});
```

Das `Form`-Objekt wertet diese Auflistung unter anderem aus, um die Ereignisse aus seiner Nachrichtenschleife »an das Steuerelement zu bringen«. Dabei induziert die Reihenfolge der Steuerelemente in der `Controls`-Auflistung die so genannte Z-REIHENFOLGE, die das logische (und eventuell optische) Hintereinander der Steuerelemente beim Zeichnen und für die Zuordnung positionsgebundener Ereignisse (Mausereignisse) bestimmt.

Die Logik der Zustellung eines Mausereignisses `MouseXxx` sieht so aus: Das `Form`-Objekt durchsucht seine `Controls`-Auflistung von Anfang an, bis es auf ein Steuerelement trifft, dessen Fensterbereich die Position überdeckt, an der das Ereignis stattgefunden hat, und ruft dessen `OnMouseXxx`-Methode auf. Findet sich an der Position kein Steuerelement, ist das Formular selbst der Adressat. Gezeichnet wird andererseits von hinten nach vorne, das heißt, das letzte Steuerelement in der Auflistung zuerst und das erste zuletzt.

Damit geht ein Mausereignis immer an das Steuerelement, das der Benutzer an der jeweiligen Position auch sieht.



Die Grundlage dieser Suche ist die API-Funktion `WindowFromPoint()`*, die einen Windows-Fensterhandle zurückgibt – entweder den eines (dann in der* `Controls`*-Auflistung zu suchenden) Steuerelements oder den Handle des Formularobjekts selbst. Windows unterhält eine eigene Fensterliste in Z-Reihenfolge; die .NET-Laufzeitumgebung sorgt für eine Synchronisation mit der* `Controls`*-Auflistung (in beiden Richtungen).*

Die sich zunächst einmal aus der Einfügereihenfolge ergebende Z-Reihenfolge lässt sich zur Laufzeit nicht nur ermitteln, sondern auch frei gestalten. Von der `Controls`-Eigenschaft her stehen Ihnen dafür die Methoden `GetChildIndex()` und `SetChildIndex()` zur Verfügung, aufseiten einzelner `Control`-Objekte deren Methoden `BringToFront()` und `SendToBack()`. Für ein Steuerelement `ctl` ist jeweils äquivalent:

```
ctl.BringToFront();  // Controls.SetChildIndex(ctl, 0);
ctl.SendToBack();    // Controls.SetChildIndex(ctl, Controls.Count-1);
```

Ereignisse

Auch wenn ein Formularobjekt nach dem Konstruktoraufruf bereits als fertig bezeichnet werden kann, erblickt es erst mit dem `Show()`-Aufruf das »Licht der Welt« und beginnt aktiv am Geschehen teilzunehmen. Die Ereignisfolge für ein Formular, das nichts weiter als einen `Show()`-Aufruf und dann einen `Close()`-Aufruf »erlebt«, sieht so aus:

`Move`	Formular wird in Position gebracht
`LocationChanged`	Eigenschaft `Location` geändert
`StyleChanged`	Formular erhält seinen Fensterstil
`Load`	Formular wird geladen
`Activated`	Formular wird aktiviert und erhält Fokus
`Paint`	Formular zeichnet sich
`Closing`	Formular soll geschlossen werden
`Closed`	Formular wird geschlossen
`Deactivate`	Formular wird deaktiviert und verliert Fokus

Das Gegenstück von `Load` ist `Close`. Beide Ereignisse treten nur einmal im Leben eines Formulars auf (Visual Basic-Programmierer aufgepasst: die Möglichkeit eines *Unload* ist nicht vorgesehen) und kennzeichnen den Beginn und

das Ende der Formularaktivität. Ein expliziter oder impliziter (beispielsweise über die SCHLIEßEN-Schaltfläche eingeleiteter) Aufruf der `Close()`-Methode generiert zunächst das `Closing`-Ereignis. Bei Behandlung dieses Ereignisses besteht die Möglichkeit, die `Cancel`-Eigenschaft des Ereignisobjekts auf `true` zu setzen und den SCHLIEßEN-Befehl damit effektiv zu verhindern. Das Paar `Activated` und `Deactivate` rahmt die Vordergrundaktivität des Formulars ein. Diese Ereignisse treten immer abwechselnd auf und kennzeichnen Beginn und Ende der interaktiven Bearbeitungsperioden des Formulars. Die Ereignisse `Enter` und `Leave` bekommt das `Form`-Objekt nicht: sie sind für Steuerelemente gedacht und zeigen an, wann ein Steuerelement den Eingabefokus erhält bzw. verliert.

Tabelle 13.1 gibt einen Überblick über die wichtigsten Ereignisse des `Form`-Objekts.

Tabelle 13.1: Wichtige Ereignisse des `Form`-Objekts

Ereignis	vererbt von	Bedeutung
`Activated`	`Form`	Tritt ein, wenn das Formular als Vordergrundfenster aktiviert wird und den Eingabefokus erhält.
`Closed`	`Form`	Tritt nach `Closing` ein, wenn das Formular geschlossen wird (Objekt kann Schließen nicht mehr verhindern).
`Closing`	`Form`	Tritt ein, wenn das Formular geschlossen werden soll. Das Objekt kann den Schließen-Vorgang verhindern.
`Deactivate`	`Form`	Tritt ein, wenn ein anderes Formular als Vordergrundfenster aktiviert wird und das Formular den Eingabefokus verliert.
`DragDrop`	`Control`	Tritt ein, wenn das Formular Ziel einer Drag&Dop-Operation ist. Geht mit `DragLeave` einher.
`DragEnter`	`Control`	Tritt ein, wenn der Mauszeiger während einer Drag&Drop-Operation in den Bereich des Formulars eintritt.
`DragLeave`	`Control`	Tritt ein, wenn der Mauszeiger während einer Drag&Drop-Operation den Bereich des Formulars verlässt. Das kann auch passieren, wenn er in ein Steuerelement eintritt, dessen `AllowDrop`-Eigenschaft `true` ist.

Tabelle 13.1: Wichtige Ereignisse des `Form`-Objekts (Forts.)

Ereignis	vererbt von	Bedeutung
`DragOver`	`Control`	Tritt ein, wenn der Mauszeiger während einer Drag&Drop-Operation im Bereich des Formulars bewegt wird (vgl. `MouseMove`). Betritt der Mauszeiger ein Steuerelement, dessen `AllowDrop`-Eigenschaft `true` ist, signalisiert das Formular `DragLeave` . Das Steuerelement signalisiert `DragEnter` und ab jetzt auch `DragOver`.
`Enter`	`Control`	Wird vom `Form`-Objekt überschrieben und unterdrückt.
`GiveFeedback`	`Control`	Tritt während einer Drag&Drop-Operation ein. Das Zielobjekt (Formular, Steuerelement) beantwortet damit eine Frage des Quellobjekts, welche Operationen (Effekte) es für den Typ des gezogenen Objekts unterstützt.
`GotFocus`	`Control`	Tritt ein, wenn das Formular den Fokus erhalten hat. (Dieses Ereignis ist veraltet und wird daher im EIGENSCHAFTEN-Fenster nicht mehr angeboten, steht aber dennoch zur Verfügung. Seine Rolle übernimmt nun das `Activate`-Ereignis.)
`Invalidated`	`Control`	Tritt ein, wenn das Formular (Steuerelement) aufgrund eines `Invalidate()`-Aufrufs neu gezeichnet werden soll.
`Layout`	`Control`	Tritt ein, wenn dem Formular (oder Steuerelement) ein neues Steuerelement hinzugefügt wird oder sich das Layout des Formulars aus anderen Gründen ändert. Lässt sich durch `SuspendLayout()` abschalten und durch `ResumeLayout()` wieder einschalten.
`Leave`	`Control`	Wird vom `Form`-Objekt unterdrückt.
`Load`	`Form`	Tritt als Folge des ersten `Show()`-Aufrufs für das Formular ein. Dieses Ereignis wird im Allgemeinen behandelt, um einmalige Initialisierungen unmittelbar vor Anzeige des Formulars vorzunehmen.
`MdiChildActivated`	`Form`	Tritt ein, wenn die `IsMdiContainer`-Eigenschaft gesetzt ist und ein untergeordnetes Formular aktiviert wird.

Ereignis	vererbt von	Bedeutung
`Paint`	`Control`	Tritt ein, wenn das Formular oder ein Teil davon neu gezeichnet werden soll.
`QueryContinueDrag`	`Control`	Tritt im Verlauf eines Drag&Drop-Vorgangs auf; ermöglicht es, die Quelle darüber zu informieren, ob es überhaupt Sinn macht, den Vorgang fortzusetzen.
`Resize`	`Control`	Tritt auf, wenn sich die Formulargröße ändert (nicht aber bei der Initialisierung des Objekts).
`XxxChanged`	`Control/Form`	Wert der Eigenschaft `Xxx` hat sich geändert.

Tabelle 13.1: Wichtige Ereignisse des `Form`-Objekts (Forts.)

Beim Durchstöbern der Klassenelemente nach Ereignissen, die ein Formularobjekt prinzipiell behandeln kann (vgl. Abbildung 13.2), fallen insbesondere zwei Benennungsmuster ins Auge: Es gibt zahlreiche `XxxChanged`-Ereignisse, die mit `Xxx`-Eigenschaften korrelieren und noch zahlreichere `OnXxx()`-Methoden, die für `Xxx`-Ereignisse zuständig sind.

Die meisten `XxxChanged`-Ereignisse – wie `LocationChanged` oder `StyleChanged` – berichten schlicht darüber, dass sich die eine oder andere Eigenschaft des Formulars geändert hat und ermöglichen es dem Objekt, darauf zu reagieren, sofern es das Klassendesign erforderlich macht.

Mit den `OnXxx()`-Methoden hat es etwas anderes auf sich. Es handelt sich hier um Methoden, die für das Auslösen des entsprechenden Ereignisses verantwortlich sind. Die `Form`- bzw. `Control`-Klasse vererbt diese Methoden als virtuelle Elemente und ermöglicht es so, durch Überschreibung exklusiv in den Ereignisablauf einzugreifen.

OnXxx() überschreiben vs. Behandlungsmethoden

In der .NET-Dokumentation wird verschiedentlich empfohlen, man solle zur Behandlung eines Ereignisses in einer abgeleiteten Klasse schlicht die entsprechende `OnXxx()`-Methode der Basisklasse überschreiben und die Behandlung dann darin vornehmen. Das ist vom Prinzip her zwar richtig, aber nicht in jedem Fall im Sinne des Erfinders. Die gewöhnliche Behandlung von Ereignissen sollten den Ereignisbehandlungsmethoden vorbehalten bleiben, deren Codegerüste der Designer automatisch einfügt und verdrahtet, wenn Sie im EIGENSCHAFTEN-Fenster auf das jeweilige Ereignis doppelklicken oder einen Methodennamen eintragen.

Das Überschreiben einer `OnXxx()`-Methode empfiehlt sich nur, wenn Sie mit Blick auf die weitere Ableitung Ihrer Klasse in die Funktionsweise der Ereignissignalisierung selbst eingreifen wollen – beispielsweise, um das Ereignis

zu unterdrücken oder durch Veränderung des Ereignisobjekts zu gestalten. Vielfach werden `OnXxx()`-Methoden auch implementiert, um sekundäre Ereignisse zu signalisieren, was prinzipiell aber auch von einer Behandlungsmethode aus möglich ist.

Der Unterschied zwischen `OnXxx()`-Methoden und gewöhnlichen Behandlungsmethoden ist die Aufruf-Reihenfolge. Implementiert eine abgeleitete Klasse eine `OnXxx()`-Methode, bekommt diese das Ereignis an vorderster Stelle zu sehen und kann die weitere Behandlung des Ereignisses gestalten.

Das folgende Beispiel einer `OnPaint()`-Überschreibung und `Paint`-Behandlung demonstriert die drei Punkte für den Eingriff ins Geschehen.

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
  // TODO spezieller Code der Klasse
  Console.WriteLine("Vor dem Paint-Ereignis");
  base.OnPaint(e);   // Ereignis signalisieren. Führt zur Abarbeitung
  // TODO spezieller Code der Klasse
  Console.WriteLine("Nach dem Paint-Ereignis");
}

private void Form1_Paint(object s, System.Windows.Forms.PaintEventArgs e)
{
  Console.WriteLine("Paint-Ereignis");
}
```

Die Debugger-Ausgabe in Folge eines `Invalidate()`- oder `Refresh()`-Aufrufs lautet:

```
Vor dem Paint-Ereignis
Paint-Ereignis
Nach dem Paint-Ereignis
```

Tatsächlich kann die Klasse ein `Xxx`-Ereignis auch unterdrücken, indem sie den `base.OnXxx()`-Aufruf einfach unterlässt. Ein Eingriff in solche Mechanismen erfordert allerdings ein sorgfältiges Studium des betreffenden Ereignisschemas. So kann es sein, dass Sie damit (ungewollt) auch sekundäre Ereignisse unterdrücken, die die Basisklasse in ihrer `OnXxx()`-Variante pflegt bzw. signalisiert.



Es gibt keine Garantie dafür, dass ein absichtlich ausgelassener Aufruf der Basisklassenvariante die entsprechende Funktionalität wirklich unterdrückt. Seltsame Ergebnisse bringt beispielsweise ein fortgeschritteneres Szenario wie das folgende:

```
class MyButton : System.Windows.Forms.Button
{
  protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
  {
```

```
    // base.OnPaint(e);
    e.Graphics.FillRectangle(System.Drawing.Brushes.White, 0, 0, 10, 10);
  }
}
```

Die Klasse `MyButton` *ist von* `Button` *abgeleitet und hat ihre eigene Zeichenroutine, die* `base.OnPaint()` *zunächst nicht aufruft. Fügt man nun ein Objekt dieses Typs in das Formular ein*

```
private MyButton myButton;
public Form1()
{
  InitializeComponent();
  this.myButton=new MyButton();
  this.myButton.Location = new System.Drawing.Point(16, 8);
  this.myButton.Size = new System.Drawing.Size(80, 40);
  this.myButton.Text = "MyButton";
  this.Controls.Add(myButton);
}
```

ist das sichtbare Ergebnis ein kleines weißes Rechteck auf einem schwarzen Rechteck (Abbildung 13.4 links). Fügt man dem Formular zudem ein echtes `Button`*-Steuerelement hinzu, funkt dessen Zeichenroutine plötzlich »dazwischen« (Abbildung 13.4 Mitte), da* `Button` *wohl einen statischen Puffer zum Zeichnen benutzt. Erst der Aufruf von* `base.OnPaint()` *bereitet dem Spuk ein Ende.*

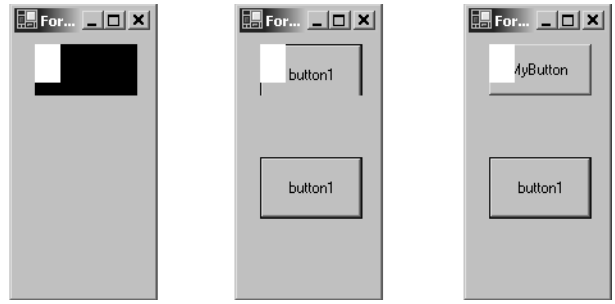


Abbildung 13.4: Seltsame Effekte im Zusammenhang mit der Paint-Routine

`OnXxx()`-Überschreibungen werden häufig dafür benutzt, sekundäre Ereignisse zu generieren, obwohl auch dies prinzipiell von einer Behandlungsmethode aus möglich ist. Ein Beispiel für eine sekundäres Ereignis wäre `Click`. Es wird aus der Abfolge `MouseDown` und `MouseUp` generiert – und zwar systemseitig, so dass eine Unterdrückung von `MouseDown` das `Click`-Ereignis nicht betrifft.

Codebeispiel – der Dreifachklick

Das folgende Codebeispiel *Dreifachklick* demonstriert die Implementierung eines `TripleClick`-Ereignisses auf der Basis einer `OnMouseUp()`-Überschreibung. Die Spezifikation dafür ist klar: Das Ereignis soll analog zu `DoubleClick`

unmittelbar vor dem dritten `MouseUp` in Folge eintreffen – vorausgesetzt, das Zeitintervall `SystemInformation.DoubleClickTime` wurde zwischen je zwei Klicks nicht überschritten. Der Code behandelt die relevanten Mausereignisse – auch das neue Ereignis – und dokumentiert die Abfolge der Ereignisse im Ausgabefenster des Debuggers. Damit das `TripleClick`-Ereignis vor dem dritten `MouseUp`-Ereignis eintrifft, darf der Aufruf von `base.OnMouseUp` nicht vor der `TripleClick`-Signalisierung erfolgen.

Die Implementierung selbst ist relativ einfach. Sie pflegt zwei als Klassenfelder vereinbarte Zustandsvariablen, um die Anzahl der erfolgten Klicks und den Zeitpunkt des jeweils letzten `MouseUp`-Ereignisses festzuhalten und bildet daraus ein Kriterium, das nach dem dritten Klick in Folge erfüllt ist.

```
int Counter;
DateTime TimeStamp;
public event System.EventHandler TripleClick;
protected override void OnMouseUp(System.Windows.Forms.MouseEventArgs e)
{
  System.TimeSpan ts = DateTime.Now - TimeStamp;   // Zeitdifferenz
  if (ts.TotalMilliseconds <= SystemInformation.DoubleClickTime)
  {                                 // erst beim zweiten Klick betreten
    if (++Counter == 2)             // Schon der dritte Klick in Folge?
    {
      Counter = 0;
      if (TripleClick != null)      // Ereignis sigalisieren?
        TripleClick(this, new System.EventArgs());
    }
  }
  else
  {
    Counter = 0;                    // Zeitdifferenz war zu groß
  }
  TimeStamp = DateTime.Now;         // Klickzeitpunkt merken
  base.OnMouseUp(e);                // Basisklassenvariante aufrufen
}

private void Form1_MouseDown(object sender, MouseEventArgs e)
{
  Console.WriteLine("MouseDown ");
}

private void Form1_MouseUp(object sender, MouseEventArgs e)
{
  Console.WriteLine("MouseUp ");
}

private void Form1_Click(object sender, System.EventArgs e)
{
  Console.WriteLine("Click ");
}
```

```
private void Form1_DoubleClick(object sender, System.EventArgs e)
{
  Console.WriteLine("DoubleClick ");
}

private void Form1_TripleClick(object sender, System.EventArgs e)
{
  Console.WriteLine("TripleClick");
}
```

Die im Ausgabefenster des Debuggers erscheinenden Meldungen sehen wie erwartet aus:

```
MouseDown
Click
MouseUp
MouseDown
DoubleClick
MouseUp
MouseDown
Click
TripleClick
MouseUp
```

13.1.2 Bildlaufleisten des Formulars verwenden

Dass die Klasse `Form` von `ScrollableControl` abstammt, schlägt sich natürlich nieder, und eine Bestandsaufnahme der von dieser Klasse vererbten Elemente fördert eine Reihe von Eigenschaften und Methoden (jedoch keine Ereignisse) zutage, die direkt oder indirekt mit der Anzeige von Bildlaufleisten im Formular zu tun haben. Tabelle 13.2 gibt einen Überblick über die in diesem Zusammenhang wichtigen Eigenschaften und Methoden.

Tabelle 13.2: Eigenschaften und Methoden des Formulars, die im Zusammenhang mit dem Bildlauf stehen.

Eigenschaft/Methode	Bedeutung
`bool AutoScroll {get; set;}`	Wenn `true`, dann ermöglicht das Formular den automatischen Bildlauf.
`Size AutoScrollMinSize {get; set;}`	Wenn (0,0), wird die Anzeige der Bildlaufleisten unterdrückt. Andernfalls bestimmt dieser Wert zuzüglich `AutoScrollMargin` die Abmessungen des Bereichs, für den das Formular einen Bildlauf durchführt und die Bildlaufleisten in Abhängigkeit von `AutoScroll` automatisch einblendet, wenn der Clientbereich des Formulars nicht ausreicht. Das Setzen dieser Eigenschaft setzt `AutoScroll` automatisch auf `true`.

Eigenschaft/Methode	Bedeutung
`Size AutoScrollMargin {get; set;}`	Dieser Wert bestimmt den zusätzlichen Randbereich, um den das Formular seine durch `AutoScrollMargin` definierte Ansicht erweitert und per Bildlauf einsehbar macht. Das Setzen dieser Eigenschaft setzt `AutoScroll` automatisch auf `true`.
`Point AutoScrollPosition {get; set;}`	Koordinaten des linken oberen Punkts für den im Clientbereich angezeigten Ausschnitt. Das Formular passt diesen Wert unter Berücksichtigung von `AutoScrollMinSize` und `AutoScrollMargin` automatisch an, wenn die Position (auch nur in einer Dimension) zu weit gesetzt wird. Setzt `AutoScroll` auf `true`. Achtung: Wird der Eigenschaft ein `Point`-Objekt zugewiesen, übernimmt sie die Koordinaten mit umgekehrten Vorzeichen, macht aus positiven Werten also negative und umgekehrt.
`bool HScroll {get; set;}`	Der Wert dieser Eigenschaft hat scheinbar keinen Einfluss auf das Geschehen und wird auch nicht gepflegt.
`void ScrollControlIntoView()`	Verschiebt den sichtbaren Ausschnitt (Ansicht) so, dass sichergestellt ist, dass das angegebene `Control`-Objekt aus der `Controls`-Auflistung sichtbar wird. Passt die Eigenschaft `AutoScrollMinSize` an, falls deren Wert dafür nicht ausreicht.
`bool VScroll {get; set;}`	Der Wert dieser Eigenschaft hat scheinbar keinen Einfluss auf das Geschehen und wird auch nicht gepflegt.

Die Handhabung der Bildlaufleisten des Formulars ist denkbar einfach. Im Grunde genommen brauchen Sie dafür nur die Eigenschaft `AutoScrollMinSize` auf den Bereich zu setzen, den Sie per Bildlauf überstreichen wollen. Alles andere passiert als Seiteneffekt bzw. automatisch: `AutoScroll` wird dabei auf `true` gesetzt, und das Formular sorgt bei einem Fokuswechsel automatisch dafür, dass immer das Steuerelement, das den Fokus erhält, im sichtbaren Ausschnitt liegt. Um einen Bildlauf zu einem bestimmten Ausschnitt durchzuführen, setzen Sie die Eigenschaft `AutoScrollPosition` auf den linken oberen Punkt des Ausschnitts. Um einen Bildlauf zu einem bestimmten Steuerelement durchzuführen, benutzen Sie die Methode `ScrollControlIntoView()`.

Viel mehr gibt es nicht zu sagen, außer dass ein Abfragen der Eigenschaft `AutoScrollPosition` aus unerfindlichen Gründen ein umgekehrtes Vorzeichen für die Koordinaten liefert.

```
AutoScrollPosition = new Point(10, 10);
string s = AutoScrollPosition.ToString();   // s = {X = -10,Y = -10}
```

In Formularen mit Statusleiste können Sie den automatischen vertikalen Bildlauf des Formulars leider nicht einsetzen, da die Statusleiste im Clientbereich sitzt und dann nicht mehr ständig sichtbar ist. Einen Lösungsvorschlag finden Sie im Abschnitt »Ärger mit der `AutoScroll`*-Eigenschaft des Formulars« auf Seite 625.*

INFO

Codebeispiel – Anzeige eines Bildes mit Bildlauf

Das folgende Codebeispiel *FormBildlauf* ist eine einfache Bildanzeige. Die Anwendung erfragt den Namen einer *.jpg*-Datei und liest deren Inhalt als Hintergrundbild des Formulars ein. Dann setzt sie die `AutoScrollMinSize`-Eigenschaft auf die Abmessungen des Bildes, um einen Bildlauf zu ermöglichen, falls dieses nicht in die Ansicht passt. Der Anwender kann den Bildlauf sowohl über die Bildlaufleisten als auch per Mausklick abwickeln. Alles, was mit den Bildlaufleisten zu tun hat, erledigt das Formular von sich aus. Die `MouseDown`-Behandlung nimmt hingegen einen aktiven Bildlauf vor, der den jeweils angeklickten Punkt zum Mittelpunkt des neuen Ausschnitts macht – vorausgesetzt, der Bildrand lässt dies zu.

```
private void Form1_Load(object sender, System.EventArgs e)
{
  OpenFileDialog ofd = new OpenFileDialog();
  ofd.Filter = "*.jpg| *.jpg";
  if (ofd.ShowDialog() == DialogResult.OK)
    this.BackgroundImage = new Bitmap(ofd.OpenFile());
  this.Activate();
  this.AutoScrollMinSize = this.BackgroundImage.Size;
}

private void Form1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{
  Point p = this.AutoScrollPosition;
  p.X = -p.X + e.X - this.ClientRectangle.Width/2;
  p.Y = -p.Y + e.Y - this.ClientRectangle.Height/2;
  this.AutoScrollPosition = p;   // Achtung: AutoScrollPosition
                                 // kehrt Vorzeichen um
  Text = AutoScrollPosition.ToString();
}
```



Abbildung 13.5: Drei unterschiedliche Bildausschnitte eines Hintergrundbilds

14 Grafik

Für Zeichenoperationen ist unter Windows traditionell das GDI (Graphics Device Interface) zuständig, das eine auf den ersten Blick zwar recht eigenwillige, auf den zweiten Blick aber doch sehr stimmige Verkapselung von Zeichenflächen, Zeichenwerkzeugen und Ausgabeoperationen darstellt. Der grundlegende Gedanke des GDI ist, dass jede Operation in einem so genannten Grafikkontext stattfindet, der das jeweilige Ausgabegerät (im Allgemeinen: Bildschirm oder Drucker) in all seinen Eigenschaften, Besonderheiten und Fähigkeiten verkapselt und dem Programmierer eine Abstraktion mit geräteunabhängigen Zugriffsmechanismen Repräsentationen und Operationen präsentiert.

.NET verwendet GDI+, eine nicht nur vom Funktionsumfang her erweiterte, sondern inzwischen auch konsequent objektorientierte Implementierung des GDI.

In diesem Kapitel geht um die Grafikausgabe und Zeichenoperationen auf der Basis der .NET-Klassenbibliothek. Neben dem obligatorischen Überblick über die verschiedenen Datenstrukturen und Klassen, die für die Grafikausgabe eine Rolle spielen, zeigen die folgenden rund 80 Seiten Konzepte und Programmiertechniken für alle vier Bereiche auf.

14.1 GDI+

Mit GDI+ vereinfachen sich nicht nur viele Ausgabeoperationen, die Sie vielleicht von der GDI-Programmierung mit C/C++ oder mit Delphi her kennen, es gibt auch eine Fülle neuer Features, die eine effizientere und vielfältigere Grafikausgabe in allen vier Bereichen – Text, Bild, Linien- und Füllfiguren – ermöglichen.

- Zu den neuen Features des GDI+ gehören:
- *Grafikpfade als eigenständige Objekte* – im GDI wurde ein Grafikpfad für die einzelne Grafikoperation im Grafikkontext nach dem Write-only-Prinzip zusammengestellt und beim Zeichnen »verbraucht«. GDI+ erlaubt es, Grafikpfade in Form von `GraphicsPath`-Objekten unabhängig vom Grafikkontext zu speichern, zu manipulieren (Erweiterung, Koordinatentransformation) und wiederzuverwenden.

- *Kardinale Splines* – im GDI konnten Linienzüge nur als Polylinien oder Polygone mit geraden Linienstücken und Ecken gezeichnet werden. Die Alternative war die Stückelung von Bezierkurven. GDI+ ermöglicht nun auch die geglättete Ausgabe von Linienzügen über kardinale Splines, für die ein Spannungsfaktor als Maß für die gleichmäßige Glättung über die gesamte Kurve hinweg anzugeben ist.
- *Matrix* – ein neues Objekt im Grafikkontext, das eine abstrakte Koordinatensystemtransformation in Form einer linearen Abbildung beschreibt. Lineare Abbildungen ermöglichen die beliebige *Skalierung* (Strecken/Stauchen) der Koordinatenachsen, *Translation* (Verschieben) des Ursprungs und *Rotation* der Zeichenfläche.
- *Alpha-Blending* – ein Konzept für partielle Transparenz, welches das vom GDI her gewohnte Farbmodell TrueColor (24-Bit-RGB) in einen 32-Bit-Farbraum mit 256 Transparenzstufen (von durchsichtig bis undurchsichtig) erweitert. Damit ist es möglich, Text, Bilder und Figuren so zu zeichnen, dass der Hintergrund mehr oder weniger stark durchscheint.
- *Farbverläufe als Pinseltypen* – vom GDI her sind Pinsel für Volltonfarben, Schraffuren und Texturen bekannt. Mit GDI+ stehen nun auch Pinseltypen für lineare Farbverläufe zwischen zwei parallelen Linien sowie für komplexe radiale Farbverläufe von einem Mittelpunkt zu beliebig vielen Randpunkten, wobei jedem Punkt eine andere Farbe zugeordnet werden kann.
- *Skalierbare Regionen* – im GDI wurden Regionen in gerätespezifischen Koordinaten repräsentiert und erlaubten ausschließlich die Translation (Verschiebung). GDI+ repräsentiert Regionen geräteunabhängig (in globalen Koordinaten) und ermöglicht so neben der Translation auch die Skalierung und Rotation von Regionen.
- *Direkte Unterstützung mehrerer Grafikformate* – während das GDI für Bitmaps nur das DIB-Format (Device Independent Bitmaps) direkt unterstützte, und ansonsten auf externe Konvertierungsfilter angewiesen war, bietet GDI+ in Form der Klassen `Image`, `Bitmap` und `Metafile` direkte Unterstützung für die Grafikformate BMP, JPEG, GIF, Exif, PNG, TIFF, ICON, WMF und EMF.

14.2 Das Prinzip – Zeichnen auf Anforderung

Das ereignisorientierte Programmiermodell von Windows sieht vor, dass eine Windows-Anwendung nicht einfach nur dann zeichnet, wenn es sich von der Programmlogik her gerade so ergibt, sondern auch und vor allem auf Aufforderung des Systems.

Das System schickt Aufforderungen zum Zeichnen intern als WM_PAINT-Nachrichten an jedes einzelne Formular (bzw. Fensterobjekt, das eine eigene Nachrichtenschleife betreibt), wenn zuvor verdeckte Fensterbereiche neu gezeichnet werden müssen, das Formular seinen Anzeigezustand wechselt (`WindowState`) oder sonstige Änderungen des Systemzustands (Anzeige-Eigenschaften) dies erfordern. Das `Form`-Objekt übersetzt diese Nachrichten seinerseits in Aufrufe der virtuellen `OnPaint()`-Methode, die wiederum je Aufruf ein `Paint`-Ereignis signalisiert – an alle, die »es interessiert« und eine Behandlungsroutine dafür registrieren.

Unabhängig vom `OnPaint()`-Aufruf geht das `Form`-Objekt zusätzlich die `Controls`-Auflistung durch und signalisiert an jedes darin enthaltene Steuerelement gleichfalls ein `Paint`-Ereignis. Handelt es sich bei einem Steuerelement um einen Container mit eigener `Controls`-Auflistung, verfährt dieses mit des Steuerelementen in seiner `Controls`-Auflistung genauso usw.

14.2.1 OnPaint() vs. Paint-Behandlung

Das allgemeine Verfahren, um im Clientbereich eines Formulars Zeichenoperationen auszuführen, besteht somit darin, dass der Besitzer eines Formularobjekts oder auch die Formularklasse selbst eine `Paint`-Behandlung registriert, die auf das `Paint`-Ereignis hin den Clientbereich des Formulars neu zeichnet. Falls die Laufzeit vernachlässigbar ist, wird dies normalerweise der gesamte Clientbereich sein, wobei alle Zeichenoperationen, die über den Clientbereich oder den Rand des sichtbaren Fensterausschnitts hinausführen, vom System automatisch beschnitten werden, ohne dass anwendungsseitig besondere Vorkehrungen nötig wären:

```
public void Form1_Paint(object sender, PaintEventArgs e)
{ ... }
```

Für aufwändigere Ausgaben empfiehlt es sich hingegen, die Eigenschaft `ClipRectangle` des `PaintEventArgs`-Ereignisobjekts auszuwerten und nur den Bereich zu zeichnen, den das System tatsächlich anfordert. Die Logik dafür ist für gewöhnlich mit einigem Aufwand verbunden, der – wie schon angedeutet – nur lohnt, wenn sich damit spürbar Laufzeit einsparen lässt.

Bei der Implementierung einer Formular- oder Steuerelementklasse besteht grundsätzlich die Möglichkeit, anstatt (oder auch ergänzend zu) einer `Paint`-Behandlung eine `override`-Variante der `OnPaint()`-Methode zu definieren.

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
  ...
}
```

Näheres dazu finden Sie im Abschnitt »OnXxx() überschreiben vs. Behandlungsmethoden« auf Seite 451. An dieser Stelle sei nur soviel gesagt, dass die Überschreibung grundsätzlich die Basisklassenvariante aufrufen oder zumindest ihrerseits ein (mit `new` überschriebenes) `Paint`-Ereignis signalisieren sollte, damit auch die gegebenenfalls besitzerseitig registrierten Behandlungsroutinen zum Aufruf kommen.

PaintEventArgs-Klasse

`Paint`-Behandlungsmethoden und `OnPaint()`-Methoden erhalten beim Aufruf ein `PaintEventArgs`-Ereignisobjekt, das zwei für den Zeichenvorgang unverzichtbare Informationen in Form schreibgeschützter Eigenschaften bereitstellt: `ClipRectangle` und `Graphics`. Erstere beschreibt den zu zeichnenden Bereich in Clientkoordinaten (in Bildpunkten mit Ursprung in der linken oberen Ecke des Bereichs), letztere ist der Grafikkontext.

Auch wenn grundsätzlich die Möglichkeit besteht, ein solches Objekt selbst zu instanziieren, werden Sie dazu kaum Anlass haben, da es eigentlich immer vom System gestellt wird. Ein denkbarer Fall wäre der Aufruf einer `Paint`-Behandlungsmethode (nicht unbedingt der klasseneigenen) von einem Punkt außerhalb der Paint-Behandlung:

```
// Im Quelltext von Form1
PaintEventArgs pEA
   = new PaintEventArgs(CreateGraphics(), this.ClientRectangle);
Form1_Paint(this, pEA);
```

Wie der folgende Abschnitt zeigt, gibt es dafür aber bessere Lösungen.

Refresh() und Invalidate()

Der Impuls für die Auffrischung der Anzeige kann allerdings auch von »innen« kommen, wann immer die Programmlogik dies erfordert. Von `Control` abstammende Objekte (also Formulare und Steuerelemente) erreichen dies entweder durch einen Aufruf der Methode `Invalidate()` oder der Methode `Refresh()`. Auch wenn beide Methoden letztlich denselben Effekt haben, dürfen sie nicht über einen Kamm geschoren werden.

Die sechsfach überladene `Invalidate()`-Methode erklärt wahlweise den gesamten Clientbereich oder einen als Rechteck oder Region beschriebenen Teilbereich für ungültig. Zusätzlich besteht die Möglichkeit anzugeben, ob nur der eigene Clientbereich oder auch der aller untergeordneten Steuerelemente neu gezeichnet werden soll. In jedem Fall löst `Invalidate()` nur mittelbar ein `Paint`-Ereignis für das jeweilige `Control`-Objekt aus, denn der Aufruf nimmt den Weg über die Nachrichtenschlange des Formulars, so dass eine Reaktion darauf frühestens nach Beendigung der aktuellen Ereignisbehandlungsmethode zu erwarten ist. (Auch wenn es falsch wäre, in diesem Zusammenhang den Begriff »asynchron« zu benutzen, es sind ja keine

Threads im Spiel, so hat die Methode zumindest den Charme eines asynchronen Aufrufs – `Invalidate()` gibt die Kontrolle sofort zurück, ohne dass der Aufrufer auf das Ergebnis der Grafikausgabe warten muss.) Der Vorteil liegt darin, dass das System so mehrere `Invalidate()`-Aufrufe zu einem einzigen `OnPaint()`-Zyklus zusammenfassen kann und die Gefahr unerwünschter rekursiver Aufrufe innerhalb des Grafiksubsystems geringer ist.

Im Gegensatz dazu nimmt die parameterlose `Refresh()`-Methode den direkten Weg. Sie erzwingt den sofortigen Aufruf der `OnPaint()`-Routine zuerst des Formulars (bzw. Steuerelements) und dann aller Steuerelemente in der `Controls`-Auflistung – grundsätzlich für den gesamten Clientbereich. Damit entspricht sie weitgehend dem Aufruf:

```
OnPaint(PaintEventArgs(CreateGraphics(), this.ClientRectangle));
```



Bei einer regulären `Paint`*-Behandlung nimmt das System ein weitere Optimierung vor: Zeichenoperationen werden nicht an den Grenzen des Fensters, sondern an den Grenzen des per (explizitem und implizitem)* `Invalidate()` *festgelegten Bereichs abgeschnitten, was unter anderem unnötige Datentransporte vom Haupt- in den Bildspeicher erspart. Bei Operationen mit einem außerhalb einer* `Paint`*-Behandlung angelegten Grafikkontext übernimmt das System dagegen ausschließlich die Beschneidung an den jeweiligen Fenstergrenzen.*

14.3 Für das Zeichnen benötigte Strukturen und Klassen

Zeichnen unter .NET ist eine kompakte Angelegenheit, die auf die üblichen Mittel der Zunft zurückgreift, im übertragenen Sinne natürlich: Man nehme Stifte und Pinsel in verschiedenen Farben und Ausführungen, lege sich Zirkel, Lineale und Schablonen zurecht und bereite den Malgrund vor ...

14.3.1 Koordinatensysteme

Der für die Ausgabe auf ein Gerät zuständige Grafikkontext wird durch ein `Graphics`-Objekt repräsentiert. Zur Ausstattung dieses Objekts gehören unter anderem ein Satz von Methoden zum Aufruf der vom GDI+ unterstützten Zeichenoperationen sowie verschiedene Eigenschaften, die eine höchst flexible Parametrisierung der von diesen Operationen verarbeiteten Koordinaten gestatten.

Die `Graphics`-Eigenschaft des `PaintEventArgs`-Ereignisobjekts liefert bei jedem Aufruf der `Paint`-Behandlung einen niegelnagelneuen Grafikkontext:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  Graphics g = e.Graphics;
  ...
```

Weltkoordinaten

Für die Ausgabe auf ein Zielgerät (Fenster oder Drucker) unterscheidet GDI+ drei Koordinatensysteme und definiert zwei Transformationen, die zwischen diesen Koordinatensystemen vermitteln. Das Koordinatensystem, in dem Sie die Koordinaten für Zeichenoperationen spezifizieren, ist das *globale Koordinatensystem*, auch *Weltkoordinatensystem* genannt. Bei der Ausgabe werden die globalen Koordinaten unter Anwendung der *globalen Transformation* erst in das *Seitenkoordinatensystem* und dann unter Anwendung der *Seitentransformation* in das *Gerätekoordinatensystem* umgerechnet. Die folgenden Abschnitte erläutern diese Begriffe.

Globale Transformation

Bei der GLOBALEN TRANSFORMATION handelt es sich um eine allgemeine lineare Abbildung, die eine beliebige Translation, Skalierung und Rotation des globalen Koordinatensystems gegenüber dem Seitenkoordinatensystem ermöglicht. Zur Repräsentation dieser Abbildung wird ein `Matrix`-Objekt verwendet, das seinerseits ein Array mit 6 `float`-Werten benutzt und dieses über die Eigenschaft `Elements` offen legt. Die ersten vier Werte im Array bilden eine 2x2-Matrix für die Skalierung und Rotation, die letzen beiden beschreiben die Translation.

Um mit Transformationen unbeschwert arbeiten zu können, sollten Sie die folgende `using`-Direktive ergänzen:

```
using System.Drawing.Drawing2D;
```

Die Klasse `Matrix` stellt vier Konstruktoren für die Instanziierung zur Verfügung. Vorgabewert für die Eigenschaft ist die IDENTISCHE ABBILDUNG, die auch der parameterlose Konstruktor liefert. Diese Abbildung zeichnet sich dadurch aus, dass sie jeden Punkt im zweidimensionalen Koordinatensystem auf sich selbst abbildet. Sie hat die Repräsentation `{1,0,0,1,0,0}`.

```
Matrix m1 = new Matrix();               // {1, 0, 0, 1, 0, 0}

Matrix m2 = new Matrix(1,0,0,1,0,0);    // {1, 0, 0, 1, 0, 0}
```

Die drei Grundoperationen

Eine Ursprungsverschiebung des globalen Koordinatensystem um 50 Einheiten nach rechts und 30 Einheiten nach unten hat die Repräsentation `{1,0,0,1,50,30}`:

```
m1.Translate(50, 30);             // {1, 0, 0, 1, 50, 30}
```

Aufgrund der vom `Graphics`-Objekt vorgenommenen Verkapselung zeigt der Aufruf

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  e.Graphics.Transform.Translate(50,30);
```

überraschenderweise keine Wirkung. Stattdessen müssen Sie die `Translate-Transform()`-Methode des `Graphics`-Objekts verwenden:

```
e.Graphics.TranslateTransform(50,30);
```

Funktionieren tut jedoch:

```
Matrix m1 = e.Graphics.Transform;
m1.Transform(50,30);
e.Graphics.Transform = m1;
```

Eine Streckung in horizontaler Richtung um den Faktor 2 und in vertikaler Richtung um den Faktor 3 hat die Repräsentation `{2,0,0,3,0,0}`. Auch dafür gibt es eigene Matrix-Methoden namens `Scale()` und `Graphics`-Methode namens `ScaleTransform()`:

```
e.Graphics.ScaleTransform(2,3);
```

Eine Drehung um 60° im Urzeigersinn hat die Repräsentation `{0.5,0.866,-0.866,0.5,0,0}` bzw. {cos(-_/3), -sin(-_/3), sin(-_/3), cos(-_/3),0,0} mit _/3 = 60°. (Das Bogenmaß wird hier gegen den Uhrzeigersinn aufgetragen, lassen Sie sich davon nicht verwirren.). Die `Matrix`-Methode dafür heißt `Rotate()`, die `Graphics`-Methode `RotateTransform()`. Beide Methoden erwarten einen Wert in Grad für eine Drehung *im* Uhrzeigersinn.

```
e.Graphics.RotateTransform(60);     // {0.5, 0.866, -0.866, 0.5, 0, 0}
```

Reihenfolge der Transformationsoperationen

Wenn Sie sich schon einmal mit der Matrixrechnung bzw. mit Linearen Abbildungen beschäftigt haben, werden Sie wahrscheinlich wissen, dass Sie die drei Operation zwar in beliebiger Reihenfolge ausführen können, das Ergebnis jeder Reihenfolge aber ein anderes ist. Sie sollten sich daher für eine feste Reihenfolge entscheiden. Am intuitivsten ist es, wenn Sie zuerst den Ursprung verschieben, dann rotieren und schließlich skalieren.

```
Matrix m1 = new Matrix();           // {1, 0, 0, 1, 0, 0}
m1.Translate(50, 30);               // {1, 0, 0, 1, 50, 30}
m1.Rotate(60);                      // {0.5, 0.866, -0.866, 0.5,50, 30}
m1.Scale(2,3);                      // {1, 1.732, -2.598, 1.5, 50, 30
```

Um bestehenden Code nicht umschreiben zu müssen, bzw. optimieren zu können, falls die Reihenfolge einmal anders herum herauskommt, existieren für alle sechs Methoden Überladungen mit zwei Parametern, bei denen der zweite Parameter die Reihenfolge bestimmt. Hat er den Vorgabewert `Matrix-Order.Prepend`, wird die Operation wie beim parameterlosen Aufruf ausge-

führt. Hat er dagegen den Wert `MatrixOrder.Append` wird die Operation so angewendet, als sei sie vor der bestehenden Transformation angewendet worden.

Die folgenden Schreibweisen würden somit dasselbe Ergebnis wie zuvor liefern:

```
// ***********************
Matrix m1 = new Matrix();
m1.Translate(50, 30 ,MatrixOrder.Prepend);
m1.Rotate(60 ,MatrixOrder.Prepend);
m1.Scale(2,3 ,MatrixOrder.Prepend);  // {1, 1.732, -2.598, 1.5, 50, 30

// ***********************
Matrix m1 = new Matrix();
m1.Rotate(60);
m1.Scale(2,3);
m1.Translate(50, 30 ,MatrixOrder.Append); // {1, 1.732, -2.598, 1.5, 50, 30

// ***********************
Matrix m1 = new Matrix();
m1.Scale(2,3, MatrixOrder.Append);
m1.Rotate(60, MatrixOrder.Append);
m1.Translate(50, 30 ,MatrixOrder.Append); // {1, 1.732, -2.598, 1.5, 50, 30
```

:-) TIPP

Als Faustregel gilt, dass Operationen ohne `MatrixOrder`*-Zusatz in der notierten Reihenfolge ausgeführt werden, wenn Sie die globale Transformation mit Blick auf den Zeichenvorgang manipulieren.*

Um also ein zu zeichnendes Rechteck per globaler Transformation erst um 30° um den Ursprung zu drehen und das Ergebnis danach um 100 Bildpunkte nach rechts und 50 Bildpunkte nach unten zu verschieben, schreiben Sie

```
e.Graphics.TranslateTransform(100, 50, MatrixOrder.Append);

e.Graphics.RotateTransform(30, MatrixOrder.Append);

e.Graphics.DrawRectangle(Pens.Black, 0,0, 30, 30);
```

bzw.

```
Matrix m = new Matrix();
m.Translate(100, 50);
m.Rotate(30);
e.Graphics.Transform = m;
e.Graphics.DrawRectangle(Pens.Black, 0,0, 30, 30);
```

Wenn Sie hingegen die Repräsentation eines Objekts mit eigener Transformationsmatrix, beispielsweise eines Grafikpfades des Typs `GraphicsPath`*, ändern, ergibt sich die notierte Reihenfolge, wenn Sie* `MatrixOrder.Append` *angeben.*

Weitere mit der Transformation zusammenhängende Operationen

Beide Objekte unterstützen noch verschiedene weitere Methoden mit komplexeren Parametersätzen, wie in Tabelle 14.1 für das `Graphics`-Objekt aufgelistet.

Tabelle 14.1: Methoden des Grafikkontexts für Transformationen

`Graphics`-Methode	Bedeutung
`void MultiplyTransform()`	Multipliziert die bestehende Transformation mit der angegebenen Transformation, was der Ausführung beider Transformationen hintereinander entspricht. In der Variante mit zwei Parametern lässt sich die Reihenfolge angeben.
`void ResetTransform()`	Setzt die Matrix auf die identische Abbildung {1,0,0,1,0,0} zurück.
`void RotateTransform()`	Dreht das globale Koordinatensystem gegenüber dem Seitenkoordinatensystem.
`void ScaleTransform()`	Skaliert das globale Koordinatensystem gegenüber dem Seitenkoordinatensystem.
`void TransformPoints()`	Transformiert ein `Point`-Array von einem der drei Koordinatensysteme in ein anderes. Ausgangs- und Zielkoordinatensystem werden über ein Element der `CoordinateSpace`-Aufzählung beschrieben. Zur Auswahl stehen: `Device`, `Page` und `World`.
`void TranslateClip()`	Verschiebt den Clipping-Bereich und ändert damit die Eigenschaft `ClipRectangle`.
`void TranslateTransform()`	Verschiebt den Ursprung des globalen Koordinatensystems gegenüber dem des Seitenkoordinatensystems.

Seitenkoordinatensystem

Das SEITENKOORDINATENSYSTEM hat seinen Ursprung unverrückbar in der linken oberen Ecke des Clientbereichs bzw. Druckdokuments und gibt die physische Maßeinheit für die beiden Koordinatenachsen vor. Der Name der entsprechenden Eigenschaft lautet `PageUnit`. Zusammen mit der Eigenschaft `PageScale`, die einen wahlfreien Skalierungsfaktor für beide Koordinatenachsen einbringt, definiert `PageUnit` eine Skalierungsabbildung vom Seitenkoordinatensystem ins Gerätekoordinatensystem, die als *Seitenabbildung* oder *Seitentransformation* bezeichnet wird.

Die folgende Betrachtung geht davon aus, dass `PageScale` nicht verändert wurde (Vorgabewert ist 1). Der Vorgabewert für die Eigenschaft `PageUnit` ist `Pixel`, ein Element aus der `GraphicsUnit`-Aufzählung. Gilt diese Einheit, sind Seitenkoordinatensystem und Gerätekoordinatensystem identisch. Tabelle

14.2 schlüsselt auf, welche Maßeinheiten Sie für das Seitenkoordinatensystem setzen können.

Tabelle 14.2: Elemente der `GraphicsUnit`-Aufzählung

Element	Bedeutung
`Display`	Einheit hat die Länge 1/75 Zoll oder 0,339 mm (Bildschirmpunkt)
`Document`	Einheit hat die Länge 1/300 Zoll oder 0,085 mm
`Inch`	Einheit hat die Länge 1 Zoll oder 25,4 mm
`Millimeter`	Einheit hat die Länge 1 mm
`Pixel`	Vorgabewert; Einheit hat die Länge 1 Bildpunkt und richtet sich nach der physischen Auflösung des Gerätes; (Die Umrechnung in ein physisches Maß kann über die `Graphics`-Eigenschaften `DpiX` und `DpiY` erfolgen.)
`Point`	Einheit hat die Länge 1/72 Zoll oder 0,352 mm (typografischer Punkt)
`World`	Als Einheit wird die für die globale Transformation geltende Einheit verwendet, also der Wert der `GraphicsUnit`-Eigenschaft des Grafikkontexts (für Seitentransformation nicht unterstützt).



Die Eigenschaften `PageUnit` *und* `PageScale` *definieren zusammen das Maß 1 für das globale Koordinatensystem und somit auch den Vorgabewert für die Stiftbreite von Pen-Objekten.*

Gerätekoordinatensystem

Das Gerätekoordinatensystem ist prinzipiell nicht frei wählbar, da es durch die physischen Gegebenheiten des Ausgabegerätes bestimmt wird. Der Bezug zum Seitenkoordinatensystem ergibt sich über die `Graphics`-Eigenschaften `DpiX` und `DpiY`, deren Werte ausdrücken, wie viele Bildpunkte pro Zoll (25,4 mm) das Gerät in der jeweiligen Richtung darstellen kann. Für einen neuen Grafikkontext sind alle Vorgabewerte so gesetzt, dass alle drei Koordinatensysteme übereinstimmen und der physische Bildpunkt des Ausgabegeräts die Maßeinheiten für beide Koordinatenachsen definiert.

14.3.2 Positionen, Abmessungen, Rechtecke

Zur Definition von Punkten, Rechtecken und Abmessungen finden sich in der .NET-Klassenbibliothek die Strukturen `Point`, `PointF`, `Size`, `SizeF`, `Rectangle`, `RectangleF`. Die Datentypen ohne Suffix »F« sind auf der Basis von `int`-Werten definiert, solche mit Suffix »F« auf der Basis von `float`-Werten. Entsprechend dual sind auch die Grafikoperationen des Grafikkontextes `Graphics` angelegt.

Da die int-Arithmetik etwas schneller als die float-Arithmetik ist, sollten die int-Versionen der Datentypen vornehmlich dann eingesetzt werden, wenn die PageUnit-Eigenschaft des Grafikkontextes auf GraphicsUnit.Pixel gesetzt ist. Ist hingegen eine Maßeinheit wie GraphicsUnit.Inch gesetzt, wären bei den int-Versionen erhebliche Rundungsverluste zu erwarten, weshalb in diesem Fall kein Weg an der float-Arithmetik vorbeiführt.

Wie zu erwarten, existiert für alle drei Ganzzahldatentypen eine implizierte Typumwandlung in die jeweiligen Gleitkomma-Gegenstücke. Für Umwandlungen in die umgekehrter Richtung stehen drei statische Methoden zur Auswahl: Xxx.Ceiling() rundet alle Datenfelder der Repräsentation auf die nächste Ganzzahl auf, Xxx.Truncate() schneidet die Nachkommastellen ab und Xxx.Round() rundet sie nach Kaufmannsart zur nächstliegenden Ganzzahl auf oder ab.

Da diese Datenstrukturen sowohl für das Zeichnen als auch für die Fensterberechung von Formularen und Steuerelementen absolut zentral sind, sollten Sie sich ausführlich mit ihrer Ausstattung und ihren Möglichkeiten auseinandersetzen. Sie werden im Folgenden einzeln bzw. paarweise vorgestellt.

Punkte – Point, PointF

Punkte erlauben die Addition und Subtraktion von Abmessungen (Size- bzw. SizeF-Strukturen). Weiterhin lassen sie sich auf Gleichheit testen und in Size-Strukturen umwandeln. Zur statischen Ausstattung der Strukturen Point und PointF gehören demnach die Operatoren +, -, ==, != sowie explizite Typumwandlungsoperatoren von Point nach Size und von PointF nach SizeF.

Tabelle 14.3 zeigt die Ausstattung auf Instanzebene.

Tabelle 14.3: Ausstattung von Point- und PointF-Strukturen

Eigenschaft, Methode	Bedeutung
`Point Empty` `PointF Empty`	Konstante für den Punkt (0,0)
`bool IsEmpty {get;}`	`true`, wenn X und Y den Wert 0 haben

`Point(int)`
`Point(Size s)`
`Point(int x, int y)`
`PointF(float x, float y)`
	Die Bedeutung der zweiparametrigen Konstruktoren ist klar. Für den Konstruktor der int-Version existieren zwei weitere einparametrige Überladungen: Die eine Variante erwartet einen int-Wert und initialisiert die X-Eigenschaft mit den unteren 16 Bit und die Y-Eigenschaft mit den oberen 16 Bit dieses Wertes. Die andere Variante erwartet einen Size-Wert, dessen Width-Eigenschaft sie als x-Wert und dessen Height-Eigenschaft sie als y-Wert interpretiert.

Tabelle 14.3: Ausstattung von `Point`- und `PointF`-Strukturen (Forts.)

Eigenschaft, Methode	Bedeutung
`int X {get; set;}` `float X {get; set;}`	x-Koordinate
`int Y {get; set;}` `float Y {get; set;}`	y-Koordinate
`void Offset()`	Verschiebt den Punkt um die angegebenen Distanzen für die x- und die y-Richtung (komponentenweise Addition).

Abmessungen – Size, SizeF

Mit Hilfe der Strukturen `Size` und `SizeF` lassen sich die Abmessungen von Rechtecken beschreiben, deren Ränder parallel zur x- und y-Achse des globalen Koordinatensystems gelegen sind.

Interessanterweise unterscheiden sich die Datentypen `Size` und `SizeF` außer im Namen der beiden Eigenschaften `Height` und `Width` eigentlich nur formal von `Point` bzw. `PointF` – von unwichtigen Details wie einer etwas anderen Konstruktor-Ausstattung einmal abgesehen. Dass es diese Datentypen überhaupt gibt, hat folgenden Hintergrund: Bei der Vektorrechnung wird in der Mathematik ein konzeptioneller Unterschied zwischen *affinen Räumen* (Punkträumen) und *Vektorräumen* gemacht. Im einen Fall rechnet man mit Punkten, die auf ein festes Koordinatensystem (mit Ursprung) bezogen sind, und im anderen Fall mit Verschiebungen (allgemeiner: Abbildungen) solcher Punkte.

Obwohl die Operationen dieser Datentypen aus rechnerischer Sicht exakt gleich definiert sind, macht es natürlich einen Unterschied, ob man ein Rechteck über zwei Punkte definiert oder über einen Aufhängepunkt und eine Verschiebung dieses Punkts. Im einen Fall kann das Rechteck allein dadurch verschoben werden, dass der Aufhängepunkt verschoben wird, im anderen Fall müssen beide Punkte verschoben werden.

Kurzum, ein `Size`- bzw. `SizeF`-Wert bezieht sich immer auf einen `Point`- bzw. `PointF`-Wert – und die Unterscheidung zwischen diesen beiden Datentypen ist der Stoff, aus dem objekteigene Koordinatensysteme gemacht sind. (An einem einzigen Punkt aufgehängt kann ein Zeichnungsobjekt nach Belieben für sich allein gedreht, gestreckt und verschoben werden.)

Zur statischen Ausstattung der Strukturen `Size` und `SizeF` gehören die Operatoren `+`, `-`, `==`, `!=` sowie explizite Typumwandlungsoperatoren von `SizeF` nach `PointF` und von `SizeF` nach `Size`.

Tabelle 14.4 zeigt die Ausstattung der Strukturen auf Instanzebene.

Eigenschaft, Methode	Bedeutung
`Point Empty`	Konstante für die Abmessungen (0,0)
`bool IsEmpty {get;}`	`true`, wenn X und Y den Wert 0 haben

`Size(Point p)`
`Size(int w, int h)`
`SizeF(PointF p)`
`SizeF(SizeF s)`
`SizeF(float w, float h)`
	Die Konstruktoren weisen keine Besonderheiten auf.

`int Width {get; set;}` `float Width {get; set;}`	Abmessung in x-Richtung
`int Height {get; set;}` `float Height {get; set;}`	Abmessung in y-Richtung

Tabelle 14.4: Ausstattung von `Size`- und `SizeF`-Strukturen

Rechtecke – Rectangle, RectangleF

Die Strukturen `Rectangle` und `RectangleF` beschreiben Rechtecke, deren Ränder parallel zur x- und y-Achse des globalen Koordinatensystems gelegen sind.

Je nach Sichtweise setzt sich eine `Rectangle`-Struktur aus einem `Point`-Wert und einem `Size`-Wert oder aus zwei `Point`-Werten zusammen – analog die `RectangleF`-Struktur. (Die zweite Sichtweise ist auf den reinen Lesezugriff beschränkt. Es existiert dafür weder ein Konstruktor dafür noch lassen sich die entsprechenden Eigenschaften direkt setzen.)

Neben den bereits eingangs erwähnten Typumwandlungsmethoden umfasst die statische Ausstattung von `Rectangle` bzw. `RectangleF` die Methoden `FromLTRB()`, `Union()`, `Inflate()` und `Intersect()`. Die Methode `FromLTRB()` generiert mangels eines entsprechenden Konstruktors ein Rechteck aus zwei `Point` bzw. `PointF`-Werten und `Union()` die Vereinigung zweier Rechtecke (definiert als kleinstes Rechteck, das beide Rechtecke umschließt). Die anderen beiden Methoden existieren auch als Instanzmethoden und sind in Tabelle 14.6 erläutert.

Tabelle 14.5 gibt einen Überblick über die Eigenschaften, Tabelle 14.6 über die Methoden der beiden Datentypen.

Eigenschaft	Bedeutung
`int Bottom {get;}` `float Bottom {get;}`	y-Koordinate des unteren Rechteckrands
`int Height {get; set;}` `float Height {get; set;}`	Höhe des Rechtecks

Tabelle 14.5: Eigenschaften von `Rectangle`- und `RectangleF`-Strukturen

Tabelle 14.5: Eigenschaften von `Rectangle`- und `RectangleF`-Strukturen (Forts.)

Eigenschaft	Bedeutung
`bool IsEmpty {get;}`	`true`, wenn `Width` und `Height` den Wert 0 haben.
`int Left {get;}` `float Left {get;}`	x-Koordinate des linken Rechteckrands
`Point Location {get; set;}` `PointF Location {get; set;}`	Aufhängepunkt des Rechtecks. Dies muss einer der vier Eckpunkte sein.
`int Right {get;}` `float Right {get;}`	x-Koordinate des rechten Rechteckrands
`Size Size {get; set;}` `SizeF Size {get; set;}`	Abmessungen des Rechtecks
`int Top {get;}` `float Top {get;}`	y-Koordinate des oberen Rechteckrands
`int Width {get; set;}` `float Width {get; set;}`	Breite des Rechtecks
`int X {get; set;}` `float X {get; set;}`	x-Koordinate des Aufhängepunkts
`int Y {get; set;}` `float Y {get; set;}`	y-Koordinate des Aufhängepunkts

Tabelle 14.6: Methoden von `Rectangle` und `RectangleF`-Strukturen

Methode	Beschreibung

`bool Contains(Point p)`
`bool Contains(int x, int y)`
`bool Contains(Rectangle r)`
`bool Contains(PointF p)`
`bool Contains(float x, float y)`
`bool Contains(RectangleF r)`
	`true`, wenn der angegebene Punkt oder das angegebene Rechteck im Rechteck enthalten sind (Randpunkte zählen auch).

`void Inflate(Size s)`
`void Inflate(int w int h)`
`void Inflate(SizeF s)`
`void Inflate(float w, float h)`
	Vergrößert die Abmessungen des Rechtecks um den angegebenen Wert. Als Alternative zu diesen Instanzmethoden stellt jeder der beiden Datentypen auch eine statische Ausführung mit gleichem Bezeichner und gleicher Funktion bereit. Sie erwartet in einem zusätzlichen Parameter das Ausgangsrechteck und gibt das vergrößerte Rechteck als Funktionswert zurück.

Methode	Beschreibung
`void Intersect(Rectangle r)` `void Intersect(RectangleF r)`	Schneidet das Rechteck mit dem angegebenen Rechteck und reduziert es auf die Schnittmenge. Ist die Schnittmenge leer, lautet das Ergebnis NULL – ein Rechteck an der Position (0,0) mit den Abmessungen (0,0). Als Alternative zu diesen Instanzmethoden stellen beide Datentypen auch eine statische Ausführung mit gleichem Bezeichner und gleicher Funktion bereit. Sie erwartet in einem zusätzlichen Parameter das Ausgangsrechteck und gibt das vergrößerte Rechteck als Funktionswert zurück.
`bool IntersectsWidth(Rectangle r)` `bool IntersectsWidth(RectangleF r)`	`true`, wenn der Schnitt mit dem angegebenen Rechteck nicht NULL (leer) ist.

`void Offset(int x, int y)`
`void Offset(float x, float y)`
`void Offset(Point p)`
`void Offset(PointF p)`
	Addiert den angegebenen Punkt zum Aufhängepunkt der Rechtecks. Dadurch ändert das Rechteck seine Position relativ.

Tabelle 14.6: Methoden von `Rectangle` und `RectangleF`-Strukturen (Forts.)

Komplexe Bereiche – die Klasse Region

Die nächste Stufe der Komplexität auf Basis der bereits vorgestellten Datenstrukturen ist die `Region`-Klasse. Ihre Objekte lassen sich für die Beschreibung beliebiger – nicht notwendigerweise zusammenhängender – Bereiche verwenden, die sich ihrerseits als *Füllbereiche* für `FillXxx`-Grafikoperationen angeben sowie als *Clippingbereiche* für die Begrenzung der Grafikausgabe setzen lassen.

Von der Konstruktion her gesehen ist ein `Region`-Objekt nichts weiter als ein Container für drei unterschiedliche Arten von Werten: `Rectangle`- bzw. `RectangleF`-Strukturen und `GraphicPath`-Objekte. Wie zu erwarten, existieren neben dem parameterlosen Konstruktor Überladungen, die eine Initialisierung jeweils einem dieser drei Datentypen ermöglichen. Um weitere Bereiche hinzuzufügen, steht eine `Union()`-Methode zur Verfügung, das Erzeugen von Schnittbereichen übernimmt die Methode `Intersect()` – für jeden der drei Datentypen und natürlich auch für `Region`-Objekte in jeweils eigenen Überladungen.

RegionData

Da eine Repräsentation als Ansammlung von `Rectangle`-, `RectangleF`- und `GraphicPath`-Werten aus formaler Sicht weder eindeutig noch minimal wäre, verwendet die `Region`-Implementierung ihrerseits ein `RegionData`-Objekt als Datenbasis, das sich nicht nur über die Methode `GetRegionData()` abfragen,

sondern auch bei der Konstruktion eines neuen `Region`-Objekts verwenden lässt. Das `RegionData`-Objekt wiederum legt eine `byte[]`-Eigenschaft namens `Data` offen und bereitet so den Weg zur Serialisierung und Deserialisierung von `Region`-Objekten.

Des weiteren besteht die Möglichkeit, von einem `Region`-Objekt durch Aufruf von `GetRegionScan()` ein `RectangleF`-Array bereitstellen zu lassen, dessen Elemente eine hinreichend gute Überdeckung des gesamten Bereichs darstellen.

Grafikpfade – GraphicsPath

Eine andere für `Region`-Objekte wichtige Klasse ist `GraphicsPath`. Die Möglichkeiten, die diese Klasse für die Bereichsdefinition bietet, sind beachtlich, und es wird Sie sicher nicht überraschen, in ihr auch die Brücke zur Schriftdefinition zu finden. Dass sich die Objekte dieser Klasse in `Region`-Objekte einfügen lassen, ist demnach auch nicht ihr Haupteinsatzbereich, sondern eröffnet nur zusätzliche Möglichkeiten für die Maskierung im Zusammenhang mit der Grafikausgabe.

Die zur Ausstattung eines jeden Grafikkontextes gehörigen Grafikoperationen `DrawPath()` und `FillPath()` sind speziell darauf ausgelegt, Grafikpfade als Umrissfiguren oder Füllfiguren bei freier Stift- und Pinselwahl zu zeichnen. Erwartungsgemäß verwendet jedes `GraphicsPath`-Objekt ein Koordinatensystem mit abstrakten Einheiten, das es erlaubt, die Bestandteile des Grafikpfads zu beschreiben und gegeneinander zu positionieren. Dieses Koordinatensystem kann über die `Transform()`-Methode des Objekts nach Belieben verbogen (das heißt: verschoben, gedreht und skaliert) werden, wobei die Anwendung der Methode *inkrementell* ist – das heißt: sich auf alle zum Zeitpunkt des Aufrufs bereits definierten Bestandteile des Pfads auswirkt. So ist es beispielsweise möglich, in einem Grafikpfad einen Schriftzug zu einem Kreis zu biegen, indem die Zeichen einzeln jeweils an der *gleichen* Position eingefügt werden und nach jedem Einfügen eine Drehung um einen Winkel erfolgt, der auf die Breite des Zeichens und den Radius abgestimmt ist. Für die Ausgabe gilt dann die in der `PageUnit`-Eigenschaft festgelegte Maßeinheit des Grafikkontextes.

Ein Blick auf die Methodenliste enthüllt, dass der Werkzeugkasten eines `GraphicsPath`-Objekts mit allem bestückt ist, was im Bereich der Grafikprogrammierung Rang und Namen hat. Dementsprechend ist die Definition komplexer Grafikpfade eher eine Wissenschaft für sich – und erfordert auf jeden Fall eine Auseinandersetzung mit Bezierkurven und kardinalen Splines. Die folgende Darstellung versteht sich daher als Einblick in Anwendungsfelder der Klasse – und keinesfalls als erschöpfende Charakterisierung des Möglichkeitspotenzials.

Für eigene Experimente mit Grafikpfaden bietet sich der folgende Coderahmen an:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  GraphicsPath gp = new GraphicsPath();
  FillGraphicsPath(gp);
  e.Graphics.DrawPath(new Pen(Color.Black, 2.5f), gp);
}

private void FillGraphicsPath(GraphicsPath gp)
{
  // Definition des Grafikpfads
}
```

Figuren

Ein Grafikpfad kann eine beliebige Anzahl von Figuren umfassen, wobei eine Figur als Abfolge von zusammenhängenden Kurvenstücken definiert ist. Stimmt der Endpunkt eines Kurvenstücks mit dem Startpunkt des nächsten Kurvenstücks nicht überein, schließt das Objekt die Lücke durch eine Linie. Der folgende Code definiert eine Kurve, die aus drei Linien besteht (Abbildung 14.1, links):

```
gp.AddLine(10, 10, 200, 20);
// gp.AddLine(200, 20, 200, 200);   // diese Linie ist überflüssig
gp.AddLine(200, 200, 10, 150);
```

Um die Figur zu einem Viereck zu schließen, ergänzen Sie einen `CloseFigure()`-Aufruf (Abbildung 14.1, Mitte):

```
gp.AddLine(10, 10, 200, 20);
gp.AddLine(200, 200, 10, 150);
gp.CloseFigure();
```

Um die beiden Linien als getrennte Figuren zu definieren, setzen Sie einen `StartFigure()`-Aufruf unmittelbar vor den Start der zweiten Figur. Wenn das `GraphicPath`-Objekt als Argument übergeben wurde, ist nicht klar, in welchem Zustand es ist. Es empfiehlt sich daher meist, die Pfaddefinition mit einem `StartFigure()`-Aufruf zu beginnen (Abbildung 14.1, rechts):

```
gp.StartFigure();
gp.AddLine(10, 10, 250, 50);
gp.StartFigure();
gp.AddLine(200, 200, 30, 150);
```

Jeder `CloseFigure()`-Aufruf schließt den mit dem letzten `StartFigure()`-Aufruf begonnen Kurvenzug und beendet zugleicht die Definition der jeweiligen Figur. Nachfolgende `AddXxx()`-Aufrufe beziehen sich auf den vorletzten

StartFigure()-Aufruf – oder, falls ein solcher nicht geschehen ist, auf die bei Konstruktion eröffnete Grundfigur des Objekts.

Für die Definition von Linienzügen steht Ihnen weiterhin die Methode AddLines() zur Verfügung, die ein Point- bzw. PointF-Array mit den aufeinander folgenden Eckpunkten erwartet.

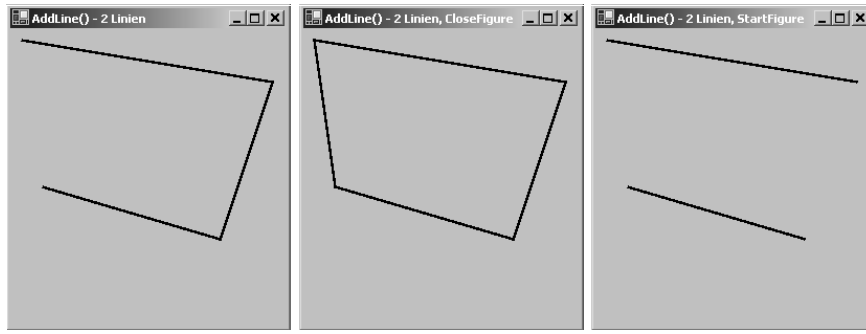


Abbildung 14.1: Variationen zweier AddLine()-Aufrufe

Kurven und abgerundete Figuren

Um Figuren mit abgerundeten Kurvenstücken zu definieren, stehen die Methoden AddEllipse(), AddArc(), AddBezier(), AddBeziers(), AddCurve() und AddClosedCurve() sowie – für Spezialfälle – AddPie() zur Verfügung.

AddEllipse() definiert eine vollständige Ellipse, AddArc() einen Ellipsenbogen über einen Startwinkel (in Grad, im Uhrzeigersinn, mit 3:00 als 0°, negative Winkel sind erlaubt) und einen Öffnungswinkel (in Grad, negative Winkel erlaubt). Die Beschreibung der Ellipse erfolgt als Rechteck (per Rectangle, RectangleF oder koordinatenweise), so dass die Hauptachsen der Ellipse immer parallel zu den Koordinatenachsen stehen. Die folgende Anweisung definiert einen Ellipsenbogen, der dem Rechteck, definiert durch den Punkt (10, 30) und die Abmessungen (250, 130), einbeschrieben ist, ab dem Winkel 30° startet und einen Öffnungswinkel von 220° aufweist (Abbildung 14.2, links):

```
gp.AddArc(10, 30, 250, 130, 30, 220);
```

Bezierkurven haben ein umfangreiches theoretisches Fundament. Wahrscheinlich werden sie Ihnen aber von Grafikprogrammen her bekannt sein. Eine Bezierkurve besteht aus einem Startpunkt, zwei Stützpunkten und einem Endpunkt. Die Paare (Startpunkt, erster Stützpunkt) und (zweiter Stützpunkt, Endpunkt) definieren zwei Tangenten, die den Austrittswinkel der Kurve aus dem Startpunkt und den Eintrittswinkel der Kurve in den Endpunkt vorgeben. Die Abstände der Stützpunkte vom Start- bzw. Endpunkt definieren den Krümmungsradius für den Austritt bzw. Eintritt. Der folgende Grafikpfad demonstriert diesen Zusammenhang recht schön (Abbildung 14.2, Mitte):

```
gp.StartFigure();
gp.AddLine(50, 250, 30, 30);
gp.StartFigure();
gp.AddBezier(30, 30, 50, 250, 200, 40, 250, 250);
gp.StartFigure();
gp.AddLine(250, 250, 200, 40);
gp.AddEllipse(25, 25, 10,10);
gp.AddEllipse(245, 245, 10,10);
```

Beachten Sie, dass die `StartFigure()`-Aufrufe notwendig sind, obwohl je ein Linienpunkt mit dem Start- und Endpunkt der Kurve übereinstimmen. (Die `AddBezier()`-Methode scheint hier unsauber implementiert zu sein.) Weiterhin zeigt der Code, dass `AddEllipse()` eine geschlossene Figur definiert und die bestehende Figur nicht erweitert.

Die Methode `AddBeziers()` erlaubt es, mehrere aneinandergrenzende Bezierkurven en bloc in eine Figur einzufügen, wobei die Bereitstellung der Parameter über ein `Point`- bzw. `PointF`-Array erfolgt. Für die erste Kurve sind 4 Punkte anzugeben, für jede weitere Kurve zusätzlich 3 Punkte. Der Endpunkt der ersten Kurve ist Startpunkt der zweiten Kurve usw. Die Kurvenkrümmungen haben keinen Bezug zueinander, so dass (nicht differenzierbare) Ecken entstehen, wenn die Stützpunkte nicht bewusst so gewählt sind, dass die Tangenten übereinstimmen. Der folgende Codeauszug definiert drei aneinandergrenzende Bezierkurven und markiert auch die Punkte (Abbildung 14.2, rechts):

```
Point[] p = new Point[] {
                          new Point(15, 15),    // Startpunkt
                          new Point(125, 25),   // 1. SP
                          new Point(250, 25),   // 2. SP
                          new Point(175, 55),   // Start-/Endpunkt
                          new Point(200, 125),  // 1. SP
                          new Point(200, 225),  // 2. SP
                          new Point(175, 215),  // Start-/Endpunkt
                          new Point(125, 250),
                          new Point(250, 125),
                          new Point(25, 200)};  // Endpunkt

Text = "AddLine() - 2 Linien, StartFigure";

Text = "AddBeziers() - 3 Bezierkurven";
gp.AddBeziers(p);
gp.AddEllipse(p[0].X-5, p[0].Y-5, 10, 10);
gp.AddEllipse(p[3].X-5, p[3].Y-5, 10, 10);
gp.AddEllipse(p[6].X-5, p[6].Y-5, 10, 10);
gp.AddEllipse(p[9].X-5, p[9].Y-5, 10, 10);
```

Es ist schwierig, mit Bezierkurven über mehrere Punkte hinweg glatte (= stetig differenzierbare) oder gar gleichmäßig geglättete Kurven hinzubekommen. Bei Verwendung *kardinaler Splines* wird dieses Unterfangen hingegen zum Sonntagsausflug: Kardinale Splines erfordern nichts weiter als Angabe

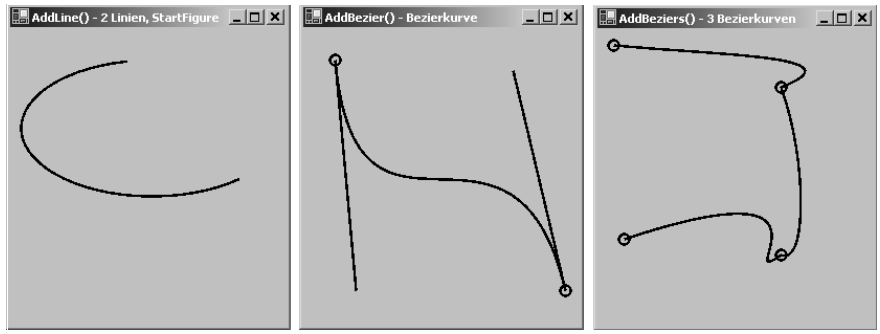


Abbildung 14.2: Kurven

von Kurvenpunkten und einem für die gesamte Kurve geltenden Glättungsfaktor. Der Vorgabewert für den Glättungsfaktor ist `0.5f` und ergibt eine Minimalkurve mit einer ausgewogenen Glättung über alle Punkte hinweg (Abbildung 14.3, links und Mitte). Ein kleinerer Glättungsfaktor bewirkt kleinere Kurvenkrümmungen an den Punkten, ein größerer sorgt für eine stärkere Krümmung. Glättungsfaktoren über `1.0f` führen zu einer Überbiegung der Kurve und etablieren weitere Schnörkel (Abbildung 14.4, links). Ist der Wert 0 (Abbildung 14.3, rechts), erhalten Sie einen Linienzug – wie mit `AddLines()` bzw. `AddPolygon()`. Tatsächlich sind auch negative Glättungen erlaubt, wobei die Frage offen bleibt, ob sie sinnvoll verwendbar sind.

Zur Definition von Splines gibt es zwei Methoden: `AddCurve()` definiert einen offenen, `AddClosedCurve()` einen geschlossenen Kurvenzug als kardinalen Spline. Anzugeben ist nichts weiter als ein `Point`- bzw. `PointF`-Array sowie optional ein Glättungsfaktor.

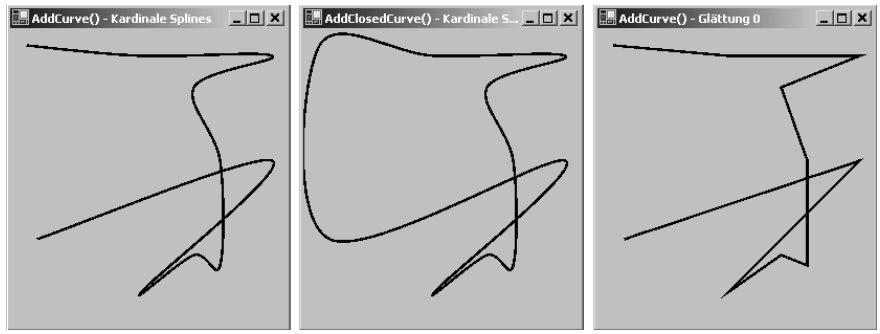


Abbildung 14.3: Kardinale Splines – links: offen Glättung 0,5; Mitte geschlossen; rechts: Glättung 0

Füllen und Maskieren

Natürlich lassen sich Grafikpfade auch als Füllfiguren ausgeben. Die Grafikoperation des Grafikkontexts dafür ist `FillPath()`. Nicht geschlossene Figuren des Grafikpfads werden dabei implizit mit `CloseFigure()` geschlossen (Abbildung 14.4, Mitte):

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  ...
  e.Graphics.FillPath(Brushes.Black, gp);
```

Ein Grafikpfad lässt sich über die `SetClip()`-Methode des Grafikkontextes auch als Maske für eine beliebige Anzahl nachfolgender Grafikoperationen setzen. Die folgende Codesequenz liefert demnach dasselbe Ergebnis wie der vorige `FillPath()`-Aufruf, obwohl eigentlich ein Rechteck von der Größe des Clientbereichs gezeichnet wird:

```
e.Graphics.SetClip(gp);
e.Graphics.FillRectangle(Brushes.Black, ClientRectangle);
```

Bei Angabe eines `Region`-Objekts für den `SetClip()`-Aufruf bleiben sicher keine Wünsche mehr offen. Der Aufruf unterscheidet dann zusätzlich noch verschiedene Kombinationsmodi mit der bestehenden Bereichsdefinition, die über die Elemente der `CombineMode`-Aufzählung ausgedrückt werden – `Complement`, `Exclude`, `Intersect`, `Replace`, `Union` und `Xor`. Eine weitere äquivalente Formulierung des zuvor gezeigten `FillPath()`-Aufrufs wäre demnach:

```
e.Graphics.SetClip(new Region(gp), CombineMode.Replace);
e.Graphics.FillRectangle(Brushes.Black, ClientRectangle);
```

Noch interessanter wird der Effekt allerdings, wenn Sie die Maske an die `Region`-Eigenschaft des Formulars binden. Als Ursprung gilt dann der Ursprung des Formulars, und die Maske blendet alles aus, was außerhalb liegt. Abbildung 14.4 rechts zeigt, was von einem `DrawImage()`-Aufruf überbleibt. Im Hintergrund des Screenshots sehen Sie einen Teil des Programmlistings und somit das Fenster des Codeeditors.

```
this.Region = new Region(gp);
Bitmap b = new Bitmap("eisbaeren.3a.jpg");
e.Graphics.DrawImage(b, -280, -360);
```



Abbildung 14.4: Kurven

Schriften

Es wird kaum überraschen, dass sich auch komplette Schriftzüge in einen Grafikpfad packen lassen. Die Möglichkeiten reichen dabei vom einfachen linearen Schriftzug, der an einem Punkt oder Rechteck ausgerichtet ist, bis hin zu kunstvollen Kompositionen, die alles ausschöpfen, was die Typografie auf Lager hat. Die für Schriftzüge zuständige Methode ist `AddString()`. Im Gegensatz zu den anderen Methoden erschließt sich ihre Anwendung nicht weitgehend von selbst, sondern erfordert je nach Vorhaben einiges an Wissen über Schriften und Typografie. Ein paar der Grundlagen finden Sie in der nun folgenden Beschreibung der Aufrufparameter eingestreut.

Die Methode steht in zwei Varianten zur Verfügung, wovon die erste ein Rechteck und die zweite einen Punkt als Positionsangabe erwartet – jeweils bezogen auf das objekteigene Koordinatensystem. Hier die Prototypen der beiden `int`-Überladungen:

```
public void AddString(string text, FontFamily ff, int fontStyle,
  float emSize, Point position, StringFormat sf);
public void AddString(string text, FontFamily ff, int fontStyle,
  float emSize, Rectangle rect, StringFormat sf);
```

Die Rechteckvariante nimmt einen automatischen Zeilenumbruch auf Wortbasis vor, wenn der Schriftzug die Breite des Rechtecks überschreitet. Das `StringFormat`-Objekt gibt unter anderem die Art der Ausrichtung an dem Punkt bzw. in dem Rechteck vor. Die für die Ausrichtung zuständigen Eigenschaften dieses Objekts sind `Alignment` und `LineAlignment` und können jeweils die Werte `Near`, `Center` oder `Far` der `StringAlignment`-Aufzählung annehmen. Damit sind insgesamt neun Ausrichtungen möglich – von »links oben« (Vorgabewert für Rechteck-Variante) über »oben mitte« und »oben rechts« usw. bis »rechts unten« (Vorgabewert für Punkt-Variante). Während die Interpretation der Ausrichtung für die Rechteckvariante klar sein dürfte, bedarf die Punktvariante der Eselsbrücke, dass sich die Bezeichner der Aufzählungswerte auf den linken oberen Punkt des Schriftzugs beziehen. Abbildung 14.5 verdeutlicht den Zusammenhang.

Die Parameter 2 bis 4 beschreiben die Schrift auf der Basis eines `FontFamily`-Objekts, getrennt nach *Schrift*, *Schriftschnitt* und *Punktgröße*. Eine kompaktere Beschreibung über ein `Font`-Objekt wie bei der `DrawString()`-Methode des Grafikkontexts ist hier leider nicht möglich.

`FontFamily`-Objekte werden unter Angabe eines Schriftnamens wie »Arial« oder »Times New Roman« konstruiert. Eine Auflistung der auf dem jeweiligen System installierten Schriften liefert die statische Methode `FontFamily.GetFamilies()`.

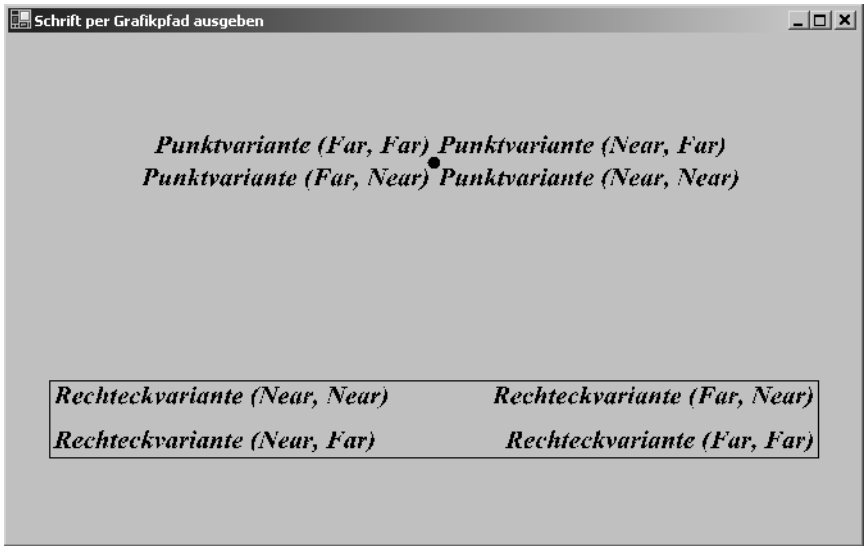


Abbildung 14.5: Ausrichtung bei der Punkt- und der Rechteckvariante

Die durch das `FontFamily`-Objekt vorgenommene Beschreibung der Schrift basiert auf den üblichen aus der Typografie übernommenen Designmaßen für Schriften und sollte als Ansammlung abstrakter Größen begriffen werden. Abbildung 14.6 zeigt, welche Maße bei einer Schrift eine Rolle spielen. Tabelle 14.7 benennt die Methoden zur Ermittlung dieser Maße sowie die Werte, die diese für die Schrift Times New Roman liefern. Wie Sie sicher wissen, werden Schriftgrößen in Punkt ausgedrückt, wobei der typografische Punkt 1/72 Zoll oder 0,353 mm entspricht. Die so genannte GEVIERTHÖHE der Schrift ist ein reines Referenzmaß für die Berechung der Punktgrößen (= Em-Höhe) und hat in Abbildung 14.6 keine Entsprechung. Für die Schrift Times New Roman hat sie beispielsweise den Wert 2048, während der Zeilenabstand das Designmaß 2355 hat. Diese Werte sind dann so zu interpretieren: Bei einer Punktgröße von 2048 hat die Schrift den Zeilenabstand 2355 Punkt. Entsprechendes gilt auch für die anderen Designmaße.

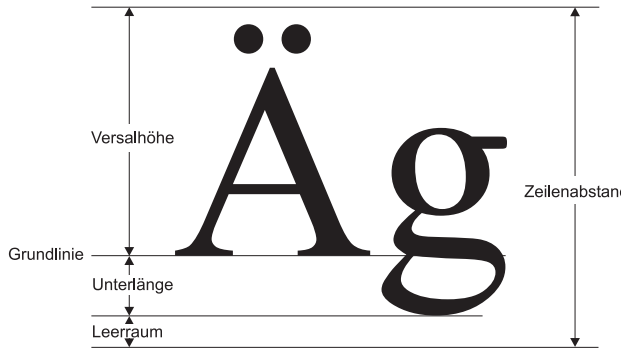


Abbildung 14.6: Designmaße für Schriften

Tabelle 14.7: Methoden zur Bestimmung der Designmaße einer Schrift

Methode	Wert für Times New Roman	Bedeutung
`int GetEmHeight()`	2048	Gevierthöhe (Bezugsmaß für die Punktberechung von Schriftgrößen)
`int GetLineSpacing()`	2355	Zeilenabstand für die Schrift
`int GetCellDescent()`	443	Unterlänge der Schrift (Buchstabenhöhe unter der Grundlinie)
`int GetCellAscent()`	1825	Versalhöhe der Schrift (Buchstabenhöhe über der Grundlinie)

Für andere Schriftgrößen lassen sich die Maße recht einfach aus diesem Zusammenhang errechnen. Um den Zeilenabstand für die Punktgröße *x* zu erhalten, multiplizieren Sie *x* mit dem Quotienten aus den Funktionswerten der Methoden `GetLineSpacing()` und `GetEmHeight()`. Für eine Times New Roman in 100 Punkt ergibt sich so theoretisch der Zeilenabstand 2355 / 2048 * 100 = 114,99 Punkt. Zur Umrechnung von Schriftpunkten in Bildpunkte muss dieser Wert weiterhin mit der vertikalen Auflösung des Grafikkontexts dividiert durch 72 – also dem Faktor `DpiY` / 72 – multipliziert werden, so dass der Zeilenabstand bei einer Bildschirmausgabe mit 96 dpi tatsächlich 153,32 Bildpunkte beträgt. Genau diesen Wert liefert auch die `GetHeight()`-Methode bzw. `Height`-Eigenschaft eines `Font`-Objekts für diese Schrift. Als Code sieht die Berechnung so aus:

```
FontFamily fFam = new FontFamily("Times New Roman");
FontStyle fs = FontStyle.Regular;                // Schriftstil
int pt = 100;                                    // Punktgröße der Schrift
// Zeilenabstand berechnen
float linespace = (float)fFam.GetLineSpacing(fs) / fFam.GetEmHeight(fs) *
                  pt * e.Graphics.DpiY / 72).ToString();  // 153,3203
```

Wie der Hinweis auf die `GetHeight()`-Methode bereits nahe legt, ist es freilich die bessere Alternative, wenn Sie sich mit den gleichen Parametern parallel ein `Font`-Objekt konstruieren und Berechnungen dieser Art über dessen Methoden abwickeln (vgl. Seite 528 ff.).

Nun fehlt noch der Schriftschnitt, damit der Aufruf der Methode `AddString()` endlich über die Bühne gehen kann. Die Methode erwartet hierfür – für .NET untypisch – inkonsequenterweise einen `int`-Wert, der sich aber durch Typumwandlung aus einem `FontStyle`-Bitvektor gewinnen lässt. Die `FontStyle`-Aufzählung definiert dafür die Werte `Bold`, `Italic`, `Regular`, `Strikeout`, `Underline`.

Codebeispiel für die Schriftausgabe

Die folgende Routine stellt einen Grafikpfad mit einem dreizeiligen Text zusammen und gibt diesen einmal in der oberen Fensterhälfte als Umriss- und einmal in der in der unteren Fensterhälfte – hier um 45° – gedreht als Füllfigur aus. Die drei Zeilen des Texts sind unterschiedlich ausgerichtet, die erste Zeile ist rechts, die zweite in der Mitte und die dritte links zum Bezugspunkt (0,0). Ein Fadenkreuz markiert den Ursprung des globalen Koordinatensystems.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  int wMedium = ClientRectangle.Width / 2;
  int hMedium = ClientRectangle.Height / 2;

  // Fadenkreuz
  e.Graphics.DrawLine(Pens.Black, wMedium, 0, wMedium, 2 * hMedium);
  e.Graphics.DrawLine(Pens.Black, 0, hMedium, 2 * wMedium, hMedium);

  GraphicsPath gp = new GraphicsPath();
  FontFamily fFam = new FontFamily("Times");        // Schriftfamilie
  FontStyle fs = FontStyle.Bold | FontStyle.Italic; // Schriftstil
  int emH = 40;                                     // Punktgröße der Schrift

  // Zeilenabstand über die em-Konstante berechnen
  float linespace = (float)fFam.GetLineSpacing(fs) / fFam.GetEmHeight(fs) *
                    pt * e.Graphics.DpiY / 72).ToString();  // 153,3203

  // Grafikpfad mit drei Zeilen zusammenbasteln
  StringFormat sf = new StringFormat();
  sf.Alignment = StringAlignment.Near;    // Ausrichtung rechts von Punkt
  gp.AddString("Times Roman", fFam, (int)fs, emH, new Point(0,0), sf);
  sf.Alignment = StringAlignment.Center;  // Ausrichtung Mitte Punkt
  gp.AddString("Times Roman", fFam, (int)fs, emH, new Point(0,lineSpace),
    sf);
  sf.Alignment = StringAlignment.Far;     // Ausrichtung links Punkt
  gp.AddString("Times Roman", fFam, (int)fs, emH,
    new Point(0, 2 * lineSpace), sf);

  // Grafikpfad in die horizontale Mitte verschieben
  Matrix matrix = new Matrix();
  matrix.Translate(wMedium, 0);
  gp.Transform(matrix);

  // Grafikpfad als Umrissschrift zeichnen
  e.Graphics.DrawPath(new Pen(SystemColors.WindowText), gp);

  // Ursprung exakt in die Mitte des Grafikpfads stetzen, drehen,
  // und Ergebnis zurück in die Mitte sowie in die untere Hälfte des
  // Formulars verschieben
  matrix = new Matrix();
  matrix.Translate(-wMedium, - 1.5f * lineSpace, MatrixOrder.Append);
  matrix.Rotate(45, MatrixOrder.Append);
  matrix.Translate(wMedium, hMedium + 1.5f * lineSpace,MatrixOrder.Append);
```

```
  // Grafikpfad ausgefüllt zeichnen
  gp.Transform(matrix);
  e.Graphics.FillPath(new SolidBrush(SystemColors.WindowText), gp);
}
```

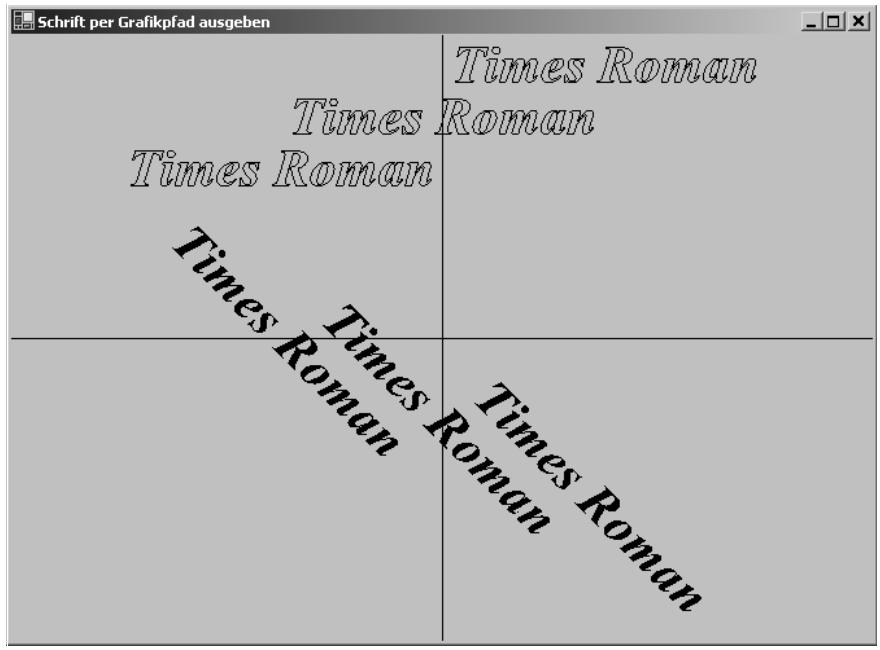


Abbildung 14.7: Schriftausgabe per Grafikpfad – einmal als Umriss-, einmal als Füllfigur

14.3.3 Farben – die Color-Struktur

Die `Color`-Struktur repräsentiert eine Farbe. Zur Darstellung von Farben verwendet Windows das ARGB-Modell, bei dem sich jede Farbe aus vier Komponentenwerten aus Wertebereich 0 bis 255 zusammensetzt. Der Alpha-Wert beschreibt die Transparenz, wobei 0 für »voll transparent« und 255 für »undurchsichtig« (*opaque*) steht. Die anderen drei Werte sind die Rot-, Grün- und Blau-Anteile der Volltonfarbe.

Die statische Ausstattung der `Color`-Struktur ermöglicht es, sowohl benannte Farben abzurufen als auch beliebige Farbwerte aus dem ARGB-Farbraum zu generieren.

Um beliebige Farbwerte zu generieren, verwenden Sie die Methode `FromArgb()`, die in drei Überladungen existiert.

```
color = Color.FromArgb(
        (int)(alpha * 0x1000000 + red * 0x10000 + green * 0x100 + blue));
color = Color.FromArgb(alpha, red, green, blue);
color = Color.FromArgb(red, green, blue);     // alpha = 255
```

Sofern Sie nicht mit Transparenz arbeiten, sollten Sie die dreiparametrige Variante verwenden, die 255 als Alphawert einsetzt. Dummerweise erwartet die einparametrige Variante einen `int`-Wert, der durch eine »harte« Typumwandlung erzwungen werden muss, weil hier die Breite des Datentyps voll ausgenutzt wird – `uint` wäre besser, ist aber eben nicht CLS-konform.

Für das Hantieren mit `Bitmap`-Objekten sind diese Methoden essenziell, denn Bildpunkte sind nichts anderes als Farbwerte. Um den `Color`-Farbwert eines Bildpunkts für ein solches Objekt abzufragen, steht Ihnen die Methode `GetPixel()` zur Verfügung. Umgekehrt lassen sich Bildpunkte mit `SetPixel()` setzen. Die folgende Schleife schneidet ein »echtes« Loch – nämlich einen voll transparenten Bereich – in eine Bitmap:

```
Bitmap b = new Bitmap(filename);
for (int x = b.Width/4; x < b.Width*3/4; x++)
for (int y = b.Height/4; y < (b.Height*3)/4; y++)
  b.SetPixel(x, y, Color.FromArgb(0));
...
```

Systemfarben – SystemColors-Klasse

Windows verwendet 26 Systemfarben, mit denen standardmäßig die Flächen und Beschriftungen der Steuerelemente und der Fensterbereiche in den verschiedenen Zuständen gezeichnet werden. Die Definition dieser Systemfarben können Sie über die SYSTEMSTEUERUNG/ANZEIGE/DARSTELLUNG interaktiv definieren und als Farbschema abspeichern.

Die `SystemColors`-Klasse erlaubt es, diese Farben über 26 statische Eigenschaften abzurufen, deren Bezeichner die Farben auch benennen. Da viele Benutzer auf ihren Systemen eigene Farbschemata einsetzen, sollten Sie als Farben für Text und Zeichnungen, deren Hintergrund Sie nicht selbst zeichnen, in jedem Fall Farben aus diesem Pool verwenden – das vermeidet den Effekt der »österreichischen Kriegsflagge«: weißer Adler auf weißem Grund.

Tabelle 14.8: Die 26 Systemfarben von Windows

Eigenschaft	von Windows verwendet für
`ActiveBorder`	Rahmen des aktiven Fensters
`ActiveCaption`	Titelleisten des aktiven Fensters
`ActiveCaptionText`	Beschriftung in der Titelleiste des aktiven Fensters
`AppWorkspace`	Hintergrundfarbe des Clientbereichs in übergeordnetem MDI-Fenster
`Control`	Hintergrundfarbe für Steuerelemente
`ControlDark`	Schattierung von 3D-Steuerelementen

Tabelle 14.8: Die 26 Systemfarben von Windows (Forts.)

Eigenschaft	von Windows verwendet für
`ControlDarkDark`	Dunkle Schattierung von 3D-Steuerelementen
`ControlLight`	Aufhellung von 3D-Steuerelementen
`ControlLightLight`	Starke Aufhellung von 3D-Steuerelementen
`ControlText`	Textfarbe für Steuerelemente
`Desktop`	Farbe des Windows-Desktop
`GrayText`	Textfarbe in deaktivierten Bereichen (`Enabled = false`)
`Highlight`	Hintergrund für ausgewählte Elemente
`HighlightText`	Textfarbe für ausgewählte Elemente
`HotTrack`	Farbe für vorselektierte Elemente, deren Aktion mit einfachem Mausklick ausgelöst werden kann
`InactiveBorder`	Rahmenfarbe für inaktive Fenster
`InactiveCaption`	Titelleistenfarbe für inaktive Fenster
`InactiveCaptionText`	Textfarbe in der Titelleiste inaktiver Fenster
`Info`	Hintergrundfarbe für Quickinfo-Fenster
`InfoText`	Textfarbe für Quickinfo-Fenster
`Menu`	Hintergrundfarbe für Menüeinträge
`MenuText`	Textfarbe für Menüeinträge
`ScrollBar`	Hintergrundfarbe für Bildlaufleisten
`Window`	Hintergrundfarbe für den Clientbereich von Fenstern
`WindowFrame`	Farbe für Fensterrahmen
`WindowText`	Textfarbe für Ausgabe von Text in Fenstern

Wie Abbildung 14.8 zeigt, geht es dabei nicht notwendigerweise um 26 verschiedene Farben, sondern um 26 Zuordnungen zu verschiedenen Elementen der Windows-Oberfläche.

KnownColor-Aufzählung und Color-Eigenschaften

Um benannte Farben abzurufen, verwenden Sie eine der statischen `Color`-Methoden `FromKnownColor()` oder `FromName()`. Erstere liefert die Farbe zu einem Wert aus der `KnownColor`-Aufzählung, die insgesamt 166 Elemente enthält,

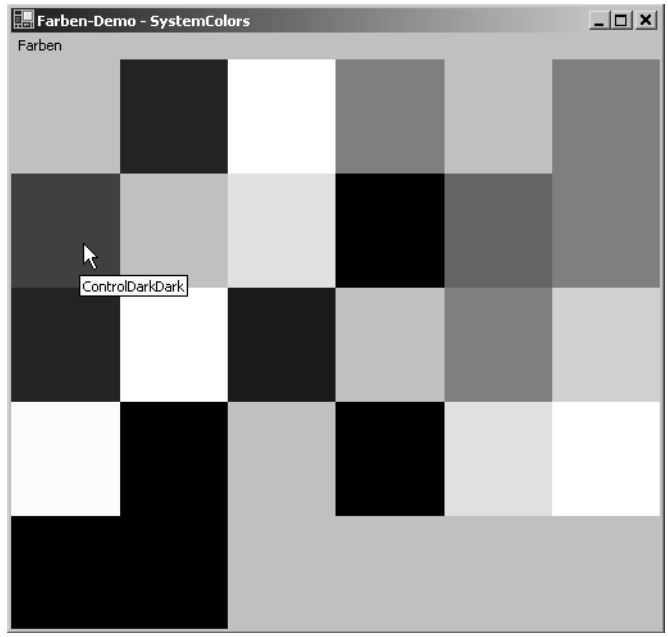


Abbildung 14.8: Die Klasse `SystemColors` enthält die über das aktuelle Farbschema für die Benutzeroberfläche definierten 26 Systemfarben. Das `ToolTip`-Fensterchen gibt darüber Aufschluss, welche Farbe welchen Namen trägt.

letztere die Farbe zu einem Farbnamen, wobei die `Color`-Klasse für jede benannte Farbe eine Eigenschaft des Typs `Color` mit den entsprechenden Namen bereitstellt – 167 an der Zahl. Obwohl diese beiden benannten Farbräume viele gleiche Farben enthalten (die `KnownColor`-Aufzählung enthält einige sogar mehrfach, weil sie auch die Farben der `SystemColors`-Klasse einschließt), sind sie doch unterschiedlich benannt – ein Tribut an die bestehende Farblandschaft von Windows. Die zusätzliche Farbe der `Color`-Klasse ist übrigens `Transparent`.

Damit haben Sie drei Möglichkeiten, benannte Farben aufzurufen:

```
Color c1 = Color.Black;
Color c2 = Color.FromKnownColor(KnownColor.Black);
Color c3 = Color.FromName("Black");
```

Jede dieser Möglichkeiten hat ihren eigenen Charme, auch wenn die `KnownColor`-Aufzählung auf den ersten Blick unmotiviert erscheinen mag. Da sie Konstanten liefert, lassen sich ihre Werte in `switch`-Konstrukten einsetzen.

Codebeispiel – Farben

Das Beispielprojekt *Farben*, dessen Ausgaben die umliegenden Abbildungen zeigen, ist ein gutes Werkzeug für die tägliche Programmierung mit der *Color*-Struktur – und demonstriert dabei auch die verschiedenen Möglichkeiten, die Sie haben, um Farben »anzufassen«.

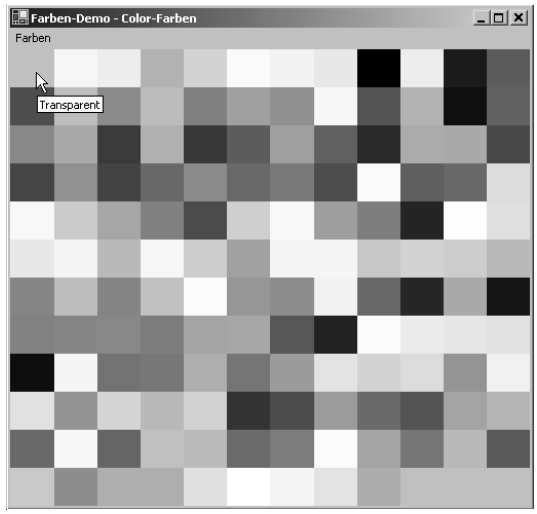


Abbildung 14.9: Die `Color`-Klasse enthält 166 »bekannte Farben«. Das `ToolTip`-Fensterchen zeigt an, welche Farbe welchen Namen trägt.

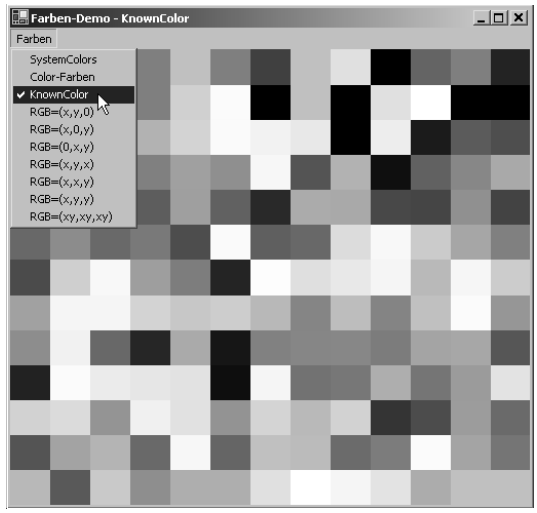


Abbildung 14.10: Die `KnownColor`-Aufzählung enthält 167 »bekannte Farben«, darunter auch die Systemfarben. Auch hier gibt ein `ToolTip`-Fensterchen darüber Aufschluss, welche Farbe welchen Namen trägt.

Ein Blick auf das Menü (Abbildung 14.10) verrät den Funktionsumfang. Die nicht gerade triviale Implementierung zerfällt in fünf Teile:

1. Pflege des Menüs
2. Zusammenstellung einer Bitmap mit einem wählbaren Farbverlauf
3. Zusammenstellen der benannten Farbenräume in einem `Color`-Array
4. Ausgabe der Farben aus dem ausgewählten benannten Farbraum als Klötzchen
5. Ausgabe der Farbnamen und Farbwerte über eine `ToolTip`-Komponente

Pflege des Menüs

Alle Menübefehle verwenden dieselbe Behandlungsmethode. Sie sind – analog zu `RadioGroup`-Steuerelementen – miteinander verschaltet und lösen sich gegenseitig aus. Die `Checked`-Eigenschaften der ersten drei Menübefehle steuern die `OnPaint()`-Methode, so dass die Zusammenstellung der Farbwerte für den ausgewählten Farbraum bei jedem Zeichnen erfolgt – es könnte sich ja seit dem letzten Zeichnen etwas geändert haben. Aus diesem Grund kommt auch die Behandlung des Ereignisses `SystemColorsChanged` mit einem schlichten Aufruf von `Invalidate()` aus. Die restlichen Menübefehle generieren jeweils eine neue Bitmap und ordnen diese der `BackgroundImage`-Eigenschaft des Formulars zu. Für die Charakterisierung der Farbverläufe ist eine eigene `enum`-Aufzählung `XY` definiert, die es möglicht, die einzelnen Farbkomponenten jeweils auszublenden oder der *x*- bzw. *y*-Koordinate sowie einer Kombination daraus (*x*/2+*y*/2) zuzuordnen.

```
private enum XY {None, X, Y, XY};   // Modi für Bitmap
private void menuFarbenItems_Click(object sender, System.EventArgs e)
{
  Size size = ClientRectangle.Size;
  MenuItem m = (MenuItem) sender;
  for(int i = 0; i < m.Parent.MenuItems.Count; i++)
    m.Parent.MenuItems[i].Checked = false;
  m.Checked = true;

  if (m.Text.IndexOf("RGB") == 0)
  {
    switch(m.Index)  // Jeweils neue Bitmap generieren
    {
      case 3:
        fillColorBitmap(size, XY.X, XY.Y, XY.None); break;
      case 4:
        fillColorBitmap(size, XY.X, XY.None, XY.Y); break;
      case 5:
        fillColorBitmap(size, XY.None, XY.X, XY.Y); break;
      case 6:
        fillColorBitmap(size, XY.X, XY.Y, XY.X); break;
      case 7:
        fillColorBitmap(size,  XY.X, XY.X, XY.Y); break;
      case 8:
        fillColorBitmap(size, XY.X, XY.Y, XY.Y); break;
      case 9:
        fillColorBitmap(size, XY.XY, XY.XY, XY.XY); break;
    }
  }
  Text = "Farben-Demo - " + m.Text;
  BackgroundImage = colorBitmap;
  Invalidate();
}
```

Bitmap mit einem wählbaren Farbverlauf

Die Methode `fillColorBitmap()` übernimmt den Aufbau des RGB-Farbverlaufs als Bitmap. Es ist natürlich nicht empfehlenswert, Farbverläufe dieses Umfangs mit einer Zeichenmethode wie `FillRectangle()` zeichnen zu wollen – daher auch die Bitmap. Allerdings werden Sie an späterer Stelle in Form der Klassen `PathGradientBrush` und `LinearGradientBrush` noch ein wesentlich einfacheres Verfahren kennen lernen, um Farbverläufe zu realisieren. In diesem Beispiel geht es primär um die Handhabung und ein Verständnis der Farben aus dem ARGB-Farbraum:

```
private Bitmap colorBitmap;
// Füllt das Datenfeld colorBitmap mit Farbverläufen
void fillColorBitmap(Size size, XY r, XY g, XY b)
{
  colorBitmap = new Bitmap(size.Width, size.Height);
  double stretchX = size.Width/256f;
  double stretchY = size.Height/256f;
  for(int x1 = 0; x1 < size.Width; x1 += 1)
    for(int y1 = 0; y1 < size.Height; y1 += 1)
    {
      int x = (int)(x1/stretchX);
      int y = (int)(y1/stretchY);
      Color color;
      // Farbkomponenten zusammenstellen
      int red =   (r==XY.X ? x : 0) + (r==XY.Y ? y : 0) +
                  (r==XY.XY ? (x+y)/2 : 0);
      int green = (g==XY.X ? x : 0) + (g==XY.Y ? y : 0) +
                  (g==XY.XY ? (x+y)/2 : 0);
      int blue =  (b==XY.X ? x : 0) + (b==XY.Y ? y : 0) +
                  (b==XY.XY ? (x+y)/2 : 0);
      color = Color.FromArgb(red, green, blue);
      colorBitmap.SetPixel(x1, y1, color);
    }
  for (int x = colorBitmap.Width/4; x < colorBitmap.Width*3/4; x++)
  for (int y = colorBitmap.Height/4; y < (colorBitmap.Height*3)/4; y++)
    colorBitmap.SetPixel(x,y,Color.FromArgb(0));
}
```

Zusammenstellen der benannten Farbenräume

Eine Aufzählung der Farben der `SystemColors`-Klasse sowie der `Color`-Struktur in einem `Color`-Array namens `colors` liefert die Methode `setColorArray()`. Sie macht vom .NET-Reflection-Mechanismus Gebrauch und verwendet deshalb ein stark fortgeschrittenes Konzept der .NET-Programmierung. Ihre doch recht »kompakte« Implementierung wird hier der Vollständigkeit halber vorgestellt, weil sie generell zeigt, wie man Klassen nach statischen Eigenschaften eines bestimmten Typs durchsucht – eine Vertiefung der benutzten Mechanismen ist an dieser Stelle leider nicht möglich.

```
Color[] colors
private void setColorArray(Type t)
{
  // alle Elemente der Klasse System.Colors ermitteln und in Array packen
  ArrayList colorsX = new ArrayList();
  System.Reflection.PropertyInfo[] colorPIs = t.GetProperties();
  for(int i = 0; i < colorPIs.Length; i++)
    if (colorPIs[i].GetGetMethod().ReturnType == typeof(Color))
      colorsX.Add(colorPIs[i].GetValue(null,null));
  colorsX.CopyTo(colors = new Color[colorsX.Count],0);
}
```

Der Aufruf der `setColorArray()`-Methode findet sich in `OnPaint()`. Obwohl der Code für die Auflistung der `enum`-Aufzählung `KnownColor` dem gleichen Muster folgt, sieht er etwas freundlicher aus. Er steckt im vorliegenden Fall *inline* in der `OnPaint()`-Methode, lässt sich aber prinzipiell für *jede* `enum`-Aufzählung verwenden, die selbst keine `GetValues()`-Methode bereitstellt – Sie müssen nur den Aufzählungstyp geeignet anpassen:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
  if (menuColor.Checked)
  {
    BackgroundImage = null;
    // alle Elemente der Klasse Color ermitteln und in Array packen
    setColorArray(typeof(Color));
    DrawColors(colors, e); // zeichnet das Array als Rechtecke
  }
  else if (menuSystemColors.Checked)
  {
    BackgroundImage = null;
    // alle Elemente der Klasse System.Colors ermitteln und in Array packen
    setColorArray(typeof(SystemColors));
    DrawColors(colors, e); // zeichnet das Array als Rechtecke
  }
  else if (menuKnownColors.Checked)
  {
    BackgroundImage = null;
    // alle Elemente der Aufzählung KnownColor ermitteln und
    // in das Array packen
    Array  arr = System.Enum.GetValues(typeof(KnownColor));
    colors = new Color[arr.Length];
    for(int i = 0; i< arr.Length;i++)
    {
      colors[i] = Color.FromKnownColor((KnownColor)arr.GetValue(i));
    }
    DrawColors(colors, e); // zeichnet das Array als Rechtecke
  }
  else
  {
    colors = null;
  }
}
```

Ausgabe der Farben als Klötzchen

Die Ausgabe der Farbklötzchen erfordert etwas Rechnerei, hat im Grunde genommen aber nur ein kreatives Detail zu bieten: Um eine gute Aufteilung zwischen Zeilen und Spalten zu finden, errechnet der Code die nächst größere Quadratzahl und nimmt diese als Grundlage für die Zellenberechnung.

```
void DrawColors(Color[] colors, PaintEventArgs e)
{
  int colorsInARow = (int) Math.Ceiling(Math.Sqrt(colors.Length));
  float cellWidth = (float)ClientRectangle.Width / colorsInARow;
  int colorsInAColumn =
    (int) Math.Ceiling((double)colors.Length/colorsInARow);
  float cellHeight = (float)ClientRectangle.Height / colorsInAColumn;
  Point p = new Point();
  for(int i = 0; i < colors.Length; i++)
  {
    // Wert für Element der Klasse System.Color
    float x = (i % colorsInARow) * cellWidth;
    float y = (i / colorsInARow) * cellHeight;
    e.Graphics.FillRectangle(new SolidBrush(colors[i]),
       x,y,cellWidth, cellHeight);
  }
}
```

Ausgabe der Farbnamen und Farbwerte als Tooltipp

Es gibt wenig Anwendungen, mit denen sich die `ToolTip`-Komponente mit so wenig Code derart eindrucksvoll in Szene setzen lässt wie bei der Ausgabe der Farbnamen und Farbwerte.

Grundlage für die Berechnung ist die Mausposition, weshalb der Code als `MouseMove`-Behandlung formuliert ist. Die Ausgabe der Farbnamen erfordert eine Abbildung der Mauskoordinaten auf einen Index für das `colors`-Array, was im Prinzip nichts anderes ist als eine Umkehrung der Situation in `OnPaint()`. Lästig (und keinesfalls überflüssig) sind dabei nur die Typumwandlungen zur Erzwingung von Gleitkomma-Arithmethik.

Bei der Ermittlung des Farbwertes in einem Farbverlauf erweist es sich als Vorteil, dass eine Bitmap vorhanden ist. Das lässt den Code dafür fast zum Einzeiler geraten. (Es würde einigen Aufwand erfordern, diesen Wert rein rechnerisch zu ermitteln, weil die direkte Abfrage von Bildpunkten eines Grafikkontexts in .NET nicht vorgesehen ist.)

```
// Setzt Text für die ToolTip-Komponente
private void Form1_MouseMove(object sender,
System.Windows.Forms.MouseEventArgs e)
{
  if(colors != null)  // benannte Farben
  {
    int colorsInARow = (int) Math.Ceiling(Math.Sqrt(colors.Length));
```

```
    int x = (int) ((float)e.X/ClientRectangle.Width * colorsInARow) ;
    int y = (int) ((float)e.Y/ClientRectangle.Height *
                   Math.Ceiling((double)colors.Length/colorsInARow));
    int i = y * colorsInARow + x;
    if (i < colors.Length)
      this.toolTip1.SetToolTip(this, colors[i].Name);
    else
      this.toolTip1.SetToolTip(this, "");
  }
  else    // ARGB-Farben
  {
    if(colorBitmap != null)
    {
      Color color = colorBitmap.GetPixel(e.X, e.Y);
      this.toolTip1.SetToolTip(this, "RGB "+ color.ToString());
    }
  }
}
```

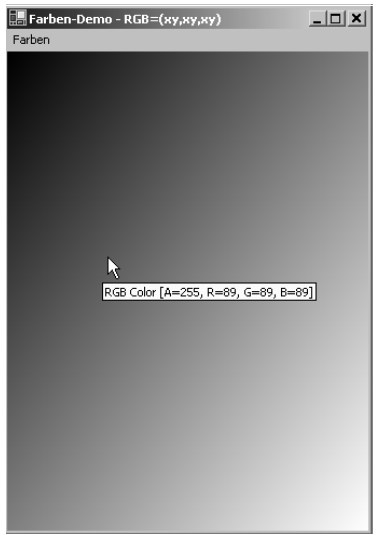




Abbildung 14.11: Anzeige der Farbwerte in einem Farbverlauf als ToolTip-Text

Übungen

- Ergänzen Sie einen Menüpunkt, der die Farben der drei benannten Farbräume nach Farbwerten sortiert.
- Erweitern Sie die Ausgabe der `ToolTip`-Komponente dahingehend, dass sie zusätzlich zu den Farbnamen auch die Farbwerte ausgibt.
- Die Bitmap für den Farbverlauf könnte größer sein. Überlegen Sie, wie Sie unter Verwendung der fehlenden Farbe (bzw. Farbkombination) an den vier Rändern nahtlos »anbauen« könnten und implementieren Sie einen Anbau als eigenständigen Menübefehl.

14.3.4 Schreibzeug – Stift und Pinsel

Während die Grafikoperationen für die Ausgabe von Bildern – also `DrawImage()`, `DrawIcon()` usw. – ohne »Schreibzeug« auskommen, erwarten alle anderen auf einen Grafikkontext bezogenen Ausgabeoperationen neben anderen Parametern ein `Pen`- oder ein `Brush`-Objekt – je nachdem, ob es sich dabei um eine Zeichen- oder um eine Fülloperation handelt. In der Regel können Sie davon ausgehen, dass `DrawXxx()`-Methoden einen STIFT bzw. ein `Pen`-Objekt erwarten und `FillXxx()`-Methoden einen PINSEL bzw. ein `Brush`-Objekt. (Ausnahmen bestätigen wie immer die Regel: `DrawString()` besteht auf ein `Brush`-Objekt. Diese Methode füllt in der Tat Grafikpfade und hätte somit eher den Namen »FillString()« verdient.)

Unter .NET zeigt sich nun, dass die eigentlich von Anfang an in der Windows-Programmierung vorhandenen Stift- und Pinsel-Abstraktionen schon immer eine sehr treffende Abbildung der wirklichen Verhältnisse waren. Tatsächlich haben sie nicht nur den Übergang zur objektorientierten Programmierung und GDI+ nahezu unbeschadet überstanden, sie können nun auch ihr Potenzial noch weiter entfalten. Die mit dem GDI+ neu hinzugekommenen Pinselarten `LinearGradientBrush` und `PathGradientBrush` führen dies eindrucksvoll vor.

Im Folgenden finden Sie einen Überblick über die Ausstattung der Klasse `Pen` sowie über die verschiedenen Ausprägungen der von der abstrakten Klasse `Brush` abstammenden Pinselklassen. Bitte erwarten Sie hier keine Vollständigkeit, da es die beiden Objekttypen durchaus in sich haben, sobald ihre benutzerdefinierten Einstellungen ins Spiel kommen.

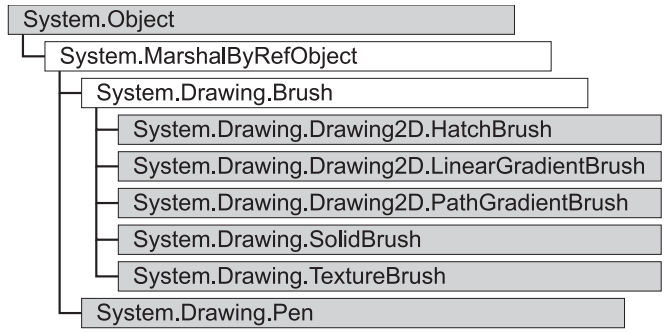


Abbildung 14.12: Die Klassen `Pen` und `Brush` in der Klassenhierarchie

Stifte – die Pen-Klasse

Die Klasse `Pen` bietet alles, was Sie von Zeichenprogrammen an Möglichkeiten für die Gestaltung von Linien kennen – und noch einiges mehr. Farben, Strichstärken, Linienstile (auch freidefinierbare) und natürlich umfangreiche Möglichkeiten für die Gestaltung von Linienenden und Linienstößen.

Falls Sie einfach nur »gewöhnliche« Linien zeichnen wollen und außer an die Farbe keine besonderen Ansprüche an das zu verwendende `Pen`-Objekt haben, sind Sie bei der `Pens`-Klasse an der richtigen Adresse. Diese Klasse bietet – in Form statischer Nur-Lesen-Eigenschaften – 141 Stifte mit den benannten Farben von `AliceBlue` bis `YellowGreen` (genauer: die 167 Farben der `KnownColor`-Auflistung abzüglich der 26 Systemfarben) zur sofortigen Verwendung an:

```
e.Graphics.DrawRectangle(Pens.Black, 10, 10, 300, 300);
```

Strichstärke

Die `Pens`-Auflistung ist zwar bequem, hat aber zwei echte Haken:

1. Sie spart die Systemfarben (vgl. `SystemColors`-Klasse) aus.
2. Die Strichstärke hat dabei grundsätzlich das Maß 1 (vgl. Seite 467), was bei unverändertem Grafikkontext einer Linie der Breite von 1 Pixel oder der so genannten *Haarlinie* entspricht.

Besser wäre eine »SystemPens«-Klasse, die Stifte für die Systemfarben mit der Strichdicke 1 bereitstellt. Die könnte man wenigstens guten Gewissens zum Zeichnen von Steuerelementen und anderen rein bildschirmorientierten Dingen einsetzen, weil sie zumindest einen vernünftigen Kontrast garantieren – unabhängig vom Farbschema des Benutzers.

Der Hintergrund für den zweiten Haken ist folgender: Auf dem Bildschirm sieht eine Strichstärke von 1 Pixel zwar gut aus und ist im Allgemeinen so auch erwünscht, auf den meisten Druckern jedoch, gleich ob Tintenstrahler, Laserdrucker oder Belichter ist von einer solchen Linie eher wenig zu sehen. Bei einer Auflösung von 600 dpi hat die Linie gerade noch eine Ausdehnung von etwa 0,04 mm und wird vom Druckwerk für gewöhnlich auch nicht mehr anständig gedruckt. Auf einem 2400-dpi-Belichter schrumpft die Strichstärke sogar auf ein Hundertstel Millimeter – 0,1 mm sollten es für Normalsichtige aber mindestens sein, besser 0,25 mm.

Falls Sie jedoch über die `PageUnit`-Eigenschaft ein physikalisches Maß – beispielsweise `GraphicsUnit.Millimeter` – einstellen, haben die Stifte der `Pens`-Klasse allesamt die Strichstärke der Maßeinheit – also 1 mm (im Falle von `GraphicsUnit.Inch` sogar 25,4 mm) – was zwar auf dem Bildschirm und auf dem Drucker in etwa gleich gut kommt, aber zuweilen eben doch recht klobig ist. Die folgende Berechnung ist auf einen Monitor mit der heute üblichen Auflösung von 1280x1024 Bildpunkten zugeschnitten und zeigt, dass 0,25 mm tatsächlich keine schlechte Wahl sind: Die Eigenschaften `DpiX` und `DpiY` eines `Graphics`-Objekts haben in dieser Auflösung und bei der Einstellung »kleine Schriftarten« den Wert 96, was auf etwa 4 Pixel pro Millimeter hinausläuft. Geht man hingegen davon aus, dass eine A4-Druckseite im Hochformat, bei voller Breite – also 21 cm – auf dem Monitor noch vernünftig lesbar sein sollte, käme man über die Berechnung 210 mm/1280 auf eine Strichstärke von 0,16 mm.



Strichstärken unterhalb der Haarlinie werden grundsätzlich auf 1 Pixel aufgerundet, verschwinden also nicht etwa komplett. Die Darstellung auf Bildschirm und Drucker klafft bei solchen Linien aber unter Umständen weit auseinander (Linienstile, Pfeilspitzen etc.).

Konstruktor

Egal, ob Sie mit einer Systemfarbe oder einer geräteunabhängigen Skalierung des Koordinatensystems arbeiten wollen: In der Praxis ist am Konstruktor der Klasse `Pen` kaum vorbeizukommen. Die zweiparametrigen Überladungen ermöglichen es, eine Strichstärke anzugeben. Als Standardstift für die Bildschirmausgabe wäre übrigens zu empfehlen:

```
e.Graphics.PageUnit = GraphicsUnit.Millimeter;
Pen pen = new Pen(SystemColors.WindowText, 0.15f);
e.Graphics.DrawRectangle(pen, 10, 10, 300, 300);
```

Der erste Parameter nimmt einen beliebigen `Color`-Wert – mithin die gesamte Palette der benannten Farben und ARGB-Farben, der zweite einen `int`- oder `float`-Wert (das »f« muss also notiert werden, wenn Dezimalstellen vorhanden sind), der die Strichstärke auf die verwendete Maßeinheit bezieht. Die Eigenschaften `Color` und `Width` ermöglichen es, auch im Nachhinein jederzeit die Farbe und die Strichstärke eines `Pen`-Objekts zu ändern – allerdings nur, wenn dieses nicht der schreibgeschützten `Pens`-Ausstattung entlehnt ist.

Besondere Effekte können Sie erzielen, wenn Sie einen Pinsel als Grundlage für ein Pen-Objekt verwenden. Die folgende `Paint`-Routine demonstriert den Einsatz eines `LinearGradientBrush`-Objekts und unterlegt den Stift mit einem Farbverlauf. Prinzipiell können Sie einem `Pen`-Objekt aber jedes `Brush`-Objekt unterjubeln.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  Rectangle rect = new Rectangle(40, 40, 200, 200);
  LinearGradientBrush lgb = new LinearGradientBrush(rect,
     Color.White, Color.DarkBlue, angle);
  Pen pen = new Pen(lgb, 40);
  // für das linke und mittlere Bild in der Abbildung auskommentiert
  // pen.Alignment = PenAlignment.Inset;
  e.Graphics.DrawRectangle(pen, rect);
}
```

Der Pinsel hat eine Zellengröße von 200x200 Pixel, beginnend ab der Position (40, 40) mit Weiß und endend mit Dunkelblau. Beim linken Bild in Abbildung 14.13 hatte `angle` den Wert 0, beim mittleren 45. Für das rechte Bild wurde zusätzlich die Eigenschaft `pen.Aligment` auf `PenAlignment.Inset` gesetzt.

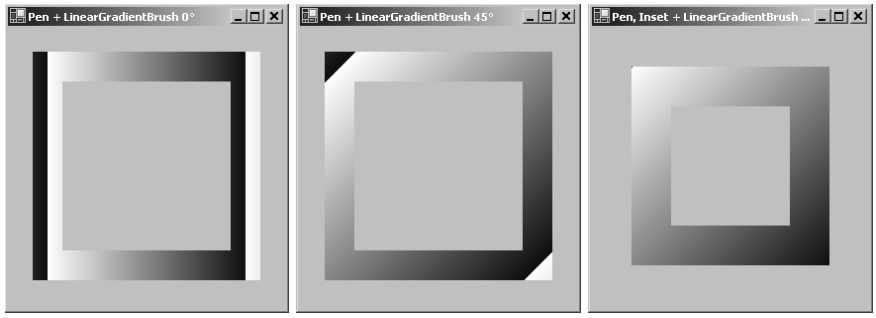


Abbildung 14.13: `Pen`-Objekt auf der Grundlage eines `LinearGradientBrush`-Objekts – drei Variationen

Einstellen des Linienstils für Verbindungspunkte

Aus den drei Variationen in Abbildung 14.13 lässt sich einiges herauslesen:

- Die Strichstärke verteilt sich gleichmäßig nach innen und außen um das angegebene Rechteck. Durch Setzen der `Alignment`-Eigenschaft auf `PenAlignment.Inset` lässt sich alternativ eine Einbeschreibung in den gezeichneten Umriss ausdrücken (rechtes Bild). Die anderen von der `PenAlignment`-Aufzählung angebotenen Werte jenseits vom Standardwert `Center` haben in diesem Zusammenhang übrigens keine Wirkung.
- Winkelausrichtung, Startpunkt und Abmessungen der `Brush`-Zelle spielen eine Rolle. Über die `PenType`-Eigenschaft lässt sich abrufen, auf welcher Basis ein Stift operiert. Die `PenType`-Aufzählung benennt die fünf möglichen Basen: `SolidColor`, `HatchFill`, `LinearGradient`, `PathGradient`, `TextureFill`.
- Die Ecken geschlossener Kurven sind spitz – über die `LineJoin`-Eigenschaft lassen sich auch abgerundete (`LineJoin.Round`) oder angefaste (`LineJoin.Bevel`) Ecken einstellen. Wie Abbildung 14.14 zeigt, sind die Linienenden davon nicht betroffen.

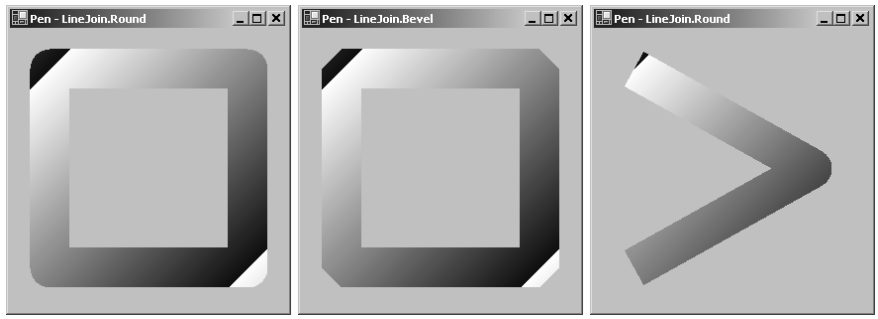


Abbildung 14.14: Wirkung der `LineJoin`-Eigenschaft

Linienenden und Linienstile

Linienenden und Linienstile sind ein Thema für sich – und nachdem die Klasse `Pen` umfangreiche Möglichkeiten für die Definition eigener Linienenden und -stile bereitet, eines mit ziemlicher Tiefe.

Bei den Linienenden sind grundsätzlich drei Fälle zu unterscheiden:

1. *Startpunkt des Zeichenvorgangs* – dieses Linienende wird über die Eigenschaft `StartCap` gesetzt. Über die `LineCap`-Aufzählung steht eine Reihe vordefinierter Linienstile zur Verfügung (Pfeile, Dreieck, Diamant, Rundung usw.). Vorgabewert ist `LineCap.Flat`. Eine besondere Bedeutung hat der Wert `LineCap.Custom`: Er »aktiviert« die Eigenschaft `CustomStartCap`, die es ermöglicht, ein benutzerdefiniertes Linienende in Gestalt eines `CustomLineCap`-Objekts zu setzen. Bei der Konstruktion eines solchen Objekts geben Sie einen Füllpfad und/oder einen Umrisspfad des Typs `GraphicsPath` an. Da selten beide Pfade benötigt werden, lässt sich einer davon auch mit `null` angeben. Dabei gelten die folgenden Regeln:

   – Der Pfad muss die Linie berühren, sonst kommt es zu einer Ausnahme.

   – Die Abmessungen der Füll- bzw. Umrissfigur sind auf eine Linienstärke von 1 zu beziehen. Eine Skalierung auf die tatsächliche Linienstärke nimmt das `Pen`-Objekt von sich aus und automatisch vor.

   – Die Pfade beziehen sich auf eine senkrecht stehende Linie (12 Uhr-Stellung), wobei der Startpunkt oben liegt und die Koordinaten (0,0) hat.

   – Die Linienstärke ist 1 und wird automatisch skaliert.

   – Linienstöße sind spitz (Standard), der dritte Parameter des `CustomLineCap()`-Konstruktors zeigt hier genauso wenig Wirkung wie eine `CloseFigure()`-Anweisung. Um geschlossene Figuren zu zeichnen, sind demnach die `GraphicsPath`-Methoden `AddPolygon()` oder `AddClosedCurve()` zu verwenden.

2. *Start- und Endpunkte von Strichelungen bei gestrichelten Linienstilen* – hierfür ist die Eigenschaft `DashCap` zuständig, wobei Sie einen von drei durch die Aufzählung `DashCap` definierten Stilen einstellen können: `Flat` (Vorgabe), `Round` und `Triangle`. Benutzerdefinierte Stile sind generell nicht möglich. `DashCap` wirkt sich nur aus, wenn die `DashStyle`-Eigenschaft auf einen anderen Wert als `Solid` gesetzt ist. Zur Auswahl stehen die in der `DashStyle`-Aufzählung vordefinierten Linienstile `Dash`, `DashDot`, `DashDotDot` und `Dot`. Der besondere Linienstil `Custom` aktiviert die Eigenschaft `DashPattern`, über die Sie einen selbstdefinierten Linienstil setzen können. Wert dieser Eigenschaft ist ein `float`-Array, dessen Elemente abwechselnd eine Strichlänge und eine Unterbrechungslänge – begin-

nend mit der Länge des ersten Strichs – definieren. Bezugsmaß ist die Strichstärke 1. Die Skalierung für größere Strichstärken nimmt das `Pen`-Objekt von sich aus vor.

3. *Endpunkt des Zeichenvorgangs* – das Ende einer Linie beschreibt die `EndCap`-Eigenschaft, für die ansonsten dasselbe wie für die `StartCap`-Eigenschaft gilt. Für benutzerdefinierte Linienstile ist die `CustomEndCap`-Eigenschaft zuständig.

Codebeispiel – Linienenden

Die folgende `Paint`-Routine aus dem Projekt *PenDemo* zeichnet verschiedene Linienenden und -stile, darunter auch zwei selbstdefinierte Linienenden. Abbildung 14.15 zeigt das Ergebnis.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  Text = "Pen - LineCap-Stile";
  Pen pen = new Pen(Color.Black, 30);

  e.Graphics.DrawLine(pen, 40, 40, 240, 40);    // Standard

  pen.StartCap = LineCap.Round;                 // Startpunkt abgerundet
  pen.EndCap = LineCap.Triangle;                // Endpunkt Dreieck
  e.Graphics.DrawLine(pen, 40, 100, 240, 100);

  pen.StartCap = LineCap.ArrowAnchor;           // beide Enden Pfeile
  pen.EndCap = LineCap.ArrowAnchor;
  e.Graphics.DrawLine(pen, 40, 170, 240, 170);

  pen.StartCap = LineCap.DiamondAnchor;         // Startpunkt Diamant
  pen.DashStyle = DashStyle.Dash;               // gestrichelt
  pen.DashCap = DashCap.Triangle;               // Striche dreieckig
  e.Graphics.DrawLine(pen, 40, 240, 240, 240);

  GraphicsPath gpFill = new GraphicsPath();     // zum Füllen (links)
  GraphicsPath gpStroke = new GraphicsPath();   // Umriss (rechts)
  gpFill.AddEllipse(-1.5f, -.55f, 3f, .8f);

  PointF[] pArr = new PointF[] { new PointF(-2.5f, -1.5f),
                                 new PointF(0, 2.5f),
                                 new PointF(2.5f, -1.5f) };
  gpStroke.AddPolygon(pArr);

  // CustomLineCap-Objekte vorbereiten
  CustomLineCap custStartLC = new CustomLineCap(gpFill, null);
  CustomLineCap custEndLC = new CustomLineCap(null, gpStroke);
  Pen pen1 = new Pen(Color.Black, 30);
  // CustomLineCap-Objekte setzen.
  pen1.CustomStartCap = custStartLC;
  pen1.CustomEndCap = custEndLC;
  e.Graphics.DrawLine(pen1, 300, 150, 500, 150);
}
```

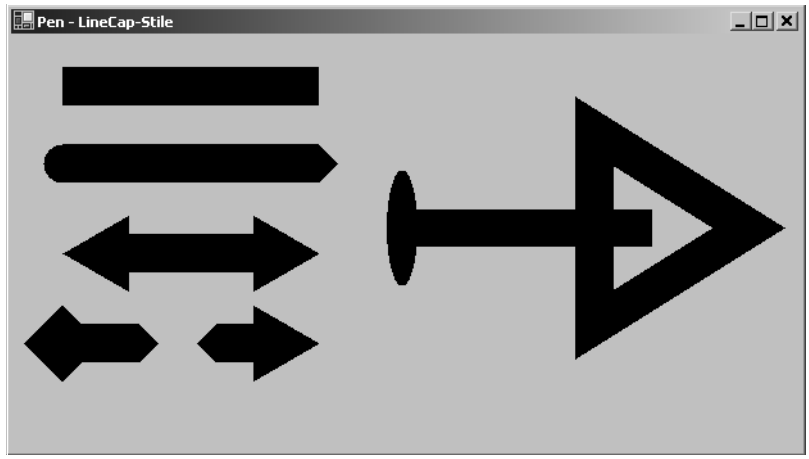


Abbildung 14.15: Linienenden – links vordefinierte, rechts selbst definierte. Alle Linien haben dieselbe Länge. Die Linie links unten ist gestrichelt und verwendet für alle drei Typen von Linienenden unterschiedliche Stile.

Pinsel – von Brush abstammende Klassen

Als abstrakte Klasse kann `Brush` selbst nicht instanziiert werden. Sie dient vielmehr, wie in Abbildung 14.12 dargestellt, als Basis für die fünf Pinselklassen `SolidBrush`, `HatchBrush`, `TextureBrush`, `LinearGradientBrush` und `PathGradientBrush`, die ihrerseits versiegelt sind. (Aus gutem Grund, denn auch in .NET ist nicht alles aus Gold, was glänzt. Diese Klassen sind reine Hüllen für entsprechende Aufrufe der Windows-API.)

Aufgabe von Pinselobjekten ist das Füllen von Füllfiguren, die als geometrische Figuren oder als Grafikpfade in Form von `GraphicsPath`-Objekten definiert sind. Grafikoperationen des Typs `FillXxx()` erwarten deshalb grundsätzlich auch ein `Brush`-Objekt, das den Vorgang des Füllens selbst beschreibt. Schriftzeichen und Linien sind nichts anderes als Grafikpfade und können daher auch mit einem Pinselobjekt gezeichnet – besser: unterlegt – werden . Die `Pen`-Klasse stellt einen eigenen Konstruktor zur Verfügung, der ein `Brush`-Objekt erwartet.

Um mit Pinseln unbeschwert zu arbeiten, sollten Sie die folgende `using`-Direktive ergänzen:

```
using System.Drawing.Drawing2D;
```

Die nachfolgenden Abschnitte geben einen Überblick über die verschiedenen Pinselarten und deren qualitative Eigenschaften, ohne auf die Fülle der Ausprägungen sowie auf weiterführende Besonderheiten im Einzelnen einzugehen.

Brushes-Klasse

Wenn Sie nichts weiter als einen gewöhnlichen Pinsel mit einer von 141 benannten Farben etwa für die Textausgabe oder das Zeichnen von Hintergründen benötigen, stellt die Klasse `Brushes` alles Nötige zur Verfügung.

```
Brush b = Brushes.Black;
```

Wie bereits im Zusammenhang mit der `Pen`-Klasse beklagt, spart diese Pseudo-Aufzählung (es sind ja statische Eigenschaften und nicht `enum`-Werte) die Systemfarben aus. Für Ausgaben, die nicht in den rein grafischen Bereich fallen, sondern etwas mit der Gestaltung der Benutzeroberfläche zu tun haben, empfiehlt sich wärmstens der Einsatz von Systemfarben, da sie automatisch auf das für das jeweilige System geltende Farbschema abgestimmt sind (exakter: nichts weiter als Spezialwerte für die Systemfarben darstellen) und bei Beachtung der zugedachten Rollen einen guten Kontrast sowie eine gute Lesbarkeit der Ausgabe garantieren.

Einfarbig – die SolidBrush-Klasse

Die Objekte der Klasse `SolidBrush` sind die einfachsten Pinsel. Sie sind einfarbig und lassen nicht nur mit jeder beliebigen ARGB-Farbe (auf Wunsch mit Transparenz), sondern insbesondere auch mit den Systemfarben konstruieren. Es ist üblich, der Objektvariablen gleich den Datentyp `Brush` zu geben, da Grafikoperationen diesen Basistyp generell auch erwarten.

Die Standardpinsel für das Zeichnen von Hintergründen und Text in Formularen wären:

```
Brush brushBkg = new SolidBrush(SystemColors.Window);
Brush brushFg = new SolidBrush(SystemColors.WindowText);
```

Ein Pinsel für eine »rosarote Brille« wäre:

```
SolidBrush brushRosa = new SolidBrush(Color.FromArgb(30, 140, 0, 20));
```

Trägt die Objektvariable den eigentlichen Datentyp – hier also `SolidBrush` – lässt sich die Farbzuordnung eines Pinsels jederzeit umdefinieren. Wie wäre es mit einem Blaufilter?

```
brushRosa.Color = Color.FromArgb(80, 0, 10, 120);
```

Schraffuren – die HatchBrush-Klasse

Die Pinsel der Klasse `HatchBrush` verkörpern zweifarbige Schraffuren, die mit einer Vorder- und einer Hintergrundfarbe gezeichnet werden. Über die `HatchStyles`-Aufzählung stehen 56 vordefinierte Schraffurstile zur Verfügung, von denen einige nichts weiter als unterschiedliche Benennungen für den gleichen Stil sind. Die Definition eigener Schraffurstile ist (ausnahmsweise einmal) nicht möglich. Verwenden Sie statt dessen Texturpinsel.

Neben den sich selbst erschließenden Eigenschaften `ForegroundColor` und `BackgroundColor` spielt noch die Eigenschaft `RenderingOrigin` des jeweiligen Grafikkontexts eine gewisse Rolle. Sie bestimmt, an welchem Punkt das an

sich unendliche Schraffurmuster beginnt und garantiert so, dass es zu keinen unliebsamen Übergängen zwischen ansonsten verträglichen Schraffuren kommt, die über unterschiedliche Grafikoperationen ausgegeben werden.

Das folgende Codebeispiel aus dem Projekt *HatchBrush-Demo* demonstriert, wie einfach der Umgang mit Schraffurpinseln im Grunde genommen ist. Das Programm zeichnet schlicht alle Schraffuren als Kästchen in den Clientbereich des Fensters und zeigt die Namen der Schraffurstile über eine `ToolTip`-Komponente an (Abbildung 14.16).

Der Code, der in seiner Struktur an das Codebeispiel aus dem Abschnitt »Zusammenstellen der benannten Farbenräume« auf Seite 490 angelehnt ist (und dort auch ausführlicher diskutiert wird), zeigt das Prinzip, wie man Aufzählungen der Art `HatchStyles` dem Benutzer per `ToolTip`-Komponente mundgerecht präsentiert. Leider sucht man dieses wirklich praktische Feature im EIGENSCHAFTEN-Fenster des Designers vergeblich, obwohl es die Lücke zwischen den vordefinierten Bezeichnern und der an sich praktischen WYSIWYG-Darstellung schließen würde. So aber hat die Darstellung im Designer oft keinen Bezug zu den Bezeichnern der `enum`-Werte.

Den vorgestellten Code in ein vom Besitzer gezeichnetes (*owner draw*) Menü, `DropDown`-Menü oder in eigene Dialoge zu integrieren, dürfte kein Problem mehr sein:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  // Alle Hatchstyles aufzählen
  Array arr = System.Enum.GetValues(typeof(HatchStyle));
  int hatchesInARow = (int) Math.Ceiling(Math.Sqrt(arr.Length));
  float cellWidth = (float)ClientRectangle.Width / hatchesInARow;
  int hatchesInAColumn =
    (int) Math.Ceiling((double)arr.Length/hatchesInARow);
  float cellHeight = (float)ClientRectangle.Height / hatchesInAColumn;

  for(int i = 0; i < arr.Length; i++)
  { // jeden Pinsel in einer Zelle ausgeben
    HatchBrush hb =
      new HatchBrush((HatchStyle) arr.GetValue(i), Color.White);
    float x = (i % hatchesInARow) * cellWidth;
    float y = (i / hatchesInARow) * cellHeight;
    e.Graphics.FillRectangle(hb, x,y, cellWidth, cellHeight);
  }
}

private void Form1_MouseMove(object sender, MouseEventArgs e)
{
  toolTip1.ShowAlways = true;
  Array arr = System.Enum.GetNames(typeof(HatchStyle));
  int hatchesInARow = (int) Math.Ceiling(Math.Sqrt(arr.Length));
  int x = (int) ((float)e.X/ClientRectangle.Width * hatchesInARow ) ;
```

```
  int y = (int) ((float)e.Y/ClientRectangle.Height *
    Math.Ceiling((double)arr.Length/hatchesInARow ));
  int i = y * hatchesInARow  + x;
  if (i < arr.Length)
    this.toolTip1.SetToolTip(this, (string) arr.GetValue(i));
  else
    this.toolTip1.SetToolTip(this, "");
}
```

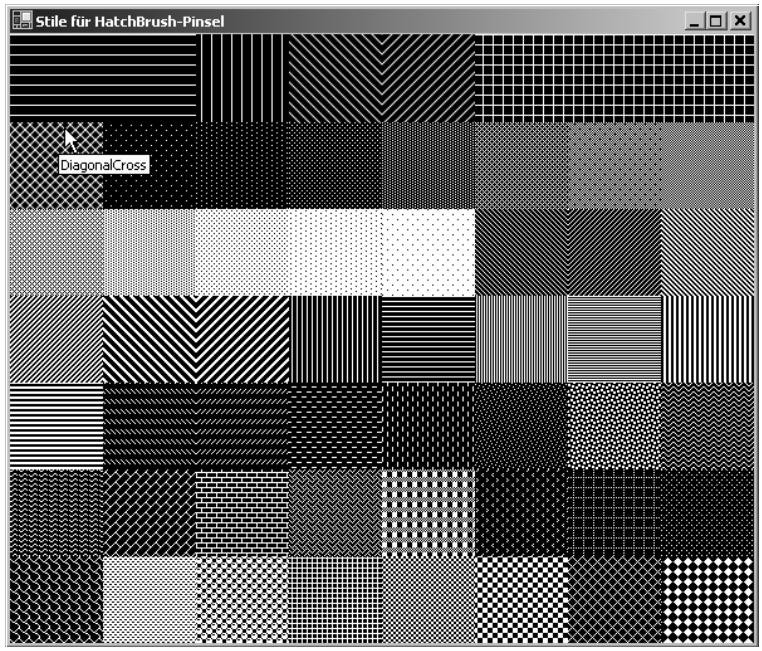


Abbildung 14.16: Das Programm *HatchBrush-Demo* gibt einen Überblick über die für `HatchBrush`-Objekte vordefinierten Schraffuren.

Texturen –die TextureBrush-Klasse

Texturpinsel sind Pinsel, die eine Grafik mit einer Bitmap ausmalen – für gewöhnlich als Kachel und im Endlosmodus. Die Kachel kann dabei schlicht wiederholt, aber auch vertikal und/oder horizontal gekippt werden.

Die Klasse `TextureBrush` stellt acht Konstruktoren bereit, denen gemeinsam ist, dass Sie ein `Image`-Objekt für die Textur erwarten. Dieses Bild kann später jederzeit über die `Image`-Eigenschaft abgerufen, jedoch nicht mehr neu gesetzt werden. Sechs Überladungen erwarten einen `Rectangle`- bzw. `RectangleF`-Wert, der den gewünschten Ausschnitt des Bildes (in Pixeln) ausdrückt. Sie müssen also nicht für jede Textur eine eigene Bitmap bereitstellen, sondern können beliebig viele Texturen in einer Bitmap zusammenfassen und dann immer nur den Ausschnitt angeben, den Sie tatsächlich für die Textur verwenden wollen. Die folgende Codezeile zeigt, wie Sie eine Bitmap einlesen, die als Ressource in der Exe-Datei gespeichert ist (ausführlichere Informationen über die Vorgehensweise finden Sie im Abschnitt »Cursorformen als Ressourcen einlesen« auf Seite 740):

```
Image image = new Bitmap(typeof(Form1), "SchwanKachelHalb.JPG");
```

Weiterhin können Sie optional einen Wert für die `WrapMode`-Eigenschaft angeben, der ausdrückt, wie der Pinsel die Textur wiederholt. Vorgabewert ist `Tile`, weitere Werte sind `TileFlipX`, `TileFlipY`, `TileFlipXY`. Der von der `WrapMode`-Aufzählung noch angebotene Wert `Clamp` zeigte bei Versuchen keine Wirkung.

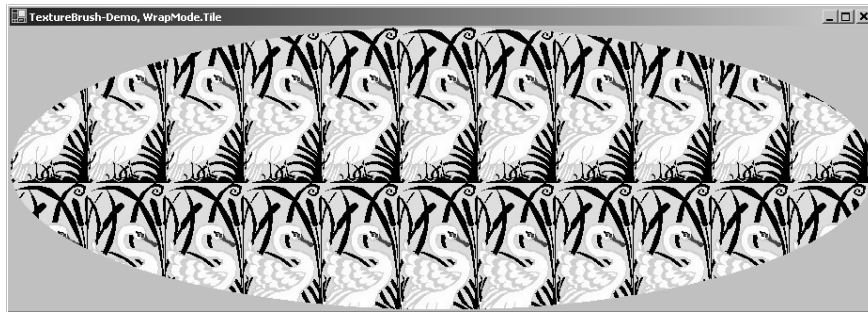
Für spezielle Anwendungen besteht die Möglichkeit, an der Textur eines fertigen `TextureBrush`-Objekts im Nachhinein Farbmanipulationen vorzunehmen. Die `TextureBrush`-Klasse stellt dafür extra zwei Konstruktoren bereit, die ein `System.Drawing.Imaging.ImageAttributes`-Objekt als Plattform für solche Operationen entgegennehmen.

Die `Transform`-Eigenschaft des Typs `Matrix` verleiht dem `TextureBrush`-Objekt seine eigentliche Flexibilität. Das Objekt stellt in diesem Zusammenhang die üblichen Methoden – `RotateTransform()`, `ScaleTransform()`, `TranslateTransform()` – bereit, über die sich die Textur drehen, verschieben und skalieren lässt, und zwar unabhängig von der globalen Transformation, die ja erst später auf die gesamte Ausgabefläche angewendet wird.

Abbildung 14.18 ist das Ergebnis der folgenden Codezeilen:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  TextureBrush tb = new TextureBrush(image);
  e.Graphics.FillEllipse(tb, ClientRectangle);
}
```

Abbildung 14.17: Die Kachel für die Textur zeigt einen Schwan.



Für Abbildung 14.18 wurde nur der Wiederholungsmodus geändert:

```
tb = new TextureBrush(image,WrapMode.TileFlipXY);
```

Und für Abbildung 14.19 kam noch eine Drehung der Textur um 90° hinzu:

```
tb.RotateTransform(90);
e.Graphics.FillEllipse(tb, ClientRectangle);
```

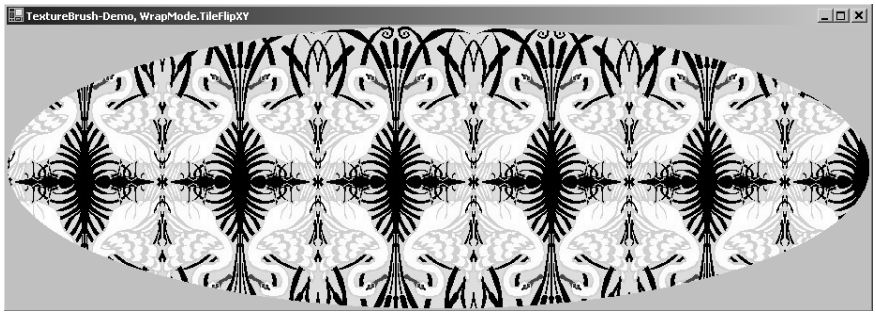


Abbildung 14.18: Angrenzende Kacheln werden horizontal bzw. vertikal gekippt.

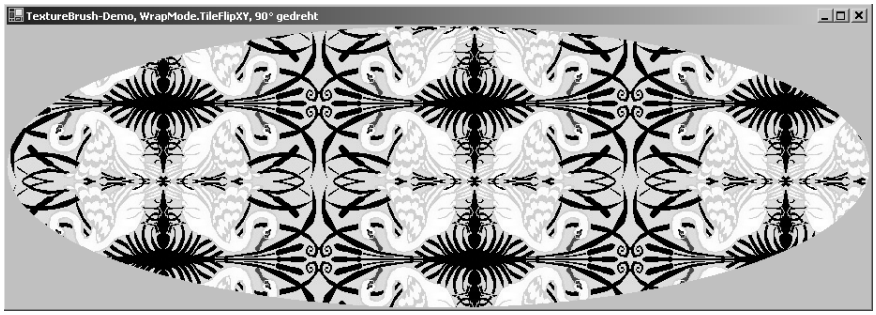


Abbildung 14.19: Die Textur um 90° gedreht

Lineare Farbverläufe – die LinearGradientBrush-Klasse

`LinearGradientBrush`-Objekte repräsentieren eine kontinuierliche Farbänderung von einer Ausgangsfarbe zu einer Zielfarbe innerhalb eines Rechtecks. Nachdem ein solcher Verlauf, wie der Name bereits sagt, eindimensional ist, arbeiten `LinearGradientBrush`-Objekte zusätzlich noch mit einer Winkelinformation für die Richtung des Verlaufs. Von der Wirkung her zeichnet ein `LinearGradientBrush`-Objekt direkt aneinandergrenzende parallele Linien, die senkrecht zur Richtung des Farbverlaufs weisen. Die Linien an sich sind einfarbig, aber von Linie zu Linie ergibt sich ein leichter Farbunterschied, der über die kürzeste »Strecke« zwischen Ausgangs- und Zielfarbe im vierdimensionalen ARGB-Farbraum berechnet wird.

Über die `WrapMode`-Eigenschaft können Sie – wie gewohnt – festlegen, wie der Pinsel den Farbverlauf als Kachel wiederholt und/oder spiegelt.

Der folgende Codeauszug zeigt, wie Sie einen »Lichteffekt« über ein natürliches Bild zeichnen, indem Sie einen Farbverlauf, von einer transparenten zu einer deckenden Farbe (hier: Weiß) hin ausgeben. Der Farbverlauf ist zudem gespiegelt und auf die halbe Bildbreite berechnet:

```
BackgroundImage = new Bitmap("SonnenuntergangInLentas.jpg");
...
Rectangle rect = ClientRectangle;
rect.Width /= 2;
rect.Height /= 2;
```

```
LinearGradientBrush lgp = new LinearGradientBrush(rect,
  Color.FromArgb(00, 255, 255, 255),
  Color.FromArgb(255, 255, 255, 255),0f);
lgp.WrapMode = WrapMode.TileFlipXY;
e.Graphics.FillRectangle(lgp,ClientRectangle);
```



Abbildung 14.20: »Sonnenuntergang in Lentas« mit künstlichem Lichteffekt als Farbverlauf ins Transparente

Komplexe Farbverläufe – PathGradientBrush

Was die Klasse `LinearGradientBrush` im Eindimensionalen und mit zwei Farben macht, überträgt die Klasse `PathGradientBrush` auf die Fläche – also ins Zweidimensionale. Sie zeichnet Farbverläufe von einem zentralen Punkt aus, der über die `CenterPoint`-Eigenschaft definiert wird, zu beliebig vielen peripheren Punkten, wobei jedem dieser Punkte eine beliebige Farbe aus dem ARGB-Farbraum zugeordnet werden kann. Neben den bereits im Zusammenhang mit der Klasse `LinearGradientBrush` vorgestellten Eigenschaften `Transform` und `WrapMode`, sieht die Klasse noch verschiedene Möglichkeiten für die Farbüberblendung vor, von denen ein Normalsterblicher allerdings eher kaum Gebrauch machen wird.

Bei Konstruktion eines `PathGradientBrush`-Objekts übergeben Sie entweder einen Graphikpfad oder besser ein `Point`-Array, um die peripheren Punkte zu beschreiben.

Damit überhaupt etwas passiert, müssen Sie im Mindestfall Werte für die Eigenschaften `CenterPoint` und `CenterColor` setzen. Die Abbildungen 14.21 bis 14.23 zeigen, wie der Farbverlauf als Linearkombination verschiedener Punkt-zu-Punkt-Farbverläufe berechnet wird, und wie Sie den zentralen Punkt prinzipiell setzen können – als Mittelpunkt, als Eckpunkt und als

Außenpunkt. Die Minimalkonfiguration sind zwei periphere Punkte, die mit dem zentralen Punkt nicht auf einer Gerade liegen dürfen, weil sich sonst keine Fläche ergibt. Damit der Farbverlauf flächendeckend wird, müssen die peripheren Punkte – gegebenenfalls zusammen mit dem zentralen Punkt – ein Rechteck bilden. Wenn Sie andere Geometrien verwenden, verstärkt sich der Kaleidoskopeffekt, da der Pinsel dann nicht überall zeichnet, sondern nur innerhalb des definierten Polygons: Die Kachel wird also so berechnet, dass das Polygon darin gerade Platz hat.

Um die Farben zu definieren, stellen Sie ein `Color`-Array zusammen und ordnen dieses der `SurroundColors`-Eigenschaft zu. Das Array darf nicht mehr Elemente haben als das bei der Konstruktion angegebene `Point`-Array – weniger sind aber zulässig. Die Indizes der beiden Arrays korrespondieren, und bei ungleicher Elementanzahl gilt die letzte Farbe für alle überzähligen Punkte.

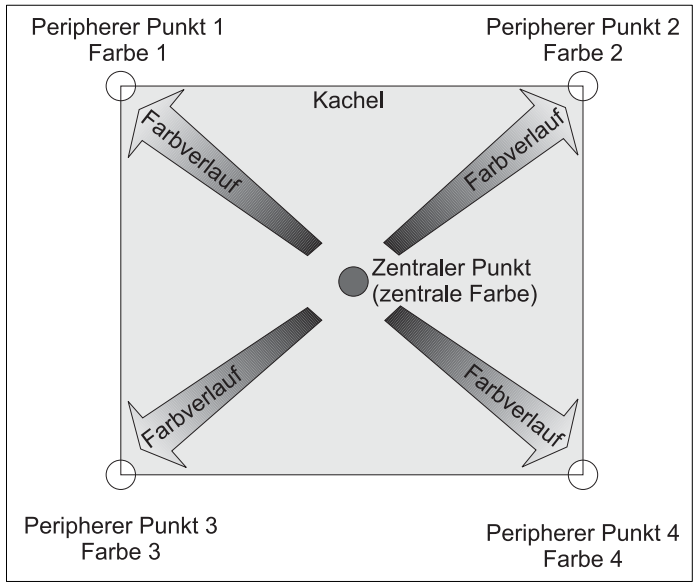


Abbildung 14.21: Normale Situation, der Farbverlauf definiert eine Kachel

Die folgende `Paint`-Routine aus dem Projekt *Farbverlauf* zeigt, wie der komplexe Farbverlauf aus Abbildung 14.24 erzeugt wird.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  Rectangle rect = ClientRectangle;
  rect.Width /= 2;
  rect.Height /= 2;
  Point[] p = new Point[] {
                            new Point(rect.Width/2, 0),
                            new Point(rect.Width, 0),
                            new Point(rect.Width, rect.Height),
                            new Point(0, rect.Height)
                          };
```

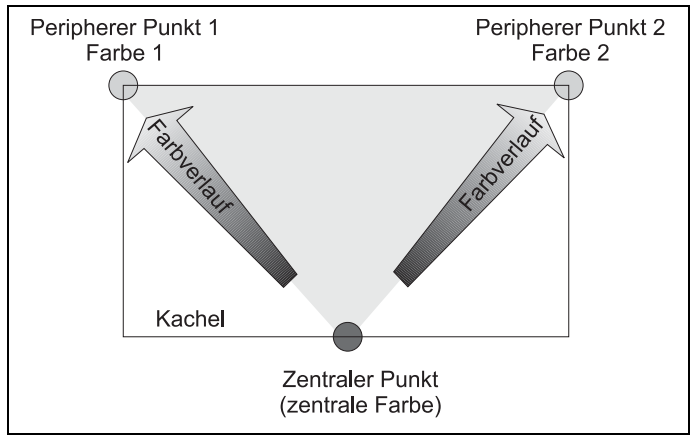


Abbildung 14.22: Minimalsituation, der Farbverlauf wird durch ein Dreieck definiert (und auch nur als Dreieck gezeichnet)

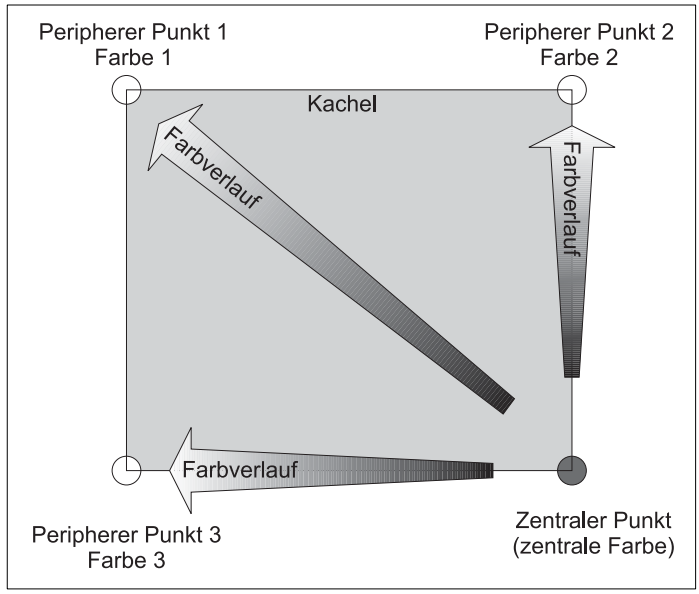


Abbildung 14.23: Asymmetrische Kachel – der Farbverlauf eignet sich gut für die Spiegelung in X/Y-Richtung

```
  Color[] c = new Color[] {
                            Color.Violet,
                            Color.DarkBlue,
                            Color.White
                           };

  PathGradientBrush pgp = new PathGradientBrush(p, WrapMode.TileFlipXY);
  pgp.CenterPoint = new Point(rect.Width/2, rect.Height/2);
  pgp.SurroundColors = c;
  pgp.CenterColor = Color.Firebrick;
  e.Graphics.FillRectangle(pgp,ClientRectangle);
}
```

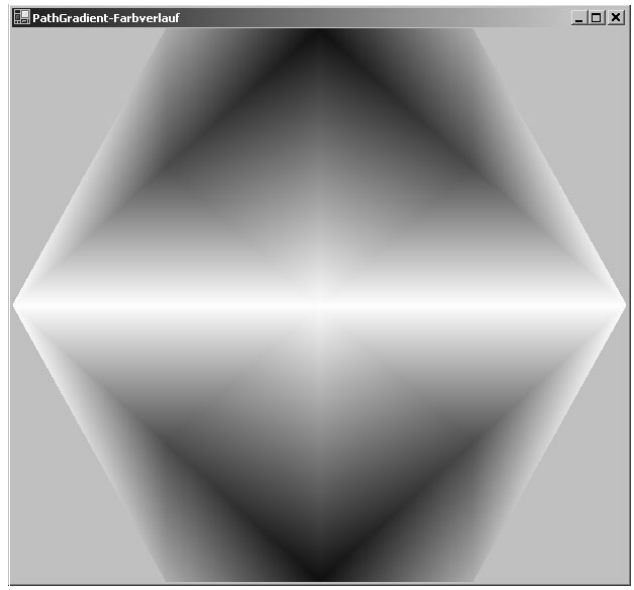


Abbildung 14.24: Farbverlauf über vier Punkte von einem zentralen Punkt aus – in Graustufen nur halb so beeindruckend

14.4 Zeichnen

Wie bereits zu Anfang dieses Kapitels ausgeführt, zeichnet ein Formular oder ein Steuerelement normalerweise nicht dann, wenn es gerade »Lust dazu hat«, sondern auf Aufforderung, indem es (oder der Besitzer) eine Behandlungsmethode für das `Paint`-Ereignis registriert oder die `OnPaint`-Methode der Basisklasse überschreibt und dadurch den automatisierten Aufruf durch den Nachrichtenbehandlungsmechanismus des Systems bedient. Auch wenn es ein paar Objekte bzw. Klassen gibt, die eigene `DrawXxx()`-Methoden zum Zeichnen zur Verfügung stellen (beispielsweise `Cursor`- und `ImageList`-Objekte oder die Klasse `ControlPaint`, die gleich einen ganzen Satz statischer Methoden zum Zeichnen von Steuerelementbestandteilen anbietet), werden Sie im Normalfall mit der wirklich üppigen Ausstattung des Grafikkontexts bestens auskommen.

14.4.1 Grafikkontext – Graphics-Klasse

Sie erhalten den Grafikkontext im Allgemeinen in Form der Eigenschaft `Graphics` des im `PaintEventArgs`-Parameter (vgl. Seite 462) eines `Paint`- oder `DrawItem`-Ereignisses übergebenen Ereignisobjekts. Obwohl die Eigenschaft genauso benannt ist wie der Datentyp, handelt es sich hier tatsächlich um ein Objekt. Auch wenn es vom Prinzip her möglich wäre, dieses Objekt in einem Datenfeld für den weiteren Gebrauch in anderen Methoden der Formular- bzw. Steuerelementklasse abzulegen, sollten Sie das nicht tun, denn der Grafikkontext kann sich »unterwegs« (beispielsweise durch Änderung

der Anzeigeeinstellungen über die Systemsteuerung) ändern und sollte als flüchtig betrachtet werden. Nicht zuletzt deshalb wird er bei jeder `Paint`-Behandlung erneut bereitgestellt.

Wie schon zu Anfang dieses Kapitels angesprochen, können Sie für Ausgaben außerhalb der `Paint`-Behandlung auch über die (von `Control` vererbte) Methode `CreateGraphics()` jederzeit einen aktuellen Grafikkontext anfordern. Standardmäßig wird dies aber nur für Animationen im Bereich der `MouseMove`-Behandlung eingesetzt, wie der Abschnitt »Codebeispiel – Gummiband« ab Seite 421 demonstriert (vgl. auch das Codebeispiel zu diesem Abschnitt).

Die `Graphics`-Klasse verkapselt die Ausgabefläche und definiert eine eigene, recht umfangreiche Schnittstelle für die Interaktion damit. In der Ausstattung eines `Graphics`-Objekts finden Sie:

- einen kompletten Satz an Zeichenoperationen (`FillXxx()` und `DrawXxx()`)
- eine Seitentransformation (`PageUnit`, `PageScale`)
- eine globale Transformation (`Transform`) mit entsprechenden Methoden (`XxxTransform()`)
- Informationen über das Ausgabegerät (in verschiedenen nicht systematisierten Eigenschaften, beispielsweise `DpiX` und `DpiY`)
- verschiedene Eigenschaften für die Parametrisierung der Ausgabe mit Blick auf Rasteroperationen, darunter: `CompositingMode`, `CompositingQuality`, `InterpolationMode`, `PixelOffsetMode`, `RenderingOrigin`, `SmoothingMode`
- *Push*- und *Pop*-Äquivalente für die stackorientierte Zustandskonservierung: `BeginContainer()` und `EndContainer()`
- ein paar weitere Hilfsmittel, etwa für die Berechnung von Stringausgaben auf der Basis eines Fonts.

Da Sie das Buch sicher nicht an dieser Stelle aufgeschlagen haben und wohl auch schon zig `Paint`-Routinen gesehen und wahrscheinlich auch selbst geschrieben haben, dürfte Ihnen ein schneller Überblick über alle Grafikoperationen dienlicher sein als weitere Beispiele. Sie sind in Tabelle 14.9 zusammengestellt. Beispiele und zusätzliche Ausführungen zum Thema Kurven finden Sie im Abschnitt »Grafikpfade – GraphicsPath« ab Seite 474, Details über das Zeichnen und Formatieren von Strings im übernächsten Abschnitt. Die Anwendung der anderen Methoden erschließt sich weitgehend über die Parametersätze der unterschiedlichen Überladungen.

Tabelle 14.9: Die Grafikoperationen des `Graphics`-Objekts

Methode	Beschreibung
`void DrawArc()`	Zeichnet einen Ellipsenbogen. Die Beschreibung der Ellipse erfolgt als `Rectangle`- bzw. `RectangleF`-Struktur (oder koordinatenweise), so dass die Hauptachsen der Ellipse immer parallel zu den Koordinatenachsen stehen. Erwartet zusätzlich einen Startwinkel (in Grad, im Uhrzeigersinn, mit 3 Uhr als 0°, negative Winkel sind erlaubt) und einen Öffnungswinkel (in Grad, negative Winkel sind erlaubt).
`void DrawBezier()`	Zeichnet eine Bezierkurve. Eine Bezierkurve besteht aus einem Startpunkt, zwei Stützpunkten und einem Endpunkt. Die Paare (Startpunkt, erster Stützpunkt) und (zweiter Stützpunkt, Endpunkt) definieren zwei Tangenten, die den Austrittswinkel der Kurve aus dem Startpunkt und den Eintrittswinkel der Kurve in den Endpunkt vorgeben. Die Abstände der Stützpunkte vom Start- bzw. Endpunkt definieren den Krümmungsradius für den Austritt bzw. Eintritt.
`void DrawBeziers()`	Zeichnet mehrere aneinandergrenzende Bezierkurven en bloc als zusammenhängende Figur. Die Bereitstellung der Parameter erfolgt über ein `Point`- bzw. `PointF`-Array. Für die erste Kurve sind 4 Punkte anzugeben, für jede weitere Kurve zusätzlich 3 Punkte. Der Endpunkt der ersten Kurve ist Startpunkt der zweiten Kurve usw. Die Kurvenkrümmungen haben keinen Bezug zueinander, so dass (mathematisch nicht differenzierbare) Ecken entstehen, wenn die Stützpunkte nicht bewusst so gewählt sind, dass die Tangenten übereinstimmen.
`void DrawCloseCurve()`	Zeichnet einen kardinalen Spline als geschlossene Kurve. Die Methode erwartet die Kurvenpunkte als Array und einen für die gesamte Kurve geltenden Glättungsfaktor. Der Vorgabewert für den Glättungsfaktor ist `0.5f` und ergibt eine Minimalkurve mit einer ausgewogenen Glättung über alle Punkte hinweg (vgl. Abbildung 14.3 auf Seite 478, links und Mitte). Ein kleinerer Glättungsfaktor bewirkt kleinere Kurvenkrümmungen an den Punkten, ein größerer sorgt für eine stärkere Krümmung. Glättungsfaktoren über `1.0f` führen zu einer Überbiegung der Kurve und etablieren weitere Schnörkel. Ist der Wert 0, erhalten Sie einen Linienzug – wie mit `AddLines()` bzw. `AddPolygon()`. Tatsächlich sind auch negative Glättungen erlaubt, wobei die Frage offen bleibt, ob sie sinnvoll verwendbar sind.

Tabelle 14.9: Die Grafikoperationen des `Graphics`-Objekts (Forts.)

Methode	Beschreibung
`void DrawCurve()`	Zeichnet einen kardinalen Spline als offene Kurve: erster und letzter Kurvenpunkt werden nicht durch eine Linie verbunden, und die Krümmung läuft hier in eine Gerade aus.
`void DrawEllipse()`	Zeichnet eine vollständige Ellipse. Die Beschreibung der Ellipse erfolgt als Rechteck (`Rectangle`, `RectangleF` oder koordinatenweise), weshalb die Hauptachsen der Ellipse immer parallel zu den Koordinatenachsen stehen.
`void DrawIcon()`	Zeichnet das angegebene `Icon`-Objekt so, dass es das angegebene Rechteck ausfüllt – gegebenenfalls erfolgt also eine Skalierung.
`void DrawIconUnstretched()`	Zeichnet das angegebene `Icon`-Objekt ohne weitere Skalierung links oben in das angegebene Rechteck – gegebenenfalls wird das Bild beschnitten.
`void DrawImage()`	Zeichnet das angegebene `Image`-Objekt oder einen Ausschnitt davon in das angegebene Rechteck – gegebenenfalls erfolgt eine Skalierung.
`void DrawImageUnscaled()`	Zeichnet das angegebene `Image`-Objekt oder einen Ausschnitt davon an die angegebene Position bzw. in das angegebene Rechteck – dabei erfolgt keine Skalierung, und überstehende Bereiche werden abgeschnitten. Diese Methode ist im Allgemeinen schneller als `DrawImage()`, kann aber keine Ausschnitte des Bildes zeichnen (vgl. aber: `ClipRectangle`-Eigenschaft).
`void DrawLine()`	Zeichnet eine Linie zwischen den angegebenen Punkten.
`void DrawLines()`	Zeichnet einen Linienzug. Erwartet die Eckpunkte in einem `Point-` oder `PointF`-Array.
`void DrawPath()`	Zeichnet einen Grafikpfad (vgl. `GraphicsPath`-Klasse).
`void DrawPie()`	Zeichnet ein Tortenstück als Ellipsenausschnitt mit Linie vom Mittelpunkt zum Start und Endpunkt des Bogens. Die Beschreibung der Ellipse erfolgt als Rechteck (`Rectangle`, `RectangleF` oder koordinatenweise), weshalb die Hauptachsen der Ellipse immer parallel zu den Koordinatenachsen stehen. Erwartet zusätzlich einen Startwinkel (in Grad, im Uhrzeigersinn, mit 3 Uhr als 0°, negative Winkel sind erlaubt) und einen Öffnungswinkel (in Grad, im Uhrzeigersinn, negative Winkel sind erlaubt).

Tabelle 14.9: Die Grafikoperationen des `Graphics`-Objekts (Forts.)

Methode	Beschreibung
`void DrawPolygon()`	Zeichnet ein Polygon als geschlossene Linienfigur, die durch ein `Point`- bzw. `PointF`-Array beschrieben wird.
`void DrawRectangle()`	Zeichnet ein Rechteck, das wahlweise in Koordinatenform (Koordinaten des linken oberen Punkts; Abmessungen) oder als `Rectangle`-Wert anzugeben ist. `RectangleF`-Werte verarbeitet die Methode nicht. Verwenden Sie dafür die Schwestermethode `DrawRectangles()`.
`void DrawRectangles()`	Zeichnet eine Folge von Rechtecken, die wahlweise durch ein `Rectangle`- oder ein `RectangleF`-Array angeben werden.
`void DrawString()`	Zeichnet einen String mit der angegebenen Ausrichtung und Schrift in das angegeben Rechteck.
`void FillClosedCurve()`	Füllt einen kardinalen Spline als geschlossene Kurve – zur Beschreibung vgl. `DrawClosedCurve()`.
`void FillEllipse()`	Füllt eine Ellipse mit dem angegebenen Pinsel – zur Beschreibung vgl. `DrawEllipse()`.
`void FillPath()`	Füllt einen Grafikpfad (vgl. `GraphicsPath`-Klasse) mit dem angegebenen Pinsel. Enthält der Grafikpfad offene Figuren, werden diese durch einen `AddLine()`-Aufruf unter Angabe des Start- und Endpunkts geschlossen.
`void FillPie()`	Füllt ein Tortenstück (Ellipsenausschnitt) mit dem angegebenen Pinsel – zur Beschreibung vgl. `DrawPie()`.
`void FillPolygon()`	Füllt einen geschlossenen Linienzug mit dem angegebenen Pinsel – zur Beschreibung vgl. `DrawPolygon()`.
`void FillRectangle()`	Füllt ein Rechteck mit dem angegebenen Pinsel – zur Beschreibung vgl. `DrawRectangle()`.
`void FillRectangles()`	Füllt eine Folge von Rechtecken mit dem angegebenen Pinsel – zur Beschreibung vgl. `DrawRectangles()`.
`void FillRegion()`	Füllt das angegebene `Region`-Objekt mit dem angegebenen Pinsel.

Pufferung

Ein Thema, das im Zusammenhang mit der Grafikausgabe immer wieder auftaucht, ist die PUFFERUNG. Wenn Sie von Visual Basic kommen, werden Sie es sich vielleicht angewöhnt haben, den *AutoDraw*-Modus einzuschalten und an beliebiger Stelle im Code zu zeichnen, ohne eigens eine *Paint*-Routine dafür einzusetzen. Dass dies funktioniert, liegt daran, dass die Basisimplementierung des Formulars eine Bitmap als Puffer für alle Ausgabeoperationen zur Verfügung stellt und diese im Rahmen einer standardmäßigen Behandlung des *Paint*-Ereignisses in den Grafikkontext kopiert.

Die Klasse `Form` unterstützt von sich aus keinen Mechanismus, der in diese Richtung weist. Alles, was gezeichnet wird, verschluckt der Grafikkontext auf Nimmerwiedersehen. Und beim nächsten `Paint`-Ereignis geht alles wieder von vorne los – zumindest für den inzwischen für ungültig erklärten Ausgabebereich, der durch die `ClipRectangle`-Eigenschaft beschrieben ist.

Je nach Art des anfallenden Zeichenaufwands kann dies tragbar sein oder auch nicht. Je aufwändiger das Rendering, desto schneller ist der Punkt erreicht, ab dem es sich auszahlt, die Ausgabe zunächst als Bitmap zu zeichnen. Wann immer eine `Paint`-Behandlung anfällt, lässt sich diese Bitmap dann stückweise mit `DrawImage()` oder auch en bloc mit `DrawImageUnscaled()` in den Grafikontext übertragen. Der Aufwand dafür hält sich in Grenzen:

```
public Form1()
{
  buffer =
    new Bitmap(ClientRectangle.Size.Width, ClientRectangle.Size.Height);
}
...
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
  // Zeichnet nur den notwendigen Ausschnitt!
  e.Graphics.DrawImage(buffer,e.ClipRectangle, e.ClipRectangle,
    GraphicsUnit.Pixel);
}
```

Im Konstruktor stellt sich natürlich die Frage, wie groß die gepufferte Zeichenfläche sein soll und was passiert, wenn sich die Fenstergröße ändert. Im Allgemeinen wird es reichen, die Bitmap exakt in den Clientbereich einzupassen und im Zuge der `Resize`-Behandlung gegebenenfalls eine größere anzulegen sowie die bereits bestehenden Inhalte per `DrawImageUnscaled()` umzukopieren. Die Logik für das Nachzeichnen des neuen Bereichs sieht vom Prinzip her genauso aus wie eine »reguläre« `Paint`-Methode, die sich auf die von Windows als ungültig erklärten Bereiche beschränkt und lässt sich sogar noch etwas effektiver gestalten. Da `ClipRectangle` den Datentyp `Rectangle` trägt (und nicht `Region`) beschreibt diese Eigenschaft das kleinste Rechteck, das alle ungültigen Bereiche einschließt. Bei Vergrößerungen des Formulars in *x*- und *y*-Richtung fällt deshalb immer der gesamte Clientbe-

reich an. Die eigene Logik kann dagegen so ausgelegt sein, dass sie in diesem Fall nur die beiden neuen Streifen zeichnet:

```
private void Form1_Resize(object sender, System.EventArgs e)
{
  Bitmap oldBuffer = buffer;
  buffer = new Bitmap(ClientRectangle.Size.Width,
    ClientRectangle.Size.Height);
  if (buffer.Width > oldBuffer.Width)
    RenderMyObjects(new Rectangle(oldBuffer.Width, 0,   // vert. Streifen
      buffer.Width-oldBuffer.Width, buffer.Height));
  if (buffer.Height > oldBuffer.Height)
    RenderMyObjects(new Rectangle(oldBuffer.Height, 0,   // hor. Streifen
      buffer.Height-oldBuffer.Height, buffer.Width));
}
```

Bei extrem aufwändigen Operationen können Sie die Logik dahingehend erweitern, dass der überlappende Bereich nur einmal gezeichnet wird.

Das Zeichnen in die Bitmap erfolgt per Grafikkontext, den Sie über die statische Methode `Graphics.FromImage()` anfordern. Der Rest ist Schema F:

```
private void RenderMyObjects(Rectangle r)
{
  Graphics g = Graphics.FromImage(buffer);
  g.FillRectangle(new SolidBrush(BackColor), r);
  ...   // Rendering
}
```

Ereignisbehandlung am Leben erhalten

In der zu Anfang dieses Teils genannten Liste der obersten Designziele für die Entwicklung von Windows-Anwendungen findet sich auch ein Punkt, der auf die Reaktivität der Benutzerstelle abzielt. Der Benutzer ist es gewohnt, dass eine Anwendung jederzeit auf Befehle reagiert und zeitaufwändige Operationen sich auch jederzeit abbrechen lassen. Als Faustregel gilt, dass eine Anwendung spätestens alle Zehntelsekunde »in die Warteschlange gucken sollte«, ob nicht irgendwelche wichtigen Ereignisse anstehen. Operationen, die länger dauern, wie etwa das Warten auf eine Verbindung oder eine Rendering-Operation, sollten erstens durch Setzen einer anderen Mauszeigerform (Sanduhr) und zweitens durch eine Fortschrittsanzeige oder einen inkrementellen Bildaufbau angezeigt werden.

Wie Sie wahrscheinlich wissen, unterbrechen sich Behandlungsmethoden für Ereignisse nicht gegenseitig, sondern werden immer schön der Reihe nach ausgeführt. Gerade beim Rendering, das beliebig zeitintensiv sein kann, müssen Sie daher in jedem Fall Maßnahmen treffen, um die Ereignisbehandlung dennoch am Leben zu erhalten, wenn eine Verletzung der Zehntelsekundenregel durch eine oder mehrere Ereignisroutinen zu erwarten ist.

Eine Möglichkeit, die sicher nicht zu verachten ist, besteht darin, aufwändige Operationen in einen oder mehrere Hintergrund-Threads auszulagern, während sich der im Vordergrund laufende Haupt-Thread um die Benutzerschnittstelle kümmert (vgl. hierzu Teil 5).

In den meisten Fällen ist dieses Geschütz aber nicht nur zu grob, sondern gleicht eher dem Motto: »Warum einfach, wenn es umständlich auch geht«.

Nach einem derartigen Statement werden Sie vermutlich nicht sonderlich überrascht sein, dass sich all dies auch mit einem einzigen periodischen Aufruf der statischen Methode `Application.DoEvents()` erledigen lässt. Der dahinter stehende Mechanismus ist zwar verblüffend einfach, er hat aber bestimmte Konsequenzen für das Codedesign. `DoEvents()` gibt die Kontrolle temporär so lange an das System zurück, bis alle inzwischen aufgelaufenen Ereignisse behandelt sind. Danach geht es weiter – bis zum nächsten `DoEvents()`-Aufruf.

Natürlich hat diese Einfachheit auch ihren Preis:

1. Es kann zum rekursiven Aufruf der unterbrochenen Behandlung kommen, wenn eine eingeschobene Behandlung ihrerseits – mittelbar oder unmittelbar – den Aufruf der Routine mit dem `DoEvents()`-Aufruf veranlasst. Rekursion per se ist kein Problem. Sie kann jedoch zum Problem werden, wenn sie systematisch etwa über eine zu schnelle Folge von `Tick`-Ereignissen angestoßen wird. Das Ergebnis ist dann früher oder später ein Stack-Überlauf.
2. Eingeschobene Behandlungsmethoden können in Datenfeldern gespeicherte globale Informationen verändern und so einen Seiteneffekt auf die unterbrochene Behandlung haben. Die Methode sollte solche Informationen also entweder lokal puffern oder auf solche Veränderungen mit einem Abbruch reagieren – und gegebenenfalls einen erneuten Aufruf ihrer selbst (beispielsweise durch `Invalidate()`) veranlassen.

Codebeispiel – eine Reise durch die Mandelbrotmenge

Das folgende aus dem Projekt *Mandelbrot* stammende Codebeispiel demonstriert verschiedene Techniken, die alle mehr oder weniger mit der Grafikausgabe im Zusammenhang stehen:

- Aufwändiges Rendering mit Einsatz der `DoEvents()`-Methode
- Pufferung der Grafikausgabe in Bitmaps
- Umgang mit `Bitmap`-Objekten
- Gummibandanimation zur Bereichsauswahl per Maus (Zoomfunktion)
- Konservierung mehrerer Puffer in einem Stack des Typs `ArrayList`

Der Code, der hinter den Abbildungen 14.25 bis 14.27 steht, integriert den Algorithmus für die Berechnung der Mandelbrotmenge in die Grafikausgabe. Das Programm zeichnet beim Start einen fest vorgegebenen Ausschnitt aus dieser Menge. Um in das Bild hineinzuzoomen, markiert man den gewünschten Ausschnitt mit der Maus unter Verwendung der linken Maustaste. Das geht auch wiederholt, bis die Rechengenauigkeit des Datentyps `float` für die Bereichsauswahl erschöpft ist und der Ausschnitt aufgrund von Rundungsfehlern nicht mehr exakt berechnet wird. Ein Klick mit der rechten Maustaste bringt dagegen das jeweils zuvor berechnete Bild wieder zum Vorschein (Herauszoomen).

Einer Fahrt durch die Mandelbrotmenge steht damit also nichts im Wege – nun ja, allzu rasant dürfte sie nicht werden, denn das Laufzeitverhalten des Programms ist angesichts des anfallenden Aufwands für das Rendering nicht gerade als vorbildlich zu bezeichnen. Je nach Größe des Formulars müssen Sie selbst auf einem halbwegs vernünftigen System schon einige Sekunden je Bild veranschlagen. Andererseits: Wäre das Beispiel nicht mit der Zehntelsekundenregel im Clinch, würden Sie es nicht an dieser Stelle finden.

Hintergrund

Die von dem Mathematiker Benoit B. Mandelbrot im Zuge seiner Beschäftigung mit so genannten »Julia-Mengen« Anfang der siebziger Jahre entdeckte Mandelbrotmenge zählt heute noch zu den augenscheinlichsten Ergebnissen der Chaosforschung und dürfte der immer noch jungen Wissenschaft ordentlich auf die Beine geholfen haben. Mandelbrot benutzte damals einen IBM-Computer, um die Menge erstmals zu visualisieren, und war umgehend von ihrer »bizarren Schönheit« bezaubert.

Der mathematische Kern der Mandelbrotmenge ist wahrlich einfach. Man untersucht eine parametrisierte Folge im komplexen Zahlenraum daraufhin, ob sie konvergiert.

$$\begin{aligned} f_0 &= 0 \\ f_{n+1} &= f_n^2 + z \end{aligned}$$

wobei der Parameter z aus der komplexen Ebene gewählt wird. Für die Visualisierung der Menge wird das Verhalten der Folge f_n untersucht, indem man z systematisch in einem bestimmten (um die Null herum gelegenen) Bereich variiert und für jedes z bis zu `cMaxIterat` (also endlich viele) Folgenglieder berechnet. Übersteigt der Betrag eines Folgenglieds einen hinreichend groß gewählten Grenzwert `cGrenze`, bricht man die Iteration ab und wertet die zu c gehörige Folge als divergent – z gehört dann *nicht* zur Mandelbrotmenge. Die Anzahl der Iterationen, die bis zum Abbruch nötig waren, lässt sich als Farbwert für den Punkt z in der komplexen Ebene interpretieren. Sind dagegen für ein z alle Folgenglieder vom Betrag her klei-

ner gleich dem Grenzwert, wertet man die zu *z* gehörige Folge als konvergent und *z* als Element der Mandelbrotmenge. Der Farbwert dafür ist eine fest definierte Farbe, beispielsweise `BackColor`.

Der Code

Allem voran eine Feststellung: Das Programm kommt um eine Pufferung der Ausgabe in einer Bitmap nicht umhin, denn es würde viel zuviel Laufzeit verschlingen, die Bildpunkte mangels einer *DrawPoint()*-Methode per `DrawRectangle()` oder `DrawLine()` direkt in den Grafikkontext zeichnen zu wollen. Das Mittel der Wahl ist ganz klar die `SetPixel()`-Methode. Als Bestandteil der Ausstattung des `Bitmap`-Objekts unterliegt sie zwar nicht der Wirkung der globalen Transformation eines nachträglich übergestülpten Grafikkontexts, doch damit lässt sich wesentlich besser leben, als umgekehrt.

Wie zu erwarten, nimmt die `Paint`-Behandlung keinerlei Rendering vor, sondern kopiert nur das gepufferte Bild. Das allerdings nicht blindlings: Da das Programm eine inkrementelle Ausgabe bereits gerenderter Bereiche vornimmt, wird die `ClipRectangle`-Eigenschaft des Ereignisobjekts beachtet. Aufforderungen zum Rendern eines neuen Ausschnitts finden sich erwartungsgemäß in den Methoden `Form1_Load()`, `Form1_MouseUp()` und `Form1_Resize()`.

Rendering

Die für das Rendering zuständige Methode heißt `Mandelbrot()`. Sie erledigt alles, was mit dem Zeichnen in den Puffer zusammenhängt und pflegt dazu eine Art eigene Koordinatensystemtransformation, die eine Umrechnung von Clientkoordinaten in den jeweils berechneten Zoombereich ermöglicht. Die Konvention für den Aufruf dieser Methode ist, dass der übergebene `Rectangle`-Wert einen Ausschnitt des aktuellen Clientbereichs beschreibt – genau den liefert die Mausauswahl per »Gummiband«. Diesen Ausschnitt rechnet sie unter Beachtung des jeweils zuvor gerenderten Ausschnitts (Zoomstufe) so um, dass der Benutzer beim nächsten Rendering nur den per Gummiband ausgewählten Bereich zu sehen bekommt. Damit die Methode beim nächsten Aufruf weiß, welchen Ausschnitt sie zuletzt gerendert hat, speichert sie ihn in dem Datenfeld `section` der Formularklasse.

Das Rendering geschieht spaltenweise. Um die Reaktivität der Benutzerschnittstelle aufrechtzuerhalten, führt die Methode nach jeder Spalte einen `DoEvents()`-Aufruf durch. Und damit der Benutzer zuschauen kann, wie sich das Bild aufbaut, wird nach jeder zehnten Spalte `Invalidate()` aufgerufen, wobei die Wirkung jeweils auf den inzwischen neu gerenderten Bereich beschränkt bleibt.

Flackern unterbinden

Tatsächlich löst der `Invalidate()`-Aufruf nicht nur eine WM_PAINT-Nachricht aus, die zum Aufruf von `OnPaint()` führt, sondern vorher noch eine

WM_ERASEBKGND-Nachricht. Diese Nachricht tritt nicht als Ereignis zum Vorschein, weshalb Sie im Designer auch keine Behandlungsroutine dafür installieren können. Sie führt aber zum Aufruf der von `Control` vererbten virtuellen `OnPaintBackground()`-Methode, deren standardmäßige Implementierung dafür sorgt, dass der neu zu zeichnende Bereich erst einmal mit Hintergrundfarbe gefüllt wird. Da die errechnete Bitmap auf jeden Fall den gesamten Clientbereich des Formulars ausfüllt, ist das im vorliegenden Fall nicht nur überflüssig, sondern auch störend, weil das sinnlose Füllen mit Hintergrundfarbe als flackernder Streifen, der über das Bild huscht (oder kriecht – je nach Rechner), sichtbar wird. Abhilfe schafft ein Überschreiben der virtuellen Methode durch eine eigene Variante mit leerem Körper.

Wiedereintritt

Wenn Sie in dem Datenfeld `section` ein Problem für den allfälligen `DoEvents()`-Aufruf riechen, liegen Sie richtig. Der Wiedereintritt in die Methode, der immer dann erfolgt, wenn der Benutzer nicht die Geduld hat, abzuwarten, bis ein Bild fertig gezeichnet wurde, verändert den Wert dieses Datenfelds und etabliert auch einen neuen Puffer. Die alte Bitmap wird in einem kleinen Stack zusammen mit dem Ausschnitt (`section`), den er repräsentiert, für den Rückweg (rechte Maustaste) konserviert. Es wäre natürlich problemlos, die Werte dieser beiden globalen Variablen lokal zwischenzuspeichern und den Puffer in jedem Fall zu Ende zu zeichnen. Die vorliegende Implementierung hat also den Mut zur Lücke und den Makel, dass der ungeduldige Benutzer auf dem »Rückweg« halbfertig gezeichnete Puffer vorfinden kann. Im Gegenzug rendert sie alle Bilder mit voller Geschwindigkeit, weil sie noch ausstehende Zeichenvorgänge umgehend abbricht. Als Zustandsvariable dafür fungiert das noch weit vor dem Wiedereintritt bereits in `Form1_MouseDown()` gesetzte `bool`-Datenfeld `stopPainting`.

Der Stack besteht aus einer `ArrayList`-Instanz namens `stack`, die als Datenfeld vereinbart ist. Die für das Hineinzoomen verantwortliche Methode `Form1_MouseUp()` erledigt die *Push*-Operation: Dazu legt sie in dieser Liste, jeweils als erstes Element, den Wert des Datenfelds `section` und die Referenz für den alten Puffer ab, bevor sie demselben Datenfeld einen Klon des alten Puffers zuordnet. Die Pop-Operation entspricht dem Herauszoomen. Sie findet in `Form1_MouseDown()` als Reaktion auf einen rechten Mausklick statt und setzt das jeweils zuletzt gespeicherte Wertpaar wieder in diese Variablen ein. Der ursprüngliche Pufferinhalt kann so einfach per `Invalidate()` wiederhergestellt werden, ohne dass ein erneutes Rendering erforderlich wäre.

Hier der gesamte Code:

```
public class Form1 : System.Windows.Forms.Form
{
  private ArrayList stack ;            // speichert Auschnitte und Puffer
  private Rectangle RubberBand ;       // Gummiband
```

```
private bool stopPainting = false;   // Flag, um Zeichnen zu stoppen
Bitmap buffer;                       // Puffer für Grafikausgabe
RectangleF section;                  // Ausschnitt
public Form1()
{
  InitializeComponent();
  stack = new ArrayList();
  buffer = new Bitmap(ClientRectangle.Size.Width,
    ClientRectangle.Size.Height);
}
static void Main()
{
  Application.Run(new Form1());
}

protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
  // Zeichnet nur den notwendigen Ausschnitt!
  e.Graphics.DrawImage(buffer,e.ClipRectangle,
    e.ClipRectangle, GraphicsUnit.Pixel);
}

// Zeichnet die Mandelbrotmenge
private void Mandelbrot(Rectangle sz)
{
  const double cGrenze = 10000;
  const double cMaxIterat = 256;
  const float StartX = -2.1f;
  const float StartDX = 2.8f;
  if (section.IsEmpty)  // erster Aufruf: initalisieren
  {
    section.X = StartX;
    section.Width = StartDX;
    section.Height = (float) sz.Height / sz.Width * StartDX;
    section.Y = -section.Height/2;
  }
  else                  // neuen Ausschnitt berechnen
  {
    section.X += section.Width  * ((float)sz.X / ClientRectangle.Width);
    section.Y += section.Height * ((float)sz.Y / ClientRectangle.Height);
    section.Width *= (float)sz.Width/ClientRectangle.Width;
    section.Height*= (float)sz.Height/ClientRectangle.Height;
  }

  // weißen Hintergrund zeichnen
  Graphics g = Graphics.FromImage(buffer);
  g.FillRectangle(Brushes.White, ClientRectangle);
  stopPainting = false;

  // Schrittweiten
  double stepx = section.Width/(ClientRectangle.Width-1);
  double stepy = section.Height/(ClientRectangle.Height-1);
  int x = 0;    // Schrittzähler
  int y = 0;
  // Algorithmus zur Errechnung der Mandelbrotmenge
  // für den Ausschnitt section
```

```
    for(double zr = section.X; zr < section.X + section.Width; zr += stepx)
    {
      double r1 , re, im;
      for(double zi=section.Y; zi<section.Y + section.Height; zi += stepy)
      {
        re = 0;                          // Realteil initialisieren
        im = 0;                          // Imaginärteil initialisieren
        for(int it = 0; it < cMaxIterat; it ++)      // Iteration für Punkt
        {
          r1 = re * re - im * im + zr;
          im = 2 * re * im + zi;
          re = r1;
          if (Math.Abs(re) > cGrenze || Math.Abs(im) > cGrenze)
          {
            Color c = Color.FromArgb((int)(0xff000000 + (it * 2222)));
            buffer.SetPixel(x, y, c);    // Punkt zeichnen
            break;
          }
        }
        y++;
      }
      y = 0; x++;
      Application.DoEvents();   // Ereignisbehandlung nach jeder Zeile
      if (stopPainting)         // Abbruch?
        return;
      if((x+1) % 10 == 0)       // neu berechneten Streifen ausgeben
        Invalidate(new Rectangle((x-11), 0, 11,ClientRectangle.Height));
    }
    // Fertig zeichnen
    Invalidate(new Rectangle((x-11),0, 11,ClientRectangle.Height));
  }

  private void Form1_Resize(object sender, System.EventArgs e)
  {
    if (ClientRectangle.IsEmpty) // Windows lässt grüßen
      return;
    // Puffer anpassen
    buffer = new Bitmap(ClientRectangle.Size.Width,
      ClientRectangle.Size.Height);
    Mandelbrot(ClientRectangle);   // neu zeichnen
  }

  private void Form1_Load(object sender, System.EventArgs e)
  {
    Show();
    Mandelbrot(ClientRectangle);   // Grundausschnitt zeichnen
  }

  // Linke Maustaste initialisiert Gummiband
  // Rechte Maustaste holt vorigen Ausschnitt zurück
  private void Form1_MouseDown(object sender, MouseEventArgs e)
  {
    if ( (e.Button & MouseButtons.Left) > 0)  // linke Maustaste?
    {
      RubberBand.Location = Control.MousePosition;
    }
```

```
  // rechte Maustaste? Dann vorigen Ausschnitt zurückholen.
  if ( stack.Count > 1 && e.Button == MouseButtons.Right)
  {
    stopPainting = true;
    buffer = (Bitmap) stack[0];    // Pop
    section = (RectangleF) stack[1];
    stack.RemoveRange(0,2);
    Invalidate();
  }
}

// Gummiband aufziehen
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
  if ( (e.Button & MouseButtons.Left) > 0)  // linke Maustaste?
  {
    if(this.ClientRectangle.Contains(new Point(e.X, e.Y)))
    {
      // Gummiband löschen
      ControlPaint.DrawReversibleFrame(RubberBand, this.BackColor,
        FrameStyle.Dashed);
      RubberBand.Size =
        (Size) Control.MousePosition - (Size) RubberBand.Location;
      // Seitenverhältnis wahren
      float ratio = (float)ClientRectangle.Height/ClientRectangle.Width;
      RubberBand.Height = (int)(RubberBand.Width * ratio);
      // Gummiband neu zeichnen
      ControlPaint.DrawReversibleFrame(RubberBand, this.BackColor,
        FrameStyle.Dashed);
    }
  }
}

// Löscht Gummiband und speichert den aktuellen Ausschnitt sowie
// Pufferinhalt auf dem Stack
private void Form1_MouseUp(object sender, MouseEventArgs e)
{
  if ( (e.Button & MouseButtons.Left) > 0)  // linke Maustaste?
  {
    // Gummiband löschen
    ControlPaint.DrawReversibleFrame(RubberBand, this.BackColor,
      FrameStyle.Dashed);
    Rectangle NewSection = RubberBand;
    RubberBand.Size = new Size(0, 0);
    NewSection.Location = PointToClient(NewSection.Location);
    if(NewSection.Width > ClientRectangle.Width / 20) // nicht zu klein?
    {
      // Aktuelle Bitmap und Ausschnitt auf Stack
      stack.Insert(0, section);  // Push
      stack.Insert(0, buffer);
      buffer = (Bitmap)buffer.Clone();
      // Fraktal neu zeichnen
      Mandelbrot(NewSection);
    }
  }
}
```

```
  protected override void OnPaintBackground(PaintEventArgs pevent)
  {
    // nichts
  }
}
```

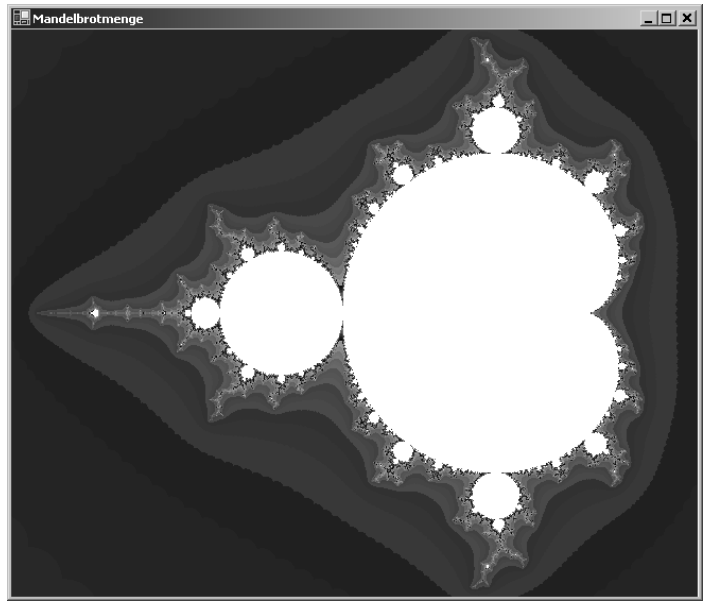


Abbildung 14.25: Ausgangsbildschirm, auch als »Apfelmännchen« bekannt

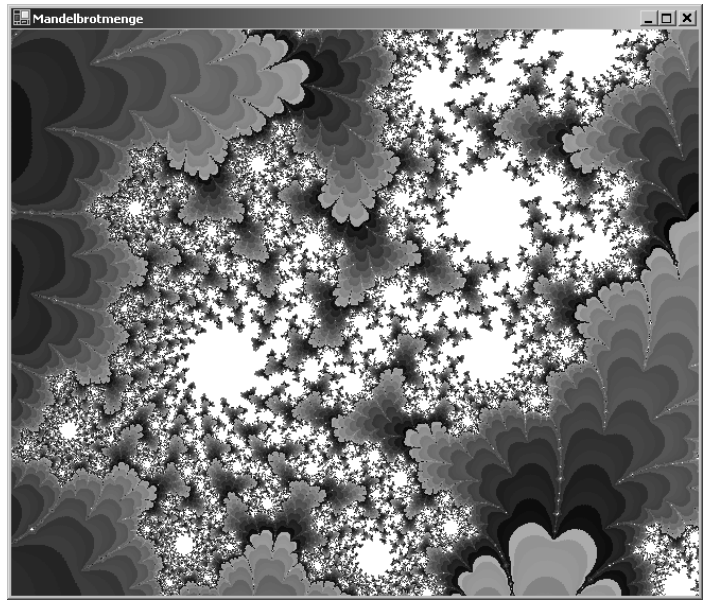


Abbildung 14.26: Höhere Zoomstufe

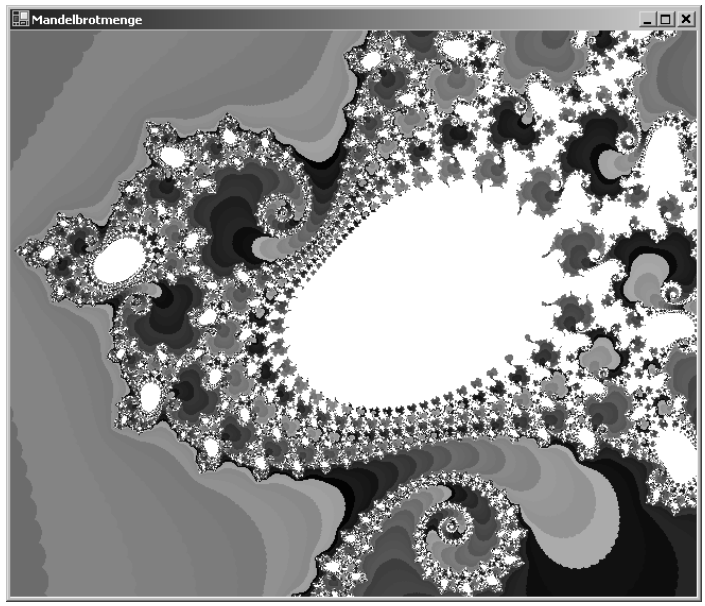


Abbildung 14.27: Unendliche Weiten ...

Ausblicke

Nicht nur die Mandelbrotmenge hat unendliche Welten zu bieten – Windows macht hier durchaus manchmal Konkurrenz. Beim Herumspielen mit dem Programm werden Ihnen vielleicht einige weitere Unvollkommenheiten aufgefallen sein, bei denen es nicht nur »Mut zur Lücke« zeigt, sondern auch gewissen Launen von Windows, bzw. der .NET-Implementierung ausgeliefert ist:

- Wenn Sie in den Desktop-Eigenschaften FENSTERINHALTE BEIM ZIEHEN ANZEIGEN angekreuzt haben, verhält sich das Programm bei Vergrößerungen seines Fensters per Maus etwas eigenartig: Es berechnet die Bitmap erst für ein um einige Pixel vergrößertes Fenster komplett neu – und dann ein zweites Mal für die tatsächlich gewünschte Fenstergröße.
- Das Programm verhält sich bei einem Maximieren per Doppelklick auf die Titelleiste anders als bei einem Klick auf das MAXIMIEREN Symbol.

Einem dieser zahlreichen Sonderfälle im Zusammenhang mit dem Zeichnen beim Vergrößern begegnet das Programm, indem es eine nicht gerade sinnvoll anmutende Abfrage in der `Resize`-Behandlung vornimmt.

```
private void Form1_Resize(object sender, System.EventArgs e)
{
  if (ClientRectangle.IsEmpty)
    return;
```

Tatsächlich kann es vorkommen, dass ein Formular beim Schließen noch ein `Resize`-Ereignis sieht, wenn es zuvor in den maximierten Zustand versetzt wurde – dummerweise mit den Abmessungen (0,0), was der `Bitmap`-Konstruktor übelnimmt.

Auch ist das Programm von der Logik her schlicht nicht darauf ausgelegt, dass der Benutzer Größenveränderungen des Fensters vornimmt. Erstens gibt es Verzerrungen, da der Ausschnitt nicht angepasst wird, und zweitens macht es sich auf dem »Rückweg« nicht gut, wenn zwischendurch die Formularabmessungen geändert wurden.

Konsequent wäre es daher, Größenänderungen per `FormBorderStyle`-Eigenschaft schlicht zu verbieten, da sich die Sache mit der Vergrößerung als erstaunlich zäh erweist. Will man ihr vor dem Hintergrund des nicht gerade trivialen Ansatzes mit der `DoEvents()`-Behandlung eine Chance geben, führen die entsprechenden Manipulationen unerfreulich tief in die Windows-Spezifikation hinein.

Da Ihnen bei ernsthafter Arbeit mit .NET aber unter Garantie ähnliche »Kleinigkeiten« begegnen werden, seien mögliche Lösungsansätze hier wenigstens kurz erwähnt, auch wenn sie absolut nichts mit Grafik zu tun haben.

Mit Größenveränderungen des Fensters und überflüssigen bzw. unerwünschten Neuberechnungen sieht es da schon anders aus: Wenn FENSTERINHALTE BEIM ZIEHEN ANZEIGEN ausgewählt ist, dann sendet Windows dem Formular bei *jeder* Mausbewegung im Rahmen einer Ziehaktion des Fensterrandes die Nachricht WM_SIZE, die ihrerseits letztlich zur Behandlungsroutine `Form_Resize()` führt. Wieso *Mandelbrot* dann nicht in Dutzenden von Neuberechnungen förmlich ertrinkt, ist schnell geklärt: Der Aufruf der Neuberechnung geschieht aus der Behandlungsroutine für WM_SIZE heraus – und so lange sich eine Anwendung aus dieser Behandlungsroutine nicht direkt zurückmeldet (also per Rücksprung, der nicht nur hier als »Operation abgeschlossen« zu werten ist), verzichtet das System sinnvollerweise darauf, ihre Warteschlange mit weiteren WM_SIZE-Nachrichten zuzustopfen.

Womit sich die Frage erhebt: Gibt es eine Benachrichtigung des Systems im Sinne von »Größenveränderung des Fensters per Ziehen mit der Maus abgeschlossen«? Die gute Nachricht ist: Es gibt sie, und sie trägt den Namen WM_EXITSIZEMOVE. Die (erste) schlechte Nachricht ist, dass die Klasse `Control` keine Behandlungsroutine dafür bereitstellt, weshalb man hier sozusagen an die Wurzel der Nachrichtenverarbeitung gehen muss:

```
bool Resizing = false;

const int WM_EXITSIZEMOVE = 0x0232;
```

```
protected override void WndProc(ref Message m)
{
  if (m.Msg == WM_EXITSIZEMOVE)
  {
    Resizing = false;
    Bitmap oldBuffer = buffer;
    buffer = new Bitmap(ClientRectangle.Size.Width,
         ClientRectangle.Size.Height);
    Mandelbrot(ClientRectangle);
  }
  // Basisklassenvariante
  base.WndProc(ref m);
}

private void Form1_Resize(object sender, System.EventArgs e)
{
  Resizing = true;
}
```

Die Methode `WndProc()` stellt bei `Control` und davon abgeleiteten Klassen bildlich gesprochen das zentrale Tor für Windows-Nachrichten dar, die konsequent hier auch fast im Rohformat erscheinen: Die Klasse `Message` ist dabei nicht mehr als ein `struct` mit einigen Eigenschaften. Die überschriebene Variante sorgt dafür, dass die in der vorigen Version von *Mandelbrot* im Rahmen von `Form1_Resize()` ausgeführte Neuberechnung nur noch einmal geschieht – und zwar bei Abschluss der Größenveränderung des Formulars.

Den Aufruf von `base.WndProc` hätte man im übrigen auch ohne weiteres in einen `else`-Zweig stellen können, weil die Formularklasse WM_EXITSIZEMOVE nicht auswertet. Für alle anderen Nachrichten ist er ein absolutes Muss – ohne ihn wäre das Formular sozusagen toter Code.

Ganz fertig ist das Programm noch nicht, weil das Unterdrücken *jeder* Löschaktion auf die Hintergrundfarbe durch `OnPaintBackground()` ein klein wenig zu viel des Guten darstellt – vor allem, wenn man das Neuzeichnen des Fensterinhalts auf den Abschluss einer Vergrößerung des Fensters verschiebt. Wie Abbildung 14.28 zeigt, verlässt sich Windows darauf, dass bei einer solchen Aktion die sozusagen als Zwischenergebnis gezeichneten Fensterränder sofort wieder mit der Hintergrundfarbe überdeckt werden.

Erledigen lässt sich das mit der bereits in den zuvor gezeigten Code kommentarlos eingeschmuggelten Variablen `Resizing`, die von `Form1_Resize` auf `true` und erst beim Abschluss (WM_EXITSIZEMOVE) wieder auf `false` gesetzt wird:

```
protected override void OnPaintBackground(
  System.Windows.Forms.PaintEventArgs pevent)
{
  if (Resizing)  // während Größenveränderungen
   base.OnPaintBackground(pevent);
}
```

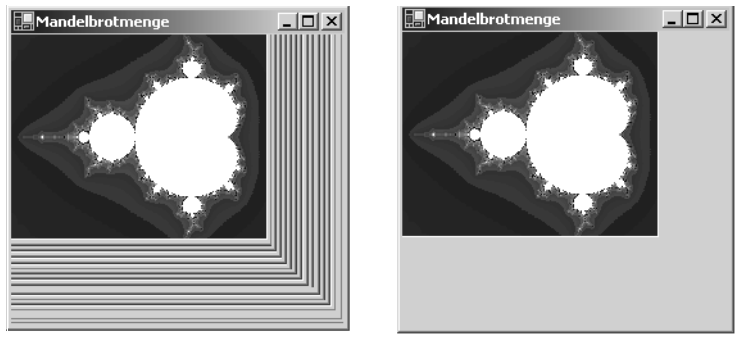


Abbildung 14.28: Bei Vergrößerungen des Formulars mit aufgeschobener Zeichenaktion darf das Löschen auf die Hintergrundfarbe nicht unterdrückt werden.

Dass *Mandelbrot* nun immer noch kein perfektes Programm darstellt, haben Sie gleich mehreren Lässlichkeiten der Windows-Entwickler zu verdanken: Bei Umschaltungen zwischen Vollbild und eingestellter Größe (Doppelklick in der Titelleiste oder Klick auf die entsprechende Box in der Titelleiste) versendet das System kein WM_EXITSIZEMOVE – und die Standardnachricht für Größenänderungen (WM_SIZE) sollte der Dokumentation zufolge Unterscheidungen zwischen diesen Umschaltungen und einer »normalen« Größenveränderung erlauben, tut es aber nicht. (`msg.WParam` bekommt zwar für die Umschaltung auf maximale Fenstergröße einen eindeutigen Wert (2), bei der Rückschaltung auf die eingestellte Größe und Größenveränderungen über die Maus steht dort dagegen in beiden Fällen der Wert 0.)

Eine denkbare Lösung dieses Dilemmas, die zur Zeit in den Internet-Newsgroups diskutiert wird und erfreulicherweise nicht noch weiter in die Abgründe von Windows führt, ist ein Timer, der bei `Form1_Resize` aktiviert wird und nach Ablauf der Wartezeit für die Neuberechnung sorgt. Die Reaktion von WM_EXITSIZEMOVE ist auch in diesem Fall noch sinnvoll und könnte so aussehen:

```
if (m.Msg == WM_EXITSIZEMOVE)
{
  ResizeTimer.Enabled = false;
  // neue Bitmap anlegen, Berechnung
  DoResizeAction();
}
```

Die hier postulierte Methode `DoResizeAction()` wird dann die Operationen ausführen, die vorher direkt bei der Bearbeitung der Nachricht stattgefunden haben. Sie wird auch beim »regulären« Ablaufen des Timers aufgerufen. Die konkrete Implementation sei Ihnen als Fingerübung überlassen.

14.4.2 Textausgabe – DrawString()

Für die Textausgabe in einen Grafikkontext stehen Ihnen zwei Wege offen:

1. *Zeichnen einzelner Strings mit jeweils fest gewählten Schriftattributen per* `DrawString()` – die Methode erwartet ein `Font`-Objekt, eine Positionsbeschreibung in Form eines Punkts oder eines Rechtecks und eine optionale Formatierungsbeschreibung in Form eines `StringFormat`-Objekts, die sich auf die Positionierung bezieht. Ist die Positionsbeschreibung als Rechteck angegeben, nimmt die Methode einen automatischen Zeilenumbruch auf der Basis von Wortzwischenräumen vor. Ausgaben, die nicht vollständig in das Rechteck passen, werden beschnitten.
2. *Ausgabe eines Grafikpfads per* `DrawPath()` *oder* `FillPath()`*, in den zuvor per* `AddString()` *ein oder mehrere Schriftzüge eingefügt wurden* – dieser Weg ist etwas verschlungener, bietet dafür aber erheblich mehr Möglichkeiten, unterschiedliche Schriften, Schriftschnitte, Schriftgrößen, Ausrichtungen sowie Drehungen und Verzerrungen zu berücksichtigen. Im Gegensatz zum ersten Weg ist die Repräsentation der Ausgabe von der Ausgabeoperation selbst getrennt. Es wird hier ein kompaktes Objekt gezeichnet, das als solches auch nach dem Zeichenvorgang noch existiert – und somit *wiederverwendbar* ist.

Dieser Abschnitt stellt nur den ersten Weg vor. Eine ausführlichere Darstellung des zweiten Wegs samt Codebeispielen finden Sie im Zusammenhang mit der `GraphicsPath`-Klasse (vgl. den Abschnitt »Schriften«, Seite 480). Dort finden Sie insbesondere auch die typografischen Grundlagen für die Berechnung von Schriftmaßen und Zeilenabständen.

Hier die Prototypen für die sechs Überladungen der Methode `DrawString()`, die alle auf der Basis von `float` arbeiten:

```
void DrawString(string text, Font f, Brush b, float x, float y);
void DrawString(string text, Font f, Brush b, float x, float y,
  StringFormat sf);
void DrawString(string text, Font f, Brush b, PointF pos);
void DrawString(string text, Font f, Brush b, PointF pos, StringFormat sf);
void DrawString(string text, Font f, Brush b, RectangleF r);
void DrawString(string text, Font f, Brush b, RectangleF r,
  StringFormat sf);
```

Font-Klasse

Aufgabe eines `Font`-Objekts ist es, die konkrete Ausgestaltung einer Schrift zu beschreiben. Dazu kombiniert es drei Informationen, die ausschließlich über den Konstruktor gesetzt werden können:

- *Schriftart* (oder Schriftfamilie) – als Datentyp für die Beschreibung einer Schriftart fungiert die Klasse `FontFamily`. Neue Objekte dieser Klasse werden unter Angabe eines Schriftnamens wie »Arial« oder »Times New Roman« konstruiert. Eine Auflistung der auf dem jeweiligen System installierten Schriften liefert die statische Methode `FontFamily.GetFamilies()`.

- *Schriftschnitt* (kursiv und/oder fett usw.) – die Beschreibung des Schriftschnitts erfolgt als `FontStyle`-Bitvektor und ist eine Kombination der Werte `Bold`, `Italic`, `Regular`, `Strikeout`, `Underline`.
- *Schriftgröße* – in der Typografie wird die Schriftgröße in der Einheit pt (Punkt) angegeben, wobei ein Punkt eine (mittlere) Zeichenhöhe von 1/72 Zoll oder 0,352 mm ausdrückt. Dieses Maß ist auch der Vorgabewert. Bei Bedarf können Sie allerdings auch ein anderes Maß setzen – beispielsweise Bildpunkte oder Millimeter, indem Sie im Konstruktor den Parameter des Typs `GraphicsUnit` auf den entsprechenden Wert der `GraphicsUnit`-Aufzählung setzen. Tabelle 14.2 auf Seite 468 gibt einen Überblick, welche Maße zur Auswahl stehen. Eine besondere Rolle spielt der Wert `World`. Er legt fest, dass das `Font`-Objekt das Maß nach der `GraphicsUnit`-Eigenschaft des Grafikkontexts richtet. Die Schriftgröße wird dabei übrigens in pt angegeben. Umgekehrt heißt dies wiederum, dass alle anderen Maße als unabhängig von der `GraphicsUnit`-Eigenschaft des Grafikkontexts zu betrachten sind.

Der Konstruktor der `Font`-Klasse ist immerhin dreizehn Mal überladen. Die gebräuchlichsten Varianten sehen so aus:

```
Font(string fontName, float points);
Font(FontFamily fFam, float points);
Font(string fontName, FontStyle fs, float points);
Font(FontFamily fFam, FontStyle fs, float points);
Font(string fontName, FontStyle fs, float points, GraphicsUnit gu);
Font(FontFamily fFam, FontStyle fs, float points, GraphicsUnit gu);
```

Tabelle 14.10: Wichtige Eigenschaften eines `Font`-Objekts

Eigenschaft	Bedeutung
`bool Bold {get;}`	`true`, wenn der Schriftschnitt das Attribut »fett« aufweist (vgl. `Style`-Eigenschaft)
`FontFamily FontFamily {get;}`	Das `FontFamily`-Objekt, auf dem die Schrift basiert
`int Height {get;}`	Zeilenhöhe der Schrift in aktuellen Grafikeinheiten des Objekts (nicht des Grafikkontexts; vgl. auch `GetHeight()`)
`bool Italic {get;}`	`true`, wenn der Schriftschnitt das Attribut »kursiv« aufweist (vgl. `Style`-Eigenschaft)
`string Name {get;}`	Schriftname (Familienname ohne Zusätze wie »Regular«)
`float Size {get;}`	Schriftgröße, ausgedrückt in dem über die Eigenschaft `Unit` festlegten Maß
`float SizeInPoints {get; }`	Schriftgröße, ausgedrückt in der Einheit pt

Tabelle 14.10: Wichtige Eigenschaften eines `Font`-Objekts (Forts.)

Eigenschaft	Bedeutung
`bool StrikeOut {get;}`	`true`, wenn der Schriftschnitt das Attribut »Durchgestrichen« aufweist (vgl. `Style`-Eigenschaft)
`FontStyle Style {get;}`	Bitvektor, der den Schriftschnitt beschreibt
`bool Underline {get;}`	`true`, wenn der Schriftschnitt das Attribut »Unterstrichen« aufweist (vgl. `Style`-Eigenschaft)
`GraphicsUnit Unit {get;}`	`enum`-Wert, der die für das Objekt geltende Maßeinheit beschreibt. Mögliche Werte sind: `Display`, `Document`, `Inch`, `Millimeter`, `Pixel`, `Point`, `World`

Neben den in Tabelle 14.10 aufgelisteten Eigenschaften eines `Font`-Objekts sollten Sie noch die dreifach überladene Methode `GetHeight()` kennen. Die parameterlose Variante dieser Methode liefert schlicht den Wert der Eigenschaft `Height`. Die anderen beiden Varianten ermöglichen es, die Schriftgröße für die Ausgabe in einem bestimmten Grafikkontext bzw. bei einer bestimmten Geräteauflösung (in dpi) zu ermitteln.

Text platzieren – StringFormat-Objekt und StringAlignment-Aufzählung

Die diversen Varianten der Methode `DrawString()` lassen sich in zwei Kategorien einteilen: Überladungen der ersten Kategorie erwarten ein Rechteck als Positionsangabe, der Rest arbeitet mit einem Punkt. Alle Angaben beziehen sich auf das objekteigene Koordinatensystem.

Die Ausgabe der Punktvarianten ist grundsätzlich einzeilig. Die Rechteckvarianten nehmen hingegen einen automatischen Zeilenumbruch auf Wortbasis (notfalls auf Zeichenbasis) vor, wenn der Schriftzug die Breite des Rechtecks überschreitet. Das `StringFormat`-Objekt gibt – unter anderem – die Art der Ausrichtung am Punkt bzw. im Rechteck vor.

Die für die Ausrichtung zuständigen Eigenschaften dieses Objekts sind `Alignment` (horizontal) und `LineAlignment` (vertikal) und können jeweils die Werte `Near`, `Center` oder `Far` der `StringAlignment`-Aufzählung annehmen. Damit sind insgesamt neun Ausrichtungen möglich – von »links oben« (Vorgabewert für die Rechteckvarianten) über »oben mitte«, »oben rechts« usw. bis »rechts unten« (Vorgabewert für die Punktvarianten). Während die Interpretation der Ausrichtung für die Rechteckvarianten klar sein dürfte, bedürfen die Punktvarianten der Eselsbrücke, dass sich die Bezeichner der Aufzählungswerte auf den linken oberen Punkt des Schriftzugs beziehen. Abbildung 14.5 aus dem Abschnitt über Grafikpfade (Seite 481) verdeutlicht den Zusammenhang.

Weiterhin können Sie für die Rechteckvarianten über die `Trimming`-Eigenschaft des `StringFormat`-Objekts festlegen, wie der Text abgeschnitten wer-

den soll, wenn er nicht ganz in das Layout-Rechteck passt. Zur Auswahl stehen sechs unterschiedliche Verfahrensweisen, die sich durch die Konstanten der `StringTrimming`-Aufzählung ausdrücken lassen (vgl. Tabelle 14.11). Beachten Sie dabei aber, dass die Höhe des Layout-Rechtecks so gewählt sein muss, dass keine Zeile durch den unteren Rand des Layout-Rechtecks abgeschnitten wird – davor schützt die `Trimming`-Eigenschaft nämlich nicht.

Tabelle 14.11: Die Elemente der `StringTrimming`-Aufzählung

Konstante	Bedeutung
`None`	Text wird gegebenenfalls abgeschnitten
`Character`	Voreinstellung. Die Textausgabe endet gegebenenfalls mit dem letzten passenden Zeichen.
`Word`	Die Textausgabe endet gegebenenfalls mit dem letzten vollständig darstellbaren Wort (Kriterium ist ein Leerzeichen).
`EllipsisCharacter`	Die Textausgabe endet gegebenenfalls mit einem Auslassungszeichen »…« und wird nach dem letzten gerade noch passenden Zeichen abgeschnitten, den Platzbedarf für das Auslassungszeichen mit eingerechnet.
`EllipsisWord`	Die Textausgabe endet gegebenenfalls mit einem Auslassungszeichen »…«, der Text wird nach dem letzten vollständig darstellbaren Wort abgeschnitten, den Platzbedarf für das Auslassungszeichen mit eingerechnet.
`EllipsisPath`	Die Textausgabe endet immer mit den letzten zwei oder drei Zeichen, davor wird aber gegebenenfalls als Ersatz für Text, der nicht mehr passt, ein Auslassungszeichen »…« eingefügt.

Abmessungen für die Textausgabe berechen

Das wahrscheinlich wichtigste Hilfsmittel für die Textausgabe neben `DrawString()` ist die `MeasureString()`-Methode des Grafikkontexts. Sie liefert den Platzbedarf, der für die Ausgabe eines gegebenen Strings bei gegebenem `StringFormat`-Objekt und maximaler Ausgabebreite anfällt. Der über den Funktionswert gelieferte `SizeF`-Wert lässt sich zur Konstruktion eines Rechtecks verwenden, in dem der String Platz hat. Der Aufruf sieht so aus.

```
SizeF sz = e.Graphics.MeasureString(text, font, maxWidth, stringFormat);
```

Theoretisch sollte die Methode als Höhe ein Vielfaches der `Height`-Eigenschaft des angegebenen `Font`-Objekts liefern. In der Praxis tut sie dies aber nicht exakt. Wie Versuche gezeigt haben, ergibt sich je nach Punktgröße der verwendeten Schrift meist ein um einige Bildpunkte größerer Wert. Zuwei-

len aber, gerade bei Punktgrößen unter 10, kann der Wert auch kleiner als erwartet ausfallen. Dass der `Width`-Wert ebenfalls nicht exakt ist, demonstriert das Codebeispiel im nächsten Abschnitt.

Um die Zeilenanzahl eines Umbruchs aus der `Height`-Eigenschaft des Funktionswerts zu ermitteln, müsste man daher also mit der `Math.Round()`-Methode arbeiten. Wesentlich eleganter ist es allerdings, sich diesen Wert von der Methode `MeasureString()` selbst berechnen zu lassen, wobei allerdings nur eine der sieben Überladungen ergänzend zum Funktionswert noch zwei `out`-Parameter des Typs `int` zurückgibt. Der eine gibt Auskunft darüber, wie viele Zeichen des Strings in das angegebene Layout-Rechteck passen, und der andere, wie viele Zeilen.



So wichtig die Methode `MeasureString()` *ist, so wenig verlässlich ist sie. Praktische Versuche haben weiterhin ergeben, dass die Methode auch abgeschnittene Zeilen in die Zeilenanzahl und die Zeichenanzahl mit einberechnet. Sie sollten daher auf jeden Fall mit dem* `Size`*-Wert weiterarbeiten, den die Methode liefert – und falls dies ein Problem ist, den Aufruf mit einer kleineren* `Height`*-Vorgabe wiederholen.*

Weiterhin besteht das Problem, dass die Methode die letzte Zeile nicht mehr am letzten Leerzeichen umbricht, sondern einfach mitten im Wort, beim letzten, gerade noch passenden Zeichen abschneidet, weshalb die gelieferte Zeichenanzahl für die Ausgabe der letzten Zeile gegebenenfalls noch einmal nach unten korrigiert werden muss. Ein Codebeispiel für einen Seitenumbruch im Zusammenhang mit der Druckausgabe finden Sie ab Seite 688.

Codebeispiel – Blocksatz

Die Rechteckvariante der Methode `DrawString()` beherrscht zwar den Zeilenumbruch für Strings, die nicht in das Layout-Rechteck passen, jedoch nur mit Flatterrand. Einen Blocksatz sieht das `StringFormat`-Objekt nicht vor, und es findet sich auch kein anderweitiges »kostenloses« Mittel dafür. Das Codebeispiel *Blocksatz* zeigt den grundsätzlichen Weg auf, der zum Blocksatz führt und deckt dabei gleichzeitig auch die Schwächen der Methode `MeasureString()` auf.

Die Aufgabenstellung lautet:

- Das Programm soll beim Start, sowie auf einen Mausklick hin eine Textdatei einlesen und deren Inhalt im Blocksatz ausgeben. Der Dateiname wird über den Standarddialog ÖFFNEN abgefragt.
- Es dürfen keine Zeilen abgeschnitten werden und es soll ein Rand von (mindestens) einem Zehntel der Breite bzw. Höhe des Clientbereichs bleiben.
- Tabulatoren und Absätze müssen nicht berücksichtigt werden.

Der Code

Das Programm selbst ist denkbar einfach gestrickt und ist nichts weiter als ein halbwegs komfortables Testbett für die zentrale Routine `PaintJustifiedInRect()`. Neben dem `Paint`-Ereignis behandelt es die Ereignisse `Load`, `Click` und `Resize`, wobei die `Load`-Behandlung auf den für `Click` zuständigen Handler zurückgreift. Diese generiert einen Dateiauswahldialog für eine Textdatei, öffnet und liest sie und splittet ihren Inhalt gleich unter Verwendung des Leerzeichens als Trennzeichenfolge in ein `string`-Array auf. Ansonsten sorgt die `Click`- ebenso wie die `Resize`-Behandlung noch dafür, dass neu gezeichnet wird. Die `Paint`-Behandlung ruft schließlich die Methode `PaintJustifiedInRect()` unter Angabe eines `Font`-Objekts und eines passend berechneten Layout-Rechtecks auf.

```
private string text;
private string[] words;

private void Form1_Load(object sender, System.EventArgs e)
{
  Form1_Click(null, null);
}

private void Form1_Paint(object sender, PaintEventArgs e)
{
  RectangleF r = new Rectangle();
  r.X = ClientRectangle.Width/10;
  r.Y = ClientRectangle.Height/10;
  r.Width = (float)ClientRectangle.Width/10 * 8;
  r.Height = (float)ClientRectangle.Height/10 * 8;
  if (words != null)
    PaintJustifiedInRect(words, new Font("Times New Roman", 15),
    r, e.Graphics);
}

private void Form1_Click(object sender, System.EventArgs e)
{
  OpenFileDialog ofd = new OpenFileDialog();
  ofd.Filter = "Textdatei (*.txt) | *.txt";
  if( DialogResult.OK == ofd.ShowDialog())
  {
    System.IO.StreamReader sr = new System.IO.StreamReader(ofd.OpenFile());
    text=sr.ReadToEnd();
    words = text.Split(null, 1000);    // die ersten maximal 1000 Wörter
  }
  Invalidate();
}

private void Form1_Resize(object sender, System.EventArgs e)
{
  Invalidate();
}
```

Allzu spannend ist dieser Code wahrlich nicht. Umso interessanter ist dagegen die Methode `PaintJustifiedInRect()`: Sie misst Stringlängen und -höhen, berechnet Wortabstände und zeichnet schließlich auch Wort für Wort, Zeile für Zeile. Trotz seines Umfangs ist der Algorithmus leicht zu überblicken, da er nur aus einer äußeren und einer inneren Schleife besteht. Die äußere Schleife durchläuft das `string`-Array und addiert die Wortbreiten sowie die minimalen Wortabstände zeilenweise auf. Passt ein Wort nicht mehr in die angefangene Zeile, errechnet die Routine aus der aufsummierten Breite und der Anzahl der auszugebenden Leerzeichen den »exakten« Wortabstand für den Blocksatz und gibt dann die Zeile Wort für Wort über die innere Schleife aus. Danach geht es mit der nächsten Zeile weiter, bis das Layout-Rechteck voll oder das Array durchlaufen ist. Im letzteren Fall darf die letzte Zeile nur linksbündig ausgegeben werden: Die Methode bastelt dazu die noch nicht ausgegebenen Wörter per `String.Join()` wieder zu einem String zusammen und zeichnet diesen mit einem einzigen `DrawString()`-Aufruf.

```
private void PaintJustifiedInRect(string[] words, Font f, RectangleF r,
Graphics g)
{
  int wordCounter = 0;
  int spacesInLine = 0;
  // Minimale Breite für den Wortabstand ist das Leerzeichen der Schrift
  float spaceSize = g.MeasureString(" ", f).Width;
  float lineLength = 0f;      // für Aufsummierung der Wortlängen
  float xPos = r.Left;        // für Layout-Rechteck der Wörter
  float yPos = r.Top;         // ...
  RectangleF layoutRect;
  StringFormat sf = new StringFormat();
  Brush b = new SolidBrush(SystemColors.WindowText);
  do
  {
    // mehrfache Leerzeichen ignorieren
    if(words[wordCounter] == "")
    {
      wordCounter++;
      continue;
    }
    lineLength += (g.MeasureString(words[wordCounter], f).Width);
    if (lineLength + spaceSize <= r.Width) // mit Leerz. im grünen Bereich?
    {
      wordCounter++;
      spacesInLine ++;
      lineLength += spaceSize;    // Leerzeichen dazu und weiter
      continue;
    }
    // Zeile ist voll, Ausgabe vorbereiten
    if (lineLength <= r.Width)    // ohne Leerzeichen im grünen Bereich?
    {
      wordCounter++;              // Wort zählen
    }
```

```
    else                          // letztes Wort + Leerzeichen wieder weg
    {
      if(spacesInLine > 0)        // nur ein Wort und das zu lang?
      {                           // Dann eben abschneiden.
        lineLength -= (g.MeasureString(words[wordCounter], f).Width +
                       spaceSize);
        spacesInLine --;
      }
    }

    float spacePerWord = 0;
    if (spacesInLine > 0)       // Mehr als ein Wort? Wortabstand berechnen
      spacePerWord = (r.Width-lineLength) / spacesInLine + spaceSize;

    // Ausgabe der Zeile vornehmen
    xPos = r.Left;
    sf.Alignment = StringAlignment.Near;  // erstes Wort links ausrichten
    for(int i = wordCounter-spacesInLine - 1; i < wordCounter; i++)
    {
     if (words[i] == "")
        continue;           // mehrfache Leerzeichen ignorieren
      float wordLength = g.MeasureString(words[i], f).Width;
      layoutRect = new RectangleF(xPos, yPos, wordLength, f.Height);
//          // Layout-Rechtecke zeichnen
//          g.FillRectangles(Brushes.White, new RectangleF[] {layoutRect});
      g.DrawString(words[i], f, b, layoutRect, sf);     // Wort zeichnen
      xPos += wordLength + spacePerWord;
      if (i < wordCounter - 1)
        sf.Alignment = StringAlignment.Center;  // mittlere Wörter
      else
        sf.Alignment = StringAlignment.Far;     // letztes Wort rechts
    }

    yPos += f.Height;       // Zeilenabstand erhöhen
    lineLength = 0;
    spacesInLine = 0;
    if (yPos + f.Height > r.Height)
      return;               // eine weitere Zeile passt nicht
  }
  while(wordCounter < words.Length); // fertig, wenn alle Wörter gemessen,
  // Restliche Strings zusammenbasteln und als Letzte Zeile zeichnen.
  string lastLine = String.Join(" ", words, wordCounter -
                    spacesInLine, spacesInLine);
  sf.Alignment = StringAlignment.Near;
  layoutRect = new RectangleF(r.Left, yPos, r.Width, f.Height);
  g.DrawString(lastLine, f, b, layoutRect, sf);
}
```

Vielleicht ist es Ihnen ja aufgefallen: So, wie der Algorithmus ausgelegt ist, sollte für die Ausgabe an und für sich kein `StringFormat`-Objekt erforderlich sein. Wie Abbildung 14.29 zeigt, ist das Ergebnis dann aber nicht gerade berauschend. Dummerweise liefert die `MeasureString()`-Methode, wie bereits beklagt, keine verlässlichen Werte – weder für die Höhe noch für die Breite. Je länger ein Wort, desto weiter steht das Layout-Rechteck hinten über.

Um diesem Effekt zu begegnen, kann man ein `StringFormat`-Objekt einsetzen, das nur das erste Wort einer Zeile linksbündig ausgibt, die mittleren hingegen zentriert und das rechte rechtsbündig. Abbildung 14.30 zeigt, dass der Fehler zwar kleiner geworden, leider aber immer noch vorhanden ist. `MeasureString()` scheint also auch bei dem internen Aufruf für die Ausrichtung sein Unwesen zu treiben. Ohne die Layout-Rechtecke fällt der Fehler allerdings kaum mehr auf (Abbildung 14.31).

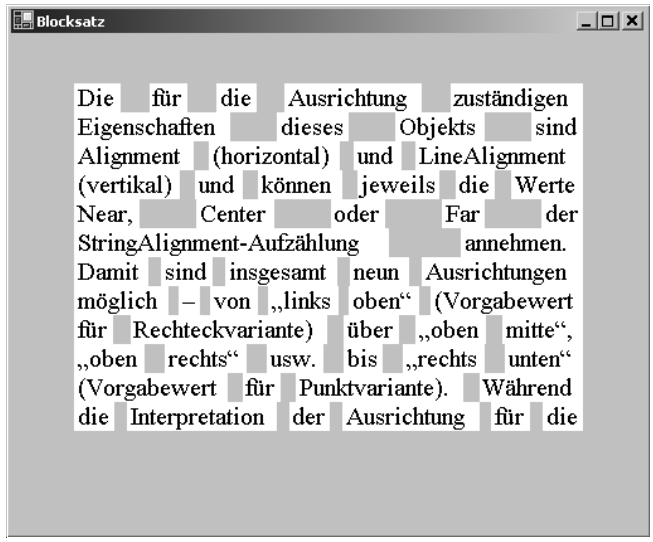


Abbildung 14.29: Obwohl die Layoutrechtecke links- und rechtsbündig sind, wirkt der Blocksatz flatterig.

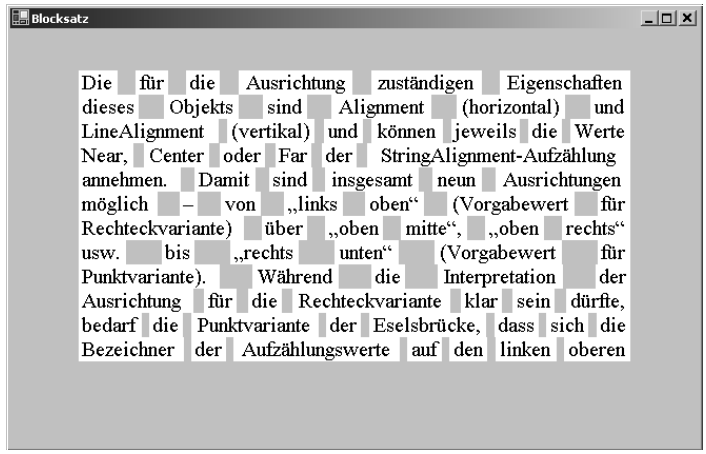


Abbildung 14.30: Die Layoutrechtecke zeigen, dass auch die Ausrichtung am rechten Rand speziell bei längeren Worten zu wünschen übrig lässt.

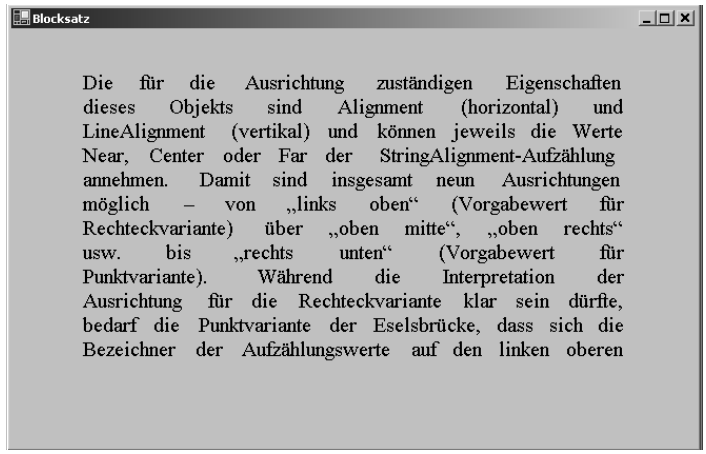


Abbildung 14.31: Das Ergebnis ist brauchbar, aber nicht perfekt.

Übung

Der vorgestellte Code nimmt nur in horizontaler Richtung einen Blocksatz vor. Erweitern Sie das Programm so, dass es auch einen vertikalen Blocksatz durchführt, also die Zeilenabstände entsprechend anpasst. (Das hört sich zunächst einmal trivial an, ist es aber nicht – vor allem, wenn man Absatzmarken berücksichtigt und zwischen zwei Absätzen mehr Platz lassen will als zwischen zwei Zeilen innerhalb eines Absatzes.)

15 Steuerelemente

Der Arbeitsablauf beim Einsetzen von Steuerelementen in einen Formular- oder Steuerelemententwurf ist weitgehend schematisiert:

1. *Schalten Sie in die Entwurfsansicht und platzieren Sie die Steuerelemente auf dem Formular- oder auch Steuerelemententwurf.* Die Reihenfolge spielt dabei eigentlich nur eine Rolle, wenn Sie die `Dock`-Eigenschaft ausnutzen wollen oder Steuerelemente in Steuerelemente platzieren wollen. (Diese Eigenschaft ermöglicht es, Steuerelemente an den Rand ihres Containers bzw. an den Rand bereits dort angehefteter Steuerelemente anzuheften). Wenn ein später eingefügtes Steuerelement als Container für bereits platzierte weitere Steuerelemente verwendet werden soll (beispielsweise `GroupBox` oder `Panel`), können Sie diese aber auch jederzeit per Drag&Drop in den Container verfrachten.

2. *Legen Sie die Initialisierungseigenschaften der Steuerelemente interaktiv im Designer fest.* Fangen Sie dabei mit den Containern an und arbeiten Sie sich schrittweise durch, bis alle Elemente wunschgemäß benannt bzw. beschriftet sind und auch an der richtigen Position stehen.

3. *Verdrahten Sie die benötigten Ereignisse der Steuerelemente einzeln, indem Sie die entsprechenden Methodengerüste interaktiv über das* EIGENSCHAFTEN-*Fenster einfügen.* Falls Sie eine Behandlungsmethode mehreren Ereignissen (und auch Steuerelementen) zuordnen wollen, fügen Sie das Methodengerüst einmal unter Vergabe eines geeigneten Bezeichners ein, implementieren Sie die Methode (was auch später noch erfolgen kann) und ordnen Sie sie dann interaktiv auch den anderen Ereignissen zu. (Kleiner Tipp: Der Designer ordnet allen neuen Instanzen einer Steuerelementklasse bestehende Behandlungsroutinen auch automatisch zu, wenn diese nach dem Schema *`Klasse_Ereignis`* benannt sind.)

4. *Implementieren Sie Feature um Feature und achten Sie dabei darauf, dass Sie zu jedem Zeitpunkt eine lauffähige Version behalten.* Testen Sie die Funktionsfähigkeit und Tauglichkeit der Ereignisbehandlung möglichst schrittweise mit jeder neu eingefügten Behandlungsmethode. Machen Sie vor jeder größeren Umstrukturierung ein Backup des Projektordners in einem Explorer-Fenster (was ohne weiteres auch im Dialog PROJEKT ÖFFNEN geht) – ein Rollback auf Ebene von VS.NET ist nur eingeschränkt über den RÜCKGÄNGIG-Befehl möglich.

15.1 Allgemeines

Gemeinsame Basisklasse der Steuerelemente ist `Control`. Diese überraschenderweise nicht abstrakte Klasse stattet die Steuerelemente (und Formulare) mit einer Unmenge an Eigenschaften, Methoden und Ereignissen aus. Abbildung 13.2 auf Seite 444 zeigt zwar die Ausstattung aus Sicht eines Formulars, doch die meisten Elemente finden sich auch bei Steuerelementen. Abbildung 15.1 zeigt den Stammbaum der meisten in der TOOLBOX anzutreffenden Steuerelemente (vgl. dazu Abbildung 13.1 auf Seite 441).

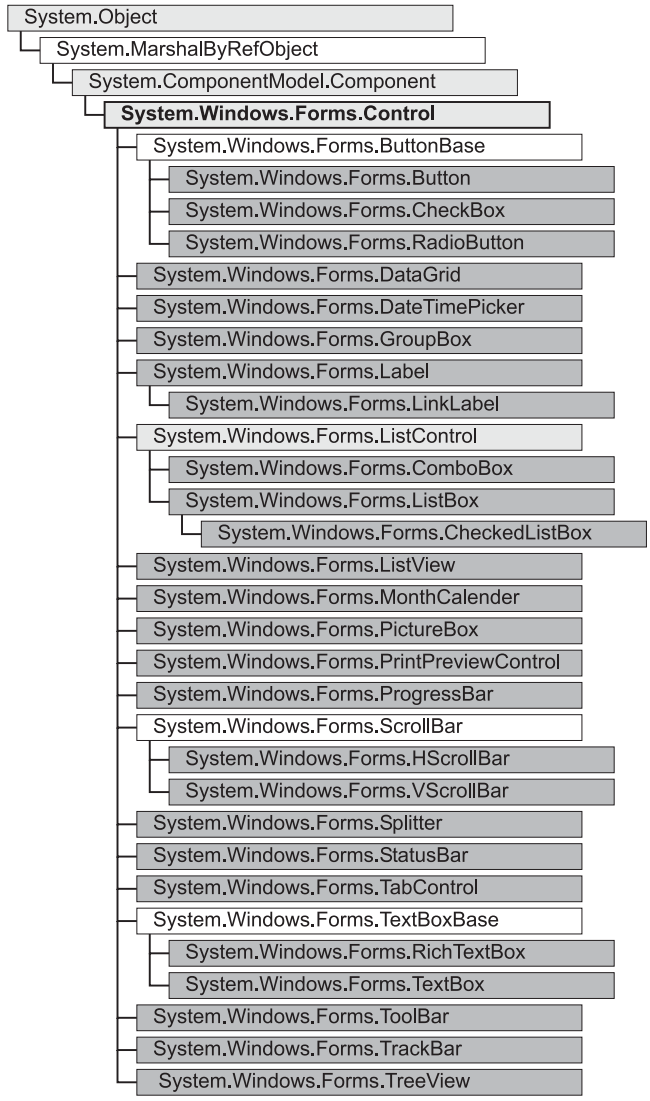


Abbildung 15.1: Klassenhierarchie – die Klasse `Control` und Elemente aus der TOOLBOX (abstrakte Klassen haben einen weißen, in der TOOLBOX befindliche Steuerelemente und Komponenten einen dunklen Hintergrund)

15.1.1 Tipps und Tricks beim Formularentwurf

Wenn Sie in die Entwurfsansicht Ihres Formulars schalten, finden Sie in der WINDOWS FORMS-Auflistung der TOOLBOX eine reiche Auswahl an Steuerelementen und Komponenten, die Sie von anderen Windows-Anwendungen her kennen. Um eine Instanz eines bestimmten Elements als neues Formularelement in dem Formularentwurf zu platzieren, markieren Sie die zugehörige Symbolschaltfläche und ziehen dann an der anvisierten Position ein Rechteck der entsprechenden Größe auf. Steuerelemente werden weitgehend so dargestellt, wie sie später zur Laufzeit erscheinen. Komponenten ohne ständige visuelle Repräsentation erscheinen hingegen als Kästchen im unteren Teil des Entwurfsfensters.

Steuerelemente mit Standardgröße wie beispielsweise `button` *können auch über einen Klick anstelle eines aufgezogenen Rahmens platziert werden und verwenden dann erst einmal die Standardvorgaben für Höhe und Breite.*



Toolbox im Weg?

Wenn Ihnen dabei die standardmäßig am linken Rand des Entwurfsfensters aufklappende TOOLBOX-Leiste im Weg ist, können Sie diese auch mit folgender Prozedur an eine andere Stelle, am besten an den rechten Fensterrand des VS-Hauptfensters, andocken:

1. Klappen Sie die Leiste auf und fixieren Sie sie durch einen Klick auf die Pin-Schaltfläche in der Titelleiste.
2. Ziehen Sie die Leiste mit der Maus an der rechten Fensterrand bis sie andockt.
3. Lösen Sie die Fixierung der Leiste durch einen weiteren Klick auf die Pin-Schaltfläche. Sie zieht sich nun an den rechten Fensterrand des Hauptfensters zurück – und bleibt im weiteren dort.

Wo sind die Komponenten?

Falls sie eine Komponente platziert haben, die nicht von `Control` abstammt bzw. im Formularlayout nicht in Erscheinung tritt (beispielsweise `Timer`), finden Sie ein Kästchensymbol dafür im unteren Bereich des Entwurfsfensters, den sie womöglich dafür etwas erweitern müssen. Die Starteigenschaften einer Komponente legen Sie wie gewohnt im EIGENSCHAFTEN-Fenster fest.

Wie bearbeite ich das richtige Element?

Der Designer markiert in der aktuellen Version gerne mal mehrere Elemente auf einmal bzw. macht das Markieren einzelner Elemente schwer. Bevor Sie also die Eigenschaften eines Elements bearbeiten, sollten Sie darauf achten, dass nur dieses Element im Formularentwurf ausgewählt ist. Falls nicht, klicken Sie auf einer freien Fläche des Formulars und wiederholen den Vorgang oder nehmen notfalls den Weg über das Kontextmenü. Da es schnell mal pas-

siert, dass man die Eigenschaften des falschen Elements verändert, prüfen Sie, ob der Bezeichner des Elements im Kombinationsfeld des EIGENSCHAFTEN-Fensters auch wirklich angezeigt wird. In der Liste dieses Felds finden Sie nicht nur einen Überblick über die bereits vergebenen Bezeichner, sondern können auch jederzeit ein anderes Element für die Bearbeitung auswählen.

Sie können natürlich auch gezielt mehrere Steuerelemente markieren und Eigenschaften, die allen gemein sind, kollektiv ändern.

Das EIGENSCHAFTEN-Fenster bietet eine alphabetische und eine nach Kategorien geordnete Ansicht der Eigenschaften sowie eine Ansicht, in der nur die Ereignisse aufgelistet sind. Der Wechsel geschieht über Klicks auf die Schaltflächen oberhalb der Liste.

Welche Eigenschaft, welches Ereignis ist wofür?

Wenn Sie eine Eigenschaft oder ein Ereignis im EIGENSCHAFTEN-Fenster markieren, erhalten Sie im unteren Bereich des EIGENSCHAFTEN-Fensters die dazugehörige Kurzbeschreibung. Falls die Beschreibung nicht sichtbar ist, müssen Sie den Fensterbereich etwas aufziehen oder im Kontextmenü des Fensters die Option BESCHREIBUNG einschalten. Zudem gibt sich das EIGENSCHAFTEN-Fenster viel Mühe, den Wertebereich einer Eigenschaft möglichst »mundgerecht« in Kombinationsfeldern oder Eigenschaftsdialogen zu präsentieren. Der Datentyp lässt sich hier in den meisten Fällen aber nur erahnen. Sie ermitteln ihn entweder über die Online-Hilfe oder über die zu der Eigenschaft eingeblendete kontextbezogene Syntaxhilfe in der Codeansicht (Inline-Hilfe).

Welche Eigenschaften hat ein Steuerelement oder eine Komponente?

Obwohl die meisten Steuerelemente und Komponenten ihre gemeinsame Abstammung nicht verleugnen und eine ganze Reihe gleicher oder zumindest ähnlicher Eigenschaften aufweisen, enthalten sie doch jeweils eigene Sätze spezifischer Eigenschaften, die auf ihre individuelle Funktionalität zugeschnitten sind. Am besten lernen Sie die Eigenschaften interaktiv im EIGENSCHAFTEN-Fenster kennen.

Dabei ist allerdings zu bedenken, dass Sie zur Entwurfszeit nur die Entwurfseigenschaften eines Steuerelements sehen. Zur Laufzeit sind im Allgemeinen noch eine Reihe zusätzlicher Eigenschaften verfügbar, viele davon nur für den Lesezugriff. Eine vollständige Auflistung aller Member einer Steuerelementinstanz präsentiert Ihnen die kontextbezogene Syntaxhilfe in der Codeansicht. Um diese – bei entsprechender Konfiguration des Editors – an sich automatisch während der Eingabe präsentierte Inline-Hilfe explizit anzufordern, setzen Sie die Schreibmarke irgendwo in den Bezeichner der Instanz und rufen den Kontextmenübefehl MEMBER AUFLISTEN auf. Informationen über Parameter von Methoden rufen Sie entsprechend über den Kontextmenübefehl PARAMETERINFO ab.

Wie füge ich Behandlungsmethoden für Ereignisse ein?

Behandlungsmethoden können Sie grundsätzlich für einzelne Steuerelemente (Komponenten) oder auch gleich für mehrere Steuerelemente definieren.

Um ein Methodengerüst für die Behandlung des Standardereignisses eines Steuerelements (Komponente) zu generieren, genügt ein Doppelklick auf das jeweilige Steuerelement (bei Komponenten: auf das Rechtecksymbol). Behandlungsmethoden für alle anderen Ereignisse fügen Sie in der Ansicht EREIGNISSE (Blitzsymbol) des EIGENSCHAFTEN-Fensters ein. Hierbei gibt es immerhin vier unterschiedliche Vorgehensweisen:

1. *Doppelklick auf den Bezeichner des Ereignisses oder die leere Zelle daneben* – der Designer generiert von sich aus einen Namen für eine neue Behandlungsmethode, der nach dem Muster: Name des Steuerelements + Unterstrich + Name des Ereignisses aufgebaut ist. Im nächsten Schritt fügt er das Gerüst einer Methode mit diesem Bezeichner ein, registriert diese als Behandlungsmethode für das Ereignis in `InitializeComponent()`, schaltet in die Codeansicht und setzt die Schreibmarke in den Rumpf der Methode.
2. *Eingabe eines Methodenbezeichners in die leere Zelle neben dem Ereignisnamen* – wie 1., mit dem Unterschied, dass der Designer den von Ihnen festgelegten Bezeichner für die Methode verwendet.
3. *Auswahl einer bereits bestehenden Methode aus der Liste unter der leeren Zelle neben dem Ereignisnamen* – der Designer registriert die bestehende Methode als Behandlungsmethode für dieses Ereignis. (Dabei werden eventuell bereits vorhandene andere Zuordnungen der Methode natürlich *nicht* gelöscht.)
4. *Eingabe eines Methodenbezeichners der aus dem Klassennamen + Unterstrich + Ereignisnamen gebildet ist* – wie 2. Zusätzlich registriert der Designer diese Methode automatisch für jedes neu eingefügte Steuerelement (Komponente) derselben Klasse.

Wie lösche ich Behandlungsmethoden für Ereignisse?

Dafür haben Sie zwei Möglichkeiten:

➡ Schalten Sie in die Entwurfsansicht, markieren Sie das betroffene Steuerelement und aktivieren Sie dann die Ansicht EREIGNISSE (Blitzsymbol) des EIGENSCHAFTEN-Fensters. Löschen Sie den Eintrag in der Zelle neben dem Ereignisnamen. Der Designer entfernt nun die Registrierung der Behandlungsmethode für das Ereignis, löscht die Methodendefinition aber nur, wenn das Methodengerüst selbst leer ist und keine weiteren Registrierungen derselben Methode existieren.

- Entfalten Sie den vom Designer gepflegten Quelltextknoten `Windows Form Designer generated code` und löschen Sie darin alle Registrierungen der Methode. Alternativ können Sie auch die Methodendefinition löschen und den Compiler Registrierungsversuche der soeben gelöschten Methode als Fehler ausweisen lassen. Über Mausklicks auf den entsprechenden Meldungen im AUSGABE-Fenster gelangen Sie dann zu den entsprechenden Stellen im Quelltext. Löschen Sie jeweils die gesamte Codezeile.

Wenn Sie eine Behandlungsmethode versehentlich per Doppelklick oder einfach nur für das falsche Steuerelement generiert haben, können Sie diesen Schritt jederzeit rückgängig machen, indem Sie in die Entwurfsansicht zurückschalten und den Befehl BEARBEITEN/RÜCKGÄNGIG oder Strg+Z anwenden.

Umbenennen von Elementen, die der Designer generiert

Bezeichner für Steuerelemente (Komponenten)

Generell sollten Sie Bezeichner für Steuerelemente oder sonstiger über den gesamten Quelltext verteilter Größen nach Möglichkeit nicht im Nachhinein ändern. Wenn es sich nicht vermeiden lässt, ändern Sie den Bezeichner nur, wenn der Quelltext fehlerlos kompilierbar ist. Verwenden Sie dann die folgende Prozedur:

1. Schalten Sie in die Codeansicht und rufen Sie den Befehl BEARBEITEN/SUCHEN UND ERSETZEN/ERSETZEN auf.
2. Markieren Sie das Kontrollkästchen AUSGEBLENDETEN TEXT DURCHSUCHEN und gegebenenfalls auch die Option ALLE GEÖFFNETEN DOKUMENTE.
3. Tragen Sie Werte in die Textfelder SUCHEN und ERSETZEN ein, achten Sie dabei aber darauf, dass auch andere Bezeichner gegebenenfalls den Suchstring enthalten können, oder der neu gewählte Bezeichner bereits für ein anderes Element vergeben sein könnte. In beiden Fällen kann die Operation den Quelltext bis zur Unbrauchbarkeit verändern und sollte schleunigst mit BEARBEITEN/RÜCKGÄNGIG widerrufen werden.
4. Kompilieren Sie den Quelltext, um zu sehen, ob auch unerwünschte Ersetzungen passiert sind.



Visual Studio .NET speichert vor dem Start des Compilers sämtliche Quelltextdateien. Die Funktion BEARBEITEN/RÜCKGÄNGIG ist danach nicht mehr verfügbar.

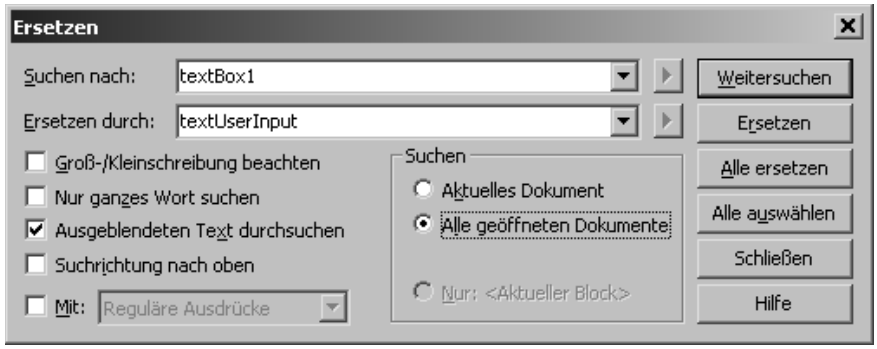


Abbildung 15.2: Der ERSETZEN-Dialog von Visual Studio .NET

Formular

Zur Änderung des Formularnamens schalten Sie in den PROJEKTMAPPEN-EXPLORER und verwenden dann dieselbe Prozedur, als würden Sie einen Dateinamen im Windows-Explorer ändern. VS.NET benennt dabei auch die *cs*.Datei um.

Namensbereich

Den Namensbereich ändern Sie, indem Sie alle entsprechend lautenden `namespace`-Direktiven im Quelltext ändern. Falls Sie zusätzlich *.ico*, *.cur* oder *.bmp*-Dateien als Ressourcen einbinden, müssen Sie auch die `Standardnamespace`-Eigenschaft der Assembly anpassen. Sie finden diese Eigenschaft im Eigenschaftsdialog des Projekts unter der Rubrik ALLGEMEIN/ALLGEMEINE EIGENSCHAFTEN.

Projekt

Zur Änderung des Projektnamens schalten Sie in den PROJEKTMAPPEN-EXPLORER und verwenden dann dieselbe Prozedur, als würden Sie einen Dateinamen im Windows-Explorer ändern. VS.NET benennt dann auch gleich die *.csproj*-Datei um und aktualisiert gegebenenfalls auch bestehende Verweise. Alternativ lässt sich der Name auch über die `Name`-Eigenschaft im EIGENSCHAFTEN-Fenster des Projekts ändern. Den Projektordner an sich benennt VS.NET nicht um.

Projektmappe

Zur Änderung des Projektmappennamens schalten Sie in den PROJEKTMAPPEN-EXPLORER und verwenden dann dieselbe Prozedur, als würden Sie einen Dateinamen im Windows-Explorer ändern. VS.NET benennt dann auch gleich die *.sln*- und die *.suo*-Dateien um. Alternativ lässt sich der Name auch über `Name`-Eigenschaft im EIGENSCHAFTEN-Fenster der Projektmappe ändern.

EXE-Datei

Visual Studio .NET verwendet den beim Anlegen eines Projekts verwendeten Namen auch für die zu erzeugende ausführbare Datei, passt diese Vorgabe bei nachträglichen Änderungen des Projektnamens aber nicht automatisch an, weshalb Sie das gegebenenfalls über die EIGENSCHAFTSSEITEN des Projekts nachholen müssen. Ausschlaggebend ist hier der über ALLGEMEINE EIGENSCHAFTEN/ALLGEMEIN erreichbare Eintrag ASSEMBLYNAME. Eine eventuell bereits existierende EXE-Datei wird dabei nicht umbenannt. (Wer nach einer ersten Compilierung den Assembly-Namen von – beispielsweise – `WindowsApp1` auf `MyApp` ändert, findet später in *Debug\Bin* sowohl *WindowsApp1.exe* als auch *MyApp.exe* vor.)

15.1.2 Eigenschaften

Tabelle 15.1 listet verschiedene Eigenschaften auf, die allen Steuerelementen gemeinsam sind. Mehr zu spezifischen Eigenschaften einzelner Steuerelemente finden Sie in dem Abschnitt, der dem jeweiligen Steuerelement gewidmet ist.

Tabelle 15.1: Auswahl an wichtigen Eigenschaften eines Steuerelements

Eigenschaft	Bedeutung
`(Name)` `string Name {get; set;}`	Im EIGENSCHAFTEN-Fenster setzen Sie mit dieser Eigenschaft auch den Bezeichner der Objektvariable, mit dem die Komponente im Quelltext erscheint. (Achtung: Bei nachträglicher Änderung dieser Eigenschaft im EIGENSCHAFTEN-Fenster übernimmt der Designer den neuen Bezeichner nur für `InitializeComponent()`. Im restlichen Quelltext bleibt der alte Bezeichner stehen und muss manuell gesucht und ersetzt werden. Ändern Sie diese Eigenschaft also als erstes und möglichst noch, bevor Sie Code schreiben, der auf das Steuerelement referiert. Um die Eigenschaft später noch zu ändern, verwenden Sie besser gleich »Suchen und Ersetzen« in der Codeansicht. Eine Änderung dieser Eigenschaft per Code ändert den Bezeichner natürlich nicht mehr.
`bool AllowDrop {get; set;}`	Diese Eigenschaft muss auf `true` gesetzt werden, damit ein Steuerelement generell als Zielobjekt einer Drag&Drop-Operation erscheinen kann (eine Behandlung der entsprechenden Ereignisse muss natürlich auch noch erfolgen).

Tabelle 15.1: Auswahl an wichtigen Eigenschaften eines Steuerelements (Forts.)

Eigenschaft	Bedeutung
`AnchorStyles Anchor {get; set;}`	Wert dieser Eigenschaft ist ein Bitvektor des Aufzählungstyps `AnchorStyles`. Die einzelnen Bits der Aufzählwerte bestimmen, zu welchen Rändern der Containerkomponente (Formular, `GroupBox`, `Panel` etc.) das Steuerelement seinen Abstand beibehält, wenn sich deren Abmessungen ändern. Das Steuerelement passt dann seine Abmessungen entsprechend an. Sind beispielsweise der rechte und linke Rand verankert, ändert das Steuerelement seine `Width`-Eigenschaft wie der Container. Die die `Height`-Eigenschaft bleibt unverändert, an ihrer Stelle wird die `Top`-Eigenschaft so angepasst, dass die relative Position des Steuerelements zwischen dem oberen und dem unteren Rand erhalten bleibt. Eine Änderung der `Anchor`-Eigenschaft kann eine Änderung der `Dock`-Eigenschaft nach sich ziehen.
`Color BackColor {get; set;}`	Hintergrundfarbe des Steuerelements. Im Entwurfsmodus stehen drei Paletten zur Verfügung: die Standardfarben (48 vordefinierte plus 16 benutzerdefinierte), die Webfarben und die Systemfarben. Bei Änderungen sollten Sie sich im Allgemeinen auf Systemfarben beschränken, damit ihre Wahl nicht so sehr aus dem jeweils vom Benutzer eingestellten Windows-Farbschema ausbricht. Farben (ohne Transparenzanteil) mit spezifischen RGB-Wert stellen Sie entweder gleich per Code ein, indem Sie den jeweiligen Farbwert über die Methode `Color.FromArgb()` generieren, oder sie notieren den RGB-Werte durch Semikolon getrennt im EIGENSCHAFTEN-Fenster – beispielsweise: `152; 152; 152`.
`int Bottom {get;}`	*y*-Koordinate des unteren Fensterrands des Steuerelements im Koordinatensystem des Containers
`Rectangle Bounds{get; set;}`	Position und Abmessungen des Steuerelementbereichs im Koordinatensystem des Containers (vgl. auch `SetBounds()`)
`bool CanFocus {get;}`	`true`, wenn das Steuerelement den Eingabefokus übernehmen kann
`bool Capture {get;}`	`true`, wenn das Steuerelement den Fokus per Maus erhalten hat und nach wie vor besitzt. ⇆, Navigationstasten und Tastenkürzel beenden diesen Zustand, stellen ihn aber nicht her.)

Tabelle 15.1: Auswahl an wichtigen Eigenschaften eines Steuerelements (Forts.)

Eigenschaft	Bedeutung
`bool CanSelect {get;}`	`true`, wenn das Steuerelement zur Laufzeit per Maus oder Tastatur ausgewählt werden kann (markierte Darstellung).
`bool CausesValidation {get; set;}`	Standardmäßig `true`. Die Eigenschaft spielt eine Rolle im Zusammenhang mit der Gültigkeitsprüfung. Gibt ein erstes Steuerelement, das unterschiedliche Zustände haben kann und dessen `CausesValidation`-Eigenschaft auf `true` gesetzt ist, den Fokus an ein zweites Steuerelement ab, dessen `CausesValidation`-Eigenschaft gleichfalls `true` ist, generiert das erste Steuerelement die Ereignisse `Validating` und `Validated`.
`Rectangle ClientRectangle {get; set;}`	Position und Abmessungen des Clientbereichs des Steuerelements (bzw. Formulars) im eigenen Koordinatensystem. Die Position ist daher immer (0,0).
`Size ClientSize {get; set;}`	Ruft die Abmessungen des Clientbereichs des Steuerelements ab und erlaubt es auch, diese zu ändern. Dementsprechend ändern sich auch verschiedene andere Eigenschaften mit Positionsbezug.
`IContainer Container {get;}`	Verweis auf die `IContainer`-Schnittstelle des Containersteuerelements. Diese Schnittstelle ermöglicht den Lesezugriff auf die `ContainerCollection`-Eigenschaft `Components` und den Aufruf der Methoden `Add()` und `Remove()`.
`bool ContainsFocus {get;}`	`true`, wenn das Steuerelement oder eines seiner untergeordneten Steuerelemente den Eingabefokus besitzt.
`ContextMenu ContextMenu {get; set ;}`	Kontextmenu des Steuerelements
`Control[] Controls {get;`	Ruft die Auflistung der dem Steuerelement untergeordneten Steuerelemente ab.
`bool Created {get;}`	`true`, wenn das Steuerelement vollständig geniert, d.h. sämtliche Initialisierungen abgeschlossen wurden – bedeutet aber nicht zwangsläufig, dass bereits ein Windows-Fensterhandle für das Steuerelement existiert.
`Cursor Cursor {get; set;}`	Objekt, das den Mauszeiger beschreibt, wenn die Maus in den Bereich des Steuerelements gerät.

Tabelle 15.1: Auswahl an wichtigen Eigenschaften eines Steuerelements (Forts.)

Eigenschaft	Bedeutung
`bool Disposing {get;}`	`true`, wenn das Steuerelement gerade zerstört wird (siehe auch `Dispose()`).
`DockStyle Dock {get; set;}`	Wert dieser Eigenschaft ist ein Element des Aufzählungstyps `DockStyle`. Es legt fest, an welchem Rand der Containerkomponente das Steuerelement (abstandslos) andockt. Eine Änderung der `Dock`-Eigenschaft kann eine Änderung der `Anchor`-Eigenschaft nach sich ziehen, die das Einstellen fester Randabstände ermöglicht.
`bool Enabled {get; set;}`	Wenn `false`, wird das Steuerelement im deaktivierten Zustand gezeichnet. In diesem Zustand nimmt es weder den Eingabefokus an noch lässt es sich auswählen.
`bool Focused {get;}`	`true`, wenn das Steuerelement den Eingabefokus besitzt.
`Color ForeColor {get; set;}`	Bestimmt die Vordergrundfarbe des Steuerelements oder ruft sie ab. Zur Farbauswahl siehe `BackColor`.
`IntPtr Handle {get;}`	Liefert einen Handle für das Fenster des Steuerelements (und zwangsläufig `null`, wenn `Created` noch den Wert `false` hat).
`bool HasChildren{get;}`	`true`, wenn das Steuerelement untergeordnete Steuerelemente besitzt.
`bool IsHandleCreated {get;}`	`true`, wenn dem Steuerelement ein Fenster zugeordnet wurde (und die `Handle`-Eigenschaft ungleich null ist).
`bool Locked {get; set;}`	Diese Eigenschaft ist nur zum Entwurfszeitpunkt sichtbar. Wenn `true`, unterbindet sie alle Layout-bezogenen Änderungen des Steuerelements, fixiert also seine Position und Größe.
`Control Parent {get;}`	Ruft das dem Steuerelement übergeordnete Steuerelement (auch Formular) ab.
`Region Region {get; set;}`	Beschreibt den für das Steuerelement gezeichneten Bereich (im Allgemeinen ein Rechteck, jedoch sind mit dieser Eigenschaft beliebige Ausschnitte definierbar (vgl. Abbildung 14.4, rechts, auf Seite 479).
`int Right {get;}`	*x*-Koordinate des rechten Fensterrands des Steuerelement im Koordinatensystem des Containers

Tabelle 15.1: Auswahl an wichtigen Eigenschaften eines Steuerelements (Forts.)

Eigenschaft	Bedeutung
`object Tag {get; set;}`	Allgemeine benutzerdefinierbare Eigenschaft, die es ermöglicht, dem Steuerelement ein beliebiges Objekt anzuheften
`int TabIndex {get; set;}`	Position in der Aktivierungsreihenfolge bei Tabulatorweiterschaltung. Den Wert dieser Eigenschaft setzen Sie beim Entwurf nicht direkt. Vielmehr benutzen Sie nach Fertigstellung des gesamten Formularentwurfs den Befehl TABULATOR-REIHENFOLGE im Menü ANSICHT, um die Aktivierungsreihenfolge komfortabel per Mausklicks festzulegen.
`bool TabStop {get; set;}`	Wenn `true`, kann der Benutzer den Fokus mit der Tabulatortaste auf das Element setzen, andernfalls nur per Maus. Zur Tabulatorreihenfolge siehe `TabIndex`.
`string Text {get; set;}`	Bestimmt die Beschriftung des Steuerelements oder ruft sie ab. Das Zeichen `'&'` ermöglicht es, `Alt`-Tastenkürzel festzulegen, über die der Benutzer dem Steuerelement per Tastatur den Eingabefokus zuschanzen kann. So legt der Text für `»Ein&schalten«` Alt+S als Tastenkürzel fest. Falls das Steuerelement den Fokus selbst nicht annehmen kann, erhält ihn das nächste in dieser Hinsicht geeignete Steuerelement entlang der Tabulatorreihenfolge. Die Eigenschaft wird von manchen Steuerelementen aber auch anderweitig verwendet: Bei einem Formular benennt sie den Text in der Titelleiste; bei einem `TextBox`- oder `RichTextBox`-Steuerelemente den Eingabewert (editierbar, wenn `ReadOnly` auf `false` gesetzt ist) usw.
`ContentAlignment TextAlign {get; set;}`	Ausrichtung der `Text`-Eigenschaft im Steuerelement. Die `ContentAlignment`-Aufzählung definiert dafür neun Positionen
`Control TopLevelControl {get;}`	Oberstes, übergeordnetes Steuerelement – im Allgemeinen das Formular.
`bool Visible {get; set;}`	Wenn `false`, ist das Steuerelement nicht sichtbar. In diesem Zustand reagiert es auch nicht auf Eingaben und nimmt auch den Fokus nicht an. Durch geeignete Pflege der `Visible`-Eigenschaften von Steuerelementen lassen sich Formulare als Mehrzweckformulare entwerfen, die zu einem Zeitpunkt immer nur den Satz an Steuerelementen anzeigen, der gerade opportun ist.

15.1.3 Methoden

Tabelle 15.2 gibt einen Überblick über den Löwenanteil der Methoden, die ein Steuerelement von der Klasse `Control` erbt. Mehr über die charakteristischen Methoden der Steuerelemente finden Sie bei der Besprechung der Steuerelemente im Einzelnen. Einige der Methoden machen nur für Steuerelemente Sinn, die als Container für andere Steuerelemente fungieren.

Tabelle 15.2: Wichtige allgemeine Methoden von `Control`-Objekten

Methode	Bedeutung
`void BringToFront()`	Setzt das Steuerelement an den Anfang der `Controls`-Auflistung seines Containers. Es wird dann zuoberst gezeichnet.
`bool Contains()`	Liefert `true`, wenn das Steuerelement Container für das im Parameter übergebene Steuerelement ist
`CreateControl()`	Nach Ausführung des Konstruktors existiert zwar das Steuerelementobjekt, jedoch ist diesem noch kein Fenster zugeordnet. Diese Methode erzwingt die Fertigstellung des Steuerelements (und aller untergeordneter Steuerelemente) – beispielsweise, um seine Fenstereinstellung abfragen oder andere auf das Fenster gerichtete Operationen ausführen zu können.
`Graphics CreateGraphics()`	Liefert ein `Graphics`-Objekt, das es ermöglicht, außerhalb der `Paint`-Behandlung in den Fensterbereich des Steuerelements zu zeichnen.
`void Dispose()`	Gibt Fenster und sonstige Ressourcen des Steuerelements sowie aller ihm gegebenenfalls untergeordneter Steuerelemente frei.
`DragDropEffects DoDragDrop()`	Startet einen Drag&Drop-Vorgang, bei dem das Steuerelement Quellobjekt ist.
`Form FindForm()`	Liefert das Formular, dem das Steuerelement untergeordnet ist.
`bool Focus()`	Setzt den Fokus auf das Steuerelement. Liefert `true`, wenn das Steuerelement den Fokus annehmen konnte – was immer dann der Fall ist, wenn seine Eigenschaft `CanFocus` den Wert `true` hat.
`Control GetChildAtPoint()`	Diese Methode macht nur für einen Container Sinn. Sie liefert eine Referenz auf das erste Steuerelement der `Controls`-Auflistung dessen Fensterbereich die angegebene Position überdeckt. Falls sich kein Steuerelement findet, liefert sie den Wert `null`.

Tabelle 15.2: Wichtige allgemeine Methoden von `Control`-Objekten (Forts.)

Methode	Bedeutung
`Control GetContainerControl()`	Liefert das nächste übergeordnete Objekt, das von der Klasse `ContainerControl` abstammt – meist ist dies das Formular.
`Control GetNextControl()`	Liefert das Steuerelement, das dem Steuerelement in der Tabulatorreihenfolge des gemeinsamen Containers nachfolgt.
`void Hide()`	Setzt die `Visible`-Eigenschaft des Steuerelements auf `false` und macht es damit unsichtbar.
`void Invalidate()`	Erklärt den Fensterbereich des Steuerelements (oder einen Teil davon) für ungültig und löst mittelbar ein `Paint`-Ereignis aus. Im Gegensatz zu `Refresh()` wird die Darstellung also nicht sofort aktualisiert, sondern frühestens nach Ende der aktuellen Ereignisbehandlung. Die Methode ist dreifach überladen. Die erste Variante ist parameterlos und erklärt den gesamten Fensterbereich des Steuerelements für ungültig, die zweite beschränkt die `Paint`-Behandlung auf ein `Rectangle`-Objekt im Fensterbereich und die dritte ermöglicht es, ein komplexes `Region`-Objekt als Clipbereich im Fensterbereich anzugeben.
`void PerformLayout()`	Erzwingt, dass das Steuerelement seine untergeordneten Elemente neu ausrichtet. (Wird standardmäßig als Reaktion auf das `Resize`-Ereignis aufgerufen.)
`Point PointToClient()`	Rechnet einen Bildschirmpunkt in das Koordinatensystem des Steuerelements um. (Der Ursprung liegt in der linken oberen Ecke des Steuerelements.)
`Point PointToScreen()`	Rechnet einen Punkt vom Koordinatensystem des Steuerelements in Bildschirmkoordinaten um. (Der Ursprung des Bildschirmkoordinatensystems liegt in der linken oberen Ecke des Bildschirms.)
`Rectangle RectangleToClient()`	Rechnet ein `Rectangle`-Objekt von Bildschirmkoordinaten in das Koordinatensystem des Steuerelements um.
`Rectangle RectangleToScreen()`	Rechnet ein `Rectangle`-Objekt vom Koordinatensystem des Steuerelements in Bildschirmkoordinaten um.

Methode	Bedeutung
`void Refresh()`	Erklärt den gesamten Fensterbereich des Steuerelements sowie die Fensterbereiche aller ihm untergeordneten Steuerelemente für ungültig (vgl. `Invalidate()`) und erzwingt dann ein unmittelbar folgendes Neuzeichnen.
`void Scale()`	Skaliert das Steuerelement und alle ihm untergeordneten Steuerelemente um die angegebenen Faktoren in *x*- und *y*-Richtung.
`bool Select()`	Aktiviert das Steuerelement, d.h. wählt es aus und setzt den Eingabefokus darauf, wenn das Element den Eingabefokus übernehmen kann. Liefert `true` bei erfolgreicher Ausführung (was immer der Fall ist, so lange `CanSelect` den Wert `true` hat).
`bool SelectNextControl()`	Ruft `Select()` für das jeweils nächste Steuerelement in der Tabulatorreihenfolge des gemeinsamen Containers auf.
`void SendToBack()`	Setzt das Steuerelement an das Ende der `Controls`-Auflistung seines Containers. Es wird dann zuunterst gezeichnet.
`void SetBounds()`	Setzt die Position und Abmessungen des Steuerelements neu.
`void Show()`	Zeigt das Steuerelement an (setzt die `Visible`-Eigenschaft auf `true`).

Tabelle 15.2: Wichtige allgemeine Methoden von `Control`-Objekten (Forts.)

15.1.4 Ereignisse

Neben den standardmäßigen Ereignissen (Maus-, Tastatur-, Drag&Drop- und Fokus-Ereignisse), die ein Steuerelement aufgrund seiner Abstammung von `Control` meldet (vgl. Tabelle 13.1 auf Seite 449), signalisiert es im Wesentlichen `XxxChanged`-Ereignisse als Reaktion auf Änderungen von Eigenschaften. Nur die wenigsten Eigenschaften eines Steuerelements werden tatsächlich durch Benutzerinteraktionen geändert – die meisten durch Code –, weshalb bereits die Behandlung weniger Ereignisse ausreicht, um den typischen Funktionsumfang eines Steuerelements in den Code zu integrieren.

Tabelle 15.3 stellt verschiedene wichtige Ereignisse vor, die von den meisten Steuerelementen signalisiert werden.

Tabelle 15.3: Wichtige allgemeine Ereignisse von `Control`-Objekten

Ereignis	Bedeutung
`ControlRemoved`	Tritt ein, wenn ein dem Steuerelement untergeordnetes Steuerelement aus der `Controls`-Auflistung entfernt wird.
`DragDrop`	Tritt ein, wenn das Steuerelement Ziel einer Drag&Dop-Operation ist. Damit es Ziel sein kann, muss seine `AllowDrop`-Eigenschaft auf `true` gesetzt sein.
`DragEnter`	Tritt ein, wenn die `AllowDrop`-Eigenschaft auf `true` gesetzt ist und der Mauszeiger während einer Drag&Drop-Operation in den Fensterbereich des Steuerelements gelangt.
`DragLeave`	Tritt ein, wenn die `AllowDrop`-Eigenschaft auf `true` gesetzt ist und der Mauszeiger während einer Drag&Drop-Operation den Bereich des Steuerelements verlässt.
`DragOver`	Tritt ein, wenn die `AllowDrop`-Eigenschaft auf `true` gesetzt ist und der Mauszeiger während einer Drag&Drop-Operation im Bereich des Steuerelements bewegt wird (vgl. `MouseMove`).
`Enter`	Geht bei einem Steuerelement dem (nicht im EIGENSCHAFTEN-Fenster aufgelisteten) `GotFocus`-Ereignis voran, wenn dieses den Focus von einem anderen Steuerelement erhält. Tritt bei Aktivierung des Formulars nur auf, wenn das Steuerelement vor der Deaktivierung den Fokus nicht besessen hat.
`GiveFeedback`	Tritt während einer Drag&Drop-Operation ein, wenn die `AllowDrop`-Eigenschaft auf `true` gesetzt ist. Das Zielobjekt (Formular, Steuerelement) beantwortet damit eine Frage des Quellobjekts, welche Operationen (Effekte) es für den Typ des gezogenen Objekts unterstützt.
`GotFocus`	Tritt ein, wenn das Steuerelement den Fokus erhalten hat und folgt unmittelbar auf das `Enter`-Ereignis. Das Ereignis ist zwar im EIGENSCHAFTEN-Fenster nicht aufgelistet, wird aber signalisiert und kann behandelt werden. Eine Behandlung dieses recht systemnahen Ereignisses ist allerdings nur in speziellen Situationen erforderlich. Behandeln Sie stattdessen das `Enter`-Ereignis.
`Invalidated`	Tritt ein, wenn das Steuerelement beispielsweise aufgrund eines `Invalidate()`-Aufrufs neu gezeichnet werden soll.
`Layout`	Tritt ein, wenn einem Container-Steuerelement ein neues Steuerelement hinzugefügt wird oder sich das Layout aus anderen Gründen ändert. Lässt sich durch `SuspendLayout()` abschalten und durch `ResumeLayout()` wieder einschalten.
`Leave`	Folgt auf das `LostFocus`-Ereignis.

Ereignis	Bedeutung
`LostFocus`	Tritt ein, wenn das Steuerelement den Fokus erhalten hat und geht dem `Leave`-Ereignis unmittelbar voran. Das Ereignis ist zwar im EIGENSCHAFTEN-Fenster nicht aufgelistet, wird aber signalisiert und kann behandelt werden. Eine Behandlung dieses recht systemnahen Ereignisses ist allerdings nur in speziellen Situationen erforderlich. Behandeln Sie stattdessen das `Leave`-Ereignis.
`Paint`	Tritt ein, wenn das Formular oder ein Teil davon neu gezeichnet werden soll.
`QueryContinueDrag`	Tritt im Verlauf eines Drag&Drop-Vorgangs auf und ermöglicht es, die Quelle darüber zu informieren, ob es überhaupt Sinn macht, den Vorgang fortzusetzen.
`Resize`	Tritt auf, wenn sich die Abmessungen des Steuerelements ändern (nicht jedoch bei der Initialisierung des Objekts)
`Validated`	Tritt nur auf, wenn das Steuerelement den Fokus an ein anderes Steuerelement (oder auch Formular) verlieren soll und die `CausesValidation`-Eigenschaft beider `true` ist. Signalisiert dann, dass die Gültigkeitsprüfung erfolgreich verlaufen ist und der Fokus tatsächlich abgegeben wird.
`Validating`	Tritt nur auf, wenn das Steuerelement den Fokus an ein anderes Steuerelement (oder auch Formular) abgeben soll und die `CausesValidation` –Eigenschaft beider `true` ist. Eine Behandlungsmethode kann eine Gültigkeitsprüfung des Elements vornehmen und die Weitergabe des Fokus gegebenenfalls durch Setzen der `Cancel`-Eigenschaft des Ereignisobjekts unterbinden – typischerweise nach Ausgabe einer Aufforderung an den Benutzer, er möge seine Eingabe korrigieren.
`XxxChanged`	Der Wert der Eigenschaft `Xxx` hat sich geändert.

Tabelle 15.3: Wichtige allgemeine Ereignisse von `Control`-Objekten (Forts.)

15.1.5 Koordinatensystem für Abmessungen und Position

Alle Abmessungen und Positionen eines Steuerelements sind in Bildpunkten gehalten, wobei als Ursprung des Koordinatensystems die linke obere Ecke des jeweiligen Containers (Formular oder Container-Steuerelement) fungiert und die *y*-Achse nach unten zeigt.

Eigenschaften	Methoden	Ereignisse
`Anchor`, `Bottom`, `Bounds`, `ClientRectangle`, `ClientSize`, `Dock`, `Height`, `Left`, `Location`, `Right`, `Size`, `Top`, `Width`	`Scale()`, `SetBounds()`	`Move`, `Resize`, `LocationChanged`, `SizeChanged`

Tabelle 15.4: Von `Control` vererbte Elemente, die mit Abmessungen und Positionen zu tun haben

15.1.6 Generieren und Einbinden von Steuerelementen zur Laufzeit

Der Designer ist eine gute Hilfe, wenn es darum geht, einzelne Steuerelemente nach visuellen Gesichtspunkten zu gruppieren, auszurichten und individuell zu initialisieren. Da er sämtliche Designinformationen in Code umsetzt, finden Sie alles, was den jeweiligen Entwurf ausmacht, im Quelltext. Damit ist es problemlos, Steuerelemente am Designer vorbei nach Bedarf auch per Code in ein Formular bzw. Containerelement (`UserControl`, `GroupBox` etc.) einzusetzen, zu initialisieren, zu verdrahten und auch wieder zu entfernen.



Wie das geht, macht einem der Designer ja vor. Am besten fügen Sie eine Steuerelementinstanz mit dem Designer ein und transferieren dann den Code in Ihre eigene `Initialize()`*-Routine.*

Stellen Sie sich vor, Sie müssen eine größere Anzahl gleichartiger Steuerelemente in einem Formular oder Container unterbringen, die alle einer gewissen Systematik folgen. In dem Fall sind Sie oft besser beraten, wenn Sie das Design rein rechnerisch – sozusagen im Blindflug – generieren. Der folgende Codeauszug aus der nächsten Beispielanwendung *Taschenrechner* zeigt, wie einfach es sein kann, zwanzig Schaltflächen zu generieren, mit ein wenig Rechenarbeit gleichmäßig über das Formular zu verteilen und zu verschalten:

```
private void InitButtons()
{
  Button b;
  string[] Tasten = new string[] {     // Beschriftung
    "M", "Recall", "Clear Reset", "±",
    "7", "8", "9", "*",
    "4", "5", "6", "-",
    "1", "2", "3", "+",
    "0", ",", "=", "/" };

  for (int i = 0; i < 20; i++)         // 20 Steuerelemente
  {
    b = new Button();
    b.Font = new Font(this.Font.Name, 14);
                                       // Tasten regelmäßig anordnen
    b.Size = new Size((int) (this.Width / 6), (int) (this.Height/8));
    double x = this.Width / 5L * (i%4 + 1) - b.Width/2;
    double y = this.Height / 7L * (i/4 + 1.5) -  b.Height/2;
    b.Location = new Point( (int) x, (int) y );
    b.Text = Tasten[i];                // Beschriften + Verdrahten
    b.Click += new System.EventHandler(this.Button_Click);
    this.Controls.Add(b);              // Container ist Formular
  }
}
```

```
protected void Button_Click(object sender, EventArgs e)
{
  Calculate(((Button) sender).Text.Substring(0, 1)); // Erstes Zeichen
}
```

Charakteristisch für den vorliegenden Code ist, dass er alle `Button`-Steuerelemente in die `Controls`-Auflistung des Containers einfügt und über eine gemeinsame Behandlungsroutine verschaltet. Letzteres muss zwar nicht sein – jedes Objekt könnte durchaus seine eigene Behandlungsmethode erhalten – doch es bringt Vorteile, wenn die Aktionen der Steuerelemente gut schematisierbar sind. Eine gemeinsame Behandlungsroutine steht natürlich vor dem Problem, dass sie das signalisierende Objekt identifizieren muss, um die richtige Aktion auszuführen, was in der Praxis meist einen zusätzlichen Verwaltungsaufwand erfordert. Falls Sie sich schon einmal gefragt haben, warum im Prototyp einer Behandlungsroutine immer ein `object`-Parameter mit von der Partie ist, erhalten Sie nun die Antwort: Ist eine Behandlungsroutine für mehrere Objekte registriert, identifiziert dieser Parameter den Absender des Ereignisses. Sie müssen nichts weiter tun, als den Parameter `sender` einer Typumwandlung zu unterziehen und haben dann den üblichen Zugriff auf die Ausstattung des Objekts.


```
protected void Button_Click(object sender, EventArgs e)
{
  Calculate(((Button) sender).Text.Substring(0, 1)); // Erstes Zeichen
}
```


Oft reicht bereits eine der Eigenschaften zur Identifikation aus (wie hier das erste Zeichen der `Text`-Eigenschaft). Beispielsweise können Sie dafür auch die ominöse Eigenschaft `Tag` verwenden, die den Datentyp `object` trägt und eigens dafür da ist, benutzerspezifische Informationen (in Form eines Objekts) an die jeweilige Instanz zu heften. Ein anderer Ansatz wäre es, die Logik schlicht auf den Index des Objekts in der `Controls`-Auflistung auszurichten und diesen via `GetChildIndex()` in Erfahrung zu bringen:

```
int ControlIndex = Controls.GetChildIndex((Control) sender);
```

Codebeispiel – Taschenrechner

Das folgende Codebeispiel *Taschenrechner* zeigt die Implementierung eines Taschenrechners auf der Basis von `Button`-Steuerelementen, der alle vier Grundrechenarten beherrscht und auch über ein M-Register zum Speichern und Abrufen von Zwischenergebnissen verfügt (Bild 15.3). Im Funktionsumfang steht er dem Windows-Rechner (in der Ansicht STANDARD) nicht viel nach. Als Anzeige des Taschenrechners dient ein `TextBox`-Steuerelement, das gleichzeitig eine wirklich billige Tastaturschnittstelle abgibt. Es reicht, das `KeyPress`-Ereignis abzufangen und die Zeichen an die Methode `Calculate()` zu übergeben. Die über die `Button`-Schaltflächen realisierte Mausschnittstelle liefert genau die gleichen Zeichen, nämlich jeweils das erste Zeichen der Schaltflächenbeschriftung.

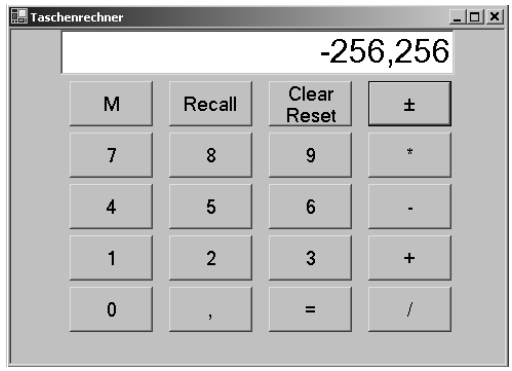


Abbildung 15.3: Ein Taschenrechner mit Maus- und Tastaturschnittstelle

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;

namespace Taschenrechner
{
  public class Form1 : System.Windows.Forms.Form
  {
    private System.Windows.Forms.TextBox display;  // Display
    private bool Decimal = false;  // true, wenn Dezimalzeichen vonhanden
    private bool Result = false;   // true, wenn Ergebnisdisplay
    private double X, M;           // Register
    enum Ops {None, Add, Subtract, Divide, Multiply};  // Operationen
    private Ops op = Ops.None;     // Aktuelle Operation

    public Form1()                 // Konstruktor
    {
      InitializeComponent();       // Textfeld initialisieren
      InitButtons();               // Schaltflächen generieren
    }                              // und initialisieren

    void InitButtons()
    {
      Button b;
      string[] Tasten = new string[] {     // Beschriftung
         "M", "Recall", "Clear Reset", "±",
         "7", "8", "9", "*",
         "4", "5", "6", "-",
         "1", "2", "3", "+",
         "0", ",", "=", "/" };
      for (int i = 0; i < 20; i++)         // 20 Tasten
      {
        b = new Button();
        b.Font = new Font(this.Font.Name, 14);
                                           // Tasten regelmäßig anordnen
        b.Size = new Size((int) (this.Width / 6), (int) (this.Height/8));
        double x = this.Width / 5L * (i%4 + 1) - b.Width/2;
        double y = this.Height / 7L * (i/4 + 1);
        b.Location = new Point( (int) x, (int) y );
```

```
    b.Text = Tasten[i];                // Beschriften + Verdrahten
    b.Click += new System.EventHandler(this.Button_Click);
    this.Controls.Add(b);              // Container ist Formular
  }
}

// Schnittstelle für Schaltflächeneingabe
protected void Button_Click(object sender, EventArgs e)
{
  Calculate(((Button) sender).Text.Substring(0, 1)); // Erstes Zeichen
}

void Calculate(string key)
{
  switch (key.ToUpper())
  {
    case "0":                 // Keine führende 0 zulassen
      if (display.Text != "" && display.Text != "-")
        goto case "9";
      break;
    case "1":
    case "2":
    case "3":
    case "4":
    case "5":
    case "6":
    case "7":
    case "8":
    case "9":                 // Die zehn Ziffern
      if (Result || display.Text == "0" )
      {
        Result = false;
        display.Text = key;
      }
      else
        display.Text += key;
      break;
    case ".":                // Dezimalzeichen
    case ",":
      if (Result)
      {
        Result = false;
        display.Text = key;
      }
      else
        if (Decimal == false)
          display.Text += ",";
      Decimal = true;
      break;
    case "M":               // angezeigten Wert speichern
      if (display.Text == "" || display.Text == "-" )
        M = 0;
      else
        M = Convert.ToDouble(display.Text);
      break;
```

```
case "R":              // gespeicherten Wert displayn
  display.Text = M.ToString();
  X = M;
  break;
case "C":              // Beim ersten Mal Clear, dann Clear All
  Decimal = false;
  if (display.Text == "")
  {
    X = M = 0;
    op = Ops.None;
  }

  else
    display.Text = "";
  break;
case "±":              // Vorzeichenswechsel
  if (display.Text.Length >=1)
  {
    if (display.Text.Substring(0,1) == "-")
      display.Text = display.Text.Substring(1);
    else
      display.Text = "-" + display.Text;
  }
  else
    display.Text = "-" + display.Text;
  break;
                       // die vier Operationen
case "+":
  this.Calculate("="); // Zwischenergebnis berechnen
  op = Ops.Add;        // Operation setzen
  break;
case "-":
  this.Calculate("="); // Zwischenergebnis berechnen
  op = Ops.Subtract;   // Operation setzen
  break;
case "*":
  this.Calculate("="); // Zwischenergebnis berechnen
  op = Ops.Multiply;   // Operation setzen
  break;
case "/":
  this.Calculate("="); // Zwischenergebnis berechnen
  op = Ops.Divide;     // Operation setzen
  break;
case "=":              // Ergebnis oder Zwischenergebnis darstellen
  try
  {
    double Y = Convert.ToDouble(display.Text);
    if (Result == false)   // Falls zweimal Operation in Folge
    {                      // keine Berechnung, neue Operation
                           // übernehmen
      switch (op)          // Operation ausführen
      {
        case Ops.None:     X = Y;  break;
        case Ops.Add:      X += Y; break;
        case Ops.Subtract: X -= Y; break;
```

```
            case Ops.Multiply: X *= Y; break;
            case Ops.Divide:   X /= Y; break;
          }
          op = Ops.None;
          display.Text = X.ToString();
          Result = true;      // nächste Ziffer löscht Display
          Decimal = false;    // Komma erlauben
        }
      }
      catch                   // sollte nicht vorkommen, aber
      {                       // sicher ist sicher
        display.Text = "Error";
        X = 0;
        Decimal = false;
        Result = true;
      }
      break;
  }
  abzeige.Select();               // Fokus auf Display
  display.SelectionStart = display.Text.Length;   // Schreibmarke
}

#region Windows Form Designer generated code
private void InitializeComponent()
{
  this.display = new System.Windows.Forms.TextBox();
  this.SuspendLayout();
  //
  // display
  //
  this.display.BackColor = System.Drawing.Color.White;
  this.display.Font = new System.Drawing.Font("Microsoft Sans Serif",
    24F, System.Drawing.FontStyle.Regular,
    System.Drawing.GraphicsUnit.Point,
    ((System.Byte)(0)));
  this.display.Location = new System.Drawing.Point(24, 0);
  this.display.Name = "display";
  this.display.Size = new System.Drawing.Size(416, 44);
  this.display.TabIndex = 0;
  this.display.Text = "";
  this.display.TextAlign = HorizontalAlignment.Right;
  this.display.KeyPress += new
    System.Windows.Forms.KeyPressEventHandler(this.display_KeyPress);
  //
  // Form1
  //
  this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
  this.ClientSize = new System.Drawing.Size(480, 324);
  this.Controls.AddRange(new Control[] {
                                        this.display});
  this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.Fixed3D;
  this.Name = "Form1";
  this.Text = "Taschenrechner";
  this.ResumeLayout(false);

}
```

```
    #endregion

    /// <summary>
    /// Der Haupteinstiegspunkt für die Anwendung.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }
    private void display_KeyPress(object sender, KeyPressEventArgs e)
    {
      e.Handled = true;
      Calculate(e.KeyChar.ToString());
    }
  }
}
```

Die Logik des nach dem Prinzip des *endlichen Automaten* implementierten Rechners kommt mit sechs Zustandsvariablen aus. Das Register für den ersten Operanden ist schlicht die `Text`-Eigenschaft des `TextBox`-Steuerelements `display`, das für den zweiten ein Datenfeld namens `X`. Zur Beschreibung der möglichen Operationen gibt es den `enum`-Typ `Ops` und natürlich ein Datenfeld dieses Typs, das den jeweils anzuwendenden Operator – der Rechner hat natürlich Infixnotation – zwischenspeichert. Außer dem Register `M` sind noch zwei Statusfelder erforderlich: Das Flag `Decimal` verhindert die mehrfache Eingabe des Dezimalzeichens, und das Flag `Result` signalisiert, dass ein Operandenwechsel nach Eingabe eines Operators stattfindet.

Die letzten beiden Anweisungen in der `Calculate()`-Methode sorgen dafür, dass die Einfügemarke im Textfeld an der richtigen Stelle bleibt und der Fokus bei Verwendung der Mausschnittstelle nicht an eine `Button`-Schaltfläche verloren geht.

```
abzeige.Select();               // Fokus auf Anzeige
anzeige.SelectionStart = anzeige.Text.Length;   // Schreibmarke
```

Die Geschichte mit dem Fokus lässt sich auch professioneller lösen. Sie leiten eine Klasse `MyButton` von `Button` ab und setzen in den Konstruktor nichts weiter als einen Aufruf der `protected`-Methode `SetStyle()`, um das `Selectable`-Flag des Steuerelements abzuschalten.

```
class MyButton : Button
{
  public MyButton()
  {
    this.SetStyle(ControlStyles.Selectable, false);
  }
}
```

Nun können Sie `MyButton` als Datentyp für die Schaltflächen vereinbaren. Da die `CanSelect`-Eigenschaft einer `MyButton`-Schaltfläche den Wert `false` hat, behält das Textfeld den Fokus auch bei der Mauseingabe.

Übung

Erweitern Sie den Taschenrechner zu einem wissenschaftlichen Taschenrechner, der die wichtigsten arithmetischen und trigonometrischen Funktionen des täglichen Lebens bereitstellt. Alles, was Sie dazu brauchen, finden Sie in der Klasse `System.Math`. Achten Sie beim Design darauf, dass die Tastaturschnittstelle möglichst erhalten bleibt.

15.2 Wichtige Steuerelemente der Toolbox

Die folgenden Abschnitte stellen die Steuerelemente aus der WINDOWS FORMS-Auflistung der TOOLBOX kurz in ihren Besonderheiten vor. Am besten lernen Sie die Elemente allerdings interaktiv kennen, indem Sie schlicht ein wenig damit experimentieren.

15.2.1 Button, CheckBox, RadioButton

Die abstrakte Klasse `ButtonBase` bildet die Grundlage für Kontrollkästchen, Optionsfelder und gewöhnliche Schaltflächen.

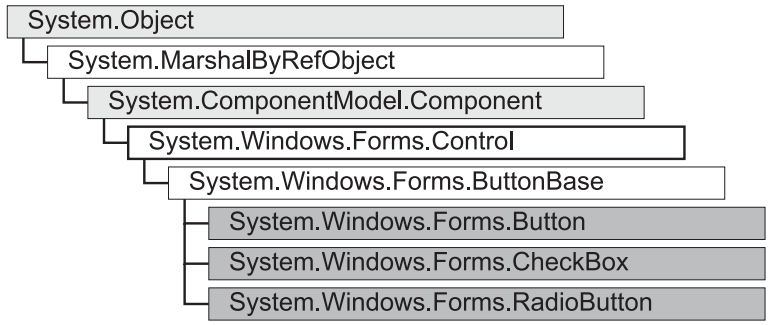


Abbildung 15.4: Ableitung von `Button`, `CheckBox` und `RadioButton`

Eigenschaft	Bedeutung
`FlatStyle` **`FlatStyle`** `{get; set;}`	Stil des Steuerelements. Zur Auswahl stehen `Standard`, `Flat`, `Popup` und `System` (für System generiert das Objekt keine Paint-Ereignisse).
`Image` **`Image`** `{get, set;}`	Hintergrund-Bitmap des Steuerelements. Erscheint nur, wenn `ImageList`-Eigenschaft an keine `ImageList`-Komponente gebunden ist.

Tabelle 15.5: Spezifische Eigenschaften der von `ButtonBase` abstammenden Steuerelemente

Tabelle 15.5: Spezifische Eigenschaften der von `ButtonBase` abstammenden Steuerelemente (Forts.)

Eigenschaft	Bedeutung
`ContentAlignment ImageAlign {get; set;}`	Ausrichtung der Hintergrund-Bitmap im Steuerelement.
`int ImageIndex {get, set;}`	Index für die `ImageList`-Eigenschaft, wählt die Hintergrund-Bitmap für das Steuerelement aus.
`ImageList ImageList {get; set;}`	`ImageList`-Komponente, die eine oder mehrere Hintergrund-Bitmaps für das Steuerelement bereitstellt. `ImageIndex` wählt die anzuzeigende Bitmap aus.
`protected bool IsDefault {get;}`	`true`, wenn die Schaltfläche Standardschaltfläche in einem Dialog ist

Button

Das `Button`-Steuerelement stellt eine einfache Schaltfläche dar, die Mausklicks des Benutzers als `Click`-Ereignis signalisiert. Sie finden zahlreiche Beispiele in diesem Buch, die mit Schaltflächen arbeiten. Um einen eigenen Schaltflächenstil zu realisieren, können Sie die `ImageXxx`-Eigenschaften der Schaltfläche verwenden und eine Hintergrund-Bitmap setzen oder das `Paint`-Ereignis behandeln. Die folgende Routine zeichnet ein schwarzes Dreieck in eine Schaltfläche:

```
private void button1_Paint(object sender, PaintEventArgs e)
{
  e.Graphics.FillPolygon(Brushes.Black, new Point[]{
        new Point(0, 0),
        new Point(button1.Width/2, button1.Height),
        new Point(button1.Width, 0) } );
}
```

Beachten Sie, dass die Routine nicht zum Aufruf kommt, wenn Sie die `FlatStyle`-Eigenschaft auf `FlatStyle.System` setzen.

Tabelle 15.6: Spezifische Eigenschaften des `Button`-Steuerelements

Eigenschaft	Bedeutung
`DialogResult DialogResult {get; set;}`	Beschreibt das Ergebnis eines modalen Dialogs, wenn dieser über die Schaltfläche verlassen wird.

Um die für dieses Steuerelement häufig verwendete `ImageList`-Eigenschaft einzusetzen, benötigen Sie eine `ImageList`-Komponente, die eine oder mehrere Bitmaps bereitstellt. Am einfachsten ist es, wenn Sie die Komponente mit dem Designer in den Formularentwurf einfügen und interaktiv mit den benötigten Bildern initialisieren. Der Designer fügt die Bilder dann automatisch der ausführbaren Datei als Ressourcen bei, was den Vorteil hat, dass

sie nicht als eigenständige Dateien mitgeschleppt werden müssen. Achten Sie darauf, dass die in der ImageList-Komponente eingestellten Bildabmessungen (in Bildpunkten) zur Schaltfläche passen. Um für die Zustände »nicht gedrückt« und »gedrückt« der Schaltfläche unterschiedliche Bitmaps der ImageList-Komponente anzuzeigen, behandeln Sie die Ereignisse MouseDown und MouseUp für das Steuerelement:

```
...
this.button1.ImageList = imageList1;
...
private void button1_MouseDown(object sender, MouseEventArgs e)
{
  button1.ImageIndex = 0;
}
private void button1_MouseUp(object sender, MouseEventArgs e)
{
  button1.ImageIndex = 1;
}
```

CheckBox

Das CheckBox-Steuerelement ist nichts anderes als das gute alte Kontrollkästchen. Im Gegensatz zum Button-Steuerelement, dessen Äußeres das CheckBox-Steuerelement optional annehmen kann, wenn die Appearance-Eigenschaft auf Appearance.Button gesetzt ist, repräsentiert es dauerhaft einen zwei- oder dreiwertigen Zustand, der sich über die Eigenschaft Checked bzw. CheckState abfragen und setzen lässt. Die Umschaltung der Benutzerschnittstelle zwischen zwei- und dreiwertig erfolgt über die Eigenschaft ThreeState. Bei Änderungen von Checked und CheckState signalisiert das Steuerelement die Ereignisse CheckedChanged und CheckStateChanged. Im Allgemeinen werden Sie nur eines dieser beiden Ereignisse behandeln, da CheckStateChanged immer zusammen mit CheckedChanged auftritt.

Tabelle 15.7: Wichtige Eigenschaften des CheckBox-Steuerelements

Eigenschaft	Bedeutung
`Appearance Appearance {get; set;}`	`Normal` – das Steuerelement zeichnet das Kontrollkästchen in unterschiedlichen Varianten (mit/ohne Häkchen, grau mit Häkchen); `Button` – das Steuerelement lässt das Kontrollkästchen weg und zeichnet eine Schaltfläche in drei Zuständen. In den markierten Zuständen erscheint die Schaltfläche dauerhaft gedrückt.
`bool AutoCheck {get; set;}`	Wenn `true`, wechselt das Steuerelement seinen Zustand automatisch bei jedem Mausklick.
`ContentAlignment CheckAlign {get; set;}`	Ausrichtung des Kontrollkästchens im Fensterbereich des Steuerelements (es gibt neun mögliche Ausrichtungen, je drei in jeder Dimension)

Tabelle 15.7: Wichtige Eigenschaften des CheckBox-Steuerelements (Forts.)

Eigenschaft	Bedeutung
`bool Checked {get; set;}`	Zweiwertiger Zustand des Kontrollkästchens
`CheckState CheckState {get; set;}`	Dreiwertiger Zustand des Kontrollkästchens; mögliche Werte sind `Unchecked`, `Checked`, `Indeterminate`
`bool ThreeState {get; set;}`	Wenn `true`, kann der Benutzer das Steuerelement per Mausklick in alle drei möglichen Zustände von `CheckState` setzen, ansonsten nur in die beiden Zustände von `Checked`.

Tabelle 15.8: Wichtige Ereignisse des CheckBox-Steuerelements

Ereignis	Bedeutung
`CheckedChanged`	`Checked`-Eigenschaft geändert (vom Programm oder durch einen Klick des Benutzers)
`CheckStateChanged`	`CheckState`-Eigenschaft geändert (vom Programm oder durch einen Klick des Benutzers)

RadioButton

Hinter dem `RadioButton`-Steuerelement versteckt sich das altbekannte Optionsfeld, das amüsanterweise mit .NET wieder zu seiner ursprünglichen, namensgebenden Bezeichnung zurückgefunden hat. Ein »Radioknopf« ist Teil einer Gruppe von Knöpfen, die sich gegenseitig auslösen, so dass immer nur ein Knopf gedrückt sein kann – wie eben bei den Stationstasten eines altertümlichen Radios.

Im Gegensatz zum Kontrollkästchen ist das Optionsfeld kein Einzelkämpfer, sondern agiert immer im Kontext weiterer Optionsfelder. Es gilt:

1. Alle einem Container direkt zugeordneten Optionsfelder bilden eine *Gruppierung*. Optionsfelder in verschiedenen Containern (gleich ob verschachtelt oder nebeneinander gelegen) gehören verschiedenen Gruppierungen an.
2. Alle Optionsfelder einer Gruppierung lösen sich gegenseitig aus. In einer Gruppierung kann maximal immer nur die `Checked`-Eigenschaft eines Optionsfelds `true` sein. Markiert der Benutzer (oder das Programm) ein anderes Optionsfeld, erscheint dieses markiert und alle anderen unmarkiert.



Es ist möglich, eine Optionsgruppe so zu initialisieren, dass keines der Optionsfelder markiert erscheint. Der Benutzer kann diesen Zustand jedoch nicht mehr herstellen: Nach dem ersten Markieren eines Optionsfeldes bleibt immer ein Optionsfeld der Gruppe markiert.

Eigenschaft	Bedeutung
`Appearance Appearance {get; set;}`	`Normal` – das Steuerelement zeichnet das Optionsfeld als Kreis bzw. Kreis mit Punkt. `Button` – das Steuerelement zeichnet keine Kreise, sondern eine Schaltfläche mit zwei Zuständen. Im markierten Zustand erscheint die Schaltfläche dauerhaft gedrückt
`bool AutoCheck {get; set;}`	Wenn `true`, wechselt das Steuerelement seinen Zustand automatisch, andernfalls muss dies per Code geschehen.
`ContentAlignment CheckAlign {get; set;}`	Ausrichtung des Optionsfelds im Fensterbereich des Steuerelements (es gibt neun mögliche Ausrichtungen, je drei in jeder Dimension)
`bool Checked {get; set;}`	Wenn `true`, wird das Optionsfeld markiert gezeichnet.

Tabelle 15.9: Wichtige Ereignisse des `Radiobutton`-Steuerelements

Ereignis	Bedeutung
`CheckedChanged`	`Checked`-Eigenschaft wurde von Programm, vom Benutzer oder vom Container geändert

Tabelle 15.10: Das zentrale Ereignis des `RadioButton`-Steuerelements

Wenn Sie von Visual Basic 6 oder Delphi her kommen, werden Sie an dieser Stelle vielleicht ein Feature schmerzlich vermissen: das Optionsfeld-Array und das darauf zugeschnittene Ereignis, das den Index des jeweils markierten Optionsfelds bereitstellt. Das Analogon in C# ist jedoch schnell zusammengezimmert:

1. Packen Sie die Optionsfelder Ihrer Gruppierung in die `Controls`-Auflistung eines `GroupBox`- oder `Panel`-Steuerelements – und zwar in der Reihenfolge, die Sie für die Indizierung wünschen. Falls wirklich noch weitere Steuerelemente in denselben Container müssen, fügen Sie diese erst danach ein. Die hier vorgestellte Routine erwartet einen Container, ein Array mit den Beschriftungen und eine gemeinsame Behandlungsroutine. Sie generiert für jede Beschriftung ein Optionsfeld, ordnet die Optionsfelder gleichmäßig untereinander im Container an und registriert die Behandlungsmethode dafür.
2. Registrieren Sie für alle Optionsfelder der Gruppierung eine gemeinsame Behandlungsmethode für das `CheckedChanged`-Ereignis, die den Index des Steuerelements ermittelt.

Hier ein Muster für den Code:

```
private void InitRadioButtons(Control GroupingControl,
  string[]Texts, EventHandler e)
{
  for (int i = 0; i < Texts.Length; i++)
  {
    RadioButton r = new RadioButton();
    r.Text = Texts[i];
    r.Left = 10;
    r.Top = (int)((0.5 + i) *
            (GroupingControl.Height / Texts.Length) - r.Height/2);
    r.CheckedChanged += e;
    GroupingControl.Controls.Add(r);
  }
}

private void ButtonGroup_CheckedChanged(object sender, System.EventArgs e)
{
  RadioButton r = (RadioButton) sender;
  DoRadio (r.Parent.Controls.GetChildIndex((Control)sender));
}
private void DoRadio(int Index)
{
  // TODO RadioButton-Aktionen Implementieren
}
```

Der Aufruf sieht so aus:

```
...
this.InitializeComponent();
InitRadioButtons(this.groupBox1,
  new string[] {"Option 1", "Option 2", "Option 3", "Option 4"},
  new System.EventHandler(this.ButtonGroup_CheckedChanged));
...
```

Eine bessere Lösung wäre natürlich ein Steuerelement. Eine Implementierung finden Sie im Projekt *RadioGroup*, eine zweite in der Bibliothek *Benutzersteuerelemente*.

15.2.2 Label und TextBox

Während das Beschriftungssteuerelement `Label` reichlich unspektakulär ist und sich in seinen Möglichkeiten weitgehend über das EIGENSCHAFTEN-Fenster erschließt, wartet das Textfeld `TextBox` mit einer Reihe interessanter Eigenschaften und Methoden auf, die seinen Funktionsumfang charakterisieren und den Umgang speziell mit der zentralen `Text`-Eigenschaft erleichtern. Tabelle 15.10 und Tabelle 15.12 stellen sie vor.

Ein interessantes Codebeispiel, das den Umgang mit einem Textfeld vorführt, ist das Steuerelement `MemTextBox`. Sie finden es im Abschnitt »Bestehende Steuerelemente modifizieren« ab Seite 656.

Tabelle 15.11: Besondere Eigenschaften von Text- und Beschriftungsfeldern

Eigenschaft	Bedeutung
`bool AcceptsReturn {get; set;}`	Wenn `true`, gibt ein mehrzeiliges Textfeld den Fokus nicht ab, wenn der Benutzer ↵ drückt, sondern setzt einen Zeilenumbruch ein.
`bool AcceptsTab {get; set;}`	Wenn `true`, gibt ein mehrzeiliges Textfeld den Fokus nicht ab, wenn der Benutzer ⇆ drückt, sondern setzt ein Tabulatorzeichen ein.
`bool AutoSize {get; set;}`	Wenn `true`, passt das Steuerelement seine Abmessungen automatisch so an, dass die `Text`-Eigenschaft (nur die erste Zeile) exakt in den Fensterbereich passt. Eine Veränderung der `Font`-Eigenschaft bewirkt damit eine Größenänderung des Steuerelements. Fixpunkt ist die linke obere Ecke – die `Anchor`-Eigenschaft wird also leider nicht beachtet.
`CharacterCasing CharacterCasing {get; set;}`	Einer der drei `enum`-Werte `Lower`, `Upper` oder `Normal`; Bestimmt, ob das Textfeld Eingaben gegebenenfalls in Groß- oder Kleinschreibung umsetzt.
`bool CanUndo {get;}`	Wenn `true`, ist der Undo-Puffer des Steuerelements besetzt; der Benutzer kann seine letzte Eingabe über Strg+Z rückgängig machen. Sinnvoll für die Aktivierung/Deaktivierung eines Menübefehls BEARBEITEN/RÜCKGÄNGIG im Programm.
`string[] Lines {get; set;}`	Dieses Array ermöglicht es, einzelne Textzeilen zu manipulieren. Ist `Multiline` auf `false` gesetzt, enthält das Array ein einziges Element mit dem Wert von `Text`.
`int MaxLength {get; set;}`	Beschränkung für die Länge der `Text`-Eigenschaft. Die Standardvorgabe 0 entspricht »unendlich« (unter Windows 9x/ME 32 KByte, unter Windows NT/XP/2000 nur durch den verfügbaren virtuellen Speicher begrenzt)
`bool Multiline {get; set;}`	Wenn `true`, kann das Textfeld auch mehrzeilige Eingaben darstellen und verarbeiten. Die `Lines`-Eigenschaft repräsentiert dann die einzelnen Zeilen.
`char PasswordChar {get; set;}`	Wenn Sie hier ein anderes Zeichen als `(char)0` zuweisen – beispielsweise ein Sternchen (»*«) – werden Eingaben nur noch mit diesem Zeichen dargestellt. Vorgesehen ist diese Eigenschaft für die Eingabe von Kennwörtern. Sie wird nicht beachtet, wenn `Multiline` auf `true` gesetzt ist.

Tabelle 15.11: Besondere Eigenschaften von Text- und Beschriftungsfeldern (Forts.)

Eigenschaft	Bedeutung
`int PreferredHeight {get; }`	Liefert die Höhe in Bildpunkten für den Fensterbereich des Steuerelements, die für die Darstellung in der gesetzten Schriftart mindestens erforderlich ist (und berücksichtigt einen eventuellen Rahmen).
`bool Readonly {egt; set;}`	Wenn `true`, kann die `Text`-Eigenschaft nicht interaktiv bearbeitet werden.
`ScrollBars ScrollBars {get; set;}`	Bestimmt, ob das Textfeld bei mehrzeiliger Anzeige Bildlaufleisten einblendet und automatisch verwaltet. Mögliche Werte sind `Both`, `Horizontal`, `None` (Vorgabe) und `Vertical`.
`string SelectedText {get; set;}`	Markierter Teilstring der `Text`-Eigenschaft
`int SelectionLength {get; set;}`	Länge der Markierung
`int SelectionStart {get; set;}`	Index des ersten markierten Zeichens der `Text`-Eigenschaft
`bool WordWrap {get; set;}`	Wenn `true` (Standardvorgabe für mehrzeilige Textfelder), wird ein automatischer Zeilenumbruch des Texts vorgenommen. Bei einzeiligen Textfeldern ignoriert.

Tabelle 15.12: Besondere Methoden von Text- und Beschriftungsfeldern

Methode	Bedeutung
`void AppendText()`	Hängt einen `string`-Wert an den Text an.
`void Clear()`	Setzt `Text` auf einen Leerstring zurück.
`void ClearUndo()`	Löscht den Undo-Puffer des Textfelds.
`void Cut()`	Schneidet die Markierung aus und setzt sie in die Zwischenablage ein, entspricht dem Tastenbefehl Strg+X.
`void Paste()`	Überträgt den Inhalt der Zwischenablage in die `Text`-Eigenschaft, entspricht den Tastenbefehl Strg+V.
`void ScrollToCaret()`	Sorgt dafür, dass in einem mehrzeiligen Textfeld die Zeile mit der Schreibmarke angezeigt wird.
`void Select()`	Markiert einen Teilstring des Texts.
`void SelectAll()`	Markiert den gesamten Text.

15.2.3 ListBox und ComboBox

Die Steuerelemente `ListBox` und `ComboBox` sind sich in ihrer Funktionsweise sehr ähnlich. Deshalb werden sie hier zusammen vorgestellt. Der wesentliche Unterschied zwischen den beiden ist, dass das Kombinationsfeld neben einer aufklappbaren Liste über ein Textfeld verfügt. Das Textfeld ermöglicht gewöhnliche Texteingaben sowie die Anzeige des jeweils ausgewählten Listeneintrags. Das Listenfeld hingegen zeigt immer eine bestimmte Menge an Listeneinträgen an, ermöglicht dafür aber die Mehrfachauswahl (`SelectionMode`-Eigenschaft) sowie die spaltenorientierte Anzeige (`MultiColumn`). Die nur bei `ComboBox` vorhandene `Text`-Eigenschaft gibt den `string`-Wert wieder, die bei beiden Steuerelementen verfügbare `SelectedIndex`-Eigenschaft den Index des aktuell ausgewählten Listeneintrags.

Beide Steuerelemente zeigen in der Standardkonfiguration eine über die Eigenschaft `Items` verwaltete, optional auch (über `Sorted`) sortierbare Liste mit Texteinträgen an, in der der Benutzer seine Auswahl per Maus oder Richtungstasten treffen kann. `Items` trägt den Datentyp `ObjectCollection`, so dass sich jeder beliebige Datentyp, und nicht nur `string`, in der Liste verwalten lassen.

Über die `DataSource`-Eigenschaft kann auch eine andere Datenquelle gebunden werden (wobei die verwendete Klasse die Schnittstelle `IList` implementieren muss), deren Elemente aber nicht veränderbar sind. Für komplexe Elementobjekte lassen sich über die Eigenschaften `DisplayMember` und `ValueMember` die Namen zweier (nicht notwendig verschiedener) Element-Eigenschaften benennen, wobei die an `DisplayMember` gebundene Eigenschaft angezeigt wird und die an `ValueMember` gebundene Eigenschaft den Wert liefert.

Standardmäßig zeichnet das Listenfeld seine Einträge selbst und bedient sich dann der `ToString()`-Methode, um beliebige Objekttypen textuell darzustellen. Alternativ kann der Besitzer des Steuerelements (im Allgemeinen die Formularklasse) das Zeichnen der Listeneinträge selbst übernehmen. Dazu setzt er die `DrawMode`-Eigenschaft auf `DrawMode.OwnerDrawFixed` oder `OwnerDrawVariable.OwnerDrawVariable` und muss dann das `DrawItem`-Ereignis behandeln und dabei den Eintrag selbst zeichnen. Dies ist in der Praxis weniger aufwändig als es klingen mag, da das Ereignisobjekt einen recht komfortablen Kontext mitliefert. Listeneinträge mit einheitlicher Höhe werden im Modus `OwnerDrawFixed` gezeichnet. Die Höhe ist über die Eigenschaft `ItemHeight` definierbar. Listeneinträge mit unterschiedlicher Höhe werden hingegen im Modus `OwnerDrawVariable` gezeichnet, wobei das Steuerelement für jedes Listenelement ein `MeasureItem`-Ereignis generiert, das der Besitzer behandeln sollte, um von `ItemHeight` abweichende Höhen elementweise zu setzen.

Tabelle 15.13: Die wichtigsten Eigenschaften der Steuerelemente `ListBox` und `ComboBox`

Eigenschaft	Bedeutung
`int ColumnWidth {get; set;}`	Nur `ListBox`: Gibt die Breite einer Spalte in Bildpunkten für die mehrspaltige Anzeige an, wenn `MultiColumn` auf `true` gesetzt ist. Hat die Eigenschaft den Wert 0, benutzt das Listenfeld den Standardwert 120.
`object DataSource {get; set;}`	Wenn `null`, listet das Steuerelement die Elemente der Eigenschaft `items` auf. Ist dieser Eigenschaft hingegen ein auflistbares (`IList` implementierendes) Objekt zugeordnet, fungiert dieses als Datenquelle. Falls das Listenfeld von Windows gezeichnet wird, können Sie über die Eigenschaften `DisplayMember` und `ValueMember` getrennt eine anzuzeigende und eine wertliefernde Eigenschaft des Datenquelleobjekts festlegen.
`string DisplayMember {get; set;}`	Name derjenigen Eigenschaft des Elementobjekts (gleich ob von `items` oder von `DataSource` geliefert), deren Wert das Steuerelement als Listeneintrage anzeigt. Wenn `""`, wird angezeigt, was die `ToString()`-Methode des Elementobjekts liefert.
`DrawMode DrawMode {get; set;}`	`enum`-Wert, der den Zeichenmodus für das Steuerelement festlegt. Wenn `Normal`, zeichnet das Steuerelement die Listeneinträge unter Anwendung der `ToString()`-Methode auf die Elementobjekte der `items`-Auflistung (oder `DataSource`-Datenquelle) selbst. Wenn `OwnerDrawFixed`, zeichnet der Besitzer die Einträge mit der festen Höhe `ItemHeight` als Antwort auf das `DrawItem`-Ereignis. Wenn `OwnerDrawVariable`, zeichnet der Besitzer die Einträge mit variabler Höhe (nur einspaltig möglich), wobei er die Höhe elementweise in Antwort auf das `MeasureItem`-Ereignis bekannt gibt.
`ComboBoxStyle DropDownStyle {get; set;}`	Nur `ComboBox`: `enum`-Wert, der die Darstellung des Listenfelds bestimmt: Bei `Simple` ist das Listenfeld ständig sichtbar; bei der Standardvorgabe `DropDown` ist es aufklappbar und der Benutzer kann beliebige Werte in das Textfeld eintippen; bei `DropDownList` ist das Listenfeld aufklappbar, und der Benutzer kann ausschließlich Werte aus der anhängenden Liste in das Textfeld auswählen bzw. Eingaben in das Textfeld wählen automatisch immer ein Element der Liste aus. Andere Eingaben sind nicht möglich.

Tabelle 15.13: Die wichtigsten Eigenschaften der Steuerelemente `ListBox` und `ComboBox` (Forts.)

Eigenschaft	Bedeutung
`int DropDownWidth {get; set;}`	Nur `ComboBox`: Breite des Listenfelds in Bildpunkten.
`int HorizontalExtent {get; set;}`	Nur `ListBox`: Anzahl der Bildpunkte, um die das Steuerelement die Ansicht beim horizontalen Bildlauf maximal verschiebt. Wenn 0, gibt es keine Begrenzung. Diese Eigenschaft wirkt sich nur aus, wenn `HorizontalScrollbar` den Wert `true` hat.
`bool HorizontalScrollbar {get; set;}`	Nur `ListBox`: Wenn `true`, zeigt das Steuerelement erforderlichenfalls eine horizontale Bildlaufleiste an, damit der Benutzer ansonsten abgeschnittene Bereiche einsehen kann. Diese Eigenschaft wirkt sich nur aus, wenn `MultiColumn` auf `false` gesetzt ist.
`bool IntegralHeight {get; set;}`	Wenn `true`, zeigt das Steuerelement in seinem Fenster nur so viele Einträge an, wie der Höhe nach vollständig gezeichnet werden können. Ansonsten erscheint der unterste dargestellte Eintrag gegebenenfalls vertikal abgeschnitten.
`int ItemHeight {get; set;}`	Anzahl der Bildpunkte für einen Listeneintrag, wenn das Besitzerobjekt die Einträge selbst zeichnet. Liefert auch den Vorgabewert für das Zeichnen im Modus `OwnerDrawVariable`.
`ObjectCollection Items {get; set;}`	Auflistung, in der das Steuerelement die Objekte für die Listeneinträge speichert. Die Elemente dieser Eigenschaft werden nur angezeigt, wenn keine Datenquelle gebunden, d.h. `DataSource` auf `null` gesetzt ist.
`int MaxDropDownItems {get; set;}`	Nur `ComboBox`: Anzahl der Elemente, die im Listenanhang des Steuerelements maximal angezeigt werden. Gibt es mehr Einträge, als angezeigt werden können, blendet das Steuerelement unaufgefordert eine Bildlaufleiste ein.
`bool MultiColumn {get; set;}`	Nur `ListBox`: Wenn `true`, zeigt das Steuerelement seine Listeneinträge spaltenweise an. Falls nicht alle Spalten vollständig angezeigt werden können, blendet es eine Bildlaufleiste ein und ermöglicht den horizontalen Bildlauf. Die Spaltenbreite wird von der Eigenschaft `ColumnWidth` bestimmt.
`int SelectedIndex {get; set;}`	Index des aktuell markierten Listeneintrags. –1 steht für »keine Auswahl« (leeres Textfeld).

Tabelle 15.13: Die wichtigsten Eigenschaften der Steuerelemente `ListBox` und `ComboBox` (Forts.)

Eigenschaft	Bedeutung
`SelectedIndexCollection SelectedIndices {get; set;}`	Nur `ListBox`: Auflistung der Indizes aller markierten Listeneinträge (vgl. `SelectionMode`)
`SelectionMode SelectionMode {get; set;}`	Nur `ListBox`: `enum`-Wert, der festlegt, wie der Benutzer Listeneinträge markieren kann. Zur Auswahl stehen: `None` (keine Auswahl möglich), `One` (nur ein Eintrag markierbar), `MultiSimple` (mehrere aufeinanderfolgende Einträge markierbar), `MultiExtended` (Einträge in beliebiger Folge markierbar).
`bool Sorted {get; set;}`	Wenn `true`, sortiert das Steuerelement seine `items`-Auflistung. Bei Verwendung einer an `DataSource` gebundenen Datenquelle muss diese Eigenschaft auf `false` gesetzt sein.
`string Text {get; set;}`	Bei `ListBox`: String-Repräsentation des zuletzt markierten Listeneintrags; bei `ComboBox`: Wert des Textfelds (der dort beim Stil `DropDown` nicht unbedingt mit einem der Listeneinträge identisch sein muss).
`string ValueMember {get; set;}`	Name derjenigen Eigenschaft des Elementobjekts (gleich, ob von `items` oder von `DataSource` geliefert), deren Wert bei Auswahl eines Listenelements der Text-Eigenschaft zugeordnet wird. Wenn `""`, erhält die `Text`-Eigenschaft den Wert, welchen die `ToString()`-Methode des zuletzt markierten Elementobjekts liefert.

Tabelle 15.14: Die wichtigsten Methoden der Steuerelemente `ListBox` und `ComboBox`

Methode	Bedeutung
`void BeginUpdate()`	Steuerelement zeichnet sich bis zum Aufruf von `EndUpdate()` nicht mehr. Sollte vor größeren Veränderungen der Liste des Steuerelements verwendet werden.
`void EndUpdate()`	Gegenstück von `BeginUpdate()`
`void ClearSelected()`	Löscht die `SelectedIndices`-Auflistung des Steuerelements und setzt `SelectedIndex` auf -1.
`int FindString()`	Sucht nach dem ersten oder nächsten Listeneintrag, der mit dem angegebenen Teilstring beginnt und liefert dessen Index.
`int FindStringExact()`	Sucht nach dem ersten oder nächsten Listeneintrag, der mit dem angegebenen Teilstring übereinstimmt und liefert dessen Index.

Methode	Bedeutung
`Rectangle GetItemRectangle()`	Liefert den Fensterbereich, in dem das Element mit dem angegebenen Index liegt. Ist das Element nicht sichtbar, liegt das Rechteck nicht im Clip-Bereich des Listenfensters.
`int GetItemHeight()`	Liefert die Höhe des Elements mit dem angegebenen Index in Bildpunkten.
`string GetItemText()`	Liefert die `ToString()`-Darstellung des Elements mit dem angegebenen Index.
`int IndexFromPoint()`	Liefert den Index des Elements, das an dem angegebenen Bildpunkt im Koordinatensystem des Steuerelements liegt. Das Element muss dazu nicht sichtbar sein.
`void SetSelected()`	Erweitert die `SelectedIndices`-Auflistung und aktualisiert die `SelectedIndex`-Eigenschaft.

Tabelle 15.14: Die wichtigsten Methoden der Steuerelemente `ListBox` und `ComboBox` (Forts.)

Ereignis	Bedeutung
`DrawItem`	Tritt in den Modi `OwnerDrawFixed` und `OwnerDrawVariable` auf, wann immer ein Listeneintrag neu gezeichnet werden muss.
`MeasureItem`	Tritt nur im Modus `OwnerDrawVariable` auf und ermöglicht es, die Höhe für jeden Listeneintrag einzeln einzustellen.
`SelectedIndexChanged`	Tritt auf, wenn der Benutzer einen anderen Listeneintrag markiert hat.
`SelectedValueChanged`	Tritt auf, wenn der Benutzer einen Listeneintrag mit einem anderen Elementwert markiert hat.
`DisplayMemberChanged`	Tritt auf, wenn die `DisplayMember`-Eigenschaft geändert wurde.
`ValueMemberChanged`	Tritt auf, wenn die `ValueMember`-Eigenschaft geändert wurde.

Tabelle 15.15: Die wichtigsten Ereignisse der Steuerelemente `ListBox` und `ComboBox`

Codebeispiel – FormSimple und FormOwnerDraw

Das Codebeispiel *ListBoxDemo* demonstriert die wichtigsten Funktionalitäten der Steuerelemente `ListBox` und `ComboBox`:

- Zeichnen durch den Besitzer
- mehrspaltige Anzeige
- Einsatz der `DataSource`-Eigenschaft

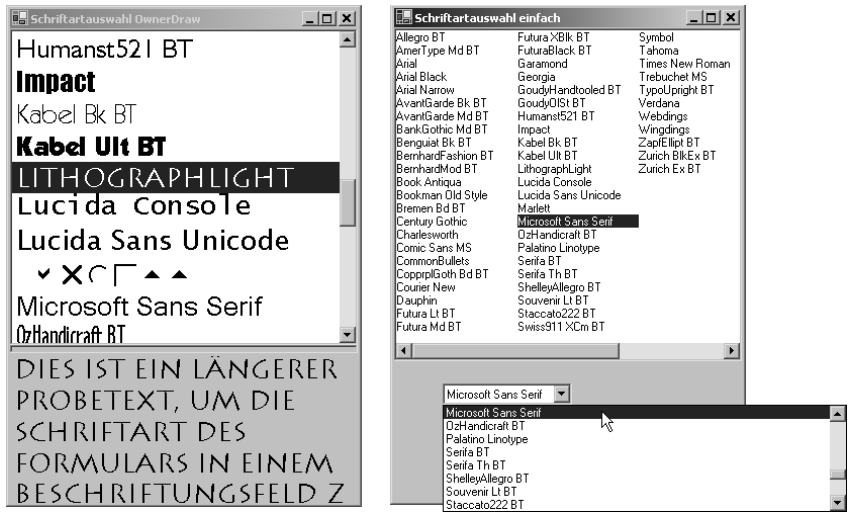


Abbildung 15.5: Die beiden Formulare der Anwendung

Die Methode `Main()` öffnet zwei Formulare, von denen das eine – `FormOwnerDraw` – ein `ListBox`-, ein `Splitter`- und ein `Label`-Steuerelement enthält. (Das `Splitter`-Steuerelement ermöglicht es, den Fensterbereich verschieden an die angedockten anderen beiden Steuerelemente aufzuteilen.) Das andere Formular trägt seinen Namen `FormSimple` zu Recht: es enthält lediglich ein `ListBox`- und ein `ComboBox`-Steuerelement.

```
static void Main()
{
   Form f = new FormOwnerDraw();
   f.Show();
   Application.Run(new FormSimple());
}
```

FormOwnerDraw

Beide Formulare zeigen Auflistungen der Systemschriftarten und ermöglichen die Auswahl. `FormOwnerDraw` macht seinem Namen alle Ehre und zeichnet jeden Schriftarteintrag in der jeweiligen Schriftart und der dafür erforderlichen Höhe. Das Beschriftungsfeld zeigt einen Probetext mit der jeweils ausgewählten Schrift an. Hier die Initialisierung des Steuerelements:

```
private void FormOwnerDraw_Load(object sender, System.EventArgs e)
 {
   listBox1.DrawMode = DrawMode.OwnerDrawVariable;
   // Alle im System installierten Schriften in die Liste aufnehmen
   foreach (FontFamily ff in FontFamily.GetFamilies(CreateGraphics()))
   {
     try   // nicht alle Fonts unterstützen FontStyle.Regular
     {
       System.Drawing.Font f = new System.Drawing.Font(ff, 20);
       listBox1.Items.Add(f);
```

```
      if (f.Name == Font.Name)
        listBox1.SelectedIndex = listBox1.Items.IndexOf(f);
    }
    catch {}
  }
  label1.Text = "Dies ist ein längerer Probetext, um die Schriftart " +
    "des Formulars in einem Beschriftungsfeld zu kontrollieren.";
}
```

Da das Formularobjekt die Listeneinträge des Steuerelements selbst zeichnet, muss es Behandlungsmethoden für die Ereignisse `MeasureItem` und `DrawItem` bereitstellen.

```
private void listBox1_MeasureItem(object sender, MeasureItemEventArgs e)
{
  // Platzbedarf für Element errechnen
  e.ItemHeight = ((Font)listBox1.Items[e.Index]).Height;
  e.ItemWidth = listBox1.ColumnWidth;
}
```

Eine Unterscheidung zwischen markierten und nicht markierten Einträgen ist nicht nötig, da das Ereignisobjekt passende Methoden und Eigenschaften bereitstellt. Die Schriftart findet sich über den Index in der `Items`-Eigenschaft:

```
private void listBox1_DrawItem(object sender, DrawItemEventArgs e)
{
  // Einzelnes Element zeichnen
  Font f = (System.Drawing.Font)listBox1.Items[e.Index];
  e.DrawBackground();
  e.Graphics.DrawString(f.Name, f, new SolidBrush(e.ForeColor), e.Bounds );
}
```

Das Formular reagiert auf das Ereignis `SelectedIndexChanged` mit dem Setzen seiner `Font`-Eigenschaft auf die ausgewählte Schriftart. Da das Beschriftungsfeld für die `Font`-Eigenschaft den Vorgabewert verwendet, ändert sich dadurch auch dessen Schriftart:

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  this.Font = (Font) listBox1.Items[listBox1.SelectedIndex];
}
```

FormSimple

Das `ListBox`-Steuerelement von `FormSimple` zeichnet sich selbst, dafür aber gleich mehrspaltig. Das Listenfeld des `ComboBox`-Steuerelements weist die Standardhöhe auf, ist jedoch verbreitert. Beide Steuerelemente zeigen zu Programmbeginn die Schriftart des Formulars als Auswahl an. Das `ListBox`-Steuerelement verwendet seine mit `string`-Werten besetzte `Items`-Auflistung als Datenquelle; Das `ComboBox`-Steuerelement bezieht seine Werte unter Ver-

wendung der Eigenschaften `DataSource`, `DisplayMember` und `ValueMember` direkt aus der `GetFamilies`-Auflistung.

Die Steuerelemente können erst nach der Initialisierung synchronisiert werden, daher kann die Verdrahtung ihrer `SelectedIndexChanged`-Ereignisse erst am Ende von `FormSimple_Load` geschehen. (Eine Alternative dazu wäre ein in `FormSimple_Load` gesetztes Flag, das bei der Behandlung dieser Ereignisse geprüft wird).

```
private System.Windows.Forms.ListBox listBox1;
private System.Windows.Forms.ComboBox comboBox1;
private void FormSimple_Load(object sender, System.EventArgs e)
{
  listBox1.DrawMode = DrawMode.Normal; // Steuerelement zeichnet sich
  listBox1.MultiColumn = true;
  // Alle im System installierten Schriften in die listBox1-Liste aufnehmen
  foreach (FontFamily ff in FontFamily.GetFamilies(CreateGraphics()))
  {
    listBox1.Items.Add(ff.Name);
    if (ff.Name == Font.Name)
      listBox1.SelectedIndex = listBox1.Items.IndexOf(ff.Name);
  }

  // Für comboBox1 muss die DataSource-Eigenschaft herhalten
  comboBox1.DataSource = FontFamily.GetFamilies(CreateGraphics());
  comboBox1.ValueMember = "Name";
  comboBox1.DisplayMember = "Name";
  comboBox1.SelectedIndex = comboBox1.FindStringExact(this.Font.Name);
  // Erst jetzt die Ereignisroutinen einhängen
  comboBox1.SelectedIndexChanged +=
    new System.EventHandler(comboBox1_SelectedIndexChanged);
  listBox1.SelectedIndexChanged +=
    new System.EventHandler(listBox1_SelectedIndexChanged);
}

private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  listBox1.SelectedIndex = comboBox1.SelectedIndex;
}

private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  comboBox1.SelectedIndex = listBox1.SelectedIndex;
}
```

15.2.4 Analoganzeigen: ScrollBar, ProgressBar, TrackBar

Zur Analoganzeige eines Wertes finden Sie in der TOOLBOX (wie sollte es auch anders sein) die vier Steuerelemente `VScrollBar`, `HScrollBar`, `ProgressBar` und `TrackBar` – zu deutsch: Bildlaufleisten, Fortschrittsleiste und Schieberegler. Obwohl nur `VScrollBar` und `HScrollBar` von einer gemeinsamen abstrakten

Basisklasse abstammen – und diese wiederum wie ProgressBar und TrackBar direkt von Control –, ist die Ausstattung der Steuerelemente doch recht ähnlich, weshalb sie hier in einen Topf geworfen werden. Der angezeigte Wert aller vier Steuerelemente ist die Value-Eigenschaft, während das Bezugsintervall durch die Eigenschaften Minimum und Maximum definiert wird. Die Fortschrittsanzeige wäre damit bereits vollständig erläutert, da sie als reine Anzeige keine Benutzerschnittstelle hat. Die anderen in Tabelle 15.16 erläuterten Eigenschaften regeln die Darstellung und die Reaktion der restlichen Steuerelemente auf Benutzeraktionen.

Werden die Steuerelemente VScrollBar und HScrollBar durch Setzen der TabStop-Eigenschaft für die Tastatureingabe »scharf« gemacht, lässt sich die Value-Eigenschaft auch über die Richtungstasten [→], [←], [↑], [↓] sowie die [Bild↑] und [Bild↓] verändern. Beide Steuerelementarten generieren zwei Ereignisse, die das Besitzerobjekt (gewöhnlich: das Formular) darüber informieren, dass sich Value verändert hat.

Tabelle 15.16: Wichtige Eigenschaften der Steuerelemente VScrollBar, HScrollBar, ProgressBar und TrackBar

Eigenschaft	Bedeutung
`int LargeChange {get; set;}`	TrackBar: Größe der Wertänderung durch die Tasten [↑] und [↓] sowie [Bild↑] und [Bild↓]. VScrollBar und HScrollBar: Größe der Wertänderung durch die Tasten [Bild↑] und [Bild↓] sowie bei Klicks in die Leiste. Zusätzlich legt dieser Wert die Breite der Bildlaufmarke bezogen auf das von Minimum und Maximum definierte Intervall fest. Bei ProgressBar ist diese Eigenschaft nicht vorhanden.
`int Maximum {get; set;}`	Maximaler Wert für die Value-Eigenschaft
`int Minimum {get; set;}`	Minimaler Wert für die Value-Eigenschaft
`Orientation Orientation {get; set;}`	TrackBar: enum-Wert, der bestimmt, ob sich das Steuerelement in horizontaler oder vertikaler Orientierung zeichnet.
`int SmallChange {get; set;}`	Wertänderung, die die Tasten [→] und [←] bewirken (bei ProgressBar nicht vorhanden)
`int TickFrequency {get; set;}`	TrackBar: Intervall von Skalenstrich zu Skalenstrich

Tabelle 15.16: Wichtige Eigenschaften der Steuerelemente VScrollBar, HScrollBar, ProgressBar und TrackBar (Forts.)

Eigenschaft	Bedeutung
`int TickStyle {get; set;}`	`TrackBar`: `enum`-Wert, der bestimmt, wo und wie die Skalenstriche und der Regler gezeichnet werden.
`int Value {get; set;}`	Von dem Steuerelement analog dargestellter Wert. Bei der Bildlaufleiste bestimmt die Differenz `Maximum – LargeChange` den Wert, der die Bildlaufmarke auf die Endposition setzt – Werte bis `Maximum` stellen zwar kein Problem dar, ändern aber die Anzeige nicht mehr. Liegt der Wert außerhalb des von `Minimum` und `Maximum` definierten Intervalls, generiert das Steuerelement eine Ausnahme.

Tabelle 15.17: Wichtige Ereignisse der Steuerelemente VScrollBar, HScrollBar, ProgressBar und TrackBar

Ereignis	Bedeutung
`Scroll`	Standardereignis; der Benutzer hat die Position der Bildlaufmarke bzw. des Reglers verändert (von `ProgressBar` nicht signalisiert)
`ValueChanged`	Der Wert der Eigenschaft `Value` hat sich verändert.

Die beiden in Tabelle 15.17 aufgelisteten Ereignisse unterscheiden sich dadurch, dass `Scroll` nur auftritt, wenn eine Benutzeraktion mit dem Steuerelement stattfindet – und sei es, wie bei der Bildlaufleiste nur ein einfacher Klick auf die Bildlaufmarke. `ValueChanged` tritt hingegen immer auf, wenn `Value` eine Wert-Änderung erfährt (also auch, wenn dies »per Programm« geschieht). Für den Schieberegler nehmen sich die beiden Ereignisse nicht viel, für die Bildlaufleiste ist `Scroll` ganz klar das interessantere, da es ein Ereignisobjekt des Typs `ScrollEventArgs` liefert, über das sich die Art der Benutzeraktion in Erfahrung bringen (Tabelle 15.18) und auch der schlussendliche Wert für die `Value`-Eigenschaft setzen lässt (`NewValue`).

Tabelle 15.18: Werte der `ScrollEventType`-Aufzählung

`enum`-Wert	Bedeutung
`EndScroll`	Bildlauf beendet
`First`	Betätigung von Pos1
`LargeDecrement`	Klick auf linken bzw. oberen Bereich zwischen Bildlaufmarke und Pfeilschaltfläche oder Betätigung von Bild↓. Das Steuerelement soll seinen Wert um (maximal) `LargeChange` herabsetzen.
`LargeIncrement`	Klick auf rechten bzw. unteren Bereich zwischen Bildlaufmarke und Pfeilschaltfläche oder Betätigung von Bild↑. Das Steuerelement soll seinen Wert um (maximal) `LargeChange` erhöhen.

enum-**Wert**	**Bedeutung**
`Last`	Betätigung von Ende
`SmallDecrement`	Klick auf linke bzw. obere Pfeilschaltfläche oder Betätigung von ← bzw. ↑. Das Steuerelement soll seinen Wert um `SmallChange` herabsetzen.
`SmallIncrement`	Klick auf rechte bzw. untere Pfeilschaltfläche oder Betätigung von → bzw. ↓. Das Steuerelement soll seinen Wert um `SmallChange` erhöhen.
`ThumbPosition`	Loslassen der Maustaste nach dem Ziehen (oder auch nur Anklicken) der Bildlaufmarke.
`ThumbTrack`	Ziehen der Bildlaufmarke mit der Maus

Tabelle 15.18: Werte der `ScrollEvent-Type`-Aufzählung (Forts.)

Der Umgang mit dem `TrackBar`-Steuerelement bedarf keiner großen Worte. Man initialisiert die Eigenschaften mit den gewünschten Werten und behandelt dann eines der beiden Ereignisse – fertig. Falls Sie sich für eine vollständige Implementierung der Schiebereglerfunktionalität als Drehknopf interessieren, werfen Sie doch einen Blick in den Abschnitt »Codebeispiel – Knob«, ab Seite 642. Dort finden Sie auch ein `TrackBar`-Steuerelement in Aktion.

Codebeispiel – Bildlaufleisten mit Zoomfunktion

Nachdem sich nicht nur in der Erbmasse des Formulars, sondern auch vieler Steuerelemente Bildlaufleistenfunktionalität findet, wird man separate Bildlaufleisten, wenn überhaupt, wohl eher zweckentfremdet einsetzen. Dabei aber sind ihre Möglichkeiten reichhaltiger als die des Schiebereglers, da sie ein Teilintervall, und nicht nur einen Wert, in einem größeren Intervall anzeigen. Erlaubt man dem Benutzer, die relative Größe des Teilintervalls zu ändern, ergibt sich ein Zoom – zugegeben bei Verlust des Realbezugs für das Gesamtintervall.

Das Projekt *Bildlaufleisten* stellt eine Bildlaufleistenfunktionalität vor, wie man sie sich ähnlich für viele Zeichenprogramme wünschen würde. Hier die Spezifikation:

1. Verschieben der Bildlaufmarke verschiebt Ansicht und Bildlaufmarke.
2. Mausklicks in die Bereiche zwischen Bildlaufmarke und Pfeilschaltfläche (`LargeXxx`) vergrößern/verkleinern den Zoomfaktor.
3. Pfeilschaltflächen verschieben die Ansicht, *nicht* jedoch die Bildlaufmarke (`SmallXxx`).

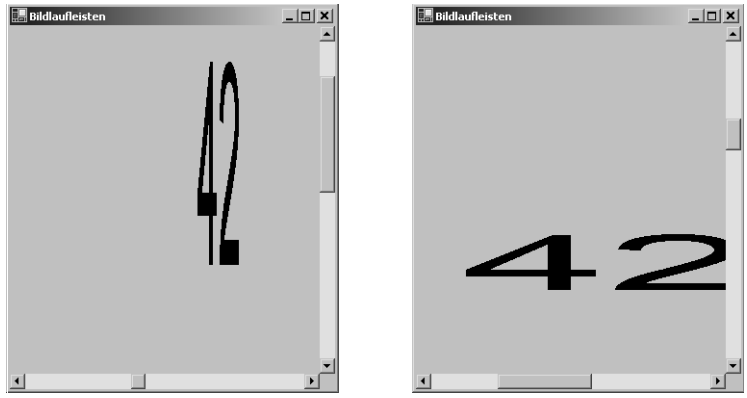


Abbildung 15.6: Die Ausgabe lässt sich in beiden Achsen zoomen und verschieben.

Der Formularentwurf enthält sichtbar die beiden Bildlaufleisten. Um sie pflegeleicht über die `Dock`-Eigenschaft an den rechten bzw. unteren Rand in den Clientbereich zu kleben, so dass ein Quadrat beim Stoß frei bleibt, ist allerdings bereits der erste Trick nötig: Die horizontale Bildlaufleiste sitzt in einem unsichtbaren `Panel`-Steuerelement, das unten an den Clientbereich andockt (`DockStyle.Bottom`). In diesem Steuerelement sitzt ein weiteres quadratisches `Panel`-Steuerelement, das seinerseits rechts andockt, wo das Quadrat frei bleiben soll (`DockStyle.Right`). Den Rest füllt die Bildlaufleiste aus (`DockStyle.Fill`).

Die Initialisierung der beiden Bildlaufleisten setzt die Eigenschaften `Minimum`, `Maximum`, `SmallChange`, `LargeChange` und `Value` so, dass beide Achsen auf 10% gezoomt sind und der sichtbare Ausschnitt in der Mitte liegt.

Das Objekt `Matrix` spielt später in der `Paint`-Behandlung die Rolle der *globalen Transformation* für die Textausgabe. Ihre Aufgabe ist es, die aktuelle Translation global (auf Ebene der Klasse) zu speichern. Die erste Translation verschiebt den Ursprung in die Mitte des Clientbereichs, sodass der sichtbare Bereich um den Ursprung herum liegt. Die Skalierung erfolgt erst unmittelbar vor dem Zeichnen, damit die Reihenfolge der beiden nicht vertauschbaren Operationen gewahrt bleibt.



Wenn Bildausschnitte im Spiel sind, arbeitet man gewöhnlich mit Verschiebungen und Skalierungen des Koordinatensystems der Ausgabefläche, statt die Koordinaten für die einzelnen Ausgabeoperationen umzurechnen. Das ist weniger (sichtbarer) Rechenaufwand und viel allgemeiner. (Genau darin liegt übrigens der Vorteil der mit jedem Grafikkontext assoziierten Transformationsmatrix.)

Das Datenfeld `MinChange` spielt eine Rolle bei der Veränderung des Zoomfaktors und definiert außerdem den unantastbaren Randbereich für die Bildlaufmarke, damit zu jedem Zeitpunkt beide Zoomoperationen möglich sind (vgl. Punkt 2 der Spezifikation).

```
private System.Windows.Forms.VScrollBar vScrollBar1;
private System.Windows.Forms.HScrollBar hScrollBar1;
private System.Drawing.Drawing2D.Matrix Matrix;

int MinChange;

private System.ComponentModel.Container components = null;

public Form1()
{
  InitializeComponent();
  // Bildlaufleisten initialisieren
  vScrollBar1.Maximum = ClientRectangle.Width - vScrollBar1.Width;
  hScrollBar1.Maximum = ClientRectangle.Height - hScrollBar1.Height;
  MinChange = hScrollBar1.Maximum/40;
  vScrollBar1.Value = (int) (vScrollBar1.Maximum * 0.45);
  vScrollBar1.LargeChange = (int) (vScrollBar1.Maximum * 0.1);
  vScrollBar1.SmallChange = vScrollBar1.LargeChange;

  hScrollBar1.Value = (int) (hScrollBar1.Maximum * 0.45);
  hScrollBar1.LargeChange = (int) (hScrollBar1.Maximum * 0.1);
  hScrollBar1.SmallChange = hScrollBar1.LargeChange;

  // Abbildung für Paint generieren und Ursprung in die Bildmitte
  Matrix = new System.Drawing.Drawing2D.Matrix();
  Matrix.Translate(vScrollBar1.Maximum/2,hScrollBar1.Maximum/2);
}
```

In der für die `Scroll`-Ereignisse *beider* Bildlaufleisten registrierten Methode `ScrollBar_Scroll()` steckt die veränderte Interpretation für die Benutzerschnittstelle der Bildlaufleisten. Das `Type`-Feld des Ereignisobjekts spezifiziert die Benutzeraktion genauer. `LargeXxx`-Nachrichten vergrößern oder verkleinern den Wert der Eigenschaft `LargeChange` (Punkt 2 der Spezifikation) des Steuerelements. Da die `Paint`-Behandlung mit diesem Wert die globale Transformation skaliert, verändert dies also den Zoomfaktor. Beachten Sie, dass die `NewValue`-Eigenschaft des Ereignisobjekts später zur `Value`-Eigenschaft wird. So führen `SmallXxx`-Nachrichten zu keiner Positionsänderung der Bildlaufmarke, bewirken aber über eine Translation der globalen Transformationsmatrix eine Verschiebung des angezeigten Bildausschnitts (Punkt 3 der Spezifikation).

```
private void ScrollBar_Scroll(object sender,
System.Windows.Forms.ScrollEventArgs e)
{
  ScrollBar sb = (ScrollBar) sender;
  int FullInterval = sb.Maximum - sb.Minimum;
  switch (e.Type)
  {
    case ScrollEventType.LargeIncrement:
      // Ansicht größer zoomen
      e.NewValue = sb.Value;
```

```
        if (sb.LargeChange < FullInterval - 4 * MinChange)
        {
          sb.LargeChange = sb.LargeChange + 2 * MinChange;
          e.NewValue = sb.Value - MinChange ;
        }
        break;
      case ScrollEventType.LargeDecrement:
        // Ansicht kleiner zoomen
        e.NewValue = sb.Value;
        if (sb.LargeChange > 2*MinChange)
        {
          sb.LargeChange = sb.LargeChange - 2*MinChange;
          e.NewValue = sb.Value + MinChange;
        }
        break;
      case ScrollEventType.SmallIncrement:
        // Gesamte Ansicht nach unten oder rechts verschieben
        e.NewValue = sb.Value;
        if (sender == this.vScrollBar1)
          Matrix.Translate(0, -sb.SmallChange);
        else
          Matrix.Translate(-sb.SmallChange, 0);
        break;
      case ScrollEventType.SmallDecrement:
        // Gesamte Ansicht nach unten oder rechts verschieben
        e.NewValue = sb.Value;
        if (sender == this.vScrollBar1)
          Matrix.Translate(0, sb.SmallChange);
        else
          Matrix.Translate(sb.SmallChange, 0);
        break;
      case ScrollEventType.SmallIncrement:
        // Gesamte Ansicht nach oben oder links verschieben
        e.NewValue = sb.Value;
        if (sender == this.vScrollBar1)
          Matrix.Translate(0, -sb.SmallChange);
        else
          Matrix.Translate(-sb.SmallChange, 0);
        break;
    }
    // Thumbnachrichten werden nicht gefiltert aber. ggf korrigiert
    // Gültigkeit von NewValue sicherstellen: Rand erzwingen
    if (e.NewValue < MinChange)
      e.NewValue = MinChange;
    if (e.NewValue > FullInterval - sb.LargeChange - MinChange)
      e.NewValue = FullInterval - sb.LargeChange - MinChange;
    Invalidate();
  }
```

Nun fehlt noch die `Paint`-Behandlung. Die Berechnung der globalen Transformation ist zwar lästig, muss aber sein, da der Zustand der Bildlaufleisten ja berücksichtigt sein will. Dafür ist die Ausgabeoperation an sich einfach: Sie zeichnet den Text grundsätzlich zentrisch zum Ursprung.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  const string text = "42";
  e.Graphics.Transform = Matrix;

  // Translation durch Bildlaufleisten ausrechnen und anwenden
  float dx = hScrollBar1.Value + hScrollBar1.LargeChange/2f
    - (hScrollBar1.Maximum - hScrollBar1.Minimum)/2f;
  float dy = vScrollBar1.Value + vScrollBar1.LargeChange/2f
    - (vScrollBar1.Maximum - vScrollBar1.Minimum)/2f;
  e.Graphics.TranslateTransform(-dx, -dy);

  // Skalierung durch Bildlaufleisten ausrechnen und anwenden
  float ScaleX = hScrollBar1.LargeChange /
    (float)(hScrollBar1.Maximum - hScrollBar1.Minimum);
  float ScaleY = vScrollBar1.LargeChange /
    (float)(vScrollBar1.Maximum - vScrollBar1.Minimum);
  e.Graphics.ScaleTransform(ScaleX, ScaleY);

  //Textausgabe
  Font font = new Font("Arial", 600);
  SizeF size = e.Graphics.MeasureString(text, font);
  e.Graphics.DrawString(text, font, Brushes.Black,
   -size.Width/2, -size.Height/2);
}
```

Übung

Wenn Sie ein wenig mit dem Programm herumspielen, werden Sie feststellen, dass es einer gewissen Übung bedarf, die Darstellung im sichtbaren Bereich zu halten bzw. wieder dahin zurück zu manövrieren, wenn sie einmal den Bildrand verlassen hat. Als Lösung dafür könnte das besagte Quadrat durch eine `Button`-Schaltfläche ersetzt und diese mit einem Reset der Anzeigeattribute belegt werden. Ein anderer Ansatz wäre es sicherzustellen, dass die Darstellung den Bildausschnitt nicht vollständig verlassen kann. Wer sich dieser Problemstellung ernsthaft widmet, wird nie mehr Schwierigkeiten mit Transformationen haben – versprochen.

15.2.5 Panel, GroupBox, Splitter

Die Steuerelemente `Panel` und `GroupBox` sind typische Container-Steuerelemente. Sie ermöglichen die Gruppierung anderer Steuerelemente zu logischen und/oder visuellen Einheiten.

Steuerelemente, die nicht direkt dem Formular untergeordnet sind, sondern einem Container-Steuerelement, werden über die `Controls`-Auflistung des Container-Steuerelements verwaltet.

Der Container übernimmt dabei auch die Rolle der Ereigniszustellung, das heißt, er gibt einen zusätzlichen Ansatzpunkt ab, um in die Ereignisbehand-

lung einzugreifen. Unterbleibt eine Ereignisbehandlung auf Ebene des Containers, bleibt er für das Ereignisgeschehen weitgehend transparent.

»Weitgehend« deshalb, weil er:

1. für seine Elemente eine untergeordnete Tabulatorreihenfolge unterhält und den Eingabefokus darin zirkulieren lässt, wenn er über die Richtungstasten oder ⇆ weitergeschaltet wird.
2. Steuerelemente auch logisch gruppiert – wie im Falle der `RadioButton`-Steuerelemente, die sich bekanntlich ja gegenseitig auslösen, so dass immer nur maximal ein Optionsfeld markiert sein kann.
3. für das Layout seiner Elemente sorgt, wenn diese über die Eigenschaften `Anchor` oder `Dock` verankert bzw. angedockt sind.

GroupBox

Das `GroupBox`-Steuerelement wird vornehmlich zur logischen und visuellen Gruppierung von Steuerelementen in Eigenschaftsseiten von Eigenschafts- und Optionendialogen eingesetzt. Im Gegensatz zum `Panel`-Steuerelement unterstreicht es die Gruppierung immer visuell, indem es einen Rahmen und eine Beschriftung (`Text`-Eigenschaft) zeichnet. Als Container-Objekt bietet es die Möglichkeit, Vorgabewerte (Farben, Schriftattribute etc.) für die untergeordneten Steuerelemente zu definieren sowie diese mit einem Hintergrundbild zu unterlegen.

Panel und Splitter

Das Steuerelement `Panel` ist für jeden Benutzer von Visual Studio .NET ein alter Bekannter, auch wenn es sich vielleicht noch nicht namentlich vorgestellt hat: Der Clientbereich des Formulars ist in der Tat nichts anderes als ein `Panel`-Steuerelement, das an das Fenster des Formulars mit `DockStyle.Fill` andockt. Damit sollte klar sein, dass sich mit einem `Panel`-Steuerelement all das machen lässt, was Sie auch im Clientbereich des Formulars veranstalten können: zeichnen, Steuerelemente hineinpacken, Hintergrundbilder anzeigen usw. – und eben auch gruppieren.

Die häufigste Anwendung des `Panel`-Steuerelements ist die Unterteilung des Clientbereichs in logisch unterschiedliche Ausgabebereiche. Setzt man dazu noch ein `Splitter`-Steuerelement ein und arbeitet mit der `Dock`-Eigenschaft, erhält man – ohne den geringsten Verwaltungsaufwand – ein flexibles Layout, in dem der Benutzer die Bereichsgrößen per Maus selbst anpassen kann.

Codebeispiel – PanelDemo

Das Codebeispiel *PanelDemo* kommt vom Prinzip her mit nur einer einzigen Zeile selbstgetippten Codes aus: einer `DrawString()`-Anweisung.

```
private void Panel_Paint(object sender, PaintEventArgs e)
{
  e.Graphics.DrawString(((Panel)sender).Name,
    new Font("Arial", 30), Brushes.Black, 20, 20);
}
```

Das Know-how steckt hier im Design. Wie in Bild 15.7 zu sehen, enthält das Formular drei `Panel`-Steuerelemente und zwei `Splitter`-Steuerelemente. Um die abgebildete Aufteilung zu erreichen, wurde als Erstes `panel1` in den Entwurf eingefügt und unten angedockt (`DockStyle.Bottom`). Als Nächstes folgte ein `Splitter`-Steuerelement, das gleichfalls unten angedockt wurde. Höhe und Stil dieses Steuerelements bestimmen das Aussehen des Randes, der sich mit der Maus ziehen lässt. Die restlichen Schritte sollten dann klar sein: Einfügen von `panel2` in den oberen Bereich und rechts andocken (`DockStyle.Right`); Einfügen eines weiteren `Splitter`-Steuerelements und rechts andocken; Einfügen von `panel3` in den restlichen Bereich links oben und rundherum andocken (`DockStyle.Fill`). Voilà.



Abbildung 15.7: Formular mit drei Panel-Steuerelementen und zwei Splittern

Beim Andocken kommt es augenscheinlich auf die Reihenfolge des Einfügens an. Um diese Reihenfolge später im Programm dynamisch zu ändern, stehen Ihnen auf Seiten der Steuerelemente die Methoden `BringToFront()` und `SendToBack()` und auf Seiten des Containers `SetChildIndex()` zur Verfügung. Bis auf das vorderste Element docken alle an einer Seite an – entweder direkt an einen Containerrand oder an den freien Rand eines dort angedockten Elements mit kleinerem Index. Das vorderste (zuletzt eingefügte) Element erhält immer das `DockStyle.Fill`-Attribut.

Eine `Click`-Behandlung, die `panel3` links, `panel2` rechts oben und `panel3` rechts unten andockt, würde so aussehen:

```
private void panel3_Click(object sender, System.EventArgs e)
{
  panel1.SendToBack();
  splitter1.SendToBack();
  panel2.SendToBack();
  splitter2.SendToBack();
  panel3.SendToBack();
  panel3.Dock = DockStyle.Left;
  splitter2.Dock = DockStyle.Left;
  panel2.Dock = DockStyle.Top;
  splitter1.Dock = DockStyle.Top;
  panel1.Dock = DockStyle.Fill;
}
```

Ein `Splitter`-Steuerelement signalisiert zwei Ereignisse: `SplitterMoving` und `SplitterMoved`. Wie die Namensgebung bereits verrät, tritt ersteres während des Ziehvorgangs auf, letzteres verkündet seinen Abschluss. Beide übermitteln ein Ereignisobjekt des Typs `SplitterEventArgs`, das zwei Koordinatenpaare zur Verfügung stellt: (`X`,`Y`) beschreiben die aktuelle Position, während (`SplitX`, `SplitY`) die schlussendlich gesetzte Position wiedergibt, die sich – ausschließlich im Zuge der Behandlung von `SplitterMoving` – aber noch anpassen lässt. Die folgende `SplitterMoving`-Behandlung implementiert ein »magnetisches« 30 Pixel-Gitter für ein `Splitter`-Objekt:

```
private void splitter2_SplitterMoving(object sender,
System.Windows.Forms.SplitterEventArgs e)
{
  const int Grid = 30;
  e.SplitX = e.X/Grid * Grid;
  e.SplitY = e.Y/Grid * Grid;
}
```

15.2.6 TreeView

Das `TreeView`-Steuerelement zählt zu den komplexeren Steuerelementen und hat eine Vielfalt von Einsatzmöglichkeiten. Wo immer eine allgemeine Baumstruktur dargestellt werden muss, bei der ein Knoten mehrere untergeordnete Knoten haben kann, sind Sie mit ihm gut bedient. Die Benutzerschnittstelle des `TreeView`-Steuerelements wird Ihnen von der erweiterten Ansicht des Windows-Explorers, vom Registrierungseditor aber auch vom Setup-Assistenten her sicher bekannt sein.

Dementsprechend gewaltig sind die in dem `TreeView`-Steuerelement steckenden Möglichkeiten. Ein Blick auf die Eigenschaften, Methoden und Ereignisse unterstreicht dies (vgl. Tabelle 15.19 bis Tabelle 15.23).

Aufbau

Der Aufbau der Wertrepräsentation durch das `TreeView`-Steuerelement ist streng rekursiv: Eine `TreeView`-Instanz verfügt über ein privates (und somit unsichtbares) `TreeNode`-Objekt (Wurzel), dessen `Nodes`-Auflistung es veröffentlicht. In diese Auflistung lassen sich beliebig viele Knoten in Form von `TreeNode`-Objekten einfügen, die ihrerseits wieder über `Nodes`-Auflistungen verfügen. Die von diesen Objekten verkörperten Knoten werden als *Stammknoten* bezeichnet und können ihrerseits wieder als Wurzel eines Teilbaums fungieren.

Darstellung

Das Steuerelement stellt die Knoten – angefangen von den Stammknoten – hierarchisch dar, indem es untergeordnete Knoten einrückt und die Abstammungsbeziehung standardmäßig durch Linien (`ShowLines`) visualisiert. An Eigenschaften visualisiert das Steuerelement zunächst einmal die `Text`-Eigenschaft. Ist dem Steuerelement eine Bildliste (`ImageList`) mit Bitmaps für die Knoten zugeordnet, zeichnet es davor noch ein Bild der Größe 16x16 Pixel. Dabei ist es weitgehend gleichgültig, welche Abmessungen die Bilder in der Bildliste tatsächlich haben, denn das `TreeView`-Steuerelement verwendet in jedem Fall seine Methode `GetThumbnailImage()` zur Skalierung. Die Eigenschaften `ImageIndex` und `SelectedImage` geben zwei Bilder an, die standardmäßig vor nicht ausgewählte bzw. ausgewählte Knoten gezeichnet werden (beispielsweise Ordner- oder Laufwerkssymbole). Für jeden Knoten lassen sich bei Bedarf auch individuell andere Bildindizes für diese beiden Zustände festlegen.

Knoten haben eine `Text`-Eigenschaft, die der Benutzer bei eingeschaltetem Bearbeitungsmodus (`BeginEdit()`, `IsEditing`, `EndEdit()`) interaktiv ändern kann. Darüber hinaus können Sie verschiedene Zustände einnehmen:

- Er kann »entfaltet« (`Expand()`, `IsExpanded`) oder »reduziert« (`Collapse()`) sein. Ist die `ShowPlusMinus`-Eigenschaft `true` (Vorgabewert), zeichnet das Steuerelement vor einen entfalteten Knoten ein Minus-Symbol und vor einen reduzierten Knoten ein Plus-Symbol. Der Benutzer kann diesen Zustand (und die Darstellung) umschalten, indem er auf diese Symbole klickt. Das Steuerelement signalisiert dann entsprechend die Ereignisse `BeforeCollapse` und `AfterCollapse` sowie `BeforeExpand` und `AfterExpand`.
- Er kann »ausgewählt« sein (`SelectedNode`, `IsSelected`). Einen Knoten, der den Fokus hat, zeichnet das Steuerelement im ausgewählten Zustand. Dazu unterlegt er die `Text`-Eigenschaft und zeichnet das Bild mit dem Index `SelectedImage`. Das Steuerelement signalisiert dann entsprechend die Ereignisse `BeforeSelect` und `AfterSelect`.
- Er kann »markiert« sein (`Checked`). Ist die `CheckBoxes`-Eigenschaft des Steuerelements `true` (Vorgabe ist `false`), zeichnet das Steuerelement

zusätzlich noch ein Kontrollkästchen vor das Bild, mit dem der Benutzer die `Checked`-Eigenschaft manipulieren kann. Das Steuerelement signalisiert dann entsprechend die Ereignisse `BeforeCheck` und `AfterCheck`.

- Er kann »im Bearbeitungsmodus« sein (`IsEditing`). Das Steuerelement signalisiert dann entsprechend die Ereignisse `BeforeLabelEdit` und `AfterLabelEdit`.

Was im Vergleich zu den Steuerelementen `ListBox` und `ListView` fehlt: Ein »Ownerdraw«, also die Möglichkeit für den Besitzer einer Steuerelementinstanz, die Knoten in Eigenregie zu zeichnen.

Tabelle 15.19: Charakteristische Eigenschaften des `TreeView`-Steuerelements

Eigenschaft	Bedeutung
`bool CheckBoxes {get; }`	Anzeige von Kontrollkästchen zwischen Plus-/Minus-Symbol und Knotenbild
`string FullPath {get; }`	Vollständiger Pfad vom Wurzelknoten bis zum aktuellen Knoten. Der Pfad wird aus den `Name`-Eigenschaften der Knoten abgetrennt durch den `PathSeparator`-String gebildet.
`int ImageIndex {get; set;}`	Index der standardmäßig für die nicht ausgewählte Darstellung verwendeten Bitmap in `ImageList`.
`ImageList ImageList {get; set;}`	Bildliste für die neben den Knoten angezeigten Bilder
`bool LabelEdit {get; set;}`	Bearbeitungsmodus für die Änderung der Knotenbeschriftung
`TreeNodeCollection Nodes {get; set;}`	Knotenliste der Wurzel
`string PathSeparator {get; set;}`	Trennzeichen für die Bildung der `FullPath`-Eigenschaft. Standardwert ist »\\«.
`int SelectedImageIndex {get; set;}`	Index der standardmäßig für die ausgewählte Darstellung verwendeten Bitmap in `ImageList`.
`TreeNode SelectedNode {get; set;}`	Ausgewählter Knoten. Der Index für das Standardbild des ausgewählten Knotens in der `ImageList` ist `SelectedImageIndex` (wobei dieses Bild aber nur angezeigt wird, wenn die `SelectedImageIndex`-Eigenschaft des Knotens nicht anders gesetzt wurde).
`bool ShowLines {get; set;}`	Darstellung von Verbindungslinien zwischen gleich- und untergeordneten Knoten. Standardwert ist `true`.

Eigenschaft	Bedeutung
`bool ShowPlusMinus {get; set;}`	Darstellung von Plus- und Minussymbol für entfaltete und nicht entfaltete Knoten. Standardwert ist `true`.
`bool ShowRootLines {get; set;}`	Darstellung von Verbindungslinien zwischen den Wurzelknoten. Standardwert ist `true`.

Tabelle 15.19: Charakteristische Eigenschaften des `TreeView`-Steuerelements (Forts.)

Methode	Bedeutung
`void CollapseAll()`	Anzeige nur der Wurzelknoten
`void Expand()`	Entfaltet den Knoten.
`void ExpandAll()`	Entfaltet den gesamten untergeordneten Teilbaum des Knotens.
`TreeNode GetNodeAt()`	Liefert den Knoten zu dem angegebenen Bildpunkt.
`int GetNodeCount()`	Liefert die Anzahl der neben- oder untergeordneten Knoten.

Tabelle 15.20: Methoden einer `TreeView`-Instanz

Ereignisse	Bedeutung
`AfterCheck`	Tritt auf, nachdem sich der Zustand eines Kontrollkästchens (vgl. `CheckBoxes`-Eigenschaft) geändert hat.
`AfterCollapse`	Tritt auf, nachdem ein Knoten reduziert wurde.
`AfterExpand`	Tritt auf, nachdem ein Knoten entfaltet wurde.
`AfterLabelEdit`	Tritt auf, nachdem die Bezeichnung eines Knotens geändert wurde.
`AfterSelect`	Tritt auf, nachdem ein Knoten ausgewählt wurde.
`BeforeCheck`	Tritt unmittelbar vor der Zustandsänderung eines Kontrollkästchens (vgl. `CheckBoxes`-Eigenschaft) auf.
`BeforeCollapse`	Tritt unmittelbar vor der Reduzierung eines Knotens auf.
`BeforeExpand`	Tritt unmittelbar vor der Entfaltung eines Knotens auf.
`BeforeLabelEdit`	Tritt unmittelbar vor der Änderung einer Knotenbezeichnung auf.
`BeforeSelect`	Tritt unmittelbar vor der Auswahl eines Knotens auf.

Tabelle 15.21: Ereignisse, die das `TreeView`-Steuerelement signalisiert

Tabelle 15.21: Ereignisse, die das `TreeView`-Steuerelement signalisiert (Forts.)

Ereignisse	Bedeutung
`ItemDrag`	Tritt auf, wenn ein Knoten per Drag&Drop innerhalb des Steuerelements an eine andere Position gezogen wird (leitet die Drag&Drop-Operation ein).
`TextChanged`	Tritt auf, wenn sich der Wert der `Text`-Eigenschaft geändert hat – beispielsweise durch Benutzerinteraktion.

Tabelle 15.22: Eigenschaften der Klasse `TreeNode`

Eigenschaft	Bedeutung
`bool Checked {get; set;}`	Zustand des dem Knoten zugeordneten Kontrollkästchens; spielt (visuell) nur eine Rolle, wenn die `CheckBoxes`-Eigenschaft des `TreeView`-Steuerelements `true` gesetzt ist. Lässt sich auch zweckentfremden.
`TreeNode FirstNode {get;}`	Erster Knoten in der Auflistung `Nodes` des Knotens.
`string FullPath {get;}`	Gesamter Pfad des Knotens vom Stammknoten (Wurzel) an (= Aneinanderreihung der `Text`-Eigenschaften aller beteiligten Knoten mit Trennzeichen; die Trennzeichenfolge liefert die `PathSeparator`-Eigenschaft des `TreeView`-Objekts).
`int ImageIndex {get; set;}`	Index der Bitmap in der Bildliste (`ImageList` des `TreeView`-Objekts), die für die Darstellung im nicht ausgewählten Zustand verwendet wird.
`int Index {get;}`	Position des Knotens in der `Nodes`-Auflistung des übergeordneten Knotens.
`bool IsEditing {get;}`	`true`, wenn der Knoten im Bearbeitungsmodus ist.
`bool IsExpand {get;}`	`true`, wenn der Knoten entfaltet ist.
`bool IsSelected {get;}`	`true`, wenn der Knoten markiert ist.
`bool IsVisible {get;}`	`true`, wenn der Knoten im Bearbeitungsmodus ist.
`TreeNode LastNode {get;}`	Letzter Knoten in der `Nodes`-Auflistung des Knotens
`TreeNode NextNode {get;}`	Nächster dem Knoten nebengeordneter Knoten
`TreeNode NextVisibleNode {get;}`	Nächster vom Steuerelement angezeigter Knoten. (Der gelieferte Knoten muss in keiner besonderen Beziehung zu dem Knoten stehen.)
`Font NodeFont {get; set;}`	Schriftart, in der die `Text`-Eigenschaft des Knotens angezeigt wird.

Eigenschaft	Bedeutung
`TreeNodeCollection Nodes {get; set;}`	Auflistung der dem Knoten untergeordneten Knoten
`TreeNode Parent {get;}`	Dem Knoten übergeordneter Knoten
`TreeNode PrevNode {get;}`	Voriger dem Knoten nebengeordneter Knoten
`TreeNode PrevVisibleNode {get;}`	Voriger vom Steuerelement angezeigter Knoten. (Der gelieferte Knoten muss in keiner besonderen Beziehung zu dem Knoten stehen.)
`int SelectedImageIndex {get; set;}`	Index der Bitmap in der Bildliste (`ImageList` des `TreeView`-Objekts), die für die Darstellung im ausgewählten Zustand verwendet wird.
`object Tag {get; set;}`	Allgemeine benutzerdefinierbare Eigenschaft, die es ermöglicht, dem Steuerelement ein beliebiges Objekt anzuheften.
`string Text {get; set;}`	Vom Steuerelement angezeigte Eigenschaft des Knotens, kann nach Aufruf von `BeginEdit()` interaktiv durch den Benutzer bearbeitet werden.
`TreeView TreeView {get; }`	Das dem Knoten übergeordnete `TreeView`-Objekt

Tabelle 15.22: Eigenschaften der Klasse `TreeNode` (Forts.)

Methoden	Bedeutung
`void BeginEdit()`	Schaltet den Knoten in den Bearbeitungsmodus um, damit der Benutzer die `Text`-Eigenschaft ändern kann.
`TreeNode Clone()`	Liefert eine Kopie des Knotens mit anhängendem Teilbaum.
`void Collapse()`	Reduziert den Knoten.
`void EndEdit()`	Beendet den Bearbeitungsmodus für den Knoten.
`void EnsureVisible()`	Sorgt dafür, dass der Knoten sichtbar ist; entfaltet dafür alle übergeordneten Knoten.
`void Expand()`	Erweitert den Knoten, so dass die ihm direkt untergeordneten Knoten sichtbar werden.
`void ExpandAll()`	Erweitert den Knoten, so dass alle ihm untergeordneten Knoten sichtbar werden (gesamter Teilbaum).

Tabelle 15.23: Methoden der Klasse `TreeNode`

Tabelle 15.23: Methoden der Klasse `TreeNode` (Forts.)

Methoden	Bedeutung
`int GetNodeCount()`	Liefert beim Aufruf mit `false` die Anzahl der dem Knoten direkt untergeordneten Knoten, beim Aufruf mit `true` die Anzahl aller Knoten im Teilbaum des Knotens.
`void Remove()`	Entfernt den Knoten aus der `Nodes`-Auflistung des übergeordneten Knotens.
`void Toggle()`	Wechselt bei jedem Aufruf zwischen erweiterter und reduzierter Darstellung.

Codebeispiel – Pfadauswahl per TreeView

Der Umgang mit dem `TreeView`-Steuerelement lässt sich natürlich am besten anhand eines praktischen Beispiels demonstrieren – seien Sie aber gewarnt, der Teufel steckt hier wirklich im Detail. Das vorgestellte Codebeispiel *TreeViewDemo* ist die Implementierung eines Pfadauswahl-Dialogs, wofür sich (leider) kein Steuerelement in der TOOLBOX findet. Um es übersichtlich zu halten, beschränkt es sich auf das lokale Dateisystem des Computers und gibt sich auch keine Mühe, unterschiedliche Laufwerksarten symbolisch zu unterscheiden. Wie Bild 15.8 zeigt, lässt sich damit aber recht gut arbeiten.

Das Formular enthält neben einer öffentlichen `Path`-Eigenschaft auch zwei Schaltflächen OK und ABBRECHEN, deren `DialogResult`-Eigenschaft gepflegt wird, was bereits darauf hindeutet, dass die Formularklasse als modaler Dialog gedacht ist. Das vorliegende Projekt *TreeViewDemo* enthält daher noch ein weiteres kleines Formular, über das sich der modale Dialog testen lässt. Ist die `Path`-Eigenschaft beim Aufruf auf den leeren String `""` gesetzt, bringt der Dialog via `GetCurrentDirectory()` den aktuellen Pfad in Erfahrung und zeigt ihn an. Ansonsten versucht er, soweit es geht, den angegebenen Pfad darzustellen. Einen Pfad ohne Laufwerksangabe bezieht er als relativen Pfad auf den aktuellen Pfad. Der Dialog ist also recht komfortabel in der Anwendung.

Entwurf

Beim Entwurf des Dialogs wurde zusätzlich zu den sichtbaren Steuerelementen noch ein `ImageList`-Steuerelement platziert und mit drei Bitmaps initialisiert. Es liefert die Symbole für das `TreeView`-Steuerelement – ein Festplattensymbol, ein geschlossener und ein geöffneter Ordner. Die Bilder selbst sind aus einem Screenshot des Explorers extrahiert und auf die kleine Symbolgröße (16x16 Pixel) zugeschnitten. Im Verzeichnis *Visual Studio .NET\Common7\Graphics* finden sich zwar allerlei Bitmaps in dieser Richtung, doch das Festplattensymbol ist nicht dabei.

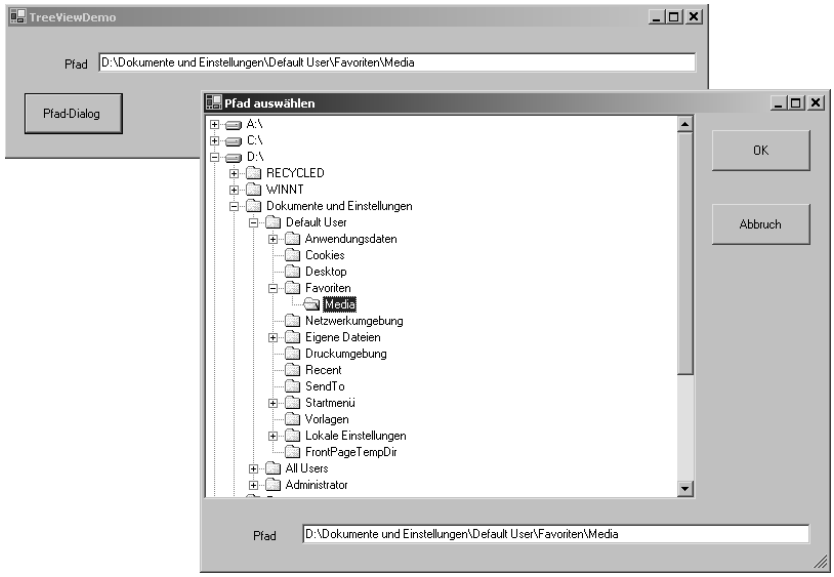


Abbildung 15.8: Das gefüllte TreeView-Steuerelement in Aktion

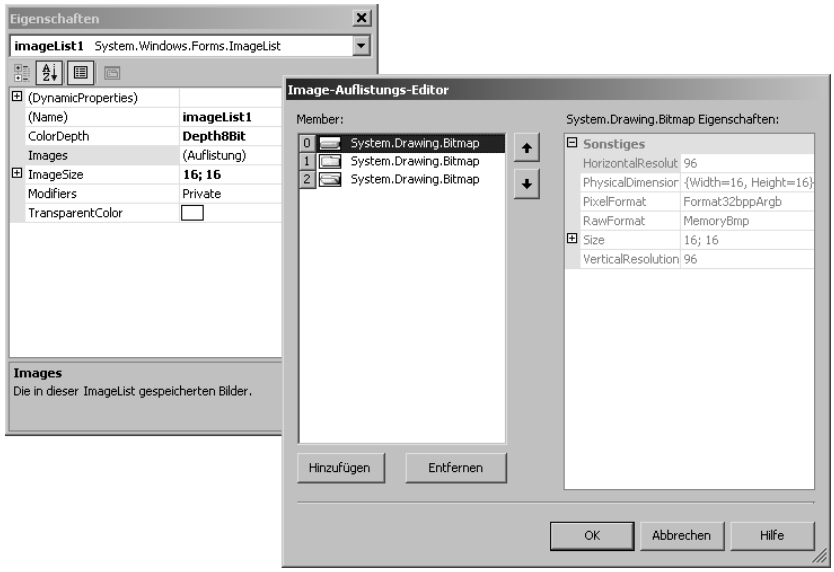


Abbildung 15.9: Bestücken des `ImageList`-Steuerelements

Implementierung

Um es gleich vorweg zu nehmen: Die Implementierung des Dialogs ist nicht trivial. Sie enthält einige Ideen und umschifft eine Reihe von Klippen, die auf den ersten Blick nicht gleich auszumachen sind. Obwohl die Logik recht eng verzahnt ist, lassen sich zwei Vorgänge unterscheiden:

- Füllen der Baumstruktur
- Suchen in der Baumstruktur

Beim Füllen kommt gleich der ersten Pferdefuß zu Tage: Es macht keinen Sinn, die gesamte Struktur des lokalen Dateisystems en bloc in das Steuerelement einsetzen zu wollen, weil das je nach Umfang des lokalen Dateisystems ohne weiteres mehrere Minuten dauern könnte.

Hier muss also eine dynamische Lösung her, die so aussieht, dass immer nur die `Nodes`-Auflistung des jeweils expandierten Knotens neu besetzt wird. Zudem findet für jeden dabei hinzugefügten Knoten eine kleine Vorschau statt, die darüber Aufschluss gibt, ob der Knoten expandiert werden kann oder nicht. Falls er expandiert werden kann, erhält er einen einzigen untergeordneten Dummy-Knoten, damit ihn das Steuerelement mit einem Plus-Symbol schmückt.

Füllen der Baumstruktur

Für die erste Ebene der Stammknoten, in der die Laufwerke erscheinen, unterbleibt die Vorschau, damit schlafende Laufwerke – CD-ROM- und Diskettenlaufwerk, aber auch heruntergefahrene Festplatten – nicht geweckt werden. In dieser Ebene lässt sich auch das Phänomen beobachten, dass beispielsweise bei einem Diskettenlaufwerk ohne eingelegte Diskette zunächst ein Pluszeichen anzeigt wird, das dann verschwindet, wenn man versucht, den Knoten zu entfalten, ohne dass eine Diskette eingelegt ist. Der Konstruktor fragt die im System installierten logischen Laufwerke ab und fügt für jedes einen Knoten in das Objekt `treeView1` ein. Jeder dieser Knoten erhält einen untergeordneten Dummy-Knoten. Der Bildindex (0 für beide Zustände) wird explizit angegeben, damit die Knoten das Laufwerkssymbol anzeigen.

```
public class SelectPath : System.Windows.Forms.Form
{
  private System.Windows.Forms.TreeView treeView1;
  private System.Windows.Forms.ImageList imageList1;
  private System.Windows.Forms.TextBox textBox1;
  private System.Windows.Forms.Button bOK;
  private System.Windows.Forms.Button bAbbruch;
  private System.Windows.Forms.Label label1;
  private System.ComponentModel.IContainer components;
  private TreeNode OpenNode;            // aktiver Knoten


  public SelectPath()
  {
    InitializeComponent();
    treeView1.ImageList = imageList1;
    treeView1.ImageIndex = 1;
    treeView1.SelectedImageIndex = 2;
```

```
    // Erst die Laufwerke einfügen
    string[] Drives = Directory.GetLogicalDrives();
    TreeNode tn;
    foreach(string Drive in Drives)
    {
      tn = new TreeNode(Drive, 0, 0, new TreeNode[]{});
      treeView1.Nodes.Add(tn);
      tn.Nodes.Add(new TreeNode("dummy"));
    }
  }
```

Mehr passiert erst einmal nicht. In diesem Zustand zeigt das Steuerelement die Laufwerke als Stammknoten an – und jedes hat ein Plus-Symbol davor. Klickt der Benutzer auf eines der Pluszeichen, signalisiert das Steuerelement die `XxxExpand`-Ereignisse. Der Code behandelt nur `BeforeExpand`.

```
private void treeView1_BeforeExpand(object sender,
   System.Windows.Forms.TreeViewCancelEventArgs e)
{
  treeView1.BeginUpdate();
  AddDirs(e.Node);
  OpenNode = e.Node;
  treeView1.EndUpdate();
}
```

Die Routine `AddDirs()` bevölkert den expandierten Knoten. Das Einrahmen dieses Aufrufs mit `BeginUpdate` und `EndUpdate` sorgt dafür, dass sich das Steuerelement während des Umbaus nicht zeichnet.

`AddDirs()` organisiert sich den Verzeichnispfad für den aktuellen Knoten über die `FullPath`-Eigenschaft und ruft die Liste der darin enthaltenen Verzeichnisse ab (`GetDirectories()`). Diese durchläuft sie, fügt für jedes Verzeichnis einen Knoten ein und prüft dann per Vorschau, ob noch untergeordnete Verzeichnisse existieren. Falls ja, erhält der Knoten einen Dummy-Knoten, damit ihn das Steuerelement mit Pluszeichen anzeigt. Falls keine untergeordneten Verzeichnisse aufzufinden sind – was wie schon gesagt beim Öffnen von Laufwerksknoten passieren kann, aber auch bei Änderungen des Dateisystems durch eine andere Anwendung –, muss eine Ausnahmebehandlung vorgenommen werden, die hier allerdings mit einem leeren Rumpf auskommt. Beachten Sie, dass die Methode wenigstens die Liste der untergeordneten Knoten löscht, um gegebenenfalls zwischenzeitlich geschehenen Veränderungen des Dateisystems Rechnung zu tragen. Es wäre natürlich auch möglich, beispielsweise durch Zweckentfremdung der `Checked`-Eigenschaft, einen einmal expandierten Knoten zu kennzeichnen und für gekennzeichnete Knoten die Kontrolle sofort zurückzugeben.

```
private void AddDirs(TreeNode tn)
{
  string path = tn.FullPath;   // Ausgangspfad
  DirectoryInfo dirInfo = new DirectoryInfo(path);
  DirectoryInfo [] Dirs;
```

```
  tn.Nodes.Clear();            // dummy löschen
  try
  {
    Dirs = dirInfo.GetDirectories();
    foreach(DirectoryInfo d in Dirs)
    {
      TreeNode dirNode = new TreeNode(d.Name);
      tn.Nodes.Add(dirNode);

      // Wenn der Knoten noch untergeordnete Knoten hat,
      // erhält er einen Dummy-Eintrag
      if ((new DirectoryInfo(dirNode.FullPath)).GetDirectories().Length >
           0)
        dirNode.Nodes.Add(new TreeNode("dummy"));
    }
  }
  catch{};
}
```

Mit diesem Code ist bereits der »Tree-Walk« möglich, das heißt, der Benutzer kann sich interaktiv im Dateisystem auf Ebene der Laufwerke und Ordner bewegen.

Suchen in der Baumstruktur

Die andere Hälfte der Miete ist die Repräsentation eines gegebenen Pfads, soweit er existiert. Werfen Sie dazu einen Blick auf die Eigenschaft `Path`, deren `set`-Accessor ja genau diese Aufgabe hat. Einen leeren String interpretiert die Methode als aktuelles Verzeichnis – angesichts der tief verschachtelten Struktur, in der die Exe-Datei der Anwendung bei der Projektentwicklung sitzt, ein guter Prüfstein. Leider funktioniert die Suche (aus bisher unerfindlichen Gründen) nur, wenn das Steuerelement sichtbar ist. Sie wird daher als Ausweg schlicht bis zur Anzeige verzögert.

```
public string Path
{
  get{ return textBox1.Text; }
  set
  {
    if (value == "")
      textBox1.Text  = System.IO.Directory.GetCurrentDirectory();
    else
      textBox1.Text  = System.IO.Path.GetFullPath(value);
    // Die Suche muss verzögert werden, bis der Dialog sichtbar
    // wird. Aktivierung in VisibleChanged-Behandlung.
    if (Visible)
      SearchNode(treeView1,textBox1.Text);
  }
}
private void treeView1_VisibleChanged(object sender, System.EventArgs e)
{
    this.SearchNode(treeView1, path);
}
```

Die Suchmethode `SearchNode()` simuliert einen Benutzer, der sich schrittweise zum gesuchten Pfad durchklickt. Das ist nötig, weil die Knotenlisten der jeweiligen Knoten erst einmal besetzt werden müssen – davon abgesehen gibt es auch keine entsprechende Suchfunktion.

Beginnend mit der Liste der Stammknoten durchsucht die Methode Knotenliste für Knotenliste (`Nodes`). Wann immer ein Knoten gefunden ist, der passt, wird dieser via `Expand()` entfaltet und die Suche mit dessen Knotenliste fortgesetzt – bis auch der letzte Knoten gefunden ist und der Pfad stimmt oder die Suche an einem Knoten abbricht, weil mit dessen Liste kein Fortkommen mehr ist. In der Praxis ergibt sich hierbei aber ein kleines Problem, da Laufwerksbezeichner mit einem umgekehrten Schrägstrich abgeschlossen werden, weshalb die `FullPath`-Eigenschaft Pfade nach dem Muster »`D:\\ordner\ordner...`« liefert. Als Workaround wird der gesuchte Pfad einfach an dieses Muster angepasst. Der Workaround in umgekehrter Richtung findet sich übrigens in der Behandlungsmethode `treeView1_AfterSelect()`. Sie übernimmt den jeweils aktuellen Pfad in das Textfeld.

```
private bool SearchNode(TreeView tv, string path)
{
  // falls path relativer Pfad ist oder missgebildet ...
  try { path = System.IO.Path.GetFullPath(path); }
  catch { return false; }
  // Sonderbehandlung, da Laufwerke mit anhängendem "\" eingefügt werden
  // und die FullPath-Eigenschaft dann das Muster
  // "D:\\ordner\ordner..." liefert
  path = path.Substring(0,3) + tv.PathSeparator + path.Substring(3);
  tv.BeginUpdate();        // nicht mehr zeichnen
  path = path.ToUpper();   // Schreibweise vereinheitlichen
  if(path.EndsWith(tv.PathSeparator)) // hängt hinten "\"?
    path = path.Substring(0,path.Length-1);       // weg damit
  TreeNodeCollection Nodes = treeView1.Nodes;
  foreach(TreeNode t in Nodes)      // alle offenen Rootknoten reduzieren
    t.Collapse();
  bool NodeFound;
  string PartialPathUpper = "";
  do                        // Pfad knotenweise entfalten
  {
    NodeFound = false;
    foreach(TreeNode t in Nodes)  // Aktuelle Ebene durchsuchen
    {
      PartialPathUpper = t.FullPath.ToUpper();
      if(path.StartsWith(PartialPathUpper))     // Passt der Knoten?
      {                     // gefunden, nächste Ebene vorbereiten
        t.Expand();         // Knoten entfalten
        OpenNode = t;       // merken
        Nodes = t.Nodes;    // Knotensammlung übernehmen
        NodeFound = true;   // Funktionswert vorbereiten
        break;              // weiter mit der nächsten Ebene
      }
    }
  }
  while (NodeFound && PartialPathUpper != path);
```

```
  OpenNode.Collapse();
  treeView1.SelectedNode = OpenNode;  // markieren
  tv.EndUpdate();          // Zeichnen wieder zulassen
  return NodeFound;
}

private void treeView1_AfterSelect(object sender,
System.Windows.Forms.TreeViewEventArgs e)
{
  string s = e.Node.FullPath;
  if (s.Length > 3)
    s = s.Substring(0,2) + s.Substring(3);
  textBox1.Text = s;
}
```

Auf die Wiedergabe des restlichen Codes wurde verzichtet, um das Beispiel nicht allzu sehr in die Länge zu ziehen.

Übung

Wenn Sie sich den Beispielcode *in medias res* erarbeiten wollen, lösen Sie doch eine der folgenden Aufgaben:

1. In einer aufgeklappten Ansicht kann es passieren, dass ein übergeordneter Knoten plötzlich nicht mehr existiert – beispielsweise wenn eine andere Anwendung ein Verzeichnis des Pfades löscht oder umbenennt. Behandeln Sie diesen Fall durch ein Fallback auf den nächsten noch existierenden übergeordneten Knoten. (Hinweis: Behandeln Sie die entsprechende Ausnahme und verwenden Sie die Suchfunktion.)
   - Überlegen Sie sich eine Strategie dafür, was passiert, wenn ein logisches Laufwerk verschwindet – beispielsweise, weil der Benutzer eine Zuordnung für ein freigegebenes Verzeichnis gelöscht hat.
2. Für die interaktive Pfadauswahl ist es nicht sinnvoll, wenn mehrere nebengeordnete Teilbäume gleichzeitig entfaltet sind. Verändern Sie die `XxxExpand`-Behandlung dahingehend, dass bei einem manuellen TreeWalk immer nur der aktuelle Pfad (alle übergeordneten Knoten mit den ihnen direkt untergeordneten Knoten) entfaltet ist.

15.2.7 Menüs

Was sich eine richtige Anwendung nennen darf, muss dem Benutzer auch eine Menüleiste mit verschiedenen Menüs zur Verfügung stellen, in denen ihr Befehlsumfang repräsentiert ist. Der Spielraum für die Ausgestaltung der Menüfunktionalität ist groß und umfasst:

- das starre Menü, das zur Entwurfszeit einmal zusammengestellt und »verdrahtet« wird – sozusagen als bessere Schaltflächensammlung, die ständig die wesentlichen Befehle der Anwendung anbietet.

- das gepflegte Menü, dessen `MenuItem`-Einträge aufgrund einer ständigen Pflege der Eigenschaften `Visible`, `Checked`, `Enabled` und `Text` jeweils den aktuellen Zustand der Anwendung widerspiegeln. Der Benutzer kann immer nur die Befehle aufrufen, die die Anwendung im aktuellen Zustand auch durchführen kann, andere Befehle erscheinen deaktiviert oder sind unsichtbar.
- Kontextmenüs mit Befehlsmengen, die auf die im Formular dargestellten Objekte individuell und kontextbezogen zugeschnitten sind.
- vom Besitzer gezeichnete Menüs, die spezifische Befehls- oder Auswahlangebote durch ihr besonderes Design visuell geprägt präsentieren.

Hauptmenü im Designer anlegen und gestalten

Um ein Formular mit einem Menü auszustatten, platzieren sie eine Instanz des `MainMenu`-Steuerelements auf dem Formularentwurf und legen die Menüstruktur sowie die Menüeinträge interaktiv im Designer fest. Achten Sie bei der Benennung der Menübefehle darauf, gleich Zugriffstasten durch Vorsetzen des Zeichens »&« vor den jeweiligen Buchstaben festzulegen, das spart später viel Arbeit. (Die Zugriffstasten für die Befehle innerhalb eines Menüs sollten Sie nach Möglichkeit eindeutig wählen.) Wenn Sie einem Menübefehl weiterhin eine Tastenkombination – beispielsweise Strg+O für ÖFFNEN – zuordnen wollen, setzen Sie die `ShortCut`-Eigenschaft entsprechend und achten darauf, dass auch die `ShowShortCut`-Eigenschaft auf `true` gesetzt ist. Der Benutzer sieht die Tastenbelegung dann zur Laufzeit und kann sie so gleich erlernen.

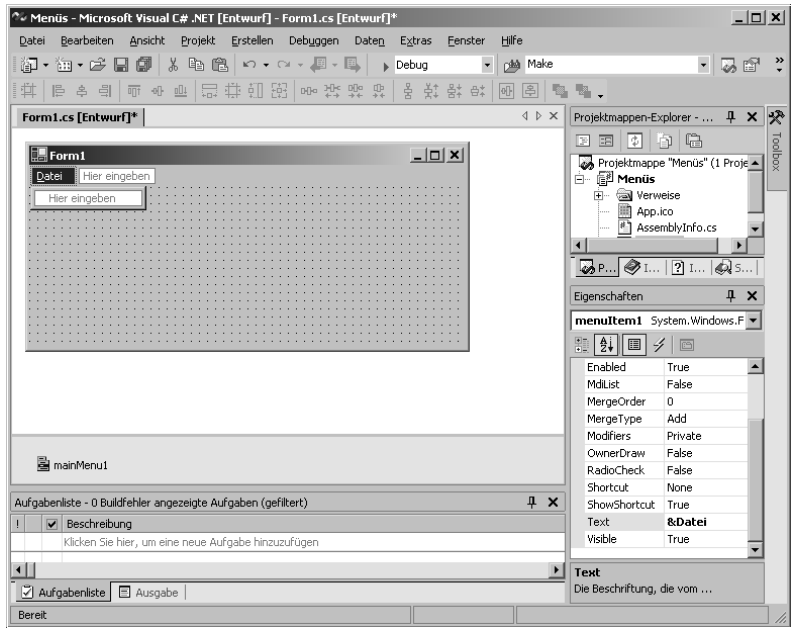


Abbildung 15.10: Der Designer konstruiert Menüeinträge vor Ort

Wenn Sie wollen, können Sie für die `MenuItem`-Objekte ein Benennungsschema verwenden, bei dem der Bezeichner die Zugehörigkeit zum übergeordneten Menü ausdrückt. Dies bewährt sich, wenn Sie an verschiedenen Stellen im Programmcode mit den Eigenschaften von `MenuItem`-Objekten arbeiten müssen (beispielsweise für die Pflege der `Checked`-Eigenschaft).

Codebeispiel – Menüs im Code pflegen

Im Gegensatz zu den anderen beiden Leisten-Steuerelementen (und komplexen Steuerelementen wie `TreeView` oder `ListView`) können Sie in einer Menüstruktur jedem einzelnen `MenuItem`-Objekt beliebige Behandlungsmethoden zuordnen. Behandelt wird üblicherweise die `Click`-Eigenschaft. Das im Folgenden diskutierte Projekt *MenüDemo* zeigt unter anderem die Pflege eines Menüs zur Laufzeit.

Die Behandlung eines Menübefehls – etwa zum Ein- und Ausschalten eines Rahmens –, der nichts weiter als seinen eigenen Zustand pflegt, sieht beispielsweise so aus:

```
private void menuToggleFrame_Click(object sender, System.EventArgs e)
{
  menuToggleFrame.Checked = ! menuToggleFrame.Checked;
  Invalidate();
}
```

Falls Sie nun denken, das sei langweilig, wird Sie der folgende Codeauszug eines Besseren belehren:

```
private Bitmap pic;
private System.Drawing.Drawing2D.Matrix transform;

... // Initialisierung von pic und transform

private void Form1_Paint(object sender, PaintEventArgs e)
{
  int RahmenDicke = 0;
  if (pic == null)
    return;

  e.Graphics.Transform = transform; // für den Zoom
  pic.SetResolution(e.Graphics.DpiX, e.Graphics.DpiY);
  if (menuFrame.Checked)
  {
    RahmenDicke = 30;
    Pen p = new Pen(new SolidBrush(pic.GetPixel(0,0)), RahmenDicke);
    e.Graphics.DrawRectangle(p,RahmenDicke/2f,RahmenDicke/2f,
      pic.Width + RahmenDicke, pic.Height + RahmenDicke);
  }
  e.Graphics.DrawImage(pic, RahmenDicke,RahmenDicke);
}
```

Es gilt übrigens als gute Programmierpraxis, die `Checked`- und `Enabled`-Eigenschaften von `MenuItem`-Objekten als offizielle Zustandsvariablen zu verwenden. Das hält den Status des Programms automatisch mit der Anzeige konsistent und vermeidet überflüssige Schattenwirtschaft.

Den Behandlungsmethoden von Menübefehlen fällt eine wichtige Aufgabe zu: Sie sind die offiziellen Einsprungpunkte für die Operationen, die der Benutzer ausführen kann. Es bietet sich daher geradezu an, diese Methoden auch programmintern für die Ausführung von Operationen sowie die Initialisierung der Anwendung aufzurufen. So würde der folgende Aufruf den Rahmen beim Programmstart einschalten.

```
public Form1()
{
  InitializeComponent();
  menuToggleFrame_Click(menuToggleFrame, new EventArgs());
  menuItemZoom_Click(menuZoom100, new EventArgs());
...
```

Ein »fauler« Aufruf könnte auch so aussehen, wenn klar ist, dass die Methode keinen der beiden Parameter auswertet:

```
menuToggleFrame_Click(null, null);
```

Bei der Methode `menuItemZoom_Click()` handelt es sich um eine kollektive Behandlungsmethode für mehrere Menübefehle, deren `Text`-Eigenschaft den Zoomfaktor wiedergibt. Es ist daher kein Zufall, dass der obige Aufruf `menuZoom100` als Senderobjekt benennt. Die `Text`-Eigenschaft dieses Objekts lautet »`&100%`« und folgt damit dem allgemeinen Muster, das die notwendige Parametrisierung der Behandlungsmethode gestattet. Hier ihre Implementierung:

```
private MenuItem LastZoom;
private System.Drawing.Drawing2D.Matrix transform;
private void menuItemZoom_Click(object sender, System.EventArgs e)
{
  if (LastZoom != null)
    LastZoom.Checked = false;
  LastZoom  = (MenuItem) sender;
  LastZoom.Checked = true;
  float scale =
    float.Parse(LastZoom.Text.Substring(1, LastZoom.Text.Length-2))/100;
  transform = new System.Drawing.Drawing2D.Matrix();
  transform.Scale(scale, scale);
  Invalidate();
}
```

Startbefehle setzen Sie nicht in den Konstruktor, sondern in die `Load`-Behandlung des Formulars:

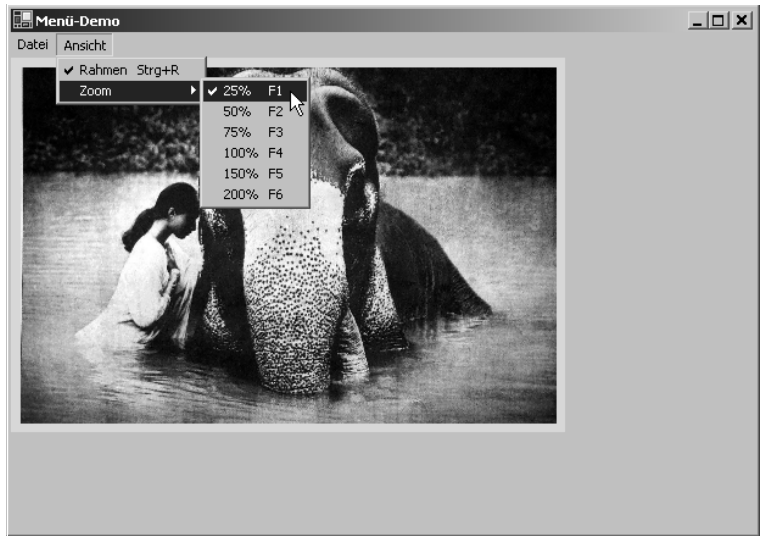


Abbildung 15.11: Das Zoom-Menü

```
private void Form1_Load(object sender, System.EventArgs e)
{
  menuOpen_Click(menuOpen, new EventArgs());
}
```

Menü zur Laufzeit erweitern

Es versteht sich, dass sich Menüs auch zur Laufzeit umbauen oder erweitern lassen. Eine häufige Aufgabe ist es beispielsweise, im DATEI-Menü eine Liste mit den zuletzt geöffneten Dateien anzubieten. Der Aufwand dafür hält sich in Grenzen.

Da `MenuItem`-Objekte leider keine `Tag`-Eigenschaft anbieten, müssen die Dateinamen in einer eigenen fifo-tauglichen Struktur gespeichert werden, wofür beispielsweise die `StringCollection`-Klasse verwendbar ist. Zur Abtrennung vom restlichen Menü empfiehlt sich eine Trennlinie in Form eines eigenen Menüeintrags, dessen `Text`-Eigenschaft auf den Wert »-« gesetzt wird. `MenuItem`-Objekte, deren `Text`-Eigenschaft diesen Wert aufweist, werden als 3D-Trennlinie gezeichnet, und nicht als eigenständiger Menüeintrag. Danach kommen die Menueinträge für die Dateinamen.

Für jeden hinzukommenden Dateinamen wird ein neues `MenuItem`-Objekt generiert, mit einer generischen Behandlungsroutine verbunden und in die `MenuItems`-Auflistung des Objekts für das DATEI-Menü eingefügt. Außerdem muss der qualifizierte Dateiname als neues Element in das `StringCollection`-Objekt eingefügt werden.

Der Rest ist eine Frage der Listenverwaltung. Überschreitet die Anzahl der gespeicherten Dateinamen eine Maximalzahl, wird der älteste Dateiname

aus den Auflistungen entfernt. Außerdem sollte ein Dateiname nicht mehrfach als Eintrag erscheinen, und die aktuell angezeigte Datei hat noch nichts in der Liste verloren. Beachten Sie hierzu die Kommentare im Code:

```
private void menuOpen_Click(object sender, System.EventArgs e)
{
  OpenFileDialog ofd = new OpenFileDialog();
  ofd.Filter = "*.jpg | *.jpg";
  if(DialogResult.OK == ofd.ShowDialog())
  {
    try { pic = new Bitmap(ofd.FileName); }
    catch {return;}
  }
  else
    return;

  // RecentFile-Liste pflegen
  const int StoredFilenames = 5;
  if (menuFile.MenuItems.Count == 3)
    menuFile.MenuItems.Add(3, new MenuItem("-")); // Trennlinie
  MenuItem mi = new MenuItem(System.IO.Path.GetFileName(ofd.FileName));
  mi.Click += new EventHandler(menuFileName_Click);
  menuFile.MenuItems.Add(4, mi);

  // Eintrag für angezeigtes Bild momentan noch unsichtbar
  mi.Visible = false;

  // existiert Eintrag schon? Wenn ja, dann rausschmeißen
  int Index;
  if((Index = FileNames.IndexOf(ofd.FileName)) >= 0)
  {
    FileNames.RemoveAt(Index);
    menuFile.MenuItems.RemoveAt(5+Index);
  }

  // Eintrag vorhanden? dann Trennlinie einschalten
  if (menuFile.MenuItems.Count > 5)
  {
    menuFile.MenuItems[5].Visible = true;
    menuFile.MenuItems[3].Visible = true;
  }
  else
    menuFile.MenuItems[3].Visible = false;

  // die andere Liste pflegen
  FileNames.Insert(0,ofd.FileName);
  if (FileNames.Count > StoredFilenames)
  {
    FileNames.RemoveAt(StoredFilenames);
    menuFile.MenuItems.RemoveAt(4 + StoredFilenames);
  }
  Invalidate();
}
```

Nun fehlt noch die `Click`-Behandlung der neuen Einträge. Sie errechnet aus dem Index des Menüeintrags einen Index für das `StringCollection`-Objekt und versucht dann, die Datei zu öffnen. Gelingt dies, muss die Methode noch die Dateinamen und die Menüeinträge für das aktuelle und das neue Bild vertauschen, damit Menü und Darstellung aufeinander abgestimmt bleiben. Daraus entstehen keine weiteren Sonderfälle, da `menuFileName_Click()` aus logischen Gründen erst dann zum Aufruf kommen kann, wenn ein Dateiname im Menü angezeigt wird, weshalb dort immer mindestens zwei Einträge und zwei Dateinamen vorhanden sind.

```
private void menuFileName_Click(object sender, System.EventArgs e)
{
  MenuItem mi = (MenuItem)sender;
  int Index = mi.Index - 4;
  try
  {
    pic = new Bitmap(FileNames[Index]);
  }
  catch {};

  // Dateinamen vertauschen
  string s = FileNames[Index];
  FileNames[Index] = FileNames[0];
  FileNames[0]= s;

  // Text vertauschen
  s = mi.Text;
  mi.Text = menuFile.MenuItems[4].Text;
  menuFile.MenuItems[4].Text = s;

  Invalidate();
}
```



Abbildung 15.12: Die Dateiliste im Datei-Menü

Kontextmenüs

Für ein ordentliches Programm ziemt es sich auch, sichtbaren Objekten Kontextmenüs zur Verfügung zu stellen, die eine schnelle, auf den jeweiligen Zustand (Kontext) der Anwendung und/oder des Objekts zugeschnittene Auswahl an Befehlen per Rechtsklick verfügbar machen.

Vom Prinzip her können Sie Kontextmenüs als Instanzen des `ContextMenu`-Steuerelements per Designer einfügen, nach dem gleichem Schema wie das Hauptmenü im Designer zusammenstellen und gleich der `ContextMenu`-Eigenschaft des jeweiligen Steuerelements zuordnen. Das Kontextmenü ist dann sofort einsatzbereit – ohne dass eine `MouseDown`-Behandlung mit `Show()`-Aufruf erforderlich wäre. Die Verdrahtung der Befehle ist eine andere Geschichte. Im Allgemeinen können Sie hier auf die Behandlungsmethoden des Hauptmenüs zurückgreifen, weshalb dieses auch *vor* den Kontextmenüs fertig gestellt und getestet werden sollte.

Unter Kontextsensitivität versteht man im Allgemeinen aber mehr, als nur vorgefertigte Menüs bereitzustellen. Kontextmenüs werden daher im Allgemeinen zur Laufzeit zusammengestellt – unter Verwendung der Menüeinträge des Hauptmenüs und eventuell weiterer Einträge für dort nicht zu findende Befehle.

Der `ContextMenu`-Konstruktor ist überladen und ermöglicht es, fertige Kontextmenüs auf der Basis von `MenuItem`-Arrays zu generieren. So einfach das klingt, die Sache hat einen Haken: Ein `MenuItem`-Objekt ist eine Systemressource und kann zu einem Zeitpunkt immer nur einem Menü angehören. Es ist daher mit Überraschungen verbunden, wenn Sie `MenuItem`-Objekte aus dem Hauptmenü in ein solches Array stecken, um daraus ein Kontextmenü zu basteln. Sie werden Ihnen später dort fehlen. Das gilt auch, wenn Sie ein leeres `ContextMenu`-Objekt generieren und die `MenuItem`-Objekte per `Add()` in die `MenuItems`-Auflistung verfrachten. Der einzig gangbare Weg führt hier über die `CloneMenu()`-Methode der `MenuItem`-Objekte, wie der folgende Codeauszug verdeutlicht:

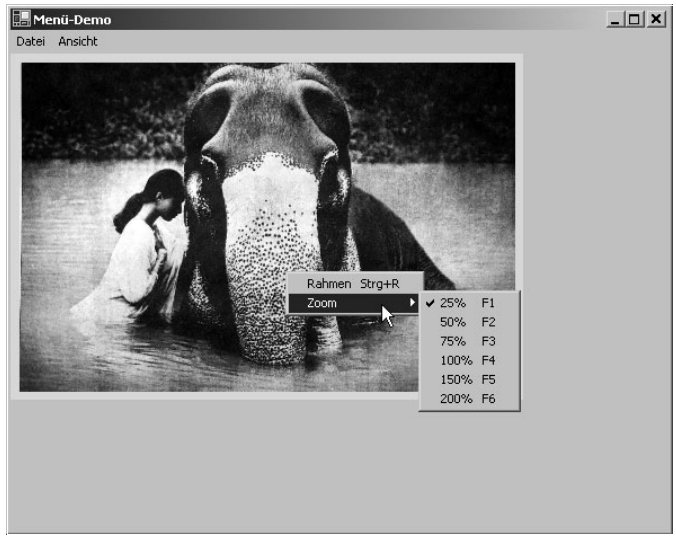
```
public Form1()
{
  InitializeComponent();
  this.ContextMenu = new ContextMenu();
  foreach(MenuItem m in menuView.MenuItems)
    this.ContextMenu.MenuItems.Add(m.CloneMenu());
...
```

Der Code ordnet dem Formularobjekt eine Kopie des Ansichtsmenüs als Kontextmenü zu. Mehr ist in diesem Fall nicht zu tun. Da die Menüeinträge zu dem Zeitpunkt bereits komplett »verdrahtet« sind, ist das neue Kontextmenü des Formulars gleichfalls sofort einsatzfähig.

Dennoch ergibt sich ein lästiges Problem, das damit zusammenhängt, dass die Behandlungsmethode immer nur den Zustand des `MenuItem`-Objekts ändert, welches das Ereignis signalisiert hat und nicht den der Kopie. Die Zustände müssen also synchronisiert werden, was die Methode `menuItemZoom_Click()` etwas komplexer ausfallen lässt.

```
private void menuItemZoom_Click(object sender, System.EventArgs e)
{
  if (LastZoom != null)
  {
    LastZoom.Checked = false;
  }
  LastZoom  = menuZoom.MenuItems[((MenuItem) sender).Index]
  LastZoom.Checked = true;
  this.ContextMenu = new ContextMenu();
  foreach(MenuItem m in menuView.MenuItems)
    this.ContextMenu.MenuItems.Add(m.CloneMenu());
  float scale = float.Parse(LastZoom.Text.Substring(1,
    LastZoom.Text.Length-2))/100;
  transform = new System.Drawing.Drawing2D.Matrix();
  transform.Scale(scale, scale);
  Invalidate();
}
```

Abbildung 15.13: Das Kontextmenü



Menüeinträge selbst zeichnen

Natürlich ist für Menüs auch die »Besitzerzeichnung« möglich. Besitzerzeichnung (owner draw) heißt, dass der Besitzer eines Objekts – im vorliegenden Fall das Formularobjekt – das Objekt zeichnet. Der Ablauf ist vom Prinzip her derselbe wie bei den `ListBox`- und `ListView`-Steuerelementen, nur dass Sie nicht immer gleich ein ganzes Menü, sondern auf der Ebene des ein-

zelnen Menübefehls zeichnen. (Es versteht sich, dass sich auf diese Weise dann auch ganze Menüs zeichnen lassen.)

Um einen Menübefehl selbst zu zeichnen, sind drei Dinge nötig:

1. Setzen Sie die `OwnerDraw`-Eigenschaft des zugehörigen `MenuItem`-Objekts auf `true`.
2. Ergänzen Sie eine Behandlungsmethode für das `MeasureItem`-Ereignis, die die Eigenschaften `ItemWidth` und `ItemHeight` des `MeasureItemEventArgs`-Ereignisobjekts auf passende Werte (in Bildpunkten) setzt. (Falls das `MenuItem`-Objekt in der Menüleiste gezeichnet wird, sollten Sie `ItemHeight` nicht verändern!)
3. Ergänzen Sie eine Behandlungsmethode für das `DrawItem`-Ereignis, die den Menüeintrag in den vom `DrawItemEventArgs`-Ereignisobjekt bereitgestellten Grafikkontext `Graphics` zeichnet.

Die ersten beiden Punkte sind schnell erledigt. Für den dritten müssen dagegen mindestens zwei Zustände unterschieden werden – und wenn die Eigenschaften `Enabled` und `Checked` des `MenuItem`-Objekts eine Rolle spielen, sind es erheblich mehr. Das reichlich mit Eigenschaften und Methoden ausgestattete `DrawItemEventArgs`-Ereignisobjekt liefert dafür aber (fast) alles Notwendige.

Codebeispiel – MenüDemoOwnerDraw

Der folgende Codeauszug aus dem Projekt *MenüDemoOwnerDraw* erweitert den Funktionsumfang der bisher diskutierten *MenüDemo*-Anwendung dahingehend, dass die Menüeinträge für die zuletzt geöffneten Dateien vor dem Dateinamen noch eine Miniatur des jeweiligen Bildes zeigen (Bild 15.14).

Die Implementierung stützt sich der Einfachheit halber auf eine Instanz des `ImageList`-Steuerelements, das Miniaturen der jeweiligen Bilder speichert und in den `ImageSize`-Abmessungen 50x50 bereitstellt. Hier erst einmal die erweiterte Fassung der Methode `menuOpen_Click()`, die zusätzlich nun auch noch die Bildliste pflegt und zwei Behandlungsmethoden für das neue `MenuItem`-Objekt registriert:

```
private void menuOpen_Click(object sender, System.EventArgs e)
{
  OpenFileDialog ofd = new OpenFileDialog();
  ofd.Filter = "*.jpg | *.jpg";
  if(DialogResult.OK == ofd.ShowDialog())
  {
    try { pic = new Bitmap(ofd.FileName); }
    catch {return;}
  }
  else
    return;
```



Abbildung 15.14: Selbst gezeichnete Menüeinträge

```
  // RecentFile-Liste pflegen
  const int StoredFilenames = 5;
  if (menuFile.MenuItems.Count == 3)
    menuFile.MenuItems.Add(3, new MenuItem("-")); // Trennlinie
  MenuItem mi = new MenuItem(System.IO.Path.GetFileName(ofd.FileName));
  mi.Click += new EventHandler(menuFileName_Click);
  mi.MeasureItem += new MeasureItemEventHandler
(menuFileName_MeasureItem);
  mi.DrawItem += new DrawItemEventHandler (menuFileName_DrawItem);
  mi.OwnerDraw = true;
  menuFile.MenuItems.Add(4, mi);
  // Eintrag für angezeigtes Bild momentan noch unsichtbar
  mi.Visible = false;

  // existiert Eintrag schon? Wenn ja, dann rausschmeißen
  int Index;
  if((Index = FileNames.IndexOf(ofd.FileName)) >= 0)
  {
    FileNames.RemoveAt(Index);
    imageList1.Images.RemoveAt(imageList1.Images.Count-Index-1);
    menuFile.MenuItems.RemoveAt(5+Index);
  }
  // Eintrag vorhanden? dann Trennlinie einschalten
  if (menuFile.MenuItems.Count > 5)
  {
    menuFile.MenuItems[5].Visible = true;
    menuFile.MenuItems[3].Visible = true;
  }
  else
    menuFile.MenuItems[3].Visible = false;
  // die anderen Listen pflegen
  FileNames.Insert(0,ofd.FileName);
```

```
    imageList1.Images.Add(pic);
    if (FileNames.Count > StoredFilenames)
    {
      FileNames.RemoveAt(StoredFilenames);
      menuFile.MenuItems.RemoveAt(4 + StoredFilenames);
      imageList1.Images.RemoveAt(0);
    }
    Invalidate();
}
```

Nachdem die `Add()`-Methode der `Images`-Eigenschaft neue Elemente nur anhängen, nicht aber einfügen kann, ist die Indizierung etwas verdreht, der Rest ist aber geradlinig. Entsprechend muss natürlich auch die `menuFileName_Click()`-Methode erweitert werden:

```
private void menuFileName_Click(object sender, System.EventArgs e)
{
  MenuItem mi = (MenuItem)sender;
  int Index = mi.Index - 4;
  try
  {
    pic = new Bitmap(FileNames[Index]);
  }
  catch {};
  // Dateinamen vertauschen
  string s = FileNames[Index];
  FileNames[Index] = FileNames[0];
  FileNames[0]= s;
  // Text vertauschen
  s = mi.Text;
  mi.Text = menuFile.MenuItems[4].Text;
  menuFile.MenuItems[4].Text = s;
  // Bild vertauschen
  int actualIdx = imageList1.Images.Count-Index-1;
  Image i = imageList1.Images[imageList1.Images.Count-1];
  imageList1.Images[imageList1.Images.Count-1] =
imageList1.Images[actualIdx];
  imageList1.Images[actualIdx] = i;

  Invalidate();
}
```

Die `MeasureItem`-Behandlung kümmert sich nur um die benötigten Abmessungen: Die Höhe `ItemHeight` richtet sich nach dem Bild. Für `ItemWidth` könnte man auch noch den Platzbedarf für den Dateinamen über die `MeasureString()`-Methode der `Graphics`-Eigenschaft einkalkulieren. Es ist aber durchaus legitim, es mit dem Vorgabewert auf sich beruhen zu lassen – längere Dateinamen werden dann zwar abgeschnitten, aber das ist in der Praxis oft besser, als wenn das Menü die halbe Bildschirmbreite ausfüllt. Der folgende Code kalkuliert den Platzbedarf für den Dateinamen mit ein und muss dabei auf die Eigenschaft `SystemInformation.MenuFont` zurückgreifen, da das Ereignisobjekt keine `Font`-Eigenschaft mitbringt und die Klasse `MainMenu` selbst keine hat:

```
private void menuFileName_MeasureItem(object sender,
  System.Windows.Forms.MeasureItemEventArgs e)
{
  MenuItem mi = (MenuItem)sender;
  e.ItemHeight = imageList1.ImageSize.Height + 4;
  int w = (int) e.Graphics.MeasureString(
                mi.Text, SystemInformation.MenuFont).Width;
  e.ItemWidth = imageList1.ImageSize.Width + 4 + w;
}
```

Das Beispiel verwendet allerdings eine mehrzeilige Wiedergabe mit automatischem Umbruch auf der Basis eines `StringFormat`-Objekts, was nicht für jeden Dateinamen die beste Lösung ist.

Die `DrawItem`-Behandlung hat etwas mehr zu tun. Dummerweise liefert das `DrawItemEventArgs`-Ereignisobjekt für den nicht ausgewählten Zustand die falsche Hintergrundfarbe, weshalb eine explizite Fallunterscheidung auf Basis der `State`-Eigenschaft erforderlich wird, um den richtigen Hintergrund zu zeichnen. Die für das System geltende Menü-Farbe kann aber beispielsweise der `KnownColor`-Auflistung entnommen werden, so dass statt `DrawBackground()` ein `FillRectangle()`-Aufruf zum gewünschten Ergebnis führt. Der Rest ist schlichtes Zeichnen in den Bereich, den die `Bounds`-Eigenschaft beschreibt:

```
private void menuFileName_DrawItem(object sender,
System.Windows.Forms.DrawItemEventArgs e)
{
  const int picOffset = 2;
  MenuItem mi = (MenuItem)sender;
  int Index = imageList1.Images.Count - 1 - (mi.Index - 4);
  if( (e.State & DrawItemState.Selected) > 0)
    e.DrawBackground();
  else
    e.Graphics.FillRectangle(
      new SolidBrush(Color.FromKnownColor(KnownColor.Menu)), e.Bounds);
  imageList1.Draw(e.Graphics, e.Bounds.X +picOffset,
    e.Bounds.Y + picOffset , Index);
  // Dateinamen ggf. mit Zeilenumbruch ausgeben
  Rectangle rect = e.Bounds;
  rect.X += imageList1.ImageSize.Width + picOffset;
  rect.Width -= imageList1.ImageSize.Width + picOffset;
  StringFormat sf = new StringFormat();
  sf.LineAlignment = StringAlignment.Center;
  e.Graphics.DrawString(mi.Text, e.Font, new SolidBrush(e.ForeColor),
    rect, sf);
}
```

Weitere Codebeispiele – Diaprojektor

Weitere Codebeispiele für vollständige Anwendungen mit/ohne Menü, Kontextmenü, Tastaturschnittstelle und Vollbildansicht finden Sie in den Projekten *Diaprojektor1*, *Diaprojektor2* und *DiaprojektorOLE*. Die drei Anwendungen sind Variationen eines Viewers für Bitmaps, der – wie ein Diaprojektor – Bilder in passender Skalierung mit Timerautomatik anzeigt. *DiaprojektorOLE* enthält, wie der Name andeuten will, eine Unterstützung für Drag&Drop.

15.2.8 Bildliste (ImageList)

Bei der Bildliste `ImageList` handelt es sich um eine Komponente, die als solche nicht von `Control` abstammt, sondern direkt von der Klasse `Component`. `ImageList` stellt die Infrastruktur für verschiedene komplexe Steuerelemente wie `TreeView`, `ListView` und `ToolBar`, die mit einer unbestimmten Zahl von Bildern gleicher Größe hantieren müssen.

Nach außen hin präsentiert sich eine Bildliste als Sammlung beliebig vieler Bitmaps, die alle dieselben Abmessungen `ImageSize` und dieselbe Farbtiefe `ColorDepth` aufweisen. Was aber nicht heißt, dass Sie nicht auch Bilder unterschiedlicher Größe und Farbtiefen in einer Bildliste speichern können. Die Bildliste speichert die Bilder nach Möglichkeit in der Originalgröße, sofern diese die maximalen Abmessungen für die `ImageSize`-Eigenschaft, 256x256 Pixel, nicht überschreitet. Größere Bilder werden auf diese Größe heruntergerechnet. Änderungen der `ColorDepth`-Eigenschaft zur Laufzeit werden mit einem Laufzeitfehler geahndet.

Für das Einfügen eines beliebigen `Bitmap`-, `Icon`- oder `Image`-Objekts per Code verwenden Sie die `Add()`-Methode der `Images`-Auflistung, die in zwei Überladungen zur Verfügung steht: In der einparametrigen Variante setzt das Steuerelement für das neue Bild den Wert der `TransparentColor`-Eigenschaft als Transparenzfarbe, in der zweiparametrigen Variante lässt sich die Transparenzfarbe unabhängig davon übergeben. Gerade mit Blick auf Symbolleisten ist es aber auch möglich, so genannte Bitmap-Streifen – Bitmaps, in denen mehrere Bilder spaltenweise nebeneinander aufgereiht sind – einzufügen. Die entsprechende Methode hört auf den Namen `AddStrip()`. Sie akzeptiert keine Bitmaps, deren Abmessungen 256x256 Pixel überschreiten. Zudem muss die Höhe des Streifens exakt auf die `Height`-Eigenschaft der `ImageSize`-Eigenschaft abgestimmt und die Breite ein ganzzahliges Vielfaches von `ImageSize.Width` sein, andernfalls signalisiert das Steuerelement Laufzeitfehler.

Tabelle 15.24: Eigenschaften der `ImageList`-Komponente

Eigenschaft	Bedeutung
`ColorDepth ColorDepth {get; set;}`	`enum`-Wert zur Bestimmung der Farbtiefe, mit dem die Bildliste die gespeicherten Bitmaps nach außen hin präsentiert. Im Angebot stehen die Werte `Depth4Bit`, `Depth8Bit` (Vorgabewert), `Depth16Bit`, `Depth24Bit` und `Depth32Bit`. Von einer Änderung dieser Eigenschaft zur Laufzeit ist abzuraten, da dies einen Laufzeitfehler erzeugt, wenn bereits ein Handle für das Steuerelement generiert wurde, das die Bildliste benutzt.
`ImageCollection Images {get; set;}`	Auflistung, in der die Bilder der Bildliste gespeichert sind.
`int Images.Count {get;}`	Anzahl der Bilder in der Bildliste
`bool Images.Empty {get;}`	`true`, wenn die Bildliste leer ist.
`Size ImageSize {get; set;}`	Bestimmt, mit welchen Abmessungen (Pixel) der für den Elementzugriff zuständige `get`-Accessor der `Images`-Eigenschaft die Bilder liefert. Sind Bitmap-Streifen im Spiel (`AddStrip()`) kann diese Eigenschaft zur Laufzeit zwar ebenfalls geändert werden, dabei werden aber die bereits gespeicherten Bilder gelöscht.
`Color TransparentColor {get; set;}`	Vorgabewert für die Transparenzfarbe. Die Bildliste verwendet diesen Wert, wenn Bitmaps über die einparametrigen Überladungen der Methoden `Add()` bzw. `AddStrip()` in die `Images`-Auflistung eingefügt werden. Änderungen dieser Eigenschaft wirken sich *nicht* auf bereits eingefügte Bitmaps aus.

Tabelle 15.25: Methoden der `ImageList`-Komponente und ihrer `Images`-Auflistung

Methode	Bedeutung
`int Images.Add()`	Fügt die angegebene Bitmap in die Bildliste ein. Die Bitmap kann eine beliebige Größe haben. Sie wird bis zu den Abmessungen 256x256 Pixel in ihrer Originalgröße gespeichert, ansonsten auf diese Größe heruntergerechnet. Die zweiparametrige Überladung liefert den Index des eingefügten Elements und ermöglicht die Angabe einer individuellen Transparenzfarbe für die Bitmap, die sich nachträglich nicht mehr ändern lässt.
`int Images.AddStrip()`	Fügt die angegebene Bitmap in die Bildliste ein, unterteilt sie aber in Abschnitte mit der über die `ImageSize`-Eigenschaft festgelegten Größe. Die Höhe der Bitmap muss `ImageSize.Height` betragen, die Breite ein ganzzahliges Vielfaches von `ImageSize.Width`. Die Kacheln werden dann in Lesrichtung in die Bildliste eingefügt. Eine zweiparametrige Überladung zur Angabe einer individuellen Transparenzfarbe ist nicht vorhanden, weshalb Sie in diesem Fall mit der `TransparentColor`-Eigenschaft arbeiten müssen. Die Methode liefert den Index des ersten eingefügten Elements.
`void Images.Clear()`	Löscht die Bildliste.
`bool Images.Contains()`	`true`, wenn die angegebene Bitmap in der Bildliste gespeichert ist.
`void Draw()`	Zeichnet ein Bild der Bildliste in der angegebenen Größe an die angegebene Position in den angegebenen Grafikkontext. Ausgabegrößen jenseits von 256x256 Pixel sind zwar möglich, wirken aber schnell »pixelig«, da die interne Repräsentation der Bilder auf diese Größe beschränkt ist.
`int Images.IndexOf()`	Liefert den Index des angegebenen Bildes.
`int Images.Remove()`	Entfernt das angegebene Bild aus der Bildliste.
`int Images.RemoveAt()`	Entfernt das Bild mit dem angegebenen Index aus der Bildliste.

Kacheln einfügen

Obwohl sich keine Methode dafür findet, eine Bitmap auch reihenweise bzw. reihen- und spaltenweise in Kacheln zu unterteilen, lässt sich dies ohne viel Aufwand erreichen. Die folgende Methode unterteilt eine Bitmap in *n* x *m* Kacheln und fügt diese in Lesrichtung in die Bildliste ein.

```
bool AddTiles(ImageList il, Image i, int rows, int columns)
{
  if (rows <=0 || columns <= 0 ||
    i.Height > 256 || i.Width > 256 ||
    i.Height % rows != 0 || i.Width % columns != 0)
    return false;
  ImageList ilTemp = new ImageList();
  // erst die Zeilen als Streifen in ilTemp einfügen
  i.RotateFlip(RotateFlipType.Rotate270FlipNone);
  ilTemp.ImageSize = new Size((int)(i.Width/rows), i.Height);
  ilTemp.Images.AddStrip(i);
  il.ImageSize = new Size((int)(i.Width / columns),
    (int)(i.Height / rows));
  // Zeilenstreifen wiederum als Streifen in il einfügen
  for (int x = 0 ; x < rows; x++)
  {
    Bitmap b = new Bitmap(ilTemp.Images[x]);
    i = b;
    i.RotateFlip(RotateFlipType.Rotate90FlipNone);
    il.Images.AddStrip(i);
  }
  return true;
}
```



Bitmaps, die als Streifen in einem `ImageList`*-Steuerelement gespeichert sind, können mit* `RotateFlip()` *nur dann gedreht werden, wenn sie quadratisch sind – was unwahrscheinlich ist. Andernfalls signalisiert die Bildliste eine Ausnahme.*

Eine alternative Formulierung dieser Methode, die Kachelgrößen bis 256 x 256 Pixel erlaubt und ohne die einschränkende Methode `AddStrip()` auskommt, zeichnet die Kacheln in neue Bitmaps und fügt diese dann mit `Add()` in die Bildliste ein:

```
int AddTilesEx(ImageList il, Image i, int rows, int columns)
{
  int w = (int) i.Width/columns;
  int h = (int) i.Height/rows;
  if (h > 256 || w > 256 ||
      h*rows != i.Height || w * columns != i.Width )
    return -1;
  int Index = il.Images.Count;
  il.ImageSize = new Size(w, h);
  for(int x = 0; x < rows*columns; x++)
  {
    Image iTemp = new Bitmap(w,h);
    Graphics g = Graphics.FromImage(iTemp);
    Rectangle srcRect = new Rectangle(w*(x % columns),
      h*(int)(x/columns), w, h);
    g.DrawImage(i,0,0,srcRect,GraphicsUnit.Pixel);
    il.Images.Add(iTemp);
  }
  return Index;
}
```

Diese Routine lässt sich auch anstelle von `AddStrip()` einsetzen. Dabei vermeidet sie nicht nur die Nachteile der Methode, sie ermöglicht insbesondere auch die spätere Skalierung der Bilder über die `ImageSize`-Eigenschaft zur Laufzeit.

Ärger mit der Transparenzfarbe

Die interaktive Bestückung einer Bildliste im Designer ist zwar weitgehend intuitiv, birgt aber doch einen Pferdefuß: Die Bildliste ordnet eingefügten Bitmaps den in der `TransparentColor`-Eigenschaft gesetzten Vorgabewert zu, sieht aber keine Möglichkeit vor, diesen Wert nachträglich zu ändern. (Das liegt daran, dass die Bildliste Bitmaps, die im RGB-Format vorliegen, beim Einfügen in das ARGB-Format umrechnet und dabei den Alphawert aller Bildpunkte mit Transparenzfarbe auf 0 setzt.) Mit anderen Worten, Sie müssen für jede einzelne Bitmap den RGB-Wert (oder Namen) der Transparenzfarbe wissen und diesen explizit der `TransparentColor`-Eigenschaft zuweisen, *bevor* Sie die Bitmap einfügen. Warum der IMAGE-AUFLISTUNGS-EDITOR keine Möglichkeit bietet, wenigstens die `TransparentColor`-Eigenschaft im Dialog zu ändern, ist unverständlich. Mit anderen Worten, Sie müssen den Auflistungseditor jedes Mal verlassen, um die `TransparentColor`-Eigenschaft neu zu setzen.

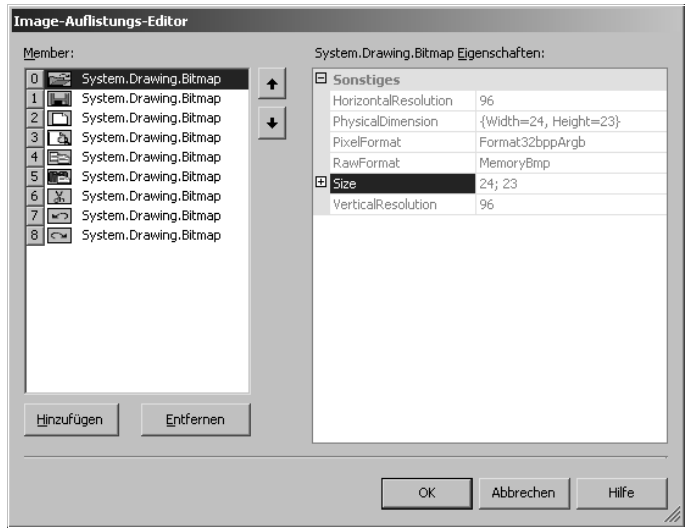


Abbildung 15.15: Der Auflistungseditor ermöglicht das interaktive Einfügen von Bitmaps in die Bildliste, die dann als Ressource zusammen mit der Exe-Datei gespeichert werden.

Dennoch empfiehlt es sich, Bildlisten bereits im Designer zu bestücken, da dies den unschlagbaren Vorteil hat, dass die Bilddaten auf diese Weise gleich im Ressourcen-Abschnitt der Exe-Datei landen und keine Dateien mitgeschleppt werden müssen. Der vom Designer generierte Code für das Auslesen der Ressource sieht übrigens so aus:

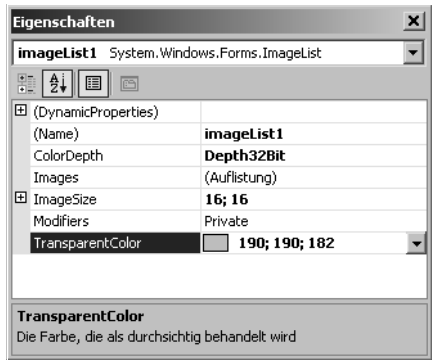


Abbildung 15.16: Für die Eigenschaft `TransparentColor` lassen sich auch RGB-Werte angeben.

```
System.Resources.ResourceManager resources =
  new System.Resources.ResourceManager(typeof(Form1));
...
this.imageList1.ImageStream =
  ((System.Windows.Forms.ImageListStreamer)
   resources.GetObject("imageList1.ImageStream")));
```

Um den Ärger mit der Transparenzfarbe für bestehende Bitmaps zu vermeiden, gibt es seit jeher die Konvention, dass der Bildpunkt mit den Koordinaten (0, 0) – also der linke obere Bildpunkt – der Bitmap Transparenzfarbe trägt. Für Bitmaps, die sich an diese Konvention halten (das sind übrigens die meisten, aber leider nicht alle Standardsymbole von Windows) kann die Transparenzfarbe automatisch bestimmt und mit der folgenden Methode auch nachträglich gesetzt werden:

```
private void MakeSymbolsTransparent(ImageList il)
{
  int Count = il.Images.Count;
  // Bilder einzeln auslesen, Tranzparenzfarbe bestimmen
  // und wieder zurückspeichern
  for(int i = 0; i < Count; i++)
  {
    Bitmap b = new Bitmap(il.Images[i]);
    il.Images.Add(b,b.GetPixel(0,0));
  }
  // Orignale Bilder rausschmeißen
  for(int i = 0; i < Count; i++)
    il.Images.RemoveAt(0);
}
```

Ein Beispiel für den praktischen Einsatz einer Bildliste finden Sie im Abschnitt »Codebeispiel – Symbol- und Statusleiste« auf Seite 626.

15.2.9 Symbolleiste – ToolBar

Ernstzunehmende Anwendungen schmücken sich meist mit einer oder mehreren Symbolleisten, deren Schaltflächen im Allgemeinen den schnellen

Zugang zu wichtigen Menübefehlen ermöglichen. Leider sieht der Designer keine automatische Zuordnung von Symbolschaltflächen zu Menübefehlen vor (`ToolBarButton`-Objekte signalisieren kein `Click`-Ereignis), weshalb Sie diese Zuordnung selbst vornehmen müssen.

Natürlich können Sie eine Symbolleiste auch per Code zusammenstellen. Das interaktive Design dürfte allerdings um einiges effizienter sein: Symbolleisten lassen sich nämlich vollständig im Designer definieren, Sie müssen dann nur eine einzige Behandlungsmethode (nämlich für das Ereignis `ButtonClick`) implementieren, um sie in das Programm einzubinden.

Symbolleiste im Designer zusammenstellen

Die Ausstattung eines Formulars mit einer Symbolleiste geschieht mit folgenden Schritten:

1. Platzieren Sie im Designer eine Instanz des `ToolBar`-Steuerelements auf dem Entwurf. In der Voreinstellung dockt die Symbolleiste am oberen Rand des Clientbereichs an.
2. Fügen Sie eine Instanz der `ImageList`-Komponente hinzu, aus der die Symbolleiste ihre Bitmaps für ihre `ToolBarButton`-Schaltflächen bezieht. Stellen Sie darin – am besten interaktiv – eine Sammlung aller Bitmaps zusammen, die Sie als Symbole für die Schaltflächen benötigen. Im Verzeichnis *Visual Studio .NET\Common7\Bitmaps* finden Sie eine kleine Auswahl üblicher Symbolleisten-Bitmaps, die Sie natürlich verwenden können. Da diese Bitmaps unterschiedliche Hintergrundfarben haben, ist es oft besser, die für die `Enabled`-Eigenschaft benötigte Transparenzfarbe zur Laufzeit einzustellen (vgl. Seite 617). Alternativ ist es möglich, die früher noch verbreiteten Bitmap-Streifen, in denen mehrere Symbolbilder zusammengefasst sind, zur Laufzeit in die Bildliste zu übernehmen.
3. Falls die Bilder nicht die richtige Größe haben, stellen Sie die Eigenschaft `ImageSize` der Bildliste auf die gewünschten Abmessungen ein. Übliche Werte sind (16;16) für kleine Symbole und (24;23) für große Symbole. Obwohl die Bildliste die gespeicherten Bilder in beliebige Abmessungen umrechnet, empfiehlt es sich, Bitmaps mit diesen Abmessungen bereitzustellen, damit die Bildqualität nicht leidet. Änderungen der Abmessungen zur Laufzeit sind problemlos, wenn die `AutoSize`-Eigenschaft der Symbolleiste auf `true` gesetzt bleibt.
4. Weisen Sie im EIGENSCHAFTEN-Fenster der `ImageList`-Eigenschaft die Bildliste zu und rufen Sie den EIGENSCHAFTEN-Dialog der `Buttons`-Auflistung auf. Sie erhalten den TOOLBARBUTTON-AUFLISTUNGS-EDITOR, in dem Sie alle Schaltflächen zusammenstellen und gleich interaktiv mit den entsprechenden Eigenschaften versehen können(Abbildung 15.17).

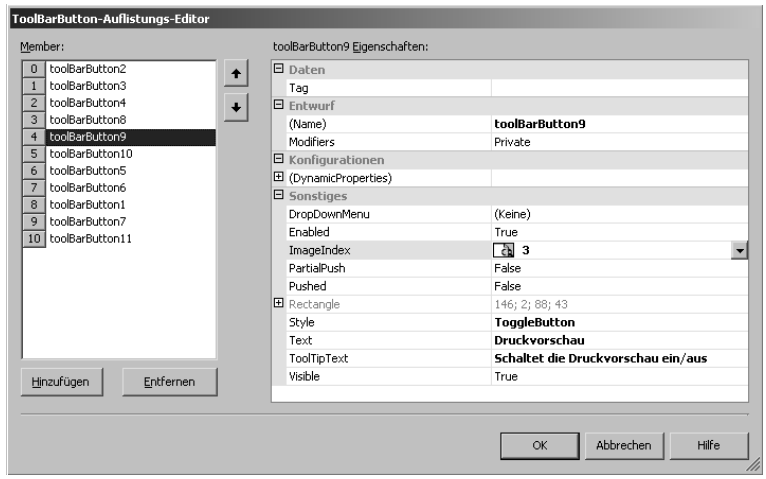


Abbildung 15.17: Einer Symbolleiste Symbolschaltflächen hinzufügen

– Weisen Sie jeder Schaltfläche über die `Style`-Eigenschaft einen von vier Stilen aus der `ToolBarButtonStyle`-Aufzählung zu. Die Voreinstellung ist `PushButton`; Die Schaltfläche funktioniert dann wie gewohnt als Taster. `ToggleButton` vergeben Sie, wenn die Schaltfläche als Ein-/Aus-Schalter mit zwei stabilen Zuständen verwendbar sein soll. (Wie beim `CheckBox`-Steuerelement lässt sich per Software noch ein dritter Zustand zuweisen, der in diesem Fall über die `PartialPush`-Eigenschaft gesteuert wird.) Falls Sie einer Schaltfläche eine Drop-Down-Liste in Form eines örtlich gebundenen Kontextmenüs zuordnen wollen, nehmen Sie die `DropDown`-Eigenschaft. Die Symbolleiste zeichnet dann (sofern ihre `DropDownButton`-Eigenschaft `true` ist) ein Pfeilsymbol rechts neben der Schaltfläche zum Öffnen des Menüs. Sie können dieser Schaltfläche über die Eigenschaft `DropDownMenu` auch gleich eine Instanz des `ContextMenu`-Steuerelements zuordnen, sofern Sie zuvor eine solche auf dem Entwurf platziert haben. (Falls nicht, verlassen Sie den Editor kurz und holen Sie dies nach. Die Menüeinträge können Sie später nachtragen – meist werden Drop-Down-Menüs von Symbolschaltflächen ohnehin vom Besitzer gezeichnet.) Der letzte Stil ist `Separator`. Er macht aus der Schaltfläche einen schlichten Abstand, den Sie verwenden können, um Schaltflächengruppen visuell voneinander abzugrenzen.

– Über die `ImageIndex`-Eigenschaft ordnen Sie allen Schaltflächen, die nicht den Stil `Separator` tragen, einen Bildindex für eine Bitmap der Bildliste zu.

– Die `Text`-Eigenschaft weisen Sie zu, wenn Sie wollen, dass – je nach Wert der `TextAlign`-Eigenschaft der Symbolleiste – unter (`ToolBarTextAlign.Underneath`) oder neben (`ToolBarTextAlign.Right`) dem Symbol der Schaltfläche eine Beschriftung erscheint. Der Wert der `ToolTipText`-Eigenschaft wird nur ausgegeben, wenn der Benutzer

mit der Maus über der Schaltfläche ca. eine Sekunde verharrt und die `ShowToolTips`-Eigenschaft der Symbolleiste auf `true` gesetzt ist. Der Text erscheint wie gewohnt in einem schwebenden Fensterchen unterhalb der Schaltfläche.

5. Nachdem alle Schaltflächen hinzugefügt sind, definieren Sie die Eigenschaften der Symbolleiste entsprechend Tabelle 15.26.

Tabelle 15.26: Eigenschaften des `ToolBar`-Steuerelements

Eigenschaft	Bedeutung
`ToolBarAppearance Appearance {get; set;}`	`enum`-Wert, der die Darstellung der Symbolleiste festlegt. Wenn `Flat`, wird nur die Symbolschaltfläche unter der Maus mit Rahmen gezeichnet, sonst alle.
`bool AutoSize {get; set;}`	Wenn `true`, passt die Symbolleiste ihre Höhe automatisch an die Höhe der Schaltflächen an (vgl. die `TextAlign`-Eigenschaft der Klasse `ToolBarButton`).
`BorderStyle BorderStyle {get; set;}`	`enum`-Wert, der die Umrandung der Symbolleiste festlegt. Zur Auswahl stehen `Fixed3D`, `FixedSingle` und `None` (Vorgabewert).
`ToolBarButtonCollection Buttons {get; set;}`	Auflistung der Schaltflächen in der Symbolleiste. Die Schaltflächen beziehen ihre Symbol-Bitmaps aus der `ImageList`-Eigenschaft der Symbolleiste.
`Size ButtonSize {get; set;}`	Standardgröße der Schaltflächen
`bool Divider {get; set;}`	Wenn `true`, zeichnet das Steuerelement eine Linie zwischen sich und der Menüleiste.
`bool DropDownArrows {get; set;}`	Bestimmt, ob die Symbolleiste einen DropDown-Pfeil neben DropDown-Schaltflächen anzeigt. (Die Anzeige wird automatisch in `InitializeComponent` auf `true` gesetzt, wenn eine Schaltfläche die `Style`-Eigenschaft `DropDown` erhält. Sie lässt sich daher nur per Code wirksam auf `false` setzen.)
`ImageList ImageList {get; set;}`	Bildliste, aus der die Symbolleiste ihre Bitmaps für die Schaltflächensymbole bezieht.
`Size ImageSize {get;}`	Abmessungen der Schaltflächensymbole in Pixel
`bool ShowToolTips {get; set;}`	Wenn `true`, blendet die Symbolleiste den Wert der `ToolTip`-Eigenschaft der betreffenden Schaltfläche als schwebendes Fenster neben dem Mauszeiger ein, wenn dieser etwa 1 Sekunde »unschlüssig« (d.h. weitgehend bewegungslos) über der Schaltfläche verharrt.

Tabelle 15.26: Eigenschaften des `ToolBar`-Steuerelements (Forts.)

Eigenschaft	Bedeutung
`ToolBarTextAlign TextAlign {get; set;}`	`enum`-Wert, der die Ausrichtung der den Schaltflächen zugeordneten `Text`-Eigenschaften in Bezug zur Symbol-Bitmap bestimmt. Zur Auswahl stehen `Unterneath` (unterhalb, Vorgabewert) und `Right` (rechts).
`bool Wrappable {get; set;}`	Wenn `true`, zeichnet die Symbolleiste ihre Schaltflächen in mehreren Reihen, falls die Breite des Fensters nicht ausreicht, um alle Schaltflächen nebeneinander darzustellen.

Tabelle 15.27: Ereignisse des `ToolBar`-Steuerelements

Ereignis	Bedeutung
`ButtonClick`	Mausklick auf die Symbolleiste. Die Information, ob eine – und welche – Schaltfläche angeklickt wurde, liefert die `Button`-Eigenschaft des Ereignisobjekts, das seinerseits den Typ `ToolBarButtonClickEventArgs` trägt.
`ButtonDropDown`	Die Pfeilschaltfläche einer DropDown-Schaltfläche wurde angeklickt. Die Information über die Schaltfläche liefert die `Button`-Eigenschaft des Ereignisobjekts (Typ: `ToolBarButtonClickEventArgs`). Falls die Eigenschaft `DropDownArrows` der Symbolleiste auf `false` gesetzt ist, wird dieses Ereignis durch einen Klick auf die DropDown-Schaltfläche selbst ausgelöst.

Tabelle 15.28: Eigenschaften des `ToolBarButton`-Objekts

Eigenschaft	Bedeutung
`ContextMenu DropDownMenu {get; set;}`	Das der Symbolschaltfläche zugeordnete Menü. Damit das Menü aufklappbar wird, muss die `Style`-Eigenschaft auf `DropDownButton` gesetzt sein. (In der kontextbezogenen Syntaxhilfe wird für diese Eigenschaft der Datentyp `Menu` ausgewiesen. Tatsächlich geht es aber um `ContextMenu`.)
`bool Enabled {get; set;}`	Wenn `false`, wird die Symbolschaltfläche deaktiviert und das Symbol als Silhouette gezeichnet. Damit das korrekt funktioniert, muss beim Einfügen der Bitmap in die Bildliste die richtige Transparenzfarbe eingestellt worden sein. Spätere Änderungen der Farbdefinition sind nicht möglich (vgl. Seite 617) .
`int ImageIndex {get; set;}`	Index des Schaltflächensymbols in der Bildliste. Für Schaltflächen ohne Bild hat diese Eigenschaft den Wert -1.
`bool PartialPush {get; set;}`	Zustand: »teilweise gedrückt« (meist für logisch voneinander abhängige Schaltflächen verwendet)

Tabelle 15.28: Eigenschaften des `ToolBarButton`-Objekts (Forts.)

Eigenschaft	Bedeutung
`bool Pushed {get; set;}`	Zustand: »gedrückt« (auch für Schaltflächen verwendbar, die nicht den Stil `ToggleButton` tragen)
`ToolBarButtonStyle Style {get; set;}`	`enum`-Wert, der den Stil der Schaltfläche bestimmt. Einer der Werte `PushButton`, `ToggleButton`, `DropDownButton`, `Separator` (siehe vorangehender Abschnitt).
`string Text {get; set;}`	Beschriftung der Schaltfläche. Erscheint abhängig von der `TextAlign`-Eigenschaft des `Toolbar`-Objekts unterhalb oder rechts neben dem Bild.
`string ToolTipText {get; set;}`	Kurzinfo zur Funktion der Schaltfläche. Der Text erscheint in einem schwebenden Fensterchen unter der Schaltfläche, wenn der Benutzer mit der Maus über der Schaltfläche verharrt und die `ShowToolTips`-Eigenschaft des `Toolbar`-Objekts gesetzt ist.

Wie in Tabelle 15.27 aufgeführt, gibt es zum Verdrahten aller `ToolBar`-Schaltflächen nur zwei spezifische Ereignisse. `ButtonClick` tritt auf, wann immer der Benutzer auf eine Schaltfläche klickt. Das Ereignisobjekt ist vom Typ `ToolBarButtonClickEventArgs` und identifiziert die Schaltfläche über die `Button`-Eigenschaft. Es gibt im Wesentlichen zwei Methoden, um die Schaltfläche einer Aktion zuzuordnen: Die `IndexOf()`-Methode der `Buttons`-Auflistung des `ToolBar`-Objekts und die `Tag`-Eigenschaft der Schaltfläche. Standardmäßig wird die erste Methode verwendet – zumal, wenn die Reihenfolge unveränderlich ist:

```
private void toolBar1_ButtonClick(object sender,
  System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
  // zugeordneten Listeneintrag ermitteln
  int Index = toolBar1.Buttons.IndexOf(e.Button);
  switch(Index)
  {
    case 0:
    // Verknüpfung mit Menübefehl
    ...
  }
```

Die zweite Methode ist allgemeiner und erlaubt beispielsweise die Verwendung von `enum`-Datentypen:

```
enum MyToolBarButton { Öffnen, Speichern, ... };
private void toolBar1_ButtonClick(object sender,
System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
  // zugeordneten Listeneintrag ermitteln
```

```
switch((MyToolBarButton )e.button.Tag)
{
  case MyToolBarButton.Öffnen:
  // Verknüpfung mit Menübefehl
  ...
}
```

Voraussetzung ist hier natürlich, dass die `Tag`-Eigenschaften der `ToolBarButton`-Objekte zuvor entsprechend definiert wurden.

Das andere Ereignis, `ButtonDropDown`, tritt auf, wenn der Benutzer das DropDown-Menü einer DropDown-Schaltfläche aufklappt. Sie behandeln dieses Ereignis, wenn der Menüinhalt *just-in-time* angepasst werden muss. Da DropDown-Menüs von Schaltflächen, aber ohnehin – wie schon gesagt – meist vom Besitzerobjekt gezeichnet werden, ist hierfür auch an anderer Stelle noch Gelegenheit (vgl. Abschnitt »Menüeinträge selbst zeichnen«, Seite 608).

Ein Beispiel für den praktischen Einsatz einer Symbolleiste finden Sie im Abschnitt »Codebeispiel – Symbol- und Statusleiste« ab Seite 626.

15.2.10 Statusleiste

Wie der Name schon sagt, ist die standardmäßig am unteren Fensterrand des Formulars angedockte Statusleiste für die Ausgabe von Statusinformationen zuständig. Vom Prinzip her ist eine `StatusBar`-Instanz sofort nach ihrer Platzierung für die einfache Textausgabe (über die `Text`-Eigenschaft) einsatzfähig. Für die Ausgabe unterschiedlicher, voneinander unabhängiger Informationen lässt sich die Statusleiste über ihre `Panels`-Auflistung in beliebig viele Bereiche unterteilen, die Sie am besten interaktiv im STATUSBARPANEL-AUFLISTUNGS-EDITOR des Designers zusammenstellen und gestalten (Abbildung 15.18). Natürlich ist die Auflistung auch per Code nach Belieben manipulierbar.

Die Ausgabemöglichkeiten eines `Panel`-Objekts beschränken sich auf die Eigenschaften `Text` und `Icon`, sieht man einmal von der mittelbaren Ausgabe über die `ToolTipText`-Eigenschaft ab. Der `Icon`-Eigenschaft muss ein Objekt vom Typ `Icon` als Wert zugeordnet werden; gewöhnliche `Bitmap`- bzw. `Image`-Objekte sind nicht geeignet. (Im Verzeichnis *Visual Studio .NET\Common7\Graphics\icons* finden Sie beispielsweise eine kleine Sammlung von *.ico*-Dateien, die sich in einer Statusleiste einsetzen lassen.)

Einen Mausklick in einen der Bereiche signalisiert das `StatusBar`-Objekt über das `PanelClick`-Ereignis. Das Ereignisobjekt von Typ `StatusBarPanelClickEventArgs` enthält unter anderem eine Eigenschaft `StatusBarPanel`, die den angeklickten Bereich identifiziert. Für spezielle Anwendungen sind weitere Informationen im Angebot, beispielsweise die Position des Mausklicks. Um

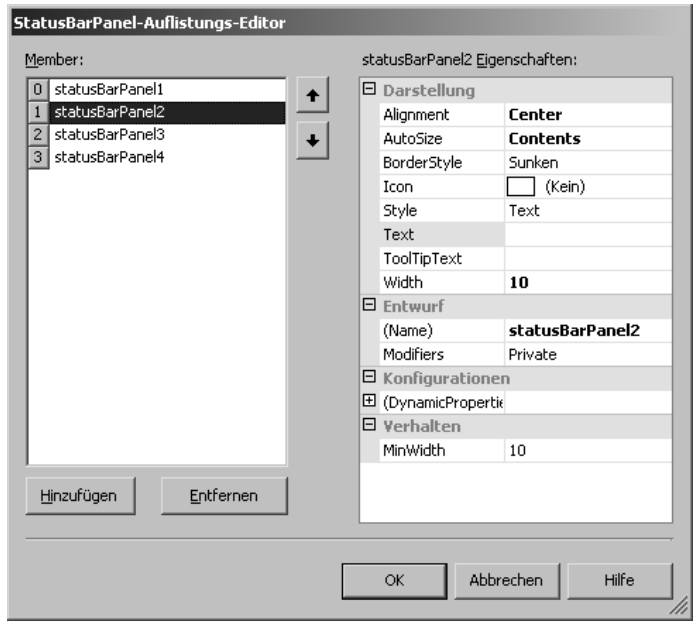


Abbildung 15.18: Der Editor zum Unterteilen der Statusleiste in verschiedene Ausgabebereiche

festzustellen, ob der Mausklick einem bestimmten Bereich galt, bedienen Sie sich der `IndexOf()`-Methode der `Panels`-Auflistung:

```
private void statusBar1_PanelClick(object sender,
System.Windows.Forms.StatusBarPanelClickEventArgs e)
{
  if (statusBar1.Panels.IndexOf(e.StatusBarPanel) == 2)
  ...
```

Ärger mit der AutoScroll-Eigenschaft des Formulars

Da eine Statusleiste über ihre `Dock`-Eigenschaft am Formularrand andockt, unterliegt sie dummerweise wie andere im Clientbereich des Formulars platzierte Steuerelemente der Wirkung der `AutoScroll`-Eigenschaft. Das heißt, ihre Sichtbarkeit hängt von der Position der vertikalen Bildlaufleiste ab. Die Lösung für dieses Problem sieht so aus:

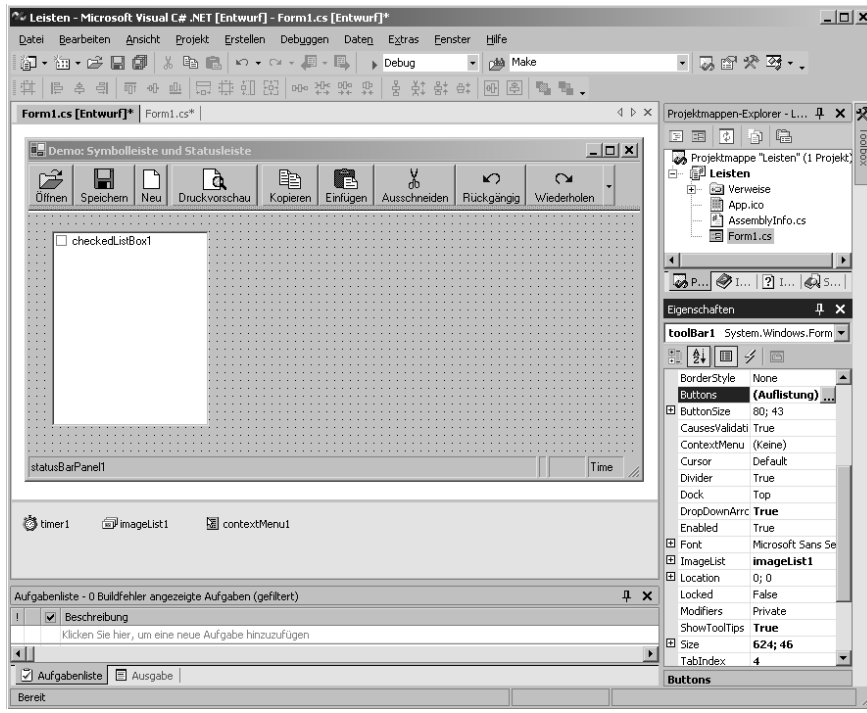
1. Fügen Sie als erstes die Statusleiste in das Formular ein.
2. Fügen Sie ein `Panel`-Steuerelement in das Formular ein, dessen `Dock`-Eigenschaft Sie auf `DockStyle.Fill` setzen. Dieses Steuerelement verfügt gleichfalls über eine `AutoScroll`-Eigenschaft.
3. Fügen Sie den gewöhnlichen Inhalt des Formulars in das `Panel`-Steuerelement ein.

Nun ist die Statusleiste immer sichtbar.

15.2.11 Codebeispiel – Symbol- und Statusleiste

Das Codebeispiel *Leisten* ist speziell darauf zugeschnitten, den Umgang mit Statusleisten und Symbolleisten zu demonstrieren. Darüber hinaus hat das Programm keinen konkreten Nutzen. Abbildung 15.19 zeigt den Entwurf.

Abbildung 15.19: Entwurfsansicht des Formulars



Zuerst ein kurzer Überblick über die implementierten Features:

1. DRUCKVORSCHAU trägt den Stil `ToggleButton` und WIEDERHOLEN den Stil `DropDownButton`. WIEDERHOLEN ist das Kontextmenü `contextMenu1` zugeordnet – beide Schaltflächen sind ohne weitere Funktion.
2. Die Symbolschaltfläche ÖFFNEN ist »verdrahtet«. Sie öffnet eine weitere Instanz des Formulars als modalen Dialog. Die anderen Symbolschaltflächen sind dahingehend verdrahtet, dass Sie den korrespondierenden Eintrag im Listenfeld `checkListBox1` auswählen.
3. Das Listenfeld `checkListBox1` listet die Bezeichnungen aller Symbolschaltflächen auf – auch die der Schaltflächen mit dem Stil `Separator`, deren `Text`-Eigenschaft zu diesem Zweck passend gesetzt wurde. Mausklicks auf die einzelnen Einträge schalten zyklisch die drei möglichen `CheckState`-Zustände durch und manipulieren die Eigenschaften `Pushed` und `Enabled` der zugeordneten Symbolschaltflächen entsprechend.

4. Die Statusleiste gibt im ersten Bereich eine Textmeldung aus, welche Symbolschaltfläche zuletzt angeklickt wurde, im zweiten Bereich den Index der angeklickten Schaltfläche, im dritten Bereich den CapsLock-Zustand (per Mausklick umschaltbar) und im vierten Bereich die aktuelle Uhrzeit.

5. Die Symbolleiste ist mit Drag&Drop-Funktionalität ausgestattet. Legt man eine oder mehrere Bilddateien (*.jpg*, *.bmp*, *.tif* etc.) darauf ab, fügt das Programm die Bilder in die Bildliste ein, generiert je Bild eine neue Symbolschaltfläche und setzt den Dateinamen als `Text`-Eigenschaft ein. Alternativ akzeptiert die Symbolleiste auch Inhalte, die originär im Bitmap-Format gehalten sind (Ziehoperationen aus Bildverarbeitungs- oder sonstigen Programmen, die dieses Format bereitstellen – beispielsweise MS Word).

6. Das Formular unterstützt die Zwischenablage. Die Tastenkombinationen Strg+V und ⇧+Einfg fügen darin gespeicherte Dateien oder Bilder als neue Schaltflächen in die Symbolleiste ein.

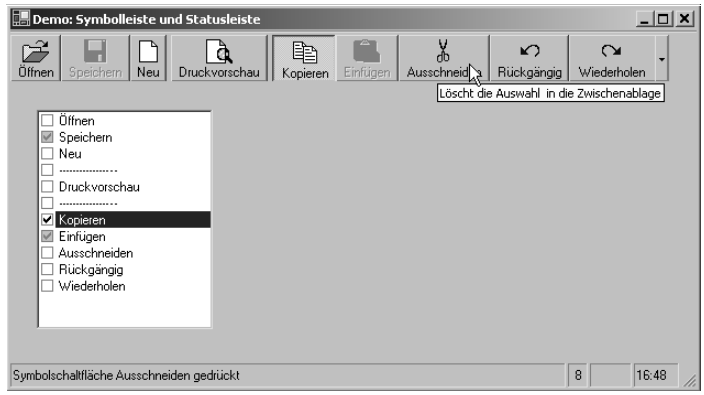


Abbildung 15.20: Das Formular in Aktion

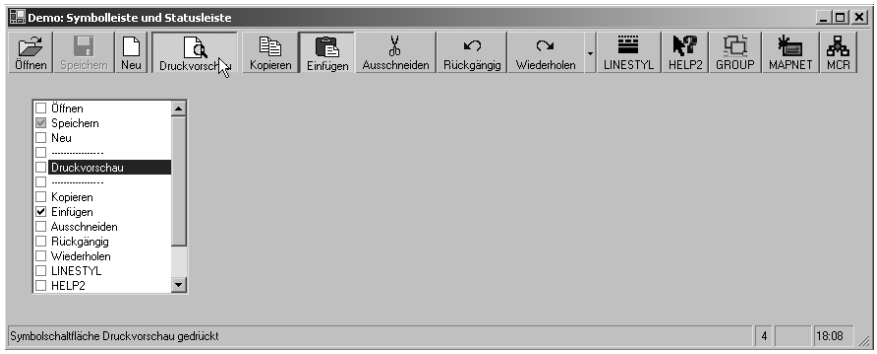


Abbildung 15.21: Die Symbolleiste mit weiteren Schaltflächen, die per Drag&Drop und über die Zwischenablage eingefügt wurden.

Implementierung

Der Konstruktor sorgt dafür, dass die Bilder in der Bildliste die richtigen Transparenzfarben erhalten und initialisiert das Listenfeld `checkedListBox1`, das für jede Symbolschaltfläche einen Eintrag mit der `Text`-Eigenschaft der Schaltfläche erhält (vgl. Abbildung 15.20).

```
public Form1()
{
  InitializeComponent();
  MakeSymbolsTransparent(imageList1);

  // ListBox mit den Text-Eignschaften der Schaltflächen füllen
  foreach(ToolBarButton b in toolBar1.Buttons)
  {
    checkedListBox1.Items.Add(b.Text, false);
  }
}
private void MakeSymbolsTransparent (ImageList il)
{
  int Count = il.Images.Count;
  // Bilder einzeln auslesen, Tranzparenzfarbe bestimmen
  // und wieder zurückspeichern
  for(int i = 0; i < Count; i++)
  {
    Bitmap b = new Bitmap(il.Images[i]);
    il.Images.Add(b,b.GetPixel(0,0));
  }
  // Orignale Bilder rausschmeißen
  for(int i = 0; i < Count; i++)
    il.Images.RemoveAt(0);
}
```

Die Methode `MakeSymbolsTransparent()` korrigiert das erwähnte Problem mit den Transparenzfarben interaktiv im Designer zusammengestellter Bildlisten. Sie geht davon aus, dass der linke obere Bildpunkt die Transparenzfarbe definiert. (Bitmaps, bei denen das nicht so ist, sollten dahingehend geändert werden.) Damit stellt sie sicher, dass Symbolschaltflächen auch richtig dargestellt werden, wenn ihre `Enabled`-Eigenschaft auf `false` gesetzt ist.

Feature 2 und ein Teil von Feature 4 stecken in der zentralen `ButtonClick`-Behandlung der Symbolleiste:

```
private void toolBar1_ButtonClick(object sender,
  System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
  // zugeordneten Listeneintrag ermitteln
  int Index = toolBar1.Buttons.IndexOf(e.Button);
  switch(Index)
  {
    case 0:
      (new Form1()).ShowDialog();
      break;
```

```
    default:
      checkedListBox1.SelectedIndex = Index;
      statusBar1.Panels[0].Text = "Symbolschaltfläche " + e.Button.Text +
        " gedrückt";
      statusBar1.Panels[1].Text = Index.ToString();
      break;
  }
}
```

Feature 3 steckt in der Behandlungsmethode für das `ItemCheck`-Ereignis des Listenfelds. Sie schaltet die drei möglichen Zustände zyklisch weiter und pflegt die Zustände der Symbolleisten:

```
private void checkedListBox1_ItemCheck(object sender,
  System.Windows.Forms.ItemCheckEventArgs e)
{
  switch (e.CurrentValue)
  {
    case CheckState.Unchecked:
      e.NewValue = CheckState.Indeterminate;
      toolBar1.Buttons[e.Index].Enabled = false;
      toolBar1.Buttons[e.Index].Pushed = false;
      break;
    case CheckState.Indeterminate:
      e.NewValue = CheckState.Checked;
      toolBar1.Buttons[e.Index].Enabled = true;
      toolBar1.Buttons[e.Index].Pushed = true;
      break;
    case CheckState.Checked:
      e.NewValue = CheckState.Unchecked;
      toolBar1.Buttons[e.Index].Enabled = true;
      toolBar1.Buttons[e.Index].Pushed = false;
      break;
  }
}
```

Die Statusleiste wird an verschiedenen Stellen im Code gepflegt. Die Behandlungsmethode für das Timer-Ereignis `Tick` gibt alle 30 Sekunden die aktuelle Uhrzeit im dritten Bereich aus:

```
private void timer1_Tick(object sender, System.EventArgs e)
{
  statusBar1.Panels[3].Text = DateTime.Now.ToShortTimeString();
  // Timer ist auf 100 msec initialisiert, damit die Zeitanzeige
  // sofort nach dem Programmstart erscheint
  timer1.Interval = 30000;
}
```

Der zweite Bereich gibt nicht nur den CapsLock-Zustand der Tastatur wieder, er generiert auch per `SendKeys.Send()` einen entsprechenden Tastendruck, der den Zustand umschaltet. (Leider scheint es in der .NET-Klassenbibliothek keine Möglichkeit zu geben, den tatsächlichen Capslock-Zustand der Tastatur abzufragen und zu setzen, weshalb Sie hier mit einer Simulation vor-

lieb nehmen müssen. Um das Beispiel übersichtlich zu halten, wurde darauf verzichtet, unverwalteten Code einzusetzen und die Funktionen `GetKeyState()` der Win32-API direkt aufzurufen.) Die Pflege dieses Bereichs übernimmt die `KeyDown`-Behandlung des Formulars, wofür die `KeyPreview`-Eigenschaft auf `true` gesetzt ist. (Den Quelltext dieser Methode, die auch die Zwischenablagenkommandos abwickelt, finden Sie im nächsten Abschnitt.)

```
private void statusBar1_PanelClick(object sender,
System.Windows.Forms.StatusBarPanelClickEventArgs e)
{
  if (statusBar1.Panels.IndexOf(e.StatusBarPanel) == 2)
  {
    SendKeys.Send("{CAPSLOCK}");
  }
}
```

Drag&Drop

Die Implementierung der Drag&Drop-Funktionalität ist etwas aufwändiger, dafür aber völlig geradlinig. Damit die Symbolleiste `DragXxx`-Ereignisse überhaupt signalisiert, muss ihre `AllowDrop`-Eigenschaft auf `true` gesetzt sein. Die Abwicklung selbst erfordert die Behandlung der Ereignisse `DragEnter` und `DragDrop`. Das Ereignisobjekt beider Ereignisse übermittelt in der `Data`-Eigenschaft eine Instanz der `IDataObject`-Schnittstelle. (Diese Schnittstelle muss jede Klasse implementieren, deren Objekte als Ole-Server auftreten wollen.) Über die `GetDataPresent()`-Methode dieses Objekts kann die `DragEnter`-Behandlung feststellen, ob der Ole-Server ein brauchbares Datenformat im Angebot hat und durch Setzen der `Effect`-Eigenschaft des Ereignisobjekts die voraussichtliche Drop-Operation ankündigen. Im vorliegenden Fall ist dies `DragDropEffects.Copy`. (Unterbleibt dieser Schritt, bricht der Ole-Server die Operation ab, da diese Eigenschaft mit `DragDropEffects.None` initialisiert ist.)

```
private void toolBar1_DragEnter(object sender,
  System.Windows.Forms.DragEventArgs e)
{
  IDataObject data = e.Data;
  if (data.GetDataPresent(DataFormats.FileDrop, true) ||
    data.GetDataPresent(DataFormats.Bitmap, true))
  e.Effect = DragDropEffects.Copy;
}
```

Die `DragDrop`-Behandlung ist für die Verwertung der abgelegten Inhalte verantwortlich. Sie delegiert dies an die Methode `OleInsert()`, die auch von der `KeyDown`-Behandlung aufgerufen wird – Kommandos für das Einfügen von Inhalten aus der Zwischenablage folgen nämlich demselben Muster:

```
private void toolBar1_DragDrop(object sender,
  System.Windows.Forms.DragEventArgs e)
{
  e.Effect = DragDropEffects.Copy;
  OleInsert(e.Data);
}

private void Form1_KeyDown(object sender, KeyEventArgs e)
{
  switch(e.KeyCode)
  {
    case Keys.CapsLock:
      if (statusBar1.Panels[2].Text == "CAPS")
        statusBar1.Panels[2].Text = "";
      else
        statusBar1.Panels[2].Text = "CAPS";
      break;
    case Keys.V:
      if (e.Modifiers == Keys.Control)
      {
        this.OleInsert(Clipboard.GetDataObject());
      }
      break;
    case Keys.Insert:
      if (e.Modifiers == Keys.Shift)
      {
        this.OleInsert(Clipboard.GetDataObject());
      }
      break;
  }
}
```

`OleInsert()` unterscheidet die beiden akzeptierten Datenformate `DataFormats.FileDrop` und `DataFormats.Bitmap`. Dateioperationen wickelt der Ole-Server dadurch ab, dass seine `GetData()`-Methode ein `string`-Array mit den Namen der betroffenen Dateien bereitstellt, mit denen der Ole-Client dann verfahren kann, wie er will. (Wenn der Client nach erfolgreicher Verwertung der Inhalte eine `Move`-Operation signalisiert, löscht der Ole-Server allerdings die übermittelten Objekte nach der Operation auf seiner Seite.) Verlangt der Client das Datenformat `DataFormats.Bitmap`, was er natürlich nur tun sollte, wenn die `GetDataPresent()`-Methode des Ole-Servers für dieses Datenformat `true` zurückgibt, liefert die `GetData()`-Methode des Ole-Servers ein `Bitmap`-Objekt.

Im ersten Fall öffnet `OleInsert()` der Reihe nach alle Dateien und fügt deren Inhalte via `NewButton()` nach automatischer Bestimmung der Transparenzfarbe (linkes oberes Pixel) in die Bildliste ein; im zweiten Fall nur die auf direktem Wege gelieferte Bitmap. Der Rest ist eine Frage der internen Verwaltung: Pro Bild muss eine neue Symbolschaltfläche hinzugefügt und ein Eintrag im Listenfeld generiert werden. Der Wert für die `Text`-Eigenschaft wird entweder aus dem Dateinamen gewonnen oder – für das Listenfeld – künstlich generiert.

Da bei einer Ole-Operation immer auch mal was schief gehen kann, empfiehlt es sich, den für Ausnahmen »empfindlichen« Code in ein `try/catch`-Konstrukt zu packen.

```
private bool OleInsert(IDataObject data)
{
  // FileDrop ist passiert - es sind mehrere Dateien möglich
  if (data.GetDataPresent(DataFormats.FileDrop, true))
    try
    {
      string [] fileNames =
        (string[]) data.GetData(DataFormats.FileDrop, true);
      foreach(string fileName in fileNames)
      {
        string buttonName =
          System.IO.Path.GetFileNameWithoutExtension(fileName);
        this.NewButton(new Bitmap(fileName), buttonName, buttonName);
      }
      return true;
    }
    catch { return false; }

  // Bitmap wurde abgelegt
  if (data.GetDataPresent(DataFormats.Bitmap, true))
    try
    {
      Bitmap b = (Bitmap) data.GetData(DataFormats.Bitmap);
      this.NewButton(b, "", "*Bitmap");
      return true;
    }
    catch { return false; }
  return false;
}

private void NewButton(Bitmap b, string buttonText, string listBoxText)
{
  Color c = b.GetPixel(0,0);
  int imageIndex = imageList1.Images.Add(b,c); // Bildliste erweitern
  // neue Symbolschaltfläche
  int buttonIndex = toolBar1.Buttons.Add(new ToolBarButton(buttonText));
  toolBar1.Buttons[buttonIndex].ImageIndex = imageIndex;
  // neuer Eintrag im Listenfeld
  checkedListBox1.Items.Add(listBoxText);
}
```

16 Steuerelemente selbst implementieren

Obwohl die Steuerelemente aus der TOOLBOX für die weitaus meisten Problemstellungen zu brauchbaren Lösungen führen dürften, wird in verschiedenen Situationen doch schnell der Wunsch laut, dem Korsett ihres standardisierten Funktionsumfangs zu entrinnen. Vielleicht wollen Sie das eine oder andere Steuerelement nur ein wenig spezialisieren, damit es besser in die Konzeption der Benutzeroberfläche für Ihre Anwendung passt, vielleicht aber auch den Funktionsumfang in der einen oder anderen Hinsicht erweitern, um ein komplexeres Ein-/Ausgabeverhalten zu erzielen. Dieser Aufgabenbereich lässt sich mit *Abänderung der Grundfunktionalität eines Steuerelements* umschreiben und lebt davon, dass die hinter einem Steuerelement stehende Klasse als Basisklasse für eigene Ableitungen verwendbar ist. Das in diesem Kapitel vorgestellte Codebeispiel, die Klasse `MemTextBox`, erweitert das `TextBox`-Steuerelement zum »Textfeld mit Gedächtnis«.

Fundamentaler ist es hingegen, *ein völlig neues Steuerelement* von Grund auf neu zusammenzuzimmern, das sein eigenes Design und seine eigene Benutzerschnittstelle mitbringt und von der Basisklasse `UserControl` oder gar `Control` aus startet. Das Codebeispiel dafür ist die Klasse `Knob`. Sie implementiert einen Drehknopf als Gegenstück zum `TrackBar`-Steuerelement, das seinen Wert als Winkelstellung repräsentiert und mit der Maus gedreht werden kann.

Der dritte mögliche Ansatz ist die *Gruppierung bestehender Steuerelemente* zu einem übergeordneten Funktionsganzen, das nach außen hin als Steuerelement in Erscheinung tritt. Das Codebeispiel dafür ist gleichfalls die Klasse `Knob`, die ein Beschriftungsfeld einbettet. (Sie können sich aber auch das gut kommentierte Codebeispiel `RadioGroup` ansehen, das im Text nicht näher erläutert wird. Es ist ein Schalteraggregat aus einer freidefinierbaren Anzahl von `RadioButton`-Steuerelementen in zweidimensionaler Anordnung.)

16.1 Wohin mit dem Quelltext für die Steuerelementklasse?

Kleinere Ableitungen, die nur für ein bestimmtes Formular gedacht sind, lassen sich ohne weiteres als untergeordnete Klasse in der jeweiligen Formularklasse definieren. Eine Entwurfsansicht dafür erhalten Sie nicht.

```
// Datei: Form1.cs
namespace MyProject2
{
  class Form1 : Form
  {
    ...
    private class MyButton : Button
    { ... }
} }
```

Bei Ableitungen, die in mehreren Formularen des Projekts eine Rolle spielen, lohnt es sich, die neue Klasse im jeweiligen Namensraum bekannt machen. Wenn die Klassendefinition nicht zu klobig ist, können Sie diese in die *.cs*-Datei einer Formularklasse setzen. Allerdings müssen Sie dann damit leben, dass der Designer immer nur für die erste Klassendefinition in der Datei eine Entwurfsansicht zeigt.

```
// Datei: Form1.cs
namespace MyProject2
{
  class Form1 : Form
  { ... }
  class UserControl1: UserControl
  { ... }
}
```

16.1.1 Als neues Element zum bestehenden Projekt hinzufügen

Umfangreichere Klassendefinitionen – und das ist der Regelfall – packen Sie besser eine eigene Datei. Die HINZUFÜGEN-Befehle im Menü PROJEKT bringen das Dialogfeld NEUES ELEMENT HINZUFÜGEN auf den Bildschirm, in dem Sie die Vorlage für die Art des gewünschten Codegerüsts auswählen und einen Dateinamen angeben, der später dann auch als Klassenname im neuen Codegerüst in Erscheinung tritt. Der Assistent verwendet dafür übrigens den ersten Namensraum des zuerst eingefügten Elements.

```
// Datei: MyControl.cs
namespace MyProject2
{
  class MyControl : UserControl
  { ... }
}
```



Da die Synchronisation zwischen dem TOOLBOX-Fenster und der Codeansicht zumindest in der den Verfassern vorliegenden Version 7.009514 nicht richtig klappt, sollten Sie für ein Steuerelement, das aus der TOOLBOX heraus mit der Maus platzierbar sein soll, auf jeden Fall die Vorlage BENUTZERSTEUERELEMENT und nicht BENUTZERDEFINIERTES STEUERELEMENT benutzen. Erstere erzeugt ein Codegerüst auf der Basis von `UserControl` *und ergänzt in der TOOLBOX ein Symbol dafür. Letztere setzt auf die Basisklasse* `Control`

auf und generiert keinen Eintrag in der TOOLBOX. Da Sie die Basisklasse und das Codegerüst im Editor noch beliebig ändern und ausdünnen können, sind hier keine Nachteile zu befürchten.

Allerdings: Die fehlende Synchronisation hat leider auch zur Folge, dass die TOOLBOX spätere Umbenennungen der Klasse nicht mitbekommt, wenn eine andere Basisklasse als `UserControl` *angegeben ist und partout auf eine Komponente mit dem ursprünglich im HINZUFÜGEN-Dialog vereinbarten Namen besteht. Als Workaround für spätere Umbenennungen anderer Steuerelementklassen ändern Sie die Basisklasse vorübergehend in* `UserControl` *und schalten dann kurz in die Entwurfsansicht um – ein Kompilieren ist dafür nicht notwendig.*

16.1.2 Steuerelementbibliothek

Die letzte Stufe der Herrlichkeit ist schließlich die Steuerelementbibliothek, in der Sie Ihre eigenen Steuerelement-Varianten als eigenständige Projekte archivieren, um sie später in anderen Projekten zur Verfügung zu haben. Dazu fügen Sie entweder der geöffneten Projektmappe über den Menübefehl DATEI/PROJEKT HINZUFÜGEN NEUES PROJEKT ein weiteres Projekt des Typs WINDOWS-STEUERELEMENTBIBLIOTHEK hinzu oder beginnen eine neue Projektmappe mit einem Projekt dieses Typs. In dieses Projekt fügen Sie dann – wie eben beschrieben – je Steuerelement ein neues Element ein.

Zudem müssen Sie später in jedem Projekt, das mit der Steuerelementbibliothek arbeitet, einen Verweis für die Steuerelementbibliothek ergänzen – auch wenn die Projekte in derselben Projektmappe zusammengefasst sind. Geben Sie dazu im PROJEKTMAPPEN-EXPLORER den Kontextmenübefehl VERWEIS HINZUFÜGEN und suchen Sie gegebenenfalls das Projekt auf der Registerkarte PROJEKTE (vgl. Abbildung 16.1).

16.2 Visuelles Steuerelementdesign unter .NET

Bevor es an die Implementierungsmuster für die eingangs unterschiedenen Aufgabenbereiche geht, wäre noch zu klären, auf welcher Grundlage das visuelle Steuerelementdesign unter .NET erfolgt.

Wenn Sie schon einmal Steuerelemente mit einer anderen Programmiersprache wie Visual Basic oder Delphi implementiert haben, werden Sie wahrscheinlich eine bestimmte Vorstellung von dem Mechanismus mitbringen, nach dem visuelles Steuerelementdesign generell funktioniert – auch wenn sich die Akzente mit .NET doch zum Teil verschoben haben. Wenn nicht, finden Sie in diesem Abschnitt auf jeden Fall einen kurzen Überblick.

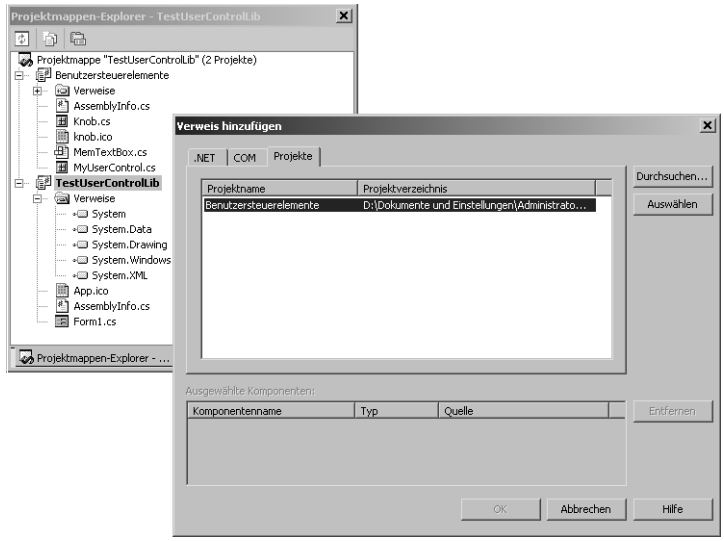


Abbildung 16.1: Verweis auf ein anderes Projekt (hier: Steuerelementbibliothek hinzufügen)

16.2.1 Codeansicht, Entwurfsansicht, Laufzeitansicht

Vom Prinzip her läuft das Steuerelementdesign nach dem gleichen Prinzip wie das Formulardesign ab. Vom gewöhnlichen Formulardesign mit fertigen Steuerelementen aus der TOOLBOX her sind Sie es gewohnt, dass Ihre Formularklasse eine *Codeansicht*, eine *Entwurfsansicht* und eine *Laufzeitansicht* hat. Wahrscheinlich haben Sie aber wenig Gedanken darauf verschwendet, wie die Entwurfsansicht zustande kommt, sondern sich einfach darüber gefreut, dass im visuellen Design Vieles so einfach ist – selbst der Lernvorgang, wie man Steuerelemente notfalls auch ohne visuelles Design handhabt; Der Code des Designers ist tatsächlich sehr einfach zu durchschauen.

Sicher wird Ihnen klar sein, dass die Laufzeitansicht eines Formulars nichts anderes als eine konkrete Instanz Ihrer Formularklasse ist, die durch die Methode `Main()` erzeugt und angezeigt wird. Was Sie in der Entwurfsansicht sehen, sind gleichfalls Instanzen: eine als Container fungierende Instanz der Basisklasse des Klassenentwurfs und Instanzen der jeweiligen Steuerelemente, die Sie darauf platziert haben. Beim Entwurf eines Formulars dient eine `Form`-Instanz, beim Entwurf eines *Benutzersteuerelements* eine `UserControl`-Instanz und beim Entwurf eines *benutzerdefinierten Steuerelements* eine `Control`-Instanz als Container.

Für diese drei Containerarten (nimmt man die für das Web-orientierte Design hinzu, sind es noch ein paar mehr) stellt VS.NET jeweils einen Designer bereit – eine Art Interpreter, der es ermöglicht, den Klassenentwurf und das Container-Layout weitgehend auf visuellem Wege zu erledigen, weil er den visuellen Entwurf auch eins zu eins in ein Codegerüst umsetzt und umgekehrt.

Ein Steuerelement spielt sich selbst

Wann immer Sie in der Entwurfsansicht ein Steuerelement aus der TOOLBOX auf dem Container platzieren, generiert der Designer eine Laufzeitinstanz der jeweiligen Steuerelementklasse, fügt sie der `Controls`-Auflistung des Containers hinzu, listet deren Eigenschaften und Ereignisse im EIGENSCHAFTEN-Fenster auf und übernimmt die Kontrolle über alle Maus- und Tastaturereignisse. Die verschiedenen Funktionen des Designers (Verschieben, Ändern der Bereichsgröße, Löschen, Kopieren, Einfügen, Festlegen der Tabulatorreihenfolge usw.) sind also überschriebenes Verhalten.

Außerdem koppelt der Designer das EIGENSCHAFTEN-Fenster direkt mit der Laufzeitinstanz, so dass Wertänderungen unmittelbar auf die Instanz durchschlagen und deren Zustand ändern, sofern die Instanz-eigene Gültigkeitsprüfung es zulässt. Ausnahmen fängt der Designer ab und zeigt die Beschreibung über ein Meldungsfenster an. Zeichnen tut sich das Steuerelement unter Laufzeitbedingungen und Berücksichtigung des aktuellen Zustands natürlich selbst – der Designer ergänzt nur den zusätzlichen Zierrat (beispielsweise Ziehrahmen und Einblendungen für die Tabulatorreihenfolge).

Der Trick mit den Ebenen

Wenn Sie eine Formularklasse implementieren, haben Sie *ein* großes Ziel: Das Formular soll sich zur Laufzeit spezifikationsgemäß verhalten – fertig. Wenn Sie ein Benutzersteuerelement implementieren, haben Sie plötzlich zwei Fronten. Das Benutzersteuerelement soll sich zur Laufzeit *und* zur Designzeit spezifikationsgemäß verhalten. Obwohl Sie Ihr primäres Augenmerk auf die Implementierung des *Laufzeitverhaltens* richten werden, spielt das *Entwurfszeitverhalten* durchaus eine wichtige Rolle. Schließlich sollte sich das Steuerelement den allgemeinen Regeln des visuellen Entwurfs beugen, die Sie vom Formulardesign mit Steuerelementen her gewohnt sind. Groß ist der Unterschied nicht, Sie sollten ihn aber kennen.

Das normale Szenario für die Entwicklung eines Steuerelements sieht so aus, dass Sie parallel dazu auf einer anderen Ebene ein Testformular implementieren, auf dem Sie eine oder mehrere Instanzen des Steuerelements platzieren, die den jeweils aktuellen Entwicklungsstand widerspiegeln. In der Regel wechseln Sie dann ständig zwischen fünf Ansichten hin und her:

- *Entwurfsansicht des Steuerelements* – falls das Steuerelement auf `UserControl` oder `Control` basiert, können Sie hier in den Fensterbereich des Containers beliebige Steuerelemente aus der Toolbox (nicht jedoch das Steuerelement selbst) platzieren.
- *Codeansicht des Steuerelements* – in diesem Fenster implementieren Sie die Ausstattung (Eigenschaften, Methoden und Ereignisse) und das Verhalten (Behandlung von Ereignissen) des Steuerelements. In der

Methode `InitializeComponent()` finden Sie den Initialisierungscode für das Containerobjekt (Basisklasseninstanz) und für die in den Entwurf eingefügten Steuerelemente.

- *Entwurfsansicht des Testformulars* – hier können Sie aus der TOOLBOX heraus eine Instanz des Steuerelements in dem Formularentwurf platzieren. Diese Instanz spiegelt immer das Entwurfszeitverhalten des Steuerelements zum Zeitpunkt des letzten erfolgreichen Erstellvorgangs wieder. Sie müssen daher immer erst den Compiler anwerfen, wenn Sie sehen wollen, wie sich eine aktuelle Änderung im Code des Steuerelements im Formularentwurf auswirkt. Die Steuerelementinstanz kann über ihre Eigenschaft `DesignMode` in Erfahrung bringen, ob sie als Entwurfs- oder als Laufzeitinstanz fungiert. Eine diesbezügliche Unterscheidung ist in der Regel aber nicht nötig.
- *Codeansicht des Testformulars* – hier bearbeiten Sie den Testcode für das Steuerelement. In der Methode `InitializeComponent()` finden Sie unter anderem den Initialisierungscode für die Eigenschaften, die Sie interaktiv in der Entwurfsansicht setzen können. (Es erscheinen nur die Eigenschaften, deren Wert vom Standardwert unmittelbar nach Konstruktion der Instanz abweicht.)
- *Laufzeitansicht des Testformulars* – diese Ansicht erhalten Sie nur, wenn Sie das Testformular starten. Sie spiegelt das Laufzeitverhalten des Steuerelements wider. Die jeweilige Instanz kann über ihre Eigenschaft `DesignMode` in Erfahrung bringen, ob sie als Entwurfs- oder als Laufzeitinstanz fungiert.

Wenn Sie es kompliziert lieben, können Sie sich eine weitere Ebene einfangen, indem Sie ein gerade in Entwicklung befindliches Steuerelement auf Ihrem Steuerelement platzieren, und dieses wie gehabt in einem Testformular oder einfach nur im anderen Steuerelement testen. Abbildung 16.2 verbildlicht die Zusammenhänge.

16.3 Implementierung eines eigenen Benutzersteuerelements

Die Entwurfszeitunterstützung für die Modifikation bestehender Steuerelemente aus der TOOLBOX lässt – wie im nächsten Abschnitt beschrieben – leider noch etwas zu wünschen übrig. Der gerade Weg für die Implementierung eigener Steuerelemente führt über die Basisklasse `UserControl`. Steuerelemente, die von `UserControl` ableitet sind, heißen in der offiziellen Dokumentation (und auch in VS) »Benutzersteuerelemente«, alle anderen selbstgeschneiderten Steuerelemente zählen zu den »benutzerdefinierten Steuerelementen« – welch subtiler Unterschied.

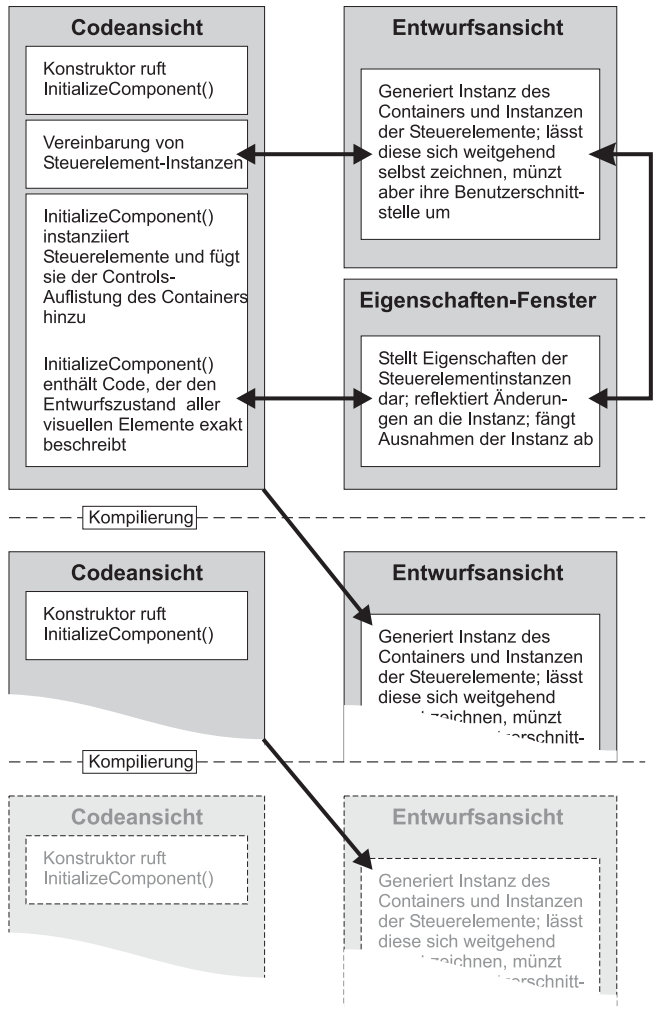


Abbildung 16.2: Der visuelle Entwurf benutzt Instanzen bereits kompilierter Steuerelement- und Containerklassen.

Das Codegerüst für ein Benutzersteuerelement erhalten Sie, indem Sie dem jeweiligen Projekt über den Befehl PROJEKT/BENUTZERSTEUERELEMENT HINZUFÜGEN eine Vorlage des Typs *Benutzersteuerelement* mit einem passend gewählten Namen hinzufügen. Wenn der Assistent seine Aufgabe erfüllt hat, finden Sie sich im Designer wieder und sehen eine Entwurfsfläche (eine Instanz der Klasse `UserControl`), die als Container für Steuerelemente aus der TOOLBOX fungieren kann. Das Benutzersteuerelementdesign folgt damit den gleichen Regeln wie das Formulardesign: Sie fügen in der Entwurfsansicht Steuerelemente ein, die Sie in der Codeansicht mit dem bestehenden Codegerüst »verdrahten«. Das Codegerüst für ein Benutzersteuerelement sieht nahezu genauso aus wie das eines Formulars, weshalb man sich doch recht schnell zuhause fühlt. Und auch die Aktivität des Designers läuft nach exakt demselben Strickmuster ab.

16.3.1 Zum Eingewöhnen

Zur Prüfung, wie sich das Benutzersteuerelement beim Formularentwurf sowie zur Laufzeit verhält, benötigen Sie noch ein startfähiges Formular. Sobald Sie die erste Testversion Ihres Benutzersteuerelements kompiliert haben, platzieren Sie Instanzen davon auf dem Formularentwurf. Diesen Instanzen liegt bereits der lauffähige Code zugrunde. Sie können sich ja mal den Spaß machen und ein Meldungsfenster im Konstruktor des Benutzersteuerelements anzeigen:

```
class MyUserControl : UserControl
{
  public MyUserControl()
  {
    MessageBox.Show ("Neue Instanz generiert");
    ...
  }
  ...
}
```

Der Dialog erscheint, wann immer der Designer eine Instanz Ihres Benutzersteuerelements anlegt. Im Einzelnen ist das, wenn Sie:

- eine neue Instanz des Benutzersteuerelements über die Toolbox auf dem Formularentwurf platzieren
- nach einem Erstellungsvorgang in die Entwurfsansicht des Formulars umschalten – und zwar je Instanz eine Meldung
- das Formular starten – gleichfalls je Instanz eine Meldung.

Wenn Sie den Aufruf hingegen in die `Paint`-Behandlung setzen (was in der Praxis nicht zu empfehlen ist), werden Sie sehen, dass die Instanz tatsächlich aktiv ist. In der Tat zeichnet sie sich nicht nur selbst, sie nimmt auch die Werte von Eigenschaften entgegen usw. Einzig die Maus- und Tastaturinteraktionen werden vom Designer überschrieben.

Fehlersuche

Wenn Sie es gewöhnt sind, Fehlern mit dem Debugger hinterher zu schnüffeln, werden Sie sich bei der Implementierung von Steuerelementen auf eine neue Situation einstellen müssen: Vom Designer angelegte Instanzen unterliegen generell nicht der Obhut des Debuggers. Probieren Sie es aus! Setzen Sie auf den `MessageBox.Show()`-Aufruf einen Haltepunkt und platzieren Sie eine weitere Instanz des Benutzersteuerelements auf dem Testformular: Der Debugger schweigt und kommt seiner Aufgabe erst nach, wenn Sie das Formular starten. Wenn Sie also einen Fehler nicht im »Blindflug« aufspüren können oder wollen und den Debugger dazu brauchen, starten Sie einfach das Testformular. Zur Laufzeit stehen Ihnen wieder alle Funktionen des Debuggers zur Verfügung.

Was tun, wenn die Entwurfsansicht blockiert ist?



Dummerweise kommt es beim Steuerelemententwurf öfter mal vor, dass der Code des gerade neu kompilierten Benutzersteuerelements eine unbehandelte Ausnahme generiert. Wenn Sie Glück haben und der Fehler in der `Paint`*-Behandlung passiert, können Sie die Fehlermeldung im Fensterbereich des Steuerelements lesen. In allen anderen Fällen schaltet der Designer die Entwurfsansicht mit einer nicht immer gerade aussagekräftigen Fehlermeldung ab und lässt sich im Allgemeinen nur durch Schließen und erneutes Öffnen der Entwurfsansicht des Formulars zu einer neuen visuellen Darstellung überreden.*

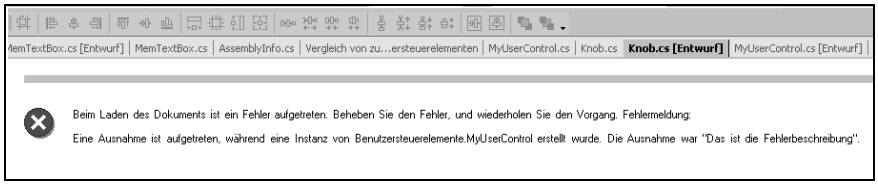


Abbildung 16.3: Auf dem Container `Knob` wurde eine Instanz von `MyUserControl` platziert, die Startschwierigkeiten hat.

Was tun, wenn der Designer spinnt?



Für eine 1.0-Version ist .NET ein wahrlich ausgereiftes Produkt. Dennoch kann es schon einmal vorkommen, dass der Designer bei einer Steuerelementklasse aus dem Tritt gerät und die Synchronisation zwischen Entwurf und Code nicht mehr gegeben ist. Sie merken das daran, dass die Entwurfsansicht nicht mehr das anzeigt, was in der Methode `InitializeComponent()` *steht. Das Problem liegt meist (aber nicht immer) an der vom Designer für die Klasse gepflegten Ressource-Datei. Wenn Reparaturversuche und ein Neustart des Systems nicht mehr helfen, bleibt Ihnen als Notnagel immer noch die folgende Prozedur:*

1. *Legen Sie ein neues Projekt (samt Projektmappe) an und löschen Sie das .cs-Element im* PROJEKTMAPPEN-EXPLORER*.*
2. *Übernehmen Sie über den Befehl* PROJEKT/VORHANDENES ELEMENT HINZUFÜGEN *alle Klassen, die intakt waren, in das Projekt.*
3. *Legen Sie für die betroffene Klasse ein neues Element gleichen Typs an und benennen Sie es entsprechend. Öffnen Sie die .cs-Datei mit der »defekten« Entwurfsansicht aus der alten Projektmappe und kopieren Sie den Code über die Zwischenablage in die neue .cs-Datei.*
4. *Kompilieren Sie das Projekt. Die Entwurfsansicht müsste nun wieder synchron sein. (Das Problem liegt meist an der vom Designer gepflegten Ressource-Datei der Klasse)*

Gewöhnen Sie sich außerdem an, in regelmäßigen Abständen, beispielsweise immer, wenn das Projekt das nächste anstehende Entwicklungsziel erreicht hat, ein Backup des Projektordners zu machen. Dazu schließen Sie

die Projektmappe und fertigen eine Kopie per Windows-Explorer (beispielsweise im Dialog PROJEKT ÖFFNEN) an. Geben Sie der Kopie einen aussagekräftigen Namen. Danach können Sie jederzeit auf das eingefrorene Projekt zurückgreifen.

16.3.2 Codebeispiel – Knob

Das in diesem Abschnitt vorgestellte Codebeispiel zeigt den Steuerelemententwurf von »der Pike auf«. Es geht um ein Drehknopf-Steuerelement, das einen Wert aus einem frei definierbaren Wertebereich als Winkelstellung eines Drehknopfs darstellt und sich mit der Maus oder der Tastatur bedienen lässt – kurzum das Gegenstück zum `TrackBar`-Steuerelement in rund. Es verfügt über eine Beschriftung, zeigt eine Skala mit einer frei definierbaren Anzahl von Skalenstrichen an, und lässt sich im Gegensatz zum `TrackBar`-Steuerelement wahlweise in beide Richtungen (gegen oder mit dem Uhrzeigersinn) orientieren, bei freidefinierbaren Endstellungen. Als Beschriftung dient ein eingebettetes `Label`-Steuerelement, dessen Eigenschaften das Steuerelement offenlegt und synchronisiert.



Abbildung 16.4: Das `Knob`-Steuerelement im Einsatz

Spezifikation

Die Spezifikation für das Steuerelement lautet:

1. Das Steuerelement soll einen Wert `Value` aus einem Wertebereich analog als Drehknopf mit entsprechender Winkelstellung zwischen zwei Endstellungen repräsentieren.
2. Der Wertebereich soll ein frei wählbares Intervall [`Minimum`, `Maximum`] über dem Datentyp `double` sein.

3. Die Endstellungen sind als frei wählbare Winkel `MinAngle` und `MaxAngle` im Bogenmaß aus dem Intervall [-2π, 2π] anzugeben. (Der Winkel 0 entspricht 3 Uhr; der Winkel wächst gegen den Uhrzeigersinn.). Ist `MinAngle` größer als `MaxAngle`, erhöht das Steuerelement seinen Wert im, sonst gegen den Uhrzeigersinn.
4. Das Steuerelement soll Wertänderungen signalisieren.
5. Das Steuerelement soll `Ticks` Skalenstriche in frei wählbarer Anzahl zwischen den Endstellungen anzeigen.
6. Das Steuerelement soll eine Beschriftung `Text` mit frei wählbarer Schriftart im unteren Bereich anzeigen.
7. Das Steuerelement soll den Fokus anzeigen.
8. Das Steuerelement soll die Tasten [→], [←], [Bild ↑] und [Bild ↓] wie das `TrackBar`-Steuerelement interpretieren und seinen Wert jeweils um `SmallChange` bzw. `LargeChange` verändern.

Grundsätzliches zum Quelltext

Die Klasse `Knob` ist aus einem Codegerüst für ein neues Benutzersteuerelement heraus gewachsen, ohne dass die ursprüngliche Basisklasse `UserControl` geändert wurde. Allerdings lässt sich der Code auch problemlos kompilieren, wenn Sie `Control` als Basisklasse einsetzen. (Die Implementierung ist dann sogar etwas geradliniger, wie sich gleich noch herausstellen wird.)

Geheimnisvolle Attribute

Der Quelltext stellt verschiedene Attribute vor, ohne diese jedoch in einen größeren Zusammenhang zu stellen. Attribute bilden eine Metaebene, die die kontextspezifische Interpretation des Quellcodes regeln – in diesem Fall für den Kontext des Designers. Nachdem Attribute gleichfalls in C# implementiert sind und entsprechende Klassen dafür in der .NET-Klassenbibliothek bereitstehen, besteht grundsätzlich die Möglichkeit, mit den üblichen Mitteln der Vererbung und Überschreibung in die jeweiligen Interpretationsmechanismen einzugreifen. Dass dies eine Wissenschaft für sich ist, versteht sich. Die theoretische Basis für Attribute und die Möglichkeiten, die Attribute bieten, zählen zu den stark fortgeschrittenen Themen und werden von diesem Buch nicht abgedeckt. Die vorgestellten Attribute lassen sich jedoch auch ohne theoretisches Fundament in ihrer Bedeutung verstehen und vorteilhaft einsetzen.

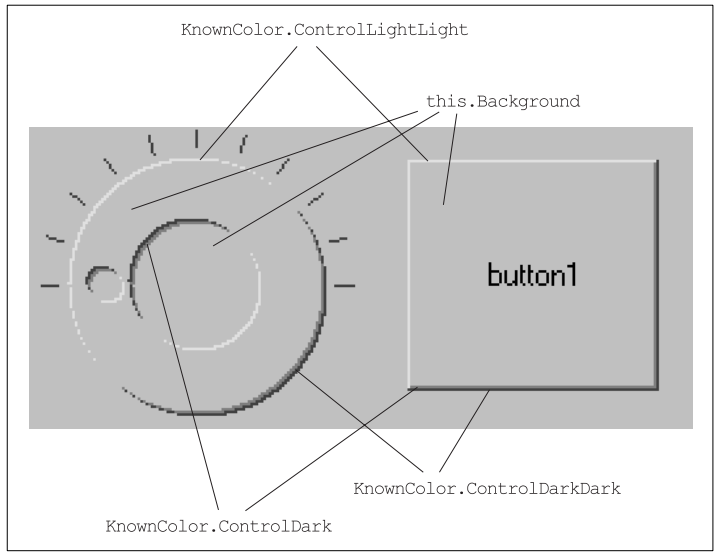
Überlegungen zum grafischen Layout

Ein Benutzersteuerelement, das sich selbst zeichnet, sollte sich möglichst unauffällig in das allgemeine Steuerelementlayout einpassen. Hierfür gibt es verschiedene Punkte, die Sie beachten sollten:

- Adaption an den allgemeinen Darstellungsstil (vgl. die `Button.FlatStyle`-Eigenschaft)
- Gebrauch der spezifischen Steuerelementfarben
- Visuelle Darstellung von Zuständen wie »markiert«, »Fokusbesitz« etc.

Abbildung 16.5 zeigt die Darstellung des `Knob`-Steuerelements neben einer Schaltfläche in Vergrößerung. Dabei sieht man recht gut, mit welch einfachen Mitteln der 3D-Effekt erreicht wird und welche Steuerelementfarben wo gezeichnet werden.

Abbildung 16.5: Grafisches Layout und Farbzuordnung des Benutzersteuerelements





Wenn Sie zum Zeichnen eines Steuerelements Darstellungselemente der üblichen Windows-Steuerelemente verwenden wollen, sollten Sie sich die Klasse `ControlPaint` *näher ansehen, die verschiedene statische Methoden für häufig anfallende Aufgaben bereitstellt.*

Implementierung

Die Implementierung gliedert sich in fünf Aufgabenbereiche:

1. Definition der Klassenelemente und deren Initialisierung im Konstruktor
2. Pflege der Eigenschaften
3. Zeichnen des Steuerelements
4. Mausschnittstelle
5. Tastaturschnittstelle

Die folgenden Abschnitte besprechen diese Aufgabenbereiche im Einzelnen.

Datenfelder und Konstruktor

Die Spezifikation des Steuerelements fordert, dass die Klasse `Knob` eine Reihe von Eigenschaften und Ereignissen implementiert. Hier die Vereinbarung der dahinter stehenden Datenfelder:

```
[DefaultEvent("ValueChanged")]  // Belegt Doppelklick im Designer
public class Knob : System.Windows.Forms.UserControl
{
  #region Datenfelder
  private double minimum, maximum,  // für Eigenschaften
    smallChange, largeChange,
    minAngle, maxAngle,
    ticks, val;
  private System.Windows.Forms.Label Caption;
  private float radius;             // Radius des Knopfes

  [Description("Tritt auf, wenn sich der Wert der Eigenschaft " +
   "Value ändert.")]
  public event System.EventHandler ValueChanged;
  [Description("Tritt auf, wenn der Benutzer die Drehknopfposition " +
  "verändert")]
  public event System.EventHandler Scroll;
  #endregion
...
}
```

Das vor der Klassenvereinbarung stehende `DefaultEvent`-Attribut teilt dem Designer mit, welches Ereignis er mit dem Doppelklick assoziieren soll. Für die Klasse `Knob` ist das `ValueChanged`. Während `ValueChanged` bei jeder Änderung der Eigenschaft `Value` signalisiert wird, kommt das andere Ereignis `Scroll` nur dann zur Ausführung, wenn die Änderungen über die Benutzerschnittstelle veranlasst wurden.

Das `Description`-Attribut ermöglicht es, eine Kurzbeschreibung für das jeweilige Element im unteren Bereich des EIGENSCHAFTEN-Fensters bereitzustellen. (Schöner wäre es natürlich, wenn der Designer auch Dokumentationskommentare verarbeiten würde. So müssen Sie doppelt kommentieren.) Abbildung 16.6 zeigt den Effekt.

Der Konstruktor versorgt die Datenfelder mit Werten und berechnet das richtige Höhe-zu-Breite-Verhältnis. Das Steuerelement muss um die Höhe des `Label`-Steuerelements höher als breit sein. Die Höhe des Label-Steuerelements ergibt sich wiederum aus der Schriftgröße über die `GetHeight()`-Methode.

Außerdem setzt der Konstruktor via `SetStyle()` verschiedene Stilattribute, damit das Steuerelement in den Genuss einer Doppelpufferung für eine flackerfreie Ausgabe kommt – es sind wirklich alle drei Attribute erforderlich,

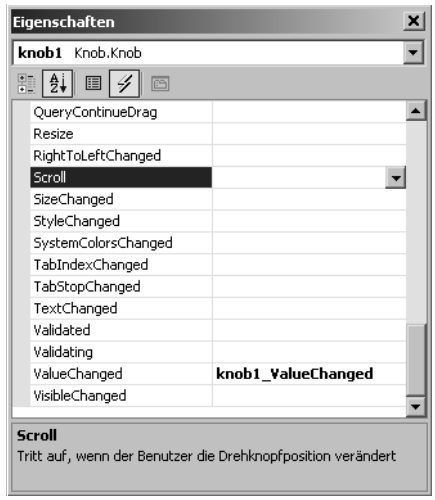


Abbildung 16.6: Die Kurzbeschreibung erscheint im unteren Bereich des Eigenschaften-Fensters.

`ControlStyles.DoubleBuffer` allein reicht nicht. Der Unterschied ist merklich – gerade bei der vielen Zeichenarbeit, die das Steuerelement erledigt. (Hinweis: Von einer Doppelpufferung können nur Steuerelemente profitieren, die sich komplett selbst zeichnen, und nicht auf eine Basisklasse angewiesen sind.)

```
#region Initialisierung
public Knob()
{
  // Doppelpufferung für Steuerelement aktivieren
  this.SetStyle(ControlStyles.UserPaint, true);
  this.SetStyle(ControlStyles.AllPaintingInWmPaint, true);
  this.SetStyle(ControlStyles.DoubleBuffer, true);

  maximum = 10;
  minAngle = Math.PI;
  ticks = 10;
  smallChange = 1;
  largeChange = 1;
  InitializeComponent();
  Caption.Height = (int) base.Font.GetHeight(); // Höhe des Textfelds
                                                // anpassen
  Height = Width + Caption.Height;    // Abmmessungen setzen
  radius = Width * 0.4f;
}
```

Eigenschaften

Der `set`-Accessor für die `Value`-Eigenschaft sorgt dafür, dass der Wert des Datenfelds `val` im zulässigen Bereich bleibt. Statt Überschreitungen durch eine Ausnahme zu ahnden, schneidet er den Wert schlicht zurecht. Außerdem löst er das `ValueChanged`-Ereignis aus, wenn sich der Wert ändert, und veranlasst per `Invalidate()` die Darstellung der neuen Zeigerposition.

```
[Description("Wert des Steuerelements im Bogenmaß")]
public double Value
{
  get { return val; }
  set
  {
    if (value != val)
    {
      if (value > maximum)
        val = maximum;
      else if (value < minimum)
        val = minimum;
      else
        if (smallChange != 0)
        val = value;
      OnValueChanged(new EventArgs());
      Invalidate();
    }
  }
}
```

Die Eigenschaft `CaptionField` liefert einen Verweis auf das dem Steuerelement untergeordnete `Label`-Steuerelement zurück. Der Besitzer eines `Knob`-Objekts kann so auf alle Eigenschaften des Textfelds zugreifen und dieses auch nach Herzenslust manipulieren. Es mag Ihnen vielleicht als selbstverständlich erscheinen, wenn die dem Textfeld interaktiv im EIGENSCHAFTEN-Fenster zugewiesenen Eigenschaften ihren Weg in das jeweilige Codegerüst finden – das ist standardmäßig aber nicht so. Der Designer nimmt von sich aus erst einmal keine Serialisierung der `Label`-Eigenschaft vor, sondern benötigt eine Extraeinladung in Form des `DesignerSerializationVisibility.Content`-Attributs. Nur wenn dieses Attribut gesetzt ist, wertet der Designer den Zustand des Beschriftungsfelds aus und serialisiert alle vom jeweiligen Standardwert abweichenden Eigenschaften in die `InitializeComponent()`-Methode des jeweiligen Containers.

```
[Description("Ermöglicht Zugriff auf Beschriftungssteuerelement")]
// Dieses Attribut ist notwendig, damit der Designer den Wert
// in InitializeComponent einträgt
[DesignerSerializationVisibility(DesignerSerializationVisibility.Content)]
public Label CaptionField
{
  get { return Caption;}
}
```



Was die »Extraeinladung« betrifft, so sei angemerkt, dass es alles andere als eine Selbstverständlichkeit ist, wenn der Designer auch Eigenschaften mit komplexen Datentypen wie Steuerelementklassen serialisiert (das heißt: veränderte Eigenschaften der jeweiligen Instanzen in die `InitializeComponent()`*-Methode des Container-Objekts zurückschreibt). Er kann dies, weil er auf der Metaebene der Attributklassen die notwendige Infrastruktur dafür vorfindet. Sie können davon ausgehen, dass der Designer alle grundlegenden*

Steuerelemente sowie alle Datentypen, die Ihnen als Eigenschaften von Steuerelementen begegnen, serialisieren kann – ein gutes Beispiel ist der Datentyp `Font`.

Trägt eine Eigenschaft hingegen einen eigenen Datentyp, lässt Sie der Designer kalt im Regen stehen. In diesem Fall müssen Sie gehörigen Aufwand treiben und verschiedene eigene Attributklassen implementieren, angefangen vom Ein-/Ausgabe-Editor für den Wert (für die `Font`*-Klasse wird hier ja der Standarddialog* `FontDialog` *aufgerufen) bis hin zur konkreten Serialisierung, damit der Designer das visuelle Design für Ihren Datentyp genauso wie das der standardmäßigen Datentypen handhabt.*



Das Herumspielen mit dem `DesignerSerializationVisibility`*-Attribut ist nicht ganz ungefährlich. Aufgrund eines den Verfassern unbekannten, dafür aber umso ärgerlicheren Fehlers in der Implementierung des Designers (.NET-Framework-Version v1.0.3705) kann es passieren, dass dieser aus dem Tritt kommt und auch die gewöhnliche Serialisierung nicht mehr korrekt vornimmt. Dies äußert sich in »vergessenen« Behandlungsmethoden und Initialisierungen und einem verworrenen Layout des Benutzersteuerelements. Versuche, das Projekt wieder gerade zu biegen, scheitern und es hat sich gezeigt, dass es das beste ist, ein neues Projekt anzulegen oder auf eine Sicherheitskopie zurückzugreifen und den an sich ja intakten Quelltext über die Zwischenablage zu kopieren.*

Was die `Font`-Eigenschaft der Klasse betrifft, so sei zunächst einmal gesagt, dass sie auf der Ebene von `Knob` eigentlich Luxus ist, denn der Zugriff darauf ist jederzeit auch über die `CaptionField`-Eigenschaft möglich. Dennoch findet man Redundanzen dieser Art recht häufig, und sie haben auch ihren tieferen Sinn. `Font` wird von `UserControl` als virtuelle Eigenschaft vererbt und gibt dem Konstruktor der Basisklasse `Control` die Möglichkeit, die aktuelle `Font`-Eigenschaft des jeweiligen Containers als Wert einsetzen. Das wirkt sich beispielsweise aus, wenn Sie eine neue Instanz des Benutzersteuerelements aus der Toolbox in ein Formular ziehen, dessen `Font`-Eigenschaft Sie zuvor verändert haben. Alle Steuerelemente »erben« automatisch diese Schriftart als induzierte Standardvorgabe. Dies hat den Vorteil, dass Sie die Schriftattribute der Steuerelemente nicht einzeln an den jeweiligen Formularstil anpassen müssen und der Designer auch nur Änderungen des induzierten Standardwertes zu serialisieren braucht. Die induzierte Standardvorgabe würde verloren gehen, wenn Sie die `Font`-Eigenschaft wie folgt nur »flach« an das Beschriftungsfeld durchreichen würden:

```
public Font Font   // ohne induzierte Standardvorgabe
{
  get { return Caption.Font; }
  set { Caption.Font = value;}
}
```

Auch die virtuelle Implementierung beschränkt sich im Wesentlichen auf das wechselseitige Hin- und Herreichen des Wertes, wobei dies aber recht

trickreich passiert und eine Ereigniskette auslöst, die bis zur Paint-Behandlung reicht. Erhält das Benutzersteuerelement ein neues Font-Objekt, setzt es zunächst die Font-Eigenschaft des Beschriftungsfelds. Das dadurch ausgelöste FontChanged-Ereignis setzt wiederum die (virtuelle) Font-Eigenschaft der Basisklasse, die letztlich der get-Accessor ebendieser Eigenschaft wieder liefert. Die FontChanged-Behandlung ändert zudem die Größe des Beschriftungsfelds, was zunächst die SizeChanged-, dann die Resize- und schließlich die Paint-Behandlung auf den Plan ruft.

```
[Description("Setzt die Schriftattribute für Beschriftung")]
  // dieses Attribut ist notwendig, damit der Designer den Wert
  // im Eigenschaftsfenster anzeigt
public override Font Font
{
  get { return base.Font; }
  set { Caption.Font = value; }
}
private void Caption_FontChanged(object sender, System.EventArgs e)
{
  base.Font = Caption.Font;
  Caption.Height = (int) base.Font.GetHeight();   // Höhe anpassen
}

private void Caption_SizeChanged(object sender, System.EventArgs e)
{
  Height = Caption.Height + Width;
}

private void Knob_Resize(object sender, System.EventArgs e)
{
  Height = Caption.Height + Width;
  radius = Width * 0.4f;
  Invalidate();
}
```

Die Text-Eigenschaft ist gleichfalls redundant. Als virtuelle Eigenschaft verfügt auch sie über einen Unterbau, an den sich allerdings ein ganzer Sack voller Fragestellungen knüpft – obwohl die Implementierung letztlich dann doch recht trivial aussieht.

Als erstes gibt es zu bemerken, dass die Basisklasse UserControl (aus welchen Gründen auch immer) Anstrengungen unternimmt, die ursprünglich von Control vererbte Text-Eigenschaft zu verdunkeln. Sie erscheint weder im EIGENSCHAFTEN-Fenster, noch wird sie im Editor aufgelistet, was daran liegt, dass in UserControl wohl ein Browsable(false)-Attribut dafür vorhanden ist. Um die Eigenschaft als override-Variante auf Ebene von Knob wieder auszugraben, ist daher ein Browsable(true)-Attribut erforderlich. Zudem wird Text vom Designer standardmäßig auch nicht serialisiert, was wohl daher kommt, dass in UserControl das DesignerSerializationVisibility-Attribut auf Hide gesetzt wurde. Änderungen der Eigenschaft im Eigenschaften-Fenster finden erst dann

ihren Niederschlag in der `Initialize`-Methode, wenn Sie explizit mit dem Wert `Visible` (oder `Content`) für das Attribut dagegenhalten.

Zweitens stellt sich natürlich die Frage, warum die Eigenschaft virtuell ist und welche Vorteile sich dadurch ergeben. Der Grund dafür ist ausschließlich in einem – recht praktischen – Feature des Designers zu suchen, das man normalerweise benutzt, ohne sich viel Gedanken darüber zu machen. Wann immer Sie eine neue Instanz eines Steuerelements in einen Container ziehen, erhält diese Vorgabewerte für die Eigenschaften `Text` und `Name` – der Klassenname klein geschrieben mit Suffix für die Instanzzählung. Instanzen der Steuerelementklasse `Knob` benennt der Designer demnach mit `knob1`, `knob2` usw.

Im vorliegenden Fall sollte der über den Designer induzierte Vorgabewert in der `Text`-Eigenschaft des Beschriftungsfelds landen, was letztlich nur möglich ist, wenn die `Text`-Eigenschaft des Benutzersteuerelements in der vorgestellten Weise »reaktiviert« wird und Wertänderungen weitergibt.

```
 [Description("Beschriftung des Steuerelements")]
  // Dieses Attribut ist notwendig, damit der Designer den Wert
  // im Eigenschaftsfenster anzeigt
[Browsable(true)]
  // Dieses Attribut ist notwendig, damit der Designer den Wert
  // in InitializeComponent einträgt
[DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)]
public override string Text
{
  get { return Caption.Text; }
  set { Caption.Text = value; }
}
```

Über die Eigenschaft `Ticks` gibt es wenig zu sagen. Obwohl sie nur positive Werte akzeptiert, kommt der Datentyp `uint` nicht in Frage, sonst wäre das Steuerelement nicht CLS-konform. Statt bei negativen Werten eine Ausnahme zu generieren, ignoriert die Eigenschaft einfach das Vorzeichen. (Einen negativen `Ticks`-Wert könnte man aber dazu verwenden, einen anderen Skalenstil zu setzen.) Zudem beschränkt die Eigenschaft den Wert nach oben hin auf 100 – noch mehr Striche würden die `Paint`-Behandlung unnötig beschäftigen.

```
[Description("Anzahl der Skalenstriche zwischen MinAngle und MaxAngle")]
public double Ticks
{
  get { return ticks; }
  set
  {
    ticks = Math.Abs(value);
    if (ticks > 100)
       ticks = 100;
    Invalidate();
  }
}
```

Die anderen Eigenschaften `Minimum`, `Maximum`, `MinAngle`, `MaxAngle`, `SmallChange` und `LargeChange` sind mehr oder weniger täglich Brot. Hier exemplarisch der Code von `Minimum` und `MaxAngle`:

```
[Description("Der minimale Value-Wert für die Drehknopfposition")]
public double Minimum
{
  get { return minimum; }
  set
  {
    if (value > maximum)
      throw new System.ArgumentOutOfRangeException (
        "Minimum muss kleiner als Maximum sein");
    minimum = value;
    Refresh();
  }
}

[Description("Winkel im Bogenmaß zwischen -2_ und 2_. Ist MaxAngle kleiner"
   + " als MinAngle, nimmt Value im Uhrzeigersinn zu, sonst ab.")]
public double MaxAngle
{
  get { return maxAngle; }
  set
  {
    const double PI2 = Math.PI*2;
    if (value < -PI2 || value > PI2)
      throw new System.ArgumentOutOfRangeException(
        "Wert zwischen -2_ und 2_ erwartet");
    // zu nah am Maximum?
    if (Math.Abs(value - minAngle) < 1E-100)
      throw new System.ArgumentException("Minimum ist zu nah am Maximum");
    else
      maxAngle = value;
    Invalidate();
  }
}
```

Zeichnen des Steuerelements

Zum Zeichnen des Steuerelements überschreibt die Klasse die virtuelle Methode `OnPaint()` – ohne die Basisklassenvariante aufzurufen. Benötigt wird ein Satz Pinsel für die verschiedenen Steuerelementfarben (vgl. Abbildung 16.5). Die Flächenelemente des Steuerelements werden als leicht gegeneinander verschobene Kreise mit geeigneten Radien gezeichnet. Ist das Steuerelement im Besitz des Fokus, zeichnet es die Markierungsfläche in der Farbe `ControlDark`, ansonsten in der Hintergrundfarbe.

Für die Berechnung der Skalenstriche und der Markierung ist ein wenig einfache Trigonometrie erforderlich. Der Winkel für die Markierung ergibt sich aber schlicht aus dem Dreisatz. Das Intervall von `Minimum` bis `Maximum` wird auf den Winkelbereich `MinAngle` bis `MaxAngle` abgebildet. Damit steht der

gesuchte Winkel `angle` im gleichen Verhältnis zu den Grenzen des Winkelintervalls wie der Wert `val` zu den Grenzen des Wertintervalls. Sie finden die Berechnung in der Hilfsmethode `Markrect()`.

```
protected override void OnPaint(PaintEventArgs e)
{
  const int offs3D = 1;
  e.Graphics.TranslateTransform(Width/2-1, Width/2-1);
  Brush b1 = new SolidBrush(
    Color.FromKnownColor(KnownColor.ControlLightLight));
  Brush b2 = new SolidBrush(this.BackColor);
  // äußerer Ring abgehoben
  Brush b3 = new SolidBrush(
    Color.FromKnownColor(KnownColor.ControlDark));
  Brush b4 = new SolidBrush(
    Color.FromKnownColor(KnownColor.ControlDarkDark));
  e.Graphics.FillEllipse(
    b1, -radius-offs3D, -radius-offs3D, 2*radius, 2*radius);
  e.Graphics.FillEllipse(
    b4, -radius+2*offs3D, -radius+2*offs3D, 2*radius, 2*radius);
  e.Graphics.FillEllipse(
    b3, -radius+offs3D, -radius+offs3D, 2*radius, 2*radius);
  e.Graphics.FillEllipse(
    b2, -radius, -radius, 2*radius, 2*radius);
  // innerer Ring abgesenkt
  e.Graphics.FillEllipse(
    b4, -radius/2-2*offs3D, -radius/2-2*offs3D, radius, radius);
  e.Graphics.FillEllipse(
    b3, -radius/2-offs3D, -radius/2-offs3D, radius, radius);
  e.Graphics.FillEllipse(
    b1, -radius/2+offs3D, -radius/2+offs3D, radius, radius);
  e.Graphics.FillEllipse(
    b2, -radius/2, -radius/2, radius, radius);
  // Skalenstriche
  if (ticks != 0)
  {
    double ma = (maxAngle > minAngle) ? maxAngle : minAngle;
    double mi = (maxAngle > minAngle) ? minAngle : maxAngle;
    for (double i = mi; i <= ma +.0001; i += (ma - mi) / ticks )
    {
      float x1 = (float) (Math.Cos(i)*radius*1.1);
      float y1 = (float) (-Math.Sin(i)*radius*1.1);
      float x2 = (float) (Math.Cos(i)*radius*1.25);
      float y2 = (float) (-Math.Sin(i)*radius*1.25);
      e.Graphics.DrawLine(new Pen(b4),x1, y1, x2, y2);
    }
  }
  // Marke zeichnen
  Rectangle rect, rect1, rect2, rect3;
  rect = rect1 = rect2 = rect3 = MarkRect();
  rect1.X -= 2*offs3D; rect1.Y -= 2*offs3D;
  rect2.X -= offs3D; rect2.Y -= offs3D;
  rect3.X += offs3D; rect3.Y += offs3D;
  e.Graphics.FillEllipse(b4, rect1 );
```

```
  e.Graphics.FillEllipse(b3, rect2 );
  e.Graphics.FillEllipse(b1, rect3);
  e.Graphics.FillEllipse(Focused ? b3 : b2, rect);
}

private Rectangle MarkRect()
{
  int r = (int) radius/8;
  float angle = (float)
   ((maxAngle - minAngle)*(val-minimum)/(maximum - minimum) + minAngle);
  return new Rectangle (
    (int) (Math.Cos(angle)*radius/1.4 - r),
    (int) (-Math.Sin(angle)*radius/1.4 - r),
    2*r, 2*r);
}
```

Mausschnittstelle

Wie erwartet, erweist sich die Mausschnittstelle als »Schmankerl« der gesamten Implementierung. Der gesamte Code steckt in der Methode `OnMouseMove()` und einer kleinen Hilfsmethode `InDragArea()`, die prüft, ob sich der Mauszeiger auf der Regelbahn befindet. Die Umrechnung der Mausposition in den Wert `Value` erfordert zwei Schritte. Im ersten Schritt berechnet die Methode den Winkel relativ zum Drehpunkt des Reglers über den Arcustangens, um dann im zweiten Schritt unter Anwendung des Dreisatzes – diesmal in der anderen Richtung – aus dem Winkel den neuen Wert für `Value` zu gewinnen. Bis auf die Tatsache, dass die verwendete Methode `Math.Atan2` nur Winkel zwischen 0 und _ liefert und deshalb ein paar Fälle zu unterscheiden sind, damit die Markierung »stetig« läuft, ist die Berechnung äußerst geradlinig. Sobald der neue Wert `newValue` berechnet ist, prüft die Methode dann noch, ob die Änderung kleiner als ein Viertel des Wertebereichs ist. Das verhindert, dass der Regler von Endstellung zu Endstellung überspringt, sowie generell, dass der Regler keine zu großen Sprünge macht, wenn die Maus mal aus der Reglerbahn gerät.

```
protected override void OnMouseMove(MouseEventArgs e)
{
  if(e.Button == MouseButtons.Left)
  {
    int x, y;
    double newAngle;
    if (InDragArea(e.X, e.Y))
    {
      x = e.X-Width/2;     // Ursprung in die Mitte
      y = e.Y-Width/2;     // ...
      if (y > 0)    // Winkelkorrektur, da Atan2 nur Winkel zwischen
        newAngle = 2* Math.PI - Math.Atan2(y,x); // -p und p liefert
      else
        newAngle = -Math.Atan2(y,x);
      if (newAngle > maxAngle && minAngle < 0 ||
          newAngle > minAngle && maxAngle < 0 )
        newAngle -= 2*Math.PI;
```

```
      double newValue =
        (newAngle - this.minAngle)*(this.maximum -this.minimum)/
        (this.maxAngle-this.minAngle) + this.minimum;
      // Überspringen von Endstellung zu Endstellung unterdrücken
      // Lauf weicher machen
      if (Math.Abs(newValue - Value) < (maximum - minimum)/4)
        Value = newValue;
    }
  }
  base.OnMouseMove(e);
}
/// <summary>
/// Hilfsmethode: Testet, ob Punkt auf der Reglerbahn liegt.
/// </summary>
private bool InDragArea(int x, int y)
{
  x -= Width/2;
  y -= Width/2;
  float mouseDist = x*x + y*y;
  float outerDist = radius * radius;
  return mouseDist < outerDist && mouseDist > outerDist/4;
}
```

Die Tastaturschnittstelle

Wann immer das Steuerelement den Fokus besitzt, kann es auch auf Tastatureingaben des Benutzers reagieren. Wie das `TrackBar`-Steuerelement reagiert es auf die Richtungstasten und auf die Tasten [Bild ↑] und [Bild ↓]. Die `OnKeyDown()`-Überschreibung unterscheidet diese Tasten und ändert den Wert entsprechend.

Leider wartet diesmal die Standardimplementierung der Basisklasse `Control` mit einem kleinen Stolperstein auf. Es zeigt sich nämlich, dass `OnKeyDown()` die Richtungstasten im Normalfall gar nicht erst zu sehen bekommt. Sie werden in Weitergabebefehle für den Fokus umgemünzt und bleiben dabei auf der Strecke. Um dem entgegen zu wirken, bleibt nichts anderes übrig, als in die höheren Sphären der Nachrichtenbearbeitung einzugreifen und die Ereignisse auf den rechten Weg zu lenken – noch bevor diese Interpretation erfolgt. Als Übeltäter ist die von `Control` vererbte Methode `ProcessCmdKey()` schnell ausgemacht. Da sie virtuell ist, kann die Überschreibung die entsprechenden Ereignisse herausfiltern und an die für die reguläre Verarbeitung von Tastaturereignissen zuständige Methode `ProcessKeyEventArgs()` weiterleiten. Alle anderen Ereignisse nehmen weiterhin den Weg über die Basisklassenvariante.

```
protected override void OnKeyDown(System.Windows.Forms.KeyEventArgs e)
{
  switch(e.KeyCode)
  {
    case Keys.Right:
      Value += smallChange;
      goto common;
```

```
    case Keys.Left:
      Value -= smallChange;
      goto common;
    case Keys.Prior:
    case Keys.Up:
      Value += largeChange;
      goto common;
    case Keys.Next:
    case Keys.Down:
      Value -= largeChange;
     common:
      e.Handled=true;
      break;
  }
}
protected override bool ProcessCmdKey(ref Message m, Keys k)
{
  if ( k == Keys.Left || k == Keys.Right || k == Keys.Up || k == Keys.Down)
    return ProcessKeyEventArgs(ref m);
  else
    return base.ProcessCmdKey(ref m, k);
}
```

Tests

Führen Sie folgende Tests mit dem Steuerelement durch:

- Platzieren Sie verschiedene Instanzen auf dem Testformular und beobachten Sie, wie sich die `Text`-Eigenschaft automatisch anpasst.
- Führen Sie das Programm aus und bedienen Sie das Steuerelement `knob1` per Maus, Tastatur und `TrackBar`-Steuerelement. Achten Sie dabei auch auf die Fokusanzeige.
- Ändern Sie verschiedene Eigenschaften einer Steuerelementinstanz im Formularentwurf. Das Steuerelement spiegelt diese Änderungen auch in der Entwurfsansicht sofort wider.
- Ändern Sie nacheinander die Eigenschaften `Font` und `CaptionField.Font` sowie Text und `CaptionField.Text` und beobachten Sie die Wechselwirkungen.
- Ändern Sie andere Eigenschaften des `CaptionField`-Objekts und beobachten Sie den Effekt auf die `InitalizeComponent()`-Methode des Testformulars.
- Ändern Sie die Eigenschaften `Font` und `BackColor` des Testformulars und platzieren Sie eine weitere Instanz des Steuerelements darauf. Beobachten Sie den Effekt.

Übung

Wenn Sie Lust haben, sich mit der Implementierung des Steuerelements weiter vertraut zu machen, lösen Sie eine oder mehrere der folgenden Aufgaben:

1. Implementieren Sie eine `Dock`-Eigenschaft, die mit `CaptionField.Dock` synchronisiert ist und es ermöglicht, das Beschriftungsfeld wahlweise oben oder unten anzudocken. Passen Sie dafür auch die `OnPaint()`-Methode an.
2. Implementieren Sie ein `Click`-Ereignis, das bei einem Mausklick in den inneren Ring ausgelöst wird. Das Steuerelement soll sich im abgesenkten Zustand zeichnen, solange die Maustaste gedrückt ist.
3. Leiten Sie eine Klasse `DiscreteStateKnob` von `Knob` ab, die einen Stufenschalter mit `Ticks` diskreten Zuständen implementiert. (Überlegen Sie sich, was Sie am Klassendesign von `Knob` ändern müssen, um mit einer einfachen Überschreibung auszukommen.)
4. Entwerfen Sie einen Stereoregler, der es ermöglicht, zwei Markierungen mit der Maus zu bedienen.

16.4 Bestehende Steuerelemente modifizieren

Sie müssen nicht immer gleich ein neues `UserControl`-Benutzersteuerelement implementieren, wenn Ihnen die Ausstattung eines bestehenden Steuerelements missfällt oder für Ihre Zwecke zu dürftig ist. Der naheliegendste Ansatz zu einem eigenen Benutzersteuerelement führt über ein existierendes Steuerelement, das auf dem Wege der Vererbung modifiziert und gegebenenfalls mit zusätzlicher Funktionalität ausgestattet wird.

Besitzerseitig lässt sich ein Steuerelement nur soweit verbiegen, wie seine öffentliche Schnittstelle – also das Sammelsurium der von der Klasse offengelegten Eigenschaften, Methoden und Ereignisse – es erlaubt. Fast alle Steuerelementklassen definieren darüber hinaus `protected`-Elemente, die nur in abgeleiteten Klassen ansprechbar oder modifizierbar sind – denken Sie etwa an die zumeist als `protected virtual` vereinbarten Ereignissignalisierungsmethoden `OnXxx()`. Das Codebeispiel *Taschenrechner* (Seite 557) behalf sich beispielsweise mit einer `Button`-Ableitung, die den Eingabefokus schlicht verweigert. Dazu wurde die Methode `SetStyle()` eingesetzt, die als `protected` ausgewiesen ist und deshalb nur auf diesem Wege aufgerufen werden kann:

```
class MyButton : Button
{
  public MyButton()
  {
    this.SetStyle(ControlStyles.Selectable, false);
  }
}
```

Bis auf den Konstruktoraufruf verhält sich die neue `MyButton`-Klasse wie die alte. Ihre Objekte haben die gleichen Elemente, das gleiche Aussehen, usw. – Vererbung in Reinkultur.

16.4.1 Überlistung des Designers

Ganz Gallien? Nun ja, da wäre noch ein kleines Dorf ... Es zeigt sich, dass der Designer auf das Spiel nicht so ohne weiteres eingeht. Die Suche nach dem neuen Steuerelement in der TOOLBOX verläuft leider ergebnislos, wenn Sie nicht den zuvor beschriebenen »offiziellen« Weg über PROJEKT/BENUTZERSTEUERELEMENT HINZUFÜGEN gehen und ein neues Benutzersteuerelement anlegen. Wenn das Steuerelement nicht einmal im Lauf seiner Entwicklung von `UserControl` oder einer wiederum davon abgeleiteten Klasse abstammt und der Code in einer eigenen Datei steckt, ist nichts zu machen.

Dennoch gibt es eine Prozedur, mit der sich zumindest leben lässt – in der Hoffnung, dass sich die Problematik irgendwann mit einem der nächsten Updates für Visual Studio zu den Akten gesellt. (Wohlgemerkt: Diese Prozedur können Sie verwenden, wenn Sie den Weg über ein neues Benutzersteuerelement nicht gehen wollen und den Code für die Steuerelementklasse in derselben Datei wie die Formularklasse halten wollen, beispielsweise als verschachtelten Typ.)

Ziel der Prozedur ist es, Instanzen abgeleiteter Steuerelementklassen mit dem Designer im Formularentwurf platzieren und wie gewohnt initialisieren zu können. Die einzelnen Schritte sind:

1. Definieren Sie die abgeleitete Steuerelementklasse. Sie müssen an dieser Stelle noch nicht die vollständige Implementierung bereitstellen. Vom Prinzip her reicht ein leeres Gerüst:

   ```
   class MyButton : Button
   {
   }
   ```

2. Platzieren Sie ein Steuerelement der Basisklasse auf dem Formularentwurf und legen Sie dessen Eigenschaften und Ereignisroutinen fest. Den Bezeichner wählen Sie schon mal so, dass er auf die abgeleitete Steuerelementklasse passt, beispielsweise: `myButton1`.

3. Ändern Sie den Datentyp des Datenfelds für das Steuerelement im Quelltext; Setzen Sie Ihre neue Steuerelementklasse ein. Aus

   ```
   private System.Windows.Forms.Button myButton1;
   ```

   wird

   ```
   private MyButton myButton1;
   ```

4. Ändern Sie – in der Methode `InitializeComponent()` – die Instanziierung des Datenfelds. Setzen Sie den passenden Konstruktor ein:

```
this.myButton1 = new MyButton();
```

Das war's. Von nun an behält der Designer den neuen Datentyp bei und listet auch die mit der abgeleiteten Klasse hinzugekommenen Eigenschaften und Ereignisse auf. Auch die Verdrahtung neuer Ereignisse klappt hervorragend.

Code im Nachhinein in eine eigene Datei auslagern

Wenn Sie sich doch noch entschließen, den Code in eine eigene Datei zu verfrachten, fügen Sie ein neues `UserControl`-Benutzersteuerelement in das Projekt ein (in der `Toolbox` findet sich nun ein Symbol dafür). Dann kopieren Sie den Code in die neue *.cs*-Datei, wo Sie die Basisklasse vorübergehend in `UserControl` ändern und – ohne zu kompilieren – nur kurz in die Entwurfsansicht schalten. Die TOOLBOX bekommt so den neuen Klassenbezeichner für das Steuerelement mit, und es lässt sich nun wie gewohnt platzieren – freilich erst, nachdem Sie die ursprüngliche Basisklasse wieder gesetzt und das Projekt erfolgreich kompiliert haben.

Übrigens, und nicht ganz unwichtig: Ein auf diese Weise geschaffenes TOOLBOX-Symbol überlebt auch den nächsten Neustart von VS ☺.

16.4.2 Codebeispiel – MemTextBox

Das in diesem Abschnitt vorgestellte Codebeispiel *MemTextBox* hat es in sich. Es zeigt die Implementierung eines Textfelds, das sich die letzten *n* Einträge merkt und während der Eingabe dezent vorschlägt. Anstatt jedoch eine Liste einzublenden, wie Sie das von der Arbeit mit VS oder dem Internet Explorer her gewohnt sind, ergänzt das Textfeld seinen Vorschlag als Markierung a la Excel-Zelle. Abbildung 16.7 zeigt das Textfeld in Aktion. Der markierte Teil ist der Vorschlag des Textfelds. Damit der Benutzer auch Werte eingeben kann, die kürzer sind als ein Vorschlag, etwa nur den Wert »`mem`«, schneidet das Textfeld bei Fokusverlust den markierten Teil des Wertes ab, übernimmt ihn also nicht automatisch. Um einen Vorschlag zu übernehmen, kann der Benutzer die Taste → oder Ende betätigen. Dies setzt die Einfügemarke an das Ende der Markierung und hebt sie auf. Zudem besteht die Möglichkeit, mehrere passende Einträge über die Taste ↑ bzw. ↓ zu durchlaufen. (Auf einen Richtungswechsel verzichtet die Implementierung allerdings.)

Spezifikation

Die Spezifikation für das Steuerelement lautet:

1. Das Steuerelement soll eine frei definierbare Anzahl von Eingaben in einem privaten First-In-First-Out-Puffer speichern und diese während nachfolgender Eingaben vorschlagen.

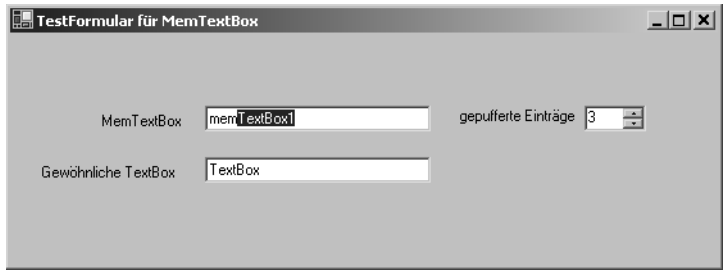


Abbildung 16.7: Vorschlag des MemTextBox-Textfelds während der Eingabe

2. Die Anzeige eines Vorschlags soll als markierter Text erfolgen.
3. Das Steuerelement soll die Tasten Auf und Ab dahingehend interpretieren, dass es jeweils den nächsten passenden Eintrag im Puffer abruft und vorschlägt.
4. Das Steuerelement soll bei Fokusverlust nur den Wert bis zum Beginn der Markierung behalten und in den Puffer übernehmen. Ist der gesamte Wert markiert, soll er hingegen gleichfalls übernommen werden.
5. Die Anzahl der gespeicherten Eingaben soll über die Eigenschaft `BufferedEntries` konfigurierbar sein.
6. Bei Wertänderungen von `BufferedEntries` sollen gespeicherte Einträge nach Möglichkeit erhalten bleiben.
7. Das Steuerelement soll sich ansonsten wie ein `TextBox`-Steuerelement verhalten.

Hintergründliches zum Quelltext

Der Quelltext der Klasse steckt in einer eigenen Datei *MemTextBox.cs*, die ursprünglich über den Befehl PROJEKT/BENUTZERSTEUERELEMENT HINZUFÜGEN dem Projekt des Testformulars hinzugefügt wurde – in der TOOLBOX findet sich somit ein Symbol dafür. Die ursprüngliche Basisklasse `UserControl` wurde dann in `TextBox` geändert. Wenn Sie die Implementierung des Steuerelements über den Befehl VORHANDENES ELEMENT HINZUFÜGEN in ein anderes Projekt übernehmen, kopiert der PROJEKTMAPPEN-EXPLORER die .cs-Datei in das jeweilige Projektverzeichnis, ohne jedoch ein Symbol dafür in der TOOLBOX zu erzeugen.

Implementierung

Die Implementierung gliedert sich in vier Aufgabenbereiche:

1. Definition der Klassenelemente und deren Initialisierung im Konstruktor
2. Verwaltung des Fifo-Puffers

3. Durchsuchen des Fifo-Puffers und Unterbreitung von Vorschlägen
4. Tastaturschnittstelle

Die folgenden Abschnitte besprechen diese Aufgabenbereiche im Einzelnen.

Datenfelder und Konstruktor

Die Ausstattung der von `TextBox` abgeleiteten Klasse `MemTextBox` mit Datenfeldern ist recht übersichtlich. Es gibt es eine Variable `maxElems` für die Anzahl der gepufferten Einträge und den Puffer `fifo`. Die anderen beiden Felder sind Zustandsvariablen und stehen im Dienst der einzelnen Methoden: `lastFound` speichert den Index des jeweils zuletzt gefundenen Vorschlags für eine eventuelle Fortsetzung der Suche (Tastaturschnittstelle), und `noSearch` erlaubt es, die Vorschlagsunterbreitung nur für die Tastatureingabe ein- und bei Fokusverlust abzuschalten. Als Datentyp für den Fifo-Eingabepuffer bietet sich `StringCollection` an, da diese Klasse Operationen für das wahlfreie Einfügen und Löschen von `string`-Einträgen an jeder beliebigen Position bereitstellt. Eine Fifo-Implementierung auf Basis des Datentyps `string[]` wäre – auch mit Blick auf die variable Eintragszahl – erheblich klobiger ausgefallen.

Der Konstruktor instanziiert den Puffer und setzt die Anzahl der möglichen Einträge auf einen Standardwert, den die Klasse als Konstante veröffentlicht. Die Eigenschaft `BufferedEntries` pflegt insbesondere `maxElems`. Weiterhin registriert der Konstruktor je eine Behandlungsmethode für die von der `TextBox`-Klasse vererbten Ereignisse `Leave`, `TextChanged` und `KeyDown`.

Sie werden sich an dieser Stelle vielleicht die Frage stellen, warum die Implementierung mit Behandlungsmethoden und nicht mit Überschreibungen der `OnXxx()`-Methoden für die Ereignisse arbeitet. Letzteres wäre natürlich möglich gewesen und wird von Microsoft an sich auch empfohlen. Da die Klasse für den Mechanismus der Ereignissignalisierung an sich jedoch nichts beizutragen hat, steht nichts dafür. Die Behandlungsmethoden sehen die Ereignisse *vor* dem Besitzer eines `MemTextBox`-Objekts, und das ist die Hauptsache. Immerhin: Sie sind mit Blick auf weitere Ableitungen der Klasse als `protected virtual` vereinbart, und lassen sich demnach überschreiben.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace MemText
{
  public class MemTextBox : TextBox
  {
    public const int DefaultNumberOfEntries = 10;
    private int maxElems;
```

```
    protected System.Collections.Specialized.StringCollection fifo;
    private int lastFound;
    private bool noSearch;

    public MemTextBox()
    {
      fifo = new System.Collections.Specialized.StringCollection();
      BufferedEntries = DefaultNumberOfEntries;
      InitializeComponent();
    }
...
  }
}
```

Der Fifo-Puffer und die BufferedEntries-Eigenschaft

Hinter der öffentlichen Eigenschaft `BufferedEntries` steht das private Datenfeld `maxElems`. Verkleinert sich dessen Wert, reduziert die Eigenschaft den Puffer unter Beachtung des Fifo-Prinzips auf die geforderte Elementzahl. Ist der Wert 0, wird der Puffer ganz geleert. Eine Anwendung kann dies mangels einer `Reset()`-Methode ausnutzen, um den Puffer zu leeren.

`BufferedEntries` taucht tatsächlich mitten unter den ererbten `TextBox`-Eigenschaften im EIGENSCHAFTEN-Fenster des Formularentwurfs auf. Damit ziemt es sich, auch eine Kurzbeschreibung für den unteren Bereich des EIGENSCHAFTEN-Fensters bereitzustellen. Ein Dokumentationskommentar nutzt leider hier nichts, Kurzbeschreibungen werden über ein `Description`-Attribut verfügbar gemacht.

```
[Description("Ruft ab, wieviele Einträge das Steuerelement maximal " +
   "speichert oder legt Maximum fest. " +
   "Pufferinhalte werden bei Änderung nach Möglichkeit übernommen. " +
   "Löst ArgumentException aus, wenn Wert negativ.")]
public int BufferedEntries
{
  get { return maxElems; }
  set
  {
    if (value < 0)               // negatives Argument?
      throw new ArgumentException(
        "Wert der Eigenschaft darf nicht negativ sein");
    maxElems = value;
    if (maxElems < fifo.Count)   // Puffer war größer?
      for(int i = maxElems; i< fifo.Count-1; i++) // älteren Rest
        fifo.RemoveAt(i);                         // abschneiden
  }
}
```

Für die Pflege des Pufferinhalts ist die Behandlungsroutine `MemTextBox_Leave()` zuständig, die zum Aufruf kommt, wenn das Textfeld den Eingabefokus verliert. Sie erklärt den nicht-markierten Teil der `Text`-Eigenschaft (wenn alles markiert ist, jedoch alles) zum Eingabewert des Textfelds und speichert ihn

als erstes Element im Puffer. Falls der Puffer dann mehr als `maxElems` Elemente hat, schneidet sie das letzte Element ab. Nachdem diese Methode die `Text`-Eigenschaft verändert, löst sie implizit ein `TextChanged`-Ereignis aus. Damit die `TextChanged`-Behandlung nicht sofort wieder einen Vorschlag unterbreitet (und die Markierung wieder ergänzt), erhält der Schalter `noSearch` den Wert `true`.

```
protected virtual void MemTextBox_Leave (object sender, System.EventArgs e)
{
  noSearch = true;                    // MemTextBox_TextChanged abschalten
  if (Text == "")
    return;
  if (SelectionLength > 0 && SelectionLength < Text.Length)
    SelectedText = "";
  fifo.Remove(Text);                  // Wenn vorhanden, Eintrag löschen
  fifo.Insert(0, Text);
  if (fifo.Count > maxElems)          // zu lang geworden?
    fifo.RemoveAt(fifo.Count-1);      // Letztes Element stutzen
}
```

Nach passenden Eingaben suchen

Die Behandlungsroutine `MemBox_TextChanged` kommt zum Aufruf, wann immer sich die `Text`-Eigenschaft ändert – gleich, ob das Programm den Wert neu setzt oder der Benutzer ein Zeichen eingibt. Damit ist sie prädestiniert dafür, die `Text`-Eigenschaft auszuwerten, den Puffer nach einem passenden Vorschlag zu durchsuchen und die mögliche Ergänzung als Markierung anzuhängen. Die `SelectionXxx`-Eigenschaften machen es möglich.

Da die Methode bei Unterbreitung eines Vorschlags ihrerseits den Wert der `Text`-Eigenschaft ändert, ruft sie sich implizit selbst auf. Obwohl keine Endlosrekursion auftreten würde, wenn die Methode den Vorschlag erneut suchen und setzen würde (die `Text`-Eigenschaft wird zwar neu gesetzt, ihr Wert bleibt aber unverändert, weshalb kein weiteres `TextChanged` zu befürchten steht), wäre es schade um die Rechenzeit. Zudem ermöglicht der für `MemTextBox_Leave()` ohnehin erforderliche Schalter `noSearch` einen unkomplizierten Abbruch bei Wiedereintritt.

```
protected virtual void MemBox_TextChanged(object sender, EventArgs e)
{
  if(noSearch) return;
  string s = Text;
  if (Text == "")
    return;
  if (fifoSearch(true, ref s))          // passenden Eintrag suchen
  {                                     // Eintrag übernehmen
    int SelStart = SelectionStart;      // und Ergänzung markieren
    noSearch = true;
    Text = s;
    noSearch = false;
```

```
    SelectionStart = SelStart;
    SelectionLength = s.Length-SelectionStart;
  }
}
```

Aufgabe der privaten Methode `fifoSearch()` ist es, den Puffer nach einem möglichen Vorschlag zu durchsuchen. Sie kennt zwei Modi, die sich darin unterscheiden, wo die Methode mit der Suche beginnt. Hat der Parameter `StartWithLastFound` den Wert `true`, beginnt die Suche an dem zuletzt gefundenen Element, ansonsten mit dem folgenden Element. Den jeweiligen Index hinterlegt die Methode in dem Datenfeld `lastFound`. Nachdem die Suche irgendwo in der Mitte des Puffers beginnen kann, verläuft sie ringförmig. Suchmuster und gefundener Wert werden über den `ref`-Parameter `Entry` ausgetauscht, da die Methode ihren Sucherfolg über den Rückgabewert kommuniziert.

```
private bool fifoSearch(bool StartWithLastFound, ref string Entry)
{
  int StartIndex = StartWithLastFound ? lastFound : lastFound + 1;
  for(int i = StartIndex; i < fifo.Count + StartIndex; i++)
    if (fifo[i % fifo.Count].StartsWith(Entry))
    {
      lastFound = i % fifo.Count;
      Entry = fifo[i % fifo.Count];
      return true;
    }
  return false;
}
```

Die Tastaturschnittstelle

Damit fehlt eigentlich nur noch die Tastaturschnittstelle, die aus der Not eine Tugend macht und nicht nur eine in der regulären Funktionsweise des Textfelds begründete Problematik korrigiert, sondern auch die Weiterschaltung zu anderen möglichen Vorschlägen per Richtungstasten implementiert.

Zur Problematik: [←] sollte jeweils das Zeichen unmittelbar vor der Schreibmarke löschen. Ist jedoch eine Markierung vorhanden, löscht diese Taste regulär aber nur die Markierung und verhält sich damit wie [Entf]. Für die Klasse `MemTexBox` ist es erforderlich, die Markierung Richtung Textanfang um ein Zeichen auszuweiten. Genau das macht die `MemTextBox_KeyDown`.

Die Weiterschaltung zum nächsten Vorschlag besteht in einem Aufruf der Methode `fifoSearch()` im entsprechenden Modus. Der Rest ist bereits gesagt.

```
protected virtual void MemTextBox_KeyDown (object sender, KeyEventArgs e)
{
  string search = null;
  switch(e.KeyCode)
  {
```

```
    case Keys.Back:          // Markierung erweitern
      if (SelectionStart > 0 && SelectionLength > 0)
      {
        SelectionStart--;
        SelectionLength++;
      }
      break;
    case Keys.Up:           // nächsten passenden Eintrag suchen
    case Keys.Down:
      search = Text.Substring(0, SelectionStart);
      if (fifoSearch(false, ref search))     // weitersuchen
      {
        int SelStart = SelectionStart;
        noSearch = true;
        Text = search;
        SelectionStart = SelStart;
        SelectionLength = Text.Length-SelStart;
      }
      e.Handled = true;
      break;
    default:
      noSearch = false;
      break;
  }
}
```

Testen des Steuerelements

Viel ist für den Test den Steuerelements nicht erforderlich: Das in Abbildung 16.7 gezeigte Formular reicht bereits aus. Es enthält zwei Textfelder, zwischen denen der Eingabefokus wechseln kann und ein `NumericUpDown`-Steuerelement zum Einstellen der Puffergröße. Um das `MemTextBox`-Steuerelement über den Designer zu platzieren, folgen Sie bitte den Ausführungen in Abschnitt 16.4.1 »Überlistung des Designers« (Seite 657). Die Koppelung mit dem `NumericUpDown`-Steuerelement ist trivial. Im vorliegenden Fall gilt der Initialisierungswert des `MemTextBox`-Steuerelements:

```
public Form1()
{
  this.InitializeComponent();
  numericUpDown1.Value = memTextBox1.BufferedEntries;
}
private void numericUpDown1_ValueChanged(object sender, System.EventArgs e)
{
  memTextBox1.BufferedEntries = (int) numericUpDown1.Value;
}
```

Nun können Sie das Steuerelement auf Herz und Nieren testen.

Übung

Das Textfeld eines `ComboBox`-Steuerelements kann zwar Vorschläge aus der anhängenden Liste unterbreiten, wenn die `DropDownStyle`-Eigenschaft auf

`DropDownList` gesetzt ist, in diesem Fall können aber keine neuen Werte eingegeben werden. Rüsten Sie ein `ComboBox`-Steuerelement mit der Funktionalität von `MemTextBox` aus. Das Steuerelement soll seine Vorschläge aus der anhängenden Liste beziehen und diese auch pflegen können.

16.5 Eigene Steuerelemente verfügbar machen

Sobald Ihr Steuerelement in seiner Testumgebung vernünftig funktioniert, werden Sie es in Ihre Anwendungen integrieren wollen. Dieser Abschnitt beschreibt, welche Möglichkeiten und Wege Ihnen dafür zur Verfügung stehen.

16.5.1 Als neues Element in das Projekt kopieren

Selbst wenn das Steuerelement in seiner Testumgebung richtig funktioniert, ist noch nicht gesagt, dass es das auch im richtigen Leben tut. Wenn Sie sich die Möglichkeit offen halten wollen, das Steuerelement für den Einsatz in Ihrem eigentlichen Projekt weiter zu bearbeiten oder auch nur seine Arbeitsweise im Debugger zu studieren, empfiehlt es sich, eine Kopie der .cs-Datei in das jeweilige Projekt übernehmen. Der Befehl dafür ist PROJEKT/VORHANDENES ELEMENT HINZUFÜGEN. Sie haben nun dieselbe Situation wie in dem Projekt, in dem Sie das Steuerelement entwickelt haben, nur dass jetzt das Formular Ihrer Anwendung an die Stelle des Testformulars tritt. Falls die Basisklasse des Steuerelements nicht `UserControl` ist, ändern Sie vorübergehend die Basisklasse in `UserControl`, schalten einmal in die Entwurfsansicht und setzen dann wieder die ursprüngliche Basisklasse ein, damit Sie in der TOOLBOX ein Symbol für das Steuerelement erhalten.

16.5.2 Steuerelemente in Steuerelementbibliotheken zusammenfassen

Falls Sie planen, ein oder mehrere eigene Steuerelemente dauerhaft über die TOOLBOX verfügbar zu machen, sind Sie mit einer Steuerelementbibliothek richtig beraten. Bei einer Steuerelementbibliothek handelt es sich um einen Projekttyp, der als Zieldatei eine *DLL* (dynamische Linkbibliothek) generiert. DLLs enthalten keinen startfähigen Code und lassen sich nur im Zusammenspiel mit einer EXE-Anwendung – also einem Projekt des Typs *Windows-Anwendung* – verwenden.

Es ist möglich, beliebig viele Steuerelemente in einer Steuerelementbibliothek zusammenzufassen. Für das Zusammenstellen einer Steuerelementbibliothek aus bereits bestehenden Steuerelementklassen verwenden Sie wie gehabt den Befehl PROJEKT/VORHANDENES ELEMENT HINZUFÜGEN (vgl. Abbildung 16.8).

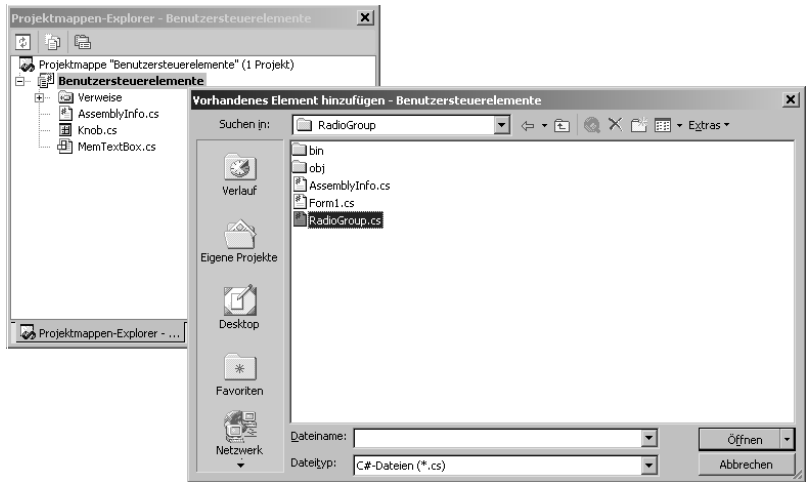


Abbildung 16.8: Zusammenstellen der Steuerelementbibliothek

Eine Steuerelementbibliothek in eine bestehende Projektmappe einfügen

Vom Prinzip her können Sie neue Steuerelemente auch sofort im Rahmen einer Steuerelementbibliothek entwickeln (PROJEKT/NEUES ELEMENT HINZUFÜGEN). Da Sie eine Steuerelementbibliothek zwar erstellen, aber nicht starten können, benötigen Sie für den Test aber auf jeden Fall ein startfähiges Projekt. Das gemeinsame Dach dafür ist die Projektmappe, die es Ihnen ermöglicht, unterschiedliche Projekte nebeneinander zu bearbeiten. Um Ihr Steuerelementbibliothek-Projekt mit einem bestehenden Projekt zu verheiraten, fügen Sie die Steuerelementbibliothek über den Befehl PROJEKT/VORHANDENES PROJEKT EINBINDEN in deren Projektmappe ein. (Es geht auch anders herum, aber dann müssen Sie das Startprojekt auf der EIGENSCHAFTSSEITE der Projektmappe anpassen.) Damit die Elemente der Steuerelementbibliothek in dem bestehenden Projekt verfügbar werden, müssen Sie dort noch über den Befehl PROJEKT/VERWEIS HINZUFÜGEN einen Verweis auf die Bibliothek ergänzen (vgl. Abbildung 16.1 auf Seite 636).

Von nun an können Sie beide Projekte nebeneinander entwickeln und die Steuerelemente aus der Steuerelementbibliothek im anderen Projekt einsetzen. Beachten Sie allerdings, dass Sie mit einer Kopie der DLL arbeiten und dass Ihre Anwendung spätere Änderungen der Bibliothek nicht automatisch mitbekommt, wenn Sie das Bibliotheksprojekt von einer anderen Projektmappe aus bearbeiten und aktualisieren.

Starke Namen und der globale Assembly Cache

Falls Sie generell nur mit einer Version der Bibliothek arbeiten wollen und nicht mit einer Kopie, setzen Sie die Eigenschaft `Lokale Kopie` des Verweises auf `false` (Abbildung 16.9). Das allein reicht aber noch nicht. Das .NET-Laufzeitsystem sucht die Datentypen aus der Bibliothek in diesem Fall im GLOBALEN ASSEMBLY CACHE, eine Sammlung von Verweisen auf signierte und offiziell registrierte DLLs. Um einen Blick in diesen Cache zu werfen,

geben Sie in Visual Studio .NET den Menübefehl EXTRAS/ASSEMBLY CACHE oder – falls Sie dieses Tool nicht installiert haben – öffnen Sie im Windows Explorer den Ordner *WINNT\Assembly*.

Die folgende Prozedur beschreibt, welche Schritte Sie unternehmen müssen, um eine Bibliothek in den globalen Assembly Cache aufzunehmen:

1. *Signieren Sie die Assembly (.dll) der Bibliothek*. Dazu starten Sie eine Kommandozeilensitzung via START/PROGRAMME/MICROSOFT VISUAL STUDIO .NET/ VISUAL STUDIO .NET TOOLS/ VISUAL STUDIO .NET EINGABEAUFFORDERUNG, wechseln in das Projektverzeichnis der Bibliothek und geben das Kommando:

   ```
   sn -k Bibliothek.snk
   ```

   Das Tool *sn.exe* (Strong Name) generiert ein für die Signierung geeignetes Schlüsselpaar und schreibt dieses in die Datei `Bibliothek.snk` – für `Bibliothek` setzen Sie am besten den Namen Ihrer Bibliothek. Beenden Sie die Kommandozeilensitzung noch nicht.

2. Wechseln Sie zurück nach Visual Studio und öffnen Sie die Datei *AssemblyInfo.cs* des Bibliotheksprojekts. Setzen Sie die folgende Pfadangabe in die bereits bestehende `AssemblyKeyFile`-Zeile:

   ```
   [assembly: AssemblyKeyFile("..\\..\\Bibliothek.snk")]
   ```

   Kompilieren Sie das Projekt. Wenn alles geklappt hat, haben Sie eine signierte Version der Bibliothek generiert, die Sie im nächsten Schritt in den globalen Assembly Cache einfügen.

3. Schalten Sie in die Kommandozeilensitzung zurück und wechseln Sie dort in das Verzeichnis, in dem sich die signierte DLL befindet – also *bin\Debug* oder *bin\Release*. Hier geben Sie nun den Befehl:

   ```
   gacutil -i Bibliothek.dll
   ```

   Wenn alles glatt gegangen ist, erhalten Sie die Meldung: `Assembly successfully added to the cache` und finden die Bibliothek nun auch im Assembly Cache *WINNT\Assembly*.

   Um die Bibliothek wieder aus dem Cache zu entfernen, lautet der Aufruf (Achtung! Keine Dateierweiterung angeben):

   ```
   gacutil -u Bibliothek
   ```

Die Alternative zu `gacutil` *ist eine zusammen mit Visual Studio .NET installierte Erweiterung des Explorers, die das Verzeichnis WINNT\Assembly auf besondere Weise darstellt – ähnlich wie beispielsweise das Fonts-Verzeichnis des Systems. Mit dieser Erweiterung reicht eine einfache Ziehaktion der DLL im Explorer (vom Projektverzeichnis in den Assembly-Cache).*



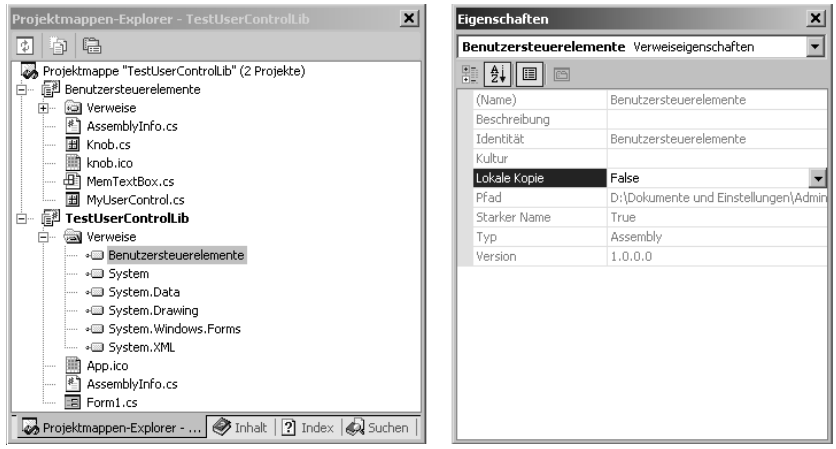


Abbildung 16.9: Mit der ursprünglichen Version der Steuerelementbibliothek weiterarbeiten

Nur einen Verweis auf eine bestehende Steuerelementbibliothek einfügen

Falls Sie nicht beabsichtigen, Änderungen am Projekt für die Steuerelementbibliothek vorzunehmen, genügt es auch, wenn Sie Ihrem bestehenden Projekt nichts weiter als einen Verweis darauf spendieren, um den Inhalt der Steuerelementbibliothek verfügbar zu machen. Auch hier können Sie sich wieder dafür entscheiden, ob Sie den augenblicklichen Stand der Steuerelementbibliothek einfrieren und eine lokale Kopie der DLL anlegen wollen oder ob das Projekt immer mit der aktuellsten Version der Steuerelementbibliothek arbeiten soll.

Leider erhalten Sie so keine Symbole in der Toolbox dafür, weshalb sich die im folgenden Unterabschnitt beschriebene Prozedur als bessere Alternative empfiehlt.

Steuerelemente einer Steuerelementbibliothek in die Toolbox einfügen

Um Steuerelemente einer Steuerelementbibliothek jederzeit in allen Anwendungen zur Verfügung zu haben, besteht grundsätzlich die Möglichkeit, die Toolbox zu erweitern. Dazu geben Sie den Befehl EXTRAS/TOOLBOX ANPASSEN, schalten in dem erscheinenden Dialogfeld auf die Seite .NET FRAMEWORK-KOMPONENTEN und klicken dort auf die Schaltfläche DURCHSUCHEN. Im erscheinenden ÖFFNEN-Dialog suchen Sie nun das Verzeichnis mit der DLL der Steuerelementbibliothek und wählen diese aus (Abbildung 16.10).

Wieder zurück im Dialog TOOLBOX ANPASSEN suchen Sie die Steuerelemente, die Sie ständig verfügbar machen wollen und setzen ein Häkchen davor. Von nun an finden Sie Symbole für die Steuerelemente in der TOOLBOX (Abbildung 16.11), und der Designer ergänzt automatisch einen Verweis auf die Steuerelementbibliothek, wenn Sie Instanzen der neuen Steuerelemente in einen Entwurf platzieren. Dieser Weg erscheint als der sinnvollste.

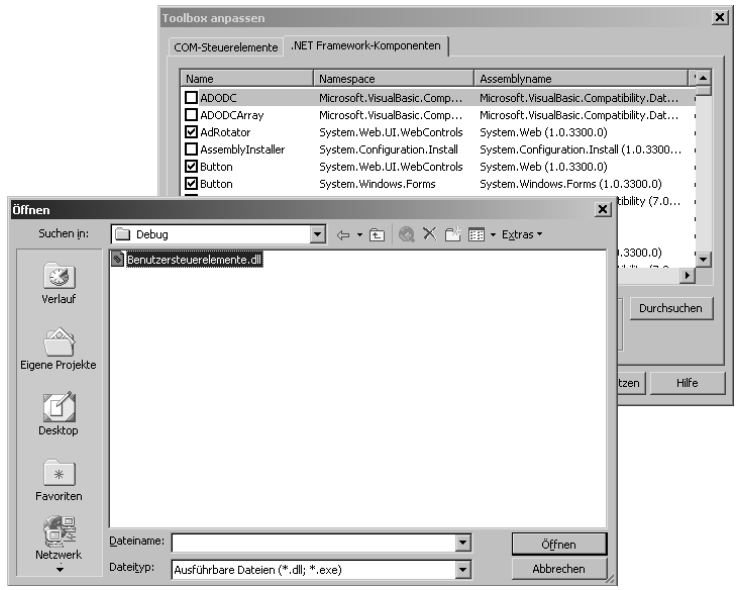


Abbildung 16.10: Steuerelemente aus der eigenen Steuerelementbibliothek dauerhaft in die Toolbox übernehmen

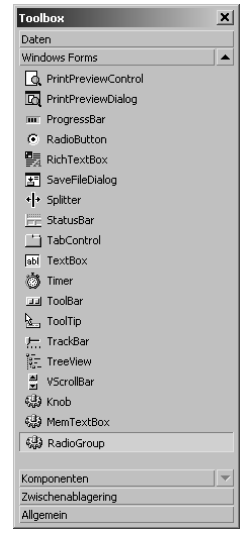


Abbildung 16.11: Die Steuerelemente aus der Steuerelementbibliothek erscheinen in der Toolbox und stehen global in allen Projekten zur Verfügung.

16.6 Eigene Benutzersteuerelemente weiter ableiten

Dass sich eigene Benutzersteuerelemente als Basis für die weitere Ableitung verwenden lassen, versteht sich alleine schon aus dem Selbstverständnis der Sprache C# heraus. Dass es dafür auch eine »offizielle« Unterstützung durch den Designer gibt, steht auf einem anderen Blatt.

Es scheint beinahe so, als würde die deutsche Übersetzung der Benutzeroberfläche hier ihr Bestes geben, um diese Möglichkeit zu verschleiern. Vielleicht haben Sie ihn ja schon (absichtlich oder versehentlich) ausprobiert, den Befehl PROJEKT/GEERBTES STEUERELEMENT HINZUFÜGEN – und sind an abstrusen Fehlermeldungen gescheitert oder gar der Meinung, dass dieser Befehl in der ansonsten gut gelungenen Benutzerführung von Visual Studio .NET zwangsweise in die Sackgasse führt. In diesem Fall sollten Sie die folgende Anleitung mit dem Untertitel »und er funktioniert doch« lesen und im Hinterkopf behalten, dass dem Befehl die Übersetzung »Ableitung von Benutzersteuerelement hinzufügen« wesentlich besser stehen würde. Analoges gilt übrigens für den Befehl PROJEKT/GEERBTES FORMULAR HINZUFÜGEN, den der Abschnitt »Teil 2 – Ableitung der Klasse MyDerivedDialog in einem eigenen Projekt« (Seite 718) weiter ausleuchtet. An dieser Stelle finden Sie auch Erläuterungen zum Thema »Vererbung und Sicherheit« und Verkapselungsstrategien.

16.6.1 Ausgangssituation

In der Codeansicht ist es kein Thema: Die Ableitung einer Klasse von einer nicht versiegelten Basisklasse funktioniert; so will es die Spezifikation der Sprache. Damit auch der Designer mitspielt und eine Entwurfsansicht anzeigt, bedarf es einer speziellen Ausgangssituation.

- Das als Basis fungierende Steuerelement muss direkt oder indirekt von `UserControl` abstammen.
- Das als Basis fungierende Steuerelement muss als Klassenbibliothek (.dll) vorliegen.

Sind diese Voraussetzungen nicht erfüllt, funktioniert die Ableitung über die Benutzerführung von Visual Studio nicht, und der Designer kann keine Entwurfsansicht für das neue Steuerelement anzeigen.

Benutzersteuerelementvererbung schrittweise

Die folgende Anleitung beschreibt die Ableitung eines neuen Benutzersteuerelements von einem bestehenden Benutzersteuerelement:

1. Öffnen Sie das Projekt, dem Sie das neue Element hinzufügen wollen, oder legen Sie ein neues Projekt dafür an.
2. Binden Sie vom PROJEKTMAPPEN-EXPLORER aus über den Kontextmenübefehl HINZUFÜGEN/VORHANDENES PROJEKT der Projektmappe das Projekt für das bestehende Steuerelement ein.

   Falls Ihnen kein Projekt für das Benutzersteuerelement zur Verfügung steht oder Sie keine Veränderungen des Benutzersteuerelements vornehmen wollen, kann dieser Schritt zusammen mit Schritt 3 entfallen.

3. Fügen Sie in das Projekt einen Verweis auf das hinzugefügte Projekt ein. Der dazu notwendige Befehl ist VERWEISE/VERWEIS HINZUFÜGEN/PROJEKTE. Achten Sie darauf, dass die Projekteigenschaft AUSGABETYP des Benutzersteuerelementprojekts auf *Klassenbibliothek* gesetzt ist und kompilieren Sie die Projektmappe.

4. Geben Sie nun den Befehl PROJEKT/GEERBTES STEUERELEMENT HINZUFÜGEN. Sie erhalten zuerst den Dialog NEUES ELEMENT HINZUFÜGEN, in dem Sie den Namen für das neue Benutzersteuerelement eingeben, und dann das Dialogfeld VERERBUNGSAUSWAHL zur Auswahl der Basisklasse für das neue Benutzersteuerelement (Abbildung 16.13).

   Falls Sie nur mit der DLL des Benutzersteuerelements arbeiten wollen und die Schritte 2 und 3 übersprungen haben, ist das Dialogfeld zunächst leer (Abbildung 16.12). In diesem Fall klicken Sie auf SUCHEN und wählen die DLL manuell aus.

Nachdem Sie den Dialog VERERBUNGSAUSWAHL bestätigt haben, finden Sie eine neue *.cs*-Datei, die Sie wie gewohnt in der Entwurfsansicht bearbeiten können – mit dem kleinen Unterschied, dass Sie nicht den leeren Clientbereich des `UserControl`-Objekts sehen, sondern das als Basis fungierende Benutzersteuerelement, wie in Abbildung 16.14 gezeigt. Der Designer hat hier also eine Instanz des Basis-Steuerelements eingefügt, die sich selbst zeichnet, und zeigt auch deren Eigenschaften im EIGENSCHAFTEN-Fenster an – gleichfalls wie gewohnt.

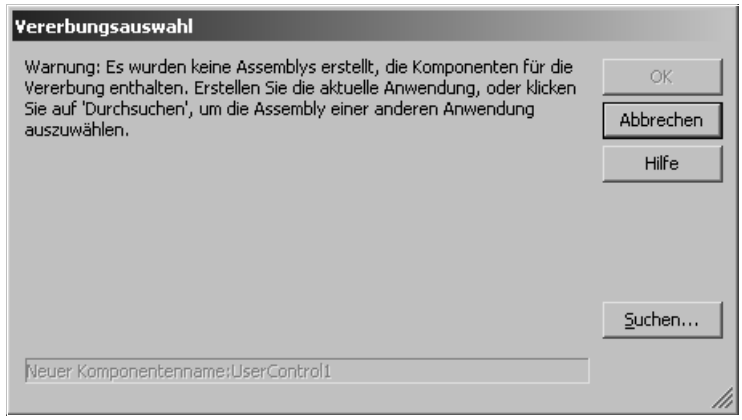


Abbildung 16.12: Hier geht es noch weiter – über die Schaltfläche SUCHEN

Abbildung 16.13: Auswahl der Basisklasse

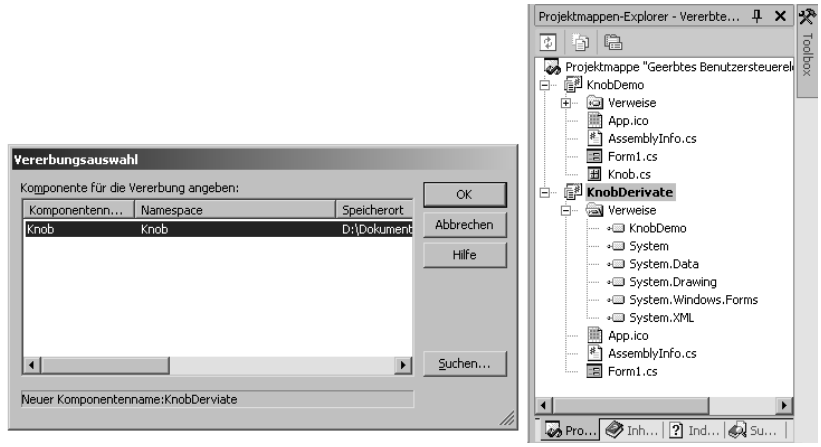
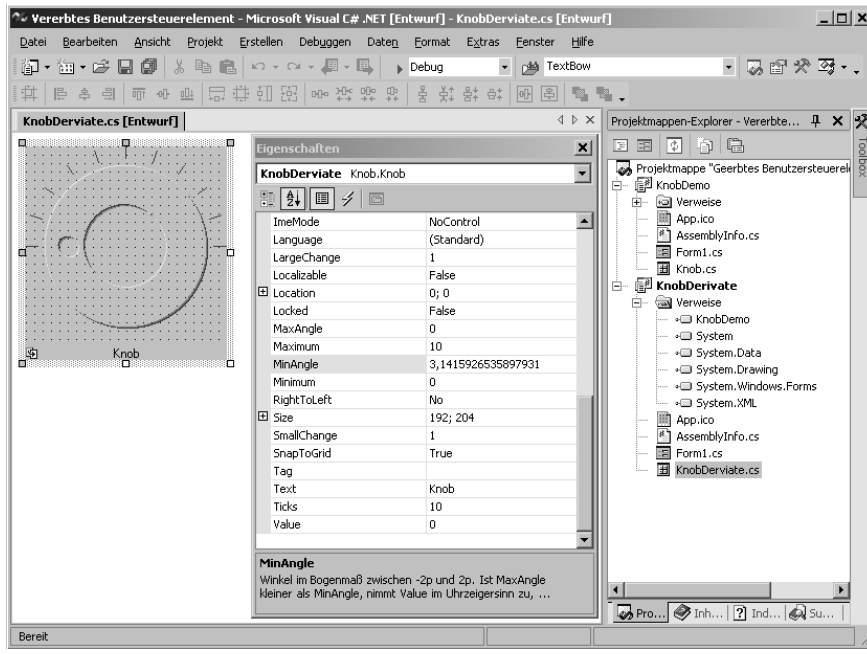


Abbildung 16.14: Entwurfsansicht des abgeleiteten Benutzersteuerelements mit EIGENSCHAFTEN-Fenster. Das als Basis fungierende Benutzersteuerelement zeichnet sich als eigenständige Instanz selbst und enthält ein Vererbungssymbol (links unten).



17 Dialogfelder

An anderer Stelle wurde bereits gezeigt, dass eine Formularinstanz jederzeit auch weitere Instanzen seiner eigenen Klasse ins Leben rufen und steuern kann. Obwohl Sie prinzipiell jedes Formular von jedem Formular aus aufrufen können und Sie sich die abenteuerlichsten Mechanismen für die Wertübergabe ausdenken können, werden Sie sich im Allgemeinen bereits beim Design festlegen, ob das Formular als *modaler Dialog* über `ShowDialog()` oder als *modusloser Dialog* über `Show()` aufgerufen wird. Diese beiden nicht gerade sehr aussagekräftigen Begriffe haben sich mal wieder als allzu wörtliche Übersetzungen der amerikanischen Ausdrücke *modal* und *modeless* in die deutsche Fachsprache eingeschlichen. Statt von »moduslosen« Dialogen (oder noch schlimmer: »Dialogen ohne Modus«) sollte man wenigstens von *nichtmodalen* Dialogen sprechen.

Ein MODALER DIALOG konzentriert alle Benutzereingaben der Anwendung so lange auf sich, bis er beendet wird. Der Benutzer hat damit bei Anzeige eines modalen Dialogs keine Möglichkeit, in ein anderes Formular derselben Anwendung (bzw. desselben Prozesses) zu wechseln.

Als NICHT-MODALER DIALOG lässt sich vom Prinzip her jedes mit `Show()` aufgerufene Formular auffassen. Kennzeichnend für den nicht-modalen Dialog ist aber nicht allein, dass der Benutzer jederzeit in das aufrufende Formular oder andere nicht-modale Dialoge bzw. Formulare wechseln kann, sondern auch, dass er das aufrufende Formular unmittelbar über jede Wertänderung informiert, damit es umgehend auf Benutzereingaben reagieren kann. Sieht man einmal davon ab, dass nicht-modale Dialoge eigene Fenster mit Schließbox sind, die der Benutzer frei anordnen kann, verhalten sie sich also nicht viel anders als Steuerelemente. Die Möglichkeit, Symbolleisten in schwebende nicht-modale Fenster zu verwandeln, kommt daher nicht von ungefähr. Sie ist naheliegend.

Dieses Kapitel beleucht das Wie und Warum sowie das programmtechnische Umfeld von Dialogen. Im ersten Teil stellt es fertige Dialog-Mechanismen und -Komponenten vor, für die in der .NET-Klassenbibliothek bereits einsatzfertige Implementierungen zu finden sind: die Klasse `MessageBox` und die sechs Windows-Standarddialoge. Der zweite Teil konzentriert sich auf die Konzeption und Implementierung eigener Dialoge und nimmt sich dabei der zentralen Themen Aufruf, Gültigkeitsprüfung und Wertübernahme an.

Der letzte Themenkomplex geht schließlich auf die Vererbung von Dialogen ein.

17.1 Modale Dialoge

Enthält ein modaler Dialog nur eine Schaltfläche – für gewöhnlich die OK-Schaltfläche –, spricht man von einem Meldungsfenster. Um Textmeldungen auszugeben oder Rückfrage zu halten, brauchen Sie allerdings nicht eigens ein Formular zu entwerfen. In solchen Fällen bedienen Sie sich einfach der Methode `MessageBox.Show()`. Obwohl die Methode verwirrenderweise den Namen `Show()` trägt, ist der Aufruf modal. Da hat wohl jemand geschlafen – »ShowDialog()« oder »ShowModal()« hätte besser ins Konzept gepasst.

17.1.1 MessageBox.Show()

Trotz ihrer zwölf Überladungen ist `MessageBox.Show()` ausgesprochen pflegeleicht. Für einfache Textmeldungen übergeben Sie der Methode ein oder zwei Strings – der erste ist der Meldungstext, der zweite der Titel des Meldungsfensters – und die Sache hat sich:

```
MessageBox.Show("Achtung Programmabsturz um 4:25", "Systemadministrator");
```

Auch Fragen des Typs JA/NEIN oder ABBRECHEN/WIEDERHOLEN/IGNORIEREN sind möglich. Dazu sind zwei weitere Dinge erforderlich: Erstens müssen Sie der Methode über einen Wert der `MessageDialogResults`-Aufzählung mitteilen, welche Schaltflächen sie anzeigen soll, und zweitens den Funktionswert – ein Element der `DialogResults`-Aufzählung – analysieren.

Tabelle 17.1: Elemente der `MessageDialogResults`-Aufzählung

enumWert	**Schaltflächen**
`AbortRetryIgnore`	ABBRECHEN – WIEDERHOLEN – IGNORIEREN
`OK`	OK (Vorgabe)
`OKCancel`	OK – ABBRECHEN
`RetryCancel`	WIEDERHOLEN – ABBRECHEN
`YesNo`	JA – NEIN
`YesNoCancel`	JA – NEIN – ABBRECHEN

Tabelle 17.2: Elemente der `DialogResult`-Aufzählung

enum-Wert	Bedeutung
`Abort`	Dialog wurde mit ABBRECHEN verlassen.
`Cancel`	Dialog wurde mit ABBRECHEN, Esc oder durch das Schließen des Fensters abgebrochen.
`Ignore`	Dialog wurde mit IGNORIEREN verlassen.
`No`	Dialog wurde mit NEIN verlassen.
`None`	Dialog wurde mit KEINE verlassen.
`OK`	Dialog wurde mit OK verlassen.
`Retry`	Dialog wurde mit WIEDERHOLEN verlassen.
`Yes`	Dialog wurde mit JA verlassen.



`MessageBox.Show()` *ist auch bei einer einfachen Meldung, die ausschließlich eine OK-Schaltfläche enthält, nicht auf das Funktionsergebnis* `DialogResult.OK` *beschränkt. Wenn der Benutzer das Dialogfeld über einen Klick in seinem Schließfeld (oder mit* Alt+F4*) beendet, liefert die Methode hier – wie auch bei allen anderen Dialogvarianten – das Funktionsergebnis* `DialogResult.Cancel`.

Optional können Sie der Methode im vierten Parameter noch ein Element der `MessageBoxIcons`-Klasse übergeben, um die Meldung mit einem der originalen Windows-Symbole für Meldungsfenster zu verzieren. Das folgende Codefragment zeigt ein typisches Szenario für einen solchen Aufruf (Abbildung 17.1):

```
DialogResult res = DialogResult.OK;
do
{
  try
  {
    WriteMyFile();
  }
  catch (Exception exc)
  {
    string msg =
     "Beim Schreiben in die Datei ist folgender Fehler aufgetreten: " +
      exc.Message;
    res = MessageBox.Show(msg, "Fehlermeldung",
            MessageBoxButtons.AbortRetryIgnore, MessageBoxIcon.Question);
  }
} while (res == DialogResult.Retry);
if (DialogResult == DialogResult.Ignore)
{
  ...
```

Abbildung 17.1:
Das Meldungfenster

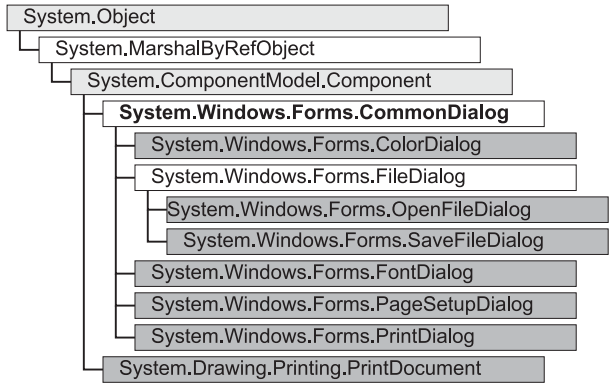


17.1.2 Standarddialoge

Seit den Zeiten von Windows 95 unterliegt nicht nur die grafische Benutzeroberfläche, sondern auch die Eingabeschnittstelle dem Gedanken des »Corporate Design«. Der Benutzer soll eine Anwendung intuitiv bedienen und einmal erlernte Arbeitsabläufe möglichst anwendungsübergreifend anwenden können. Ein wichtiges Ergebnis dieser Entwicklung sind die Standarddialoge. Das System stellt sie bereit, um Vorgänge wie die Datei- oder Druckerauswahl, die im Funktionsumfang fast jeder Anwendung enthalten sind, erstens nach einem übergeordneten Schema abzuwickeln und zweitens nicht jedes Mal neu implementieren zu müssen.

Ein Blick in die TOOLBOX von VS.NET fördert sechs Komponenten zu Tage – allesamt alte Bekannte, die von der abstrakten Basisklasse `CommonDialog` abstammen und zu den Standarddialogen gerechnet werden (vgl. Bild 17.2).

Abbildung 17.2:
Ableitung der Standarddialoge



Der Einsatz dieser Komponenten in eigenen C#-Anwendungen läuft nach Schema F:

1. Wählen Sie die gewünschte Dialogkomponente in der TOOLBOX aus (vgl. Abbildung 17.3).
2. Fügen Sie die Komponente durch Aufziehen oder schlichtes Klicken in das Entwurfsfenster ein. (Komponenten erscheinen in einem abgetrenn-

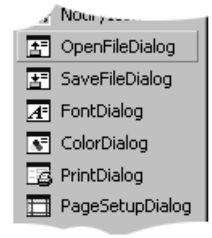


Abbildung 17.3: Die fünf Standarddialoge sind in der TOOLBOX zu finden.

ten Fensterbereich unterhalb des Formularentwurfs). Der Designer vereinbart dafür wie gewohnt in der Formularklasse ein Instanzdatenfeld; das zugehörige Objekt wird in der Methode `InitializeComponent()` erzeugt und initialisiert. Für die Komponente `OpenFileDialog` sieht das so aus:

```
public System.Windows.Forms.OpenFileDialog openFileDialog1;
...
private void InitializeComponent()
{
  this.openFileDialog1 = new System.Windows.Forms.OpenFileDialog();
 ...
}
```

3. Legen Sie die Dialog-Eigenschaften, soweit erforderlich und möglich, im EIGENSCHAFTEN-Fenster fest.

4. Ergänzen Sie den Code für weitere individuelle Initialisierungen sowie den allfälligen `ShowDialog()`-Aufruf, der den Dialog als modalen Dialog anzeigt. (Ein modaler Dialog konzentriert alle an die Anwendung gerichteten Eingaben auf sich und blockiert damit alle anderen Fenster der Anwendung.) Für ein `OpenFileDialog`-Objekt sieht das etwa so aus:

```
bool OpenFile()
{
  this.openFileDialog1.Filter = "Textdatei (*.txt; *.asc)|*.txt;*.asc";
  if (this.openFileDialog1.ShowDialog() == DialogResult.OK)
     ...
}
```

5. Werten Sie das Ergebnis von `ShowDialog()` aus. Falls der Benutzer den Dialog nicht abgebrochen hat, fragen Sie die Eigenschaften des Dialogobjekts ab und erhalten so die Benutzereingaben.

6. Wenn nötig, implementieren Sie Behandlungsmethoden für die Ereignisse, die der Dialog in bestimmten Konfigurationen generiert. Die `OpenFileDialog`-Komponente signalisiert beispielsweise ein `FileOk`-Ereignis, wenn der Benutzer auf OK klickt und ermöglicht – noch während des `ShowDialog()`-Aufrufs – die Prüfung der Eingaben. Die folgende Behandlungsmethode akzeptiert beispielsweise keine UNC-Dateinamen:

```
private void openFileDialog1_FileOk(object sender,
  System.ComponentModel.CancelEventArgs e)
{
  if (this.openFileDialog1.FileName.StartsWith("\\\\"))
  {
    MessageBox.Show("UNC-Namen (?????\\\\Server\\...) nicht erwünscht",
                    "Fehler");
    e.Cancel = true;
  }
}
```

Wenn Sie den Umgang mit einer Komponente verstanden haben, werden Sie auch mit den anderen keine großen Verständnisschwierigkeiten haben.

Tabelle 17.3 und Tabelle 17.4 zeigen exemplarisch die Ausstattung der Komponente `OpenFileDialog` in fast vollständiger Form. Im Allgemeinen verraten bereits die Bezeichnung und der Datentyp der einzelnen Elemente, wofür sie gedacht sind. Nehmen Sie sich am besten ein paar Minuten Zeit und studieren Sie die Elemente der `OpenFileDialog`-Komponente. Überlegen Sie sich, für welche Einsatzbereiche sie eine Rolle spielen. Die anderen Komponenten sind nach dem gleichen Schema gestrickt.

Tabelle 17.3: Wichtige Eigenschaften eines `OpenFileDialog`-Objekts

Eigenschaft	Bedeutung
`bool AddExtension {get; set;}`	Bestimmt, ob das Steuerelement die Standarderweiterung `DefaultExt` anhängt, wenn der Benutzer keine Erweiterung angibt.
`string DefaultExt {get; set;}`	Standarderweiterung, wird in Abhängigkeit von `AddExtension` angehängt, wenn der Benutzer keine Erweiterung angibt.
`bool CheckFileExists {get; set;}`	Wenn `true`, stellt das Steuerelement sicher, dass der angegebene Dateiname eine existierende Datei bezeichnet.
`bool CheckPathExists {get; set;}`	Wenn `true`, stellt das Steuerelement sicher, dass der angegebene Pfad existiert.
`bool DereferenceLinks {get; set;}`	Wenn `true`, liefert das Steuerelement anstatt von .lnk-Dateinamen deren Verknüpfungsziele.
`string FileName {get; set;}`	Dateiname samt Pfad (vollqualifiziert). Enthält nach Anzeige des Dialogs die Auswahl des Benutzers. Der Wert kann nach Anzeige des Dialogs auch `null` sein, falls die Eigenschaft zuvor nicht initialisiert wurde und der Benutzer den Dialog abbricht. (Abgebrochene Dialogfelder sollte man allerdings sowieso bestenfalls in Scherzprogrammen auswerten.)

Eigenschaft	Bedeutung
`bool Multiselect {get; set;}`	Wenn `true`, erlaubt das Dialogfeld dem Benutzer die Auswahl mehrerer Dateinamen.
`string [] FileNames {get; set;}`	Dateinamen samt Pfad (vollqualifiziert). Enthält nach Anzeige des Dialogs die Auswahl des Benutzers. Bei Einfachauswahl enthält das Array nur ein Element. Der Wert kann nach Anzeige des Dialogs auch `null` sein (vgl. `FileName`).
`string Filter {get; set;}`	Filterspezifikation; Kann beliebig viele *paarige* Einträge des Typs *Filterbezeichnung* | *Filterausdruck* enthalten, Platzhalterzeichen sind erlaubt). Hier ein Muster: `"alle (*.*) | *.* | Dokumente (*.doc;*.dok) | *.doc; *.dok"`
`int FilterIndex {get; set;}`	Index des bei Anzeige des Dialogs wirksamen Filters (vgl. `Filter`-Eigenschaft)
`bool ShowReadOnly {get; set;}`	Wenn `true`, Anzeige des Kontrollkästchens SCHREIBGESCHÜTZT ÖFFNEN, und Auswahlmöglichkeit durch den Benutzer.
`bool ReadOnlyChecked {get; set;}`	Zustand des Kontrollkästchens SCHREIBGESCHÜTZT ÖFFNEN. `true` entspricht einem Häkchen.
`bool RestoreDirectory {get; set;}`	Wenn `true`, übernimmt der Dialog die Pfadauswahl des Benutzers für nachfolgende Aufrufe nicht.
`string Title {get; set;}`	Text in Titelleiste des Dialogs. Wenn `null`, lautet die Standardvorgabe "Öffnen".
`bool ValidateNames {get; set;}`	Wenn `true`, stellt der Dialog sicher, dass der Benutzer nur gültige Win32-Dateinamen eingeben kann.

Tabelle 17.3: Wichtige Eigenschaften eines `OpenFileDialog`-Objekts (Forts.)

Methode	Bedeutung
`DialogResult ShowDialog()`	Anzeige des Dialogs
`System.IO.Stream OpenFile()`	Öffnet die über die Eigenschaft `FileName` spezifizierte Datei für Leseoperationen
`void Reset()`	Setzt den Dialog auf die Standardvorgaben zurück

Tabelle 17.4: Wichtige Methoden eines `OpenFileDialog`-Objekts

Die `ShowDialog()`-Methoden aller Komponenten geben einen Wert des `enum`-Typs `DialogResult` zurück, der Aufschluss darüber gibt, mit welcher Schaltfläche der Benutzer den Dialog beendet hat (Tabelle 17.2).

Codebeispiel – Standarddialoge

Das Projekt *Standarddialoge* demonstriert den prinzipiellen Umgang mit den sechs Standarddialogen. Der in Abbildung 17.4 gezeigte Formularentwurf sieht für den Aufruf eines jeden Dialogs ein `Button`-Steuerelement vor. Weiterhin finden sich im unteren Bereich des Entwurfsfensters sieben Komponenten – je eine für die sechs Standarddialoge sowie eine `PrintDocument`-Komponente, die für den Aufruf der Komponenten `PrintDialog` und `PageSetupDialog` sowie für Ausdrucke benötigt wird. Den oberen Bereich des Formulars nehmen ein für die mehrzeilige Textanzeige mit Bildlauf eingerichtetes `TextBox`-Steuerelement (links) und ein `PictureBox`-Steuerelement für die Anzeige von Bildern (rechts) ein.

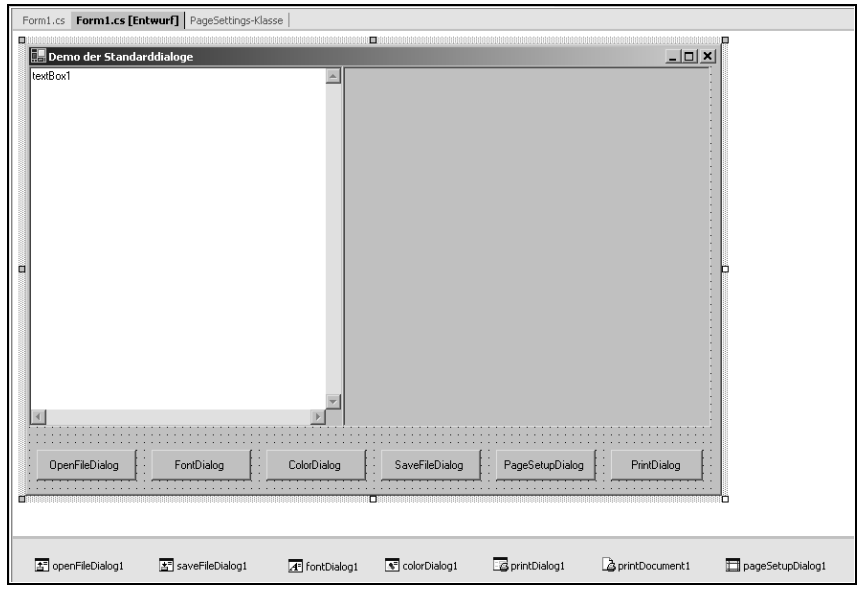


Abbildung 17.4: Formularentwurf für das Projekt *Standarddialoge*

Funktionsumfang und Benutzerschnittstelle

Der Funktionsumfang des Programms ist recht beachtlich, wobei die Wirkung der sechs Schaltflächen weitgehend selbsterklärend und aufeinander abgestimmt ist. Der Benutzer hat die Möglichkeit, wahlweise eine Text- oder Grafikdatei zu öffnen, um sie im Text- bzw. Bildfeld anzuzeigen. Text im Textfeld lässt sich nicht nur bearbeiten und abspeichern, sondern in den Schrift- und Farbeinstellungen verändern und ausdrucken. Schrift- und Farbeinstellungen werden allerdings nicht abgespeichert.

Die Schaltfläche OPENFILEDIALOG ruft den Dialog *Öffnen* auf (Abbildung 17.5). Je nachdem, welchen Filter der Benutzer einstellt, kann er darin eine Text- oder eine Bilddatei auswählen, deren Anzeige dann im Textfeld bzw. Bildfeld des Formulars erfolgt (vgl. Abbildung 17.6).

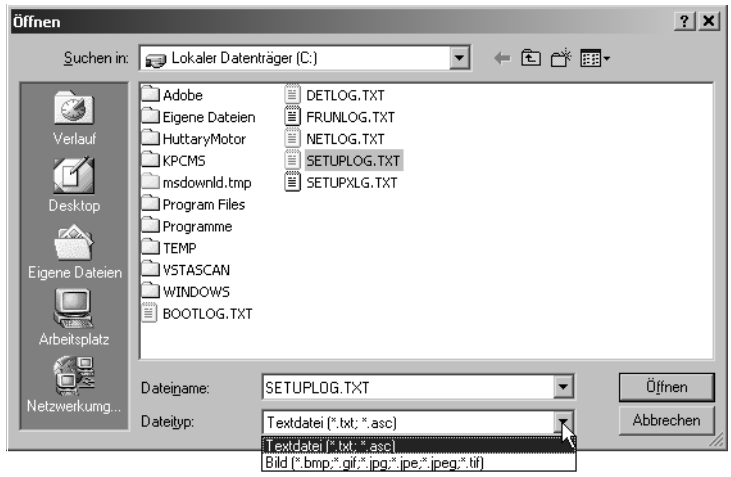


Abbildung 17.5: Dateiauswahl mit der Komponente *FileOpenDialog*

Die Schaltfläche FONTDIALOG ruft den Dialog SCHRIFTART auf den Plan (vgl. Abbildung 17.6). Der Benutzer kann darüber die `Font`-Eigenschaft des `TextBox`-Steuerelements einstellen und die Wirkung seiner Einstellung auch gleich testen, wenn er die Schaltfläche ÜBERNEHMEN anklickt.

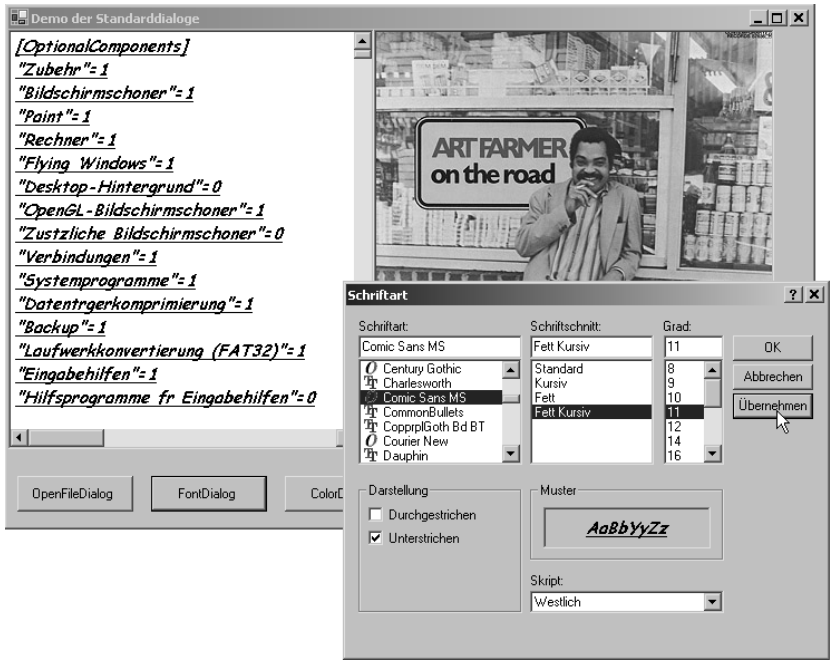


Abbildung 17.6: Einstellen der Schriftattribute für das Textfeld über den Standarddialog SCHRIFTART. Die Schaltfläche ÜBERNEHMEN macht das Ergebnis sichtbar, obwohl der Dialog noch offen ist.

Die Schaltfläche COLORDIALOG zeigt den Dialog FARBEN an und ermöglicht es, die Schriftfarbe für das Textfeld einzustellen. Abbildung 17.7 zeigt den auf die volle Größe entfalteten Dialog und seine Wirkung auf das Textfeld.

Da der Dialog keine ÜBERNEHMEN-Schaltfläche besitzt, muss er erst geschlossen werden, um die Wirkung der Farbauswahl auf das Textfeld zu sehen – für die Abbildung wurde er deshalb erneut geöffnet.

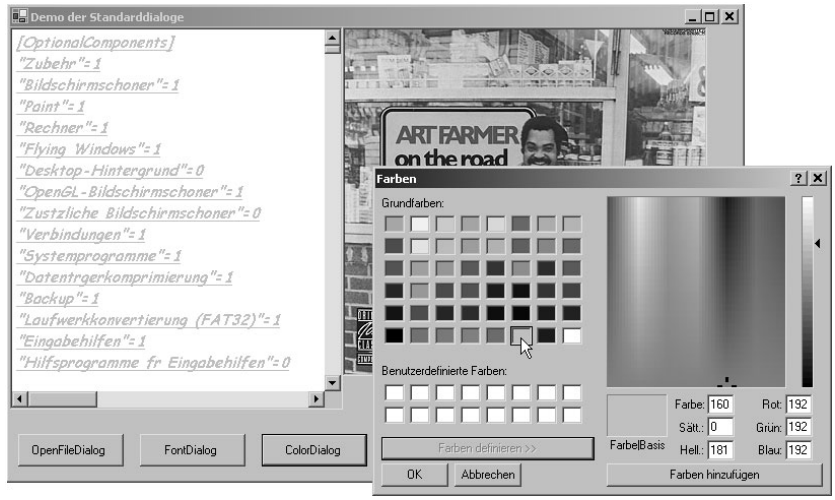


Abbildung 17.7: Einstellen der Schriftfarbe für das Textfeld über den Standarddialog FARBE

Das Textfeld ist editierbar. Der Benutzer kann darin gelesene Texte ändern bzw. eigene Texte schreiben und über die Schaltfläche SAVEFILEDIALOG wiederum als Textdatei speichern; Schriftattribute und Farbe gehen dabei natürlich verloren. Abbildung 17.8 zeigt das Fenster der Anwendung mit einem kleinen Text und davor den Dialog DATEI SPEICHERN UNTER, der nachfragt, ob die neue Datei angelegt werden soll. Beachten Sie, dass der eingegebene Dateiname keine Erweiterung enthält, diese aber automatisch vom Dialog ergänzt wird.

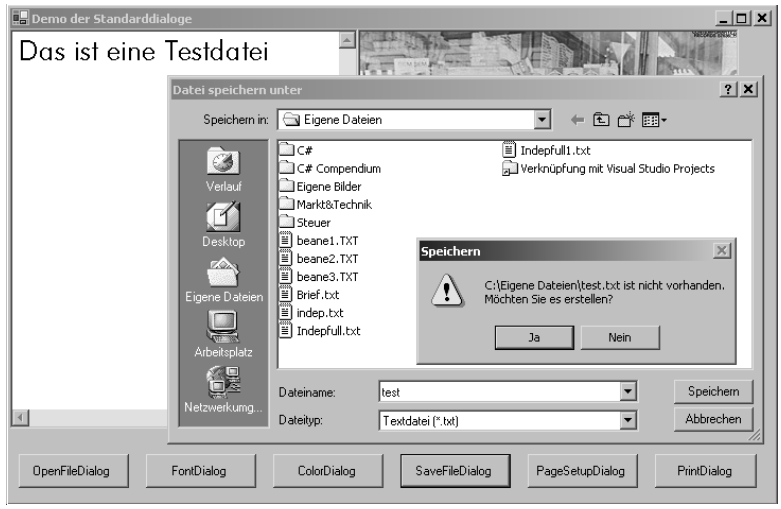


Abbildung 17.8: Speichern des Textfeldinhalts als Textdatei. Die Komponente *FileSaveDialog* ist für die Auswahl des Speicherorts und die Eingabe des Dateinamens verantwortlich. Bei gesetzter `CreatePrompt`-Eigenschaft fragt der Dialog von sich aus nach, ob eine neue Datei erstellt werden soll.

Der Dialog SEITE EINRICHTEN, den die PAGESETUPDIALOG-Schaltfläche auf den Bildschirm bringt, ermöglicht verschiedene Einstellungen, die für die Druckerausgabe relevant sind: Papierart und -fach, Hochformat/Querformat, Maße für die unbedruckten Ränder und – über die Schaltfläche DRUCKER – einen im System oder Netzwerk bekannten Drucker.

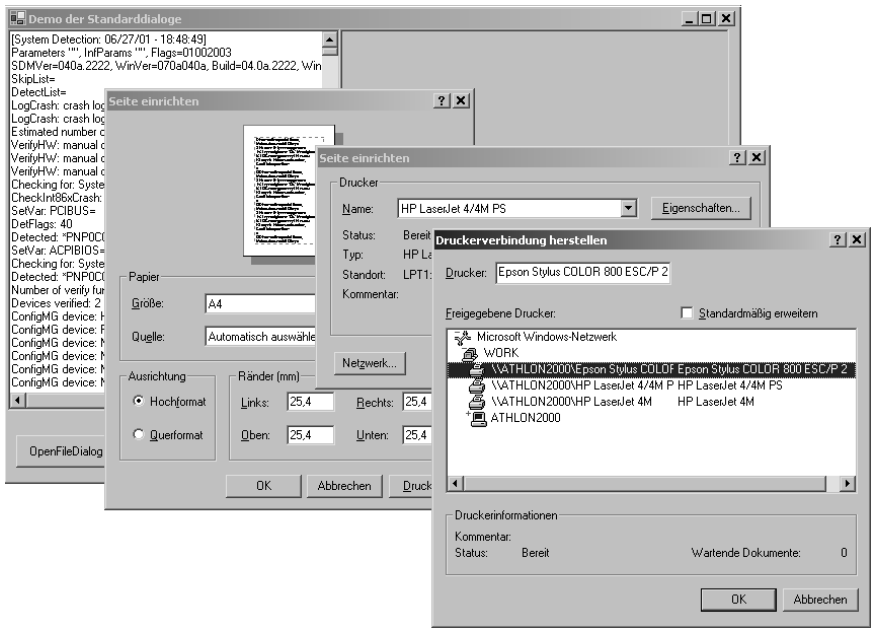


Abbildung 17.9: Der Standarddialog SEITE EINRICHTEN führt auch zur Auswahl des Druckers und der Druckerverbindung über das Netzwerk.

Die letzte der sechs Schaltflächen, PRINTDIALOG, ruft den Dialog DRUCKEN auf, über den ein Ausdruck des Textfeldinhalts in der gewählten Schrift und Farbe möglich ist. Der Ausdruck ist mehrseitig und berücksichtigt die im Dialog SEITE EINRICHTEN gewählte Seitenausrichtung sowie Ränder.

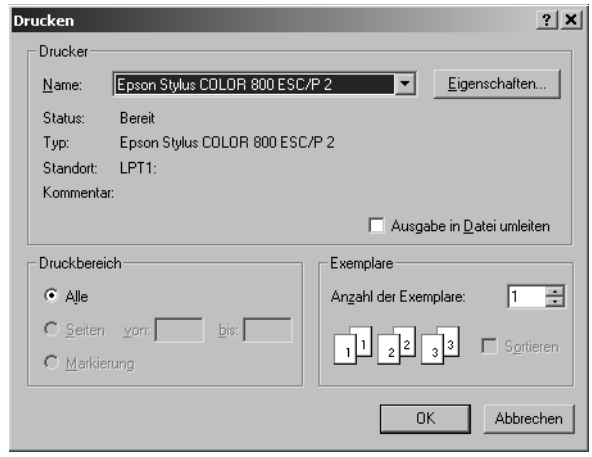


Abbildung 17.10: Der Standarddialog DRUCKEN spiegelt unter anderem die im Dialog SEITE EINRICHTEN getroffenen Druckereinstellungen wider.

Werfen Sie nun einen Blick auf den Code, der all dies hervorbringt. Es versteht sich, dass alle Komponenten und Steuerelemente mit dem Designer eingefügt und auch deren Ereignisbehandlung damit »verdrahtet« wurden. Der größte Teil des Codes steckt daher bereits im Codegerüst und wird hier nicht mehr weiter besprochen.

Der restliche (und in den folgenden Abschnitten aufgelistete) Code zerfällt schlicht in die Funktionalitäten der sechs Schaltflächen.

OpenFileDialog

Die Ausstattung und auch der Umgang mit der Komponente `OpenFileDialog` wurden bereits exemplarisch vorgestellt. Vor dem Aufruf der `ShowDialog()`-Methode generiert `buttonOpenFile_Click` zwei Filter: einen für Textdateien und einen für Bilddateien, wobei jeder Filter mehr als ein Format anzeigt. Nach dem Aufruf bestimmt die Methode anhand des eingestellten Filters, welche Dateiart der Benutzer gewählt hat und liest entweder die Textdatei en bloc in das Textfeld ein oder die Bilddatei in eine neue Bitmap für die `Image`-Eigenschaft des Bildfelds.

```
using System.ComponentModel;
...
private string TextFileName;

private void buttonOpenFile_Click(object sender, System.EventArgs e)
{
  string filter = "Textdatei (*.txt; *.asc)|*.txt;*.asc";
  filter += " | Bild (*.bmp;*.gif;*.jpg;*.jpe;*.jpeg;*.tif) | "
            + *.bmp;*.gif;*.jpg;*.jpe;*.jpeg;*.tif";
  this.openFileDialog1.Filter = filter;
  if (this.openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    switch (this.openFileDialog1.FilterIndex) // FilterIndex ist 1-basiert
    {
      case 1:     // Textdatei lesen
        System.IO.StreamReader sr = new System.IO.StreamReader(
                                      this.openFileDialog1.OpenFile());
        textBox1.Text=sr.ReadToEnd();
        sr.Close();
        break;
      case 2:     // Bild einlesen
        this.pictureBox1.Image = new Bitmap(this.openFileDialog1.FileName);
        break;
    }
  }
}
```

Um die Sache etwas »konservativer« zu gestalten, behandelt der Code das von der Komponente generierte `FileOk`-Ereignis dahingehend, dass er nur Dateinamen akzeptiert, deren Pfad nicht mit der Zeichenfolge `"\\"` beginnt. Wird das Ereignisfeld `Cancel` in der Behandlungsmethode auf `true` gesetzt,

verhält sich die Komponente schlicht so, als habe der Benutzer nicht auf OK geklickt, d.h. sie setzt die Dateiauswahl fort:

```
private void openFileDialog1_FileOk(object sender,
System.ComponentModel.CancelEventArgs e)
{
  if (this.openFileDialog1.FileName.StartsWith("\\\\"))  // Netzwerkpfad?
  {
    MessageBox.Show("Datei muss auf auf lokalem Datenträger liegen" ,
                    "Fehler");
    e.Cancel = true;
  }
}
```

FontDialog

Die Komponente `FontDialog` verfügt über eine Reihe von Eigenschaften, über die sich die Anzahl der angebotenen Schriften und Schriftschnitte genauer vorgeben lässt. In der Regel werden Sie hier aber mit den Standardvorgaben bestens auskommen. Die Eigenschaft `Font` lässt sich hier guten Gewissens als Ein-/Ausgabeparameter der Komponente bezeichnen.

```
private void buttonFont_Click(object sender, System.EventArgs e)
{
  this.fontDialog1.Font = this.textBox1.Font;
  this.fontDialog1.ShowApply = true;    // "Übernehmen" anzeigen
  if (this.fontDialog1.ShowDialog() == DialogResult.OK)
    this.textBox1.Font = this.fontDialog1.Font;
}
```

Interessant ist allerdings die Eigenschaft `ShowApply`. Ist sie beim `ShowDialog()`-Aufruf auf `true` gesetzt, zeigt das Dialogfeld die Schaltfläche ÜBERNEHMEN an und generiert als Reaktion darauf das Ereignis `Apply` – ohne `ShowDialog()` zu beenden. Die Behandlungsmethode aktualisiert in der Regel die Anzeige und wertet dafür die `Font`-Eigenschaft der Komponente aus.

```
private void fontDialog1_Apply(object sender, System.EventArgs e)
{
  this.textBox1.Font = this.fontDialog1.Font;
}
```

ColorDialog

Das Formular der Komponente `ColorDialog` hat zwei Gestalten: Eine kompakte Ansicht, in der nur die 48 Farben der Systempalette sowie die 16 benutzerdefinierbaren Farben angezeigt werden, und eine erweiterte Ansicht, die zusätzlich auch die Auswahl aus dem »kontinuierlichen« TrueColor-Farbraum ermöglicht. Die benutzerdefinierten Farben lassen sich über den Dialog interaktiv definieren sowie als Array en bloc der Eigenschaft `CustomColors` zuweisen. Beachten Sie, dass die Eigenschaft eine Kopie des Arrays anfertigt und das Setzen einzelner Array-Elemente keine Veränderungen bewirkt.

```
this.colorDialog1.CustomColors = new int[16]{6916092, 15195440, 16107657,
                                             1836924,  3758726, 12566463,
                                             7526079, 7405793, 6945974,
                                             241502, 2296476, 5130294,
                                             3102017, 7324121, 14993507,
                                             11730944
                                            };
```

In der Beispielanwendung sieht der Code für die Komponente `ColorDialog` wahrlich harmlos aus:

```
private void buttonColor_Click(object sender, System.EventArgs e)
{
  this.colorDialog1.Color = this.textBox1.ForeColor;
  if (this.colorDialog1.ShowDialog() == DialogResult.OK)
    textBox1.ForeColor  = this.colorDialog1.Color;
}
```

SaveFileDialog

Auch der Code für die Anzeige der Komponente `SaveFileDialog` enthält keine Spezialitäten. Er gibt einen Dateinamen, eine Standarderweiterung und einen Filter vor und weist den Dialog an, den Benutzer neue Dateinamen per Meldungsdialog quittieren zu lassen. Die Ausgabe selbst erfolgt dann ohne weiteren Aufwand über die `Write()`-Methode eines unter Angabe dieses Dateinamens geöffneten `StreamWriter`-Objekts.

```
private void buttonSaveFile_Click(object sender, System.EventArgs e)
{
  this.saveFileDialog1.FileName = this.TextFileName;
  this.saveFileDialog1.CreatePrompt = true; // Bei neuer Datei nachfragen
  this.saveFileDialog1.DefaultExt = "txt";  // Erweiterung .txt vorgeben
  this.saveFileDialog1.AddExtension = true; // Erweiterung ggf. ergänzen
  this.saveFileDialog1.Filter = "Textdatei (*.txt)|*.txt";
  if (this.saveFileDialog1.ShowDialog() == DialogResult.OK) // Dialog
  {
    // Ausgabedatei öffnen
    System.IO.StreamWriter tw =
      new System.IO.StreamWriter(this.saveFileDialog1.FileName);
    tw.Write(textBox1.Text);   // Text von textBox1 en bloc schreiben
    tw.Close();
    this.TextFileName = this.saveFileDialog1.FileName;  // Namen merken
  }
}
```

Druckvorgang

Der restliche Code des Programms hängt im weiteren und engeren Sinne mit dem Druckvorgang zusammen. Der Standarddialog SEITE EINRICHTEN ermöglicht die Einstellung der Ränder für das Druckdokument, der Standarddialog DRUCKEN wickelt den Druckvorgang an sich ab. Gedruckt wird der gesamte Inhalt des Textfelds – in der eingestellten Farbe, mit der eingestellten Schrift und linksbündig mit Flatterrand. Dazu nimmt der Code

einen rudimentären Zeilen- und Seitenumbruch vor, weshalb gegebenenfalls auch mehrere Seiten herauskommen. Seien sie also vorsichtig, was Sie versuchshalber drucken.

PageSetupDialog und PrintDialog

Das Szenario für den Einsatz von `PageSetupDialog` und `PrintDialog` ist ein wenig komplexer als das der bisher vorgestellten Dialoge, weil diese Komponenten auf Basis einer `PrintDocument`-Komponente operieren, die vor dem Aufruf der jeweiligen `ShowDialog()`-Methode an die Eigenschaft `Document` gebunden werden muss. Wie erwartet, zeigt das `PageSetupDialog`-Objekt nicht nur die für das `PrintDocument`-Objekt geltenden und in der Eigenschaft `DefaultPageSettings` gespeicherten globalen Seiteneinstellungen (Drucker, Seitenausrichtung und Ränder) an, sondern schreibt bestätigte Änderungswünsche des Benutzers auch wieder automatisch dahin zurück.



Dummerweise wurde bei der Lokalisierung der `PageSetupDialog`*-Komponente etwas geschlampt. Es ist mal wieder eine fehlende Umrechnung zwischen Inch und Millimeter daran schuld – der ja angeblich auch die Marssonde Polar Lander der NASA zum Opfer gefallen sein soll. Obwohl die grundlegende Einheit aller Seiteneinstellungen eines Druckdokuments das 100stel Zoll ist, zeigt die deutsche Variante der Komponente zumindest in der bei Drucklegung des Buchs aktuellen .NET-Framework-Version v1.0.3705 die* `Margins`*-Werte der Eigenschaft* `PageSettings` *ohne weitere Umrechnung als Millimeterwerte an, rechnet dann aber die nach Bestätigung des Dialogs eingesammelten Werte brav von Millimeter nach Zoll zurück, was – gut sichtbar bei erneutem Aufruf des Dialogs mit unveränderten Einstellungen – den bekannten Faktor 2,54 ins Spiel bringt. Der folgende Code enthält daher einen Workaround dafür.*

Damit ist der Rahmen für den Aufruf der `PageSetupDialog`-Komponente bereits klar:

```
private void buttonPageSetup_Click(object sender, System.EventArgs e)
{
  // NetzwerkSchaltfläche in Dialog Seite einrichten/Drucker
  this.pageSetupDialog1.ShowNetwork = true;
  // Druckdokument zuordnen
  this.pageSetupDialog1.Document = this.printDocument1;
  // Fehler in PageSettingsDialog korrieren: Inch in Millimeter umrechnen
  this.printDocument1.DefaultPageSettings.Margins = InchToMM(
    this.printDocument1.DefaultPageSettings.Margins);
  this.pageSetupDialog1.ShowDialog();
}

private System.Drawing.Printing.Margins InchToMM(
  System.Drawing.Printing.Margins m)
{
  const float Inch = 2.541045f;
  m.Left = (int) (m.Left * Inch);
```

```
  m.Top = (int) (m.Top * Inch);
  m.Bottom = (int) (m.Bottom * Inch);
  m.Right = (int) (m.Right * Inch);
  return m;
}
```

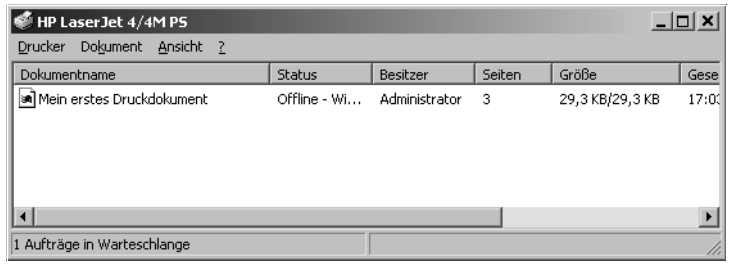
Aufgabe der `PrintDialog`-Komponente ist es, die Parameter für den anstehenden Druckvorgang in Erfahrung zu bringen und grünes Licht für den Start des Ausdrucks zu geben. Damit der Benutzer die Option SEITENBEREICH aktivieren und einzelne Seiten für den Ausdruck angeben kann, muss die Eigenschaft `AllowSomePages` auf `true` gesetzt sein. Die Seitenbereichseinstellungen selbst lassen sich über die `PrinterSettings`-Eigenschaft abfragen und auch vorgeben – freilich erst, nachdem der Komponente das `PrintDocument`-Objekt zugeordnet wurde. Das Codebeispiel verzichtet auf die Möglichkeit einer benutzerseitigen Vorgabe von Seitenbereichen und überlässt die Implementierung dem Leser als Übung.

Die Eigenschaft `DocumentName` legt den Namen fest, unter dem der Ausdruck in der Druckerwarteschlange erscheint (vgl. Abbildung 17.11).

```
private void buttonPrint_Click(object sender, System.EventArgs e)
{
  printDocument1.DocumentName = "Mein erstes Druckdokument";
  this.printDialog1.Document = printDocument1;
  if (this.printDialog1.ShowDialog() == DialogResult.OK)
    this.printDocument1.Print();
}
```

Druckausgabe über die PrintDocument-Komponente

Abbildung 17.11: Das Druckdokument in der Druckerwarteschlange enthält drei Seiten.



Den Ausdruck an sich wickelt das `PrintDocument`-Objekt ab. Dessen Instanzmethode `Print()` generiert vier Ereignisse mit aussagekräftigen Bezeichnern: `BeginPrint`, `QueryPageSettings`, `PrintPage` und `EndPrint`. Eine Behandlung von `BeginPrint` und `EndPrint` ist optional und ermöglicht die Ausführung zusätzlichen Codes, der gegebenenfalls noch vor bzw. nach dem Ausdruck anfällt – etwa eine Nachfrage à la »Wollen Sie wirklich gleich alle 5935 Seiten auf ihrem neuen Proof-Drucker ausbelichten?«. Die Behandlungsmethoden können den Ausdruck abbrechen, indem sie die `Cancel`-Eigenschaft des `PrintEventArgs`-Ereignisobjekts auf `true` setzen. Der Benutzer hat außerdem die

Möglichkeit, den Ausdruck über das Statusfenster der Druckausgabe abzubrechen, das den Fortschritt des Spoolings als Seitenzahl anzeigt (Bild 17.12).

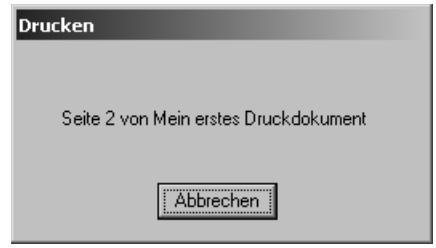


Abbildung 17.12: Statusfenster der Druckausgabe

Der Druckvorgang ist seitenorientiert, wobei `Print()` für jede auszugebende Seite zuerst ein `QueryPageSettings`-Ereignis und dann ein `PrintPage`-Ereignis generiert. Eine Behandlung von `QueryPageSettings` ist nur erforderlich, falls zu erwarten ist, dass die `DefaultPageSettings`-Eigenschaft des `PrintDocument`-Objekts die Seiteneinstellungen für einzelne Druckseiten nicht richtig beschreibt. Das `QueryPageSettingsEventArgs`-Ereignisobjekt liefert das dafür benötigte `PageSettings`-Objekt als Kopie der `DefaultPageSettings`-Eigenschaft. Es geht dann weiter an die für die Anfertigung der einzelnen Druckseite verantwortliche `PrintPage`-Behandlung. Veränderungen des `PageSettings`-Objekts wirken sich nicht auf die `DefaultPageSettings`-Eigenschaft aus, sondern nur auf die aktuelle Druckseite.

Der Seitenaufbau für die Druckausgabe ist das Geschäft der `PrintPage`-Behandlung. Rein logisch passiert hier nichts anderes als in einer `Paint`-Routine: Eine Methode zeichnet in einen Grafikkontext – das heißt: Sie wendet verschiedene Grafikoperationen auf eine vorgegebene Zeichenfläche mit bestimmten Attributen an und benutzt dafür ein Koordinatensystem, Stifte, Pinsel, Schriften und einen Satz von Zeichenmethoden. Der Grafikkontext ist nichts anderes als ein Objekt, das die Zeichenfläche verkapselt und die dazu gehörigen Attribute sowie Grafikoperationen in Form von Eigenschaften und Methoden bereitstellt (vgl. Kapitel 14, »Grafik« (Seite 459ff)). Das Standardkoordinatensystem richtet sich nach der tatsächlichen oder virtuellen Auflösung des jeweiligen Ausgabegeräts. Der Ursprung für eine Druckseite liegt links oben, wobei die *y*-Achse nach unten gerichtet ist. Als voreingestellte Maßeinheit gelten 100stel Zoll.

Angenommen, Sie haben einen Grafikkontext für die Druckausgabe namens `e`. Um ein im Koordinatenursprung beginnendes Quadrat mit der Seitenlänge 10 cm zu zeichnen, schreiben Sie:

```
e.Graphics.DrawRectangle(Pens.Black, 0, 0, 10000 / 2.54f, 10000 / 2.54f);
```

Eleganter ist es natürlich, wenn Sie das Koordinatensystem über die `GraphicsUnit`-Eigenschaft des Gerätekontexts schlicht auf Millimeter umstellen:

```
e.Graphics.PageUnit = GraphicsUnit.Millimeter;
e.Graphics.DrawRectangle(Pens.Black, 0, 0, 100, 100);
```

Die Behandlungsmethode für das `PrintPage`-Ereignis findet in ihrem Parameter `PrintPageEventArgs` also einen fertigen Grafikkontext für den jeweiligen Drucker vor – als `Graphics`-Eigenschaft. Für die string-basierte Textausgabe ist die Methode `DrawString()` zuständig.

Erfreulicherweise versteht diese Methode einen Zeilenvorschub als Aufforderung, unter Einhaltung des für die jeweilige Schrift vordefinierten Standardzeilenabstands eine neue Zeile zu beginnen und kann mehrzeilige Texte somit tatsächlich en bloc ausdrucken, was bei der starken Grafikorientierung der Ausgabe keine Selbstverständlichkeit ist. Bei Angabe eines passend initialisierten `StringFormat`-Objekts sowie eines Begrenzungsrechtecks als Parameter für den Aufruf von `DrawString()` lassen sich aber auch automatische Zeilenumbrüche für die rechts- oder linksbündige oder auch zentrierte Ausgabe erreichen.

Wenn Sie nun aber erwartet haben, dass dieser Automatismus auch gleich für einen Seitenumbruch sorgt – leider nein. Dafür sind weder Mittel vorhanden, noch ist es einem `StringFormat`-Objekt gegeben, von sich aus eine Druckausgabe abzuwickeln. Mit anderen Worten: Alles was über den Zeilenumbruch hinausgeht, fällt in Ihren Aufgabenbereich.

`DrawString()` zeichnet mit einer vorgegebenen Schrift und Farbe in ein Begrenzungsrechteck, das als `Rectangle`-Objekt vorgegeben ist. Enthält eine Zeile zu viele Zeichen oder der gesamte Text zu viele Zeilen, um noch auf die Druckseite bzw. den eingestellten Seitenbereich zu passen, wird gnadenlos abgeschnitten.

Der `Print()`-Aufruf generiert zunächst einmal die Ereignisse für eine Druckseite. Um eine weitere Druckseite auszugeben, muss die `PrintPage`-Behandlung die Eigenschaft `HasMorePages` ihres `PrintPageEventArgs`-Parameters auf `true` setzen. Die `Print()`-Methode generiert dann weitere `QueryPageSettings`- und `PrintPage`-Ereignisse – so lange, bis die `PrintPage`-Behandlung `HasMorePages` irgendwann auf `false` belässt. Gleichzeitig sieht der Benutzer den bekannten `Drucken`-Dialog, der die durch die Druckerwarteschlange übertragenen Druckseiten mitzählt (Abbildung 17.12).

Offensichtlich ist die Schriftausgabe in einen Grafikkontext ist ein eigenes und nicht gerade einfaches Kapitel, so dass es wenig Sinn macht, an dieser Stelle beliebig weit ins Detail zu gehen. (Sie finden die notwendigen Details aber im Abschnitt »Textausgabe – DrawString()«, Seite 527, sowie in dem dazugehörigen Beispielprojekt *Blocksatz*.) Der hier verwendete Code macht sich zumindest die Mühe, den Text aus dem Textfeld linksbündig im Flattersatz auszugeben und einen vernünftigen Seitenumbruch zu erreichen.

Den Löwenanteil der Arbeit bei der Berechnung des Seitenumbruchs erledigt die (leider recht unselige) `MeasureString()`-Methode des Grafikkontexts. Sie berechnet die Anzahl der Zeilen und die Anzahl der Zeichen für die aktuelle Druckseite. Da `MeasureString()` die Tendenz hat, bis zur nächsten ganzen Zeile aufzurunden, sollte die Höhe des Druckbereichs bereits zuvor auf ein ganzes Vielfaches der Zeilenhöhe abgerundet werden. Über die Zeichenanzahl lässt sich der Startindex für die jeweils nächste Druckseite berechnen. `printDocument1_PrintPage()` hinterlegt diesen Wert im Datenfeld `printStartIndex` der Formularklasse und setzt die `HasMorePages`-Eigenschaft des `PrintPageEventArgs`-Ereignisobjekts auf `true`, falls sich über den Startindex erkennen lässt, dass noch nicht alle Zeichen gedruckt sind.

Einen weiteren Workaround erfordert die Tatsache, dass die Methode `MeasureString()` im Zusammenspiel mit dem `StringFormat`-Objekt an sich zwar einen leerzeichenbasierten Zeilenumbruch berechnet, dummerweise jedoch nicht für die letzte Zeile. Diese bricht sie aus unerfindlichen Gründen mitten im Wort um, weshalb gegebenenfalls eine Korrektur der Anzahl der für die Seite auszugebenden Zeichen erforderlich wird, wenn die Zeile nicht zufällig mit einem Zeilenvorschub oder einem Leerzeichen endet. Der Workaround sieht so aus, dass der Code von sich aus das letzte Leerzeichen sucht und mit dem gefundenen Index die Anzahl korrigiert. Der Übersichtlichkeit halber nimmt die vorliegende Implementierung zwar eine Prüfung vor, *ob* ein Leerzeichen gefunden wurde, verzichtet aber auf eine weitere Analyse, ob dieses tatsächlich in der letzten Zeile sitzt. `DrawString()` gibt die Seite letztlich also auch nur bis zu diesem Leerzeichen aus – egal, wo es steht.

```
private int printStartIndex = 0;

private void printDocument1_PrintPage(object sender,
  System.Drawing.Printing.PrintPageEventArgs e)
{
  Brush PrintBrush = new SolidBrush(textBox1.ForeColor);
  StringFormat sf = new StringFormat();
  // Höhe des Ausgabebereichs auf ganze Zeilenanzahl verkürzen
  RectangleF box = e.MarginBounds;
  float fontHeight = textBox1.Font.GetHeight(e.Graphics);

  // Höhe des Ausgabebereichs auf ganze Zeilen abrunden
  box.Height = (int) fontHeight * (int)(box.Height / fontHeight);

  // Anzahl der auszugebenden Zeichen berechnen
  int linesDummy;
  int charsToPrint;

  box.Size =
   e.Graphics.MeasureString(textBox1.Text.Substring(printStartIndex),
     textBox1.Font, box.Size, sf, out charsToPrint, out linesDummy);

  // Seitenumbruch sollte auch bei Leerzeichen stattfinden.
  // (MeasureString() bricht in der letzten Zeile leider mitten im
  // Wort um.)
  string printString =
    textBox1.Text.Substring(printStartIndex, charsToPrint);
```

```
  if (!printString.EndsWith("\n"))  // Zeilenvorschub? dann OK.
  {
    int lastWhiteSpace = printString.LastIndexOf(" ");
    if  (lastWhiteSpace > 0)
      charsToPrint =  lastWhiteSpace + 1;
  }

  // Eine Seite zeichnen
  e.Graphics.DrawString(textBox1.Text.Substring(printStartIndex,
    charsToPrint), textBox1.Font, PrintBrush, e.MarginBounds);
  printStartIndex += charsToPrint;

  // Weitere Seiten?
  if (printStartIndex < textBox1.Text.Length -1)
    e.HasMorePages = true;
  else
    printStartIndex = 0;                          // für nächsten Ausdruck
}
```

Übung

Erweitern Sie den Funktionsumfang des Beispielprojekts *Standarddialoge* dahingehend, dass es

1. die Auswahl mehrerer Text- und Bilddateien im Rahmen eines einzigen ÖFFNEN-Aufrufs ermöglicht
2. diese Dateien im Text- bzw. Bildfeld nacheinander und zyklisch anzeigt. Implementieren sie wahlweise eine Maus- oder Tastaturschnittstelle für die Weiterschaltung der Anzeige.
3. eine Seitenbereichsauswahl im DRUCKEN-Dialog ermöglicht und bei der Druckausgabe berücksichtigt.

17.1.3 Eigene modale Dialoge

Das Aufrufschema für selbst entworfene modale Dialoge sieht genauso aus wie das für die Standarddialoge. Neben den eigentlichen Dialogelementen fügen Sie Schaltflächen ein – am besten mit den gewohnten Beschriftungen OK, ABBRECHEN, JA, NEIN, usw. –, über die der Benutzer den Dialog verlassen kann, und sorgen dafür, dass jede Schaltfläche einen spezifischen `DialogResult`-Wert zurückgibt, anhand dessen der Aufrufer feststellen kann, wie der Dialog beendet wurde.

Rückgabewert des Dialogs

Nachdem ein Formular nicht per `return`-Anweisung, sondern durch einen (expliziten oder impliziten) Aufruf der `Close()`-Methode geschlossen wird, ist der `ShowDialog()`-Aufruf so definiert, dass er den Wert der `DialogResult`-Eigenschaft des Formularobjekts als Funktionswert liefert. Die Wertrückgabe für eine Schaltfläche WIEDERHOLEN sieht demnach so aus:

```
private void buttonRetry_Click(object sender, System.EventArgs e)
{
  this.DialogResult = DialogResult.Retry;
  this.Close();
}
```

Vom Prinzip her können Sie natürlich auch jedes andere Ereignis, etwa einen Mausklick in das Formular, auf diese Weise behandeln. Ob es jedoch sinnvoll ist, die Erwartungshaltung des Benutzers auf diese Weise zu brüskieren, ist eine andere Frage.

Zur Erwartungshaltung gehört auch, dass es der Benutzer in modalen Dialogen gewohnt ist:

- eine stärker umrahmte Standardschaltfläche vorzufinden, über die er den Dialog per ↵ verlassen kann
- den Dialog jederzeit mit Esc abbrechen zu können.

Die Implementierung dieses Standardverhaltens ist vonseiten des `Form`-Objekts gut vorbereitet. Sie brauchen nichts weiter zu tun, als dem Formularobjekt die beiden für diese Fälle vorgesehenen Schaltflächen bekannt zu machen:

```
public MyDialog()
{
  InitializeComponent();
  this.AcceptButton = buttonOK;
  this.CancelButton = buttonCancel;
  ...
```

Den Rest, das heißt, die Überwachung der Tastaturschnittstelle, das Auslösen der Ereignisse (deren Behandlung nun aber nicht mehr zwingend ist) und das Setzen der `DialogResult`-Eigenschaft auf die Werte `DialogResult.OK` bzw. `DialogResult.Cancel` übernimmt das Basisklassenobjekt des Formulars – Vererbung ist doch ein Segen.

Ausgestaltung des Dialogs

Da modale Dialoge die Aufmerksamkeit des Benutzers ganz auf sich lenken wollen, erscheinen sie passenderweise in der Mitte des Bildschirms (`CenterScreen`) oder zumindest des Anwendungsfensters (`CenterParent`). Zudem verwenden sie meist einen einfacheren Fensterstil mit festen Abmessungen, wobei die Titelleiste lediglich eine Schließbox (und kein Systemmenü) enthält. Diese Einstellungen lassen sich über die entsprechenden `Form`-Eigenschaften bereits bei der Formulargestaltung interaktiv im Designer vornehmen. Hier die Codeseite:

```
private void InitializeComponent()
{
  ...
  this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedSingle;
  this.MaximizeBox = false;
  this.MinimizeBox = false;
  this.Name = "myDialog";
  this.ShowInTaskbar = false;
  this.Text = "Modaler Dialog";
  this.StartPosition = System.Windows.Forms.FormStartPosition.CenterParent;
```

Der rigideste Rahmenstil ist `FixedDialog`. Er wird nicht nur für Dialoge, sondern auch als Container für Werkzeugleisten, schwebende Symbolleisten usw. gerne verwendet und liefert ein nicht skalierbares Fenster ohne Systemmenü.

Eigenschaftsdialoge und allgemeine Dialoge

Typische modale Dialoge sind Eigenschaftsdialoge, in denen der Benutzer Eigenschaften eines oder mehrerer Objekte setzen kann – der besseren Verständlichkeit halber geht die weitere Diskussion von einem einzelnen Objekt aus. Die Grenzen sind hier allerdings fließend und es ist ohne weiteres vertretbar, auch OPTIONEN- und EINSTELLUNGEN-Dialoge, die eine Auswahl an Eigenschaften vieler Objekte präsentieren, zu den Eigenschaftsdialogen zu rechnen – denken Sie dabei einfach an die Standarddialoge.

Um die aktuellen Werte (und Wertebereiche) der Eigenschaften des Objekts übersichtlich zu präsentieren, enthält der Formularentwurf eines Eigenschaftsdialogs ein Sammelsurium geeignet gewählter und gruppierter Steuerelemente. Der Aufrufer des Eigenschaftsdialogs (häufig das Objekt selbst, das dafür eine Methode bereitstellt) hat dafür Sorge zu tragen, dass diese Steuerelemente entweder noch *vor* dem `ShowDialog()`-Aufruf, spätestens aber durch den Konstruktor des Formularobjekts eine geeignete Initialisierung erfahren. Schließlich sollen sie den aktuellen Zustand des Objekts widerspiegeln.

Techniken für die Wertübergabe und -übernahme

Wie bereits angeklungen, bieten sich dafür zwei Techniken (aber auch Mischformen derselben) an, die Sie danach auswählen, wie viel der Eigenschaftsdialog über den potenziellen Aufrufer weiß.

Dialog verwaltet das Objekt eigenständig

Ist der Dialog dem Aufrufer mehr oder weniger auf den Leib geschrieben, kann das Formularobjekt eine Eigenschaft mit dem Datentyp des Objekts (oder eben eines Objekts, das zumindest die Eigenschaften besitzt, um die es geht) offen legen. Der Aufrufer ordnet dieser Eigenschaft dann einfach das Objekt vor dem `ShowDialog()`-Aufruf zu und verlässt sich darauf, dass das Formularobjekt alles Weitere erledigt. Nach Rückgabe der Kontrolle findet

er dann entweder ein verändertes Objekt als Wert der Eigenschaft vor – oder eben nicht, wenn der Benutzer keine Änderungen vorgenommen bzw. den Dialog abgebrochen hat. Prominente Beispiele dafür wären die Standarddialoge `PageSetupDialog` und `PrintDialog`.

```
// Im Quelltext des Aufrufers
// Wertübergabe: Setzen der als Aufrufschnittstelle fungierenden
// Eigenschaft auf das betroffene Objekt.
myDialog.ObjectToChange = this.objectToChange;
DialogResult res = myDialog.ShowDialog();
if (res == DialogResult.OK)
{
  Invalidate();     // Objekt ist bereits manipuliert
}
// Aufrufer arbeitet mit dem (ggf. veränderten) Objekt objectToChange
// weiter, als sei nichts passiert.
```

Aufrufer verwaltet das Objekt

Ist der Dialog so allgemein, dass er vielen höchst unterschiedlich gearteten Aufrufern dienlich sein kann, wird er eine Schnittstelle in Form einer Sammlung von Eigenschaften mit allgemeinen Datentypen offen legen (`public`). Der Aufrufer trägt dafür Sorge, dass er diese Eigenschaften vor dem Aufruf setzt und nach dem Aufruf – in Abhängigkeit vom Ergebnis des Aufrufs – auswertet. Der Dialog initialisiert seine Steuerelemente über die `set`-Accessoren dieser Eigenschaften und liest die Benutzereingaben über die `get`-Accessoren wieder aus. Prominente Beispiele dafür wären die Standarddialoge `OpenFileDialog` und `SaveFileDialog`.

```
// Im Quelltext des Aufrufers
... // Wertübergabe: Setzen der Dialogeigenschaften
do
  if (myDialog.ShowDialog() == DialogResult.Cancel)
    break;
while (ApplyChangesToObject() == false)
...

private bool ApplyChangesToObject()
{
  ...  // Wertübernahme: Auslesen der Dialogeigenschaften
       // Gültigkeitsfehler gegebenenfalls melden
```

Auch wenn Ihnen der Unterschied auf den ersten Blick nicht so gravierend erscheinen mag, so lebt er doch von dem Dilemma, wem die Verantwortung für die verbindliche Übernahme der Benutzereingaben zufällt – dem Aufrufer oder dem Dialog. Dementsprechend findet auch die Gültigkeitsprüfung im ersten Fall durch das Dialogobjekt (hier in direkter Reaktion auf die Ereignisse der Steuerelemente) und im zweiten Fall durch den Aufrufer statt – eben da, wo das Objekt selbst manipuliert wird.

Schaltfläche Übernehmen – Gültigkeitsprüfung und Rückruf

Vielfach ist es erwünscht, dass der Benutzer noch im Eigenschaftsdialog die Wirkung seiner Eingaben sehen (oder sonstwie erfahren) kann – falls er sie mangels Visualisierungsmöglichkeit nicht direkt zu Gesicht bekommt, erübrigt sich die vorgestellte Technik. Wenn es sich bei dem bearbeiteten Objekt nicht gerade um ein `Control`-Objekt handelt, das selbst über eine `Paint`-Routine verfügt, ist der Aufrufer für die Darstellung des Objekts verantwortlich. Ein prominenter Vertreter eines solchen Dialogs ist der Standarddialog `FontDialog`.

Zur Einleitung eines noch während der modalen Anzeige stattfindenden Zeichenvorgangs bietet ein Dialog im Allgemeinen eine Schaltfläche namens ÜBERNEHMEN an, derer sich der Benutzer jederzeit bedienen kann – sinnvollerweise aber nur, wenn er Eingaben vorgenommen hat. Auch in diesem Zusammenhang sind wieder die beiden bereits vorgestellten Szenarien zu unterscheiden.

Dialog verwaltet das Objekt

Falls der Dialog für die Übernahme der Eingaben verantwortlich ist, wird dieser bei der Behandlung des zugehörigen `Click`-Ereignisses das Objekt entsprechend der Benutzereingaben manipulieren und dann den Aufrufer per Ereignis benachrichtigen, dass er es zeichnen soll. Erkennt der Dialog dabei aber Gültigkeitsverletzungen, unterbleibt der Rückruf vorerst, bis der Benutzer korrekte Eingaben gemacht hat. Die OK-Behandlung muss bei Gültigkeitsverletzungen noch drastischere Maßnahmen ergreifen: Sie muss verhindern, dass der Dialog beendet wird. Das Mittel dafür ist eine `Closing`-Behandlung.

```
// im Quelltext der Dialogklasse MyDialog
public event EventHandler Applied;  // Rückruf
private bool noClose = false;
private void buttonApply_Click(object sender, System.EventArgs e)
{
  if (ApplyChangesToObject())
    OnApplied(e);
}

private void buttonOK_Click(object sender, System.EventArgs e)
{
  if (ApplyChangesToObject())
    OnApplied(e);
  else
    noClose = true;
}
```

```
private MyDialog_Closing(object sender,
  System.ComponentModel.CancelEventArgs e)
{
  if (noClose)
    e.Cancel = noClose = false;
}

protected virtual void OnApplied(EventArgs e)
{
  if (Applied != null)
    Applied(this, e);
}
private bool ApplyChangesToObject()
{
  ...
}
//************************************************************
// im Quelltext des Aufrufers
...
MyDialog myDialog = new MyDialog();
Applied += new EventHandler(this.myDialog_Applied);
...
myDialog.ObjectToChange = this.objectToChange;  // Wertübergabe
if (myDialog.ShowDialog() == DialogResult.OK)
{
  myDialog_Applied(null, null);
}

private void myDialog_Applied(object sender, System.EventArgs e)
{
  ...  // Wertübernahme: Verwerten der Eigenschaft(en)
  Refresh();
}
```

Aufrufer verwaltet das Objekt

Im anderen Fall signalisiert der Dialog dem Aufrufer, dass er die Eingaben übernehmen soll, um es dann zu zeichnen. Damit fällt der Schwarze Peter der Gültigkeitsprüfung dem Aufrufer zu, und dieser hat nur ein Mittel, sich über Fehleingaben zu beschweren: Er kann einen weiteren modalen Dialog aufrufen. Erlaubt ist hier alles, was modal ist, beispielsweise eine `MessageBox`-Meldung oder ein eigener Dialog.

```
// im Quelltext der Dialogklasse MyDialog
public event EventHandler Apply;  // Rückruf
private void buttonApply_Click(object sender, System.EventArgs e)
{
    OnApply(e);
}
protected virtual void OnApply(System.EventArgs e)
{
  if (Apply!= null)
    Apply(this, e);
}
```

```
// ****************************************************
// im Quelltext des Aufrufers
...
MyDialog myDialog = new MyDialog();
Apply += new EventHandler(this.myDialog_Apply);
...
do                                              // Aufruf des Dialogs
  if (myDialog.ShowDialog() == DialogResult.Cancel)
    break;
while (this.ApplyChangesToObject() == false)    // Eingabe ungültig?
...

private void myDialog_Apply(object sender, System.EventArgs e)
{
  if (ApplyChangesToObject())   // Werte übernehmen, wenn gültig
    Refresh();                  // anzeigen
}
private bool ApplyChangesToObject()
{
  ...     // Gültigkeitsprüfung
  if(error)
  {
    MessageBox.Show("Fehler: Eingabe ... ist ungültig");
    return false;
  }
  return true;
}
```

17.2 Nicht-modale Dialoge

Wie bereits zu Beginn dieses Kapitels angesprochen, lässt sich grundsätzlich jedes mit `Show()` oder `myForm.Visible = true` aufgerufene Formular als nicht-modaler Dialog begreifen.

Im engeren Sinne versteht man unter einem NICHT-MODALEN DIALOG ein Formular, das gleichberechtigt neben anderen Formularen der Anwendung angezeigt wird und dem Benutzer grundsätzlich die Wahl lässt, in welchem Formular er Eingaben vornehmen möchte.

Das ist aber noch nicht alles. Im Allgemeinen gilt für nicht-modale Dialoge die »Berichtspflicht«, das heißt, sie müssen ihren Besitzer durch Signalisierung entsprechender Ereignisse über Änderungen ihres Zustands informieren. Der Spielraum dafür, wann sich ein nicht-modales Formular beim Besitzer meldet, ist groß. Vom Prinzip her könnte es jedes vom Benutzer verursachte oder auch sonstige Ereignis (beispielsweise Timer-Ticks) melden, meist sind es aber nur `Click`-Ereignisse auf Schaltflächen wie SUCHEN oder ÜBERNEHMEN oder Klicks auf Symbole (Symbolleisten in eigenen Fenstern), für die ein Rückruf erfolgt.

Da für nicht-modale Dialoge im Allgemeinen keine Verkapselung notwendig ist, kann der Dialog die entsprechenden Steuerelemente einfach als `public` vereinbaren, um dem Besitzer die Möglichkeit zu geben, seine Behandlungsroutinen gleich an Ort und Stelle einzuklinken:

```
// Im Quelltext des Besitzers eines Suchen/Ersetzen-Formulars searchForm
searchForm.buttonSearch.Click += new EventHandler(this.search_Click);
searchForm.buttonReplace.Click += new EventHandler(this.replace_Click);
searchForm.buttonReplaceAll.Click  += new
EventHandler(this.replaceAll_Click);
searchForm.Closed += new EventHandler(searchForm_Closed);
```

Dies ist die effektivste Programmiertechnik. Für die Implementierung ist es fast transparent, auf welchem Formular die Steuerelemente sitzen, deren Ereignisse behandelt werden müssen. An Stelle von `this` wird hier eben das Formular `searchForm` als Besitzer eingesetzt.

Im Einzelfall kann es natürlich nötig sein, dass ein Dialog auch eigene Ereignisse generiert. Auch das ist *business as usual*:

```
// Im Quelltext eines Suchen/Ersetzen-Formulars

public event EventHandler Reduce;
public event EventHandler Expand;
```

Nun wäre es natürlich schlecht, wenn sich ein Suchen/Ersetzen-Dialog oder eine schwebende Symbolleiste – beides archetypische Vertreter für nicht-modale Dialoge – jedes Mal durch ihren Besitzer (für gewöhnlich das Hauptformular der Anwendung) verdeckt würden, sobald der Benutzer diesen für die Eingabe aktiviert. Aus diesem Grund, sollte die Eigenschaft `Owner` eines nicht-modalen Dialogs immer auf den Besitzer verweisen, damit die Laufzeitumgebung für die gewünschte Z-Ordnung der Fenster sorgen kann:

```
searchForm.Owner = this;
```

Ohne diese Zuordnung würde das Dialogfeld unter anderem auch in der Task-Leiste erscheinen, sich bei Fensterwechseln über Alt+⇆ hineindrängeln wollen usw.

17.2.1 Codebeispiel – Suchen/Ersetzen-Dialog

Das Projekt *SuchenErsetzen* umfasst die beiden in Abbildung 17.13 gezeigten Formulare. Das Hauptformular `Form1` enthält ein Textfeld mit gesetzter `Multiline`-Eigenschaft, in dem sich – gesteuert von einem nicht-modalen Dialog – Suchen- und Ersetzen-Operationen durchführen lassen.

Die Implementierung des Suchdialogs beschränkt sich im Wesentlichen auf den Export der beiden Textfelder und der vier Schaltflächen. Einzig das

Click-Ereignis der Schaltfläche buttonExpand wird formularintern behandelt. Um alles andere kümmert sich das Hauptformular.

```
public class Search : System.Windows.Forms.Form
{
  public System.Windows.Forms.TextBox textSearch;
  public System.Windows.Forms.TextBox textReplace;
  public System.Windows.Forms.Button buttonSearch;
  public System.Windows.Forms.Button buttonReplaceAll;
  public System.Windows.Forms.Button buttonReplace;
  public System.Windows.Forms.Button buttonExpand;
  private System.Windows.Forms.Label label1;
  private System.Windows.Forms.Label label2;
  public Search()
  {
    InitializeComponent();
  }
  public void buttonExpand_Click(object sender, System.EventArgs e)
  {
    if(buttonExpand.Text == "&Erweitern")
    {
      buttonExpand.Text = "&Reduzieren";
      this.Height = 200;
      textReplace.Enabled = true;
      buttonReplace.Enabled = true;
      buttonReplaceAll.Enabled = true;
    }
    else
    {
      buttonExpand.Text = "&Erweitern";
      this.Height = 100;
      textReplace.Enabled = false;
      buttonReplace.Enabled = false;
      buttonReplaceAll.Enabled = false;
    }
  }
}
```

Das Hauptformular unterhält eine Instanz des Suchdialogs im Datenfeld searchForm, die beim ersten Aufruf der über die Menübefehle SUCHEN und ERSETZEN aktivierten Methode InitSearchForm()angelegt und initialisiert wird. Zur Initialisierung gehört die Registrierung von drei Behandlungsmethoden für die Schaltflächen sowie eine für das Closing-Ereignis, und dort wird auch die Owner-Eigenschaft des Suchdialogs auf this gesetzt

Die Behandlung der Schaltflächen strebt nicht so sehr nach Perfektion. Sie ist bewusst einfach gehalten und gibt sich Mühe, die Operationen ohne viel Drumherum richtig umzusetzen. Erwarten Sie aber nicht zuviel davon, wenn Sie das Programm testen.

Eine Behandlung des Closing-Ereignisses ist Pflicht, da der Benutzer den Suchen/Ersetzen-Dialog jederzeit schließen kann und searchForm dabei sein

Fenster und auch die Werte der beiden Textfelder einbüßen würde. Die Methode verhindert das Schließen und ruft stattdessen die `Hide()`-Methode des Formulars auf. Das Schließen des Fensters beim Beenden der Anwendung ist hingegen gesichert, da das `Closing`-Ereignis bei einem »Ableben« des Besitzers ausbleibt.



Das über `searchForm` *referenzierte C#-Objekt bliebe auch beim Schließen des Fensters erhalten – schlicht deshalb, weil über das Feld* `searchForm` *noch eine Referenz darauf besteht. Was bei einem vollständigen* `Close()` *unwiederbringlich abgebaut wird, sind die GDI-Objekte – also das Fenster und die darin enthaltenen Steuerelemente. Anders gesagt: Ein einmal geschlossenes Formular lässt sich auch dann nicht wiederbeleben, wenn die umhüllenden C#-Datenstrukturen aufgrund einer weiterhin existierenden Referenz über das Schließen hinaus erhalten bleiben.*

Alle Operationen starten grundsätzlich ab der Schreibmarke. Diese müssen Sie also an den Anfang oder irgendwo in die Mitte des Textes setzen, nachdem Sie etwas eingetippt oder über die Zwischenablage eingefügt haben.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace NichtModalerDialog
{
  public class Form1 : System.Windows.Forms.Form
  {
    private System.Windows.Forms.TextBox textBox1;
    private System.Windows.Forms.MainMenu mainMenu1;
    private System.Windows.Forms.MenuItem menuItem1;
    private System.Windows.Forms.MenuItem menuSearch;
    private System.Windows.Forms.MenuItem menuReplace;
    private Search searchForm;
...
    private void menuSearch_Click(object sender, System.EventArgs e)
    {
      InitSearchForm();
      searchForm.buttonExpand.Text = "&Reduzieren";
      searchForm.buttonExpand_Click(null, null);
    }

    private void menuReplace_Click(object sender, System.EventArgs e)
    {
      InitSearchForm();
      searchForm.buttonExpand.Text = "&Erweitern";
      searchForm.buttonExpand_Click(null, null);
    }
```

```
private void InitSearchForm()
{
  if (searchForm == null)
  {
    searchForm = new Search();
    // Ereignisroutinen einhängen
    searchForm.buttonSearch.Click +=
       new EventHandler(this.search_Click);
    searchForm.buttonReplace.Click +=
       new EventHandler(this.replace_Click);
    searchForm.buttonReplaceAll.Click  +=
       new EventHandler(this.replaceAll_Click);
    searchForm.Closing += new CancelEventHandler(searchForm_Closing);
    searchForm.Owner = this;
  }
  searchForm.Show();
}

private void search_Click(object sender, System.EventArgs e)
{
  int pos = textBox1.Text.IndexOf(searchForm.textSearch.Text,
    textBox1.SelectionStart + textBox1.SelectionLength);
  if (pos >= 0)
  {
    textBox1.SelectionStart = pos;
    textBox1.SelectionLength = searchForm.textSearch.Text.Length;
  }
  else   // nicht gefunden, Schreibmarke hinter letzen Fund
  {
    textBox1.SelectionStart += textBox1.SelectionLength;
    textBox1.SelectionLength = 0;
  }
}
private void replace_Click(object sender, System.EventArgs e)
{
  search_Click(null, null);   // nächstes Vorkommen suchen
  if (textBox1.SelectionLength > 0)
  {
    textBox1.SelectedText = searchForm.textReplace.Text; // ersetzen
    textBox1.SelectionLength = searchForm.textReplace.Text.Length;
  }
}
private void replaceAll_Click(object sender, System.EventArgs e)
{
  do
  {
    search_Click(null, null);   // nächstes Vorkommen suchen
    if (textBox1.SelectionLength > 0) // gefunden?
      textBox1.SelectedText = searchForm.textReplace.Text; // ersetzen
    else
      break;                    // fertig
  }
  while(true);
}
```

```
    private void searchForm_Closing(object sender, CancelEventArgs e)
    {
      searchForm.Hide();
      e.Cancel = true;
    }
  }
}
```

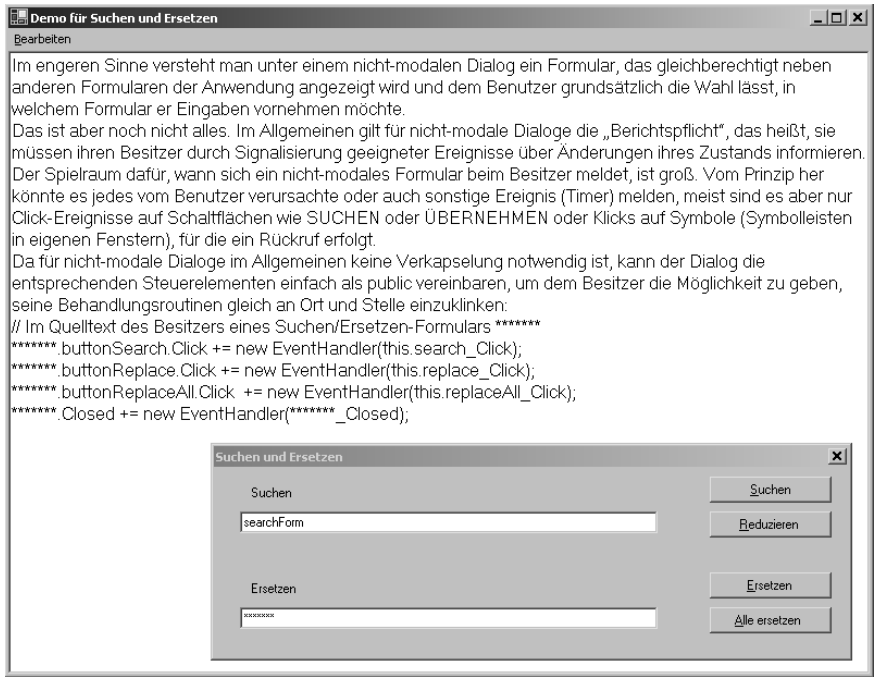


Abbildung 17.13: Das Formular mit Text aus diesem Abschnitt nach einer ALLE ERSETZEN-Operation

17.3 Gestaltung

Bei der inhaltlichen Ausgestaltung eines Dialogfelds haben Sie alle Freiheiten und können nach Herzenslust alles einsetzen, was die Zunft hergibt. Anstatt sich jedoch nach der Devise »Sein ist Design« unüberlegt zu verkünsteln, sollten Sie sich zuerst bestehende Dialoge – beispielsweise die des Betriebssystems oder einer typischen Anwendung – ansehen und deren Aufbau studieren. Sicherlich werden Ihnen dabei viele Details auffallen, die Sie als Benutzer bisher als selbstverständlich genommen haben, hinter denen aber einige Überlegung und jahrelange Erfahrung in der Präsentation von Information steckt. Und dann war da noch die Geschichte mit dem »Was der Bauer nicht kennt, das frisst er nicht«: Es ist schlicht kontraproduktiv, dem späteren Benutzer mit jedem Dialog einen neuen Intelligenztest aufzuerlegen – eine Erfahrung, die man heutzutage speziell bei Webdesigns öfter macht, als einem lieb ist.

Weiterhin stellt sich die Frage, ob ein Dialogentwurf bereits der Weisheit letzter Schluss sein soll oder ob es vorstellbar ist, dass er als Ausgangsbasis

für weitere, speziellere Dialoge fungieren kann. Ein naheliegendes Beispiel dafür wären Nachfolgeversionen der Anwendung, deren zusätzliche Features Erweiterungen auch der Dialoge nach sich ziehen. Dementsprechend sollte nicht nur der Entwurf, sondern auch die Kommunikation mit dem Aufrufer eines Dialogs so gestaltet sein, dass einer abgeleiteten Klasse weder Wege verbaut sind noch Türen offen stehen, Unsinn zu treiben. Das vernünftige Design einer Schnittstelle für die Dialogvererbung gehört zu den anspruchsvolleren Aufgaben bei der Programmierung, erfordert Fingerspitzengefühl und sollte nicht auf die leichte Schulter genommen werden.

17.3.1 TabControl-Steuerelement

Ein besonders in Eigenschaftsdialogen häufig anzutreffendes Strukturierungsmittel ist das `TabControl`-Steuerelement. Als Plattform, die speziell darauf ausgerichtet ist, beliebig viele Dialoge in einen einzigen Dialog zusammenzufassen, ist es in geradezu idealer Weise dafür geeignet, Dialogen mit gut kategorisierbaren Inhalten – eine vernünftige Kategorisierung lässt sich fast immer finden – den Charme und die Übersichtlichkeit eines Karteikastens mit Registerkarten zu verleihen. Darüber hinaus ist dieses Steuerelement eine gute Eintrittskarte für die Formularvererbung.

Tabelle 17.5 und Tabelle 17.6 geben einen kompakten Überblick über den interessanteren Teil der Ausstattung von `TabControl`.

Tabelle 17.5: Eigenschaften des `TabControl`-Steuerelements

Eigenschaft	Bedeutung
`TabAlignment Alignment {get; set;}`	`enum`-Wert, der die Ausrichtung der Registerkarten im Steuerelement bestimmt. Zur Auswahl stehen die Werte `Top` (oberer Rand links, Reiter waagrecht, Voreinstellung), `Bottom` (unterer Rand links, Reiter waagrecht) sowie `Left` und `Right` (linker bzw. rechter Rand oben, Reiter senkrecht).
`TabAppearance Appearance {get; set;}`	`enum`-Wert, der das Erscheinungsbild der Reiter mit den Beschriftungen der Registerkarten bestimmt. Zur Auswahl stehen die Werte `Normal` (Voreinstellung), `Button` (Reiter erscheinen als Schaltflächen mit Umrandung), `FlatButton` (nur der angezeigte Reiter erscheint als Schaltfläche im flachen Design).

Tabelle 17.5: Eigenschaften des `TabControl`-Steuerelements (Forts.)

Eigenschaft	Bedeutung
`TabDrawMode DrawMode {get; set;}`	`enum`-Wert, der bestimmt, ob der Besitzer die Reiter (die hier in der kontextbezogenen Hilfe fälschlicherweise als »Registerkarte« erscheinen) selbst zeichnet (`OwnerDrawFixed`) oder nicht (`Normal`). Ist der Wert auf `OwnerDrawFixed` gesetzt, generiert das Steuerelement für jeden Reiter ein `DrawItem`-Ereignis (jedoch kein `MeasureItem`-Ereignis) und ermöglicht dem Besitzer das Zeichnen des Reiters in der festen Größe `ItemSize` – unabhängig von der `SizeMode`-Einstellung.
`bool HotTrack {get; set;}`	Wenn `true`, verändert die Beschriftung im Reiter ihre Darstellung, wenn der Mauszeiger darüber fährt.
`ImageList ImageList {get; set;}`	Ermöglicht die Bereitstellung einer Bildliste auf Containerebene. `TabPage`-Instanzen können von dieser Liste Gebrauch machen – müssen das aber nicht, da sie selbst über eine Eigenschaft dieses Namens (und Typs) verfügen.
`Size ItemSize {get; set;}`	Bestimmt die Höhe und eine Breite für die Reiter aller Registerkarten. Den Wert für die Breite beachtet das Steuerelement nur, wenn die `SizeMode`-Eigenschaft auf `Fixed` gesetzt ist. Ist die Beschriftung eines Reiters länger als mit `ItemSize.Width` darstellbar, zeichnet das Steuerelement diesen Reiter so groß, dass die Beschriftung vollständig erscheint. Bei senkrechter Ausrichtung (`Alignment`) vertauschen die Eigenschaften ihre Rollen.
`bool Multiline {get; set;}`	Wenn `true`, zeigt das Steuerelement die Reiter der Registerkarten in mehreren Zeilen an und verringert dabei die effektive Höhe aller Registerkarten. Hat diese Eigenschaft den Wert `false` und enthält das Steuerelement mehr Registerkarten, als es in einer Zeile anzeigen kann, stellt es Bildlaufpfeile am rechten bzw. unteren Rand der Reiterleiste dar, mit denen sich die Darstellung scrollen lässt.
`Point Padding {get; set;}`	Bestimmt den Randabstand der Beschriftung zur linken (im Modus `Normal` auch zur rechten) oberen Ecke des Reiters. Bei Änderung dieses Wertes ändert sich die `ItemSize`-Eigenschaft nicht automatisch.
`int RowCount {get;}`	Anzahl der Zeilen bei mehrzeiliger Anzeige

Eigenschaft	Bedeutung
`int SelectedIndex {get; set;}`	Index des angezeigten `TabPage`-Steuerelements
`TabPage SelectedTab {get; set;}`	Aktuell angezeigtes `TabPage`-Steuerelement
`TabSizeMode SizeMode {get; set;}`	`enum`-Wert zur Bestimmung, wie das Steuerelement die horizontalen Abmessungen für die einzelnen Reiter berechnet. Zur Auswahl stehen die Werte: `Normal` – Vorgabe; Das Steuerelement berechnet die Breite eines Reiters über die Länge der Beschriftung und den Wert der `Padding`-Eigenschaft. `FillToRight` – bei einzeiliger Anzeige wie `Normal`, bei mehrzeiliger Anzeige berechnet das Steuerelement einen »Umbruch« und dehnt die Reiter so weit aus, dass sie die gesamte Breite des Steuerelements füllen. `Fixed` – das Steuerelement verwendet den Wert der Eigenschaft `ItemSize.Width` als Breite für die Reiter und schneidet den überhängenden Teil der Beschriftung ab.
`int TabCount {get; set;}`	Anzahl der in der Auflistung `TabPages` verwalteten `TabPage`-Instanzen.
`TabPageCollection TabPages {get; set;}`	Auflistung der von dem Steuerelement verwalteten Registerkarten.

Tabelle 17.6: Methoden und Ereignisse des `TabControl`-Steuerelements

Methode/Ereignis	Bedeutung
`DrawItem`	Wird signalisiert, wenn die Eigenschaft `DrawMode` auf `OwnerDrawFixed` gesetzt ist und ein Reiter, beschrieben durch die `Index`-Eigenschaft des Ereignisobjekts, neu gezeichnet werden muss. Der Zeichenbereich hat eine feste Größe und richtet sich nach der `ItemSize`-Eigenschaft. Er wird wie üblich durch die Eigenschaft `Bounds` des Ereignisobjekts beschrieben. `MeasureItem`-Ereignisse signalisiert das Steuerelement nicht.
`Rectangle GetTabRect()`	Liefert das Rechteck, in dem der Reiter des (als Index) angegebenen `TabPage`-Objekts gezeichnet wird. Es gilt das Koordinatensystem des `TabControl`-Objekts.
`SelectedTabChanged`	Wird signalisiert, nachdem der Benutzer eine andere Registerkarte aktiviert hat.

Formal gesehen handelt es sich bei dem `TabControl`-Steuerelement um einen Container, der über eine `TabPages`-Auflistung verfügt. Nach Platzierung des Steuerelements auf einem Formularentwurf sieht dieses aus wie eine große unbeschriftete Schaltfläche und quittiert jeden Versuch, andere Steuerelemente darauf zu platzieren mit einer Fehlermeldung, dass nur »TabPages« (Registerkarten) erlaubt seien. Ein `TabPage`-Steuerelement werden Sie in der TOOLBOX aber vergebens suchen. Des Rätsels Lösung findet sich in einem zusätzlichen Bereich des zugehörigen Eigenschaften-Fensters bzw. im Kontextmenü des `TabControl`-Steuerelements, die zwei Operationen anbieten: REGISTERKARTE HINZUFÜGEN und TAB ENTFERNEN. (Eine konsequentere Übersetzung für die zweite Operation wäre natürlich »Registerkarte entfernen« gewesen.)

TabPages-Auflistung und TabPage-Klasse

Das Steuerelement stellt zur Bearbeitung seiner `TabPages`-Auflistung aber auch einen TABPAGE-AUFLISTUNGS-EDITOR zur Verfügung, in dem Sie nach Herzenslust Registerkarten einfügen, löschen und umordnen sowie deren (zahlreiche) Eigenschaften einsehen und auf eigene Werte setzen können. Analog zu Menüs und Symbolleisten fügt der Designer für jede `TabPage`-Instanz ein Datenfeld und den entsprechenden Initialisierungscode in das Codegerüst des Formulars ein. Abbildung 17.14 zeigt das Entwurfsszenario.

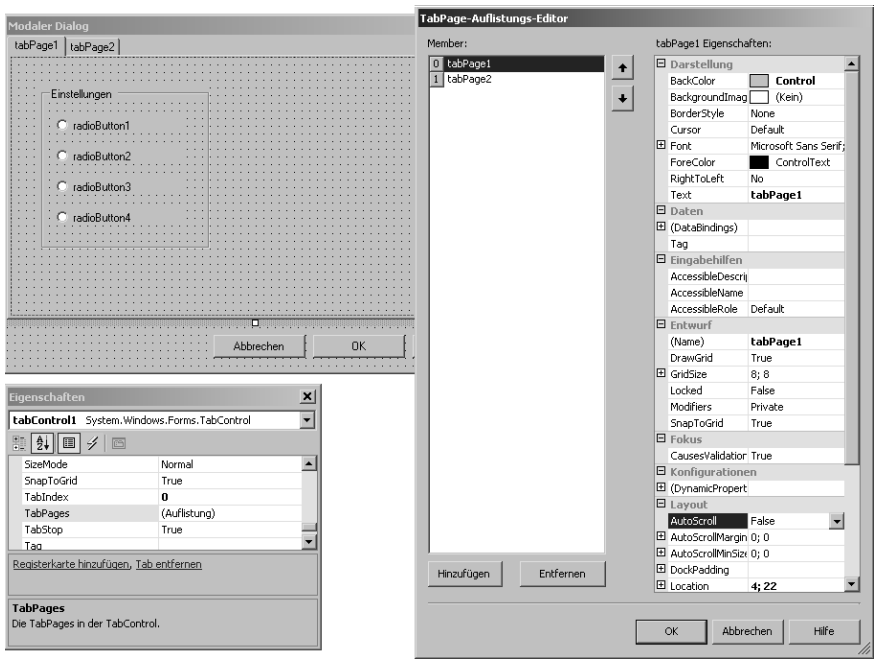


Abbildung 17.14: Entwurf eines Dialogs mit TabControl-Steuerelement

Die Fülle der von einer Registerkarte offengelegten Eigenschaften überrascht nur so lange, bis man in Betracht zieht, dass jede Registerkarte ihrer-

seits ein Container ist und für sich einen Dialog ersetzen kann – weshalb die `TabPage`-Klasse den Löwenanteil der Eigenschaften ihrer direkten Basisklasse `System.Windows.Forms.Panel` öffentlich erbt. Da sich die Größe einer Registerkarte immer nach ihrem Container richtet, fehlen aber Eigenschaften wie `Dock`, `Anchor`, während bequeme Eigenschaften wie `ImageList`, `ImageIndex` und `Tag` hinzugekommen sind.

Als Container besitzt ein `TabPage`-Objekt selbstverständlich eine `Controls`-Auflistung, in der alle auf der Registerkarte befindlichen Steuerelemente vertreten sind. An die Stelle der `Controls`-Auflistung des Formulars tritt nun die `Controls`-Auflistung der einzelnen Registerkarten. Solange Sie die `Controls`-Auflistung einer Registerkarte über den Designer pflegen, fällt diese hierarchische Organisation gar nicht auf: Der Designer erzwingt die eindeutige Benennung aller auf dem Formularentwurf befindlichen Steuerelemente, gleich, in welchem Container sie stecken, weshalb der Zugriff darauf aus Sicht der Formularklasse wie üblich »flach« ist. Mit anderen Worten: Sowohl der Initialisierungscode als auch der Code für die Wertübernahme merken von der visuellen Strukturierung durch `TabControl`-Objekte nichts.

Dynamischer Entwurf und Serialisierung

Wenn Sie natürlich anfangen, ein `TabControl`-Objekt am Designer vorbei zur Laufzeit mit weiteren `TabPage`-Objekten auszustaffieren und diese wiederum mit Steuerelementen vollzupacken, ohne auf Ebene der Formularklasse je eigene Datenfelder dafür vorzusehen, müssen Sie den »Instanzenweg« beschreiten und sich über die `Controls`-Auflistung des `TabControl`-Objekts zur `Controls`-Auflistung des gewünschten `TabPage`-Objekts durchhangeln, um dort das entsprechende Steuerelement herauszufischen. Was hier etwas kompliziert klingt, ist die generische Strategie für die Serialisierung und Deserialisierung von `TabControl`-Steuerelementen bzw. Formularentwürfen im Allgemeinen. Der Designer geht nicht anders vor, wenn er beim Umschalten von der Entwurfsansicht in die Codeansicht das Codegerüst pflegt.

17.4 Formularvererbung

Vererbung ist – wie schon gesagt – eine feine Sache. Je komplexer und spezieller allerdings eine Klasse wird, desto schwieriger wird das vorausschauende Design. Selbst das beste Klassendesign ist irgendwann an dem Punkt angelangt, wo es konkrete Designentscheidungen treffen muss, die sich später als Stolperstein erweisen können und am Ende der Vererbungslinie in ein wüstes »Zurechtbiegen« und Durchreichen von Eigenschaften, Methoden, Parametern und Ereigniskaskaden ausarten – bis sich der Mehraufwand auch angesichts des zunehmend verschlechternden Leistungsverhaltens nicht mehr lohnt und die Linie ausgereizt ist.

Trotz dieses gerade etwas düster gezeichneten Ausblicks hat die Formularvererbung an sich eine ganze Reihe an bestechenden Vorzügen, die in den folgenden Abschnitten kurz diskutiert werden.

17.4.1 Spezialisierung der Basisklasse

In diesem Szenario, das Sie bestens kennen werden, weil es die Standardsituation des Formularentwurfs im Designer umfasst, nutzt die abgeleitete Klasse die ererbte Implementierung wie eine Bibliothek und gründet ihre eigene Implementierung auf diesen Kern. Die Basisklasse kann neben Code, neuen Eigenschaften und Ereignissen auch Steuerelemente wie Bildlaufleisten oder Gestaltungselemente bereitstellen. Das letzte Codebeispiel dieses Kapitels (Seite 713) nimmt dieses Szenario auf und stellt einen typischen Eigenschaftsdialog vor, der sich durch Ableitung erweitern lässt.

Formularschablonen – die Basisklasse als Initialisierungsklasse

Ein etwas eingeschränkterer Blickwinkel auf dasselbe Szenario sind die FORMULARSCHABLONEN. In diesem Fall wird eine, im Allgemeinen direkt von `Form` abgeleitete Basisklasse als Vorlage für eine ganze Schar von Formularen eingesetzt und dient dem alleinigen Zweck, dem Gedanken des *Corporate Design* folgend Vorgaben für Stilattribute, Eigenschaften und andere Ausstattungsmerkmale zu machen – und das an zentraler Stelle (vgl. Abbildung 17.15). Der Vorteil dieses Ansatzes: Auf diese Weise ist die Möglichkeit gegeben, eine Einstellung oder ein Merkmal mit globaler Wirkung für die gesamte Klassenhierarchie zu ändern. Änderungen schlagen nicht nur unmittelbar auf alle abgeleiteten Klassen durch, sie sind auch im Nachhinein jederzeit noch möglich.

Eine recht naheliegende Designstrategie wäre beispielsweise, Vererbung als wohldefinierte Schnittstelle für die Aufteilung der visuellen und prozeduralen Ausgestaltung von Formularen und Dialogen auf verschiedene Arbeitsgruppen in einem Entwicklungsteam einzusetzen. Wenn Sie jemals gezielt mit Dokumentvorlagen etwa in einer Textverarbeitung oder in einem Zeichenprogramm gearbeitet haben, werden Sie eine Vorstellung davon haben, welches Potenzial in diesem Ansatz steckt.

17.4.2 Modifikation der Basisklasse

Vererbung kann man auch unter dem Gedanken sehen, der Basisklasse bestimmte Eigenheiten austreiben zu wollen – etwa den Fensterstil zu ändern oder ein Verhalten zu korrigieren, das so nicht übernommen werden kann oder soll. Die Grenzen zum »Einbruch« in die Basisklasse beziehungsweise zur Zweckentfremdung sind hier aber schnell überschritten. Der folgende Abschnitt diskutiert zwei in diesem Zusammenhang mögliche Strategien.

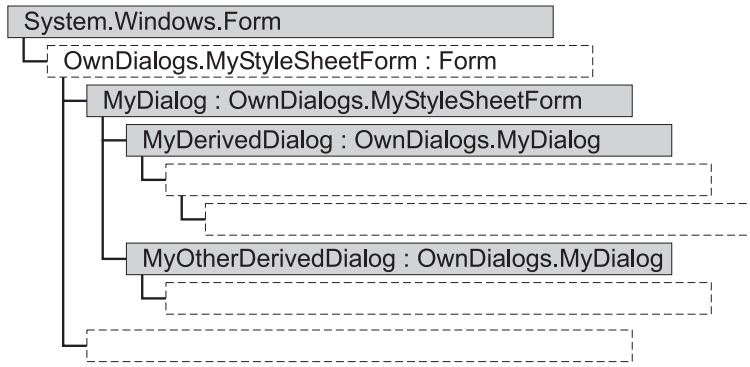


Abbildung 17.15: Klassenhierarchie für die Formularvererbung

17.4.3 Mögliche Implementierungsstrategien

Wenn es um die Planung von Formularklassen – spezieller noch: Dialogklassen geht – steht man schnell vor der Frage, ob es Sinn macht, eine Klasse so auszulegen, dass sie als potenzielle Basisklasse auftreten kann – oder eben nicht. Wer sich dagegen entscheidet, erspart sich auf jeden Fall viel Gedankenarbeit und »überflüssige« Implementierung in Form von Sicherheits- und Gültigkeitstests – eben all das, was anfällt, wenn eine Klasse gegen Missbrauch durch Vererbung wasserdicht gemacht werden soll.

protected-Vereinbarung aller wichtigen Größen

Für den »Hausgebrauch« im eigenen Unternehmen, wo Ausfälle (unbehandelte Ausnahmen oder entartetes Verhalten) nicht so schwer ins Gewicht fallen, können Sie sich für die Strategie der unverkapselten Implementierung entscheiden und allen für die Vererbung wichtigen Größen – Datenfeldern, Methoden und gegebenenfalls auch Steuerelementen – schlicht eine `public`- oder besser noch eine `protected`-Vereinbarung verpassen. Abgeleitete Klassen können dann mit diesen Größen arbeiten, als seien sie die eigenen. Auf jeden Fall müssen Sie dafür entweder eine umfangreiche Spezifikation schreiben, die letztlich die gesamte Implementierung und alle darin verborgenen Fallstricke erläutert oder aber gleich den Quelltext offen legen. Alles Andere wäre Unsinn.

Natürlich müssen Sie sich bei einem solchen, weder wasserdichten noch irgendwelchen Kompatibilitätsüberlegungen folgenden Ansatz die Frage gefallen lassen, warum Sie dafür überhaupt Vererbung einsetzen wollen und nicht gleich einfach mit einer Kopie des Quelltextes der Ausgangsklasse starten (oder diese zur Verfügung stellen), um diese schlicht zurechtzuschneidern. Das ist einfacher, verhindert ein eventuell notwendiges Umschiffen von Erblasten, ermöglicht Verbesserungen, das Ausmerzen von Fehlern, umfangreichere Tests – und ist *last not least* in vielen Fällen sicher effizienter.

Verkapselung – schmale aber wohldefinierte Schnittstellen

Für den Einsatz in kommerziell vertriebenen Klassenbibliotheken scheidet der offene Ansatz natürlich aus. Es ist nicht tragbar, eine Schnittstelle in Umlauf zu bringen, die nicht gegen alle Eventualitäten gesichert ist. Da spielen Haftungsfragen ebenso eine Rolle wie Sicherheitsfragen – ein Image ist schnell beschädigt.

Eine Klasse definiert grundsätzlich zwei Schnittstellen: die Schnittstelle zum Objektbesitzer und die Vererbungsschnittstelle. Letztere ist eine Obermenge der ersteren und umfasst neben den `internal`- und `public`-Elementen eben noch die `protected`-Elemente. Anders gesagt: Ein Objekt einer abgeleiteten Klasse »sieht« alles, was ein Besitzer des Basisklassenobjekts sieht, und darüber hinaus noch etwas mehr. Hier liegt bereits das Problem: Nichts und niemand hält eine abgeleitete Klasse davon ab, ererbte `protected`-Elemente ihrerseits als `public`-Elemente der Öffentlichkeit auszuliefern. Ein Klasse sollte daher nur solche Elemente als `protected`-Elemente vererben, für die sie *alle* Wertveränderungen voll im Griff hat. Die Betonung des Wortes »alle« ist hier entscheidend, wie das folgende nach außen hin harmlos wirkende Codefragment zeigt:

```
class Form1 : Form
{
  private TextBox textBox1;
  protected TextBox TextBox1
  {
    get{ return textBox1; }
  }
  ...
}
```

Der Code bemüht sich zwar, den Schreibzugriff auf das Datenfeld `textBox1` zu unterbinden, die Tür für eine Manipulation aller Eigenschaften des Textfelds steht aber weit offen, da die Schreibsperre nur für die Objektvariable gilt und nicht für die Eigenschaften des daran gebundenen Objekts. Die abgeleitete Klasse könnte beispielsweise die `Dispose()`-Methode des Objekts aufrufen und dieses damit über den Jordan schicken, ohne dass die Basisklasse etwas davon merkt (nein, ein Ereignis namens `Disposing` signalisiert `TextBox` nicht.) Der nächste Zugriff der Basisklassenimplementierung auf das Objekt endet in diesem Fall unweigerlich mit einer Ausnahme, die mit ziemlicher Sicherheit nicht abgefangen wird (und wohl auch schlecht abzufangen ist).

Es gibt aber noch schlimmere Szenarien. Stellen Sie sich vor, der Dialog nimmt über völlig andere Steuerelemente, die er nicht offen legt, eine Kennwortabfrage vor. In diesem Fall stellt der obige Code nicht nur ein Absturzrisiko dar, sondern ein Sicherheitsloch erster Güte. Ein potenzieller Angreifer könnte sich über die `Parent`-Eigenschaft des Textfelds zum Formu-

larobjekt und von dort zur `Controls`-Auflistung hin durchhangeln. Nun ist es ein Leichtes, das für die Kennwortabfrage zuständige Textfeld ausfindig zu machen (schließlich ist der Datentyp ja klar und die `PasswortChar`-Eigenschaft ein sicheres Erkennungsmerkmal) und von der abgeleiteten Klasse aus zu »automatisieren«, während ein davor gezeichneter Klon des Textfeldes in aller Ruhe die Kennwörter einsammelt und die Ableitung über das Internet mal eben schnell »nach Hause telefoniert«. Eine schlichtere Variante dieses Angriffs käme voraussichtlich auch mit einer eingeschmuggelten Ereignisbehandlung für das `TextChanged`-Ereignis des Textfelds aus.

Wenn es darum geht, die `Text`-Eigenschaft eines Textfelds für ein abgeleitetes Objekt les- und manipulierbar zu gestalten, sollten Sie sich also besser an dem folgenden Muster orientieren:

```
class Form1 : Form
{
  private TextBox textBox1;
  protected string TextBox1
  {
    get { return textBox1.Text; }
    set { textBox1.Text = value; }
  }
  ...
}
```

Es gibt aber noch einen Mittelweg, der die Schnittstelle elegant hält und auf eine *Doppelpufferung* hinausläuft (und den übrigens auch viele .NET-Klassen gehen):

```
class Form1 : Form
{
  private TextBox textBox1;
  private TextBox textBoxbuffer;
  protected string TextBox1
  {
    get
    {
      textBoxBuffer.Text = textBox1.Text;
      textBoxBuffer.Font = textBox1.Font;
      ...
      return textBoxbuffer;
    }
    set
    {
      textBoxBuffer = value;
      textBox1.Text = value.Text;
      textBox1.Font = value.Font;
      ...
    }
  }
  ...
}
```

Die Ableitung kann hier zwar ein Textfeldobjekt abfragen und setzen, womöglich auch einen `Show()`-Aufruf dafür veranstalten, der Zugriff auf das originale Textfeld bleibt ihr aber verwehrt, denn die Basisklasse pickt sich nur die Eigenschaften heraus, die zur Spezifikation der Vererbungsschnittstelle gehören.

Als Faustregel gilt, dass eine Basisklasse bedenkenlos nur solche Datenfelder als Eigenschaften oder über Methoden offen legen sollte,

- *die einen Werttyp tragen* – die Manipulation von Werttypen lässt sich im `set`-Accessor gut überwachen.
- *deren Zuweisung eine Wertkopie erzeugt* – hier wären in erster Linie Strings zu nennen.
- *deren Datentyp sie selbst exportiert* – da die Basisklasse den Datentyp selbst implementiert, hat sie so auch den Zugriff auf alle Elemente des Datenfelds im Griff.
- *die kausal unabhängig sind* – die Basisklassenimplementierung darf an keiner Stelle Vorannahmen über den Wert des Elements machen, sondern muss Lesezugriffe darauf so gestalten, dass jederzeit Werte aus dem gesamten Wertebereich des zugrunde liegenden Datentyps verarbeitet werden können.

Die strikte Beachtung dieser Regeln kann gerade bei umfangreichen Formularen bzw. Dialogen einen erheblichen Aufwand bedeuten und auch das Öffnen einer Formularklasse für die Vererbung generell in Frage stellen.

Genug des erhobenen Zeigefingers. Wo Vererbung draufsteht, sollte auch Vererbung drin sein. Das folgende Codebeispiel zeigt, wie es geht.

17.4.4 Codebeispiel – Eigenschaftsdialog mit Vererbung

Die Projektmappe *MyDerivedDialog* umfasst Projekte für die Formularklassen `MyDialog` und `MyDerivedDialog`, die zwei für den modalen Aufruf konzipierte Eigenschaftsdialoge implementieren. Wie die Namensgebung bereits andeutet, ist `MyDerivedDialog` eine Ableitung von `MyDialog`. Beide Dialoge zeigen Registerkarten in einem `TabControl`-Steuerelement – `MyDerivedDialog` eine mehr als `MyDialog` –, legen die aktuellen Einstellungen der darauf befindlichen Steuerelemente bei Verlassen des Dialogs über OK in der Registrierung ab und lesen sie beim nächsten `ShowDialog()`-Aufruf wieder von dort ein.

Um dem Arrangement noch einen praxisnahen Rahmen zu geben, findet der Start des `MyDerivedDialog`-Formulars über ein `NotifyIcon`-Steuerelement statt, das sich nach dem Programmstart in den Systembereich der Taskleiste einnistet und über ein eigenes Kontextmenü verfügt. Jenseits des genannten

Spezifikationskerns demonstriert der Code also verschiedene weitere Techniken. Hier ein Überblick:

- Modaler Dialog
- Umgang mit dem `TabControl`-Steuerelement
- Formularvererbung mit Überschreibung virtueller Methoden
- Zugriff auf die Registrierung, Öffnen von Schlüsseln, Lesen und Schreiben von Werten
- Umgang mit dem `NotifyIcon`-Steuerelement
- Start einer Anwendung ohne Formular (`ApplicationContext`)

Teil 1 – das Projekt MyDialog

Da die Klasses `MyDialog` als Basisklasse für `MyDerivedDialog` konzipiert ist, geht die folgende Implementierungsbeschreibung auch auf die diesbezüglichen Designentscheidungen ein. Falls Sie sich nicht für die Formularvererbung interessieren, sollte das der Sache keinen Abbruch tun. Der Dialog funktioniert natürlich auch im Stand-Alone-Betrieb.

Zentrale Datenstruktur ist das `TabControl`-Steuerelement `tabControl1`, das eine `protected`-Vereinbarung erhält und so auch – »für den hausinternen Gebrauch« – potenziellen Ableitungen zur freien Verfügung steht:

```
protected System.Windows.Forms.TabControl tabControl1;
```

Wie Abbildung 17.16 verdeutlicht, zeigt das `MyDialog`-Formular im oberen Teil zwei Registerkarten mit verschiedenen Steuerelementen und im unteren Teil drei Schaltflächen für das modale Gerüst des Dialogs an. Die Steuerelemente wurden im Designer eingefügt, automatisch benannt und spielen für die Zwecke des Beispiels eine untergeordnete Rolle.

Die OK-Schaltfläche ist so verdrahtet, dass der Dialog die Werte der Steuerelemente auf den Registerkarten unter dem Schlüssel `\\HKEY_LOCAL_MACHINE\Software\MyDialog` in die Registrierung schreibt und den Dialog dann beendet:

```
public const string OwnRegistryKey = "Software\\MyDialog";

private void buttonOK_Click(object sender, System.EventArgs e)
{
  WriteTabsToRegistry(OwnRegistryKey);
  Close();
}
```

Die Methode `WriteTabsToRegistry()` arbeitet mit der Bibliotheksklasse `Registry`, die verschiedene statische Methoden bzw. Eigenschaften für den Lese- und Schreibzugriff auf die Hauptschlüssel der Registrierung bereitstellt.

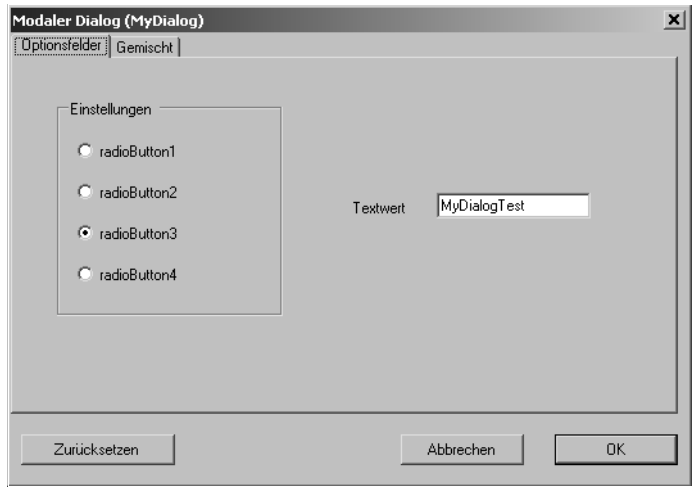


Abbildung 17.16: Laufzeitansicht des `MyDialog`-Formulars

`CreateSubKey()` legt einen neuen Unterschlüssel unter dem genannten Pfad an bzw. öffnet ihn, wenn er bereits existiert. Rückgabewert dieses Aufrufs ist ein Objekt des Typs `RegistryKey`, eine Art verkapselter Datei-Handle. Über dieses Objekt können nun Schreib- und Leseoperationen durch Aufruf der Instanzmethoden `SetValue()` und `GetValue()` abgewickelt werden. `SetValue()` verlangt im ersten Parameter einen Bezeichner für den zu speichernden Wert und im zweiten Parameter den Wert selbst, wie für die assoziative Speicherung üblich. Strings und Aufzählungswerte werden im Stringformat *REG_SZ*, alles andere im binären Format *REG_DWORD* in der Registrierung gespeichert – was für das spätere Auslesen natürlich eine Rolle spielt. Hier erst einmal der Code für das Schreiben in die Registrierung:

```
using Microsoft.Win32;
...

// Schreibt Inhalte der Registerkarte unter dem angegebenen Schlüssel
// in die Registry und liefert den geöffneten Schlüssel oder null,
// wenn ein Fehler aufgetreten ist.
protected virtual RegistryKey WriteTabsToRegistry(string Key)
{
  Microsoft.Win32.RegistryKey regKey;
  try   // Kann schiefgehen, bspw. wenn Schreibrecht für Registry fehlt
  {
    regKey = Microsoft.Win32.Registry.LocalMachine.CreateSubKey(Key);
    // Schreiben der Registerkartenwerte
    // Erste Registerkarte
    for(int i = 0; i < groupBox1.Controls.Count; i++)
      if (((RadioButton)groupBox1.Controls[i]).Checked)
        regKey.SetValue(groupBox1.Text, i);
    regKey.SetValue(label1.Text, textBox1.Text);
    // Zweite Registerkarte
    regKey.SetValue(checkBox1.Text, checkBox1.CheckState);
    regKey.SetValue(checkBox2.Text, checkBox2.CheckState);
```

```
    regKey.SetValue(checkBox3.Text, checkBox3.CheckState);
    regKey.SetValue(label2.Text, trackBar1.Value);
    return regKey;
  }
  catch (Exception e)
  {
    MessageBox.Show(e.Message,
     "Fehler beim Schreiben in die Registrierung");
  }
  return null;
}
```

Ausgelesen wird die Registrierung in Reaktion auf das `Load`-Ereignis, also gleich bei Aufruf des Formulars, sowie auf das `Click`-Ereignis der Schaltfläche ZURÜCKSETZEN. Die Methode dafür heißt `ReadTabsFromRegistry()`. Vielleicht werden Sie sich fragen, ob ihren Aufruf nicht auch der Konstruktor übernehmen könnte. Das funktioniert in der Tat – aber nur, solange keine Formularvererbung (speziell: keine `override`-Variante der Methode `ReadTabsFromRegistry()`) im Spiel ist. Denn bei Konstruktion des Formularobjekts der abgeleiteten Klasse kommt als erstes der Konstruktor der Basisklasse zum Aufruf. Würde dieser die virtuelle Methode `ReadTabsFromRegistry()` aufrufen, käme die gegebenenfalls existierende Überschreibung der abgeleiteten Klasse zum Zuge, und würde in ihrem Versuch scheitern, auf Steuerelementinstanzen des eigenen Formulars zuzugreifen – die existieren nämlich zu diesem Zeitpunkt noch nicht. Fehler dieser Art sind schwer zu finden und lassen sich nur durch die umsichtige Planung einer Klasse als Basisklasse bereits im Vorfeld vermeiden.

Beim allerersten Aufruf dieser Methode wird sie weder einen Schlüssel noch Werte unter dem Schlüssel vorfinden. Für diesen Fall greift die `InitalizeComponent()`-Initialisierung des Formulars. Fehlen Einträge für einzelne Werte bei existierendem Schlüssel, stellt die zweiparametrige Überladung von `GetValue()` einen Vorgabewert bereit, der hier der Einfachheit halber fest kodiert ist, aber natürlich auch die Werte der `InitalizeComponent()`-Initialisierung verwenden könnte. Generell ist im Zusammenhang mit `GetValue()` darauf zu achten, in welchem Datenformat die Methode einen Wert liefert (*REG_SZ* oder *REG_DWORD*) – der Rückgabetyp `object` ist hier etwas irreführend. Beachten Sie dazu vor allem den `Enum.Parse()`-Aufruf – und auch den Vorgabewert – für Werte des Aufzählungstyps `CheckState`.

```
private void MyDialog_Load(object sender, System.EventArgs e)
{
  buttonReset_Click(null, null);   // Werte einlesen
}

private void buttonReset_Click(object sender, System.EventArgs e)
{
  ReadTabsFromRegistry(OwnRegistryKey);
}
```

```
// Liest den angebene Schlüssel als Unterschlüssel von \\HKEY_LOCAL_MACHINE
// und setzt die Registerkarten entsprechend
protected virtual RegistryKey ReadTabsFromRegistry(string Key)
{
  RegistryKey regKey;
  if ((regKey = Registry.LocalMachine.OpenSubKey(Key, true)) != null)
  {
    // Initialisieren der Registerkartenwerte

    // Erste Registerkarte
    int Index = 0;
    Index = (int) regKey.GetValue(groupBox1.Text, 0);
    ((RadioButton)groupBox1.Controls[Index]).Checked = true;
    textBox1.Text = (string) regKey.GetValue(label1.Text, "");
    // Zweite Registerkarte
    checkBox1.CheckState = (CheckState)
      System.Enum.Parse(typeof(CheckState),
       (string) regKey.GetValue(checkBox1.Text, "Unchecked"));
    checkBox2.CheckState = (CheckState)
      System.Enum.Parse(typeof(CheckState),
        (string) regKey.GetValue(checkBox2.Text, "Unchecked"));
    checkBox3.CheckState = (CheckState)
      System.Enum.Parse(typeof(CheckState),
        (string) regKey.GetValue(checkBox3.Text, "Unchecked"));
    trackBar1.Value = (int) regKey.GetValue(label2.Text);
  }
  return regKey;
}
```

Dass die beiden Methoden für den Zugriff auf die Registrierung einen Wert vom Typ `RegistryKey` zurückgeben, ist gleichfalls eine Design-Entscheidung, die auf das Konto Vererbung geht: Eine abgeleitete Klasse soll die Basisklassenvariante aufrufen und dann mit dem bereits geöffneten Schlüssel weiterarbeiten können. Und noch ein Tribut an die Vererbung: Ein expliziter `Close()`-Aufruf für das `RegistryKey`-Objekt findet sich nicht. Die Implementierung baut hier auf den impliziten Aufruf beim Abbau des Objekts. Würde die Basisklasse den Handle freigeben, könnte sie die abgeleitete Klasse damit brüskieren und umgekehrt.

Damit Sie die Formularklasse testen können, verfügt diese über eine Methode `Main()`, die ohne großes Brimborium – und ohne `Application.Run()`-Aufruf – eine Instanz des Dialogs per `ShowDialog()` aufruft. (`Application.Run()` nimmt einen nicht-modalen Aufruf des übergebenen Formulars vor.)

```
[STAThread]
static void Main()
{
  new MyDialog().ShowDialog();
}
```

Wenn Sie den Dialog als alleinstehende Anwendung testen wollen, öffnen Sie vom PROJEKTMAPPEN-EXPLORER aus den Eigenschaftsdialog des Pro-

jekts *MyDialog* und setzen Sie unter ALLGEMEINE EIGENSCHAFTEN den AUSGABETYP auf *Windows-Anwendung*. Wird auf das Projekt für diesen Dialog von einem anderen Projekt aus verwiesen, weil `MyDialog` dort als Basisklasse auftritt, muss *Klassenbibliothek* als AUSGABETYP gesetzt sein.

Teil 2 – Ableitung der Klasse MyDerivedDialog in einem eigenen Projekt

Designziele für die Klasse `MyDialog` waren unter anderem, dass sie

- als Basisklasse für modale Dialoge verwendbar ist, die weitere Registerkarten einbringen
- die Infrastruktur für das Auslesen und die Speicherung von Werten in die Registrierung stellt.

Die weitere Beschreibung geht vom komplexeren Fall aus, dass für die Basisklasse und die abgeleitete Klasse verschiedene Projekte existieren – was eher der Praxis entspricht und die parallele Entwicklung beider Dialoge vereinfacht. Natürlich können Sie grundsätzlich auch die *.cs*-Datei der Basisklasse in das jeweilige Projekt einfügen, müssen dann aber aufpassen, dass Sie nicht plötzlich verschiedene Varianten der Basisklasse erhalten, wenn später noch Änderungen erforderlich sein sollten. Der Projektverweis ist auf jeden Fall die bessere Wahl.

Formularvererbung schrittweise

Die folgende Anleitung beschreibt die Formularvererbung bezogen auf das Projekt *MyDerivedDialog* schrittweise:

1. Legen Sie ein neues Projekt mit dem Typ *Windows-Anwendung* und dem Namen *MyDerivedDialog* an.
2. Binden Sie vom PROJEKTMAPPEN-EXPLORER aus über den Kontextmenübefehl HINZUFÜGEN/VORHANDENES PROJEKT der Projektmappe das Projekt `MyDialog` in die Projektmappe ein. (Alternativ könnten Sie, wie angesprochen, auch nur die Datei *MyDialog.cs* zum Projekt selbst hinzufügen. Schritt 3 entfällt dann.)
3. Fügen Sie in das Projekt *MyDerivedDialog* einen Verweis auf das hinzugefügte Projekt ein: VERWEISE/VERWEIS HINZUFÜGEN/PROJEKTE. Damit `MyDialog` als Basisklasse auswählbar ist, muss die Projekteigenschaft AUSGABETYP des Projekts *MyDialog* auf *Klassenbibliothek* gesetzt sein.
4. Löschen Sie das Hauptformular *MyDerivedDialog.cs* des Projekts im Projektmappen-Explorer und fügen Sie ein Formular dieses Namens erneut über den Kontextmenübefehl HINZUFÜGEN/GEERBTES FORMULAR HINZUFÜGEN des Projekts *MyDerivedDialog* ein. Dabei können Sie auswählen, welche Komponente aus der aktuellen Projektumgebung Sie als Basis verwenden wollen (Abbildung 17.17). Alternativ können

Sie das Hauptformular auch behalten und gleich in der Codeansicht die neue Basisklasse eintragen. Da der Designer aber bereits Einstellungen (beispielsweise die Abmessungen) für das Formular gesetzt hat, müssen Sie zusätzlich noch die `this`-Initialisierungen in `InitializeComponent()` löschen, damit das Formular alle Werte der Basis übernimmt.

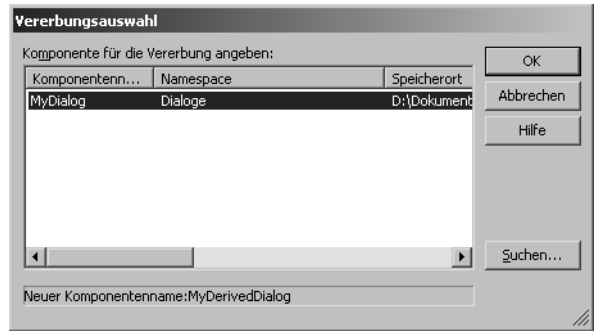


Abbildung 17.17: Auswahl der Basis für die Formularvererbung

Abbildung 17.18 gibt die Entwurfsansicht des neuen Formulars unmittelbar nach Schritt 4 wieder. Die Ansicht zeigt alle Elemente der Basis, kennzeichnet diese aber mit einem Ableitungssymbol, damit Sie im weiteren Verlauf des Entwurfs sehen, was ererbt ist, und was nicht. Im EIGENSCHAFTEN-Fenster sehen Sie die Vorgabewerte für die Eigenschaften der Steuerelemente, können diese aber nicht ändern, sofern das jeweilige Steuerelement keine `protected`- oder `public`-Vereinbarung trägt.

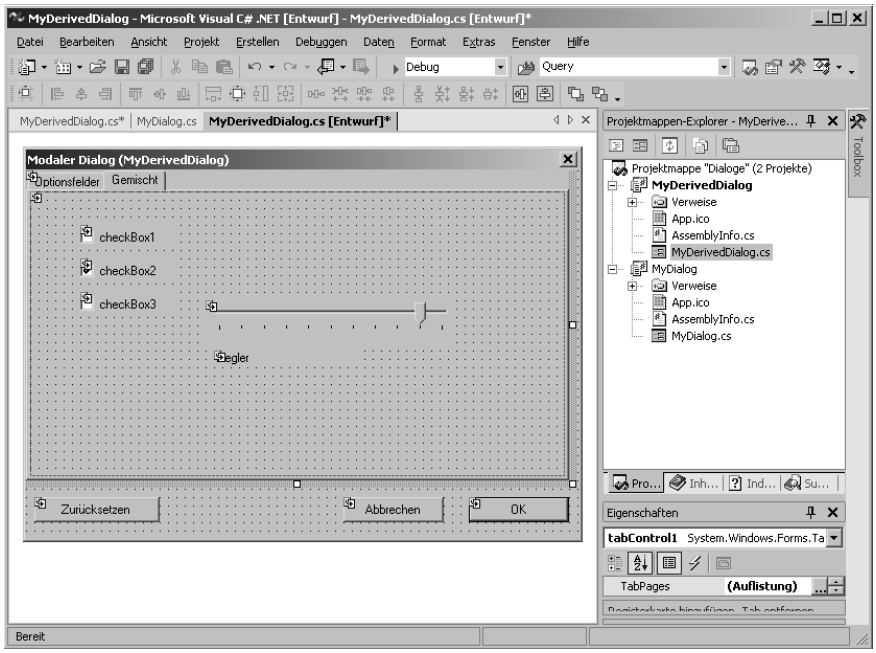


Abbildung 17.18: Entwurfsansicht des abgeleiteten Formulars unmittelbar nach der Ableitung

Die Schnittstelle zur Basisklasse

Im vorliegenden Fall trägt allein das `TabControl`-Steuerelement `tabControl1` eine `protected`-Vereinbarung und gestattet daher das interaktive Einfügen weiterer Registerkarten im TABPAGE-AUFLISTUNGS-EDITOR. Abbildung 17.18 zeigt die auf Ebene der Klasse `MyDerivedDialog` hinzugefügte dritte Registerkarte.

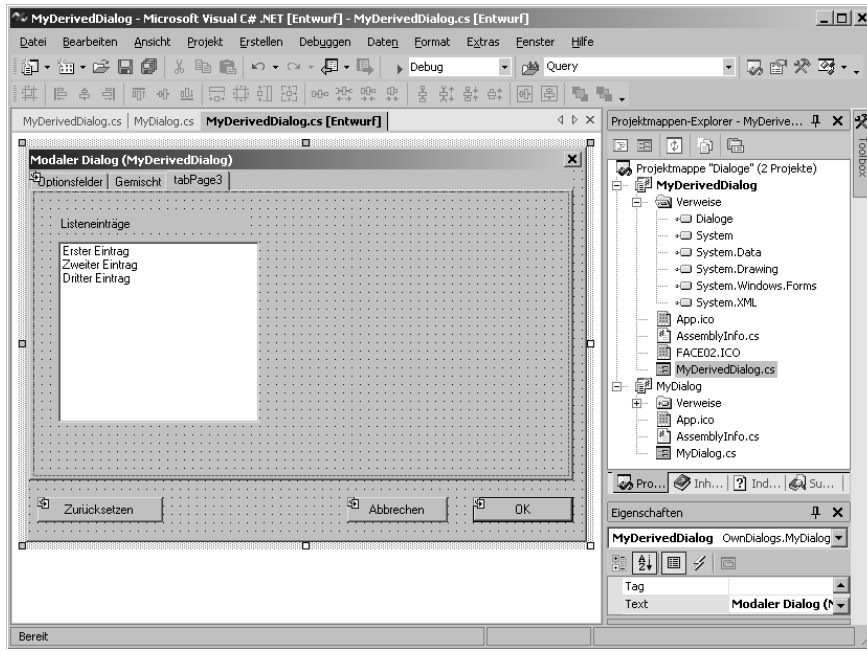


Abbildung 17.19: Entwurfsansicht des abgeleiteten Formulars nach Einfügen einer weiteren Registerkarte

Der Editor listet die Registerkarten der Basisklasse und deren Eigenschaften auf, belegt sie aber augenscheinlich mit einem Schreib- bzw. Löschschutz, der zur Laufzeit nicht mehr vorhanden ist (als nur zur Designzeit existiert). Mit anderen Worten: Das `tabControl1`-Steuerelement öffnet Tür und Tor für allerlei Unsinn. Code, der den Grundstein für Laufzeitfehler in der Basisklasse legt, wäre beispielsweise:

```
this.tabControl1.TabPages.RemoveAt(0);
```

Nachdem der Code der Basisklasse auf Elemente in dieser Registerkarte referiert, wäre es unter dem Aspekt der Typsicherheit wünschenswert, wenn Löschaktionen dieser Art von vornherein gar nicht erst möglich wären. Wie bereits im Vorspann diskutiert, gibt es verschiedene Strategien, um für diesen Fall mehr Sicherheit zu erreichen. Sie fallen jedoch alle recht codeintensiv aus. Um eine bessere Verkapselung zu erreichen, würden sich im vorliegenden Fall zwei Strategien anbieten:

1. `MyDialog` könnte einen eigenen von `TabControl` abgeleiteten Datentyp `MyTabControl` bereitstellen, der den Zugriff auf die Registerkarten der Basisklasse verhindert bzw. überwacht und Übergriffe auf die `Controls`-Auflistung des Formulars unterbindet.
2. `MyDialog` könnte einen von `TabPage` abgeleiteten Datentyp mit ausgeblendeter `Parent`-Eigenschaft sowie mit Operationen zum Einfügen und Entfernen von Objekten dieses Typs bereitstellen. Eine abgeleitete Klasse kann dann eigene Registerkarten einfügen und löschen, während die Registerkarten der Basisklasse geschützt bleiben.

Eine Ausarbeitung dieser Ansätze können Sie bei Bedarf selbst vornehmen. Um aus der Klasse `MyTabControl` ein richtiges Steuerelement zu machen, lesen Sie bitte Kapitel 16, »Steuerelemente selbst implementieren«. Leider verbauen Sie sich mit beiden Ansätzen Möglichkeiten für das interaktive Design mit dem Designer, so dass die Aussichten für eine »schöne« oder gar schnelle Lösung eher trüb sind.

Neben `tabControl1` vererbt `MyDialog` noch die folgenden Elemente, deren Namensgebung bereits alles sagt:

```
public const string OwnRegistryKey = "Software\\MyDialog";
protected virtual RegistryKey WriteTabsToRegistry(string Key);
protected virtual RegistryKey ReadTabsFromRegistry(string Key);
```

Da die Klasse `MyDerivedDialog` eigene Werte in die Registrierung schreiben bzw. daraus lesen soll, überschreibt sie diese Methoden mit eigenen Implementationen, die aber ihrerseits wiederum die Basisklassenvarianten aufrufen – das übliche Muster also. Weiterhin überschreibt die Klasse auch die Konstante `OwnRegistryKey` und verwendet ihren eigenen Pfad in der Registrierung. Der Code selbst enthält keinerlei Spezialitäten – und ist seinerseits natürlich wieder so ausgelegt, dass die weitere Vererbung klappt.

```
public new const string OwnRegistryKey = "Software\\MyDerivedDialog";

// Wird von der Basisklasse aufgerufen
protected override RegistryKey WriteTabsToRegistry(string Key)
{
  RegistryKey regKey = base.WriteTabsToRegistry(OwnRegistryKey);
  regKey.SetValue(label3.Text, listBox1.SelectedIndex);
  return regKey;
}
// Wird von der Basisklasse aufgerufen
protected override RegistryKey ReadTabsFromRegistry(string Key)
{
  RegistryKey regKey = base.ReadTabsFromRegistry(OwnRegistryKey);
  if (regKey != null)
    listBox1.SelectedIndex = (int) regKey.GetValue(label3.Text, 0);
  return regKey;
}
```

Teil 3 – Modaler Aufruf über eine NotifyIcon-Komponente

Der Code für den Dialog ist damit bereits erledigt, und einem `ShowDialog()`-Aufruf steht nichts mehr im Wege. Nicht nur, um diesen Aufruf etwas dramatischer zu gestalten, sondern auch, um ein weiteres gängiges Konzept für die Aktivierung ständig in Bereitschaft befindlicher Anwendungen vorzustellen, enthält das Projekt eine weitere Klasse `StartCode`, die den Aufruf des modalen Dialogs über eine `NotifyIcon`-Komponente inszeniert: Nach Programmstart erscheint zunächst nur ein Smiley-Symbol in der Taskleiste von Windows, das wiederum den Dialog per Mausklick auf den Plan ruft – und das beliebig oft.

Instanzen der `NotifyIcon`-Komponente nisten sich im Systembereich der Taskleiste (system tray) als Symbol ein und signalisieren im Wesentlichen Mausereignisse an ihren Besitzer. Da sie über eine `ContextMenu`-Eigenschaft verfügen, lassen sie sich mit einem Kontextmenü versehen, das mit beliebigen Befehlen bestückt werden kann. Die vorliegende Implementierung verwendet ein Kontextmenü mit zwei Befehlen: ÖFFNEN und BEENDEN. Ersterer entspricht dem Mausklick und öffnet den Dialog, letzter beendet die gesamte Anwendung. (Einen anderen Weg zur Beendigung der Beispielanwendung gibt es übrigens nicht.)

Implementierung

Erwartungsgemäß stellt die Klasse `StartCode` die Methode `Main()`. Ihre Fassung darf mit Recht als ungewöhnlich bezeichnet werden. Offensichtlich benötigt das Programm kein eigenes Formular, da seine dürftige Benutzerschnittstelle in der Taskleiste bestens untergebracht ist. Um eine formularlose Anwendung zu starten, die sich auch wieder abbrechen lässt, wird schlicht eine andere Überladung der Methode `Application.Run()` verwendet, die ein `ApplicationContext`-Objekt als Parameter erwartet. (Sie können diesem Objekt übrigens über die Eigenschaft `MainForm` bei Bedarf später jederzeit noch ein Formular zuordnen). Es empfiehlt sich, ein statisches Datenfeld für dieses Objekt einzuführen, da es die »Austrittskarte« aus dem fahrenden Zug verkörpert, indem es die Methode `ExitThread()` bereitstellt.

Die Initialisierung der `NotifyIcon`-Instanz umfasst im Wesentlichen die Definition eines Kontextmenüs und die Registrierung zweier Behandlungsmethoden für die drei möglichen `Click`-Ereignisse (Klick auf das Symbol und Klicks auf die beiden Menübefehle). Die `Click`-Behandlung startet den modalen Dialog `MyDerivedDialog`, die andere beendet das Programm.

```
using System;
using System.Windows.Forms;
using System.ComponentModel;
using System.Drawing;
```

```
namespace OwnDialogs
{
  public class StartCode
  {
    static NotifyIcon notifyIcon;
    static bool DialogVisible = false;
    static ApplicationContext applicationContext;
    [STAThread]
    static void Main()
    {
      notifyIcon = new NotifyIcon();
      notifyIcon.Icon = new Icon(typeof(MyDerivedDialog), "FACE02.ICO");
      notifyIcon.Click += new EventHandler(notifyIcon_Click);
      notifyIcon.Text = "MyDerivedDialog";     // erscheint als ToolTip
      notifyIcon.ContextMenu =
        new ContextMenu(new MenuItem[] {
           new MenuItem("Beenden", new EventHandler(menuClose_Click)),
           new MenuItem("Öffnen", new EventHandler(notifyIcon_Click))
                                      });
      notifyIcon.Visible = true;

      applicationContext = new ApplicationContext();
      Application.Run(applicationContext);
    }

    // Zeigt den Dialog an
    private static void notifyIcon_Click(object sender, EventArgs e)
    {
      if (!DialogVisible)   // Mehrfachaufruf verhindern
      {
        DialogVisible = true;
        new MyDerivedDialog().ShowDialog();
        DialogVisible = false;
      }
    }

    // Beendet das Programm
    private static void menuClose_Click(object sender, EventArgs e)
    {
      notifyIcon.Dispose();
      applicationContext.ExitThread();
    }
  }
}
```

Abbildung 17.20 zeigt einen Ausschnitt des Windows-Desktops mit den wesentlichen Details: NotifyIcon, Kontextmenü und schlussendlich auch den Dialog selbst.

Mit dieser Plattform macht es richtig Spaß, die Funktionsweise des Dialogs zu testen. Wenn Sie sehen wollen, was in der Registrierung passiert, starten Sie den Registrierungseditor, indem Sie den Menübefehl START/AUSFÜHREN aufrufen und dann *Regedit* eintippen. Das Messgerät steht, nun können Sie

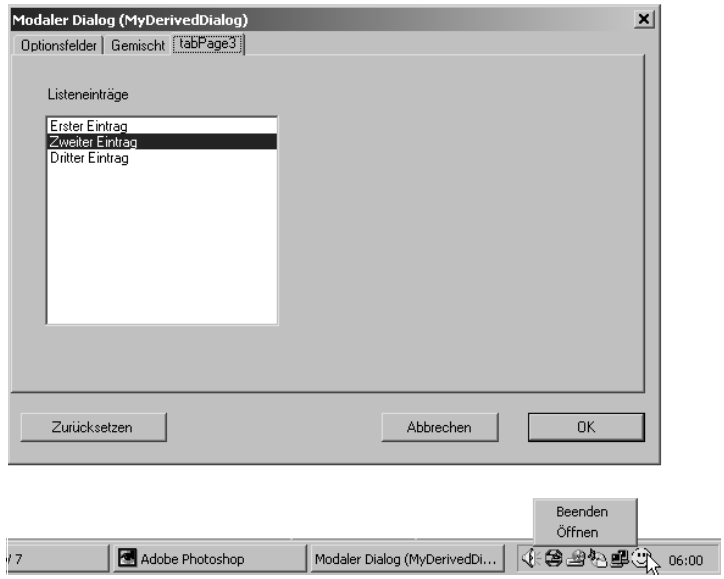


Abbildung 17.20: Auszug aus dem Windows-Desktop mit dem Dialog und dem NotifyIcon (Smiley-Symbol) im Systembereich der Taskleiste, über das der Dialog aufgerufen und das Programm beendet wird.

das Smiley-Symbol anklicken. Um die Wirkung der OK-Schaltfläche des Dialogs zu beobachten, geben Sie den Befehl ANSICHT/AKTUALISIEREN im Registrierungseditor (Abbildung 17.21).

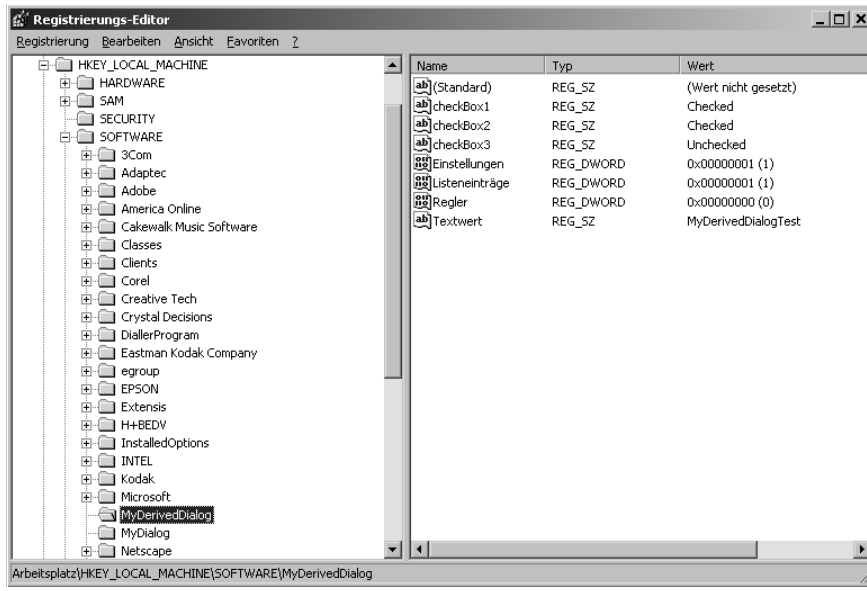


Abbildung 17.21: Hinterlassenschaften des Dialogs `MyDerivedDialog` in der Registrierung

Nach den Tests löschen Sie die Schlüssel am besten über den Registrierungseditor aus der Registrierung.

18 Zwischenablage und Drag&Drop

Im Grunde genommen sind die Zwischenablage und Drag&Drop nichts weiter als zwei Gesichter ein und derselben Technologie, die auf den Namen OLE hört. Beide Mechanismen ermöglichen den Datentransfer von einem *Quellobjekt* zu einem *Zielobjekt* (im Falle der Zwischenablage sogar zu beliebig vielen Zielobjekten), wobei Quellobjekt und Zielobjekt auch unterschiedlichen Anwendungen angehören dürfen.

Dass der Datentransfer auch die Prozessgrenze und womöglich sogar die Systemgrenze überschreiten kann, davon merken Sie als Programmierer nichts. Einzig der Handshake, das heißt die Verständigung über das gewünschte Datenformat (als Tribut an das Typsystem) und das Angeben einer Operation sind vielleicht etwas ungewohnt.

18.1 Zwischenablage

Die Zwischenablage ist in der .NET-Klassenhierarchie über die direkt von `System.Object` abstammende, versiegelte und nicht instanziierbare `Clipboard`-Klasse repräsentiert.

18.1.1 Steuerelemente mit eigener Zwischenablagenfunktionalität

Bei Experimenten mit der Zwischenablage sollten Sie im Hinterkopf behalten, dass viele Steuerelemente, beispielsweise die von der Klasse `TextBoxBase` abgeleiteten Steuerelemente `TextBox` und `RichTextBox`, bereits von sich aus eine Programmierschnittstelle oder zumindest eine Tastaturschnittstelle für die Zwischenablage mitbringen. Die Implementierung einer Menü- bzw. Symbolleistenschnittstelle für die Zwischenablagenfunktionalität kann daher vom Prinzip her direkt mit den entsprechenden Methoden arbeiten. Kommt als Quell- oder Zielobjekt nur ein Textfeld `textBox1` in Frage, könnte die Behandlung der Menübefehle schlicht so aussehen:

```
#if Variante1
private void menuCopy_Click(object sender, System.EventArgs e)
{
  textBox1.Copy();
}
```

```
private void menuCut_Click(object sender, System.EventArgs e)
{
  textBox1.Cut();
}
private void menuPaste_Click(object sender, System.EventArgs e)
{
  textBox1.Paste();
}
#endif
```

Kommen mehrere Textfelder als Quelle oder Ziel in Frage, müsste man das betroffene Steuerelement irgendwie ausfindig machen und beispielsweise nach dem folgenden Muster ansprechen:

```
#if Variante2
private void menuCopy_Click(object sender, System.EventArgs e)
{
  try
  {
    TextBoxBase tbb = (TextBoxBase)ActiveControl;
    tbb.Copy();
  }
  catch  {}
}
...
#endif
```

Der Königsweg ist allerdings die folgende Formulierung:

```
#if Variante3
private void menuCopy_Click(object sender, System.EventArgs e)
{
  SendKeys.Send("^C");
}
private void menuCut_Click(object sender, System.EventArgs e)
{
  SendKeys.Send("^X");
}
private void menuPaste_Click(object sender, System.EventArgs e)
{
  SendKeys.Send("^V");
}
#endif
```

Der Code schickt mittels `SendKeys.Send()` eine Tastatureingabe (Strg+C, Strg+X, Strg+V) an das Steuerelement, das den Fokus besitzt, und überlässt alles Weitere dem Steuerelementobjekt. (Die syntaktischen Details, wie das `string`-Argument für `Send()` aussehen muss, finden Sie in guter Darstellung in der Online-Hilfe). Wenn dieses Zwischenablagenfunktionalität implementiert, verarbeitet es das Ereignis, ansonsten eben nicht. Auf diese Weise können Sie alle Steuerelemente erreichen, die von sich aus mit der Zwischenablage zusammenarbeiten, beispielsweise also auch Listen.

Der soeben vorgestellte Code findet sich in dem Projekt *Zwischenablage*. Sie können damit – wohlgemerkt über die Menüschnittstelle – Text von Textfeld zu Textfeld sowohl innerhalb einer Anwendung als auch anwendungsübergreifend austauschen. Die dritte Variante bezieht auch das auf dem Formular befindliche `NumericUpDown`-Steuerelement mit ein.

Um zwischen den einzelnen Varianten umzuschalten, passen Sie die `#define`-Anweisung in der ersten Zeile des Codes an.

18.1.2 Die Zwischenablage direkt ansprechen

In nicht allen Fällen lässt sich die tatsächliche Interaktion mit der Zwischenablage an ein Steuerelement delegieren. Oft müssen spezifische Inhalte gewissermaßen »manuell« in die Zwischenablage einlagert bzw. daraus entnommen werden, beispielsweise, wenn es um die Implementierung der Zwischenablagenfunktionalität für eigene Steuerelement- oder Formularklassen geht – und erst recht, wenn komplexe Inhalte, etwa ein Datensatz, über mehrere Steuerelemente verteilt bzw. aufzuteilen sind.

Die Klasse `Clipboard` stellt nur zwei statische Methoden bereit, `SetDataObject()` und `GetDataObject()`, deren Bezeichner schon fast eine klarere Sprache sprechen als ihre Prototypen:

```
public static void SetDataObject(object data);
public static void SetDataObject(object data, bool copy);
public static IDataObject GetDataObject();
```

Tabelle 18.1: Methoden der Schnittstelle `DataObject`

Methode	Bedeutung
`object GetData()`	Liefert die Daten aus dem Objekt in dem gewünschten Datenformat bzw. Datentyp. Falls das Datenformat nicht verfügbar ist bzw. eine Konvertierung in den gewünschten Datentyp nicht möglich ist, ist der Rückgabewert `null`.
`bool GetDataPresent()`	Liefert eine Aussage darüber, ob die von dem Objekt repräsentierten Daten in dem gewünschten Datenformat bzw. Datentyp verfügbar sind.
`string[] GetFormats()`	Liefert eine Auflistung der Datenformate, in denen das Objekt den repräsentierten Wert liefern kann.
`void SetData()`	Lagert Daten (in Form eines Objekts) in das Objekt ein. Diese Methode existiert in vier überladenen Varianten. Die einparametrige Variante übernimmt das angegebene Objekt und benutzt dessen Typinformation. Die beiden zweiparametrigen Varianten ermöglichen zusätzlich die explizite Angabe eines Datenformats als `string`-Wert oder eines Datentyps als `Type`-Wert. Die dreiparametrige Variante erlaubt weiterhin die Angabe eines `bool`-Werts, der ausdrückt, ob eine Konvertierung in kompatible Formate zulässig ist.

Daten einlagern – SetDataObject()

Um etwas in die Zwischenablage hineinzustecken, rufen Sie die Methode `SetDataObject()` auf und übergeben dieser ein Objekt, das vom Prinzip her einen beliebigen (!) – also auch eigenen Datentyp – tragen darf, solange dieser serialisierbar ist. Wenn Sie wollen, dass dieses Objekt auch nach Beendigung Ihrer Anwendung noch erhalten bleibt, verwenden Sie die zweiparametrige Überladung und versorgen den zweiten Parameter mit `true`. (Vielleicht fühlen Sie sich nun an die Rückfragen einiger Programme wie Photoshop oder Corel Draw erinnert, die beim Schließen einen entsprechenden Dialog hervorbringen.) Ist der zweite Parameter `false` oder fehlt er, wird das in die Zwischenablage übertragene Objekt bei Programmende mit abgebaut – und die Zwischenablage geleert.

Das wirft natürlich sofort die Frage auf, ob die Zwischenablage nun eine Referenz oder einen Klon des Objekts speichert. Die Antwort dürfte nicht überraschen. Es ist beides möglich.

Die Zwischenablage hat somit zwei Operationsmodi:

- *frühe Wertbindung* – in diesem Modus (zweiter Parameter des `SetDataObject()`-Aufrufs ist `true`) generiert `SetDataObject()` einen echten Klon, so dass `GetDataObject()` das Objekt in dem Zustand liefert, wie es eingelagert wurde.
- *späte Wertbindung* – in diesem Modus (zweiter Parameter des `SetDataObject()`-Aufrufs fehlt oder ist `false`) lagert `SetDataObject()` bei einem Verweistyp nur eine Referenz auf das Objekt ein, so dass `GetDataObject()` das Objekt in seinem jeweils aktuellen Zustand liefert. Werttypen werden hingegen kopiert. (Natürlich ist `string` mal wieder die rühmliche Ausnahme, weil letztlich ja die Implementierung der Zuweisungsoperation für den jeweiligen Datentyp darüber entscheidet, ob ein Klon oder eine Referenz bei der Zuweisung herauskommt.)

Der Aufruf von `SetDataObject()` für Textfelder sieht so aus.

```
private void menuClone_Click(object sender, System.EventArgs e)
{
  menuClone.Checked = !menuClone.Checked;
}

#if Variante4
private void menuCopy_Click(object sender, System.EventArgs e)
{
  Control c = ActiveControl;
  Clipboard.SetDataObject(c.Text.ToUpper(), menuClone.Checked);
}
```

```
private void menuCut_Click(object sender, System.EventArgs e)
{
  Control c = ActiveControl;
  Clipboard.SetDataObject(c.Text.ToUpper(), menuClone.Checked);
  c.Text = "";
}
```

Die `Checked`-Eigenschaft des Menüeintrags `menuClone` gibt den Operationsmodus vor, der angesichts des eingelagerten `string`-Werts allerdings nur beim Programmende Wirkung entfaltet (String-Werte werden ja grundsätzlich kopiert). Da der Code mit der `Text`-Eigenschaft des Steuerelements mit dem Eingabefokus arbeitet, bleibt das `NumericUpDown`-Steuerelement außen vor.

GetDataObject()

Um etwas aus der Zwischenablage herauszuholen, rufen Sie die Methode `GetDataObject()` auf. Sie liefert eine Instanz der Schnittstelle `IDataObject`, die ihrerseits drei in diesem Zusammenhang wichtige Methoden bereitstellt (Tabelle 18.1). `GetFormats()` vermittelt einen Überblick über die verfügbaren Datenformate und Datentypen, wobei unter »Datenformat« eine Familie von Datentypen zu verstehen ist. Die Klasse `DataFormats` definiert eine Reihe von `string`-Konstanten für Datenformate, die das von der Zwischenablage standardmäßig für die Einlagerung von Objekten verwendete Container-Objekt des Typs `DataObject` von sich aus versteht.

Die direkt von `object` *abstammende Klasse* `DataObject` *implementiert die Schnittstelle* `IDataObject` *und steht vom Prinzip her als Basisklasse für die Implementierung eigener Datenformate und Containerkonzepte (beispielsweise für die komprimierte Serialisierung) zur Verfügung.*



Häufig anzutreffende `DataFormats`-Konstanten sind: `Text`, `FileDrop`, `Bitmap`, `Html`, `Rtf`, `Dib`, `Tiff`, `EnhancedMetaFile`, `CommaSeparatedValue`, `UniCodeText` und `WaveAudio`.

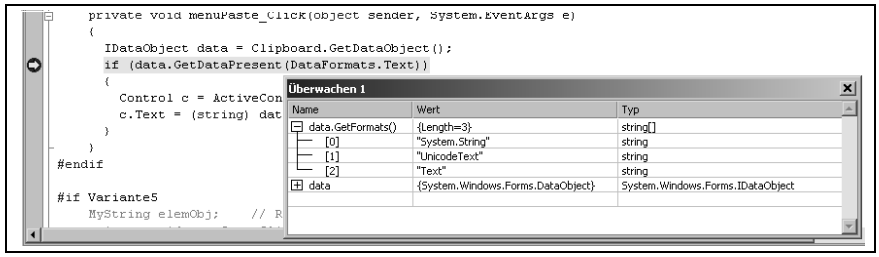


Abbildung 18.1: Ergebnis von `GetFormats()` für das Datenformat `Text`, nachdem zuvor ein String in die Zwischenablage kopiert wurde.

Damit Sie eine Vorstellung davon erhalten, was die `GetFormats()`-Methode liefert, setzen Sie auf die `if`-Anweisung des folgenden Codes einen Haltepunkt und sehen sich das `data`-Objekt in einem Überwachungsfenster des Debuggers einmal genauer an – freilich erst, nachdem Sie etwas in die Zwischenablage hingepackt haben (Abbildung 18.1 und Abbildung 18.2).

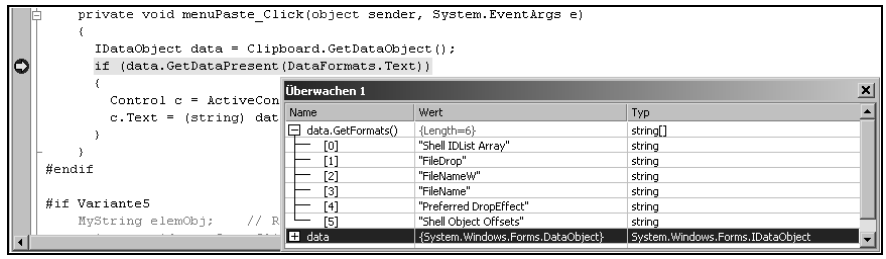


Abbildung 18.2: Ergebnis von `GetFormats()` für das Datenformat `FileDrop`, nachdem zuvor einige Dateien über den Windows-Explorer in die Zwischenablage kopiert wurden.

```
private void menuPaste_Click(object sender, System.EventArgs e)
{
  IDataObject data = Clipboard.GetDataObject();
  if (data.GetDataPresent(DataFormats.Text))
  {
    Control c = ActiveControl;
    c.Text = (string) data.GetData(DataFormats.Text);
  }
}
#endif
```

Nachdem ein Zielobjekt im Allgemeinen nicht wissen kann, welche Inhalte und Datenformate (bzw. Datentypen) das Quellobjekt in die Zwischenablage gepackt hat, muss es sich zuerst darüber informieren, inwieweit das erwartete Datenformat überhaupt verfügbar ist. Falls das Objekt ohnehin nur mit einem bestimmten Datenformat bzw. Datentyp etwas anfangen kann, können Sie sich auch den `GetFormats()`-Aufruf und die Auswertung des Ergebnisses sparen und stattdessen gleich mit der `GetDataPresent()`-Methode prüfen, ob das `IDataObject`-Objekt die Daten in dem gewünschten Datenformat oder Datentyp bereitstellen kann.

Referenztypen und eigene Datentypen

Wie bereits erwähnt, schluckt die Zwischenablage auch eigene Datentypen, sofern diese serialisierbar sind. Die folgende, fünfte Variante des Codes demonstriert die frühe und die späte Wertbindung für Referenztypen am Beispiel eines eigenen `class`-Datentyps. Anstatt der verwendeten Klasse `MyString`, deren Definition Sie gleich im nächsten Abschnitt finden, können Sie alles Mögliche in ein `DataObject`-Objekt packen, angefangen von `string`-Arrays bis hin zu Auflistungen beliebiger Objekte mit serialisierbaren Datentypen – auch wenn diese Datentypen ihrerseits wieder Auflistungen sind.

```
#using ClipboardClasses
#if Variante5
MyString elemObj;    // Referenztyp
private void menuCopy_Click(object sender, System.EventArgs e)
{
  if(elemObj == null)   // instanziiert?
  {
    elemObj = new MyString();
```

```
    menuCut.Text = "&Ref-Wert ändern";
  }
  elemObj.Elem = ActiveControl.Text;
Clipboard.SetDataObject(elemObj, menuClone.Checked);
}
// Achtung: veränderte Semantik. Ändert Wert des Reftyps elemObj,
// was ggf. den Inhalt der Zwischenablage ändert
private void menuCut_Click(object sender, System.EventArgs e)
{
  if (menuCut.Text == "&Ausschneiden")
  {
    elemObj = new MyString();
    menuCut.Text = "&Ref-Wert ändern"
    elemObj.Elem = ActiveControl.Text;
    Clipboard.SetDataObject(elemObj, menuClone.Checked);
    ActiveControl.Text = "";
  }
  else
    elemObj.Elem = ActiveControl.Text;
}
private void menuPaste_Click(object sender, System.EventArgs e)
{
  IDataObject data = Clipboard.GetDataObject();
  if (data.GetDataPresent(typeof(MyString)))
  {
    MyString ms = (MyString) data.GetData(typeof(MyString));
    ActiveControl.Text = ms.Elem;
    int ten = ms.Len;
  }
}
```

Eigene Datentypen anwendungsübergreifend verwenden

Für den anwendungsinternen Bedarf kann `MyString` vom Prinzip her auch ein privater Datentyp der Klasse `Form1` sein. Wenn Sie einen Datenaustausch mit anderen Anwendungen durchführen wollen, brauchen Sie auf jeden Fall eine eigene Assembly mit dem Ausgabetyp *Klassenbibliothek* (*.dll*). Dazu gehen Sie wie folgt vor:

1. Fügen Sie der Projektmappe (einer der Anwendungen) ein eigenständiges Projekt mit dem Typ *Klassenbibliothek* hinzu und definieren Sie darin alle Klassen, deren Objekte Sie anwendungsübergreifend per Zwischenablage (oder Drag&Drop) transferieren wollen. Kompilieren Sie das Projekt, um die Assembly der Klassenbibliothek zu erhalten.
2. Fügen Sie in jede Anwendung, die von diesen Datentypen Gebrauch macht, einen Verweis auf die Assembly ein. Eine Anleitung, wie Sie dies machen und welche Möglichkeiten Sie dabei haben (Stichworte: »starke Namen« und »Assembly Cache«), finden Sie beispielsweise im Abschnitt »Eine Steuerelementbibliothek in eine bestehende Projektmappe einfügen« (Seite 666).

Von nun an können Sie die Datentypen in allen Projekten verwenden und nach Herzenslust über die Zwischenablage transferieren. Um die importierten Datentypen ohne Namensraumzusätze ansprechen zu können, empfiehlt sich eine `using`-Direktive für den jeweiligen Namensraum. Hier die noch ausstehende Definition der Klasse `MyString`. Beachten Sie bitte das `Serializable`-Attribut:

```
using System;
namespace ClipboardClasses
{
  [Serializable]
  public class MyString
  {
    public string Elem = "";
    public int Len  // Eigenschaft
    {
      get { return Elem.Length;}
    }
  }
}
```

Vorteile der späten Wertbindung

Wenn Sie diesen Code testen oder auch nur durchdenken, achten Sie bitte darauf, dass der Menübefehl AUSSCHNEIDEN nach dem ersten Aufruf des KOPIEREN-Befehls nicht nur eine andere Beschriftung, sondern auch eine veränderte Semantik erhält. Trägt der Menübefehl KLON kein Häkchen, ändert der Code aufgrund der späten Wertbindung den »Inhalt« der Zwischenablage, indem er nur den Wert des ursprünglich eingelagerten Objekts manipuliert – und zwar *ohne* Aufruf der `SetDataObject()`-Methode.

Dieses Verfahren ist zu empfehlen, wenn große Datenmengen oder viele verschiedene Repräsentationen eines Objekts (dazu gleich noch mehr) bereitgestellt werden müssen. Die Eigenschaft `Len` der Beispielklasse `MyString` ist nicht nur der Form halber da: Wie sich mit einem darauf gesetzten Haltepunkt zeigen lässt, kommt der `get`-Accessor tatsächlich erst im Zuge von `menuPaste_Click()` zur Ausführung, also immer dann, wenn das Objekt aus der Zwischenablage kopiert und tatsächlich ein Zugriff auf die Eigenschaft erfolgt. In der Praxis heißt das, dass der Datentransfer auf die Daten und Datenformate beschränkt bleiben kann, die das Zielobjekt auch benötigt – und dass das Quellobjekt so lange als Server für die Zwischenablage zur Verfügung bleiben muss, bis diese einen anderen Inhalt erhält und die Referenz den Weg alles Irdischen geht. Nicht zuletzt auch aus diesem Grund fragen viele Anwendungen, ob der Inhalt der Zwischenablage nach Programmende erhalten bleiben soll und packen vor ihrem Ableben gegebenenfalls noch einen Klon hinein.

Mehrere Datenformate und Datentypen en bloc bereitstellen

Der Aufruf der `SetDataObject()`-Methode im bisher vorgestellten Beispielcode war noch nicht die ganze Wahrheit. Tatsächlich instanziiert die `SetDataObject()`-Methode ein neues `DataObject`-Objekt und packt das im ersten Parameter übergebene Objekt über einen Aufruf der `IDataObject`-Methode `SetData()` unter Angabe des entsprechenden Datenformats hinein – vorausgesetzt natürlich, dass dieses nicht bereits seinerseits schon ein `DataObject`-Objekt ist.

Wie Sie sicher schon vermutet haben, liegt hier der Ansatzpunkt für die Bereitstellung unterschiedlicher Datenformate: Anstatt wie bisher nur ein Objekt per `SetDataObject()` in die Zwischenablage einzulagern, haben Sie auch die Möglichkeit, das Bündel selbst zu schnüren, indem Sie der `SetDataObject()`-Methode ein zuvor fertig bepacktes Container-Objekt des Typs `DataObject` übergeben.

Die sechste Variante des Beispielcodes demonstriert diesen Weg anhand der beiden `PictureBox`-Steuerelemente des Formulars, über deren Sinn und Zweck sich die vorangehenden Abschnitte fröhlich ausgeschwiegen haben. Die Hilfsmethode `packDataObject()` packt zwei Objekte, ein `string`-Objekt und das `Image`-Objekt des oberen `PictureBox`-Steuerelements zusammen in das bereitgestellte `DataObject`-Objekt, das danach via `SetDataObject()` in die Zwischenablage wandert.

```
#if Variante6
private void menuCopy_Click(object sender, System.EventArgs e)
{
  packClipboard();
}
private void packClipboard()
{
  DataObject data = new DataObject();
  Control c = ActiveControl;
  data.SetData(c.Text);
  data.SetData(pictureBox1.Image);
  Clipboard.SetDataObject(data, menuClone.Checked);
}
private void menuCut_Click(object sender, System.EventArgs e)
{
  packClipboard();
  ActiveControl.Text = "";
  pictureBox1.Image = null;
}
private void menuPaste_Click(object sender, System.EventArgs e)
{
  IDataObject data = Clipboard.GetDataObject();
  if (data.GetDataPresent(DataFormats.Text))
  {
    Control c = ActiveControl;
    c.Text = (string) data.GetData(DataFormats.Text);
  }
```

```
    if (data.GetDataPresent(DataFormats.Bitmap))
    {
      pictureBox2.Image = (Image) data.GetData(DataFormats.Bitmap);
    }
  }
#endif
```

`menuPaste_Click()` holt die beiden Objekte nach Möglichkeit wieder heraus, steckt das Bild aber in die untere Picturebox. Obwohl die Zwischenablage bei diesem Experiment ein höchst spezielles Format enthält – nämlich die Kombination von Text und Grafik, kann der Text von jedem Client gelesen werden, der einen String erwartet, und das Bild von jedem Client, der mit Bitmaps zurechkommt. Umgekehrt kann die Demo-Anwendung auch jeden Text und jedes Bild anzeigen. Probieren Sie es doch einmal mit einem Screenshot des Anwendungsfensters (Alt+PrintScreen) oder des gesamten Bildschirms (PrintScreen).



Derartige Kombinationen sind alles andere als ungewöhnlich: Microsoft Excel stellt in die Zwischenablage kopierte Ausschnitte von Tabellenblättern beispielsweise unter anderem als Text sowie als (notwendigerweise statische) Bitmap vor – und natürlich im programmeigenen Format zum Einfügen in andere (Excel-)Tabellenblätter.

Bestehende Inhalte der Zwischenablage erhalten

Die allerwenigsten Anwendungen gehen sorgsam mit dem Inhalt der Zwischenablage um: Meistens reicht bereits ein versehentlicher Druck auf Strg+C um eine zuvor mühsam zusammengestellte Liste durch ein einzelnes Leerzeichen zu ersetzen. Anders gesagt: Ablageoperationen in die Zwischenablage geschehen meist im Blindflug – was aber gerade bei Anwendungen, die mit exotischen Datenformaten kommunizieren, nicht sein müsste.

Leider lässt sich der Inhalt der Zwischenablage nicht einfach nachträglich noch über die `SetData()`-Methode des von `GetDataObject()` gelieferten `IDataObject`-Schnittstellenobjekts erweitern, denn ihr Aufruf geht schlicht ins Leere. Der steinigere Weg, der eine Kopie der gesamten Zwischenablage in ein neues `DataObject`-Objekt vorsieht, ist so steinig auch wieder nicht. Zumindest beschwert sich das `DataObject`-Objekt nicht darüber, wenn ein Datenformat mehrfach eingelagert wird, was dann ja passieren kann. Es überschreibt den neuen Inhalt stoisch mit dem alten – und erspart damit die Filterung der Datenformate. (Es bleibt Ihnen natürlich freigestellt, sich um solche Filterungen zu kümmern, sie spart Ressourcen und Laufzeit.)

Die nicht überschreibende Version von `packClipboard()` sieht damit so aus:

```
#if Variante7
...
private void packClipboard()
{
  IDataObject idata = Clipboard.GetDataObject();
  DataObject data = new DataObject();
  foreach(string format in idata.GetFormats())
    data.SetData(format, false, idata.GetData(format));
  Control c = ActiveControl;
  if (c.Text != "")
    data.SetData(DataFormats.Text, c.Text);
  if (pictureBox1.Image != null)
    data.SetData(DataFormats.Bitmap, pictureBox1.Image);
  Clipboard.SetDataObject(data, menuClone.Checked);
}
...
#endif
```



Abbildung 18.3: Formular der Anwendung Zwischenablagen-Demo in der Variante 7

Übung

Implementieren Sie eine Variante 8 des Projekts, die das Datenformat `FileDrop` des Windows-Explorers unterstützt. Beginnen Sie mit der Implementierung für das Zielobjekt und ergänzen Sie dann die für das Quellobjekt (prüfen Sie auch die Korrektheit von Dateipfaden):

- Erstes Ziel ist, dass Sie eine Sammlung von Dateien, die Sie vom Explorer aus in die Zwischenablage kopieren, in das jeweiligen Textfeld einfügen können (nur Dateinamen samt Pfad).
- Zweites Ziel ist, dass Sie eine bearbeitete (= teilweise gelöschte) Auswahl dieser Dateinamen (im gleichen Datenformat) von der Anwendung aus zurück in die Zwischenlage kopieren.

Wenn Sie alles richtig gemacht haben, müssten Sie wiederum im Explorer per BEARBEITEN/KOPIEREN Kopien nur der ausgewählten Dateien anfertigen können – am besten in ein leeres Verzeichnis.

18.2 Drag&Drop

Wenn Sie den vorigen Abschnitt über die Zwischenablage gelesen haben, sind Sie gut vorbereitet auf Drag&Drop. Falls nicht, sollten Sie dies, wenn nicht zuerst, so auf jeden Fall begleitend tun. Er vermittelt den grundlegenden Zusammenhang zwischen den Klassen `DataObject` und `DataFormats` und stellt auch die Techniken der *frühen* und der *späten Wertbindung* vor. Diesen Hintergrund brauchen Sie für ein tiefer gehendes Verständnis von Drag&Drop auf jeden Fall.

Der Rest ist schnell erzählt und betrifft im Wesentlichen den (bei der Zwischenablage nur mittelbar vorhandenen) »Handshake« zwischen Quellobjekt und Zielobjekt sowie das visuelle Feedback für den Benutzer.

18.2.1 Ablauf der Operation

Die Rollenverteilung ist klar: Das Quellobjekt initiiert einen Ziehvorgang (Drag) – meist als Reaktion auf ein `MouseDown`-Ereignis – durch einen Aufruf seiner Methode `DoDragDrop()`. (Dieser Aufruf ist synchron und kehrt erst zurück, wenn die Operation abgeschlossen ist.) Dabei übergibt es ein Objekt eines beliebigen serialisierbaren Datentyps oder aber ein fertig gepacktes `DataObject`-Objekt sowie einen Bitvektor des `enum`-Typs `DragDropEffects`, der die zulässigen Operationen ausdrückt. Wie das »Packen« im Einzelnen vor sich geht und welche Möglichkeiten Sie dabei haben, auch exotische Datenformate zu übertragen, was gerade bei Drag&Drop ein häufiger Anwendungsfall ist, finden Sie im vorigen Abschnitt »Zwischenablage« ausführlichst beschrieben. Der Bitvektor ist meist eine Oder-Kombination der vom Explorer her bekannten Operationen `Copy`, `Link` und `Move`. Darüber hinaus gibt es noch die selten benutzte Operation `Scroll`. Die `DragDropEffects`-Aufzählung enthält zudem noch die Werte `All` und `None`, deren Bedeutung sich über die Namensgebung erschließt.

```
// in der Klasse des Quellobjekts
private void textBoxSource_MouseDown(object sender, MouseEventArgs e)
{
  // Containerobjekt leitet DD ein!
  if (DoDragDrop(textBoxSource.Text, DragDropEffects.All) ==
      DragDropEffects.Move)
    textBoxSource.Text = "";
}
```

Einladung zum Drag&Drop

Da während des Ziehvorgangs noch nicht klar ist, welches Objekt tatsächlich das spätere Zielobjekt sein wird, benachrichtigt das System jedes als OLE-Client in Frage kommende Objekt, wenn die Maus dessen Fensterbereich betritt (`DragEnter`), überstreicht (`DragOver`) und verlässt (`DragLeave`). Ein Objekt kommt nur dann als Zielobjekt in Frage, wenn seine `AllowDrop`-Eigenschaft `true` ist.

```
// in der Klasse des Zielobjekts
private void InitializeComponent()
{
  ...
  this.textBoxTarget.AllowDrop = true;
  this.textBoxTarget.DragOver +=
    new System.Windows.Forms.DragEventHandler(textBoxTarget_DragOver);
  this.textBoxTarget.DragDrop +=
    new System.Windows.Forms.DragEventHandler(textBoxTarget_DragDrop);
  this.textBoxTarget.DragEnter +=
    new System.Windows.Forms.DragEventHandler(textBoxTarget_DragEnter);
 ...
}
```

Kommunikation mit einem potenziellen Zielobjekt

Ein potenziell aufnahmebereiter Empfänger informiert sich dann beim `DataObject`-Objekt, in welchen Datenformaten dieses die Daten liefern kann und signalisiert – sofern ein verwertbares Format dabei ist – seine Bereitschaft, indem er seinerseits die von ihm unterstützten Operationen kundtut. Dazu weist er in Antwort auf eintreffende `DragEnter`- und `DragOver`-Ereignisse sowie – im Allgemeinen – unter Auswertung des Tastaturzustands `KeyState` der `Effect`-Eigenschaft des Ereignisobjekts die anvisierten Operation als `DragDropEffects`-Bitmaske zu. (Zur Analyse des Tastaturzustands gleich mehr.)

```
// in der Klasse des Zielobjekts
private void textBoxTarget_DragOver(object sender, DragEventArgs e)
{
  switch(e.KeyState)
  {
    case 0x05:  // Umschalt + Linke Maustaste
    case 0x09:  // Strg + Linke Maustaste
    case 0x0d:  // Umschalt + Strg Linke Maustaste
    case 0x21:  // Alt + Linke Maustaste
    case 0x25:  // Alt + Umschalt + Linke Maustaste
    case 0x29:  // Alt + Strg + Linke Maustaste
      e.Effect = DragDropEffects.Move;
      break;
    default:    // keine Taste und Alt + Strg + Umschalt
      e.Effect = DragDropEffects.Copy;
      break;
  }
}
```

Findet eine `DragOver`-Behandlung statt, ist *keine* `DragEnter`-Behandlung erforderlich und umgekehrt. Sie können aber auch für beide Ereignisse dieselbe Behandlungsmethode registrieren. Damit das `DragDrop`-Ereignis beim Lösen der Maustaste generiert wird, ist nur wichtig, dass die zuletzt aufgerufene `DragXxx`-Behandlungsmethode eine Operation ungleich `None` gesetzt hat. Welche Operation das Zielobjekt schlussendlich bei der `DragDrop`-Behandlung anfordert, hat damit übrigens nichts zu tun. Die gesetzte Operation wirkt sich nur darauf aus, welches visuelle Feedback (Cursorform) der Benutzer erhält.

Feedback an den Benutzer

Im Zuge der `DragOver`-Behandlung analysiert das Zielobjekt den Tastaturzustand und ordnet diesem die Operation zu, die es im Falle eines `DragDrop`-Ereignisses ausführen würde – oder eben keine: der Vorgabewert für `Effect` ist `None`. Wie der angeführte Beispielcode zeigt, erfordert die Auswertung der `KeyState`-Eigenschaft ein für die Programmierung mit .NET höchst selten anzutreffendes Verfahren. Mangels Konstanten müssen die Bitmasken für den Tastaturzustand als Zahlenwert zusammengestellt werden. Warum es keine `KeyStates`-Aufzählung gibt, die diese Bit-Gymnastik erspart, ist unverständlich. Nehmen Sie es als Rückblick in Zeiten »längst überholter« Programmierpraktiken und achten Sie einfach beim nächsten Komponenten-Update darauf, ob Microsoft darin Versäumtes nicht vielleicht nachgeholt hat.

Tabelle 18.2: Bitmasken für die Auswertung der `KeyState`-Eigenschaft

Bitmaske	Tastatur-/Mauszustand
0x01	Linke Maustaste gedrückt
0x02	Rechte Maustaste gedrückt
0x04	⇧ gedrückt
0x08	Strg gedrückt
0x10	Mittlere Maustaste gedrückt
0x20	Alt gedrückt

Es steht Ihnen frei, die Masken selbst zu definieren und den Code etwas sprechender zu gestalten:

```
// in der Klasse des Zielobjekts
[Flags]
enum KeyStates {None, MouseLeft, MouseRight, Shift = 4, Control = 8,
MouseMiddle = 16, Alt = 32};
```

```
private void textBoxTarget_DragOver(object sender,
System.Windows.Forms.DragEventArgs e)
{
  switch((KeyStates)e.KeyState & ~KeyStates.MouseLeft)
  {
    case KeyStates.Shift:
    case KeyStates.Control:
    case KeyStates.Shift | KeyStates.Control:
    case KeyStates.Alt:
    case KeyStates.Shift | KeyStates.Alt: Maustaste
    case KeyStates.Alt | KeyStates.Control:
      e.Effect = DragDropEffects.Move;
      break;
    default:
      e.Effect = DragDropEffects.Copy;
      break;
  }
}
```

GiveFeedBack-Ereignis

Das `DataObject`-Objekt kombiniert die vom Quellobjekt und Zielobjekt gesetzten Bitvektoren für die Operationen über ein bitweises AND und gibt, indem es das `GiveFeedback`-Ereignis signalisiert, dem Quellobjekt im Gegenzug die Möglichkeit, eine eigene Cursorform zu setzen, um dem Benutzer die jeweils ausgewählte Operation zu visualisieren. Falls das Quellobjekt keine Behandlungsroutine dafür registriert oder die Eigenschaft `UseDefaultCursors` auf `true` setzt, setzt das `DataObject`-Objekt von sich aus die standardmäßigen Cursorformen des Systems. Der folgende Code zeigt eine mögliche Implementierung der `GiveFeedback`-Behandlung:

```
// in der Klasse des Quellobjekts
private void textBoxSource_GiveFeedback(object sender,
  System.Windows.Forms.GiveFeedbackEventArgs e)
{
  e.UseDefaultCursors = true;
#if OwnCursors
  e.UseDefaultCursors = false;
  switch(e.Effect)
  {
    case DragDropEffects.Copy:
      Cursor = cur[0];
      break;
    case DragDropEffects.Move:
      Cursor = cur[1];
      break;
    case DragDropEffects.Link:
      Cursor = cur[3];
      break;
    default:
      Cursor = cur[2];
      break;
  }
#endif
```

Vielleicht werden Sie sich bei dem Code fragen, wo denn nun die unmittelbar vor dem Ablegen des Objekts eingestellte Cursor-Form wieder auf den Standardwert zurückgesetzt wird. Der `default`-Zweig kann diese Aufgabe nicht übernehmen, da es keinen Effekt gibt, der dies ausdrücken würde. Außerdem tritt das `GiveFeedback`-Ereignis im Übrigen nach dem Ablegen des Objekts auch nicht mehr auf. Die Antwort ist ein echtes Aha-Erlebnis und bedarf wohl keines weiteren Kommentars:

```
// in der Klasse des Quellobjekts
private void textBoxSource_MouseDown(object sender,
  System.Windows.Forms.MouseEventArgs e)
{
  // Containerobjekt leitet DD ein!
  if (DoDragDrop(textBoxSource.Text, DragDropEffects.All) ==
      DragDropEffects.Move)
    textBoxSource.Text = ""; // wird erst nach Abschluss der gesamten
  Cursor = Cursors.Default;  // DD-Operation ausgeführt
}
```

Cursorformen als Ressourcen einlesen

Das `cur`-Array initialisiert der Konstruktor – durch Einlesen von eingebetteten Ressourcen. Hierbei handelt es sich um ein gängiges Verfahren, das Sie für alle möglichen Ressourcen der Typen *.bmp, .gif, .jpg, .jpe, .jpeg, .ico* und *.cur* anwenden können. Es hat den Vorteil, dass Sie – mit Blick auf eine etwaige Auslieferung – für Ihre Anwendung nicht etliche externe Dateien bereitstellen, verwalten und zusammenhalten müssen, zumal die Anwendung vielleicht ja nicht mehr funktioniert, wenn versehentlich eine davon gelöscht, verschoben, verändert oder auch nur umbenannt wurde.

Die notwendigen Schritte zum Einbetten von Ressourcen der genannten Typen sind:

1. Fügen Sie die Dateien über den Menübefehl PROJEKT/VORHANDENES ELEMENT HINZUFÜGEN in das Projekt ein. Visual Studio erzeugt dabei eine Kopie aller Dateien im Projektverzeichnis. (Sie verlieren also nicht die Originaldatei, wenn Sie später wieder welche davon löschen.)
2. Setzen Sie im EIGENSCHAFTEN-Fenster die `BuildAction`-Eigenschaft aller Ressourcedateien auf *Eingebettete Ressource*.
3. Lesen Sie die Ressourcen nach dem folgenden Muster ein, wobei Sie peinlichst auf die Schreibweise des Dateinamens achten müssen, da C# auch hier penibel zwischen Groß- und Kleinschreibung unterscheidet.

   ```
   cur = new Cursor[4];
   cur[0] = new Cursor(typeof(Form1),"H_CROSS.CUR");
   ...
   ```

   Die hier verwendete Überladung des Konstruktors existiert analog auch für die Datentypen `Bitmap` und `Icon`. Sie erwartet im ersten Parameter die

Typinformation einer beliebigen (!) Klasse, die dem Standardnamensraum der die Ressource bereitstellenden Assembly angehört. Dieser Namensraum lässt sich im EIGENSCHAFTEN-Dialog über die Eigenschaft `Standardnamespace` des jeweiligen Projekts setzen, dem die Ressource hinzugefügt wurde und muss gegebenenfalls nachträglich noch angepasst werden, wenn Sie den für das Codegerüst automatisch generierten Namensraum nicht beibehalten haben. Kurzum, er sollte mit dem Namensraum der im ersten Parameter genannten Klasse übereinstimmen, andernfalls erhalten Sie beim Versuch, die Ressourcen einzulesen, kuriose Ausnahmen mit wenig aussagekräftigen Fehlermeldungen.

QueryContinueDrag

Das Ereignis `QueryContinueDrag` gehört auch noch zum Feedbacksystem, wird aber im Allgemeinen nur selten behandelt. Tatsächlich wird es vom Formular gar nicht erst an das Quellobjekt weitergeleitet, sondern muss auf Formularebene (bzw. Container-Ebene) behandelt werden. Eine installierte Behandlungsroutine kommt in regelmäßigen Zeitabständen, etwa alle 50 Millisekunden zum Aufruf und kann den Vorgang von sich aus abbrechen, indem sie die `Action`-Eigenschaft des Ereignisobjekts auf den Wert `DragAction.Cancel` setzt. Das Ereignisobjekt transportiert zudem noch die Eigenschaften `KeyState` und `EscapePressed`. Letztere ist Grundlage für das Standardverhalten – und sollte auch für die Behandlungsmethode ein Kriterium für den sofortigen Abbruch sein:

```
// in der Klasse des Quellobjekts
private void textBoxSource_QueryContinueDrag(object sender,
  System.Windows.Forms.QueryContinueDragEventArgs e)
{
  if(e.EscapePressed)
  {
    e.Action = DragAction.Cancel;
    System.Console.WriteLine(
      "{0} Drag-Vorgang auf Benutzerwunsch abgebrochen", DateTime.Now);
  }
}
```

An weiteren Aktionen stehen noch `Continue` und `Drop` im Angebot. Beide werden allerdings nur in ganz speziellen Situationen, beispielsweise bei Implementierung von Eingabehilfen für Behinderte (Stichwort: Accessibility) oder beim Abspielen aufgezeichneter Vorgänge Anwendung finden.

Ablegen des Objekts

Sobald der Benutzer den gezogenen Inhalt auf einem empfangswilligen Zielobjekt ablegt, kann dieses damit beginnen, den Inhalt auszuwerten. Wie das im Einzelnen vor sich geht und welche Möglichkeiten Sie dabei haben, auch exotische Datenformate zu übertragen – was gerade bei Drag&Drop ja ein häufiger Anwendungsfall ist – finden Sie im vorigen Abschnitt »Zwischenablage« ausführlichst beschrieben.

Bei der DragDrop-Behandlung kann sich das Zielobjekt, wie schon angedeutet, über die während der DragOver- bzw. DragEnter-Behandlung zuletzt gesetzte Aktion hinwegsetzen und eine andere Aktion ausführen. Ob es sinnvoll ist, den Benutzer derartig zu verwirren, sei dahingestellt. Auf jeden Fall sollte das Zielobjekt bei einem Misslingen der Operation anzeigen, dass etwas schiefgegangen ist und die Operation None setzen. Das ist besonders wichtig, wenn der Benutzer eine Verschiebeoperation durchführen wollte und der Inhalt auf Seiten des Quellobjekt nun doch nicht gelöscht werden darf. Bei dem Datenformat DataFormats.Text mag dies vielleicht überflüssig erscheinen, bei anderen Datenformaten kann aber durchaus etwas schief gehen, und dies sollte das Quellobjekt in jedem Fall mitgeteilt bekommen.

```
private void textBoxTarget_DragDrop(object sender,
System.Windows.Forms.DragEventArgs e)
{
  if (e.Data.GetDataPresent(DataFormats.Text, true))
    try
    {
      textBoxTarget.Text = (string) e.Data.GetData(DataFormats.Text);
    }
  catch
    {
      e.Effect = DragDropEffects.None;
      return;
    }
}
```

Hier noch einmal der Code um den DoDragDrop()-Aufruf herum, der die Move-Operation abschließt:

```
// in der Klasse des Quellobjekts
private void textBoxSource_MouseDown(object sender, MouseEventArgs e)
{
  // Containerobjekt leitet DD ein!
  if (DoDragDrop(textBoxSource.Text, DragDropEffects.All) ==
      DragDropEffects.Move)
    textBoxSource.Text = "";
}
```

18.2.2 Zusammenfassung

Hier noch einmal ein schneller Überblick über die Abwicklung einer einfachen Drag&Drop-Operation ohne Schnörkel.

Aus Sicht des Quellobjekts

Grundsätzlich kann jedes Steuerelement eines Formulars und auch das Formular selbst die Rolle des Quellobjekts einnehmen. Die Aufgaben aufseiten des Quellobjekts sind:

1. *Einleiten der Drag&Drop-Operation* in Reaktion auf eine Benutzeraktion – im Einzelnen heißt das: Bereitstellen des zu übermittelnden Objekts und Aufruf der `DoDragDrop()`-Methode.
2. *Optional:* Für das Quellobjekt kann eine `GiveFeedback`-Behandlung erfolgen, wenn die Darstellung eigener Cursorformen erwünscht ist.
3. *Optional:* Für den Container (im Allgemeinen: das Formular) kann eine `QueryContinueDrag`-Behandlung erfolgen, wenn eine Ablaufkontrolle erwünscht ist – beispielsweise, um den Vorgang auch programmseitig abzubrechen.
4. *Löschen des übermittelten Objekts*, wenn eine `Move`-Operation ausgeführt wurde.

Aus Sicht des Zielobjekts

Damit ein Formular oder Steuerelement als Zielobjekt eines Drag&Drop-Vorgangs auftreten kann, müssen mindestens drei Voraussetzungen gegeben sein:

1. Die `AllowDrop`-Eigenschaft des Objekts muss `true` sein. Ist diese Eigenschaft `false`, erhält das Objekt keine `DragXxx`-Ereignisse und das Quellobjekt zeigt die Cursorform `Cursors.No` (Verbotsschild) an.
2. Für das Objekt muss eine Behandlungsmethode für das `DragEnter`- oder `DragDrop`-Ereignis registriert sein – wahlweise auch für beide, was jedoch wenig Sinn macht. Diese Methode muss eine Operation ungleich `None` setzen. Mit einer `DragEnter`-Behandlung kommen Sie aus, wenn das Zielobjekt ohnehin nur *eine* Operation (im Allgemeinen `Copy`) unterstützt, andernfalls muss eine `DragDrop`-Behandlung mit Unterscheidung des Tastatur- bzw. Mauszustands erfolgen, damit ein visuelles Feedback möglich ist.
3. Für das Objekt muss eine Behandlungsmethode für das `DragDrop`-Ereignis registriert sein, die das übermittelte `DataObject`-Objekt auswertet. Falls die Operation misslingt, sollte diese Methode die Operation `None` setzen.

Codebeispiel

Ein typisches Codebeispiel für eine einfache exotische Drag&Drop-Operation finden Sie ab Seite 630. Es demonstriert das Einfügen neuer Symbolschaltflächen in eine Symbolleiste per Drag&Drop.

Teil 4 Verteilte Programme und Webanwendungen

Kapitel 19: XML 747

Kapitel 20: Einführung in verteilte Programme mit .NET 809

Kapitel 21: ASP.NET allgemein 869

Kapitel 22: Webanwendungen 881

Kapitel 23: Webdienste 941

Dieser Teil hat mit reinen Windows-Anwendungen nur noch insoweit zu tun, als Formulare auch hier eines der möglichen Ein- und Ausgabemedien darstellen.

Nach einer Einführung in XML geht es erst um die von .NET für XML zur Verfügung gestellten Klassen; als nächstes stehen einfache Client-/Server-Systeme und dann Mehrschichtsysteme auf dem Programm, bevor dieser Buchteil dann auf die Programmierung mit dem Internet Explorer als Host, Mail und verteilte Datenbanksysteme mit ASP.NET eingeht.

Web-Formulare, die Server- und Clientseite von Webdiensten, beiderseitiges Caching und Zustandshaltung wären bei anderen Programmiersprachen genauso wie die Kommunikation über Firewalls hinweg mit SOAP Themen für einige wenige Gurus. Mit C# und der .NET-Laufzeitumgebung lassen sich dagegen nicht nur Clients, sondern auch Server ausgesprochen geradlinig implementieren.

19 XML

XML (Extensible Markup Language) ist eine der Basistechnologien von .NET, die nicht nur auf der Anwendungs-, sondern auch auf der Entwicklungsseite die Rolle eines tragenden Elements spielt: ADO.NET und SOAP basieren genauso auf XML wie die Konfigurationsdateien von VS.NET und die Inline-Dokumentation.

Dieses Kapitel bietet einen anwendungsorientierten Überblick über den Einsatz von XML in .NET.

Die maßgeblichen Referenzen für XML und verwandte Sprachen liefert `www.w3.org`. Sehr gute und umfassende Einführungen und Tutorials dazu finden Sie zum Beispiel unter `www.netzwelt.com/selfhtml/xml/index.htm` oder `www.w3schools.com`.

19.1 Einführung in XML

XML wurde im Februar 1998 vom World Wide Web Consortium W3C als Recommendation (Deutsch: Empfehlung) veröffentlicht und ist damit ein anerkannter Standard. Dass es in so kurzer Zeit zum »ASCII des Web« werden konnte, verdeutlicht die Notwendigkeit eines solchen Standards.

Die folgenden Abschnitte geben eine Einführung in XML. Wenn Sie mit diesen Sprachen schon vertraut sind, können Sie sie guten Gewissens überspringen.

19.1.1 Was ist XML?

XML ist eine Meta-Sprache zur Beschreibung von Daten. Das heißt, es ist eine Sprache zur Beschreibung von Sprachen zur Beschreibung von Daten. Scherz beiseite: Sie denken sich also selbst eine Beschreibung Ihrer Daten aus und formulieren diese Regeln dann in XML:

```
<DerDienst>Unser Dienst ist in <AnzahlLänder>9</AnzahlLänder> Ländern
verfügbar und kostet nur <PreisProMonat>9</PreisProMonat> Euro. Sie
erreichen uns von <VonUhr>9</VonUhr> Uhr bis <BisUhr>19</BisUhr> Uhr.</
DerDienst>
```

Durch XML bleibt die Bedeutung der Daten erhalten. So ist im obigen Beispiel jedem sofort die Bedeutung der Zahlen klar, weil sie schlicht und ergreifend daneben steht. Sollte der Beispiel-Dienst demnächst in zehn Ländern verfügbar sein, lässt sich diese Information in XML-Dokumenten automatisch aktualisieren. In allen anderen Dokumenten ist manuelles Suchen und Ersetzen angesagt.

Außerdem ist XML textbasiert, damit in jedem Editor darstellbar, und eventuelle Beschädigungen lassen sich relativ leicht reparieren. Dagegen reicht bei Binärformaten wie WinWord oder Excel ein gekipptes Bit für einen Totalverlust. Im Gegensatz zu diesen Formaten kann XML auch keine Makros enthalten und ist damit garantiert virenfrei.

Und schließlich ist XML ein Standard des World Wide Web Consortiums (W3C) und somit auf beliebigen Plattformen verfügbar. XML bietet sich also auch zum Austausch von Daten an.

Damit andere die Daten auswerten können, muss Einigkeit über ihre Bedeutung bestehen. Dazu treffen die Beteiligten entsprechende Absprachen. Ein Beispiel dafür ist WML (Wireless Markup Language), das WAP-Handies (Wireless Application Protocol) verwenden – eine Sprache, die vom WAP-Forum, einem Zusammenschluss von Industrieunternehmen, entwickelt wurde. Ein anderes Beispiel ist SVG (Scalable Vector Graphics), ein offizieller W3C-Standard für Vektorgrafiken.

Nachteilig an XML ist der Platzbedarf. Das ist bei der Ausstattung heutiger Rechner kein Problem hinsichtlich Speicher- und Plattenplatz, kann aber bei der Übertragung zu Performance-Problemen führen.

19.1.2 XML, XHTML, XSL, etc.: Wie hängt was zusammen?

XML ist eine Metasprache zur Beschreibung von Daten – wie das Beispiel des vorangehenden Abschnitts demonstriert hat, aber nicht unbedingt das Idealformat zur ihrer Darstellung. Deshalb werden die Daten zur Anzeige in ein anderes Format transformiert, und das ist oft HTML (HyperText Markup Language). Reines HTML ist relativ einfach, kann von jedem Browser dargestellt werden, produziert aber bei aufwändigerer Formatierung ebenfalls recht umfangreiche Datenmengen, weshalb man zur detaillierten Festlegung der Darstellung eine dritte Sprache benutzt: CSS (Cascading Style Sheets). CSS kann man sich als eine wiederverwendbare Zusammenfassung von Formatierungen vorstellen, analog zur Dokumentvorlage eines Textverarbeitungssystems. Die Transformation von XML zu HTML erledigt XSLT (Extensible Stylesheet Language Transformations). Das Ergebnis der Transformation kann aber auch ein neues XML-Dokument sein, ein Textdokument, ein PostScript-Dokument oder ein Dokument in einem beliebigen anderen Format.

Eigentlich braucht man zur Darstellung von XML nicht den Umweg über HTML und CSS zu gehen. Denn dafür sind die XSL-FO (XSL Formatting Objects) vorgesehen, die eine noch bessere Kontrolle der Darstellung erlauben als CSS. Insbesondere gewährleisten XSL-FO eine Entsprechung von Bildschirmdarstellung und Druck. Die XSL-FO bilden zusammen mit XSLT die XSL (Extensible Stylesheet Language), werden aber leider von Microsoft (noch?) nicht unterstützt.

Da jeder seine XML-Regeln selbst definieren darf, muss er prüfen können, ob ein XML-Dokument diesen Regeln entspricht. Dazu dient XSD (XML Schema Definition Language). Allerdings ist XSD erst auf dem Weg, das ursprüngliche Format DTD (Document Type Definition) zu ersetzen. DTD ist XSD zwar in den Ausdrucksmöglichkeiten unterlegen, aber es war eben zuerst da. XSD-Implementierungen werden gerade erst nachgereicht. In der Dokumentation von Visual Studio finden Sie auch noch Hinweise auf XDR (XML Data Reduced). Inhaltlich ist davon zwar vieles in XSD eingeflossen, XDR ist aber kein Standard und wird auch keiner werden.

XSL und XSD sind in XML beschrieben, jedes XSL- und jedes XSD-Dokument ist also gleichzeitig ein XML-Dokument und kann deshalb wie ein XML-Dokument ausgewertet bzw. verarbeitet werden. Um diese Möglichkeit auch für HTML nutzen zu können, wurde der HTML-Standard überarbeitet. Das Resultat ist XHTML. Die Unterschiede liegen vor allem im Wegfall von Varianten, sodass sich die Arbeit von Web-Designern und Browser-Programmierern wesentlich vereinfacht.

Das XML Informationset (Infoset) definiert, wie ein XML-Dokument zu interpretieren ist (Datenmodell).

Auch der Zugriff auf die in XML-Dokumenten enthaltenen Informationen ist standardisiert. Diese werden meistens im Document Object Model (DOM) dargestellt. Das ist eine Baumdarstellung, in der man sich von einem Element zum nächsten hangeln, Gruppen bilden, Elemente hinzufügen, verändern oder entfernen kann. Es ist also nicht nötig, sich mit der XML-Syntax herumzuschlagen oder mit riesigen Strings zu hantieren. Entsprechende Klassen sind in .NET enthalten, außerdem lassen sich externe XML-Prozessoren wie der des IE einbinden.

Neben dem DOM existiert noch das Simple API for XML (SAX). Während DOM-Prozessoren ein XML-Dokument vollständig im Speicher halten, lesen SAX-Prozessoren es Stück für Stück und lösen nach jedem Lesevorgang ein Ereignis aus. Bei sehr großen Dokumenten ergeben sich daraus potenziell Geschwindigkeitsvorteile, außerdem sinkt der Speicherverbrauch. Dafür muss sich das Programm, das die Ereignisse erhält, den aktuellen Zustand merken, was schnell sehr aufwändig wird. SAX ist kein offizieller Standard und wird in .NET nicht unterstützt.

In .NET sind dafür die `XmlTextReader`- und `XmlTextWriter`-Klassen enthalten, die ähnlich wie SAX-Prozessoren ein XML-Dokument Stück für Stück verarbeiten. Dabei führen sie die einzelnen Schritte aber erst auf Anforderung hin durch (»Pull-Modell«) und helfen so, die mit dem »Push-Modell« der SAX-Prozessoren verbundenen Komplikationen im erträglichen Rahmen zu halten.

XPath selektiert Teile eines XML-Dokuments, ist für XML-Dokumente also so etwas wie SQL für Datenbanken. Allerdings bietet XPath spezielle Ausdrücke der Art »ist-ein-Kind-von«, um die Relationen zwischen XML-Knoten zu beschreiben. XPath wird in XSLT und XPointer benutzt und als W3C Recommendation von den aktuellen XML-Prozessoren unterstützt.

XPointer dient in Zusammenarbeit mit XLink zur Identifikation von Teilen eines XML-Dokuments in URIs (Uniform Resource Identifier). Dadurch ergeben sich weit über die Hyperlink-Fähigkeiten von HTML hinausgehende Verbindungsmöglichkeiten. XPointer ist eine W3C Candidate Recommendation, XLink hat den Weg zur offiziellen Empfehlung des W3C bereits geschafft.

Neben den genannten Akronymen und Standards gibt es noch viele weitere, zum Beispiel XML Query oder XML Encryption, die in naher Zukunft praxisrelevant werden. Ein gelegentlicher Blick auf die Seite `www.w3.org` empfiehlt sich für Interessierte deshalb auf jeden Fall.

19.1.3 XML-Syntax

Ziel dieses Abschnittes ist es, einen Überblick über den Aufbau von XML-Dokumenten zu geben. Insbesondere zeigt er, welche Daten wie in XML verpackt und transportiert werden können. Dagegen fehlen Details wie zum Beispiel die Processing Instructions.

Weitere Informationen zur XML-Syntax finden Sie über die am Kapitelanfang genannten Links ins Internet.

Elemente und Attribute

Ein XML-Dokument besteht aus einem oder mehreren Elementen. Jedes Element ist durch so genannte Tags (Deutsch: Etikett, Anhänger) begrenzt, die aus frei wählbarem Text in spitzen Klammern bestehen. Dabei enthält das schließende Tag einen Schrägstrich, gefolgt vom gleichen Text wie im öffnenden Tag. Im Gegensatz zu HTML unterscheidet XML zwischen Groß- und Kleinschreibung. Im Beispiel vom Kapitelanfang enthält das Element *DerDienst* sowohl weitere Elemente als auch Daten.

```
<DerDienst>Unser Dienst ist in <AnzahlLänder>9</AnzahlLänder> Ländern
verfügbar und kostet nur <PreisProMonat>9</PreisProMonat> Euro. Sie
```

```
erreichen uns von <VonUhr>9</VonUhr> Uhr bis <BisUhr>19</BisUhr> Uhr.</
DerDienst>
```

Elementnamen müssen mit einem Buchstaben oder einem Unterstrich beginnen. Darauf kann eine beliebige Anzahl von Buchstaben, Ziffern, Unterstrichen, Bindestrichen oder Punkten folgen. Die Bezeichner dürfen jedoch nicht mit den Buchstaben »XML« beginnen, egal, ob groß oder klein geschrieben.

Ein Element kann neben Daten und anderen Elementen auch ATTRIBUTE enthalten, beides wird unter dem Begriff KNOTEN zusammengefasst. Attribute tauchen vor allem in stark strukturierten Dokumenten auf.

```
<Kontakt>
  <VorName>Jens</VorName>
  <NachName>Meier-Schmitz</NachName>
  <TelefonFax Vorwahl="01234" Einwahl="56" Telefon="78" Fax='79' />
</Kontakt>
```

Ein Attribut kann nur innerhalb eines Elements existieren und besteht aus einem Namen, dem Gleichheitszeichen und einem Wert in Anführungszeichen – wahlweise einfache oder doppelte. Damit sind allerdings ausschließlich die Zeichen 0x0027 bzw. 0x0022 gemeint, nicht irgendwelche typografischen Anführungszeichen wie Textverarbeitungsprogramme sie generieren. Für die Namen von Attributen gelten dieselben Regeln wie für Elementnamen, und in jedem Element darf ein Attributname nur einmal vorkommen. Die Reihenfolge von Attributen in einem Element ist gleichgültig und bei der Verarbeitung mit einem XML-Prozessor auch nicht mehr zu ermitteln. Alle Daten in Elementen und Attributen betrachtet der XML-Prozessor als Text.

Im zweiten Beispiel sehen Sie auch ein LEERES ELEMENT. Ein leeres Element enthält weder Daten noch andere Elemente, und deshalb kann es wie im Beispiel mit einem Schrägstrich geschlossen werden. Es wäre genauso zulässig, statt des Schrägstrichs ein schließendes Tag zu verwenden. Attribute zählen nicht als Inhalt eines Elements, man kann sie sich als Meta-Informationen zum Element vorstellen.

Die Zeilenumbrüche und Einrückungen im zweiten Beispiel dienen nur der besseren Lesbarkeit. Der XML-Prozessor kümmert sich nicht um sie.

Die Informationen in einem XML-Dokument haben immer die Form eines Baums. Ein Element kann also mehrere Unterelemente haben, aber nur ein übergeordnetes Element. Außerdem enthält jedes XML-Dokument automatisch ein WURZEL-ELEMENT (Englisch: root). Dieses Element kann neben Kommentar-Elementen (siehe unten) und Processing Instructions (Anweisungen für den XML-Prozessor) genau ein DOKUMENT-ELEMENT enthalten,

nämlich das Eltern-Element der eigentlichen Daten. Im zweiten Beispiel ist *Kontakt* das Dokument-Element, und wenn ein neues *Kontakt*-Element hinzukäme, müsste man beide wiederum in einem übergeordneten Element verpacken.

Elemente mit gleichem Namen können beliebig oft vorkommen, auch auf unterschiedlichen Hierarchieebenen. Die Elemente erscheinen bei der Verarbeitung in der Dokumentreihenfolge.

Elemente müssen korrekt verschachtelt sein, solche aus HTML gewohnten Konstruktionen sind also illegal.

```
Das ist <b>in <i>XML </b>illegal</i>.
```

In XML muss es so aussehen:

```
Das ist <b>in <i>XML </i></b><i>legal</i>.
```

Eine besondere Art des Elements sind Kommentare:

```
<!-- Das ist ein XML-Kommentar -->
```

Ein Kommentar-Element kann beliebigen Text enthalten, aus naheliegenden Gründen aber keine aufeinanderfolgenden Bindestriche und auch keinen Bindestrich am Ende. Es kann überall dort stehen, wo ein Element erlaubt ist, außerdem auch direkt unter dem Wurzel-Element. Das heißt natürlich auch, dass Kommentare nicht innerhalb eines Elements oder Attributs stehen dürfen.

Namensräume

Es kommt häufig vor, dass zwei XML-Dokumente den gleichen Elementnamen mit unterschiedlicher Bedeutung verwenden. Das Element *Preis* könnte zum Beispiel auch den Gewinn in einer Lotterie enthalten. Damit diese Elemente im selben Dokument benutzt werden können, gibt es das Konzept der Namensräume, das aus Programmiersprachen wie C++ oder C# bekannt ist.

Einen Namensraum deklarieren Sie als Attribut; er gilt für das entsprechende Element sowie alle seine Kind-Elemente. Dabei kann er in diesen Kind-Elementen auch lokal überschrieben werden. Die Deklaration sieht folgendermaßen aus, wobei in einem Element auch mehrere Namensräume deklariert werden können.

```
<EinElement Xmlns:Lotterie="MeinLotterieNamensraumUri">
```

Die Zugehörigkeit eines Elements zu einem Namensraum lässt sich am vorangestellten Namensraum-Namen erkennen.

```
<Lotterie:Preis>Teddybär</Lotterie:Preis>
```

Zusätzlich lässt sich ein Standard-Namensraum definieren.

```
<EinElement Xmlns="MeinNamensraumUri">
```

Jedes Element ohne vorangestellte Nennung eines Namensraums gehört automatisch zum Standard-Namensraum.

Zeichensätze in XML

Ist Ihnen aufgefallen, dass die im ersten Beispiel dieses Kapitels verwendeten Tags Umlaute enthalten? In einer perfekten Welt wäre das auch kein Problem – aber leider kommt beim Öffnen dieses Beispiels im Internet Explorer nichts weiter heraus als eine Fehlermeldung (siehe Abbildung 19.1).

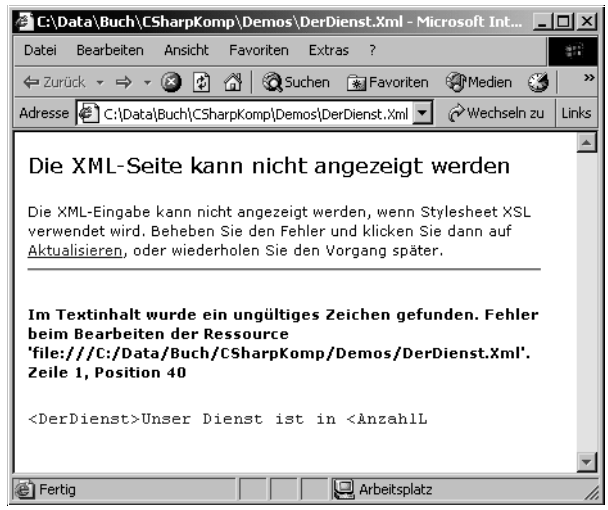


Abbildung 19.1: Umlaute in XML-Dokumenten sind eigentlich gültig ...

Dieser Fehler und seine Verwandten haben schon zu einiger Frustration geführt, deshalb folgt hier eine kurze Einführung in die Tiefen der Textkodierung und der Zeichensätze.

Der Internet Explorer interpretiert den Text hier als UTF-8 (Unicode Transformation Format). UTF-8 kodiert die ersten 128 Zeichen, also den ASCII- (American Standard Code for Information Interchange) bzw. ANSI-Zeichensatz (American National Standards Institute), mit 7 Bit. In diesem Zeichensatz wurde auch das Beispiel abgespeichert.

Das Zeichen »ä« liegt aber auf Position 228, und damit in den zweiten 128 Zeichen. Diese werden in UTF-8 mit 2 bis 5 Bytes kodiert, das mit einem Byte kodierte »ä« wird also nicht erkannt. Das würde auch für das »ä« in den Daten gelten.

Abhilfe lässt sich auf drei verschiedene Arten schaffen. Zum einen je nach Betriebssystem und Textverarbeitung durch Speichern des Textes als UTF8 – ein Format, das beispielsweise auch vom Editor (Notepad) unter Windows 2000 unterstützt wird. Die zweite Möglichkeit besteht aus der Speicherung im UNICODE-Format. Unicode, oder UTF-16, kodiert jedes Zeichen mit 16 Bit. Damit lassen sich über 65.000 Zeichen unterscheiden, womit die aktuellen Schriftsprachen dargestellt werden können. Jeder XML-Prozessor muss UTF-8 und UTF-16 unterstützen, die Kodierung ermittelt er aus den ersten Zeichen.

Beim Öffnen einer UTF-kodierten XML-Datei in einem Text-Editor finden sich oft drei oder vier merkwürdige Zeichen am Anfang. Das ist die so genannte BYTE ORDER MARK (BOM), in der die Kodierung des Dokuments angegeben wird. Das ist also kein Übertragungsfehler, und jeder XML-Prozessor muss damit umgehen können.

Es gibt aber noch wesentlich mehr Kodierungsmöglichkeiten und Zeichensätze, und offensichtlich erkennt sie der XML-Prozessor nicht immer richtig. Sie können die Kodierung deshalb auch explizit angeben, und das ist die dritte Lösungsmöglichkeit. Für den ASCII-Zeichensatz sieht man in Westeuropa meistens die folgende Angabe:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Das ist eine XML-DEKLARATION, die den genauen Typ des XML-Dokuments festlegt. Obwohl die XML-Deklaration in die Zeichen `"<??>"` eingeschlossen ist, handelt es sich hier nicht um eine Processing Instruction (PI).

Diese Zeile muss als erste im Dokument erscheinen, vor ihr ist auch kein Kommentar oder Whitespace erlaubt. Und ausnahmsweise müssen die Attribute in genau der gezeigten Reihenfolge stehen. Das *version*-Attribut hat bis heute immer den Wert *1.0*, und *ISO-8859-1* enthält die ersten 128 Zeichen des ASCII-Zeichensatzes sowie die in Westeuropa benutzten Sonderzeichen. ISO-8859-2 enthält Sonderzeichen für Ost- statt für Westeuropa usw. Wenn man diese Zeile hinzufügt, wird das Dokument korrekt dargestellt.

Leider hat die Lösung noch einen kleinen Schönheitsfehler. Denn wenn Sie den Text `"Euro"` durch das Zeichen »€« ersetzen, erscheint im Internet Explorer nur ein Kästchen. Das »€« ist in ISO-8859-1 nämlich nicht enthalten, in den anderen aktuellen ISO-8859-Varianten natürlich erst recht nicht. Solange die Dokumente unter Windows verarbeitet werden, kann man als Kodierung *windows-1252* angeben. Das ist eine Übermenge von ISO-8859-1, die auch schon das »€« enthält. Diese Kodierung ist kein offizieller Standard, wird aber auch außerhalb der Windows-Plattform durchaus unterstützt. »Erfreulicherweise« ist das € auch in Unicode enthalten. Mehr zum Euro-Problem in der EDV finden Sie zum Beispiel unter `www.cs.tut.fi/~jkorpela/html/euro.html`.

Wenn der Internet Explorer den Fehler »Switch from current encoding to specified encoding not supported.« meldet, wurde entweder das Dokument in Unicode abgespeichert und im encoding-Attribut eine Single-Byte Kodierung wie ISO-8859-1 angegeben, oder umgekehrt.



Spezielle Zeichen in XML

Würde das Zeichen »<« einmal nicht den Anfang eines Tags markieren, käme der XML-Prozessor ziemlich durcheinander – und deswegen ist es einfach verboten, dieses Zeichen irgendwo anders zu benutzen. Ersetzt wird es durch die entsprechende ENTITY REFERENCE »<«. Daraus folgt, dass auch das Zeichen »&« eine besondere Bedeutung hat und durch seine Entity Reference ersetzt werden muss. Insgesamt kennt XML fünf eingebaute Entity References:

- & für &
- ' für '
- > für >
- < für <
- " für "

Entity References in Attribut- und Element-Werten ersetzt ein XML-Prozessor automatisch durch die entsprechenden Zeichen, in Kommentaren tut er das nicht.

Mehr als die fünf hier aufgeführten Entity References kennt der XML-Prozessor tatsächlich nicht. Aus HTML bekannte Entity References wie   (»non breaking space« = nicht trennendes Leerzeichen) oder &auml (»a umlaut« = ä) müssen durch ihre Unicode Codes ersetzt werden – siehe weiter unten.



Beim Aufnehmen von HTML- oder XML-Code in ein XML-Dokument ist das manuelle Ersetzen dieser speziellen Zeichen natürlich lästig und führt auch zu völlig unleserlichen Dokumenten. Deshalb können Elemente auch CDATA-Abschnitte (Character Data) enthalten, in denen der XML-Prozessor aber keine Entity References ersetzt.

```
<DerText><![CDATA[Hier können Sie <, >, &, " und ' benutzen.]]></DerText>
```

Der XML-Prozessor betrachtet alles bis zum "]]>" als Text, logischerweise darf aber die Zeichenkombination "]]>" nicht vorkommen. Während das zufällige Vorkommen dieser Kombination in Textdaten noch unwahrscheinlich ist, können Binärdaten wie Grafiken durchaus Überraschungen bergen. Diese Daten müssen also vorher in ein ungefährliches Format übersetzt werden, wobei man allerdings den Performance-Verlust durch das

Übersetzen nicht vergessen darf. Im Allgemeinen wird das BASE64-Format benutzt, das zur Kodierung nur die 64 Zeichen »A«-»Z«, »a«-»z«, »0«-»9«, »+« und »/« verwendet. Damit brauchen die Daten natürlich keinen umschließenden CDATA-Abschnitt mehr; dieser bleibt also HTML- und XML-Code vorbehalten.

Allerdings sind nicht nur die als Entity References aufgelisteten Zeichen »gefährlich«. Neben den druckbaren Zeichen sind in XML nur die Zeichen 0X0009 (horizontaler Tabulator), 0X000A (Zeilenvorschub/LF) und 0X000D (Wagenrücklauf/CR) erlaubt.

Zeichen, die auf der Tastatur nicht zu finden sind, können im Unicode eingefügt werden. Das kann sowohl mit dem Dezimalwert als auch mit dem Hexadezimalwert des Codes geschehen. Diese Einfügemöglichkeit besteht immer, auch wenn kein Unicode verwendet wird. Das Ganze nennt sich CHARACTER REFERENCE und hat die folgende Form:

```
<DerText>Das sind 2 Copyright-Zeichen: &#169; &#XA9;</DerText>
```

19.2 XSLT und XPath einsetzen

Mit XSLT und XPath können Sie ein XML-Dokument transformieren, das heißt in ein anderes Dokument überführen. Dabei stellt XSLT das Grundgerüst für die Beschreibung der Transformation bereit, und XPath selektiert die gewünschten Teile für die Transformation.

Quell- und Zieldokument müssen nicht unbedingt als Datei existieren: Sie können zum Beispiel mit Streams arbeiten, und das Zieldokument bleibt meistens ein virtuelles Dokument, der so genannte ERGEBNISBAUM (Englisch: result tree).

Die Beispiele und Erläuterungen dieses Abschnitts beziehen sich auf den XML-Prozessor des Internet Explorers (IE). Das hat den Vorteil, dass das Öffnen der XML-Datei im IE reicht, um das Ergebnis der Transformation zu sehen. Die Beispiele würden aber auch mit jedem anderen XML-Prozessor funktionieren, zum Beispiel mit der in .NET eingebauten Klasse `XslTransform`, die eine eigenständige Implementation darstellt und deshalb ohne den IE auskommt.

:-) TIPP

Sie können XML-Dateien direkt im IE öffnen, durch sein Standard-Stylesheet zeigt er das Dokument in einer Baumstruktur. Die Adresse `res://msxml.dll/defaultss.xsl` *zeigt das dazu verwendete Standard-Stylesheet.*

Zu beachten ist allerdings, dass XML und Konsorten relativ neue Standards sind und nicht von allen XML-Prozessoren richtig und vollständig imple-

mentiert werden. So benutzt zum Beispiel der XML-Prozessor des IE bis zur Version 5.0 einen Microsoft-eigenen XSLT-Dialekt. Andererseits implementieren XML-Prozessoren oft nützliche Erweiterungen der Standards: Im IE finden sich zum Beispiel sehr praktische Suchmethoden.

19.2.1 Codebeispiel – Die erste Transformation

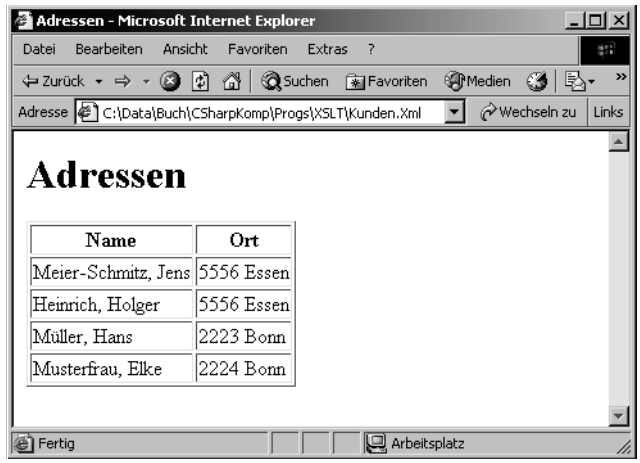
Zunächst ein einfaches Beispiel zur Transformation von XML. Die folgende XML-Datei soll als HTML-Dokument im IE gezeigt werden.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="Adressen.Xsl"?>
<Kunden>
  <Kunde>
    <VorName>Jens</VorName>
    <NachName>Meier-Schmitz</NachName>
    <TelefonFax Vorwahl="01234" Einwahl="56" Telefon="78" Fax='79' />
    <Adresse Stadt="Essen" Plz="5556" Straße="Emscher Weg 19"/>
    <Umsatz Vorjahr="30000" Rabatt="12"/>
  </Kunde>
  <Kunde>
    <VorName>Holger</VorName>
    <NachName>Heinrich</NachName>
    <TelefonFax Vorwahl="01234" Einwahl="56" Telefon="77" Fax='79' />
    <Adresse Stadt="Essen" Plz="5556" Straße="Emscher Weg 19"/>
    <Umsatz Vorjahr="10000"/>
  </Kunde>
  <Kunde>
    <VorName>Hans</VorName>
    <NachName>Müller</NachName>
    <TelefonFax Vorwahl="04321" Einwahl="555" Telefon="66" Fax='77' />
    <Adresse Stadt="Bonn" Plz="2223" Straße="Heimhöhe 13"/>
    <Umsatz Vorjahr="20000" Rabatt="9"/>
  </Kunde>
  <Kunde>
    <VorName>Elke</VorName>
    <NachName>Musterfrau</NachName>
    <TelefonFax Vorwahl="04321" Einwahl="666" Telefon="888" />
    <Adresse Stadt="Bonn" Plz="2224" Straße="Im Berg 21"/>
    <Umsatz Vorjahr="50000" Rabatt="15"/>
  </Kunde>
</Kunden>
```

Die erste Zeile ist schon bekannt, die zweite verweist auf ein Stylesheet mit dem XSLT-Code zur Transformation in ein HTML-Dokument. Das Stylesheet soll Namen und Adressen der Kunden in einer einfachen Tabelle ausgeben, wie in Abbildung 19.2 zu sehen.

```
<?xml version="1.0"?>
<html
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xsl:version="1.0">
```

Abbildung 19.2: Die Kunden als HTML-Tabelle




```
  <head>
    <title>Kunden</title>
  </head>
  <body>
    <h1>Kunden</h1>
    <table border="1">
      <tr><th>Name</th><th>Ort</th></tr>
      <xsl:for-each select="Kunden/Kunde">
        <tr>
          <td>
            <xsl:value-of select="NachName"/>
            <xsl:text>, </xsl:text>
            <xsl:value-of select="VorName"/>
          </td>
          <td>
            <xsl:value-of select="Adresse/@Plz"/>
            <xsl:text> </xsl:text>
            <xsl:value-of select="Adresse/@Stadt"/>
          </td>
        </tr>
      </xsl:for-each>
    </table>
  </body>
</html>
```

Auch hier belegt die erste Zeile die XML-Konformität. Wenn es um Küken statt um Kunden ginge, müsste das *encoding*-Attribut einen entsprechenden Zeichensatz festlegen.

Das Element *html* bildet das Dokument-Element des XML(!)-Dokuments und legt den *xsl*-Namensraum sowie die XSL-Version fest. Dass es sich hier um ein XML-Dokument handelt, lässt sich einfach demonstrieren, wenn man einmal eines der <*html*>-Tags durch <*Html*> ersetzt: Der IE moniert dann prompt die nicht übereinstimmenden Tags.

Die Deklaration <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl"> bezieht sich auf eine überholte XSL(T)-Version. Diese Version wird nur von Microsofts XML-Prozessoren in alten Versionen des IE unterstützt.

:-)
TIPP

Auf die einleitenden beiden Zeilen folgt ein Haufen HTML, in dem sich die *xsl*-Elemente verstecken. Das *xsl:for-each*-Element ist erwartungsgemäß eine SCHLEIFE, und zwar die einzige, die XSLT kennt. Das *select*-Attribut bestimmt, für welche Knoten die Schleife durchlaufen wird. Der Wert des Attributs ist immer ein XPATH-Ausdruck und muss eine KNOTENLISTE liefern. In diesem Fall liefert er alle *Kunde*-Elemente, die Kinder von *Kunden*-Elementen sind. Das *Kunden*-Element als Dokument-Element wird hier automatisch selektiert, die Elementnamen werden als Text eingegeben, und mit dem Schrägstrich wechselt man zur nächsttieferen Ebene.

Die *xsl*-Elemente innerhalb der Schleife beziehen sich auf das jeweils selektierte *Kunde*-Element. Das *xsl:value-of*-Element schreibt den Wert seines *select*-Attributs in den Ergebnisbaum, in der ersten Tabellenzelle ist das der Text der *NachName-* und *VorName*-Elemente.

Auch das *xsl:text*-Element schreibt seinen Text in den Ergebnisbaum, hier ist das ein Komma und ein Leerzeichen.

Selbst wenn man nur ein Leerzeichen einfügen wollte, müsste man das *xsl:text*-Element bemühen; das ist auf den ersten Blick verwunderlich. Aber rundherum wimmelt es von Leerzeichen, die nicht im Ergebnisbaum landen sollen. Genauer gesagt handelt es sich um WHITESPACE (Deutsch: unsichtbare Zeichen), der neben Leerzeichen auch Tabulator-, Zeilenvorschub- und Wagenrücklauf-Zeichen umfasst. Deshalb ist das Standardverhalten jedes XML-Prozessors, nur-Whitespace-Elemente zu ignorieren. Sie können dieses Verhalten mit *xsl:strip-space-* und *xsl:preserve-space*-Elementen kontrollieren.

Generell ist vor allem die Idee des Ergebnisbaums entscheidend für das Verständnis von Transformationen. Dieser Baum wird aus Elementen zusammengebaut – also nicht aus Textstücken, wie man das von der Arbeit mit Strings und Dateien gewöhnt ist.

Der Code für die zweite Tabellenzelle ist ähnlich aufgebaut, allerdings werden hier Attribut-Werte eingetragen. Mit dem Zeichen »@« im XPath-Ausdruck schalten Sie von Elementen auf Attribute um.

In der Quellcodeansicht des IE erscheint nur das XML-Dokument, auch ein Blick auf den Text in der Adressleiste macht das deutlich – das Ergebnisdokument existiert nur im Speicher.

Diese XSL-Datei ist allerdings ein Sonderfall und ähnelt einem Programm, das aus einem einzigen Code-Block besteht. In dieser XSL-Datei ließen sich auch keine globalen Variablen (siehe unten) oder ähnliches deklarieren. Im Allgemeinen werden Sie Ihre XSL-Dateien aus TEMPLATES (Vorlagen) aufbauen: Vorlagen sind die Funktionen der Sprache XSLT.

19.2.2 Codebeispiel – Vorlagen einsetzen

Das nächste Beispiel bietet ähnliche Funktionalität wie das vorhergehende – die XML-Datei bleibt sogar völlig unverändert –, benutzt aber Vorlagen. Das Ergebnis ist eine Fax-Liste, wie in Abbildung 19.3 zu sehen.



Abbildung 19.3: Die mit einer Vorlage generierte Fax-Liste

Das dazu verwendete Stylesheet enthält zwei *xsl:template*-Elemente, deren *match*-Attribute den Typ der zu verarbeitenden Knoten bestimmen. Der Schrägstrich im ersten *match*-Attribut steht dabei für das Dokument-Element, und deshalb wendet der XML-Prozessor diese Vorlage zuerst an.

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:template match="/">
    <html>
      <head>
        <title>Fax-Liste</title>
      </head>
      <body>
        <h1>Fax-Liste</h1>
        <ul>
          <xsl:apply-templates select="Kunden/Kunde"/>
        </ul>
      </body>
    </html>
  </xsl:template>
```

```
  <xsl:template match="Kunde">
    <li>
      <xsl:value-of select="VorName"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="NachName"/>
      <xsl:text> Fax: </xsl:text>
      <xsl:value-of select="TelefonFax/@Vorwahl"/>
      <xsl:text>/</xsl:text>
      <xsl:value-of select="TelefonFax/@Einwahl"/>
      <xsl:text>-</xsl:text>
      <xsl:value-of select="TelefonFax/@Fax"/>
    </li>
  </xsl:template>
</xsl:stylesheet>
```

Die Vorlage baut die grundlegende HTML-Struktur auf und enthält nur ein einziges *xsl*-Element: *xsl:apply-templates*. Durch dieses Element werden nun diejenigen Vorlagen angewendet, die Knoten des im *select*-Attribut genannten Typs verarbeiten. In diesem Fall also die zweite Vorlage, die für jeden Kunden einen Listeneintrag mit der Fax-Nummer generiert.

Allerdings darf man sich die Zusammenarbeit der Vorlagen nicht so vorstellen, dass bei jedem Aufruf der zweiten *Vorlage* ein *Kunde*-Element übergeben wird. Das *match*-Attribut ist ein Muster, kein Parameter – und das aktuelle Element eine globale Variable, die in jedem *template*-Element zur Verfügung steht.

Jetzt folgt das "was-man-alles-machen-könnte-wenn-man-wollte", zum Beispiel die erste Vorlage so ändern:

```
<xsl:template match="Kunden">
  <html>
    <head>
      <title>Fax-Liste</title>
    </head>
    <body>
      <h1>Fax-Liste</h1>
      <ul>
        <xsl:apply-templates select="Kunde"/>
      </ul>
    </body>
  </html>
</xsl:template>
```

Der XML-Prozessor findet jetzt keine Vorlage mehr für das Dokument-Element und greift deshalb auf eine eingebaute Standardvorlage zurück, die *xsl:apply-templates* ohne *select*-Attribut aufruft. Dadurch wird automatisch nach Vorlagen für die Kind-Elemente des Dokument-Elements gesucht, der XML-Prozessor ruft also die *Kunden*-Vorlage auf. Da dieses Verhalten allen XML-Prozessoren angeboren ist, besteht also keine Verpflichtung, eine Vorlage für das Dokument-Element zu kodieren. Es ist aber eine gute Idee, auf diese Weise den Einstiegspunkt zu verdeutlichen.

Die Funktionalität von *xsl:apply-templates* ohne *select*-Attribut ist nicht an das Dokument-Element gebunden: *<xsl:apply-templates/>* statt *<xsl:apply-templates select="Kunde"/>* würde zum gleichen Ergebnis führen.

Die *Kunde*-Vorlage könnte ihrerseits *<xsl:apply-templates/>* aufrufen, um zum Beispiel die *VorName*- und *NachName*-Elemente über entsprechende Vorlagen zu verarbeiten. Diese Vorlagen würden dann in der Reihenfolge aufgerufen, in der die *VorName*- und *NachName*-Elemente im Quelldokument erscheinen. Dieses Verfahren ist also immer vorteilhaft, wenn die Struktur des Quelldokuments ohnehin übernommen werden soll.

Wenn mehrere Vorlagen einen Knoten verarbeiten könnten, sucht sich der XML-Prozessor nach bestimmten Regeln eine davon aus oder er meldet einen Fehler – er wendet sie also nicht nacheinander an.

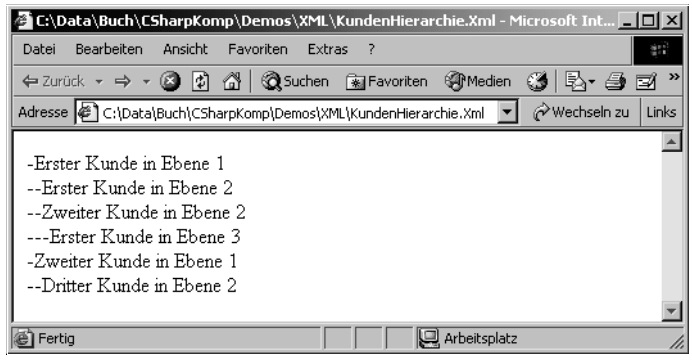
19.2.3 Benannte Vorlagen, Parameter und Variablen einsetzen

Benannte Vorlagen dienen hauptsächlich als Hilfsfunktionen, oder wenn die Anwendung von Mustern wegen komplexer XML-Dokumente zu unübersichtlich wird.

Codebeispiel – Einrückungen entsprechend der Element-Hierarchie ausgeben

Das in diesem Abschnitt vorgestellte Beispiel geht davon aus, dass *Kunde*-Elemente nicht nur als Kinder des *Kunden*-Elements, sondern an verschiedenen Positionen erscheinen. Die Ausgabe soll diese Anordnung durch entsprechende Einrückung widerspiegeln – siehe Abbildung 19.4.

Abbildung 19.4: Ausgabe mit Einrückung entsprechend der Element-Hierarchie




```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl"
  href="KundenHierarchieMitBenanntenTemplates.Xsl"?>
<Kunden Tipp="Die Kundenliste">
  <Kunde Tipp="Erster Kunde in Ebene 1">
    <Kunde Tipp="Erster Kunde in Ebene 2">
    </Kunde>
```

```
    <Kunde Tipp="Zweiter Kunde in Ebene 2">
      <Kunde Tipp="Erster Kunde in Ebene 3">
      </Kunde>
    </Kunde>
  </Kunde>
  <Kunde Tipp="Zweiter Kunde in Ebene 1">
    <Kunde Tipp="Dritter Kunde in Ebene 2">
    </Kunde>
  </Kunde>
</Kunden>
```

Die Elemente sind alle vom selben Typ, das *Tipp*-Attribut gibt Aufschluss über Ebene und Reihenfolge.

Das Stylesheet dazu sieht so aus:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- KundenHierarchieMitBenanntenTemplates.Xsl -->
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:template match="Kunden">
    <xsl:for-each select="//Kunde">
      <xsl:variable name="Einrückung">
        <xsl:call-template name="GibEinrückung"/>
      </xsl:variable>
      <xsl:call-template name="KundenTemplate">
        <xsl:with-param name="Einrückung" select="$Einrückung"/>
      </xsl:call-template>
    </xsl:for-each>
  </xsl:template>


  <xsl:template name="GibEinrückung">
    <xsl:variable name="Ebene" select="count(ancestor::node())"/>

    <xsl:value-of select="substring('----------', 0, $Ebene)"/>
  </xsl:template>


  <xsl:template name="KundenTemplate">
    <xsl:param name="Einrückung"/>

    <xsl:value-of select="$Einrückung"/>
    <xsl:value-of select="@Tipp"/><br/>
  </xsl:template>
</xsl:stylesheet>
```

Die erste Vorlage selektiert das *Kunden*-Element und bildet dann eine Schleife über alle *Kunde*-Elemente, egal, in welcher Ebene sie auftreten. Das erledigt der XPath-Ausdruck *//Kunde*. XPath-Ausdrücke, die mit dem Schrägstrich beginnen, beziehen sich immer auf das *Dokument*-Element – sie finden also auch Knoten oberhalb des aktuellen.



Ein XPath-Ausdruck wie //Kunde ist sehr bequem für den Programmierer. Der XML-Prozessor hat dadurch aber jede Menge Arbeit, denn er muss das gesamte Dokument nach Kunde-Elementen durchsuchen. Das kostet Laufzeit. Exaktes Formulieren eines XPath-Ausdrucks lohnt sich also.

Dann wird die Variable *Einrückung* deklariert und erhält den Wert der *GibEinrückung*-Vorlage.

Als benannte Vorlage enthält *GibEinrückung* im *xsl:template*-Element statt des *match*-Attributs ein *name*-Attribut. Dementsprechend enthält auch das aufrufende *xsl:call-template*-Element ein *name*-Attribut. Die Werte beider Attribute müssen natürlich auch in der Groß- und Kleinschreibung übereinstimmen.

Die *GibEinrückung*-Vorlage deklariert eine lokale Variable *Ebene*. Diese Deklaration erfolgt anders als oben aber in einem leeren Element, und der Wert wird direkt über das *select*-Attribut zugewiesen. Der XPath-Ausdruck *ancestor::node()* liefert eine Knotenliste mit allen Vorfahren des aktuellen Knotens, *count()* liefert die Anzahl der darin enthaltenen Knoten. Eine Beschreibung der XPath-Ausdrücke und -Funktionen finden Sie unter den am Kapitelanfang genannten Links.

Die XPath-Funktion *substring()* liefert die dem Wert von *Ebene* entsprechende Anzahl Bindestriche, dabei wird dem Variablennamen das Dollarzeichen vorangesetzt. Das *xsl:value-of*-Element fügt diese Bindestriche in den Ergebnisbaum ein.

Dabei landen die Bindestriche natürlich in der Variablen *Einrückung*. Diese konnte nicht als leeres Element deklariert werden, weil Vorlagen im *select*-Attribut nicht angewendet werden dürfen.

Zum Schluss wird die *KundenTemplate*-Vorlage mit dem Parameter *Einrückung* aufgerufen. Dazu erhält der Aufruf ein zusätzliches *xsl:with-param*-Element und die Vorlage ein zusätzliches *xsl:param*-Element. In der Vorlage wird auf den Parameter genauso zugegriffen wie auf eine Variable.

Diese Vorlage könnte genauso gut über ein Muster angewendet werden, also mit einem *xsl:apply-templates*-Element in der ersten Vorlage und einem *match*-Attribut. Es ist übrigens legal, einer Vorlage sowohl ein *match*- als auch ein *name*-Attribut zu spendieren.

Noch ein paar Bemerkungen zu XSLT-Variablen. Zum einen sind sie nur sichtbar im übergeordneten Element unterhalb ihrer Deklaration, aber nicht innerhalb ihrer Deklaration. Das verdeutlicht das folgende Beispiel:

```
<xsl:template match="Kunden">
  <xsl:for-each select="//Kunde">
<!--    <xsl:value-of select="$Einrückung"/> FEHLER -->
    <xsl:variable name="Einrückung">
      <xsl:call-template name="GibEinrückung"/>
<!--      <xsl:value-of select="$Einrückung"/> FEHLER -->
    </xsl:variable>
<!--    <xsl:value-of select="$Einrückung"/> OK -->
    <xsl:call-template name="KundenTemplate">
      <xsl:with-param name="Einrückung" select="$Einrückung"/>
    </xsl:call-template>
  </xsl:for-each>
<!--  <xsl:value-of select="$Einrückung"/> FEHLER -->
</xsl:template>
```

Zum anderen unterscheiden sich XSLT-Variablen nicht nur durch das Dollarzeichen beim Aufruf von ihren aus der prozeduralen Programmierung bekannten Vettern: XSLT-Variablen kann man nur bei ihrer Deklaration einen Wert zuweisen, und es ist unmöglich, diesen Wert später zu ändern!

Das ist bedingt durch ein Konzept namens FUNKTIONALE PROGRAMMIERUNG. Dieses Konzept bietet die parallele Ausführung vieler Anweisungen und Vorlagen, fordert dafür aber die Freiheit von Nebeneffekten. Wenn Variablen verändert werden dürften, könnte die Ausführung einer Anweisung aber Nebenwirkungen auf andere Anweisungen haben. Die Reihenfolge der Ausführung wäre dann nicht mehr egal.

Die funktionale Programmierung führt zu ungewohnten Klimmzügen, zum Beispiel finden sich in Stylesheets häufig rekursive Vorlagenaufrufe (siehe Abschnitt »Gruppieren und Zwischensummen durch Rekursion bilden«, Seite 775).

19.2.4 Sortieren und Nummerieren

Hier ist noch einmal das XML-Dokument aus dem ersten Beispiel des Kapitels. Es soll nun zuerst sortiert und dann nummeriert werden.

Codebeispiel – Sortieren

Das in diesem Abschnitt vorgestellte Beispiel gibt die Tabelle nach Postleitzahlen und Nachnamen sortiert aus – siehe Abbildung 19.5.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="Sortieren.Xsl"?>
<Kunden>
  <Kunde>
    <VorName>Jens</VorName>
    <NachName>Meier-Schmitz</NachName>
    <TelefonFax Vorwahl="01234" Einwahl="56" Telefon="78" Fax='79' />
    <Adresse Stadt="Essen" Plz="5556" Straße="Emscher Weg 19"/>
    <Umsatz Vorjahr="30000" Rabatt="12"/>
```



Abbildung 19.5: Nach Postleitzahlen und Nachnamen sortiert

```
  </Kunde>
  <Kunde>
    <VorName>Holger</VorName>
    <NachName>Heinrich</NachName>
    <TelefonFax Vorwahl="01234" Einwahl="56" Telefon="77" Fax='79' />
    <Adresse Stadt="Essen" Plz="5556" Straße="Emscher Weg 19"/>
    <Umsatz Vorjahr="10000"/>
  </Kunde>
  <Kunde>
    <VorName>Hans</VorName>
    <NachName>Müller</NachName>
    <TelefonFax Vorwahl="04321" Einwahl="555" Telefon="66" Fax='77' />
    <Adresse Stadt="Bonn" Plz="2223" Straße="Heimhöhe 13"/>
    <Umsatz Vorjahr="20000" Rabatt="9"/>
  </Kunde>
  <Kunde>
    <VorName>Elke</VorName>
    <NachName>Musterfrau</NachName>
    <TelefonFax Vorwahl="04321" Einwahl="666" Telefon="888" />
    <Adresse Stadt="Bonn" Plz="2224" Straße="Im Berg 21"/>
    <Umsatz Vorjahr="50000" Rabatt="15"/>
  </Kunde>
</Kunden>
```

Das zugehörige Stylesheet sieht ebenfalls fast so aus wie im ersten Beispiel des Kapitels.

```
<?xml version="1.0"?>
<!-- Sortieren.Xsl -->
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
```

```
  <xsl:template match="/">
      <html>
        <head>
          <title>Adressen</title>
        </head>
        <body>
          <h1>Adressen</h1>
          <p>Nach Postleitzahlen und Nachnamen sortiert.</p>
          <table border="1">
            <tr><th>Name</th><th>Ort</th></tr>
            <xsl:apply-templates select="Kunden/Kunde">
              <xsl:sort select="Adresse/@Plz" data-type="number"/>
              <xsl:sort select="NachName" data-type="text"/>
            </xsl:apply-templates>
          </table>
        </body>
      </html>
  </xsl:template>

  <xsl:template match="Kunde">
    <tr>
      <td>
        <xsl:value-of select="NachName"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="VorName"/>
      </td>
      <td>
        <xsl:value-of select="Adresse/@Plz"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="Adresse/@Stadt"/>
      </td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

Neu sind hier die beiden *xsl:sort*-Elemente, für die das *xsl:apply-templates*-Element um ein schließendes Tag erweitert wurde. Im *select*-Attribut dieser Tags wird das Sortierkriterium aufgeführt. Beide Sortierungen werden nacheinander durchgeführt, dabei sortiert die zweite nur die unentschiedenen Fälle der ersten.

Das *data-type*-Attribut ist vor allem für die Sortierung von Zahlenwerten wichtig. Denn voreingestellt ist die Sortierung nach Textregeln, die zum Beispiel die Zahlen 10 und 100 vor den Zahlen 2 und 3 einsortiert.

Durch weitere Attribute, beispielsweise *order* zur Festlegung auf- oder absteigender Sortierung oder *lang* zur Festlegung der sprachabhängigen Sortierregeln, lässt sich die Sortierung noch genauer anpassen.

Codebeispiel – Nummerieren

In diesem Beispiel soll zusätzlich eine laufende Nummer in die Tabelle eingetragen werden. Abbildung 19.6 zeigt das Ergebnis, das sich bereits mit zwei zusätzlichen Zeilen in der zweiten Vorlage erreichen lässt.

```
<xsl:template match="Kunde">
  <tr>
    <td>
      <xsl:value-of select="position()"/>
      <xsl:text>.) </xsl:text>
      <xsl:value-of select="NachName"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="VorName"/>
    </td>
    <td>
      <xsl:value-of select="Adresse/@Plz"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="Adresse/@Stadt"/>
    </td>
  </tr>
</xsl:template>
```

Abbildung 19.6: Die Adressen sortiert und nummeriert




```
<xsl:template match="Kunde">
  <tr>
    <td>
      <xsl:value-of select="position()"/>
      <xsl:text>.) </xsl:text>
      <xsl:value-of select="NachName"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="VorName"/>
    </td>
    <td>
      <xsl:value-of select="Adresse/@Plz"/>
      <xsl:text> </xsl:text>
```

```
      <xsl:value-of select="Adresse/@Stadt"/>
    </td>
  </tr>
</xsl:template>
```

Position() ist eine XPath-Funktion und liefert die Position des Knotens im aktuellen Kontext. »Im aktuellen Kontext« heißt hier: im sortierten Zustand.

Dagegen liefert das *xsl:number*-Element die Position des Knotens in Dokumentreihenfolge, also vor dem Sortieren. Dieses Element bietet aber umfangreichere Formatierungsmöglichkeiten als die *format-number()*-Funktion. Zum Beispiel sorgt diese Zeile für die Nummerierung mit römischen Zahlen:

```
<xsl:number value="position()" format="I"/>
```

Das *xsl:number*-Element formatiert den im *value*-Attribut angegebenen Wert so, wie es das *format*-Attribut festlegt. Durch zusätzliche Attribute lassen sich Formatierung und Zählung noch weiter anpassen.

19.2.5 Bedingungen und Prädikate einsetzen

Für Bedingungen und Prädikate muss noch einmal die Fax-Liste aus den ersten Beispielen herhalten, das XML-Dokument entstammt dem vorhergehenden Beispiel.

Codebeispiel – Bedingungen einsetzen

Jetzt sollen nur Personen ausgegeben werden, die auch eine Faxnummer haben. Frau Musterfrau fehlt das *Fax*-Attribut, sie fällt also aus der Liste heraus (vgl. Abbildung 19.7).

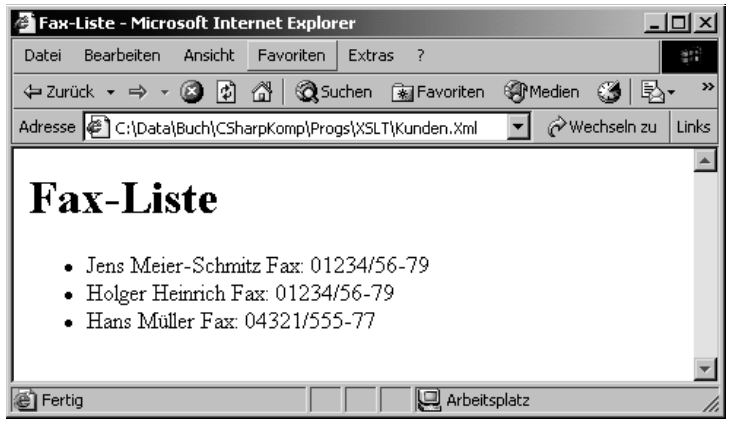


Abbildung 19.7: Die auf Kunden mit Faxnummer reduzierte Liste

Das Stylesheet dazu sieht so aus:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- FaxMitBedingung.Xsl -->
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:template match="Kunden">
    <html>
      <head>
        <title>Fax-Liste</title>
      </head>
      <body>
        <h1>Fax-Liste</h1>
        <ul>
          <xsl:apply-templates select="Kunde"/>
        </ul>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="Kunde">
    <xsl:if test="TelefonFax/@Fax">
      <li>
        <xsl:value-of select="VorName"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="NachName"/>
        <xsl:text> Fax: </xsl:text>
        <xsl:value-of select="TelefonFax/@Vorwahl"/>
        <xsl:text>/</xsl:text>
        <xsl:value-of select="TelefonFax/@Einwahl"/>
        <xsl:text>-</xsl:text>
        <xsl:value-of select="TelefonFax/@Fax"/>
      </li>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Hinzugekommen ist in der zweiten Vorlage das *xsl:if*-Element. Sein *test*-Attribut prüft hier allerdings nur das Vorhandensein des *Fax*-Attributs. Um auch leere *Fax*-Attribute auszufiltern, müsste der Test so aussehen (weitere Informationen zu logischen Operatoren finden Sie unter den Links am Kapitelanfang):

```
<xsl:if test="TelefonFax/@Fax and TelefonFax/@Fax != ''">
```

Codebeispiel – Prädikate einsetzen

Den gleichen Effekt erreichen auch PRÄDIKATE. Dadurch lassen sich die zu verarbeitenden Knoten schon im *xsl:apply-templates*-Element einschränken und so unnötige Aufrufe der zweiten Vorlage einsparen. Die *xsl:if*-Elemente in der zweiten Vorlage können jetzt natürlich entfallen; die erste Vorlage sieht mit Prädikat folgendermaßen aus:

```
<xsl:template match="Kunden">
  <html>
    <head>
      <title>Fax-Liste</title>
    </head>
    <body>
      <h1>Fax-Liste</h1>
      <ul>
        <xsl:apply-templates
           select="Kunde[TelefonFax/@Fax and TelefonFax/@Fax != '']"/>
      </ul>
    </body>
  </html>
</xsl:template>
```

Die Bedingung erscheint also einfach in eckigen Klammern hinter dem Ausdruck, der die Knotenliste liefert. Diese Liste wird dann zusätzlich durch das Prädikat eingeschränkt.

Wenn das Prädikat einen Integer-Wert liefert, wird er auf die Position des Knotens in der Liste bezogen. Diese Zeile liefert also nur den Eintrag für »Holger Heinrich«:

```
<xsl:apply-templates select="Kunde[2]"/>
```

Wie in Bedingungen können Sie auch in Prädikaten XPath-Funktionen benutzen. Die folgende Zeile liefert zum Beispiel nur den letzten Eintrag:

```
<xsl:apply-templates select="Kunde[last()]"/>
```

Codebeispiel – xsl:choose einsetzen

Erstaunlicherweise kennt XSLT kein *xsl:else*- oder *xsl:else-if*-Element. Um beispielsweise bei fehlender Faxnummer die Telefonnummer auszugeben (vgl. Abbildung 19.8), benutzt man deshalb das *xsl:choose*-Element.

```
<xsl:template match="Kunde">
  <li>
    <xsl:value-of select="VorName"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="NachName"/>
    <xsl:text> Fax: </xsl:text>
    <xsl:value-of select="TelefonFax/@Vorwahl"/>
    <xsl:text>/</xsl:text>
    <xsl:value-of select="TelefonFax/@Einwahl"/>
    <xsl:text>-</xsl:text>
    <xsl:choose>
      <xsl:when test="TelefonFax/@Fax and TelefonFax/@Fax != ''">
        <xsl:value-of select="TelefonFax/@Fax"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="TelefonFax/@Telefon"/>
```

```
        <xsl:text> (Telefon)</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
  </li>
</xsl:template>
```

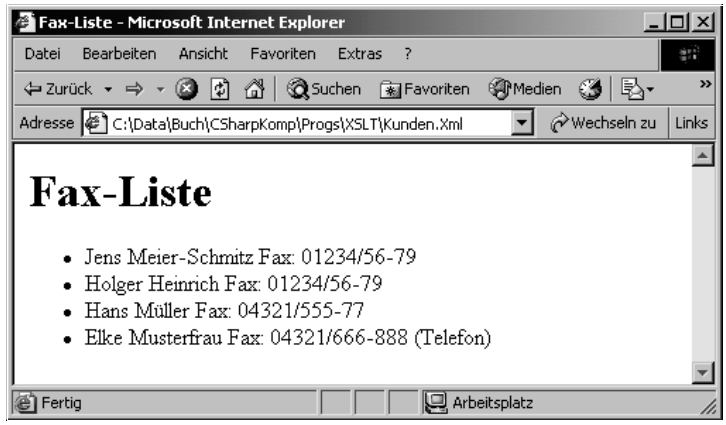


Abbildung 19.8: Fax-Liste mit xsl:choose

Das *xsl:choose*-Element erfüllt denselben Zweck wie ein `switch` in C#, ist aber etwas anders strukturiert: Es stellt in erster Linie den Rahmen für eine beliebige Zahl von *xsl:when*-Elementen dar, die jedes für sich allein ein oder mehrere Bedingungen prüfen. Die XML-Spezifikation garantiert, dass das erste *xsl:when*-Element ausgeführt wird, dessen Prüfbedingung *true* ergibt. Sie legt aber nicht explizit fest, dass in diesem Fall alle weiteren *when*-Zweige des *choose*-Elements übersprungen werden, und ein Äquivalent zu `break` gibt es hier nicht. Aus diesem Grund sollte man *choose* und *when* eher als eine Folge von `if`-Anweisungen betrachten – und Konstrukte wie das folgende vermeiden.

```
<xsl:choose>
  <xsl:when test1= "Meier">
    <xsl:value-of select="Meier gefunden"/>
  </xsl:when>
  <xsl:when test2= "0000">
    <xsl:value-of select="Nummer 0000 gefunden"/>
  </xsl:when>
  ...
```

Hier ist durch die Spezifikation nicht exakt festgelegt, was geschieht, wenn beide Bedingungen zutreffen.

In einem Punkt unterscheidet sich *xsl:choose* aber definitiv von einer Folge einzelner Prüfungen: Wenn keins der *xsl:when*-Elemente zum Ergebnis *true* kommt, wird das *xsl:otherwise*-Element ausgeführt. Dieses Element ist zwar optional, sollte in einem ordentlichen Stylesheet aber nie fehlen (Stichwort: Fehlertoleranz).

19.2.6 Elemente und Attribute einfügen

Wozu würde man Elemente und Attribute in ein XML-Dokument einfügen wollen, das (zumindest in diesen Beispielen) ohnehin nur in der Vorstellung des Browsers existiert? Ganz einfach: damit der Browser sie bei der Darstellung benutzt.

Codebeispiel – Elemente und Attribute einfügen

Das klassische Beispiel zur Benutzung von Attributen ist eine Zebrastreifen-Tabelle wie in Abbildung 19.9: Gerade und ungerade Zeilen bekommen durch unterschiedliche Werte im *class*-Attribut eigene Hintergrundfarben.

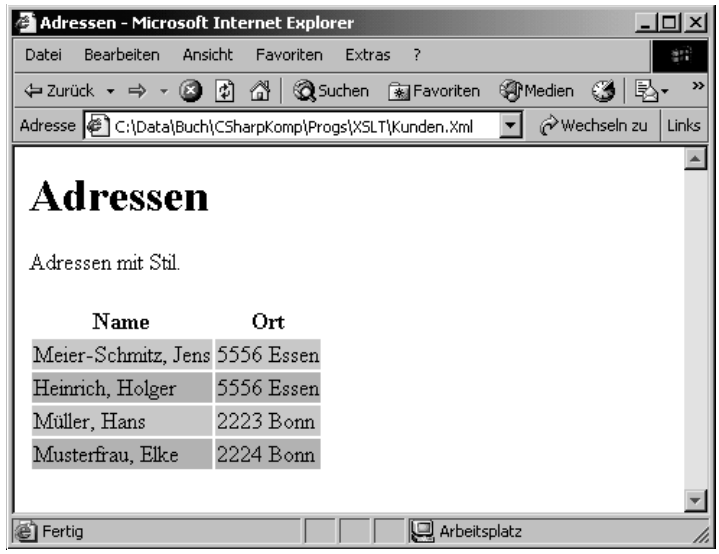


Abbildung 19.9: Adressen mit Stil

Das benutzte XML-Dokument ist schon in den vorhergehenden Beispielen erschienen, und auch das XSL-Dokument ist in großen Teilen bekannt.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- AdressenMitEinfügung.Xsl -->
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:template match="/">
      <html>
        <head>
          <style type="text/css">
            .Gerade {background-color: Aqua}
            .UnGerade {background-color: Aquamarine}
          </style>
          <title>Adressen</title>
        </head>
```

```
        <body>
          <h1>Adressen</h1>
          <p>Adressen mit Stil.</p>
          <table>
            <tr><th>Name</th><th>Ort</th></tr>
            <xsl:apply-templates select="Kunden/Kunde"/>
          </table>
        </body>
      </html>
  </xsl:template>

  <xsl:template match="Kunde">
    <tr>
      <xsl:attribute name="class">
      <xsl:choose>
        <xsl:when test="position() mod 2 = 0">
          <xsl:text>Gerade</xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>UnGerade</xsl:text>
        </xsl:otherwise>
      </xsl:choose>
      </xsl:attribute>
      <td>
        <xsl:value-of select="NachName"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="VorName"/>
      </td>
      <td>
        <xsl:value-of select="Adresse/@Plz"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="Adresse/@Stadt"/>
      </td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

Neu ist die Ergänzung der erste Vorlage um ein *style*-Element, das zwei Klassen mit unterschiedlichen Hintergrundfarben definiert.

Die zweite Vorlage ordnet die *tr*-Elemente abwechselnd einer dieser Klassen zu, was wie üblich über das *class*-Attribut des *tr*-Elements geschieht. Dieses Attribut wird über das *xsl:attribute*-Element angelegt, seinen Wert bekommt es über das darin enthaltene *xsl:text*-Element. Genauso lassen sich auch Elemente anlegen, nur wird statt des *xsl:attribute*-Elements ein *xsl:element*-Element verwendet.

Lange String-Basteleien wären hier, wie auch in XSLT allgemein, der falsche Ansatz. XSLT baut einen Ergebnisbaum auf, anstatt ein Textdokument aus einzelnen Zeichen zusammenzusetzen. Mit dem Verständnis dieses Konzepts ist XML ganz einfach.

19.2.7 Gruppieren und Zwischensummen durch Rekursion bilden

In den bisherigen Beispielprogrammen waren die Kundenadressen bestenfalls sortiert, jetzt sollen jeweils die Kunden einer Stadt in einer eigenen Tabelle aufgeführt werden. Abbildung 19.10 zeigt das Ergebnis.

Dieses Ergebnis kann man mit verschiedenen Methoden erreichen. Aber die meisten funktionieren dann irgendwie doch nicht, und der Rest ist zu langsam. Deshalb benutzt man zum Gruppieren die MUENCH-METHODE, so genannt nach ihrem Erfinder Steve Muench, einem Oracle-Programmierer und Buchautor.

Codebeispiel – Gruppieren mit der Muench-Methode

Dieses Beispielprogramm zeigt das Gruppieren mit der Muench-Methode.

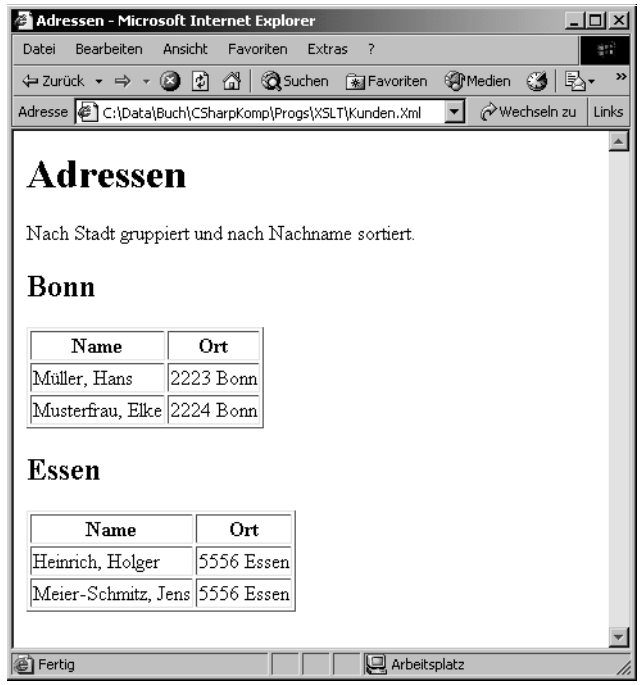


Abbildung 19.10: Adressen mit Gruppierung

Für dieses einfache Beispiel hätte man zugegeben am besten die HTML-Datei mit der Hand kodiert, aber für größere Datenmengen und kompliziertere Datenstrukturen lohnt sich das Nachvollziehen des Stylesheets.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- AdressenMitGruppierung.Xsl -->
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
```

```
  <xsl:key name="Städte" match="Kunden/Kunde" use="Adresse/@Stadt"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Adressen</title>
      </head>
      <body>
        <h1>Adressen</h1>
        <p>Nach Stadt gruppiert und nach Nachname sortiert.</p>
        <xsl:for-each select="Kunden/Kunde[generate-id(.)=
          generate-id(key('Städte', Adresse/@Stadt)[1])]">
          <xsl:sort select="Adresse/@Stadt"/>
          <h2><xsl:value-of select="Adresse/@Stadt"/></h2>
          <table border="1">
            <tr><th>Name</th><th>Ort</th></tr>
            <xsl:for-each select="key('Städte', Adresse/@Stadt)">
              <xsl:sort select="NachName"/>
              <xsl:apply-templates select="."/>
            </xsl:for-each>
          </table>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="Kunde">
    <tr>
      <td>
        <xsl:value-of select="NachName"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="VorName"/>
      </td>
      <td>
        <xsl:value-of select="Adresse/@Plz"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="Adresse/@Stadt"/>
      </td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

Hier fällt zuerst diese Zeile auf:

```
<xsl:key name="Städte" match="Kunden/Kunde" use="Adresse/@Stadt"/>
```

Das *xsl:key*-Element stellt man sich am besten wie einen Datenbankindex vor. Der Index wird unter dem in *name* angegebenen Namen verfügbar sein, und liefern wird er die mit *match* selektierten Knoten. Diese werden eingeschränkt über die Werte der in *use* angegebenen Knoten, deren Pfad relativ zu den in *match* angegebenen Knoten ausgewertet wird. In diesem Beispiel enthält der Index also alle *Kunde*-Elemente, und zugegriffen wird über das *Stadt*-Attribut ihres *Adresse*-Elements.

In den *match*- und *use*-Attributen sind keine Variablen erlaubt, sonst wären zirkuläre Definitionen möglich. Aber weil das *xsl:key*-Element vor den globalen Variablen ausgewertet wird, kann sein Resultat in diesen benutzt werden.

Wie bei einem Datenbank-Index dauert das Aufbauen seine Zeit, dafür ist später der Zugriff schneller. Ausgewertet wird ein solcher Index ausschließlich über die *key()*-Funktion.

Benutzt wird der Index zum ersten Mal um eine Liste zu erzeugen, die einen Kunden für jede Stadt enthält.

```
<xsl:for-each select="Kunden/Kunde[generate-id(.)=
  generate-id(key('Städte', Adresse/@Stadt)[1])]">
```

In dieser Zeile passieren mehrere Dinge auf einmal:

- *Kunden/Kunde* erzeugt eine Liste aller Kunden-Elemente.
- Für jedes dieser Elemente wird geprüft, ob es die Bedingung des Prädikats erfüllt.
- Diese Bedingung ist, dass es der erste Kunde in seiner Stadt ist: *key('Städte', Adresse/@Stadt)[1]*.
- Eine Schleife läuft über die fertige Liste.

Die *key()*-Funktion liefert eine Liste mit allen *Kunde*-Elementen, die den angegebenen Wert im *Stadt*-Attribut ihres *Adresse*-Elements haben. Das entspricht der Definition des Index im *xsl:key*-Element. Das »innere« Prädikat *[1]* schränkt diese Liste auf ihr erstes Element ein.

Über *generate-id()* wird der aktuelle Kunde mit dem von *key()* gelieferten verglichen. Dabei liefert *generate-id()* zum übergebenen Knoten einen eindeutigen Wert. Das heißt, ein Knoten liefert immer den gleichen, innerhalb des Programms einzigartigen Wert – Verwechslungen sind also ausgeschlossen.

Die so erhaltene Liste wird nach Städten sortiert, und für jede Stadt wird eine Überschrift und eine Tabelle ausgegeben.

Dabei bildet die folgende Zeile eine Schleife über alle *Kunde*-Elemente in dieser Stadt:

```
<xsl:for-each select="key('Städte', Adresse/@Stadt)">
```

Diese Liste wird nach Nachnamen sortiert, und jedes der *Kunde*-Elemente wird über die zweite Vorlage in die Tabelle eingetragen.

Codebeispiel – Bilden von Zwischensummen

Die nächsten Beispiele befassen sich mit dem Bilden von Zwischensummen. Das erste Resultat sieht wie in Abbildung 19.11 aus.

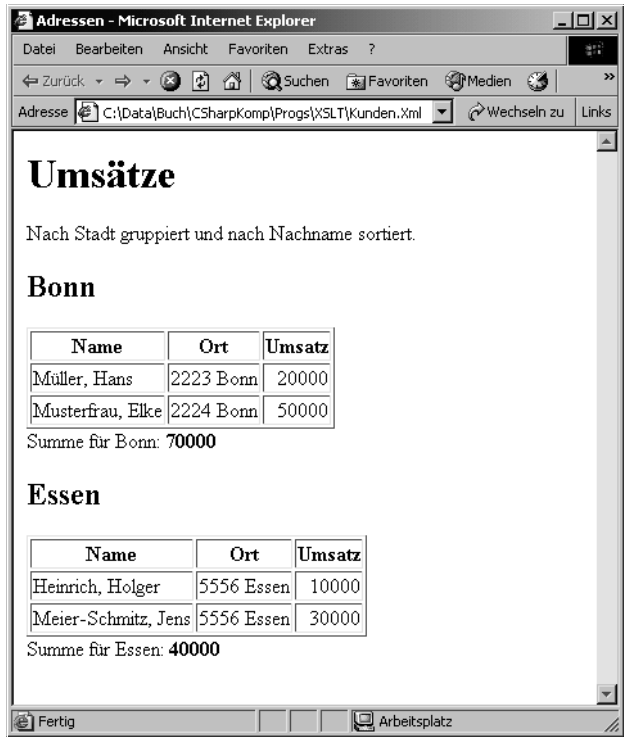


Abbildung 19.11: Adressen mit Umsatz

Jede Tabelle erhält also eine zusätzliche Spalte mit den Umsätzen, und außerdem eine Unterschrift mit der Summe dieser Umsätze.

Das ist recht einfach zu bewerkstelligen:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- AdressenMitZwischensummen.Xsl -->
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:key name="Städte" match="Kunden/Kunde" use="Adresse/@Stadt"/>
    <xsl:template match="/">
    <html>
      <head>
          <style type="text/css">
             .Umsatz {text-align: right}<!-- oder justify -->
           </style>
        <title>Umsätze</title>
      </head>
```

```
      <body>
        <h1>Adressen</h1>
        <p>Nach Stadt gruppiert und nach Nachname sortiert.</p>
        <xsl:for-each select="Kunden/Kunde[generate-id(.)=
          generate-id(key('Städte', Adresse/@Stadt)[1])]">
          <xsl:sort select="Adresse/@Stadt"/>
          <h2><xsl:value-of select="Adresse/@Stadt"/></h2>
          <table border="1">
            <tr><th>Name</th><th>Ort</th><th>Umsatz Vorjahr</th></tr>
            <xsl:for-each select="key('Städte', Adresse/@Stadt)">
              <xsl:sort select="NachName"/>
              <xsl:apply-templates select="."/>
            </xsl:for-each>
          </table>
          <xsl:text>Summe für </xsl:text>
          <xsl:value-of select="Adresse/@Stadt"/>
          <xsl:text>: </xsl:text>
          <b><xsl:value-of
            select="sum(key('Städte', Adresse/@Stadt)/Umsatz/@Vorjahr)"/>
          </b>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="Kunde">
    <tr>
      <td>
        <xsl:value-of select="NachName"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="VorName"/>
      </td>
      <td>
        <xsl:value-of select="Adresse/@Plz"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="Adresse/@Stadt"/>
      </td>
      <td class="Umsatz">
        <xsl:value-of select="Umsatz/@Vorjahr"/>
      </td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

Der Stil *Umsatz* richtet die Zahlen rechtsbündig aus. Wenn die Zahlen Nachkommastellen hätten, würde man hier die Ausrichtung *justify* verwenden.

Dann bekommt das Tabellengerüst eine zusätzliche Spalte, in die die zweite Vorlage die Umsatzwerte einträgt. Dabei wird natürlich der oben definierte Stil benutzt.

Schließlich sorgt diese Zeile für den Eintrag der Summe:

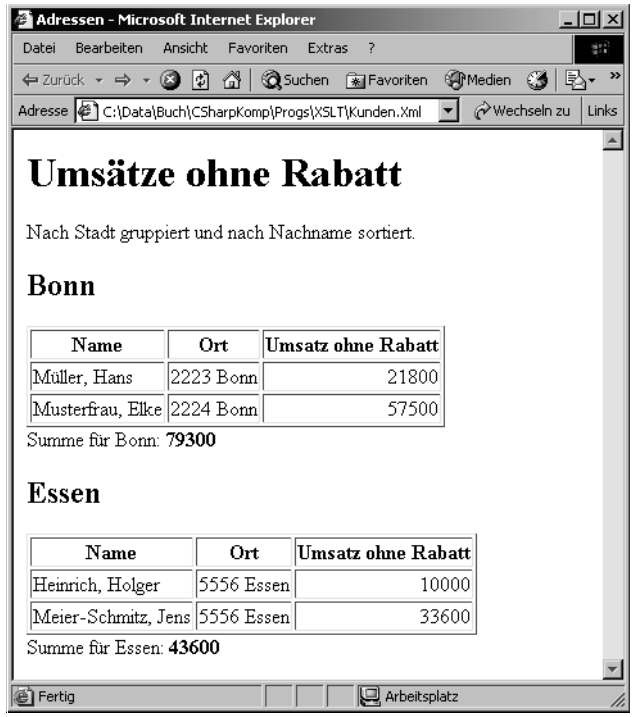
```
<xsl:value-of select="sum(key('Städte', Adresse/@Stadt)/Umsatz/@Vorjahr)"/>
```

Wie in den vorangegangenen Beispielen liefert die *key()*-Funktion eine Liste mit allen *Kunde*-Elementen zur gewünschten Stadt, dazu liefert *../Umsatz/@Vorjahr* die gesuchten Attribute. Die Funktion *sum()* bildet schließlich die Summe der Umsätze.

Codebeispiel – Bilden von Zwischensummen mit Rekursion

Im letzten Beispiel konnte die Summe problemlos aus den vorhandenen Attributen ermittelt werden. Wesentlich schwieriger ist die Berechnung der Umsätze ohne Rabatt. Diese muss man für jeden Kunden anhand seines persönlichen Rabattes berechnen, eine einfache Summenbildung wie im vorherigen Beispiel ist also nicht mehr möglich. Abbildung 19.12 zeigt das Ergebnis.

Abbildung 19.12: Adressen mit Umsatz ohne Rabatt



Der grundsätzliche Aufbau des Stylesheets bleibt gleich, es kommt aber eine Vorlage zur Berechnung mit rekursivem Aufruf dazu.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- AdressenMitZwischensummenUndRekursion.Xsl -->
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
```

```
<xsl:key name="Städte" match="Kunden/Kunde" use="Adresse/@Stadt"/>

<xsl:template match="/">
  <html>
    <head>
        <style type="text/css">
           .Umsatz {text-align: right}<!-- oder justify -->
         </style>
      <title>Adressen</title>
    </head>
    <body>
      <h1>Umsätze ohne Rabatt</h1>
      <p>Nach Stadt gruppiert und nach Nachname sortiert.</p>
      <xsl:for-each select="Kunden/Kunde[generate-id(.)=
        generate-id(key('Städte', Adresse/@Stadt)[1])]">
        <xsl:sort select="Adresse/@Stadt"/>
        <h2><xsl:value-of select="Adresse/@Stadt"/></h2>
        <table border="1">
          <tr><th>Name</th><th>Ort</th><th>Umsatz ohne Rabatt</th></tr>
          <xsl:for-each select="key('Städte', Adresse/@Stadt)">
            <xsl:sort select="NachName"/>
            <xsl:apply-templates select="."/>
          </xsl:for-each>
        </table>
        <xsl:text>Summe für </xsl:text>
        <xsl:value-of select="Adresse/@Stadt"/>
        <xsl:text>: </xsl:text>
        <b>
        <xsl:call-template name="BerechneUmsatzOhneRabattSumme">
          <xsl:with-param
            name="Kunden" select="key('Städte', Adresse/@Stadt)"/>
        </xsl:call-template>
        </b>
      </xsl:for-each>
    </body>
  </html>
</xsl:template>

<xsl:template match="Kunde">
  <tr>
    <td>
      <xsl:value-of select="NachName"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="VorName"/>
    </td>
    <td>
      <xsl:value-of select="Adresse/@Plz"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="Adresse/@Stadt"/>
    </td>
    <td class="Umsatz">
      <xsl:call-template name="BerechneUmsatzOhneRabatt">
        <xsl:with-param name="Kunde" select="."/>
      </xsl:call-template>
    </td>
  </tr>
</xsl:template>
```

```
  <xsl:template name="BerechneUmsatzOhneRabatt">
    <xsl:param name="Kunde"/>

    <xsl:choose>
      <xsl:when test="$Kunde/Umsatz/@Rabatt and
        $Kunde/Umsatz/@Rabatt !=''">
        <xsl:value-of
          select="round($Kunde/Umsatz/@Vorjahr *
            ((100 + $Kunde/Umsatz/@Rabatt) div 100))"/>
      </xsl:when>
      <xsl:otherwise> <!-- sonst NaN -->
        <xsl:value-of select="$Kunde/Umsatz/@Vorjahr"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>


  <xsl:template name="BerechneUmsatzOhneRabattSumme">
    <xsl:param name="Kunden"/>

    <xsl:choose>
      <xsl:when test="$Kunden">
        <xsl:variable name="UmsatzOhneRabatt">
          <xsl:call-template name="BerechneUmsatzOhneRabatt">
            <xsl:with-param name="Kunde" select="$Kunden[1]"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:variable name="LaufendeSumme">
          <xsl:call-template name="BerechneUmsatzOhneRabattSumme">
            <xsl:with-param name="Kunden"
              select="$Kunden[position() != 1]" />
          </xsl:call-template>
        </xsl:variable>
        <xsl:value-of select="$UmsatzOhneRabatt + $LaufendeSumme" />
      </xsl:when>
      <xsl:otherwise>0</xsl:otherwise>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

Neu ist hier zunächst die Vorlage *BerechneUmsatzOhneRabatt*. Sie erwartet ein *Kunde*-Element als Parameter und berechnet dessen Umsatz ohne Rabatt. Wenn das *Umsatz*-Element dieses Kunden kein *Rabatt*-Attribut hat, liefert die Vorlage den Umsatz unverändert zurück. Ein fehlendes oder leeres *Rabatt*-Attribut würde sonst für die Berechnung den Wert *NaN* (Not a Number) liefern, eine spezielle XSLT-Konstante. Kunden ohne Umsatz gibt es im gegebenen Beispiel nicht.

Beim Aufruf zur Berechnung aus der *Kunde*-Vorlage heraus müsste das *Kunde*-Element eigentlich nicht als Parameter übergeben werden, weil es sowieso der aktuelle Knoten ist.

Neu ist auch die Vorlage *BerechneUmsatzOhneRabattSumme*, die die Summe für die Tabelle berechnet. Sie bekommt beim Aufruf eine Liste aller *Kunde*-Elemente der Tabelle übergeben, lässt für jedes dieser Elemente den Umsatz ohne Rabatt berechnen und liefert die Summe zurück. Weil es in XSLT keine Möglichkeit gibt, einer Variablen nach der Initialisierung einen Wert zuzuweisen, arbeitet die Vorlage rekursiv.

Zunächst prüft die Vorlage, ob die im Parameter übergebene Liste leer ist. Falls ja, wird der Aufruf mit dem Ergebnis *0* beendet.

Falls die Liste mindestens ein Element enthält, wird *BerechneUmsatzOhne-Rabatt* mit dem ersten Element der Liste aufgerufen; das Resultat landet in der Variablen *UmsatzOhneRabatt*. Danach ruft sich die Vorlage selbst auf, und zwar mit allen Elementen außer dem ersten. Das Ergebnis wird in der Variablen *LaufendeSumme* gespeichert. Schließlich gibt die Vorlage die Summe der beiden Variablen *UmsatzOhneRabatt* und *LaufendeSumme* zurück.

Bei jedem Selbstaufruf wird die Liste der *Kunde*-Elemente also um ein Element kürzer, bis sie schließlich leer ist und die Rekursion endet.

19.3 XML-Klassen in .NET

.NET bietet mehrere Möglichkeiten zum Erzeugen und Verarbeiten von XML-Dokumenten. Zum einen können Sie das XML DOM (Document Object Model) benutzen. Das XML DOM ist vom World Wide Web Consortium standardisiert und bietet ein XML-Dokument als Baum an, der es über Typen wie *Document*, *Element*, *NodeList* und deren Eigenschaften und Methoden verfügbar macht. Das XML DOM erlaubt, zu Elementen und Attributen zu navigieren, Elemente und Attribute hinzuzufügen, zu selektieren, zu verändern und zu entfernen.

Das XML DOM wird in .NET von der `XmlDocument`-Klasse bereitgestellt. Die Knotentypen eines XML-Dokuments werden von Klassen wie `XmlAttribute` oder `XmlElement` repräsentiert. Durch die `XmlDocument`-Klasse steht ein XML-Dokument als recht intuitives Objektmodell zur Verfügung, das alle Navigations- und Bearbeitungsmöglichkeiten bietet. Da immer das ganze XML-Dokument geladen und dafür die entsprechenden Objekte erzeugt werden müssen, können große XML-Dokumente zu Performance- und Ressourcen-Problemen führen.

Navigation und Selektion erfolgen bei `XmlDocument`-Instanzen entweder über das XML DOM selbst oder über die spezialisierte Klasse `XPathNavigator`. Diese Klasse und ihre Verwandten, wie `XPathNodeIterator` und `XPathExpression`, stellen eine Abstraktionsschicht oberhalb des XML DOM bereit. Die

Idee hinter dieser Schicht ist ein einheitliches Zugriffsmodell für beliebige Datenspeicher. Denkbar wären beispielsweise `XPathNavigator`-Implementationen für ZIP-Dateien oder das Dateisystem.

Zur Transformation eines XML-Dokuments steht die darauf spezialisierte Klasse `XPathDocument` zur Verfügung. Veränderungen der Daten wie das Setzen von Werten sind aber mit dieser Klasse nicht möglich, außerdem geschehen Navigation und Selektion ausschließlich über `XPathNavigator`-Instanzen.

Transformationen erledigt die Klasse `XslTransform`, und zwar sowohl für Dokumente aus `XPathDocument`-Instanzen als auch für Dokumente aus `XmlDocument`-Instanzen. Und, man ahnt es schon: Das Ganze funktioniert über `XPathNavigator`-Instanzen.

Die zweite Möglichkeit zum Erzeugen und Verarbeiten von XML-Dokumenten sind die Klassen `XmlTextReader` und `XmlTextWriter`. Sie bieten einen Stream-basierten Zugriff auf XML-Dokumente, bauen also keinen Baum von Objekten auf, und haben dadurch Performance-Vorteile insbesondere bei großen Dokumenten. Sie werden auch von der `XmlDocument`-Klasse zum Lesen bzw. Schreiben von XML-Dokumenten benutzt.

Wie bereits zu Anfang des Kapitels erwähnt, bietet die .NET-Laufzeitumgebung zwar keine SAX-Implementation (Simple API for XML), dafür aber die Einbindung des XML-Prozessors des IE. Diese Möglichkeit ist insbesondere interessant, wenn Sie auf dessen liebgewonnene Erweiterungen wie `TransformNode()` nicht verzichten können oder wollen. Die Beispiele im Abschnitt »XSLT und XPath einsetzen« auf Seite 756 benutzen alle diesen XML-Prozessor.

Neben den genannten Klassen existieren noch Klassen wie `XmlDataDocument` für spezielle XML-Anwendungen, in diesem Fall die Kombination mit der `DataSet`-Klasse.

Die folgenden Abschnitte zeigen Einsatzbeispiele dieser Klassen.

19.3.1 XML erzeugen und transformieren

Das in diesem Abschnitt vorstellte Beispiel demonstriert die DOM- und Stream-basierten Varianten beim Erzeugen und Transformieren von XML. Da sich 400 Zeilen Code hinter dem Screenshot in Abbildung 19.13 verbergen, darf er trotz eingeschränkter Aussage- und Anziehungskraft ins Buch.

Für dieses Projekt benötigen Sie folgende `using`-Anweisungen:

```
using System.Xml;
using System.IO;
```

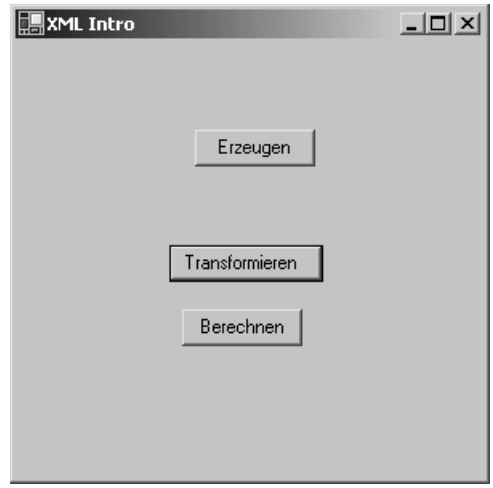


Abbildung 19.13: Nur drei Schaltflächen darauf, aber 400 Zeilen Code darunter ...

```
using System.Text;
using System.Xml.Xsl;
using System.Xml.XPath;
```

Codebeispiel – Erzeugen von XML

Dieses Beispiel soll eine XML-Datei mit Kunden erzeugen, wie sie schon im Abschnitt »Codebeispiel – Die erste Transformation« (Seite 757) verwendet wurde. Um die Möglichkeiten (und Umständlichkeiten) der verschiedenen .NET-Klassen zu zeigen, wird die Datei in drei Schritten angelegt: Die ersten beiden Kunden erzeugt die `XmlDocument`-Klasse, den dritten erzeugt die `XmlTextWriter`-Klasse; der vierte Kunde wird schließlich aus einem anderen XML-Dokument importiert. Damit sich die einzelnen Schritte besser nachvollziehen lassen, werden die Zwischenergebnisse jeweils in eine Datei geschrieben.

Hier ist noch einmal das Endergebnis:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Kunden>
  <Kunde>
    <VorName>Jens</VorName>
    <NachName>Meier-Schmitz</NachName>
    <TelefonFax Vorwahl="01234" Einwahl="56" Telefon="78" Fax="79" />
    <Adresse Stadt="Essen" Plz="5556" Straße="Emscher Weg 19" />
    <Umsatz Vorjahr="30000" Rabatt="12" />
  </Kunde>
  <Kunde>
    <VorName>Holger</VorName>
    <NachName>Heinrich</NachName>
    <TelefonFax Vorwahl="01234" Einwahl="56" Telefon="77" Fax="79" />
    <Adresse Stadt="Essen" Plz="5556" Straße="Emscher Weg 19" />
    <Umsatz Vorjahr="10000" />
  </Kunde>
```

```
  <Kunde>
    <VorName>Hans</VorName>
    <NachName>Müller</NachName>
    <TelefonFax Vorwahl="04321" Einwahl="555" Telefon="66" Fax="77" />
    <Adresse Stadt="Bonn" Plz="2223" Straße="Heimhöhe 13" />
    <Umsatz Vorjahr="20000" Rabatt="9" />
  </Kunde>
  <Kunde>
    <VorName>Elke</VorName>
    <NachName>Musterfrau</NachName>
    <TelefonFax Vorwahl="04321" Einwahl="666" Telefon="888" Fax="" />
    <Adresse Stadt="Bonn" Plz="2224" Straße="Im Berg 21" />
    <Umsatz Vorjahr="50000" Rabatt="15" />
  </Kunde>
</Kunden>
```

Die Behandlungsmethode für das `Click`-Ereignis der ERZEUGEN-Schaltfläche ruft für jeden der Schritte eine Methode auf.

```
private void btnErzeugen_Click(object sender, System.EventArgs e)
{
  ErzeugeXmlMitDom();
  ErzeugeXmlMitWriter();
  ErzeugeXmlMitImport();
}//btnErzeugen_Click
```

Die Methode `ErzeugeXmlMitDom()` sieht folgendermaßen aus:

```
private void ErzeugeXmlMitDom()
{
  XmlNode TempElem, TempAttr1, Kommentar, Kunde, Kunden;
  XmlAttribute TempAttr; //Oder von XmlNode auf XmlAttribute casten
  XmlText TempTextElem;

  XmlDocument doc = new XmlDocument();

  //Kundenliste mit LoadXml() erzeugen
  doc.LoadXml("<Kunden/>");
  Kunden = doc.SelectSingleNode("Kunden");

  //Einen Kommentar erzeugen
  Kommentar = doc.CreateComment("Erstes XML-Dokument");
  doc.PrependChild(Kommentar);
  //Man könnte auch eine PI mit CreateProcessingInstruction() erzeugen,
  //oder Elemente einem Namensraum zuordnen

  //Ersten Kunden über Knoten-Objekte erzeugen
  Kunde = doc.CreateElement("Kunde");
  //-VorName- und NachName-Elemente erzeugen
  TempElem = doc.CreateElement("VorName");
  TempElem.InnerText = "Jens";
  Kunde.AppendChild(TempElem);
  TempElem = doc.CreateElement("NachName");
```

```
//TempElem.InnerText = "Meier-Schmitz";
TempTextElem = doc.CreateTextNode("Meier-Schmitz");
TempElem.AppendChild(TempTextElem);
Kunde.AppendChild(TempElem);
//-TelefonFax-Element mit Attributen erzeugen
TempElem = doc.CreateElement("TelefonFax");
TempAttr1 = doc.CreateAttribute("Vorwahl");
TempAttr1.Value = "01234";
TempElem.Attributes.Append((XmlAttribute)TempAttr1);
TempAttr = doc.CreateAttribute("Einwahl");
TempAttr.Value = "56";
TempElem.Attributes.Append(TempAttr);
TempAttr = doc.CreateAttribute("Telefon");
TempAttr.Value = "78";
TempElem.Attributes.Append(TempAttr);
TempAttr = doc.CreateAttribute("Fax");
TempAttr.Value = "79";
TempElem.Attributes.Append(TempAttr);
Kunde.AppendChild(TempElem);
//-Adresse-Element mit Attributen erzeugen
TempElem = doc.CreateElement("Adresse");
TempAttr = doc.CreateAttribute("Stadt");
TempAttr.Value = "Essen";
TempElem.Attributes.Append(TempAttr);
TempAttr = doc.CreateAttribute("Plz");
TempAttr.Value = "5556";
TempElem.Attributes.Append(TempAttr);
TempAttr = doc.CreateAttribute("Straße");
TempAttr.Value = "Emscher Weg 19";
TempElem.Attributes.Append(TempAttr);
Kunde.AppendChild(TempElem);
//-Umsatz-Element mit Attributen erzeugen
TempElem = doc.CreateElement("Umsatz");
TempAttr = doc.CreateAttribute("Vorjahr");
TempAttr.Value = "30000";
TempElem.Attributes.Append(TempAttr);
TempAttr = doc.CreateAttribute("Rabatt");
TempAttr.Value = "12";
TempElem.Attributes.Append(TempAttr);
Kunde.AppendChild(TempElem);
Kunden.AppendChild(Kunde);

//Zweiten Kunden über String erzeugen
Kunde = doc.CreateElement("Kunde");
Kunde.InnerXml ="<VorName>Holger</VorName>" +
  "<NachName>Heinrich</NachName>" +
  "<TelefonFax Vorwahl='01234' Einwahl='56' Telefon='77' Fax='79' />" +
  "<Adresse Stadt='Essen' Plz='5556' Straße='Emscher Weg 19'/>" +
  "<Umsatz Vorjahr='10000'/>";
Kunden.AppendChild(Kunde);

//Kundenliste in Datei schreiben
XmlTextWriter tw = new XmlTextWriter(
  Application.StartupPath + "\\Kunden1.Xml",
  Encoding.GetEncoding("ISO-8859-1"));
```

```
  try
  {
    tw.Formatting = Formatting.Indented;
    tw.Indentation = 2;
    tw.QuoteChar = '"';
    doc.Save(tw);
  }
  finally
  {
    tw.Close();
  }//finally
}//ErzeugeXmlMitDom
```

Nach dem Erzeugen einer `XmlDocument`-Instanz entsteht darin das *Kunden*-Element, wodurch automatisch das Wurzel-Element (*DocumentElement*) existiert. Die dabei benutzte Methode `LoadXml()` ersetzt den gesamten Dokumentinhalt durch den übergebenen String, erzeugt also ein *Kunden*-Element, und `SelectSingleNode()` selektiert dieses Element.

Die Methode `CreateComment()` erzeugt einen Kommentar. Dabei wird das typische Erzeugungs-Muster des DOM benutzt: Nur die entsprechenden Factory-Methoden des Dokuments können neue Knoten erzeugen, und diese Knoten müssen dann mit einer Methode wie `AppendChild()` oder `PrependChild()` ihrem übergeordneten Element hinzugefügt werden. Von diesem Erzeugungs-Muster existieren nur zwei Abweichungen: das Laden aus einem String (durch `LoadXml()` bzw. `InnerXml` oder `OuterXml`) sowie der Import von Knoten aus anderen XML-Dokumenten. Beide Varianten werden noch gezeigt.

Das erste *Kunde*-Element und seine Attribute entstehen wieder nach dem typische Erzeugungs-Muster des DOM. Auffallend ist dabei, dass über die `Value`-Eigenschaft nur der Wert eines Attributs gesetzt wird. Zugriffe auf den Wert eines Elements geschehen über verschiedene andere Eigenschaften: `InnerText` enthält den gesamten Text des Elements und seiner Kind-Elemente, `InnerXml` enthält das gleiche plus die XML-Tags der Kind-Elemente und ihrer Attribute und `OuterXml` enthält zusätzlich die XML-Tags des Elements selbst. Beim Setzen dieser Eigenschaften gehen also eventuell vorhandene Kind-Elemente verloren. Um das zu vermeiden, erzeugt man einen Knoten vom Typ `XmlText` und fügt ihn dann dem Element hinzu. Zum Zugriff auf ihren Inhalt bieten Knoten des Typs `XmlText` die Eigenschaft `Data`.

Elemente und Attribute des zweiten Kunden entstehen über die `InnerXml`-Eigenschaft des *Kunde*-Elements aus einem String.

Das Schreiben des ersten XML-Dokuments erledigt die `XmlTextWriter`-Klasse, die dabei gleich noch für eine vernünftige Formatierung sorgt. Um Sonderzeichen nicht ersetzen zu müssen, erfolgt die Speicherung in der Kodierung *ISO-8859-1* (siehe Abschnitt »Zeichensätze in XML«, Seite 753). `XmlDocument.Save()` würde zwar ebenfalls die Speicherung erlauben, bietet aber keine Möglichkeit zur Festlegung des Zeichensatzes.

Den dritten Kunden erzeugt die Methode `ErzeugeXmlMitWriter()` mit der `XmlTextWriter`-Klasse.

```
private void ErzeugeXmlMitWriter()
{
  //Ohne Formatierung schreiben
  XmlTextWriter w = new XmlTextWriter(
    Application.StartupPath + "\\Kunden2.Xml",
    Encoding.GetEncoding("ISO-8859-1"));

  try
  {
    w.WriteStartDocument();
    w.WriteComment("Zweites XML-Dokument");
    //Man könnte auch eine PI mit WriteProcessingInstruction() erzeugen,
    //oder Elemente einem Namensraum zuordnen
    w.WriteStartElement("Kunden");
    //Dritten Kunden erzeugen
    w.WriteStartElement("Kunde");
    //-Element mit WriteStartElement() / WriteEndElement() erzeugen
    w.WriteStartElement("VorName");
    w.WriteString("Hans");
    w.WriteEndElement();//VorName
    //-Element mit WriteElementString() erzeugen
    w.WriteElementString("NachName", "Müller");
    //-Element mit WriteStartElement() / WriteEndElement() erzeugen
    w.WriteStartElement("TelefonFax");
    //--Attribut mit WriteStartAttribute() / WriteEndAttribute() erzeugen
    w.WriteStartAttribute("Vorwahl", "");
    w.WriteString("04321");
    w.WriteEndAttribute();//Vorwahl
    //--Attribute mit WriteAttributeString()erzeugen
    w.WriteAttributeString("Einwahl", "555");
    w.WriteAttributeString("Telefon", "66");
    w.WriteAttributeString("Fax", "77");
    w.WriteEndElement();//TelefonFax
    //-Element mit WriteStartElement() / WriteEndElement() erzeugen
    w.WriteStartElement("Adresse");
    //--Attribute mit WriteAttributeString()erzeugen
    w.WriteAttributeString("Stadt", "Bonn");
    w.WriteAttributeString("Plz", "2223");
    w.WriteAttributeString("Straße", "Heimhöhe 13");
    w.WriteEndElement();//Adresse
    //-Element mit WriteStartElement() / WriteEndElement() erzeugen
    w.WriteStartElement("Umsatz");
    //--Attribute mit WriteAttributeString()erzeugen
    w.WriteAttributeString("Vorjahr", "20000");
    w.WriteAttributeString("Rabatt", "9");
    w.WriteEndElement();//Umsatz
    w.WriteEndElement();//Kunde
    w.WriteEndElement();//Kunden
    w.WriteEndDocument();
    w.Flush();
  }
```

```
  finally
  {
    w.Close();
  }
}//ErzeugeXmlMitWriter
```

Hier übernimmt die `XmlTextWriter`-Instanz also nicht nur das Schreiben der Datei, sondern auch das Erzeugen des Inhalts. Im Gegensatz zur DOM-basierten Variante muss man hier selbst über Methoden wie `WriteStartDocument()`, `WriteEndDocument()`, `WriteStartElement()`, `WriteEndElement()` und `WriteStartAttribute()`, `WriteEndAttribute()` für das korrekte Öffnen und Schließen der Tags sorgen. Für einfache Elemente und Attribute stehen auch Methoden wie `WriteElementString()` und `WriteAttributeString()` zur Verfügung. `XmlTextWriter` basiert auf der abstrakten Klasse `XmlWriter` und ist beim Zugriff auf XML-Daten mit weniger Überbau verbunden als `XmlDocument`-Instanzen.

Jetzt zum dritten Schritt der Kundenvermehrung, dem Import von Knoten aus anderen XML-Dokumenten durch die Methode `ErzeugeXmlMitImport()`.

```
private void ErzeugeXmlMitImport()
{
  //Die ersten beiden Kunden sind schon da
  XmlDocument doc = new XmlDocument();
  doc.Load(Application.StartupPath + "\\Kunden1.Xml");

  XmlNode Kunden = doc.SelectSingleNode("Kunden");
  XmlNode Kunde = null;

  //Den dritten Kunden mit XmlTextReader laden
  XmlTextReader tr = new XmlTextReader(
    Application.StartupPath + "\\Kunden2.Xml");

  try
  {
    tr.MoveToContent();//Kunden
    tr.Read();//1. Kunde
    Kunde = doc.CreateElement("Kunde");
    string s = tr.ReadInnerXml();
    Kunde.InnerXml = s; //Geht nur mit Hilfsvariable!?
    Kunden.AppendChild(Kunde);
  }
  finally
  {
    tr.Close();
  }

  //Den vierten Kunden mit ImportNode() erzeugen:
  XmlDocument doc2 = new XmlDocument();
  doc2.LoadXml("<Kunde>" +
    "<VorName>Elke</VorName>" +
    "<NachName>Musterfrau</NachName>" +
    "<TelefonFax Vorwahl='04321' Einwahl='666' Telefon='888' Fax='' />" +
    "<Adresse Stadt='Bonn' Plz='2224' Straße='Im Berg 21'/>" +
```

```
    "<Umsatz Vorjahr='50000' Rabatt='15'/>" +
    "</Kunde>");
  Kunde = doc2.SelectSingleNode("Kunde");
  Kunde = doc.ImportNode(Kunde, true);
  Kunden.AppendChild(Kunde);

  //Resultat speichern
  XmlTextWriter tw = new XmlTextWriter(
    Application.StartupPath + "\\Kunden.Xml",
    Encoding.GetEncoding("ISO-8859-1"));
  try
  {
    tw.Formatting = Formatting.Indented;
    tw.Indentation = 2;
    tw.QuoteChar = '"';
    doc.Save(tw);
  }
  finally
  {
    tw.Close();
  }//finally
}//ErzeugeXmlMitImport
```

Hier entsteht zuerst eine `XmlDocument`-Instanz mit den beiden zuerst erzeugten Kunden. Dann wandert der dritte Kunde in eine `XmlTextReader`-Instanz. Diese Klasse bietet wie `XmlTextWriter` einen Stream-basierten Zugriff auf XML-Daten und Navigationsmethoden wie `MoveToAttribute()` oder `MoveToNextAttribute()`, aber im Gegensatz zur Klasse `XmlDocument` keine Möglichkeit, Daten zu verändern.

Der dritte Kunde wird mit der `ReadInnerXml()`-Methode ausgelesen und über die `InnerXml`-Eigenschaft in eine neues *Kunde*-Element des ersten XML-Dokuments hineinpraktiziert.

Der vierte und letzte Kunde entsteht in einer neuen `XmlDocument`-Instanz. Um ihn der ersten `XmlDocument`-Instanz hinzuzufügen, muss das Programm ihn zuerst deren `ImportNode()`-Methode übergeben. Der zweite Parameter legt dabei fest, ob die Kind-Elemente mit importiert werden. Erst nachdem der Knoten importiert wurde, akzeptiert ihn die `AppendChild()`-Methode.

Zum Schluss übernimmt wieder eine `XmlTextWriter`-Instanz die Formatierung und Speicherung des Dokuments.

Beim Vergleich des Ergebnisses in der Datei *Kunden.Xml* mit dem Erzeugungsaufwand erscheint die Arbeit mit den .NET-Klassen auf den ersten Blick sehr umständlich. Aber in der Praxis wird kaum jemand Daten auf diese Weise generieren. Und abgesehen von der `Value`-Eigenschaft, die unter anderem für Text-Elemente `null` liefert, sind die Objekt-Modelle recht intuitiv geraten.

Der nächste Abschnitt demonstriert, wie sich die gerade erzeugten Daten mit den XML-Klassen von .NET in eine HTML-Datei transformieren lassen.

Codebeispiel – Transformieren von XML mit einem Stylesheet

Diese Transformation benutzt ein schon aus dem Abschnitt »Bedingungen und Prädikate einsetzen« (Seite 769) bekanntes Stylesheet zur Erzeugung der in Abbildung 19.14 wiedergegebenen Fax-Liste.

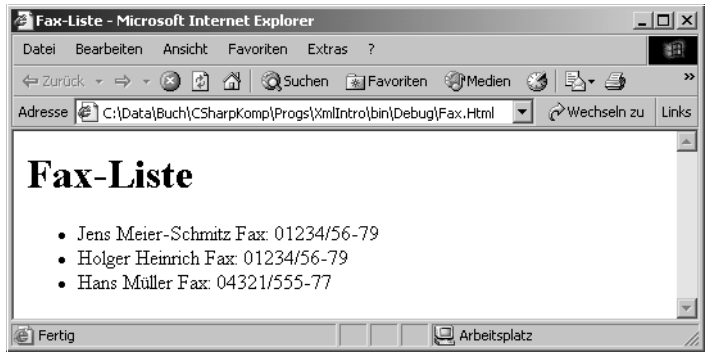


Abbildung 19.14: Das Resultat der Transformation

Hier ist noch einmal der Code des Stylesheets:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- FaxMitPrädikaten.Xsl -->
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:template match="Kunden">
    <html>
      <head>
        <title>Fax-Liste</title>
      </head>
      <body>
        <h1>Fax-Liste</h1>
        <ul>
          <xsl:apply-templates
            select="Kunde[TelefonFax/@Fax and TelefonFax/@Fax != '']"/>
        </ul>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="Kunde">
    <li>
      <xsl:value-of select="VorName"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="NachName"/>
      <xsl:text> Fax: </xsl:text>
      <xsl:value-of select="TelefonFax/@Vorwahl"/>
      <xsl:text>/</xsl:text>
```

```
      <xsl:value-of select="TelefonFax/@Einwahl"/>
      <xsl:text>-</xsl:text>
      <xsl:value-of select="TelefonFax/@Fax"/>
    </li>
  </xsl:template>

</xsl:stylesheet>
```

Dieser Code wurde bereits auf Seite 766 besprochen, weshalb es gleich mit der Behandlungsroutine für Klicks auf die Schaltfläche TRANSFORMIEREN weitergeht.

```
private void btnTransformieren_Click(object sender, System.EventArgs e)
{
  XslTransform xslt = new XslTransform();
  xslt.Load(Application.StartupPath + "\\FaxMitPrädikaten.Xsl");
  xslt.Transform(
    Application.StartupPath + "\\Kunden.Xml",
    Application.StartupPath + "\\Fax.Html");
  Process.Start(Application.StartupPath + "\\Fax.Html");
}//btnTransformieren_Click
```

Die Transformation kapselt ein `XslTransform`-Objekt, das zunächst mit dem Stylesheet initialisiert wird. Seine `Transform()`-Methode erwartet die zu transformierende XML-Datei und den Namen der zu erzeugenden HTML-Datei. Tatsächlich handelt es sich bei dieser Version mit zwei Parametern um eine von neun überladenen Varianten, weshalb man denken sollte, hier für jeden Anlass etwas Passendes zu finden. Der nächste Abschnitt zeigt aber, dass dem leider nicht so ist.

Codebeispiel – Transformieren von XML mit den XML-Klassen

Dieser Abschnitt kommt noch einmal auf das Beispiel aus dem Abschnitt »Gruppieren und Zwischensummen durch Rekursion bilden« (Seite 775) zurück – nur dass es in dem hier vorstellten Beispielprojekt *XMLIntro* nicht um die Implementation über ein Stylesheet, sondern über die entsprechenden XML-Klassen von .NET geht. Abbildung 19.15 zeigt noch einmal das erwartete Ergebnis.

Dieses Beispiel bietet gleich zwei Implementationen. Die zwar nicht weniger kompliziert sind, aber immerhin mehr als 10 Prozent kürzer als das Original mit seinen 106 Zeilen XSLT-Code: Mit `XmlDocument` werden 92, mit `XpathNavigator` 87 Zeilen benötigt. Zunächst die Implementation mit der `XmlDocument`-Klasse:

```
private void btnBerechnen_Click(object sender, System.EventArgs e)
{
  XmlDocument doc = new XmlDocument();
  doc.Load(Application.StartupPath + "\\Kunden.Xml");
  Byte[] b = ErzeugeHtml(doc);
```

Abbildung 19.15: Gruppiert und sortiert mit den XML-Klassen




```
  //In Datei schreiben
  FileStream fs = new FileStream(
    Application.StartupPath + "\\Kunden.Html", FileMode.Create);
  try
  {
    fs.Write(b, 0, b.Length);
  }
  finally
  {
    fs.Close();
    Process.Start(Application.StartupPath + "\\Kunden.Html")
  }
}//btnBerechnen_Click
```

Hier wird das vorhin generierte XML-Dokument geladen, über `ErzeugeHtml()` transformiert und das Ergebnis schließlich in eine Datei geschrieben.

Die für das Erzeugen des HTML-Dokuments zuständige Methode `ErzeugeHtml()` liefert ein `byte`-Array zurück, das gleich über ein `FileStream`-Objekt geschrieben werden kann.

```
private byte[] ErzeugeHtml(XmlDocument doc)
{
  //Kunden nach Namen sortieren
  XslTransform xslt = new XslTransform();
  xslt.Load(Application.StartupPath + "\\Sortierung.Xsl");
```

```
  MemoryStream ms = new MemoryStream();
  XmlTextWriter tw = new XmlTextWriter(
    ms, Encoding.GetEncoding("ISO-8859-1"));
  tw.Formatting = Formatting.Indented;
  tw.Indentation = 2;
  tw.QuoteChar = '"';
  xslt.Transform(doc, null, tw);
  tw.Flush();
  ms.Position = 0;
  doc.LoadXml(Encoding.GetEncoding(
    "ISO-8859-1").GetString(ms.ToArray()));

  StringBuilder sb = new StringBuilder();
  //HTML-Dokument bis zur ersten Tabelle erzeugen
  sb.Append("<html><head>");
  sb.Append("<meta http-equiv='Content-Type' content='text/html'" +
    "charset='ISO-8859-1'/>");
  sb.Append(
    "<style type='text/css'>.Umsatz {text-align: right}</style>");
  sb.Append("<title>Umsätze ohne Rabatt</title></head><body>");
  sb.Append("<h1>Umsätze ohne Rabatt</h1>");
  sb.Append("<p>Nach Stadt gruppiert und nach Nachname sortiert.</p>");
  //Liste der voneinander verschiedenen Städte erzeugen
  XmlNodeList KundenStädte = doc.SelectNodes(
    "Kunden/Kunde[not(Adresse/@Stadt=" +
    "preceding-sibling::Kunde/Adresse/@Stadt)]");
  //Liste der voneinander verschiedenen Städte sortieren
  ArrayList SortierteKundenStädte = new ArrayList(KundenStädte.Count);
  for (int i = 0; i < SortierteKundenStädte.Capacity; i++)
    SortierteKundenStädte.Add(
      KundenStädte[i]["Adresse"].GetAttribute("Stadt", ""));
  SortierteKundenStädte.Sort();
  //Für jede Stadt eine Tabelle mit Überschrift und Summe erzeugen lassen
  for (int i = 0; i < SortierteKundenStädte.Count; i++)
  {
    string Stadt = (string)SortierteKundenStädte[i];
    XmlNodeList Kunden = doc.SelectNodes(
      "Kunden/Kunde[Adresse/@Stadt='" + Stadt + "']");
    sb.Append(ErzeugeHtmlTabelle(Kunden, Stadt));
  }
  //HTML-Dokument schließen und Ergebnis zurückliefern
  sb.Append("</body></html>");
  Encoding enc = Encoding.GetEncoding("ISO-8859-1");
  return enc.GetBytes(sb.ToString());
}//ErzeugeHtml mit Dom
```

Der erste Teil der Methode sortiert die Kunden nach Namen. Diese Aufgabe könnten auch, wie in der nächsten Implementation zu sehen, `XPathNavigator`- und `XPathExpression`-Instanzen erledigen. Hier soll aber eine Lösung implementiert werden, die ausschließlich mit dem XML DOM arbeitet. Da das XML DOM aber keine eingebaute Möglichkeit zum Sortieren bietet, muss hier wieder ein Stylesheet herhalten (eine Implementation mit einer `ArrayList`-Instanz finden Sie weiter unten).

```
<?xml version="1.0"?>
<!-- Sortierung.Xsl -->
<xsl:stylesheet
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      version="1.0">
<!--
  <xsl:output method="xml" encoding="iso-8859-1" version="1.0"
    indent="yes"/>
-->

  <xsl:template match="/Kunden">
    <xsl:copy>
      <xsl:for-each select="Kunde">
        <xsl:sort select="NachName" data-type="text"/>
        <xsl:copy-of select="."/>
      </xsl:for-each>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Die Vorlage im Stylesheet selektiert zuerst das *Kunden*-Element und kopiert es in den Ergebnisbaum. Dann selektiert es alle *Kunde*-Elemente, sortiert sie nach ihrem *NachName*-Element und kopiert sie ebenfalls in den Ergebnisbaum.

Leider kann man das *xsl:output*-Element hier nicht einsetzen, denn die `XslTransform`-Klasse fühlt sich nur zuständig für den Aufbau des Ergebnisbaums, aber nicht für seine Serialisierung.

Das muss wie in den vorhergehenden Beispielen die `XmlTextWriter`-Klasse erledigen. Da sie eher auf das Schreiben in eine Datei spezialisiert ist, sind für die Rückgabe der Daten an die `XmlDocument`-Instanz einige Verrenkungen nötig. Alle hier verwendeten Methoden bieten überladene Versionen mit den verschiedensten Parameter-Typen an, aber leider passt keine Kombination so richtig.

Die Kunden sind jetzt nach Namen sortiert, als Nächstes baut die `StringBuilder`-Instanz den Anfang des HTML-Dokuments auf. Dann liefert die Methode `SelectNodes()` eine `XmlNodeList`-Instanz der unterschiedlichen Städte, dazu wird der gleiche XPath-Ausdruck verwendet wie im ursprünglichen Beispiel. Der Typ `XmlNodeList` wird im XML DOM allgemein zur Zusammenfassung von Knoten benutzt. Daneben existiert noch die Klasse `XmlDocumentFragment`, deren Instanzen Baum-Fragmente verwalten.

Das Sortieren erfolgt zur Abwechslung über eine `ArrayList`-Instanz. Eine Schleife hängt die Namen der Städte an diese Liste an, die anschließend über den Aufruf ihrer Methode `Sort()` alphanumerisch sortiert wird.

Das Erzeugen der Tabellen mit Überschrift und der Summe für jede Stadt übernimmt die Methode `ErzeugeHtmlTabelle()`. Sie erhält dazu als Parameter eine `XmlNodeList`-Instanz mit den Kunden der entsprechenden Stadt sowie den Namen der Stadt. Den Rückgabewert der Methode fügt die `StringBuilder`-Instanz dem HTML-Dokument hinzu und schließt es nach Abarbeitung aller Städte. Schließlich liefert `ErzeugeHtml()` das HTML-Dokument als `byte`-Array zurück.

Der letzte Teil der Implementation ist die Methode `ErzeugeHtmlTabelle()`.

```
private string ErzeugeHtmlTabelle(XmlNodeList Kunden, string Stadt)
{
  StringBuilder sb = new StringBuilder();

  //Überschrift und Kopf für Tabelle erzeugen
  sb.Append("<h2>" + Stadt + "</h2>");
  sb.Append("<table border='1'>");
  sb.Append("<tr><th>Name</th><th>Ort</th>" +
    "<th>Umsatz ohne Rabatt</th></tr>");
  //Einträge erzeugen
  int Summe = 0;
  foreach (XmlNode Kunde in Kunden)
  {
    sb.Append("<tr><td>");
    sb.Append(Kunde["NachName"].InnerText + ", " +
      Kunde["VorName"].InnerText);
    sb.Append("</td><td>");
    sb.Append(Kunde["Adresse"].GetAttribute("Plz", "") + " " +
      Kunde["Adresse"].GetAttribute("Stadt", ""));
    sb.Append("</td><td class='Umsatz'>");
    int Umsatz = int.Parse(Kunde["Umsatz"].GetAttribute("Vorjahr", ""));
    if (Kunde["Umsatz"].GetAttribute("Rabatt", "") != "")
    {
      int Rabatt = int.Parse(Kunde["Umsatz"].GetAttribute("Rabatt", ""));
      int UmsatzOhneRabatt =
        (int)decimal.Round(Umsatz * (100 + Rabatt) / 100, 0);
      Summe += UmsatzOhneRabatt;
      sb.Append(UmsatzOhneRabatt.ToString());
    }
    else
    {
      Summe += Umsatz;
      sb.Append(Umsatz.ToString());
    }
    sb.Append("</td></tr>");
  }
  //Den Rest erzeugen und Ergebnis zurückliefern
  sb.Append("</table>");
  sb.Append("Summe für <b>" + Stadt + "</b>: " + Summe.ToString());
  return sb.ToString();
}//ErzeugeHtmlTabelle mit DOM
```

Über eine `StringBuilder`-Instanz setzt die Methode zuerst Überschrift und Kopf der Tabelle zusammen. Die `foreach`-Schleife generiert dann die einzelnen Zeilen, was mit dem XML DOM recht intuitiv geschieht. Schließlich liefert `ErzeugeHtmlTabelle()` das Resultat als String zurück.

Alles in allem unterscheidet sich die Generierung des HTML-Dokuments mit dem XML DOM wenig von anderen prozeduralen Ansätzen.

Das Gegenteil trifft auf die folgende Implementation mit `XPathNavigator`, `XPathExpression` und anderen speziell auf Transformationen ausgerichteten Klassen zu. Mit ihnen lässt sich die Behandlungsroutine für Klicks auf BERECHNEN folgendermaßen schreiben:

```
private void btnBerechnen_Click(object sender, System.EventArgs e)
{
  XPathDocument doc = new XPathDocument(
      Application.StartupPath + "\\Kunden.Xml");
  XPathNavigator nav = doc.CreateNavigator();
  Byte[] b = ErzeugeHtml(nav);
  //In Datei schreiben
  FileStream fs = new FileStream(
    Application.StartupPath + "\\Kunden.Html", FileMode.Create);
  try
  {
    fs.Write(b, 0, b.Length);
  }
  finally
  {
    fs.Close();
  }
}//btnBerechnen_Click
```

Da hier keine Manipulation über das XML DOM stattfinden wird, kann eine `XPathDocument`-Instanz statt einer `XmlDocument`-Instanz das Laden der XML-Datei übernehmen. Diese Instanz liefert hier ein Objekt vom Typ `XPathNavigator`, das ist eine Art Cursor zur Navigation im XML-Dokument. Die `XPathNavigator`-Instanz ist hier nötig, weil nur sie XPath-Ausdrücke kompilieren kann.

Die Verwendung der `XPathNavigator`-Klasse zeigt die überarbeitete Methode `ErzeugeHtml()`. Dieser Methode wird jetzt eine `XPathNavigator`-Instanz übergeben, statt der `XmlDocument`-Instanz in der Implementation mit dem XML DOM.

```
private byte[] ErzeugeHtml(XPathNavigator nav)
{
  StringBuilder sb = new StringBuilder();
  //HTML-Dokument bis zur ersten Tabelle erzeugen
  sb.Append("<html><head>");
  sb.Append("<meta http-equiv='Content-Type' content='text/html'" +
    "charset='ISO-8859-1'/>");
```

```
  sb.Append(
    "<style type='text/css'>.Umsatz {text-align: right}</style>");
  sb.Append("<title>Umsätze ohne Rabatt</title></head><body>");
  sb.Append("<h1>Umsätze ohne Rabatt</h1>");
  sb.Append("<p>Nach Stadt gruppiert und nach Nachname sortiert.</p>");
  //Sortierte Liste der voneinander verschiedenen Städte erzeugen
  XPathExpression xp = nav.Compile(
    "Kunden/Kunde[not(Adresse/@Stadt=" +
    "preceding-sibling::Kunde/Adresse/@Stadt)]");
  xp.AddSort("Adresse/@Stadt", XmlSortOrder.Ascending,
    XmlCaseOrder.None, "", XmlDataType.Text);
  XPathNodeIterator KundenStädte = nav.Select(xp);
  XPathExpression xpStadt = KundenStädte.Current.Compile(
    "string(Adresse/@Stadt)");
  while (KundenStädte.MoveNext())
  {
    //HTML für jede Stadt erzeugen lassen
    string Stadt = (string)KundenStädte.Current.Evaluate(xpStadt);
    XPathExpression xpKunden = nav.Compile(
      "Kunden/Kunde[Adresse/@Stadt='" + Stadt + "']");
    xpKunden.AddSort("NachName", XmlSortOrder.Ascending,
      XmlCaseOrder.None, "", XmlDataType.Text);
    XPathNodeIterator Kunden = nav.Select(xpKunden);
    sb.Append(ErzeugeHtmlTabelle(Kunden, Stadt));
  }
  //HTML-Dokument schließen und Ergebnis zurückliefern
  sb.Append("</body></html>");
  Encoding enc = Encoding.GetEncoding("ISO-8859-1");
  return enc.GetBytes(sb.ToString());
}//ErzeugeHtml mit XPathNavigator
```

`ErzeugeHtml()` beginnt wie gehabt mit dem Aufbau des HTML-Kopfs und der Überschrift. Selektion und Sortierung der Kundenstädte übernimmt jetzt aber eine `XPathExpression`-Instanz. Diese lässt sich in etwa mit Stored Procedures vergleichen und hat keinen öffentlichen Konstruktor: Instanzen von `XPathExpression` kann ausschließlich die Factory-Methode `XPathNavigator.Compile()` liefern.

Zur Selektion dient wieder der gleiche XPath-Ausdruck wie in den bisherigen Implementationen. Die Methode `AddSort()` fügt dann der `XPathExpression`-Instanz eine Sortierung hinzu. Das ist wesentlich eleganter als das, was bei der Implementation mit dem XML DOM zu sehen war.

Die `Select()`-Methode der Klasse `XPathNavigator` führt schließlich Selektion und Sortierung durch. Statt einer `XPathExpression`-Instanz könnte man hier auch einen XPath-Ausdruck übergeben, aber dabei ließe sich keine Sortierung festlegen. Zusätzlich müsste man natürlich überlegen, ob ein bei jedem Zugriff wiederholtes Analysieren des XPath-Ausdrucks mehr Nutzen bringt als das einmalige Kompilieren und Speichern. Und sicher würde man die `XPathExpression`-Instanzen nicht bei jedem Methoden-Aufruf neu erzeugen, das geschieht hier nur der besseren Übersicht wegen.

Das Resultat der `Select()`-Methode der Klasse `XPathNavigator` ist eine `XPathNodeIterator`-Instanz. Sie bietet ähnliche Navigationsmöglichkeiten wie die in der vorherigen Implementation benutzte `XmlNodeList`-Klasse, ist aber ausschließlich für Lesezugriffe verwendbar.

Die `while`-Schleife zeigt eine weitere Anwendung von `XPathExpression`-Instanzen: die Auswertung durch die `Evaluate()`-Methode der Klasse `XPathNavigator`. Wichtig dabei ist, dass der XPath-Ausdruck kein Element oder Attribut liefert, sondern durch eine XPath-Funktion wie *string()* den Wert des Elements oder Attributs.

Die Selektion der Kunden nach Wohnort und ihre Sortierung nach Nachnamen erfolgt wiederum über eine `XPathExpression`-Instanz. Das Resultat wird dann der Methode `ErzeugeHtmlTabelle()` übergeben.

```
private string ErzeugeHtmlTabelle(
  XPathNodeIterator Kunden, string Stadt)
{
  XPathExpression xpStadt = Kunden.Current.Compile(
    "string(Adresse/@Stadt)");
  XPathExpression xpName = Kunden.Current.Compile(
    "concat(string(NachName), ', ', string(VorName))");
  XPathExpression xpOrt = Kunden.Current.Compile(
    "concat(string(Adresse/@Plz), ' ', string(Adresse/@Stadt))");
  XPathExpression xpHatRabatt = Kunden.Current.Compile(
    "boolean(Umsatz/@Rabatt and Umsatz/@Rabatt !='')");
  XPathExpression xpUmsatz = Kunden.Current.Compile(
    "number(Umsatz/@Vorjahr)");
  XPathExpression xpUmsatzOR = Kunden.Current.Compile(
    "round(Umsatz/@Vorjahr * (100 + Umsatz/@Rabatt) div 100)");

  //Überschrift und Kopf für Tabelle erzeugen
  StringBuilder sb = new StringBuilder();
  sb.Append("<h2>" + Stadt + "</h2>");
  sb.Append("<table border='1'>");
  sb.Append("<tr><th>Name</th><th>Ort</th>" +
    "<th>Umsatz ohne Rabatt</th></tr>");
  //Einträge erzeugen
  int Summe = 0;
  int Umsatz = 0;
  while (Kunden.MoveNext())
  {
    sb.Append("<tr><td>");
    sb.Append((string)Kunden.Current.Evaluate(xpName));
    sb.Append("</td><td>");
    sb.Append((string)Kunden.Current.Evaluate(xpOrt));
    sb.Append("</td><td class='Umsatz'>");
    //oder mit GetAttribute, dann aber vorher auf Umsatz-Element
    //positionieren:
    if ((bool)Kunden.Current.Evaluate(xpHatRabatt))
      Umsatz = Convert.ToInt32(Kunden.Current.Evaluate(xpUmsatzOR));
```

```
    else
      Umsatz = Convert.ToInt32(Kunden.Current.Evaluate(xpUmsatz));
    Summe += Umsatz;
    sb.Append(Umsatz.ToString() + "</td></tr>");
  }
  //Den Rest erzeugen und Ergebnis zurückliefern
  sb.Append("</table>");
  sb.Append("Summe für <b>" + Stadt + "</b>: " + Summe.ToString());
  return sb.ToString();
}//ErzeugeHtmlTabelle mit XPathNodeIterator
```

`ErzeugeHtmlTabelle()` macht extensiven Gebrauch der Klasse `XPathExpression`, die hier sowohl zum Ermitteln der einzutragenden Werte als auch zum Test auf das Vorhandensein des *Rabatt*-Attributs benutzt wird. Auch hier liefern XPath-Funktionen wie *string()*, *concat()* und *boolean()* das Ergebnis im gesuchten Typ.

Nachdem die Auswertungslogik in den `XPathExpression`-Instanzen steckt, beschränkt sich das Ermitteln der Werte auf den Aufruf der `Evaluate()`-Methode und die Umwandlung in den richtigen Typ. Obwohl die XPath-Funktion *round()* in *xpUmsatzOR* einen Integer-Wert zurückliefert, muss auch der Umsatz ohne Rabatt mit der `Convert()`-Methode in einen Integer-Wert umgewandelt werden – sonst gibt es eine Fehlermeldung.

19.3.2 Validieren von XML

Die bisherigen XML-Dokumente waren alle WOHLGEFORMT, das heißt, ihr Inhalt entsprach dem XML-Standard. Und der Versuch, ein nicht wohlgeformtes XML-Dokument zu laden, würde eine Ausnahme auslösen. Die automatische Prüfung auf Einhaltung des XML-Standards ist eine Art der in diesem Abschnitt besprochenen Validierung.

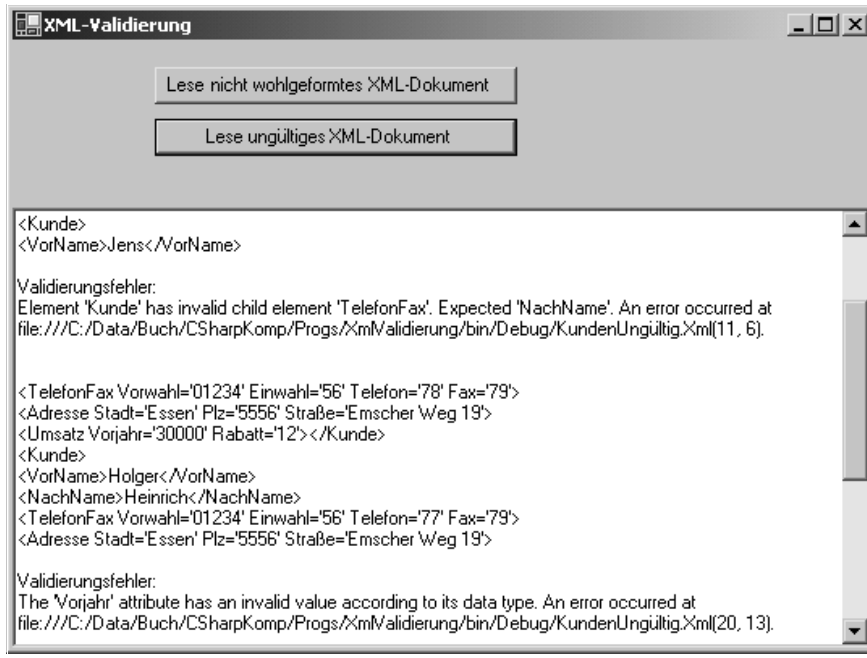
Für XML-Dokumente können Sie zusätzlich eigene, über den XML-Standard hinausgehende, Regeln festlegen. Diese Regeln können zum Beispiel das Vorhandensein und die Reihenfolge von Elementen oder den Wertebereich von Attributen bestimmen. Ein XML-Dokument, das diesen zusätzlichen Regeln entspricht, ist GÜLTIG. Die optionale Prüfung auf Einhaltung dieser zusätzlichen Regeln ist die zweite Art der hier besprochenen Validierung.

Natürlich muss jedes gültige XML-Dokument auch wohlgeformt sein – sonst ist es per Definition kein XML-Dokument.

Codebeispiel – Validieren von XML

Das folgende Beispielprojekt *XmlValidierung* zeigt anhand verschiedener XML-Dokumente die Möglichkeiten und Auswirkungen der Validierung.

Abbildung 19.16: Validieren eines ungültigen XML-Dokuments



Zum Nachprogrammieren dieses Beispiels brauchen Sie ein Formular mit zwei Schaltflächen, einem Textfeld und den in Tabelle 19.1 gezeigten Eigenschaftswerten.

Tabelle 19.1: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	`Text`	XML-Validierung
Obere Schaltfläche	`Name`	btnLeseNichtWohlgeformtesXmlDokument
Obere Schaltfläche	`Text`	Lese nicht wohlgeformtes XML-Dokument
Untere Schaltfläche	`Name`	btnLeseUngültigesXmlDokument
Untere Schaltfläche	`Text`	Lese ungültiges XML-Dokument
Textfeld	`Name`	txtResultat
Textfeld	`Dock`	Bottom
Textfeld	`MultiLine`	true
Textfeld	`ScrollBars`	Both
Textfeld	`Text`	(leer)

Gehen Sie dann in die Code-Ansicht und fügen Sie folgende `uses`-Anweisungen hinzu:

```
using System.Text;
using System.Xml;
using System.Xml.Schema;
```

Deklarieren Sie noch eine globale Variable.

```
private StringBuilder m_sb = null;
```

Hier ist der Code für die obere Schaltfläche und die Methode `ZeigeErgebnis()`:

```
private void btnLeseNichtWohlgeformtesXmlDokument_Click(
  object sender, System.EventArgs e)
{
    XmlTextReader r = new XmlTextReader(
      Application.StartupPath + "\\KundenElementFehler.Xml");
    ZeigeErgebnis(r);
}//btnLeseNichtWohlgeformtesXmlDokument_Click

private void ZeigeErgebnis(XmlReader r)
{
  m_sb = new StringBuilder();
  try
  {
    while (r.Read())
   {
      switch (r.NodeType)
      {
        case XmlNodeType.Comment:
          m_sb.Append("\r\n<!--" + r.Value + "-->");
          break;
        case XmlNodeType.Element:
          m_sb.Append("\r\n<" + r.Name);
          while (r.MoveToNextAttribute())
          {
            m_sb.Append(" " + r.Name + "='" + r.Value + "'");
          }
          m_sb.Append(">");
          break;
        case XmlNodeType.EndElement:
          m_sb.Append("</" + r.Name + ">");
          break;
        case XmlNodeType.Text:
          m_sb.Append(r.Value);
          break;
        case XmlNodeType.XmlDeclaration:
          m_sb.Append("<?xml version='1.0'?>");
          break;
      }//switch
    }//while
  }//try
```

```
  catch (Exception x)
  {
    //Diese Ausnahme bricht das Parsing ab
    m_sb.Append("\r\n\r\nFehler:\r\n" + x.Message + "\r\n");
  }
  txtResultat.Text = m_sb.ToString();
}//ZeigeErgebnis
```

Das Einlesen des XML-Dokuments geschieht wie gehabt über eine `XmlTextReader`-Instanz – nur lässt der Dokumentname das drohende Unheil wohl erahnen. Das Einlesen über eine `XmlDocument`-Instanz würde übrigens dieselben Effekte zeigen, da `XmlDocument` zum Lesen intern ebenfalls `XmlTextReader` benutzt. Die Methode `ZeigeErgebnis()` wertet in einer `while`-Schleife die von der `XmlTextReader`-Instanz gelieferten Knoten aus und schreibt sie schließlich in das Textfeld. Die Auswertung in der `switch`-Anweisung ist hier recht einfach, aber man kann sich leicht vorstellen, wie aufwändig die Verarbeitung komplizierterer XML-Dokumente wird – dann doch lieber DOM.

Da das zu lesende Dokument *KundenElementFehler.Xml* ein nicht geschlossenes Element enthält, kann der Parser es nicht auswerten und löst eine Ausnahme aus. Das geschieht bei allen Verstößen gegen die XML-Regeln, also auch bei fehlenden Hochkommas um Attribut-Werte oder bei falsch verschachtelten Elementen. Durch entsprechende Manipulationen des Dokuments lässt sich der Parser auf die Probe stellen.

Nach Entfernen des nicht geschlossenen Elements akzeptiert der Parser das Dokument. Aber der Code aus den bisherigen Beispielen für die Auswertung würde natürlich scheitern, denn er erwartet, dass der Aufbau des Dokuments einem bestimmten Schema folgt. So muss zum Beispiel jedes *Kunde*-Element ein *VorName*- und ein *NachName*-Element enthalten. Erfreulicherweise kann der Parser das prüfen, wozu er natürlich das Schema des Dokuments benötigt.

Die Beschreibung dieses Schemas erfolgt in der XML Schema Definition Language (XSD), ein Standard des World Wide Web Consortiums (W3C). Der Standard wird hier nicht im Detail erläutert, nähere Informationen finden Sie unter den Links am Kapitelanfang.

Statt nun lange Dokumentationen zu wälzen, lassen Sie sich die XSD-Datei einfach von einem Tool erzeugen. Rufen Sie dazu über die Kommandozeile von Visual Studio (Start/Programme/Microsoft Visual Studio .NET/Visual Studio .NET Tools/Visual Studio .NET-Eingabeaufforderung) das Tool *XSD.EXE* mit vollständigem Pfad und Namen der Datei *Kunden.Xml* auf. Wichtig ist natürlich, dazu keine der XML-Dateien mit eingebauten Fehlern zu benutzen.

Die im Folgenden benutzte XSD-Datei für *Kunden.Xml* sieht so aus:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="Kunden" xmlns=""
  xmlns:xs="http://www.w3.org/2001/XMLSchema" >
  <xs:element name="Kunden">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Kunde">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="VorName" type="xs:string" />
              <xs:element name="NachName" type="xs:string" />
              <xs:element name="TelefonFax">
                <xs:complexType>
                  <xs:attribute name="Vorwahl" type="xs:string" />
                  <xs:attribute name="Einwahl" type="xs:string" />
                  <xs:attribute name="Telefon" type="xs:string" />
                  <xs:attribute name="Fax" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="Adresse">
                <xs:complexType>
                  <xs:attribute name="Stadt" type="xs:string" />
                  <xs:attribute name="Plz" type="xs:integer" />
                  <xs:attribute name="Straße" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="Umsatz">
                <xs:complexType>
                  <xs:attribute name="Vorjahr" type="xs:integer" />
                  <xs:attribute name="Rabatt" type="xs:integer"
                    default="0" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Gegenüber der generierten Originalversion fehlen die Festlegungen zur Datenbindung, außerdem wurden einige Attribut-Typen von *string* auf *integer* umdeklariert und für das *Rabatt*-Attribut ein *Default*-Wert vorgegeben.

Dazu gehört natürlich eine entsprechend präparierte Version von *Kunden.Xml*.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!--KundenUngültig.Xml:
  beim 1. Kunden fehlt NachName-Element
  beim 2. Kunden ist Vorjahr-Attribut des Umsatz-Elements leer
-->
```

```
<Kunden
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="Kunden.xsd">
  <Kunde>
    <VorName>Jens</VorName>
    <TelefonFax Vorwahl="01234" Einwahl="56" Telefon="78" Fax="79" />
    <Adresse Stadt="Essen" Plz="5556" Straße="Emscher Weg 19" />
    <Umsatz Vorjahr="30000" Rabatt="12" />
  </Kunde>
  <Kunde>
    <VorName>Holger</VorName>
    <NachName>Heinrich</NachName>
    <TelefonFax Vorwahl="01234" Einwahl="56" Telefon="77" Fax="79" />
    <Adresse Stadt="Essen" Plz="5556" Straße="Emscher Weg 19" />
    <Umsatz Vorjahr="" />
  </Kunde>
  (Restliche Kunde-Elemente der Übersichtlichkeit wegen entfernt)
</Kunden>
```

Die Datei *KundenUngültig.Xml* verweist in ihrem *Kunden*-Element auf die bereits vorgestellte Schemadatei *Kunden.Xsd*. Das XML entspricht aber diesem Schema nicht, denn zum einen fehlt ein *NachName*-Element, und zum anderen ist ein *Vorjahr*-Attribut leer. Dieses Attribut darf nicht leer sein, weil es in der Schemadatei als *integer* deklariert wurde.

Nach diesen Vorbereitungen nun endlich zurück zum Code. Die zur Demonstration dieses Fehlers verwendete Methode sieht so aus:

```
private void btnLeseUngültigesXmlDokument_Click(
  object sender, System.EventArgs e)
{
  XmlTextReader r = new XmlTextReader(
    Application.StartupPath + "\\KundenUngültig.Xml");
  XmlValidatingReader vr = new XmlValidatingReader(r);
  vr.ValidationEventHandler +=
    new ValidationEventHandler(BehandleValidierungsfehler);
  ZeigeErgebnis(vr);
}//btnLeseUngültigesXmlDokument_Click
```

Das Einlesen besorgt auch hier wieder eine `XmlTextReader`-Instanz. Zur Validierung kommt eine `XmlValidatingReader`-Instanz hinzu, die automatisch das in *KundenUngültig.Xml* referenzierte XSD-Dokument lädt. Bei einem häufiger verwendeten XSD-Dokument lohnt sich das Caching über die `Schemas`-Eigenschaft des `XmlValidatingReader`-Objekts.

Bevor das Lesen beginnt, erhält die `XmlValidatingReader`-Instanz noch eine Behandlungsroutine für Validierungsfehler, die den Eintrag der Fehlermeldung in ein Textfeld übernimmt.

```
private void BehandleValidierungsfehler(
  object sender, ValidationEventArgs args)
{
  //Nach dieser Ausnahme geht das Parsen weiter
  m_sb.Append("\r\n\r\nValidierungsfehler:\r\n" +
    args.Message + "\r\n\r\n");
}//BehandleValidierungsfehler
```

Wenn eine Behandlungsroutine eingesetzt wird, geht das Lesen des Dokuments auch nach einem Verstoß gegen das Schema weiter. Ohne diese Zuordnung würde eine Ausnahme ausgelöst, und das Lesen wäre beendet.

Da die `Load()`-Methode der `XmlDocument`-Klasse eine `XmlValidatingReader`-Instanz akzeptiert, funktioniert die Validierung im XML DOM genauso wie mit der `XmlTextReader`-Instanz.

Durch XML-Schemas lässt sich also der korrekte Aufbau von XML-Dokumenten recht wartungsfreundlich sicherstellen. Ohne das Schema müsste das Programm die im Screenshot zu sehenden Fehlermeldungen selbst generieren, was im Detail eine Menge Arbeit macht und die eigentliche Programmlogik verdeckt. Allerdings kostet die Validierung auch nicht gerade wenig Rechenzeit, weshalb man sie sobald wie möglich ausschalten wird. Die XML-Dokumente können dabei unverändert bleiben, das Setzen der `ValidationType`-Eigenschaft der `XmlValidatingReader`-Instanz auf `ValidationType.None` genügt.

Die Benutzung von XML-Schemas bringt noch einen zusätzlichen Nutzen: das automatische Eintragen von Standardvorgaben. Auf diese Weise ist der zweite Kunde zu einem *Rabatt*-Attribut mit dem Wert *0* gekommen. Der auswertende Code könnte den Umsatz ohne Rabatt nun ohne Prüfung des Attributs ausrechnen. Allerdings muss man auch hier den Nutzen gegen den Performance-Verlust abwägen. Im Allgemeinen wird es besser sein, beim Generieren des XML-Dokuments auch das Attribut und seinen Standardwert zu erzeugen.

20 Einführung in verteilte Programme mit .NET

Der Begriff VERTEILTE PROGRAMME beschreibt Programme, deren Bestandteile auf verschiedenen Rechnern laufen, die über ein Netzwerk verbunden sind. Aus wie vielen Teilen ein solches Programm besteht, hängt von den jeweiligen Anforderungen ab. Je nach Anzahl der Bestandteile spricht man von einem Zweischicht-, Dreischicht- oder n-Schichtprogramm. (Der englische Begriff »multi-tiered« ist übrigens der Seefahrt entlehnt und wie üblich recht plakativ: »tiers« steht in diesem Fall für die Ruderreihen auf einer Galeere.)

Im nächsten Abschnitt finden Sie eine kurze Erläuterung der Beweggründe für die Schichtenaufteilung. Danach geht es um die technische Realisierung, also die Antwort auf die Frage, welche Möglichkeiten .NET zur Implementierung der Programmteile und der Kommunikation zwischen ihnen bietet. Abbildung 20.1 zeigt einige typische Mehrschichtsysteme unter .NET.

20.1 Beweggründe für die Schichtenaufteilung

Solange ein Programm von einem einzelnen Benutzer an einem einzelnen Rechner ausgeführt werden kann, genügt für die Datenverwaltung ein Desktop-Datenbanksystem wie Access, Paradox oder DBase. Die Aufteilung des Programms in Benutzerschnittstelle, Geschäftslogik und Datenzugriff ist meist nur logischer Natur und nicht zwingend – sie erleichtert lediglich die Wartung des Programms. Die vorangehenden Kapitel haben sich auf Programme dieser Art konzentriert.

Daten, die mehreren Benutzern gemeinsam zur Verfügung stehen sollen, legt man auf einen Datei-Server. Im Einschichtsystem liegt die zur Bearbeitung benötigte Datenbank-Engine nicht auf dem Server, sondern immer noch lokal auf den Systemen der Benutzer. Deshalb müssen die Daten zur Bearbeitung automatisch auf den lokalen Rechner kopiert werden. Ein besonderes Kommunikationsprotokoll ist dafür nicht nötig, die Übertragung der Daten regeln die beteiligten Betriebssysteme automatisch. Dieser Ansatz ist mit einem halben Dutzend Benutzern und geringen Datenmengen durchaus normal und vertretbar.

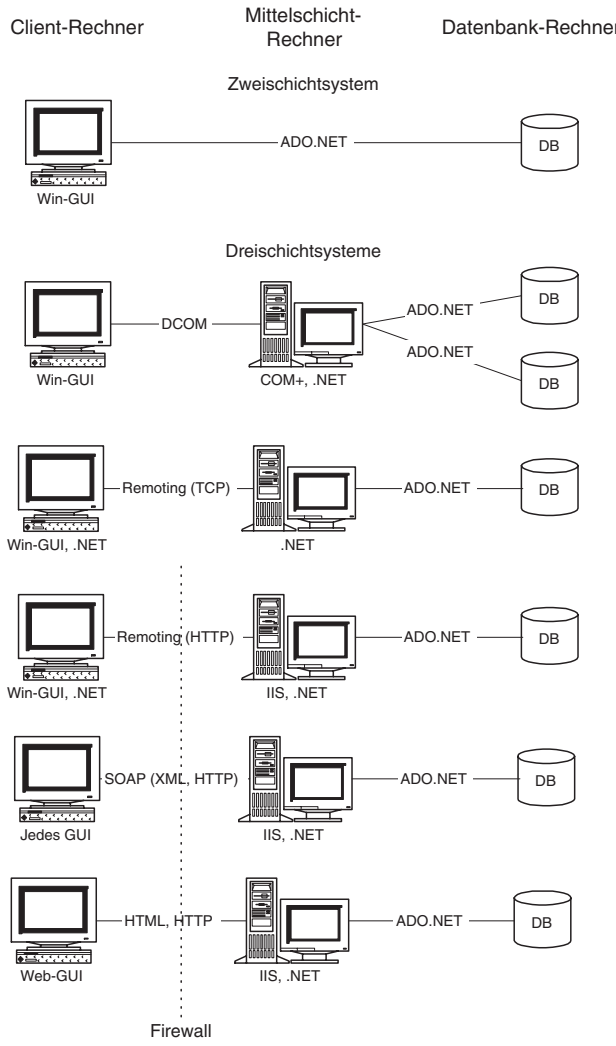


Abbildung 20.1: Typische Mehrschichtsysteme unter .NET

Werden Datenmengen und/oder Benutzerzahlen größer, leidet zuerst die Performance und schließlich auch die Stabilität des Systems. Deshalb benutzt man dafür andere Datenbanksysteme, deren Engines auf dem Server liegen, dort (lokal) die Daten verarbeiten, die Abfrageergebnisse zusammenstellen, und nur die Ergebnisse dieser Operationen zum jeweiligen Client senden. Mit dem Aufbau solcher Systeme befasst sich dieser Teil des Buches.

20.1.1 Zweischichtsysteme

Zweischichtsysteme findet man häufig als Industriesteuerungen: Eine Schicht bildet der Datenbankrechner, die andere Schicht bilden einige Dutzend Client-Rechner in der Werkshalle. In der ursprünglichen Variante des

Zweischichtsystems stellen die Clients beim Hochfahren eine Verbindung zur Datenbank her. Diese Verbindung bleibt dann bis zum Herunterfahren des Clients bestehen. Jede dieser Verbindungen belegt natürlich Ressourcen auf dem Datenbankrechner.

Auf den Client-Rechnern laufen im Allgemeinen so genannte FAT CLIENTS. »Fat« heißt, sie enthalten sehr viel Geschäftslogik. Einer der Gründe dafür ist, dass die Clients in einer prozeduralen Programmiersprache geschrieben werden. Diese bietet wesentlich mehr Ausdrucksmöglichkeiten und damit eine schnellere Fertigstellung des Programms als die Programmiersprachen der Datenbanksysteme. Außerdem beschränken sich die Programmierumgebungen der Datenbanksysteme bei der Fehlersuche oft auf Funktionen, die eher an die 70er Jahre erinnern. Und schließlich hat jedes Datenbanksystem seine eigene Syntax und seine eigenen Datentypen, Datenbank-Code ist also oft nicht portabel. Der andere Grund für die Verwendung von Fat Clients liegt in der Entlastung des Datenbankrechners. Denn zum einen wachsen die Kosten für die Hardware überproportional zur Größe, zum anderen sind die Lizenzgebühren für großkalibrige Datenbanken erschreckend hoch.

Das Kommunikationsprotokoll zwischen Clients und Server wird vom Datenzugriff bestimmt. Jeder Client-Rechner benötigt dazu eine spezielle Software, welche die eigentliche Kommunikation mit der Datenbank übernimmt. Beispielsweise benutzt eine Oracle-Datenbank das SQLNet-Protokoll (auf den Client-Rechnern wird der SQLNet-Client installiert); Interbase verwendet eine spezielle TCP/IP-Variante, die auf den Clients eine entsprechende DLL erfordert usw.

Wichtig für den erfolgreichen Einsatz eines Zweischichtsystems sind also drei Dinge: Erstens muss die Anzahl der Clients in einem für diese Architektur akzeptablen Rahmen bleiben. Zweitens sollten sie räumlich eng beieinander stehen, weil die Client-Software bei Änderungen der Geschäftslogik ausgetauscht werden muss. Und drittens muss sich die EDV-Umgebung kontrollieren lassen, weil die proprietären Kommunikationsprotokolle oft schon an der nächsten Firewall scheitern.

Diese so genannten CLIENT-/SERVER-SYSTEME sind durch ihre geringere Komplexität und die entfallenden Kosten für Middleware (siehe weiter unten) deutlich kostengünstiger als Dreischichtsysteme.

20.1.2 Dreischichtsysteme

Bei Dreischichtsystemen kommt gegenüber den Zweischichtsystemen noch eine MITTLERE SCHICHT dazu, deren Hauptaufgabe auch hier die Entlastung des Datenbankrechners ist: Dazu wird ein Teil der Geschäftslogik von den Client-Rechnern auf einen speziellen Rechner verlagert, und nur dieser Rechner kommuniziert direkt mit der Datenbank.

Dass die gesamte Geschäftslogik hier in der mittleren Schicht liegt, ist allerdings eher eine Lehrbuchmeinung: Um den Benutzern Wartezeiten und dem System unnötige Belastungen zu ersparen, wird der Client-Rechner auch hier alles entscheiden, was er alleine entscheiden kann. Und manche Operationen lassen sich nicht nur am sichersten, sondern auch hinsichtlich der Performance am besten als Stored Procedure der Datenbank implementieren – also direkt auf dem Server.

Die Geschäftslogik auf dem Mittelschichtrechner benutzt dort im Allgemeinen eine zugekaufte Infrastruktur, die so genannte MIDDLEWARE. Diese Software-Bausteine, wie Object Broker und Transaktionsmonitore, übernehmen große Teile der für diese Architektur nötigen Infrastrukturaufgaben.

Die Kommunikation zwischen Mittelschicht- und Datenbankrechner läuft wie bei den Zweischichtsystemen ab. Zur Kommunikation zwischen Client- und Mittelschichtrechner wird das von der Mittelschicht bevorzugte Protokoll verwendet. Auch dieses Protokoll ist proprietär, es gelten also die gleichen Einschränkungen wie bei der Kommunikation mit der Datenbank.

Dreischichtsysteme können mehrere 10 000 Benutzer versorgen, sind aber wegen des Protokoll-Problems auf kontrollierte Umgebungen beschränkt.

20.1.3 n-Schichtsysteme

n-Schichtsysteme fügen den drei bereits beschriebenen Schichten noch eine oder mehrere hinzu. Beispielsweise kann eine dieser zusätzlichen Schichten das Protokoll-Problem lösen, indem sie das HTTP-Protokoll zur Kommunikation mit den Clients benutzt. (Das heißt übrigens nicht zwingend, dass die Clients mit einer HTML-Oberfläche arbeiten müssen.) Eine andere Schicht kann die Lastverteilung zwischen mehreren Rechnern der Mittelschicht übernehmen, indem sie Anforderungen ihrer Clients in einer Warteschlange speichert. Denn meistens ist eine Abarbeitung in Echtzeit nicht nötig; es reicht, dass sie zeitnah erfolgt.

n-Schichtsysteme lösen also alle bisher angesprochenen Probleme, sind aber naturgemäß entsprechend komplex und deshalb aufwändiger bei der Implementierung und Wartung.

Bis vor wenigen Jahren waren n-Schichtsysteme in kleinen und mittleren Projekten nicht zu realisieren, weil sich die Hersteller die entsprechenden Software-Bausteine ausgesprochen gut bezahlen ließen. Das hat sich inzwischen – vor allem Dank Microsoft – geändert.

20.2 Technologien für Mehrschichtsysteme unter .NET

Dieser Abschnitt gibt einen Überblick, welche technischen Prinzipien Microsoft bei der Implementation mehrschichtiger Systeme in der .NET-Laufzeitumgebung eingesetzt hat.

20.2.1 Technologien der Datenbankschicht

Bei den Datenbanken bleibt alles beim alten: Sie können in Lizenzgebühren für ein Datenbanksystem wie Oracle oder Microsoft SQL Server investieren oder Freeware-Systeme wie PostgresSql oder Interbase einsetzen. Bis auf den Microsoft SQL Server laufen alle diese Datenbanksysteme auch auf mindestens einem nicht-Windows Betriebssystem (wie beispielsweise Linux).

Die Kommunikation mit der Datenbank geschieht über ADO.NET. Die entscheidenden Neuerungen gegenüber ADO sind zum einen die `DataSet`-Klasse, die mehrere Tabellen und die Beziehungen zwischen ihnen enthalten kann – theoretisch also auch die ganze Datenbank. Eine `DataSet`-Instanz kann ohne Verbindung zur Datenbank existieren, Änderungen an den Daten zwischenspeichern und sich selbst serialisieren. Außerdem ist XML integraler Bestandteil von ADO.NET (während es bei ADO noch ein nachträglich aufgepfropfter Zusatz war). Beispielsweise ist der gesamte Inhalt der `DataSet`-Instanz über die neue `XmlDataDocument`-Klasse als XML verfügbar. Alles in allem ist ADO.NET spezialisiert auf den Einsatz in Drei- und n-Schichtsystemen.

20.2.2 Technologien der Mittelschicht

In der Mittelschicht benutzen Sie die COM+ Services. Diese waren unter Windows NT 4 noch Einzelprodukte: Microsoft Transaction Server (MTS) und Microsoft Message Queue Server (MSMQ). Die MTS-Dienste bieten neben der namengebenden Transaktionsunterstützung beispielsweise Object Pooling und Just-In-Time Activation sowie ein ausgefeiltes Sicherheitsmodell.; die MSMQ-Dienste können unter anderem mit asynchroner Kommunikation durch Message Queues aufwarten. Durch transaktionale Message Queues geht eine eingestellte Nachricht garantiert nicht verloren. Dadurch lassen sich diese Dienste auch zur Lastverteilung einsetzen.

Allerdings wurden die COM+ Services unverändert ins .NET Framework übernommen, basieren also auf COM. Dementsprechend müssen Sie beim Einsatz der COM+ Services mit zwei Nachteilen leben. Zum einen müssen Sie die betroffenen Assemblies als COM-Komponenten registrieren. Zum anderen leidet die Performance bei der Kommunikation mit dem unverwalteten COM+ Code.

20.2.3 Technologien zur Client-Kommunikation

In den Technologien zur Client-Kommunikation stecken die entscheidenden Neuerungen der .NET-Architektur gegenüber der bisher von Microsoft propagierten Distributed interNet Applications Architecture (DNA). Entscheidend ist die Abkehr von COM mit seiner inhärenten Komplexität und den Problemen des proprietären DCOM-Protokolls. Welcher COM-Programmierer hat nicht schon Stunden mit dem Aufspüren irgendwelcher esoterischer Konfigurationsprobleme verbracht und dabei bis zum Ellenbogen in der Registry gesteckt!

Die .NET-Architektur ersetzt registrierungsbedürftige COM-Komponenten durch selbstbeschreibende Assemblies; das Firewall-geschädigte DCOM musste Web Services und .NET Remoting weichen, die auf Web Standards wie HTTP und XML beruhen. Auf gut Deutsch: (D)COM ist »legacy«, aus Pietätsgründen sagt man aber »classic«.

Die Internet Information Services (IIS) aus der DNA gibt es immer noch, und sie bekommen zusätzliche Bedeutung: Neben der Versorgung von Web Clients mit HTML-Oberfläche sind sie jetzt auch für die Web Services und das Remoting über den HTTP-Kanal zuständig. Der Client kann damit seine Windows-Oberfläche behalten, aber das DCOM-Protokoll wird durch das HTTP-Protokoll ersetzt. Bei den Web Services werden die über HTTP transportierten Daten im SOAP-Protokoll (Simple Object Access Protocol) verpackt. Dieses Protokoll ist standardisiert, und damit sind zum ersten Mal plattformübergreifende n-Schichtsysteme wirklich realisierbar: Windows-Clients können einen Linux Web Service benutzen; umgekehrt – mit einem Linux-Client und einem Windows Web Service – funktioniert es genauso.

Durch die Plattformunabhängigkeit des IL-Codes besteht außerdem die Aussicht, dass ursprünglich für Windows geschriebene Assemblies später einmal auch auf anderen Betriebssystemen laufen werden. Zum Beispiel befasst sich das Mono-Projekt mit der Portierung von .NET auf Linux, und Microsoft portiert den Kern von .NET auf FreeBSD.

20.3 Internet-Programmierung mit .NET

Dieser Abschnitt präsentiert Informationen und Beispiele zu den Klassen, die .NET für den Zugriff auf Ressourcen im Internet bietet. (Wenn Sie noch keine Erfahrung mit den Internet-Protokollen haben oder Ihre Kenntnisse auffrischen wollen, lesen Sie Anhang C, Internet-Protokolle.) Die wichtigsten Punkte der folgenden Seiten im Überblick:

- *IE-Hosting* – darunter versteht man das Einbinden des Internet Explorers in ein Programm. Dadurch kann das Programm nicht nur normale Webseiten anzeigen, sondern auch noch auf diverse andere Ressourcen zugreifen.
- `DNS`-Klasse – diese Klasse ermittelt zu einer IP-Adresse den Host-Namen und umgekehrt.
- `TcpClient`-Klasse – mit dieser Klasse lassen sich unter anderem Whois-Abfragen durchführen und Mails empfangen.
- `SmtpMail`-, `MailMessage`-, `WebClient`- und `WebRequest`-Klasse – diese Klassen dienen zum Senden von Mails und für Suchanfragen im Internet.
- `WebClient`-Klasse – mit dieser Klasse lässt sich ein einfacher Screen Scraper basteln.
- `WebRequest`- und `WebResponse`-Klasse – diese Klassen bauen mit wenigen Zeilen ein Framework zum transparenten Zugriff auf beliebige Ressourcentypen auf.

Eine Implementation mit einfachen TCP- oder UDP-Clients und -Servern finden Sie hier nicht – dazu enthält die Online-Dokumentation ausreichend Beispiele.

20.3.1 IE-Hosting

Durch IE-Hosting lassen sich mit wenig Aufwand beeindruckende Berichte, Grafiken und Animationen oder Videos in Anwendungen integrieren, weil dabei einfach die Darstellungsmöglichkeiten des Internet Explorers (IE) verwendet werden. Diese erschöpfen sich nicht in der Darstellung von HTML-, GIF- und JPEG-Dateien. Mit entsprechenden Stylesheets ist auch die Anzeige von XML-Dateien möglich, und reine Textdaten kann der IE natürlich auch darstellen. Außerdem beherrscht er auch exotischere Formate, wie zum Beispiel das Microsoft-eigene Vektorgrafikformat VML (Vector Markup Language). Und schließlich gibt es noch eine Vielzahl von Plug-Ins, beispielsweise erhalten Sie bei `www.adobe.com` ein Plug-In zur Darstellung des vom W3C standardisierten Vektorgrafikformats SVG (Scalable Vector Graphics). Tatsächlich können Sie mit IE-Hosting nicht nur alles darstellen, was Sie jemals im Internet gesehen haben – da die Daten lokal verfügbar sind, bereiten auch schicke Animationen und Videos keine Probleme.

Die Grenzen des IE-Hosting treten eher beim Drucken auf. Kopf- und Fußzeilen sind ein Problem, ebenso der Seitenumbruch an den passenden Stellen. Bei solchen Anforderungen empfiehlt sich eher der Einsatz von Word, das sich auch recht gut programmieren und in ein anderes Programm integrieren lässt.

Codebeispiel – Webbrowser mit dem Explorer-Steuerelement

Das in diesem Abschnitt vorgestellte Beispielprojekt *IEHosting* benutzt den Internet Explorer als ganz normalen Webbrowser: Sie können eine beliebige Internetadresse eintippen und mit VOR- und ZURÜCK-Schaltflächen navigieren.

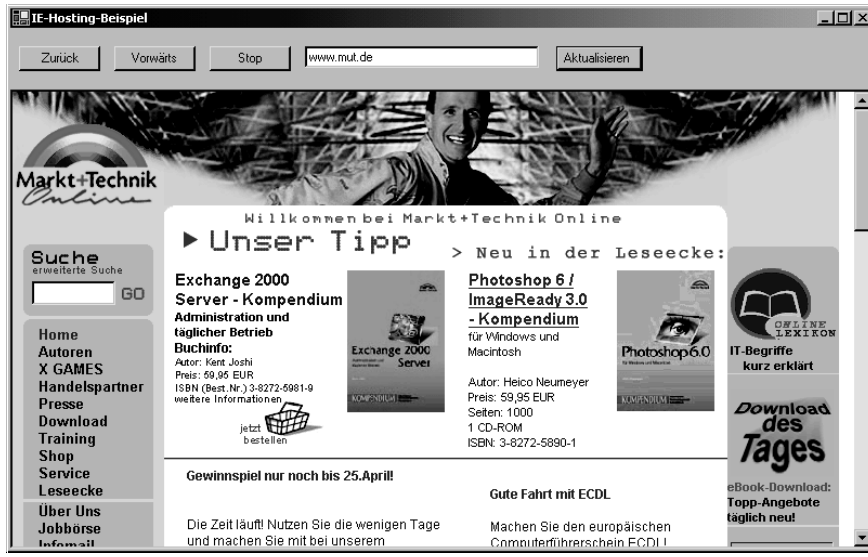


Abbildung 20.2: Ein einfacher Webbrowser mit dem IE-Steuerelement

Das `Explorer`-Steuerelement gehört nicht zum Standardinhalt der TOOLBOX. Sie fügen es folgendermaßen hinzu:

1. Befehl TOOLBOX ANPASSEN im Kontextmenü der TOOLBOX wählen
2. Register COM-STEUERELEMENTE wählen
3. Eintrag MICROSOFT WEBBROWSER (*%WinDir%\System32\ShDocVw.Dll*) wählen
4. OK klicken.

Das `Explorer`-Steuerelement erscheint jetzt als letzter Eintrag in Ihrer TOOLBOX. Da der IE über eine COM-Schnittstelle programmiert wird, sind beim Einbinden automatisch zwei Dateien im Unterverzeichnis *...\bin\Debug* des Projekts entstanden: *AxInterop.SHDocVw.dll* zur Interaktion mit dem Formular und *Interop.SHDocVw.dll* zur Interaktion mit dem Code.



Das Generieren der Dateien dauert auf einem 350 MHz Pentium ca. 8 Minuten, also nicht irritieren lassen.

Die Schaltfläche zum Navigieren und das Textfeld zur Eingabe der Internetadresse setzen Sie auf ein `Panel`-Steuerelement. Normalerweise würde man hier zuerst an eine Symbolleiste denken, darauf lassen sich aber direkt keine Textfelder platzieren. Also behilft man sich mit einem `Panel`-Steuerelement.

Setzen Sie also ein `Panel`-Steuerelement auf das Formular und stellen Sie die `Dock`-Eigenschaft des Panels auf `Top`. Ziehen Sie dann eine Instanz des `Explorer`-Steuerelements auf das Formular und stellen Sie seine `Dock`-Eigenschaft auf `Fill`. Es ist wichtig, zuerst das `Panel`-Steuerelement auf das Formular zu setzen, weil sonst das `Explorer`-Steuerelement den gesamten Ausgabebereich des Formulars füllt, ohne Platz für das `Panel`-Steuerelement zu lassen.

Setzen Sie nun vier Schaltflächen und ein Textfeld auf das `Panel`-Steuerelement, wie in Abbildung 20.2 zu sehen, und initialisieren Sie die in Tabelle 20.1 aufgeführten Eigenschaften.

Tabelle 20.1: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	`Text`	IE-Hosting-Beispiel
Erste Schaltfläche	`Name`	btnVorwärts
Erste Schaltfläche	`Text`	Vorwärts
Erste Schaltfläche	Code im Click-Ereignis	axWebBrowser1.GoForward();
Zweite Schaltfläche	`Name`	btnZurück
Zweite Schaltfläche	`Text`	Zurück
Zweite Schaltfläche	Code im Click-Ereignis	axWebBrowser1.GoBack();
Dritte Schaltfläche	`Name`	btnStop
Dritte Schaltfläche	`Text`	Stop
Dritte Schaltfläche	Code im Click-Ereignis	axWebBrowser1.Stop();
Textfeld	`Name`	txtAdresse
Textfeld	`Text`	(leer)
Vierte Schaltfläche	`Name`	btnAktualisieren
Vierte Schaltfläche	`Text`	Aktualisieren

Der Code für die Behandlung des `Click`-Ereignisses der AKTUALISIEREN-Schaltfläche lautet:

```
private void Aktualisieren_Click(object sender, System.EventArgs e)
{
  object flagArg = 0;
  object strArg = "";

  axWebBrowser1.Navigate(
    txtAdresse.Text, ref flagArg, ref strArg, ref strArg, ref strArg);
}
```

Fertig ist der Webbrowser!

Was der IE sonst noch kann

Der IE kann nicht nur sehr verschiedene Daten darstellen, er unterstützt auch erstaunlich viele Protokolle. Bei der Eingabe von Internetadressen wie `www.mut.de` verwendet er automatisch das HTTP-PROTOKOLL. Wenn Sie dagegen eine FTP-Adresse eingeben, verwendet er automatisch das FTP-PROTOKOLL. Bei `ftp.microsoft.com` zeigt Ihnen der IE also die Verzeichnisstruktur des Microsoft Download-Bereichs und Sie können sich darin bewegen wie in Ihrem lokalen Dateisystem.

Mit dem FILE-PROTOKOLL können Sie auf Dateien Ihrer Festplatte zugreifen. Allerdings werden diese nicht im IE angezeigt, sondern wie bei einem Doppelklick geöffnet. Also Vorsicht mit *BAT*-Dateien. Probieren Sie es einmal mit `"file:///c:/boot.ini"` oder schlicht `"c:/boot.ini"` aus. Auch mit diesem Protokoll können Sie sich Verzeichnisse ansehen.

Daneben gibt es auch exotischere Protokolle. So packt das ABOUT-PROTOKOLL jeden eingegebenen Text in HTML ein und stellt ihn sofort dar. Probieren Sie es mit `"about:Hallo Welt"` aus und lassen Sie sich den Quelltext anzeigen. Die Länge des ausgebbaren Textes ist allerdings begrenzt.

Und damit stellt sich die Kernfrage des IE-Hosting: Wie stelle ich damit eigene, lokale Inhalte dar? Die einfachste Antwort sind natürlich lokale HTML-Dateien, einschließlich verlinkter Stylesheets und Grafiken. Auch für dynamisch generierte Inhalte ist das meistens akzeptabel.

Aber was ist, wenn Sie die Inhalte nicht auf die Festplatte schreiben können oder wollen? Nun, dann können Sie sich die nächstaufwändigere Lösung ansehen: das RES-PROTOKOLL. Mit diesem Protokoll können Sie Ihre Inhalte aus beliebigen Dateien laden, beispielsweise aus einer Ressourcen-DLL. Der Benutzer sieht nur die DLL und zumindest Nicht-Programmierer haben keinen Zugriff auf die eigentlichen Daten. Auf diese Weise werden auch die Fehlermeldungen des IE gespeichert. Geben Sie einfach mal eine falsche Internetadresse an, und sobald die Fehlermeldung erscheint, sehen Sie in der Adresszeile ihre Herkunft. Im IE-Hosting-Beispiel müssen Sie dazu mit der rechten Maustaste in den Ausgabebereich klicken und im Kontextmenü den Befehl EIGENSCHAFTEN wählen. Im EIGENSCHAFTEN-Fenster sehen Sie dann den Adresseintrag.

Die Königsklasse des IE-Hosting stellen Asynchronous Pluggable Protocol (APP) Handler dar (eine offizielle deutsche Übersetzung scheint es für diesen Begriff nicht zu geben). Durch diese kaum auszusprechenden Wunderdinge können Sie Ihre Inhalte auch dynamisch generieren. Denn mit einem APP Handler definieren Sie ein neues, eigenes Protokoll. So hat es Microsoft mit seinem ms-help-Protokoll getan, auf dem die Hilfe von Visual Studio basiert.



Dank des ms-help-Protokolls können Sie die Online-Dokumentation von Visual Studio auch im Internet Explorer lesen, was bei umfangreichen Themen und längerem Lesen recht angenehm ist. Um die URL zu erhalten, klicken Sie mit der rechten Maustaste auf das Thema, wählen im Kontextmenü den Befehl Eigenschaften und kopieren den Text unter Adresse in die Adressleiste des IE.

Alternativ können Sie auch die externe Hilfe benutzen. Zwischen interner und externer Hilfe schalten Sie folgendermaßen um: Menü Extras, Befehl Optionen, Ordner Umgebung, Eintrag Hilfe, entsprechende Option wählen. Die neue Option wird erst nach Neustart von Visual Studio wirksam.

Für jede Ressource, die der IE über dieses Protokoll laden will, ruft er Ihren APP Handler auf. Dann können Sie diese Ressource aus der Datenbank laden, im Code zusammenbasteln, beim Laden dekomprimieren oder was auch immer. Ein APP Handler ist eigentlich ein recht einfaches Programm, das vier Methoden bereitstellt und in der Registry eingetragen sein muss. Unter *HKEY_CLASSES_ROOT\PROTOCOLS\Handler\ms-help* finden Sie zum Beispiel den Eintrag, der auf den APP Handler für die Hilfe von Visual Studio verweist. Leider muss jeder APP Handler drei COM-Schnittstellen (`IInternet`, `IInternetProtocolRoot`, `IInternetProtocol`) implementieren. Das ist nicht trivial und eine genauere Beschreibung sprengt den Rahmen dieses Buches. Dasselbe gilt für die Ersetzung des IE-Kontextmenüs durch ein eigenes: Dazu müssen Sie die COM-Schnittstelle `IDocHostUi` implementieren.



Um das Kontextmenü nur zu unterdrücken, können Sie das folgende <body>-Tag in Ihren HTML-Dokumenten benutzen: <body oncontextmenu="return false">.

Damit keine Missverständnisse entstehen: Fast alles, was in diesem Abschnitt besprochen wurde, funktioniert nur mit dem Internet Explorer als Browser, und das auch nur mit aktuellen Versionen.

20.3.2 DNS-Abfragen mit der DNS-Klasse

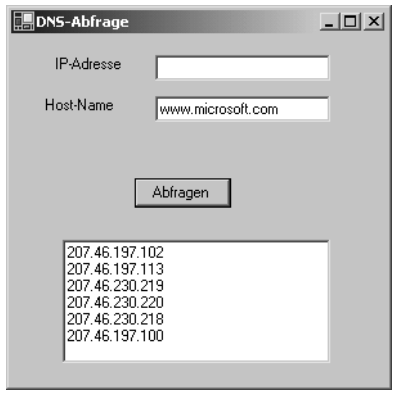
Eine DNS-Abfrage (DNS = Domain Name System) liefert zu einem Host-Namen wie `www.microsoft.com` die zugehörige IP-Adresse, zum Beispiel 207.46.197.100. Eine DNS-Abfrage führt Ihr Browser immer automatisch durch, denn zur internen Kommunikation benutzen die beteiligten Systeme ausschließlich die IP-Adresse.

Das ganze geht auch umgekehrt und heißt dann DNS-Rückwärts-Abfrage (Reverse Lookup). Intelligente Tools wie *tracert* führen diese Abfrage automatisch durch. Oft haben Sie aber nur die IP-Adresse, und dann ist eine erfolgreiche DNS-Rückwärts-Abfrage hilfreich. Wie das Wort »erfolgreich« schon andeutet, kann eine DNS-Rückwärts-Abfrage auch scheitern: Es lässt sich einfach nicht zu jeder IP-Adresse ein Host-Name ermitteln.

Codebeispiel – DNS-Abfragen und -Rückwärts-Abfragen

Das Beispielprojekt *DNS* demonstriert beide Arten der Abfrage und benutzt dazu die `DNS`-Klasse aus `System.Net`.

Abbildung 20.3: Eine DNS-Abfrage



Legen Sie ein neues Projekt vom Typ *Windows-Anwendung* an und setzen Sie zwei `Label`-Steuerelemente, drei Textfelder und eine Schaltfläche auf das Formular. Stellen Sie dann die in Tabelle 20.2 gezeigten Eigenschaften ein.

Tabelle 20.2: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	`Text`	DNS-Abfrage
Erstes Label	`Text`	IP-Adresse
Zweites Label	`Text`	Host-Name
Erstes Textfeld	`Name`	txtIpAdresse

Steuerelement	Eigenschaft	Wert
Erstes Textfeld	Text	207.46.197.100
Zweites Textfeld	Name	txtHostName
Zweites Textfeld	Text	www.microsoft.com
Schaltfläche	Name	btnAbfragen
Schaltfläche	Text	Abfragen
Drittes Textfeld	Name	txtResultat
Drittes Textfeld	Text	(leer)
Drittes Textfeld	MultiLine	true

Tabelle 20.2: Eigenschaftswertei (Forts.)

Gehen Sie dann in die Code-Ansicht und fügen Sie die Direktive `using System.Net;` am Dateianfang ein. Deklarieren Sie ein Datenfeld in der Formularklasse.

```
private bool m_IpAdresseAuswerten = false;
```

Ergänzen Sie schließlich noch die folgenden Ereignisbehandlungen:

```
private void btnAbfragen_Click(object sender, System.EventArgs e)
{
  string s;

  try
  {
    if (m_IpAdresseAuswerten)
    {
      IPAddress ip = IPAddress.Parse(txtIpAdresse.Text);
      IPHostEntry he = Dns.GetHostByAddress(ip);

      if (he.Aliases.Length == 0)
        s = he.HostName;
      else
        s = string.Join("\r\n", he.Aliases);
    }//if (m_IpAdresseAuswerten)
    else
    {
      IPHostEntry he = Dns.GetHostByName(txtHostName.Text);
      IPAddress [] ip = he.AddressList;
      s = "";
      for (int i = 0; i < ip.Length; i++)
        s += ip[i].ToString() + "\r\n";
    }//else
    txtResultat.Text = s;
  }//try
```

```
  catch (Exception x)
  {
    MessageBox.Show(x.Message);
  }//catch
}//btnAbfragen_Click


private void txtIpAdresse_KeyPress(
  object sender, System.Windows.Forms.KeyPressEventArgs e)
{
  m_IpAdresseAuswerten = true;
}


private void txtHostName_TextChanged(object sender, System.EventArgs e)
{
  m_IpAdresseAuswerten = false;
}
```

Die Variable `m_IpAdresseAuswerten` erspart dem Programm eine zweite Schaltfläche. `btnAbfragen_Click()` benötigt eine Fehlerbehandlung, denn hier ist wirklich eine notwendig. Zum einen, weil man schnell mal einen Tippfehler macht, besonders bei den IP-Adressen. Zum anderen, weil das DNS Reverse Lookup nicht von jedem DNS-Server unterstützt wird, und die DNS-Klasse auf abgewiesene Anfragen mit einer Ausnahme reagiert. Das ist auch der Grund, warum in diesem Beispiel `"www.microsoft.com"` statt beispielsweise `"www.mut.de"` benutzt wird. Der Code selbst sollte einigermaßen selbsterklärend sein. Auffällig ist hier zum einen, dass von der `DNS`-Klasse nur statische Methoden benutzt werden; zum anderen, dass es sowohl mehrere Host-Namen zu einer IP-Adresse als auch mehrere IP-Adressen zu einem Host-Namen geben kann.

20.3.3 Whois-Abfragen mit der TcpClient-Klasse

Eine Whois-Abfrage (Who is ... = Wer ist ...) liefert Informationen über den Eigentümer einer Domain oder einer IP-Adresse. Ursprünglich war das zum Lösen technischer Probleme gedacht. Heute handelt es sich bei den zu lösenden Problemen leider hauptsächlich um Spam- (Werbung) oder Hacker-Angriffe. Durch die Informationen aus einer Whois-Abfrage finden Sie einen Ansprechpartner für entsprechende Beschwerden.

Bis 1999 wurden Domains vom InterNIC, das heißt von der Firma Network Solutions in den USA verwaltet. Die USA haben dieses Monopol inzwischen aufgegeben, und je nach Art und geographischer Lage der Domain ist jetzt einer der Network Solutions-Konkurrenten zuständig. Da es keine einheitliche Datenbank gibt, muss man sich die nötigen Informationen oft aus mehreren Quellen zusammenstellen. Gute Beschreibungen zum Verfolgen von Spammern und Hackern finden Sie beispielsweise unter `www.apnic.net/info/faq/abuse/index.html`.

Erfreulicherweise gibt es aber auch Quellen, die zuständigkeitsübergreifend Auskunft geben, zum Beispiel `www.allwhois.com` oder `www.samspade.org`.

Mit dem folgenden Beispielprogramm können Sie verschiedene Verzeichnisse abfragen. Dabei bieten

1. APNIC (Asian Pacific Network Information Centre) für den Asiatisch/Pazifischer Raum
2. ARIN (American Registry for Internet Numbers) für Amerika und südliche Teile Afrikas
3. RIPE NCC (Réseaux IP Européens, Network Coordination Center) für Europa und Nordafrika

auch Informationen zu IP-Adressen. DeNIC (für deutsche Domains) dagegen bietet nur Informationen zu Domains.

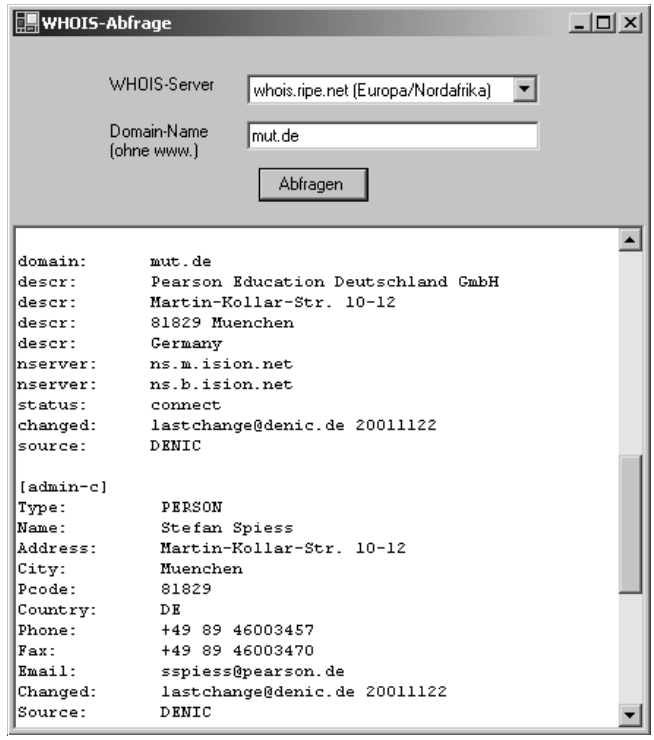


Abbildung 20.4: Eine Whois-Abfrage

Codebeispiel – Whois-Abfrage

Das Beispielprojekt *WHOIS* zeigt, wie das geht. Legen Sie ein neues Formular an und setzen Sie darauf ein `Panel`-Steuerelement und ein Textfeld. Stellen Sie die `Docking`-Eigenschaft des `Panel`-Steuerelements auf `Top` und die des Textfelds auf `Fill`. Die weiteren Eigenschaftswerte zeigt Tabelle 20.3.

Tabelle 20.3: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	`Text`	Whois-Abfrage
Formular	`AcceptButton`	btnAbfragen
Erstes Label	`Text`	Whois-Server
Zweites Label	`Text`	Domain-Name (ohne www.)
Kombinationslistenfeld	`Name`	cboWhoisServer
Kombinationslistenfeld	`DropDownStyle`	DropDownList
Oberes Textfeld	`Name`	txtDomainName
Oberes Textfeld	`Text`	mut.de
Schaltfläche	`Name`	btnAbfragen
Schaltfläche	`Text`	Abfragen
Unteres Textfeld	`Name`	txtResultat
Unteres Textfeld	`Text`	(leer)
Unteres Textfeld	`Font`	Courier New
Unteres Textfeld	`MultiLine`	True
Unteres Textfeld	`ScrollBars`	Vertical

Tragen Sie dann noch die folgenden Einträge in die `Items`-Eigenschaft des Kombinationslistenfelds ein:

```
whois.apnic.net (Asien/Pazifik)
whois.arin.net (Amerika / Südl. Afrika)
whois.denic.de (Deutschland)
whois.internic.net (.com, .net, .edu, .org)
whois.networksolutions.com (.com, .net, .edu, .org)
whois.ripe.net (Europa/Nordafrika)
```

Leider gibt es keine Möglichkeit, den ausgewählten Eintrag zur Design-Zeit festzulegen. Die folgende Zeile in der `Form1_Load()`-Routine verhindert, dass das Kombinationslistenfeld beim Programmstart keinen Wert anzeigt.

```
cboWhoisServer.SelectedIndex = 5; //Ripe
```

Fügen Sie `using`-Anweisungen für `System.Net.Sockets` und `System.IO` hinzu. Tragen Sie dann den folgenden Code als Behandlung für das `Click`-Ereignis der Schaltfläche ein:

```
private void btnAbfragen_Click(object sender, System.EventArgs e)
{
  TcpClient cli = null;
  NetworkStream ns = null;
  string WhoisServer = "";
  txtResultat.Text = "";

  try
  {
    Cursor.Current = Cursors.WaitCursor;
    WhoisServer =
      cboWhoisServer.Text.Substring(0, cboWhoisServer.Text.IndexOf(" "));
    //Anfrage senden
    cli = new TcpClient(WhoisServer, 43);
    ns = cli.GetStream();
    StreamWriter sw = new StreamWriter(ns);
    sw.Write(txtDomainName.Text + "\r\n"); //"\r\n" verhindert Hängen
    sw.Flush();
    //Resultat zeigen
    StreamReader sr = new StreamReader(ns);
    string s =  sr.ReadToEnd();
    //Ripe, DeNIC und NetworkSolutions benutzen die Zeichen x000A+x0025
    //als Zeilenumbrüche im Vorspann. Die eigentlichen Daten sind mit
    //dem Zeichen x000A unterteilt. Andere Formate, wie das von ApNIC,
    //werden hier nicht berücksichtigt.
    //\x000A=\n; \x000D=\r; \x0025=%
    s = s.Replace("\x000A\x0025", "\x000D\x000D");
    s = s.Replace("\x000A", "\x000D\x000A");
    s = s.Replace("\x000D\x000D", "\x000D\x000A");
    s = s.Replace("\x0025", "");
    txtResultat.Text = s;
    //Aufräumen
    ns.Close();
    cli.Close();
  } //try
  catch (Exception x)
  {
    MessageBox.Show(x.Message);
  } //catch
  finally
  {
    //Aufräumen
    if (ns != null)
      ns.Close();
    if (cli != null)
      cli.Close();
    Cursor.Current = Cursors.Default;
  } //finally
}   //btnAbfragen_Click
```

Schauen Sie sich den Code im Detail an. Weil die Abfrage durchaus mehrere Sekunden dauern kann, setzt man den Sanduhr-Cursor – der natürlich auch nach Auftreten einer Ausnahme wieder zurückgesetzt werden muss. Der erste echte Arbeitsschritt des Programms besteht in der Abfrage des ausgewählten Kombinationslistenfeld-Eintrags, also des Namens des Whois-Servers.

Jetzt kommt die eigentliche Abfrage. Dazu benutzt das Programm die `TcpClient`-Klasse und übergibt ihrem Konstruktor den Namen des Servers und den zu kontaktierenden Port. Whois-Abfragen sprechen immer den Port Nummer `43` an. Bei diesem Konstruktor verbindet sich die `TcpClient`-Instanz sofort mit dem Server. Alternativ könnte man auch den parameterlosen Konstruktor aufrufen und die eigentliche Verbindung dann mit der `Connect()`-Methode herstellen.

Die auszutauschenden Daten werden sehr schön von der `NetworkStream`-Klasse gekapselt und schließlich von einer `StreamReader`-Instanz in einen String kopiert. Nach ein bisschen Formatierungsarbeit wird die Antwort schließlich im unteren Textfeld gezeigt.

Wichtig ist, alle Aufräumarbeiten auch im Falle einer Ausnahme durchzuführen. Wenn keine Verbindung mit dem Server zu Stande kommt, scheitert der Konstruktor der `TcpClient`-Klasse. Aus diesem Grund muss das Programm vor dem Schließen der Verbindung und der `NetworkStream`-Instanz auf eine `null`-Referenz prüfen.

20.3.4 Mails senden mit den Klassen SmtpMail und MailMessage

Mit der `SmtpMail`-Klasse können Sie Mails aus Ihren Programmen versenden. Das Akronym SMTP steht für Simple Mail Transfer Protocol und definiert den Standard für die Übertragung von Mails im Internet. Es beschränkt sich allerdings auf die Übertragung zum nächsten Server und umfasst nicht die Verteilung zu den Clients. Dafür ist das POP3-Protokoll zuständig, dessen Besprechung Sie im nächsten Abschnitt finden.

Einsatzbeispiele für das Senden von Mails aus Programmen sind ein automatischer Notruf an den Administrator wegen einer überlaufenden Festplatte oder das Senden von Mails aus einem Web-Formular. Das Versenden von Mails mit der `SmtpMail`-Klasse ist unter optimalen Bedingungen in einer Zeile erledigt.

Allerdings liegt ein großes Problem im Schaffen der »optimalen Bedingungen«. Denn zum einen ist die `SmtpMail`-Klasse als Wrapper um die CDO Version 2 bzw. CDOSYS (Collaboration Data Objects for Windows 2000) implementiert. Dadurch steht diese Funktionalität nur auf Windows 2000, Windows .NET Server und Windows XP zur Verfügung – fehlt also unter Windows 9x und Windows NT. Dort sind über Outlook (nicht Outlook

Express) oder Exchange zwar die CDO Version 1.x bzw. CDONTS installiert, diese basieren aber auf MAPI (Messaging API). MAPI ist ein Microsoft-eigener Kommunikationsstandard, der zum Beispiel auch Faxdienste unterstützt. CDOSYS basiert dagegen auf SMTP und NNTP (Network News Transport Protocol), zwei Internet-Standards. CDOSYS ist zu seinen Vorgängern völlig inkompatibel, allerdings sind die Vorgängerversionen aus Gründen der Rückwärtskompatibilität auch in Windows 2000 verfügbar. Als ActiveX-Objekte werden die CDOSYS über COM Interop angesprochen. Bei einem so einfachen Protokoll wie SMTP stellt sich übrigens die Frage, warum bei der Implementation der `SmtpMail`-Klasse der Umweg über die CDOSYS gegangen wurde.

»Optimale Bedingungen« muss auch Ihr Mail-Server herstellen, das kann beispielsweise der SMTP-Dienst des IIS sein oder ein Exchange-Server. Beide müssen für das Weiterleiten konfiguriert sein. Dieses Beispiel verwendet den SMTP-Dienst, bei der Besprechung des Codes finden Sie auch Hinweise zur Benutzung eines beliebigen SMTP-Servers.

Die folgenden Punkte beschreiben die Konfiguration des SMTP-Dienstes auf dem lokalen Rechner auf Basis der Standard-Installation.

- Zuerst muss das gute Stück natürlich installiert und lauffähig sein. Die Konfiguration können Sie auch bei beendetem Dienst durchführen. Auf jeden Fall ist es ratsam, den Dienst nach Änderungen an der Konfiguration zu beenden und neu zu starten, weil Änderungen bei laufendem Dienst oft nicht richtig übernommen werden.
- Zum Konfigurationsdialog kommen Sie über SYSTEMSTEUERUNG/VERWALTUNG/INTERNETDIENSTE-MANAGER. Im Fenster INTERNET-INFORMATIONSDIENSTE öffnen Sie den Knoten des Servers und wählen dann den Befehl EIGENSCHAFTEN im Kontextmenü des Eintrags mit dem Briefsymbol. Bei einer Standardinstallation hat der SMTP-Server den Namen *Virtueller Standardserver für SMTP*.
- Im EIGENSCHAFTEN-Dialog aktivieren Sie zuerst die Protokollierung. Dann klicken Sie auf die Schaltfläche EIGENSCHAFTEN und im Fenster ERWEITERTE PROTOKOLLIERUNGSEIGENSCHAFTEN wählen Sie die Registerkarte ERWEITERTE EIGENSCHAFTEN. Hier markieren Sie dann alle Kontrollkästchen. Es gibt intelligentere Einstellungen, aber während der Programmentwicklung ist diese Art von Rundschlag durchaus vertretbar. Außerdem bekommen Sie so am besten einen Überblick über die Möglichkeiten des Protokolls. Übernehmen Sie die Änderungen.
- Wechseln Sie zur Registerkarte ZUGRIFF und klicken Sie dann auf WEITERGABE. Klicken Sie im Fenster WEITERGABEEINSCHRÄNKUNGEN auf HINZUFÜGEN und tragen Sie die IP-Adresse *127.0.0.1* ein. Das ist die so genannte Loopback-Adresse des lokalen Rechners. Übernehmen Sie die

Änderungen. Jetzt leitet der SMTP-Dienst Ihre, und nur Ihre, Mails weiter. Diese Einschränkung ist wichtig, denn viele Spammer können ihr Unwesen nur treiben, weil sie immer wieder ungesicherte SMTP-Server zum Einschleusen ihrer Mails finden. Auch, wenn Ihr Spiel- und Bastel-Server durch eine Firewall vor Zugriffen von außen geschützt ist, sollten Sie ihn realistisch konfigurieren. Denn sonst kommt es beim Übergang von der Entwicklungs- auf die Produktionsumgebung zu Überraschungen mit Showstopper-Qualität.

- Wechseln Sie jetzt zur Registerkarte ÜBERMITTLUNG und klicken Sie dann auf ERWEITERT. Tragen Sie in die Zeile SMART HOST den Namen Ihres eigentlichen Mail-Servers ein. Diesen Namen finden Sie zum Beispiel in Outlook Express unter EXTRAS/KONTEN/EIGENSCHAFTEN (des E-Mail Kontos), Registerkarte SERVER, Zeile POSTAUSGANG (SMTP). An diesen Server leitet der SMTP-Dienst Ihre Mails weiter. Sie können statt des Namens auch die IP-Adresse in eckigen Klammern eingeben. Übernehmen Sie die Änderungen.
- Wenn der Übermittlungs-Server keinen anonymen Zugriff unterstützt, klicken Sie in der Registerkarte ÜBERMITTLUNG auf AUSGEHENDE SICHERHEIT und machen dann im Fenster AUSGEHENDE SICHERHEIT die nötigen Angaben.
- Das war's. Beenden Sie jetzt den Dienst und starten Sie ihn neu, damit die Änderungen auch richtig übernommen werden.

Und jetzt die versprochene Mail-Sende-Zeile. Hier fehlt nichts: die `Send()`-Methode ist statisch.

```
System.Web.Mail.SmtpMail.Send("Ich@Domaene1.de", "Du@Domaene2.de",
  "Der Betreff", "Der Text");
```

Codebeispiel – Mails mit Attachment

Wenn Sie zusätzliche Funktionalität brauchen und zum Beispiel ein Attachment mitsenden möchten, benutzen Sie eine andere Version der `Send()`-Methode. Diese erwartet als Parameter ein `MailMessage`-Objekt und wird im Beispielprojekt *Mail* vorgestellt.

Um das Beispielprojekt nachzuvollziehen, setzen Sie sieben `Label`-Steuerelemente, sieben Textfelder, drei Schaltflächen und ein `OpenFileDialog`-Steuerelement auf ein neues Formular und stellen dann die in Tabelle 20.4 gezeigten Eigenschaften ein.

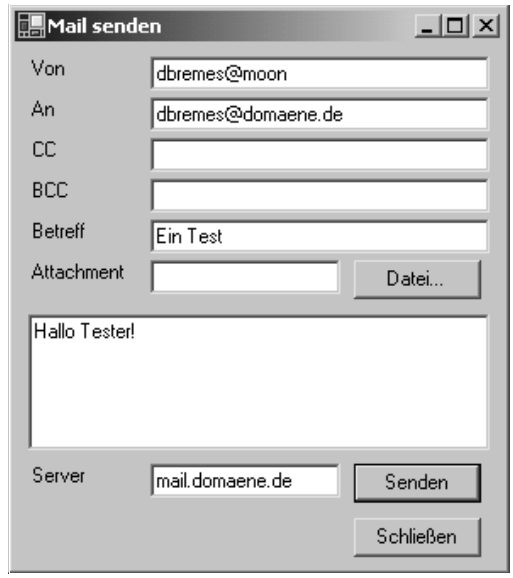


Abbildung 20.5: Mail senden

Steuerelement	Eigenschaft	Wert
Formular	Text	Mail senden
Erstes Label	Text	Von
Zweites Label	Text	An
Drittes Label	Text	CC
Viertes Label	Text	BCC
Fünftes Label	Text	Betreff
Sechstes Label	Text	Attachment
Siebtes Label	Text	Server
Erstes Textfeld	Name	txtVon
Erstes Textfeld	Text	(leer)
Zweites Textfeld	Name	txtAn
Zweites Textfeld	Text	(leer)
Drittes Textfeld	Name	txtCc
Drittes Textfeld	Text	(leer)
Viertes Textfeld	Name	txtBcc
Viertes Textfeld	Text	(leer)

Tabelle 20.4: Eigenschaftswerte

Tabelle 20.4: Eigenschaftswerte (Forts.)

Steuerelement	Eigenschaft	Wert
Fünftes Textfeld	Name	txtBetreff
Fünftes Textfeld	Text	(leer)
Sechstes Textfeld	Name	txtAttachment
Sechstes Textfeld	Text	(leer)
Siebtes Textfeld	Name	txtNachricht
Siebtes Textfeld	Text	(leer)
Erste Schaltfläche	Name	btnDatei
Erste Schaltfläche	Text	Datei
Zweite Schaltfläche	Name	btnSenden
Zweite Schaltfläche	Text	Senden
Dritte Schaltfläche	Name	btnSchließen
Dritte Schaltfläche	Text	Schließen

Richten Sie einen Verweis auf `System.Web.dll` ein und fügen Sie eine `using`-Anweisung für den Namensraum `System.Web.Mail` hinzu. Legen Sie dann folgende Behandlungsmethoden für die `Click`-Ereignisse der Schaltflächen an:

```
private void btnDatei_Click(object sender, System.EventArgs e)
{
  openFileDialog1.FileName = txtAttachment.Text;
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    txtAttachment.Text = this.openFileDialog1.FileName;
  }
}//btnDatei_Click


private void btnSchließen_Click(object sender, System.EventArgs e)
{
  Close();
}//btnSchließen_Click


private void btnSenden_Click(object sender, System.EventArgs e)
{
  try
  {
    //Mail zusammenbauen
    MailMessage mail = new MailMessage();
    mail.From = txtVon.Text;
    mail.To = txtAn.Text;
```

```
    mail.Cc = txtCc.Text;
    mail.Bcc = txtBcc.Text;
    mail.Subject = txtBetreff.Text;
    mail.Body = txtNachricht.Text;
    if (txtAttachment.Text.Length > 0)
      mail.Attachments.Add(
        new MailAttachment(txtAttachment.Text, MailEncoding.Base64));

    //Mail senden
    SmtpMail.Send(mail);
    MessageBox.Show("Mail gesendet an: " + txtAn.Text);
  }
  catch(Exception x)
  {
    MessageBox.Show(x.Message.ToString());
  }
}//btnSenden_Click
```

Neben den `From`- und `To`-Eigenschaften des Einzeiler-Beispiels setzen Sie hier also noch Kopieempfänger in den CC- (Carbon Copy) und BCC- (Blind Carbon Copy) Feldern. Außerdem kann der Benutzer ein Attachment mitsenden, das im Base64-Format kodiert wird. Diese Kodierung ist nötig, weil SMTP auch auf Systemen läuft, die Zeichen nur mit sieben Bit kodieren (vor allem UNIX-Systeme). Diese Kodierung macht den Text zwar unlesbar, stellt aber keine Verschlüsselung dar: Jeder SMTP-Client macht daraus automatisch wieder lesbaren Text.

Zusätzlich können Sie noch weitere Eigenschaften setzen, zum Beispiel die Priorität der Nachricht oder Format und Kodierung des Inhalts. Und nun zum interessanten Teil: Was funktioniert alles nicht, und wie bringt man es trotzdem zum Funktionieren?

Das entscheidende Manko ist die fehlende Möglichkeit, Benutzername und Passwort für die Anmeldung an einem SMTP-Server anzugeben. Den Server selbst können Sie in der `SmtpServer`-Eigenschaft der `SmtpMail`-Klasse festlegen. Aber das macht eben nur Sinn, wenn der Server keine Anmeldung fordert. Das Beispiel benutzt deshalb den vorhin konfigurierten lokalen SMTP-Dienst.

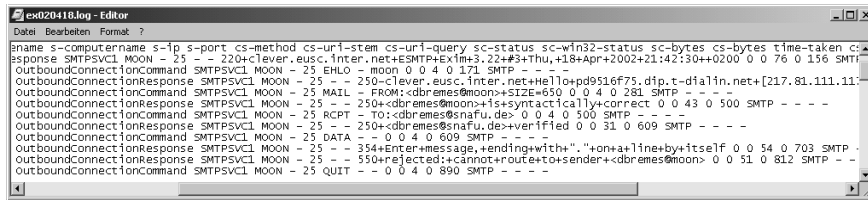
In der Vergangenheit kam es bei .NET-Release-Wechseln immer wieder zu Überraschungen mit ehemals lauffähigen Anwendungen der `SmtpMail`-Klasse. Wenn das Beispielprogramm nicht funktioniert, sollten Sie deshalb mit verschiedenen Angaben für die `SmtpServer`-Eigenschaft der `SmtpMail`-Klasse experimentieren. Gute Kandidaten sind `"localhost"`, `"127.0.0.1"`, `"smarthost"` und die IP-Adresse des Rechners.

Zur Problemlösung haben Sie neben dem gerade beschriebenen ingenieurmäßigen Vorgehen (erst mal alle Knöpfe drücken, und wenn es dann noch nicht funktioniert, kräftig dagegen treten) noch die Methode des scharfen

Hinsehens. Am besten fangen Sie mit dem Senden einer lokalen Nachricht an, also von "name@rechner" an "name@rechner". Was Sie als "name" angeben, ist egal. Wichtig ist die Angabe des Rechnernamens. Wenn der SMTP-Dienst nicht läuft, bleibt die Nachricht im Verzeichnis *Inetpub\mailroot\Pickup* liegen – und zwar ohne eine Ausnahme auszulösen! Wenn alles gut läuft, transportiert der Dienst die Nachricht über das *Queue*-Verzeichnis ins *Drop*-Verzeichnis. Damit ist seine Arbeit erledigt, die weitere Verteilung wäre Aufgabe eines POP3-Servers.

Was der SMTP-Dienst gemacht hat, protokolliert er wie vorhin konfiguriert. Die Log-Datei liegt unterhalb des *System32*-Verzeichnisses in einem Unterverzeichnis mit dem Namen des Dienstes, bei der Standard-Installation also *SMTPSVC1*.

Abbildung 20.6: Auszug aus der Log-Datei



Ein erfolgreicher Mail-Transport läuft nach folgendem Muster ab:

- Der Sender baut eine TCP-Verbindung zum Empfänger auf. Dieser sendet daraufhin eine Nachricht mit dem Code 220.
- Der Sender schickt eine HELO-Nachricht. Wenn er Erweiterungen unterstützt, sendet er eine EHLO-Nachricht. Der Empfänger antwortet darauf mit dem Code 250 und die Übertragung kann beginnen.
- Der Sender schickt eine MAIL-Nachricht mit der Absenderangabe und der Empfänger antwortet darauf mit dem Code 250.
- Der Sender schickt eine RCPT-Nachricht mit der Empfängerangabe und der Empfänger antwortet darauf mit dem Code 250. Dieser Nachrichtenaustausch wird für jeden Empfänger wiederholt.
- Der Sender schickt eine DATA-Nachricht und der Empfänger antwortet darauf mit dem Code 354.
- Der Sender schickt die Header und den Text der Mail. Der Empfänger antwortet mit dem Code 250, sobald er eine Zeile bekommen hat, die nur aus einem Punkt besteht.
- Der Sender schickt eine QUIT-Nachricht und der Empfänger antwortet darauf mit dem Code 221. Damit ist die Kommunikation beendet.

Wenn etwas schief geht, bleiben die Nachrichten entweder kommentarlos im *Queue*-Verzeichnis liegen, oder Sie erhalten die wenig aussagekräftige Fehlermeldung »Could not access 'CDO.Message' object«. Zur Fehlersuche empfiehlt sich auch hier wieder der Netzwerkmonitor. Auf Rechnern ohne Netzwerkmonitor muss man mit dem Protokoll des SMTP-Servers zurechtkommen. Abbildung 20.7 zeigt, dass die SMTP-Server durchaus hilfreiche Fehlermeldungen liefern. Schade nur, dass die CDO sie nicht weitergeben.

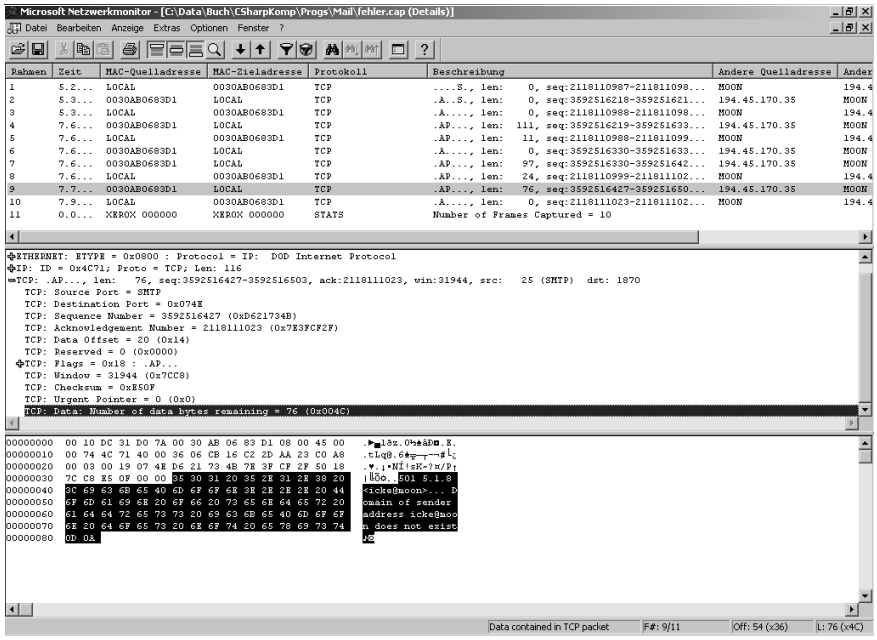


Abbildung 20.7: Mit dem Netzwerkmonitor auf Fehlersuche

Auf Rechnern mit Windows XP tritt häufig das Phänomen auf, dass die `SmtpMail`*-Klasse zwar in einem Windows- oder Konsolen-Programm funktioniert, aber nicht in ASP.NET. Abhilfe schafft das explizite Setzen der* `SmtpServer`*-Eigenschaft.*

:-) TIPP

Die Ursache liegt darin, dass die CDO versuchen, den lokalen SMTP-Server aus der IIS Meta-Datenbank (Metabase) zu ermitteln. Unter Windows 2000 und Windows .NET Server hat ASP.NET die nötigen Rechte, unter Windows XP dagegen nicht. Es ist allerdings zu erwarten, dass dieses Problem mit einem der nächsten Service Packs korrigiert wird.

20.3.5 Mails empfangen mit der TcpClient-Klasse

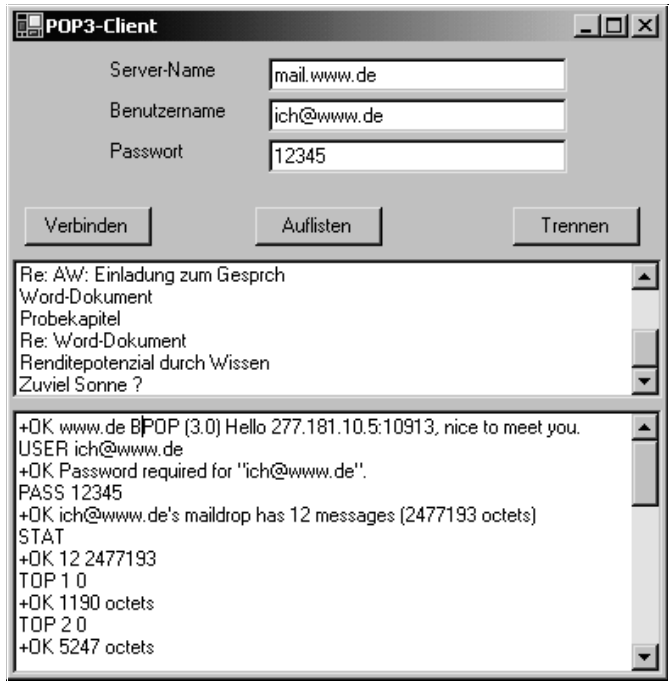
Mail-Clients benutzen POP3 (Post Office Protocol Version 3), um Mails zu empfangen. POP3 ist in RFC 1939 und anderen definiert und wie alle anderen Internetprotokolle ein einfaches, textbasiertes Frage- und Antwort-Spiel.

Codebeispiel – Mail-Emfang

Das Beispielprogramm *Pop3* stellt das automatisierte Abfragen eines Mail-Kontos durch die `TcpClient`-Klasse vor. Diese Automatisierung wird interessant, sobald auch die Auswertung der Mails automatisiert durchgeführt werden kann. Es geht also nicht um die Programmierung eines Outlook-Klons, dementsprechend ist die Funktionalität des Beispielprogramms auch stark eingeschränkt.

Das Beispielprogramm kann sich mit einem POP3-Server verbinden, die Betreffzeilen der vorhandenen Mails auflisten und die Verbindung wieder beenden.

Abbildung 20.8: Mails empfangen mit der TcpClient-Klasse



Setzen Sie ein `Panel`-Steuerelement und danach ein Textfeld, ein Listenfeld und ein Splitter-Steuerelement auf ein neues Formular. Stellen Sie die `Docking`-Eigenschaft des `Panel`-Steuerelements auf `Top`, die `Docking`-Eigenschaft des Textfelds und des Splitter-Steuerelements auf `Bottom` und die `Docking`-Eigenschaft des Listenfelds auf `Fill`. Setzen Sie noch drei `Label`-Steuerelemente, drei Textfelder und drei Schaltflächen auf das `Panel`-Steuerelement und stellen Sie dann die in Tabelle 20.5 gezeigten Eigenschaften ein.

Tabelle 20.5: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	Text	POP3-Client
Erstes Label	Text	Server-Name
Zweites Label	Text	Benutzername
Drittes Label	Text	Passwort
Erstes Textfeld	Name	txtServerName
Erstes Textfeld	Text	(leer)
Zweites Textfeld	Name	txtBenutzerName
Zweites Textfeld	Text	(leer)
Drittes Textfeld	Name	txtPasswort
Drittes Textfeld	Text	(leer)
Erste Schaltfläche	Name	btnVerbinden
Erste Schaltfläche	Text	Verbinden
Zweite Schaltfläche	Name	btnAuflisten
Zweite Schaltfläche	Text	Auflisten
Dritte Schaltfläche	Name	btnTrennen
Dritte Schaltfläche	Text	Trennen
Listenfeld	Name	lstMails
Unteres Textfeld	Name	txtResultat
Unteres Textfeld	MultiLine	True
Unteres Textfeld	ScrollBars	Vertical

Fügen Sie `using`-Anweisungen für `System.Net.Sockets`, `System.Text`, `System.IO` und `System.Text.RegularExpressions` hinzu. Legen Sie dann die folgenden globalen Variablen an:

```
private enum Kommandos {PASS, QUIT, STAT, TOP, USER};
private TcpClient m_cli = null;
private NetworkStream m_stm = null;
private StreamReader m_sr = null;
```

Die `Kommandos`-Aufzählung enthält nur einen Teil der möglichen POP3-Kommandos. Interessant wären zum Beispiel noch *NOOP*, mit dem Sie die Verbindung zum Server am Leben halten können. Oder *RETR* zum Abfragen

einer Mail, *DELE* zum Löschen einer Mail und *RSET* zum Rückgängigmachen des Löschens. Das oben verwendete Kommando *TOP* ist optional, wird aber von den meisten Servern unterstützt. Beschreibungen von POP3 finden Sie überall im Internet, beispielsweise unter `www.rfc-editor.org`.

Die Behandlungsmethode des `Click`-Ereignisses der VERBINDEN-Schaltfläche stellt eine TCP-Verbindung zum Server her und meldet den Benutzer mit Name und Passwort an. Voraussetzung ist natürlich ein bestehendes Benutzerkonto.

```
private void btnVerbinden_Click(object sender, System.EventArgs e)
{
  try
  {
    txtResultat.Text = "";
    //Port Nummer für POP3-Protokoll ist 110
    m_cli = new TcpClient(txtServerName.Text, 110);
    m_stm = m_cli.GetStream();
    m_sr = new StreamReader(m_stm);
    //Server sagt schon vor dem 1. Kommando "Hallo"
    string s = m_sr.ReadLine();
    ZeigeStatus(s);
    //Anmelden
    FührеKommandoAus(m_stm, m_sr, Kommandos.USER, txtBenutzerName.Text);
    FührеKommandoAus(m_stm, m_sr, Kommandos.PASS, txtPasswort.Text);
  }
  catch (Pop3Exception x)
  {
    MessageBox.Show(x.Message);
  }
}//btnVerbinden_Click
```

Die Methode `ZeigeStatus()` protokolliert die Kommunikation zwischen Server und Client im unteren Textfeld. `FührеKommandoAus()` sendet das übergebene Kommando an den Server und prüft dessen Antwort. Hier ist der Code sowie die nötigen Hilfsmethoden:

```
private string FührеKommandoAus(
    NetworkStream stm, StreamReader sr, Kommandos Kommando, string Text)
{
  Sende(stm, Kommando, Text);
  return LeseAntwort(sr);
}//FührеKommandoAus


private string LeseAntwort(StreamReader sr)
{
  string s = sr.ReadLine();
  PrüfeAntwort(s);
  ZeigeStatus(s);
  return s;
}//LeseAntwort
```

```
private void PrüfeAntwort(string Antwort)
{
  if ((Antwort == null) || (Antwort == ""))
    throw new Pop3KeineAntwortException();
  if (!Antwort.StartsWith("+OK"))
    throw new Pop3ErrException(Antwort);
}//PrüfeAntwort


private void Sende(NetworkStream stm, Kommandos Kommando, string Text)
{
  string Zeile;
  byte[] Bytes;

  Zeile = Kommando.ToString() + " " + Text;
  ZeigeStatus(Zeile);
  Zeile += "\r\n";
  Bytes = Encoding.ASCII.GetBytes(Zeile.ToCharArray());
  stm.Write(Bytes, 0, Bytes.Length);
}//Sende


private void ZeigeStatus(string NeuerStatus)
{
  txtResultat.Text += NeuerStatus + "\r\n";
}//ZeigeStatus
```

Interessant ist beim oben vorgestellten Code vor allem die Methode `PrüfeAntwort()`. Zum einen kann `sr.ReadLine()` in `LeseAntwort()` auch `null` zurückgeben, und zwar wenn der Server die Verbindung vorher beendet hat. Zum anderen führt beispielsweise eine falsche Kommandoreihenfolge zu Fehlermeldungen des Servers. Diese Fehlermeldungen beginnen mit dem Text `"-ERR"`. Die Methode `PrüfeAntwort()` prüft beide Möglichkeiten und löst im Fall der Fälle eine Ausnahme aus. Dadurch werden beide Fehlermeldemechanismen in die .NET-Ausnahmebehandlung integriert.

Neben `PrüfeAntwort()` berücksichtigen auch die Behandlungsmethode für die `Click`-Ereignisse der Schaltflächen nur Ausnahmen, die von `Pop3Exception` abgeleitet sind.

```
public abstract class Pop3Exception : ApplicationException
{
  //Basisklasse für Programm-eigene Exception-Klassen
  public Pop3Exception(string FehlerMeldung)
    : base(FehlerMeldung)
  {
  }//ctor
}//class Pop3Exception

public class Pop3ErrException : Pop3Exception
{

  public Pop3ErrException(string Antwort)
```

```
    : base(Antwort.Substring(4)) //-ERR abschneiden
  {
  }//ctor
}//class Pop3ErrException

public class Pop3KeineAntwortException : Pop3Exception
{

  public Pop3KeineAntwortException()
    : base("Keine Antwort vom Server bekommen")
  {
  }//ctor
}//class Pop3KeineAntwortException
```

`Pop3Exception` wird also von `ApplicationException` abgeleitet, und das ist auch der einzige Lebenszweck der Klasse `ApplicationException`. Microsoft stellt sich eine zweigeteilte Ausnahmenhierarchie vor: Alle CLR-definierten Ausnahmen sind von `SystemException` abgeleitet, alle programmdefinierten von `ApplicationException`. Allerdings hält sich Microsoft selbst nicht an diesen Standard: Zum einen sind einige `Reflection`-Ausnahmen von `ApplicationException` abgeleitet, zum anderen ist beispielsweise `IsolatedStorageException` ein direkter Nachfahre von `Exception`.

Außerdem dürfte sich die Ausnahmebehandlung eher an der Ursache der Ausnahme orientieren als daran, ob sie in einer Methode des Systems oder des Programms aufgetreten ist. Ein `catch`-Filter fragt also zum Beispiel auf `MemberAccessException` ab, aber nicht auf `ApplicationException`. Es ist auch nicht so, dass von `SystemException` abgeleitete Ausnahmen generell schwerwiegendere Ursachen haben. Eine eigene Ausnahmenhierarchie auf `ApplicationException` aufzubauen, ist also ebenso arbeitsintensiv wie sinnfrei. Stattdessen wird man wie hier die bestehende Hierarchie nur erweitern, um Funktionalität hinzuzufügen.

Zurück zum Code. Von der abstrakten Klasse `Pop3Exception` werden die beiden tatsächlich auszulösenden Ausnahmen abgeleitet: `Pop3ErrException` und `Pop3KeineAntwortException`. `Pop3ErrException` schneidet von der im Konstruktor übergebenen Fehlermeldung des Servers das `"-ERR"` ab, `Pop3KeineAntwortException` denkt sich dort einen eigenen Text aus. Alle drei Konstruktoren rufen dann ebenso wie der Konstruktor von `Pop3Exception` einfach den Konstruktor der jeweiligen Basisklasse auf.

Die Behandlungsmethode des `Click`-Ereignisses der AUFLISTEN-Schaltfläche trägt die Betreff-Zeilen der vorhandenen Mails in das Listenfeld ein.

```
private void btnAuflisten_Click(object sender, System.EventArgs e)
{
  string s = "";
  Match m = null;
```

```
  try
  {
    lstMails.Items.Clear();
    //Anzahl der vorhandenen Nachrichten ermitteln
    s = FührеKommandoAus(m_stm, m_sr, Kommandos.STAT, "");
    m = Regex.Match(s, @"\+OK (\d+) ");
    int AnzahlMails = int.Parse(m.Groups[1].Value);
    //Betreff-Zeilen der Mails auflisten
    for (int i = 1; i <= AnzahlMails; i++)
    {
      s = LeseMail(m_stm, m_sr, i);
      m = Regex.Match(s, "^Subject: (.*)", RegexOptions.Multiline);
      s = m.Groups[1].Value;
      s = s.Remove(s.Length - 1, 1);//CR abschneiden
      lstMails.Items.Add(s);
    }
  }
  catch (Pop3Exception x)
  {
    MessageBox.Show(x.Message);
  }
}//btnAuflisten_Click
```

Hier wird zuerst die Anzahl der vorhandenen Nachrichten durch das *STAT*-Kommando ermittelt. Der Server antwortet darauf mit einer Zeile wie `"+OK 12 12345"`. Darin liefert der erste Ziffernblock die Anzahl der Mails und der zweite deren Größe in Bytes. Hier interessiert nur die Anzahl der Mails, und die ermittelt der Reguläre Ausdruck. Den könnte man vereinfachen, denn das Leerzeichen hinter dem Ziffernblock reicht bereits als Begrenzung. Aber so ist auf den ersten Blick klar, was gesucht wird: der erste, und nicht der zweite Zahlenblock.

Die Schleife listet dann die Mails auf, älteste zuerst. Dabei übernimmt ein weiterer Regulärer Ausdruck die Suche nach dem Inhalt der Betreffzeile. Das Zeichen `"^"` am Anfang des Ausdrucks beschränkt die Suche nach dem Text `"Subject"` auf den Zeilenanfang, das gleiche Wort an anderer Stelle wird also nicht gefunden. Wichtig ist hier, die Option `RegexOptions.Multiline` zu setzen, denn sonst beschränkt das Zeichen `"^"` die Suche auf den Anfang des Strings.

Hier ist der Code für das Lesen der Mails. Da ohnehin nur die Header ausgewertet werden, holt die Methode `LeseMail()` den Rumpf der Nachricht gar nicht ab.

```
private string LeseMail(
  NetworkStream stm, StreamReader sr, int NachrichtenNummer)
{
  //0 holt nur die Header, -1 die ganze Mail,
  //sonst alle Header und entsprechende Zeilenzahl
  Sende(stm, Kommandos.TOP, NachrichtenNummer + " 0");
  return LeseMehrzeiligeAntwort(sr);
}//LeseMail
```

```
private string LeseMehrzeiligeAntwort(StreamReader sr)
{
  LeseAntwort(sr); //Erste Zeile ist etwas wie "+OK 1234 Octets"
  StringBuilder sb = new StringBuilder();
  string s = sr.ReadLine();
  while (!s.StartsWith("."))
  {
    sb.Append(s + "\r\n");
    s = sr.ReadLine();
  }
  return sb.ToString();
}//LeseMehrzeiligeAntwort
```

Vor der Methode `LeseMehrzeiligeAntwort()` hat sich der Code nur mit dem Lesen einzeiliger Antworten befasst, dazu reichte ein `ReadLine()` des `StreamReader`-Objekts. Bei mehrzeiligen Antworten hat nur die erste Zeile das gewohnte OK-Format. Danach kommen dann die eigentlichen Daten in beliebig vielen Zeilen. Den Schluss bildet eine Zeile, die nur einen Punkt enthält. `LeseMehrzeiligeAntwort()` bastelt daraus wieder eine Mail zusammen.

Zu guter Letzt noch der Code für das Trennen der Verbindung:

```
private void btnTrennen_Click(object sender, System.EventArgs e)
{
  try
  {
    FührеKommandoAus(m_stm, m_sr, Kommandos.QUIT, "");
    GebeRessourcenFrei();
  }
  catch (Pop3Exception x)
  {
    MessageBox.Show(x.Message);
  }
}//btnTrennen_Click


private void GebeRessourcenFrei()
{
  if (m_cli != null)
  {
    m_cli.Close();
    m_cli = null;
  }
  if (m_sr != null)
  {
    m_sr.Close();
    m_sr = null;
  }
  if (m_stm != null)
  {
    m_stm.Close();
    m_stm = null;
  }
}//GebeRessourcenFrei
```

Der Code arbeitet mit globalen Variablen für die `NetworkStream`- und die `StreamReader`-Instanzen, weil der Stream die TCP-Verbindung repräsentiert und bis zum expliziten Trennen offen bleiben muss.

Wie eingangs erwähnt, zeigt das Beispiel nur die automatisierte Abfrage eines Mail-Kontos. Deswegen fehlt neben einer vollständigen Implementation der POP3-Kommandos auch das Konvertieren des angezeigten Betrefftextes. Je nach Format der Mail gehen deshalb Umlaute und andere Sonderzeichen verloren oder werden durch andere Zeichen ersetzt. Dasselbe würde diesen Zeichen auch im Rumpf der Mail zustoßen.

20.3.6 Ein Screen Scraper mit den Klassen WebClient und Regex

Ein Screen Scraper, zu Deutsch Bildschirmkratzer, kratzt nicht etwa erschlagene Fliegen oder den Nikotinfilm vom Bildschirm, sondern die Buchstaben von seiner Rückseite. Solche Klimmzüge muss man machen, wenn man sonst nicht an die Daten kommt.

Die Ursprünge des Bildschirmkratzens liegen im Zeitalter der textbasierten Terminals. Heute, im Internet-Zeitalter, kratzt man die Buchstaben von den Webseiten. Das ist nötig, wenn man die Daten maschinell weiterverarbeiten möchte und der Server nur Webseiten und keine Web Services anbietet.

Codebeispiel – Automatisierte Auswertung von Webseiten

Die Aufgabe des Beispielprojekts *ScreenScraper* gliedert sich in zwei Teile: das Sammeln der Daten und ihre Auswertung. Das folgende Beispielprogramm sammelt Daten von den Webseiten zweier Wintersportorte, sucht die Schneehöhen heraus und zeigt sie in einem Textfeld. Im Sommer können Sie stattdessen die Windstärken an zwei Segel- bzw. Surfrevieren abfragen. Da Sie zum Sammeln keine Oberfläche brauchen, programmieren Sie nicht etwa den IE, sondern benutzen die `WebClient`-Klasse, die genau für solche Aufgaben gedacht ist. Die Auswertung erledigt die `Regex`-Klasse, die ebenfalls genau dafür eingerichtet ist.

Um das Beispiel nachzustellen, setzen Sie ein `Panel`-Steuerelement und danach ein Textfeld auf ein neues Formular. Stellen Sie die `Docking`-Eigenschaft des `Panel`-Steuerelements auf `Top` und die `Docking`-Eigenschaft des Textfelds auf `Fill`. Setzen Sie noch eine Schaltfläche und zwei Optionsschaltflächen auf das `Panel`-Steuerelement und stellen Sie dann die in Tabelle 20.6 gezeigten Eigenschaften ein.

Abbildung 20.9: Der Screen Scraper in Aktion

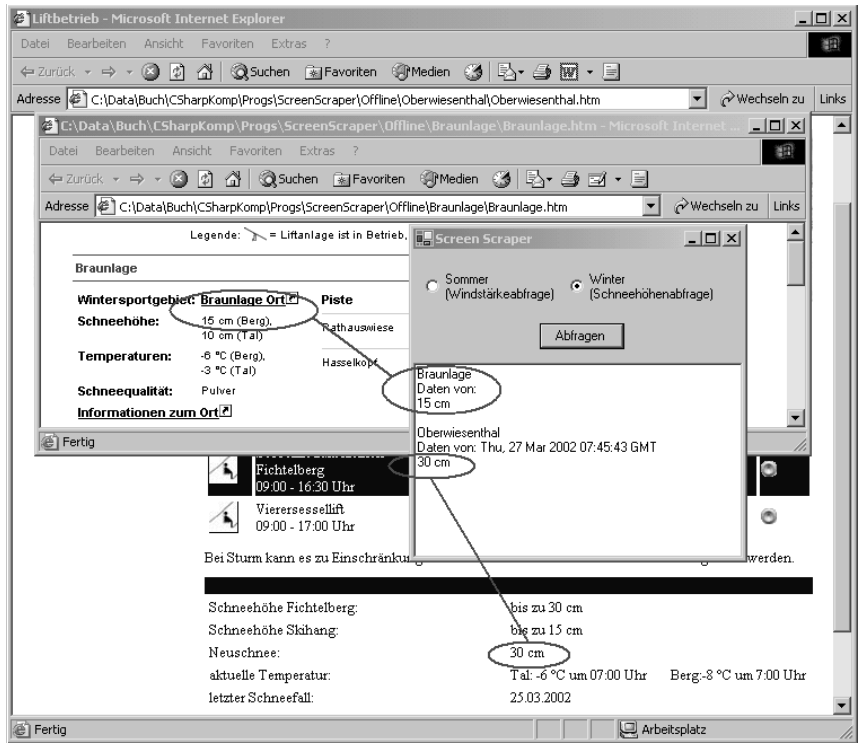


Tabelle 20.6: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	`Text`	Screen Scraper
Formular	`AcceptButton`	btnAbfragen
Schaltfläche	`Name`	btnAbfragen
Schaltfläche	`Text`	Abfragen
Linke Optionsschaltfläche	`Name`	optSommer
Linke Optionsschaltfläche	`Text`	Sommer (Windstärkeabfrage)
Linke Optionsschaltfläche	`Name`	optWinter
Linke Optionsschaltfläche	`Text`	Winter (Schneehöhenabfrage)
Textfeld	`Name`	txtResultat
Textfeld	`Text`	(leer)
Textfeld	`MultiLine`	True
Textfeld	`ScrollBars`	Vertical

Fügen Sie `using`-Anweisungen für die Namensbereiche `System.Net`, `System.Text` und `System.Text.RegularExpressions` hinzu. Tragen Sie dann den folgenden Code in die Methode für das `Click`-Ereignis der Schaltfläche ein.

```
private void btnAbfragen_Click(object sender, System.EventArgs e)
{
  txtResultat.Text = "";
  Cursor.Current = Cursors.WaitCursor;
  try
  {
    if (optSommer.Checked)
    {
      txtResultat.Text += Abfrage("Sylt",
        "http://www.sylt-wind.de/index.htm",
        @" Uhr:\s*(\d+\.\d+ kt)", "Windstärke") + "\r\n\r\n";
      txtResultat.Text += Abfrage("St. Peter-Ording",
        "http://www.st.peter-ording.de/wetter_spo.htm",
        @"Windst&auml;rke.*>(.*\d+ BF)", "Windstärke") + "\r\n\r\n";
    }
    else
    {
      txtResultat.Text += Abfrage("Braunlage",
        "http://www.braunlage.de/schneenews/Wintersportgebiete/inhalt.asp",
          @"Braunlage Ort.*Schneeh&ouml;he.*>(\d+ cm) \(Berg\).*braunlage",
          "Schneehöhe") + "\r\n\r\n";
      txtResultat.Text += Abfrage("Oberwiesenthal",
        "http://www.oberwiesenthal.de/« +
        «Willkommen/Tagesinfo/Liftbetrieb/liftbetrieb.htm",
        @"Schneehöhe Skihang.*>(.*\d+ cm).*Neuschnee", "Schneehöhe") +
        "\r\n\r\n";
    }
  }
  finally
  {
    Cursor.Current = Cursors.Default;
  }
}//btnAbfragen_Click
```

Und jetzt noch die Methode `Abfragen()`:

```
private string Abfragen(
  string Name, string WebAdresse, string RegEx, string TextZumWert)
{
  string s = Name + "\r\n";
  WebClient wc = new WebClient();
  try
  {
    Byte[] Bytes = wc.DownloadData(WebAdresse);
    string Html = Encoding.UTF8.GetString(Bytes);
    Match m = Regex.Match(Html, @RegEx, RegexOptions.Singleline);
    s += "Daten von: " + wc.ResponseHeaders["Last-Modified"] + "\r\n";
    s += TextZumWert + ": " + m.Result("$1");
  }
```

```
  catch (Exception x)
  {
    MessageBox.Show(x.Message);
  }
  return s;
}//Abfragen
```

So weit, so gut. Die Logik in der Behandlungsmethode für das `Click`-Ereignis der Schaltfläche dürfte sich selbst erklären. Interessanter ist die Methode `Abfragen()` – vor allem ihr vorletzter Parameter sieht recht bedrohlich aus. Der Rest des Abschnitts konzentriert sich auf die Winterversion mit der Schneehöhenabfrage, die sommerliche Abfrage der Windstärken funktioniert aber genauso.

Ein Blick auf die Methode `Abfragen()`. Der erste Parameter dient nur der späteren Zuordnung der Daten. Im Beispiel wurden Ortsnamen eingegeben, in größeren Skigebieten könnte das auch der Name eines Bergs oder Teilgebiets sein.

Das Herunterladen der Seite übernimmt die `WebClient`-Klasse. Dazu bekommt ihre `DownloadData()`-Methode die Adresse der herunterzuladenden Webseite übergeben. Wichtig ist, dabei auch das Protokoll anzugeben, also `"http:"`. Außerdem steht in der Adressleiste des IE nicht immer die richtige Adresse, wenn die Daten in Frames dargestellt werden. Wenn Sie nicht die erwarteten Daten bekommen, klicken Sie mit der rechten Maustaste auf die auszuwertenden Daten und wählen im Kontextmenü den Befehl EIGENSCHAFTEN. Im EIGENSCHAFTEN-Fenster sehen Sie dann die anzusprechende Adresse.

Die `DownloadData()`-Methode gibt ein `byte`-Array zurück, das die `Encoding`-Klasse in einen String konvertiert. Diese Konvertierung ist nötig, weil die `Regex`-Klasse nicht mit `byte`-Arrays oder Streams umgehen kann. Die erste Version des Programms benutzte zum Konvertieren die ASCII-Kodierung. Statt `"Schneehöhe"` erschien aber `"SchneehC6he"`, und ein Blick in die Header der Seite zeigte, dass sie in UTF-8 kodiert war. Da die Auswertung einer ASCII-kodierten Seite mit UTF-8 keine Nachteile bringt, erfolgt sie jetzt allgemein in diesem Format.

Weiter geht es mit der Auswertung der heruntergeladenen Daten und damit dem Einsatz der `Regex`-Klasse und des dritten Parameters. Hier wird ein so genannter REGULÄRER AUSDRUCK (Regular Expression = Regex) übergeben, der den Wert für die Schneehöhe am Skihang auslesen soll. Wie im Screenshot zu sehen, liegt dieser Wert zwischen den Texten `"Schneehöhe Skihang"` und `"Neuschnee"`. Dazwischen suchen Sie Daten, die nach einem Tag-Ende (`">"`) mit einem beliebigen Text beginnen, hier `"bis zu"`. Darauf folgen eine oder mehrere Ziffern, ein Leerzeichen und der Text `"cm"`. Der Reguläre Ausdruck dazu sieht so aus:

```
"Schneehöhe Skihang.*>(.*\d+ cm).*Neuschnee"
```

Details zum Aufbau Regulärer Ausdrücke finden Sie in der Online-Dokumentation, Thema »Reguläre Ausdrücke von .NET Framework«.

Die `Match()`-Methode der `Regex`-Klasse liefert den bzw. die gesuchten Textteile. Als Parameter erwartet sie den zu durchsuchenden Text, einen Regulären Ausdruck und gegebenenfalls zusätzliche Optionen. Um im Regulären Ausdruck einfache Rückstriche verwenden zu können, setzen Sie ihm ein `"@"` voran. Das ist eine sehr angenehme Möglichkeit der Sprache C#, die auch schon bei der Parameterübergabe genutzt wurde. Die Option `RegexOptions.Singleline` ist wichtig: Ohne sie würde das `"."` im Regulären Ausdruck als »jedes Zeichen außer Zeilenvorschüben« interpretiert. Und Zeilenvorschübe enthält das zu untersuchende HTML mit Sicherheit.

Statt der gefundenen Daten schreiben Sie aber zuerst das Datum der letzten Aktualisierung in den Rückgabe-String. Das lässt sich leicht aus den Header-Einträgen der Seite auslesen. Der entsprechende Eintrag ist aber nicht immer vorhanden, wie die Abfrage für Braunlage zeigt.

Zum Schluss kommt dann endlich die Schneehöhe in den Rückgabe-String. Das erledigt die `Result()`-Methode des `Match`-Objekts, die als Parameter den Namen der Gruppe erwartet. Da der Reguläre Ausdruck nur eine Gruppe enthält, muss der Wert *$1* sein. Alternativ könnte man die Gruppe auch mit `Groups[1].Value` erhalten. Und wie immer bei der Internet-Programmierung, darf die Fehlerbehandlung nicht vergessen werden.

Das Beispiel können Sie beliebig ausbauen und beispielsweise die aktuelle Temperatur auslesen. Ganz ordentliche Menschen implementieren statt der Methode mit String-Rückgabe eine Klasse mit entsprechenden Eigenschaften.

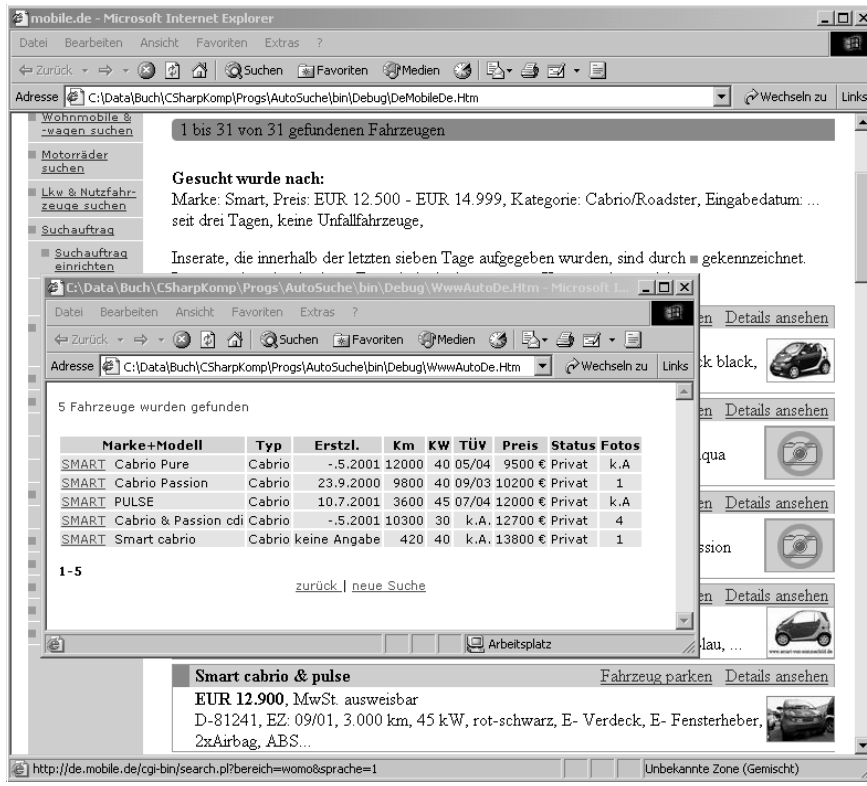
Im nächsten Abschnitt finden Sie noch eine Erweiterung dieses Lösungsansatzes: die Übergabe von Parametern an die abzufragende Webseite.

20.3.7 Eine automatisierte Auto-Suche mit der WebClient-Klasse

Die automatisierte Auto-Suche ist auch ein Screen Scraper wie im vorherigen Abschnitt. In diesem Fall sucht sie aber nicht brusttiefen Pulverschnee, sondern ein Auto(chen): ein Smart Cabrio.

Die Webseiten, die Sie für die Autosuche abfragen, benötigen deshalb zusätzlich Angaben über den gewünschten Autotyp, das maximale Alter, den maximalen Preis etc. Diese Angaben übergeben Sie bei Aufruf der Webseite als Parameter. Das schränkt das Ergebnis soweit ein, dass das Programm sich eine Auswertung des Resultats spart und die komplette Seite als Datei ablegt.

Abbildung 20.10: Die heruntergeladenen Resultate



Codebeispiel – Automatisierte Suche im Web

Da im Beispielprogramm *AutoSuche* alles hartkodiert ist und die Anzeige per Internet Explorer erfolgt, brauchen Sie nur ein Formular mit einer Schaltfläche. Stellen Sie die in Tabelle 20.7 gezeigten Eigenschaften ein.

Tabelle 20.7: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	`Text`	Auto-Suche
Formular	`AcceptButton`	btnAbfragen
Schaltfläche	`Name`	btnAbfragen
Schaltfläche	`Text`	Abfragen

Fügen Sie `using`-Anweisungen für `System.Collections.Specialized`, `System.Net`, `System.Text` und `System.IO` hinzu. Tragen Sie dann den folgenden Code in die Methode für das `Click`-Ereignis der Schaltfläche ein.

```
private void btnAbfrage_Click(object sender, System.EventArgs e)
{
  Cursor.Current = Cursors.WaitCursor;
  NameValueCollection Parameter = new NameValueCollection();
  try
  {
    //Suche auf www.auto.de mit HTTP GET
    Parameter.Add("Modelle", "1060");
    Parameter.Add("Marke", "0");
    Parameter.Add("Preisv", "0");
    Parameter.Add("Preisb", "0");
    Parameter.Add("MwSt", "0");
    Parameter.Add("Farbe", "0");
    Parameter.Add("Typ", "1");
    Parameter.Add("Kategorie", "0");
    Parameter.Add("PLZBereich", "-1");
    Parameter.Add("Erstzulassung", "0");
    Parameter.Add("KmStand", "20000");
    Parameter.Add("PSLeistung", "0");
    Parameter.Add("Sortierungnach", "AutodePreis");
    Parameter.Add("suche", "Suche+starten");
    Parameter.Add("AModell", "");
    Parameter.Add("MOVE", "1");
    Parameter.Add("FahrzeugID", "1");
    Parameter.Add("Fahrzeug", "PKW");
    AutoMitHttpGetSuchen(Parameter,
      "http://www.automail.de/SucheErgebnis.asp",
      Application.StartupPath + "\\WwwAutoDe.Htm");
    //Suche auf de.mobile.de mit HTTP POST
    Parameter.Clear();
    Parameter.Add("bereich", "pkw");
    Parameter.Add("sprache", "1");
    Parameter.Add("DoSearch", "1");
    Parameter.Add("FormMake", "214");
    Parameter.Add("FormModel", "");
    Parameter.Add("FormPrice", "125000000-149999999");
    Parameter.Add("FormEZ", "-");
    Parameter.Add("FormKilometer", "-");
    Parameter.Add("FormPower", "-");
    Parameter.Add("FormCategory", "13");
    Parameter.Add("FormColor", "...beliebig");
    Parameter.Add("FormLand", ".");
    Parameter.Add("FormPLZ", "");
    Parameter.Add("FormDurchmesser", "0");
    Parameter.Add("FormDate", "3");
    Parameter.Add("FormSort", "0");
    Parameter.Add("x", "37");
    Parameter.Add("y", "7");
    AutoMitHttpPostSuchen(Parameter,
      "http://de.mobile.de/cgi-bin/search.pl",
      Application.StartupPath + "\\DeMobileDe.Htm");
  }
```

```
  finally
  {
    Cursor.Current = Cursors.Default;
  }
}//btnAbfragen_Click
```



Der Zeichensatz für die Werte der Parameter ist eingeschränkt. Beispielsweise müssen Leerzeichen durch `"+"` *ersetzt werden. Diese Konvertierung übernimmt die Methode* `UrlEncode()` *der Klasse* `HttpUtility`*.*

Und das ist der Code für die Funktionen `AutoMitHttpGetSuchen()` und `AutoMitHttpPostSuchen()`:

```
private void AutoMitHttpGetSuchen(
  NameValueCollection Parameter, string WebAdresse, string DateiName)
{
  WebClient wc = new WebClient();
  wc.QueryString = Parameter;
  try
  {
    wc.DownloadFile(WebAdresse, DateiName);
    Process.Start(DateiName);
  }
  catch (Exception x)
  {
    MessageBox.Show(x.Message);
  }
}//AutoMitHttpGetSuchen


private void AutoMitHttpPostSuchen(
  NameValueCollection Parameter, string WebAdresse, string DateiName)
{
  WebClient wc = new WebClient();
  FileStream fs = null;
  try
  {
    Byte[] Bytes = wc.UploadValues(WebAdresse, "POST", Parameter);
    fs = new FileStream(DateiName, FileMode.Create);
    fs.Write(Bytes, 0, Bytes.Length);
    Process.Start(DateiName);
  }
  catch (Exception x)
  {
    MessageBox.Show(x.Message);
  }
  finally
  {
    if (fs != null)
      fs.Close();
  }
}//AutoMitHttpPostSuchen
```

Die beiden Webseiten funktionieren unterschiedlich: die eine erwartet ein HTTP GET, die andere ein HTTP POST. Beim HTTP GET werden die Parameter an die URL gehängt, die dann die Form `"http://www.automail.de/SucheErgebnis.asp?Modelle=1060&Marke=0"` hat. Die Länge des Strings ist begrenzt, deshalb muss man für größere Mengen von Parametern ein HTTP POST benutzen. Das HTTP POST transportiert die Parameter im Rumpf: Sie sind also nicht in der Adresszeile des IE sichtbar und laden deshalb den Benutzer nicht zu Experimenten ein.

In beiden Fällen werden die Parameter als `NameValueCollection` übergeben. Das Problem ist natürlich, alle Parameter und ihre Werte richtig zu raten – nein, das war nur ein Scherz: Der Netzwerkmonitor liefert alles im Klartext. Nach dem Starten des Netzwerkmonitors füllen Sie einmal die Webseite aus und wählen dann im Netzwerkmonitor-Menü SAMMELN den Befehl STARTEN. Dann schicken Sie Ihre Abfrage ab und wählen schließlich im Netzwerkmonitor-Menü SAMMELN den Befehl BEENDEN UND ANZEIGEN.

Beim HTTP GET stehen die interessanten Daten in der Zeile mit dem ersten `"GET"`. Ein Doppelklick darauf zeigt alle Parameter im untersten Teil des Fensters (vgl. Abbildung 20.11).

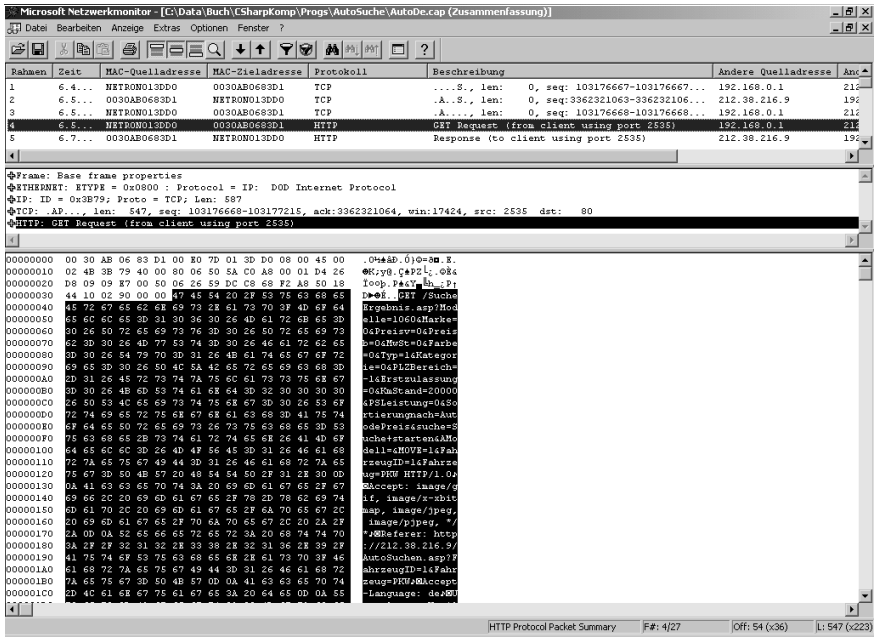


Abbildung 20.11: Beim HTTP GET werden die Parameter an die URL angehängt.

Das HTTP POST wird auf mehrere Frames verteilt, weshalb Sie bei dieser Technik nicht nur in der Zeile mit dem ersten `"POST"`, sondern auch im darauf folgenden `"Request"` nachsehen müssen (vgl. Abbildung 20.12).

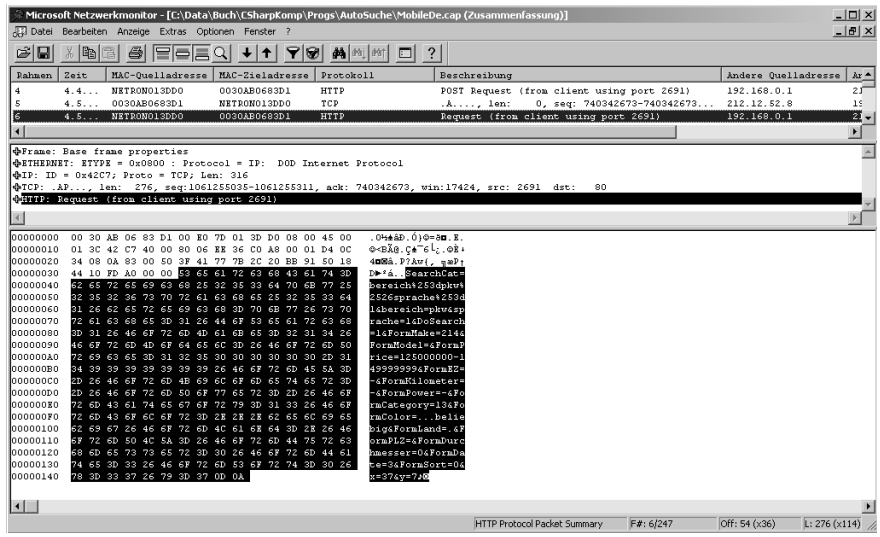


Abbildung 20.12: Beim HTTP POST werden die Parameter im Rumpf der Nachricht transportiert.

Der Netzwerkmonitor ist zwar in vielen Windows-Versionen verfügbar, gehört aber nicht zum Umfang der Standard-Installation. Wenn bei Ihnen die Suche über SYSTEMSTEUERUNG/SOFTWARE/WINDOWS-KOMPONENTEN erfolglos bleiben sollte – es gibt eine größere Zahl ähnlicher Programme im Internet.

Aber zurück zum Code. Wenn Sie schon den Abschnitt über das Beispielprojekt *ScreenScraper* gelesen haben, ist die Funktion `AutoMitHttpGetSuchen()` nichts Neues. Lediglich das Setzen der `QueryString`-Eigenschaft und das Herunterladen in eine Datei mit der `DownloadFile()`-Methode sind hinzugekommen.

In der `AutoMitHttpPostSuchen()`-Methode kommt ein bisschen Code zum Speichern der Datei hinzu. Das ist nötig, weil die `DownloadFile()`-Methode nur mit HTTP GET arbeitet.

20.3.8 Eine asynchrone Auto-Suche mit den Klassen WebClient und der WebRequest

Dieser Abschnitt stellt eine weitere Implementation der Auto-Suche vor, die im Gegensatz zu ihrem Vorgänger asynchron arbeitet. Das bedeutet zum einen, dass sie die beiden Webseiten nicht nacheinander abfragt, sondern parallel. Und es bedeutet zum anderen, dass der Benutzer während der Suche weiterarbeiten kann. Zugegeben, das Programm bietet keine Funktionalität, die man zum Weiterarbeiten nutzen könnte. Aber das ist eine andere Geschichte.

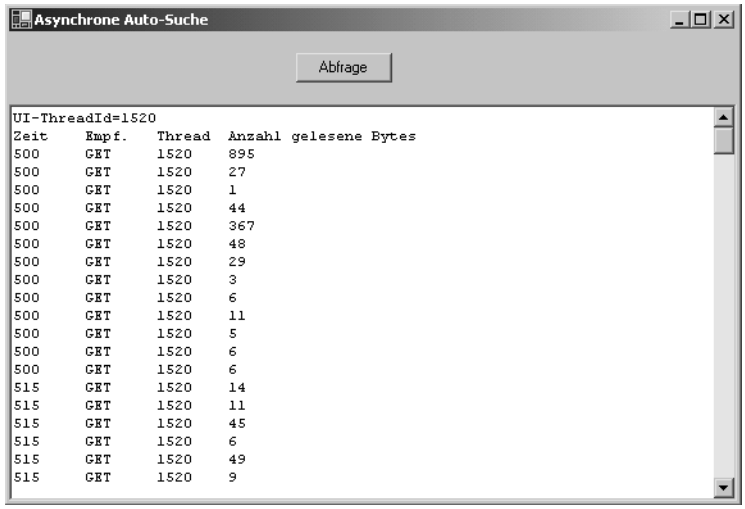


Abbildung 20.13: Asynchrone Auto-Suche

Entscheidend ist, dass jetzt mehrere Threads die Arbeit erledigen. Neben dem Haupt-Thread, auf dem die Oberfläche läuft, existiert für jede Suche ein eigener Thread und eventuell auch mehrere. Diese Threads erzeugt und verwaltet .NET selbst. Sie brauchen dabei nichts zu tun, als einige Vorsichtsmaßnahmen einzuhalten. Mit Critical Sections, Mutexen und Semaphoren brauchen Sie sich also nicht herumzuschlagen (vgl. hierzu Teil 5).

Was ändert sich nun genau gegenüber dem ursprünglichen Beispielprogramm? In den neuen Methoden `AutoMitAsyncHttpGetSuchen()` und `AutoMitAsyncHttpPostSuchen()` wird die jeweilige Webseite nicht mehr vollständig heruntergeladen. Stattdessen wird das Herunterladen nur angestoßen, und sobald der erste Teil der Antwort ankommt, geben die Methoden die Kontrolle an den Aufrufer zurück. Beim Anstoßen wird eine Rückrufmethode übergeben, die der jeweilige Such-Thread bei jedem Eintreffen einer Teilantwort aufruft. Diese Methode schreibt beim Eintreffen der letzten Teilantwort alles in eine Datei.

Dabei werden der Dateiname und die bisher erhaltenen Teilantworten nicht etwa in globalen Variablen gespeichert. Stattdessen stecken sie in einem speziellen Zustands-Objekt, das wie die Rückrufmethode beim Anstoßen des Herunterladens als Parameter übergeben wurde.

Die Rückrufmethoden sind auch für das Schreiben der Einträge in das Textfeld verantwortlich. Dort erscheinen die vergangene Zeit in Millisekunden, GET oder POST, die ThreadId und die Anzahl der gelesenen Bytes. Nur hier, wo Sie von anderen Threads auf den Oberflächen-Thread zugreifen, ist eine kleine Vorsichtsmaßnahme (Stichwort: Synchronisierung) in Gestalt eines `Delegate`-Aufrufs nötig.

Während der Suche ist die Schaltfläche deaktiviert, da eine doppelte Abfrage derselben Daten wenig Sinn macht und das Programm aus dem Tritt bringt. Um die Schaltfläche wieder zu aktivieren, rufen die Methoden `AutoMitAsyncHttpGetSuchen()` und `AutoMitAsyncHttpPostSuchen()` am Ende der Suche die Methode `AutoSucheBeendet()` auf. Da sich dabei theoretisch zwei Threads in die Quere kommen können, ist auch diese Methode Thread-sicher implementiert.

Um den Benutzer auch akustisch über das Ende der Suche zu informieren, gibt die Methode `AutoSucheBeendet()` einen Piepton aus. Erstaunlicherweise gibt es in .NET keine Funktion wie `MessageBeep()` unter Windows. Man muss diese Funktion also per `DllImport` verfügbar machen.

Und schließlich wollen auch hier unverwaltete Ressourcen freigegeben werden. Neben der schon bekannten Implementierung im `finally`-Block sehen Sie hier noch eine weitere Klasse mit der Basis `IDisposable`.

Codebeispiel – Asynchrones Suchen mit WebClient und WebRequest

Um das Beispielprojekt *AutoSucheAsync* nachzustellen, setzen Sie ein `Panel`-Steuerelement und danach ein Textfeld auf ein neues Formular. Stellen Sie die `Docking`-Eigenschaft des `Panel`-Steuerelements auf `Top` und die `Docking`-Eigenschaft des Textfelds auf `Fill`. Setzen Sie noch eine Schaltfläche auf das `Panel`-Steuerelement und stellen Sie dann die in Tabelle 20.8 gezeigten Eigenschaften ein.

Tabelle 20.8: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	`Text`	Asynchrone Auto-Suche
Formular	`AcceptButton`	btnAbfragen
Schaltfläche	`Name`	btnAbfragen
Schaltfläche	`Text`	Abfragen
Textfeld	`Font`	Courier New
Textfeld	`Name`	txtResultat
Textfeld	`MultiLine`	True
Textfeld	`ScrollBars`	Vertical
Textfeld	`Text`	(leer)

Fügen Sie `using`-Anweisungen für `System.Net`, `System.Text`, `System.Runtime.InteropServices`, `System.IO` und `System.Collections.Specialized` hinzu. Definieren Sie außerdem zwei globale Variablen.

```
private int m_AktiveAutoSuchen = 0;
private int m_StartTickCount = 0;
```

Tragen Sie dann den folgenden Code in die Methode für das `Click`-Ereignis der Schaltfläche ein:

```
private void btnAbfrage_Click(object sender, System.EventArgs e)
{
  //Vorbereitungen
  btnAbfrage.Enabled = false;
  m_StartTickCount = Environment.TickCount;
  txtResultat.Text =
    "UI-ThreadId=" + AppDomain.GetCurrentThreadId().ToString() + "\r\n";
  txtResultat.Text += "Zeit\tEmpf.\tThread\tAnzahl gelesene Bytes\r\n";
  NameValueCollection Parameter = new NameValueCollection();

  //Suche auf www.auto.de mit HTTP GET
  Parameter.Add("Modelle", "1060");
  Parameter.Add("Marke", "0");
  Parameter.Add("Preisv", "0");
  Parameter.Add("Preisb", "0");
  Parameter.Add("MwSt", "0");
  Parameter.Add("Farbe", "0");
  Parameter.Add("Typ", "1");
  Parameter.Add("Kategorie", "0");
  Parameter.Add("PLZBereich", "-1");
  Parameter.Add("Erstzulassung", "0");
  Parameter.Add("KmStand", "10000");
  Parameter.Add("PSLeistung", "0");
  Parameter.Add("Sortierungnach", "AutodePreis");
  Parameter.Add("suche", "Suche+starten");
  Parameter.Add("AModell", "");
  Parameter.Add("MOVE", "1");
  Parameter.Add("FahrzeugID", "1");
  Parameter.Add("Fahrzeug", "PKW");
  AutoMitAsyncHttpGetSuchen(Parameter,
    "http://www.automail.de/SucheErgebnis.asp",
    Application.StartupPath + "\\WwwAutoDe.Htm");

  //Suche auf de.mobile.de mit HTTP POST
  Parameter.Clear();
  Parameter.Add("bereich", "pkw");
  Parameter.Add("sprache", "1");
  Parameter.Add("DoSearch", "1");
  Parameter.Add("FormMake", "214");
  Parameter.Add("FormModel", "");
  Parameter.Add("FormPrice", "100000000-124999999");
  Parameter.Add("FormEZ", "-");
  Parameter.Add("FormKilometer", "-10000");
  Parameter.Add("FormPower", "-");
  Parameter.Add("FormCategory", "13");
  Parameter.Add("FormColor", "...beliebig");
  Parameter.Add("FormLand", ".");
  Parameter.Add("FormPLZ", "");
  Parameter.Add("FormDurchmesser", "0");
  Parameter.Add("FormDate", "3");
```

```
  Parameter.Add("FormSort", "0");
  Parameter.Add("x", "37");
  Parameter.Add("y", "7");
  AutoMitAsyncHttpPostSuchen(Parameter,
    "http://de.mobile.de/cgi-bin/search.pl",
    Application.StartupPath + "\\DeMobileDe.Htm");
}//btnAbfragen_Click
```

Ergänzen Sie außerdem die Methode `AutoMitAsyncHttpGetSuchen()`.

```
private void AutoMitAsyncHttpGetSuchen(
  NameValueCollection Parameter, string WebAdresse, string DateiName)
{
  WebClient wc = new WebClient();

  lock(this)
    m_AktiveAutoSuchen++;

  ZustandDerHttpGetSuche z = new ZustandDerHttpGetSuche();
  z.DateiName = DateiName;
  try
  {
    wc.QueryString = Parameter;
    z.WebClientStream = wc.OpenRead(WebAdresse);
    z.WebClientStream.BeginRead(z.Puffer, 0, z.Puffer.Length,
      new AsyncCallback(HttpGetEmpfänger), z);
  }
  catch (Exception x)
  {
    MessageBox.Show(x.Message);
  }
}//AutoMitAsyncHttpGetSuchen
```

Die Methode `AutoMitAsyncHttpGetSuchen()` inkrementiert zuerst die Variable für die Anzahl der aktiven Suchen. Das ist nötig, um die in `btnAbfragen_Click()` deaktivierte Schaltfläche nach dem Ende aller Suchen wieder verfügbar zu machen. Der Aufruf von `lock()` verhindert, dass der zweite Such-Thread die Variable zum gleichen Zeitpunkt inkrementieren kann. Dazu später mehr.

`AutoMitAsyncHttpGetSuchen()` erzeugt dann ein Zustandsobjekt, das diverse Variablen speichert. Da die `WebClient`-Klasse keine asynchrone Version von `DownloadFile()` besitzt, müssen Sie ihre `OpenRead()`-Methode benutzen. Diese sendet die Anfrage und liefert einen Stream zurück, also nichts Asynchrones bis hierhin.

Das fängt erst in der nächsten Zeile an, denn die Methode `BeginRead()` des Streams gibt die Kontrolle sofort zurück. Sie erwartet als dritten Parameter eine Rückrufmethode in Form einer Delegatinstanz, die sie jeweils beim Eintreffen der Daten aufruft. Die Daten werden dabei in einem `byte`-Array geliefert, das in den ersten drei Parametern beschrieben wird. Der vierte Parameter ist das Zustandsobjekt, das ebenfalls an die Rückrufmethode übergeben wird. Diese schließt auch den Stream.

Der Prototyp der Rückrufmethode wird durch den im Namensraum `System` zu findenden Delegaten `AsyncCallback` beschrieben.

```
public delegate void AsyncCallback(IAsyncResult ar);
```

Die Rückrufmethode darf also keinen Wert zurückgeben und muss einen einzelnen Parameter akzeptieren, nämlich einen `IAsyncResult`-Schnittstellenzeiger. Über diesen Parameter ist unter anderem das vorhin übergebene Zustandsobjekt verfügbar. Hier ist der Code dieser Rückrufmethode:

```
public void HttpGetEmpfänger(IAsyncResult ar)
{
  FileStream fs = null;

  try
  {
    ZustandDerHttpGetSuche z = (ZustandDerHttpGetSuche)ar.AsyncState;
    int AnzahlGelesenerBytes = z.WebClientStream.EndRead(ar);
    //Bericht-Eintrag ausgeben
    txtResultat.Invoke(
      new SchreibeBerichtDelegate(SchreibeBericht),
      new Object[] {Environment.TickCount - m_StartTickCount,
      "GET", AppDomain.GetCurrentThreadId(), AnzahlGelesenerBytes});
    //Daten einlesen
    if (AnzahlGelesenerBytes > 0)
    {
      z.Antwort.Write(z.Puffer, 0, AnzahlGelesenerBytes);
      z.WebClientStream.BeginRead(z.Puffer, 0, z.Puffer.Length,
        new AsyncCallback(HttpGetEmpfänger), z);
    }
    else
    {
      fs = new FileStream(z.DateiName, FileMode.Create);
      fs.Write(z.Antwort.ToArray(), 0, (int)z.Antwort.Length);
      z.Dispose();
      AutoSucheBeendet();
    }
  }
  catch (Exception x)
  {
    MessageBox.Show(x.Message);
  }
  finally
  {
    if (fs != null)
      fs.Close();
  }
}//HttpGetEmpfänger
```

Das Zustandsobjekt steckt in der `AsyncState`-Eigenschaft des `ar`-Parameters. Über dieses Zustandsobjekt ist das `Stream`-Objekt der Web-Anfrage zugänglich. Mit der `EndRead()`-Methode dieses Objekts lässt sich ermitteln, wie viele Bytes in den Puffer übertragen wurden, um dessen Inhalt in einen anderen

Stream zu übertragen, in das Antwortobjekt. Durch einen erneuten Aufruf von `BeginRead()` werden die nächsten Daten abgeholt. Sobald diese Daten verfügbar sind, kommt erneut die Rückrufmethode zum Zuge, bis das Ende des Streams erreicht ist.

Wenn keine Daten mehr verfügbar sind, schreibt die Rückrufmethode die gesamte Antwort in eine Datei. Dann ruft sie die Methode `Dispose()` des Zustandsobjekts auf, damit es seine unverwalteten Ressourcen freigeben kann. Der Aufruf der Methode `AutoSucheBeendet()` führt nach Abschluss beider Suchen zur erneuten Aktivierung der Schaltfläche. Sowohl das Zustandsobjekt als auch die Methode `AutoSucheBeendet()` werden später noch näher erläutert.

Als Nächstes sehen Sie, wie das Ausgeben der Einträge in das Textfeld implementiert ist. Die Implementation der Rückrufmethode zeigte schon, dass das Textfeld eine Methode namens `Invoke()` hat. Diese Methode synchronisiert den Thread, der die Rückrufmethode ausführt, mit dem Thread der Oberfläche. Das ist nötig, weil die von .NET zur Verfügung gestellten Hüllklassen für alle Oberflächenobjekte, also auch Formulare, nicht Threadsicher sind: Zugriffe und Methodenaufrufe sollten nur aus dem Thread heraus erfolgen, der das jeweilige Steuerelement angelegt hat. Üblicherweise ist das der Haupt-Thread des Programms, alle Oberflächenobjekte laufen also im selben Thread.

Um die Arbeit mit Hintergrund-Threads zu vereinfachen, hat Microsoft allen Steuerelementen eine Methode `Invoke()` spendiert. Andere Klassen, wie `ArrayList` und ihre Verwandten aus dem `System.Collections` Namensraum, bieten eine Methode namens `Synchronized()`. Diese liefert einen Threadsicheren Wrapper. Wenn beides fehlt, müssen Sie selbst Hand anlegen. Dazu können Sie in C# beispielsweise die Methode `lock()` benutzen, ein Beispiel dazu folgt gleich.

Aber zurück zum Code. Im Aufruf der Methode `Invoke()` wurde ein `SchreibeBerichtDelegate` namens `SchreibeBericht` instanziiert. Außerdem wurde ein `object`-Array mit vier Elementen übergeben. Hier sehen Sie die Definition des Delegaten und der passenden Methode:

```
public delegate void SchreibeBerichtDelegate(
  int Zeit, string Empfänger, int ThreadId, int AnzahlGelesenerBytes);
public void SchreibeBericht(
  int Zeit, string Empfänger, int ThreadId, int AnzahlGelesenerBytes)
{
  txtResultat.Text += string.Format(
  "{0}\t{1}\t{2}\t{3}\r\n", new object[] {
  (Environment.TickCount - m_StartTickCount).ToString(),
  Empfänger,
  AppDomain.GetCurrentThreadId().ToString(),
  AnzahlGelesenerBytes.ToString()});
}//SchreibeBericht
```

Die darzustellenden Werte werden also 1:1 durchgereicht und als String formatiert. Als Belohnung für diese paar Zeilen spendiert .NET Multi-Threading-Fähigkeit.

Die Methode `AutoSucheBeendet()` wird von den Such-Threads nach Abschluss ihrer eigentlichen Arbeit aufgerufen und entscheidet, ob die Schaltfläche wieder aktiviert werden kann. Da es sehr wohl denkbar ist, dass zwei oder mehrere Suchoperationen zur selben Zeit enden, ist hier eine Synchronisierung notwendig.

```
private void AutoSucheBeendet()
{
  lock(this)
  {
    MessageBeep(1);
    AktiveAutoSuchen--;
    if (AktiveAutoSuchen == 0)
     btnAbfrage.Enabled = true;
  }
}
```

Wie in Teil 5 noch ausführlicher beschrieben, kann man mit `lock()` dafür sorgen, dass sich zwei oder mehr Threads bei Zugriffen auf ein und dasselbe Objekt nicht in die Quere kommen. Bei zwei gestarteten Suchvorgängen sieht das Szenario folgendermaßen aus:

1. Die erste Suche ist beendet. Der Thread meldet über `lock(this)` eine Sperre für das Formularobjekt an, die er erst nach Ausführung *aller* Aktionen (Erniedrigen des Zählers, Prüfung und gegebenenfalls Aktivierung der Schaltfläche) wieder abgibt.
2. Während der erste Thread diese Operationen durchführt, ist auch der zweite Thread fertig und ruft ebenfalls `AutoSucheBeendet()` auf. Er wird durch das `lock(this)` so lange blockiert, bis der erste Thread den Codeblock vollständig abgearbeitet hat.

Ohne diese Sperre wäre es möglich, dass der erste Thread den Zähler exakt in dem Moment prüft, indem ihn der zweite Thread herabsetzt.

Im Allgemeinen wird man `this` als Argument für `lock()` übergeben, in statischen Methoden und zum Schutz statischer Variablen ist es `typeof(Meinklassenname)`. (Jedes Programm kann eine beliebige Zahl solcher Sperren und Sperrobjekte verwenden – entscheidend ist, dass dieselben Werte an jeder Stelle auch durch dieselben Sperrobjekte geschützt werden.)

Wie schon gesagt, gibt es keine Methode zum Ausgeben eines schlichten akustischen Signals im Framework, weshalb das Beispielprogramm die API-Methode `MessageBeep()` per Plattformaufruf einbindet. Hier ist die Deklaration dazu:

```
[DllImport("user32.dll")]
public static extern bool MessageBeep(uint soundtype);
```

Jetzt zum letzten Teil des Puzzles, den Zustandsklassen. Sie vereinfachen die Übergabe von Daten zwischen dem Programmteil, der die asynchrone Operation anstößt, und der Rückrufmethode. Wie Sie bereits gesehen haben, wird eine Instanz der Zustandsklasse im dritten Parameter der Methode `BeginRead()` des Streams übergeben. Diese Instanz steht in der Rückrufmethode über die `AsyncState`-Eigenschaft ihres Parameters zur Verfügung.

Die hier benutzte Zustandsklasse speichert den Namen, unter dem die Webseite gespeichert werden soll. Außerdem speichert sie den Stream, durch dessen `BeginRead()`- und `EndRead()`-Methoden die Daten abgerufen werden, sowie einen Stream zum Zwischenspeichern der bisher gelesenen Daten. Leider gibt es keine Möglichkeit, die Daten direkt vom einen in den anderen Stream zu übertragen. Stattdessen muss man sie über ein `byte`-Array (oder `StreamReader`-/`StreamWriter`-Instanzen) kopieren. Auch dieses Array stellt die Zustandsklasse bereit. Hier sind die Definitionen für die benutzte Zustandsklasse sowie für ihre Basisklasse:

```
private class ZustandDerSuche : IDisposable
{
  public string DateiName = "";

  public virtual void /*IDisposable*/ Dispose()
  {
    //Die Basisklasse hat keine unverwalteten Ressourcen
  }//Dispose
}//class ZustandDerSuche


private class ZustandDerHttpGetSuche : ZustandDerSuche
{
  public MemoryStream Antwort = new MemoryStream();
  //Max. Paketgröße im Ethernet = 1500 Bytes
  public Byte[] Puffer = new Byte[1500];//Paketgröße
  public Stream WebClientStream = null;

  public override void Dispose()
  {
    if (WebClientStream != null)
      WebClientStream.Close();
    if (Antwort != null)
      Antwort.Close();
  }//Dispose
}//class ZustandDerHttpGetSuche
```

Die Basisklasse existiert, weil es neben der Zustandsklasse `ZustandDerHttpGetSuche` noch eine für die Suche mit HTTP POST gibt. Aufgabe der Basisklasse ist dabei vor allem die Verpflichtung ihrer abgeleiteten Klassen zur Implementierung der Schnittstelle `IDisposable`, die nur eine einzige Methode

namens `Dispose()` umfasst und von Clients vor dem Ungültigwerden der Referenz für eine Instanz der Zustandsklasse aufgerufen wird. Das ist nötig, um die unverwalteten Ressourcen dieser Klasse freizugeben. Wie Sie sehen, macht die Methode das in `ZustandDerHttpGetSuche` für die beiden Streams. Das `byte`-Array wird vom Garbage-Collector entsorgt, wie alle anderen verwalteten Ressourcen auch.

Nun stellt sich natürlich die Frage, welche Objekte tief in ihrem Innern unverwaltete Ressourcen beherbergen und deshalb explizit durch Aufruf einer Methode wie `Close()` freigegeben werden müssen. Die Antwort ist ganz einfach: alle, die eine Methode `Close()` haben. Wenn Sie das für den `MemoryStream` anzweifeln, haben Sie Recht. Aber es ist sicherer, sich ein bedingungsloses Schließen anzugewöhnen. Die Methode `Dispose()` hat den gleichen Zweck, auch sie sollte, wenn vorhanden, bedingungslos aufgerufen werden.

Und warum der Umweg über die `IDisposable`-Schnittstelle, statt die Methode einfach zu implementieren? Das wird klar, sobald Sie Objekte verschiedener Typen gemeinsam zum Beispiel in einer `ArrayList` verwalten. Wenn alle Objekte `IDisposable` implementieren, können Sie in einer Schleife einfach in den Typ `IDisposable` umwandeln und so die `Dispose()`-Methoden aller Objekte aufrufen.

Trotz des Umwegs ist die vorgestellte `Dispose()`-Implementation noch recht einfach. Häufig werden Sie stattdessen das so genannte Dispose Pattern sehen. Dieses Lösungsmuster gibt die unverwalteten Ressourcen wahlweise über die Methode `Dispose()` oder im Finalizer frei. Dabei können in `Dispose()` auch die unverwalteten Ressourcen freigegeben werden, die in .NET Objekten gekapselt sind. Beim Aufruf des Finalizers existieren diese Objekte aber eventuell schon gar nicht mehr. Dort können also nur ungekapselte Ressourcen freigegeben werden. Wenn diese Ressourcen bereits freigegeben wurden, sollte der (aufwändige) Finalizer-Lauf unterbleiben. Das führt zu folgender Implementierung, die nicht im Beispielprojekt enthalten ist:

```
private class ZustandDerSuche : IDisposable
{
  public string DateiName = "";

  //Finalizer, kein Destruktor
  ~ZustandDerSuche()
  {
    GibUnverwalteteRessourcenFrei(false);
  }//finalize

  public void /*IDisposable*/ Dispose()
  {
    GC.SuppressFinalize(this);
    GibUnverwalteteRessourcenFrei(true);
  }//Dispose
```

```
  protected virtual void GibUnverwalteteRessourcenFrei(
    bool IstExpliziterAufruf)
  {
    //Die Basisklasse hat keine unverwalteten Ressourcen
  }//GibUnverwalteteRessourcenFrei
}//class ZustandDerSuche


private class ZustandDerHttpGetSuche : ZustandDerSuche
{
  public MemoryStream Antwort = new MemoryStream();
  public Byte[] Puffer = new Byte[1500];
  public Stream WebClientStream = null;

  protected override void GibUnverwalteteRessourcenFrei(
    bool IstExpliziterAufruf)
  {
    if (IstExpliziterAufruf)
    {
     //Unverwaltete Ressourcen freigeben
      if (WebClientStream != null)
        WebClientStream.Close();
      if (Antwort != null)
        Antwort.Close();
     }// if (IstExpliziterAufruf)
     //Unverwaltete und ungekapselte (!) Ressourcen freigeben
  }//GibUnverwalteteRessourcenFrei
}//class ZustandDerHttpGetSuche
```

Den zweiten Teil des Programms bildet die Abfrage per HTTP POST. Die Methode `AutoMitAsyncHttpPostSuchen()` ist völlig anders aufgebaut als die Methode `AutoMitAsyncHttpGetSuchen()`. Weil mit der `WebClient`-Klasse kein asynchrones HTTP POST möglich ist, kommt hier die `WebRequest`-Klasse zum Einsatz. Wie üblich bedeuten mehr Einstellmöglichkeiten auch mehr Arbeit.

```
private void AutoMitAsyncHttpPostSuchen(
  NameValueCollection Parameter, string WebAdresse, string DateiName)
{
  Stream stm = null;

  lock(this)
    m_AktiveAutoSuchen++;

  //Parameter in Body schreiben
  StringBuilder sb = new StringBuilder();
  for (int i = 0; i < Parameter.Count; i++)
  {
    //Annahmen: 1 Value / Key, HttpUtility.UrlEncode unnötig
    sb.Append(Parameter.GetKey(i) + "=" +
      Parameter.GetValues(i).GetValue(0) + "&");
  }
```

```
  try
  {
    WebRequest req = WebRequest.Create(WebAdresse);
//  HttpWebRequest req = (HttpWebRequest)WebRequest.Create(WebAdresse);
//  req.UserAgent = "Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0)";
//  req.Referer =
//    "http://de.mobile.de/cgi-bin/search.pl?bereich=pkw&sprache=1";
//  req.Accept =
//    "image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*";
//  req.Headers["Accept-Language"] = "de";
    req.Method = "POST"; //vor Aufruf von GetRequestStream() setzen
    string s = sb.ToString();
    if (s.EndsWith("&"))
    {
      s.Remove(s.Length - 1, 1);
      req.ContentLength = s.Length; //vor dem Schreiben setzen
      stm = req.GetRequestStream();
      StreamWriter sw = new StreamWriter(stm, Encoding.ASCII);
      sw.Write(s);
      sw.Close();
    }
    else
    {
      req.ContentLength = 0;
    }
    req.ContentType = "application/x-www-form-urlencoded";
    ZustandDerHttpPostSuche z = new ZustandDerHttpPostSuche();
    z.Anfrage = req;
    z.DateiName = DateiName;
    req.BeginGetResponse(new AsyncCallback(HttpPostEmpfänger), z);
  }
  catch (Exception x)
  {
    MessageBox.Show(x.Message);
  }
}//AutoMitAsyncHttpPostSuchen
```

Die hier benutzte Klasse `WebRequest` hat keine `Parameter`-Eigenschaft, deshalb müssen die übergebenen Parameter über einen Stream gesetzt werden. Dazu erzeugen Sie zuerst einen String im gleichen Format wie beim HTTP GET. Da String-Objekte beim Anfügen eines zusätzlichen Strings hinter den Kulissen erst zerstört und dann mit dem Gesamt-String wieder aufgebaut werden, empfiehlt sich zum Zusammenbauen des Strings ein `StringBuilder`-Objekt. Wie zu sehen, könnte die `NameValueCollection` mehr als einen Wert pro Schlüssel speichern.

`WebRequest` ist eine abstrakte Klasse, und ihre Methode `Create()` ist kein Konstruktor, sondern eine statische Factory-Methode. Je nach Protokoll der übergebenen URL liefert sie ein `HttpWebRequest`- oder ein `FileWebRequest`-Objekt. Im Allgemeinen ist eine explizite Typumwandlung unnötig. Aber wenn ein misstrauischer Server auf Angaben wie *UserAgent* oder *Referrer* achtet, können Sie den hier auskommentierten Code benutzen.

Falls Sie hier ein `stm.Close()` vermissen: Das ist hier unnötig. Die `StreamWriter`-Instanz schließt bei `sw.Close()` freundlicherweise auch den darunter liegenden Stream.

Der Rest funktioniert wie schon bei der Methode `AutoMitAsyncHttpGetSuchen()` beschrieben. Hier ist der Code für die Methode `HttpPostEmpfänger()`:

```
public void HttpPostEmpfänger(IAsyncResult ar)
{
  FileStream fs = null;
  StreamReader sr = null;
  WebResponse resp = null;

  try
  {
    ZustandDerHttpPostSuche z = (ZustandDerHttpPostSuche)ar.AsyncState;
    WebRequest req = z.Anfrage;
    resp = req.EndGetResponse(ar);
    //resp.ContentLength muss vom Server nicht gesetzt werden, und
    //stm.CanSeek ist false, Peek funktioniert also auch nicht.
    //Also umständlich über StreamReader lesen - der findet selbst raus,
    //wo Schluss ist.
    sr = new StreamReader(resp.GetResponseStream());
    string s = sr.ReadToEnd();
    //Bericht-Eintrag ausgeben
    txtResultat.Invoke(
      new SchreibeBerichtDelegate(SchreibeBericht),
      new Object[] {Environment.TickCount - m_StartTickCount,
      "POST", AppDomain.GetCurrentThreadId(), s.Length});
    //Daten einlesen
    if (s.Length > 0)
    {
      fs = new FileStream(z.DateiName, FileMode.Create);
      fs.Write(Encoding.ASCII.GetBytes(s), 0, s.Length);
      z.Dispose();
      AutoSucheBeendet();
    }
  }
  catch (Exception x)
  {
    MessageBox.Show(x.Message);
  }
  finally
  {
    if (fs != null)
      fs.Close();
    if (sr != null)
      sr.Close();
    if (resp != null)
      resp.Close();
  }
}//HttpPostEmpfänger
```

20.3.9 Austauschbare Protokolle mit den Klassen WebRequest und WebResponse

Der Begriff »Austauschbare Protokolle« (Pluggable Protocols) kommt Ihnen vielleicht unbekannt vor, aber Sie haben das Konzept schon in den vorhergehenden Beispielen dieses Kapitels gesehen und zwar in Form der Klassen `WebRequest`, `HttpWebRequest` und `FileWebRequest`.

Durch austauschbare Protokolle stellt .NET einen Mechanismus zur Verfügung, der passend zum Protokollnamen eines URI (Uniform Resource Identifier) die richtige Anfrageklasse instanziiert. Wenn der URI mit `"http:"` oder `"https:"` beginnt, wird ein `HttpWebRequest` instanziiert, bei `"file:"` ein `FileWebRequest`. Beide sind von `WebRequest` abgeleitet und bieten deshalb Methoden wie `GetResponse()` bzw. deren asynchrone Varianten `BeginGetResponse()` und `EndGetResponse()`. Daneben bieten sie protokollspezifische Eigenschaften und Methoden, `HttpWebRequest` zum Beispiel die `Referer`- oder die `UserAgent`-Eigenschaft.

Dementsprechend liefern `GetResponse()` und `EndGetResponse()` auch Response-Objekte mit protokollspezifischen Eigenschaften und Methoden. Die Klassen dieser Objekte sind von `WebResponse` abgeleitet, und .NET enthält bereits die Klassen `HttpWebResponse` und `FileWebResponse`.

Diesen Mechanismus können Sie um beliebige Protokolle erweitern. Neben Standard-Internetprotokollen wie FTP (File Transfer Protocol) oder NNTP (Network News Transfer Protocol) können das auch völlig selbstgestrickte Protokolle sein. Lassen Sie sich von den Namen `WebRequest` und `WebResponse` nicht in die Irre führen: TCP/IP ist zwar meistens, aber nicht zwingend mit von der Partie, wie das file-Protokoll ja schon zeigt. Allerdings bleibt die Magie auf die Instanz beschränkt, die das Protokoll registriert, eine rechnerweite Benutzung ist nicht vorgesehen. Außerdem können auch keine vorhandenen Protokolle ersetzt oder sonst wie beeinflusst werden.

Austauschbare Protokolle könnte man leicht mit »Asynchronous Pluggable Protocols« verwechseln. Aber obwohl die Idee die gleiche ist, sind Funktionalität und Implementation völlig verschieden.

Ein naheliegendes Beispiel für austauschbare Protokolle wäre FTP. Eigentlich ist es sogar erstaunlich, dass .NET keine Implementierung liefert. Leider liefert dieser Abschnitt auch keine, denn FTP ist nicht auf zwei Buchseiten zu implementieren – im Internet finden Sie aber diverse Implementierungen. Da die Online-Dokumentation von Visual Studio kein vollständiges Beispiel zu austauschbaren Protokollen enthält, finden Sie hier ein »Hallo Welt«-Beispiel, das die Zusammenarbeit der beteiligten Klassen und Methoden verdeutlicht.

Codebeispiel – Austauschbare Protokolle

Die Oberfläche in dem Beispielprojekt *PlugProt* kommt mit zwei Schaltflächen aus, deshalb wurde hier der Platz für den üblichen Screenshot und die Tabelle mit den Eigenschaftswerten eingespart.

Beginnen Sie ein neues Projekt vom Typ *Windows-Anwendung* und fügen Sie eine Klassendatei mit folgendem Inhalt hinzu:

```
using System;
using System.Net;

namespace PlugProt
{
  public class PpRequestCreator : IWebRequestCreate
  {
    public PpRequestCreator()
    {
    }//ctor

    public WebRequest Create(Uri Url)
    {
      return new PpRequest(Url);
    }//Create
  }//class PpRequestCreator

  public class PpRequest : WebRequest
  {
    private System.Uri m_Url;

    private PpRequest()
    {
      // Default-Konstruktor unsichtbar machen
    }//ctor

    internal PpRequest(System.Uri Url)
    {
      if (Url.ToString().ToUpper().StartsWith("PP:"))
        m_Url = Url;
      else
        throw new ArgumentException("Url muss mit 'PP:' beginnen", "Url");
    }//ctor

    public override WebResponse GetResponse()
    {
      return new PpResponse(m_Url);
    }//GetResponse

    public override Uri RequestUri
    {
      get {return m_Url;}
    }//RequestUri

  }//class PpRequest
}//namespace PlugProt
```

Das ist die absolute Minimal-Implementation für eine von `WebRequest` abgeleitete Klasse und ihre Instanziierung. Die Klasse `PpRequest` macht zuerst ihren Standardkonstruktor unsichtbar. Das ist nötig, weil sie nur durch die Factory-Methode von `PpRequestCreator` instanziierbar sein soll und jede Klasse immer einen parameterlosen Standardkonstruktor hat. Dieser Konstruktor ist "unkaputtbar": Auch, wenn er nicht im Quelltext erscheint, ist er vorhanden.

Dann definiert `PpRequest` einen internen Konstruktor. Dieser ist also nur von innerhalb der Assembly definierten Klassen aufrufbar. Auch das ist eher ein Klimmzug als eine Lösung, denn nach dem Muster für austauschbare Protokolle soll der Konstruktor nur von `PpRequestCreator` aufrufbar sein. In C# stellt eine Datei aber keine Grenze dar, wie zum Beispiel in Delphi. C# kennt als Grenze nur die Klasse und die Assembly.

Passend zur Signatur der Factory-Methode von `PpRequestCreator` hat der interne Konstruktor einen Parameter vom Typ `System.Uri`. Nach einem groben Test merkt sich `PpRequest` die URI und benutzt sie in `GetResponse()`, um ein Objekt vom Typ `PpResponse` zurückzugeben. Entsprechend dem Muster für austauschbare Protokolle muss die benutzte Klasse von `WebResponse` abgeleitet sein. Die übergebene URL kann zusätzlich über die Eigenschaft `RequestUri` abgefragt werden, was für das Beispiel aber nicht unbedingt nötig ist.

Die Klasse `PpRequestCreator` implementiert die Schnittstelle `IWebRequestCreate` und verpflichtet sich damit zur Bereitstellung einer Methode `Create()`, die einen Parameter vom Typ `System.Uri` entgegen nimmt und ein Objekt vom Typ `WebRequest` zurückliefert. Auch das ist typisch für das Muster für austauschbare Protokolle: Statt der Superklassen-Instanz wird natürlich ein `PpRequest`-Objekt geliefert.

Damit ist der Anfrageteil erledigt, es folgt der Antwortteil. Legen Sie dazu eine neue Klassendatei mit folgendem Inhalt an:

```
using System;
using System.Net;
using System.IO;

namespace PlugProt
{
  public class PpResponse : WebResponse
  {
    MemoryStream m_ms;

    private PpResponse()
    {
      // Default-Konstruktor unsichtbar machen
    }//ctor
```

```
    //Versteckter Konstruktor.
    //PpResponse-Instanz nur über WebRequest erhältlich.
    internal PpResponse(Uri Url)
    {
      //Url auswerten ...
      m_ms = new MemoryStream();
      StreamWriter sw = new StreamWriter(m_ms);
      sw.Write("Hallo Welt!");
      sw.Flush();
      //sw.Close(); würde auch Stream schließen.
      m_ms.Seek(0, SeekOrigin.Begin);
    }//ctor

    public override long ContentLength
    {
      get {return m_ms.Length;}
      //set;
    }//ContentLength

    public override Stream GetResponseStream()
    {
      return m_ms;
    }//GetResponseStream

  }//class PpResponse
}//namespace PlugProt
```

Entsprechend dem Muster für austauschbare Protokolle ist die Klasse `PpResponse` von `WebResponse` abgeleitet. Auch hier wird der Standardkonstruktor versteckt und ein interner implementiert. Der wertet die übergebene URL aber nicht aus, sondern initialisiert den Antwort-Stream einfach mit `"Hallo Welt!"`. Der Antwort-Stream kann über die Methode `GetResponseStream()` abgefragt werden. Damit ist der entscheidende Teil des Musters implementiert, die Eigenschaft `ContentLength` wird im Beispiel nicht benutzt. Hinweise zu allen anderen ererbten Methoden und Eigenschaften finden Sie in den Code-Kommentaren im Beispiel-Ordner.

Die folgende Besprechung des Testaufbaus liefert den letzten Teil des Puzzles: Wie findet .NET die zum Protokollnamen passende Klasse?

Setzen Sie dazu zwei Schaltflächen auf das Formular, nennen Sie sie `btnRegistrieren` und `btnLaden`, fügen Sie `using`-Anweisungen für `System.Net` und `System.IO` hinzu und geben Sie dann den folgenden Code in die Methoden zur Behandlung der `Click`-Ereignisse der Schaltflächen ein:

```
private void btnLaden_Click(object sender, System.EventArgs e)
{
  PpRequest req;
  PpResponse resp;
```

```
  try
  {
    req = (PpRequest)WebRequest.Create("PP:Test");
    resp = (PpResponse)req.GetResponse();
    StreamReader sr = new StreamReader(resp.GetResponseStream());
    MessageBox.Show(sr.ReadToEnd(),
      this.Text, MessageBoxButtons.OK, MessageBoxIcon.Information);
  }
  catch (NotSupportedException)
  {
    MessageBox.Show(
      "Vor dem Laden muss der Handler" +
      " für austauschbare Protokolle registriert werden!",
      this.Text, MessageBoxButtons.OK, MessageBoxIcon.Information);
  }
}//btnLaden_Click


private void btnRegistrieren_Click(object sender, System.EventArgs e)
{
  bool Success;
  PpRequestCreator rc;
  rc = new PpRequestCreator();

  Success = WebRequest.RegisterPrefix("PP:", rc);
  if (! Success)
  {
    this.btnLaden.Enabled = false;
    MessageBox.Show("Konnte Handler für austauschbare Protokolle" +
      " nicht registrieren.",
      this.Text, MessageBoxButtons.OK, MessageBoxIcon.Stop);
  }
}//btnRegistrieren_Click
```

Im Code der REGISTRIEREN-Schaltfläche findet sich des Rätsels Lösung: Der statischen Methode `RegisterPrefix()` der `WebRequest`-Klasse werden einfach Protokollname und das zugehörige Instanziierungs-Objekt übergeben.

Der Code der LADEN-Schaltfläche könnte genauso gut `WebRequest`- und `WebResponse`-Objekte benutzen, weil `PpRequest` und `PpResponse` keine neuen Mitglieder implementieren. Da beide Basisklassen abstrakt sind, erzeugt der Zugriff auf nicht überschriebene Mitglieder eine Ausnahme vom Typ `NotSupportedException`.

21 ASP.NET allgemein

ASP.NET ist das .NET-Angebot für das Internet: Webanwendungen und Webdienste. Während Webanwendungen ihre Funktionalität dem Benutzer über HTML-Oberflächen anbieten, haben Webdienste anstelle einer Oberfläche eine API: Ihre Funktionalität wird ausschließlich von anderen Programmen genutzt, nicht von menschlichen Benutzern.

Trotz dieser »oberflächlichen« Unterschiede haben Webanwendungen und Webdienste vieles gemeinsam, insbesondere benutzen sie in großen Teilen dieselbe Infrastruktur. Mit diesen Gemeinsamkeiten befasst sich dieses Kapitel.

Hier geht es also um die technischen Grundlagen, auf denen die Kapitel Webanwendungen (Seite 881) und Webdienste (Seite 941) aufbauen. Je nach Interessenlage und verfügbarer Zeit können Sie dieses Kapitel zuerst lesen oder die Praxiskapitel vorziehen und dieses dann als Referenz benutzen.

21.1 Wie kommt die Endung .NET an ASP?

Nachdem Microsoft die Entwicklung von HTML-Oberflächen mit ASP revolutioniert hat, könnte man vermuten, dass ASP.NET aus der Portierung und dem Anhängen des .NET-Postfix besteht. Erfreulicherweise ist ASP.NET aber eine ebenso konsequente wie ambitionierte Weiterentwicklung von ASP.

Das Revolutionäre an ASP war der Produktivitätssprung durch die Integration von Script-Engines, durch die Möglichkeit zur Einbindung von ActiveX-Komponenten sowie durch eine IDE (Integrated Development Environment). Dementsprechend manifestiert sich die Konsequenz der Weiterentwicklung zu ASP.NET neben der Integration »richtiger« Programmiersprachen, deren Klassen zur Laufzeit kompiliert werden, auch in der Übernahme des aus der Windows-Programmierung bekannten Steuerelemente- und Ereignis-Modells sowie in der Benutzung der »normalen« IDE.

Die Ambitioniertheit des Schritts von ASP zu ASP.NET zeigt sich darin, dass ASP.NET von Grund auf neu geschrieben wurde. Nur dadurch konn-

ten Probleme optimal gelöst werden, die sich im Laufe des ASP-Einsatzes gezeigt haben. Neben Wartungsproblemen durch die Vermischung von HTML und Script-Code waren das auch Performance-Probleme durch interpretierten statt kompilierten Code und die Thread-Abhängigkeit von in Visual Basic geschriebenen ActiveX-Komponenten. Die Benutzung der Registry und der IIS Metabase sowie die Notwendigkeit, die IIS (Internet Information Services) zum Austausch von Komponenten herunterzufahren, brachten weitere Wartungsprobleme mit sich.

Hier ist eine Liste der Verbesserungen, ohne Anspruch auf Vollständigkeit:

- Kompilierter statt interpretierter Code, dadurch Performance-Gewinn.
- Beliebige Programmiersprache nutzbar statt Beschränkung auf VBScript und JScript.
- Durch *CodeBehind* werden Code und HTML voneinander getrennt.
- Caching der Ausgabe konfigurierbar, dadurch Performance-Gewinn.
- Sitzungszustand auslagerbar, dadurch Einsatz in Web-Farmen möglich.
- Automatischer Neustart von Anwendungen, dadurch Schutz gegen Speicherlecks, Deadlocks und ungewollt beendete Programme.
- Konfiguration über XML-Dateien, dadurch Weitergabe per XCopy möglich.
- Komponenten können bei laufender Anwendung ersetzt werden, Herunterfahren des Servers ist nicht mehr nötig.
- Verbesserte Möglichkeiten zur Implementation und Konfiguration der Sicherheit.
- Mehr und bessere Steuerelemente.
- Nutzung der .NET-Laufzeit-Umgebung mit Garbage Collection, Ausnahme-Behandlung etc.
- Unterstützung von Webdiensten.

Und noch eine gute Nachricht: ASP und ASP.NET können parallel genutzt werden. Die IIS entscheiden anhand der Erweiterung des Dateinamens, welche Technologie genutzt wird. Statt der Erweiterung *.asp* benutzen Webanwendungen, bzw. deren einzelne Web Forms-Seiten die Erweiterung *.aspx*, Webdienste die Erweiterung *.asmx*.

21.2 Bearbeitung einer HTTP-Anfrage in ASP.NET

HTTP-Anfragen werden von den IIS an die HTTP-Pipeline von ASP.NET weitergeleitet, die durch einen ausgeklügelten Erweiterungs- und Konfigurations-Mechanismus den jeweiligen Anforderungen angepasst werden kann.

ASP.NET ist im Gegensatz zu ASP relativ unabhängig von den IIS. Zum einen konnte dadurch ASP.NET von Grund auf neu geschrieben werden, ohne Kompromisse für die parallele Nutzung bestehender ASP-Lösungen machen zu müssen. Zum anderen wird so die Portierung von ASP.NET auf andere Web Server und Betriebssysteme erleichtert. Neben dem Apache zum Beispiel auf Linux könnte das auch ein Westentaschen-Server in einem Router oder in einer Drehbank sein.

Abbildung 21.1 zeigt die an der Bearbeitung einer HTTP-Anfrage in ASP.NET beteiligten Module und Prozesse.

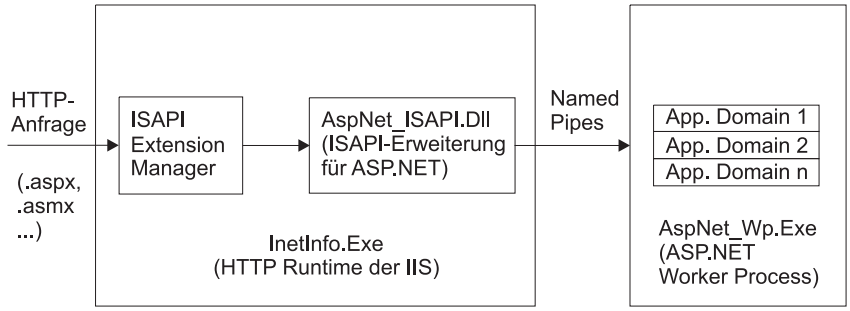


Abbildung 21.1: Behandlung einer HTTP-Anfrage durch ASP.NET

Die HTTP-Anfrage wird aufgrund der Erweiterung ihres Dateinamens vom ISAPI Extension Manager an die *AspNet_IsApi.Dll* weitergeleitet. Die entsprechenden Erweiterungen wie *ASPX* und *ASMX* sind dafür beim ISAPI Extension Manager registriert. Der Anschluss von ASP.NET an die IIS erfolgt also wie schon in ASP vorgesehen über eine *ISAPI.DLL*.

AspNet_IsApi.Dll leitet die Anfrage über Named Pipes weiter an *AspNet_Wp.Exe* – den Arbeitsprozess von ASP.NET. Durch die Trennung in zwei unabhängige Prozesse kann eine amoklaufende Anwendung den eigentlichen Serverprozess nicht gefährden.

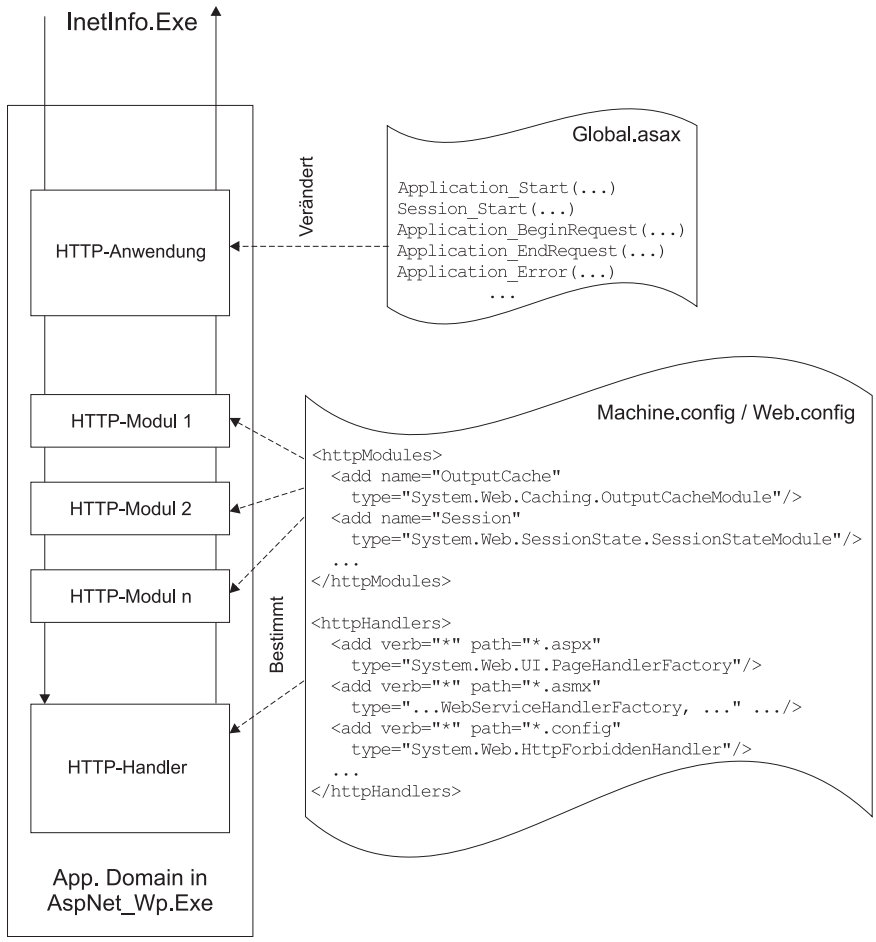
Der Begriff HTTP-Anwendung oder kurz Anwendung (Englisch: Application) bezeichnet dabei die Gesamtheit aller in einem Verzeichnis vorhandenen relevanten Dateien. Neben den Konfigurationsdateien sind das also vor allem die Dateien mit den Erweiterungen *ASPX* und *ASMX* sowie die Dateien im *bin*-Verzeichnis.

Alle Anfragen werden im selben Prozess bearbeitet, gegeneinander isoliert sind sie durch Anwendungsdomänen (Application Domains). Anwendungsdomänen sind ein neues Konzept von .NET und bieten Ähnliches für Sicherheit und Fehlertoleranz wie Prozesse, benötigen aber weniger Systemressourcen und weniger Zeit für den Kontextwechsel.

21.2.1 Funktionsweise der HTTP-Pipeline

Im Prozess *AspNet_Wp.Exe* wird die Anfrage durch die HTTP-Pipeline von ASP.NET bearbeitet. Den Aufbau dieser Pipeline zeigt Abbildung 21.2.

Abbildung 21.2: HTTP-Pipeline



Zuerst wird der HTTP-Anfrage ein Objekt vom Typ `HttpApplication` zugewiesen. Diese Objekte dienen der Steuerung des Ablaufs bei der Bearbeitung.

Die eigentliche Bearbeitung findet dann in HTTP-Modulen, HTTP-Handlern und in der Datei *Global.asax* bzw. der zugehörigen CodeBehind-Datei *Global.asax.cs* statt. Dadurch wird die HTTP-Pipeline gebildet.

Je nach Erweiterung des Dateinamens in der Anfrage wählt ASP.NET den HTTP-Handler aus, der auf diese Art der Anfrage spezialisiert ist. Für Anfragen nach Dateien mit der Erweiterung *ASPX* ist das zum Beispiel ein Objekt vom Typ `System.Web.UI.Page` – der Typ, von dem alle Web Forms-Seiten abgeleitet sind. Dieser Mechanismus ist leicht erweiterbar, denn Sie können beliebige eigene Typen definieren, die nur die Schnittstelle `IHttpHandler` implementieren, bzw. die Schnittstelle `IHttpAsyncHandler` für asynchrone HTTP-Handler implementieren müssen. HTTP-Handler sind mit ISAPI-DLLs unter ASP vergleichbar.

HTTP-Module können vor *und* nach der Bearbeitung der Anfrage durch den HTTP-Handler eingreifen, also sowohl die Anfrage aufbereiten als auch die Antwort nachbereiten. Dazu sind verschiedene Ereignisse zu behandeln, die vom `HttpApplication`-Objekt angeboten werden und den Ablauf der Bearbeitung widerspiegeln. Somit bieten sich HTTP-Module auch als Filter für Vorgänge wie Verschlüsselung oder Kompression an und müssen – erwartungsgemäß – die Schnittstelle `IHttpModule` implementieren. Wenn mehrere HTTP-Module dasselbe Ereignis bearbeiten, erfolgt die Bearbeitung in der Reihenfolge der Registrierung. HTTP-Module sind mit ISAPI-Filtern unter ASP vergleichbar.

Und schließlich kann der Code in der Datei Global.asax.cs in die Bearbeitung eingreifen. Auch der Anschluss dieses Codes erfolgt über Ereignisse. Diese werden entsprechend dem Auf- und Abbau der Anwendung und der Session sowie bei Fehlern und nach der Authentifizierung des Clients generiert. Eine detaillierte Beschreibung der Ereignisreihenfolge finden Sie in der Online-Dokumentation unter *HttpApplication-Klasse, Informationen zur HttpApplication-Klasse*. Wenn Code in der *Global.asax.cs* und ein HTTP-Modul dasselbe Ereignis bearbeiten, erfolgt die Bearbeitung zuerst im HTTP-Modul.

21.2.2 Objekte der HTTP-Pipeline

Zum Transport durch die HTTP-Pipeline wird jede Anfrage in ein Objekt des Typs `HttpContext` verpackt. Über dessen Eigenschaften sind neben der Anfrage selbst (`Request`-Eigenschaft) auch ihre Antwort (`Response`-Eigenschaft), ihr HTTP-Handler und diverse andere Objekte ansprechbar.

ASP.NET verwaltet für jede ANWENDUNG einen Pool von Objekten des Typs `HttpApplication` und sorgt über das Pooling dafür, dass nicht für jede Anfrage ein neues Objekt erzeugt werden muss, Objekte zu jedem Zeitpunkt aber maximal einer Anfrage zugeordnet sind. `HttpApplication`-Objekte bieten über ihre Eigenschaften zum Beispiel Zugriff auf das `HttpContext`-Objekt und eine Liste der HTTP-Module für diese Anfrage (`Modules`-Eigenschaft). Sie erlauben außerdem über ihre Ereignisse den Zugriff auf die verschiedenen Bearbeitungsschritte der Anfrage.

Bei der Instanziierung von Objekten des Typs `HttpApplication` wird der Code in der Datei *Global.asax.cs* zum Bestandteil des `HttpApplication`-Objekts. Dadurch können Sie nicht nur Code für die Bearbeitung von Ereignissen des `HttpApplication`-Objekts integrieren, sondern auch auf Ereignisse von HTTP-Modulen reagieren und Variablen mit unterschiedlicher Sichtbarkeit und Lebensdauer definieren. Mehr Informationen zu den Möglichkeiten von *Global.asax.cs* finden Sie in der Online-Dokumentation. Die Datei *Global.asax.cs* befindet sich immer im Wurzelverzeichnis der Anwendung.

Während sich `HttpApplication`-Objekte und der Code in der Datei *Global.asax.cs* auf »automagische« Weise vereinigen, müssen Sie HTTP-Module und HTTP-Handler kompilieren und das Resultat entweder in das *bin*-Verzeichnis der Anwendung oder in den Global Assembly Cache kopieren. Erfreulicherweise ist kein Herunterfahren der IIS notwendig: Veränderungen führen zu einer neuen Anwendungsinstanz in einer neuen Domäne, die dann alle neuen Anfragen bearbeitet, während zum Zeitpunkt der Veränderung laufende Anfragen durch die alte Domäne mit den ursprünglichen Versionen abgearbeitet werden.

HTTP-Handler müssen die Schnittstellen `IHttpHandler` bzw. `IHttpAsyncHandler` implementieren. In deren Methode `ProcessRequest()` bzw. `BeginProcessRequest()` erhalten die HTTP-Handler eine Referenz auf ein Objekt des Typs `HttpContext`, über das sie auf die Anfrage zugreifen können.

HTTP-Module müssen die Schnittstelle `IHttpModule` implementieren, in deren Methode `Init()` sie ebenfalls eine Referenz auf ein Objekt des Typs `HttpContext` erhalten.

21.2.3 Konfiguration der HTTP-Pipeline

Die HTTP-Pipeline konfigurieren Sie über entsprechende Einträge in den Dateien *Machine.config* und *Web.config*; zur Bearbeitung neuer Dateinamenerweiterungen müssen Sie zusätzlich die IIS konfigurieren.

Machine.config ist eine XML-Datei, die für alle Anwendungen auf einem Rechner maßgeblich ist. Diese Datei existiert auf jedem Rechner genau ein-

mal für jede .NET Installation und befindet sich im Verzeichnis *%WinDir%\Microsoft.NET\Framework\<Version>\CONFIG*.

Web.config-Dateien sind genauso aufgebaut wie die *Machine.config*-Datei, befinden sich aber in den Anwendungsverzeichnissen oder den dazugehörigen Unterverzeichnissen. Ihr Geltungsbereich ist das Verzeichnis selbst und alle seine Unterverzeichnisse, wobei die lokale Konfiguration jede globale verdeckt. Insbesondere verdeckt die Konfiguration in einer *Web.config*-Datei eine anderslautende Konfiguration in der Datei *Machine.config*.

Einige Elemente, wie die *add*-Elemente für HTTP-Handler und HTTP-Module, wirken kumulativ. In der Datei *Machine.config* legen Sie also die Grundausstattung an HTTP-Handlern und HTTP-Modulen fest, die Sie für spezielle Anwendungen durch entsprechende Einträge in *Web.config*-Dateien erweitern. Durch *remove*- und *clear*-Elemente können Sie verfügbare HTTP-Handler und HTTP-Module auch wieder entfernen.

HTTP-Handler verfügbar machen

Welcher HTTP-Handler eine Anfrage bearbeiten soll, bestimmen entsprechende Einträge in der *Machine.config*- und in den *Web.config*-Dateien.

Im Folgenden finden Sie einen Auszug aus der *Machine.config*-Datei, der die Einbindung einiger standardmäßig verwendeter HTTP-Handler zeigt (im Original sind es 20).

```
<httpHandlers>
  <add verb="*" path="*.vsdisco"
    type="System.Web.Services.Discovery.DiscoveryRequestHandler,
    System.Web.Services, Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a" validate="false"/>
  <add verb="*" path="*.aspx" type="System.Web.UI.PageHandlerFactory"/>
  <add verb="*" path="*.asmx"
    type="System.Web.Services.Protocols.WebServiceHandlerFactory,
    System.Web.Services, Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a" validate="false"/>
  <add verb="*" path="*.ashx" type="System.Web.UI.SimpleHandlerFactory"/>
  <add verb="*" path="*.cs" type="System.Web.HttpForbiddenHandler"/>
  <add verb="GET,HEAD" path="*" type="System.Web.StaticFileHandler"/>
</httpHandlers>
```

Das erste Element fügt den HTTP-Handler für die *.vsdisco*-Erweiterung hinzu, ist aber normalerweise aus Sicherheitsgründen auskommentiert. Das zweite und dritte Element machen Webanwendungen und Webdienste verfügbar. Im *type*-Attribut wird hier statt der eigentlichen Implementation jeweils eine Factory-Klasse aufgeführt, wodurch die Anfrage durch spezialisierte Typen behandelt werden kann.

Im vierten Element sehen Sie einen speziellen HTTP-Handler. Die hier genannte `SimpleHandlerFactory`-Klasse kompiliert *.ashx*-Dateien, eine spezielle Art von HTTP-Handlern.

Das fünfte Element zeigt, wie Anfragen abgeblockt werden können: Jede Anfrage nach einer Datei mit der Erweiterung *.cs* wird als nicht erlaubt zurückgewiesen. Im *path*-Attribut kann statt einer Erweiterung auch der vollständige Name angegeben werden.

Das sechste Element zeigt schließlich die Funktion des *verb*-Attributs: hier werden die HTTP-Kommandos aufgelistet, für die dieses Element gelten soll.



Zusätzlich zur Einbindung in der Machine.config- und in den Web.config-Dateien müssen Sie die Erweiterung des Dateinamens in den IIS registrieren. Erläuterungen dazu finden Sie im Abschnitt »Dateinamenerweiterung registrieren« auf Seite 877.

Da die `httpHandlers`-Elemente kumulativ wirken, existieren *remove*- und *clear*-Elemente zum Entfernen von HTTP-Handlern.

HTTP-Module verfügbar machen

HTTP-Module werden analog zu den HTTP-Handlern verfügbar gemacht, auch dies zeigt wieder ein Auszug aus der Datei *Machine.config*:

```
<httpModules>
  <add name="OutputCache" type="System.Web.Caching.OutputCacheModule"/>
  <add name="Session" type="System.Web.SessionState.SessionStateModule"/>
  <add name="WindowsAuthentication"
    type="System.Web.Security.WindowsAuthenticationModule"/>
  <add name="FormsAuthentication"
    type="System.Web.Security.FormsAuthenticationModule"/>
  <add name="PassportAuthentication"
    type="System.Web.Security.PassportAuthenticationModule"/>
  <add name="UrlAuthorization"
    type="System.Web.Security.UrlAuthorizationModule"/>
  <add name="FileAuthorization"
    type="System.Web.Security.FileAuthorizationModule"/>
</httpModules>
```

Hier werden die allen Anwendungen zur Verfügung stehenden Caching-, State-, Authentifizierungs- und Authorisierungs-Objekte verfügbar gemacht. Wie bei den HTTP-Handlern können Sie hier Ihre eigenen HTTP-Module für alle Anwendungen anbieten. HTTP-Module, die nur von bestimmten Anwendungen genutzt werden sollen, führen Sie in den entsprechenden *Web.config*-Dateien auf. Auch für HTTP-Module existieren *remove*- und *clear*-Elemente.

Den Arbeitsprozess von ASP.NET konfigurieren

Durch eine sinnvolle Konfiguration des Arbeitsprozesses von ASP.NET können Sie die Stabilität Ihrer Anwendungen erhöhen. Konfigurierbar sind dabei die Bedingungen, unter denen der Prozess neu gestartet wird. Diese Bedingungen kann man unterscheiden – in *reaktive*, wie zum Beispiel eine maximale Länge der Warteschlange oder eine maximale Speicherbelegung, und in *proaktive*, wie zum Beispiel ein automatisches Herunterfahren nach einer bestimmten Zeit. Außerdem können Sie festlegen, welche Vorfälle in das Ereignisprotokoll eingetragen werden sollen.

Die Konfiguration geschieht über die Datei *Machine.config*, und zwar im *processModel*-Element. In dieser Datei finden Sie auch eine ausführliche Beschreibung der Konfigurationsmöglichkeiten.

21.3 Konfiguration der IIS

Obwohl die Konfiguration von ASP.NET fast ausschließlich in den Dateien *Machine.config* und *Web.config* stattfindet, gibt es zwei Szenarien, in denen Sie die IIS konfigurieren müssen: zum Anlegen eines virtuellen Verzeichnisses und zum Registrieren einer Dateinamenerweiterung.

Die Konfiguration der IIS wird in der *IIS Metabase* gespeichert, einer binären Datei. Da Teile dieser Datei mit einem für den jeweiligen Rechner spezifischen Wert verschlüsselt sind, kann man sie nicht auf einen anderen Rechner kopieren. Zur Automatisierung der Konfiguration stehen verschiedene Kommandozeilenwerkzeuge wie *FpSrvAdm.Exe* zur Verfügung. Außerdem finden sich im Verzeichnis *Inetpub\AdminScripts* diverse Skripts für diese Aufgabe.

21.3.1 Dateinamenerweiterung registrieren

Damit die IIS nicht nur statische Inhalte wie HTML-Dateien liefern können und außerdem erweiterbar sind, müssen Sie die Programme kennen, die dynamische Inhalte generieren sollen. Der Typ des Inhalts wird dabei über die Dateinamenerweiterung der Anfrage festgelegt. Einige Erweiterungen wie asp, asmx und aspx sind mit ihren entsprechenden Programmen bereits bei der IIS-Installation registriert worden. Wenn Sie einen HTTP-Handler programmieren, werden Sie oft eine neue Dateinamenerweiterung verwenden wollen, die Sie als Erstes registrieren müssen.

Dazu gehen Sie folgendermaßen vor:

1. Öffnen Sie den Internetdienste-Manager (START/PROGRAMME/VERWALTUNG).
2. Wählen Sie im linken Bereich die Website, für die Sie die Erweiterung registrieren wollen.

3. Klicken Sie mit der rechten Maustaste auf diese Site und wählen Sie den Befehl EIGENSCHAFTEN im Kontextmenü.
4. Klicken Sie in der Registerkarte VERZEICHNIS bzw. BASISVERZEICHNIS auf die Schaltfläche KONFIGURATION.
5. Wählen Sie im Fenster ANWENDUNGSKONFIGURATION die Registerkarte ANWENDUNGSZUORDNUNGEN und klicken Sie auf die Schaltfläche HINZUFÜGEN.
6. Füllen Sie das Fenster HINZUFÜGEN/BEARBEITEN DER ZUORDNUNG VON ANWENDUNGSERWEITERUNGEN aus.

21.3.2 Virtuelle Verzeichnisse anlegen und Anwendungen anmelden

Während die IIS statische Inhalte wie HTML-Dateien über eine entsprechende Pfadangabe in der Anfrage problemlos liefern können, solange sie nur unterhalb des *Inetpub\wwwroot*-Verzeichnisses liegen, muss für dynamische Inhalte ein virtuelles Verzeichnis angelegt werden. Anders ausgedrückt: Eine Anwendung setzt sich aus den Dateien unterhalb ihres virtuellen Wurzelverzeichnisses zusammen.

Wenn Sie eine Anwendung, zum Beispiel einen Webdienst, mit Visual Studio anlegen, richtet die Entwicklungsumgebung das virtuelle Verzeichnis automatisch für Sie ein. Bei manuell angelegten oder auf Ihren Rechner kopierten Anwendungen müssen Sie dagegen in den IIS selbst deren Wurzelverzeichnis zu einem virtuellen Verzeichnis machen und die Anwendungen einzeln anmelden. Dazu gehen Sie folgendermaßen vor:

1. Öffnen Sie den INTERNETDIENSTE-MANAGER (START/PROGRAMME/VERWALTUNG).
2. Wählen Sie im linken Bereich den Ordner, den Sie zu einem virtuellen Verzeichnis machen wollen.
3. Wählen Sie im Kontextmenü des Ordners den Befehl ALLE TASKS/BERECHTIGUNGS-ASSISTENT.
4. Beantworten Sie die Fragen des Assistenten.

Statt der Schritte 2 und 3 können Sie auch im Kontextmenü der Standardwebsite den Befehl NEU und dann im Untermenü den Befehl VIRTUELLES VERZEICHNIS wählen.



Die Beispielprojekte lassen sich nur dann verändern und somit weiterbearbeiten, wenn Sie die Ordner nicht in das Verzeichnis für die Standardwebseite kopieren.

Anstatt eines virtuellen Verzeichnisses können Sie mit einer analogen Prozedur auch eine neue Site anlegen. Setzen Sie in beiden Fällen den Pfad für das Basisverzeichnis auf die lokale Kopie des Ordners wwwroot von der Begleit-CD.

Um eine kopierte Anwendung in einer Site anzumelden, wählen Sie im Kontextmenü des Ordners den Befehl EIGENSCHAFTEN *und klicken im erscheinenden* EIGENSCHAFTEN*-Dialog in der Registerkarte* VERZEICHNIS *auf die Schaltfläche* ERSTELLEN*.*

22 Webanwendungen

Webanwendungen sind Microsofts aktuelle Technologie zur Erstellung dynamischer Benutzeroberflächen auf der Basis von HTML. Dabei gehen Sie genauso vor wie beim Formularentwurf für Windows-Programme: Sie ziehen ein paar Steuerelemente auf ein Formular, geben Initialisierungswerte für Eigenschaften an und unterlegen verschiedene Ereignisse mit Code – fertig. Zur Anfrage eines Webbrowsers generiert ASP.NET dann automatisch HTML, sodass der Entwurf im Browser darstellbar ist. Dabei werden sogar Besonderheiten des Browsers berücksichtigt, beispielsweise, welche HTML-Version er verarbeiten und ob er mit Cascading Stylesheets oder Script-Code umgehen kann. Zusätzlich bieten Webanwendungen bzw. ASP.NET noch Unterstützung beim Caching, beim Halten des Zustands und für andere praktische Dinge.

Vor dem Sprung in praktische Webanwendungen finden Sie im nächsten Abschnitt eine Blitzeinführung in Konzepte wie HTML, CSS, HTML-Formulare und Zustandskonservierung. Wenn Sie damit schon vertraut sind, können Sie gerne weiterblättern.

22.1 Das Web ohne Web Forms

Dieser Abschnitt erläutert zuerst die Grundlagen des HTML-Einsatzes und zeigt dann die Praxis an einigen Beispielen.

22.1.1 Grundlagen von HTML

Die folgenden Unterabschnitte zeichnen in aller Kürze die stürmische Entwicklung von HTML in den letzten zehn Jahren nach.

Statisches HTML

Als Tim Berners-Lee sich Anfang der neunziger Jahre an einem Schweitzer Forschungsinstitut das World Wide Web (WWW, oder kurz Web) ausdachte, wollte er damit Wissenschaftlern den Zugriff auf Dokumente erleichtern. Dabei wurden an thematisch sinnvollen Stellen eines Dokuments Hyperlinks angebracht, über die weitere Dokumente mit vertiefenden Informationen zu diesen Themen erreicht werden konnten. Da bereits

die grundlegende Spezifikation davon ausging, dass sich diese zusätzlichen Dokumente irgendwo in einem Netzwerk befinden würden, ergab sich ein weltweites HYPERTEXTSYSTEM.

Für dieses weltweite System waren einige Standards nötig. Zum einen mussten natürlich die Dokumente ein einheitliches Format haben, dafür wurde HTML (HyperText Markup Language) entwickelt. HTML ist ein einfaches Textformat, das Textteile durch TAGS (Marken) mit Auszeichnungen wie Kursiv- und Fettschrift oder Funktionalität wie Hyperlink oder Aufzählungselement versieht.

Zusätzlich musste ein Transportprotokoll definiert werden, das die beteiligten Rechner verstanden. Dieses Protokoll ist HTTP (HyperText Transfer Protocol). Es ist ebenfalls textbasiert, sehr einfach und dadurch ohne großen Aufwand zu implementieren.

Die physische Verbindung zwischen den Rechnern, das INTERNET, existierte zu diesem Zeitpunkt bereits: Es war in den sechziger Jahren in den USA als militärisches Projekt zur Verbindung verteidigungsrelevanter Forschungsinstitute ins Leben gerufen worden und verband die lokalen Netze der einzelnen Forschungsinstitute. Da das Ganze auch nach einem Atomschlag funktionieren sollte, wurden die Verbindungen als NETZ gestaltet: Selbst beim Ausfall mehrerer Knoten konnten Informationen ihr Ziel noch über die verbliebenen Knoten erreichen. Zur Nutzung des Netzes durch immer mehr Forschungsinstitute und Universitäten wurde 1977 mit TCP/IP (Transmission Control Protocol/Internet Protocol) ein verbessertes, einheitliches Protokoll entwickelt. Details zu TCP/IP finden Sie im Anhang.

Ziel der Erfinder sowohl des Internet als auch des Web waren also einfache, robuste Protokolle. Die gewünschte Funktionalität beschränkte sich auf den Austausch von Text- und Grafikdokumenten – und das Sicherheitskonzept bestand aus dem Ausschluss der Außenwelt.

Neue Möglichkeiten und Notwendigkeiten für Formatierung, Programmierung und Sicherheit von HTML

Als die Nutzerzahl des Web in der zweiten Hälfte der Neunziger förmlich explodierte, sollte plötzlich jedes Pixel auf dem Bildschirm blinken und hupen. Und statt Dokumenten sollten vollständige Programmoberflächen im Browser erscheinen. Außerdem saßen jetzt auch Leute vor dem Bildschirm, die weniger von wissenschaftlich-anthroposophischen Beweggründen geleitet wurden als von profanen Interessen – bis hin zur Langeweile. Einiger dieser Leute konfrontierten das Web mit unvorhergesehenen Sicherheitsproblemen.

Den neuen Anforderungen an Layoutmöglichkeiten begegneten die Browser-Hersteller mit HTML-»Erweiterungen«, also selbstdefinierten Tags, die

außer ihrem eigenen Browser natürlich niemand unterstützte. Der Funktionalität wurde mit Client-Scripting (JavaScript, VBScript) und HTML-Objektmodellen (dHTML, DHTML – ja, das ist ein Unterschied) sowie binären Erweiterungen wie Plug-Ins und ActiveX-Komponenten auf die Sprünge geholfen. Für die Sicherheit wurde der Secure Sockets Layer (SSL) eingeführt, der zusammen mit HTTP das HTTPS (HTTP Secure) zur verschlüsselten Datenübertragung ergibt.

In der evolutionären Auslese der letzten Jahre ist alles außer SSL/HTTPS, JavaScript, dem Plug-In-Konzept und wenigen Java Applets untergegangen. Dafür gibt es diverse neue Standards: Cascading Style Sheets (CSS) und ein aktualisiertes HTML (HTML 4.01 bzw. XHTML 1.0) übernehmen die Formatierung. Dazu kommen die Scalable Vector Graphics (SVG), ein auf XML basierendes Vektorgrafikformat, dessen dynamische und interaktive Möglichkeiten es auch für die Gestaltung von Bedienoberflächen einsetzbar machen. Mit CSS3 und seinen Behaviours sowie dem DOM (Document Object Model) Level 2 HTML steht eine einheitliche Programmierschnittstelle zur Verfügung. Und schließlich unterstützt nun auch Microsoft JavaScript (VBScript gibt es in .NET nicht mehr) und hat es sogar standardisieren lassen (ECMAScript).

Die Referenz zu diesen Standards finden Sie bei `www.w3.org`; `selfhtml.teamone.de` und `www.w3schools.com` liefern gute und umfassende Erläuterungen und Tutorials.

Die Serverseite

Webserver, wie Microsofts Internet Information Services (IIS) oder der Apache (siehe `www.apache.org`), stellen die Infrastruktur insbesondere zur Kommunikation mit Web-Clients zur Verfügung. Ebenso bieten sie Schnittstellen für Programme oder Engines, die das eigentliche Erzeugen von HTML (und Scripts, CSS etc.) übernehmen.

Ursprünglich waren diese Programme auf das so genannte CGI (Common Gateway Interface) zugeschnitten, das nach einem recht simplen Strickmuster arbeitet: Ein CGI-Programm wird vom Webserver als separater Prozess gestartet, bekommt die Parameter einer HTTP-Anfrage per Kommandozeile übergeben und schreibt daraufhin ein HTML-Dokument auf die Festplatte, das der Webserver dann an den Client liefert. Diese Lösung bringt zwar eine maximale Entkopplung von Server und HTML-Engine, lässt aber (vor allem unter Windows NT) hinsichtlich der Performance sehr zu wünschen übrig und führt zu einer kaum wartbaren Vermischung von HTML und Programmlogik.

Das Performance-Problem ging Microsoft mit den ASP (Active Server Pages) an. ASP erlaubt die Integration von Scripts in HTML, die resultierende Datei wird mit der Namenserweiterung *.asp* gespeichert. Die IIS inter-

pretieren bei einer Anfrage die *.asp*-Datei, führen dabei das Script aus und senden Resultat im HTML-Format an den Client. ASP ist einfacher zu programmieren und (oft) schneller und ressourcenschonender als CGI. HTML und Programmlogik sind allerdings auch hier immer noch vermischt.

ASP.NET ist die Weiterentwicklung von ASP: Durch CodeBehind werden HTML und Programmlogik voneinander getrennt, außerdem werden die String-Basteleien durch ein objektorientiertes Programmiermodell ersetzt. Im Gegensatz zu ASP-Seiten, die von oben nach unten abgearbeitet werden und dadurch praktisch von einer einzigen Methode gebildet werden, erfolgt die Programmierung bei ASP.NET ereignisgesteuert und damit in Form mehrerer, auf die jeweilige Aufgabe spezialisierter Methoden. Und schließlich wird der Code nur beim ersten Zugriff (sowie nach jeder Veränderung) kompiliert, nicht bei jeder Anfrage erneut interpretiert – was natürlich der Performance zugute kommt.

22.1.2 HTML in praktischen Beispielen

Die folgenden Abschnitte zeigen anhand kleiner Beispielprogramme die Möglichkeiten und Grenzen der Standards. Alle Dokumente wurden nicht mit Visual Studio, sondern mit dem Windows-Editor Notepad erzeugt, damit das Wesentliche auch zu erkennen ist.

Codebeispiel – Statisches HTML

Der ursprüngliche Verwendungszweck des Web bestand im Zugriff auf statische HTML-Dokumente. Abbildung 22.1 zeigt, wie das verwendete Beispiel-Dokument *HalloWelt.htm* aus dem Ordner *wwwroot\HtmlStatisch* im Internet Explorer (IE) aussieht.

Statische HTML-Dokumente können Sie durch Doppelklick auf die Datei im Explorer öffnen. Wie in der Abbildung 22.1 zu sehen, erscheint bei so aufgerufenen Dokumenten der vollständige Pfad in der Adressleiste des IE. Hierzu ist kein Webserver nötig, die Datei könnte in einem beliebigen Verzeichnis liegen – auch außerhalb von *Inetpub\wwwroot*.

```
<html>
  <head>
    <title>Hallo Welt!</title>
  </head>
  <body>
    <h2>Hallo Welt!</h2>
    <p> Das ist der Text des Beispiels. Obwohl inhaltlich nicht
        gerade gehaltvoll, ist der Text nicht zwecklos: </p>
    <p> Er zeigt, wie Sie Absätze, Zeilenumbrüche <br/> und
      Textauszeichnungen wie <b>Fettschrift</b> kodieren.</p>
    <p>Sogar eine kleine Liste    gibt es:
      <ul>
        <li>_</li>
```



Abbildung 22.1: Anzeige eines statischen HTML-Dokuments

```
      <li>&yen;</li>
      <li>&#036;</li>
    </ul>
  </p>
  <p>Tipp: Für Zeichensätze in HTML gilt dasselbe wie für
    Zeichensätze in XML.</p>
 </body>
</html>
```

HTML-Dokumente sind ähnlich wie XML-Dokumente aufgebaut, man kann aber keine eigenen Tags definieren. Jedes HTML-Dokument muss mit dem Tag *<html>* beginnen und mit dem Tag *</html>* enden. Der auszugebende Text steht zwischen den *<body>*-Tags, wobei die in diesem Bereich verwendeten Tags, wie hier *<h2>*, meistens der Formatierung des Textes dienen. Das ist ein großer Unterschied zu XML, dessen Tags nur die Bedeutung des Inhalts beschreiben, nicht seine Darstellung. Umgekehrt kann man aus den Tags eines HTML-Elements nicht erkennen, welche Bedeutung es hat.

Whitespace wie Leerzeichen und Zeilenumbrüche wird in HTML als Wortgrenze formatiert. Mehrere Leerzeichen werden dabei als ein Leerzeichen interpretiert, der Text demonstriert es mit den Leerzeichen zwischen »Liste« und »gibt«. Ebenso werden Zeilenumbrüche im Text ignoriert und als ein Leerzeichen dargestellt, Sie sehen es im ersten Absatz. Dementsprechend sind spezielle Elemente für Zeilenumbrüche nötig: *
* (break) für den Zeilenumbruch, * * und * * (paragraph) für den Absatz. Ob spezielle Zeichen, wie das Euro-Symbol oder Umlaute, in HTML direkt angegeben werden können, hängt vom Zeichensatz ab. Näheres dazu enthält der Abschnitt »Zeichensätze in XML« auf Seite 753. Die Auswirkungen einer geänderten Zeichensatzinterpretation können Sie über den Befehl CODIERUNG im Kontextmenü des IE testen.

Zwischen den <*head*>-Tags stehen Meta-Informationen, die intern verarbeitet werden, aber nicht in der Ausgabe erscheinen. Solche Meta-Informationen sind zum Beispiel Stichwörter für Internet-Suchmaschinen, Stylesheets oder Scripts. Auch der Titel des Dokuments zwischen den <*title*>-Tags gehört dazu, er erscheint in der Titelleiste des Browsers.

Entsprechend dem Standard XHTML 1.0 finden Sie in diesem Buch nur Element- und Attributnamen in Kleinbuchstaben. Außerdem werden alle Tags geschlossen: Auf ein öffnendes <*xxx*> folgt also grundsätzlich irgendwann ein schließendes </*xxx*>.

Codebeispiel – HTML mit Cascading Style Sheets

Cascading Style Sheets (CSS) verwenden ein ähnliches Prinzip wie Druckformatvorlagen in einer Textverarbeitung wie Microsoft Word: Sie ersetzen Formatierungs-Tags in HTML, um eine Trennung von Inhalt und Darstellung zu erreichen. Die Zentralisierung der Darstellungsregeln ermöglicht auch eine dokumentübergreifende Vereinheitlichung der Darstellung. Außerdem bietet CSS wesentlich weitreichendere Formatierungsmöglichkeiten als die HTML-Erweiterungen einzelner Browser-Hersteller. In Abbildung 22.2 wird das Beispiel-Dokument *HalloWeltCss.htm* durch CSS formatiert, um einen Eindruck von diesen Möglichkeiten zu geben.

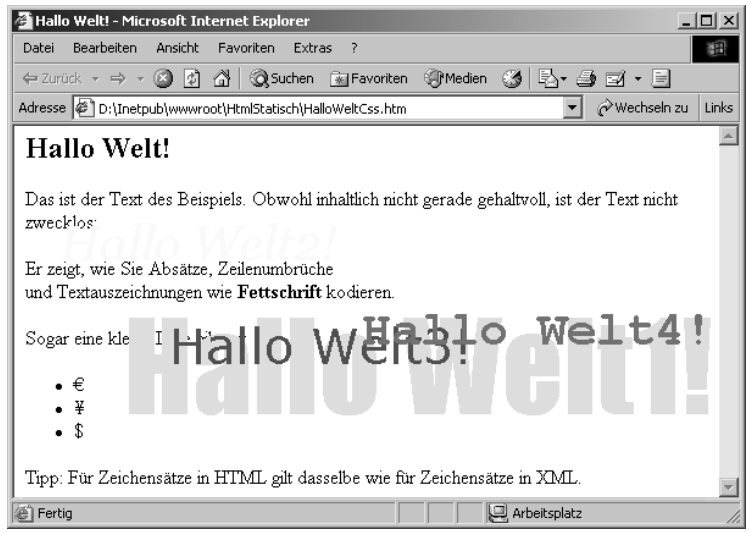


Abbildung 22.2: HTML mit Cascading Style Sheets formatiert

Der Code dafür sieht so aus:

```
<html>
  <head>
    <style>
      #p1 {color: #DDD;
        font: 100px/1 "Impact", "Helvetica Narrow", sans-serif}
```

```
      #p2 {color: #FFC; font: italic 40px/1 "Georgia", serif}
      #p3 {color: #080; font: 40px/1 "Verdana", sans-serif}
      #p4 {color: #37F;
        font: bold 40px/1 "Courier New", "Courier", monospace}

      #p1 {text-align: right;  margin: -170px 0 20px 0}  /* top right */
      #p2 {text-align: left;   margin: -190px 0 130px 5%}  /* top left */
      #p3 {text-align: right;  margin: -87px 35% 47px 0}  /* center */
      #p4 {text-align: right;  margin: -98px 0 58px 0}  /* center right */
    </style>
    <title>Hallo Welt!</title>
  </head>
  <body>
    <h2>Hallo Welt!</h2>
    <p> Das ist der Text des Beispiels. Obwohl inhaltlich nicht
        gerade gehaltvoll, ist der Text nicht zwecklos: </p>
    <p> Er zeigt, wie Sie Absätze, Zeilenumbrüche <br/> und
      Textauszeichnungen wie <b>Fettschrift</b> kodieren.</p>
    <p>Sogar eine kleine Liste    gibt es:
      <ul>
        <li>_</li>
        <li>&yen;</li>
        <li>&#036;</li>
      </ul>
    </p>
    <p>Tipp: Für Zeichensätze in HTML gilt dasselbe wie für
      Zeichensätze in XML.</p>
    <p id="p1">Hallo Welt1!</p>
    <p id="p2">Hallo Welt2!</p>
    <p id="p3">Hallo Welt3!</p>
    <p id="p4">Hallo Welt4!</p>
  </body>
</html>
```

Während die Absätze im HTML-Dokument brav untereinander stehen, tauschen sie im IE ihre Position und passen sich sogar Veränderungen der Fenstergröße an. Das sind allerdings die höheren Weihen des Einsatzes von CSS, und bevor Sie in Ehrfurcht erstarren: Das gezeigte Stylesheet ist, auf gut Deutsch gesagt, aus `www.w3.org/Style` bzw. `www.w3.org/Style/map.css` geklaut.

Wenn Sie sich den Quelltext im IE ansehen (Menü ANSICHT, Befehl QUELLTEXT), erscheint das Dokument nicht anders als in Notepad. Das formatierte Dokument existiert also nur im Speicher des IE.

Die Gestaltungsregeln dieses Beispiels sind durch ein *style*-Element im Kopf des Dokuments eingebettet. Jede Regel besteht dabei aus einem SELEKTOR, dem in geschweiften Klammern eine Liste von EIGENSCHAFTEN und WERTEN folgt.

Der Selektor bestimmt, auf welche Elemente die Regel angewendet werden soll. Dabei kann eine Regel auf beliebig viele Elemente angewendet werden, und beliebig viele Regeln können auf ein Element angewendet werden. Bei

Konflikten hat die spezielle Regel den Vorrang vor der allgemeinen. Aus dieser Möglichkeit der Überlagerung von Regeln kommt der Begriff CASCADING in CSS.

In diesem Beispiel dient das *id*-Attribut als Selektor, er hat deshalb die Form #<*id-Wert*>. Meistens werden aber Elemente oder Klassen als Selektoren verwendet, beispielsweise:

```
body {background: #fffff0; font-family: Verdana, Arial, sans-serif;}
.Tip {border-left: 0.3em solid red; padding-left: 0.5em;}
```

Im Dokument existiert dabei (sinnvollerweise) mindestens ein Element der Klasse *Tip*, zum Beispiel:

```
<span class="Tip">Tip: Jetzt nur nicht die Geduld verlieren!</span>
```

Wenn die Regel für Elemente der Klasse *Tip* eine Hintergrundfarbe festlegen würde, hätte diese Priorität vor der für *body*-Elemente allgemein festgelegten Hintergrundfarbe.

Eingebettete CSS verwenden Sie, wenn sie nur auf den Inhalt eines Dokuments angewendet werden sollen. Zur einheitlichen Formatierung mehrerer Dokumente werden externe Style Sheets benutzt – Dateien mit der Endung .CSS, die ausschließlich (und in genau derselben Formulierung wie bei der eingebetteten Variante) Regeln enthalten. In den zu formatierenden Dateien wird das *style*-Element durch einen entsprechenden Link ersetzt, beispielsweise:

```
<link rel="stylesheet" type="text/css" href="Default.Css"/>
```

Wenn wie im ursprünglichen Beispiel jede Regel ohnehin nur auf ein Element angewendet wird, kann sie auch als so genannter INLINE STYLE im Element selbst untergebracht werden. Zum Beispiel:

```
<p style="text-align: right; margin: -170px 0 20px 0; color: #DDD;
font: 100px/1 'Impact', 'Helvetica Narrow', sans-serif">Hallo Welt1!</p>
```

Die Regeln werden also einfach einem `style`-Attribut zugewiesen, bis auf die einfachen Anführungszeichen bleibt ihr Inhalt gleich. Visual Studio verwendet diese Technik. Auch hier hat wieder das Spezielle den Vorrang vor dem Allgemeinen, also *inline* vor *eingebettet* vor *extern* (vgl. *HalloWeltCssInline.htm*).

Welches Element welche Eigenschaften unterstützt und welche Werte dafür in welcher Notation und Reihenfolge zulässig sind, ist ein Kapitel für sich. Auch der genaue Aufbau von Selektoren und die Prioritäts-Regeln sind wesentlich komplexer als hier dargestellt. Mehr Informationen dazu finden Sie unter den Links am Kapitelanfang.

Es soll nicht verschwiegen werden, dass ältere Browser mit CSS erhebliche Probleme haben. Wie bei vielen anderen Standards beginnt eine brauchbare Unterstützung für CSS erst mit den 5er Versionen von IE und Netscape Navigator. Aber selbst die aktuellen Browserversionen unterstützen noch nicht alle Details des CSS2-Standards.

Codebeispiel – Dynamisches HTML

Dynamisches HTML wird erst zum Zeitpunkt der Anfrage erzeugt. Dazu muss der WWW-Publishing-Dienst der IIS laufen, und die ASP- bzw. ASP.NET-Datei muss in einem virtuellen Verzeichnis liegen. Wie Sie ein virtuelles Verzeichnis anlegen, beschreibt der Abschnitt »Virtuelle Verzeichnisse anlegen« auf Seite 878.

Die Variation des Beispiels in Abbildung 22.3 zeigt, wie Sie Scripts in ASP- bzw. ASP.NET-Seiten integrieren. In der Adressleiste ist zu sehen, dass hier nicht über das Dateisystem zugegriffen wird, sondern über den Webserver.



Abbildung 22.3: Anzeige eines dynamischen HTML-Dokuments

Dieses Beispiel ist eine in in VBScript kodierte ASP-Seite, wie das folgende Listing zeigt. Alles in den <%%>-Tags wird dabei als Script interpretiert und ausgeführt, dabei schreibt die *Write()*-Methode des *Response*-Objekts in den Ausgabe-Stream. Zum Ermitteln und Formatieren von Datum und Zeit werden verschiedene VBScript-Funktionen benutzt. Das geschieht übrigens ohne Ausnahme alles auf dem Server – im Browser kommt ausschließlich reines HTML an.

```
<html>
  <head>
    <title>Hallo Welt!</title>
  </head>
  <body>
    <h2>Hallo Welt!</h2>
    <p>Heute ist
      <%response.write(FormatDateTime(date(), vblongdate))%>.<br/>
      Es ist <%response.write(time())%> Uhr.</p>
  </body>
</html>
```

Und was passiert, wenn Sie die Datei unter der Erweiterung *aspx* speichern, um ASP.NET zu nutzen? Wahrscheinlich erst einmal gar nichts, weil der Prozess *AspNet_Wp.Exe* noch nicht gestartet worden ist. Und wenn man das nachholt, erscheint eine Fehlermeldung, denn VBScript wird unter ASP.NET nicht mehr unterstützt.

Das folgende Listing zeigt den funktionierenden Code für ASP.NET. Das Ergebnis im IE ist gleich, allerdings wird hier VB.NET anstelle von VBScript benutzt, und der Code wird kompiliert statt interpretiert. VB.NET ist die Standardsprache von ASP.NET und wird verwendet, wenn nichts anderes angegeben ist.

```
<html>
  <head>
    <title>Hallo Welt!</title>
  </head>
  <body>
    <h2>Hallo Welt!</h2>
    <p>Heute ist <%response.write(now().ToLongDateString())%>.<br/>
      Es ist <%response.write(now().ToLongTimeString())%> Uhr.</p>
  </body>
</html>
```

In C# sieht das Ganze so aus:

```
<%@ Page Language="CS"%>
<html>
  <head>
    <title>Hallo Welt!</title>
  </head>
  <body>
    <h2>Hallo Welt!</h2>
    <p>Heute ist <%Response.Write(DateTime.Now.ToLongDateString());%>.<br/>
      Es ist <%Response.Write(DateTime.Now.ToLongTimeString());%> Uhr.</p>
  </body>
</html>
```

Das Ergebnis im IE ist gleich, man kann aber schon auf den ersten Blick die syntaktischen Unterschiede zwischen VB.NET und C# erkennen. Entscheidend für die Verwendung von C# unter ASP.NET ist das *Language*-Attribut in der *Page*-Anweisung der ersten Zeile.



Im Gegensatz zu ASP erlaubt ASP.NET kein Mischen mehrerer Programmiersprachen auf einer Seite.

Bei allen drei Variationen des Beispiels lässt sich leicht erkennen, welches Durcheinander die Vermischung von Code und HTML verursacht. An die Zustände in Dateien mit realistischem Inhalt (und entsprechender Größe) mag man da gar nicht denken. Deshalb benutzen Sie in ASP.NET normaler-

weise *CodeBehind*. Dazu später mehr, denn der nächste Abschnitt beschäftigt sich erst einmal mit einer weiteren wichtigen Anwendung von HTML: den Formularen.

Codebeispiel – Formulare in HTML

Durch Formulare kann man in HTML so etwas wie eine Programmoberfläche gestalten. Die Gestaltungsmöglichkeiten sind zwar nicht mit denen von Windows-Programmen vergleichbar, für eine einfache Interaktion mit dem Benutzer genügen sie aber. In der Beschränkung auf das Wesentliche liegt auch der Vorteil von HTML-Formularen: Sie werden mit wenigen Tags beschrieben, sind deshalb plattformunabhängig und lassen sich leichter und sicherer übertragen als EXE- oder sonstige Binär-Dateien.

Dieser Abschnitt zeigt nur die grundlegende Funktionsweise von HTML-Formularen. Mehr Informationen dazu finden Sie unter den Links am Kapitelanfang.

Die hier vorgestellte Beispielanwendung *HtmlForm* besteht aus drei Formularen und einer CSS-Datei. Das erste Formular, *Anmeldung.aspx*, implementiert eine Benutzeranmeldung, verwendet Script-Code zur Validierung der Eingaben und demonstriert die Probleme der Zustandslosigkeit. Das zweite Formular, *AnmeldungCli.aspx*, zeigt die Positionierung von HTML-Steuerelementen, das dritte, *Resultat.aspx*, der Vollständigkeit halber eine Zusammenfassung der Eingaben.

Alle drei Formulare werden zuerst im althergebrachten ASP-Stil implementiert, um syntaktischen Verwirrungen vorzubeugen, sind sie aber als gültige ASP.NET-Dateien kodiert. An dieser Implementation werden Probleme und Beschränkungen des althergebrachten Stils deutlich. Die Anwendung wird dann im nächsten Abschnitt noch einmal mit dem Visual Studio im neuen ASP.NET-Stil implementiert und zeigt die dort vorhandenen Lösungsmöglichkeiten.

Das Anmeldeformular

Abbildung 22.4 zeigt das Anmeldeformular.

Hier der Code des Anmeldeformulars *Anmeldung.aspx*:

```
<%@ Page language="C#"%>
<html>
  <head>
    <title>Anmeldeformular</title>
    <link rel="stylesheet" type="text/css" href="Default.Css"/>
    <script runat="server">
      public void PrüfeEingabe()
      {
```

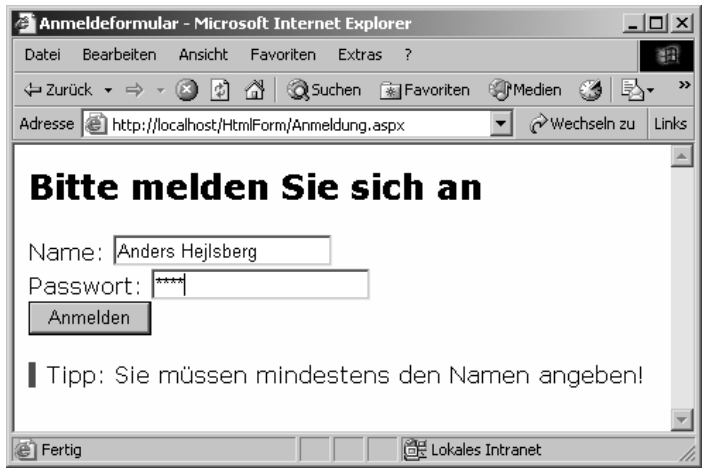


Abbildung 22.4: Das Anmeldeformular im IE

```
        if ((Request.QueryString["txtName"] != null) &&
          (Request.QueryString["txtName"] != ""))
          this.Response.Redirect("Auswahl.aspx?txtName=" +
            Request.QueryString["txtName"]);
      }
    </script>
  </head>
  <body>
    <% PrüfeEingabe(); %>
    <h2>Bitte melden Sie sich an</h2>
    <form method="get" action="Anmeldung.aspx">
      Name: <input type="text" name="txtName"><br>
      Passwort: <input type="password" name="txtPassword"><br>
      <input type="submit" value="Anmelden">
    </form>
    <span class="Tip">Tipp: Sie müssen mindestens den Namen angeben!</span>
  </body>
</html>
```

Ein HTML-Formular entsteht durch ein *form*-Element, jede HTML-Seite (ohne Frames) kann nur ein Formular enthalten. Das *action*-Attribut des <*form*>-Tags legt fest, was beim Senden der Formulardaten geschehen soll. Im Allgemeinen wird hier eine URL angegeben, an die die Daten gesendet werden sollen. In diesem Beispiel werden die Formulardaten an das Formular selbst gesendet, auf den ersten Blick dürfte das unverständlich erscheinen – es macht aber Sinn, denn das Formular enthält ja auch den Code zu seiner Validierung. Dazu später mehr.

Das *method*-Attribut des <*form*>-Tags enthält die Werte *post* oder *get*. Sie bestimmen, wie die Formulardaten an den Server übermittelt werden. Beim *get* werden die Daten an die URL der Nachricht an den Server gehängt, in Abbildung 22.5 ist das zu sehen. Nachteilig an dieser Methode ist die Sicht-

barkeit der Daten und die Begrenzung ihrer Länge. Beim *post* werden die Formulardaten im Rumpf der Nachricht an den Server übertragen.

Innerhalb des <*form*>-Tags können HTML-Steuerelemente eingesetzt werden, in diesem Fall sind das zwei Textfelder und eine Schaltfläche. Entscheidend für die Art der Steuerelemente sind neben dem Tag auch die Werte der *type*-Attribute: *text* erzeugt ein einzeiliges Textfeld, *password* ein Textfeld, das Eingaben nur als Sternchen zeigt. *submit* erzeugt schließlich eine Schaltfläche, deren Betätigung die Formulardaten absendet.

Kommen wir nun zum Funktionsablauf und damit zum Skript. Beim Verarbeiten der Anfrage trifft der Server auf die Zeile <% *PrüfeEingabe();* %>, wobei er den Text zwischen den %-Zeichen entsprechend der Direktive in der ersten Zeile des Dokuments als C#-Code auswertet. Das sieht zwar genauso aus wie unter ASP, hier wird aber nichts interpretiert, sondern beim ersten Zugriff compiliert. Außerdem machen Semikolon und Klammern klar, dass es sich hier um C# handelt, nicht um VBScript. Übrigens liefert der Server das serverseitige Skript nicht an den Client, wie ein kurzer Blick in den Quelltext des Dokuments im IE zeigt.

Unter ASP.NET müssen Methoden in einem script-Element definiert werden. Eine Definition im <%%>-Block wie unter ASP ist nicht mehr zulässig.



Der Server führt dann die Methode `PrüfeEingabe()` aus, die innerhalb der <*script*>-Tags im Kopf des Dokuments definiert ist. Entscheidend ist hier der Wert *server* des *runat*-Attributs. Ohne dieses Attribut würde der Server den Code für ein clientseitiges Script halten und ihn übertragen, statt ihn auszuführen. `PrüfeEingabe()` prüft Vorhandensein und Inhalt des Parameters *txtEingabe*, in dem der Inhalt des entsprechenden Textfelds übertragen wird. Bei der ersten Anfrage schlägt die Prüfung fehl, denn das Formular wird ohne Formulardaten in der URL aufgerufen (siehe Abbildung 22.4). Die Methode `PrüfeEingabe()` tut dann nichts weiter, und das Formular wird an den Client gesendet.

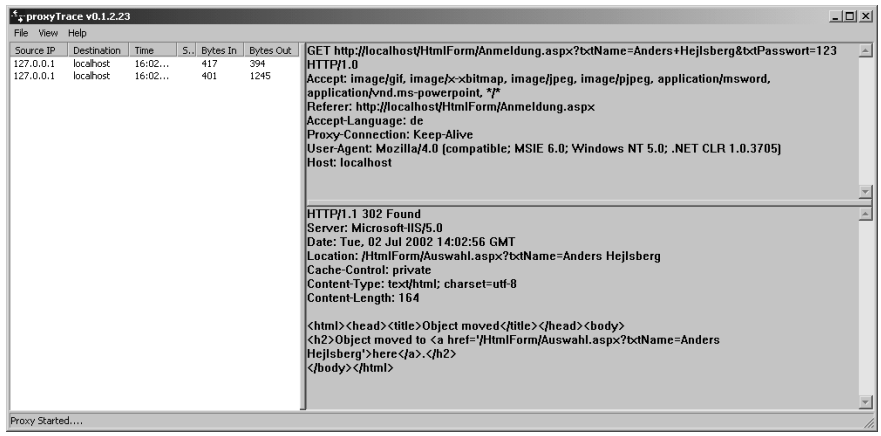
Wenn der Benutzer jetzt das Formular ausfüllt und auf ANMELDEN klickt, wird der Inhalt der beiden Textfelder übertragen. `PrüfeEingabe()` führt dann die Anweisung

```
this.Response.Redirect(
  "Auswahl.aspx?txtName=" + Request.QueryString["txtName"]);
```

aus. Dadurch wird der Benutzer zum zweiten Formular, *Auswahl.aspx*, umgeleitet. Außerdem wird diesem Formular der Benutzername als Parameter übergeben. Mit dem Programm *ProxyTrace* kann man das sehr schön

sehen (Informationen zum Einsatz von ProxyTrace finden Sie im Abschnitt »Debugging von Webdiensten« auf Seite 952.)

Abbildung 22.5: Die übergebenen Parameter und der Redirect in ProxyTrace



Der Browser fordert nach der Umleitung automatisch das neue Dokument an, das ist der zweite Eintrag im linken Fensterbereich der Abbildung 22.5.

Wenn der Benutzer das Feld *Name* leer gelassen hat, wird er nicht umgeleitet, landet also wieder im Anmeldeformular. Allerdings ist dabei auch das vorher eventuell ausgefüllte *Passwort*-Feld leer. Dessen Wert müsste der serverseitige Code explizit in das neu zu sendende Formular eintragen. Dass so etwas bei umfangreicheren Formularen schnell aufwändig wird, ist ein grundlegendes Problem der Formular-Programmierung in HTML. Unter ASP.NET steht Ihnen dazu als Lösung der Anzeigestatus zur Verfügung.

Das Anmeldeformular mit clientseitigem Skript

Allerdings ist es nicht unbedingt geschickt, die Validierung auf dem Server durchzuführen. Der Client könnte selbst validieren und damit den Server entlasten sowie dem Benutzer unnötige Wartezeiten ersparen. Dazu setzt man clientseitiges Skripting ein (vgl. *AnmeldungCli.aspx*).

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="Default.Css"/>
    <title>Anmeldeformular</title>
    <script language="JavaScript">
      function PrüfeEingabe()
      {
        return document.frmAnmeldung.txtName.value != "";
      }
    </script>
  </head>
```

```
  <body>
    <h2>Bitte melden Sie sich an</h2>
    <form method="get" name="frmAnmeldung"
      action="auswahl.aspx" onsubmit="return PrüfeEingabe()">
      Name:  <input type="text" name="txtName"><br>
      Passwort: <input type="password" name="txtPasswort"><br>
      <input type="submit" value="Anmelden">
    </form>
    <span class="Tip">Tipp: Sie müssen mindestens den Namen angeben!</span>
  </body>
</html>
```

Hier fällt zunächst einmal das veränderte *script*-Element auf, das hier ein *language*-Attribut mit dem Wert `"JavaScript"` hat; außerdem fehlt das *runat*-Attribut. Hier ist also weder C# drin, noch wird der Code auf dem Server ausgeführt. Die Methode *PrüfeEingabe()* liefert *false* zurück, wenn das Textfeld *Name* leer ist.

Im *<form>*-Tag steht jetzt ein *name*-Attribut, über dessen Wert im oben erläuterten Skript auf das Formular zugegriffen wird. Außerdem ist ein *onsubmit*-Attribut hinzugekommen, das die Methode *PrüfeEingabe()* aufruft. Wenn sie *true* zurückliefert, wird das im *action*-Attribut genannte Formular aufgerufen. Dabei werden in der URL automatisch die Werte aus den Textfeldern übergeben.

Nachteilig ist bei clientseitigem Skript allerdings, dass der Benutzer den Code über den Befehl QUELLTEXT ANZEIGEN im Kontextmenü des IE anschauen kann.

Das Stylesheet

Vor der Besprechung des zweiten Formulars noch kurz ein Blick in das Stylesheet *default.css*. Sie binden es mit der Zeile `"<link rel="stylesheet" type="text/css" href="Default.Css"/>"` im *head*-Element der HTML-Datei ein; es besteht lediglich aus zwei Zeilen:

```
body {background: #fffff0; font-family: Verdana, Arial, sans-serif;}
.Tip {border-left: 0.3em solid red; padding-left: 0.5em;}
```

Die erste Zeile sorgt für eine angenehme Hintergrundfarbe und eine am Bildschirm gut lesbare Schrift. Die zweite Zeile formatiert die im **-Tag enthaltene letzte Zeile des HTML-Dokuments (über ihr *class*-Attribut).

Es wäre schön, wenn man den Text »Tipp: ...«, der auch in den beiden anderen Formularen erscheinen soll, ebenfalls in der Regel angeben könnte. Das ist in CSS2 auch so vorgesehen, wird aber selbst von der neuesten Version des IE nicht unterstützt.

Das Auswahlformular

Das zweite Formular zeigt neben dem (mühsamen) Einsatz von HTML-Steuerelementen auch deren althergebrachte Positionierung durch eine Tabelle. In Abbildung 22.6 erkennen Sie neben der schlichten Eleganz des Formulars auch, wie Parameter zur Übertragung kodiert werden: In der Adresszeile des IE sehen Sie, dass zum Beispiel Leerzeichen durch ein *%20* ersetzt werden, entsprechend ihrem hexadezimalen ASCII-Wert. Das Ganze nennt man auch »URL kodiert«.

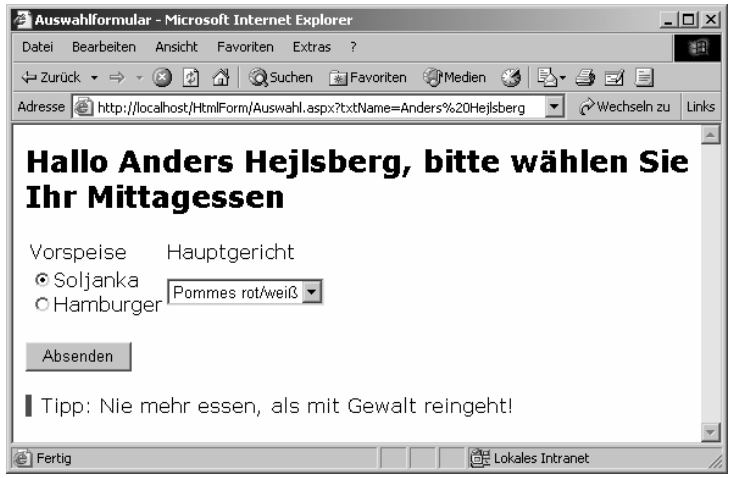


Abbildung 22.6: Einsatz und Positionierung von HTML-Steuerelementen

Der Code für das Auswahlformular *Auswahl.aspx* sieht wie folgt aus:

```
<%@ Page Language="C#"%>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="Default.Css"/>
    <title>Auswahlformular</title>
  </head>
  <body>
    <h2>Hallo <% Response.Write(Request.QueryString["txtName"]); %>,
      bitte wählen Sie Ihr Mittagessen</h2>
    <form method="get" action="Resultat.aspx">
      <table>
        <tr><td>Vorspeise</td><td>Hauptgericht</td></tr>
        <tr>
          <td>
            <input type="radio" name="optVorspeise"
              value="Soljanka" checked="checked"/>Soljanka<br/>
            <input type="radio" name="optVorspeise"
              value="Hamburger"/>Hamburger
          </td>
          <td>
            <select name="lstHauptgericht">
              <option>Pommes rot/weiß</option>
```

```
            <option>Hamburger</option>
          </select>
        </td>
      </tr>
    </table>
    <input type="hidden" name="txtName"
      value="<% Response.Write(Request.QueryString["txtName"]); %>"/><br>
    <input type="submit" value="Absenden"/>
  </form>
  <span class="Tip">Tip: Nie mehr essen, als mit Gewalt reingeht!</span>
 </body>
</html>
```

Das Auswahlformular bindet dasselbe Stylesheet ein wie das Anmeldeformular: Das stellt ein einheitliches Erscheinungsbild sicher und vereinfacht die Wartung.

Ein *script*-Element existiert hier nicht, aber in den <%%>-Blöcken wird per Skript zweimal der Name des Benutzers ausgegeben. Dieser Name wurde dem Formular beim Aufruf als Parameter in der URL übergeben, Abbildung 22.6 zeigt es. Auf diese in der URL übergebenen Werte greifen Sie mit der `QueryString`-Auflistung des `Request`-Objekts zu. Dieses Objekt vom Typ `HttpRequest` kapselt Informationen zur Anfrage, beispielsweise `Browser`, `Cookies`, `UrlReferrer`, `UserHostAdress`, aber auch zur Anwendung, wie `PhysicalPath` oder `ServerVariables`.

Die Ausgabe erledigt die `Write()`-Methode des `Response`-Objekts. Dieses Objekt ist vom Typ `HttpResponse` und kapselt die Ausgabe. Das umfasst neben dem Schreiben auch das Setzen von Cookies (`Add()`-Methode der `Cookies`-Auflistung) oder Headern (`AddHeader()`, `AppendHeader()`) sowie das Setzen des Status (zum Beispiel mit der `StatusDescription`-Eigenschaft). Außerdem ermöglicht das `Response`-Objekt durch verschiedene Eigenschaften und Methoden die Verwaltung des Cachings.

Die erste Ausgabe schreibt den Namen des Benutzers in die Begrüßung, die zweite trägt ihn als Wert eines versteckten *input*-Steuerelements ein. Das ist die klassische Art, bei der Programmierung von HTML-Formularen Zustandsinformationen zu speichern. So spart man sich die aufwändige Verwaltung des Zustands am Server, die außerdem mit steigender Benutzerzahl zu Ressourcenproblemen führt. Nachteilig ist dabei natürlich wieder, dass der interessierte Benutzer die Werte im Quelltext sieht – und bei größeren Datenmengen zwangsläufig die Übertragungsdauer steigt. (Nötig ist die Speicherung – und Rückgabe – des Benutzernamens, weil auch das dritte Formular eine personalisierte Begrüßung implementieren soll.)

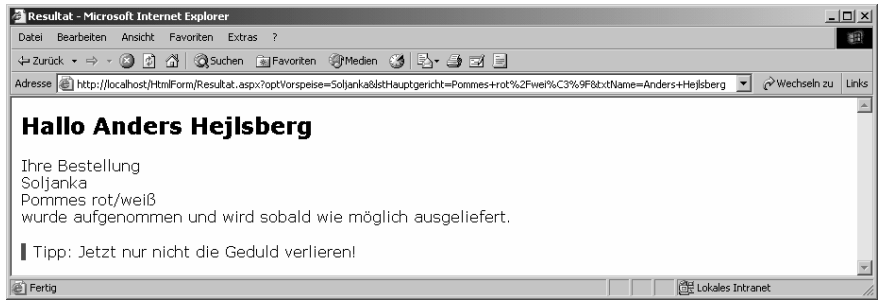
Die sichtbaren Steuerelemente dieses Auswahlformulars ermöglichen die Zusammenstellung eines Essens für den robusteren Magen. Die Positionierung der Steuerelemente durch eine Tabelle schafft ein aufgeräumtes Design und ist das Standardverfahren in der klassischen HTML-Programmierung.

Beim Zurücksenden des Formulars an den Server wird das Dokument *Resultat.aspx* angefordert. (Diese Anforderung geschieht über das *action*-Attribut des *form*-Elements.) Dabei wird die URL wieder um einige Parameter erweitert, das bestimmt der Wert *get* im *method*-Attribut des *form*-Elements. Übertragen werden dabei die Werte der *value*-Attribute, sowohl aus dem versteckten *input*-Element als auch aus der Auswahl des Benutzers.

Das Resultatformular

Das Resultatformular zeigt die Kellner-übliche Reaktion auf eine Bestellung: ihre Wiederholung und die Vertröstung auf später (Abbildung 22.7).

Abbildung 22.7: Das Resultatformular



Der Code für das Resultatformular (*Resultat.aspx*) ist ähnlich dem des Auswahlformulars.

```
<%@ Page Language="C#"%>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="Default.Css"/>
    <title>Resultat</title>
  </head>
  <body>
    <h2>Hallo <% Response.Write(Request.QueryString["txtName"]); %></h2>
    <p>Ihre Bestellung<br/>
    <% Response.Write(Request.QueryString["optVorspeise"] + "<br/>" +
      Request.QueryString["lstHauptgericht"] + "<br/>"); %>
    wurde aufgenommen und wird sobald wie möglich ausgeliefert.</p>
    <span class="Tip">Tip: Jetzt nur nicht die Geduld verlieren!</span>
  </body>
</html>
```

Auch das Resultatformular schreibt die Werte seiner Aufrufparameter wieder in die Ausgabe, diesmal allerdings mit zusätzlichen HTML-Tags.

Interessant ist an diesem Mini-Projekt zum einen die Einfachheit der verwendeten Lösungsmuster, zum anderen leider auch das absehbare Wartungsproblem durch die Vermischung von Oberflächen- und funktionalem Code.

Zum Vergleich mit den folgenden Implementationen in den Abschnitten »Web Forms-Seiten mit HTML-Serversteuerelementen« und »Web Forms-Seiten mit Webserver-Steuerelementen« (Seite 899 bzw. Seite 910) ist auch noch festzuhalten, dass bei der gesamten Kommunikation zwischen Client und Server gerade einmal 5000 Bytes durch die Leitung wanderten. Das lässt sich mit dem Netzwerkmonitor oder einem Tool wie ProxyTrace (siehe Abschnitt »Debugging von Webdiensten« auf Seite 952) leicht feststellen.

22.2 Web Forms-Seiten mit HTML-Serversteuerelementen

Mit den HTML-Serversteuerelementen bietet ASP.NET eine neue Möglichkeit zur Implementation von HTML-Oberflächen. Design und Programmierung der Oberfläche erfolgen dabei wie von Windows-Anwendungen gewohnt in der IDE von Visual Studio: Sie ziehen ein Steuerelement aus der TOOLBOX auf ein Formular, stellen Eigenschaften für Steuerelement und Formular im EIGENSCHAFTEN-Fenster ein und bearbeiten Ereignisse von Steuerelement und Formular im Code-Editor. HTML-Serversteuerelemente finden Sie auf der Registerkarte HTML der TOOLBOX, das Formular und die Steuerelemente können Sie sowohl in der Entwurfs- als auch in der HTML-Ansicht ansehen und bearbeiten.

HTML-Serversteuerelemente kapseln die bekannten HTML-Tags, also neben Steuerelement-Tags wie <*input*> oder <*select*> auch allgemeine HTML-Tags wie <*div*> oder <*span*> sowie ganze Tabellen. Neu ist, dass die IDE relativ komfortable Bearbeitungsmöglichkeiten für diese Tags und ihre Attribute sowie für die Programmierung bietet. Trotzdem sehen Sie im HTML-Designer genau das HTML, das später an den Client geschickt wird.

Weil der Quelltext von Web Forms-Seiten mit HTML-Serversteuerelementen dem von HTML-Formularen, die mit ASP programmiert sind, recht ähnlich ist, eignet sich diese Art von ASP.NET-Projekt auch zur Migration von ASP-Projekten. Allerdings sollte man entsprechende Tests nur mit Kopien der ASP-Projekte durchführen, denn ASP.NET respektiert vorhandenen Quelltext so wenig, wie man das von Microsoft-Werkzeugen gewohnt ist.

Die automatische Formatierung des Quelltextes können Sie über die Optionen unter EXTRAS/OPTIONEN/TEXT-EDITOR/HTML/XML/FORMAT einschränken. Manches lässt sich aber trotzdem nicht abschalten: Beispielsweise fügt Visual Studio grundsätzlich ein selected in das erste <option>-Element eines HTML-Listenfeldes ein.





Um die in diesem Kapitel vorgestellten Beispielprojekte schrittweise nachzustellen, legen Sie jeweils ein neues Projekt vom Typ ASP.NET-Webanwendung an. Falls Sie die fertigen Projekte von der Begleit-CD in eine Website kopiert haben, folgen Sie der Anleitung im Abschnitt »Virtuelle Verzeichnisse anlegen und Anwendungen anmelden« (Seite 878), um sie in Visual Studio .NET ausführen und weiterbearbeiten zu können.

Codebeispiel – Web Forms-Seiten mit HTML-Serversteuerelementen

Das Beispielprojekt *WfFormHtml* implementiert noch einmal dieselbe Funktionalität, die Sie schon aus dem Abschnitt »Codebeispiel – Formulare in HTML« von Seite 891 kennen: eine aus drei Formularen bestehende Anwendung zum Zusammenstellen eines Mittagessens.

Das Anmeldeformular

Zur Demonstration der Möglichkeiten von Visual Studio wird das Aussehen des Anmeldeformulars nicht durch ein Stylesheet festgelegt, sondern durch die entsprechenden Einstellungsmöglichkeiten der IDE. Abbildung 22.8 zeigt das Anmeldeformular im Betrieb, wobei gleich das im Vergleich zur Erstimplementation angenehm aufgeräumte Äußere auffällt.



Abbildung 22.8: Das Anmeldeformular mit HTML-Serversteuerelementen

Ziehen Sie für das Anmeldeformular folgende Steuerelemente von der Registerkarte HTML der TOOLBOX auf das Formular: drei `Label`-Steuerelemente (die Überschrift wird im HTML-Code erstellt), ein `Submit Button`-Steuerelemlent, ein `Text Field`-Steuerelement und ein `Password Field`-Steuerelement. Achtung: Beim Aufklappen der TOOLBOX erscheint erst die Registerkarte WEB FORMS, Sie müssen erst zur Registerkarte HTML wechseln.

Da die einzelnen Eigenschaften auf sehr unterschiedliche Arten eingestellt werden, finden Sie hier statt einer Tabelle mit Eigenschaftswerten eine detaillierte Beschreibung des Vorgehens.

So stellen Sie die Eigenschaften des Formulars ein: Seinen Titel geben Sie in die *title*-Eigenschaft ein (wird automatisch aus dem Dateinamen generiert), seine Hintergrundfarbe legen Sie mit der *bgColor*-Eigenschaft fest. Dazu klicken Sie im EIGENSCHAFTEN-Fenster auf die kleine, durch drei Punkte gekennzeichnete Schaltfläche und wählen im Fenster FARBAUSWAHL die Registerkarte BENANNTE FARBEN. Im Bereich WEITERE wählen Sie dann in der dritten Spalte von rechts die vierte Farbe von oben (*Ivory*).

Da die Überschrift in diesem Formular nicht verändert wird, ist für sie kein `Label`-Steuerelement nötig. Stattdessen gehen Sie in die HTML-Ansicht und legen im *body*-Element ein *h1*-Element mit der Überschrift an – Dank IntelliSense muss man hier nicht alle Tags manuell eingeben. Sie können jede Web Forms-Seite sowohl in der Entwurfs- als auch in der HTML-Ansicht bearbeiten.

Den Text in den drei `Label`-Steuerelementen können Sie nach Anklicken direkt eingeben, das jeweilige Steuerelement wird dabei schraffiert umrahmt.

Die Formatierung des untersten `Label`-Steuerelements stellen Sie über seine *style*-Eigenschaft ein. Klicken Sie dazu im EIGENSCHAFTEN-Fenster auf die kleine, durch drei Punkte gekennzeichnete Schaltfläche und wählen Sie im linken Bereich des Fensters STIL-GENERATOR die Option KONTUREN. Wählen Sie dort im Bereich RAHMEN als zu ändernde Kante *Links* aus, als Stil *Durchgehende Linie*, als Breite *Mittel* und als Farbe *Red*. Tragen Sie jetzt noch im Bereich ABSTAND in der Zeile LINKS den Wert *0.5* ein und wählen Sie als Einheit *em*.



Bei mehrmaligem Ändern dieser Formatierung erscheint der rote Streifen eventuell zweimal. In der HTML-Ansicht sieht man dann, dass Visual Studio die Formatierung einmal in das span- und einmal in das umgebende div-Element eingefügt hat. Entfernen Sie die entsprechenden Attribute aus dem span-Element.

Benennen Sie das `Text Field`- und das `Password Field`-Steuerelement durch Eintragen der Werte *txtName* und *txtPasswort* in die jeweilige *id*-Eigenschaft, die Beschriftung der Schaltfläche tragen Sie in ihre *value*-Eigenschaft ein.

Wenn Sie jetzt wie bei Windows-Formularen üblich auf die Schaltfläche doppelklicken, um den Validierungscode einzugeben, erhalten Sie eine Fehlermeldung: Um serverbasierten Code eingeben zu können, müssen Sie das

Steuerelement erst in ein HTML-Serversteuerelement umwandeln. Klicken Sie dazu mit der rechten Maustaste auf das Steuerelement und wählen Sie im Kontextmenü den Befehl ALS SERVERSTEUERUNG AUSFÜHREN. In der linken oberen Ecke des Steuerelements erscheint jetzt ein grünes Rechteck mit einem Pfeil darin (außerdem im *<input>*-Tag der Schaltfläche ein neues Attribut: *runat="server")*. Diese Markierung und dieses Attribut kennzeichnen alle HTML-Serversteuerelemente – ohne sie kann ein Steuerelement der Registerkarte HTML in serverbasiertem Code nicht angesprochen werden.

Der Betrieb als HTML-Serversteuerelement hat noch einen Vorteil: ASP.NET speichert automatisch den Inhalt des Steuerelements, sodass Eingaben beim erneuten Zeigen des Formulars wieder erscheinen – ohne weiteres Zutun des Programmierers wie bei ASP.

Doppelklicken Sie jetzt wieder auf die Schaltfläche, um den Code für die Behandlung Ihres `ServerClick`-Ereignisses in die CodeBehind-Datei einzugeben. In dieser Datei können Sie übrigens auch sehen, dass die HTML-Serversteuerelemente automatisch als Mitglieder der Formularklasse definiert wurden – genau wie bei Windows-Formularen.

Ergänzen Sie jetzt die Methode für das `ServerClick`-Ereignis der Schaltfläche:

```
private void Submit1_ServerClick(object sender, System.EventArgs e)
{
  if (txtName.Value != "")
    this.Response.Redirect("Auswahl.aspx?txtName=" +
      Request.Form["txtName"]);
}
```

Der Methodenkopf und die erste Codezeile sehen genauso aus wie bei einem Windows-Formular; erst der Aufruf des Auswahlformulars zeigt, dass es sich hier um Web Forms-Seiten handelt.

Das Anmeldeformular mit clientseitigem Skript

HTML-Serversteuerelemente lassen sich auch mit clientseitigem Skript kombinieren. Um das auszuprobieren, kopieren Sie am besten das Anmeldeformular über das Kontextmenü des PROJEKTMAPPEN-EXPLORERS.

Gehen Sie dann in die HTML-Ansicht und fügen Sie unterhalb des *<head>*-Tags ein *script*-Element ein; der HTML-Designer unterstützt Sie dabei mit IntelliSense. Das fertige *script*-Element soll so aussehen:

```
<script language="javascript">
  function PrüfeEingabe()
  {
    return document.Form1.txtName.value != "";
  }
</script>
```

Die Anführungszeichen um `"javascript"` fügt der Designer übrigens automatisch ein, sobald Sie in die Entwurfsansicht gehen.

Erweitern Sie nun das <*form*>-Tag um ein *onsubmit*-Attribut. Hier müssen Sie die Anführungszeichen um den Attributwert allerdings selbst eingeben, das Leerzeichen verwirrt den Designer nämlich:

```
<form id="Form1" method="post" runat="server"
  onsubmit="return PrüfeEingabe()">
```

Gehen Sie jetzt in die Entwurfsansicht und entfernen Sie die Markierung des Befehls ALS SERVERSTEUERUNG AUSFÜHREN im Kontextmenü der Schaltfläche.

Als Letztes sind noch einige Änderungen in der CodeBehind-Datei (<*Projektname*>*.aspx.cs*) nötig. Wählen Sie dort in der MEMBER-Liste (rechts oben im Editor) den Eintrag `Page_Load` und ergänzen Sie den Code wie folgt:

```
private void Page_Load(object sender, System.EventArgs e)
{
  if (IsPostBack)
    this.Response.Redirect("Auswahl.aspx?txtName=" +
      Request.Form["txtName"]);
}
```

Dieser Code ist ähnlich zu dem im Abschnitt »Codebeispiel – Formulare in HTML« von Seite 891, allerdings wird hier die `Form`-Auflistung ausgewertet und nicht die `QueryString`-Auflistung. Das ist nötig, weil Web Forms-Seiten mit einem HTTP POST an den Server gesendet werden. Die Parameter werden dabei nicht in der URL übertragen, sondern im Rumpf der Nachricht.

Entfernen Sie jetzt noch den Code für die Schaltfläche und ändern Sie ebenfalls den Klassennamen, der ist ja im Original des Formulars schon definiert.

Wie funktioniert das Ganze nun? Die Änderungen im HTML-Code führen zur Validierung der Eingabe vor dem Senden der Daten an den Server. Allerdings wird bei Web Forms-Seiten der Wert eines *action*-Attributs ignoriert, obwohl IntelliSense sogar die Eingabe unterstützt (beim Testen sieht man in der Quelltext-Ansicht des IE, dass im *action*-Attribut immer der Name der Web Forms-Seite selbst steht).

Deshalb muss die Weiterleitung zum nächsten Formular explizit kodiert werden, zum Beispiel im Ereignis `Load` des `Page`-Objekts. Dieses Ereignis tritt jedes Mal auf, wenn der Server die Seite lädt. Das geschieht natürlich auch vor dem ersten Senden an den Client, bei dem keine Weiterleitung erwünscht ist, weshalb der Code vorher die `IsPostback`-Eigenschaft prüft.

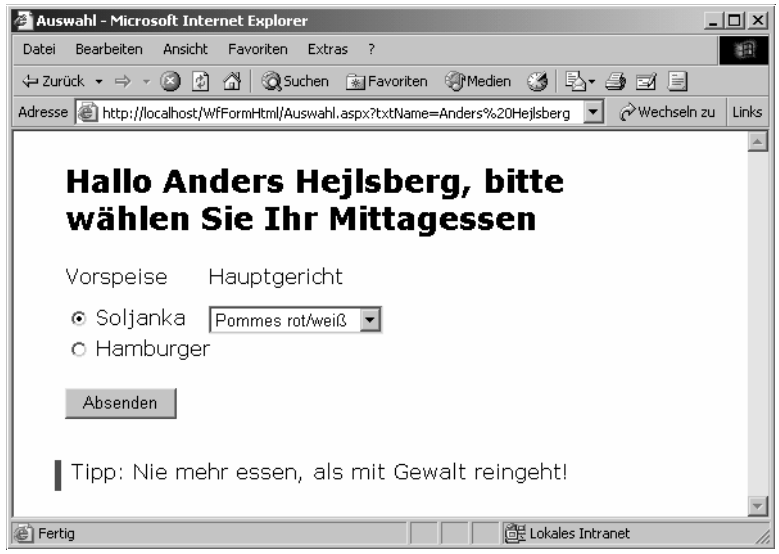
Über das Kontextmenü des PROJEKTMAPPEN-EXPLORERS können Sie festlegen, welche Web Forms-Seite als Startseite benutzt wird.

Wie die `Page_Load()`-Methode zeigt, bieten Web Forms-Seiten ein ähnliches, ereignisbasiertes Programmiermodell wie Windows-Formulare. Da die Ereignisse auf dem Server bearbeitet werden, dürfen sie nicht so häufig auftreten wie zum Beispiel `MouseMove`. Web Forms-Seiten haben deshalb nur relativ wenige Typen von Ereignissen.

Das Auswahlformular

Auch das Auswahlformular ist eine Neu-Implementation der Version aus Abschnitt »Codebeispiel – Formulare in HTML« von Seite 891, Abbildung 22.9 zeigt es noch einmal.

Abbildung 22.9: Das Auswahlformular



Fügen Sie dem Projekt eine neue Web Forms-Seite namens *Auswahl.aspx* hinzu und setzen Sie folgende HTML-Serversteuerelemente darauf: zwei `RadioButton`-Steuerelemente, sechs `Label`-Steuerelemente (zwei davon zum Anzeigen des Textes neben den Optionsschaltflächen), ein `DropDownListBox`-Steuerelement und ein `Submit Button`-Steuerelement. Stellen Sie dann die in Tabelle 22.1 gezeigten Eigenschaften ein, eine detaillierte Erläuterung des Vorgehens finden Sie im Abschnitt »Das Anmeldeformular« auf Seite 910.

Tabelle 22.1: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Oberstes Label	`id`	lblÜberschrift
Label über Optionsschaltflächen	(Eingabe)	Vorspeise
Label über DropDownComboBox	(Eingabe)	Hauptgericht
Obere Optionsschaltfläche	`id`	optSoljanka

Tabelle 22.1: Eigenschaftswerte (Forts.)

Steuerelement	Eigenschaft	Wert
Obere Optionsschaltfläche	`name`	optVorspeise
Label neben oberer Optionsschaltfläche	(Eingabe)	Soljanka
Untere Optionsschaltfläche	`id`	optHamburger
Untere Optionsschaltfläche	`name`	optVorspeise
Label neben unterer Optionsschaltfläche	(Eingabe)	Hamburger
Kombinationslistenfeld	`name`	lstHauptgericht
Submit Button	`value`	Absenden

Der Wert der `id`-Eigenschaft bestimmt den Namen, unter dem das Steuerelement im Code angesprochen wird, der Wert der `name`-Eigenschaft bestimmt den Namen des zurückgegebenen Parameters. Optionsschaltflächen mit gleicher `name`-Eigenschaft bilden automatisch eine Gruppe, in der nur eine Optionsschaltfläche markiert sein kann.

Es ist nicht möglich, die Werte für die Liste des Kombinationslistenfeldes im EIGENSCHAFTEN-Fenster einzugeben, das geschieht später in der HTML-Ansicht.

Um den Text der Überschrift per Code verändern zu können, muss sie in einem `Label`-Steuerelement verwaltet werden. Setzen Sie die Einfügemarke in dieses Steuerelement und wählen Sie in der Symbolleiste FORMATIERUNG den Eintrag *Überschrift 2* im Listenfeld BLOCKFORMAT. Dieses Listenfeld ist nur beim Editieren im `Label`-Steuerelement verfügbar.

Leider funktioniert die dabei vom HTML-Designer eingefügte Formatierung nicht wie gedacht: Wie Abbildung 22.10 zeigt, hat der Designer ein *h2*-Element in einem *div*-Element erzeugt – der Text wird aber später dem *div*-Element zugewiesen, und dem fehlt die Formatierung.

Verschieben Sie deshalb das *div*-Element in das *h2*-Element und machen Sie das *div*- zu einem *span*-Element. Abbildung 22.11 zeigt das Ergebnis.

Geben Sie jetzt noch die Werte für die Liste des Kombinationslistenfeldes ein, Abbildung 22.12 zeigt die entscheidenden Zeilen.

Gehen Sie zurück in die Entwurfsansicht. Wählen Sie dann das oberste `Label`-Steuerelement, die Optionsschaltflächen, das Kombinationslistenfeld und die Schaltfläche zusammen aus, und benutzen Sie nach einem Rechtsklick im Kontextmenü den Befehl ALS SERVERSTEUERUNG AUSFÜHREN. Doppelklicken Sie dann auf die Web Forms-Seite und ergänzen Sie die Methode für das `Load`-Ereignis des `Page`-Objekts.

Abbildung 22.10: Das h2-Element im div-Element funktioniert nicht,



Abbildung 22.11: .... aber ein span-Element im h2-Element tut es dann.

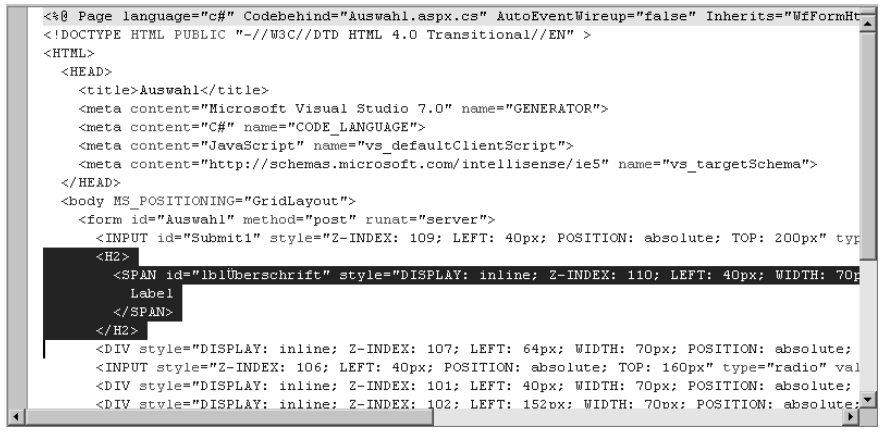
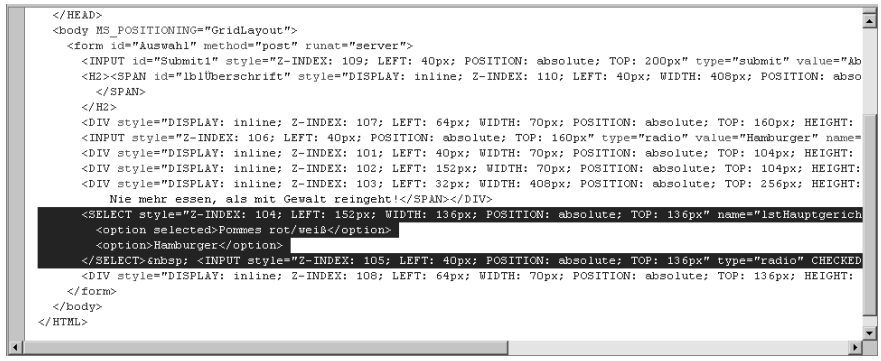


Abbildung 22.12: Die Listenwerte in der HTML-Ansicht




```
private void Page_Load(object sender, System.EventArgs e)
{
  lblÜberschrift.InnerText =
    "Hallo " + Request.QueryString["txtName"] +
    ", bitte wählen Sie Ihr Mittagessen";
  ViewState["Name"] = Request.QueryString["txtName"];
}
```

Der Code erzeugt beim Laden des Formulars den Begrüßungstext dynamisch aus dem beim Aufruf übergebenen Parameter. Da der Name des Benutzers an das nächste Formular weitergegeben werden soll, muss er zusätzlich gespeichert werden. Anders als bei der ursprünglichen Implementation erfolgt die Speicherung hier aber nicht in einem eigens angelegten versteckten Textfeld, sondern im `ViewState`-Objekt. Dieses Objekt benutzt jede Web Forms-Seite zum Speichern der Steuerelement-Inhalte, zusätzlich können Sie hier Ihre eigenen Werte ablegen.

Wählen Sie jetzt in der MEMBER-Liste den Eintrag `Submit1_ServerClick` und ergänzen Sie den generierten Methodenrumpf wie folgt:

```
private void Submit1_ServerClick(object sender, System.EventArgs e)
{
  string Vorspeise;

  if (optSoljanka.Checked)
    Vorspeise = optSoljanka.Value;
  else
    Vorspeise = optHamburger.Value;
  Response.Redirect("Resultat.Aspx?" +
    "Name=" + ViewState["Name"] +
    "&optVorspeise=" + Vorspeise +
    "&lstHauptgericht=" + lstHauptgericht.Value);
}
```

Das `Submit Button`-Steuerelement ruft jetzt die (noch zu erstellende) Web Forms-Seite *Resultat.aspx* mit den nötigen Parametern zur Ausgabe der Bestellung auf.

Die ersten beiden Formulare sind nun bereit zum Test. Lassen Sie bei der Anmeldung einmal das Namensfeld leer, und geben Sie nur ein Passwort ein: Wie erwartet erscheint erneut das Anmeldeformular, aber im Gegensatz zur Implementation im Abschnitt »Codebeispiel – Formulare in HTML« auf Seite 891 bleibt Ihre Eingabe im Passwortfeld erhalten.

ASP.NET speichert die Inhalte der HTML-Serversteuerelemente dazu automatisch im `ViewState`-Objekt. Dieses Objekt funktioniert analog zu den `Session`- und `Application`-Objekten, die Sie auch in ASP.NET-Webanwendungen benutzen können. Eine Beschreibung ihrer Möglichkeiten finden Sie im Abschnitt »fortgeschrittene Techniken für das Caching« auf Seite 971.

Das Stylesheet

Die Web Forms-Seiten benutzen zur Formatierung das gleiche Stylesheet wie die ursprüngliche Implementation im Abschnitt »Codebeispiel – Formulare in HTML« auf Seite 891. Dieses Mal bietet aber der CSS-Editor Unterstützung beim Anlegen des Stylesheets.

Fügen Sie dazu dem Projekt ein leeres Stylesheet hinzu (PROJEKT/NEUES ELEMENT HINZUFÜGEN/STYLESHEET) und nennen Sie es *Default.css*. Ergänzen Sie es wie folgt:

```
body
{ background-color:Ivory;
  font-family: Verdana, Arial, Sans-Serif;
}
.Tip
{ border-left: solid 0.3em red;
  padding-left: 0.5em
}
```

Der CSS-Editor bietet durch IntelliSense eine Auswahl sowohl für die verfügbaren Attribute als auch für deren Werte. Die Farbe *Ivory* ist ihm allerdings unbekannt, sie wurde manuell eingegeben.

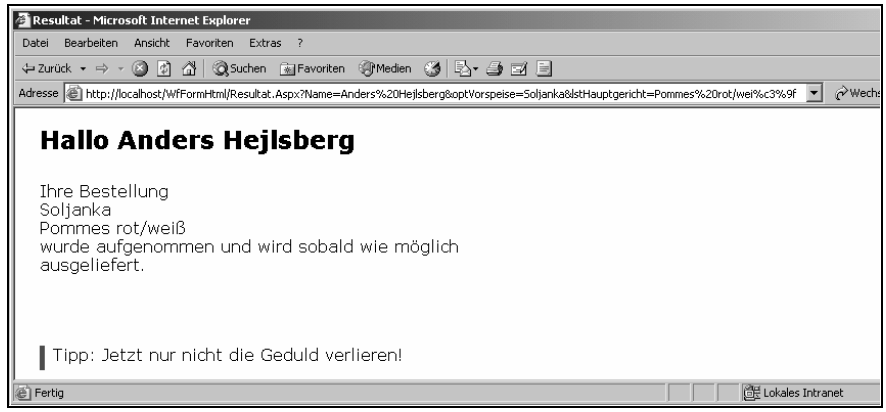
Zum Zuweisen des Stylesheets doppelklicken Sie einfach im PROJEKTMAPPEN-EXPLORER auf die Datei *Auswahl.aspx* und ziehen dann das Stylesheet aus dem PROJEKTMAPPEN-EXPLORER in den HTML-Designer. Das Auswahlformular wird jetzt elfenbeinfarben dargestellt, und in der HTML-Ansicht können Sie sehen, dass der Designer einen Link zum Stylesheet eingefügt hat.

Zum Formatieren des unteren Labels geben Sie im EIGENSCHAFTEN-Fenster `"Tipp"` als `class`-Wert ein und drücken EINGABE – sofort erscheint der rote Streifen am linken Rand des Labels.

Das Resultatformular

Für das Resultatformular fügen Sie dem Projekt eine neue Web Forms-Seite hinzu und nennen sie *Resultat.aspx*. Abbildung 22.13 zeigt, wie das Ergebnis im Betrieb aussieht.

Abbildung 22.13: Das Resultatformular



Für das Resultatformular benötigen Sie drei `Label`-Steuerelemente (Registerkarte HTML der TOOLBOX). Formatieren Sie das obere wieder als *Überschrift 2* (einschließlich der Korrekturen im HTML-Code), machen Sie das obere und das mittlere zu HTML-Serversteuerelementen und geben Sie den Text in das untere ein.

Ziehen Sie dann wieder das Stylesheet in die Entwurfsansicht des Resultatformulars und weisen Sie schließlich dem unteren `Label`-Steuerelement wieder die Klasse *Tipp* zu.

Doppelklicken Sie jetzt auf das Formular und ergänzen Sie die Methode `Page_Load()` wie folgt:

```
private void Page_Load(object sender, System.EventArgs e)
{
  lblÜberschrift.InnerText = "Hallo " + Request.QueryString["Name"];
  lblBestellung.InnerHtml = "Ihre Bestellung<br/>" +
    Request.QueryString["optVorspeise"] + "<br/>" +
    Request.QueryString["lstHauptgericht"] + "<br/>" +
    "wurde aufgenommen und wird sobald wie möglich ausgeliefert.";
}
```

Hier fällt auf, dass der Text für `lblBestellung` nicht über die `InnerText`-, sondern über die `InnerHtml`-Eigenschaft gesetzt wird: Das ist immer nötig, wenn der Text HTML-Tags enthält.

Die Werte der beim Aufruf übergebenen Parameter liefert die `QueryString`-Auflistung, denn das Formular wurde per HTTP GET aufgerufen.

Ein schneller Test mit ProxyTrace zeigt, dass zur Aufnahme der Bestellung rund 11.000 Bytes transportiert werden, also doppelt so viele wie bei der klassischen Version aus dem Abschnitt »Codebeispiel – Formulare in HTML« von Seite 891. Das macht sich im üblichen Intranet-Szenario des Entwicklers nicht bemerkbar, kann aber schon deutliche Auswirkungen auf die Skalierbarkeit einer Anwendung unter Produktionsbedingungen haben.

Der Effekt wird sich in realistischeren Anwendungen mit mehr Steuerelementen noch verstärken, denn dort fallen die HTTP-Header mit ihrer von der Implementation unabhängigen Größe weniger stark ins Gewicht als in diesem Mini-Beispiel mit seinen wenigen Steuerelementen.

Allerdings kann man die Datenmenge wahrscheinlich einschränken, wenn man die Steuerelemente in eine Tabelle (`HtmlTable`) einfügt. Ob und wie sich dabei einsparen lässt, hängt von den Layout-Details ab. Zumindest sieht der resultierende HTML-Code um einiges aufgeräumter aus.

22.3 Web Forms-Seiten mit Webserver-Steuerelementen

Mit Webserver-Steuerelementen programmieren Sie HTML-Oberflächen – und zwar genau so, wie Sie es von Windows-Anwendungen her gewohnt sind: Sie erzeugen die Oberfläche aus vorgefertigten Steuerelementen, stellen deren Eigenschaften in einem EIGENSCHAFTEN-Fenster ein und schreiben Code zur Behandlung ihrer Ereignisse in einem Code-Editor.

Webserver-Steuerelemente bieten dabei einen wesentlich höheren Abstraktionsgrad als HTML-Serversteuerelemente (siehe Abschnitt »Web Forms-Seiten mit HTML-Serversteuerelementen« auf Seite 899). Während diese »einfach« gängige HTML-Tags kapseln, beruhen Webserver-Steuerelemente auf einem einheitlichen Programmiermodell mit einem einheitlichen Satz von Eigenschaften.

Webserver-Steuerelemente generieren zur Laufzeit ein auf den Client-Browser abgestimmtes HTML, eventuell zusätzlich CSS und Script – man stellt sie sich deshalb am besten als programmierbare Generatoren vor. Dementsprechend ist der Inhalt der HTML-Ansicht nicht unbedingt identisch mit dem, was an den Client gesendet wird. Im Gegensatz zu früheren Microsoft-Technologien senden Webserver-Steuerelemente aber keine ActiveX-Komponenten oder sonstige Binärdaten an den Client.

Neben den zu erwarteten Implementationen der üblichen HTML-Steuerelemente existieren sehr komplexe Webserver-Steuerelemente wie zum Beispiel ein Kalender.

Webserver-Steuerelemente finden Sie auf der Registerkarte WEB FORMS der TOOLBOX. Wie die HTML-Serversteuerelemente können Sie auch die Webserver-Steuerelemente sowohl in der Entwurfs- als auch in der HTML-Ansicht ansehen und bearbeiten; außerdem lassen sich beide Typen von Serversteuerelementen auch gemischt verwenden.

Codebeispiel – Web Forms-Seiten mit Webserver-Steuerelementen

Das Beispielprojekt *WfFormWeb* ist die dritte und letzte Implementation der Funktionalität, die Sie schon aus dem Abschnitt »Codebeispiel – Formulare in HTML« von Seite 891 kennen: eine aus drei Formularen bestehende Anwendung zum Zusammenstellen eines Mittagessens. Diesmal mit Webserver-Steuerelementen.

Das Anmeldeformular

Auch in dieser Implementation soll das Aussehen des Anmeldeformulars nicht durch ein Stylesheet festgelegt werden, sondern durch die entsprechenden Einstellungsmöglichkeiten der IDE: Abbildung 22.14 zeigt das Resultat in der Entwurfsansicht.

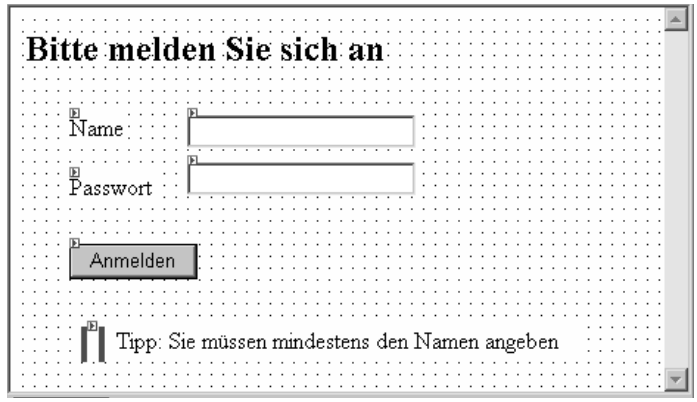


Abbildung 22.14: Das Anmeldeformular in der Entwurfsansicht

Wie Abbildung 22.14 durch die Symbole in den linken oberen Steuerelementecken zeigt, werden Webserver-Steuerelemente immer auf dem Server ausgeführt – eine Umschaltmöglichkeit wie bei HTML-Serversteuerelementen besteht nicht.

Fügen Sie dem Projekt eine neue Web Forms-Seite mit dem Namen *Anmeldung.aspx* hinzu. Ziehen Sie von der Registerkarte WEB FORMS der TOOLBOX drei `Label`- (die Überschrift entsteht gleich in der HTML-Ansicht), zwei `TextBox`- und ein `Button`-Steuerelement auf das Formular und tragen Sie für die jeweiligen `Text`-Eigenschaften die in Abbildung 22.14 zu sehenden Werte ein.

Nennen Sie die Textfelder *txtName* und *txtPasswort* und stellen Sie für das Kennwort-Feld die `TextMode`-Eigenschaft auf *Password*. Setzen Sie dann die `bgColor`-Eigenschaft des Formulars auf *Ivory*. Dazu zeigen Sie im Wert-Editor die Registerkarte BENANNTE FARBEN an und wählen im Bereich WEITERE die vierte Farbe von oben in der dritten Spalte von rechts.

Wechseln Sie jetzt in die HTML-Ansicht und legen Sie die Überschrift als *h2*-Element an. Ergänzen Sie außerdem noch das *style*-Attribut des Tipp-Labels um folgende Werte:

```
PADDING-LEFT: 0.5em; BORDER-LEFT-COLOR: red; BORDER-LEFT-STYLE: solid;
```

Leider existiert für `Label`-Steuerelemente keine Möglichkeit, diese Formatierungen im EIGENSCHAFTEN-Fenster oder mit einem Designer zu erzeugen – CSS-Wissen bleibt also nach wie vor wichtig.

Bei der Rückkehr in die Entwurfsansicht irritiert die Verdoppelung des roten Streifens; das Problem beschränkt sich aber auf die Entwurfsansicht, im Client-Browser wird nur einer der Streifen zu sehen sein.

Der Code für die ANMELDEN-Schaltfläche sieht fast genauso aus wie bei der Implementation durch HTML-Serversteuerelemente. Trotzdem wird auch

hier wieder der vereinheitlichte Eigenschaftensatz deutlich: Der Benutzername steckt in der `Text`-Eigenschaft.

```
private void btnAnmelden_Click(object sender, System.EventArgs e)
{
  if (txtName.Text != "")
    this.Response.Redirect("Auswahl.aspx?txtName=" + txtName.Text);
}
```

Validierung auf dem Client

Auch hier ergibt sich natürlich wieder der Wunsch, die Validierung auf dem Client durchzuführen. Im Gegensatz zu den beiden vorhergehenden Implementationen brauchen Sie dafür aber keinen JavaScript-Code zu schreiben, sondern können stattdessen ein Überprüfungssteuerelement benutzen.

Ein Überprüfungssteuerelement kapselt die Überprüfungsregeln für ein anderes Steuerelement, wobei einem Steuerelement auch mehrere Überprüfungssteuerelemente zugeordnet werden können.

Für verschiedene Arten der Überprüfung existieren eigene Überprüfungssteuerelement-Implementationen: So gibt es zum Beispiel ein `RequiredFieldValidator`-, ein `RangeValidator`- und ein `RegularExpressionValidator`-Steuerelement.

Überprüfungssteuerelemente finden Sie auf der Registerkarte WEB FORMS der TOOLBOX, es handelt sich also um Webserver-Steuerelemente. Trotzdem können sie die Validierung auf dem Client durchführen, denn sie generieren dazu JavaScript (in Abhängigkeit von den Fähigkeiten des Browsers). Zusätzlich führen Sie immer auch noch eine Validierung auf dem Server durch, da die Validierung auf dem Client durch direktes Senden der Daten umgangen werden kann.

Der Einsatz eines Überprüfungssteuerelements gestaltet sich erfreulich einfach: Setzen Sie ein `RequiredFieldValidator`-Steuerelement auf das Anmeldeformular neben das Name-Feld, und setzen Sie dann die `ControlToValidate`-Eigenschaft des Steuerelements auf `txtName` sowie seine `ErrorMessage`-Eigenschaft auf `"Bitte geben Sie den Namen ein"`.

Das Resultat können Sie sofort testen: Wenn das Name-Feld leer ist, erfolgt beim Klick auf ANMELDEN kein Zugriff auf den Server – stattdessen erscheint die Fehlermeldung des Überprüfungssteuerelements. Die Spuren seines clientseitigen Scripts können Sie auch in der Quelltext-Ansicht des IE sehen, allerdings nur in Form eines Verweises auf eine JavaScript-Datei.

Das Auswahlformular

Das Auswahlformular in der Entwurfsansicht zeigt Abbildung 22.15.



Abbildung 22.15: Das Auswahlformular in der Entwurfsansicht

Für das Auswahlformular benötigen Sie eine neue Web Forms-Seite namens *Auswahl.aspx* sowie drei `Label`-, ein `RadioButtonList`- und ein `DropDownList`-Steuerelement. Stellen Sie dann die in Tabelle 22.2 gezeigten Eigenschaften ein.

Tabelle 22.2: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Oberstes Label	`ID`	lblÜberschrift
Linkes Label	`Text`	Vorspeise
Rechtes Label	`Text`	Hauptgericht
Optionsschaltflächengruppe	`ID`	optVorspeise
Kombinationslistenfeld	`ID`	lstHauptgericht
Schaltfläche	`ID`	btnAbsenden
Schaltfläche	`Text`	Absenden
Unteres Label	`CssClass`	Tipp
Unteres Label	`Text`	Tipp: Nie mehr essen, als mit Gewalt reingeht!

Zum Anlegen der Optionsschaltflächen klicken Sie auf die kleine, durch drei Punkte gekennzeichnete Schaltfläche der Eigenschaft `Items` und geben im LISTITEM-AUFLISTUNGS-EDITOR ihre Beschriftung in die `Text`-Eigenschaft ein. Setzen Sie außerdem die `Selected`-Eigenschaft der ersten Optionsschaltfläche auf `True`.

Auf die gleiche Art füllen Sie dann die Liste des Kombinationslistenfeldes, hier brauchen Sie aber keinen Eintrag zu selektieren.

Ziehen Sie jetzt das Stylesheet aus dem PROJEKTMAPPEN-EXPLORER auf das Formular, um die Formatierung anzuwenden. Gehen Sie dann in die HTML-Ansicht und umgeben Sie das *asp:Label*-Element der Überschrift mit einem *h2*-Element.

Wechseln Sie wieder in die Entwurfsansicht und legen Sie folgende Methoden für das `Load`-Ereignis des Formulars und für das `Click`-Ereignis der Schaltfläche an:

```
private void Page_Load(object sender, System.EventArgs e)
{
  lblÜberschrift.Text =
    "Hallo " + Request.QueryString["txtName"] +
    ", bitte wählen Sie Ihr Mittagessen";
  ViewState["Name"] = Request.QueryString["txtName"];
}


private void btnAbsenden_Click(object sender, System.EventArgs e)
{
  Response.Redirect("Resultat.Aspx?" +
    "Name=" + ViewState["Name"] +
    "&optVorspeise=" + optVorspeise.SelectedItem.Text +
    "&lstHauptgericht=" + lstHauptgericht.SelectedItem.Text);
}
```

Beim Vergleich mit der Implementation durch HTML-Serversteuerelemente fällt auf, dass der Wert einer Optionsschaltfläche hier wesentlich leichter auszulesen ist. Ansonsten unterscheidet sich der Code nur durch die Namen der Eigenschaften.

Das Resultatformular

Das Resultatformular in der Entwurfsansicht zeigt Abbildung 22.16.

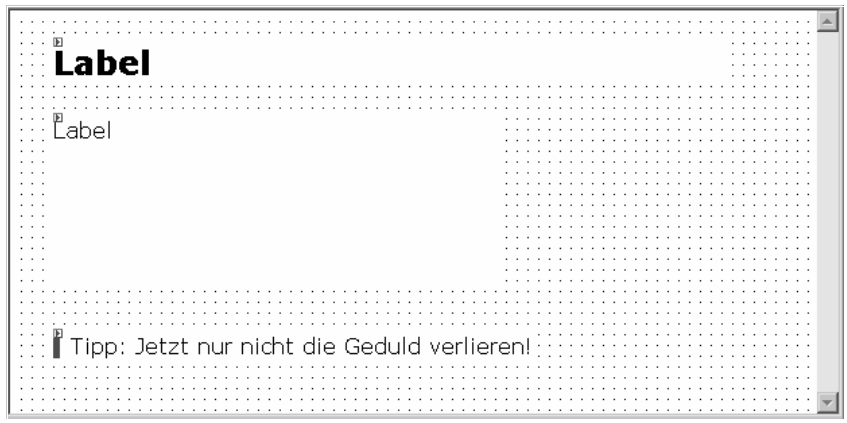


Abbildung 22.16: Das Resultatformular in der Entwurfsansicht

Fügen Sie dem Projekt eine neue Web Forms-Seite namens *Resultat.aspx* hinzu. Setzen Sie auf diese Web Forms-Seite drei Webserver-Steuerelemente vom Typ `Label` und geben Sie die in Abbildung 22.16 gezeigten Beschriftungen in die `Text`-Eigenschaft der Steuerelemente ein.

Ziehen Sie jetzt das Stylesheet aus dem PROJEKTMAPPEN-EXPLORER auf das Formular um die Formatierung anzuwenden. Gehen Sie dann in die HTML-Ansicht und umgeben Sie das *asp:Label*-Element der Überschrift mit einem *h2*-Element.

Wechseln Sie wieder in die Entwurfsansicht und legen Sie folgende Methode für das `Load`-Ereignis des Formulars an:

```
private void Page_Load(object sender, System.EventArgs e)
{
  lblÜberschrift.Text = "Hallo " + Request.QueryString["Name"];
  lblBestellung.Text = "Ihre Bestellung<br/>" +
    Request.QueryString["optVorspeise"] + "<br/>" +
    Request.QueryString["lstHauptgericht"] + "<br/>" +
    "wurde aufgenommen und wird sobald wie möglich ausgeliefert.";
}
```

Das ist fast der gleiche Code wie in der Implementation durch HTML-Serversteuerelemente, nur der Name der Eigenschaft für die Beschriftung des `Label`-Steuerelements ist unterschiedlich.

Im Vergleich mit der Implementation durch HTML-Serversteuerelemente ergibt sich beim IE6 auch die gleiche Datenmenge für die Durchführung einer Bestellung. Bei anderen Browsern können die Datenmengen unterschiedlich sein, denn Webserver-Steuerelemente generieren ihr HTML ja Browser-spezifisch.

22.4 Web Forms-Seiten und Datenbindung

ASP.NET definiert das Konzept der Datenbindung wesentlich universeller als ASP und geht dabei sogar über frühere Entwicklungsumgebungen für Windows-Anwendungen, zum Beispiel Visual Basic 6, weit hinaus.

Zum einen fasst ASP.NET den Begriff der Datenquelle wesentlich weiter. So muss ein Objekt zum Liefern einer bindungsfähigen Datenmenge nur eine der Schnittstellen `Icollection`, `IEnumerable` oder `IListSource` implementieren – das können neben den erwarteten `DataTable`- oder `DataView`-Instanzen zum Beispiel auch `ArrayList`- oder `StringCollection`-Instanzen sein. Außerdem lassen sich nicht nur Datenmengen, sondern auch einzelne Daten per Datenbindung zuweisen: die Werte von Variablen und Eigenschaften ebenso wie die Rückgabewerte von Methoden und Ausdrücken.

Zum anderen lassen sich Daten nicht mehr nur an eine bestimmte Steuerelement-Eigenschaft binden, wie beispielsweise die `Text`-Eigenschaft eines Textfeldes; sie lassen sich mit dem gleichen Konzept auch in einer Web Forms-Seite ausgeben, ohne dass dazu ein Steuerelement nötig wäre.

22.4.1 Web Forms-Seiten und Serversteuerelemente an Daten binden

Unter ASP.NET benutzen Sie zur Ausgabe bzw. Bindung von Daten eine einfache deklarative Syntax, die an einen Skript-Block erinnert.

```
<%# NameDerDaten %>
```

Erst beim Aufruf der `DataBind()`-Methode wird die Datenbindung ausgeführt. Diese Methode besitzen sowohl das `Page`-Objekt als auch alle Serversteuerelemente, wobei der Aufruf dieser Methode am übergeordneten Steuerelement automatisch zum Aufruf der Methode für alle seine untergeordneten Steuerelemente führt.

Der folgende Auszug einer CodeBehind-Datei aus dem Beispielprojekt *WfData* stellt als zu bindende Daten einen String und ein `string`-Array zur Verfügung; beachten Sie den Aufruf von `DataBind()` in der `Page_Load()`-Methode.

```
public class WebForm1 : System.Web.UI.Page
{
  protected System.Web.UI.WebControls.CheckBoxList CheckBoxList1;
  protected System.Web.UI.HtmlControls.HtmlSelect Select1;

  public string Nachricht = "Hallo Welt";
  public string[] Programmiersprachen = null;

  private void Page_Load(object sender, System.EventArgs e)
  {
    Programmiersprachen =
      new String[4]{"C#", "Delphi", "Java", "Visual Basic"};
    DataBind();
  }
}
```

Das nächsten Listing demonstriert die Datenbindung in der HTML-Ansicht – sowohl für die Ausgabe in der Web Forms-Seite als auch zum Füllen eines Listenfelds und einer Gruppe von Kontrollkästchen. Da das Listenfeld ein HTML-Steuerelement ist, muss es als Serversteuerelement ausgeführt werden. Der besseren Übersicht halber wurden die Steuerelement-Positionsangaben aus dem Listing entfernt.

```
<form id="Form1" method="post" runat="server">
  <P><%#Nachricht%></P>
  <P>
  <SELECT style="Z-INDEX: 101; size=5
    DataSource="<%#Programmiersprachen%>"
    id=Select1 name=Select1 runat="server">
  </SELECT>
  <asp:CheckBoxList id="CheckBoxList1"
    DataSource="<%#Programmiersprachen%>"
    runat="server" style="Z-INDEX: 102; >
  </asp:CheckBoxList>
  </P>
</form>
```

Abbildung 22.17 zeigt das unspektakuläre Ergebnis.

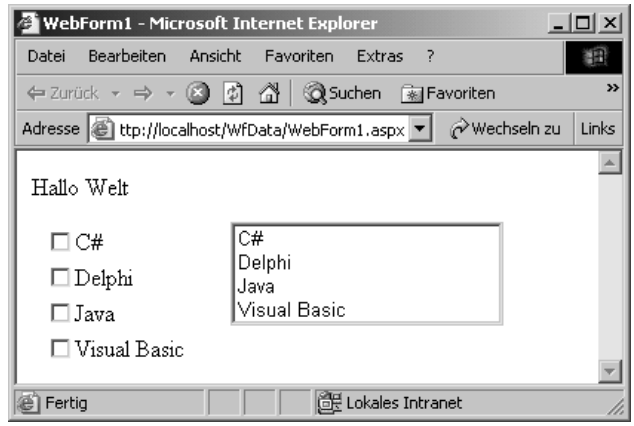


Abbildung 22.17: Web Forms-Seiten und Serversteuerelemente an Daten binden

Wenn eine Datenquelle mehrere Felder bietet, legen Sie zur Datenbindung zusätzlich die `DataTextField`-Eigenschaft fest; wenn sie mehrere Tabellen bietet, legen Sie auch noch die `DataMember`-Eigenschaft fest. Alle diese Zuordnungen können Sie natürlich auch im EIGENSCHAFTEN-Fenster vornehmen.

22.4.2 Spezielle Webserver-Steuerelemente für Ausgaben in Listenform

Neben den zu erwartenden Webserver-Steuerelementen wie `TextBox`, `Label` oder `ListBox` bietet ASP.NET drei listengebundene Webserver-Steuerelemente mit besonderen Funktionen: `DataGrid`, `DataList` und `Repeater`. Jedes dieser Steuerelemente erlaubt die Darstellung einer Datenmenge mit mehreren Feldern, also zum Beispiel den Inhalt einer Datenbank-Tabelle.

Alle drei Steuerelemente bieten neben einer weitreichenden Gestaltungsfreiheit der Oberfläche zusätzlich die Möglichkeit, zusammen mit den Daten auch Steuerelemente anzuzeigen – zum Beispiel ein `LinkButton`-Webserver-Steuerelement. Dadurch ergibt sich indirekt die Möglichkeit zur Veränderung der angezeigten Daten, wozu natürlich ein Postback zum Server nötig ist.

Die listengebundenen Webserver-Steuerelemente erzeugen für jeden Eintrag eine Instanz des dazu angezeigten Steuerelements und übernehmen auch die Weiterleitung der von dieser Instanz ausgelösten Ereignisse (event bubbling). Der Server analysiert dann die zurückgesendeten Daten und ermittelt anhand von Namenskonventionen die aufzurufende Methode.

Bei den auslösbaren Ereignissen handelt es sich zum einen um vordefinierte Ereignisse (`cancel`, `edit`, `delete` und `update`), für die listengebundene Webserver-Steuerelemente entsprechende Ereignisbehandlungsmethoden bieten. Die Verknüpfung können Sie sowohl über Attribute wie *OnCancelCommand* in der HTML-Ansicht vornehmen als auch über die entsprechenden Einträge im EIGENSCHAFTEN-Fenster.

Zum zweiten können Sie neben diesen vordefinierten Ereignissen auch eigene definieren, die Sie dann entweder durch das *OnItemCommand*-Attribut oder über den Eintrag *ItemCommand* im EIGENSCHAFTEN-Fenster mit einer entsprechenden Methode verbinden.

Zum Zugriff auf die Daten werden meistens Konstrukte wie das folgende verwendet:

```
<input id="txtDateiname" type="text" runat="server"
  value='<%# DataBinder.Eval(Container.DataItem, "Dateiname") %>'  />
```

Dabei enthält `Container` die erste Instanz eines übergeordneten Steuerelements, das die Schnittstelle `INamingContainer` implementiert – je nach Typ des listengebundenen Webserver-Steuerelements ist das ein Objekt vom Typ `DataListItem`, `DataGridItem` oder `RepeaterItem`. Die `DataItem`-Eigenschaft des `Container`-Objekts verweist automatisch auf den aktuell auszugebenden Eintrag des listengebundenen Webserver-Steuerelements. Schließlich liefert die statische Methode `Eval()` der `DataBinder`-Klasse den Wert aus dem angegebenen Feld.

Die Formatierung der einzelnen Einträge erfolgt über Vorlagen (Templates), wobei die einsetzbaren Vorlagentypen zwischen den einzelnen listengebundenen Webserver-Steuerelementen variieren.

Generell lässt sich sagen, dass das `DataGrid`-Webserver-Steuerelement die meiste eingebaute Funktionalität besitzt und jeden Eintrag als Zeile in einer Tabelle darstellt. Das `Repeater`-Webserver-Steuerelement bildet das andere

Extrem, bietet praktisch gar keine eingebaute Funktionalität, dafür aber den größten Gestaltungsspielraum. Das `DataList`-Webserver-Steuerelement liegt bezüglich eingebauter Funktionalität und Gestaltungsspielraum in der Mitte und stellt jeden Eintrag als Zelle in einer Tabelle dar.

22.4.3 Das DataGrid-Steuerelement

Das `DataGrid`-Steuerelement zeigt seine Daten tabellenartig an: jeden Datensatz in einer Zeile, jedes Feld in einer Spalte. Dabei bietet es im Gegensatz zu den beiden anderen listengebundenen Webserver-Steuerelementen eine Standarddarstellung, ist also ohne die Definition von Vorlagen sofort einsetzbar.

Dieses Steuerelement bietet als einziges die Möglichkeit, seine Daten seitenweise darzustellen (Paging). Außerdem ist eine Sortierfunktion vorbereitet.

Allerdings lässt sich das Erscheinungsbild der Daten nicht so weit wie bei den beiden anderen listengebundenen Webserver-Steuerelementen bestimmen: Es fehlt zum Beispiel die Möglichkeit, Separatoren wie beim `Repeater`-Steuerelement oder *EditItemTemplate*-Vorlagen wie beim `DataList`-Steuerelement anzulegen.

Codebeispiel – Paginieren und Sortieren mit dem DataGrid-Steuerelement

Das Beispielprojekt *WfDataGrid* zeigt zunächst das grundsätzliche Vorgehen beim Binden eines Steuerelements wie `DataGrid`, `DataList` oder `Repeater` an eine Datenmenge. Dabei bietet das `DataGrid`-Steuerelement als einziges eine Standarddarstellung der Datenmenge. Abbildung 22.18 zeigt das Ergebnis: schlicht, aber effektiv.

Setzen Sie ein `DataGrid`-Steuerelement (Registerkarte WEB FORMS) auf einen Web Form-Entwurf. Binden Sie dann das Steuerelement an eine Datenmenge, indem Sie die CodeBehind-Datei – gegebenenfalls unter Anpassung von Dateipfad und -filter – wie folgt ergänzen:

```
//...
using System.IO;

namespace WfDataGrid
{
  public class WebForm2 : System.Web.UI.Page
  {
    protected System.Web.UI.WebControls.DataGrid DataGrid1;

  protected const string Pfad =
    "D:\\Programme\\Microsoft Visual Studio .NET\\FrameworkSDK\\Samples" +
    "\\QuickStart\\aspplus\\images";
```

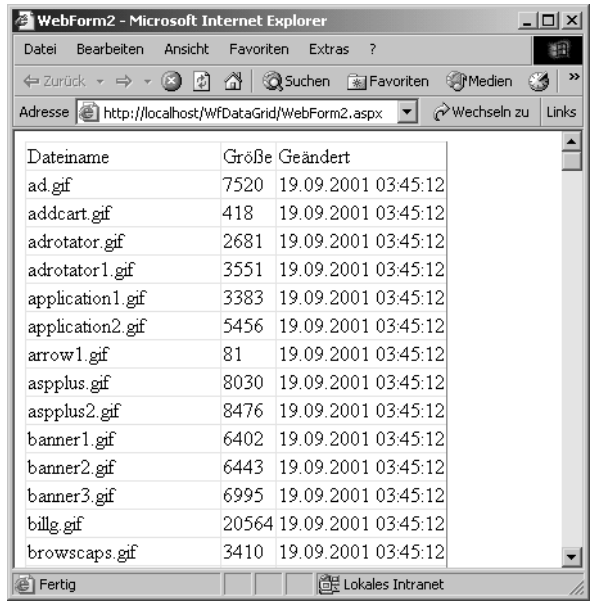


Abbildung 22.18: Das DataGrid-Steuerelement mit Standarddarstellung

```
private void BindeGrid()
{
  DataGrid1.DataSource = (DataTable)Session["Daten"];
  DataGrid1.DataBind();
}//BindeGrid


private DataTable ErzeugeDaten()
{
  DataTable t = new DataTable();
  t.Columns.Add(
    new DataColumn("Dateiname", Type.GetType("System.String")));
  t.Columns.Add(
    new DataColumn("Größe", Type.GetType("System.Int32")));
  t.Columns.Add(
    new DataColumn("Geändert", Type.GetType("System.DateTime")));

  DataRow r = null;
  DirectoryInfo di = new DirectoryInfo(Pfad);
  foreach (FileInfo fi in di.GetFiles("*.gif"))
  {
    r = t.NewRow();
    r["Dateiname"] = fi.Name;
    r["Größe"] = fi.Length;
    r["Geändert"] = fi.LastWriteTime;
    t.Rows.Add(r);
  }
  return t;
}//ErzeugeDaten
```

```
private void Page_Load(object sender, System.EventArgs e)
{
  if (!IsPostBack)
  {
    DataTable t = ErzeugeDaten();
    Session.Add("Daten", t);
  }
  BindeGrid();
}//Page_Load
```

Beim ersten Senden der Web Forms-Seite an den Client lässt die `Page_Load()`-Methode eine Datenmenge erzeugen und an das `DataGrid`-Steuerelement binden. Für folgende Seitenaufrufe speichert `Page_Load()` die Datenmenge im `Session`-Objekt. Je nach Rahmenbedingungen könnte es auch sinnvoll sein, sie im `Application`-, `Cache`- oder `ViewState`-Objekt unterzubringen – oder sie bei jedem Seitenaufruf neu aufzubauen.

Die Methode `ErzeugeDaten()` generiert aus den *.gif*-Dateien der Beispiele von Visual Studio eine Tabelle mit den drei Feldern `Dateiname`, `Größe` und `Geändert`; `BindeGrid()` bindet das `DataGrid`-Steuerelement an die gespeicherte Datenmenge.

Tatsächlich erschöpft sich der Code zur Datenbindung also in zwei Zeilen, der Rest ist für die Datengenerierung zuständig. Damit ist das Beispiel bereits lauffähig und liefert das in Abbildung 22.18 zu sehende Resultat.

Natürlich lässt sich das Aussehen der Tabelle interessanter gestalten. Rufen Sie dazu im Kontextmenü des `DataGrid`-Steuerelements den Befehl AUTOMATISCHE FORMATIERUNG auf, wählen Sie als Farbschema *Farbig 1* und klicken Sie OK. Die Entwurfsansicht stellt das Steuerelement sofort im gewählten Farbschema dar.

Rufen Sie jetzt im Kontextmenü des `DataGrid`-Steuerelements den Befehl EIGENSCHAFTENGENERATOR auf. Selektieren Sie dann im Fenster DATAGRID1 EIGENSCHAFTEN die Registerkarte FORMAT und wählen Sie zum Beispiel den Schriftnamen *AvantGarde Bk BT*; klicken Sie auf ÜBERNEHMEN, um das Resultat im Steuerelement zu sehen. Auf diese Art und Weise können Sie das Aussehen der generierten Tabelle nahezu beliebig gestalten.

Bei dieser Gestaltung darf man zwei Dinge nicht aus den Augen verlieren: Zum einen muss die gewählte Schrift auf dem Rechner des Benutzers verfügbar sein, deshalb werden Sie im Allgemeinen mehrere alternative Schriften angeben. Zum anderen überschreiben spezielle Festlegungen die allgemeineren; so würde zum Beispiel die für das `DataGrid`-Steuerelement allgemein festgelegte Schrift in der Kopfzeile durch eine dafür festgelegte Kopfzeilen-Schrift ersetzt.

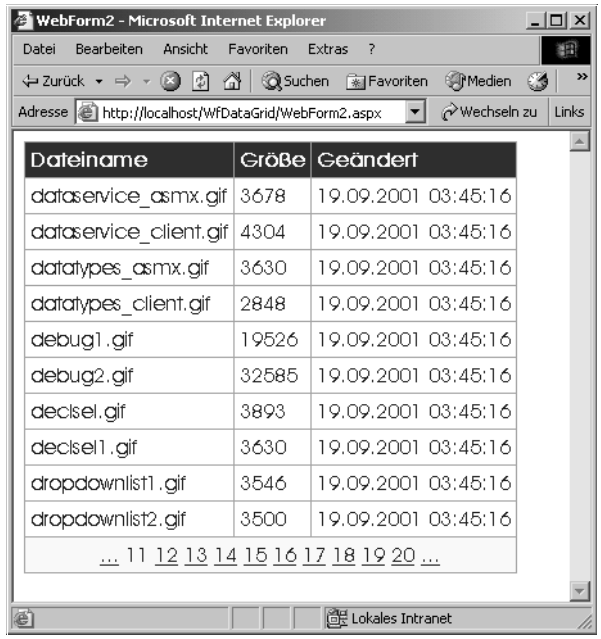
Auch die Funktionalität der Tabelle lässt sich erweitern, zum Beispiel um eine Paginierung. Rufen Sie dazu wieder den Eigenschaftengenerator auf und wählen Sie die Registerkarte PAGING. Markieren Sie dort das Kontrollkästchen PAGING und als Modus *Seitenzahlen*. Klicken Sie OK – die Oberfläche der neuen Funktionalität ist schon im Steuerelement zu sehen.

Allerdings ist die neue Funktionalität noch nicht verfügbar, dazu fehlen genau zwei Zeilen Code. Legen Sie über das EIGENSCHAFTEN-Fenster eine Methode für das `PageIndexChanged`-Ereignis des Steuerelements an und ergänzen Sie sie folgendermaßen:

```
private void DataGrid1_PageIndexChanged(object source,
  System.Web.UI.WebControls.DataGridPageChangedEventArgs e)
  {
  DataGrid1.CurrentPageIndex = e.NewPageIndex;
  BindeGrid();
}//DataGrid1_PageIndexChanged
```

Abbildung 22.19 zeigt das Ergebnis.

Abbildung 22.19: Das DataGrid-Steuerelement mit Paging



Ebenso leicht wie das Paging lässt sich dem Steuerelement auch eine Sortierung hinzufügen. Rufen Sie dazu wieder den Eigenschaftengenerator auf, wählen Sie die Registerkarte ALLGEMEIN und markieren Sie dort das Kontrollkästchen SORTIEREN ZULASSEN. Nach einem Klick auf OK ist auch hier

wieder neue Funktionalität in der Oberfläche erkennbar: Die Spaltenüberschriften werden als Hyperlinks formatiert.

Die Implementation benötigt diesmal drei Zeilen. Legen Sie dazu eine Methode zum Behandeln des `SortCommand`-Ereignisses an und ergänzen Sie sie wie folgt:

```
private void DataGrid1_SortCommand(object source,
  System.Web.UI.WebControls.DataGridSortCommandEventArgs e)
{
  DataTable t = (DataTable)DataGrid1.DataSource;
  t.DefaultView.Sort = e.SortExpression;
  BindeGrid();
}//DataGrid1_SortCommand
```

Da die `Sort`-Eigenschaft der `DefaultView`-Klasse mit dem Spaltennamen zufrieden ist, übergeben Sie hier einfach den Wert der `SortExpression`-Eigenschaft. Durch Anhängen eines `" ASC"` oder `" DESC"` könnten Sie auch auf- und absteigend sortieren. Abbildung 22.20 zeigt das Ergebnis der (standardmäßig aufsteigenden) Sortierung.

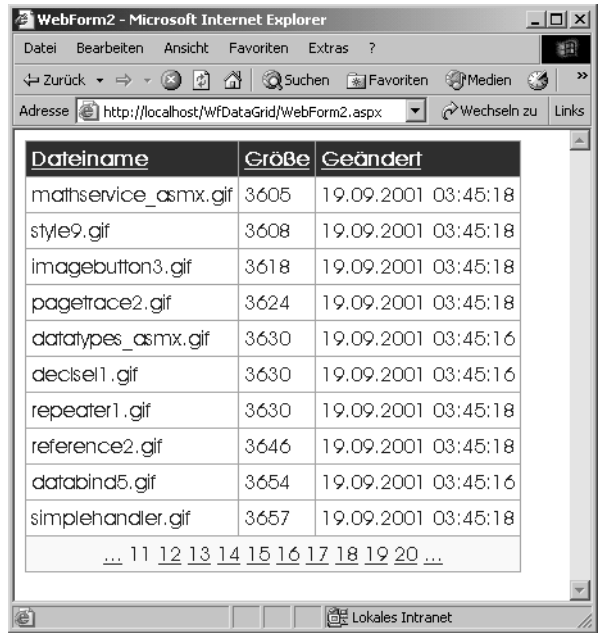


Abbildung 22.20: Das DataGrid-Steuerelement mit Paging und Sortierung

22.4.4 Das Repeater-Steuerelement

Das `Repeater`-Steuerelement bietet im Vergleich zum `DataGrid`-Steuerelement sehr wenig eingebaute Funktionalität, erlaubt aber eine größere Flexibilität bei der Darstellung der Daten. Diese Darstellung legen Sie durch Vorlagen fest, die Sie als untergeordnete HTML-Elemente in der HTML-Ansicht des Steuerelements erstellen; durch ein vorgegebenes Namensschema werden sie den Elementen zugeordnet.

Allerdings stellt das `Repeater`-Steuerelement nichts dar, solange Sie nicht mindestens eine Vorlage erstellen. Diese Vorlage legt das normale Aussehen der einzelnen Einträge fest und besteht aus einem *ItemTemplate*-Element. Zusätzlich können Sie die Gestaltung des Element-Übergangs durch ein *SeparatorTemplate*-Element sowie Kopf- und Fußzeilen in *HeaderTemplate*- bzw. *FooterTemplate*-Elementen definieren; den Zebrastreifeneffekt des folgenden Beispiels erreichen Sie durch Anlegen eines *AlternatingItemTemplate*-Elements.

Codebeispiel – Eine Zebrastreifentabelle mit dem Repeater-Steuerelement

Die Seite *Repeater1.aspx* im Beispielprojekt *WfRepeater* implementiert eine Zebrastreifentabelle wie in Abschnitt »Codebeispiel – Elemente und Attribute einfügen« auf Seite 773, benutzt dazu aber ein `Repeater`-Steuerelement: Abbildung 22.21 zeigt das Resultat.

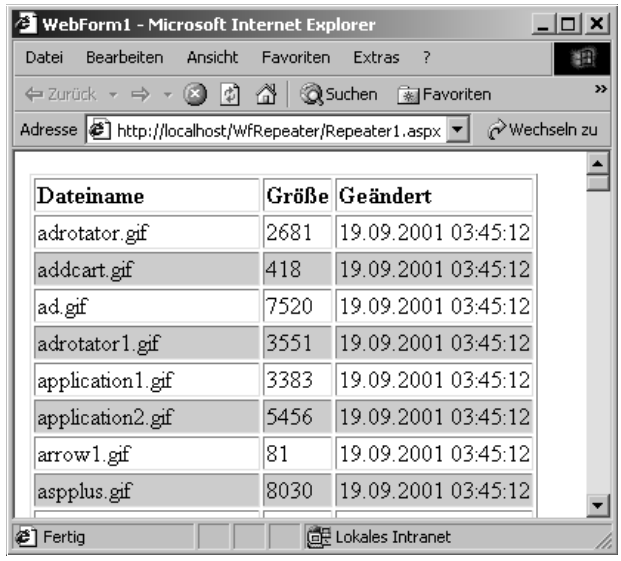


Abbildung 22.21: Eine Zebrastreifentabelle mit dem Repeater-Steuerelement

Setzen Sie ein `Repeater`-Steuerelement in einen neuen Web Form-Entwurf, wechseln Sie in die HTML-Ansicht und ergänzen Sie das *form*-Element wie folgt:

```
<form id="Form1" method="post" runat="server">
  <asp:repeater id="Repeater1" runat="server">
    <HeaderTemplate>
      <table border="1">
        <tr>
          <td><b>Dateiname</b></td>
          <td><b>Größe</b></td>
          <td><b>Geändert</b></td>
        </tr>
    </HeaderTemplate>
    <ItemTemplate>
      <tr>
        <td>
          <%# DataBinder.Eval(Container.DataItem, "Dateiname") %>
        </td>
        <td>
          <%# DataBinder.Eval(Container.DataItem, "Größe") %>
        </td>
        <td>
          <%# DataBinder.Eval(Container.DataItem, "Geändert") %>
        </td>
      </tr>
    </ItemTemplate>
    <AlternatingItemTemplate>
      <tr>
        <td bgcolor="lightblue">
          <%# DataBinder.Eval(Container.DataItem,
                                "Dateiname") %>
        </td>
        <td bgcolor="lightblue">
          <%# DataBinder.Eval(Container.DataItem,
                                "Größe") %>
        </td>
        <td bgcolor="lightblue">
          <%# DataBinder.Eval(Container.DataItem,
                                "Geändert") %>
        </td>
      </tr>
    </AlternatingItemTemplate>
    <FooterTemplate>
      </table>
    </FooterTemplate>
  </asp:repeater>
</form>
```

Den Code zum Generieren der Einträge und zur Anbindung an das Steuerelement zeigt das Listing auf Seite 919.

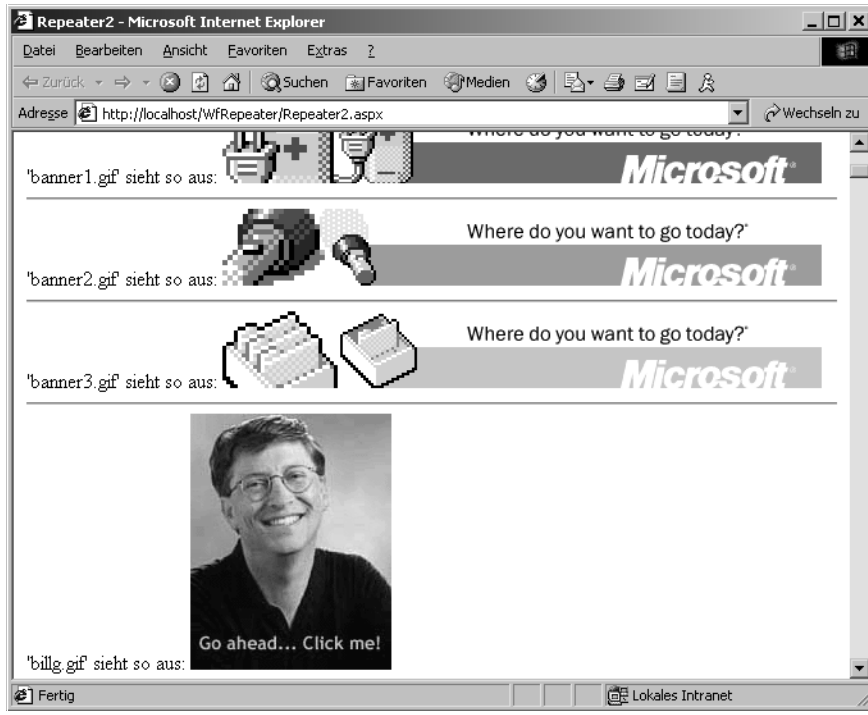
Die Vorlage *HeaderTemplate* definiert den Kopf des zu erzeugenden Dokuments als Anfang und Kopfzeile einer dreispaltigen Tabelle, die durch die Vorlage *FooterTemplate* abgeschlossen wird.

Die beiden Vorlagen *ItemTemplate* und *AlternatingItemTemplate* definieren das Aussehen der einzelnen Einträge – in diesem Fall als Zeilen einer Tabelle. Dabei benutzt das `Repeater`-Steuerelement diese beiden Vorlagen automatisch abwechselnd, sodass sich der gewünschte Zebrastreifeneffekt ergibt.

Codebeispiel – Datendarstellung ohne Tabelle

Die Seite *Repeater2.aspx* zeigt, wie umfangreich die Gestaltungsmöglichkeiten des `Repeater`-Steuerelements sind: Es stellt die einzelnen Einträge nicht in einer Tabelle dar, sondern in einem völlig frei gewählten Format – in diesem Fall als Liste mit Trennung durch einen Querstrich. Das Resultat ist in Abbildung 22.22 zu sehen.

Abbildung 22.22: Datendarstellung ohne Tabelle



Setzen Sie ein `Repeater`-Steuerelement in einen neuen Web Form-Entwurf, wechseln Sie in die HTML-Ansicht und ergänzen Sie das *form*-Element wie folgt:

```
<form id="Repeater2" method="post" runat="server">
  <asp:Repeater id="Repeater1" runat="server">
```

```
    <ItemTemplate>
      '<%# DataBinder.Eval(Container.DataItem, "Dateiname") %>'
        sieht so aus:
      <asp:Image runat=server
        ImageUrl=<%# Pfad + "\\" +
          DataBinder.Eval(Container.DataItem, "Dateiname") %>
          ID="Image1"/>
    </ItemTemplate>
    <SeparatorTemplate>
      <hr />
    </SeparatorTemplate>
  </asp:Repeater>
</form>
```

Den Code zum Generieren der Einträge und zur Anbindung an das Steuerelement zeigt das Listing auf Seite 920.

Die *ItemTemplate*-Vorlage beschränkt sich auf die Ausgabe des Dateinamens, eines kurzen Textes, und die Anzeige der Grafik in einem `Image`-Webserver-Steuerelement. Ebenso kurz fällt die *SeparatorTemplate*-Vorlage aus: sie schreibt das *<hr/>*-Tag zur Abgrenzung der einzelnen Einträge.

Gerade die Kürze des Codes lässt die Gestaltungsmöglichkeiten erkennen: sie sind nur durch den Sprachumfang von HTML begrenzt, denn das `Repeater`-Steuerelement erzeugt selbst kein HTML, das der Ausgabe wie beim `DataGrid`- und `DataList`-Steuerelement eine Tabellenform aufzwingen würde. Es wendet nur die Vorlagen an, deren einzige Einschränkung daraus besteht, dass das Endergebnis gültiges HTML sein muss.

22.4.5 Das DataList-Steuerelement

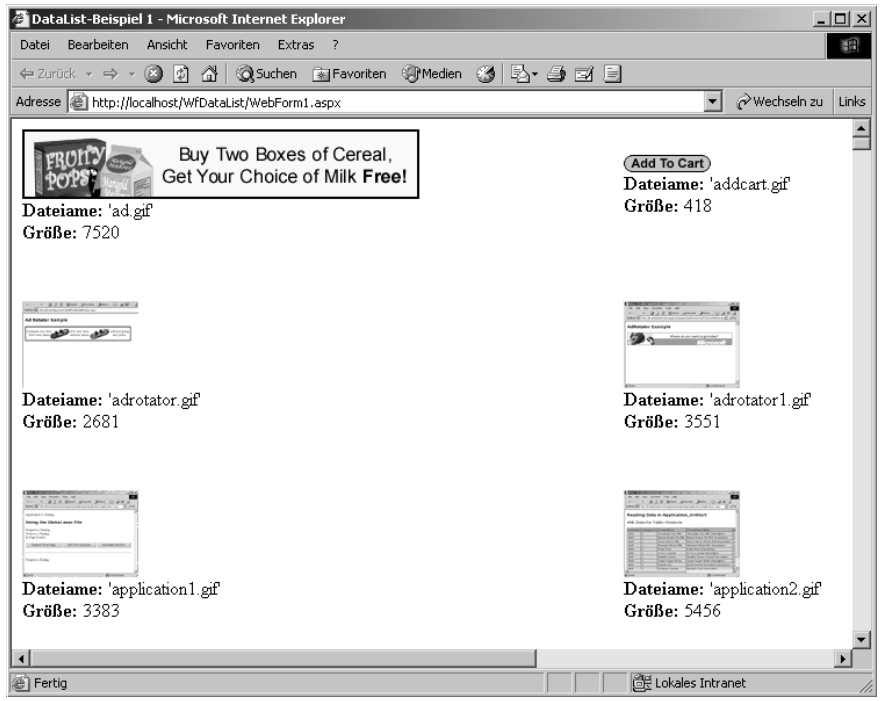
Das `DataList`-Steuerelement bietet als einziges der listengebundenen Webserver-Steuerelemente die Möglichkeit, Einträge auf mehrere Spalten zu verteilen. Dazu generiert es eine HTML-Tabelle und schreibt jeden Eintrag in eine Zelle, wobei die Anzahl der Zellen pro Zeile einstellbar ist. Für einfache Listen können Sie auch das Layout *Fluss* benutzen, dann generiert das Steuerelement statt einer Tabelle für jeden Eintrag ein *span*-Element.

Zusätzlich zur Arbeit in der HTML-Ansicht haben Sie die Möglichkeit, die Vorlagen in einem speziellen Designer zu bearbeiten (Befehl VORLAGE BEARBEITEN des Kontextmenüs). Und ebenso wie beim `DataGrid`-Steuerelement können Sie aus vorgefertigten Formatierungen wählen (Befehl AUTOMATISCHE FORMATIERUNG des Kontextmenüs).

Codebeispiel – Listeneinträge auf mehrere Spalten verteilen

Die Seite *WebForm1.aspx* des Beispielprojekts *WfDataList* demonstriert die eingebaute Fähigkeit des `DataList`-Steuerelements, Listeneinträge auf mehrere Spalten zu verteilen – Abbildung 22.23 zeigt das Resultat.

Abbildung 22.23: Listeneinträge auf mehrere Spalten verteilen



Setzen Sie ein `DataList`-Steuerelement auf eine neue Web Forms-Seite und wählen Sie im Kontextmenü des Steuerelements den Befehl EIGENSCHAFTENGENERATOR. Legen Sie dort die Anzahl der Spalten auf *2* fest, wählen Sie als RICHTUNG *Vertikal* und klicken Sie OK.

Ergänzen Sie dann in der HTML-Ansicht das *form*-Element wie folgt:

```
<form id="Form1" method="post" runat="server">
  <asp:datalist id="DataList1" style="Z-INDEX: 101; LEFT: 8px;
    POSITION: absolute; TOP: 8px" runat="server"
    RepeatDirection="Horizontal" RepeatColumns="2">
    <ItemTemplate>
      <div style="PADDING-BOTTOM: 3em; PADDING-RIGHT: 3em;">
        <asp:Image id=Image1 runat="server" ImageUrl='<%# Pfad + "\\" +
          DataBinder.Eval(Container.DataItem, "Dateiname") %>'>
        </asp:Image><br/>
        <b>Dateiame:</b>
          '<%# DataBinder.Eval(Container.DataItem, "Dateiname") %>'<br/>
        <b>Größe:</b> <%# DataBinder.Eval(Container.DataItem, "Größe") %>
```

```
      </div>
    </ItemTemplate>
  </asp:datalist>
</form>
```

Den Code zum Generieren der Einträge und zur Anbindung an das Steuerelement zeigt das Listing auf Seite 919.

Der Code beschränkt sich auf die Definition einer *ItemTemplate*-Vorlage für die einzelnen Einträge. Auffällig ist dabei allerdings die Verpackung des eigentlichen Eintragsinhalts in einem *div*-Element. Sie ist nötig, um den Einträgen ein *style*-Attribut zur Oberflächengestaltung zuweisen zu können, denn die *table*- und *td*-Elemente des `DataList`-Steuerelements sind nicht zugänglich.

Weil das `DataList`-Steuerelement die Einträge nicht wie das `DataGrid`-Steuerelement auf mehrere Tabellenzellen verteilt, sorgen *
*-Tags für Zeilenumbrüche innerhalb der Zelle.

Codebeispiel – Detailanzeige in einer Tabelle

Häufig ist die Listendarstellung der Daten nur der Einstieg in die Detailansicht eines Eintrags. So selektiert der Benutzer zum Beispiel einen Kunden anhand eines Kürzels in einer Auswahlliste und sieht sich dann dessen detaillierte Informationen an: die klassische Master/Detail-Ansicht.

Die Seite *WebForm2.aspx* demonstriert, wie das `DataList`-Steuerelement solche Oberflächen-Features durch `LinkButton`-Webserver-Steuerelemente unterstützt. In Abbildung 22.24 sehen Sie das Beispiel in Aktion.

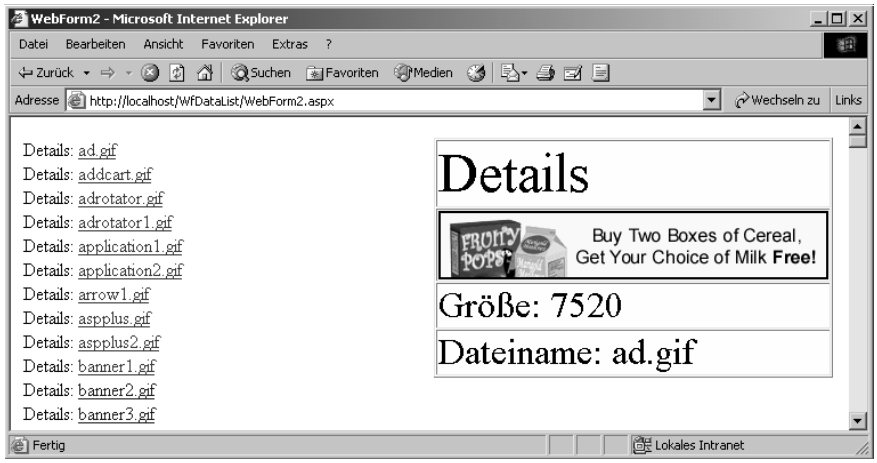
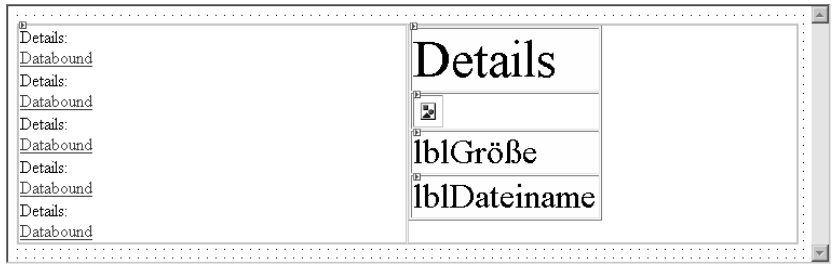


Abbildung 22.24: Detailanzeige in einer Tabelle

Die Darstellung in Abbildung 22.24 besteht aus drei Tabellen: eine zeigt die Liste und wird vom `DataList`-Steuerelement generiert, eine zeigt die Details, und beide sind zur Positionierung von einer dritten umschlossen. In Abbildung 22.25 sehen Sie die Entwurfsansicht des Programms.

Abbildung 22.25: Die Entwurfsansicht des Programms



Verschachteln Sie zum Nachprogrammieren zwei HTML-Steuerelemente vom Typ `Table` (Registerkarte HTML der TOOLBOX) und passen Sie sie wie in Abbildung 22.25 zu sehen an. Geben Sie dann der Detailtabelle die `id` *DetailTabelle*, und tragen Sie in ihre oberste Zelle den Text *Details* ein.

Setzen Sie dann in die Detailtabelle noch ein `Image`- (`id` = *Grafik)* und zwei `Label`-Webserver-Steuerelemente (`id` = *lblGröße* bzw. *lblDateiname*) und schließlich noch ein `DataList`-Steuerelement in die umgebende Tabelle.

Wechseln Sie anschließend in die HTML-Ansicht und ergänzen Sie dort das *form*-Element wie folgt:

```
<form id="WebForm2" method="post" runat="server">
  <TABLE id="Table1" style="Z-INDEX: 101; LEFT: 8px; WIDTH: 724px;
    POSITION: absolute; TOP: 16px; HEIGHT: 96px" cellSpacing="1"
    cellPadding="1" width="724">
    <tr>
      <td width="50%">
        <asp:datalist id="DataList1" runat="server" Width="71px"
          OnSelectedIndexChanged="ZeigeDetails" DataKeyField="Dateiname"
          Height="160px">
          <ItemTemplate>
            Details:
            <asp:LinkButton runat=server CommandName="Select" Text=
             '<%# DataBinder.Eval(Container.DataItem, "Dateiname") %>' />
          </ItemTemplate>
        </asp:datalist></td>
      <td width="50%" valign="top">
        <TABLE id="DetailTabelle" style="Z-INDEX: 101" cellSpacing="1"
          cellPadding="1" border="1" runat="server">
          <TR>
            <TD style="FONT-SIZE: 3em;
              BACKGROUND-COLOR: ivory">Details</TD>
          </TR>
          <TR>
```

```
            <TD id="GrafikAnzeige"><asp:image id="Grafik"
              runat="server"></asp:image></TD>
          </TR>
          <TR>
            <TD style="FONT-SIZE: 2em; BACKGROUND-COLOR: ivory">
              <asp:Label id="lblGröße"
                runat="server">lblGröße</asp:Label></TD>
          </TR>
          <TR>
            <TD style="FONT-SIZE: 2em; BACKGROUND-COLOR: ivory">
              <asp:Label id="lblDateiname"
                runat="server">lblDateiname</asp:Label></TD>
          </TR>
        </TABLE>
      </td>
    </tr>
  </TABLE>
</form>
```

Die umgebende Tabelle positioniert hier die beiden darzustellenden Tabellen durch die *width*-Attribute ihrer *td*-Elemente; außerdem sorgt das *valign*-Attribut dafür, dass die Detailtabelle am oberen Seitenrand erscheint.

Bei der Detailtabelle ist das Hinzufügen des *runat*-Attributs entscheidend, da diese Tabelle vom serverseitigen Code sichtbar und unsichtbar geschaltet wird. Die Zellen zur Textdarstellung wurden außerdem um *style*-Attribute ergänzt.

Das `DataList`-Steuerelement enthält in seiner *ItemTemplate*-Vorlage ein `LinkButton`-Webserver-Steuerelement, das zur Laufzeit einen Hyperlink generiert. Das Anklicken dieses Hyperlinks führt aber nicht zu einem weiteren Dokument, sondern löst ein Ereignis auf dem Server aus: das `SelectedIndexChanged`-Ereignis, denn das *CommandName*-Attribut des `LinkButton`-Webserver-Steuerelements hat den Wert `"Select"`.

Dazu legt das *OnSelectedIndexChanged*-Attribut des `DataList`-Steuerelements fest, welche Methode dieses Ereignisses bearbeiten soll – hier also `ZeigeDetails()`. Zusätzlich bestimmt das *DataKeyField*-Attribut ein Feld der Datenmenge, durch das der selektierte Eintrag eindeutig identifiziert werden kann.

Die Methode `ZeigeDetails()` befindet sich in der CodeBehind-Datei.

```
protected void ZeigeDetails(Object sender, EventArgs e)
{
  string Dateiname =
    DataList1.DataKeys[DataList1.SelectedIndex].ToString();
  DataTable t = (DataTable)Session["Daten"];
  DataRow[] r = t.Select("Dateiname='" + Dateiname + "'");
  Grafik.ImageUrl = Pfad + "\\" + Dateiname;
```

```
  lblGröße.Text =  "Größe: " + r[0]["Größe"].ToString();
  lblDateiname.Text = "Dateiname: " + Dateiname;
  DetailTabelle.Visible = true;
}//ZeigeDetails
```

`ZeigeDetails()` ermittelt zuerst den Namen der selektierten Datei. Die `DataKeys`-Auflistung des `DataList`-Steuerelements wird dazu automatisch mit den Werten der im *DataKeyField*-Attribut angegebenen Spalte gefüllt, und `SelectedIndex` dient als Index in diese Auflistung.

Alle auszugebenden Detailangaben lassen sich direkt aus dem Dateinamen zusammensetzen – bis auf die Dateigröße, für die noch einmal auf die Tabelle zugegriffen werden muss.

Die Detailtabelle wird erst beim Zeigen der Details sichtbar geschaltet; beim Programmstart soll sie mangels Daten unsichtbar sein, was die folgende Zeile in der `Page_Load()`-Methode erledigt:

```
private void Page_Load(object sender, System.EventArgs e)
{
  if (!IsPostBack)
  {
    DetailTabelle.Visible = false;
  // …
  }
}//Page_Load
```

Den restlichen Code zum Generieren der Einträge und zur Anbindung an das Steuerelement zeigt das Listing auf Seite 920.

Codebeispiel – Detailanzeige in einer SelectedItemTemplate-Vorlage

Der Code für die Seite *WebForm3.aspx* implementiert die gleiche Funktionalität wie die vorhergehende, allerdings benutzt er dazu eine *SelectedItemTemplate*-Vorlage statt einer Detailtabelle. Durch diesen eingebauten Mechanismus kommt das Programm mit weniger Code aus, schränkt damit andererseits natürlich seine Formatierungsmöglichkeiten ein.

Meistens dürften die Formatierungsmöglichkeiten der *SelectedItemTemplate*-Vorlage aber genügen – wie Abbildung 22.26 zeigt, reizt das Beispiel sie bei Weitem nicht aus.

Für dieses Beispiel benötigen Sie ein `DataList`-Steuerelement mit folgenden Ergänzungen in der HTML-Ansicht (die *FooterStyle*- und *FooterStyle*-Elemente wurden mit dem Eigenschaftengenerator erzeugt):

```
<form id="WebForm3" method="post" runat="server">
  <asp:datalist id="DataList1" style="Z-INDEX: 101; LEFT: 16px;
```

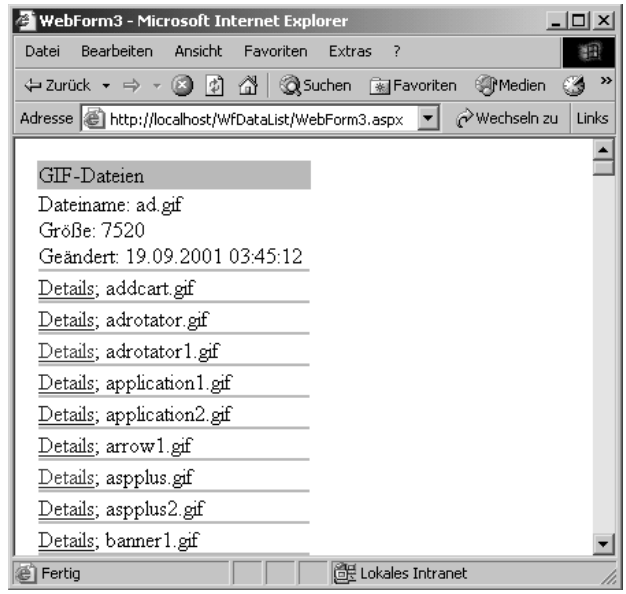


Abbildung 22.26: Detailanzeige in einer SelectedItemTemplate-Vorlage

```
    POSITION: absolute; TOP: 16px" runat="server">
    <HeaderTemplate>
      GIF-Dateien
    </HeaderTemplate>
    <ItemTemplate>
      <div style="BORDER-BOTTOM: lightsalmon thin solid">
        <asp:LinkButton id="button1" Text="Details"
          CommandName="select" runat="server" />
        <%# DataBinder.Eval(Container.DataItem, "Dateiname") %>
      </div>
    </ItemTemplate>
    <SelectedItemTemplate>
      <div style="BORDER-BOTTOM: lightsalmon thin solid">
        Dateiname:
        <%# DataBinder.Eval(Container.DataItem, "Dateiname") %>
        <br />
        Größe:
        <%# DataBinder.Eval(Container.DataItem, "Größe") %>
        <br />
        Geändert:
        <%# DataBinder.Eval(Container.DataItem, "Geändert") %>
      </div>
    </SelectedItemTemplate>
    <FooterTemplate>
       
    </FooterTemplate>
    <FooterStyle BackColor="LightSalmon"></FooterStyle>
    <HeaderStyle BackColor="LightSalmon"></HeaderStyle>
  </asp:datalist>
</form>
```

Die *HeaderTemplate*- und *FooterTemplate*-Elemente sorgen in Zusammenarbeit mit den *HeaderStyle*- und *FooterStyle*-Elementen für farbige Streifen in Kopf- und Fußzeile. Da die Fußzeile keinen Text enthält, benötigt sie ein `" "` (nonbreaking space) damit der farbige Streifen trotzdem erscheint.

Das *ItemTemplate*-Element verpackt seine Daten auch in diesem Beispiel wieder in einem zusätzlichen *div*-Element, um durch dessen *style*-Attribut das gewünschte Erscheinungsbild zu realisieren. Der eigentliche Inhalt der Tabellenzelle besteht wieder aus einem `LinkButton`-Webserver-Steuerelement und dem Dateinamen.

Das `LinkButton`-Webserver-Steuerelement löst auch hier ein `SelectedIndexChanged`-Ereignis aus, allerdings erfolgt dessen Verknüpfung mit der entsprechenden Methode auf dem Server diesmal nicht durch das *OnSelectedIndexChanged*-Attribut des `DataList`-Steuerelements. Stattdessen benutzt das Programm die bei Windows-Formularen üblichen Delegaten.

```
private void DataList1_SelectedIndexChanged(
  object sender, System.EventArgs e)
{
  BindeDataList();
}
```

Da die Methode nicht auf die zugrunde liegende Tabelle zugreift, kommt das `DataList`-Steuerelement ohne *DataKeyField*-Attribut aus.

Den Code zum Generieren der Einträge und zur Anbindung an das Steuerelement zeigt das Listing auf Seite 920.

Codebeispiel – Daten ändern in einer EditItemTemplate-Vorlage

Auf der Beispielseite *WebForm4.aspx* bietet das `DataList`-Steuerelement durch seine *EditItemTemplate*-Vorlage eine Möglichkeit, die angezeigten Daten zu ändern. Diese Vorlage wird genauso eingebunden wie die *SelectedItemTemplate*-Vorlage des letzten Beispiels. Abbildung 22.27 zeigt das laufende Programm.

Als Basis können Sie das `DataList`-Steuerelement aus dem letzten Beispiel übernehmen. Passen Sie dann das *form*-Element folgendermaßen an:

```
<form id="WebForm4" method="post" runat="server">
  <asp:datalist id="DataList1" style="Z-INDEX: 101; LEFT: 8px;
    POSITION: absolute; TOP: 8px" runat="server"
    OnCancelCommand="DataList1_Cancel"
    OnEditCommand="DataList1_Edit"
    OnUpdateCommand="DataList1_Update">
    <HeaderTemplate>
      GIF-Dateien
    </HeaderTemplate>
```

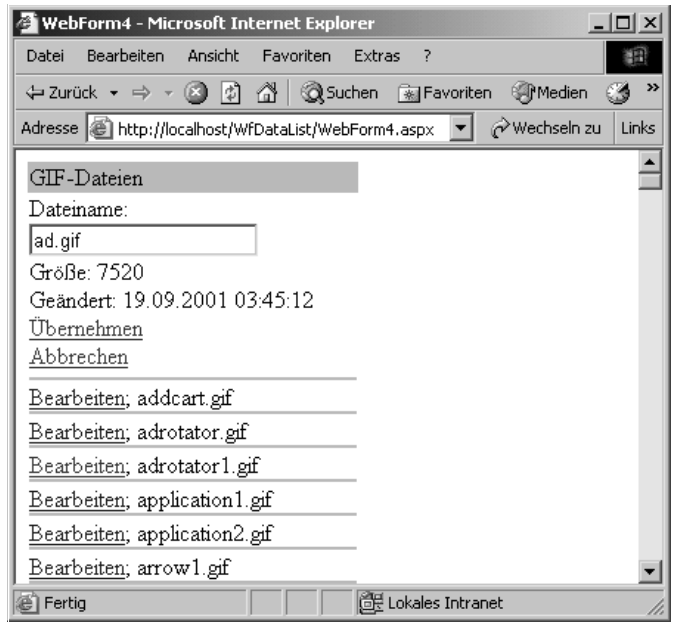


Abbildung 22.27: Daten ändern in einer EditItem-Template-Vorlage

```
<ItemTemplate>
  <div style="BORDER-BOTTOM: lightsalmon thin solid">
    <asp:LinkButton CommandName="edit" runat="server"
      id="button1" Text="Bearbeiten" />
    <%# DataBinder.Eval(Container.DataItem, "Dateiname") %>
  </div>
</ItemTemplate>
<EditItemTemplate>
  <table style="BORDER-BOTTOM: lightsalmon thin solid" width="100%">
    <tr>
      Dateiname:
      <input id="txtDateiname" type="text" runat="server" value=
        '<%# DataBinder.Eval(Container.DataItem, "Dateiname") %>' />
      <br />
      Größe:
      <%# DataBinder.Eval(Container.DataItem, "Größe") %>
      <br />
      Geändert:
      <%# DataBinder.Eval(Container.DataItem, "Geändert") %>
      <br />
      <asp:LinkButton CommandName="Update" Runat="server"
        ID="Linkbutton1">Übernehmen</asp:LinkButton>
      <br />
      <asp:LinkButton CommandName="Cancel" Runat="server"
        ID="Linkbutton2">Abbrechen</asp:LinkButton>
    </tr>
  </table>
</EditItemTemplate>
<FooterTemplate>
   
</FooterTemplate>
```

```
    <FooterStyle BackColor="LightSalmon"></FooterStyle>
    <HeaderStyle BackColor="LightSalmon"></HeaderStyle>
  </asp:datalist>
</form>
```

Die *HeaderTemplate*- und *FooterTemplate*-Elemente bleiben unverändert, beim *ItemTemplate*-Element ändern sich die Werte der *CommandName*- und *Text*-Attribute des `LinkButton`-Webserver-Steuerelements. Das *SelectedItemTemplate*-Element wird durch ein *EditItemTemplate*-Element ersetzt, das ein Textfeld zur Bearbeitung des Dateinamens sowie je ein `LinkButton`-Webserver-Steuerelement zum Übernehmen und zum Verwerfen der Änderungen enthält.

Die Anbindung der drei `LinkButton`-Steuerelemente an ihre Server-Methoden erfolgt über entsprechende Attribute des `DataList`-Steuerelements. Man könnte hier allerdings genauso gut wie im vorhergehenden Beispiel eine auf Delegaten basierende Lösung implementieren.

Hier sind die drei Methoden zur Behandlung der Ereignisse in der CodeBehind-Datei:

```
protected void DataList1_Cancel(
  Object sender, DataListCommandEventArgs e)
{
  DataList1.EditItemIndex = -1;
  BindeDataList();
}
protected void DataList1_Edit(Object sender, DataListCommandEventArgs e)
{
  DataList1.EditItemIndex = e.Item.ItemIndex;
  BindeDataList();
}
protected void DataList1_Update(
  Object sender, DataListCommandEventArgs e)
{
  // Hier gehört der Code fürs Update hin ...
  String Dateiname =
    ((HtmlInputText)e.Item.FindControl("txtDateiname")).Value;
  DataList1.EditItemIndex = -1;
  BindeDataList();
}
```

Um einen Eintrag in der *EditItemTemplate*-Vorlage editieren zu können, genügt das Setzen der `EditItemIndex`-Eigenschaft des `DataList`-Steuerelements auf den entsprechenden Index; dementsprechend genügt das Setzen dieser Eigenschaft auf *–1*, um das Editieren zu beenden.

Interessant ist hier noch das Auslesen des neuen Werts in der `DataList1_Update()`-Methode: Der Wert ist nur über das entsprechende Steuerelement zu erreichen.

Was jetzt noch fehlt, ist der Code zum Generieren der Einträge und zur Anbindung an das Steuerelement; die entsprechende Implementation zeigt das Listing auf Seite 920.

Codebeispiel – Mit clientseitigem Skript eine Sicherheitsabfrage vor dem Löschen implementieren

Das Löschen von Daten soll der Benutzer häufig explizit bestätigen, um Datenverluste durch Bedienungsfehler zu verhindern. Dieses Beispiel zeigt, wie man mit clientseitigem Skript eine Sicherheitsabfrage vor dem Löschen implementiert.

Die Sicherheitsabfrage ist mit einer Zeile JavaScript-Code implementiert – das Problem ist, ein `LinkButton`-Steuerelement zur Ausführung des clientseitigen Skripts zu überreden: Dazu bekommt das Steuerelement bei seiner Erzeugung dynamisch ein `OnClick`-Attribut mit dem entsprechenden JavaScript-Code hinzugefügt.

Abbildung 22.28 zeigt das Programm in Aktion.

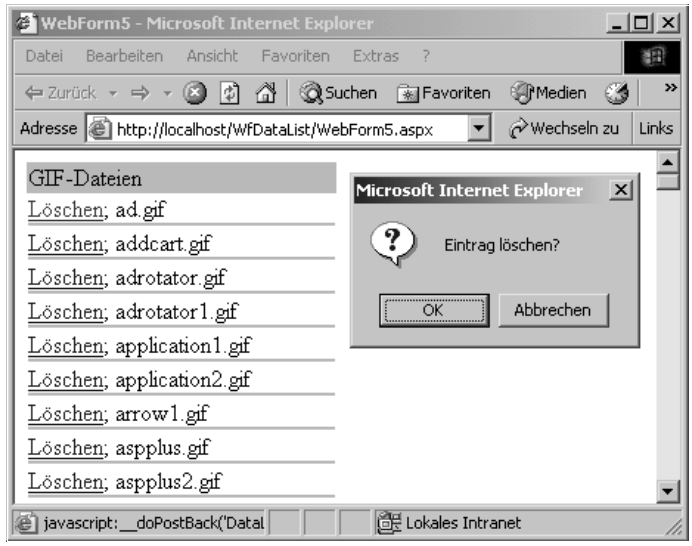


Abbildung 22.28: Sicherheitsabfrage vor dem Löschen

Da Oberflächengestaltung und Programmierung des `DataList`-Steuerelements hier sehr ähnlich wie im vorangehenden Beispiel verlaufen, kopieren Sie am einfachsten von dort auf eine neue Web Form. Ändern Sie dann in der HTML-Ansicht den Inhalt des *form*-Elements wie folgt:

```
<form id="WebForm5" method="post" runat="server">
  <asp:datalist id="DataList1"
    OnDeleteCommand="DataList1_DeleteCommand"
    OnItemCreated="DataList1_ItemCreated"
```

```
    style="Z-INDEX: 101; LEFT: 8px; POSITION: absolute; TOP: 8px"
    runat="server">
    <HeaderTemplate>
      GIF-Dateien
    </HeaderTemplate>
    <ItemTemplate>
      <div style="BORDER-BOTTOM: lightsalmon thin solid">
        <asp:LinkButton CommandName="delete" runat="server"
          id="btnDelete" Text="Löschen" />
        <%# DataBinder.Eval(Container.DataItem, "Dateiname") %>
      </div>
    </ItemTemplate>
    <FooterTemplate>
       
    </FooterTemplate>
    <FooterStyle BackColor="LightSalmon"></FooterStyle>
    <HeaderStyle BackColor="LightSalmon"></HeaderStyle>
  </asp:datalist>
</form>
```

Gegenüber der Originalversion ergeben sich also folgende Veränderungen:

- Das *EditItemTemplate*-Element ist entfallen.
- Im *ItemTemplate*-Element erhält das `LinkButton`-Webserver-Steuerelement neue Werte für das *CommandName*-, das *id*- und das *Text*-Attribut.
- Im *datalist*-Element werden die vorhandenen Attribute zur Verbindung mit Ereignisbehandlungsmethoden durch *OnDeleteCommand*- und *OnItemCreated*-Attribute ersetzt.

Neben dem in dem Listing auf Seite 920 gezeigten Code zum Generieren der Einträge und zur Anbindung an das Steuerelement enthält die CodeBehind-Datei noch diese beiden Ereignisbehandlungsmethoden:

```
protected void DataList1_ItemCreated(object sender,
  System.Web.UI.WebControls.DataListItemEventArgs e)
{
  if (e.Item.ItemType == ListItemType.Item)
  {
    WebControl btn = (WebControl)e.Item.FindControl("btnDelete");
    btn.Attributes.Add ("onclick",
      "return confirm('Eintrag löschen?');");
  }
}

protected void DataList1_DeleteCommand(object source,
  System.Web.UI.WebControls.DataListCommandEventArgs e)
{
  //Hier gehört der Code fürs Löschen hin ...
}
```

Die Methode `DataList1_ItemCreated()` wird beim Erzeugen jedes Eintrags (außer Kopf- und Fußzeile) aufgerufen und verbindet dessen `LinkButton`-Webserver-Steuerelement mit einer Zeile JavaScript für die Sicherheitsabfrage. Wenn der Benutzer diese Sicherheitsabfrage verneint, liefert der JavaScript-Code `False` zurück und unterbindet damit das Postback der Web Form. Und ohne Postback wird natürlich die Methode `DataList1_DeleteCommand()` auf dem Server nicht ausgeführt.

Da die Methode `DataList1_DeleteCommand()` den Eintrag nicht wirklich löscht, benutzen Sie zum Testen am besten einen Haltepunkt.

23 Webdienste

Webdienste sind Programme für Programme. Anders als beim Remoting, das die Bestandteile eines Programms verbindet, integrieren Webdienste mehrere voneinander unabhängige Programme. Dementsprechend verfügen Webdienste auch über Standards zur Beschreibung von Programmschnittstellen und Festlegungen zur Suche nach Servern, die einen bestimmten Dienst anbieten.

Die Triebfeder hinter der Entwicklung von Webdiensten liegt in der immer weiter zunehmenden geschäftlichen Integration. So sollen Kundensysteme im Lieferantensystem vollautomatisch Bestellungen aufgeben können oder vollautomatische Auktionen von Waren oder Aufträgen möglich sein. Ein anderes Anwendungsbeispiel wäre ein Reisebüroverbund, dessen Kunden über ein einziges Programm die Angebote aller angeschlossenen Reisebüros nutzen können. Dabei integrieren die Webdienste auch die unterschiedlichen EDV-Plattformen der Reisebüros. Auch Großunternehmen mit heterogenen EDV-Plattformen profitieren von dieser Form der Enterprise Application Integration (EAI). Und nicht zuletzt sind Webdienste die technische Grundlage völlig neuer Geschäftsmodelle, wie Software als Dienstleistung.

Webdienste sind eine Weiterentwicklung der ursprünglichen manuellen Nutzung des Internet. Deshalb basieren sie auf denselben, von allen Plattformen unterstützten Standards. Das ist ihr entscheidender Vorteil gegenüber anderen Standards wie IIOP (Internet InterORB Protocol) oder DCOM (Distributed COM), die sich wegen technischer Probleme und »politischer« Überlegungen nie wirklich durchsetzen konnten.

Der Vorteil der Webdienste gegenüber Webseiten ist, dass Webdienste nicht die Programmoberfläche bestimmen, sondern nur die benutzten Protokolle. Ein Client-Programm kann deshalb genauso gut eine Unix- wie eine Windows-Oberfläche haben oder auch auf einem Handy oder Palmtop laufen.

Ebenso frei können auch Betriebssystem und Implementation des Servers gewählt werden. Oft werden dafür vorhandene Webserver wie Apache oder Microsofts IIS »aufgebohrt«, grundsätzlich kann aber jedes Programm Webdienst spielen, das die entsprechenden Protokolle einhält.

Allerdings sind Webdienste selbst für Internet-Verhältnisse noch relativ jung. Es gibt deshalb leider noch diverse Probleme mit der Interoperabilität, und es fehlt zum Beispiel die Transaktionsunterstützung. Auch die Sicherheit ist nicht eingebaut, lässt sich für HTTP-basierte Webdienste allerdings leicht durch HTTPS realisieren.

Unter all diesen faszinierenden technischen Möglichkeiten schimmern natürlich auch immer wirtschaftliche Interessen durch. Denn wer es schafft, jetzt Software-Patente, halb offene Protokolle oder Dienstleistungen als de facto-Standards durchzusetzen, hat eine Lizenz zum Gelddrucken und kann die Konkurrenz in Schach halten. Das bekannteste Beispiel ist sicher Microsofts Passport-Dienst, die meisten Versuche finden aber von der Öffentlichkeit zunächst unbemerkt statt. So verzichtete IBM erst auf öffentlichen Druck auf die Geltendmachung »plötzlich« auftauchender Software-Patente in seinen Beiträgen zu ebXML, wenig später wiederholte sich das Ganze in Zusammenarbeit mit Microsoft bei ihrem Standardisierungsvorschlag WSDL. (Zu den diversen Akronymen siehe nächster Abschnitt.)

Offen ist dagegen der Ausgang anderer Standardisierungsinitiativen von Microsoft und IBM, wie WS-Security, das die Verschlüsselung und digitale Signatur von SOAP-Nachrichten regelt und ebenfalls juristische Kuckuckseier enthält. Die logische Fortsetzung dieser Aktivitäten ist das völlige Umgehen der bestehenden Standardisierungsgremien: Die jüngst aus dem Nichts erschienene Web Services Interoperability Organization (WS-I) ist ein gutes Beispiel dafür.

Die Patent-Probleme an sich sind nicht neu, man denke an Amazons »1 Click Technology« oder den Versuch von British Telecom, ein Patent auf Hyperlinks durchzusetzen. Erfreulicherweise haben die Inhaber derartiger »Patente« bisher meist den Kürzeren gezogen.

23.1 Akronyme, Akronyme ...

Webdienste benutzen im Allgemeinen HTTP (Hypertext Transfer Protocol) zum Transport der Daten. Andere Implementierungen, zum Beispiel auf Basis von FTP (File Transfer Protocol) oder TCP (Transmission Control Protocol), sind aber auch möglich.

Verpackt werden die Daten in XML (Extended Markup Language). Den genauen Aufbau der XML-Nachrichten und -Elemente legt SOAP (Simple Object Access Protocol) fest. Mit SOAP konkurriert der einfachere Standard XML-RPC (XML Remote Procedure Calls, von UserLand, Microsoft und Developmentor), eigentlich ist XML-RPC aber eher ein Vorfahre von SOAP.

Die Beschreibung der Programmierschnittstellen erfolgt in WSDL (Web Services Description Language, von Microsoft, IBM and Ariba). Diese Sprache hieß bei Microsoft früher SDL (Services Description Language).

Damit andere den Webdienst auch finden, existiert UDDI (Universal Description, Discovery and Integration, von Microsoft, IBM and Ariba). UDDI umfasst sowohl die Registraturen zum Veröffentlichen von Webdiensten als auch die Abfragesprache der Registraturen. Diese Sprache ist wiederum SOAP-basiert. Neben UDDI existiert auch der einfachere Standard DISCO (Discovery) von Microsoft, der keine Registraturen umfasst, sondern lokale Webdienste auflistet.

Ein mit SOAP konkurrierender Standard mit speziellem Anwendungs-Schwerpunkt ist ebXML (Electronic Business XML), das andere Vorstellungen sowohl von der Verpackung der Daten als auch von den Protokollen zu ihrer Beschreibung und Lokalisierung hat. ebXML wird von OASIS (Organization for the Advancement of Structured Information Standards) und UN/CEFACT (United Nations body for Trade Facilitation and Electronic Business) unterstützt.

Zusammenfassend lässt sich sagen, dass Webdienste zurzeit eher eine Beispiel-Implementation sind. Nur SOAP ist vom World Wide Web Consortium (W3C) abgesegnet. Übrigens existiert dort schon seit einigen Jahren eine »Web Services Activity« mit dem Ziel, den endgültigen Protokollsatz für Webdienste zu entwickeln – das Entwicklerleben bleibt also auch in Zukunft interessant.

23.2 Codebeispiel – Mini-Webdienst

Ganz recht, hier kommt das unvermeidliche »Hallo Welt«-Beispiel in Form der Projekts *WsHelloWorld*. Denn Webdienste sind dank Visual Studio zwar sehr einfach zu implementieren, das Konzept ist aber relativ neu und recht umfangreich. Dieses Beispiel demonstriert deshalb die grundsätzliche Funktionsweise und die Bestandteile eines Webdienstes.

23.2.1 Der Server

Legen Sie ein neues Projekt vom Typ *ASP.NET-Webdienst* an. Vergeben Sie den Namen *WsHelloWorld*. Damit Visual Studio dieses Projekt anlegen kann, muss der WWW-Publishing-Dienst laufen.

Wechseln Sie in die Code-Ansicht der automatisch angelegten Datei *Service1.asmx.cs* und entfernen Sie die Kommentarzeichen in der Implementation der Methode `HelloWorld()`. Entscheidend ist das Methoden-Attribut `[WebMethod]`, erst damit ist die Methode des Webdienstes verfügbar.

Das war's schon – nun lässt sich der Webdienst mit F5 starten.

- Der Internet Explorer (IE) erscheint mit einer Seite, die den Webdienst und seine Methoden beschreibt. Die Warnung vor der Verwendung des Namensraums `http://tempuri.org` können Sie erst einmal ignorieren, das ist für Demoprojekte in Ordnung. Aktuell interessiert nur der Kopfteil der Seite. Klicken Sie dort auf den HELLOWORLD-Link.
- Es erscheint eine neue Seite, auf der Sie die Methode `HelloWorld()` ausprobieren können. Klicken Sie dazu auf die AUFRUFEN-Schaltfläche.
- Wieder erscheint eine neue IE-Instanz, und zwar mit dem Resultat des Methodenaufrufs. Die zu besichtigende Seite enthält diesmal den Text `"Hello World"`, verziert mit einigen XML-Tags. Test bestanden! Schließen Sie dieses Fenster, nachdem Sie den Inhalt ausgiebig bewundert haben.

OK, eine Ausgabe wie diese erwartet man wohl von einem »Hallo Welt«-Beispiel, der Webdienst scheint also zu funktionieren. Eine HTML-Seite mit dem Text `Hallo Welt` hätte man aber auch einfacher haben können, notfalls auch einschließlich der XML-Tags. Was macht also den Webdienst aus?

Ganz einfach: die Kombination mit dem `"Web"`, denn `"Dienste"` an sich sind ja nichts Neues. Webdienste und ihre Clients kommunizieren miteinander über standardisierte Internetprotokolle, im Allgemeinen das Web-Protokoll HTTP. Und da der IE ebenfalls HTTP unterstützt, konnte Visual Studio in Kooperation mit den Internet Information Services (IIS) vollautomatisch und kostenlos einen Test-Client für den neuen Webdienst generieren. Wie Sie den Server über HTTP, aber mit einem Windows-basierten Client aufrufen, zeigt der nächste Abschnitt.

23.2.2 Der Client

Webdienste bzw. ihre Clients sind nicht auf HTML-Oberflächen festgelegt, das demonstriert der folgende selbstgestrickte Test-Client in Form des Projekts *WsHelloWorldClient*. Fügen Sie dazu dem bestehenden Projekt ein neues hinzu:

- Markieren Sie im Menü DATEI den Befehl NEU und wählen Sie den Befehl PROJEKT.
- Wählen Sie die Vorlage WINDOWS-ANWENDUNG und die Option ZU PROJEKTMAPPE HINZUFÜGEN. Klicken Sie OK, nachdem Sie den Namen *WsHelloWorldClient* vergeben haben.

Ihre Projektmappe enthält nun zwei Projekte, den Webdienst `WsHelloWorld` und `WsHelloWorldClient`. Damit das zweite Projekt zum Client wird, müssen Sie ihm Webserver und Webdienst bekanntgeben.

- Klicken Sie im PROJEKTMAPPEN-EXPLORER mit der rechten Maustaste auf den VERWEISE-Eintrag des Test-Clients und wählen Sie im Kontextmenü den Befehl WEBVERWEIS HINZUFÜGEN.
- Geben Sie im Textfeld des erscheinenden Fensters die Adresse des Webdienstes ein: `http://localhost/WsHelloWorld/Service1.asmx` und drücken Sie EINGABE. Im linken Teil des Fensters erscheint jetzt die Beschreibung des Webdienstes, die Sie beim ersten Test schon gesehen haben.
- Klicken Sie auf VERWEIS HINZUFÜGEN.

Visual Studio erzeugt jetzt einige neue Dateien und Einträge im PROJEKTMAPPEN-EXPLORER. Allerdings ist die entscheidende Datei nicht zu sehen: *Web References\localhost\Reference.cs* unterhalb Ihres Client-Projekts. Sie können sie im PROJEKTMAPPEN-EXPLORER sichtbar machen, indem Sie dort die Schaltfläche ALLE DATEIEN ANZEIGEN und dann das Pluszeichen vor der Datei *Reference.Map* klicken.

Diese Datei enthält den Code für einen so genannten PROXY. Ein Proxy, zu Deutsch »Stellvertreter«, sitzt zwischen Client und Server. Er bietet dem Client exakt die gleichen Eigenschaften und Methoden an wie der Server. Wenn der Client eine dieser Eigenschaften oder Methoden aufruft, leitet der Proxy sie an den Server weiter. Genauso stellt der Proxy für den Server einen Client dar. Dabei erledigt er immer Zusatzaufgaben, in diesem Fall setzt er Methoden-Aufrufe bzw. deren Rückgabewerte in HTTP-Nachrichten um und umgekehrt.

Hier ist ein Auszug aus dem automatisch generierten Proxy-Code:

```
//namespace und using-Anweisungen entfernt

//Attribute entfernt
public class Service1 :
  System.Web.Services.Protocols.SoapHttpClientProtocol {

  public Service1()
  {
    this.Url = "http://localhost/WsHelloWorld/Service1.asmx";
  }

  //Attribut entfernt
  public string HelloWorld()
  {
    object[] results = this.Invoke("HelloWorld", new object[0]);
    return ((string)(results[0]));
  }

  //Asynchrone Methoden entfernt
}
```

Diesen Code hat *Wsdl.Exe* generiert. Bei einem manuellen Aufruf dieses Dienstprogramms können Sie entsprechende Parameter angeben, um auch für andere Protokolle Code zu generieren oder die URL aus einer Konfigurationsdatei zu lesen. Mehr Informationen zu *Wsdl.Exe* finden Sie in der Online-Dokumentation.

Der Name der Proxy-Klasse ist `Service1`, genau wie der Name des Webdienstes. Abgeleitet wurde die Klasse von `SoapHttpClientProtocol`, der auf SOAP spezialisierten Basisklasse für Proxies.

Der Konstruktor setzt die `Url`-Eigenschaft auf die URL des Webdienstes. Die Vorlage für Webdienste benutzt das CodeBehind Modell, deshalb haben Sie die Veränderungen in der Datei *Service1.asmx.cs* statt in *Service1.asmx* vorgenommen. Eine kurze Besprechung der einzelnen Dateien finden Sie in Tabelle 23.1 auf Seite 951.

Erwartungsgemäß hat die Methode `HelloWorld()` denselben Prototyp wie auf dem Server. Den Aufruf der Server-Methode erledigt `Invoke()`. Diese Methode erwartet als ersten Parameter den Namen der aufzurufenden Methode, danach ein Array von Objekten mit ihren Parametern. Zurück liefert sie ebenfalls ein Array von Objekten. Dieses Array enthält den Rückgabewert sowie alle Referenz- und Rückgabe-Parameter.

Ein Beispiel für die HTTP-Nachrichten, die `Invoke()` erzeugt, ist beim ersten Test schon auf einer der HTML-Seiten erschienen. Um diese Seite (vgl. Abbildung 23.1) noch einmal zu sehen, müssen Sie nur die Datei *Service1.asmx* im IE öffnen. Die IIS erzeugen dann dynamisch alle nötigen HTML-Dateien. Allerdings nutzt es nichts, die Datei direkt zu öffnen, zum Beispiel über den Explorer. Stattdessen müssen Sie die Webadresse der Datei in die Adresszeile des IE eingeben, nur so werden die IIS überhaupt in den Ablauf eingeschaltet. Die Adresse können Sie aus der Eigenschaft WEBVERWEIS HINZUFÜGEN des Ordners *localhost* im PROJEKTMAPPEN-EXPLORER kopieren (siehe Abbildung 23.4 auf Seite 954).

Der Aufruf für `HelloWorld()` ist also ein `HTTP POST` auf `localhost/WsHelloWorld/Service1.asmx`. Diese Nachricht enthält den HTTP-Header *SOAPAction*, der Geräten wie Firewalls nähere Informationen zum Zweck der Nachricht geben soll. Der Rumpf der Nachricht entspricht der SOAP-Spezifikation: Eingehüllt in ein *Envelope*-Element erscheinen die eigentlichen Daten im *Body*-Element. In diesem Beispiel werden keine Parameter übertragen, im *Body*-Element erscheint also nur der Methodenname `HelloWorld()`. Das *Envelope*-Element enthält neben dem *Body*-Element noch die Definition der verwendeten Namensräume. Es könnte zusätzlich noch *Header*-Elemente enthalten, die benutzerdefinierte Informationen unabhängig von Methodenaufrufen übertragen. Diese Dreiteilung in Envelope, Header und Body ist typisch für SOAP-Nachrichten.

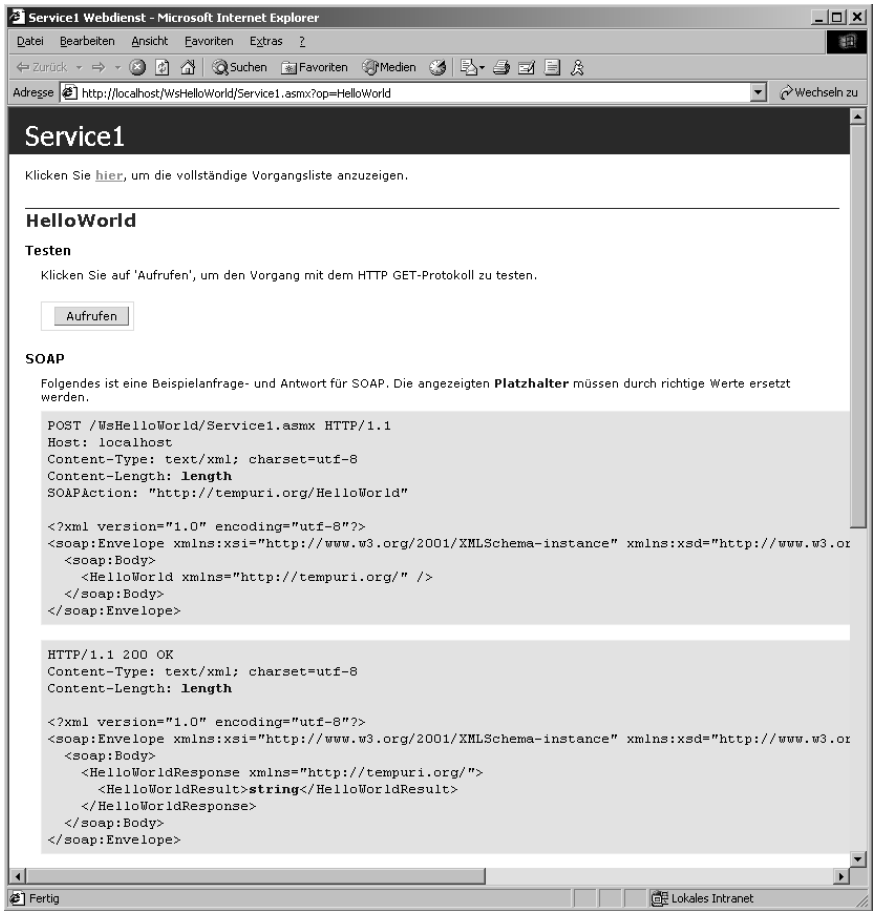


Abbildung 23.1: SOAP-Nachrichten des »Hallo Welt«-Beispiels

Die Antwort folgt dem gleichen Aufbauschema, im *Body*-Element steckt hier der Rückgabewert. Die Elementnamen *HelloWorldResponse* und *HelloWorldResult* werden automatisch aus dem Namen der Methode gebildet. Auf den genauen Aufbau von SOAP soll hier nicht eingegangen werden, denn Visual Studio nimmt Ihnen die Arbeit des Kodierens ohnehin ab. Weitere Informationen zu SOAP finden Sie beispielsweise unter `www.w3.org/TR/SOAP/`.

Es bleibt noch die Frage, woher Visual Studio die nötigen Informationen zum Generieren des Proxies hat: Sie stecken in der WSDL-Datei, Abbildung 23.2 zeigt sie im IE. Diese Datei erzeugen die IIS automatisch, wenn die *asmx*-Datei mit dem *WSDL*-Parameter aufgerufen wird. Visual Studio erzeugt außerdem eine lokale Kopie dieser Datei im Verzeichnis des Proxies.

Die *WSDL*-Datei kann recht komplex werden. Ein detailliertes Wissen über ihr Innenleben ist nicht unbedingt nötig, weil sie ohnehin automatisch generiert wird. Hier folgt deshalb nur ein kurzer Überblick über den allgemeinen

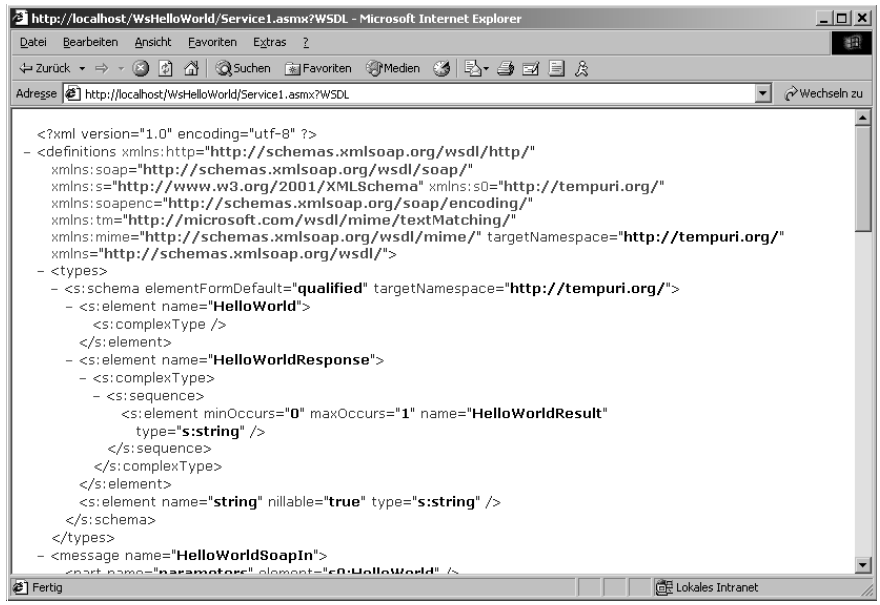


Abbildung 23.2: Die WSDL-Datei des »Hallo Welt«-Beispiels

Aufbau einer WSDL-Datei. Um das Dokument etwas zu kürzen, wurden die Elemente entfernt, die sich nicht auf SOAP, sondern auf HTTP GET und HTTP POST beziehen. Beide Formate übertragen die Daten nicht als XML, sondern als URL-kodierte Name/Wert-Paare und werden von anderen SOAP-Clients und -Servern nicht unterstützt.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions
  xmlns:http=http://schemas.xmlsoap.org/wsdl/http/
  xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
  xmlns:s=http://www.w3.org/2001/XMLSchema
  xmlns:s0=http://tempuri.org/
  xmlns:soapenc=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:tm=http://microsoft.com/wsdl/mime/textMatching/
  xmlns:mime=http://schemas.xmlsoap.org/wsdl/mime/
  targetNamespace=http://tempuri.org/
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema
      elementFormDefault="qualified"
      targetNamespace="http://tempuri.org/">
      <s:element name="HelloWorld">
        <s:complexType />
      </s:element>
      <s:element name="HelloWorldResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1"
              name="HelloWorldResult" type="s:string" />
          </s:sequence>
```

```
        </s:complexType>
      </s:element>
      <s:element name="string" nillable="true" type="s:string" />
    </s:schema>
  </types>
  <message name="HelloWorldSoapIn">
    <part name="parameters" element="s0:HelloWorld" />
  </message>
  <message name="HelloWorldSoapOut">
    <part name="parameters" element="s0:HelloWorldResponse" />
  </message>
  <!—message-Elemente für HTTP GET und POST entfernt -->
  <portType name="Service1Soap">
    <operation name="HelloWorld">
      <input message="s0:HelloWorldSoapIn" />
      <output message="s0:HelloWorldSoapOut" />
    </operation>
  </portType>
  <!—portType-Elemente für HTTP GET und POST entfernt -->
  <binding name="Service1Soap" type="s0:Service1Soap">
    <soap:binding transport=http://schemas.xmlsoap.org/soap/http
      style="document" />
    <operation name="HelloWorld">
      <soap:operation soapAction=http://tempuri.org/HelloWorld
        style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
  <!—binding-Elemente für HTTP GET und POST entfernt -->
  <service name="Service1">
    <port name="Service1Soap" binding="s0:Service1Soap">
      <soap:address
        location="http://localhost/WsHelloWorld/Service1.asmx" />
    </port>
  <!—port-Elemente für HTTP GET und POST entfernt -->
  </service>
</definitions>
```

Bei der Deutung der WSDL-Datei fängt man am besten unten an. Dort steht das *service*-Element, das den Webdienst definiert. Die Definition besteht aus dem Namen sowie dem zu benutzenden PORT (oder mehreren). Dieser Port, auch Endpunkt genannt, wird durch eine URL sowie eine Bindung (BINDING) beschrieben.

Die Bindung besteht aus der Festlegung des Protokolls sowie den bei der Übertragung zu verwendenden Datenformaten und ist im *binding*-Element definiert. In diesem Beispiel soll die Übertragung per *HTTP* erfolgen, das *style*-Attribut legt den genauen Aufbau des zu übertragenden XML-Doku-

ments fest. Im *operation*-Element finden Sie den Namen der aufzurufenden Methode sowie den im HTTP-Header *SOAPAction* einzutragenden Wert. Die beiden *soap:body*-Elemente legen fest, dass zur Interpretation der übertragenen Werte ein spezielles Schema zur Verfügung steht (siehe *types*-Element).

Im *portType*-Element finden sich die für den Methodenaufruf zu verwendenden Nachrichten, hier *HelloWorldSoapIn* und *HelloWorldSoapOut*.

Beide Nachrichten erscheinen noch einmal in den entsprechenden *message*-Elementen und sind im *types*-Element definiert.

Wie schon gesagt, brauchen Sie WSDL-Dokumente im Allgemeinen nicht manuell zu erzeugen oder zu verändern. Nur wenn Server und Client sich nicht über die Standards einig sind, muss man hier nacharbeiten.

Zurück zum Client. Fügen Sie dessen Formular eine Schaltfläche und diese Behandlungsmethode für das `Click`-Ereignis hinzu:

```
private void btnWebServiceAufrufen_Click(
  object sender, System.EventArgs e)
{
  localhost.Service1 ws = new localhost.Service1();
  MessageBox.Show(ws.HelloWorld());
}
```

Hier wird eine Instanz des Proxies erzeugt. Da eine entsprechende `using`-Anweisung fehlt, muss der Namensraum `localhost` mit angegeben werden. Der Methodenaufruf lässt dann schon nicht mehr erkennen, dass hier ein Webdienst aufgerufen wird: Die Details der Kommunikation übernimmt ja der Proxy.

Legen Sie zum Schluss noch den Client als Startprojekt der Projektmappe fest. Erklären Sie dazu im PROJEKTMAPPEN-EXPLORER das Projekt *WsHelloWorldClient* über den Befehl ALS STARTPROJEKT FESTLEGEN zum Startprojekt.

Das war's, Sie können den Webdienst jetzt mit dem neuen Client testen. Eventuell passiert nach dem ersten Klicken des WEBSERVICEAUFRUFEN-Buttons sekundenlang gar nichts: Der Prozess *AspNet_Wp.Exe* muss erst gestartet werden.

Da der Zugriff auf *localhost*-Adressen (bzw. 127.0.0.1) nicht über die Netzwerkkarte läuft, entfällt der Netzwerk-Monitor als Debugging-Werkzeug. Informationen zu alternativen Debugging-Möglichkeiten finden Sie im Abschnitt »Debugging von Webdiensten« auf Seite 952.

Tabelle 23.1 zeigt eine kurze Auflistung der beteiligten Dateien.

Tabelle 23.1: Die Dateien eines Webdienst mit Test-Clients

Verzeichnis	Datei	Funktion
Verzeichnis der Projektmappe	*WsHelloWorld.sln*	Projektverwaltung
	WsHelloWorld.suo	Projektoptionen
Client-Verzeichnis unterhalb des Projektmappen-Verzeichnisses	**.**	Normale Dateien jedes Projekt vom Typ *Windows-Anwendungs*
Obj\Debug\TempPE-Verzeichnis unterhalb des Client-Verzeichnisses	*Web References.localhost.Reference.cs.dll*	Temporäre Proxy-Dll
Web References\localhost-Verzeichnis unterhalb des *Client*-Verzeichnisess	*Reference.cs*	Proxy-Code
	Reference.map	Wurde zum Generieren des Proxy-Code benutzt
	Service1.disco	Verweis auf WSDL-Datei
	Service1.wsdl	Kopie der Beschreibung des Webdienst-API
Inetpub\wwwroot\WsHelloWord (localhost\WsHelloWorld)	**.**	Normale Dateien jeder Webanwendung
	Service1.asmx	Webdienst-Datei
	Service1.asmx.cs	CodeBehind-Datei für Webdienst
	Service1.asmx.resx	Ressourcen Datei für Webdienst
	WebService1.vsdisco	Nimmt bestimmte Verzeichnisse vom Dynamic Discovery aus

Wie Ihnen beim Anlegen des Projekts sicher schon aufgefallen ist, lässt sich das Verzeichnis eines Webdienstes nicht so frei bestimmen wie das eines Projekts vom Typ *Windows-Anwendung*. Webdienste müssen immer im Basisverzeichnis der Site (im Allgemeinen *Inetpub\wwwroot*) liegen.

Das in der Tabelle 23.1 erwähnte DYNAMIC DISCOVERY ermöglicht ein vereinfachtes Auffinden von Webdiensten auf Entwicklungsrechnern. Dazu geben Sie beim Hinzufügen einer Webreferenz in Visual Studio die Datei *localhost/default.vsdisco* an. Dadurch listet ASP.NET automatisch alle auf dem Server verfügbaren Webdienste auf.

Beim Dynamic Discovery werden einfach das Verzeichnis der angegebenen *vsdisco*-Datei sowie alle darin nicht ausgeschlossenen Unterverzeichnisse nach *asmx*-, *vsdisco*- und *disco*-Dateien durchsucht. Das ist eine praktische Microsoft-Erfindung, die aber eben nur von Microsoft-Servern unterstützt wird. Implementiert ist Dynamic Discovery übrigens durch eine `HttpHandler`-Klasse.

Um sicherheitstechnische Überraschungen auf Produktions-Servern zu vermeiden, wird Dynamic Discovery bei der Installation automatisch ausgeschaltet. Auf Entwicklungsrechnern können Sie es nachträglich wahlweise für einzelne oder alle Webdienste des Rechners einschalten.

Um alle Webdienste eines Servers für Dynamic Discovery verfügbar zu machen, entfernen Sie in der Datei *%windir%\Microsoft.NET\Framework\<Version>\CONFIG\machine.config* den Kommentar um den Eintrag `"<add verb="*" path="*.vsdisco" ...>"`. Außerdem müssen Sie dem *ASPNET*-Benutzer das Leserecht für die IIS Metabase geben. Sie haben dafür zwei Möglichkeiten: Entweder fügen Sie den ASPNET-Benutzer der Gruppe *VS Developers* hinzu, oder Sie erweitern seine Rechte für die IIS Metabase. Leider geht das nicht über die Administrations-Oberfläche, sondern nur per Programm über die IIS Admin Objekte. Ein geeignetes Programm, *MetaAcl.Exe*, ist bei Microsoft zum Download verfügbar.

Um Dynamic Discovery nur für bestimmte Websites freizuschalten, können Sie den Eintrag in die entsprechende *Web.config*-Datei einfügen. Dazu kopieren Sie am besten den Eintrag aus *Machine.config* und tragen ihn in einem neuen *httpHandlers*-Element unterhalb des *system.web*-Elements ein. Das Leserecht für die IIS Metabase ist hier nicht nötig.

23.2.3 Debugging von Webdiensten

Beim Versuch, im Einzelschritt durch den Webdienst zu gehen, erlebt man erst einmal eine unangenehme Überraschung: Haltepunkte werden einfach ignoriert. Das liegt daran, dass der Webdienst in einem anderen Prozess als dem Startprojekt der Projektmappe läuft, nämlich im Arbeitsprozess der IIS (*AspNet_Wp.exe*).

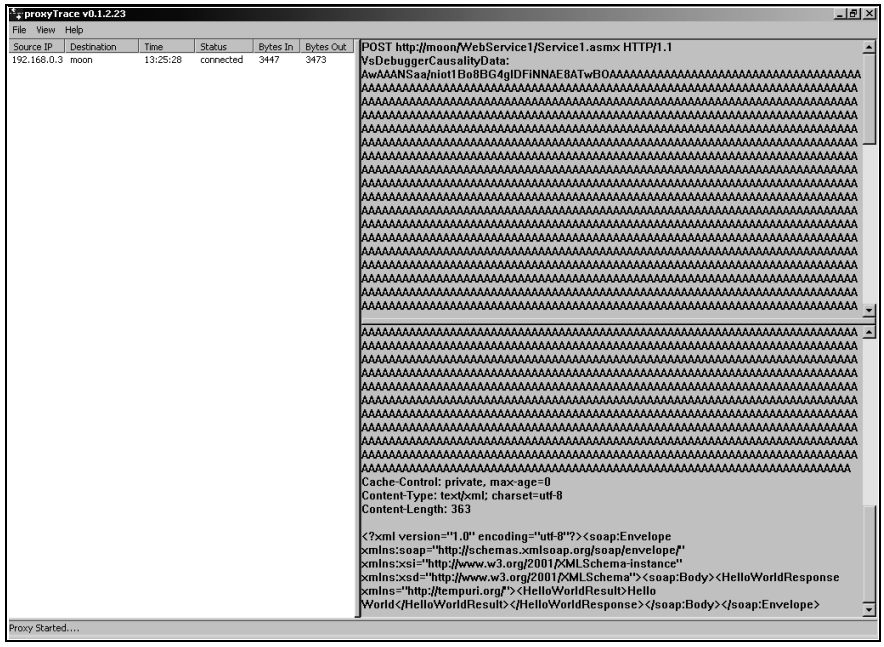
Um den Webdienst im Debugger zu untersuchen, wählen Sie im Menü DEBUGGEN den Befehl PROZESSE. Im Fenster PROZESSE wählen Sie dann den Prozess *AspNet_Wp.Exe*, klicken auf ANFÜGEN und schließen das Fenster.

Wenn der Prozess nicht zu sehen ist, rufen Sie den Webdienst einmal auf. Jetzt sollten Ihre Haltepunkte funktionieren.

Nun können Sie sowohl im Client als auch im Server Haltepunkte setzen, es fehlt aber noch eine Debugging-Möglichkeit für die Übertragungsschicht. Der Netzwerk-Monitor ist nur verwendbar, wenn Client und Server auf verschiedenen Rechnern laufen.

ProxyTrace

Für Zugriffe auf den lokalen Webserver brauchen Sie ein Zusatz-Tool wie ProxyTrace. Wie der Name schon vermuten lässt, arbeitet dieses Tool als Proxy, reicht die Daten einfach durch, und macht sie dabei sichtbar. ProxyTrace ist frei verfügbar, und steht über `www.pocketsoap.com/tcptrace/pt.asp` zum Herunterladen zur Verfügung.


proxyTrace v0.1.2.23
File View Help
Source IP Destination Time Status Bytes In Bytes Out
192.168.0.3 moon 13:25:28 connected 3447 3473
POST http://moon/WebService1/Service1.asmx HTTP/1.1
VsDebuggerCausalityData:
AwAANSaa/niot1Bo8BG4gIDFiNNAE8ATwBOAAA

Headers die Zusammenarbeit verweigern. Die gute: Dieser Header erscheint nicht, wenn Sie den Client außerhalb der Entwicklungsumgebung starten.

Neben den anderen Headern ist natürlich auch die SOAP-Nachricht zu sehen. Fehlern sollte man mit diesem Tool also schnell auf die Schliche kommen. Allerdings klappt die Aktualisierung der rechten Seite nicht so richtig, wenn über eine Verbindung mehrere Nachrichten übertragen werden. Klicken Sie dann zwischendurch auf eine andere Verbindung im linken Teil.

Zur Installation entpacken Sie die heruntergeladene Zip-Datei in ein beliebiges Verzeichnis. Beim Start fragt ProxyTrace nach der Nummer des zu überwachenden Ports und schlägt *8080* vor. Sie können hier eine beliebige Portnummer eingeben, und müssen nur dafür sorgen, dass auch Ihr Client diese Portnummer benutzt. ProxyTrace spielt für Ihren Client dann Server an diesem Port, reicht die Anfragen aber einfach an den eigentlichen Server unter der Portnummer *80* weiter (die ihrerseits in ProxyTrace unveränderlich vorgegeben ist). Natürlich können Sie auf diese Art auch mehrere Proxies hintereinander schalten, zum Beispiel neben ProxyTrace auch noch WebWasher (siehe `www.webwasher.com`) laufen lassen.

Um ProxyTrace mit einem Webdienst einzusetzen, müssen Sie zwei Veränderungen am Client vornehmen. Zum einen müssen Sie die URL der Webreferenz ändern, sonst funktioniert ProxyTrace genauso wenig wie der Netzwerk-Monitor. Ersetzen Sie *localhost* durch den Namen Ihres Rechners, im Beispiel ist das *moon*. Ändern Sie dazu im PROJEKTMAPPEN-EXPLORER die Eigenschaft WEBVERWEIS HINZUFÜGEN, Abbildung 23.4 zeigt es.

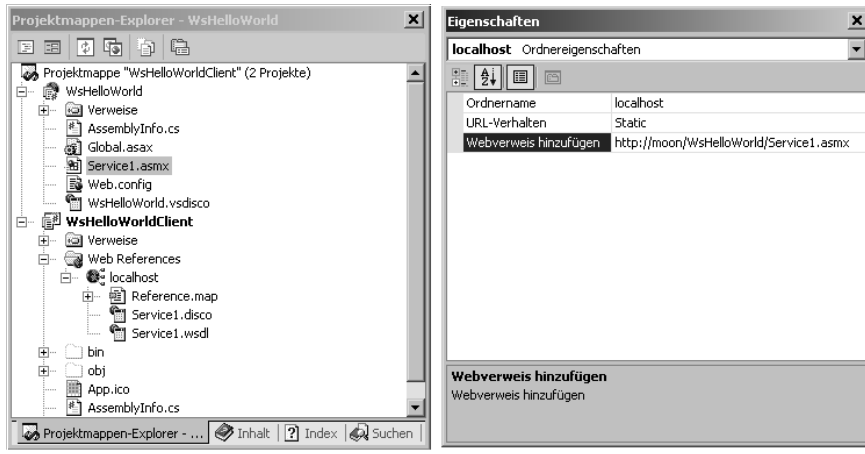


Abbildung 23.4: Die geänderte URL der Webreferenz

Zum anderen weisen Sie den Client an, die für ProxyTrace eingestellte Portnummer zu benutzen. Dazu fügen Sie eine Referenz auf `System.Net` hinzu und ändern den Code folgendermaßen:

```
private void btnWebServiceAufrufen_Click(
  object sender, System.EventArgs e)
{
  localhost.Service1 ws = new localhost.Service1();
  ws.Proxy = new WebProxy("http://moon:8080");
  MessageBox.Show(ws.HelloWorld());
}
```

Der Client verbindet sich jetzt nicht mehr direkt mit dem Server, sondern mit dem angegebenen Proxy. Im Code ist die Angabe der Portnummer entscheidend, statt des Rechnernamens könnten Sie hier auch *localhost* angeben.

Um ProxyTrace mit dem IE als Client einzusetzen, zum Beispiel beim Zugriff auf die Datei *localhost/default.vsdisco,* gehen Sie folgendermaßen vor:

1. Wählen Sie im Menü EXTRAS des IE den Befehl INTERNETOPTIONEN.
2. Wählen Sie im Fenster INTERNETOPTIONEN die Registerkarte VERBINDUNGEN und klicken Sie auf EINSTELLUNGEN.
3. Markieren Sie im Fenster EINSTELLUNGEN FÜR LOKALES NETZWERK (LAN) die Option PROXYSERVER FÜR LAN VERWENDEN und klicken Sie auf ERWEITERT.
4. Geben Sie im Fenster PROXYEINSTELLUNGEN für den Typ HTTP als Adresse des Proxyservers `"127.0.0.1"` ein und als Port den gleichen Wert wie für PROXYTRACE, also zum Beispiel `"8080"`.
5. Schließen Sie alle Fenster mit OK. Die neuen Einstellungen gelten für alle Instanzen des IE.

23.3 Codebeispiel – Asynchrone Methoden und Timeouts

Wenn ein Webdienst über das Internet aufgerufen wird, können bis zum Eintreffen der Antwort schon mal einige Sekunden vergehen. Um die Nerven des Benutzers dadurch nicht unnötig zu strapazieren, lässt man einen zweiten Thread auf die Antwort warten. Die Programmoberfläche läuft wie gewohnt über den primären Thread, der Benutzer kann also ungestört weiterarbeiten.

Der Webdienst der Beispielanwendung *WebServiceAsync* bietet die gleiche `HelloWorld()`-Methode wie die aus dem vorherigen Abschnitt. Um asynchrone Methoden und Timeouts erfahrbar zu machen, enthält sie aber einen Aufruf von `Thread.Sleep()`. Die im Tiefschlaf zu verbringende Zeit in Millisekunden wird als Parameter übergeben.

```
[WebMethod]
public string HelloWorld(int Schlafzeit)
{
  Thread.Sleep(Schlafzeit);
  return "Hello World";
}
```

Der Client enthält neben den asynchronen Methoden auch eine synchrone Methode, deren reguläre Aufgabe es ist, die Reaktion auf Timeouts zu demonstrieren. Da er umfangreicher als im letzten Beispiel wird, lohnt sich das Anlegen einer neuen Projektmappe. Fügen Sie dem Webdienst auch hier wieder ein Projekt vom Typ *Windows-Anwendung* hinzu und setzen Sie dort eine Webreferenz auf den Webdienst.

Erfreulicherweise ist in .NET die Möglichkeit zum asynchronen Aufruf von Methoden bereits integriert. Die beim Setzen der Webreferenz generierten Proxy-Dateien enthalten zusätzlich auch asynchrone Versionen der Methodenaufrufe (die im letzten Beispiel lediglich der Übersichtlichkeit halber aus den Listings entfernt wurden). Hier sind die drei für den neuen Proxy generierten Methoden, wieder zu finden in *Web References\localhost\Reference.cs* unterhalb des Client-Projekts:

```
public string HelloWorld(int Schlafzeit)
{
  object[] results =
    this.Invoke("HelloWorld", new object[] {Schlafzeit});
  return ((string)(results[0]));
}

public System.IAsyncResult BeginHelloWorld(
  int Schlafzeit, System.AsyncCallback callback, object asyncState)
{
  return this.BeginInvoke(
    "HelloWorld", new object[] {Schlafzeit}, callback, asyncState);
}

public string EndHelloWorld(System.IAsyncResult asyncResult)
{
  object[] results = this.EndInvoke(asyncResult);
  return ((string)(results[0]));
}
```

Das Muster asynchroner Methodenaufrufe basiert auf Delegaten und ist immer gleich (siehe dazu auch Kapitel 24, Multithreading, in Teil 5):

- Die `BeginX()`-Methode liefert einen Wert vom Typ `System.IAsyncResult` zurück. Über dessen `IsCompleted`-Eigenschaft lässt sich feststellen, ob die Methode bereits abgearbeitet wurde.

- Als Parameter erwartet die `BeginX()`-Methode zuerst die gleichen Parameter wie die synchrone Methode, in diesem Fall also `Schlafzeit`. Darauf folgen optional eine Rückruf-Methode (in Form eines Delegaten) und ein `asyncState`-Objekt. In diesem Beispiel werden beide nicht genutzt, entsprechenden Code finden Sie aber im Abschnitt »Eine asynchrone Auto-Suche mit den Klassen WebClient und der WebRequest« auf Seite 850.
- Die `EndX()`-Methode liefert einen Wert vom gleichen Typ wie die synchrone Methode.
- Als Parameter erwartet die `EndX()`-Methode zuerst die Rückgabe- und Referenz-Parameter der synchronen Methode, natürlich in der gleichen Reihenfolge wie dort. Als letzten Parameter erwartet die `EndX()`-Methode ein Objekt vom Typ `IAsyncResult` – das gleiche, das die `BeginX()`-Methode geliefert hat. Wichtig zu wissen ist, dass die `EndX()`-Methode bei ihrem Aufruf erst nach Abarbeitung zurückkehrt. Um asynchron zu arbeiten, muss man also entweder eine Rückruf-Methode einsetzen oder die `IsCompleted`-Eigenschaft abfragen.

Nun zurück zum Client-Projekt, das in Abbildung 23.5 zu sehen ist.



Abbildung 23.5: Client für asynchrone Methoden und Timeouts in Webdiensten

Setzen Sie zwei Gruppenfelder, ein `Label`-Steuerelement, ein Textfeld, eine Schaltfläche und einen Zeitgeber auf das Formular. Setzen Sie dann in das obere Gruppenfeld ein `Label`-Steuerelement, ein Textfeld und eine Schaltfläche. Und schließlich bekommt die untere GroupBox noch zwei `Label`-Steuerelemente und drei Schaltflächen. Stellen Sie weiterhin die in Tabelle 23.2 gezeigten Eigenschaften ein.

Tabelle 23.2: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	Text	Asynchroner Webdienst Client
Oberstes Label	Text	Schlafzeit in sec
Oberstes Textfeld	Name	txtSchlafzeit
Oberstes Textfeld	Text	7
Oberes Gruppenfeld	Text	Synchroner Aufruf
Label im oberen Gruppenfeld	Text	Timeout in sec
Textfeld im oberen Gruppenfeld	Name	txtTimeout
Textfeld im oberen Gruppenfeld	Text	5
Schaltfläche im oberen Gruppenfeld	Name	btnHelloWorld
Schaltfläche im oberen Gruppenfeld	Text	HelloWorld
Unteres Gruppenfeld	Text	Asynchroner Aufruf
Obere Schaltfläche im unteren Gruppenfeld	Name	btnBeginHelloWorld
Obere Schaltfläche im unteren Gruppenfeld	Text	BeginHelloWorld
Linkes Label im unteren Gruppenfeld	Text	Bisherige Dauer in sec:
Rechtes Label im unteren Gruppenfeld	Name	lblDauer
Rechtes Label im unteren Gruppenfeld	Text	0
Linke Schaltfläche im unteren Gruppenfeld	Name	btnAbbrechen
Linke Schaltfläche im unteren Gruppenfeld	Text	Abbrechen
Rechte Schaltfläche im unteren Gruppenfeld	Name	btnEndHelloWorld
Rechte Schaltfläche im unteren Gruppenfeld	Text	EndHelloWorld

Steuerelement	Eigenschaft	Wert
Unterste Schaltfläche	Name	btnBeenden
Unterste Schaltfläche	Text	Beenden
Zeitgeber	Interval	1000

Tabelle 23.2: Eigenschaftswerte (Forts.)

Schalten Sie dann in die Code-Ansicht und fügen Sie die Anweisung `using System.Net;` ein. Deklarieren Sie diese Variablen in der Formularklasse.

```
private IAsyncResult m_ar = null;
private localhost.Service1 m_ws = null;
private int m_Dauer = 0;
```

Und das ist der Code für den synchronen Aufruf:

```
private void btnHelloWorld_Click(object sender, System.EventArgs e)
{
  m_ws = new localhost.Service1();
  m_ws.Timeout = int.Parse(txtTimeout.Text) * 1000;
  try
  {
    m_ws.HelloWorld(int.Parse(txtSchlafzeit.Text) * 1000);
    m_ws = null;
  }
  catch (WebException x)
  {
    m_ws = null;
    MessageBox.Show(x.Message);
  }
}//btnHelloWorld_Click
```

Hier werden die `Timeout`-Eigenschaft des Proxies und die Schlafzeit des Webdienstes gesetzt. Im Beispiel sind das 5 und 7 Sekunden, deswegen erhalten Sie nach 5 Sekunden eine Ausnahme vom Typ `WebException`. Leider ist am Typ der Ausnahme nicht zu erkennen, ob es sich um ein Timeout oder einen anderen Fehler handelt. Bis zum Auftreten des Fehlers oder der Rückkehr der synchronen Methode ist die Programmoberfläche eingefroren, der synchrone Aufruf verdeutlicht also das »normale« Verhalten.

Legen Sie jetzt die folgenden Methoden für den asynchronen Aufruf an:

```
private void btnAbbrechen_Click(object sender, System.EventArgs e)
{
  timer1.Enabled = false;
  m_ws.Abort();
}//btnAbbrechen_Click
```

```
private void btnBeenden_Click(object sender, System.EventArgs e)
{
  Close();
}//btnBeenden_Click


private void btnBeginHelloWorld_Click(
  object sender, System.EventArgs e)
{
  InitialisiereProgramm();
  timer1.Enabled = true;
  m_ws = new localhost.Service1();
  m_ar = m_ws.BeginHelloWorld(
    int.Parse(txtSchlafzeit.Text) * 1000, null, null);
}//btnBeginHelloWorld_Click


private void btnEndHelloWorld_Click(object sender, System.EventArgs e)
{
  timer1.Enabled = false;
  MessageBox.Show(m_ws.EndHelloWorld(m_ar));
}//btnEndHelloWorld_Click


private void InitialisiereProgramm()
{
  lblDauer.Text = "0";
  m_ar = null;
  m_Dauer = 0;
  m_ws = null;
}//InitialisiereProgramm


private void timer1_Tick(object sender, System.EventArgs e)
{
  string s = m_ar.IsCompleted ? "" : "nicht ";

  m_Dauer++;
  lblDauer.Text = m_Dauer.ToString() +
    string.Format(" (Ist {0}fertig!)", s);
}//timer1_Tick
```

`btnBeginHelloWorld_Click()` setzt zuerst die Ausgabe und einige Variablen zurück. Dann wird der Zeitgeber aktiviert und die `BeginHelloWorld()`-Methode des Proxies aufgerufen. Dadurch wartet ein Thread aus dem Threadpool der Anwendung auf die Abarbeitung durch den Webdienst, der Benutzer kann also mit dem Client weiterarbeiten.

Zum Beispiel könnte er mit der ABBRECHEN-Schaltfläche die `Abort()`-Methode des Proxies aufrufen. Diese Methode ist im Proxy-Code nicht zu sehen, weil sie über `SoapHttpClientProtocol` und `HttpWebClientProtocol` von `WebClientProtocol` geerbt wurde. Der laufende Zeitgeber zeigt über die Methode `timer1_Tick()` mit einem `Label`-Steuerelement die abgelaufene Zeit an; über die

Abfrage der `IsCompleted`-Eigenschaft der `IAsyncResult`-Schnittstelle signalisiert er auch die Beendigung des Webdienstes.

Der Benutzer kann jederzeit über `btnEndHelloWorld_Click()` die `EndHelloWorld()`-Methode des Proxies aufrufen. Tut er das, bevor der Webdienst ausgeschlafen hat, blockiert `EndHelloWorld()`. Abhilfe schafft eine Rückruf-Funktion. Diese wird mit einem `AsyncCallDback`-Objekt verbunden, das Sie dann als zweiten Parameter beim Aufruf von `BeginHelloworld()` übergeben. Am Ende des Webdienstes wird Ihre Rückruf-Funktion automatisch aufgerufen. Ein Beispiel dazu finden Sie im Abschnitt »Eine asynchrone Auto-Suche mit den Klassen WebClient und der WebRequest« auf Seite 850.

Setzen Sie jetzt noch den Client als Startprojekt und probieren Sie dann das Ganze aus.

23.4 Einfaches Zustand halten, Caching und mehrere Webdienste in einem Projekt

Dieser Abschnitt zeigt zwei Techniken, die zur Grundausstattung vieler Webdienste gehören: Zustand halten und Caching. Da beide Techniken die Zwischenspeicherung von Daten zum Inhalt haben, kann man die Begriffe teilweise synonym verwenden. Vorgestellt wird jeweils die am einfachsten zu implementierende Variante, und der Abschnitt konzentriert sich auf das »Wie und Warum«. Außerdem finden Sie hier Informationen zur Arbeit mit mehreren Webdiensten in einem Projekt.

23.4.1 Codebeispiel – Zustand halten

Webdienste sind von Natur aus *zustandslos*: Nach Abarbeitung einer Client-Anfrage merken sie sich nicht den Sitzungszustand. Das ist im Allgemeinen vorteilhaft, denn dadurch werden zwischen den Client-Anfragen keine Ressourcen belegt.

Aber nicht immer kann ein Webdienst seine Aufgabe in einem Methodenaufruf erledigen, manchmal muss er sich Zwischenergebnisse oder Benutzereinstellungen merken. Dieses Merken eines Zustands nennt man Zustand (Englisch: State) halten, und es kann genauso gut wenige Sekunden wie mehrere Jahre umfassen.

Dabei stellen sich zwei Fragen: Wo und wie werden die gebunkerten Daten gespeichert, und woher weiß der Server, zu welchem Client sie gehören? Die Identifikation des Clients erfolgt bei Webseiten im Allgemeinen über eine SessionId, die in Cookies, versteckten Textfeldern oder in der URL zwischen Client und Dienst transportiert wird. Für Webdienste fallen mangels

Oberfläche die versteckten Textfelder weg, dafür können sie SOAP-Header benutzen. Festzuhalten ist auf jeden Fall, dass für Webdienste kein verbindlicher Standard zum Transport der SessionId oder sonstiger Zustandsinformationen existiert – Client und Server müssen sich also jeweils auf ein Verfahren einigen. Das Speichern der eigentlichen Daten übernimmt im Allgemeinen der Server, je nach Randbedingungen hält er die Daten im Speicher oder lagert sie aus. Am Ende des Abschnitts finden Sie eine kurze Beschreibung der Möglichkeiten in ASP.NET.

Erfreulicherweise hat Microsoft seinen Kunden die Implementation des Zustandhaltens abgenommen, zumindest für einfache Fälle. Sie brauchen im Konstruktor des `WebMethod`-Attributs nur den Parameter `EnableSession` auf `true` zu setzen, und können Ihre Daten im `Session`-Objekt speichern.

Das Beispielprojekt *WsCachingUndState* zeigt diese Möglichkeit. Um es nachzustellen, legen Sie ein neues Webdienst-Projekt an und fügen die folgenden beiden Methoden in die CodeBehind-Datei ein:

```
[WebMethod(EnableSession=true)]
public string GibName()  {
  string s = "Den Namen habe ich leider vergessen";

  if (Session["Name"] != null)
    s = Session["Name"].ToString();
  return s;
}


[WebMethod(EnableSession=true)]
public void MerkeName(string Name)
{
  Session["Name"] = Name;
}
```

Wie schon angekündigt, erhält der Konstruktor des `WebMethod`-Attributs hier den Wert `true` für den benannten Parameter `EnableSession`. Dadurch kann der Code zum Datenspeichern das `Session`-Objekt benutzen, der Zugriff auf die Daten erfolgt wie bei jeder `Collection` über Name-/Wert-Paare.

Nach der Compilierung dieses Projekts geht es mit dem in Abbildung 23.6 zu sehenden Test-Client weiter: Fügen Sie der Projektmappe ein Projekt vom Typ *Windows-Anwendung* hinzu und setzen Sie eine Webreferenz auf den Webdienst.

Setzen Sie eine Schaltfläche und ein Gruppenfeld auf das Formular. Setzen Sie dann in das Gruppenfeld ein `Label`-Steuerelement, ein Textfeld und zwei Schaltflächen. Stellen Sie schließlich die in Tabelle 23.3 gezeigten Eigenschaften ein.



Abbildung 23.6: Der Client für die Demonstration des Zustandshaltens

Tabelle 23.3: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	`Text`	Caching und Zustand halten testen
Gruppenfeld	`Text`	Zustand halten testen
Label	`Text`	Name
Textfeld	`Name`	txtName
Textfeld	`Text`	(leer)
Linke Schaltfläche im Gruppenfeld	`Name`	btnNameSetzen
Linke Schaltfläche im Gruppenfeld	`Text`	NameSetzen
Rechte Schaltfläche im Gruppenfeld	`Name`	btnNameLesen
Rechte Schaltfläche im Gruppenfeld	`Text`	NameLesen
Unterste Schaltfläche	`Name`	btnBeenden
Unterste Schaltfläche	`Text`	Beenden

Gehen Sie dann in die Code-Ansicht und erweitern Sie die `uses`-Anweisungen um `using System.Net;`. Deklarieren Sie dieses Datenfeld in der Formularklasse.

```
private localhost.MerkService m_ms = null;
```

Hier ist der Code für die drei Schaltflächen:

```
private void btnBeenden_Click(object sender, System.EventArgs e)
{
  Close();
}

private void btnNameSetzen_Click(object sender, System.EventArgs e)
{
  m_ms.MerkeName(txtName.Text);
}

private void btnNameLesen_Click(object sender, System.EventArgs e)
{
  MessageBox.Show(m_ms.GibName());
}
```

Während der Code kaum Unerwartetes enthält, wird Sie vermutlich überraschen, dass der Client so (noch) nicht funktioniert: Ohne das explizite Zurücksenden des vom Webdienst gelieferten Cookies beim Aufrufen von Webdienst-Methode tut sich hier nämlich gar nichts. Erfreulicherweise ist das Problem mit einer Zeile behoben: Sie spendieren dem Webdienst nach seiner Erzeugung ein `CookieContainer`-Objekt, das sich um alles kümmert. Hier ist der Code für den Konstruktor des Formulars:

```
public Form1()
{
  InitializeComponent();

  m_ms = new localhost.MerkService();
  m_ms.CookieContainer = new CookieContainer();
}
```

Das war's auch schon. Setzen Sie den Client als Startprojekt und probieren Sie den Merk-Service aus.

Versuchen Sie das Ganze auch einmal mit einem zweiten Client. Die für den Start notwendige Exe-Datei finden Sie im Verzeichnis *WsCachingUndState\Client\bin\debug*. Wie zu erwarten, merkt sich der Webdienst für jeden der Clients einen (eigenen) Namen.

Wenn alle Clients auf dieselben Daten zugreifen sollen, benutzen Sie statt des `Session`- das `Application`-Objekt, das im Gegensatz zum `Session`-Objekt auch ohne Attribut-Manipulation verfügbar ist. Damit sich die Clients beim Schreiben nicht in die Quere kommen, bietet die Klasse `Application` die Methoden `Lock()` und `UnLock()`.

Für beide Objekte gilt, dass die darin gespeicherten Daten während der Lebenszeit des Objekts erhalten bleiben. Für das `Session`-Objekt ist das die Dauer der Verbindung, für das `Application`-Objekt ist es die Lebensdauer der

Anwendungsdomäne. Denken Sie daran, dass die aktuelle Anwendungsdomäne durchaus zerstört und durch eine neue ersetzt werden kann, zum Beispiel wenn sie zu viel Speicher belegt oder die *Global.asax*-Datei geändert wird. Um Daten zu löschen, bieten beide Objekte verschiedene `Remove()`-Methoden.

Für das `Session`-Objekt können Sie festlegen, wo es seine Daten speichern soll. Voreinstellung ist im Prozess, das ist natürlich die schnellste, aber nicht unbedingt die ressourcenschonendste Variante. Weitere Möglichkeiten sind außerhalb des Prozesses oder über einen SQL-Server, wobei beide Speicherorte auf einem anderen Rechner liegen können. Dadurch wird der Webserver entlastet, und außerdem kann dieses Konzept auch in Web-Farmen eingesetzt werden. Das Übertragen der Daten kostet dann natürlich mehr Zeit. Die Einstellung nehmen Sie in der Datei *Web.config* oder *Machine.config* vor, und zwar im *sessionState*-Element. Dort können Sie auch festlegen, ob Cookies benutzt werden. Für das `Application`-Objekt sind solche Einstellungen nicht möglich. Weitere Konfigurationsinformationen finden Sie in der Online-Dokumentation unter *Sitzungsstatus* (im Text heißt es dann *Sitzungszustand*).

Übrigens ist das `Session`-Objekt einer der Gründe, warum man Webdienste von `System.Web.Services.WebService` ableiten sollte. Mit einer anderen Basisklasse müssten Sie auf das `Session`-Objekt über eine Zeile wie

```
HttpSessionState hss = HttpContext.Current.Session;
```

zugreifen, dazu benötigen Sie noch

```
using System.Web.SessionState;
```

Über die `Session`-Eigenschaft sind auch andere Eigenschaften des Sitzungskontexts verfügbar, zum Beispiel die `SessionID` und Informationen zur Thread-Sicherheit.

23.4.2 Caching

Unter Caching versteht man das Zwischenspeichern von Daten: anstatt sie erneut zu berechnen oder zusammenzustellen, sendet der Server einfach die bereits vorhandenen Daten.

Dabei wird natürlich gewonnene Rechenzeit gegen verbrauchten Speicherplatz getauscht. Caching wird also umso interessanter, je aufwändiger das Zusammenstellen der Daten ist. Neben der Rechenzeit des Webservers gehen hier insbesondere auch Zeitverluste durch Datenbankzugriffe ein. Und je »haltbarer« die Daten sind, desto mehr bieten sie sich für das Caching an. Abträglich für das Caching ist neben einer großen Datenmenge

vor allem eine starke Parametrisierung der Daten, bei Dutzenden von Versionen multiplizieren sich auch kleine Datenteilchen zu großen Datenmengen.

Dieses Problem hat der Webdienst dieses Beispiels garantiert nicht: Er liefert lediglich die Uhrzeit – allerdings in »gebrauchter« Form. Das heißt, er ermittelt die Zeit nicht bei jeder Anfrage neu, sondern erst nach Ablauf eines einstellbaren Verfallsdatums. Dabei berücksichtigt er auch unterschiedliche Parameter-Werte. Die Benutzeroberfläche zeigt Abbildung 23.7.

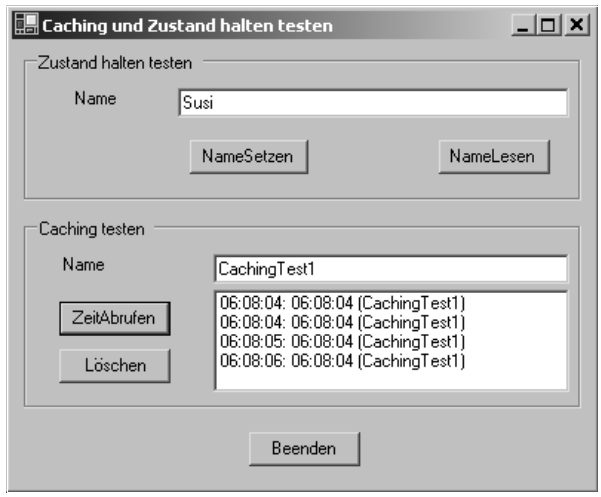


Abbildung 23.7: Der Client fürs Caching

Fügen Sie dem Server einen neuen Webdienst hinzu und nennen Sie ihn `"ZeitService"`. Dieser Webdienst bekommt eine einzige Methode, die die aktuelle Serverzeit einschließlich der Sekunden als String zurückgibt.

```
[WebMethod(CacheDuration=10)]
public string GibZeit(string Name)
{
  return DateTime.Now.ToLongTimeString() + " (" + Name + ")";
}
```

Das Interessante an dieser Methode ist der benannte Parameter `CacheDuration` im Konstruktor des `WebMethod`-Attributs. Mit diesem Parameter legen Sie die Speicherdauer des Rückgabewerts fest, in diesem Fall 10 Sekunden. Wenn während dieser Zeit ein Client `GibZeit()` aufruft, liefert ASP.NET die zuletzt zurückgegebene Antwort, ohne die Methode aufzurufen. Erst, wenn die gespeicherte Antwort älter als 10 Sekunden ist, ruft ASP.NET tatsächlich `GibZeit()` auf. Die Methode hängt den Wert des `Name`-Parameters an die Antwort, um zu zeigen, dass ASP.NET für jeden Wert dieses Parameters eine eigene Version der Antwort speichert. Anders als beim Caching von HTML-Seiten ist dieses Verhalten nicht auf bestimmte Parameter einschränkbar.

Kompilieren Sie den Webdienst und setzen Sie im Test-Client eine Webreferenz darauf. Erweitern Sie dann den Test-Client durch ein Gruppenfeld mit zwei Schaltflächen und einem Listenfeld und stellen Sie dafür die in Tabelle 23.4 gezeigten Eigenschaften ein.

Tabelle 23.4: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Gruppenfeld	Text	Caching testen
Label	Text	Name
Textfeld	Name	txtCachingName
Listenfeld	Name	lstZeit
Obere Schaltfläche im Gruppenfeld	Name	btnZeitAbrufen
Obere Schaltfläche im Gruppenfeld	Text	ZeitAbrufen
Untere Schaltfläche im Gruppenfeld	Name	btnLöschen
Untere Schaltfläche im Gruppenfeld	Text	Löschen

Deklarieren Sie für den neuen Webdienst noch eine Variable in der Formularklasse.

```
private localhost1.ZeitService m_zs = null;
```

Der neue Webdienst wird wie der alte im Konstruktor des Formulars instanziiert und mit einem `CookieContainer`-Objekt verbunden.

```
public Form1()
{
  InitializeComponent();
  m_ms = new localhost.MerkService();
  m_ms.CookieContainer = new CookieContainer();
  m_zs = new localhost1.ZeitService();
  m_zs.CookieContainer = new CookieContainer();
}
```

Hier ist noch der Code für die beiden neuen Schaltflächen:

```
private void btnZeitAbrufen_Click(object sender, System.EventArgs e)
{
  lstZeit.Items.Add(DateTime.Now.ToLongTimeString() + ": " +
    m_zs.GibZeit(txtCachingName.Text));
}
private void btnLöschen_Click(object sender, System.EventArgs e)
{
  lstZeit.Items.Clear();
}
```

Damit der Caching-Effekt sichtbar wird, erscheint im Listenfeld neben der vom Webdienst gelieferten auch die lokal ermittelte Zeit. Erwartungsgemäß liefert der Webdienst nur etwa alle 10 Sekunden die richtige Zeit und berücksichtigt dabei auch den Wert des übergebenen Parameters.

23.4.3 Mehrere Webdienste in einem Projekt implementieren

In diesem Abschnitt wurde zwar schon ein Projekt *WsCacheUndState* vorgestellt, das mit mehreren Webdiensten arbeitet, da dort die Webreferenzen aber auf die *asmx*-Dateien gesetzt sind, erscheinen die Webdienste im Client-Code so, als ob sie nichts miteinander zu tun hätten. Eigentlich würde man ja erwarten, dass nach Eingabe von `"localhost."` eine Liste mit den Einträgen `MerkService` und `ZeitService` ausklappt.

Das kann man als Schönheitsfehler ansehen, denn die beiden Instanzen `localhost` und `localhost1` können sich ja sowohl das `CookieContainer`- als auch `WebProxy`-Objekt teilen. Sie können den Rest des Abschnitts also getrost überspringen, wenn Ihnen solche "esoterischen" Dinge erst einmal unwichtig sind. Andererseits sollten interne Design-Entscheidungen, wie die Aufteilung der Webdienste auf Module, für den Client unsichtbar bleiben.

Variante 1: Auslagerung des Codes aus den asmx.cs-Dateien in entsprechende Klassen

Die im Allgemeinen beste Variante ist die Auslagerung des Codes aus den *asmx.cs*-Dateien in entsprechende Klassen. Diese Klassen enthalten dann die Implementation der Webdienste und lassen sich auch von beliebigen anderen Clients benutzen oder im Global Assembly Cache unterbringen. Die asmx.cs-Dateien werden durch eine einzige ersetzt, die nur noch als Fassade fungiert und sich auf Instanziierung der Klassen und Aufrufe ihrer Methoden beschränkt.

Variante 2: Editieren des Proxy-Codes

Alternativ können Sie den Proxy-Code editieren. Dazu benutzen Sie am besten eine Kopie des Client-Projekts. Entfernen Sie darin die beiden Webreferenzen und löschen Sie die entsprechenden Ordner im PROJEKTMAPPEN-EXPLORER. Die Ordner müssen Sie dort eventuell mit der Schaltfläche ALLE DATEIEN ANZEIGEN erst sichtbar machen.

Setzen Sie jetzt eine Webreferenz auf die *vsdisco*-Datei des Server-Projekts. Wenn Sie ProxyTrace benutzen wollen, können Sie statt `"localhost"` auch den Namen des Rechners angeben und brauchen dann die URL der Webreferenz nicht mehr zu bearbeiten (siehe Abbildung 23.8).

Wie zu erwarten, enthält die *vsdisco*-Datei Informationen zu beiden Webdiensten des Verzeichnisses. Nachdem Sie auf VERWEIS HINZUFÜGEN geklickt

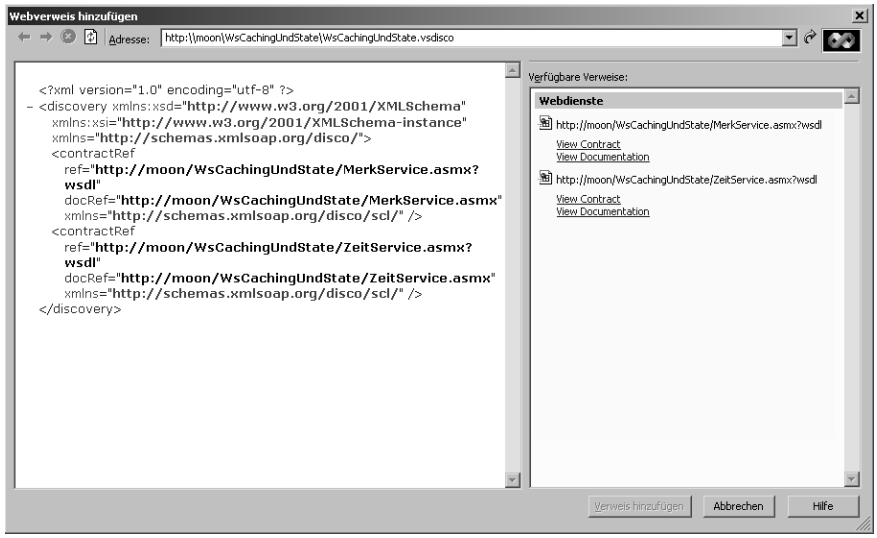


Abbildung 23.8: vsdisco-Datei des Server-Projekts

haben, erscheinen beide im PROJEKTMAPPEN-EXPLORER auch brav als Webreferenz.

Aber wenn Sie jetzt `"localhost."` (bzw. `"<Rechnername>."`) eingeben, um einen Webdienst zu instanziieren, erscheint nach wie vor nur ein Eintrag in der Vorschlagsliste. Ein Blick in die Datei *Reference.cs* zeigt, dass dort auch nur der Code für den `MerkService`-Proxy steht. Das liegt an der Beschränkung des Tools WSDL.EXE, das sich beim Generieren des Proxy-Codes auf einen einzigen Webdienst beschränkt. (Mehr dazu im folgenden Abschnitt.)

Als Lösung bietet sich in diesem Beispiel die manuelle Erweiterung des Proxy-Codes an. Kopieren Sie dazu einfach die Klasse `MerkService` im Proxy. Entfernen Sie dann in der Kopie die Definitionen der `MerkeName`-Methoden und ändern Sie dort alle Vorkommen von `"MerkService"` in `"ZeitService"` sowie alle Vorkommen von `"GibName"` in `"GibZeit"`.

Jetzt erscheinen in der Vorschlagsliste beide Webdienste. Ändern Sie dementsprechend im Client-Code auch beide Vorkommen von `"localhost1"` in `"localhost"`.

Damit ist das gewünschte Ziel erreicht. Allerdings ist Vorsicht geboten, denn beim nächsten Aktualisieren der Webreferenz würde der selbstgebastelte Proxy-Code überschrieben.

Variante 3: Attribute benutzen

Mit Attributen können Sie WSDL.EXE dazu bewegen, Proxy-Code für beliebig viele Webdienste zu erzeugen.

Der erste Schritt ist die Verwendung von SOAP-Namensräumen. Das geschieht mit dem `WebService-Attribut`, dessen Konstruktor Sie den benannten Parameter `Namespace` übergeben. Für die Klasse `MerkService` sieht das so aus, wobei der Name des Namensraums natürlich nicht auflösbar sein muss:

```
[WebService(Namespace="http://moon/EinService")]
public class MerkService : System.Web.Services.WebService
{
  //...
}
```

Sie können dem Konstruktor des `WebService-Attributs` übrigens auch den benannten Parameter `Description` übergeben und damit den Webdienst beschreiben.

Sobald jeder Webdienst seinen eigenen Namensraum hat, erzeugt WSDL.EXE auch den nötigen Proxy-Code in der Datei *Reference.cs*. Außerdem haben Sie so den Namensraum *tempuri.org* ersetzt, der für Produktions-Server sowieso nicht akzeptabel ist. Damit keine Missverständnisse aufkommen: Der Namensraum des Webdienstes ist natürlich völlig unabhängig vom Namensraum Ihres C#-Projekts.

Leider hat die Lösung noch einen Schönheitsfehler: Hinter dem Klassennamen würde jetzt das Postfix `"Soap"` erscheinen. Um auch das noch geradezubiegen, legen Sie zuerst den Namen auf Klassen-Ebene fest. Dazu benutzen Sie das `WebServiceBinding`-Attribut.

```
[WebServiceBinding(Name="MerkService")]
[WebService(Namespace="http://moon/EinService")]
public class MerkService : System.Web.Services.WebService
{
  //...
}
```

Jetzt binden Sie mit dem `SoapDocumentMethod`-Attribut noch alle Methoden an diesen Service. Dazu benötigen Sie eine `using`-Anweisung für `System.Web.Services.Protocols`. Der Wert des benannten Parameters `Binding` muss natürlich dem eben angegebenen Namen entsprechen. Auch dazu wieder ein Beispiel aus der Klasse `MerkService`:

```
[WebMethod(EnableSession=true)]
[SoapDocumentMethod(Binding="MerkService")]
public string GibName()  {
  //...
}
```

Zur besseren Übersicht ist hier der komplette Code der Klasse `ZeitService`. Wie zu sehen, muss der neue Namensraum nicht auflösbar sein, nur eindeutig.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols; //für SoapDocumentMethod

namespace WsCachingUndState
{
  [WebServiceBinding(Name="ZeitService")]
  [WebServiceAttribute(Namespace="http://moon/NochEinService")]
  public class ZeitService : System.Web.Services.WebService
  {
    public ZeitService()
    {
      InitializeComponent();
    }

    #region Component Designer generated code
    //generierten Code entfernt
    #endregion


    [WebMethod(CacheDuration=10)]
    [SoapDocumentMethod(Binding="ZeitService")]
    public string GibZeit(string Name)
    {
      return DateTime.Now.ToLongTimeString() + " (" + Name + ")";
    }
}
```

Geschafft. Nach Erstellen des Webdienst-Projekts und Aktualisieren der Webreferenz im Client sind beide Webdienste dort unter `localhost` bzw. `<Rechnername>` verfügbar. Die Webreferenz muss natürlich wie in der zweiten Variante die vsdisco-Datei referenzieren.

23.5 Fortgeschrittene Techniken für das Caching

Wie im vorhergehenden Abschnitt beschrieben, lässt sich das Caching für Webdienste durch das `WebMethod`-Attribut einschalten und beeinflussen. Dieses Caching erledigt ASP.NET dann ohne Aufruf der Methoden des Webdienstes.

Neben dieser Alles-oder-nichts-Variante bietet Ihnen ASP.NET auch die Möglichkeit, beliebige Teilergebnisse eines Webdienstes vorzuhalten. Dazu dient das `Cache`-Objekt, das Einträge im Schlüssel/Wert-Format verwaltet. Das `Cache`-Objekt ermöglicht Ihnen auch, Regeln für das Löschen dieser Einträge aus dem Cache festzulegen. Sichtbarkeit und Lebensdauer der Daten im `Cache`-Objekt entsprechen denen des `Application`-Objekts.

Das Beispielprojekt *WsCaching* (vgl. Abbildung 23.9) zeigt einige der Möglichkeiten des `Cache`-Objekts.

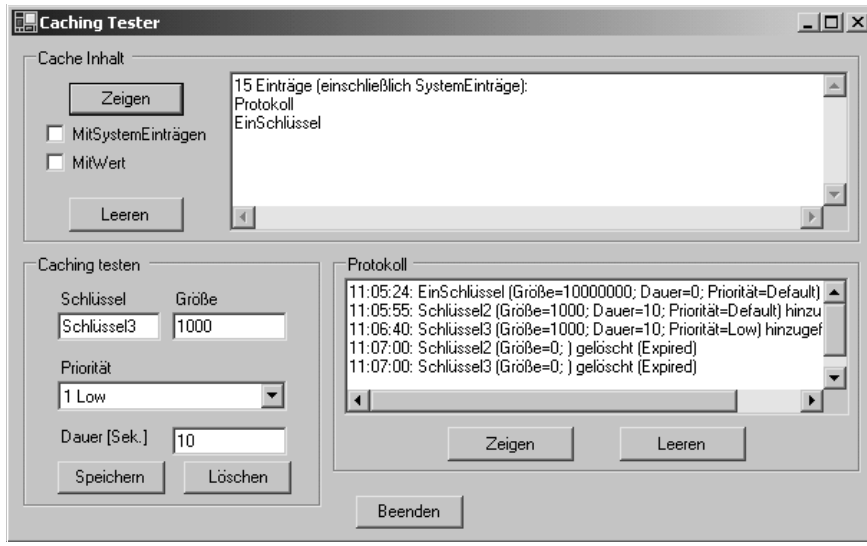


Abbildung 23.9: Demonstration der Möglichkeiten des Cache-Objekts

Legen Sie ein neues Webdienst-Projekt an und fügen Sie die folgenden drei Methoden zum Ansehen und Löschen der Inhalte des Cache-Objekts in die CodeBehind-Datei ein:

```
private string GibCacheEintrag(DictionaryEntry de, bool MitWert)
{
  string s = de.Key.ToString();
  if (MitWert)
  {
    s += " = " + de.Value;
    s += "\r\n-----------------------------------------";
  }
  return s + "\r\n";
}//GibCacheEintrag

[WebMethod]
public String GibCacheInhalt(
  bool MitSystemEinträgen, bool MitWert)
{
  StringBuilder sb = new StringBuilder();

  sb.Append(Context.Cache.Count +
    " Einträge (einschließlich SystemEinträge):\r\n");
  foreach(DictionaryEntry de in Context.Cache)
  {
    if (MitSystemEinträgen)
      sb.Append(GibCacheEintrag(de, MitWert));
    else {
      String s = de.Key.ToString();
```

```
      if (!s.StartsWith("ISAPIWorkerRequest.")
        && !s.StartsWith("System."))
        sb.Append(GibCacheEintrag(de, MitWert));
    }
  }//foreach
  return sb.ToString();
}//GibCacheInhalt

[WebMethod]
public void LeereCache()
{
  foreach(DictionaryEntry de in Context.Cache)
  {
    String s = de.Key.ToString();
    if (!s.StartsWith("ISAPIWorkerRequest")
      && !s.StartsWith("System"))
      Context.Cache.Remove(s);
  }//foreach
}//LeereCache
```

Die Methode `GibCacheInhalt()` gibt unter Zuhilfenahme von `GibCacheEintrag()` einen String zurück, der die Anzahl der Einträge im Cache sowie die Einträge selbst auflistet. Über den Parameter `MitSystemEinträgen` lassen sich die nicht vom Testprogramm angelegten Einträge ausblenden. Optional lässt sich mit dem Parameter `MitWert` der Wert jedes Eintrags sichtbar machen, der Schlüssel ist immer enthalten. `LeereCache()` tut genau das, was der Name sagt.

Die folgende Methode erledigt die Speicherung eines Eintrags im Cache, wobei neben Schlüssel und Größe des Eintrags auch Regeln zu seiner Löschung angegeben werden können.

```
[WebMethod]
public void SpeichereEintrag(
  string Schlüssel, int Dauer, int Priorität, int Größe)
{
  //Dauer-Parameter auswerten
  TimeSpan ts;
  if (Dauer != 0)
    ts = TimeSpan.FromSeconds(Dauer);
  else
    ts = Cache.NoSlidingExpiration;
  //Priorität-Parameter auswerten
  CacheItemPriority cip;
  Type t = typeof(CacheItemPriority);
  if (Enum.IsDefined(t, Priorität))
    cip = (CacheItemPriority)Enum.ToObject(t, Priorität);
  else
    throw new ArgumentOutOfRangeException("Priorität", Priorität,
      "Priorität muss zwischen 1 und 6 liegen.");
  //Größe-Parameter auswerten
  StringBuilder sb = new StringBuilder(Größe);
  for (int i = 0; i < Größe; i++)
    sb.Append('X');
```

```
  //Eintrag speichern und protokollieren
  Context.Cache.Insert(Schlüssel,
    sb, null, Cache.NoAbsoluteExpiration,
    ts, cip,
    new CacheItemRemovedCallback(this.LöschRückruf));
  Protokolliere(Schlüssel, FormatiereOptionen(ts, cip),
    Größe, "hinzugefügt");
}//SpeichereEintrag
```

Der Parameter `Dauer` enthält die Zeit in Sekunden, die der Eintrag nach dem letzten Zugriff mindestens im Cache verbleiben soll. Da die `Insert()`-Methode des `Cache`-Objekts dafür einen `TimeSpan`-Wert erwartet, muss der Wert in `Dauer` entsprechend interpretiert werden. Sinn der Übergabe als Integer-Wert ist natürlich, den Webdienst auch von Clients aufrufen zu können, die nicht für die .NET-Laufzeitumgebung geschrieben sind.

Das Gleiche gilt für den Parameter `Priorität`. Seine Umwandlung in den Typ `CacheItemPriority` ist eine gute Demonstration dafür, wie man so etwas unter .NET typsicher implementiert.



Mit dem folgenden Code erhalten Sie eine Liste der möglichen Werte für `CacheItemPriority`*:*

```
CacheItemPriority[] ac =
  (CacheItemPriority[])Enum.GetValues(typeof(CacheItemPriority));
StringBuilder sb1 = new StringBuilder();
foreach (CacheItemPriority c in ac)
  sb1.Append(string.Format("{0:D} {1:G}", c, c) + "\r\n");
```

Schließlich wird vor dem eigentlichen Speichern des Eintrags noch sein Wert entsprechend dem Parameter `Größe` als mehr oder weniger umfangreicher String generiert.

Beim Aufruf der `Insert()`-Methode des `Cache`-Objekts werden zuerst Schlüssel und Wert des Eintrags übergeben. Existiert bereits ein Eintrag mit diesem Schlüssel, wird er zuerst komplett gelöscht, bevor das Programm einen neuen Eintrag einfügt. Als dritter Parameter könnte ein Objekt vom Typ `CacheDependency` übergeben werden. Dadurch würde der Eintrag automatisch gelöscht, sobald sich die im Objekt referenzierte Datei oder der darin referenzierte Eintrag im Cache ändert.

Als vierten Parameter können Sie einen Verfallszeitpunkt vom Typ `DateTime` angeben. Darauf verzichtet das Beispiel und übergibt stattdessen im fünften Parameter die schon beschriebene Mindest-Lebensdauer nach dem letzten Zugriff.

Der sechste Parameter legt die Priorität des Eintrags fest. Bei knappem Speicher werden die Einträge mit geringer Priorität zuerst gelöscht. Je höher die »Kosten« für die Erzeugung des Eintrags sind und je öfter er gebraucht wird, desto höher sollte man also seine Priorität setzen.

Als letzten Parameter erhält die Methode einen Delegaten für eine Rückruf-Methode, die beim Löschen des Eintrags aufgerufen wird. Aufgabe der Methode ist hier die Protokollierung der Löschoperationen. Das Speichern dokumentiert die nächsten Zeile durch Aufruf der Methode `Protokolliere()`.

Das detaillierte Protokollieren der einzelnen Speicher- und Lösch-Vorgänge erleichtert das Experimentieren. Dazu dienen die folgenden Methoden:

```
private string FormatiereOptionen(TimeSpan ts, CacheItemPriority cip)
{
  return "Dauer=" + ts.Seconds + "; Priorität=" + cip.ToString();
}//FormatiereOptionen

[WebMethod]
public string GibProtokollInhalt()
{
  string s = "Protokoll leer";
  StringBuilder Protokoll =
    (StringBuilder)Context.Cache.Get("Protokoll");
  if (Protokoll != null)
    s = Protokoll.ToString();
  return s;
}//GibProtokollInhalt

[WebMethod]
public void LeereProtokoll()
{
  StringBuilder Protokoll =
    (StringBuilder)Context.Cache.Get("Protokoll");
  if (Protokoll != null)
    Context.Cache.Remove("Protokoll");
}//LeereProtokoll

public void LöschRückruf(
  string Schlüssel, object O, CacheItemRemovedReason Grund)
{
  string s = "gelöscht (" + Grund.ToString() + ")";
  Protokolliere(Schlüssel, "", O, s);
}//LöschRückruf

private void Protokolliere(
  string Schlüssel, string Optionen, int Größe, string Operation)
{
  StringBuilder Protokoll =
    (StringBuilder)Context.Cache.Get("Protokoll");
  if (Protokoll == null)
    Protokoll= new StringBuilder();
  Protokoll.Append(DateTime.Now.ToLongTimeString() + ": " +
    Schlüssel + " (Größe=" + Größe.ToString() + "; " +
    Optionen + ") " + Operation + "\r\n");
  Context.Cache.Insert("Protokoll", Protokoll);
}//Protokolliere
```

Die Methode `LöschRückruf()` wird automatisch beim Löschen eines Eintrags aufgerufen und erhält dabei neben Schlüssel und Wert des gelöschten Eintrags auch den Grund für die Löschung. Schlüssel und Grund reicht sie an `Protokolliere()` weiter. Diese Methode führt das Protokoll auf der Basis eines `StringBuilder`-Objekts – natürlich ebenfalls im Cache. Wie zu sehen, ist die `Insert()`-Methode auch mit erheblich weniger Parametern zufrieden.

Der Inhalt des Protokolls ist über `GibProtokollInhalt()` abrufbar; das gesamte Protokoll kann über `LeereProtokoll()` gelöscht werden. `FormatiereOptionen()` formt aus den übergebenen Parametern einen String, der dann an `Protokolliere()` übergeben werden kann.

Jetzt fehlt nur noch die Methode `LöscheEintrag()`, die einen Eintrag unabhängig von den beim Speichern angegebenen Löschregeln aus dem Cache entfernt.

```
[WebMethod]
public void LöscheEintrag(string Schlüssel)
{
  Context.Cache.Remove(Schlüssel);
}//LöscheEintrag
```

Fügen Sie noch `using`-Anweisungen für `System.Web.Caching` und `System.Text` hinzu. Wenn die Kompilierung fehlerfrei verläuft, können Sie sich dann der Clientseite zuwenden.

Für den Client fügen Sie der Projektmappe ein Projekt vom Typ *Windows-Anwendung* hinzu, setzen Sie es als Startprojekt und fügen Sie eine Webreferenz auf den Webdienst hinzu. Setzen Sie noch die nötigen Steuerelemente auf das Formular und stellen Sie die in Tabelle 23.5 gezeigten Eigenschaften ein.

Tabelle 23.5: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	`Text`	Caching Tester
Oberes Gruppenfeld	`Text`	Cache Inhalt
Obere Schaltfläche im oberen Gruppenfeld	`Name`	btnCacheinhaltZeigen
Obere Schaltfläche im oberen Gruppenfeld	`Text`	Zeigen
Untere Schaltfläche im oberen Gruppenfeld	`Name`	btnCacheLeeren
Untere Schaltfläche im oberen Gruppenfeld	`Text`	Leeren

Tabelle 23.5: Eigenschaftswerte (Forts.)

Steuerelement	Eigenschaft	Wert
Oberes Kontrollkästchen im oberen Gruppenfeld	`Name`	chkMitSystemEinträgen
Oberes Kontrollkästchen im oberen Gruppenfeld	`Text`	MitSystemEinträgen
Unteres Kontrollkästchen im oberen Gruppenfeld	`Name`	chkMitWert
Unteres Kontrollkästchen im oberen Gruppenfeld	`Text`	MitWert
Textfeld im oberen Gruppenfeld	`Name`	txtCacheInhalt
Textfeld im oberen Gruppenfeld	`Multiline`	True
Textfeld im oberen Gruppenfeld	`ScrollBars`	Both
Textfeld im oberen Gruppenfeld	`WordWrap`	False
Linkes Gruppenfeld	`Text`	Caching testen
Linkes oberes Label im linken Gruppenfeld	`Text`	Schlüssel
Rechtes oberes Label im linken Gruppenfeld	`Text`	Größe
Linkes oberes Textfeld im linken Gruppenfeld	`Name`	txtSchlüssel
Rechtes oberes Textfeld im linken Gruppenfeld	`Name`	txtGröße
Mittleres Label im linken Gruppenfeld	`Text`	Priorität
Kombinationslistenfeld	`Name`	cboPriorität
Kombinationslistenfeld	`DropDownStyle`	DropDownList
Kombinationslistenfeld	`Items`	1 Low 2 BelowNormal 3 Default 4 AboveNormal 5 High 6 NotRemovable
Unteres Label im linken Gruppenfeld	`Text`	Dauer [Sek.]

Tabelle 23.5: Eigenschaftswerte (Forts.)

Steuerelement	Eigenschaft	Wert
Unteres Textfeld im linken Gruppenfeld	`Name`	txtDauer
Linke Schaltfläche im linken Gruppenfeld	`Name`	btnSpeichern
Linke Schaltfläche im linken Gruppenfeld	`Text`	Speichern
Rechte Schaltfläche im linken Gruppenfeld	`Name`	btnLöschen
Rechte Schaltfläche im linken Gruppenfeld	`Text`	Löschen
Rechtes Gruppenfeld	`Text`	Protokoll
Textfeld im rechten Gruppenfeld	`Name`	txtProtokoll
Textfeld im rechten Gruppenfeld	`Multiline`	True
Textfeld im rechten Gruppenfeld	`ScrollBars`	Both
Textfeld im rechten Gruppenfeld	`WordWrap`	False
Linke Schaltfläche im rechten Gruppenfeld	`Name`	btnProtokollZeigen
Linke Schaltfläche im rechten Gruppenfeld	`Text`	Zeigen
Rechte Schaltfläche im rechten Gruppenfeld	`Name`	btnProtokollLeeren
Rechte Schaltfläche im rechten Gruppenfeld	`Text`	Leeren
Untere Schaltfläche	`Name`	btnBeenden
Untere Schaltfläche	`Text`	Beenden

Nach soviel Oberflächen-Design ist der einzugebende Code erfreulich kurz. Deklarieren Sie zuerst für den Webdienst ein Datenfeld in der Formularklasse:

```
private localhost.CacheService m_cs = null;
```

Erweitern Sie dann den Konstruktor des Formulars:

```
public Form1()
{
  InitializeComponent();

  cboPriorität.SelectedIndex = 2;
  m_cs = new localhost.CachingService();
  m_cs.CookieContainer = new CookieContainer();
  txtGröße.Text = "10000000";
  txtDauer.Text = "0";
}
```

Jetzt noch ein paar Zeilen für die Schaltflächen:

```
private void btnBeenden_Click(object sender, System.EventArgs e)
{
  Close();
}

private void btnCacheinhaltZeigen_Click(
  object sender, System.EventArgs e)
{
  String s = m_cs.GibCacheInhalt(
    chkMitSystemEinträgen.Checked, chkMitWert.Checked);
  txtCacheInhalt.Text = FügeCrEin(s);
}

private void btnCacheLeeren_Click(object sender, System.EventArgs e)
{
  m_cs.LeereCache();
}
private void btnLöschen_Click(object sender, System.EventArgs e)
{
  m_cs.LöscheEintrag(txtSchlüssel.Text);
}

private void btnProtokollLeeren_Click(
  object sender, System.EventArgs e)
{
  m_cs.LeereProtokoll();
}

private void btnProtokollZeigen_Click(
  object sender, System.EventArgs e)
{
  txtProtokoll.Text = FügeCrEin(m_cs.GibProtokollInhalt());
}

private void btnSpeichern_Click(object sender, System.EventArgs e)
{
  m_cs.SpeichereEintrag(
    txtSchlüssel.Text, int.Parse(txtDauer.Text),
    cboPriorität.SelectedIndex + 1, int.Parse(txtGröße.Text));
}
```

```
private string FügeCrEin(string Original)
{
  //Bedingt durch die XML-Spezifikation kommen Zeilenumbrüche
  //statt als CrLf nur als Lf an. FügeCrEin() kompensiert das.
  return Original.Replace("\n", "\r\n");
}
```

Bis auf die Methode `FügeCrEin()` also nichts Unerwartetes. Und `FügeCrEin()` tut genau, was Name und Kommentar sagen.

Beim Experimentieren mit dem Programm werden Sie vielleicht ein überraschendes Verhalten feststellen: Statt bei Speichermangel einen oder mehrere Einträge aus dem Cache zu löschen, fliegt der ganze Prozess aus dem Speicher und wird automatisch neu gestartet. Das lässt sich am besten mit dem Task-Manager beobachten, *AspNet_Wp.Exe* heißt der zuständige Prozess.

Hier handelt es sich aber nicht um einen Bug, sondern um ein Feature: Während unter ASP ein fehlerhaftes Programm soviel Speicher belegen konnte, dass der Rechner schließlich neu gestartet werden musste, erledigt ASP.NET dieses Problem durch Beenden und Neustarten des Prozesses automatisch. Wie viel Speicher der Prozess maximal belegen darf, können Sie im *processModel*-Element der *Machine.config*-Datei durch das *memoryLimit*-Attribut einstellen. Und mit dem *requestQueueLimit*-Attribut dieses Elements können Sie gleich noch ein weiteres Problem bekämpfen: Deadlocks.

Leider lässt sich nicht feststellen, wie viel Speicher das `Cache`-Objekt belegt. Außerdem enthält es neben den vom Programm explizit gespeicherten Einträgen noch alles, was im `Session`-Objekt (im Prozess) oder von ASP.NET im Ausgabe-Cache für die Seite gespeichert wurde. Über die Leistungsindikatoren ist zumindest die Anzahl der Einträge abfragbar.

Auf jeden Fall muss man sich also nicht nur eine sinnvolle Caching-Strategie überlegen, sondern ihr Funktionieren auch ausgiebig testen. Mit verschiedenen Speicherorten des `Session`-Objekts und dem kreativen Einsatz von Datei-Abhängigkeiten beim Speichern im `Cache`-Objekt sollten sich aber auch komplizierte Szenarien in Web-Farmen in den Griff bekommen lassen.

Zum Schluss des Abschnitts noch zwei Bemerkungen zu den Möglichkeiten des `Cache`-Objekts. Zum einen bietet es neben der `Insert()`- noch eine `Add()`-Methode mit folgenden Unterschieden:

- `Add()` tut nichts, wenn ein Eintrag mit dem angegebenen Schlüssel schon existiert. Ausnahmen werden in diesem Fall nicht erzeugt.
- `Add()` ist nicht überladen, es existiert nur eine Version mit sieben Parametern.
- `Add()` liefert den eingefügten Eintrag zurück.

Außerdem existiert keine eingebaute Möglichkeit, `Cache`-Einträge bei Veränderungen in einer Datenbank ungültig werden zu lassen. Im SQL-Server kann man sich dazu beispielsweise eine erweiterte Stored Procedure schreiben.

23.6 Webdienste erweitern

Dieser Abschnitt zeigt zwei fortgeschrittene Techniken bei der Implementation von Webdiensten: SOAP-Erweiterungen und SOAP-Header. Zu beiden Techniken finden Sie zuerst eine grundlegende Erläuterung, darauf folgt ein Praxisbeispiel, das sie gemeinsam demonstriert.

23.6.1 SOAP-Erweiterungen verwenden

Eine SOAP-Erweiterung ermöglicht den Zugriff auf den Datenstrom eines SOAP-basierten Webdienstes, und zwar in verschiedenen Zuständen. Dabei können Sie den Datenstrom nicht nur auswerten, sondern auch verändern. SOAP-Erweiterungen bieten sich dadurch beispielsweise zum Komprimieren oder Verschlüsseln der Daten an. Ein Beispiel zum Protokollieren der übertragenen Daten finden Sie in der Online-Dokumentation unter »SOAP, Erweiterungen«.

Bild 23.10 zeigt eine SOAP-Erweiterung im Datenfluss eines Webdienstes.

Die HTTP-Laufzeitumgebung der IIS (InetInfo.Exe) leitet Anfragen von Clients, die sich auf URLs mit der Namenserweiterung *asmx* beziehen, in die HTTP Pipeline im Prozess *AspNet_Wp.Exe* weiter. In dieser Pipeline wandert die Anfrage, gegebenenfalls nach Passieren von HTTP-Modulen, zum HTTP-Handler für Webdienste. Dieser HTTP-Handler stellt nicht nur den Kontext für die eigentliche Anwendung bereit, sondern bietet durch SOAP-Erweiterungen auch die Möglichkeit zum Zugriff auf die übertragenen Daten.

Eine SOAP-Erweiterung kann beim Senden und Empfangen an jeweils zwei Punkten auf den Datenfluss zugreifen:

1. *BeforeDeserialize* – die Anfrage ist in dem Format, in dem sie den HTTP-Handler erreichte, im Allgemeinen geht es hier also um XML, das dem SOAP-Standard entspricht.
2. *AfterDeserialize* – die Anfrage ist in dem Format, in dem der Methodenaufruf stattfinden wird. Hier handelt es sich um .NET-Objekte.
3. *BeforeSerialize* – die Antwort ist in dem Format, das der Methodenaufruf geliefert hat. Hier handelt es sich um .NET-Objekte.
4. *AfterSerialize* – die Antwort ist in dem Format, in dem sie an den Client gesendet wird. Im Allgemeinen geht es hier also wieder um XML, das dem SOAP-Standard entspricht.

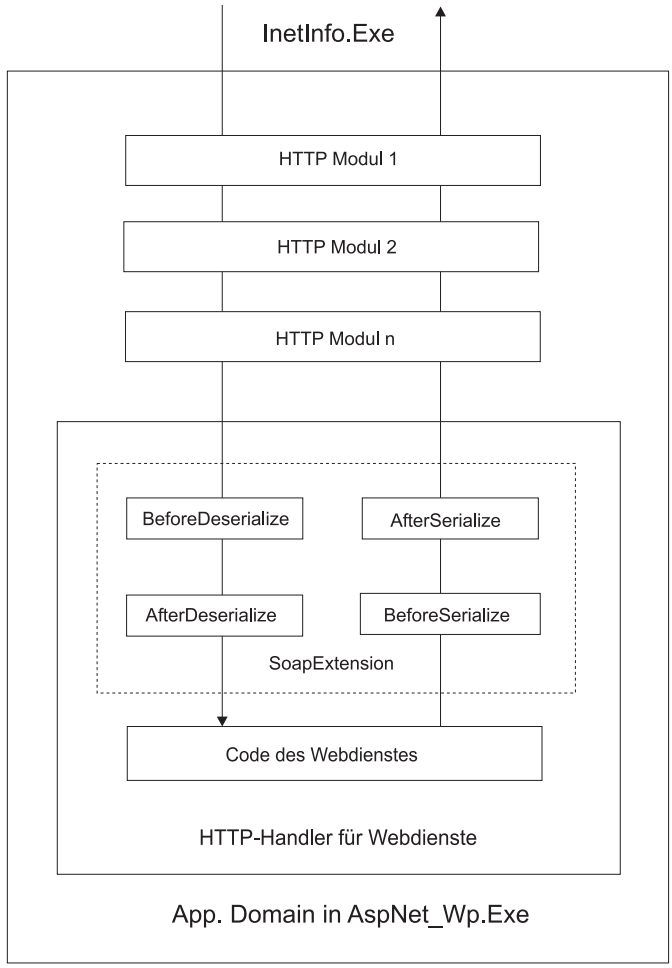


Abbildung 23.10: Datenfluss eines Webdienstes

Je nach Aufgabenstellung wird eine SOAP-Erweiterung an unterschiedlichen Punkten des Datenflusses eingreifen. Zum Beispiel wird eine SOAP-Erweiterung zum Komprimieren oder Protokollieren der Daten ihre Arbeit im Rahmen von *AfterSerialize* erledigen.

Eine SOAP-Erweiterung lässt sich sowohl auf alle als auch (im Gegensatz zu HTTP-Modulen) nur auf ausgewählte Methoden eines Webdienstes anwenden. Außerdem lassen sich beliebig viele SOAP-Erweiterungen nacheinander einsetzen. SOAP-Erweiterungen können auch clientseitig installiert werden.

Die Klasse SoapExtension

Jede SOAP-Erweiterung entsteht durch Ableitung von der Klasse `SoapExtension`. Dazu ist mindestens die Implementation folgender Methoden nötig:

- `ProcessMessage()` – diese Methode wird an jedem der vier genannten Punkte aufgerufen und liefert in ihrem Parameter ein Objekt vom Typ `SoapMessage`. Dieses Objekt ermöglicht durch seine Eigenschaften und Methoden den Zugriff auf die Daten, beispielsweise enthält `Stream` den gesamten Datenstrom, allerdings nur für den Lesezugriff. Außerdem lässt sich anhand der `Stage`-Eigenschaft feststellen, an welchem Punkt des Datenflusses die Methode aufgerufen wurde.
- `Initialize()` – diese Methode wird beim Instanziieren der SOAP-Erweiterung aufgerufen. Das geschieht beim Aufruf jeder Methode, auf die diese `SoapExtension-Klasse` angewendet werden soll. Die `SoapExtension`-Instanz bleibt dann bis zur Rückkehr des Methodenaufrufs bestehen. `Initialize()` wird also zu häufig aufgerufen, um darin aufwändige Initialisierungen durchzuführen, wie zum Beispiel das Herstellen einer Datenbankverbindung. Deshalb erhält `Initialize()` einen Parameter vom Typ `Object`, der die in `GetInitializer()` zurückgelieferten `Initialisierungs-Informationen transportiert`.
- `GetInitializer()`- diese Methode wird beim ersten Instanziieren der SOAP-Erweiterung aufgerufen. Das ist der erste Aufruf einer Methode, auf die diese `SoapExtension-Klasse` angewendet werden soll. `GetInitializer()` existiert in zwei überladenen Versionen und liefert `Initialisierungs-Informationen als Wert vom Typ Object zurück`. Dieser Wert wird während der Lebensdauer der Anwendung gespeichert und jedem Aufruf von `Initialize()` als Parameter übergeben.

Ebenfalls wichtig ist die Methode `ChainStream()`: Sie muss überschrieben werden, wenn Sie Daten verändern wollen. `ChainStream()` erhält als Parameter ein `Stream`-Objekt mit den Original-Daten und liefert ein `Stream`-Objekt mit den veränderten Daten. Allerdings müssen Sie die Daten auf jeden Fall in das neue `Stream`-Objekt kopieren, auch wenn sie gar nicht verändert wurden.

Damit mehrere SOAP-Erweiterungen nacheinander auf eine Methode angewendet werden können, muss eine entsprechende Konfigurationsmöglichkeit existieren. Dazu können Sie die Gruppe und die Priorität einer SOAP-Erweiterung festlegen: Je niedriger der jeweilige Wert ist, desto näher sitzt die SOAP-Erweiterung an den übertragenen Daten. Die SOAP-Erweiterung mit dem niedrigsten Wert wird also für ankommende Daten als erste aufgerufen, für zu sendende als letzte. Dabei zählt zuerst der Gruppenwert (*0* oder *1*), erst bei gleichem Gruppenwert zählt der Prioritätswert.

Wie Sie den Gruppen- und den Prioritätswert setzen, hängt von der Konfigurationsart ab. Zum einen können Sie durch Hinzufügen eines Elements in der entsprechenden *Web.config*- oder der *Machine.config*-Datei die SOAP-Erweiterung auf alle Methoden der durch diese Datei konfigurierten Webdienste anwenden. Für eine SOAP-Erweiterung der Klasse `Abrechnung.AbrechnungsExtension` in der Assembly `Abrechnung` sähe das Element so aus:

```
<system.web>
  <webServices>
    <soapExtensionTypes>
      <add type="Abrechnung.AbrechnungsExtension, Abrechnung"
        priority="1" group="0"/>
     </soapExtensionTypes>
  </webServices>
</system.web>
```

Alternativ können Sie eine SOAP-Erweiterung auf ausgewählte Methoden eines Webdienstes anwenden. Dazu dekorieren Sie die entsprechenden Methoden mit einem Attribut, das vom Typ `SoapExtensionAttribute` abgeleitet wurde.

Die Klasse SoapExtensionAttribute

Die Klasse `SoapExtensionAttribute ist die Basisklasse für alle Attribute zur Anwendung von` SOAP-Erweiterungen. In Ihrer Ableitung müssen Sie mindestens diese Eigenschaften implementieren:

- `ExtensionType` – liefert den Typ der anzuwendenden SOAP-Erweiterung.
- `Priority` – liefert den Prioritätswert.

Die Instanz Ihrer `SoapExtensionAttribute`-Klasse wird an die Methode `GetInitializer()` der `SoapExtension`-Instanz übergeben, sodass eine Parametrisierung durch Eigenschaften der `SoapExtensionAttribute`-Klasse möglich ist. `GetInitializer()` enthält dabei auch noch einen Parameter des Typs `LogicalMethodInfo`, Informationen wie den Methodennamen braucht eine `SoapExtensionAttribute`-Instanz also nicht zu transportieren.

Übrigens wurde hier nicht etwa die `Group`-Eigenschaft vergessen: SOAP-Erweiterungen, die über ein Attribut angewendet werden, bekommen automatisch einen mittleren Gruppenwert (sollte hier der Wert 0,5 gemeint sein?).

23.6.2 SOAP-Header verwenden

SOAP-Header sind ein optionaler Bestandteil von SOAP-Nachrichten. Sie werden zwar mit der Nachricht transportiert, aber außerhalb des *body*-Elements. Dementsprechend transportieren sie Daten, die nichts mit dem eigentlichen Methodenaufruf zu tun haben: zum Beispiel eine BenutzerId oder eine SessionId. SOAP-Header können auch vom Server zu Client gesendet werden, im Folgenden wird aber erst einmal die andere Richtung betrachtet.

SOAP-Header werden in .NET von der `SoapHeader`-Klasse abgeleitet, in Ihrer Ableitung definieren Sie die für Ihren Header nötigen Eigenschaften.

Dann definieren Sie eine `public`-Variable dieses Typs in der Klasse Ihres Webdienstes. Dieser Variable weist .NET später automatisch den übertragenen Header zu.

Die Methoden, bei deren Aufruf der SOAP-Header übertragen werden soll, dekorieren Sie schließlich mit einem Attribut vom Typ `SoapHeaderAttribute`. Dabei können Sie über benannte Parameter festlegen:

- `Direction` – die Transportrichtung des Headers. Möglich sind `In` (Standardwert), `Out` und `InOut`.
- `Required` – ist der Header erforderlich? Möglich sind `true` (Standardwert bei `Direction = In`) und `false`. Wenn ein erforderlicher Header fehlt, wird automatisch eine `Ausnahme` ausgelöst.
- `MemberName` – der Name der `public`-Variablen zur Aufnahme des Headers.

Laut SOAP-Spezifikation kann der Client verlangen, dass der Server einen Header versteht, also richtig verarbeitet. In diesem Fall steht die geerbte `MustUnderstand`-Eigenschaft der übergebenen `SoapHeader`-Instanz auf `true`, und der Server muss die Verarbeitung quittieren. Das geschieht durch Setzen ihrer `DidUnderstand`-Eigenschaft auf `true`, anderenfalls löst .NET entsprechend der SOAP-Spezifikation eine `SoapHeaderException` aus, die als *SoapFault*-Element übertragen wird.

23.6.3 Codebeispiel – SOAP-Header und SOAP-Erweiterungen

Das in diesem Abschnitt vorgestellte Beispielprojekt *WsGrafikSpender* implementiert einen kostenpflichtigen Grafikspender: Der Benutzer gibt einen Text und Design-Festlegungen an, wie die zu verwendende Schriftart, und das Programm liefert eine entsprechende Grafik. Diese Grafik kann dann zum Beispiel als Schaltfläche in einer Webseite verwendet werden.

Derartig wertvolle Dienste können natürlich nicht kostenlos sein, deshalb protokolliert das Programm für die spätere Rechnungsstellung über eine SOAP-Erweiterung die Anzahl der Zeichen im Text.

Und damit der Benutzer zur Rechnungsstellung identifiziert werden kann, fordert die grafikspendende Methode einen SOAP-Header mit seiner BenutzerId.

Ausbaustufe 1 – der reine Grafikspender

Die Projektmappe *WsGrafikSpender* enthält wie üblich neben dem Webdienst auch einen Test-Client. Dessen Oberfläche zeigt Abbildung 23.11.

Abbildung 23.11:
Der Test-Client



Um das Beispiel nachzustellen, legen Sie als Erstes das Projekt für den Webdienst an, setzen Sie eine Referenz auf *System.Drawing.Dll* und fügen Sie die folgenden `using`-Anweisungen hinzu:

```
using System.Drawing;//Benötigt Referenz auf System.Drawing.Dll
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
using System.IO;
```

Der Webdienst stellt drei Methoden zur Verfügung.

```
    [WebMethod(Description=
  "Liefert eine PNG-Grafik mit dem übergebenen Text und Design.")]
public byte[] ErzeugeGrafik(
  string Text, string SchriftName, float SchriftGröße,
  string HgFarbe, string VgFarbe, int Schatten, float Scherung)
{
  const int Auflösung = 96;//96 DPI für kleine Schriftarten
  Bitmap bm = null;
  Brush b = null;
  Color c = Color.White;
  Font f = null;
  Graphics g = null;
  int ScherungsOffset = 0;

  try
  {
    b = new SolidBrush(Color.FromName(VgFarbe));
    c = Color.FromName(HgFarbe);
    //Auflösungs- bzw. Geräte-unabhängig die Bitmap-Größe ermitteln
    f = new Font(SchriftName, SchriftGröße);
    bm = new Bitmap(1, 1);
    bm.SetResolution(Auflösung, Auflösung);
```

```
    g = Graphics.FromImage(bm);
    Size sz = g.MeasureString(Text, f).ToSize();
    //Annahme: Schatten >= 0
    sz.Height += Schatten;
    sz.Width += Schatten;
    //Bei Scherung wird der String breiter und steht nicht mehr mittig,
    //einigermaßen ausbügeln:
    if (Scherung != 0)
    {
      int ScherungsZuschlag = Convert.ToInt32(
        g.MeasureString("M", f).ToSize().Width * Math.Abs(Scherung));
      sz.Width += ScherungsZuschlag;
      if (Scherung > 0)
        ScherungsOffset =
          ScherungsZuschlag - Convert.ToInt32(Scherung * 40);
      else
        ScherungsOffset = ScherungsZuschlag + Convert.ToInt32(Scherung);
    }

    //Grafik erzeugen
    bm = new Bitmap(bm, sz);
    bm.SetResolution(Auflösung, Auflösung);
    g = Graphics.FromImage(bm);
    if (Scherung != 0)
    {
      Matrix m = new Matrix();
      m.Shear(Scherung, 0);
      g.Transform = m;
    }
    g.Clear(c);
    if (Schatten != 0)
    {
      g.DrawString(Text, f, Brushes.Black,
        ScherungsOffset + Schatten, Schatten);
    }
    g.DrawString(Text, f, b, ScherungsOffset, 0);
  }
  finally {
    b.Dispose();
    f.Dispose();
    g.Dispose();
  }
  //In byte-Array konevertieren und zurückgeben.
  //XmlSerializer kodiert byte-Arrays automatisch mit Base64.
  MemoryStream ms = new MemoryStream();
  bm.Save(ms, ImageFormat.Png);
  return ms.ToArray();
}//ErzeugeGrafik

[WebMethod(Description=
  "Liefert die Namen der auf dem Server verfügbaren Farben.")]
public string[] GibFarbNamen()
{
  return Enum.GetNames(typeof(KnownColor));
}//GibFarbNamen
```

```
[WebMethod(Description=
  "Liefert die Namen der auf dem Server verfügbaren Schriften.")]
public string[] GibSchriftNamen()
{
  FontFamily[] arFf = FontFamily.Families;
  string[] arStr = new string[arFf.Length];
  for (int i = 0; i < arFf.Length; i++)
    arStr[i] = arFf[i].Name;
  return arStr;
}//GibSchriftNamen
```

Die (später) kostenpflichtige Methode heißt `ErzeugeGrafik()` und liefert die mit den übergebenen Parametern erzeugte Grafik im PNG-Format (Portable Network Graphics – der GIF-Nachfolger) zurück. Erfreulich ist, dass das zurückgegebene `byte`-Array beim Übertragen automatisch im Base64-Format kodiert wird und damit SOAP- bzw. XML-tauglich ist. Die Implementationsdetails sind hier sekundär und zum Beispiel im aktuellen Petzold nachzulesen (Charles Petzold: Windows Programmierung mit C#, Microsoft Press).

Die Methoden `GibFarbNamen()` und `GibSchriftNamen()` liefern `string`-Arrays mit den verfügbaren Farben und Schriftarten, die der Client dann in entsprechenden Listenfeldern anbieten kann.

Kompilieren Sie das Projekt, fügen Sie der Projektmappe ein neues Projekt vom Typ *Windows-Anwendung* hinzu und setzen Sie dort eine Webreferenz auf den Webdienst.

Setzen Sie die in Abbildung 23.11 zu sehenden Steuerelemente auf das Formular des Test-Clients und stellen Sie die in Tabelle 23.6 gezeigten Eigenschaften ein.

Tabelle 23.6: Eigenschaftswerte

Steuerelement	Eigenschaft	Wert
Formular	`Text`	Grafikspender-Test
Formular	`AcceptButton`	btnErzeugen
Oberstes Label	`Text`	Beschriftung
2. Label	`Text`	Schriftname
3. Label	`Text`	Schriftgröße
4. Label	`Text`	HgFarbe
5. Label	`Text`	VgFarbe
6. Label	`Text`	Schatten
7. Label	`Text`	Scherung

Tabelle 23.6: Eigenschaftswerte (Forts.)

Steuerelement	Eigenschaft	Wert
Oberstes Textfeld	Name	txtBeschriftung
2. Textfeld	Name	txtSchriftgröße
Oberstes Kombinationslistenfeld	Name	cboSchriftName
Oberstes Kombinationslistenfeld	DropDownStyle	DropDownList
2. Kombinationslistenfeld	Name	cboHgFarbe
2. Kombinationslistenfeld	DropDownStyle	DropDownList
3. Kombinationslistenfeld	Name	cboVgFarbe
3. Kombinationslistenfeld	DropDownStyle	DropDownList
Oberstes Auf-Ab-Steuerelement	Name	updSchatten
Oberstes Auf-Ab-Steuerelement	Maximum	5
2. Auf-Ab-Steuerelement	Name	updScherung
2. Auf-Ab-Steuerelement	DecimalPlaces	2
2. Auf-Ab-Steuerelement	Increment	0,25
2. Auf-Ab-Steuerelement	Maximum	1,5
2. Auf-Ab-Steuerelement	Minimum	-1,5
Obere Schaltfläche im Gruppenfeld	Text	ZeitAbrufen
Schaltfläche	Name	btnErzeugen
Schaltfläche	Text	Erzeugen
Bildfeld	Name	pictureBox1
Bildfeld	Dock	Bottom

Deklarieren Sie für den Webdienst noch eine Variable in der Formularklasse.

```
private localhost.Generator m_gen = null;
```

Erweitern Sie dann den Konstruktor des Formulars folgendermaßen zum Initialisieren der Steuerelemente:

```
    public Form1()
{
  InitializeComponent();

   m_gen = new localhost.Generator();
  cboSchriftName.Items.AddRange(m_gen.GibSchriftNamen());
  cboSchriftName.SelectedIndex = 0;
  txtSchriftgröße.Text = "24";

  cboHgFarbe.Items.AddRange(m_gen.GibFarbNamen());
  cboHgFarbe.SelectedIndex = 30;

  cboVgFarbe.Items.AddRange(m_gen.GibFarbNamen());
  cboVgFarbe.SelectedIndex = 50;
}
```

Hier ist schließlich noch der Code für die ERZEUGEN-Schaltfläche:

```
private void btnErzeugen_Click(object sender, System.EventArgs e)
{
  localhost.Kunde kd = new localhost.Kunde();
  Byte[] Puffer = m_gen.ErzeugeGrafik(txtBeschriftung.Text,
    cboSchriftName.Text, float.Parse(txtSchriftgröße.Text),
    cboHgFarbe.Text, cboVgFarbe.Text,
    Decimal.ToInt32(updSchatten.Value),
    Decimal.ToSingle(updScherung.Value));
  MemoryStream ms = new MemoryStream(Puffer);
  pictureBox1.Image = new Bitmap(ms);
}
```

Damit wäre der Grafikspender betriebsbereit.

Ausbaustufe 2 – SOAP-Erweiterung zum Protokollieren einsetzen

Der im vorherigen Abschnitt vorgestellte Grafikspender wird in diesem Beispiel um eine SOAP-Erweiterung zum Protokollieren der erzeugten Grafiken erweitert. Fügen Sie der Projektmappe dafür ein neues Projekt vom Typ *Klassenbibliothek* namens *Abrechnung* hinzu. Dieses Projekt wird sowohl das Erweiterungsattribut als auch die SOAP-Erweiterung aufnehmen.

Setzen Sie eine Referenz auf *System.Web.Services.Dll* und fügen Sie die folgenden `using`-Anweisungen ein:

```
using System.Web.Services.Protocols;
using System.Xml;
using System.IO;
```

Hier ist der Code für das Erweiterungsattribut:

```
[AttributeUsage(AttributeTargets.Method)]
public class AbrechnungsAttribute : SoapExtensionAttribute {
  int m_Priorität = 0;
```

```
  public override Type ExtensionType {
    get { return typeof(AbrechnungsExtension); }
  }//ExtensionType
  public override int Priority {
    get { return m_Priorität; }
    set { m_Priorität = value; }
  }//Priority
}//class AbrechnungsAttribut
```

Und schließlich der Code für die SOAP-Erweiterung selbst:

```
public class AbrechnungsExtension : SoapExtension {
  private XmlDocument m_doc = null;

  public override object GetInitializer(
    LogicalMethodInfo MethodInfo, SoapExtensionAttribute Attribut)
  {
    return null;
  }//GetInitializer


  public override object GetInitializer(Type WebServiceType)
  {
    return null;
  }//GetInitializer


  public override void Initialize(object Initialisierer)
  {
    m_doc = new XmlDocument ();
    try
    {
      m_doc.Load("C:\\Abrechnung.Xml");
    }
    catch (FileNotFoundException)
    {
      m_doc.LoadXml("<Grafiken/>");
    }
  }//Initialize


  public override void ProcessMessage(SoapMessage Nachricht)
  {
    if(Nachricht.Stage == SoapMessageStage.AfterDeserialize)
    {
      //Grafik-Element erzeugen
      XmlElement Grafik = m_doc.CreateElement("Grafik");
      XmlElement Grafiken = (XmlElement)m_doc.SelectSingleNode("Grafiken");
      Grafiken.AppendChild(Grafik);
      //Zeitpunkt-Attribut erzeugen
      XmlAttribute Zeitpunkt = m_doc.CreateAttribute("Zeitpunkt");
      Zeitpunkt.Value = DateTime.Now.ToString();
      Grafik.Attributes.Append(Zeitpunkt);
      //Text-Element erzeugen
      XmlElement Text = m_doc.CreateElement("Text");
```

```
      Text.InnerText = (string)Nachricht.GetInParameterValue(0);
      Grafik.AppendChild(Text);
      m_doc.Save("C:\\Abrechnung.Xml");
    }
  }//ProcessMessage

}//class AbrechnungsExtension
```

Das Einlesen des XML-Dokuments zum Protokollieren der erzeugten Grafiken erfolgt der Einfachheit halber in der `Initialize()`-Methode, normalerweise würde man es in den `GetInitializer()`-Methoden tun.

Die Methode `ProcessMessage()` prüft zunächst, ob der Aufruf am richtigen Punkt im Datenfluss erfolgte. Dann erzeugt sie ein neues *Grafik*-Element mit einem Zeitstempel im Attribut und einem Kind-Element, das den Text für die Grafik enthält. Dieser Wert ist nur an diesem Punkt des Datenflusses über `GetInParameterValue()` verfügbar.

Die hier vorgestellte Implementation einer SOAP-Erweiterung ist relativ einfach. Wesentlich aufwändiger wird es, wenn Sie Daten verändern wollen. Im Internet finden Sie dazu diverse Implementationen zur Verschlüsselung von Daten.

Kompilieren Sie jetzt das Projekt und setzen Sie eine Referenz darauf im Webdienst-Projekt. Um die SOAP-Erweiterung dort nutzen zu können, sind noch zwei Kleinigkeiten nötig. Zum einen dekorieren Sie die `ErzeugeGrafik()`-Methode des Webdienstes mit dem neuen Attribut, der Namensraum der `AbrechnungsAttribute`-Klasse lautete hier also `Abrechnung`:

```
[Abrechnung.AbrechnungsAttribute()]
[WebMethod(Description=
  "Liefert eine PNG-Grafik mit dem übergebenen Text und Design.")]
public byte[] ErzeugeGrafik(...)
{
  //...
}
```

Zum anderen müssen Sie noch die neue DLL von *<Projektverzeichnis>\bin\debug* nach *Inetpub\wwwroot\<Anwendungsverzeichnis>\bin* kopieren. Beim Aufruf der `ErzeugeGrafik()`-Methode findet sie .NET dort automatisch.

Wenn Sie die SOAP-Erweiterung auf alle Methoden des Webdienstes anwenden wollen, können Sie das mit dem folgenden Eintrag in der *Web.config*-Datei der Anwendung erledigen:

```
<system.web>
  <webServices>
    <soapExtensionTypes>
```

```
      <add type="Abrechnung.AbrechnungsExtension, Abrechnung"
        priority="1" group="0"/>
     </soapExtensionTypes>
  </webServices>
</system.web>
```

Der erste Wert im *type*-Attribut entspricht der Angabe im Attribut der `ErzeugeGrafik()`-Methode des Webdienstes, der Wert nach dem Komma ist der Name der Assembly.

Jetzt wird Ihre SOAP-Erweiterung auch für die Methoden `GibFarbNamen()` und `GibSchriftNamen()` aufgerufen, die beide keine Parameter haben. Der Code der SOAP-Erweiterung in `ProcessMessage()` würde deshalb eine Ausnahme auslösen. Die folgende Änderung verhindert das und trägt den Methodennamen in die XML-Datei ein:

```
public override void ProcessMessage(SoapMessage Nachricht)
{
  if(Nachricht.Stage == SoapMessageStage.AfterDeserialize)
  {
    //Grafik-Element erzeugen
    //...
    Grafik.Attributes.Append(Zeitpunkt);
    if (Nachricht.MethodInfo.Name == "ErzeugeGrafik")
    {
      //Text-Element erzeugen
      XmlElement Text = m_doc.CreateElement("Text");
      Text.InnerText = (string)Nachricht.GetInParameterValue(0);
      Grafik.AppendChild(Text);
    }
    else {
      //Text-Element erzeugen
      XmlElement Hilfsmethode = m_doc.CreateElement("Hilfsmethode");
      Hilfsmethode.InnerText = Nachricht.MethodInfo.Name;
      Grafik.AppendChild(Hilfsmethode);
    }
    m_doc.Save("C:\\Abrechnung.Xml");
  }
}//ProcessMessage
```

Der Webdienst kann jetzt die erzeugten Grafiken und sogar den Aufruf der Hilfsmethoden protokollieren, zur Rechnungsstellung fehlt noch die Identifikation des Kunden. Das geschieht über einen SOAP-Header.

Ausbaustufe 3 – Identifikation des Kunden über einen SOAP-Header

Die nun vorgestellte Implementation überträgt einfach eine KundenId beim Aufruf der `ErzeugeGrafik()`-Methode des Webdienstes.

Definieren Sie dazu eine entsprechende von `SoapHeader` abgeleitete Klasse im Projekt der SOAP-Erweiterung (der Namensraum ist hier `Abrechnung`).

```
public class Kunde : SoapHeader {
  public int Id;
}//class Kunde
```

Außerdem soll der neue Header natürlich mitprotokolliert werden.

```
public override void ProcessMessage(SoapMessage Nachricht)
{
  if(Nachricht.Stage == SoapMessageStage.AfterDeserialize)
  {
    //Grafik-Element erzeugen
    //...
    Grafik.Attributes.Append(Zeitpunkt);
    //KundenId-Attribut erzeugen
    XmlAttribute KundenId = m_doc.CreateAttribute("KundenId");
    foreach (SoapHeader sh in Nachricht.Headers)
    {
      if (sh is Kunde)
        KundenId.Value = ((Kunde)(sh)).Id.ToString();
    }
    Grafik.Attributes.Append(KundenId);
    if (Nachricht.MethodInfo.Name == "ErzeugeGrafik")
    {
      //...
    }
    m_doc.Save("C:\\Abrechnung.Xml");
  }
}
//ProcessMessage
```

Kompilieren Sie das Projekt neu, und kopieren Sie die DLL wieder in das `bin`-Verzeichnis der Anwendung.

Legen Sie dann im Webdienst-Projekt eine `public`-Variable vom Typ `Kunde` an und dekorieren Sie die `ErzeugeGrafik()`-Methode mit dem neuen Attribut.

```
public class Generator : System.Web.Services.WebService
{
  public Abrechnung.Kunde kd = null;
  public Generator()
  {
    //...
  }
  #region Component Designer generated code
  //...
  #endregion
  [SoapHeader("kd")]
  [Abrechnung.AbrechnungsAttribute()]
  [WebMethod(Description=
    "Liefert eine PNG-Grafik mit dem übergebenen Text und Design.")]
```

```
  public byte[] ErzeugeGrafik(...)
  {
    //.
  }
  //...
```

Dem Attribut-Konstruktor wird also der Name der `public`-Variablen übergeben, nicht etwa der Klassenname des Attributs.

In der Methode `ErzeugeGrafik()` könnten Sie auf die KundenId einfach in der Form `kd.Id` zugreifen. Wenn der Header optional wäre, müssten Sie vorher auf `null` prüfen. Eigentlich ist die `public`-Variable hier unnötig, denn die Auswertung des Headers erfolgt ja in der SOAP-Erweiterung. Aber wenn die Variable fehlt, kann die Webreferenz im Test-Client nicht aktualisiert werden.

Aktualisieren Sie jetzt die Webreferenz im Client-Projekt. Der folgende Auszug aus dem entstandenen Proxy-Code zeigt die für den Header wichtigen Zeilen.

```
namespace Client.localhost {
    //using ...;


    /// <remarks/>
    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.ComponentModel.DesignerCategoryAttribute("code")]
    [System.Web.Services.WebServiceBindingAttribute(
      Name="GeneratorSoap", Namespace="http://tempuri.org/")]
    public class Generator :
      System.Web.Services.Protocols.SoapHttpClientProtocol {

        public Kunde KundeValue;

        /// <remarks/>
        public Generator()
        {
            this.Url = "http://localhost/WsGrafikSpender/Generator.asmx";
        }

        /// <remarks/>
        [System.Web.Service  .Protocols.SoapHeaderAttribute("KundeValue")]
        [System.Web.Service  .Protocols.SoapDocumentMethodAttribute(
          "http://tempuri.org/ErzeugeGrafik",
          RequestNamespace="http://tempuri.org/",
          _u82 ?esponseNamespace="http://tempuri.org/",
          Use=System.Web.Services.Description.SoapBindingUse.Literal,
          ParameterStyle=
            System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
        [return:  System.Xml.Serialization.XmlElementAttribute(
          DataType="base64Binary")]
        public System.Byte[] ErzeugeGrafik(string Text, string SchriftName,
```

```
          System.Single SchriftGröße, string HgFarbe, string VgFarbe,
          int Schatten, System.Single Scherung)
        {
            object[] results = this.Invoke("ErzeugeGrafik", new object[] {
                        /*Parameterliste*/});
            return ((System.Byte[])(results[0]));
        }

        /// <remarks/>
        //Asynchrone Variante und Hilfsmethoden entfernt
        [System.Xml.Serialization.XmlTypeAttribute(
          Namespace="http://tempuri.org/")]
        [System.Xml.Serialization.XmlRootAttribute(
          Namespace="http://tempuri.org/", IsNullable=false)]
    public class Kunde : SoapHeader {

        /// <remarks/>
        public int Id;
    }
}
```

Jetzt fehlt nur noch das Setzen des Header-Werts im Client.

```
private void btnErzeugen_Click(object sender, System.EventArgs e)
{
  localhost.Kunde kd = new localhost.Kunde();
  kd.Id = 111;
  m_gen.KundeValue = kd;
  Byte[] Puffer = m_gen.ErzeugeGrafik(...) {
  }
  //...
}
```

Damit ist die Implementation des Beispiels abgeschlossen, aber natürlich lässt sie Wünsche offen. Abgesehen von »Kleinigkeiten« wie der Auswertung der XML-Datei ist das zum Beispiel ihr Aufbau: Ein Zusammenhang zwischen den Elementen für Grafikerzeugung und Aufruf der Hilfsfunktionen kann nur anhand der KundenIds und der Zeitstempel erraten werden. Dazu könnte man beispielsweise einen zusätzlichen Header mit einer SessionId benutzen. Und natürlich könnte man die `string`-Arrays mit den Namen der Farben und Schriftarten zwischenspeichern, und und und ...

Teil 5 Weiterführendes

Kapitel 24: Multithreading 999
Kapitel 25: .NET und COM 1041
Kapitel 26: Legacy Code und Windows API 1069

In diesem Teil geht es zuerst einmal um Multithreading, für das C# – wer hätte es gedacht – einige verblüffend einfache Lösungen und neue Konzepte in petto hat. Das darauffolgende Kapitel wirft dann einen Blick über den Zaun: Wie bestückt man existierende Software mit .NET-Komponenten und umgekehrt, obwohl die Online-Dokumentation hier kaum weiterhilft? COM- und ActiveX-Komponenten lassen sich geradlinig in .NET-Anwendungen integrieren, in umgekehrter Richtung geht es mit ein wenig Sorgfalt auch – was die Migration natürlich sehr erleichtert, weil man Existierendes stückweise ersetzen und nicht auf einen Schlag umbauen muss.

Auf- und Rückrufe der Windows-API sowie anderer DLLs – und vor allem, was darüber nicht in der Online-Dokumentation steht –, runden diesen Buchteil schließlich ab.

24 Multithreading

Dass .NET und seine Sprachen Multithreading unterstützen, also die Aufteilung einer Anwendung in mehrere Ausführungsstränge, versteht sich von selbst. Dieses Kapitel beschreibt die grundlegenden Klassen und Synchronisationsmechanismen, wobei – wie wäre es auch anders zu erwarten – Delegaten eine wichtige Rolle spielen.



Bevor es zur Sache geht, noch ein Hinweis, der in der besten aller Welten unnötig wäre: Die deutschen Übersetzer der Online-Dokumentation und insbesondere der von VS angezeigten Kurzbeschreibungen zeigen sich beim Thema Threads leider weitgehend überfordert. In einigen Fällen hilft die wortweise Rückübersetzung ins Englische (mit der »Planung« eines Threads ist eben das Scheduling gemeint, in diesem Kontext also die Zuteilung von Zeitscheiben bei der Vergabe von Rechenzeit), aber bei Behauptungen wie der, dass `Sleep(0)` *den aufrufenden Thread »anhält, sodass andere wartende Threads ausgeführt werden können« (richtig wäre: Abgabe der für die aktuelle Zeitscheibe verbleibenden Rechenzeit) ist man froh, dass zumindest die Kommentare und Ausgaben der Codeauszüge unübersetzt geblieben sind.*

24.1 Einfache Beispiele

Dieser Abschnitt stellt einfache Beispiele für Multithreading zusammen mit den Voraussetzungen vor. Dabei geht es erst einmal um Thread-Klassen der Kategorie *incommunicado:* Ihre Objekte bekommen eine bestimmte Aufgabe übertragen, erledigen sie, und enden dann. Obwohl derartige Threads nicht nur in C# immer wieder gerne hergenommen werden, um die grundlegenden Mechanismen zu demonstrieren, sollten Sie im Kopf behalten, dass die Anforderungen an real verwendbare Threads üblicherweise wesentlich höher ausfallen – und meist auch anders, weshalb das hier vorgestellte Material in der Praxis in einigen Punkten zu relativieren ist.

24.1.1 Grundvoraussetzungen

Zur Thread-Programmierung empfiehlt es sich, die folgenden Namensräume für den Quelltext zu öffnen:

```
using System.Threading;
using System.Diagnostics;
```

In `Threading` ist unter anderem die Klasse `Thread` definiert; `Diagnostics` stellt die Klasse `Trace` zur Verfügung, die über (ebenfalls statische) Methoden wie `WriteLine()` das Verfolgen von Abläufen auch in Hintergrund-Threads erlaubt.



Während Haltepunkte in Hintergrund-Threads für VS.NET überhaupt kein Problem darstellen, sollte man mit Rückmeldungen über die Windows-Oberfläche via `MessageBox.Show()` *ausgesprochen vorsichtig sein: Sie führen zwar nicht direkt zum Absturz des Programms (wie es beispielsweise bei Delphi der Fall wäre), verzerren aufgrund der automatischen Synchronisierung mit dem Vordergrund aber das Zeitverhalten von Threads derart, dass sie nur in wenigen Fällen brauchbar sind.* `Trace.WriteLine()` *arbeitet dagegen über die API-Routine* `OutputDebugString`*, die auch für Aufrufe aus dem Hintergrund ausgelegt ist.*

Die zweite für die Thread-Programmierung notwendige Zutat ist ein x-beliebiges Objekt mit einer parameterlosen `void`-Methode, die den Einsprungpunkt für neue Threads verkörpert:

```
public class MyThread
{
  public void Execute()
  {
    // ...
  }
}
```

Der Aufruf dieser Methode in einem eigenen Thread sieht dann so aus (hier als Behandlung eines `Click`-Ereignisses formuliert):

```
private void bStartNewThread_Click(object sender, System.EventArgs e)
{
  // ein neues Objekt (thread-individuelle Daten)
  MyThread MyThreadObject = new MyThread();

  // Delegat für den Einsprungpunkt des Threads generieren
  Thread NewThread = new Thread(new ThreadStart(MyThreadObject.Execute));
  // Start des Threads
  NewThread.Start();
}
```

Der Code generiert eine Instanz der eigenen Klasse `MyThread` sowie eine Instanz der .NET-Klasse `Thread`, deren Konstruktor eine Delegatinstanz des .NET-Typs `ThreadStart` für die Startmethode des anzulegenden Threads erwartet. `Start()` ruft diese Methode im Kontext des Threadobjekts `NewThread` auf.



Die deutsche Version der Online-Dokumentation führt in die Irre. Der Konstruktor der Klasse Thread erwartet nicht einen Verweis »auf Methoden, die bei Beginn der Ausführung aufgerufen werden müssen« – etwa im Sinne einer Initialisierung bzw. eines »OnStart«-Rückrufs, sondern eben schlicht den Verweis auf diejenige Methode eines (im übrigen beliebigen) Objekts, die den gesamten Ausführungsrumpf des Threads darstellt.

Im Prinzip war's das bereits: Die über den Delegaten angegebene Methode wird nun in einem eigenen (Hintergrund-)Thread ausgeführt. Wenn Sie mehrere Instanzen eines solchen Threads parallel oder nacheinander ausführen lassen wollen, benötigen Sie für jede dieser Instanzen einen eigenen Delegaten, nicht aber unbedingt auch jeweils ein eigenes Objekt der Klasse `MyThread`. (Da `ThreadStart` eine parameterlose Methode deklariert, wird man allerdings in der Praxis meistens mit individuellen Objekten arbeiten, deren Datenfelder jeweils individuelle Daten speichern.)

Einbetten des Thread-Objekts

Für OOP-Fans mit Hang zum Purismus noch ein Nachsatz. All das lässt sich in eine einzige Klasse verpacken, obwohl `Thread` – weil versiegelt – als Basisklasse nicht zur Verfügung steht.

```
class MyThread
{
  Thread Thread;
  public MyThread()
  {
    this.Thread = new Thread(new ThreadStart(Execute));
    this.Thread.Start();
  }
  void Execute()
  {
    Console.WriteLine("Hi, ich bin der Thread");
  }
}
```

Da der Thread hier bereits bei Konstruktion gestartet wird, reicht also der folgende Ausdruck zum Auslösen von Hintergrundaktivitäten:

```
new MyThread();
```

Leichter geht es nicht mehr. Falls eine Kontrolle des Threads erwünscht ist, wie im folgenden Abschnitt diskutiert, würde man allerdings einen etwas weniger verkapselten Ansatz wählen.

```
class MyThread
{
  public readonly Thread Thread;
  public MyThread()
  {
```

```
    this.Thread = new Thread(new ThreadStart(Execute));
  }
  void Execute()
  {
    Console.WriteLine("Hi, ich bin der Thread");
  }
}
```

Nun hat der Besitzer per Datenfeld Zugriff auf das `Thread`-Objekt (austauschen darf es dieses Objekt allerdings nicht) und kann den Start und die weitere Steuerung selbst übernehmen.

```
MyThread NewObject = new MyThread();
// Initialisierung individueller Datenfelder, soweit vorhanden
// ...
NewObject.Thread.Start();
```

24.1.2 Die Klasse Thread im Überblick

Die wesentlichen Eigenschaften und Methoden der Klasse `Thread` sind schnell aufgezählt. Tabelle 24.1 gibt einen Überblick.

Tabelle 24.1: Die wichtigsten Eigenschaften und Methoden der Klasse `Thread`

Eigenschaft/Methode	Bedeutung
`void Start()`	Startet einen neuen Thread
`void Suspend()`	Hält Thread an
`void Resume()`	Setzt Thread fort
`ThreadPriority Priority {get; set;}`	Legt die Ausführungspriorität des Threads im Verhältnis zu den anderen Threads der Anwendung (nicht: die Priorität der Anwendung im Verhältnis zu anderen Anwendungen) fest.
`ThreadState ThreadState {get;}`	`enum`-Wert, der Auskunft über den aktuellen Status des Threads gibt. Threads werden grundsätzlich im Status `Unstarted` angelegt, der seinerseits `Suspended` (also angehalten) entspricht; sie werden erst durch Aufruf ihrer Methode `Start()` zu Kandidaten bei der Vergabe von Rechenzeit.
`void Join()`	hält den *aufrufenden* Thread (Besitzer des Thread-Objekts) so lange an, bis der Thread beendet ist.

Hier ein Codebeispiel für den Einsatz der Methode `Join()`. Der Vordergrund-Thread spaltet einen Hintergrund-Thread ab und wartet dann später auf dessen Beendigung.

```
// Start des Threads
NewThread.Start();
// Operationen im Vordergrund parallel zum Hintergrund-Thread
// ...
// Vordergrund anhalten, bis Hintergrund-Thread beendet ist
NewThread.Join();
```

24.1.3 Codebeispiel – Threads ohne eigene Klasse

Die allereinfachste Form quasi-paralleler (auf Multiprozessor-Systemen: echt paralleler) Programmierung kommt sogar ohne eine separate Klasse aus. Wie das Codebeispiel *ThreadwithoutClass* demonstriert, hält Sie niemand davon ab, in einer Methode der Klasse `Form1` eine andere Methode *desselben* Objekts in einem separaten Thread auszuführen.

```
int ForegroundCounter, BackgroundCounter;

public void ExecThisInASeparateThread()
{
  // in einem separaten Thread auszuführende Anweisungen
  for (int x = 0; x < 1000; x++)
    BackgroundCounter++;
}

private void button1_Click(object sender, System.EventArgs e)
{
  BackgroundCounter = 0;
  ForegroundCounter = 0;

  Thread NewThread =
    new Thread(new ThreadStart(ExecThisInASeparateThread));
  NewThread.Start();
  // Im Vordergrund auszuführende Anweisungen, parallel zum Hintergrund
  for (int y = 0; y < 1000; y++)
    ForegroundCounter++;
  // Vordergrund ist fertig. Warten auf den Hintergrund-Thread
  NewThread.Join();

  MessageBox.Show(string.Format("FGCounter: {0}, BGCounter: {1}",
    ForegroundCounter, BackgroundCounter));
}
```

In diesem Beispiel geht zwar nichts schief – aber auch nur deshalb, weil Vorder- und Hintergrund-Thread mit verschiedenen Variablen arbeiten und der Vordergrund-Thread erst auf `BackgroundCounter` *zugreift, nachdem er das Ende des Hintergrund-Threads abgewartet hat.*



24.2 Synchronisation für Variablenzugriffe

Zwei Threads gleicher Priorität erhalten insgesamt in etwa auch dieselbe Rechenzeit. Zu welchen Zeitpunkten das System einem einzelnen Thread Rechenzeit zuteilt, und in wie vielen »Portionen«, ist aber lediglich statistisch vorhersagbar.

Bei einfachen Variablen käme man unter Umständen sogar noch mit unsynchronisierten Zugriffen durch: Wenn die im Hintergrund arbeitende Methode aus dem zuvor gezeigten Beispiel ebenfalls `ForegroundCounter` erhöhen würde, also

```
public void ExecThisInASeparateThread()
{
  // in einem separaten Thread ausführende Anweisungen
  for (int x = 0; x < 1000; x++)
    ForegroundCounter++;
}
```

dann hätte dieser Zähler hinterher eben den Stand 2000. Wer etwas Ähnliches mit einer komplexen Variablen probiert (beispielsweise einem String), kann von Glück reden, wenn er damit sofort Schiffbruch erleidet – und sich damit eine längere Fehlersuche erspart. Der Grund dafür liegt darin, dass eine Anweisung wie

```
ForegroundString += "xyz";
```

im Gegensatz zur Erhöhung einer Integervariablen aus mehreren Teilen und einer Vielzahl von Prozessorbefehlen besteht: Der Ermittlung der neuen Stringlänge, dem Belegen eines Bereichs entsprechender Größe auf dem Heap, dem Einschreiben der neuen Daten in diesen Bereich usw. Einer der vielen möglichen Wege in die Katastrophe sieht dann so aus:

1. Vordergrund-Thread: Anhängen von 3 Zeichen an den String, also Ermittlung des neuen Platzbedarfs, Belegen eines Bereichs auf dem Heap – beispielsweise für 6+3 = 9 Zeichen.
2. Die Zeitscheibe für den Vordergrund-Thread ist abgelaufen, das System schaltet auf den Hintergrund-Thread um.
3. Hintergrund: Anhängen von 5 Zeichen an den String, also dieselben Schritte – Belegen eines Bereichs auf dem Heap – beispielsweise mit 6 (oder schon 9?) + 5 = 11 bzw. 14 Zeichen, Einkopieren der Daten.
4. Die Zeitscheibe für den Hintergrund-Thread ist abgelaufen, das System schaltet auf den Vordergrund-Thread um. Wie viele Zeichen der Vordergrund-Thread nun kopiert (9, 11, 14?) – und woher diese stammen (aus dem alten oder neuen Bereich?), ist mehr oder minder zufällig.



Die Umschaltung zwischen Threads geschieht über Interrupts, die die Befehlsausführung des aktuellen Threads unterbrechen (freilich erst nach Abschluss des aktuellen Maschinenbefehls), seinen Kontext (Prozessorregister, Stack) zwischenspeichern, den Kontext des jeweils nächsten Threads wiederherstellen und dessen (zuvor irgendwann einmal auf dieselbe Weise unterbrochene) Befehlsfolge fortsetzen. Das Erhöhen einer Integervariablen geschieht üblicherweise mit einem einzelnen Befehl – und nur deshalb führt das gezeigte Beispiel nicht zwangsläufig zu einem ähnlichen Durcheinander wie das Anhängen von Zeichen an einen String.

Je nach Systemlast gibt Windows einem einzelnen Thread pro Drehung der Zeitscheibe zwischen 20 und 200 Millisekunden Rechenzeit. Diese Werte sind nicht von Microsoft dokumentiert oder gar garantiert, sondern das Ergebnis einer empirischen Messung und sollen ausschließlich eine praktische Vorstellung vermitteln.

24.2.1 lock

Tatsächlich ist die Synchronisation von Variablenzugriffen in C# derart einfach, dass die Vorrede in diesem Abschnitt übermäßig lang erscheinen mag.

```
lock(<Referenztyp>)
{
  // ... Zugriffe
}
```

Die Anweisung `lock`, die intern in zwei Methodenaufrufe der Klasse `System.Threading.Monitor` umgesetzt wird, legt eine CRITICAL SECTION an – ein Objekt, das sich zu jedem Zeitpunkt nur im Besitz *eines* Threads befinden kann. Das Szenario sieht hier so aus:

1. Thread A fordert über `lock` eine Sperre an und erhält den Zuschlag.
2. Während Thread A noch mit Zugriffen beschäftigt ist, läuft seine Zeitscheibe ab. Das System gibt Thread B Rechenzeit.
3. Thread B fordert über `lock` eine Sperre für dasselbe Objekt an – und wird nun so lange angehalten (lies: bekommt seine Rechenzeit entzogen), bis Thread A seine Sperre wieder freigegeben hat.

`lock` sorgt nicht auf irgendeine geheimnisvolle Weise dafür, dass einem Thread der Zugriff auf bestimmte Speicherzellen oder Objekte verwehrt wird. Vielmehr könnte man solche Sperren und andere Synchronisationsobjekte aus der Sicht eines Threads wesentlich besser als Aufrufe von Funktionen beschreiben, deren Ausführung immer genau so lange dauert, bis das Objekt für den aufrufenden Thread zur Verfügung steht und in Besitz genommen werden kann.

Anders gesagt: Wenn Sie in Thread A den Zugriff auf ein Objekt `X` in ein `lock`-Konstrukt einbetten, dann schützen Sie also *diesen Thread* davor, zur Unzeit an X herumzumanipulieren – und nicht etwa die Variable selbst. Woraus auch folgt: Entweder betten Sie in *jedem Thread* Zugriffe auf X in ein `lock` ein oder der Schutz ist wertlos.



Im Zusammenhang mit `lock` *erscheint in der C#-Dokumentation verschiedentlich auch der Begriff Mutex – eine Abkürzung für »mutually exclusive«, was sich gut mit »gegenseitig ausschließend« übersetzen lässt. In der Dokumentation anderer Programmiersprachen wird etwas genauer unterschieden: »Critical Sections« (in C# also:* `lock`*) gelten anwendungsweit, »echte« Mutexe (in C#: Objekte der Klasse* `Mutex`*) lassen sich dagegen auch zur anwendungsübergreifenden Synchronisation verwenden und sind – wen wundert's – mit einem erheblich höheren Verwaltungsüberbau verbunden.*

Das von `lock` erwartete Argument muss entweder einen Referenztyp tragen (also im Heap gespeichert sein), wenn eine Objektinstanz zu schützen ist:

```
// Sperren des zu manipulierenden Objekts
lock(ForegroundString)
{
  ForegroundString = ForegroundString + "xyz";
}
```

oder es muss ein Klassentyp sein, wenn statische Datenfelder einer Klasse zu schützen sind.

```
// Sperren der anynomen Klasseninstanz
lock(typeof(Form1))
{
  // Zugriff auf statische Datenfelder der Klasse Form1
}
```

Ein auf dem Stack liegender Wert eines Werttyps – etwa eine `int`-Variable – ist als Argument also nicht zulässig.

Eine Anwendung kann eine beliebige Anzahl gleichzeitig aktiver Sperren für Objekte und Klassen unterhalten. Das System erlaubt ohne Probleme das mehrfache Anfordern ein und derselben Sperre innerhalb eines Threads:

```
public void ProcA()
{
  lock(ForegroundString)
  {
    // ...
  }
}
```

```
public void ProcB()
{
  lock(ForegroundString)
  {
    ProcA();
    // ...
  }
}
```

Hier steht also nicht zu befürchten, dass sich `ProcB` nach Anfordern des Mutex durch den Aufruf von `ProcA` selbst bis in alle Ewigkeit blockiert: Windows erkennt, dass es hier um ein und denselben Thread geht. (Bei Aufrufen von `ProcA` aus einem anderen Thread heraus ist die Sperre selbstverständlich wirksam.)

24.2.2 Deadlocks

Wenn zwei oder mehr Threads zwei oder mehr Synchronisationsobjekte anfordern, ist bei der Reihenfolge der Anforderungen Sorgfalt geboten. Das Standardbeispiel für zwei Threads, die sich gegenseitig bis zum manuellen Hinauswurf des Prozesses blockieren, sieht so aus:

```
public void ThreadAProc()
{
  lock(RefA)
  {
    // ...
    lock(RefB)
    {
       // ...
    }
  }
}

public void ThreadBProc()
{
  lock(RefB)
  {
    // ...
    lock(RefA)
    {
       // ...
    }
  }
}
```

Wenn Windows hier von Thread A nach Thread B umschaltet, nachdem Thread A die Sperre `RefA` in Besitz genommen hat, dann holt sich Thread B die Sperre `RefB` und wird bei der Anforderung von `RefA` stillgelegt. Weiter geht es mit Thread A, der aber nun seinerseits auf die Freigabe von `RefB` wartet – und das bis in alle Ewigkeit.

Der Ausweg ist allerdings erheblich schneller beschrieben als das Problem selbst: Wenn zwei Threads mehrere Sperren anfordern, dann sollten sie das grundsätzlich in derselben Reihenfolge tun – im gegebenen Beispiel muss Thread B also genauso wie Thread A zuerst `RefA` und erst dann `RefB` anfordern.

24.2.3 Wettrennen (Race)

Der Ausdruck *race condition* steht in diesem Zusammenhang für zwei Threads, die sich buchstäblich ein Rennen liefern, und zwar mit ungewissem Ausgang. Fehler dieser Art sind erheblich schwerer zu entdecken als schlichte Deadlocks. Ein einfaches Beispiel:

```
public void ThreadAProc(int index)
{
  lock(GlobalList)
   {
     GlobalList.Items.RemoveAt(index);
   }
}
public void ThreadBProc()
{
  for (int x = 0; x < GlobalList.Items.Count; x++)
  {
    // ... irgendwelche (zeitaufwändigen) Operationen
    lock(GlobalList)
     {
       GlobalList.Items[x] = ...  // irgendeine Manipulation
     }
  }
}
```

Hier ist also Thread A für das Anhängen von Elementen an eine Liste zuständig, Thread B für das Bearbeiten der Listenelemente – und nachdem beide Threads ihre Zugriffe auf die Liste per `lock` synchronisieren, sollte das eigentlich auch funktionieren.

In der Praxis tut es das leider nicht immer, weil die Klammerung mit `lock` in `ThreadBProc` zu eng angesetzt ist: Zwischen der Prüfung `x < GlobalList.Count` und dem Zugriff auf Element `x` bleibt ein Zeitfenster offen, in dem Thread A Listenelemente löschen und so dafür sorgen kann, dass die darauffolgende Indizierung in Thread B ins Leere geht oder dasselbe Element zweimal bearbeitet. Sicher wäre daher:

```
public void ThreadBProc()
{
  lock(GlobalList)
  {
    for (int x = 0; x < GlobalList.Items.Count; x++)
    {
```

```
      // ... irgendwelche (zeitaufwändigen) Operationen
      GlobalList.Items[x] = ...  // irgendeine Manipulation
    }
  }
}
```

Besser wäre es natürlich so, wenn die Programmlogik es zulässt:

```
public void ThreadBProc()
{
  // ... irgendwelche (zeitaufwändigen) Operationen

  lock(GlobalList)
  {
    for (int x = 0; x < GlobalList.Items.Count; x++)
    {
      GlobalList.Items[x] = ...  // irgendeine Manipulation
    }
  }
}
```

24.3 Steuerelemente

Während die Routinen der Windows-API und sämtliche Windows-Steuerelemente wie Listenfelder, Treeviews usw. natürlich »threadfest« sind, sich also ohne jede weitere Vorsorge auch quasi-parallel von verschiedenen Threads aus aufrufen lassen, können die .NET-Hüllklassen für Steuerelemente diese Eigenschaft nicht für sich in Anspruch nehmen. Eine Anweisung wie

```
MyListBox.Items.Add("xyz");
```

läuft, was das Windows-Steuerelement betrifft, letztlich an irgendeiner Stelle auf den Aufruf der API-Funktion `SendMessage` an die Windows-Listbox hinaus, und bei dieser Routine sorgt Windows von selbst dafür, dass sie ausschließlich im Kontext eines einzigen, systemeigenen (Vordergrund-) Threads ausgeführt wird. Die C#-Hüllklasse `ListBox` führt vor allem aus Gründen der Performance eine eigene Stringliste des Typs `ObjectCollection`. Bei unsynchronisierten Zugriffen auf diese Liste ergibt sich zwangsläufig dasselbe Durcheinander wie bei anderen C#-Sammelklassen. Die Basisklasse `Control` definiert deshalb eine Eigenschaft `InvokeRequired` mit dem Typ `bool`. Wenn ihre Abfrage `true` ergibt, dann ist der abfragende Thread nicht mit dem Thread identisch, der das Steuerelement (und die Hüllklasse) angelegt hat, weshalb eine explizite Synchronisation vor dem Zugriff notwendig wird. Derartige Synchronisationen sind das Thema des nächsten Abschnitts.

24.4 Synchrone und asynchrone Rückrufe

Wenn man den üblichen Beispielen zur Thread-Programmierung folgt, dann sieht »typische« Hintergrund-Thread in einer real existierenden Anwendung folgendermaßen aus:

```
public void Execute()
{
  // Rückmeldung: Gestartet
  ReportStatus(0, string.Format("Thread {0} gestartet", ThreadNumber));
  // Arbeitsteil
  for (int x = 0; x < 5; x++)
  {
    ReportStatus(1, string.Format("Thread {0}: Iteration {1} von 5",
      ThreadNumber, x+1));
    Thread.Sleep(2000);
  }
  // Rückmeldung: Ende
  ReportStatus(2, string.Format("Thread {0} beendet", ThreadNumber));
}
```

Die Rückmeldung über den Start des Threads ist optional, weil der Besitzer eines Threads im Allgemeinen schließlich selbst »weiß«, wann er einen neuen Thread anlegt. Die Rückmeldung über das Ende des Threads kann bzw. muss entfallen, wenn die Anwendung an irgendeiner Stelle mit `Join()` auf das Ende des Threads wartet. (Ansonsten ergäbe sich sofort ein Deadlock: Der Vordergrund-Thread bleibt bis zum Ende des Hintergrund-Threads blockiert, während der Hintergrund-Thread kurz vor seinem Ende noch einmal den Vordergrund-Thread aufrufen will. Mehr dazu in einem der folgenden Abschnitte.)

Die eigentliche Arbeit eines Threads wird üblicherweise in einer Schleife erledigt, die je nach zu lösender Aufgabe eine fixe Zahl von Iterationen oder ein Abbruchkriterium im Sinne von `while (!Abbruchflag)` vorsieht – und eben eine Rückmeldung an den Besitzer (im Allgemeinen: der Vordergrund-Thread). Beispielsweise:

```
while (!Terminated)
{
  DoSomeThingorWaitForSomething ();
  ReportAndStoreNewData();
}
```

So in etwa könnte der Rumpf eines Threads aussehen, der auf das nächste Datenpaket, den nächsten Tastendruck, das Ende einer Teilberechnung usw. wartet (bzw. diese Operation selbst ausführt), die neuen Daten in irgendeine Liste einfügt – und dann dem Vordergrund-Thread per Rückruf mitteilt, dass ein Teilschritt erledigt ist, eine neue Mail zur Verfügung steht, eine Taste gedrückt wurde etc.

24.4.1 Delegaten

Wie nicht anders zu erwarten, kommen bei derartigen Rückrufen Delegaten zum Einsatz. Im einfachsten Fall:

```
public class MyThread
{
  // Vereinbarung des Delegattyps
  public delegate void MainCBDelegate (int Status, string Msg);
  // Vereinbarung einer Delegatinstanz
  public MainCBDelegate ReportStatus;
  ...
}

public class Form1 : System.Windows.Forms.Form
{ ...

  public void StatusFromThread(int Status, string Msg)
  {
    // ... Auswertung von Status und Msg
  }

  private void StartNewThread()
  {
    // ein neues Objekt (thread-individuelle Daten)
    MyThread NewObject = new MyThread();

    // Delegat für die Rückruf-Methode
    NewObject.ReportStatus =
       new MyThreadObject.MainCBDelegate (StatusFromThread);

    // Delegat für die auszuführende Methode, Thread mit diesem Delegaten
    Thread NewThread = new Thread(new ThreadStart(NewObject.Execute));
    // Start des Threads
    NewThread.Start();
  }
}
```

Das Anlegen des Delegaten für den Rückruf und das Registrieren des Rückrufs im Feld `ReportStatus` des Objekts geschieht also noch im Vordergrund-Thread und unterscheidet sich in keiner Weise von dem Prozedere, das für die Ereignisbehandlung notwendig ist: Anstelle einer direkten Zuweisung ließe sich deshalb auch der Operator += verwenden.

Der Grundsatz des business as usual gilt allerdings nicht nur für das Anlegen und Einsetzen, sondern auch – wen wundert's – für Rückrufe über den Delegaten: Sie finden wie ganz normale Aufrufe über Funktionszeiger (Verzeihung, Delegaten) statt – von einem Hintergrund-Thread aus also im Kontext dieses Hintergrund-Threads und damit *unsychronisiert*.

24.4.2 Synchronisation über Steuerelemente

Delphi-Programmierer, die in der Dokumentation zu C# bereits vergeblich nach einem Äquivalent zur Methode *Synchronize* der VCL-Klasse *TThread* gesucht haben, werden wohl beim Anblick dieser Überschrift genauso erleichtert aufatmen wie COM-Programmierer, die zum Thema »Serialisierung« lediglich Anmerkungen zum Speichern von Ressourcen gefunden haben: Ganz recht, es gibt auch in C# die Möglichkeit, die Befehlsausführung in den Kontext des Vordergrund-Threads zu verlagern. Der folgende Codeauszug zeigt, wie:

```
public class MyThread
{
  public Control SyncControl;
  // ...
  // eine Instanz dieses Delegatentyps
  public StatusToMainDelegate CallMainMsgStatus;

  private void ReportStatus(int Status, string Msg)
  {

    // Parameter als object-Array verpacken (boxing)
    object[] P = new object[] { Status, Msg };

    // synchron zum Haupt-Thread über ein Control des Haupt-Threads
    SyncControl.Invoke(CallMainMsgStatus, P);
  }
}
```

Der einzige zusätzliche Schritt, der beim Anlegen eines solchen Threads notwendig wird, ist das Binden einer beliebigen Steuerelementinstanz an das Feld `SyncControl` des `MyThread`-Objekts.

```
private System.Windows.Forms.ListBox boxStatus;
...
private void bStartNewThread_Click(object sender, System.EventArgs e)
{
  // ein neues Objekt (thread-individuelle Daten)
  MyThread NewObject = new MyThread ();
  NewObject.ThreadNumber = ++GlobalThreadNumber;

  // Delegat für die Rückruf-Methode
  NewObject.CallMainMsgStatus = new
    MyThread.StatusToMainDelegate(CBFromThread);
  NewObject.SyncControl = boxStatus;

  // Delegat für die auszuführende Methode, Thread mit diesem Delegaten
  Thread NewThread = new Thread(new ThreadStart(NewObject.Execute));
  // Start des Threads
  NewThread.Start();
}
```

Wie die hier eingesetzte Rückrufmethode `CBFromThread` aussieht, dazu mehr im nächsten Abschnitt.

Wer die Analogie noch ein wenig weiter treiben will, definiert eine Hüllklasse mit einem Konstruktor, der auf einem `Control`-Objekt des Vordergrund-Threads besteht – und schon sind die gewohnten Verhältnisse wieder hergestellt: Wie sich in der C#-Dokumentation nachlesen lässt, erwartet die Methode `Control.Invoke()` einen Delegaten (in einer überladenen Variante zusätzlich ein `object`-Array für Parameter) und sorgt dafür, »dass dieser Delegat in dem Thread ausgeführt wird, in dem das Steuerelement angelegt wurde« – gemeint ist natürlich: »im Kontext des Threads«.

Die Frage ist allerdings, ob man – außer für Ports – auch in einer neuen Sprache um jeden Preis zu Bekanntem zurückkehren will. Und als zweites sollte stutzig machen, dass `Invoke()` ausschließlich als Methode von `Control` und davon abgeleiteten Klassen zu finden ist.

24.4.3 Synchronisation auf neuen Wegen

Tatsächlich haben die Microsoft-Entwickler hier ein weiteres Mal die Gunst der neuen Sprache genutzt, um alte Zöpfe abzuschneiden: Eine »harte« Synchronisation mit dem »GUI-Thread«, also dem Vordergrund-Thread einer Anwendung, ist nämlich ausschließlich für Steuerelemente notwendig – und sonst nirgends. Jede andere Art von Thread-Synchronisation findet in C# ausschließlich über die Synchronisationsobjekte statt, die das System für solche Fälle zu bieten hat – und nicht über den beliebten (durchaus aber auch mal verfluchten) Trick, dass die für das Fenstersystem von Windows spezifische API-Funktion `SendMessage` sozusagen automatisch im Vordergrund-Thread läuft.

Was bei dieser Neuerung ebenfalls herausgekommen ist, mag von gestandenen ProgrammierInnen sehr wohl als ein Aufzäumen des Pferdes von hinten gesehen werden (wobei man sich aber gleich darüber im Klaren sein sollte, dass hier eine Menge Gehirnschmalz investiert wurde): Nicht der zurückrufende Thread hat die Kontrolle über die Synchronisation, sondern die dabei aufgerufene Routine des Besitzers. Die Routine eines Threads, die den Benutzer – beispielsweise per Listenfeld oder Label – über seinen aktuellen Status informiert, könnte beispielsweise ohne weiteres so aussehen:

```
// Aufruf aus einem Hintergrund-Thread heraus
private void ReportStatus(int Status, string Msg)
{
  CallMainMsgStatus(Status, Msg);
}
```

Wo diese Information landet, und ob eine Synchronisierung notwendig ist, wird nicht im Code des Threads, sondern gegebenenfalls erst in der über

den Delegaten aufgerufenen Methode entschieden. Ein Codebeispiel für eine solche – zugegeben etwas unübersichtliche – »nachträgliche« Synchronisierung findet sich auf der Begleit-CD unter dem Namen *ThreadSyncInCB*.

```
// Thread-Rückruf (callback)
public void CBFromThread(int Status, string Msg)
{
  if (boxStatus.InvokeRequired)
  {  // Aufruf aus einem anderen Thread heraus
    CBFromThreadDelegate d = new CBFromThreadDelegate(CBFromThread);
    // Parameter als object-Array
    object[] P = new object[] {Status, Msg};
    // Synchronisation mit dem Vordergrund-Thread durch rekursiven Aufruf
    boxStatus.Invoke(d, P);
  }
  else
  { // Aufruf aus dem Vordergrund-Thread (oder nachsynchronisiert)
    boxStatus.Items.Add(Msg);
    boxStatus.SelectedIndex = boxStatus.Items.Count-1;
    switch(Status)
    // ...
```

24.4.4 Asynchron ausgeführte Methoden

Das zuvor erwähnte Aufzäumen des Pferdes von hinten geht bei C# allerdings noch ein erhebliches Stück weiter. Tatsächlich hält Sie niemand davon ab, eine *beliebige* Methode zu einem – ebenfalls beliebigen – Zeitpunkt in einem separaten Thread auszuführen, ohne dafür extra eine eigene Klasse anzulegen.

Möglich wird das über zwei Methoden namens `BeginInvoke()` und `EndInvoke()`, die bei jedem Delegaten mit von der Partie sind. Ein Beispiel dazu, das sich auf der Begleit-CD unter dem Namen *ThreadIAsyncResult* wiederfindet:

```
public delegate int ExecInBGDelegate(int Start, int Stop);

public int ExecInBG(int Start, int Stop)
{
  for (int x = Start; x < Stop; x++)
  {
    Thread.Sleep(200);
  }
  return Stop;
}
private void button1_Click(object sender, System.EventArgs e)
{
  ExecInBGDelegate d = new ExecInBGDelegate(ExecInBG);

  IAsyncResult ar = d.BeginInvoke(1, 20, null, null);
  boxStatus.Items.Add("Thread gestartet");
```

```
  // Statusabfrage (optional)
  while(!ar.IsCompleted)
  {
    boxStatus.Items.Add("Warten auf den Thread");
    boxStatus.Update();
    Thread.Sleep(100);
  }

  // Aufruf beenden (blockiert den Aufrufer)
  int Res = d.EndInvoke(ar);
  boxStatus.Items.Add(String.Format("Thread beendet. Ergebnis: {0}", Res));
}
```

Ob es sich bei diesen beiden Methoden von Delegaten um einen Nachgedanken gehandelt hat, ist unklar – jedenfalls fehlen sie sowohl in der kontextbezogenen Syntaxergänzung als auch in der Online-Dokumentation, weshalb hier eine referenzartige Beschreibung folgt.

BeginInvoke, EndInvoke und IAsyncResult

Die Methode `BeginInvoke()` eines Delegaten führt die Methode, auf die der Delegat verweist, in einem eigenen, neu angelegten Thread – und somit asynchron – aus. Die Syntax lautet:

```
IAsyncResult BeginInvoke([DelegateParam,...], Delegate AsyncDelegate,
  object AsyncDelegatParam);
```

Die *DelegateParam*-Parameter müssen der Signatur des Delegaten entsprechen und werden hier ausnahmsweise nicht über ein `object`-Array, sondern einzeln angegeben.

`AsyncDelegat` ist ein weiterer Delegat, der beim Ende des Threads aufgerufen wird, um den Besitzer über das Ende des Threads zu informieren. Allerdings ist hier auch die Übergabe von `null` zulässig.

`object` ist ein beliebiges Objekt, das der durch `AsyncDelegat` bezeichneten Routine als Parameter übergeben wird. Hier ist ebenfalls der Wert `null` zulässig.

`BeginInvoke()` kehrt sofort zum Aufrufer zurück. Das direkte Funktionsergebnis ist eine Instanz einer `IAsyncResult`-Schnittstelle, über deren Methoden unter anderem der Status abgefragt oder – mit `AsyncWaitHandle.WaitOne()` – auf das Ende des Threads gewartet werden kann. (Diese Schnittstelle ist im Gegensatz zu `BeginInvoke()` vollständig dokumentiert.)

Die Methode `EndInvoke()` erwartet die von `BeginInvoke()` gelieferte `IAsyncResult`-Schnittstelleninstanz. Sie blockiert den Aufrufer gegebenenfalls so lange, bis der Thread beendet ist (vgl. `Join()`), und liefert ein der Signatur des Delegaten entsprechendes Ergebnis – das Funktionsergebnis der asynchron ausgeführten Methode.

24.5 Interaktion mit Threads

Um noch einmal auf den Anfang dieses Kapitels zurückzukommen: Die meisten Beispiele zur Thread-Programmierung gehen von Threads aus, die nach dem Start bestenfalls ab und an eine Rückmeldung von sich geben, und irgendwann nach getaner Arbeit enden.

In realen Problemstellungen geht es aber leider meist nicht um schlichte Datenlieferanten, sondern um Interaktionen. Ein zumindest vom Prinzip her einfaches Beispiel dafür ist eine Anwendung, die sprachgesteuert ihre Befehle entgegennimmt, wobei die Spracherkennung in einem Hintergrund-Thread ihren Dienst tut.

Die möglichen Reaktionen auf einen erkannten Befehl lassen sich in drei Kategorien aufteilen:

1. Reguläre Befehle, die sich nach dem Standardschema abarbeiten lassen: Auf einen Befehl wie – beispielsweise – »Linie zeichnen« führt die Anwendung die gewünschte Operation aus, danach geht es mit der Spracherkennung für den nächsten Befehl weiter.
2. Befehle, die eine Reprogrammierung des Erkenner-Threads erfordern – beispielsweise das Öffnen eines bestimmten Menüs, verbunden mit dem Wechsel der Vokabulars.
3. Den Befehl »Programmende«.

Das Entscheidende dabei ist, dass alle drei Kategorien zwangsläufig in einer Rückrufroutine des Threads abgearbeitet werden müssen. Befehle der Kategorie 1 sind dabei problemlos – sie entsprechen im Wesentlichen dem Verhalten eines Threads, der in unregelmäßiger Folge Daten liefert und dann unverändert weiterläuft.

Bei Befehlen der zweiten Kategorie sieht es schon schwieriger aus, weil bei einem Schema wie

```
private int CBFromRecognizer(int CmdKind, string Cmd)
{ switch(CmdKind)
  case 1: // Kategorie 1
    ExecuteCommand(Cmd);
    break;
  case 2: // Kategorie 2
    TerminateRecognizer();
    WaitForRecognizerEnd();  // <- Deadlock!
    CreateThreadWithNewVocabulary(Cmd);
    // ...
```

bereits auf den ersten Blick zu erkennen sein sollte, dass das Programm bis in alle Ewigkeit auf das Ende des Threads wartet (der ja seinerseits in der Rückrufroutine festsitzt).

Der Befehl »Programmende« (Kategorie 3) verlangt deshalb eine Sonderbehandlung, weil dort aus dem Rückruf heraus nicht nur der aufrufende Thread beendet werden muss, sondern eben auch das gesamte Programm (oder das jeweilige Formular). Wenn dabei Ressourcen freigegeben werden sollen, die der Thread für seine Arbeit benötigt, muss das Programm in irgendeiner Weise auf das Ende des Threads warten.

24.5.1 Die Beispielklasse Recognizer

Ein denkbares Lösungsschema für diese Problemstellung basiert auf einer Klasse (hier: `Recognizer`) mit den folgenden Eigenschaften:

- Einer Methode (im gegebenen Beispiel: `Execute()`), die eine geschlossene Schleife mit einem Abbruch-Flag implementiert und beim Vorhandensein neuer Daten (hier: »erkannter Befehl«) einen unsychronisierten Rückruf ausführt. Im Beispiel bestehen diese erkannten Befehle aus schlichten, bei der Initialisierung des Programms vorgegebenen Integer- und Stringlisten:

```
while (!Terminated)
{
   if (x >= Data.WordCount) x = 0;
   // "Befehl erkannt", Rückruf (unsynchronisiert)
    OnRecognition(Data.Category(x), Data.Word(x));
    Thread.Sleep(300);
    x++;
}
```

- Diese Routine wird beim ersten Start des Recognizers über `BeginInvoke()` aufgerufen, also in einen separaten Thread verpackt. Das von diesem Aufruf gelieferte `IAsyncResult`-Schnittstellenobjekt steht der Anwendung über ein Datenfeld des Formularobjekts zur Verfügung.

- Die Kategorie 1 – das Erkennen eines »regulären Befehls« – ist schnell abgehandelt:

```
private void OnRecognitionCB(int Category, string Cmd)
{
   switch(Category)
     case 1:
       Trace.WriteLine("CB: Befehl: "+Cmd);
       break; // Weiter im existierenden Thread
```

- Die Anwendung erzeugt nicht nur beim Start, sondern auch bei jedem Parameterwechsel sowohl ein neues `Recognizer`-Objekt als auch einen neuen Thread. Der laufende Thread wird in der (von ihm selbst aufgerufenen) Rückrufroutine beendet.

```
     case 2:
       Trace.WriteLine("CB Neue Parameter: "+Cmd);
       TheRecognizer.Terminated = true;
```

```
      switch (Cmd)
      {
        case "Vokabular 2":
          NewRecognizer(RecogDataSet2);
          break;
        case "Vokabular 3":
          NewRecognizer(RecogDataSet3);
          break;
      }
      break; // neue Thread läuft an, alter Thread läuft aus
```

Einer der wesentlichen Punkte dabei ist, dass ein solcher Thread ausläuft (und danach mangels weiterer Referenzen von der Garbage Collection entsorgt wird), ohne dass dabei synchronisierte Rückrufe mit im Spiel wären (für Delphi-Programmierer: »OnTerminate«). Ansonsten entstünde, wie zuvor erläutert, ein Deadlock, weil das Programm innerhalb einer Rückrufroutine des Threads auf das Ende eben dieses Threads wartet.

Die Kategorie 3, nämlich der erkannte Befehl »Programmende«, wirft dagegen unerwartete Probleme auf.

```
    case 3:
      Trace.WriteLine("CB - Programmende");
      TheRecognizer.Terminated = true;
      bStartRecog.Enabled = true;
      if (bCloseOnTermination.Checked)
#if DEADLOCK
       Close();   // Denkfehler!
#else
       CloseRequestFromThread();
#endif
       break;  // Programmende
     }
   }
```

Der direkte Aufruf von `Close()` führt hier zu einem Deadlock. Der Grund dafür ist etwas verzwickter und erst nachvollziehbar, wenn man sich die Behandlungsmethode `Form1_Closing()` ansieht.

```
private void Form1_Closing(object sender, CancelEventArgs e)
{
  if (RecognizerStatus != null && !RecognizerStatus.IsCompleted)
  {  // Programmende "von außen", d.h. nicht durch
     // einen Recognizer-Befehl
     Trace.WriteLine("FormClose: Warten auf Recognizer-Ende");
     TheRecognizer.Terminated = true;
     RecognizerStatus.AsyncWaitHandle.WaitOne();
     RecognizerDelegate.EndInvoke(RecognizerStatus);
     Trace.WriteLine("FormClose: Recognizer-Ende erreicht");
  }
  // gemeinsam genutzte Ressourcen aller Recognizer-Instanzen freigeben
  FreeGlobalRecognizerResources();
  //
}
```

Wenn das Programm nicht durch einen erkannten Befehl des Recognizers, sondern auf andere Art und Weise – beispielsweise über einen Menübefehl – beendet wird, muss es seinerseits aktiv für das Beenden des Recognizers sorgen, bevor es globale Ressourcen freigibt, auf die alle Instanzen des Recognizers gemeinsam aufbauen.

Der springende Punkt: Wie sich beim schlichten Experimentieren herausstellte, verwendet die Methode `System.Windows.Forms.Form.Close()` zum Weiterreichen der erforderlichen Windows-Nachricht (WM_CLOSE) ausschließlich direkte Aufrufe bzw. die API-Funktion `SendMessage` – was bedeutet, dass der Weg bis zum Stillstand beim direkten Aufruf dieser Methode von der Rückruffunktion aus so aussehen würde:

1. Rückruf des aktiven Threads mit dem Befehl »Programmende«.
2. Aufruf von `Form.Close` aus der Rückrufroutine heraus.
3. `Form.Close` versendet die Windows-Nachricht WM_CLOSE per `SendMessage`, wartet also auf das Beenden dieser Aktion.
4. Die sich ergebenden Kaskade von Windows-Nachrichten führt irgendwann zum Aufruf von `Form1_Closing`.

In `Form1_Closing` wartet die Anwendung nun auf das Ende des Recognizer-Threads – und tut das bis zum St.-Nimmerleinstag, weil der Rückruf des aktiven Threads nach wie vor nicht beendet ist.

24.5.2 Entkoppelung

Die Lösung dieses Problems hört auf den Namen »Entkoppelung«. Irgendwie muss dafür gesorgt werden, dass der Prozessor in jedem Fall wieder aus der Rückrufroutine heraus ist, bevor das Programm auf das Thread-Ende wartet. Dafür bieten sich unter anderem drei Möglichkeiten an, von denen die erste den klassischen Ansatz darstellt:

- `PostMessage`: In der Rückrufroutine wird eine selbstdefinierte Nachricht (oder gleich ein WM_CLOSE) in die Warteschlange der Anwendung eingereiht – verbunden mit der guten Hoffnung, dass Windows nicht noch vor dem Beenden des Rückrufs auf die Idee kommt, zum Vordergrund-Thread umzuschalten. So begründet diese Hoffnung aufgrund des äußerst kleinen Zeitfensters auch ist: Wirklich sauber wird sie erst, wenn man die Priorität des Hintergrund-Threads vorher heraufsetzt und so dafür sorgt, dass die Nachricht unter allen Umständen erst nach Ende des Rückrufs bearbeitet wird. Alles in allem fällt dieser Ansatz in die Kategorie »Bastelei am System«.
- Entkoppelung per Timer. Das Programm definiert einen Timer, der erst in der Rückrufroutine aktiviert wird und ordnet ihm eine Methode

`Form1_FormCloseTimerElapsed()` zu, die ihrerseits `Close()` aufruft. Als Timer-Intervall bieten sich 100 Millisekunden an, wobei auch hier wieder die (wenn auch berechtigte) Hoffnung mit von der Partie ist, dass das reichen möge.

- Entkoppelung über einen separaten Thread. Das mag überkompliziert erscheinen, stellt aber einen Weg dar, der definitiv kein Zeitfenster für Probleme offen lässt. Dieser Thread wird in der Rückruffunktion des Recognizers angelegt, wartet seinerseits auf das Ende des Recognizer-Threads, und ruft dann `Close()` auf. Die Prüfung innerhalb von `Form1_Closing()`, ob der Recognizer-Thread beendet wurde, ergibt in diesem Fall grundsätzlich `true`.

24.6 Codebeispiel – interaktive Threads

Dieser Abschnitt listet das zuvor besprochene Programm *InteractiveThread* auf und liefert einige weitere Details dazu. Abbildung 24.1 zeigt den Formularentwurf.

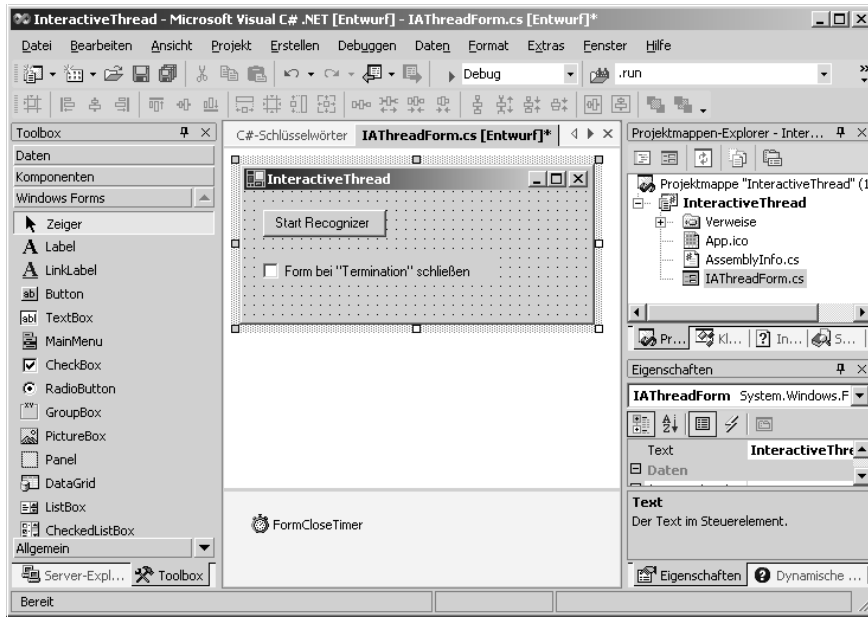


Abbildung 24.1: Der Formularentwurf im Designer

Die Schaltfläche trägt den Namen `bStartRecognizer`, bei Klicks darauf wird die Methode `bStartRecogClick()` des Formularobjekts aufgerufen. Die Checkbox hört auf den Namen `cCloseOnTermination` und hat keine verbundenen Ereignisse; für `FormCloseTimer` wurde die Eigenschaft `Interval` auf 100 msec und die Eigenschaft `Enabled` auf `false` gesetzt. Für Timer-Ereignisse ist die Methode `FormCloseTimer_Elapsed` zuständig.

Ausgaben des Programms – von der »Spracherkennung« gelieferte Befehle sowie Statusmeldungen der verschiedenen Threads – geschehen über die `Trace`-Klasse. Während des Programmlaufs sollte also das Ausgabefenster des Debuggers geöffnet sein.

```
// Listing: InteractiveThreads - IAThreadForm.cs
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Threading;
using System.Diagnostics;
```

Der Namensraum `System.Threading` enthält Methoden wie `Thread.Sleep()` und `System.Diagnostics` definiert die Klasse `Trace`, deren statische Methode `WriteLine()` für die Ausgabe der Statusmeldungen verwendet wird. Diese beiden `using`-Anweisungen müssen Sie selbst hinzufügen.

```
namespace InteractiveThread
{
  public class RecogData
  {
    public readonly int WordCount;
    private int[] categories;
    private string[] words;
    public int Category(int index)
    {
      if (index > WordCount) index = WordCount-1;
      return categories[index];
    }

    public string Word(int index)
    {
      if (index > WordCount) index = WordCount-1;
      return words[index];
    }
```

`RecogData` ist eine reichlich einfache Klasse zur Repräsentation des aktuellen Recognizer-Vokabulars. Der Recognizer liefert für jede »Erkennung« sowohl eine Kategorie (mögliche Werte: 1 bis 3) als auch das erkannte Wort als String.

```
  public class Recognizer
  {
    public bool Terminated;  // Abbruch für Schleife in Execute
    public int ThreadNumber;    // einfacher Zähler: ID für Trace
    public object Soundcard;    // = Referenz auf GlobalSoundcard
    private RecogData Data;     // Thread-inidividuelle Parameter
    // Rückruf bei "Erkennung" eines Befehls
    public delegate void OnRecognitionDelegate(int Category, string Word);
    public OnRecognitionDelegate OnRecognition;
```

Die Feldvariable `Terminated` der Klasse `Recognizer` stellt das Abbruchkriterium für die im Thread ausgeführte Schleife dar. Bei `ThreadNumber` handelt es sich um die Kopie eines globalen Zählers, der beim Anlegen jedes neuen Recognizer-Objekts um Eins heraufgesetzt wird und mit in den Statusausgaben erscheint. `Soundcard` ist eine Referenz auf ein globales `object` und wird zur Synchronisierung der Zugriffe auf globale Ressourcen (für einen Recognizer eben beispielsweise eine Soundkarte) gebraucht – mehr dazu bei der Besprechung von `Execute()` auf den folgenden Seiten. `OnRecognition` ist schließlich der Delegat für den unsynchronisierten Rückruf nach der Erkennung eines Befehls und wird beim Anlegen eines Recognizer-Objekts auf die Formularmethode `OnRecognitionCB()` gesetzt, die die Auswertung übernimmt.

```
public Recognizer(RecogData NewData)
{
  Data = NewData;
}
```

Der Konstruktor übernimmt die für den jeweiligen Thread spezifischen Daten. Im Prinzip ist eine derartige Trennung unnötig: Man hätte genauso gut entweder auch noch einzelne Felder für das Vokabular in der Klasse `Recognizer` vorsehen – oder sämtliche Daten inklusive `Terminated`, `Soundcard` usw. in der Klasse `RecogData` unterbringen und ein entsprechendes Objekt beim Start des Threads übergeben können. Hier führen recht viele Wege nach Rom.

```
// asynchron, per BeginInvoke aus dem Hauptprogramm / CB aufgerufen
public void Execute()
{ int x = 0;

  Trace.WriteLine(String.Format("{0}: Warten auf Soundcard-Lock",
    ThreadNumber));
  lock(Soundcard)
  {   Trace.WriteLine(String.Format("{0}: Soundcard-Lock erhalten",
        ThreadNumber));
      while (!Terminated)
      {
        if (x >= Data.WordCount) x = 0;
        // "Befehl erkannt", Rückruf (unsynchronisiert)
        OnRecognition(Data.Category(x), Data.Word(x));
        Thread.Sleep(300);
        x++;
      }
  }
  Trace.WriteLine(String.Format("{0}: Soundcard-Lock freigegeben",
    ThreadNumber));
}
}
```

Die Recognizer-Methode `Execute()` wird über `BeginInvoke()` aufgerufen und in einem eigenen Thread ausgeführt. Das einzige noch nicht besprochene Detail ist die Sperre über `Soundcard`, die in dieser Demonstration für die Synchronisation des Zugriffs auf globale Ressourcen steht. Das Szenario sieht hier folgendermaßen aus:

1. Thread 1 macht sich an der Ressource (hier: der Soundkarte) zu schaffen, nachdem er sie für andere Instanzen seiner selbst gesperrt hat.
2. Thread 1 erkennt einen Befehl zum Parameterwechsel und führt einen Rückruf aus, in dessen Verlauf ein Thread 2 angelegt und bei Thread 1 das Flag `Terminated` gesetzt wird.
3. Thread 2 startet und wird beim Abruf von `Soundcard` erst einmal so lange still gelegt, bis Thread 1 diese Ressource wieder freigegeben hat, d.h. endgültig ausgelaufen ist.

Ohne diese Schutzmaßnahme wäre die Abfolge der Zugriffe auf die globale Ressource undefiniert: Es bestünde ohne weiteres die Möglichkeit, dass Thread 2 bereits zu Zugriffen auf die Ressource kommt, während Thread 1 noch daran arbeitet.

```
public class IAThreadForm : System.Windows.Forms.Form
{ // per Designer erzeugte Komponenten
  private System.ComponentModel.Container components = null;
  private System.Windows.Forms.Button bStartRecog;
  private System.Windows.Forms.CheckBox cCloseOnTermination;
  private System.Timers.Timer FormCloseTimer;

  // vorgegebene Befehlslisten zur "Erkennung"
  private RecogData[] RecogDataSets;
  private object GlobalSoundcard; // Soundcard-Synchronisation
  private int GlobalThreadNumber = 0; // Thread-ID für Trace
```

Das Array `RecogDataSets` enthält sozusagen das Gesamtvokabular des Recognizers und wird in einer Methode namens `AllocateGlobalRecognizerResources()` besetzt, welche hier stellvertretend für die Initialisierung von Ressourcen steht, die alle Instanzen des Recognizers gemeinsam benötigen. In dieser Methode findet auch die Initialisierung von `GlobalSoundcard` statt.

```
  private Recognizer TheRecognizer; // aktuelles Recognizer-Objekt

  // Delegat für Recognizer.Execute in einem eigenen Thread
  private delegate void RecogExecuteDelegate();
  RecogExecuteDelegate RecognizerDelegate;
  IAsyncResult RecognizerStatus;  // Ergebnis von BeginInvoke
```

`TheRecognizer` wird beim Anlegen eines neuen Recognizer-Objekts neu besetzt (was den Nebeneffekt hat, dass dabei auch die letzte Referenz auf das Vorgänger-Objekt verschwindet und sich die Garbage Collection seiner annehmen kann). `RecognizerStatus` ist das Funktionsergebnis von `BeginInvoke()`; die Speicherung in einem Feld des Formularobjekts erlaubt Statusabfragen des Threads zu jedem Zeitpunkt. Diese Variable wird ebenfalls beim Anlegen überschrieben (d.h. die Referenz auf den jeweiligen Vorgänger gelöscht).

```
private void AllocateGlobalRecognizerResources()
{ // Initialisieren der drei Befehlslisten
  RecogDataSets = new RecogData[]
   {
      new RecogData(3,
       new int[] {1,1,2},
       new string[] { "Wort 1.1", "Wort 1.2", "Vokabular 2"}),
     new RecogData(3,
       new int[] {1,1,2},
       new string[] { "Wort 2.1", "Wort 2.1", "Vokabular 3"}),
     new RecogData(3,
       new int[] {1,1,3},
       new string[] { "Wort 2.1", "Wort 2.1", "Programmende"})
   };
  // Synchronisation für die Soundkarte
  GlobalSoundcard = new object();
}
```

`AllocateGlobalRecognizerResources()` ist für das Belegen der Ressourcen zuständig, die allen Instanzen des Recognizers gemeinsam zur Verfügung stehen.

```
private void FreeGlobalRecognizerResources()
{
  RecogDataSets = null;
  GlobalSoundcard = null;
}
```

Das Gegenstück zu `AllocateGlobalRecognizerResources()` gibt die gemeinsam genutzten Ressourcen wieder frei und darf erst aufgerufen werden, wenn auch die letzte Instanz des Threads ausgelaufen ist. Im gegebenen Beispiel könnte man die Freigabe natürlich dem Garbage Collector überlassen, weshalb diese Routine in erster Linie den Stellvertreter für »richtige« Ressourcen (wie offene Dateien oder Mikrofonkanäle) spielt.

```
public IAThreadForm()
{
  InitializeComponent();
  // von allen Recognizer-Instanzen gemeinsam genutzte Ressourcen
  AllocateGlobalRecognizerResources();
}
```

Der Konstruktor der Formularklasse übernimmt das Belegen der gemeinsam genutzten Ressourcen – die `Form1_Closing()` wieder freigibt. Das Verlegen der Belegung auf den Start der ersten Recognizer-Instanz (also die Methode `bStartRecogClick()`) ist kein Problem, wenn man eine Prüfung á la `if (GlobalSoundcard == null)` einbaut und so eine eventuelle Doppelbelegung vermeidet. Eine analoge Freigabe (nämlich nach dem Ende des letzten Threads) kann dagegen nicht einfach in der Rückrufroutine erfolgen, dafür wäre eine Modifikation der Warteroutine `WaitForThreadEnd()` erforderlich, die auf das Ende des aktuellen Recognizer-Threads wartet (siehe dort).

```
protected override void Dispose( bool disposing )
// ...
#region Windows Form Designer generated code
// ...
#endregion

[STAThread]
static void Main()
{
  Application.Run(new IAThreadForm());
}
```

Der vom Designer generierte Code erschien nicht interessant genug, um hier wiedergegeben zu werden, zumal er keinerlei Besonderheiten enthält.

```
// Neues Recognizer-Objekt anlegen und starten
private void NewRecognizer(RecogData Data)
{
  TheRecognizer = new Recognizer(Data); // überschreibt alte Instanz
  // Rückmeldung
  TheRecognizer.OnRecognition = new
     Recognizer.OnRecognitionDelegate(OnRecognitionCB);
  TheRecognizer.Soundcard = GlobalSoundcard;  // Syncobject
  TheRecognizer.ThreadNumber = GlobalThreadNumber++;  // ID für Trace
  // Anlegen des neuen Threads und Start
  RecognizerDelegate = new RecogExecuteDelegate(TheRecognizer.Execute);
  RecognizerStatus = RecognizerDelegate.BeginInvoke(null, null);
  Trace.WriteLine("Neues Recognizer-Objekt angelegt und gestartet");
}
```

`NewRecognizer()` wird sowohl beim ersten Start des Recognizers (via `bStartRecogClick()`) als auch aus der Rückrufroutine heraus (für neue Parametersätze) ausgerufen. Diese Methode legt nicht nur ein neues Recognizer-Objekt an (und übergibt dabei das zu verwendende Vokabular als Objekt `Data`), sondern startet auch einen neuen Thread. Durch das Neubesetzen von `TheRecognizer` erlischt die (nach dem Auslaufen des Threads letzte) Referenz auf ein eventuelles Vorgänger-Objekt; Analoges gilt für das `IAsyncResult`-Schnittstellenobjekt `RecognizerStatus`. (Ein explizites `EndInvoke()` findet sich in diesem Programm ausschließlich für die letzte Thread-Instanz im Rahmen von `Form1_Closing()`. Die anderen Thread- und Schnittstellen-Instanzen fallen der Garbage Collection anheim, sobald keine weiteren Referenzen mehr auf sie bestehen.)

```
private delegate void WaitForThreadEndDelegate();

private void WaitForThreadEnd()
{
  Trace.WriteLine("WaitEnd: Warten auf Recognizer-Thread");
  RecognizerStatus.AsyncWaitHandle.WaitOne();
  Trace.WriteLine("WaitEnd: Recognizer-Thread beendet");
  Close();
}
```

```
    private void FormCloseTimer_Elapsed(object sender,
      System.Timers.ElapsedEventArgs e)
    {
      Close();
    }

    private void CloseRequestFromThread()
    {
#if USETIMER
      FormCloseTimer.Enabled = true;
#else
      WaitForThreadEndDelegate d =
        new WaitForThreadEndDelegate(WaitForThreadEnd);
      d.BeginInvoke(null, null);
#endif
    }
```

Die Methode `CloseRequestFromThread()` wird bei Erkennung des Befehls »Programmende« aus der Rückrufroutine des Recognizers aufgerufen, läuft also im Kontext des Hintergrund-Threads. Da `Form1_Closing()` wie erläutert seinerseits auf das Ende des Recognizer-Threads wartet, darf die Formularmethode `Close()` hier nicht direkt aufgerufen werden, sondern erst nach einer Entkoppelung, die hier entweder über einen Timer (`FormCloseTimer_Elapsed()`) oder über einen weiteren Hintergrund-Thread stattfindet: `WaitForThreadEnd()` wartet auf das Ende des Recognizer-Threads und ruft erst dann `Close()` auf.

Die Frage, ob hier nicht der Teufel durch den Beelzebub ersetzt wurde, ist durchaus berechtigt: `WaitForThreadEnd()` sorgt zwar offensichtlich für eine Entkoppelung – aber wird hier nicht `Close()` ebenfalls aus einem Hintergrund-Thread (nur eben einem anderen) heraus aufgerufen, und stand da nicht weiter vorne in diesem Kapitel, dass für Windows-Komponenten eine Synchronisation mit dem Vordergrund-Thread erforderlich ist?

Obwohl beide Feststellungen durchaus zutreffend sind, funktioniert die Sache. Der Grund: `Close()` arbeitet über die API-Funktion `SendMessage()`, bei der Windows in Eigenregie für eine Synchronisation mit dem Vordergrund sorgt. (Hier zeigt sich allerdings ein weiteres Mal, dass eine Klassenbibliothek ohne Quelltexte nur die Hälfte wert ist: Die Frage, ob `Close()` vor dem `SendMessage()` noch irgendwelche anderen, im unsynchronisierten Kontext eventuell kritischen Aktionen anzettelt, bleibt offen.)

Wenn Sie von allen Recognizer-Instanzen gemeinsam genutzte Ressourcen nicht erst beim Schließen der Form, sondern nach dem Auslaufen des letzten Threads wieder freigeben wollen (vgl. Seite 1024), können Sie `WaitForThreadEnd()` und `CloseRequestFromThread()` um einen entsprechenden `bool`-Parameter zur Unterscheidung zwischen »Ressourcen freigeben« und »Fenster schließen« erweitern.

```
//    private delegate void WaitForThreadEndDelegate(bool DoClose);
//    private void WaitForThreadEnd(bool DoClose)
//    {
//      // ...
//      FreeGlobalRecognizerResources();
//      if (DoClose) Close();
//    }
//
//    private void FormCloseTimer_Elapsed(object sender,
//      System.Timers.ElapsedEventArgs e)
//    {
//      Close();
//    }
//
//    private void CloseRequestFromThread(bool DoClose)
//    {
//#if USETIMER
//      FormCloseTimer.Enabled = true;
//#else
//      WaitForThreadEndDelegate d =
//        new WaitForThreadEndDelegate(WaitForThreadEnd);
//      d.BeginInvoke(DoClose, null, null);
//#endif
//    }
```

Dabei ist allerdings zu beachten, dass die Freigabe der globalen Ressourcen nun aus dem Hintergrund heraus geschieht.

```
// unsynchronisierter Rückruf des Recognizers bei "Befehl erkannt"
private void OnRecognitionCB(int Category, string Cmd)
{
  switch(Category)
  {
    case 1:
      Trace.WriteLine("CB: Befehl: "+Cmd);
      break; // Weiter im existierenden Thread
    case 2:
      Trace.WriteLine("CB Neue Parameter: "+Cmd);
      TheRecognizer.Terminated = true;
      switch (Cmd)
      {
        case "Vokabular 2":
          NewRecognizer(RecogDataSets[1]);
          break;
        case "Vokabular 3":
          NewRecognizer(RecogDataSets[2]);
          break;
      }
      break; // neue Thread läuft an, alter Thread läuft aus

    case 3:
      Trace.WriteLine("CB - Programmende");
      TheRecognizer.Terminated = true;
      bStartRecog.Enabled = true;
```

```
          if (cCloseOnTermination.Checked)
#if DEADLOCK
            Close();   // Denkfehler!
#else
            CloseRequestFromThread();
#endif
          break;  // Programmende
        }
      }
```

Da die Rückrufroutine des Recognizers bereits ausführlich erläutert wurde, sollte hier eine Beschreibung im Telegrammstil reichen: Nach der Bearbeitung normaler Befehle der Kategorie 1 (die hier aus der schlichten Ausgabe über `Trace` besteht) geht es im existierenden Thread weiter, bei Vokabularwechseln (Kategorie 2) wird per `NewRecognizer()` ein neues Objekt und ein neuer Thread angelegt, der alte Thread läuft ohne weitere Interaktion aus. Auf den Befehl »Programmende« reagiert die Anwendung mit dem Aufruf der Methode `CloseRequestFromThread()`, die ihrerseits abhängig vom Symbol `USETIMER` für einen entkoppelten `Close()`-Aufruf per Timer bzw. über einen weiteren Thread sorgt.

```
    private void bStartRecogClick(object sender, System.EventArgs e)
    {
      bStartRecog.Enabled = false;
      NewRecognizer(RecogDataSets[0]);
    }
```

Ein Klick auf die Schaltfläche »Start Recognizer« legt das erste Recognizer-Objekt an, übergibt dabei das erste Vokabular aus der Liste, und sorgt über das Deaktivieren der Schaltfläche dafür, dass es bei einer Instanz des Recognizers bleibt. An dieser Stelle könnte man wie erwähnt auch für das Belegen der globalen Ressourcen sorgen, wobei dem Aufruf von `AllocateGlobalRecognizerResources()` aber eine Prüfung auf »bereits belegt« vorausgehen muss.

```
    private void Form1_Closing(object sender,
      System.ComponentModel.CancelEventArgs e)
    {
      if (RecognizerStatus != null && !RecognizerStatus.IsCompleted)
      { // Programmende "von außen", nicht durch einen Recognizer-Befehl
        Trace.WriteLine("FormClose: Warten auf Recognizer-Ende");
        TheRecognizer.Terminated = true;
        RecognizerStatus.AsyncWaitHandle.WaitOne();
        RecognizerDelegate.EndInvoke(RecognizerStatus);
        Trace.WriteLine("FormClose: Recognizer-Ende erreicht");
      }
      // gemeinsam genutzte Ressourcen aller Recognizer-Instanzen freigeben
      FreeGlobalRecognizerResources();
      // damit die Ausgaben von Trace sichtbar bleiben
      MessageBox.Show(this, "Programm beendet", this.Text);
    }
  }
}
```

Diese Methode wird beim Schließen des Formulars (exakter: bei Erhalt der Windows-Nachricht `WM_CLOSEREQUEST`) aufgerufen und ist sowohl bei einem direkten Aufruf der Methode `Close()` als auch beim Schließen des Fensters über das Systemmenü bzw. per Alt+F4 mit von der Partie. In den letzteren beiden Fällen muss sie einen eventuell laufenden Recognizer-Thread beenden und auf sein Ende warten, bevor sie die dem Recognizer zu Grunde liegenden Ressourcen freigibt.

Die Ausgabe über `MessageBox.Show()` dient, wie im dazugehörigen Kommentar beschrieben, ausschließlich dem Offenhalten des Ausgabe-Fensters des Debuggers. Dieser Aufruf ließe sich im Prinzip auch in der Methode `Main()` (nach `Application.Run()`) unterbringen; dort steht der Text der Titelleiste nicht mehr zur Verfügung, weil das Formularobjekt nach dem Rücksprung aus `Run()` bereits abgebaut ist.

24.6.1 Der Programmablauf

Die Rückmeldungen des Programms im Ausgabefenster sehen typischerweise so aus:

```
Neues Recognizer-Objekt angelegt und gestartet
0: Warten auf Soundcard-Lock
0: Soundcard-Lock erhalten
CB: Befehl: Wort 1.1
CB: Befehl: Wort 1.2
CB Neue Parameter: Vokabular 2
Neues Recognizer-Objekt angelegt und gestartet
0: Soundcard-Lock freigegeben
1: Warten auf Soundcard-Lock
1: Soundcard-Lock erhalten
CB: Befehl: Wort 2.1
CB: Befehl: Wort 2.1
CB Neue Parameter: Vokabular 3
Neues Recognizer-Objekt angelegt und gestartet
2: Warten auf Soundcard-Lock
1: Soundcard-Lock freigegeben
2: Soundcard-Lock erhalten
CB: Befehl: Wort 2.1
CB: Befehl: Wort 2.1
CB - Programmende
2: Soundcard-Lock freigegeben
WaitEnd: Warten auf Recognizer-Thread
WaitEnd: Recognizer-Thread beendet
```

Wie durch die Hervorhebung angedeutet, ist die Reihenfolge beim Start einer neuen Thread-Instanz und dem Beenden des Vorgängers nicht immer dieselbe. Tatsächlich haben wir ein gutes Dutzend Versuche gebraucht, um einmal eine Überschneidung zwischen zwei Thread-Instanzen zu erwischen – und genau darin liegt die Verzwicktheit bei der Thread-Programmierung: Je nach Problemstellung kann es auch ohne `lock` oder mit einem `lock` an der

leider nur fast richtigen Stelle x Mal gut gehen, und dann eben das eine oder andere Mal nicht.

In einer anderen Situation – nämlich dem Klick auf das Schließfeld des Fensters bei laufendem Recognizer – wird dagegen sofort klar, wozu die Synchronisation gut ist:

```
Neues Recognizer-Objekt angelegt und gestartet
0: Warten auf Soundcard-Lock
0: Soundcard-Lock erhalten
CB: Befehl: Wort 1.1
CB: Befehl: Wort 1.2
CB Neue Parameter: Vokabular 2
Neues Recognizer-Objekt angelegt und gestartet
0: Soundcard-Lock freigegeben
1: Warten auf Soundcard-Lock
1: Soundcard-Lock erhalten
CB: Befehl: Wort 2.1
FormClose: Warten auf Recognizer-Ende
1: Soundcard-Lock freigegeben
FormClose: Recognizer-Ende erreicht
```

Hier ist die Synchronisation in jedem Fall notwendig: Ohne sie würde `Form1_Closing()` die Ressourcen des Recognizers abräumen, während der Thread noch läuft.

24.7 Weitere Synchronisationsobjekte

Außer den von C# aus über `lock` erreichbaren »Critical Sections« hat Windows noch eine Reihe weiterer Synchronisationsobjekte zu bieten, auf die dieser Abschnitt noch kurz eingeht. (Allzu sehr in die Tiefe geht dieses Material nicht: Wer sich für Dinge wie `Interlocked.CompareExchange()` oder die Interprozess-Synchronisation über `Mutex` interessiert, sei auf die Online-Dokumentation verwiesen.)

24.7.1 Codebeispiel –Threads mit Work-Liste

Die Grundlage zur Diskussion der Problemstellung und zur praktischen Demonstration liefert das Beispielprojekt *InteractiveThread2*, das sich von ihrem im vorangehenden Abschnitt vorgestellten Vorläufer nur in wenigen Punkten unterscheidet.

- Die (nach wie vor unsynchronisierte) Rückrufroutine des Threads beschränkt sich darauf, erkannte Befehle in einer Liste namens `PendingCommands` zu speichern, die natürlich während des Zugriffs gesperrt wird.

```
// unsynchronisierter Rückruf des Recognizers bei "Befehl erkannt"
private void OnRecognitionCB(int Category, string Cmd)
{
  // neues Ereignis an die Liste anhängen
  lock(PendingCommands)
  {
    PendingCommands.Add(new RecogCommand(Category, Cmd));
    SignalNewItem();
  }
}
```

`SignalNewItem` ist sozusagen ein Platzhalter: Diese Methode muss dem Vordergrund-Thread auf irgendeine Weise mitteilen, dass ein neues Listenelement vorhanden ist.

- Die zur Speicherung verwendete Klasse ist so einfach, wie man es auch erwarten sollte.

```
public class RecogCommand
{
  public readonly int Category;
  public readonly string Word;

  public RecogCommand(int Category, string Word)
  {
    this.Category = Category;
    this.Word = Word;
  }
}
```

- `PendingCommands` ist eine Instanz der Klasse `ArrayList` und wird zusammen mit dem Formularobjekt konstruiert.

```
public class IAThreadForm2 : System.Windows.Forms.Form
 { // ...
   IAsyncResult RecognizerStatus;  // Ergebnis von BeginInvoke
   ArrayList PendingCommands = new ArrayList();
```

- Das Abarbeiten eines Listenelements geschieht schließlich über eine Methode `EvaluateNextCmd()`, die dem Rückruf aus der Vorgängerversion recht ähnlich sieht:

```
  private bool EvaluateNextCmd()
    { RecogCommand NextCmd;

     lock(PendingCommands)
      {
       if (PendingCommands.Count == 0) return false;
        NextCmd = (RecogCommand)PendingCommands[0];
        PendingCommands.Remove(NextCmd);
        // true für weitere Elemente
        Result = PendingCommands.Count > 0;
      }

      Debug.Assert(!listBox1.InvokeRequired,
       "EvaluateNextListItem: falscher Thread-Kontext");
```

```
    switch(NextCmd.Category)
    {
      case 1:
        listBox1.Items.Add(String.Format("{0}: {1}",
           NextCmd.Category, NextCmd.Word));
        break; // Weiter im existierenden Thread
      case 2:
        listBox1.Items.Add("CB Neue Parameter: "+NextCmd.Word);
        TheRecognizer.Terminated = true;
        switch (NextCmd.Word)
        {
          case "Vokabular 2":
            NewRecognizer(RecogDataSets[1]);
            break;
          case "Vokabular 3":
            NewRecognizer(RecogDataSets[2]);
            break;
        }
        break; // neue Thread läuft an, alter Thread läuft aus

      case 3:
        listBox1.Items.Add("CB - Programmende");
        TheRecognizer.Terminated = true;
        bStartRecog.Enabled = true;
        EndWaitThreadOrTimer(false);
        if (bCloseOnTermination.Checked)
          Close();   // hier ist das kein Denkfehler
        break;  // Programmende
    }
    return Result;
  }
```

Die wesentlichen Unterschiede sind hier:

- Die Methode wird ausschließlich im Kontext des Vordergrund-Threads ausgeführt. (Falls einmal doch nicht, bricht das Programm aufgrund der Prüfung durch `Assert()` mit einer Exception ab.)
- Die Befehle werden nicht direkt vom Recognizer geliefert, sondern aus `PendingCommands` gelesen (und unmittelbar danach aus dieser Liste entfernt). Das Funktionsergebnis zeigt an, ob die Liste noch weitere Elemente enthält.
- `Trace.WriteLine()` wurde durch die Speicherung in einer Listbox (listBox1) ersetzt.
- Da die Methode ausschließlich im Vordergrund ausgeführt werden soll, führt der direkte Aufruf von `Close()` hier nicht zu einem Deadlock; eine entsprechende Entkoppelung per Thread oder Timer ist (an dieser Stelle) unnötig.
- Die Implementation der bei Befehlen der Kategorie 3 aufgerufenen Methode `EndWaitThreadOrTimer` hängt von der jeweils verwendeten Synchronisierungstechnik ab. Mehr dazu in den folgenden Abschnitten.

- Beim Start der ersten Recognizer-Instanz wird eine Methode `StartWaitThreadOrTimer()` aufgerufen, die das Gegenstück zu `EndWaitThreadOrTimer()` darstellt (und deren Implementation ebenfalls von der verwendeten Synchronisierungstechnik abhängt).
- Beim Schließen des Formulars muss nicht nur der Recognizer-Thread, sondern auch der zur Entkopplung verwendete Thread bzw. Timer beendet werden.

```
private void Form1_Closing(object sender,
   System.ComponentModel.CancelEventArgs e)
{
  if (RecognizerStatus != null && !RecognizerStatus.IsCompleted)
  { // Programmende "von außen", d.h. nicht durch Recognizer-Befehl
    Trace.WriteLine("FormClose: Warten auf Recognizer-Ende");
    TheRecognizer.Terminated = true;
    EndWaitThreadOrTimer(true);
    RecognizerStatus.AsyncWaitHandle.WaitOne();
    RecognizerDelegate.EndInvoke(RecognizerStatus);
```

- Und schließlich: Das Formular enthält mit `listBox1` eine zusätzliche Listbox zur Anzeige der erkannten Befehle.

Was nun noch fehlt, ist die Antwort auf die Frage, wie das Programm nach dem Anhängen neuer Elemente an `PendingCommands` zum Aufruf von `EvaluateNextCmd()` kommt.

Die ersten beiden Möglichkeiten haben nur wenig mit neu vorzustellenden Synchronisationsobjekten zu tun: Denkbar wäre zum einen ein Aufruf der API-Funktion `PostMessage()` in `SignalNewItem()` in Kombination mit einer Empfängermethode, zum anderen – wieder einmal – ein schlichter Timer, der alle 100 msec prüft, ob `PendingCommands` mehr als 0 Elemente enthält. In diesem Fall wäre `SignalNewItem()` überflüssig.

```
private void PendingCmdTimer_Elapsed(object sender,
  System.Timers.ElapsedEventArgs e)
{ int CurCount;
  lock(PendingCommands)
  {
    CurCount = PendingCommands.Count;
  }
  if (CurCount > 0)
     EvaluateNextCmd();
}
```

Da sich `lock` wie bereits erwähnt von ein und demselben Thread aus ohne weiteres auch mehrfach verwenden lässt, ist die folgende Formulierung ebenfalls möglich:




```
private void PendingCmdTimer_Elapsed(object sender,
  System.Timers.ElapsedEventArgs e)
{
```

```
  lock(PendingCommands)
  {
    while (PendingCommands.Count > 0)
      EvaluateNextCmd();
  }
}
```

Der Unterschied zwischen den beiden Varianten: Die erste lässt dem Hintergrund-Thread nach dem Bearbeiten jedes Befehls die Möglichkeit, weitere Elemente in die Liste einzufügen, die zweite blockiert die Liste so lange, bis sämtliche Elemente bearbeitet sind.

24.7.2 Events

Die Bezeichnung EVENT steht bei der Windows-Programmierung für Objekte, die zu jedem Zeitpunkt einen von zwei Zuständen – »signalisiert« oder »unsignalisiert« – einnehmen, wobei ein Thread darauf warten kann, dass ein anderer Thread ein solches Objekt in einen signalisierten Zustand versetzt. Vorausgesetzt, `MyEvent` steht für ein solches Objekt, dann blockiert die Anweisung

```
MyEvent.WaitOne();
```

den aufrufenden Thread so lange, bis ein anderer Thread `MyEvent` über die Methode `Set()` in den signalisierten Zustand versetzt.

C# bzw. die .NET-Bibliotheken definieren in diesem Zusammenhang zwei Klassen namens `ManualResetEvent` und `AutoResetEvent`, die sich nur in einem Punkt unterscheiden: `AutoResetEvent`-Objekte fallen von selbst wieder in den unsignalisierten Zustand, nachdem das System die Blockierung aller auf sie wartenden Threads aufgehoben hat, `ManualResetEvent`-Objekte bleiben dagegen so lange signalisiert, bis ihre Methode `Reset()` aufgerufen wird.

Eine Version des Programms `InteractiveThread`, die ein `AutoResetEvent`-Objekt zur Benachrichtigung über neue Befehle einsetzt, baut auf den folgenden Komponenten auf:

- Das Formular enthält ein Datenfeld für ein entsprechendes Objekt, dessen Konstruktor mit `false` (unsignalisierter Zustand) aufgerufen wird:

  ```
  private AutoResetEvent PendingCmdEvent = new AutoResetEvent(false);
  ```

- Die beim Anhängen neuer Befehle an `PendingCommands` aufgerufene Methode `SignalNewCommand()` signalisiert dieses Objekt:

  ```
  private void SignalNewCommand()
  {
    PendingCmdEvent.Set();
  }
  ```

Eine Methode des Formulars, die in einem eigenen Thread ausgeführt wird, übernimmt das Warten auf neue Listenelemente:

```
private void PendingCmdEventThread()
{
  while (!PendingCmdEventThreadTerminated)
  {
    PendingCmdEvent.WaitOne();
    if (!PendingCmdEventThreadTerminated)
    {
       EvaluateNextCmdDelegate d = new
       EvaluateNextCmdDelegate(EvaluateNextCmd);
       // existierende Listenelemente der Reihe nach bearbeiten
       while ((bool)listBox1.Invoke(d))
        ;
    }
  }
}
```

`WaitOne()` hält den Thread also so lange an, bis `PendingCmdEvent` signalisiert wird, und ruft dann `EvaluateNextCmd()`, wobei der Aufruf hier über `listBox1.Invoke()` geschieht – also synchronisiert zum Vordergrund-Thread. Die Formulierung über eine `while`-Schleife trägt der Möglichkeit Rechnung, dass mehr als ein Befehl eingegangen ist, bevor das Programm zur Bearbeitung kommt – schließlich sieht man `PendingCmdEvent` nicht an, wie oft die `Set()`-Methode dieses Objekts aufgerufen wurde. Dass `PendingCmdEventThreadTerminated` hier gleich zweimal geprüft wird, hat einen Grund: Irgendwann wird das Programm diesen Thread auch einmal beenden wollen und muss ihn dazu erst einmal über eine Signalisierung des Events wecken.

```
private void EndWaitThreadOrTimer(bool SetSignal)
{
  PendingCmdEventThreadTerminated = true;
  if (SetSignal)
    SignalNewCommand();
}
```

Eine Alternative zu diesem Verfahren – also der Signalisierung des Events, *ohne* dass die Liste einen neuen Befehl enthalten würde – stellt der Einsatz einer anderen Überladung von `WaitOne()` dar.

```
//      private void PendingCmdEventThread()
//      {
//        while (!PendingCmdEventThreadTerminated)
//        {
//          if (PendingCmdEvent.WaitOne(100, false))
//          {
//            EvaluateNextCmdDelegate d = new
//              EvaluateNextCmdDelegate(EvaluateNextCmd);
//            while ((bool)listBox1.Invoke(d))
//                ;
//          }
```

```
//        }
//     }
//
//    private void EndWaitThreadOrTimer(bool SetSignal)
//    {
//      PendingCmdEventThreadTerminated = true;
//    }
```

Diese Variante von `WaitOne()` kehrt im 100-Millisekunden-Takt zum aufrufenden Thread zurück, wobei das Funktionsergebnis nur dann `true` ist, wenn der Event signalisiert wurde. (`EndWaitThreadOrTimer` ist noch einmal aufgelistet, weil diese Methode dann in keinem Fall eine Signalisierung enthalten darf – die würde sonst mit dem Erhalt eines neuen Befehls verwechselt.)

➡ Das Anlegen und Starten des Warte-Threads bietet keine weiteren Überraschungen oder Neuigkeiten.

```
private void StartWaitThreadOrTimer()
{
  PendingCmdEventThreadTerminated = false;
  PendingCmdEventDelegate d =
    new PendingCmdEventDelegate(PendingCmdEventThread);
  d.BeginInvoke(null, null);
}
```

Das war's – jedenfalls fast: Was noch aussteht, ist die Antwort auf die Frage, wieso die Signalisierung in `EndWaitThreadOrTimer()` von einem Parameter (`SetSignal`) abhängig gemacht wurde. Wenn Sie noch einmal zu den Listings von `EvaluateNextListItem()` und `Form1_Closing()` (ab Seite 1031) zurückblättern, werden Sie feststellen, dass diese Methode von `EvaluateNextListItem()` aus mit `false`, von `Form1_Closing()` aus dagegen mit `true` aufgerufen wird. Dass es beim Schließen des Formulars nicht nur notwendig ist, das »Terminated«-Flag des Warte-Threads zu setzen, sondern ihn auch noch aufzuwecken, damit er das auch mitkriegt, ist klar – aber wieso ist dieses Entblockieren von `EvaluateNextListItem()` aus nicht notwendig, nachdem diese Methode doch ebenfalls ausschließlich im Vordergrund läuft? Weil `EvaluateNextListItem()` zwar nicht im Kontext des Warte-Threads, aber aus ihm heraus (nämlich über `listBox1.Invoke()`) aufgerufen wird. Was dort nach dem Rücksprung als Nächstes kommt, ist die Prüfung des »Terminated«-Flags beim nächsten Schleifendurchlauf – und damit ein Abbruch.

So marginal diese Detail erscheinen mag: Kleinkrämerei ist es nicht. Wenn der `PendingCmdEvent` aus `EvaluateNextListItem` heraus signalisiert wird und der Warte-Thread danach abbricht, dann bleibt dieses Objekt im signalisierten Zustand. Auffällig wird dieser Fehler erst, wenn man das Häkchen der Checkbox FORM BEI TERMINATE SCHLIEßEN entfernt und nach dem Durchlauf der ersten drei Thread-Instanzen ein zweites Mal auf START RECOGNIZER klickt: In diesem Fall kommt der Warte-Thread sofort zum Zuge – und stürzt sich auf eine leere Liste.

Die letzte Frage wäre wohl, wo nun der Unterschied zwischen einem Timer mit einem Intervall von 100 Millisekunden und einem Event mit einem Timeout derselben Größe liegt? Der Timer arbeitet unabhängig von der Zahl und Geschwindigkeit eintreffender Befehle stur in seinem Takt, während ein Event *sofort* reagiert: Die dort angegebenen 100 Millisekunden sind eine maximale Wartezeit (die nur ein einziges Mal zu spüren ist – nämlich dann, wenn das Programm den Thread beenden will und deshalb bewusst einen Timeout produziert).

24.7.3 Semaphoren

Synchronisationsobjekte dieser Art sind eigentlich in erster Linie für das Pooling von Ressourcen gedacht: Eines der Standard-Einsatzbeispiele ist ein Dienst mit *m* asynchron laufenden Clients, denen *n* Instanzen (Soundkanäle, Mailslots, Named Pipes, Dateien ...) zur Verfügung stehen, wobei *m* größer als *n* werden kann. Als Prinzip gilt in einem solchen Fall: Wer zuerst kommt, mahlt zuerst – und wer zu spät kommt, muss so lange warten, bis *irgendeiner* der Vorgänger die Ressource wieder abgibt. Anders gesagt: Eine Semaphore ist ein Sperre mit einem Zähler. Wenn dieser Zähler mit *x* initialisiert wird, kann sie gleichzeitig von *x* verschiedenen Threads angefordert werden. Der Zähler wird bei jedem Zuschlag um Eins erniedrigt und bei jeder Besitzabgabe wieder erhöht. Ein Thread, der eine Semaphore anfordert, deren Zähler momentan auf 0 steht, wird so lange angehalten, bis irgendein anderer Thread den Besitz wieder abgibt.

Ein weiterer, nicht so offensichtlicher Einsatzbereich von Semaphoren ist die Bearbeitung von Listen, die von einem Thread befüllt und von einem oder mehreren anderen Threads abgearbeitet werden. Das Prinzip sieht auf den ersten Blick schwer nach einer Art Missbrauch aus.

- Die Datenquelle (also der »Befüllerthread«) hängt ein neues Element an die Liste an und gibt dann den Besitz an der Semaphore ab – obwohl er ihn nie angefordert hatte.

```
private void AddElem(object Elem)
{
  lock(TheList)
  {
    TheList.Add(Elem);
    TheListSemaphore.Release ();
}
```

- Der Datenabnehmer fordert die Semaphore an. Wenn er den (exakter: einen) Zuschlag erhält, entfernt er ein Element aus der Liste, bearbeitet es – und gibt den Zuschlag nie wieder ab.

```
private void WorkerThread()
{
  while(!WorkerThreadTerminated)
  {
    // Warten, bis die Liste mindestens ein Element enthält
    TheListSemaphore.Acquire();
    if(!WorkerThreadTerminated)
       break;
    lock(TheList)
    {
      object Elem = TheList[0];
      TheList.RemoveAt(0);
    }
      ProcessListEntry(Elem);
  }
}
```

Die zweite Prüfung des »Terminated«-Flags hat hier denselben Zweck wie im Beispiel des vorangehenden Abschnitts: Sie sorgt dafür, dass sich der Thread nicht nur aus sich selbst heraus, sondern auch von außen beenden lässt – mithin, wenn die Liste keine Elemente enthält. (Die für das »externe« Beenden des Threads zuständige Methode setzt erst die Abbruchbedingung und erhöht dann den Zähler der Semaphore je `Release()`-Aufruf um Eins.)

Die Frage, ob das denn wohl so ohne weiteres geht – zwei Threads, von denen einer ständig Zuschläge wieder abgibt, die er nie erhalten hat, und ein zweiter, der sich ausschließlich bedient, ohne jemals wieder etwas herauszurücken, lässt sich bei C# und der .NET-Laufzeitumgebung hundertprozentig mit »Ja« beantworten. Der recht amüsante Grund für diese Gewissheit: Obwohl Windows im Systemkern nicht nur die Implementation für Critical Sections (lock) und Mutexe, sondern auch die für Events und Semaphoren liefert, haben sich die Microsoft-Entwickler bei der .NET-Laufzeitumgebung eine C#-Klasse für letztere gespart. Oder eben: Dinge, die man sowieso selbst schreiben muss, haben den Vorteil, dass man wenigstens genau weiß, was drin ist.

Eine Semaphorenklasse

Dieser Abschnitt bessert das (scheinbare) Versäumnis der Microsoft-Entwickler nach und stellt die Implementation einer Klasse `Semaphore` vor. So sehr unsereins auch nach ewigen Ruhm lechzt – dafür ist das Meisterwerk leider um einige Größenordnungen zu klein ausgefallen:

```
public class Semaphore
  {
    private int Count;

    public Semaphore(int Count) { this.Count = Count; }
```

```
    public void Acquire()
    {
      lock(this)
      {
        while(Count == 0)
          Monitor.Wait(this);
        Count--;
      }
    }

    public void Release()
    {
      lock(this)
      {
        Count++;
        Monitor.Pulse(this);
      }
    }
  }
```

Beide Methoden sperren vor Zugriffen erst einmal das Objekt über `lock`. Wenn der Zähler momentan den Stand 0 hat, blockiert `Acquire` den aufrufenden Thread über `Monitor.Wait()` so lange, bis irgendein anderer Thread `Release()` aufgerufen hat. `Release()` erhöht den Zähler um Eins und sorgt über `Monitor.Pulse()` dafür, dass der *erste* wartende Thread (und nicht etwa eventuell mehrere Threads auf einmal) erlöst wird.

Der Standardkonstruktor von `Semaphore` muss übrigens nicht überladen werden, weil `Count` durch `new Semaphore()` auf 0 gesetzt wird.

Mit dieser Klasse lässt sich nun eine weitere Variante von `InteractiveThread` schreiben:

- Das Formularobjekt erhält ein Datenfeld für ein entsprechendes Objekt, dessen Konstruktor mit `0` (kein Listenelement verfügbar) aufgerufen wird.

  ```
  private Semaphore PendingCmdSemaphore = new Semaphore(0);
  ```

- Die beim Anhängen neuer Befehle an `PendingCommands` aufgerufene Methode `SignalNewCommand()` gibt den (nie von diesem Thread aus angeforderten) Besitz ab, erhöht also den Zähler um Eins.

  ```
  private void SignalNewCommand()
  {
    PendingCmdSemaphore.Release();
  }
  ```

- Eine Methode des Formularobjekts, die in einem eigenen Thread ausgeführt wird, übernimmt das Warten auf neue Listenelemente, das hier durch eine Anforderung der Semaphore geschieht:

```
private void PendingCmdSemaphoreThread()
{
  while (!PendingCmdSemaphoreThreadTerminated)
  {
     PendingCmdSemaphore.Acquire();
     if (!PendingCmdSemaphoreThreadTerminated)
     {
         EvaluateNextCmdDelegate d =
            new EvaluateNextCmdDelegate(EvaluateNextCmd);
         listBox1.Invoke(d);
     }
  }
}
```

`Acquire()` hält den Thread so lange an, bis der Zähler der Semaphore mindestens auf Eins steht (lies: die Liste mindestens ein Element enthält), und setzt ihn bei der Erteilung des Zuschlags gleich wieder herab.



Eine `while`*-Schleife für den Aufruf von* `EvaluateNextCmd()` *ist hier definitiv unnötig, weil Semaphoren (im Gegensatz zu Events) sehr wohl exakt mitzählen, wie oft »neues Listenelement« signalisiert worden ist. (Tatsächlich liegt in dieser Eigenschaft der eigentliche Witz von Semaphoren gegenüber Events, wenn es um Listenverarbeitungen geht.)*

Der Aufruf von `EvaluateNextCmd()` geschieht wie gehabt synchron zum Vordergrund-Thread und die zweimalige Prüfung von `PendingCmdSemaphoreThreadTerminated` ermöglicht auch hier das Beenden des Threads »von außerhalb«:

```
private void EndWaitThreadOrTimer(bool SetSignal)
{
   PendingCmdSemaphoreThreadTerminated = true;
   if (SetSignal)
      SignalNewCommand();
}
```

Wie auf Seite 1036 erläutert, wird `EndWaitThreadOrTimer()` bei der Erkennung des Recognizer-Befehls »Programmende« (also von `EvaluateNextCmd()` und damit vom Warte-Thread selbst aus) mit `false`, von `Form1_Closing()` dagegen mit `true` aufgerufen. Ohne diese Unterscheidung stünde der Zähler der Semaphore nach Beenden der Threads aufgrund des Recognizer-Befehls »Programmende« auf Eins.



Zum Anlegen und Starten des Warte-Threads gibt es auch hier nichts Neues zu sagen – bestenfalls, dass man dazu ohne weiteres denselben Delegatentyp wie für die Variante mit Events hätte verwenden können.

```
private void StartWaitThreadOrTimer()
{
   PendingCmdSemaphoreThreadTerminated = false;
   PendingCmdSemaphoreDelegate d =
        new PendingCmdSemaphoreDelegate(PendingCmdSemaphoreThread);
   d.BeginInvoke(null, null);
}
```

25 .NET und COM

COM, das *Common Object Model* von Microsoft, kann in mehrfacher Hinsicht als die Vorläufertechnologie von .NET gesehen werden. Ob und wann .NET diese Technik vollständig ablösen wird, ist zum gegenwärtigen Zeitpunkt noch unklar: Zum einen bauen nicht nur die existierenden Windows-Versionen, sondern auch die aktuelle Implementation der .NET-Laufzeitumgebung (nicht zuletzt: ADO.NET) massiv auf COM auf, zum anderen existiert auch außerhalb von Microsoft eine wahrhaft gigantische Codebasis, beispielsweise in Form von ActiveX-Steuerelementen. Dieses Kapitel geht kurz auf die Grundlagen von COM ein, erläutert dann den Einsatz von COM-Objekten in C#-Anwendungen und widmet sich schließlich der Frage, wie man das Pferd sozusagen von hinten aufzäumt – nämlich .NET-Komponenten in COM einsetzt.

25.1 Die COM-Technologie im Überblick

Microsofts COM hat eine recht bewegte Geschichte und eine Vielzahl von Namen hinter sich. Man könnte guten Gewissens sagen, dass sich hinter diesen drei Buchstaben alles verbirgt, was Microsoft und anderen in den Jahrzehnten (sic) vor .NET zum Thema »Wiederverwendbarkeit von Code« eingefallen ist.

25.1.1 Konzepte für wiederverwendbaren Code

Die beiden grundlegenden Varianten des Wiederverwendungskonzepts sind derart bekannt, dass hier eine einfache Aufzählung ausreichen sollte:

- *Statische Bibliotheken:* Benötigte Hilfsfunktionen werden zum Zeitpunkt der Programmerstellung aus Bibliotheksmodulen in die ausführbare Datei kopiert. Die Nachteile: Eine Suite mit zehn verschiedenen Programmen, die sämtlich eine Funktion *xyz* benötigen, enthält den Code von *xyz* zehn Mal in zehn verschiedenen .EXE-Dateien. Nach Korrekturen von *xyz* müssen alle zehn Programme neu kompiliert bzw. gebunden werden. Und schließlich: Wenn *xyz* einen Umfang von 40 KByte hat, dann wird die Suite um 400 KByte größer. Die Vorteile sind dieselben wie die Nachteile: Was ein Programm als direkten Bestandteil seiner .EXE-Datei hat, kann ihm keiner mehr nehmen. »Versionitis« findet also ausschließlich beim Hersteller statt.

- *Dynamische Bibliotheken:* Hilfsfunktionen werden in DLLs zusammengefasst; das Betriebssystem sorgt in Eigenregie für das Laden der benötigten DLLs und das Einsetzen von Einsprungadressen in den Code. Vorteile: Die Korrektur einer von zehn Programmen verwendeten Funktion *xyz* findet nur an einer Stelle statt, die notwendige Recompilierung beschränkt sich auf die DLL. Der Platzbedarf der zuvor als Beispiel verwendeten Suite sinkt um 360 KByte – und zwar nicht nur auf der Festplatte, sondern auch im Hauptspeicher, weil Windows den Code einer einmal geladenen DLL allen Prozessen gemeinsam zur Verfügung stellt. Das gilt nicht nur bei Hilfsfunktionen für Anwendungen, sondern auch für das System selbst, dessen Module fast ausnahmslos in Form von DLLs gehalten sind. Würde jede Anwendung das Laden einer weiteren Kopie von USER32, GDI usw. in den Hauptspeicher erforderlich machen, käme auch ein modernes System mit 256 MByte RAM oder mehr recht schnell in Platznöte (von Rechnern mit 4 MByte, wie sie zu den Anfangszeiten von Windows üblich waren, erst gar nicht zu reden). Die Nachteile sind auch hier eng mit den Vorteilen verwandt: Als Entwickler hat man nur noch begrenzt die Kontrolle darüber, welche Version einer DLL die eigene Anwendung hinterher zu Gesicht bekommt. Anders gesagt: »Versionitis« findet nicht nur beim Hersteller, sondern auch und vor allem beim Endabnehmer statt.

Was bei Diskussionen des Themas »statische vs. dynamische Bibliotheken« üblicherweise unter den Tisch fällt: Über Wiederverwendbarkeit bei der Programmentwicklung sagen diese beiden Speicherformate nicht das Geringste aus. Solange man sich auf eine Programmiersprache beschränkt, sind statische Bibliotheken amüsanterweise sogar geeigneter als ihre dynamischen Gegenstücke, weil sie spezifisch auf die jeweilige Programmiersprache zugeschnitten werden können. Das gilt vor allem, was die darin enthaltenen METADATEN betrifft – also zusätzliche Informationen, beispielsweise über Klassennamen, zugeordnete Methoden und ihre Signaturen, Sichtbarkeit etc. Kein Wunder also, dass die Klassenbibliotheken bekannter Programmiersprachen statisch gehalten sind. (Von der Möglichkeit, Metadaten in Bibliotheken unterzubringen, macht allerdings nur die Sprache C++ Gebrauch, die Signaturen über Erweiterungen der Funktionsnamen verschlüsselt – Stichwort: Name Mangling.)

DLLs sollen dagegen von allen Programmiersprachen aus verwendbar sein – und weil Microsoft es vor gut zwei Jahrzehnten erst einmal versäumt hat, einen sprachübergreifenden Standard für Metadaten zu definieren, ist dort der kleinste gemeinsame Nenner angesagt: Die Aufrufkonventionen von Pascal, keine Unterscheidung zwischen Groß- und Kleinbuchstaben in den Funktionsnamen, und keinerlei zusätzliche Informationen über den Typ von Parametern und Rückgabewerten.

Fast könnte man sagen: Dass die von Microsoft später eingeführten Techniken zur Wiederverwendung von Code auf DLLs und nicht auf statischen Bibliotheken aufbauen, liegt nicht zuletzt daran, dass mit DLLs schon einmal ein sprachübergreifender Standard vorhanden war – auch wenn sich dieser Standard im Wesentlichen dadurch auszeichnet, dass ihm eigentlich *alles* fehlt.



Obwohl die Namenserweiterung .lib traditionell für statische Bibliotheken steht, arbeiten Microsofts und Borlands C/C++ mit einer großen Zahl vergleichsweise kleiner .lib-Dateien, die Namen wie User32.lib, Gdi32.lib etc. tragen – und kein einziges Byte Code enthalten, sondern tatsächlich »nur« Metadaten: Aus ihnen bezieht der Linker des Entwicklungssystems die Information, welche Querverweise er für den Windows-Linker hinterlassen muss, damit letzterer beim Start einer Anwendung die Einsprungadressen der von User32.dll, Gdi32.dll usw. exportierten API-Routinen einsetzen kann.

25.1.2 COM-Metadaten

Was neben dem nackten Code von Routinen für die objektorientierte Programmierung notwendig ist, sind zusätzliche Informationen über die Zusammenhänge zwischen Klassen-, Methoden- und Feldnamen, die dazugehörigen Signaturen, die Sichtbarkeit der einzelnen Elemente usw.

Für statische Bibliotheken behelfen sich die bekannten Programmiersprachen hier mit entsprechenden, separaten Quelltextdateien (Interface-Deklarationen bei Delphi, per *#include* eingefügte Header bei C++), ohne die Klassensysteme wie Microsofts MFC oder Borlands VCL praktisch unbenutzbar wären.

Das Ergebnis der Bemühungen von Microsoft, für die Implementation von COM-Objekten in DLLs ein entsprechendes, aber eben sprachübergreifendes Pendant zu schaffen, hört auf den Namen IDL (Interface Description Language) und sieht im Rohformat ungefähr so aus:

```
[ object, dual, uuid(…), helpstring(…), QUEUEABLE ]
interface IShip:IDispatch
{
   [propput, id(1)] HRESULT CustomerId ([in] long CustId);
   [propput, id(2)] HRESULT OrderId ([in] long OrderID);
   [id(3)] HRESULT LineItem ([in] long Qty);
   [id(4)] HRESULT Process ();
}
```

Hier wird also der Zusammenhang zwischen einem Namen (`IShip`), einem (in IDL noch als UUID = »universally unique identifier« bezeichneten) GUID, diversen Attributen, und vor allem den dazugehörigen Methodennamen und ihren Bezeichnern festgelegt. Dabei kommt ein System generalisierter (OLE-)

Datentypen zum Einsatz, das zwar als Vorläufer des CLS gesehen werden kann, aber nur im geschichtlichen Sinne: Da diese Systematik erst geschaffen wurde, nachdem sich Programmiersprachen wie C, C++, Delphi und VB längst etabliert hatten, sind bei praktisch allen komplexen Datentypen mehr oder weniger aufwändige Konvertierungen nötig, und von einer gemeinsamen Verwaltung können COM-Programmierer nur träumen.

Frühe und späte Bindung

Für den Zugriff auf die per IDL beschriebenen Daten bzw. die von einer COM-Schnittstelle wie `IShip` definierten Methoden stehen im Wesentlichen zwei Möglichkeiten zur Verfügung:

- *Frühe Bindung*: Zum Kompilierungszeitpunkt der Anwendung (also des COM-Clients) steht eine COM-Typbibliothek (*.tlb*-Datei) zur Verfügung, die ebendiese Beschreibung enthält. Compiler und Linker setzen in diesem Fall anstelle von Methodennamen wie `OrderID` echte Indizes ein (im Sinne von: direkter Aufruf der Methode, deren Adresse über den Index 2 des Schnittstellenzeigers erreichbar ist).
- *Späte Bindung:* Compiler und Linker übergeben Methodennamen an die (fix vorgegebene) Methode `IDispatch.Invoke()`, die sich ihrerseits zur Laufzeit um die Ermittlung echter Einsprungadressen kümmert. Vorausgesetzt wird dabei, dass die COM-Schnittstelle `IDispatch` implementiert (und – ganz recht – dass die IDL-Beschreibung für `IDispatch()` ihrerseits irgendwo fix in das Entwicklungssystem eingebaut ist).

Auch bei früher Bindung spielt die Registrierung von Windows eine zentrale Rolle: Sie muss unter anderem die Information nachliefern, welche DLL oder EXE die Implementation einer Schnittstelle wie `IShip` enthält (Name und Pfad; dazu kommen Informationen über das Thread-Modell usw.).



Das »Anmelden« von COM-Komponenten – sei es mit einem Kommandozeilen-Parameter wie /regserver oder mit der speziell dafür vorgesehenen Konsolenanwendung regsvr32 – sorgt für ebendiese Einträge in der Registrierung.

COM-Komponenten müssen für den Einsatz mit .NET nicht nur auf dem Entwicklungssystem registriert sein, sondern auch auf dem Zielsystem. Das bedeutet, dass Sie im Setup Ihrer .NET-Anwendung einen entsprechenden Aufruf von regsvr32 einbauen müssen.

25.1.3 Die Grenzen von COM

COM stellt neben der sprachunabhängigen Abstraktion von Schnittstellen eine weitgehend abwärtskompatible Zusammenfassung mehrerer Technologien dar, die von Microsoft einmal zu ganz anderen Zwecken erfunden oder

übernommen worden sind – beispielsweise dem Einbetten und Verknüpfen von Objekten (OLE 1 und 2), der Automatisierung von Anwendungen über einfache Scriptsprachen (vor allem: Visual Basic for Applications) und der dazu notwendigen prozessübergreifenden Kommunikation. Auch wenn das Konzept zentral registrierter Schnittstellen auf den ersten Blick hin eingängig erscheinen mag: Sobald es um die Details geht, kann der Einsatz außerordentlich komplex werden. Der Grund dafür liegt in der Entwicklungsgeschichte: Sie ist derart von Paradigmenwechseln durchsetzt, dass die Implementation aufeinandergetürmtes Stückwerk werden *musste*. Genauso natürlich ist, dass die Compiler-Entwickler angesichts der komplexen Mechanismen mit »Fix-fertig«-Bibliotheken gekontert haben (Microsoft mit der MFC für C++ und entsprechenden Erweiterungen für VB, Borland mit einem Klassensystem in der VCL). Das Ergebnis dieser Bemühungen ist einerseits, dass sich einfache COM-Anwendungen recht schnell schreiben lassen, aber eben nur mit einer Untermenge der tatsächlichen Möglichkeiten – und wer die ausschöpfen will, wird über die Klassensysteme mit einer zusätzlichen, nicht minder komplexen Schicht konfrontiert. Kein Wunder, dass über die COM-Programmierung eine Unzahl dickleibiger Bücher geschrieben worden ist. (Hier geht es wohlgemerkt um die Anwendung – nicht etwa um die Implementation eigener Hüllklassensysteme. Die ist dem Vernehmen nach nur wenigen Auserwählten auf diesem Planeten in Gänze verständlich.)

Das zweite, nicht minder große Manko wird im Zusammenhang mit .NET (und auch in diesem Buch) derart oft thematisiert, dass es hier nur noch einmal kurz erwähnt werden soll: Wie die Praxis gezeigt hat, ist die zentrale Registrierung von Schnittstellen eben nur auf den ersten Blick eine gute Idee, weil sie den Parallelbetrieb unterschiedlicher Versionen einer Schnittstelle nur noch über Tricks zulässt. Ein Beispiel dafür ist DirectX mit Schnittstellen wie *IDirectDraw*, *IDirectDraw2*, *IDirectDraw4* – wobei es hier wohlgemerkt nicht um Korrekturen ein und derselben Version, sondern um eine fortlaufende Entwicklung mit *unterschiedlichem Funktionsumfang* geht. DirectX 8.0 und 8.1 definieren dagegen ein und dieselben Schnittstellen in unterschiedlichen *Überarbeitungszuständen*: Wer DirectX 8.1 installiert hat, kommt an die Implementationen aus der Version 8.0 definitiv nicht mehr heran.

Mit einem dritten COM-Problem verhält es sich ähnlich, was die Häufigkeit der Erwähnung im Zusammenhang mit .NET betrifft: Das Schnittstellenkonzept für das Objektmodell von .NET ist weder ein direkter Vertreter noch Bestandteil des COM-Konzepts, dem in dieser Hinsicht vor allem eines fehlt: virtuelle Methoden und die Möglichkeit, sie in abgeleiteten Klassen zu überschreiben. Zwar hält Sie in COM niemand davon ab, eine Schnittstelle wie `IShip` vollständig über eine neue Schnittstelle `IShip2` zu verkapseln und dort die öffentliche Methode `IShip2.CustomerID()` neu zu definieren, aber an die Stellen in der Implementation von `IShip`, an denen die »Basisklassenvariante« von `CustomerID` aufgerufen wird, kommen Sie nur heran, wenn Sie über die Quelltexte der Implementation verfügen.

DCOM

Der vierte und wohl letzte gewichtige Grund, der Microsoft förmlich dazu zwang, sich etwas Neues zu überlegen, hat nicht einmal direkt etwas mit dem Grundprinzip von COM zu tun (obwohl die für die prozessübergreifende Kommunikation verwendeten Techniken – Stichwort: Marshalling – natürlich auch bei der Kommunikation zwischen Prozessen auf verschiedenen Computern mit von der Partie sind): DCOM (Distributed COM) krankt im Gegensatz zum Konkurrenzmodell CORBA daran, dass diese Erweiterung zu einem Zeitpunkt entworfen wurde, als man bei Microsoft dem Internet noch keinen allzu hohen Stellenwert einräumte. Zum einen greift das Sicherheitsmodell für den Betrieb verteilter Anwendungen über das Internet viel zu kurz, zum anderen – und für Microsoft letztendlich schmerzhafter – findet die Kommunikation zwischen DCOM-Komponenten über ein proprietäres, weitgehend ungeschütztes Protokoll statt, dem jede Firewall sinnvollerweise recht schnell den Garaus macht.

COM+

Diese Weiterentwicklung von COM – vermutlich die letzte in der langen Reihe – geschah erst einmal unter dem Namen *Microsoft Transaction Services* und offensichtlich parallel zu .NET. COM+ bemüht sich, einige der mittlerweile erkannten Schwachstellen ihrer Vorgänger auszubügeln; unter anderem sind dort nicht nur Transaktionen, sondern auch Rückrufe zum Signalisieren von Ereignissen vorgesehen (wofür man mit COM noch eine eigene Schnittstelle benötigt, die Clients dem Server zur Verfügung stellen müssen. Damit werden sie selbst zum Server und sorgen so fast automatisch für ein prächtiges Durcheinander im Sicherheitsmodell.) Dass dabei eine weitere, noch komplexere Schicht herausgekommen ist, liegt nicht in der Schuld der Entwickler, sondern der unvermeidlichen Forderung nach Abwärtskompatibilität.

Eine weitere Verbesserung stellen verschiedene systemseitige Dienste dar, die darauf abzielen, COM-Objekte in puncto Kommunikationsgeschwindigkeit und Initialisierung Beine zu machen – was gerade im Zusammenhang mit dem Betrieb von Webservern schon immer ganz oben auf der Wunschliste stand. Ersteres wird durch zusätzliche Forderungen an die Implementation von COM-Objekten erreicht, die eine Ausführung in einem gemeinsamen Prozessraum und somit schnelle *in-process*-Aufrufe ohne Marshalling gestatten – die verwaltete Umgebung der .NET-Konzeption lässt also grüßen; zweiteres lässt sich durch ein Pooling einmalig initialisierter Objekte erreichen, bringt für die Implementation der Objekte aber die zusätzliche Forderung mit sich, dass sie statuslos sein müssen. Für die Bearbeitung von HTTP-Anfragen ist das ideal, das Protokoll selbst ist ja statuslos, für andere Anwendungsbereiche und für COM-Objekte generell ist dies jedoch keine Alternative.

25.2 .NET-Hüllklassen für COM

Die Integration von COM-Objekten in die .NET-Laufzeitumgebung geschieht über STELLVERTRETERKLASSEN in der jeweiligen Sprache (im gegebenen Fall also: C#). Klassen dieser Art unterscheiden sich syntaktisch in keiner Weise von anderen C#-Klassen und können selbstverständlich auch als Basis weiterer C#-Klassen dienen. Technische Unterschiede gibt es natürlich schon, und deshalb hat man bei Microsoft für COM-Stellvertreterklassen eine eigene Bezeichnung geschaffen: Runtime Callable Wrapper (»zur Laufzeit aufrufbare Hülle«) oder kurz RCW.

25.2.1 Codebeispiel – Anlegen von RCWs

Die automatisierte Generierung einer Hüllklasse in VS.NET setzt voraus, dass zu der COM-Schnittstelle eine Typbibliothek in Form einer *.tlb*-Datei existiert. Wie zuvor erläutert, liefern derartige Dateien die Metadaten (Methodennamen, Parametertypen usw.), die in einer COM-DLL nicht enthalten sind. Die Vorgänge werden nun anhand des Beispielprojekts *RCW-PhotoshopImport* demonstriert.

Der Menübefehl PROJEKT/VERWEIS HINZUFÜGEN aktiviert das gleichnamige Dialogfeld, das auf seiner Karteikarte COM sämtliche in der Registrierung des Systems verzeichneten COM-Schnittstellen anzeigt. Wie in Abbildung 25.1 am Beispiel von Adobe Photoshop zu sehen, müssen der Komponentenname und der Name der *.tlb*-Datei nicht unbedingt übereinstimmen; der Pfad zur *.tlb*-Datei wird ebenfalls den Registrierungsinformationen entnommen.

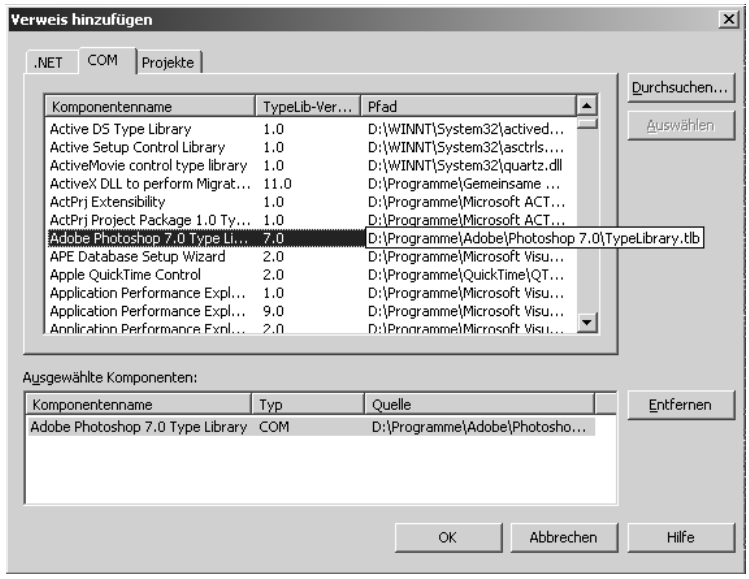


Abbildung 25.1: Erweiterung des Projekts um einen Verweis auf eine COM-Komponente

Das Ergebnis dieser Aktion ist eine DLL im Stil von .NET (also mit Metadaten in Form eines *Manifest*) im Unterordner *obj* des aktuellen Projekts. Der Name dieser DLL ergibt sich durch den in der *.tlb*-Datei angegebenen Namen, dem .NET die Zeichenfolge »interop.« voranstellt – für Adobe Photoshop beispielsweise: *interop.PhotoshopTypeLibrary.dll*.

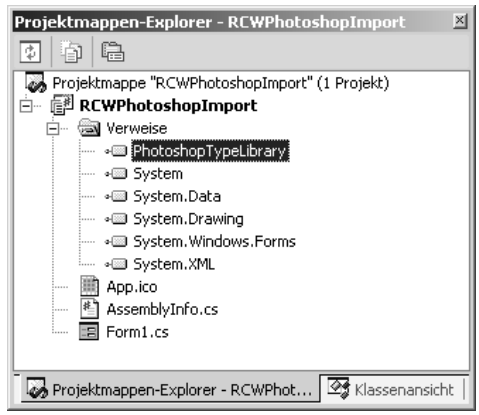


Abbildung 25.2: Der für die Photoshop-Hüllklasse erzeugte Namensraum

Wie in Abbildung 25.2 zu sehen, legt VS.NET beim Generieren der Hüllklasse auch einen neuen Namensraum an. Das Anlegen einer Instanz der COM-Hüllklasse geschieht mit derselben Syntax wie bei anderen C#-Klassen, wenn man einmal davon absieht, dass ausschließlich ein parameterloser Konstruktor zur Verfügung steht (dessen Implementation letztlich aus einem Aufruf der API-Funktion `CoCreateInstance()` besteht).

```
private void button1_Click(object sender, System.EventArgs e)
{
  PhotoshopTypeLibrary.PhotoshopApplication PS;
  PhotoshopTypeLibrary.IAutoPSDoc PSDoc;

  PS = new PhotoshopTypeLibrary.PhotoshopApplication();
  PSDoc = PS.Open("C:\temp\test.jpg");
  // ...
```

Selbstverständlich ist es auch in diesem Fall möglich, über ein entsprechendes `using` Tipparbeit einzusparen.

Das Hilfsprogramm TlbImp

Beim Anlegen von Hüllklassen über PROJEKT/VERWEIS HINZUFÜGEN ruft VS.NET das Hilfsprogramm TlbImp (für »Type Library Importer«) auf und belässt es bei den Standardvorgaben dieses Utilities: Konfigurationsmöglichkeiten aus VS.NET heraus gibt es hier nicht.

Wenn Sie sich selbst um die Benennung der DLL oder den entstehenden Namensraum kümmern wollen, können Sie *TlbImp.exe* auch über ein Kon-

solenfenster aufrufen und dabei entsprechende Schalter einsetzen: `/out:PhotoShopLib.DLL` legt beispielsweise den Namen der DLL explizit auf »PhotoShopLib« fest, `/name:PhotoShop` den Namensraum auf »PhotoShop« (vgl. Abbildung 25.3). Beim Aufruf ohne Parameter (oder mit `/?`) gibt *TlbImp.exe* eine Liste der verfügbaren Schalter aus.



Abbildung 25.3: Direkter Aufruf von TlbImp.exe

TlbImp.exe wird bei einer Standardinstallation von VS.NET im Ordner \Programme\Microsoft Visual Studio .NET\FrameworkSDK\bin gespeichert.



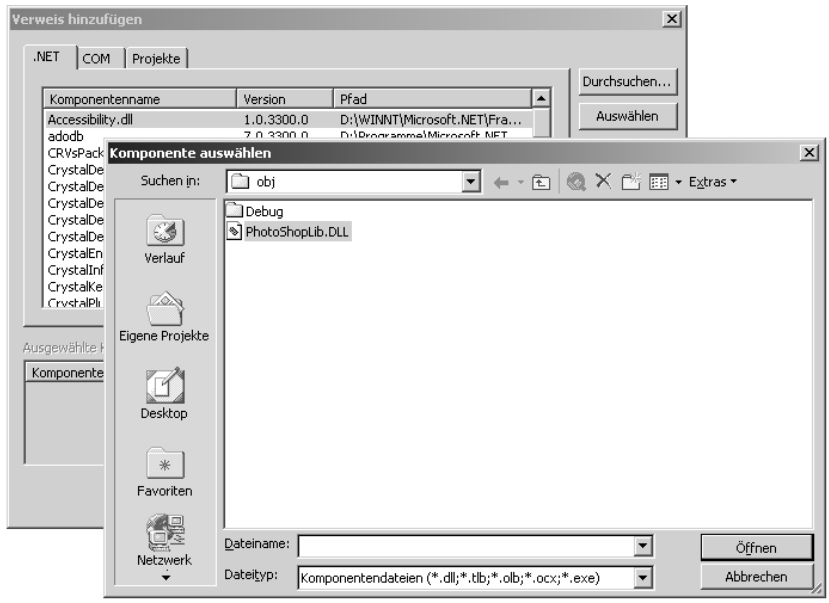
Es empfiehlt sich deshalb sehr, das zum Start benötigte Konsolenfenster nicht direkt, sondern über START\PROGRAMME\MICROSOFT VISUAL STUDIO .NET\VISUAL STUDIO .NET TOOLS\VISUAL STUDIO .NET EINGABEAUFFORDERUNG zu öffnen, weil diese Verknüpfung für das Einsetzen der entsprechenden Suchpfade sorgt.

Die Referenz auf die von *TlbImp.exe* erzeugte DLL müssen Sie dann über PROJEKT/VERWEIS HINZUFÜGEN selbst einbauen – wobei es nun im gleichnamigen Dialogfeld nicht um die Abteilung COM, sondern eben um .NET geht, und dort die Schaltfläche DURCHSUCHEN verwendet werden muss – vgl. Abbildung 25.4. Falls Sie es versehentlich von der Karteikarte COM aus versuchen, ist das allerdings auch kein Unfall: VS.NET merkt über den Dateiinhalt, worum es geht. (Tatsächlich ist es anders herum: COM-DLLs werden als solche erkannt, .NET-DLLs gelten schlicht als »Datei«. Funktionieren tut die Sache trotzdem.)

Direkte Deklaration

Wenn keine TLB vorhanden ist, geht es zur Not auch mit der direkten Codierung der Metadaten. Microsoft demonstriert das auf der MSDN-Website `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vaconIntroductionToCOMInteroperability.asp`) anhand des Mediaplayers.

Abbildung 25.4:
Eintrag der Referenz




```
using System;
using System.Runtime.InteropServices;

namespace QuartzTypeLib
{
  // Deklaration von IMediaControl als COM-Schnittstelle, die
  // von IDispatch abgeleitet ist:
  [Guid("56A868B1-0AD4-11CE-B03A-0020AF0BA770"),
      InterfaceType(ComInterfaceType.InterfaceIsDual)]
  interface IMediaControl   // Basis-Schnittstellen nicht aufgelistet
  {
      // IUnknown-Methoden werden *nicht* aufgelistet
        void Run();
        void Pause();
        void Stop();
        void GetState( [In] int msTimeout, [Out] out int pfs);
        void RenderFile(
            [In, MarshalAs(UnmanagedType.BStr)] string strFilename);
  ...
```

25.2.2 Der Einsatz

Unabhängig davon, wie die Klassendeklaration zu Stande gekommen ist, bietet die kontextbezogene Hilfe auch in diesem Fall Unterstützung. Abbildung 25.5 zeigt ein Beispiel dazu.

Wie auch nicht anders zu erwarten, bietet die .NET-Laufzeitumgebung für Instanzen von Stellvertreterklassen exakt dieselben Annehmlichkeiten wie für »normale« Objekte. Für das in Abbildung 25.5 angedeutete Beispiel gilt:

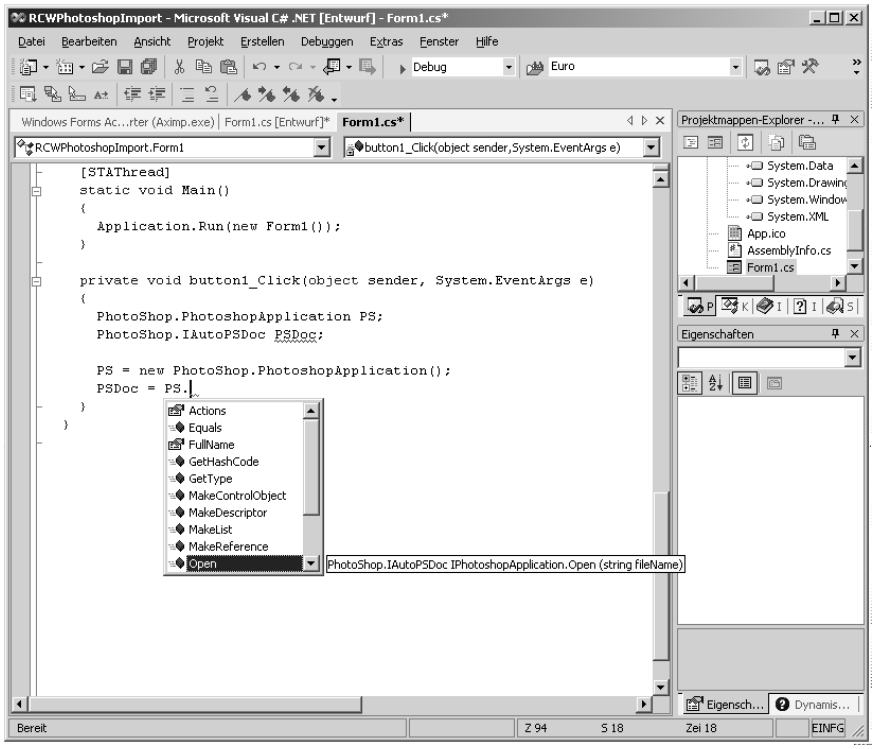


Abbildung 25.5: Die Syntaxhilfe ist auch für COM-Hüllklassen verfügbar

- Die Deklaration einer Instanz der Stellvertreterklasse sorgt für das Laden des RCW (hier: *PhotoshopLib.DLL*).
- Der Konstruktor des RCW erledigt (über die API-Funktion `CoCreateInstance`) für das Heraussuchen der Registrierungsinformationen und das Laden des COM-Servers (Photoshop.EXE).
- Der Garbage Collector räumt irgendwann die Instanzen der Stellvertreterklasse wieder ab und erniedrigt damit die entsprechenden COM-Referenzzähler, was letztendlich erst zum Entladen des COM-Servers und dann zum Entladen der RCW-DLL führt. Der in anderen Programmiersprachen bei der COM-Programmierung unumgängliche `finally`-Block erübrigt sich also – wieder einmal.

25.2.3 Fehlerprüfungen

Da beim Einsatz von COM zwei externe Dateien geladen werden müssen und diese Ladeoperationen *beide* erst auf Anforderung hin geschehen, sind entsprechende Fehlerprüfungen ein unbedingtes Muss.

Die Deklaration einer Stellvertreterklasseninstanz ist mit Sicherheit der unkritischere der beiden Schritte: Sie kann nur dann zu einer Exception führen, wenn die RCW-DLL fehlt, also die Installation beschädigt wurde.

Mit der Konstruktion einer Instanz der Stellvertreterklasse sieht es dagegen etwas anders aus: Auch unter normalen Umständen hat das Programm damit zu rechnen, dass der entsprechende COM-Server nicht auf dem Zielsystem installiert ist.

Sinnvollerweise sorgt man dafür, dass die entsprechenden Prüfungen nicht erst dann stattfinden, wenn der Benutzer die jeweilige Funktion der Anwendung tatsächlich einsetzen will. Ein denkbarer Ort und Zeitpunkt ist nach dem Laden des Formulars (der Konstruktor ist aber genauso gut geeignet).

```
private void Form1_Load(object sender, System.EventArgs e)
{
  try
  {
    // Exception, wenn die RCW-DLL fehlt
    PhotoshopTypeLibrary.PhotoshopApplication PS;

    // Exception, wenn Photoshop nicht auf dem System registriert ist
    PS = new PhotoshopTypeLibrary.PhotoshopApplication();
    button1.Enabled = true;
  }
  catch
  {
    button1.Enabled = false;
    MessageBox.Show("Photoshop nicht verfügbar, Funktion abgeschaltet.");
  }
}
```

Der unvermeidliche Nachteil einer solchen »vorauseilenden« Prüfung liegt darin, dass sie die Startzeit der Anwendung auch dann erhöht, wenn der Benutzer die COM-Komponente gar nicht einsetzt. Die (optimistische) Variante nimmt die Prüfung erst auf Anforderung vor (und sorgt dann gegebenenfalls dafür, dass sich der Fehler nicht wiederholt).

```
private void button1_Click(object sender, System.EventArgs e)
{
  // optimistischer Ansatz: RCW-DLL wird als vorhanden angenommen
  PhotoshopTypeLibrary.PhotoshopApplication PS;
  PhotoshopTypeLibrary.IAutoPSDoc PSDoc;

  try
  {
    PS = new PhotoshopTypeLibrary.PhotoshopApplication();
    // ...
  }
  catch
  {
    button1.Enabled = false;
    MessageBox.Show("Photoshop nicht verfügbar, Funktion abgeschaltet.");
  }
```

25.2.4 ActiveX-Steuerelemente

ActiveX-Steuerelemente stellen in mehrfacher Hinsicht eine Ausnahme der in den vorangehenden Abschnitten erläuterten Regeln dar.

- Damit sich ein (beliebiges) visuelles Steuerelement in ein Formular einsetzen lässt, muss seine Klasse von `System.Windows.Forms.Control` abgeleitet sein.
- OCX-Dateien enthalten im Gegensatz zu normalen COM-DLLs die zur Erzeugung der Hüllklassen benötigten Metadaten direkt. Eine *.tlb*-Datei ist deshalb unnötig.

Um der Forderung nach einer spezifischen Basisklasse gerecht zu werden, legt VS.NET für ActiveX-Steuerelemente nicht eine, sondern *zwei* Klassen (in zwei DLLs) an: Eine Hüllklasse für die COM-DLL, und eine zweite, von `Control` abgeleitete Klasse, die ihrerseits die COM-Hüllklasse verkapselt.

Zum Erzeugen dieser beiden Hüllklassen für ein ActiveX-Steuerelement in VS.NET verwenden Sie das über das Kontextmenü der TOOLBOX erreichbare Dialogfeld TOOLBOX ANPASSEN. Wie Abbildung 25.6 zeigt, listet die Karteikarte COM-STEUERELEMENTE dieses Dialogfelds die beim System registrierten ActiveX-Komponenten auf.

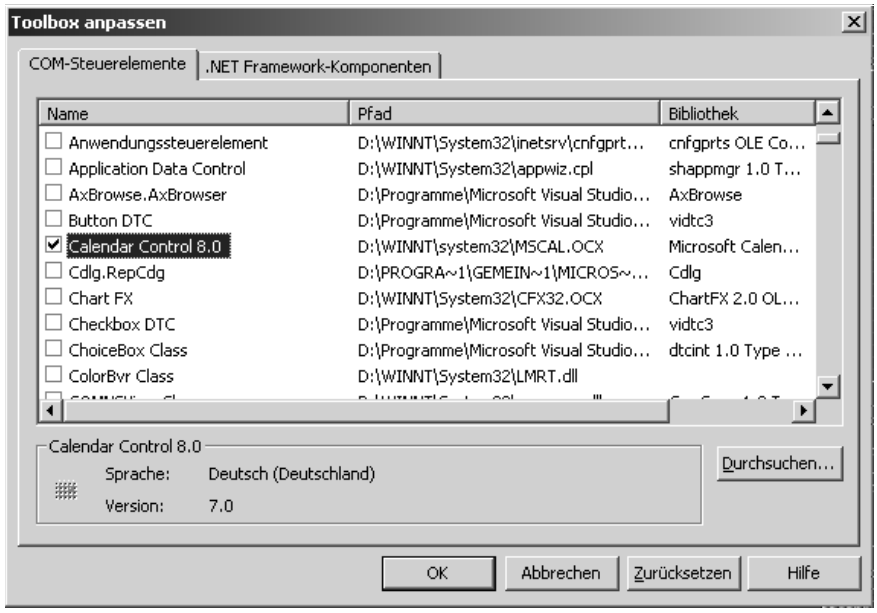


Abbildung 25.6: Hinzufügen des Kalender-Steuerelements zur Toolbox

Beim Klick auf OK erzeugt VS.NET eine (.NET-)DLL – im gegebenen Beispiel, also der Auswahl des Kalender-Steuerelements aus Microsoft Office: AXInterop.MSACAL.DLL für MSACAL.OCX –, speichert sie je nach Projekteinstellung im Unterordner obj\Debug bzw. obj\Bin und erweitert das Projekt um entsprechende Referenzen und Namensräume – nämlich sowohl auf die zugrunde liegende COM-Stellvertreterklasse (MSACAL) als auch um die ActiveX-Hüllklasse (AxMSCAL).

Ein auf diese Weise eingebettetes Steuerelement lässt sich auf dieselbe Weise im Designer verwenden wie die in der .NET-Klassenhierarchie vordefinierten Steuerelemente der Toolbox (vgl. Abbildung 25.7). Das gilt sowohl für die kontextbezogene Syntaxhilfe als auch für seine Eigenschaften und eventuelle Ereignisse.

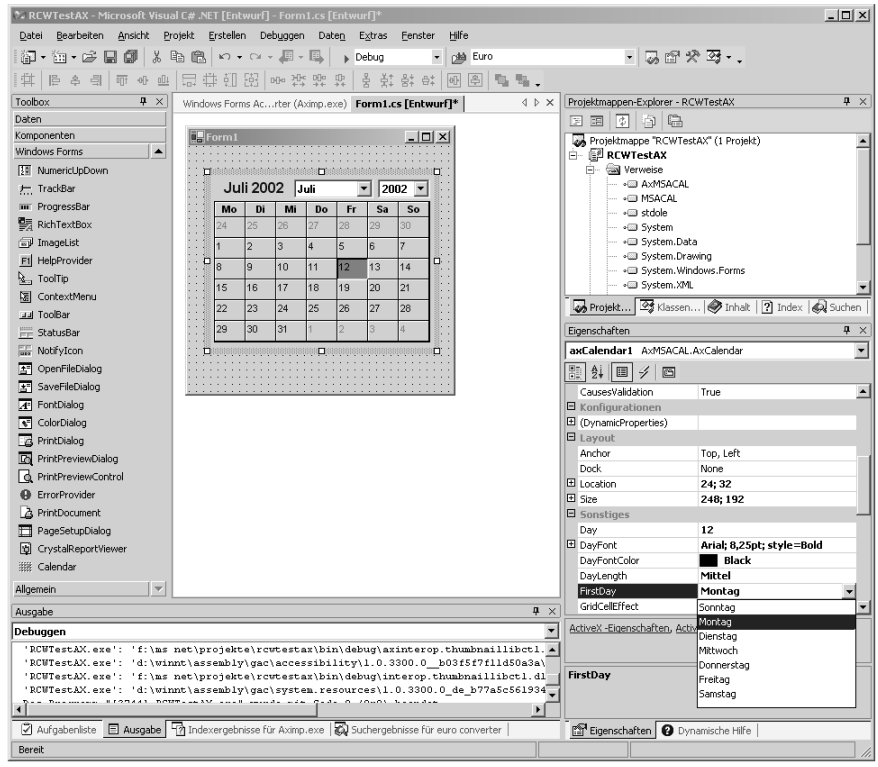


Abbildung 25.7: Ein importiertes ActiveX-Steuerelement im Designer

Das Hilfsprogramm AXImp

Für den Import von ActiveX-Steuerelementen gilt Ähnliches wie beim Anlegen von Hüllklassen ohne visuelle Komponente: VS.NET verwendet ein externes Hilfsprogramm dafür und arbeitet mit dessen Standardvorgaben.

Das für ActiveX-Steuerlemente zuständige Konsolenprogramm heißt *AXImp.exe* (für »Active X Importer«). Es bietet über verschiedene Schalter analog zu *TlbImp.exe* die Möglichkeit, Namensräume, DLL-Namen usw. festzulegen.



AXImp.exe wird bei einer Standardinstallation von VS.NET in Programme\Microsoft Visual Studio .NET\bin untergebracht und benötigt seinerseits die Datei aximp.resources.DLL, die bei der Installation auf einem deutschen Windows im Unterordner ….NET\bin\DE landet.

Es empfiehlt sich deshalb hier noch mehr als bei TlbImp, das zum Start benötigte Konsolenfenster nicht direkt, sondern über START\PROGRAMME\MICROSOFT VISUAL STUDIO .NET\VISUAL STUDIO .NET TOOLS\VISUAL STUDIO *.NET Eingabeaufforderung zu öffnen, weil diese Verknüpfung für das Einsetzen der entsprechenden Suchpfade sorgt.*



Der Schalter /source von AXImp.exe mag die Hoffnung erwecken, hier werde eine C#-gemäße Deklaration für die COM-Stellvertreterklasse generiert. Leider ist dem nicht so: AXImp.exe beschränkt sich beim Erzeugen von Listings auf die von `Control` *abgeleitete Klasse. Da aber auch dieses Listing recht aufschlussreich sein kann, wird es einem der folgenden Abschnitte kurz besprochen.*

Codebeispiel – Microsoft Calendar-Steuerelement

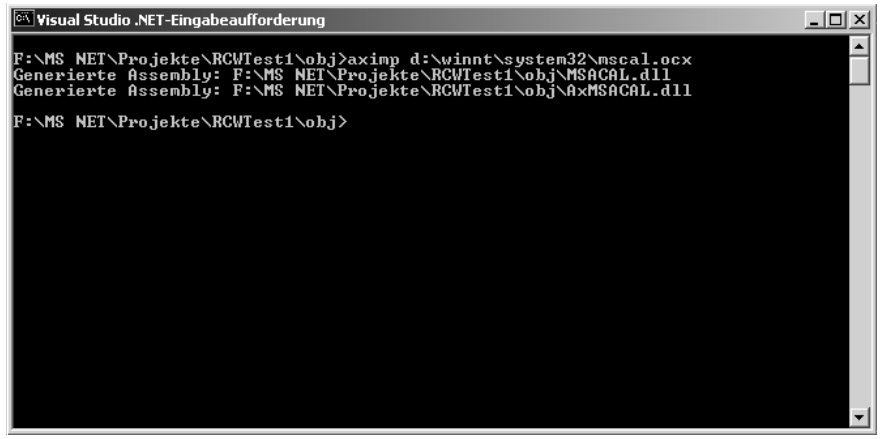
Das Beispielprojekt *MSACALImport* demonstriert im Wesentlichen die Projektstruktur für den Einsatz eines ActiveX-Steuerelements. Die Kunst liegt hier, wie beim Kochen, eigentlich nur in der Vorbereitung der Zutaten.

Abbildung 25.8 zeigt einen Aufruf von *AXImp.exe* mit den Standardvorgaben des Programms für das Kalender-Steuerelement, das als Bestandteil von Microsoft Office installiert wird. Das Ergebnis sind zwei DLLs: *MSACAL.dll* mit der COM-Hüllklasse, *AxMSCAL.dll* mit der Verkapselung als Ableitung von `Control`.

Wie zuvor erwähnt, sollte das verwendete Konsolenfenster über die Verknüpfung *Visual Studio .NET Eingabeaufforderung* geöffnet werden.

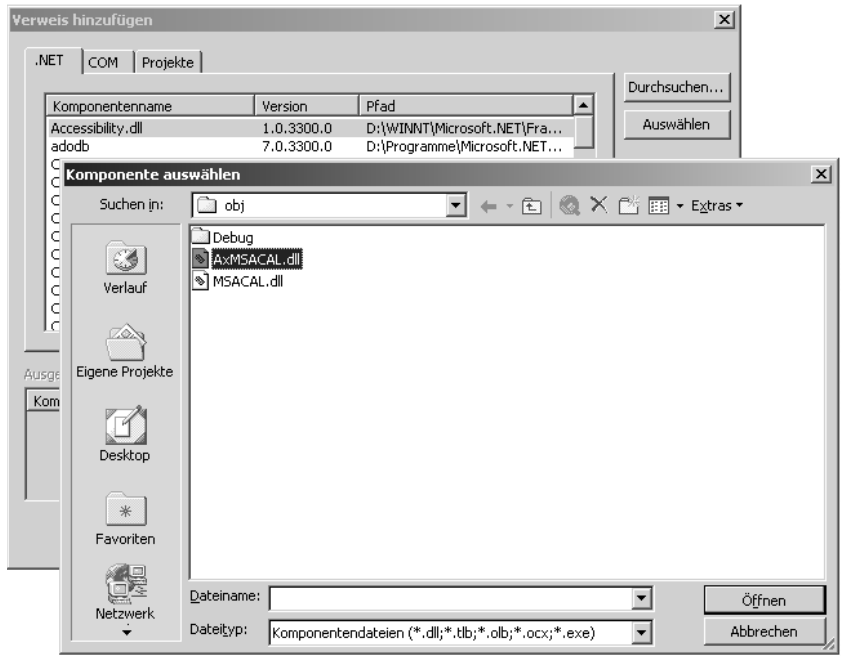
Der nächste notwendige Schritt besteht aus dem Anlegen von Projektverweisen auf die beiden DLLs. Technisch gesehen ist ein Verweis auf *AxMSACAL.dll* ausreichend – es ist aber sicher nicht schlecht, wenn in der Abteilung VERWEISE der Projektmappe *alle* selbsterzeugten DLLs erscheinen, die das Programm voraussetzt.

Abbildung 25.8: Aufruf von AXImport über die Kommandozeile ohne zusätzliche Optionen



Wenn Sie das ActiveX-Steuerelement über die TOOLBOX zugänglich machen wollen, verwenden Sie wie zuvor beschrieben das Dialogfeld TOOLBOX ANPASSEN und dort die Schaltfläche DURCHSUCHEN. Geht es lediglich um die Namensräume und Referenzen, dann tut es auch PROJEKT/VERWEIS HINZUFÜGEN und dort dann ebenfalls DURCHSUCHEN – siehe Abbildung 25.9.

Abbildung 25.9: Manueller Nachtrag der Projektverweise



Das »manuelle« Anlegen einer Instanz dieses Steuerelements kann dann beispielsweise so aussehen:

```
private void button1_Click(object sender, System.EventArgs e)
{
  AxMSACAL.AxCalendar MSCal;

  // neues Schnittstellenobjekt
  MSCal = new AxMSACAL.AxCalendar();

  // übergeordnetes Fenster: das Formular
  MSCal.Parent = this;
  // Position und Größe
  MSCal.Size = new System.Drawing.Size(300,200);
  MSCal.Location = new System.Drawing.Point(0, 60);
  // Eintragen in der Liste
  this.Controls.Add(MSCal);

  MSCal.Today();  // aktuelles Kalenderdatum setzen

  button1.Enabled = false; // ein Kalender ist genug

}
```

Das Ergebnis zeigt Abbildung 25.10.

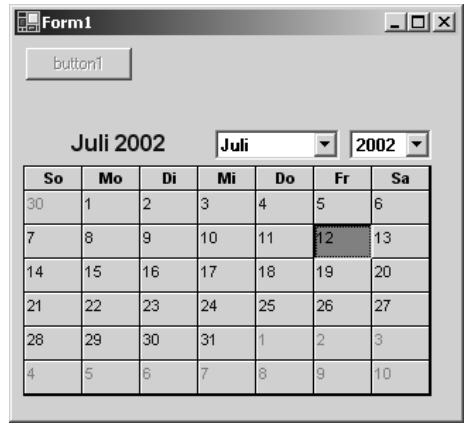


Abbildung 25.10: Das zur Laufzeit angelegte Kalender-Steuerelement in einem Formular

Listings mit dem Schalter /source von AXImp

Wer sich die von `Control` abgeleitete Klasse zur Verkapselung eines solchen Steuerelements genauer ansehen will, weist *AXImp.exe* über den Schalter `/source` an, einen C#-Quelltext mit der Klassendeklaration zu erzeugen. Wie bereits beklagt, geht es dabei allerdings nicht um die Deklaration des eingebetteten Steuerelements, sondern ausschließlich um die Ableitung von `Control`, wie das folgende Fragment zeigt:

```
//------------------------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.0.3705.209
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
//------------------------------------------------------------------

[assembly: System.Reflection.AssemblyVersion("7.0.0.0")]
[assembly: System.Windows.Forms.AxHost.TypeLibraryTimeStamp(
       "26.06.1998 00:00:00")]

namespace AxMSACAL
{
  [System.Windows.Forms.AxHost.ClsidAttribute(
     "{8e27c92b-1264-101c-8a2f-040224009c02}")]
  [System.ComponentModel.DesignTimeVisibleAttribute(true)]
  [System.ComponentModel.DefaultEvent("AfterUpdate")]
  [System.ComponentModel.DefaultProperty("_Value")]

  public class AxCalendar : System.Windows.Forms.AxHost
  {
    private MSACAL.ICalendar ocx;
    private AxCalendarEventMulticaster eventMulticaster;
    private System.Windows.Forms.AxHost.ConnectionPointCookie cookie;
    public AxCalendar() :
      base("8e27c92b-1264-101c-8a2f-040224009c02")
      {
        this.SetAboutBoxDelegate(new AboutBoxDelegate(AboutBox));
      }
...
```

25.2.5 Untersuchungen mit ILDASM

Wenn die Methoden und Eigenschaften einer COM-Klasse oder eines ActiveX-Steuerelements nicht an anderer Stelle dokumentiert sind, ist das Ausprobieren über die kontextbezogene Syntaxhilfe von VS.NET doch etwas mühselig – und wie zuvor gezeigt, beschränkt sich die Quelltext-Erzeugung von *AXImp.exe* auf die Ableitung von `Control`, weshalb *TlbImp.exe* konsequent ein äquivalenter Schalter komplett fehlt.

Sowohl bei fensterlosen COM-Klassen als auch bei ActiveX-Steuerelementen hilft ILDASM weiter. Dieses (endlich einmal nicht auf der Konsole basierende) Dienstprogramm ist bei einer Standardinstallation von VS.NET in *Programme\Microsoft Visual Studio .NET\FrameworkSDK\Bin* untergebracht, irritierenderweise in einer Standardinstallation aber weder über das START-Menü noch über das Menü EXTRAS von VS.NET erreichbar. Angesichts der Fülle von Informationen, die ILDASM aus dem Manifest einer .NET-DLL herausholt (vgl. Abbildung 25.11) empfiehlt es sich eigentlich, gleich beide Menüs um entsprechende Wahlpunkte zu bereichern.



Abbildung 25.11: ILDASM zeigt auch bei ActiveX-Steuerelementen die Methoden und Eigenschaften der Basisklasse

Vorausgesetzt wird dabei natürlich die Auswahl der entsprechenden DLL – im gegebenen Beispiel also nicht *Interop.AxMSACAL.DLL*, sondern *Interop.MSACAL.DLL*.

25.2.6 COM ohne Stellvertreterklassen und RCWs

Nostalgiker, die – je nach Provenienz – wehmütig an die Konvertierung von COM-Schnittstellenzeigern über Präprozessor-Makros mit mehreren Ebenen (C++) oder explizite Typumwandlungen in Integer bei Zuweisungen (Delphi) denken, lässt C# zwar recht bewusst mit ihren Sehnsüchten allein. Dennoch bietet der Namensraum `Runtime.InteropServices` die Möglichkeit zur direkten Anforderung einer COM-Schnittstelleninstanz; die Methode `InvokeMember()` der Klasse `Type` erlaubt den Aufruf benannter Methoden.

```
using System.Runtime.InteropServices;
// ...
private void button2_Click(object sender, System.EventArgs e)
{ Type PhotoshopAppType;
  PhotoshopAppType = Type.GetTypeFromProgID("Photoshop.Application.7");
```

```
object PS, PSDoc;
PS = Activator.CreateInstance(PhotoshopAppType);

object[] Arguments = { "C:\\Test.jpg"} ;
PSDoc = PhotoshopAppType.InvokeMember("Open",
   System.Reflection.BindingFlags.InvokeMethod,
   null, PS, Arguments);
// ...
```

Obwohl diese Technik RCW-DLLs und Stellvertreterklassen genauso einspart wie das dazu notwendige Setup, wird man sie wohl bestenfalls als kurzfristigen Notnagel verwenden. Da Aufrufe über die Namen von Methoden geschehen, muss die COM-Schnittstelle von `IDispatch` abgeleitet sein – und Fehler jeder Art (unpassende Parameter, falsch geschriebene bzw. nicht vorhandene Methoden) werden erst zur Laufzeit offensichtlich.

25.3 .NET-Komponenten in COM

Wer einer neuen Technologie zum Durchbruch auf breiter Front verhelfen will, sorgt im Allgemeinen nach Kräften dafür, dass sie in irgendeiner Weise abwärtskompatibel ist, schafft also einen möglichst einfachen Weg zur Übernahme bereits existierender Daten, Komponenten usw. Das Ergebnis ist üblicherweise eine Einbahnstraße, die ausschließlich vom Alten zum Neuen führt, aber eben nicht andersherum – und das ist aus geschäftspolitischer Sicht ja auch durchaus gewünscht. Wieso Microsoft dann eine Menge Energie darin investiert hat, .NET-Komponenten auch von COM-Anwendungen aus nutzbar zu machen? Die einzig denkbare Antwort darauf: Weil man es in Redmond mit .NET wirklich ernst meint. Das mag sich zunächst paradox anhören, ist es aber nicht: Eine wirklich vorteilhafte Technologie setzt sich nicht über Marktbeherrschung und absichtlich verbaute Rückwege, sondern aufgrund ihrer inneren Werte durch. (Diesen Satz können sich einige andere Hersteller gerne mal hinter den Spiegel schreiben.)

Die eigentliche Motivation von Microsoft sieht natürlich trotzdem anders aus: Wenn sich .NET-Komponenten problemlos in das Gerüst von COM einsetzen lassen, dann ist die Hemmschwelle zum Einsatz der neuen Technik wesentlich niedriger, weil die Migration dann auch Schritt für Schritt geschehen kann.

25.3.1 Voraussetzungen

Was sich von einer .NET-Klasse per COM-Schnittstelle einsetzen lässt, ist natürlich auf die Möglichkeiten von COM begrenzt. Allzu schmerzhaft sind die Einschränkungen allerdings nicht.

- Konstruktoren mit Parametern sind nicht verwendbar, weil COM ausschließlich die parameterlose Variante kennt.
- Statische Elemente sind nicht über die COM-Schnittstelle aufrufbar (aus der Implementation heraus dagegen schon).
- Konstanten teilen das Schicksal statischer Elemente: Sie stehen den Methoden der .NET-Klasse zur Verfügung, sind aber nicht direkt über die COM-Schnittstelle abfragbar, sondern nur über den Umweg einer (nicht statischen) Zugriffsmethode.
- Der Aufruf überladener Methoden ist möglich, wenn auch nur mit ein wenig Nachhilfe auf Seiten des Programmierers.

Die weiteren Voraussetzungen sind ein COM-gerechter Eintrag in der Registrierung und die Installation im globalen Cache der .NET-Laufzeitumgebung (was seinerseits wiederum einen starken Namen voraussetzt).

25.3.2 Codebeispiel – .NET-DLL als COM-Komponente

Da an einer Demonstration mindestens vier Programme (nämlich zwei Compiler und zwei Utilities) beteiligt sind, über die sich recht viel schreiben ließe, verzichtet dieser Abschnitt komplett auf eine theoretische Einleitung. Um das Beispielprojekt *Net_for_COMDemo* nachzustellen, legen Sie ein neues Projekt in VS.NET mit dem Vorlagentyp KLASSENBIBLIOTHEK an.

Bei der hier zu Demonstrationszwecken vorgestellten Klasse ist der Name fast genauso lang wie der gesamte Quelltext:

```
using System;

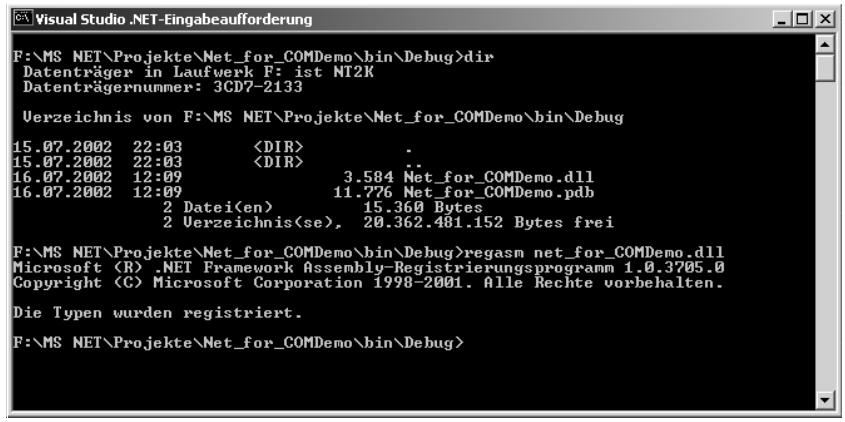
namespace Net_for_COMDemo
{
  public class Euro
  {
    public static double Factor = 1.955;

    public Euro() { }
    public double GetFactor() { return Factor; }
    public double CurVal;
    public double GetDM() { return CurVal * Factor; }
  }
}
```

Registrierung mit RegAsm

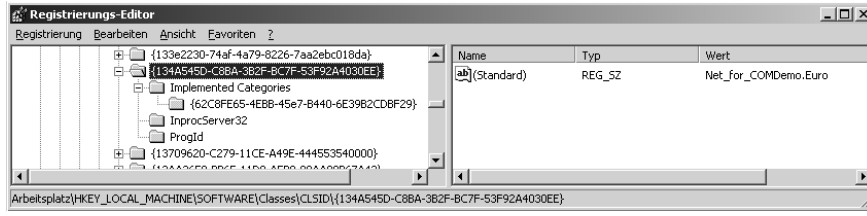
Die kompilierte DLL wird mit dem Utility *RegAsm* (»Register Assembly«) registriert. Dazu ist wieder einmal ein Konsolenfenster notwendig, das über die Verknüpfung Visual Studio .NET-Eingabeaufforderung gestartet wurde – siehe Abbildung 25.12.

Abbildung 25.12: Aufruf von RegAsm zum Eintrag eines Assemblies als COM-Schnittstelle



Mit *Regedit* lässt sich nachprüfen, dass *RegAsm* nun für die Klasse `Net_For_COMDemo.Euro` einen COM-gemäßen Schlüssel in die Registrierung eingetragen hat (der auf Ihrem System mit Sicherheit unter einem anderen GUID läuft) – siehe Abbildung 25.13.

Abbildung 25.13: Der von RegAsm erzeugte Teilschlüssel in Classes\CLSID



INFO

RegAsm wird zusammen mit dem .NET-Laufzeitsystem installiert, ist also auch auf Client-Systemen verfügbar und muss dort als Teil der Installation einer als COM-Komponente verpackten Assembly aufgerufen werden.

Wenn Sie nicht explizit dafür sorgen, enthält eine Assembly keine GUIDs für die jeweiligen Klassen, weshalb RegAsm die GUID-Generierung bei Bedarf in Eigenregie nachholt – mit der Folge, dass eine Klasse wie `Euro` *auf jedem Client-System einen anderen GUID bekommt.*

Das stellt zwar nicht unbedingt ein Problem dar, weil reines Scripting – beispielsweise mit VBS – über Klassennamen arbeitet, und eine für frühe Bindung verwendete .tlb-Datei selbstverständlich auch einen (dann auf allen Systemen gleichen) GUID enthält. Schön ist es trotzdem nicht.

Abhilfe schafft ein im Quelltext der .NET-Klasse explizit angegebener GUID:

```
namespace Net_for_COMDemo
{
  [Guid("CAD5F914-C8CB-4657-B89A-2D1D9342D963")]
  public class Euro
  {
    public static double Factor = 1.955;
```

GUIDs lassen sich übrigens in VS.NET über den Menübefehl EXTRAS/GUID ERSTELLEN erzeugen (hinter dem das externe Programm `guidgen.exe` *steckt). Im von Guidgen dargestellten Dialogfeld wählen Sie dazu das Format 4 (»Registry«); mit einem Klick auf die Schaltfläche COPY wird der neue GUID in die Zwischenablage kopiert und kann von dort aus in den Quelltext eingefügt werden.*

Wie Abbildung 25.14 zeigt, ist als COM-Server für diese Klasse allerdings nicht die DLL selbst eingetragen, sondern *mscoree.dll*, und anstelle eines Pfades wie *F:\MS NET\Projekte…* finden sich hier Versionsangaben. Damit die .NET-Laufzeitumgebung die neue Klasse anderen Anwendungen als COM-Schnittstelle zur Verfügung stellen kann, muss *Net_For_COMDemo* im Global Assymbly Cache installiert werden, was einen starken Namen voraussetzt. Dafür ist sind mehrere Wechsel zwischen VS.NET und dem Konsolenfenster notwendig (wobei man durchaus fragen darf, wieso sich das eigentlich nicht über ein kleines Dialogfeld in VS.NET abwickeln ließe – und zwar in einem Rutsch).

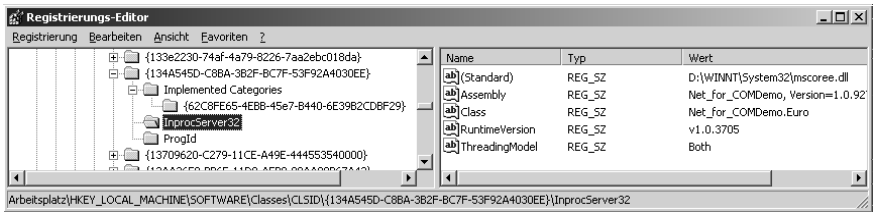


Abbildung 25.14: Für COM-Schnittstellen von .NET-Komponenten verwendet die Laufzeitumgebung eine zentrale Server-DLL

Zuordnung eines starken Namens

Wie bereits erwähnt, stellt ein starker Name die Voraussetzung für den Eintrag in den globalen Cache dar (und auf nichts anderes als diesen Cache bezieht sich die COM-Server-DLL mscoree). Zum Anlegen der Schlüsselpaardatei wechseln Sie im Konsolenfenster in den Unterordner *obj* des Projekts:

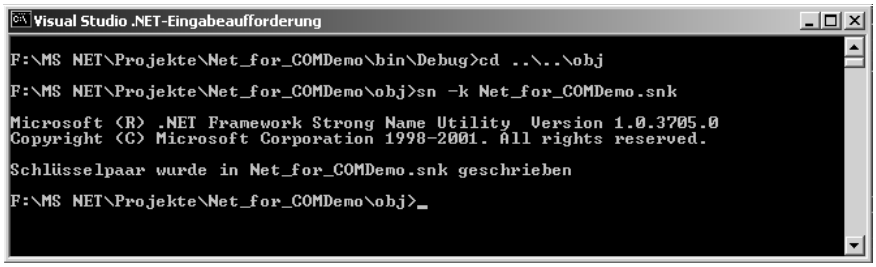


Abbildung 25.15: Anlegen des Schlüsselpaars mit dem Kommandozeilenprogramm sn.exe

Im Editor von VS.NET laden Sie die zum Projekt gehörige Datei *AssemblyInfo.cs* und tragen den Namen der Schlüsseldatei dort ein. Der entsprechende Abschnitt ist kurz vor dem Ende der Datei und enthält als Standardvorgabe einen Leerstring:

```
// (*) Das verzögern der Signierung ist eine erweiterte Option. Weitere
//  Informationen finden Sie in der Microsoft .NET Framework-Dokumentation.
//
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("..\\Net_for_COMDemo.snk")]
[assembly: AssemblyKeyName("")]
```

Ohne das vorangestellte ..\\ würde VS.NET die SNK-Datei im Unterordner *obj\Debug* bzw. *obj\Release* suchen.

Installation im globalen Cache

Nach einer erneuten Kompilierung mit VS.NET, bindet der Compiler nun auch den starken Namen in die DLL ein (und sollte dafür mit ERSTELLEN/NEU ERSTELLEN gestartet werden), besteht der letzte Schritt schließlich aus der Installation im globalen Cache. Dazu können Sie entweder erneut im Konsolenfenster nach *Bin\Debug* wechseln und dort das Utility *gacutil* (mit */i Net_for_COMDemo.dll*) aufrufen oder schlicht im Explorer die Datei *Net_for_COMDemo.dll* in den globalen Cache von .NET ziehen, der standardmäßig den Ordner *\WINNT\assembly* belegt. Wie Abbildung 25.16 zeigt, wird dieser Ordner dank einer zusammen mit .NET installierten Shell-Erweiterung (*shfusion.dll*) auf spezielle Weise dargestellt.

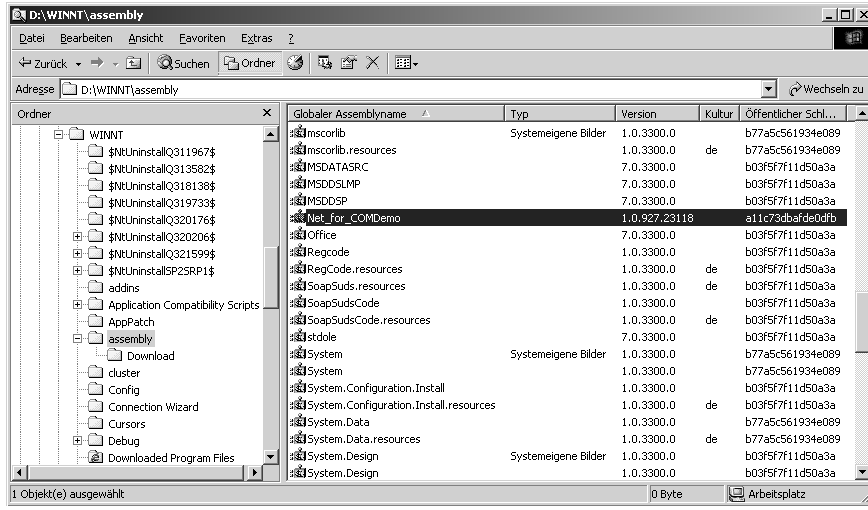


Abbildung 25.16: Der globale Cache von .NET in der (durch shfusion.dll erweiterten) Darstellung des Explorers



Da der Cache ausschließlich Verweise enthält, läuft das Eintragen von Assemblies aus anderen Ordnern über Ziehen und Ablegen – die Kombination Strg+C / Strg+V *ist hier gesperrt.*

Späte Bindung

Zum ersten Ausprobieren bietet sich VB Script an – nicht nur, weil dort grundsätzlich mit später Bindung gearbeitet wird, sondern auch, weil dieser

Compiler auf jedem System verfügbar ist. Tippen Sie also das folgende Prögrämmchen in den Windows-Editor ein und speichern Sie es unter dem Namen *NETComTest.vbs*:

```
Option Explicit
Dim objEuro, DMResult, EuroValue

EuroValue = InputBox("Betrag in Euro?")

Set objEuro = CreateObject("Net_for_COMDemo.Euro")
objEuro.CurVal = EuroValue
DMResult = objEuro.GetDM
Call MsgBox(DMResult)
```

`CreateObject()` fordert eine COM-Schnittstelleninstanz für die Klasse an. Wie die nächsten beiden Zeilen demonstrieren, sind öffentliche Eigenschaften (`CurVal`) und Methoden (`GetDM`) des Euro-Objekts in der für COM-Objekte gewohnten Weise zu erreichen

Frühe Bindung

Mit dem Programm *TlbExp* als Gegenstück zu *TlbImp* lässt sich aus einem Assembly eine *.tlb*-Datei erzeugen, die sich zumindest theoretisch in keinem Punkt von einer *.tlb*-Datei für eine »echte« COM-Schnittstelle unterscheidet. Vorausgesetzt wird dabei die explizite Vergabe des Attributs `ClassInterfaceType.AutoDual` – ansonsten ist das Ergebnis eine weitgehend leere IDL-Beschreibung.

```
using System;

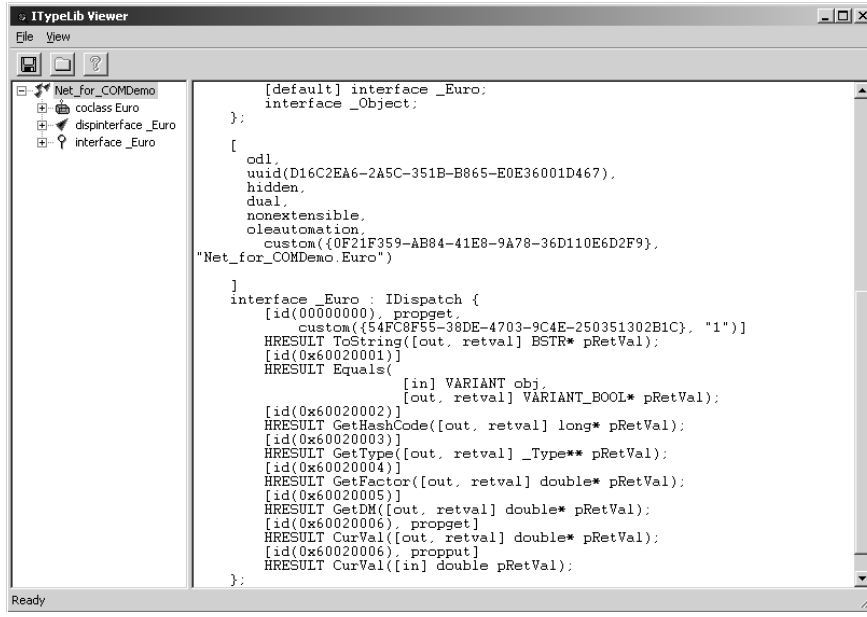
namespace Net_for_COMDemo
{
  [ClassInterface(ClassInterfaceType.AutoDual)]
  [Guid("CAD5F914-C8CB-4657-B89A-2D1D9342D963")]
  public class Euro
  {
    public static double Factor = 1.955;

    public Euro() { }
    public double GetFactor() { return Factor; }
    public double CurVal;
    public double GetDM() { return CurVal * Factor; }
  }
}
```

Nach entsprechender Kompilierung, erneuter Registrierung mit `RegAsm` und dem Aufruf von *Tlbexp* für *Net_for_COMDemo.dll* lässt sich im OLE Viewer von Visual Studio 6 sehen, was die dabei erzeugte Datei *Net_for_-COMDemo.tlb* an Informationen enthält. (Zum Start dieses Tools wählen Sie den Befehl EXTRAS/OLE/COM-OBJEKTKATALOG.) Suchen Sie im ersten Schritt den Eintrag *Net_for_COMDemo* für die Datei im Teilbaum *TypeLi-*

braries. Ein Doppelklick darauf liefert eine Detailansicht in dem (modalen) Fenster ITYPELIB VIEWER. Wie Abbildung 25.17 zeigt, erscheint das öffentliche Datenfeld `CurVal` als (`get`/`set`) Eigenschaft, auch die Methoden `GetDM()` und `GetFactor()` sind mit von der Partie – und was als statisch deklariert wurde, hat trotz des Modifizierers `public` keinen Weg in die *.tlb*-Datei gefunden.

Abbildung 25.17: Die TLB-Datei im Klartext





Registrierung und Erzeugung der .tlb-Datei lassen sich alternativ auch in einem einzigen Aufruf von RegAsm erledigen. Der entsprechende Schalter heißt /tlb, die Angabe eines .tlb-Dateinamens ist optional.

```
> regasm Net_For_COMDemo.dll /tlb
```

Bitte beachten: Die Wahl zwischen RegAsm und TlbExp haben Sie nur auf dem Entwicklungssystem. Für die Registrierung einer als COM verpackten .NET-Komponente auf dem Zielsystem ist in jedem Fall RegAsm erforderlich.

Eine praktische Erprobung mit Borland Delphi scheitert im Moment allerdings daran, dass die zusammen mit .NET installierte Typenbibliothek *mscorlib.tlb* für die Datentypen Byte, Single und Double Deklarationen enthält, die der Delphi-Compiler als fehlerhaft moniert.

```
Double = packed record
  // <- Fehlermeldung: Typ Double noch nicht vollständig definiert
  m_value: Double;
end;
```

Da sich dieses Problem durch den gesamten Quelltext von immerhin gut 10000 Zeilen zieht, sind die restlichen Überlegungen an dieser Stelle vorläufig eher theoretischer Natur.

- Delphi generiert für die COM-Schnittstelle seinerseits eine Stellvertreterklasse und folgt dabei den Borland'schen Namenskonventionen. Dass dabei eine Klasse *TEuro* herauskommt – honny soit qui mal y pense.
- Objekte dieser Klasse lassen sich dank der frühen Bindung wie normale VCL-Objekte einsetzen – beispielsweise:

```
procedure TForm1.Button1Click(Sender: TObject);
var EuroVal, DMResult: Double;
begin
  EuroVal := StrToFloat(Edit1.Text);
  EuroCalc.CurVal := EuroVal;
  DMResult := EuroCalc.GetDM;
  ShowMessage(FloatToStr(DMResult));
end;
```



Das Problem ist natürlich inzwischen auch Borland bekannt, und dort hat wenigstens einer der Redner für die australische BorCon reagiert: Glenn Stephens stellt mit einem Assistenten für Delphi zwar keine abschließende Lösung, aber eine Umgehung vor. Dieser Assistent ist sowohl über seine eigene Website (`www.glennstephens.com.au/DotNetWizards.zip`*) als auch über* `www.borland.com` *(»Search«: .NET assembly Import and Export Wizards for Delphi) kostenlos als Quelltext herunterladbar. Weitere und bessere Lösungen werden aber sicher nicht lange auf sich warten lassen.*

Nach ein klein wenig Nachbesserei (Verzeichnis ToolsAPI per Projektoptionen in den Suchweg für Dateien aufnehmen, Delphi-Versionsnummer in der Funktion DotNetDirectoryLocations.GetDelphiPath() für die Suche in der Registrierung von 5.0 auf 6.0 aktualisieren) lässt sich dieser Assistent auch installieren und ausführen. Ärgerlicherweise zeigt sich dann, dass Delphi 6 in diesem Punkt eben nicht Delphi 5 entspricht: Der Compiler bleibt wiederum in MSCorlib_TLB mit einem Syntaxfehler hängen. Es mag sein, dass dieses Tool Benutzern von Delphi 5 weiterhilft – darum dieser Hinweis.

25.3.3 Grenzen

Obwohl eine als COM-Schnittstelle verpackte .NET-Komponente einen nicht unbeträchtlichen Überbau mit sich herumschleppt (auf den Clients wie Delphi dann noch einmal eine weitere Schicht in Form einer Stellvertreterklasse draufpacken), ließ sich mit einigen Experimenten zeigen, dass die Verluste in puncto Laufzeit im erträglichen Rahmen bleiben – vor allem, weil das doppelte und dreifache Umschichten von Parametern gegenüber dem Zeitbedarf von COM selbst praktisch nicht ins Gewicht fällt.

Der wesentliche Punkt: Eine als COM getarnte .NET-Komponente ist kein »besseres COM« – sondern eben ein Zwitter, der sowohl auf der .NET- als auch auf der COM-Seite Installation und Wartung verlangt, und mit sämtlichen Problemen behaftet ist, die mit .NET eigentlich abgeschafft werden sollten. Das betrifft sowohl die nicht mögliche Koexistenz verschiedener Versionen (es sei denn, man schafft jeweils eigene COM-Schnittstellen dafür) als auch die Rechtevergabe und die Installation.



Falls Sie spaßeshalber (etwa für Laufzeitmessungen) den Reimport einer von .NET aus exportierten COM-Komponente versuchen, bleibt es bei einer Fehlermeldung.



Abbildung 25.18: So leicht lassen sich die Hüllklassen dann doch nicht hereinlegen.

26 Legacy Code und Windows API

Da sich .NET der Philosophie des Unternehmens Microsoft gemäß als ganzheitlicher (böse Zungen würden vermutlich sagen: alles erschlagender) Ansatz versteht, wird man eigentlich bei jeder Problemstellung über kurz oder lang in der .NET-Klassenbibliothek fündig. Der typische Fall, in dem man eben doch die geordneten Verhältnisse der .NET-Umgebung verlassen und auf eigentlich Überholtes zurückgreifen wird, ist deshalb weniger bei Aufrufen der Windows-API zu suchen, als beim Einbinden von DLLs, die es – egal aus welchem Grund – im Moment nur in dieser Form gibt, und die dem erklärten Willen von Microsoft zufolge dennoch kein unüberwindliches Hindernis auf dem Weg zu .NET sein sollen.

Ein anderer, nicht minder arger Stolperstein auf dem Weg zu .NET wäre es, wenn früher einmal in C erstellter Code grundsätzlich komplett neu geschrieben werden müsste, weil dort nun mal unvermeidlich mit Zeigern hantiert wird.

Dieses Kapitel beschreibt die Unterstützung, die C# im Umgang mit DLLs und Erbschaften aus früheren Zeiten zur Verfügung stellt. Wie Sie zu Recht erwarten können, geht es dabei weniger um eine Wiederholung des Materials in der Online-Dokumentation als um die Dinge, die man dort nicht findet – oder nur in versteckter oder missverständlicher Form.

26.1 DLL-Aufrufe

Obwohl es im Regelfall weniger um die Windows-API als eher um DLLs geht, die Sie irgendwann einmal erworben haben (oder um das firmeneigene Buchhaltungsmodul, in dem x Jahre C/C++-Entwicklung stecken), laufen DLL-Aufrufe bei .NET unter der Rubrik Platform Invocation Services, was sich in etwa mit »plattformspezifische Aufrufe« übersetzen lässt (und angesichts der Tatsache, dass DLLs ein Windows-Spezifikum darstellen, ja auch durchaus seine Richtigkeit hat). Das allgemein gebräuchliche Kürzel für solche Aufrufe ist PInvoke.

Ausnahmen bestätigen natürlich auch hier die Regel: Die in einem der folgenden Abschnitte vorgestellte Funktion zum Herunterfahren von Windows dürfte schon rein deshalb nicht direkt in .NET enthalten sein, weil sie sozusagen das Spielfeld sprengt.

26.1.1 Deklaration externer Routinen

Obwohl eine von einer DLL exportierte Routine wohl kaum den Kontext einer C#-Klasse voraussetzt, sind die C#-Entwickler ihrer Philosophie treu geblieben: Das deklarative Drumherum eines Imports findet wie auch alles andere im Rahmen einer Klassendeklaration statt; eine auf diese Weise deklarierte externe Routine erscheint dem restlichen Programm gegenüber als Methode der jeweiligen Klasse. Im einfachsten Fall:

```
class MyClass
{
  // ... <reguläre C#-Methoden>
  [DllImport("DLLName.dll")]
  Modifizierer static extern Ergebnistyp Funktionsname([Parameter]);
     ...
```

Als Modifizierer sind hier ausschließlich `public` und `internal` erlaubt.

Mit diesen syntaktischen Mitteln lässt sich bereits ein einfaches Beispielprogramm schreiben.

```
[STAThread]
static void Main()
{
  Application.Run(new Form1());
}

[System.Runtime.InteropServices.DllImport("user32.dll")]
public static extern bool MessageBeep(uint soundtype);

private void button1_Click(object sender, System.EventArgs e)
{
  MessageBeep(0);
}
```

In einer realen Anwendung nimmt man natürlich den Namensraum `System.Runtime.Interopservices` per `using` in den Quelltext auf. (Und in einer noch realeren Anwendung verwendet man `MessageBox.Show()`.)

DllImportAttribute

Die bei `DllImport` verwendbaren Parameter gehen in den Konstruktor eines `DllImportAttribute`-Objekts ein. Die vollständige Form sieht inklusive der Standardvorgaben so aus:

```
[DllImport("DllName", EntryPoint="Funktionsname", ExactSpelling=false,
SetLastError=false, CharSet=CharSet.Auto,
CallingConvention=CallingConvention.WinApi)]
```

Einige Anmerkungen zu den einzelnen Parametern, die so nicht in der – bei diesem Thema ansonsten recht reichhaltigen – Online-Dokumentation zu finden sind, gibt Tabelle 26.1.

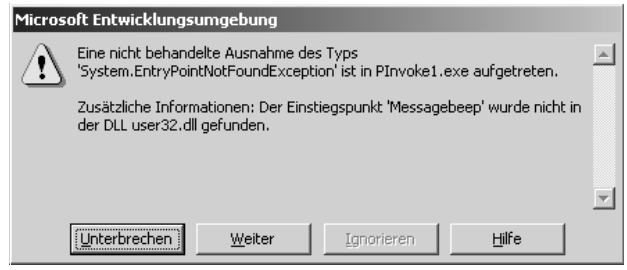
Tabelle 26.1: Parameter von `DllImport-Attribute`

Parameter	Bedeutung
DllName	Legt den Namen der Datei fest, in der die Routine gesucht wird und muss an erster Stelle erscheinen. Die Angabe der Dateiendung ist obligatorisch – mit »user32« anstelle von »user32.DLL« kann die Laufzeitumgebung nichts anfangen. Die Angabe eines Suchpfades ist zulässig und optional – wenn er fehlt, verwendet das System die Standard-Suchreihenfolge (Startverzeichnis der Anwendung, aktuelles Verzeichnis, Windows-Verzeichnis, System bzw. System32 usw.).

EntryPoint	Gibt den Namen der exportierten Routine an, ist optional, und kann wie alle weiteren in diesem Abschnitt besprochenen Parameter an beliebiger Stelle außer der ersten erscheinen. Wenn dieser Parameter fehlt, entnimmt der Compiler den Routinennamen der `extern`-Deklaration, die in jedem Fall unmittelbar auf `DllImport` folgen muss. Die beiden folgenden Deklarationen sind äquivalent:
	`[DllImport("xyz.dll", EntryPoint="MyFuncW")`
	`public static extern void MyFunc();`
	
	`[DllImport("xyz.dll")]`
	`public static extern void MyFuncW();`
	
	Im ersten Fall wird die externe Routine im restlichen Programm als `MyFunc()` angesprochen, im zweiten Fall mit ihrem »echten« Namen `MyFuncW()`. Wie dieses Beispiel bereits andeutet, sind derartige Umbenennungen vor allem dann nützlich, wenn es um die Unterscheidung zwischen ANSI- und Unicode-Versionen von API-Routinen geht, die gegenüber dem restlichen Programm transparent sein soll.
	
	Notwendig ist die explizite Unterscheidung zwischen den A- und W-Varianten von API-Funktionen nur in Ausnahmefällen, weil alle Windows-Versionen hier mit Aliasnamen arbeiten: `MessageBox` führt unter Windows 2000 beispielsweise zum Aufruf der Funktion, die dort auch unter `MessageBoxW` erreichbar ist, unter Windows 98 entspricht `MessageBox` dagegen `MessageBoxA`.

Tabelle 26.1: Parameter von `DllImport-Attribute` (Forts.)

Parameter	Bedeutung

ExactSpelling	Sollte der Dokumentation zufolge dafür sorgen, dass die .NET-Laufzeitumgebung und Windows nicht zwischen Groß- und Kleinschreibung bei den Funktionsnamen unterscheiden. In der aktuellen Version von C# scheint dieser Parameter allerdings wirkungslos zu bleiben. Die Rechnung für eine Deklaration wie
	`[DllImport("user32.dll", ExactSpelling=false)]`
	`public static extern bool Messagebeep(uint soundtype);`
	
	`private void bMessageBeep_Click(object sender, EventArgs e)`
	`{`
	`  Messagebeep(0);  // <- anstelle von MessageBeep`
	`}`
	
	bekommt man unerfreulicherweise erst zur Laufzeit präsentiert.
	 

SetLastError	Erlaubt die Festlegung, ob ein von der exportierten Routine eventuell in der Systemvariablen `LastError` hinterlassener Statuscode im weiteren berücksichtigt werden soll oder nicht. Die Logik eines Aufrufs sieht offensichtlich so aus:
	`// Aufruf einer exportierten Routine`
	`if (SetLastError) WinAPISetLastError(0);`
	`  CallDLLRoutine(...);`
	`if (SetLastError)`
	`  CopyWinAPILastErrorToNET();`
	
	Ein auf diese Weise festgehaltener Stand von `LastError` lässt sich dann über die Methode `Marshal.GetLastWin32Error` abfragen. Anders gesagt: Mit dieser Methode lässt sich der System-Fehlerstatus der jeweils letzten per PInvoke aufgerufenen Routine ermitteln, bei der *SetLastError* auf *true* gesetzt war. Dies wird deshalb derart ausführlich erwähnt, weil sich die deutschen Übersetzer mit dem Text »Der `SetLastError`-Parameter gibt an, ob die Methode den zuletzt aufgetretenen Fehler in Win32 beibehält« in der Online-Dokumentation verewigt haben.

Parameter	Bedeutung
Charset	Legt fest, ob Strings bei der Übergabe als Parameter im Unicode-Format bleiben (`Charset.Unicode`), grundsätzlich in Ansi-Strings unter Berücksichtigung der landesspezifischen Einstellungen des Zielsystems umgewandelt werden (`Charset.Ansi`) – oder sich die .NET-Laufzeitumgebung am Zielsystem orientiert (`Charset.Auto`), also unter Windows NT und Windows 2000/XP Unicode, unter Windows 98/Me dagegen Ansi verwendet. Die Standardvorgabe `Charset.Auto` muss beispielsweise dann durch eine explizite Angabe ersetzt werden, wenn Sie eine API-Routine wie `MessageBoxA` (also die Ansi-Variante) unter Windows 2000 aufrufen wollen.
CallingConvention	Legt unter anderem fest, ob die aufrufende oder die aufgerufene Routine die Parameter wieder vom Stack räumt. Die Standardvorgabe `CallingConvention.WinAPI` berücksichtigt die diesbezüglichen Unterschiede zwischen Windows CE und den anderen Varianten automatisch; für eigene DLLs, die Klassen direkt exportieren (und für deren Methoden einen `this`-Zeiger erwarten), ist `CallingConvention.ThisCall` eine interessante Möglichkeit.

Tabelle 26.1: Parameter von `DllImportAttribute` (Forts.)



Theoretisch wäre es denkbar, dass die .NET-Laufzeitumgebung und/oder Windows bei einer Angabe wie `[DllImport("C:\\Temp\\user32.dll ")]` *erst einmal prüfen, ob user32.dll bereits geladen ist – weshalb man hier je nach System unweigerlich System\user32.dll oder System32\user32.dll serviert bekommen würde.*

Tatsächlich unterscheidet das System nicht nur über den DLL-Namen, sondern auch über den Suchpfad (soweit angegeben). Für das im Folgenden auszugsweise wiedergegebene Codebeispiel DllPath wurde mit Delphi eine Datei user32.dll erstellt, die ebenfalls eine Routine `MessageBeep()` *exportiert:*

```
[DllImport("F:\\MS NET\\Projekte\\PInvoke1\\user32.dll",
  EntryPoint = "MessageBeep")]
public static extern bool DelphiMessageBeep(uint soundtype);

private void bDelphiMessageBeep_Click(object sender, System.EventArgs e)
{
  DelphiMessageBeep(0);  // Aufruf der DLL in F:\Projekte...
}

[DllImport("user32.dll", EntryPoint = "MessageBeep")]
public static extern bool SystemMessageBeep(uint soundtype);

private void bMessageBeep_Click(object sender, System.EventArgs e)
{
  SystemMessageBeep(0);  // Aufruf der DLL in System bzw. System32
}
```

26.1.2 Ladezeitpunkte und Fehlerprüfung

Während Programmierumgebungen wie Microsoft C/C++ und Delphi beim Einbinden von DLLs schlicht entsprechende Füllstellen für den Windows-Linker produzieren, also das Lokalisieren und Laden dieser Dateien grundsätzlich dem System überlassen, fahren C# und .NET hier eine Art zweigleisiger Strategie: Das Laden grundlegender DLLs des Frameworks (wie *mscorlib.dll*) übernimmt der Windows-Linker, der Rest – die Implementation von Klassen und per PInvoke Eingebundenes – wird erst dann gesucht und geladen, wenn es notwendig ist (Stichworte: `LoadLibrary()` und `GetProcAddress()`).

Auf der Habenseite kann dieses Prinzip eine geringere Speicherbelastung und kürzere Startzeiten von Anwendungen verzeichnen. Der dafür zu entrichtende Preis sieht so aus, dass man die Quittung für eine Deklaration wie

```
[DllImport("user32.dl")]   // <- Tippfehler: hätte user32.dll heißen sollen
public static extern bool MessageBeep(uint soundtype);
```

eben erst dann in Form einer Fehlermeldung zu sehen bekommt, wenn die Anwendung die Routine tatsächlich aufruft.

26.1.3 Codebeispiel – Ladezeitpunkt einer DLL

Zu welchem Zeitpunkt eine DLL geladen und wann sie wieder aus dem Speicher hinausbefördert wird, lässt sich mit einer in Delphi geschriebenen DLL demonstrieren, die entsprechende Meldungen ausgibt. Den folgenden Code finden Sie im Beispielprojekt *DLLLoadTime*:

```
library DelphiDLLTest; // Delphi-DLL für den Einsatz mit .NET
uses Windows;

{$R *.res}

procedure HelloWorld; stdcall; export;
begin
  MessageBox(0, 'Hello, world','Delphi-DLL',0);
end;

procedure MessageBeep(MsgType: Integer); stdcall; export;
begin
  MessageBox(0, 'MessageBeep', 'Beep', 0);
end;

exports
  HelloWorld,
  MessageBeep;

function DllEntryPoint(iReason: Integer):Boolean;
  procedure ShowMessage(Msg: String);
```

```
  begin
    MessageBox(0, PChar(Msg), 'Delphi-DLL', 0);
  end;
begin
  case iReason of
    Dll_Process_Attach: ShowMessage('Process Attach');
    Dll_Thread_Attach: ShowMessage('Thread Attach');
    Dll_Thread_Detach: ShowMessage('Thread detach');
    Dll_Process_Detach: ShowMessage('Process detach');
  end;
  Result := True;  // Laden erfolgreich
end;

begin
  // globaler Funktionszeiger von Delphi für Rückmeldungen
  // des Systems (hier vor allem: Ausblenden aus dem Adressraum)
  DllProc := @DLLEntryPoint;
  // fürs Einblenden in einen Adressraum muss die Meldungsroutine
  // direkt aufgerufen werden
  DllEntryPoint(Dll_Process_Attach);
end.
```

Das dazugehörige C#-Programm definiert für den Aufruf der exportierten Funktion `HelloWorld()` eine eigene Klasse.

```
public class DelphiHello
{
  [DllImport("F:\\MS NET\\Projekte\\PInvoke1\\DelphiDLLTest.dll")]
  public static extern void HelloWorld();

  public void CallHelloWorld()
  {
    HelloWorld();
  }

}
```

Das Formular von `PInvoke1` enthält eine einzige Schaltfläche; die dazugehörige Behandlungsroutine legt ein `DelphiHello`-Objekt an, ruft seine Methode `CallHelloWorld()` auf, und überlässt dann dem Garbage Collector die Aufräumarbeiten.

```
private void bDelphiHello_Click(object sender, System.EventArgs e)
{ DelphiHello dh = new DelphiHello();

  MessageBox.Show("DelphiHello-Instanz angelegt");
  dh.CallHelloWorld();
}
```

Die Reihenfolge der Rückmeldungen beim ersten Klick:

```
DelphiHello-Instanz angelegt
Delphi-DLL: Process attach
Delphi-DLL: Hello, world
```

Die Rückmeldung »Process detach« fehlt. Die Schlussfolgerungen daraus sind anfänglich etwas verblüffend:

- Das Laden der DLL geschieht nicht beim Anlegen des Objekts, sondern erst beim Aufruf der Methode, die ihrerseits die importierte DLL-Funktion aufruft.
- Die DLL bleibt nach dem erstmaligen Laden im Speicher, obwohl das `DelphiHello`-Objekt durch die Garbage Collection abgeräumt wird. Folglich fehlt bei weiteren Klicks auch die Rückmeldung »Process attach«.

Tatsächlich bekommen Sie die Meldung »Process detach« (die hier gleichbedeutend ist mit »DLL wird aus dem Speicher geworfen«) erst beim Ende des Programms zu sehen.

Allerdings handelt es sich hier nicht etwa um eine Lässlichkeit der Entwickler, sondern um eine konsequente Umsetzung der Prinzipien, denen C# ansonsten folgt: Klassen mit statischen Methoden werden beim ersten Zugriff (in Form einer anonymen Instanz) initialisiert, die im weiteren bis zum Ende des Prozesses zur Verfügung steht. Dieser durchaus erwünschte Effekt hängt übrigens mit dem JIT-Compiling zusammen. (Eine Möglichkeit, derartige Instanzen während des Programmlaufs wieder abzuräumen, besteht unseres Wissens nicht. Allerdings hüllt sich die Online-Dokumentation von C# auch beim Thema »Ladezeitpunkte« in völliges Schweigen.)



Wie bereits erwähnt, geht C# bei der Initialisierung statischer Instanzen einen anderen Weg als C++ und Delphi: Dort werden derartige Objekte nicht beim ersten Zugriff, sondern im Rahmen des Programmstarts besetzt.

26.1.4 Fehlerprüfungen

Wer Routinen der Windows-API per PInvoke aufrufen will, kann es wohl guten Gewissens den eingebauten Mechanismen der .NET-Laufzeitumgebung überlassen, den Benutzer im Falle eines Falles zu benachrichtigen – schließlich ist der Umstand »user32.dll nicht gefunden« praktisch gleichbedeutend mit »Kein Windows installiert«. Für andere DLLs empfiehlt sich dagegen dringend ein mit `try...catch` geklammerter Aufruf irgendeiner importierten Routine zu einem wohldefinierten Zeitpunkt – wie es das Codebeispiel *DllCheck* demonstriert:

```
[DllImport("F:\\MS NET\\Projekte\\PInvoke2\\DelphiDLLTest2.dll")]
internal static extern uint GetVersion();

private void DLL1TestFunc()
{
  GetVersion();
}
```

```
private bool CheckDLLImports()
{ bool Res = true;
  try
  { // der Reihe nach bei allen DLLs einmal anklopfen
    DLL1TestFunc();
    // DLL2TestFunc();
    // ...
  }
  catch
  {
    Res = false;
  }
  return Res;
}
// ...
private void Form1_Load(object sender, System.EventArgs e)
{
  if (!CheckDLLImports())
  {
    MessageBox.Show("Fehler beim Laden zusätzlicher DLLs", "PInvoke2");
    Close();
  }
}
```

Eine mögliche Alternative besteht aus einem Rückgriff auf eine Technik aus noch älteren Zeiten:




```
[DllImport("kernel32.dll",SetLastError=true)]
static private extern int LoadLibrary(string libname);

[DllImport("kernel32.dll",SetLastError=true)]
static private extern void FreeLibrary(int libhandle);

// DLL zur Prüfung laden und gleich wieder hinauswerfen
private bool CheckDLL(string DLLName)
{ int LibHandle;
  LibHandle = LoadLibrary(DLLName);
  if (LibHandle != 0) FreeLibrary(LibHandle);
  return (LibHandle != 0);  // true für "DLL gefunden"
}
private bool CheckDLLImports()
{
  return
   (CheckDLL("F:\\MS NET\\Projekte\\PInvoke2\\DelphiDLLTest2.dll")
    /* && CheckDLL("...") */
   );
}
```

Über einen weiteren Import aus kernel32.dll, nämlich der API-Funktion `GetProcAddress()`*, ließe sich auch noch sicherstellen, dass nicht nur die jeweilige DLL, sondern auch die gewünschte exportierte Routine vorhanden ist. (Danach würde Sie niemand davon abhalten, das Ergebnis von* `GetProcAddress` *gleich in einen entsprechend typisierten Delegaten umzusetzen und auf diese Weise den gesamten Import »zu Fuß« zu erledigen. Ob das sinn-*

voll ist, erscheint angesichts der von C# im Zusammenhang mit einem `DllImport`*-Attribut gebotenen Möglichkeiten zur Konvertierung von Datentypen allerdings fraglich.)*

26.2 Marshalling

Bei Parametern und Ergebnissen mit einfachen Datentypen wie `int` und `bool` greift eine automatische Typumwandlung, die auch mit Gleitkommazahlen usw. verlässlich ihre Dienste tut.

Allerdings kann die Sache bereits mit Strings etwas schwieriger werden, weil es auf der anderen Seite immerhin vier Varianten und somit den Zwang zur Konvertierung gibt: *LPStr*, *LPWStr* (also Zeiger auf mit NULL abgeschlossene ANSI- bzw. Unicode-Strings), *BStr* (Unicode mit vorangestelltem Wort für die Zeichenzahl) und *LPTStr* (ANSI oder Unicode je nach Plattform). In einigen – erfreulich wenigen – Fällen lässt sich die Art der Konvertierung nicht allgemein festlegen, und man muss »zu Fuß« nachhelfen. (Das Protobeispiel dafür sind natürlich Konvertierungsroutinen zwischen Zeichensätzen – etwa von ANSI-EBCDIC nach Unicode und zurück –, die zwangsläufig mit Zeigern auf unterschiedliche Strings arbeiten.)

26.2.1 MarshalAs und UnmanagedType

Zur Festlegung, in welchem Format ein Parameter übergeben werden soll, definiert C# den Aufzählungstyp `UnmanagedType`, der von `AnsiBStr` (Ansi-String mit vorangestelltem Längenbyte) über `I1` bis `I8` (Integer mit 1, 2, 4 und 8 Bytes) und `IUnknown` (COM-Schnittstellenzeiger) bis `VBByRefStr` (Referenz auf einen String zur Bearbeitung durch die Laufzeitfunktionen von Visual Basic) so ziemlich alles an Formaten anbietet, was des Windows-Programmierers Herz begehren kann.

Eingesetzt wird dieser Aufzählungstyp hauptsächlich als Parameter für den Aufruf des Konstruktors der Attributklasse `MarshalAsAttribute`. Das im Folgenden auszugsweise wiedergegebene Codebeispiel *CompleteDecl* treibt das auf die Spitze:

```
[DllImport("user32.dll", EntryPoint = "MessageBoxA")]
  public extern static int MessageBox(
   [MarshalAs(UnmanagedType.U4)] uint hWnd,
   [MarshalAs(UnmanagedType.LPStr)] string lpText,
   [MarshalAs(UnmanagedType.LPStr)] string lpCaption,
   [MarshalAs(UnmanagedType.U4)] uint uType);
  private void button1_Click(object sender, System.EventArgs e)
  {
      MessageBox(0, "Text der Meldung", "Überschrift", 0);
  }
```

Da `MessageBoxA` ausschließlich mit Zeigern auf ANSI-Strings arbeitet und `uint` sowieso einem vorzeichenlosen Integer mit 32 Bit entspricht, hätte es im gegebenen Beispiel allerdings auch eine wesentlich kürzere Angabe getan:

```
[DllImport("user32.dll", EntryPoint = "MessageBoxA",
   CharSet = CharSet.Ansi)]
public extern static int MessageBox(
  uint hWnd, string lpText, string lpCaption, uint uType);
```



`MarshalAs` *wird nicht nur bei Funktionsprototypen, sondern auch bei Strukturdefinitionen eingesetzt – siehe die folgenden Abschnitte. Dort geht es unter anderem auch um den (hier unterschlagenen) optionalen Parameter* `SizeConst`.

26.2.2 Strings und Referenzparameter

Windows bietet eine ganze Reihe von API-Funktionen, die ihre Ergebnisse in einem vom Aufrufer zur Verfügung gestellten Stringpuffer hinterlassen wollen. Der einfachste Weg, dieser Forderung zu genügen, führt über `StringBuilder`-Objekte, die den von ihnen belegten Speicherplatz letztlich sowieso über Zeiger zur Verfügung stellen – auch wenn es Zeiger bekanntlich in C# nicht gibt.

Codebeispiel – Abfrage des Benutzernamens

Stellvertretend für alle API-Funktionen, die Ergebnisse in aufruferseitig bereitgestellten Stringpuffern frei wählbarer Größe hinterlassen, soll hier der Einsatz der Routine `WNetGetUser` demonstriert werden. Das dazugehörige Formular enthält eine einzige Schaltfläche und ist deshalb nicht abgebildet. Hier relevante Quelltext aus dem Bespielprojekt *GetUserName*:

```
using System.Text;   // StringBuilder
using System.Runtime.InteropServices;
// ...
[DllImport("MPR.DLL", EntryPoint = "WNetGetUserA", CharSet = CharSet.Ansi)]
internal static extern uint WNetGetUser(string LocalResourceName,
  StringBuilder lpBuffer,
  [MarshalAs(UnmanagedType.U4)] ref int BufLen);
```

Die API-Dokumentation definiert `WNetGetUserA()` folgendermaßen:

```
// C!
DWORD WNetGetUser(
    LPCTSTR lpName,     // lokaler Ressourcenname (const)
    LPTSTR lpUserName,  // Stringpuffer für den Benutzernamen
    LPDWORD lpnLength   // Zeiger auf die Pufferlänge
   );
```

Obwohl hier mit drei Zeigern gearbeitet wird, unterscheidet sich die Behandlung der drei Parameter kräftig: Der erste ist ein reiner `in`-Parameter, der zweite fällt in die Kategorie `out`, der dritte wird sowohl gelesen (zur Ermittlung der maximalen Pufferlänge) als auch geschrieben (tatsächliche Zeichenzahl) und ist somit ein `ref`-Parameter.

Wie im C#-Quelltext zu sehen, reicht im ersten Fall das Standard-Marshalling, das hier für die Umsetzung des Unicode-Strings `LocalResourceName` in einen Zeiger auf einen ANSI-String sorgt; und beim dritten Parameter tut es ein schlichtes `ref`, um der Laufzeitumgebung anzuzeigen, dass er nicht als Wert, sondern eben als Adresse der entsprechenden C#-Variablen übergeben werden soll.

Der zweite Parameter ist *per se* eine Referenz, weil das Standard-Marshalling `StringBuilder`-Objekte in einen Zeiger umsetzt (der hier nicht stellvertretend für das Objekt selbst, sondern eben für den von ihm belegten Speicherplatz steht). Die notwendige Konvertierung des von der API-Funktion gelieferten ANSI-Strings in Unicode findet ebenfalls als Teil des Standard-Marshalling statt.

Da die Routine einiger Vorbereitungen in Form eines entsprechend konstruierten `StringBuilder`-Objekts und einer Längenangabe bedarf, ist sie als `internal` deklariert und damit ausschließlich über eine andere Methode derselben Assembly zugänglich.

```
public string GetUserName()
{  int BufLen = 256;

  StringBuilder objBuffer = new StringBuilder(BufLen);
  WNetGetUser("", objBuffer, ref BufLen);
  return objBuffer.ToString();
}
```

`GetUserName()` legt also ein `StringBuilder`-Objekt mit entsprechender Kapazität an (wobei es im gegebenen Beispiel auch 32 anstelle von 256 Zeichen getan hätten). Die Variable `BufLen` muss auch hier als Referenz übergeben werden. Da die Routine einen String (und nicht das `StringBuilder`-Objekt) zurückgeben soll, wird abschließend dessen Methode `ToString()` eingesetzt.

Dem Aufruf von `GetUserName()` merkt man dann nicht mehr an, dass hinter der Methode ein wenig zusätzliche Detailarbeit steckt:

```
private void button1_Click(object sender, System.EventArgs e)
{
  MessageBox.Show(GetUserName(), "Benutzername");
}
```

Die Alternative zur Übergabe als Referenz (`ref`) ist bei PInvoke die Deklaration mit [In, Out]. Details dazu finden Sie im nächsten Abschnitt, ein Codebeispiel auf Seite 1088.



26.2.3 Strukturtypen als Parameter

Zum Einbinden einer Windows-DLL in ein Programm, das mit einer der »klassischen« Programmiersprachen geschrieben worden ist, benötigt man eine Header-Datei in derselben Programmiersprache, die Prototypen der einzelnen Funktionen und vor allem die verwendeten Strukturtypen deklariert. C/C++-ProgrammiererInnen haben hier schon immer die wenigsten Sorgen gehabt, VB-Fans können zumindest bei den von der Windows-API verwendeten Typen auf eine spezielle »Import-Bibliothek« zurückgreifen. Bei Delphi gilt dagegen seit jeher das Prinzip Hoffnung – nämlich, dass sich irgendjemand (meist aus der JEDI-Gruppe) bereits die Mühe gemacht haben möge, entsprechende Deklarationen als Delphi-Quelltext zusammenzutippen.

Die erste schlechte Nachricht für C#-Programmierer ist, dass auch diese Sprache natürlich nicht ohne (in C# geschriebene) Deklarationen »externer« Strukturtypen auskommt. Die zweite: Da das »Hintertürchen« PInvoke in erster Linie für das Einbinden von DLLs vorgesehen ist, die eben *nicht* Bestandteil der Windows-API sind, gibt es für C# keine »Windows-Typenbibliothek« á la VB.

Um es noch einmal zu betonen: Da DLLs keine Metadaten zu Signaturen exportierter Funktionen und den verwendeten Strukturtypen enthalten, sind Sie hier wie in einer der klassischen Programmiersprachen völlig auf sich selbst gestellt. Nichts und niemand – außer unvermeidlich folgenden Schutzverletzungen – hält Sie davon ab, nach entsprechender Deklaration eine API-Routine wie `MessageBox()` mit fünf Gleitkomma-Parametern (anstelle von zwei vorzeichenlosen Integern und zwei Stringzeigern) aufzurufen oder `FindFirst` eine Struktur unterzujubeln, in der Felder vertauscht sind oder auf falschen Offsets stehen. Hier ist also äußerste Sorgfalt angesagt.



StructLayout

Die gute Nachricht ist, dass man sich in Redmond dazu entschlossen hat, auch bei notwendigen Übeln ganze Arbeit zu leisten. C# bietet bei der Definition »externer« Strukturen mit dem Attribut `StructLayout` einige Möglichkeiten, die das von anderen Programmiersprachen her leidvoll gewohnte Ratespiel – wo in einer Struktur jetzt welches Feld landet, mit welcher Größe, und mit welchem Typ – ein für allemal beenden.

Die beiden im Zusammenhang mit DLL-Aufrufen wichtigen Varianten von `StructLayout` unterscheiden sich über den Aufzählungstyp `LayoutKind`, der dem Konstruktor der dazugehörigen Klasse `StructLayoutAttribute` übergeben wird:

`LayoutKind.Sequential` ordnet die Felder der Struktur in exakt derselben Reihenfolge an, wie sie in der Deklaration angegeben sind, und entspricht weitgehend dem, was man von klassischen Programmiersprachen gewohnt ist. Die Ausrichtung der Felder an Wort- oder DWord-Grenzen wird dabei durch den benannten Parameter `Pack` festgelegt, für den Zweierpotenzen im Bereich von 0 bis 256 zulässig sind. Ein Beispiel:

```
[StructLayout(LayoutKind.Sequential, Pack = 4)]
public class FindData
{
  public int  fileAttributes = 0;

  // im Original eine eingebettete Struktur des Typs FILETIME
  public int  creationTime_lowDateTime = 0 ;
  public int  creationTime_highDateTime = 0;
  // ...
```

`LayoutKind.Explicit` erlaubt bzw. fordert die explizite Festlegung des Offsets jedes einzelnen Feldes über zusätzliche Attribute des Typs `FieldOffset`:

```
[StructLayout(LayoutKind.Sequential)]
public class FindData
{
  [FieldOffset(0)]
  public int  fileAttributes = 0;
  // im Original eine eingebettete Struktur des Typs FILETIME
  [FieldOffset(4)]
  public int  creationTime_lowDateTime = 0 ;
  [FieldOffset(8)]
  public int  creationTime_highDateTime = 0;
  // ...
```

Der Parameter `Pack` wird in diesem Fall ignoriert. Die Fehlermeldung, dass einem Feld ohne `FieldOffset` ein »structoffset«-Attribut fehle, dürfte noch aus einer Vorversion von VS.NET stammen, und wird wohl in einem der nächsten Service Packs korrigiert. Eine Prüfung, ob sich die einzelnen Felder aufgrund ihrer festgelegten Offsets überschneiden, findet *nicht* statt – tatsächlich lassen sich auf diese Weise nämlich auch *Unions* definieren (siehe übernächster Abschnitt).



In einer dritten Variante – der Standardvorgabe `LayoutKind.Auto` *– ordnet der Compiler die Felder von Datenstrukturen in Eigenregie so an, dass eine möglichst optimale Balance zwischen Ausrichtung an Doppelwortgrenzen einerseits und Platzbedarf andererseits entsteht. Die Details dieser Anordnung sind nicht dokumentiert, und DLL-Aufrufe mit diesem Layout wären in jedem Fall nicht sicherer als ein Roulette-Gewinn.*

struct vs. class

`StructLayout` ist sowohl auf Klassen (`class`) als auch auf Strukturen (`struct`) anwendbar, die bei DLL-Aufrufen sowieso weitgehend austauschbar sind. Die Frage, welches der beiden Schlüsselwörter zum »Nachdeklarieren« externer Strukturen das sinnvollere ist, lässt sich nicht allgemein beantworten – kein Wunder, sonst hätten sich die Microsoft-Entwickler wohl auch auf eine der beiden Möglichkeiten beschränkt.

Zur Verdeutlichung zwei Deklarationen:

```
public struct XYStruct
{
  public int x, y;
}
public class XYClass
{
  public int x, y;
}
```

Die Unterschiede im »Handling« sind die üblichen:

- `XYStruct`-Variablen werden auf dem Stack gespeichert, Instanzen von `XYClass` auf dem Heap.
- Deklaration und Initialisierung sehen so aus:

```
XYStruct xyS;
xyS.X = 0; xyS.Y = 0;  // Initialisierung Feld für Feld

XYClass xyC = new XYClass();  // Initialisierung über den Konstruktor
```

- Bei Übergaben an importierte Funktionen wird es interessanter:

```
[DllImport("xyz.dll")]
internal static StructAPIFunc(ref XYStruct xyS);
[DllImport("xyz.dll")]
internal static ClassAPIFunc([In, Out] XYClass xyS);
CallAPIFunc(ref xyS);
CallAPIFunc(xyC);
```

Einen technischen Grund dafür gibt es meines Wissens nicht: In beiden Fällen legt die .NET-Laufzeitumgebung einen Puffer auf dem Stack an und setzt dort die Werte der einzelnen Felder ein, wobei ggf. die beim Marshalling erforderlichen Typumwandlungen stattfinden, und übergibt die Stack-Adresse an die aufzurufende Funktion.

Wenn die Struktur als Referenzparameter deklariert (bzw. bei der Übergabe von Klasseninstanzen: das Attribut `Out` angegeben) ist, dann findet nach dem Aufruf der importierten Funktion ein Marshalling in umgekehrter Richtung statt.

Die direkte Antwort auf die Frage »struct oder class« muss deshalb wohl lauten: »Es kommt hauptsächlich darauf an, was Sie mit der Datenstruktur sonst noch vorhaben«.



Ein Praxisbeispiel für die Deklaration einer API-Datenstruktur über `struct` *finden Sie auf Seite 1095.*

Größenangaben und Unions

Zu `StructLayout` lässt sich optional ein benannter Parameter `Size` angeben, der dann die Gesamtgröße der Struktur bestimmt:

```
[StructLayout(LayoutKind.Sequential, Pack = 4, Size = 8)]
public class FindData
{
  public int  fileAttributes = 0;              // 4 Bytes
  public int  creationTime_lowDateTime = 0 ;   // 8 Bytes
  public int  creationTime_highDateTime = 0;   // 12 Bytes
  // ...
```

Wie dieses Beispiel schon andeutet, wird auf diese Weise lediglich festgelegt, wie viele Bytes die .NET-Laufzeitumgebung transportiert – als Prüfung, ob eine »nachdeklarierte« Struktur tatsächlich einem Original (mit bekannter Größe) entspricht, ist `Size` nicht gedacht. Vielmehr geht es hier offensichtlich um Strukturen mit »offenem Ende« – beispielsweise:

```
[StructLayout(LayoutKind.Sequential, Size = 256)]
public class ExampleData
{
  public int  ActualSize = 0;  // Größenangabe
  public byte AnyData;         // Platz für bis zu 252 Bytes
};
```

Diese Struktur bietet also im Feld `AnyData` Platz für maximal 252 Bytes, obwohl die deklarierten Datenfelder insgesamt gerade einmal 6 Bytes belegen.

Der Platz für String-Datenfelder in Strukturen (in C: char-Arrays) wird dagegen nicht über das Attribut `StructLayout` festgelegt, sondern individuell über den optionalen Parameter `SizeConst` von `MarshalAs`:

```
[StructLayout(LayoutKind.Sequential, Pack = 4, Size = 8)]
public class FindData
{
  // ...
  public int  dwReserved1 = 0;
  [ MarshalAs( UnmanagedType.ByValTStr, SizeConst=260 )]
  public String  fileName = null;
  // ..
};
```

Dieses Beispiel definiert also ein Datenfeld `fileName`, das innerhalb der Struktur für 260 Zeichen (bei Unicode also 520 Bytes) Platz bietet. Den Rest (nämlich die Umwandlung zwischen ANSI und Unicode sowie die Konvertierung von und nach `string`) übernimmt wiederum die .NET-Laufzeitumgebung ohne weiteres Zutun. (`UnmanagedType.ByValTStr` steht für einen Zeichenarray mit fixer Elementzahl, das abhängig vom dem zusammen mit `StructLayout` angegebenen Zeichensatz ANSI- oder Unicode-Zeichen enthält.)

Da der C#-Compiler absichtlich nicht prüft, ob sich per `FieldOffset` angeordnete Felder überschneiden, ist mit diesem Attribut auch die Definition von Unions (in Delphi: Varianten) möglich. Im Beispielprojekt *uniondemo* ist eine solche Struktur definiert:

```
[StructLayout( LayoutKind.Explicit)]
  public class CVarRec
{
  [FieldOffset(0)] public int Kind;
  // gleiche Offsets = union
  [FieldOffset(4)] public int IntVal;
  [FieldOffset(4)] public double DoubleVal;
}
```

Die Übergabe einer solchen Struktur enthält keinerlei Besonderheiten.

```
[DllImport ("DelphiDLLTest4.dll")]
public static extern void ShowVarRec(CVarRec VR);

private void bStart_Click(object sender, System.EventArgs e)
{ CVarRec VR = new CVarRec();
  // Übergabe der Union mit besetztem IntVal-Feld
  VR.Kind = 0; VR.IntVal = 4;
  ShowVarRec(VR);
  // noch einmal, aber mit besetztem DoubleVal-Feld
  VR.Kind = 1; VR.DoubleVal = 3.17;
  ShowVarRec(VR);
}
```

Die – wieder einmal in Delphi geschriebene – Gegenseite sieht so aus:

```
library DelphiDLLTest4;  // variante Records
uses Windows, SysUtils;

{$R *.res}
type
  TVarRec = packed record
    Kind: Integer;
    case Integer of
     0: (IntVal: Integer);
     1: (DoubleVal: Double);
    end;
  PVarRec = ^TVarRec;
```

```
procedure ShowVarRec(Rec: PVarRec); stdcall; export;
  procedure MsgBox(Msg: String);
  begin
    MessageBox(0, PChar(Msg), 'Delphi-DLL',0);
  end;
begin
  with Rec^ do
    case Kind of
     0: MsgBox(Format('Integer: %d',[IntVal]));
     1: MsgBox(Format('Double: %f',[DoubleVal]));
    else
      MsgBox(Format('Unknown Kind: %d',[Kind]));
    end;
end;

exports
  ShowVarRec;

begin
end.
```

Das Ergebnis finden Sie (als Montage) in Abbildung 26.1.

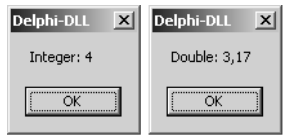


Abbildung 26.1: DLL-Aufrufe mit einer union (Montage)

Codebeispiel – Dateilisten mit FindFirst/FindNext

Die in diesem Abschnitt vorgestellte Beispielanwendung *FindFirst* basiert auf einem leicht nachkorrigierten Beispiel von Microsoft. Es listet die Dateien im aktuellen Verzeichnis auf und verwendet dazu die API-Funktionen `FindFirst()`/`FindNext()`, die ihrerseits den in den vorangehenden Abschnitten schon öfter in Teilen zitierten Strukturtyp WIN32_FIND_DATA bzw. die dafür »nachdeklarierte« C#-Klasse `FindData` verwenden.

Das Formular enthält diesmal einige Elemente mehr, deren Bedeutung sich aber wohl ohne längere Erklärungen aus Abbildung 26.2 ergeben dürfte. Auffällig ist dort, dass die Dateien in ihrer »natürlichen« Reihenfolge erscheinen, also nicht alphabetisch sortiert, und das diplomatisch als `alternateFilename` benannte Feld der Struktur nur dann mit einem DOS-Dateinamen im 8.3-Format besetzt ist, wenn der »lange« Dateiname nicht diesen Konventionen genügt.

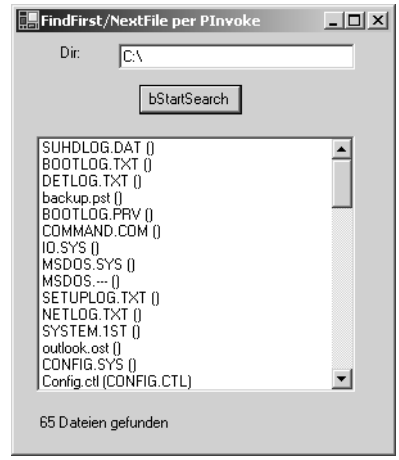


Abbildung 26.2: Die Ergebnisse von FindFirstFile/ FindNextFile

Der Quelltext ist hier wie üblich nur auszugsweise wiedergegeben:

```
// Die Strukturdefinition in C
/*
typedef struct _WIN32_FIND_DATA
{
  DWORD    dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
  DWORD    dwReserved0;
  DWORD    dwReserved1;
  TCHAR    cFileName[ MAX_PATH ];
  TCHAR    cAlternateFileName[ 14 ];
} WIN32_FIND_DATA, *PWIN32_FIND_DATA;
*/
// Der "nachdeklarierte" Strukturtyp in C#
[StructLayout( LayoutKind.Sequential)]
  public class FindData
  {
    public int  fileAttributes = 0;
    // creationTime ist im Original eine FILETIME-Struktur
    public int  creationTime_lowDateTime = 0 ;
    public int  creationTime_highDateTime = 0;

    // noch eine FILETIME-Struktur(lastAccessTime)
    public int  lastAccessTime_lowDateTime = 0;
    public int  lastAccessTime_highDateTime = 0;
    // und noch eine (lastWriteTime)
    public int  lastWriteTime_lowDateTime = 0;
    public int  lastWriteTime_highDateTime = 0;

    public int  nFileSizeHigh = 0;
    public int  nFileSizeLow = 0;
```

```
    public int  dwReserved0 = 0;
    public int  dwReserved1 = 0;

    // im Microsoft-Beispiel auf 256 gesetzt, deshalb stimmt dort
    // der Offset von alternateFileName nicht
    [ MarshalAs( UnmanagedType.ByValTStr, SizeConst=260 )]
    public String  fileName = null;

    [ MarshalAs( UnmanagedType.ByValTStr, SizeConst=14 )]
    public String  alternateFileName = null;  // DOS-Dateiname
  }
```

Bei `StructLayout` wurde bewusst kein Zeichensatz angegeben (was auf die Standardvorgabe `CharSet = CharSet.Auto` hinausläuft), damit sich die Klasse `FindData` sowohl für die ANSI- als auch für die Unicode-Version von `FindFirstFile` verwenden lässt. `UnmanagedType.ByValTStr` wird konsequent bei der Ausführung unter Windows 9x als ANSI, ansonsten als Unicode behandelt. Im zweiten Fall belegt der Dateiname immerhin 520 Bytes in der Struktur.

```
[ DllImport("Kernel32.dll")]
public static extern uint FindFirstFile(string fileName,
  [ In, Out ] FindData findFileData );

[ DllImport("Kernel32.dll")]
public static extern bool FindNextFile(uint hFindFile,
  [ In, Out ] FindData findFileData );

[ DllImport( "Kernel32.dll")]
public static extern bool FindClose(uint hFindFile);
```

Die Attribute `[In, Out]` signalisieren, dass zwei Transporte notwendig sind: von C# zu den unverwalteten Datentypen vor dem Aufruf der API-Funktion, und von den unverwalteten Datentypen zurück zu C# nach dem Aufruf.



Hier zeigt sich wieder einmal, dass einfache Variablen eben auch in C# keine Objekte sind. Wenn die exportierte Routine einen Zeiger auf einen schlichten Integer erwartet, sind `ref` *und* `[In, Out]` *austauschbar – bei Klassen wie* `FindData` *dagegen nicht: Mit einer Referenz auf eine C#-Klasse können API-Funktionen nicht viel anfangen (vgl. Seite 1083).*

```
private void bStartSearch_Click(object sender, System.EventArgs e)
{
  uint hFind;
  string Dir = eDir.Text;  // Pfad
  FindData SearchRec = new FindData();
  // minimalistische Prüfung des Suchwegs
  if (Dir == "") Dir = "C:\\";
  if (Dir[Dir.Length-1] != '\\') Dir += '\\';

  // Listenfeld entleeren und erste Datei suchen
  listBox1.Items.Clear();
```

```
  hFind = FindFirstFile(Dir+"*.*", SearchRec);
  if (hFind != -1)
  {
    while (true)
    { // Verzeichnisse und . bzw .. bei der Ausgabe übergehen
      if ((SearchRec.fileAttributes & 0x10) == 0)
        // DOS-Dateiname, soweit vorhanden, in Klammern
        listBox1.Items.Add(String.Format("{0} ({1})",
           SearchRec.fileName, SearchRec.alternateFileName));
      if (!FindNextFile(hFind, SearchRec))
        break;
    }
    FindClose(hFind);  // Suche beenden
  }
  lTotalFiles.Text = String.Format("{0} Dateien gefunden",
    listBox1.Items.Count);
}
```

Beim Klick auf die Schaltfläche `bStartSearch` übernimmt das Programm den String im (hier als `Dir` benannten) Eingabefeld, sorgt auf recht minimalistische Weise dafür, dass dabei (meistens) ein Suchpfad herauskommt, und startet die Suche über den Aufruf von `FindFirstFile()`. Wenn sich dabei zumindest ein Dateieintrag findet, der den Suchkriterien entspricht, liefert `FindFirstFile()` eine entsprechend besetzte Struktur und einen Handle zurück, der zur Fortsetzung der Suche über `FindNextFile()` verwendet und zu guter Letzt ordentlich ans System zurückgegeben wird.

26.2.4 Marshalling direkter Funktionsergebnisse

`MarshalAs` ist für direkte Funktionsergebnisse genauso verwendbar wie für Parameter. Um Zweideutigkeiten bei der Zuordnung zu vermeiden, haben sich die Entwickler hier ein zusätzliches Attribut einfallen lassen, zu dem die Online-Dokumentation erstaunlicherweise überhaupt nichts im Index findet – wahrscheinlich deshalb, weil das in den Betaversionen von C# noch anders gelöst war. Die Syntax sieht recht bedrohlich aus:

```
[return: MarshalAs(UnmanagedType.Typ)] internal static extern
  CTS-Ergebnistyp Funktionsname(Parameter...)
```

Sinnvoll bzw. notwendig ist eine solches explizites Marshalling unter anderem bei DLL-Funktionen oder Schnittstellen, die ihrerseits Zeiger auf COM-Schnittstellen zurückliefern – beispielsweise:

```
[return: MarshalAs(UnmanagedType.IUnknown)] CTS-Objekttyp Methode(...)
```

Wer das `return:` in einer solchen Deklaration vergisst, wird vom Compiler im schönsten Denglisch an die Betaphase erinnert: »Das Feature ‚MarshalAs on Method' sollte nicht mehr verwendet werden. Benutzen Sie stattdessen ‚return: location override with MarshalAs attribute'«.

Strukturtypen als Ergebnisse

Die meisten Programmiersprachen erlauben die Definition von Funktionen, die als direktes Ergebnis Strukturen beliebiger Komplexität zurückgeben. (Oder, exakter: simulieren diese Möglichkeit. Die Implementation sieht üblicherweise so aus, dass der Aufrufer eine entsprechende Struktur auf dem Stack belegt und einen versteckten Zeiger darauf an die Funktion übergibt.)

Kurz und schlecht: C# unterstützt diese Möglichkeit im Rahmen von PInvoke nicht – was auch kein Wunder ist, weil sich so etwas sprachübergreifend nur als Stückwerk realisieren ließe. Hier müssen Sie also auf Referenzparameter ausweichen (wenn man Möglichkeiten wie die nachträgliche Zerstückelung eines int64 in X- und Y-Koordinaten einer Struktur wie POINT mal außen vorlässt).

26.3 Rückrufe

Objektorientierte Programmiersprachen haben bei der Zusammenarbeit mit der (eben nicht objektorientierten) Windows-API normalerweise spätestens dann ein Problem, wenn es um Rückrufe geht: Routinen wie `EnumFonts()`, `EnumWindows()` etc. erwarten eine Rückruffunktion und legen vor dem Aufruf dieser Funktion brav die zu liefernden Informationen als Parameter auf dem Stack – mehr aber auch nicht.

Da Methoden in C++ als allerersten – versteckten – Parameter einen `this`-Zeiger als Referenz auf die jeweilige Objektinstanz erwarten (der bei Delphi als `Self` bezeichnet wird, was die Sache aber auch nicht besser macht) und ein solcher Parameter bei Rückrufen aus der Windows-API heraus schlicht fehlt, muss man sich bei diesen Programmiersprachen mit Konstrukten auf globaler Ebene behelfen. Das folgende Delphi-Fragment verdeutlicht, wie holprig so ein Konstrukt ausfallen kann:

```
function EnumWindowsCB(hWindow, UserValue: Cardinal): Boolean; stdcall;
begin
  // harte Typumwandlung von UserValue in einen Objektzeiger
  Result := TMyObject(UserValue).DoEnumWindowsCB(hWindow);
end;

function TMyObject.DoEnumWindowsCB(hWindow: Cardinal): Boolean;
begin
  // ... Auswertung
  Result := True;
end;
procedure TMyObject.StartEnumWindows
begin
  EnumWindows(@EnumWindowsCB, Cardinal(Self));
end;
```

Dass es hier von Ungereimtheiten und Falltüren förmlich wimmelt, dürfte auch den LeserInnen auffallen, die sich noch nie mit Delphi beschäftigt haben: `StartEnumWindows()` muss die Adresse der Rückruffunktion als untypisierten Zeiger und die Referenz auf die Objektinstanz (`Self`) gar als Integerwert übergeben, den die globale Funktion `EnumWindowCB()` dann »hart« wieder in eine Objektreferenz umsetzt; die eigentliche Rückrufroutine `DoEnumWindowsCB()` hat einen Parameter weniger, weil `UserValue` für den Self-Zeiger missbraucht wird – und außerdem darf man sich noch bedanken, dass hier von Typprüfungen der Parameter jede Spur fehlt, weil das Ganze sonst kaum über die Kompilierung hinauskäme.

In C# zeigt sich an dieser Stelle wieder einmal, wie umfassend und raffiniert das Konzept der Delegaten wirklich ist: Wie das folgende Codebeispiel zeigt, wird auch für Rückrufe aus der Windows-API heraus nicht mehr als ein schlichter – und typsicherer(!) – Delegat gebraucht.

Codebeispiel – EnumWindows

Wie Abbildung 26.3 zeigt, enthält das Formular des Beispielprojekts *EnumWindows* eine Schaltfläche (`bStartEnum`), die eine Listbox (`listBox1`) mit den Titeln aller aktuell existierenden Fenster füllt. Das `Label`-Feld `lTotalCount` gibt am Ende der Aufzählung den Wert von `listBox1.Items.Count` wieder.

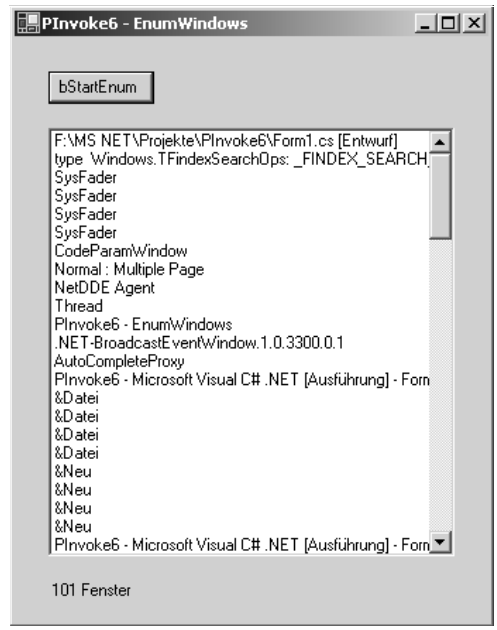


Abbildung 26.3: Die Ergebnisse von EnumWindows. Da hier nicht auf Sichtbarkeit der Fenster geprüft wird, ist die Liste recht lang.

Der dazugehörige Quelltext nimmt neben `InteropServices` auch den Namensraum `System.Text` per `using` auf, weil zur Abfrage der Fenstertitel ein `String-`

Builder-Objekt zum Einsatz kommt. Das folgende Listing kommt direkt zur Sache, d.h. übergeht wie üblich den vom Designer generierten Code:

```
public delegate bool EnumWindowsCBDelegate(uint hWindow, uint UserValue);

[DllImport("user32")]
public static extern int
   EnumWindows(EnumWindowsCBDelegate CB, uint UserValue);
[DllImport("user32")]
public static extern void GetWindowText(uint hWindow,
      StringBuilder WindowCaption, int nMaxCount);
```

Der Delegat EnumWindowsCBDelegate definiert den Prototyp der Rückruffunktion für EnumWindows() so, wie es auch in der API-Dokumentation steht; EnumWindows selbst wird als Funktion importiert und dabei so deklariert, dass sie im ersten Parameter die Adresse der Rückruffunktion als Delegatinstanz erwartet.

Die API-Funktion GetWindowText() liefert den Titel eines (über seinen Windows-Handle adressierten) Fensters.

```
public bool EnumWindowsCB(uint hWindow, uint UserValue)
{ StringBuilder WindowCaption = new StringBuilder(256);

  GetWindowText(hWindow, WindowCaption, WindowCaption.Capacity);
  if (WindowCaption.Length > 0)  // titellose Fenster übergehen
    listBox1.Items.Add(WindowCaption.ToString());
  return true;  // Aufzählung fortsetzen
}
```

Die über den Delegaten aufgerufene Methode hat (wie es sich in C# gehört) denselben Prototyp wie der Delegat, und UserValue steht zur freien Verwendung zur Verfügung. (Das Pferd sozusagen von hinten aufzuzäumen und nach entsprechender Änderung der Prototypen nun per UserValue ein C#-Objekt wie listBox1 übergeben zu wollen, setzt ein Aushebeln des CTS über diverse Konvertierungen voraus, ist also nicht zu empfehlen.)

EnumWindowsCB setzt zur Abfrage des jeweiligen Fenstertitels die bereits auf Seite 1079 besprochene Technik mit einem StringBuilder-Objekt ein; Fenster mit leerem Titel werden bei der Ausgabe übergangen.

```
private void bStartEnum_Click(object sender, System.EventArgs e)
{ EnumWindowsCBDelegate EnumCB = new EnumWindowsCBDelegate(EnumWindowsCB);

  listBox1.Items.Clear();
  EnumWindows(EnumCB, 0);
  lTotalCount.Text = String.Format("{0} Fenster", listBox1.Items.Count);
}
```

Die Behandlungsroutine für den Klick auf `bStartEnum` legt eine Delegatinstanz für die Methode `EnumWindowsCB` an, der dann als Rückrufadresse an EnumWindows übergeben wird. Der Parameter `UserValue` (in der API-Dokumentation: `lpContext`) wird *nicht* gebraucht: Wie zuvor erwähnt, wäre es tatsächlich gar nicht mal so einfach, dort einen Verweis auf eine andere Objektinstanz unterzubringen. Die Ausnahme von der Regel: Eine entsprechende Signatur von `EnumWindowsCB` und `EnumWindowsCBDelegate` vorausgesetzt, könnte man über `UserValue` ohne jeden Trick (und wiederum vollkommen typsicher) einen zweiten Delegaten übergeben.

26.4 Codebeispiel – Herunterfahren von Windows

Die bisher in diesem Kapitel vorgestellten Codebeispiele haben sich der Nachvollziehbarkeit halber bewusst an Bekanntes gehalten – auf gut Deutsch: sind in der Praxis samt und sonders überflüssig: Wenn man in C# für das Aufzählen von Fenstern oder Schriftarten, das Auflisten von Dateien in einem Verzeichnis usw. tatsächlich über PInvoke auf Funktionen der Windows-API zurückgreifen müsste, dann wäre es um die Zukunft von .NET wohl nicht sonderlich rosig bestellt. Die .NET-Klassenhierarchie ist reichhaltig genug, um jedenfalls für solche Anliegen eigene Mittel bereitzustellen – die letzten Endes natürlich nichts anderes als Verkapselungen der entsprechenden API-Aufrufe sind.

Zum Abschluss dieses langen Abschnitts soll aber wenigstens eine Funktion vorgestellt werden, für die es in den Bibliotheken von .NET tatsächlich kein Äquivalent gibt: Das Herunterfahren von Windows und das damit verbundene Hantieren mit Sicherheitsattributen auf Systemebene. (Wieso den Bibliotheken eine solche Funktion fehlt? Vermutlich, weil beim Sicherheitsmodell von .NET ja immer noch das Sandkastenprinzip mitschwingt – und dort ist ja das vollständige Abräumen der Spielwiese kaum vorgesehen.)

Das Formular des Codebeispiels *ShutDownWindows* (siehe Abbildung 26.4) zeichnet sich erneut nicht gerade durch seine Fülle von Steuerelementen und Verzierungen aus. Die dem Kombinationsfeld `comboBox1` anhängende Liste wird in `FormLoad()` mit den möglichen Werten für API-Funktion `ExitWindowsEx` gefüllt. Zunächst die einfache Klasse zur Verknüpfung der textuellen Repräsentation mit den dazugehörigen Werten:

```
private class StrIntPair
{
  private string m_Name;
  private int m_Value;

  public StrIntPair(string g_Name, int g_Value)
  {
    m_Name = g_Name;    m_Value = g_Value;
  }
```

```
  public string Name { get { return m_Name; } }
  public int Value { get { return m_Value; } }
}
```

Wenn man Möglichkeiten wie die Kombination der Flags für den Aufruf von `ExitWindowsEx()` mit einer – notfalls gewaltsamen – Durchsetzung der eigenen Absichten (EWX_FORCE und EWX_FORCEIFHUNG) außen vor lässt, reichen vier Einträge:

```
private void Form1_Load(object sender, System.EventArgs e)
{
  comboBox1.Items.Add(new StrIntPair("EWX_LOGOFF", 0));
  comboBox1.Items.Add(new StrIntPair("EWX_SHUTDOWN", 1));
  comboBox1.Items.Add(new StrIntPair("EWX_REBOOT", 2));
  comboBox1.Items.Add(new StrIntPair("EWX_POWEROFF", 8));

  comboBox1.DisplayMember = "Name";
  comboBox1.ValueMember = "Value";
  comboBox1.SelectedIndex = 0;  // Logoff
}
```

Die Behandlungsroutine für Klicks auf die Schaltfläche `bShutdownWindows` übergibt den aktuell ausgewählten Wert aus `comboBox1` einer weiteren Methode und übernimmt auch gleich die Reaktion auf (an dieser Stelle recht unwahrscheinliche) Fehler:

```
private void bShutDownWindows_Click(object sender, System.EventArgs e)
{ int ExitWindowsFlags = ((StrIntPair) comboBox1.SelectedItem).Value;

  if (!CallExitWindows(ExitWindowsFlags))
    MessageBox.Show("ShutDown-Aufruf gescheitert", "ExitWindows-Demo", 0);
}
```

Ginge es ausschließlich um Windows 9x/Me, dann wäre `CallExitWindows()` mit der Deklaration der API-Funktion und einem einfachen Aufruf erledigt.

```
[DllImport("user32.dll", SetLastError=true)]
internal static extern bool ExitWindowsEx(int Flags, int Reserved);

public bool CallExitWindows(int ExWFlags)
{
#if Win9xOnly
  return ExitWindowsEx(ExWFlags, 0);
#else
  // ...
#endif
}
```

Nur – und wie nach dieser Vorrede wohl auch kaum anders zu erwarten – ist dem nicht so. Windows NT/2000/XP fordern mit ihrem Sicherheitsmodell für jede Aktion von `ExitWindowsEx()` mit Ausnahme eines schlichten Abmeldens (EXW_LOGOFF), dass der aufrufende Prozess das Privileg

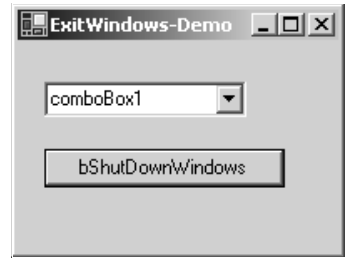


Abbildung 26.4: Das Formular des Codebeispiels

SE_SHUTDOWN_NAME hat. Dieses Privileg muss er erst einmal über eine weitere API-Routine namens `AdjustTokenPrivileges()` beantragen, die ihrerseits aber erst nach Öffnen des jeweiligen Prozess-Tokens verwendbar ist.

Wie das folgende, vollständige Listing zeigt, kommt mit Konstanten und dem von `AdjustTokenPrivileges` als Parameter erwarteten Strukturtyp doch einiges an Deklarationen zusammen:

```
[DllImport("user32.dll", SetLastError=true)]
internal static extern bool ExitWindowsEx(int Flags, int Reserved);

[DllImport("kernel32.dll")]
internal static extern IntPtr GetCurrentProcess();

[DllImport("advapi32.dll", SetLastError=true) ]
internal static extern bool OpenProcessToken(IntPtr h,
  int acc, ref IntPtr phtok);

[DllImport("advapi32.dll", SetLastError=true) ]
internal static extern bool LookupPrivilegeValue(string host,
  string name, ref long pluid);

[StructLayout(LayoutKind.Sequential, Pack=1)]
internal struct TokenPrivileges
{
  public int  Count;  // Zahl der Luid/Attr-Paare (hier: immer 1)
  public long  Luid;  // "locally unique identifier" (im Gegensatz zu GUID)
  public int  Attr;   // Attribut (enable, disable etc.)
}
```

Der Typ `TokenPrivileges` ist hier zur Abwechslung einmal nicht über `class`, sondern über `struct` deklariert. Die Konsequenzen dieser Deklaration sind (vgl. auch in Kapitel 7, Datentypen, den Abschnitt »Komplexe Werttypen«):

- Variablen des Typs `TokenPrivileges` werden nicht per `new` angelegt, sondern schlicht deklariert (und nicht auf dem Heap, sondern auf dem Stack gespeichert).
- Da es keinen Konstruktor gibt, müssen die Felder einzeln initialisiert werden.

➡ Die Übergabe an eine Routine wie `AdjustTokenPrivileges` (die einen Zeiger auf den Strukturtyp `TokenPrivileges` erwartet), ist mit einem schlichten `ref` (sowohl deklarierten Prototyp als auch im Aufruf) erledigt. Beim Einsatz von `class` wäre im Prototyp die Kennzeichnung mit `[In]` fällig (`bool DisableAl, [In] TokenPrivileges NewState, ...`), beim Aufruf entfiele das `ref`.

```
[DllImport("advapi32.dll", SetLastError=true)]
internal static extern bool AdjustTokenPrivileges(IntPtr hToken,
  bool DisableAll, ref TokenPrivileges NewState,
  int len, IntPtr PrevState, IntPtr relen );

  internal const int  SE_PRIVILEGE_ENABLED = 0x00000002;
  internal const int  TOKEN_QUERY    = 0x00000008;
  internal const int  TOKEN_ADJUST_PRIVILEGES = 0x00000020;

public bool CallExitWindows(int ExWFlags)
{
#if Win9xOnly
  return ExitWindowsEx(ExWFlags, 0);
#else
  bool Success = true;
  TokenPrivileges tp;
  IntPtr hToken = IntPtr.Zero;

  // Mit dem aktuellen Prozess verbundenes Token für Abfrage und
  // Veränderung öffnen. Dieser Aufruf scheitert unter Windows 9x/ME
  if (OpenProcessToken(GetCurrentProcess(),
    TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, ref hToken))
  {
    // gewünschte Veränderung: Privileg aktivieren
    tp.Count = 1; tp.Luid = 0; tp.Attr = SE_PRIVILEGE_ENABLED;
    // LUID für das Privileg "Shutdown" abfragen
    Success &= LookupPrivilegeValue(null, "SeShutDownPrivilege",
      ref tp.Luid);
    // Privilegien verändern
    Success &= AdjustTokenPrivileges(hToken, false, ref tp,
        0, IntPtr.Zero, IntPtr.Zero);
  }
  Success &= ExitWindowsEx(ExWFlags, 0);
  return Success;
#endif
}
```

Unter Windows 9x und Windows Me sind alle drei hier eingesetzten API-Routinen zur Manipulation von Prozess-Tokens zwar vorhanden (was ja recht erfreulich ist – ansonsten müsste man hier auch noch Ausnahmen abfangen), aber nur in Form leerer Hüllen, die grundsätzlich `false` als direktes Ergebnis zurückliefern. `CallExitWindows()` unterscheidet auf diese Weise zwischen den Windows-Versionen: Wenn `OpenProcessToken()` den Wert `false` zurückgibt, geht es aller Wahrscheinlichkeit nach um Windows 9x/Me

(und falls nicht: Wenn diese Routine scheitert, kann man sich die restlichen Vorbereitungen für `ExitWindowsEx()` auch unter Windows NT, XP und 2000 getrost sparen).



Das Funktionsergebnis `true` *zeigt hier lediglich an, dass das System nichts an den einzelnen API-Aufrufen zu bemängeln hatte. Das tatsächliche Herunterfahren des Systems findet asynchron statt – und wenn auch nur ein einziger Prozess sich nicht einverstanden erklärt (das heißt auf die von* `ExitWindowsEx()` *an alle Prozesse versandte Windows-Nachricht WM_QUERYENDSESSION mit »jetzt [noch] nicht« reagiert), wird die Operation abgebrochen. Wer unter allen Umständen eine Abmeldung und/oder das Herunterfahren bzw. Neustarten des Systems erreichen will, muss die Bitkonstante EWX_FORCE (Wert: 0x04) zusätzlich in den* `Flags`*-Parameter für* `ExitWindowsEx()` *einbauen:*

```
CallExitWindows(ExwFlags | 0x04);
```

26.5 Nicht gesicherter Code

Microsoft hat sich bei C# und der .NET-Laufzeitumgebung alle Mühe gegeben, das Zeigerkonzept... ja, eben: nicht mit Gewalt zu verbannen, wie es unter anderem Java tut, sondern überflüssig zu machen. Ein Beweis dafür, dass dieser Versuch weitgehend gelungen scheint: Meine Suche nach einem Beispielprogramm für diesen Abschnitt, bei dem Zeiger wirklich zwingend erforderlich sind, förderte wenig Substanzielles zu Tage. Die Betonung liegt hier auf dem Wort »zwingend«: Wer darauf besteht, die Spaltenordnung eines `ListView`-Steuerelements per `SendMessage` abzufragen und darüber stolpert, dass die dafür notwendige Struktur einen `char`-Zeiger enthält, kann entweder achselzuckend zum Schlüsselwort `unsafe` greifen oder – wie von Microsoft vorsehen – entsprechende Methoden der Klasse `Marshal` benutzen. `AllocHGlobal()` belegt Speicher auf dem globalen Heap von Windows, `PtrToStringAnsi()` konvertiert den Inhalt eines solchen Speicherbereichs in einen String, `FreeHGlobal()` gibt den globalen Speicherbereich wieder frei.

Zugegeben sprengen derartige Techniken auch den Rahmen dieses Buchs, und außerdem geht es ja nicht immer um API-Funktionen: Vor allem C-Programmierer werden schon fast zwangsläufig eigene Bibliotheksroutinen in petto haben, die nicht nur mit Zeigern auf Strings arbeiten, sondern auch mit Zeigern auf Zeiger-Arrays, deren Elemente auf Strukturen mit Zeigern verweisen. Und wer eine in C/C++ oder Delphi höchst effizient verwaltete, hierarchisch organisierte Liste mit den dazugehörigen Iterations- bzw. Sortierverfahren in seinem Fundus hat, mag sich sicher eher dazu bewegen lassen, den Code nach C# zu portieren, wenn dabei nicht als Erstes die gesamte Logik auseinander fällt.

26.5.1 Voraussetzungen

Damit der C#-Compiler »unsicheren« Code akzeptiert, sind zwei bzw. drei Voraussetzungen zu erfüllen:

1. Die explizite Kennzeichnung von Zeigerdeklarationen usw. (mehr dazu weiter unten) mit dem Schlüsselwort `unsafe`.
2. Die Angabe des Compilerschalters `/unsafe` beim Aufruf des Compilers über die Kommandozeile bzw. die Einstellung UNSICHERE CODEBLÖCKE ZULASSEN im EIGENSCHAFTEN-Fenster des Projekts – siehe Abbildung 26.5. Bei dieser Einstellung sollte man darauf achten, in KONFIGURATION (links oben im Dialogfeld) vorher ALLE KONFIGURATIONEN auszuwählen, sonst gibt es beim späteren Umschalten vom Debug- in den Release-Modus erst einmal eine Menge Fehlermeldungen.
3. Die dritte Voraussetzung gilt nur für verteilte Anwendungen und macht sich erst zur Laufzeit bemerkbar: Module mit unsicherem Code, die nicht auf dem lokalen System gespeichert sind, führt die .NET-Laufzeitumgebung nur dann aus, wenn sie von einem Benutzer mit Administratorrechten für das jeweilige Netzwerk signiert sind. (Was auch bedeutet: Über eine Internetverbindung lassen sich solche Module bestenfalls dann betreiben, wenn eine VPN-Verbindung zu einem LAN besteht – oder man sämtliche Sicherheitsmechanismen der Laufzeitumgebung abschaltet.)

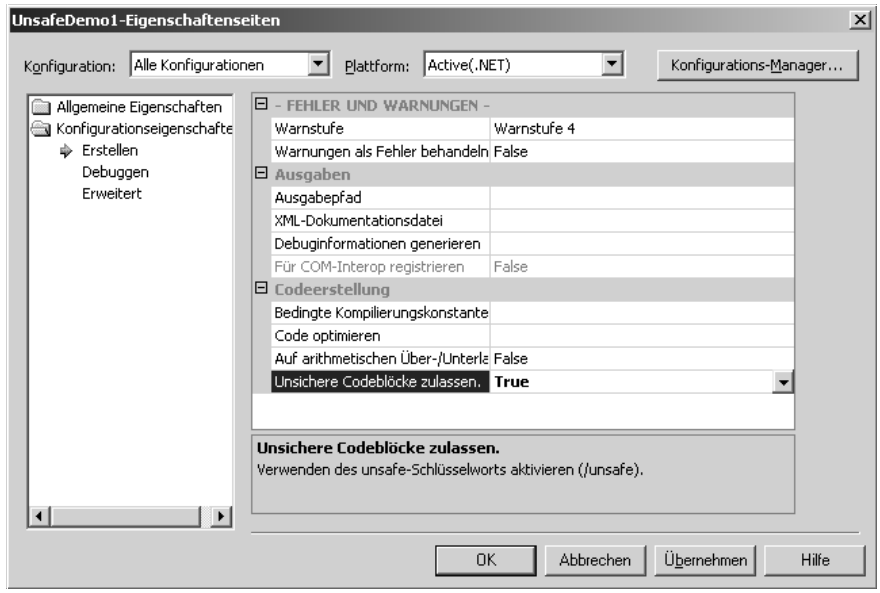


Abbildung 26.5: Das Äquivalent zum Schalter /unsafe in den Projekteigenschaften

Unsicher vs. unverwaltet

Dass .NET für unsicheren Code also einiges an Drumherum erwartet, sich bei Aufrufen x-beliebiger DLLs dagegen mit schlichten Deklarationen zufrieden gibt, mag zur Annahme verleiten, dass unsicherer Code automatisch auch unverwaltet sei. Etwas verwirrender noch: Bei importierten DLL-Funktionen hat die .NET-Laufzeitumgebung definitiv keine Kontrolle darüber, welche Operationen ausgeführt werden (weshalb sie eigentlich per se als unsicher zu betrachten wären). Tatsächlich scheint man hier der Praxis den Vorzug vor stringenter Logik gegeben zu haben: DLL-Aufrufe finden mit unverwalteten Daten statt, kommen aber ohne die explizite Deklaration als »unsicher« (und ihre Folgen) aus, und als `unsafe` gekennzeichneter Code arbeitet sehr wohl mit dem CTS, so lange man nicht über Methoden wie `Marshal.AllocHGlobal()` explizit dafür sorgt, dass dort unverwaltete Speicherbereiche ins Spiel kommen.

26.5.2 Syntax und Geltungsbereiche

Die Kennzeichnung von Code mit `unsafe` kann zum einen auf der Ebene der Klasse oder einzelner Methoden geschehen:

```
public unsafe class xyz
{  // alle Methoden, Deklarationen usw. innerhalb der Klasse
}

public class class Form1 : System.Windows.Forms.Form
{
  // ...
  unsafe void button1_Click(object sender, System.EventArgs e)
  {
    // alle Deklarationen und Operationen innerhalb der Methode
  }
}
```

Wie das zweite Beispiel zeigt, geht `unsafe` nicht in den Prototyp ein: Der Compiler beschwert sich nicht, obwohl es bei `button1_Click` um eine per Delegat aufgerufene Methode geht.

Bei Typvereinbarungen für Klassen und Strukturen ist `unsafe` entweder für die gesamte Deklaration anzugeben oder für jedes Element einzeln – lokale Blockstrukturen sind hier generell nicht erlaubt.

```
public class xyz
{
  int x;
  unsafe
  {               // Compiler-Fehlermeldung: Typ wird erwartet
    int* a;
    char* b;
  }
  // ...
```

Lokale Variablen lassen sich hingegen jederzeit in `unsafe`-Blockstrukturen vereinbaren und es versteht sich, dass hier keine anderen Regeln gelten als für Anweisungsblöcke generell.

```
public void MyFunc()
{ int MyInt; char MyChar;
  unsafe
  { // von der Syntax her zulässig, ABER:
    int* p = &MyInt;
    char* x = &MyChar;
    // Anweisungen
  }            // hier endet der Geltungsbereich von p und x

  p* = 25;     // Compiler-Fehlermeldung: p ist unbekannt
}
```

Obwohl man sinnvollerweise bestrebt sein wird, als `unsafe` gekennzeichnete Codeblöcke möglichst klein zu halten, dürfte es wohl in der Praxis so aussehen, dass man entweder komplette Methoden auf diese Art deklariert – oder eben lokal konzentrierte Code-Abschnitte, in denen sich dann im schönsten C-Chinesisch arbeiten lässt.

```
public void MyFunc()
{
  // ...
  if (x ==4)
  unsafe
  { char* p = stackalloc char[256], p1 = p;
    for (int z = 0; z < 256; z++, p1++)
      *p1 = char(z);
    // ...
  }
```

Hier wird mit dem (ebenfalls nur bei `unsafe` erlaubten) Operator `stackalloc` Platz für 256 Zeichen auf dem Stack reserviert, dessen Startadresse im `char`-Zeiger `p` landet; ein zweiter `char`-Zeiger namens `p1` bekommt danach den Wert von `p` zugewiesen. Die darauffolgende Schleife initialisiert den Speicherbereich mit den Unicodes von 0 bis 255.



Wie auch in der Online-Dokumentation beschrieben, gibt es bei der Deklaration von Zeigervariablen einen sicher gewöhnungsbedürftigen Unterschied zwischen C/C++ und C#:

```
// C/C++
char *p, p1;   // char-Zeiger p, char-Variable p1
char *p, *p1;  // char-Zeiger p, p1

// C#
char* p, p1;   // char-Zeiger p, p1
```

Klassen mit unsicheren Methoden

Dieser Abschnitt existiert hauptsächlich, weil in den Usenet-Newsgroups des öfteren die Frage gestellt wird, ob der Einsatz einer Klasse mit unsicheren Methoden die Kompilierung des gesamten Programms mit */unsafe* bzw. entsprechenden Projekteinstellungen notwendig macht. Die Antwort ist ein klares Nein: Die Kompilierung in diesem Modus ist nur dann notwendig, wenn das Programm derartige Methoden auch aufruft. Ein Beweis dafür ist die Klasse `string`, die praktisch in jedem Programm mit von der Partie ist – und mehrere als `unsafe` deklarierte Konstruktorvarianten wie beispielsweise `unsafe public String(char*)` zur Verfügung stellt. Erst wenn man eine dieser Varianten tatsächlich benutzt, wird der Schalter `/unsafe` fällig und wirkt sich dann aber auch nur auf den mit `unsafe` deklarierten Code aus. Der */unsafe*-Schalter per se macht also eine Assembly nicht zum unsicheren Code. Er erlaubt letztlich nur den Einsatz des `unsafe`-Modifizierers.

26.5.3 Möglichkeiten und Grenzen

Die Online-Dokumentation erläutert recht breit, welche zusätzlichen Möglichkeiten unsicherem Code zur Verfügung stehen, und gibt Beispiele dazu. Die folgende Auflistung wird deshalb nur bei den Dingen ausführlicher, die nicht in dieser Dokumentation zu finden sind:

- `*`-Operator zur Dereferenzierung von Zeigern:

```
int *p = &MyInt;
*p = 4;  // setzt MyInt auf 4
```

- `->`-Operator zum Zugriff auf Felder einer Struktur:

```
struct MyStruct
{ int a, b;
}
MyStruct xyz;   // Variable des Typs MyStruct
MyStruct* Pxyz = &MyStruct;  // Zeiger auf MyStruct

Pxyz->a = 4;    // xyz.a = 4;
```

- `[]`-Operator zur Indizierung (mit Zeigerarithmetik):

```
char* pC; int* pI;
// ...
pI[7] = 4;  // = (* int)((int)pI + 7 * sizeof(int))* = 4
pC[2] = "a";  // = (* char) ((int) pC + 2 * sizeof(char))* = "a"
```

- `&`-Operator zur Ermittlung der Adresse einer Variablen:

```
char c; char* pC = &c;
```

Bei Variablen auf dem Stack ist die Adresse unveränderlich, bei Datenfeldern von Objektinstanzen, die ja auf dem Heap gehalten werden, sieht es dagegen anders aus, weil dort die Garbage Collection (fast) jederzeit für Verschiebungen sorgen kann. Die Dokumentation empfiehlt deshalb, für derartige Instanzen grundsätzlich das (wiederum nur bei `unsafe` verwendbare) Schlüsselwort `fixed` einzusetzen:

```
public class MyClass
{
  public int x;
}

private unsafe void button1_Click(object sender, System.EventArgs e)
{ MyObj = new MyClass();

  fixed(int* Px = @MyObj.x)
  {
    // ...
    *Px = 4;
  } // ab hier kann die Garbage Collection MyObj wieder verschieben
```

Wirklich notwendig ist ein derartiges »Festnageln« nur, wenn die .NET-Speicherverwaltung zur Lebenszeit der Zeigervariablen durch das Anlegen neuer Objektinstanzen direkt gefordert wird. (Die Garbage Collection arbeitet über einen Thread mit niedrigster Priorität – einfach dem laufenden Code dazwischenplatzen kann sie deshalb nicht.)

- Operatoren ++ und -- zum Erhöhen und Erniedrigen von Zeigern:

```
int *pI;
char *pC;

pI++;  // = (uint)pI += 4;
pC--;  // = (uint)pC -= 2;
```

- Operatoren + und – für arithmetische Operationen mit Zeigern:

```
int *pI;
char *pC;

pI += 4;  // = (uint)pI += sizeof(int)*4;
pC -= 2;  // = (uint)pC -= sizeof(char)*2;
```

- Operatoren ==, !=, <, >, <= und => für Adressvergleiche:

```
Int* p, pStart, pEnd;

for (p = pStart; p < pEnd; p++)
  *p = ...;
```

- `stackalloc`-Operator zum Reservieren von Speicher auf dem Stack:

```
int* p = stackalloc int[200];  // Platz für 200 ints
```

Hier ist bei größeren Arrays Vorsicht angesagt, weil die .NET-Laufzeitumgebung pro Thread (also auch für den primären Thread einer Anwendung) rund 1 MByte Stack reserviert – unabhängig davon, wie viel virtueller Speicher tatsächlich auf dem Zielsystem zur Verfügung steht.

- `sizeof`-Operator zur Ermittlung des tatsächlichen Platzbedarfs von Strukturen und Variablen:

```
int IntSize = sizeof(int);  // 4
```

Eine technische Notwendigkeit, `sizeof` ebenfalls die Hürde `/unsafe` voranzustellen, gibt es sicher nicht. Dieser Operator dürfte allerdings einer der wenigen sein, für den es sich lohnt, die Kompilierung »unsicheren« Codes auch in einem normalen Programm zuzulassen – und dort ausschließlich auf den Debug-Modus zu beschränken. Beispielsweise:

```
#if DEBUG
  if (sizeof(MyVar) == 2)
  { // Debug-Prüfungen für Unicode
  }
 else
  { // Debug-Prüfungen für ANSI
  }
#endif
```

Anhang

Anhang A: Erfahrungen mit der Installation 1107
Anhang B: Einsatz des Debuggers 1115
Anhang C: Internet-Protokolle 1125
Anhang D: Was ist auf der CD-ROM? 1139

Von einer Übersicht über die Begleit-CD abgesehen ist der Anhang kein Nachschlagewerk: Hier geht es vielmehr um Dinge, die thematisch nicht so recht in die Buchteile gepasst haben – wie Details zu Internetprotokollen (TCP, IP, HTTP) und ihre Untersuchung im Netzwerk-Monitor, Fehlersuche bei paralleler Entwicklung mit verschiedenen Sprachen (Delphi, MSVC, C#) und die Installation von .NET sowohl auf der Entwicklungsmaschine als auch auf den jeweiligen Zielsystemen.

A Erfahrungen mit der Installation

Obwohl man von einem Entwicklungssystem mit Laufzeitumgebung und Debugger erwarten darf, dass es mehr Veränderungen am System vornimmt als eine Textverarbeitung, kommt man bei der Installation von Visual Studio .NET mit maximal zwei Neustarts des Systems aus, wenn alles gut geht.

Mit den Checklisten dieses Anhangs sollten Sie dafür sorgen können, dass es auch Ihre Installation auf der von Microsoft vorgegebenen Ideallinie ins Ziel schafft. Entstanden sind diese Listen auf der Basis eigener – nicht immer schmerzfreier – Erfahrungen und dem Durchkämmen der einschlägigen Gruppen im Usenet.

A.1 Das Entwicklungssystem

Microsoft gibt auf `http://msdn.microsoft.com/vcsharp/productinfo/sysreq.asp` die folgenden Minimalvoraussetzungen für die Installation von Visual Studio .NET an:

- Betriebssystem: XP Professional, 2000 Professional, 2000 Server, NT 4.0 Workstation oder NT 4.0 Server.
- Hauptspeicher: 64 MByte für Windows NT 4.0 Workstation, 96 MByte für 2000 Professional, 160 MByte für XP, 192 MByte für die Server-Varianten.
- Freier Platz auf der Festplatte: 600 MByte auf dem Systemlaufwerk, 2 GByte auf dem Installationslaufwerk.
- Grafikkarte mit 800x600 in 256 Farben.
- Prozessor: PII-450.

Windows 9x und Me sind als Entwicklungssystem definitiv nicht vorgesehen; bei Windows XP Home scheitert die Installation an den Erweiterungen für Front Page, die ihrerseits einen laufenden Internet Information Server voraussetzen, leider komplett.

Eine praxisgerechte Ausstattung sieht wie üblich ein wenig anders aus: 256 (besser: 512) MByte RAM, 3 GByte freier Platz auf der Festplatte und

1024x768 in Hi- oder TrueColor als Minimum. Eine Desktop-Größe von 1280 x 1024 ist bei der Entwicklung verteilter Anwendungen oder von Webdiensten wahrlich kein überflüssiger Luxus: Angesichts eines halben Dutzends aktiver Fenster hätte sich unsereins auch öfter mal 1600x1200 gewünscht (plus Großvater-Leselupe, wenn man das auf einem 19er-Bildschirm versucht).

Beim Prozessor scheint Microsoft noch mehr als bei der Grafikkarte als Zielgruppe den überaus duldsamen Masochisten im Auge zu haben. Es entspricht sicher der Wahrheit, dass VS.NET auch mit einem PII-450 nach einigen Stunden den Startbildschirm fast vollständig aufgebaut hat: Tatsache ist aber, dass sich einer der Autoren durch das Start- und Compilierverhalten von Visual Studio .NET nach einigen Wochen Fingertrommelns endgültig motiviert sah, seinen PIII-550 gegen einen Pentium 4 mit 2 GHz auszuwechseln, bei dieser Gelegenheit von 256 auf 512 MByte RAM aufzustocken – und diesen doch recht tiefen Griff in die Brieftasche wirklich nicht bereut hat. (Das gilt für Windows 2000 Advanced Server – unter NT 4, speziell der Workstation-Version, sollte man auch mit ein oder zwei PS weniger unter der Haube auskommen.)

A.1.1 Internetanbindung und Service Packs

Bei älteren Microsoft-Produkten fiel eine permanente Internetanbindung noch in die Kategorie wünschenswertes, aber optionales Extra. Bei Visual Studio .NET ist eine Verbindung zum Internet fast schon Pflicht – nicht nur wegen den unvermeidlichen Service-Packs. Vorausgesetzt wird:

- Windows 2000: Service Pack 2. Bei Drucklegung war bereits das Service Pack 3 verfügbar (Umfang: 33 MByte in der Express Installation, 124 MByte komplett)
- Windows NT 4: Service Pack 6 (Umfang des aktuellen und letzten Service Packs 6a: 34 MByte sowohl für die Client- als auch die Server-Version)
- Windows XP: Zum Zeitpunkt der Drucklegung war hier noch kein Service Pack gefordert.

Das Service Pack 2 für die .NET-Laufzeitumgebung nimmt sich mit seinen 6 MByte Umfang gegenüber den anderen Downloads noch recht bescheiden aus und dürfte auch bei einer ISDN-Verbindung kein allzu großes Problem darstellen. Was ISDN-Einwahlverbindungen dagegen allgemein im Zusammenhang mit Visual Studio .NET etwas problematisch macht, ist die Online-Dokumentation: Sie zeigt nicht direkt an, ob ein Link lokal auflösbar ist oder zur MSDN-Website führt. (Wenn also nach einem Klick der Einwahldialog erscheint, dann sind die gesuchten Daten eben nicht auf der eigenen Platte.)

A.1.2 Zeitbedarf und Umfang der Installation

Dieser Punkt bekommt deshalb eine eigene Überschrift, weil VS.NET auch den bisherigen Rekordhalter in dieser Disziplin (Office 2000 unter Windows 9x) um Längen schlägt: Rechnen Sie je nach Aktualitätsgrad Ihres Systems mit *anderthalb bis zweieinhalb Stunden*. Die von den Fortschrittsanzeigen des Installationsdialogs angegebenen Schätzungen sind, mit Verlaub gesagt, Hausnummern.

Nicht alle der über die Online-Dokumentation erreichbaren Beispielprogramme werden im Rahmen einer Standardinstallation auf die Festplatte kopiert, weshalb man bei ausgedehnterer Sucherei öfter einmal die Aufforderung erhält, die Installations-CD bzw. DVD einzulegen. Einer der Autoren hat deshalb einfach die gesamte DVD erst einmal auf die Platte kopiert und von dort aus die Installation gestartet – seitdem ist an dieser Front Ruhe, wenn auch auf Kosten von immerhin 3,1 GByte zusätzlich belegtem Speicherplatz für die Enterprise-Version.



Sämtliche Installationsdateien in ein Verzeichnis auf der Festplatte zu kopieren und die Installation von dort aus zu starten, scheint auch die einfachste Lösung für die Probleme zu sein, von denen gelegentlich im Zusammenhang mit der auf vier CDs ausgelieferten Version von Visual Studio .NET berichtet wird –vorausgesetzt natürlich, Sie können etwas über 3 GByte Plattenplatz erübrigen. Wenn Sie die Wahl haben, verwenden Sie als Speicherort für Installationsdateien und Ziel der Installation zwei unterschiedliche physische Laufwerke. Da im Verlauf der Installation über 17000 meist kleine Dateien angelegt werden, kommt ein einzelnes IDE-Laufwerk arg ins Schwitzen und verlängert die Affäre um schätzungsweise eine halbe Stunde. (Mit SCSI-Laufwerken dürfte das erheblich unkritischer sein.)

A.1.3 Konflikte mit anderen Programmen

Wie bei einem Paket des Umfangs von Visual Studio .NET wohl auch nicht anders zu erwarten, kann es mit einer Reihe anderer Anwendungen Probleme geben, an denen nicht einmal immer Microsoft selbst Schuld ist.

Antiviren-Software

Sowohl bei der Installation von VS.NET als auch im laufenden Betrieb der Entwicklungsumgebung sind Scripts an einigen wenigen (aber eben leider kritischen) Stellen mit von der Partie, weshalb das Script Blocking von Virenschutzprogrammen wie Norton Antivirus 200x ausgeschaltet werden muss – und das eben leider dauerhaft. Wieso man bei Microsoft meinte, darauf nicht verzichten zu können, wird wohl das Geheimnis der Entwickler bleiben: Einer der Autoren sah sich in Folge dazu gezwungen, sein ansonsten durchaus wohlgelittenes Outlook Express durch Eudora zu ersetzen (und die anderen beiden arbeiten sowieso seit längerem mit Netscape).



Die von der Symantec-Website herunterladbare, auf 30 Tage befristete Trialware-Version von Norton Antivirus 200x ist auch unter den Server-Versionen von Windows 2000 lauffähig – im Gegensatz zur Kaufversion dieses Pakets, die sich unter Windows 2000 Advanced Server mit dem freundlichen Hinweis verabschiedet, man möge doch bitte die (ungleich teurere) »Corporate Edition« erwerben. Das Problem, dass sich die Trialware nach Ablauf von 30 Tagen zwar abschaltete, Scripts aber nach wie vor blockierte und sich zu allem Überfluss auch nicht mehr richtig deinstallieren ließ, scheint nicht nur einer der Autoren gehabt zu haben, weshalb die Firma inzwischen ein separates Deinstallationsprogramm namens rnav.exe zum Download bereitstellt. (Erreichbar ist dieser Download über `http://www.symantec.com` *und dort die Eingabe »uninstall antivirus« in der Suchfunktion der Abteilung »Service & Support«.)*

Upgrades von Visual Studio Trialware und/oder Betaversionen

Erstaunlicherweise ist hier keiner der sonst üblichen Migrationspfade vorgesehen. Der einzig gangbare Weg: Deinstallieren Sie die alte Version komplett, bevor Sie die neue Version aufspielen. Bei einem der Autoren findet sich trotzdem in SYSTEMSTEUERUNG/SOFTWARE noch ein Icon für eine ältere Version – was aber daran liegen mag, dass es dort um die US-Version von VS.NET ging. Das Mischen unterschiedlicher Sprachversionen scheint – wieder einmal – allgemein ein Problem zu sein, das man am besten vollständig umschifft.

Parallelinstallation mit Visual Studio 6

Visual Studio 6 sollte vor VS.NET installiert werden, da VS.NET eine ganze Reihe von Komponenten aktualisiert – ansonsten gibt es einige kleine Überschneidungen: So wurde beispielsweise davon berichtet, dass Systemprozesse nicht mehr im Debugger von VS 6 sichtbar sind, aber auch eine ganze Reihe fertiger Anwendungen mit verschiedenen neuen DLLs nicht oder teilweise nicht mehr laufen.

Für die Entwicklung muss man in jedem Fall die Hilfestellungen voneinander trennen, auch wenn das Plattenplatz im Gigabyte-Bereich kostet: Die letzte Ausgabe der MSDN Library, die sich noch mit Visual Studio 6 verträgt, hat das Datum Oktober 2001 – mit neueren Versionen kommt VS 6 nicht mehr zurecht.

Für den Einsatz der Kommandozeilen-Compiler und verschiedener nur in der Kommandozeilenversion verfügbarer Tools beider Entwicklungsumgebungen sind separate Eingabeaufforderungen notwendig, die über eigene Varianten von MSCVARS.BAT zahlreiche Umgebungsvariablen wie LIB und INCLUDE setzen.

Nach einer Installation von VS.NET mit Visual Studio 6 angelegte Packages sollten auf jeden Fall auf einem weiteren Rechner (ohne VS.NET) geprüft oder im Idealfall gleich dort angelegt werden, weil ansonsten zwangsläufig einige Aktualisierungen dazukommen, die man zwar für VS.NET, nicht aber für MSVC braucht – beispielsweise Internet Explorer 6 und MDAC.

Andere Entwicklungsumgebungen

Auf dem System eines der Autoren (raten Sie mal, welchem) läuft neben VS.NET und Visual Studio 6 auch noch Delphi 6. Abgesehen davon, dass sich die Entwicklungsumgebungen beim Start gerne einmal beschweren, ihr eigener JIT-Debugger sei nicht gesetzt, scheint das eine problemlose Angelegenheit zu sein.

A.1.4 Windows Component Update

Der einzige echte Fallstrick, über den einer der Autoren dann auch prompt gestolpert ist, verbirgt sich im Installationsdialog. Wie Abbildung A.1 zeigt, ist der zwar in drei Schritte unterteilt – er hakt die bereits ausgeführten Teile aber nicht ab oder deaktiviert sie. Wenn nach dem Component Update und dem daraufhin fälligen Neustart der Schritt 1 der Installation ein weiteres Mal in leuchtenden Farben prangt, liegt an sich die Überlegung nahe, dass dieses Update dann wohl noch nicht abgeschlossen sei. Was das Setup bei einem weiteren Klick auf dieses Icon allerdings tatsächlich tut: Es fährt ein und dasselbe Component Update erneut ab – nicht mehr und nicht weniger.



Abbildung A.1: Die einzelnen Schritte im Installationsdialog bleiben nach ihrer Ausführung weiterhin wählbar

A.2 Die Clientseite: .NET Framework

Die .NET-Laufzeitumgebung – nicht zu verwechseln mit dem im Lieferumfang von VS.NET enthaltenen, wesentlich umfangreicheren .NET Framework SDK – steht über `http://msdn.microsoft.com/netframework/default.asp` zusammen mit dem zum Zeitpunkt der Drucklegung – August 2002 – aktuellen Service Pack 2 zum kostenfreien Download zur Verfügung. Das Framework umfasst gut 20 MByte, das Service Pack 6 MByte. Eine Version dieses Frameworks, in die das Service Pack 1 oder 2 bereits integriert ist, scheint nicht geplant zu sein, man muss also beide Packungen separat herunterladen und – ärgerlicher – das Zielsystem zweimal neu starten. (Da das Service Pack seinerseits anscheinend die .NET-Laufzeitumgebung benutzt, bekommt man bei ihm vor dem ersten Neustart nach der Installation des Basispakets lediglich eine Systemfehlermeldung zu sehen.)

Im Gegensatz zur Entwicklungsumgebung ist die .NET-Laufzeitumgebung unter *allen* aktuellen Windows-Versionen lauffähig – also auch unter Windows 98/Me sowie der XP Home Edition und den Desktop-Varianten von Windows 2000. Für NT 4.0 wird das Service Pack 6a (nicht etwa: Service Pack 6) vorausgesetzt, für Windows 2000 das Service Pack 2 »empfohlen«.

Explizite Angaben, wie viel Hauptspeicher erforderlich seien, macht Microsoft nicht. Eigene, zugegeben recht flüchtige Tests ergaben einen Basisbedarf von rund 10 MByte – wobei einige DLLs der Laufzeitumgebung anscheinend auch über das Ende der .NET-Anwendung hinaus im Speicher bleiben.

Eine weitere Voraussetzung ist der Internet Explorer in der Version 5.01 (wobei sich hier schon aufgrund der bekannten Sicherheitsprobleme die Installation der Version 6.0x empfiehlt).

Soll ein System als Server für verteilte Anwendungen betrieben werden (was hier auch ohne weiteres mit Windows 98 oder Me möglich ist), sind einige weitere Voraussetzungen zu erfüllen:

- Für ADO.NET: Auf dem Server muss mindestens die Version 2.6 der Microsoft Data Access Components (MDAC) installiert sein; empfohlen wird – wen wundert's – die aktuelle Version 2.7.
- Für ASP.NET: Der Internet Information Server mit den aktuellen Sicherheits-Updates.

Beide Zusätze müssen vor der Installation des .NET Framework installiert sein. Den Hinweis, dass die IIS nicht ohne Sicherheits-Updates auf das eigene System losgelassen werden sollten, hat einer der Autoren erst einmal großzügig beiseite gewischt – und teuer bezahlt, nämlich mit einer Neuinstallation ab FDISK. Auch wenn der Nimda-Virus inzwischen passé ist: Wenn Sie dieses Buch in den Händen halten, wird unter Garantie irgendein anderes menschliches Trauerwürstchen erfolgreich versucht haben, seine »15 Minuten Berühmtheit« zu erzwingen.

B Einsatz des Debuggers

Was bei anderen Programmiersprachen des Programmierers täglich Brot ist – nämlich das Durchkämmen des Speichers nach verloren gegangenen Objekten und Strukturen, die Suche nach verwaisten oder mehrfach freigegebenen Speicherbereichen etc. – fällt bei C# aufgrund des CTS, der referenzbasierten Lebenszeit und des Sprachdesigns weitgehend in die Kategorie Exotika. Umso erfreulicher, dass der Debugger von Visual Studio .NET hier trotzdem *mehr* Unterstützung liefert als alle seine Vorgänger aus dem Hause Microsoft zusammen.

Da die automatische Überwachung, die Anzeige lokaler Variablen und die Aufrufliste (im Programmierer-Jargon: der Aufruf-Stack) weitgehend selbsterklärend sind und sich auch dieser Anhang nicht als Wiederholung der Online-Hilfe verstehen will, geht es hier nach einem (ultrakurzen) Abriss der Grundfunktionen um einige Dinge, die den Autoren als nicht so offensichtlich erschienen. Das restliche Material konzentriert sich auf die Umgebung eines Programms, bei der C# – und damit die Online-Dokumentation – nur noch teilweise eine Rolle spielen.

B.1 Grundfunktionen des Debuggers

Zur Ausführung eines C#-Programms unter Kontrolle des Debuggers müssen die folgenden Voraussetzungen erfüllt sein:

- Kompilierung im Debug-Modus, wobei die Symbole DEBUG und TRACE definiert sein sollten (Standardvorgabe)
- Start über DEBUGGEN/STARTEN bzw. F5

Kompilierung im Debug-Modus

Dieser Modus stellt die Standardvorgabe für neue Projekte dar. Die Umschaltung zwischen Debug- und Release-Modus geschieht über den Menübefehl ERSTELLEN/KONFIGURATIONS-MANAGER.

Die Eigenschaften für den Debug-Modus werden über die EIGENSCHAFTSSEITEN des Projekts festgelegt, die mit dem Menübefehl PROJEKT/EIGENSCHAFTEN erreichbar sind. (Dort findet sich ein zweiter Weg zum Konfigurations-

Manager: Über die gleichnamige Schaltfläche, die allerdings erst aktiv wird, nachdem man den Abschnitt »Konfigurationseigenschaften« gewählt hat.)

Bedingte Symbole zur Kompilierung und Ablaufverfolgung

Für den Debug-Modus sind standardmäßig die beiden Symbole DEBUG und TRACE definiert und klammern ihrerseits die (statischen) Methoden der `Debug`- und `Trace`-Objekte, die beide im Namensraum `System.Diagnostics` definiert sind. In ein Programm eingestreute Aufrufe dieser Methoden zur Ablaufverfolgung – beispielsweise `Trace.WriteLine` – bleiben ohne das Symbol DEBUG folgenlos. Die (hypothetische) Implementation von `Trace` dürfte so aussehen:

```
namespace System.Diagnostics
{
  public class Trace
  {
    public static void WriteLine(...)
    {
      #if TRACE
      // ...
     #endif
    }
// ...
```

Aus diesem Grund ist eine eigene Klammerung von Trace-Aufrufen nicht notwendig – anstelle von

```
using System.Diagnostics;
// ...
#if TRACE
  Trace.WriteLine(NewElem == null, ...);
#endif
```

können Sie also schreiben:

```
Trace.WriteLine(NewElem == null, ...);
```

Für die Klasse `Debug` und das Symbol DEBUG gilt Analoges.



`Trace` und `Debug` wickeln Ausgaben über die API-Funktion `OutputDebugString()` ab und sind deshalb zur Ablaufverfolgung in Multithread-Programmen nur bedingt brauchbar, weil diese Funktion synchronisiert zum primären (»GUI«-)Thread der Anwendung läuft.

Von `http://www.sysinternals.com` – einer Website, die allgemein eine unschätzbare Fundgrube für die Windows-Programmierung darstellt – ist unter anderem ein Prögrämmchen herunterladbar, das derartige Ausgaben in einem eigenen Fenster sammelt. Bitte beachten: Beim Start einer

Anwendung innerhalb der Entwicklungsumgebung leitet VS.NET `OutputDebugString()` *zur Darstellung im eigenen Ausgabe-Fenster selbst um (und verzichtet leider auf das Weiterreichen).*

Die in der Online-Dokumentation aufgestellte Behauptung, die Klassen `Trace` *und* `Debug` *seien erst nach Definition der entsprechenden Symbole verfügbar, ist irreführend. Richtiger wäre: Aufrufe der statischen Methoden dieser beiden Klassen bleiben folgenlos, wenn die entsprechenden Symbole nicht definiert sind.*

Haltepunkte

Haltepunkte lassen sich mit F9 oder einem Klick in die linke Spalte des Editors sowohl setzen als auch wieder entfernen, was sowohl vor als auch erst nach dem Start einer Anwendung geschehen kann. Voraussetzung ist, dass das Programm im Debug-Modus kompiliert und gestartet wurde. Beim Erreichen eines Haltepunkts übernimmt der Debugger die Kontrolle, zeigt automatisch im Fenster LOKAL die an der unterbrochenen Anweisung beteiligten Variablen an und ermöglicht unter anderem die gezielte Veränderung von Werten. Wie Abbildung B.1 zeigt, geschieht die Unterbrechung *vor* Ausführung der Anweisung: Das Feld `TS.s3` hat dort offensichtlich noch den Wert 0.

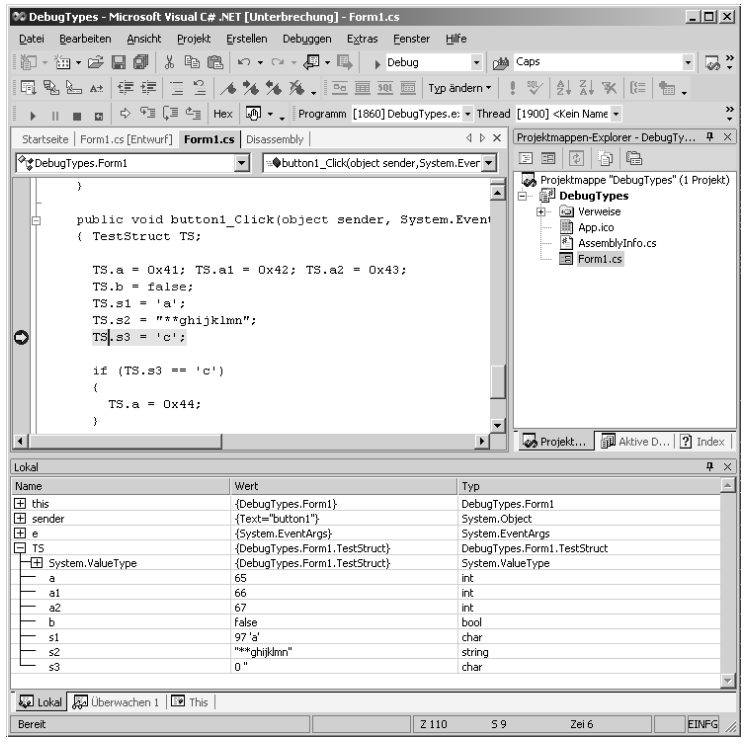


Abbildung B.1: Wie am Wert von `TS.s3` zu sehen, findet die Unterbrechung vor der Ausführung der Anweisung statt.

Die Fortsetzung eines per Haltepunkt unterbrochenen Programms geschieht übrigens wiederum mit F5 (oder über den Menübefehl DEBUGGEN/WEITER, der den »normalen« Menübefehl DEBUGGEN/STARTEN ersetzt).



Ein Feature sucht man bei Visual Studio .NET vergebens: Die Möglichkeit, das Programm an einer bestimmten Stelle fortzusetzen – etwa, nachdem man nach einer Endlos-Verfolgung in den falschen Zweig einer `if`*-Anweisung getappt ist. Ein solches »Hoppla, zurück!« ist bei Delphi und MSVC auf dem Umweg über das Fenster Disassembly und eine gezielte Veränderung des Programmzählers möglich, bei VS.NET dagegen nicht, weil im (über DEBUG/FENSTER zu erreichenden) Fenster REGISTER das EIP-Register schlicht fehlt. Schade.*

Speicherauszüge im Rohformat

In der besten aller Welten würde man spätestens mit C# auch bei der Fehlersuche ausschließlich mit Objekten und strenger Typisierung auskommen. Leider ist dem noch lange nicht so: Vor allem bei Aufrufen von API-Funktionen ist eben doch noch manchmal der Blick aufs Rohformat der verwendeten Strukturen angesagt. Während man sich bei MSVC für so etwas mit wüsten expliziten Typumwandlungen und Zusätzen á la `*((char *)&p->Data),20` im Debugger herumschlagen musste, haben die Entwickler bei VS.NET ganze Arbeit geleistet: Über DEBUGGEN/FENSTER/ARBEITSSPEICHER stehen vier voneinander unabhängige Fenster zur Anzeige von Speicherinhalten im Rohformat zur Verfügung.

Die Eingabe der Startadresse eines Speicherauszugs ist anfänglich etwas gewöhnungsbedürftig: Die Namen von `struct`- und `class`-Instanzen führen zur sofortigen Anzeige der Adresse der jeweiligen Instanz; mit einem Klick auf die rechts neben dem Eingabefeld angebrachte Schaltfläche AKTUALISIEREN kann man zwischen Namen und Adressen umschalten. Für einfache Variablen und Felder von Strukturen gilt dagegen der jeweilige Wert als Startadresse.

Anders gesagt: Im gegebenen Beispiel kommt bei der Eingabe von »TS« die Startadresse der `struct`-Variablen heraus, bei der Eingabe von »TS.a« dagegen ein Speicherauszug ab der Adresse 0x41 (nämlich dem aktuellen Wert des Feldes `TS.a`).

Außerdem gibt es anscheinend noch einige weitere Rätsel zu lösen bzw. Nacharbeiten zu leisten. Grundlage für Abbildung B.2 war die bereits erwähnte `struct`-Klasse, deren Deklaration so aussieht:

```
[StructLayout(LayoutKind.Sequential)]
public struct TestStruct
{
  public int a, a1, a2;
  public bool b;
```

```
  public char s1;
  public string s2;
  public char s3;
}
```

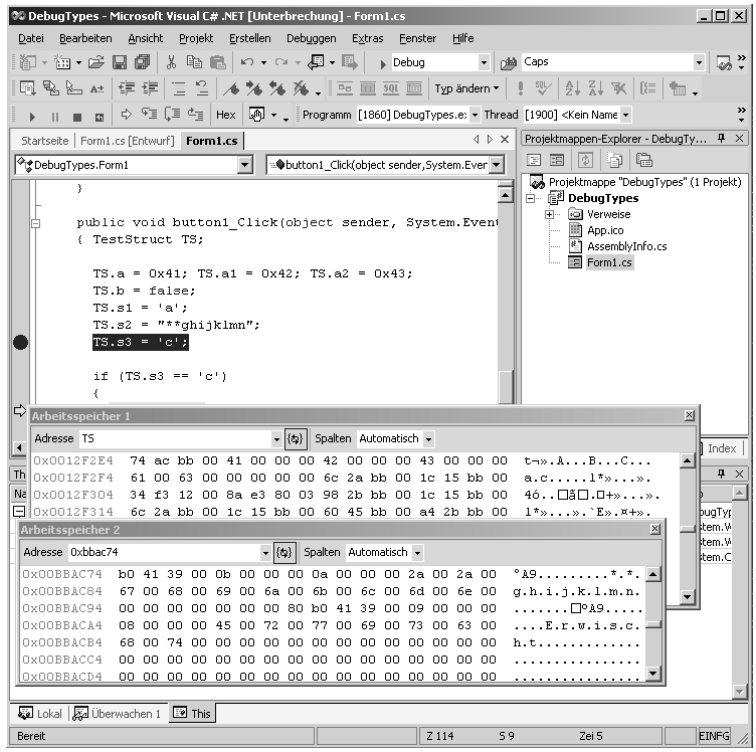


Abbildung B.2: Die `struct`-Instanz `TS` und ihr `string`-Feld `s2` als Speicherauszug. Hundertprozentig ausgereift wirkt das Ganze noch nicht.

Die `int`-Felder `a1`, `a2` und `a3` mit den Werten 0x41, 0x42 und 0x43 sind im Speicherauszug recht schnell wiederzufinden, desgleichen die (Unicode)-Werte der `char`-Felder `s1` und `s3` (nämlich »a« und »c«). So weit, so gut. Mit der Reihenfolge hapert es aber trotz `StructLayout.Sequential` kräftig: Die Adresse des `string`-Werts `s2` steht am Anfang der Struktur, und das `bool`-Feld `b` ist ganz an ihrem Ende gelandet, nämlich hinter `s3`. (Zeigen lässt sich das, indem man dieses Feld im AUTO-Fenster des Debuggers auf `true` setzt und dann die Anzeige des Speicherauszugs aktualisiert.)

Denkbar wäre, dass das Attribute `StructLayout` nur beim Marshalling berücksichtigt wird und der Compiler ansonsten nach Gutdünken verfährt. Allerdings fällt auch der Debugger darauf herein: Wenn man im zweiten Speicherauszugfenster anstelle der zu Fuß abgetippten Adresse »TS.s2« eingibt, erscheint ein Speicherauszug ab der Adresse 0x00000000.

B.2 Fehlersuche in DLLs

Für Windows-DLLs gilt auch im Zusammenhang mit C# die übliche Regel: An Schutzverletzungen in Modulen wie GDI, USER usw. ist in 99,999% das eigene Programm schuld, das an irgendeiner Stelle Unsinn auf den Stack gepackt hat, weshalb die Suche in jedem Fall dort beginnen sollte – also nicht in der DLL. Wer darauf beharrt, dass »seine« Schutzverletzung zu den restlichen 0,001% gehört und nach einer (im Normalfall mehrtägigen) Suche gar den Beweis dafür in den Händen hält, der darf Microsoft eine Mail schicken, bevor er schließlich genau dasselbe tut wie ein Pragmatiker, nur eben einige Dutzend Arbeitsstunden später – nämlich sein Programm so modifizieren, dass das Problem nicht mehr auftritt.

Oder auch: Das Hineinbohren in fremde DLLs kann ausgesprochen spannend sein – lehrreich dazu, der praktische Nährwert geht aber fast immer gegen Null.

Anders sieht die Situation aus, wenn Sie über den Quellcode einer DLL verfügen, sei es nun, dass Sie ihn selbst geschrieben haben oder er als Bestandteil eines Pakets mitgeliefert wurde. In diesem Fall lohnt das Anwerfen des zweiten Compilers/Debuggers allemal, auch – oder erst recht – wenn sich bereits beim ersten Haltepunkt in der DLL zeigt, welch fürchterlichen Unsinn man auf der C#-Seite gebaut hat.

B.2.1 Fehlersuche mit Delphi

Die Fehlersuche in einer mit Delphi geschriebenen und von C# aus benutzten DLL entspricht hundertprozentig dem Standardmodell:

- DLL-Projekt mit Delphi laden, Haltepunkte an gewünschter Stelle (mit F5) setzen.
- Die von C# erzeugte .EXE-Datei über Ausführen/Parameter als Host angeben, bei Bedarf Kommandozeilen-Parameter hinzufügen.
- Start mit F9. (Dass VS.NET diese beiden Funktionstasten genau andersherum als Delphi belegt, hält der Autor dieses Anhangs nach wie vor für einen kleinen Seitenhieb aus Redmond in Richtung der Borland-Entwickler.)

Delphi und VS.NET kommen sich zwar nicht bei der Installation gegenseitig in die Quere, leider aber bei den Debuggern, die beide mit dem üblichen Verfahren arbeiten (nämlich dem zeitweiligen Einsetzen des Prozessorbefehls `INT 03` und dme Umbiegen des Interruptvektors 3 auf sich selbst). Aus diesem Grund ist ein Debugging ausschließlich wechselseitig möglich, nicht

aber zur selben Zeit. Das Szenario – am Beispiel *uniondemo* in *Kapitel 26, »Legacy Code und Windows API«*, (im Abschnitt *»Größenangaben und Unions«*) verdeutlicht – sieht hier folgendermaßen aus:

- In VS.NET: Start des C#-Programms – nicht mit F5, sondern mit F11 (»Einzelschritt«). Die etwas umständlichere Alternative mit demselben Ergebnis ist ein Haltepunkt auf der Anweisung `Application.Run(...)`.
- In Delphi: RUN/ATTACH TO PROCESS, Heraussuchen des Prozesses uniondemo.exe aus der Liste (vgl. Abbildung B.3) und Klick auf ATTACH.

Möglich ist das »Anhängen«, weil die .NET-Laufzeitumgebung DLLs nicht beim Start der jeweiligen Anwendung, sondern erst auf Anforderung hin lädt (vgl. Kapitel 26, »Legacy Code und Windows API«, im Abschnitt *»Ladezeitpunkte und Fehlerprüfung«*). Der einzige, aber leider arg wertmindernde Schönheitsfehler daran: Der Delphi-Debugger wird hier als zweiter gestartet, übernimmt den Interruptvektor – und wird auch für Haltepunkte aktiv, die Sie im C#-Quelltext gesetzt haben. Dass Delphi bei einem solchen »unerwarteten« `INT 03` etwas hilflos auf die Darstellung als reines Disassembly ausweicht und den Befehl einfach so stehen lässt (vgl. Abbildung B.5), ist dann nur konsequent.

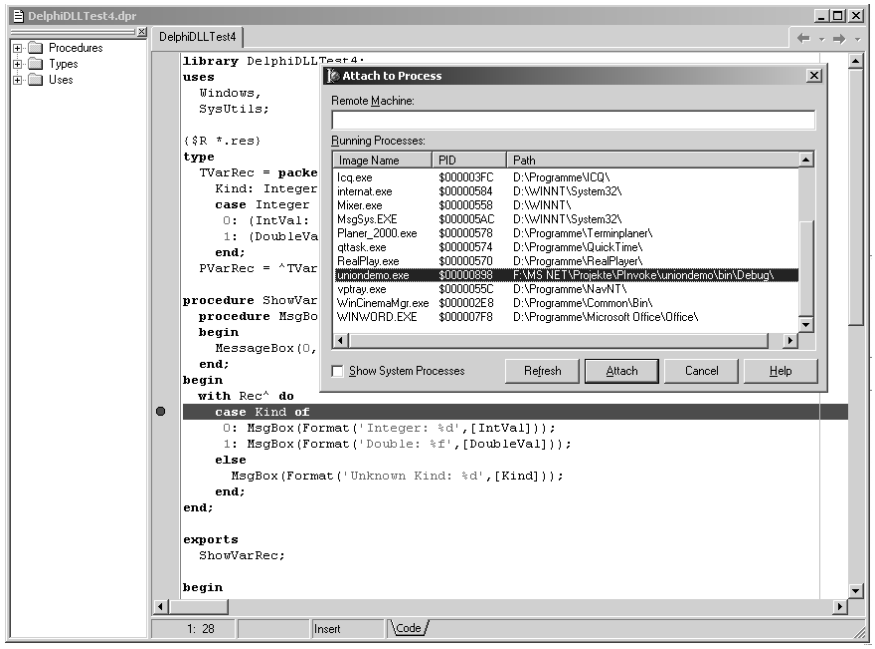


Abbildung B.3: Auswahl des Prozesses, in dessen Adressraum die DLL eingeblendet werden soll, in Delphi (US-Version).

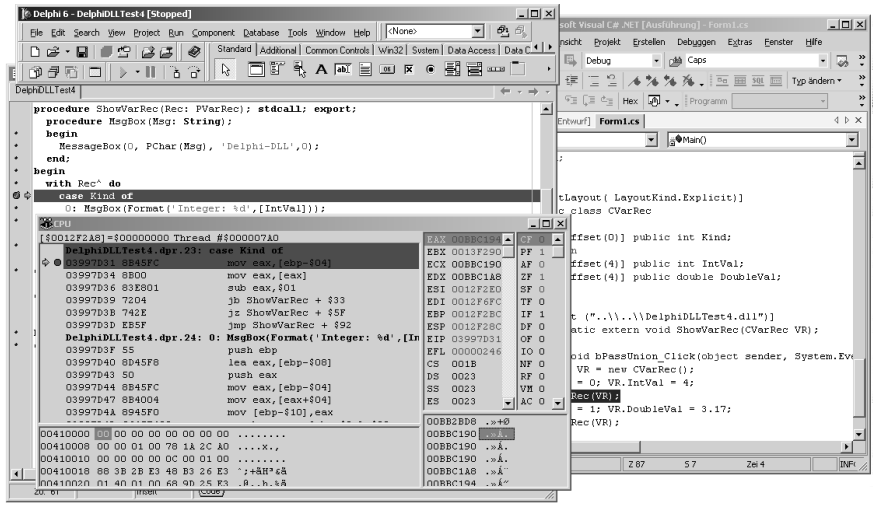


Abbildung B.4: Ein auf der Delphi-Seite gesetzter Haltepunkt in der DLL wurde erreicht.

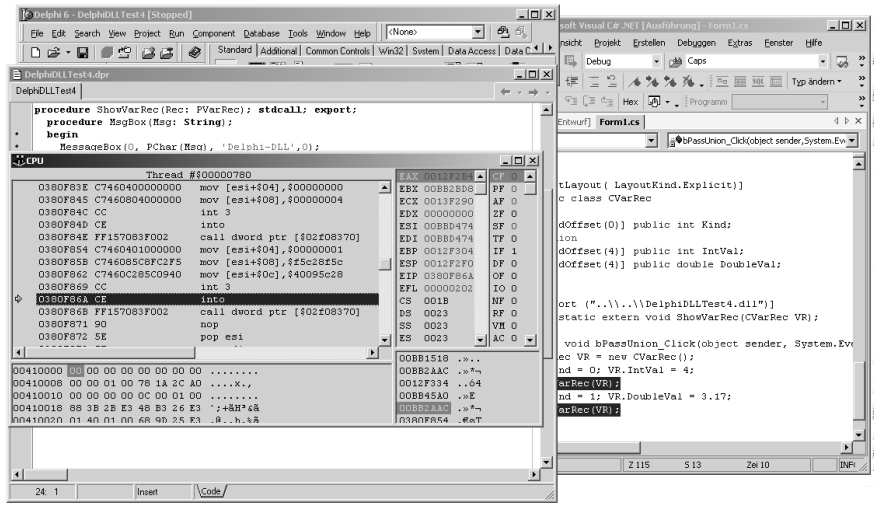


Abbildung B.5: Dieser Haltepunkt stammt leider vom VS-Debugger, genauso wie der einige Zeilen höher erscheinende `INT 03`.



Der Delphi-Debugger hat in einem solchen Fall keine Chance, das durch `INT 03` *ersetzte Befehlscode-Byte gegen den ursprünglichen Wert auszutauschen. Wie auch in Abbildung B.5 zu sehen, wird der nächste auszuführende Befehl zwangsläufig verfälscht. Wer ein auf diese unglückliche Weise unterbrochenes Programm einfach fortsetzen lässt, muss mit den Konsequenzen leben (die sich bei uniondemo allerdings darauf beschränken, dass die DLL auch die Gleitkommazahl als Integer ausgeben will).*

Der einzige Trost: Wenn Ihr System über ausreichend Hauptspeicher verfügt (256 MByte sollten es schon sein), können Sie beide Entwicklungsumgebungen laden und so mit minimalem Aufwand zwischen den beiden Debuggern umschalten, wobei eben zu jedem Zeitpunkt immer nur einer der beiden aktiv sein darf. (Wer kein Zen-Meister ist, bringt spätestens beim dritten Umschalten F5 und F9 durcheinander – aber diese Tastenbelegung ließe sich ja in beiden IDEs ändern.)

B.2.2 Fehlersuche mit MSVC

Die Unterschiede zwischen Delphi und MSVC sind hier erheblich kleiner, als man glauben mag: Die Menüs haben andere Namen, F5 und F9 funktionieren in beiden IDEs gleich – und die Debugger kommen sich genauso ins Gehege wie bei Delphi und VS.NET.

Zur Fehlersuche in einer mit MSVC geschriebenen DLL, die von einer C#-Anwendung aus genutzt wird, sind hier die folgenden Schritte notwendig:

- DLL-Projekt mit MSVC laden, Haltepunkte an gewünschter Stelle (mit F9) setzen.
- Die von C# erzeugte .EXE-Datei über Ausführen/Parameter als Host angeben, bei Bedarf Kommandozeilen-Parameter hinzufügen. Das läuft bei MSVC über PROJECT/SETTINGS und dort die Registerkarte DEBUG, weitere Unterschiede zu Delphi gibt es nicht – was leider auch heißen soll, dass MSVC mit den Debug-Informationen des C#-Programms nicht das Geringste anfangen kann. (Andersherum sieht die Geschichte leider auch nicht besser aus: VS.NET hat mit den von MSVC in eine DLL eingebauten Zusatzinformationen zur Fehlersuche ebenfalls nichts am Hut.)
- Start mit F5.

Erstaunlicherweise hat man es bei Microsoft anscheinend für überflüssig gehalten, sich Gedanken um ein Protokoll zu machen, das wenigstens bei den hauseigenen Debuggern Haltepunkte auseinanderdividiert: Wenn man ein C#-Programm von VS.NET aus startet und sich dann mit MSVC über BUILD/START DEBUG/ATTACH TO PROCESS an diesen Prozess anhängt (vgl. Abbildung B.6), dann hat eben der MSVC-Debugger bei allen Haltepunkten das erste und letzte Wort. Und wie Abbildung B.7 leider zeigt, stellt der sich bei mit VS.NET gesetzten Haltepunkten keine Spur schlauer an als das Gegenstück aus dem Hause Borland.

Abbildung B.6: Anhängen an einen Prozess in MSVC

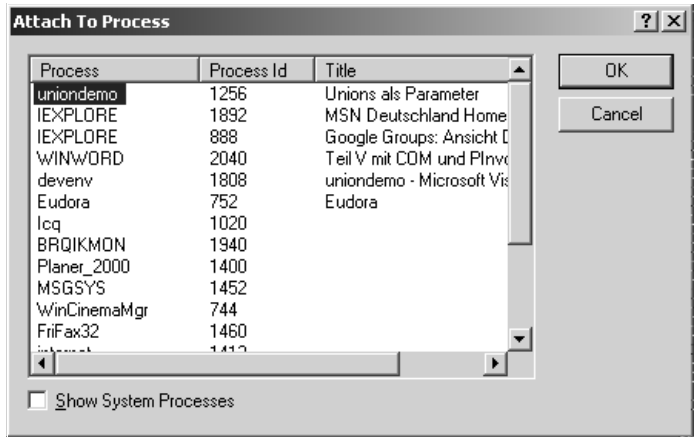
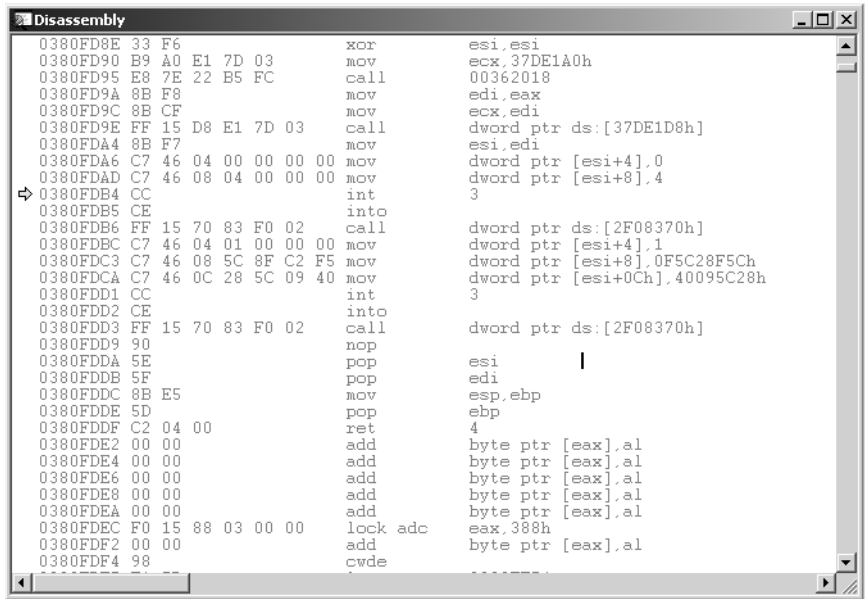


Abbildung B.7: Ein mit VS.NET gesetzter Haltepunkt – diesmal im (genauso überforderten) Debugger von MSVC



C Internet-Protokolle

Wenn Sie `www.mut.de` in die Adresszeile Ihres Browsers eingeben, erscheint – eine funktionierende Internetverbindung vorausgesetzt – die Homepage des Markt+Technik-Verlags. Dieser Anhang beantwortet die Frage, wie das eigentlich funktioniert, warum es so funktioniert, und wie .NET diese Funktionalität zur Kommunikation zwischen den Komponenten verteilter Programme benutzt.

C.1 HTTP

Um dem Benutzer das mit der Eingabe `www.mut.de` angeforderte Dokument zeigen zu können, stellt der Browser (Client) dem Server eine Anfrage im HyperText Transfer Protocol (HTTP). Dieses Protokoll wurde, wie der Name schon vermuten lässt, zur Übertragung von Text-Dokumenten konzipiert. Auf eine Anfrage des Clients folgt dabei genau eine Antwort des Servers, danach wird die Verbindung beendet. Aus Performance-Gründen kann die Verbindung in der neuesten Version 1.1 von HTTP auch erhalten bleiben. Trotzdem ist und bleibt HTTP ein zustandsloses Protokoll, das heißt der Server »vergisst« beantwortete Anfragen sofort. Sobald die Kommunikation also über den simplen Austausch (der Client kann auch ein Dokument an den Server senden) einzelner Dokumente hinausgeht, müssen Client und Server den Stand ihrer Kommunikation außerhalb von HTTP verwalten, zum Beispiel mit Cookies.

Das Fehlen der Zustandsverwaltung in HTTP bedeutet letztlich zwar zusätzlichen Aufwand, aber nur über ein zustandsloses Protokoll können Server die Internet-üblichen Client-Mengen bedienen. Denn zum einen kann der Server so zwischen einzelnen Anfragen eines Clients die Anfragen anderer Clients beantworten. Bei einem zustandsorientierten Protokoll müsste der Server die für diesen Client benötigten Ressourcen reserviert halten. Zum anderen ist der Server für die Freigabe seiner Ressourcen nicht auf eine formelle Abmeldung der Clients angewiesen (was auch gut so ist: Ansonsten müsste man den Internet Explorer ähnlich wie Windows herunterfahren, Verbindungszusammenbrüche würden serverseitig eine Spezialbehandlung erfordern usw.).

HTTP ist ein rein textbasiertes Protokoll, weshalb man den Inhalt von HTTP-Nachrichten recht einfach sichtbar machen und untersuchen kann. In Abbildung C.1 sehen Sie die Anfrage nach der Homepage des Markt+Technik-Verlags im Netzwerkmonitor. Dieses Programm gehört zum Lieferumfang vieler Windows-Versionen, allerdings nicht zur Standard-Installation. Wenn Sie es nicht unter SYSTEMSTEUERUNG/SOFTWARE/WINDOWS-KOMPONENTEN finden: Das Internet bietet Zugriff auf eine ganze Reihe von Programmen mit gleicher Funktionalität.

INFO

Der Netzwerkmonitor (`netmon.exe`*) wird von der Windows-Installation im Verzeichnis System bzw. System32 untergebracht. Einen Eintrag im START-Menü erzeugt diese Installation augenscheinlich nicht.*

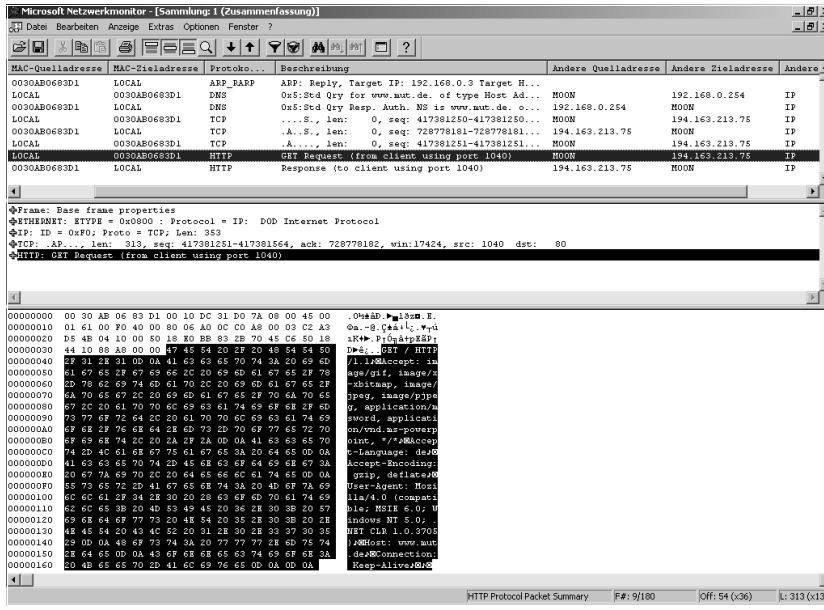


Abbildung C.1: Die HTTP-Anfrage im Netzwerkmonitor

Der vereinfachte Aufbau der HTTP-Anfrage sieht etwa so aus:

```
GET / HTTP/1.0
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR
1.0.3705)
Host: www.mut.de
Connection: Close
```

Das erste Wort beschreibt den gesendeten Befehl, in diesem Fall ein HTTP-GET. Darauf folgen Pfad und Name des angeforderten Dokuments. Hier wurde kein Name angegeben, der Server wird deshalb seine Standard-Seite senden. Als Letztes steht die verwendete HTTP-Version in der ersten Zeile.

In der zweiten Zeile nennt der Client die von ihm akzeptierten Formate, in diesem Fall alles. Er könnte auch einzelne Formate nennen, zum Beispiel *image/jpeg*. In der dritten Zeile stellt sich der Client vor, in der vierten wird der Server genannt. Die letzte Zeile legt fest, dass die Verbindung nach dem Senden der Antwort abgebaut werden soll.

Den ersten Teil der Antwort des Servers zeigt Abbildung C.2 im Netzwerkmonitor. Sie erkennen dort auch den Cookie, den der Server zum Verwalten der Verbindung sendet.

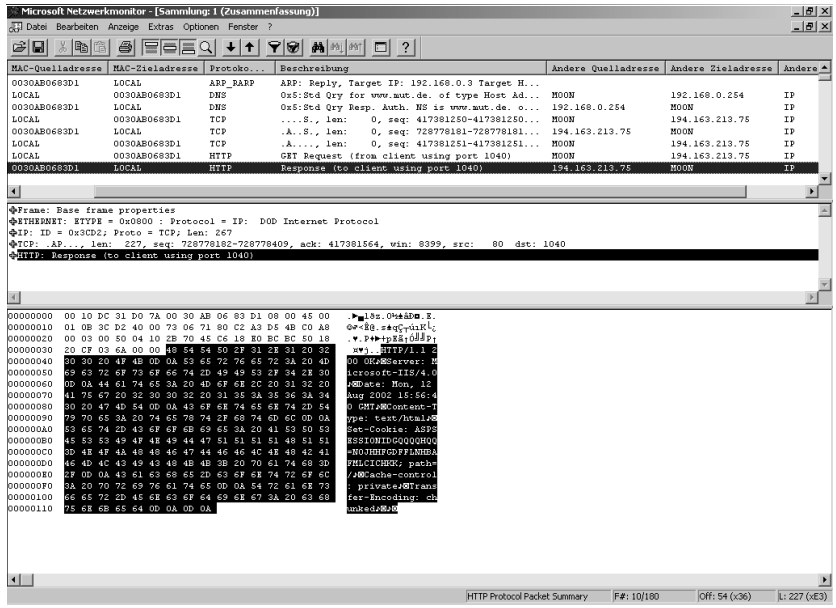


Abbildung C.2: Der erste Teil der HTTP-Antwort im Netzwerkmonitor

Den zweiten Teil der Antwort zeigt Abbildung C.3, hier ist der Anfang des HTML-Dokuments zu erkennen.

Die Antwort besteht insgesamt aus 35 Einzelnachrichten; zusätzlich fordert der Client weitere Dokumente an – beispielsweise Grafiken, auf die das eigentliche Dokument verweist. Der grundsätzliche Aufbau der gesamten Antwort sieht so aus:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Date: Mon, 18 Mar 2002 15:12:11 GMT
Content-Type: text/html
Cache-Control: private

<html>
<head>
...
```

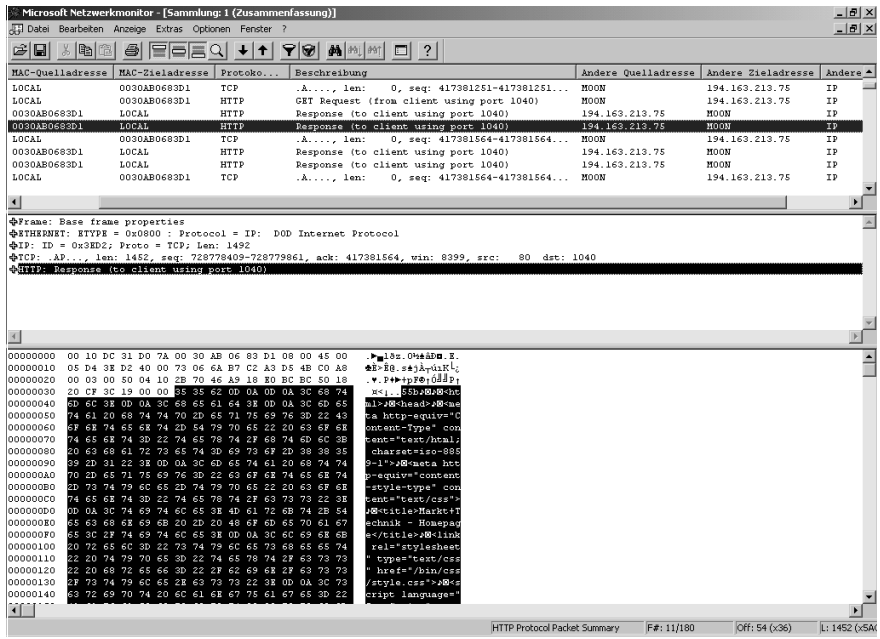


Abbildung C.3: Der zweite Teil der HTTP-Antwort im Netzwerkmonitor

```
</head>
<body>
...
<body>
</html>
```

Die Antwort besteht also aus dem Kopf und einem Rumpf, der hier nur andeutungsweise wiedergegeben ist. Beide sind durch eine Leerzeile getrennt. Hier erscheint keine wie auch immer geartete Client-Adresse – es stellt sich also die Frage, wie die Antwort den Weg zum Client überhaupt gefunden hat. Tatsächlich kümmert sich HTTP überhaupt nicht um den Weg, den der Datentransport nimmt: Dafür ist das TCP-Protokoll zuständig.

C.2 TCP

Das Transmission Control Protocol (TCP) ist zuständig für die korrekte Übertragung der Daten zwischen Client und Server. Das ist durchaus eine Herausforderung, denn zum einen werden die Daten nicht »am Stück« übertragen, sondern in einzelnen Paketen. Zum anderen werden sie nicht über eine physische Verbindung gesendet, sondern über ein Netzwerk, das jedes Paket unabhängig von den anderen weiterleitet: die Pakete nehmen also unter Umständen unterschiedliche Wege – und treffen beim Empfänger nicht unbedingt in der Reihenfolge ein, wie sie versendet werden.

Das Modell ist zwar komplizierter als beispielsweise beim (analogen) Telefonieren, dafür aber leistungsfähiger: Aufgrund der Unterteilung in Pakete können mehr Kommunikationsverbindungen gleichzeitig etabliert werden, als physische Leitungen vorhanden sind. Außerdem ist dieses Modell weniger störanfällig: Verloren gegangene oder beschädigte Pakete werden einfach neu gesendet, was bei kompletten Dokumenten zu aufwändig wäre. Und beim Ausfall einer Leitung oder eines Weiterleitungspunkts (ROUTER) wird einfach ein anderer Weg benutzt. Das geschieht automatisch, denn die beteiligten Router lernen selbstständig. Diese eingebaute »Unkaputtbarkeit« verdankt das Internet seinen militärischen Wurzeln.

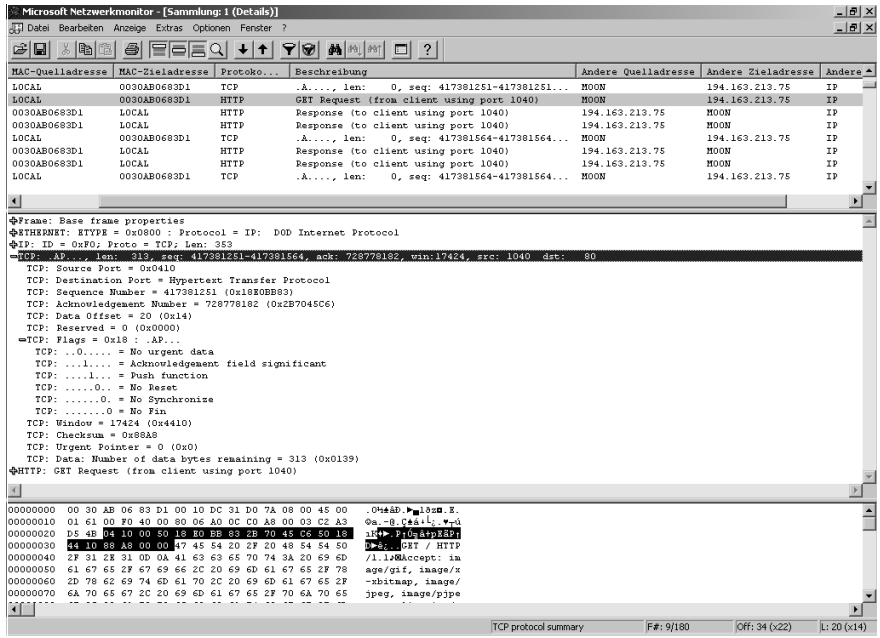
Um die korrekte Übertragung der Daten zu gewährleisten, fügt die TCP-Implementation den HTTP-Daten beim Aufteilen in Pakete zusätzliche Informationen hinzu. So fallen durch Angabe der noch ausstehenden Bytes verloren gegangene Pakete auf (da die Router Pakete ihrerseits weiter aufteilen dürfen, kann die TCP-Implementation hier nicht die Anzahl der noch ausstehenden Pakete benutzen). Dank einer ebenfalls hinzugefügten Sequenznummer lassen sich die empfangenen Pakete wieder in die richtige Reihenfolge bringen. Schließlich berechnet die TCP-Implementation für den Inhalt des Paketes eine Prüfsumme und kann so beschädigte Pakete erkennen.

Damit auf einem Rechner mehrere Programme gleichzeitig über TCP kommunizieren können, benutzen sie so genannte PORTS. Diese Ports dienen nur der Kanalisierung des Datenverkehrs, sie bieten keine zusätzliche Funktionalität. Der Client hat sich im Beispiel den Port *1040* ausgesucht und spricht den Port *80* des Servers an. IP-Adresse und Port-Nummer von Server und Client identifizieren eine TCP-Verbindung eindeutig. Ein Client-Programm kann auch über mehrere Ports gleichzeitig kommunizieren – so macht es Ihr Browser, wenn Sie mehrere Fenster gleichzeitig öffnen.

Alle diese Informationen werden wiederum in einem Kopf untergebracht – HTTP-Kopf und -Rumpf bilden die Nutzlast, eine Verschlüsselung findet auf TCP-Ebene nicht statt. Das TCP-Paket sieht also genauso aus wie bei HTTP gezeigt, nur kommen am Anfang einige Binärdaten dazu. Abbildung C.4 zeigt den TCP-Header der Anfrage im Netzwerkmonitor.

Die TCP-Implementation auf der Empfängerseite wertet den Kopf aus, sortiert die Pakete und sendet eine Empfangsbestätigung, wenn alles korrekt übertragen wurde. Erhält der Sender innerhalb einer vorgegebenen Zeitspanne keine Empfangsbestätigung, sendet er die betroffenen Pakete von sich aus erneut. Auch hier erkennt man das auf Robustheit ausgelegte Design des Internets. Aus Performance-Gründen wird die Empfangsbestätigung immer für eine bestimmte Menge von Paketen gesendet, nicht für jedes einzelne.

Abbildung C.4: Der TCP-Header der Anfrage im Netzwerkmonitor



Offensichtlich kümmert sich auch die TCP-Implementation nicht um den Weg der Pakete zum Empfänger. Wie HTTP delegiert sie diese Aufgabe an ein untergeordnetes Protokoll namens IP.

C.3 IP

Das Internet Protocol (IP) ist zuständig für die Weiterleitung der Pakete. Dazu benutzt es allerdings eine IP-Adresse statt der im Browser eingegebenen Adresse. Denn eine Adresse wie `"www.mut.de"` ist wohl für Menschen leicht erinnerbar, für Maschinen aber schwer zu verarbeiten. Also muss sie in einem Zwischenschritt übersetzt werden, das erledigen so genannte Domain Name System (DNS) Server. In kleinen Netzen kann dafür auch die Hosts-Datei benutzt werden. (Diese Datei enthält eine einfache textuelle Liste, die Computernamen IP-Adressen zuordnet.)

Welchen DNS-Server Ihr Rechner ansprechen soll, wird bei der Konfiguration des Internetzugangs festgelegt bzw. beim Zustandekommen einer Einwahlverbindung vom Provider übermittelt. Sollte dieser DNS-Server die Antwort nicht kennen, konsultiert er automatisch andere DNS-Server. Schließlich sendet er als Antwort die IP-Adresse, hier *194.163.213.75*.

Eine IP-Adresse ist weltweit, das heißt im gesamten Internet, eindeutig. Sie besteht aus 4 Byte (= 32 Bit), die meistens in Vierergruppen dezimal geschrieben werden, zum Beispiel *192.168.0.1*. Diese Gruppierung unter-

stützt nicht nur die Lesbarkeit, sie verdeutlicht auch die Funktion des Internets als Zusammenschluss eigenständiger Netze. Diese so genannten Subnetze haben eigene IP-Adressbereiche, die von der ICANN (Internet Corporation for Assigned Names and Numbers, früher IANA (Internet Assigned Numbers Authority) vergeben werden. So bekommt auch ein Internet Service Provider (ISP), wie zum Beispiel T-Online, einen IP-Adressbereich zugewiesen. Und sobald Sie mit Ihrem Rechner online gehen, erhält er (exakter: das angeschlossene Modem) vom ISP automatisch eine IP-Adresse aus diesem Bereich zugeordnet: Erst danach können Datenpakete quer durch das Internet zu Ihrem Rechner geleitet werden.

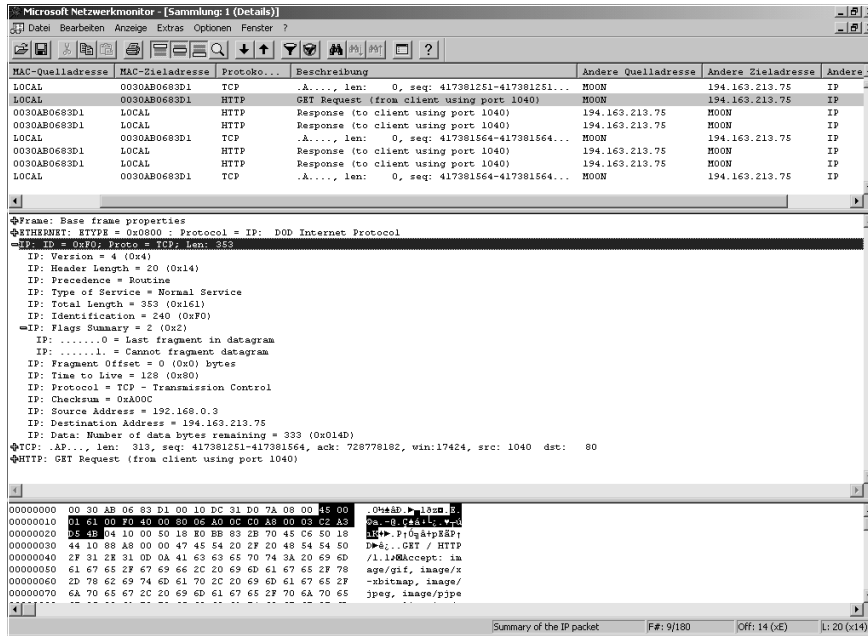
Natürlich macht es keinen Sinn, ein Paket, das nur zum nächsten Schreibtisch will, durch das Internet zu schicken. IP-Adressen, die im eigenen SUBNETZ liegen, müssen also erkannt werden können. Dazu dient die Subnetzmaske, die Sie vielleicht schon von der Konfiguration Ihres Rechners kennen. In kleinen Netzen hat diese Maske meistens die Form *255.255.255.0*. Dadurch beschreiben die ersten drei Vierergruppen der IP-Adresse das Subnetz (Netzwerk-Adresse) und die letzte Vierergruppe den Rechner (Host-Adresse). Da die Adressen *0* (das Subnetz selber) und *255* (Broadcast an alle Hosts im Subnetz) reserviert sind, kann ein solches Subnetz maximal 254 Hosts lokal verbinden. Nicht jeder Host ist unbedingt ein Rechner, denn auch andere Geräte, z.B. Router, brauchen eine eigene IP-Adresse. Es sind über 2 Millionen dieser so genannten CLASS-C-NETZE möglich.

Die CLASS-A- und CLASS-B-Netze können wesentlich mehr Hosts enthalten, dafür sind deutlich weniger solcher Netze möglich. Da diese Netze alle schon vergeben sind, müssen neue bei Bedarf aus einzelnen Class-C-Netzen zusammengesetzt werden. Dieses Problem wird, neben anderen, durch die neue Version des IP-Protokolls (IPv6) gelöst. Die IP-Adressen in IPv6 sind nicht etwa 40, sondern gleich 128 Bit lang, wodurch eine astronomische Menge an Adressen zur Verfügung steht.

Trotzdem ist es nicht nötig, für jeden Rechner eine IP-Adresse zu registrieren. Denn die meisten Rechner kommunizieren schon aus Sicherheitsgründen nicht direkt mit dem Internet, sondern über einen dazwischen geschalteten Router. Nur dieser erhält wie oben beschrieben eine offizielle IP-Adresse und ist vom Internet aus direkt ansprechbar. Die Rechner bekommen im Class-C-Netz eine lokale IP-Adresse im Bereich von *192.168.0.0* bis *192.168.255.255*, für die anderen Netz-Klassen gibt es ähnliche Festlegungen. Pakete mit Adressen aus diesem Bereich werden im Internet nicht weitergeleitet und können deshalb ihr Subnetz nicht verlassen. Wenn einer der Rechner mit dem Internet kommunizieren will, wendet er sich an den Router. Dieser braucht dazu ebenfalls eine lokale IP-Adresse, die in der Rechner-Konfiguration dann als STANDARD-GATEWAY eingetragen wird. Der Router hat demnach zwei IP-Adressen.

Nun zurück zum Paket des Beispiels. Auch die IP-Implementation fügt dem zu sendenden Paket einen Kopf hinzu. Dieser enthält neben den IP-Adressen von Sender (hier *192.168.0.3*) und Empfänger (hier *194.163.213.75*) noch Angaben wie die Version des IP-Protokolls (*4*) und den Typ des Nutzlast-Protokolls (*TCP*). Außerdem enthält der Kopf zum Beispiel die so genannte »Time to Live«-Information (*128*), abgekürzt TTL. Diese Angabe bestimmt, dass das Paket maximal 128-mal weitergeleitet werden darf. Jeder Router setzt diese Zahl um Eins herab und wenn sie auf *0* fällt, wird das Paket vernichtet. Das verhindert, dass Pakete ewig durch das Internet vagabundieren. Abbildung C.5 zeigt den IP-Header der Anfrage im Netzwerkmonitor.

Abbildung C.5: Der IP-Header der Anfrage im Netzwerkmonitor



Das Paket sieht immer noch genauso aus wie in der ersten Abbildung, nur die Binärdaten am Anfang haben sich wieder vermehrt. In dieser Form erreicht das Paket den Treiber der Netzwerkkarte.

C.4 Der Weg durchs Netz

Der TCP/IP-Stapel hat jetzt seine Arbeit erledigt, die Details der Weiterleitung richten sich nun nach dem verwendeten Netzwerk. Da die allermeisten Leser ein ETHERNET-Netzwerk verwenden werden, konzentriert sich die folgende Beschreibung auf diesen Standard.

Der Treiber der Netzwerkkarte muss das Paket zuerst einmal neu adressieren, denn Netzwerkkarten identifizieren ihr Gegenüber anhand der MAC-ADRESSE (Media Access Control). Auch diese Adresse ist weltweit eindeutig,

entsprechende Adress-Bereiche werden den Netzwerkkarten-Herstellern von einer Organisation zugeteilt. Die MAC-Adresse ist 48 Bit groß und kann deshalb nicht als IP-Adresse im Paket untergebracht werden: Erstens bietet IPv4 nur 32 Bit dafür, zweitens werden MAC-Adressen vom jeweiligen Hersteller fix in die Netzwerkkarte eingetragen, weshalb man ohne weiteres auch in einem LAN höchst unterschiedliche MAC-Adressen vorfindet – einen Zusammenhang zwischen Werten und räumlicher Anordnung bzw. Verbindungen der einzelnen Computer gibt es hier nicht.

Um die MAC-Adresse zur IP-Adresse zu erfahren, sendet die Netzwerkkarte einen Broadcast an alle anderen Netzwerkkarten im Subnetz. Eine davon antwortet: Im Beispiel wird das der Router sein, denn der ist zuständig für alle außerhalb des Subnetzes liegenden IP-Adressen. Die Netzwerkkarte adressiert dann das Paket mit der gemeldeten MAC-Adresse, und es kann endlich den Rechner verlassen. Abbildung C.6 zeigt den BROADCAST und die Antwort darauf sowie den Ethernet-Header des Beispiel-Pakets, den hier zu sehenden Frame-Header hat der Netzwerkmonitor zum Speichern zusätzlicher Informationen hinzugefügt.

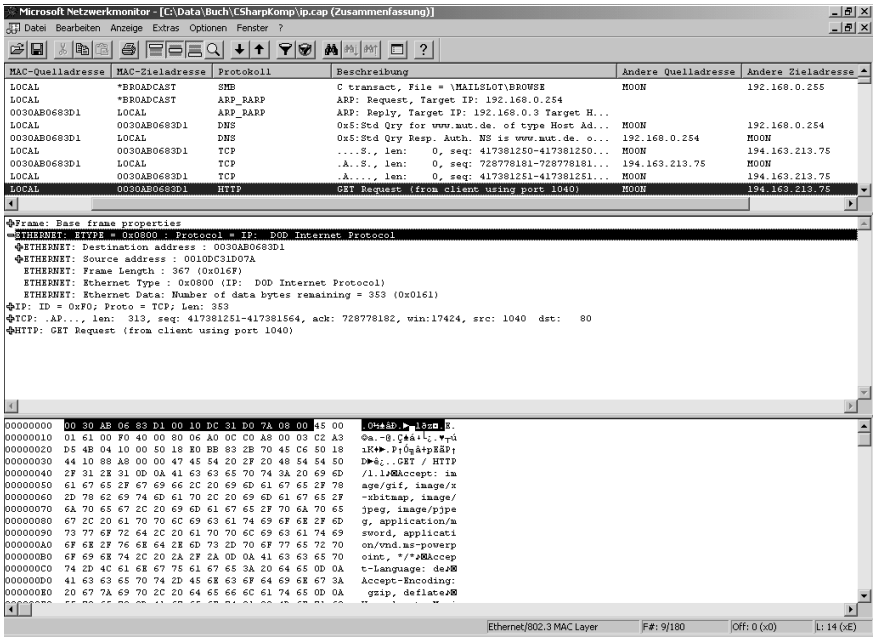


Abbildung C.6: Der Ethernet-Header der Anfrage im Netzwerkmonitor

Intelligenterweise merkt sich die Netzwerkkarte diese Zuordnung zwischen MAC- und IP-Adresse, sodass sie beim nächsten Mal nicht erneut fragen muss. Die ganze Prozedur zum Ermitteln der MAC-Adresse ist im Adress Resolution Protocol (ARP) definiert. ARP-Anfrage und -Antwort sehen Sie in Abbildung C.6, Abbildung C.7 zeigt die Benutzung des (zum Lieferumfang von Windows gehörigen) Programms `arp.exe`, das die gespeicherten Zuordnungen auflistet.

Abbildung C.7: Das Programm ARP.exe listet gespeicherte Zuordnungen auf



Jede Netzwerkkarte sieht sich alle Pakete an, die sie erreichen, nimmt aber nur solche an, die an sie adressiert sind. (Die Ausnahme von der Regel sind spezielle Netzwerk-Analysatoren oder Sniffer.) Neben denen mit der eigenen MAC-Adresse sind das z.B. noch Broadcast-Pakete. Die im lokalen Netzwerk gelegene Seite des Routers erkennt dabei, dass die IP-Adresse des Empfängers außerhalb des Subnetzes liegt und gibt das Paket an die Internetseite weiter.

Dabei erledigt der Router noch eine andere Aufgabe: Er trägt statt der lokalen IP-Adresse (hier *192.168.0.3*) des Clients seine eigene Internet-IP-Adresse ein, die er im Allgemeinen dynamisch vom ISP bezogen hat. Wenn mehrere Clients über dieselben IP-Portnummer kommunizieren wollen, muss der Router diese zusätzlich durch eindeutige Nummern ersetzen. Bei ankommenden Paketen führt er dann eine entsprechende Rückübersetzung der IP-Adressen und Port-Nummern durch. Eine solche Adressumsetzung sorgt auch dafür, dass sich ein Rechner im LAN nicht ungebeten über das Internet ansprechen lässt, sondern nur Pakete erhält, die er zuvor angefordert hat. Ein Router, der zusätzlich noch eine Paketfilterung vornimmt, d.h. die Weiterleitung auf Pakete mit bestimmten Portnummern beschränkt, ist bereits eine einfache Firewall.

Die internetseitige Netzwerkkarte des Routers kann das Paket im Allgemeinen nicht direkt an den Server senden, da sie dorthin keine physische Verbindung hat. Solche Verbindungen hat sie nur zu anderen Routern, die das Paket dann weiterleiten. Dazu besitzt jeder Router eine Routing-Tabelle mit den von ihm zu erreichenden benachbarten Routern. Diese Tabelle kann fest einprogrammiert sein oder mit verschiedenen Protokollen dynamisch aktualisiert werden. Aus ihr und einem Entscheidungs-Algorithmus kann ein Router für jede IP-Adresse den »besten« Router für die Weiterleitung ermitteln. Dieses Verfahren setzt sich fort, bis das Paket die Netzwerkkarte des Ziel-Rechners erreicht; den Weg kann man mit Werkzeugen wie Tracert.Exe sichtbar machen (Abbildung C.8). Für diesen Zweck finden Sie im

Internet sogar Implementationen, die den Weg grafisch auf einer Karte darstellen: VisualRoute, NeoTrace (die Express-Version ist Freeware) und viele andere.

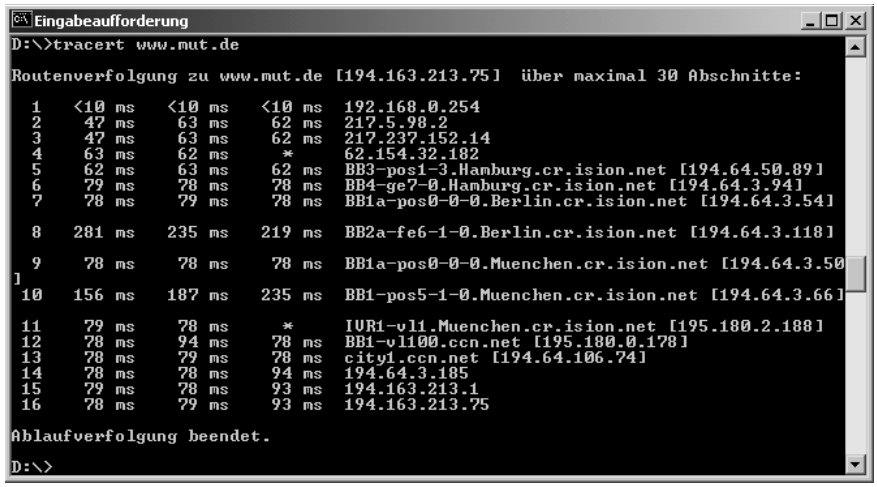


Abbildung C.8: Ablaufverfolgung mit tracert (»trace route«)

Jeder Router reagiert automatisch auf Änderungen seiner Umgebung, und eine (leicht angreifbare) Zentrale gibt es nicht. Das paketorientierte Weiterleiten durch ein Netz erschwert auch das Abhören. Einzelne Router abzuhören ist zwar leicht möglich, weil aber nicht alle Pakete eines Dokuments ihren Weg über denselben Router nehmen, fällt dem Angreifer nur ein Teil des Dokuments in die Hände. Gleichzeitig fallen riesige Datenmengen an, die selbst mit der heute verfügbaren Technik kaum auszuwerten sind. Nicht schlecht für ein System, das vor über 20 Jahren erdacht wurde.

Man sollte sich allerdings immer vor Augen halten, dass die Daten in keiner Weise verschlüsselt werden und mit relativ einfachen Mitteln erfasst und gespeichert werden können. Über jede Netzwerk-Steckdose in einem Flur, in der Lobby oder in einem Tagungsraum ist der Zugriff auf alle Daten des herausgeführten Subnetzes möglich – nur ein Funknetz ist noch leichter angreifbar. Abhilfe schafft der Einsatz von Verschlüsselung wie Secure Sockets Layer (SSL) oder das darauf aufsetzende HTTPS.

Nun zurück zum Paket des Beispiels, das mittlerweile am Ziel angekommen ist: Die Netzwerkkarte des Zielrechners übergibt das Paket an die IP-Implementation, welche die Informationen im IP-Kopf überprüft und dann entfernt. Das gleiche macht die TCP-Implementation für ihre Informationen. Dabei leitet sie das Paket automatisch an den HTTP-Server weiter, denn der lauscht an Port 80 (exakter: hat sich als Empfänger für diesen Port bei TCP angemeldet). Diese Portnummer ist fest für HTTP vergeben, der Client muss bei der Adressierung also nicht raten.

Nachdem der HTTP-Server unter der Portnummer 80 die Verbindungs-Anforderungen angenommen hat, kann er für die weitere Kommunikation mit dem Client nun eine andere Port-Nummer vorgeben. Von dieser sendet er dann die einzelnen Pakete des Dokuments, und der Client kann die Einzelteile anhand der Server-Portnummer wieder zusammenfügen. Das ist wichtig, denn ein Client fordert häufig weitere Dokumente vom Server an, bevor das erste vollständig angekommen ist. Das passiert zum Beispiel mit Grafiken und Stylesheets eines größeren HTML-Dokuments. Bei Anforderung des nächsten Dokuments spricht der Client wieder die Port-Nummer 80 an.

Im nächsten Abschnitt finden Sie das verbindende Rahmenwerk zu den gerade beschriebenen Internetprotokollen und zu ihrer Nutzung in .NET.

C.5 Der Protokollstapel

Der Begriff »Protokollstapel« bezeichnet einen Stapel von Kommunikations-Protokollen (wobei hier nicht ein Stack gemeint ist, sondern die Aufteilung in mehrere Schichten). Jedes dieser Protokolle implementiert eine bestimmte Funktionalität und definiert eine Schnittstelle zur Benutzung durch das Protokoll in der direkt darüber- und darunter liegenden Schicht. Durch diese Standardisierung können Sie beliebige Teile des Protokollstapels selbst implementieren.

Die oberste Schicht des Protokollstapels ist die Anwendung, die kommunizieren will, die aller unterste die Netzwerk-Hardware. Dazwischen existieren je nach Referenz-Modell verschieden viele Schichten. Abbildung C.9 zeigt das bekannteste Modell, das OSI-Modell, das sieben Schichten definiert.

Abbildung C.9: Der Protokollstapel

OSI-Schicht	OSI-Name	TCP/IP-Schicht	TCP/IP-Protokoll	.NET-Klasse	Nutzung
7	Anwendung	Anwendung	HTTP, FTP, SMTP, POP, DNS, ...	WebClient, HttpWebRequest/Response, WebRequest/Response, ...	Webdienste, Webanwendungen, Remoting über HTTP
6	Darstellung				
5	Sitzung		SSL		
4	Transport	Transport	TCP, UDP, ...	TcpListener, -Client, UdpClient, Socket, ...	Remoting über TCP, DCOM
3	Vermittlung	Internet	IP (ICMP), ARP, ...	—	—
2	Sicherung	Netzwerk	—	—	—
1	Übertragung	(Hardware) z.B. Ethernet	—	—	—

Bei der Beschreibung der einzelnen Protokolle wurde bereits deutlich, dass die Daten in kleinen Paketen zwischen den Protokollen ausgetauscht werden. Serverseitig fügt dabei das jeweils unten liegende Protokoll dem erhaltenen Paket zusätzliche Informationen hinzu, den so genannten HEADER.

Das kann zum Beispiel eine Prüfsumme sein, mit der die Integrität der Daten sichergestellt wird. Auf der Clientseite existiert der gleiche Protokollstapel. Dort werden die Daten von unten nach oben hochgereicht, wobei jedes Protokoll nur den Header auswertet, der vom selben Protokoll auf der Senderseite hinzugefügt wurde. Dann entfernt es diesen Header und reicht das restliche Paket nach oben hin weiter. Dieses Vorgehen ist bei allen Protokollstapeln gleich.

C.5.1 Der TCP/IP-Stapel

Bisher war vom abstrakten Modell des Protokollstapels die Rede. Dazu existieren verschiedene, mehr oder weniger weit verbreitete praktische Implementationen – neben TCP/IP beispielsweise Novells IPX/SPX. Diese Beschreibung hier widmet sich ausschließlich dem Sieger der Protokoll-Evolution, nämlich TCP/IP: Es gibt praktisch kein Betriebssystem (mehr), das keine Implementation eines TCP/IP-Stapels besitzt. Die im Beispiel beschriebene Kommunikation über HTTP und die darunter liegenden Protokolle funktioniert also mit allen Betriebssystemen: Zum Beispiel kann ein Windows-Client einen Unix-Server ansprechen und umgekehrt. Der TCP/IP-Stapel wurde in den späten Siebzigern mit dem Ziel entwickelt, mehrere Netze miteinander zu verbinden. Daher der Name Internet Protocol (IP). Abbildung C.9 zeigt die Zuordnung der Protokolle zu Schichten.

Zur Implementation eines TCP/IP-Stapels gehören auch die Protokolle ICMP und UDP. Das User Datagram Protocol (UDP) hat die gleiche Aufgabe wie TCP, führt aber keine Fehlerkorrektur durch – darum muss sich die Anwendung selber kümmern. Dafür ist der Durchsatz höher. UDP wird zum Beispiel für Internet-Radio eingesetzt. Das Internet Control Messaging Protocol (ICMP) dient zur Fehlerbehandlung und Kommunikationskontrolle, nicht zum Datentransport. Sie haben sicher schon den Ping-Befehl benutzt, den dieses Protokoll zur Verfügung stellt.

Genauso wie Sie sich zwischen TCP und UDP entscheiden, wählen Sie ein Protokoll der darüber liegenden Ebene. So benutzt ein Browser HTTP, ein Newsreader das Network News Transport Protocol (NNTP), ein Mail-Client benutzt das Post Office Protocol (POP3) usw. Jedem dieser Protokolle ist serverseitig ein TCP-Port zugeordnet, an dem ein entsprechendes Programm lauscht. Diese Ports sind so genannte »well known ports« und haben Nummern von *0* bis *1023*, die von der IANA (heute ICANN) vergeben wurden. Das sind zum Beispiel *80* für HTTP, *110* für POP3 und *119* für NNTP. Daneben gibt es noch »registered ports« mit Nummern von *1024* bis *49151* und schließlich die »dynamic and/or private ports« im Bereich von *49152* bis *65535*. Die Kombination von Protokoll und Port nennt man auch »Socket«, zu Deutsch »Steckdose« – denn damit können neue Protokolle in den bestehenden Stapel »eingesteckt« werden. Diese Art der Programmierung hat sogar einen eigenen Namen: Socket-Programmierung.

Auch UDP benutzt Port-Nummern. Diese Nummern sind zwar technisch unabhängig von den Nummern der TCP-Ports, der Übersichtlichkeit wegen wurden sie aber analog zu diesen vergeben.

Client-Programme suchen sich für jede Verbindung eine beliebige freie Port-Nummer. Und natürlich kann ein Rechner gleichzeitig Client und Server sein. Das Programm Netstat.exe listet bestehende Verbindungen auf – Abbildung C.10 zeigt seinen Einsatz.

Abbildung C.10: Bestehende Verbindungen mit Netstat.exe aufgelistet



Der Protokollstapel ist also eine Art Baukastensystem. Häufig wird zum Beispiel zwischen TCP und HTTP ein Secure Sockets Layer (SSL) eingefügt (siehe Abbildung C.9). Das ergibt dann das HTTPS-Protokoll (HTTP Secure), der HTTPS-Server horcht an Port 443. Wegen der Implementation im TCP/IP-Stapel kann SSL auch von allen anderen höheren Protokollen genutzt werden.

SSL ist übrigens eine Erfindung, die von der Firma Netscape offengelegt und dem W3C zur Standardisierung übergeben wurde. Dieser Standard wurde 1999 unter dem Namen TLS 1.0 veröffentlicht.

C.5.2 Der Protokollstapel in .NET

Die in .NET für verteilte Applikationen unterstützten Protokolle benutzen ebenfalls den TCP/IP-Baukasten: Remoting über den TCP-Kanal benutzt direkt TCP (der Port ist frei wählbar), ebenso DCOM (Port 135); Remoting über den HTTP-Kanal benutzt direkt HTTP, und SOAP setzt auf HTTP auf. Dass Remoting über den TCP-Kanal und DCOM an Firewalls scheitern, ist nun leicht verständlich: Die meisten Firewalls lassen ausschließlich HTTP durch. Abbildung C.9 zeigt die Einordnung von Remoting, DCOM, Webdiensten und Webanwendungen in den Protokollstapel.

D Was ist auf der CD-ROM?

Auf der Begleit-CD finden Sie zweierlei:

- Das deutsche *.NET Framework SDK* und das *.NET Framework Service Pack 2*
- Beispielprojekte mit den kompletten Quellcodes zu allen im Buch vorgestellten Codebeispielen

D.1 .NET Framework SDK

Das *deutsche .Microsoft .NET Framework Software Development Kit (SDK)* beinhaltet das deutsche .NET Framework und sowie alle Komponenten, die Sie zum Schreiben, Erstellen, Testen und Bereitstellen von gebrauchsfertigen .NET Framework-Anwendungen benötigen – darunter Compiler für die .NET-Sprachen, eine umfangreiche deutsche Dokumentation, zahlreiche Beispiele und verschiedene wichtige Befehlszeilenwerkzeuge.

Das SDK darf allerdings nicht mit einer integrierten Entwicklungsumgebung wie Visual Studio .NET verwechselt werden. Es handelt sich vielmehr um eine Minimalausstattung, die von Microsoft zum Ausprobieren und für erste Gehversuche mit den .NET-Sprachen von der Kommandozeile aus kostenfrei verbreitet wird, um eine möglichst schnelle Verbreitung des darin enthaltenen .NET-Framework (Redistributable) und der .NET-Komponenten zu erreichen. Für die ernsthafte Software-Entwicklung, wie sie in diesem Buch beschrieben wird, kommen Sie an Visual Studio .NET nicht vorbei.

Für die Installation des SDK starten Sie das Programm *Setup.exe* und folgen den weiteren Anweisungen des Installationsassistenten. Danach installieren Sie das Service Pack 2 durch Aufruf der Datei *NDP10_SP_Q321898-_De.exe*.

Obwohl möglich, ist es nicht empfehlenswert, das SDK in mehreren Sprachen auf einem System zu installieren – schon gar nicht unter Verwendung der Standardvorgaben. Sollten Sie auf Ihrem System beispielsweise eine englische Betaversion des SDK oder von Visual Studio haben, müssen Sie diese zuerst deinstallieren, bevor Sie mit der Installation des deutschen *.NET Framework*

SDK beginnen. Gegen eine mehrsprachige Installation nur des .NET-Framework (Redistributable) und der .NET-Komponenten spricht hingegen nichts

Falls auf Ihrem System bereits eine Installation des deutschen SDK oder von Visual Studio .NET als Release-Version (mit oder ohne Service Pack 1) vorhanden ist, steht es Ihnen frei, nur das Service Pack 2 zu installieren.

D.2 Übersicht über die Beispielprojekte

In den Verzeichnissen *Code/Teil 2* bis *Code/Teil 5* der CD-ROM finden Sie den in den einzelnen Buchteilen vorgestellten Beispielcode in Form fertiger Visual Studio. NET-Projekte – jedes Projekt in einem eigenen Ordner.

D.2.1 Installation

Für die Installation des Beispielcodes verwenden Sie die Kopierfunktionalität des Windows-Explorers.

Um die Projekte mit Visual Studio .NET weiterzubearbeiten, kopieren Sie entweder das gesamte Verzeichnis (ca. 20 MB) auf eine lokale Festplatte Ihres Systems oder nur die Projektordner, die Sie interessieren. In beiden Fällen müssen Sie nach dem Kopiervorgang für alle kopierten Dateien und Verzeichnisse das Schreibschutzattribut entfernen. Dazu markieren Sie den oder die Ordner in einem Fenster des Windows Explorers und rufen das EIGENSCHAFTEN-Fenster auf. Im nächsten Schritt entfernen Sie das Häkchen für das Attribut SCHREIBGESCHÜTZT, klicken auf ÜBERNEHMEN und bestätigen, dass die Attributänderung auch für alle untergeordneten Verzeichnisse erfolgen soll.

Für den Einsatz der zum Teil 4 gehörigen Beispielprogramme empfiehlt es sich auf jeden Fall, den gesamten Ordner *Teil 4* zu kopieren, damit die bestehende Verzeichnisstruktur erhalten bleibt. Um die Web-Anwendungen einsetzen zu können, müssen Sie das Verzeichnis *Code\Teil 4\wwwroot* über das Fenster INTERNET-INFORMATIONSDIENSTE des INTERNETDIENSTE-MANAGER entweder als virtuelles Verzeichnis einer bestehenden Site oder gleich als neue Site einrichten. Außerdem ist es erforderlich, die einzelnen Anwendungen als solche anzumelden. Klicken Sie dazu im EIGENSCHAFTEN-Fenster der Anwendung auf die Schaltfläche ERSTELLEN (nach der Anmeldung lautet die Beschriftung dieser Schaltfläche ENTFERNEN.)

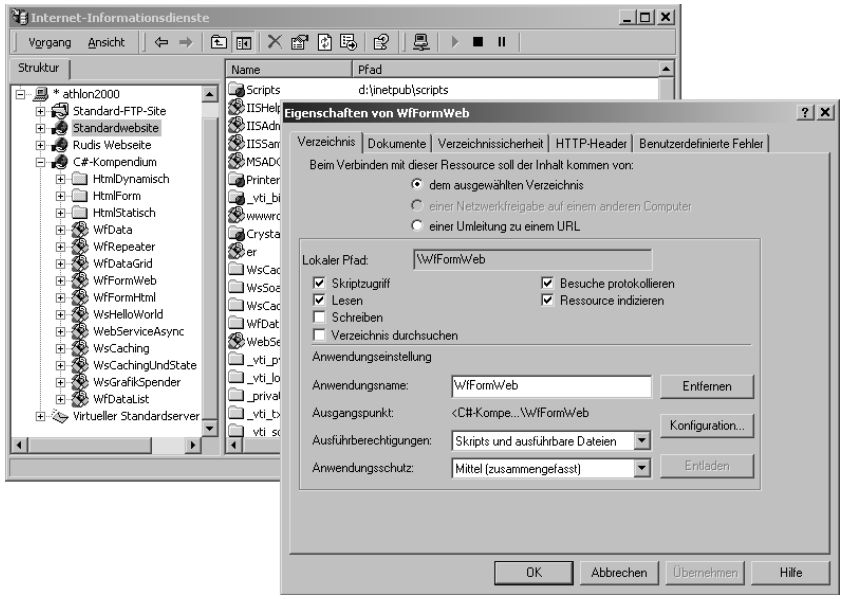


Abbildung D.1: Anmelden einer Web-Anwendung im Internetdienste-Manager

D.2.2 Teil 2 – Die Sprache C#

PrimNumbers (ab Seite 157)

Das Programm stellt eine ungewöhnliche (weil rein prozedurale) Repräsentation für endliche Primzahlenfolgen vor. Der Code zeigt die vollständige Implementierung einer aufzählbaren Klasse auf Basis der Schnittstellen `IEnumerator` und `IEnumerable` sowie die eines rein prozedural arbeitenden Indexers.

TextReader (ab Seite 176)

Die Windows-Anwendung liest eine Textdatei zeilenweise ein, um sie in einem auf dem Formular befindlichen Multiline-Textfeld auszugeben. Die Anwendung demonstriert den unterschiedlichen Zeitbedarf für Stringmanipulationen mit den Datentypen `string` und `StringBuilder`.

DelegateDemo (ab Seite 202)

Ein kleines als Konsolenanwendung ausgelegtes Testprogramm für Delegaten, das den Einsatz einer generischen Methode `LogicTableTernary()` für die Ausgabe einer Wahrheitstabelle für ternäre (=dreiwertige) Logikoperationen demonstriert.

MultiCastDelegateDemo (ab Seite 206)

Konsolenprogramm, das den Umgang mit Delegaten für mehrfache Funktionsaufrufe noch einmal in einem komplexeren Zusammenhang demonstriert.

Typumwandlung (ab Seite 224)

Konsolenprogramm, das die Definition von Typumwandlungsoperatoren für zwei Klassen demonstriert, deren Objekte in der einen Richtung implizit und in der anderen Richtung explizit typkompatibel sind.

Bouncing (ab Seite 254)

Windows-Anwendung á la »TV-Tennis«, die eine Formularklasse für eine timer-basierte Ballanimation vorstellt. Demonstriert unter anderem die Instanzzählung durch ein statisches Datenfeld.

DisposeDemo (ab Seite 292)

Zeigt das Zusammenspiel zwischen `Open()`, `Close()`, `Dispose()` und einem Destruktor für eine Klasse namens `MyDatabaseConnection`. Die Klasse stellt Methoden für den Aufbau, die Statusabfrage und das explizite Schließen einer Datenbankverbindung bereit, ohne jedoch faktisch eine Verbindung mit einer Datenbank herzustellen. »Vergisst« der Client, die zuvor geöffnete Datenbankverbindung zu schließen, sorgt der Destruktor der Klasse dafür, dass die Verbindung automatisch abgebaut wird. Eine von der Implementierung gepflegte Statusvariable ermöglicht es, die Geschehnisse über die Methode `PrintStatus` an der Konsole gut zu verfolgen.

Eigenschaften (ab Seite 297)

Das Codebeispiel zeigt eine einfache Klasse mit vier Eigenschaften, deren Objekte einen Preis mit prozentualem Mehrwertsteueranteil repräsentieren. Geboten sind »Erhalt der Integrität« und »Gültigkeitsprüfung«: Neue Objekte müssen mit einem Mehrwertsteuerfaktor initialisiert werden, der zwischen 0 und 1 liegt. Eine Änderung der `Brutto`-Eigenschaft ändert bei unverändertem Steueranteil auch die Nettoeigenschaft – und umgekehrt. Eine Änderung des Steuerfaktors führt zu einer Änderung der `Brutto`-Eigenschaft bei unveränderter `Netto`-Eigenschaft.

Indexer (ab Seite 301)

Zeigt die Implementation eines Indexers für die Klasse `Preis` aus dem Beispielprojekt *Eigenschaften*.

Indexer1 (ab Seite 303)

Beispiel für die Implementation einer indizierbaren Eigenschaft.

OperatorExample (ab Seite 311)

Zeigt die Implementierung verschiedener Operatoren für eine Vektorklasse namens `Point3D`. Um die Auswertungsreihenfolge von Ausdrücken, in denen diese Operatoren vorkommen, verfolgbar zu machen, bedient sich der Code der in der Klasse `Trace` der durch ein `Conditional`-Attribut bedingt vereinbarten Methode `Msg`. Die Ablaufverfolgung wird durch Definition des Symbols `TRACEMODE` eingeschaltet.

Schnittstellen (ab Seite 324)

Demonstriert die formale Seite des Umgangs mit Schnittstellenklassen. Der Code definiert drei Schnittstellenklassen `MyInterface1`, `IMyInterface2` und `IMyInterface`, wobei die ersten beiden als Basis in der Vererbungsliste der dritten aufgeführt sind. Zudem definiert er eine Klasse `MyClass`, die alle drei (!) Schnittstellen implementiert.

EnumeratorBeispiel (ab Seite 328)

Das Konsolenprogramm definiert eine eigene Schnittstelle `IGeneralEnumerable` auf der Basis von `IEnumerable`, die zusätzlich zu `GetEnumerator()` eine Methode `Item()` für den Elementzugriff und eine Eigenschaft `ItemCount` für Elementanzahl fordert. Eine Klasse, die diese Schnittstelle implementiert, ist unabhängig von ihrem Elementtyp aufzählbar.

AbstrakteKlassen (ab Seite 332)

Beispielprojekt, das den Einsatz einer abstrakten Klasse mit zwei abstrakten und drei nicht-abstrakten Elementen zeigt.

AusnahmenDemo (ab Seite 351)

Demonstriert verschiedene Techniken, die im Zusammenhang mit der Signalisierung und Behandlung von Ausnahmen stehen.

StringOps (ab Seite 364)

Stellt die Methoden `Intern()` und `IsInterned()` vor. Sie geben darüber Aufschluss, wie Strings in .NET implementiert sind.

StringFormat (ab Seite 371)

Demonstriert eine Auswahl von Stringformatierungen für verschiedene Datentypen und Formatierungsausdrücke sowie den Zusammenhang zwischen den Überladungen von `String.Format()` und `Console.WriteLine()`. Die Konsolenausgabe des Programms ist weitgehend selbsterklärend.

StringBuilder (ab Seite 374)

Zeigt verschiedene Möglichkeiten, wie man einen Teilstring aus einem `StringBuilder`-Objekt gewinnt.

ErstellZeit (ab Seite 383)

Die formularlose Windows-Anwendung ist ein kleines, aber durchaus nützliches Tool, das die Auswahl eines Verzeichnisses gestattet und davon ausgehend rekursiv in allen Unterverzeichnissen die Erstell- und Zugriffszeiten aller Dateien und Verzeichnisse auf die aktuelle Uhrzeit ändert.

Checksummen (ab Seite 396)

Die Windows-Anwendung ist ein Testbett für fünf Methoden, die jeweils auf unterschiedliche Weise dasselbe Ergebnis – die Bytesumme einer Datei – berechnen und dabei verschiedene Arten von Streams und Dateizugriffen vorstellen. Vier davon operieren synchron, die fünfte asynchron. Das Formular der Anwendung ermöglicht den Aufruf der Methoden und gibt die relevanten Größen – darunter auch den Zeitbedarf – in der Statusleiste aus.

D.2.3 Teil 3 – Windows-Anwendungen

Gummiband (ab Seite 421)

Das kleine Programm behandelt drei der vier primären Mausereignisse und zeichnet ein »Gummiband« in das Formular, wenn die Maus mit gedrückter linker Taste gezogen wird.

Gummiband1 (ab Seite 424)

Zeigt eine professionellere Implementierung der Anwendung *Gummiband* mit Mitteln der Klasse `ControlPaint`.

Zeitanzeige (ab Seite 429)

Das Beispielprojekt *Zeitanzeige* nimmt eine einfache Anzeige der Systemzeit mit Stunden, Minuten und Sekunden über ein `Label`-Steuerelement vor.

Tastaturanalyse (ab Seite 433)

Demonstriert die Behandlung von zwei der drei Ereignisse für ein Textfeld und nutzt dabei ein Feature, das C# »so ganz nebenbei« standardmäßig für Objekte von Aufzählklassen unterstützt, die mit dem `[Flags]`-Attribut vereinbart sind: Die `ToString()`-Methode drückt den Wert eines Bitvektors als Kombination der Bezeichner der in der Aufzählung für die einzelnen Bitfelder vereinbarten Konstantenwerte aus. Das Programm ermöglicht es, interaktiv zu erforschen, welchen Konstantennamen die `Keys`-Aufzählung für welche Taste definiert.

Tastaturvorschau (ab Seite 437)

Bei diesem Programm betreibt das Formular eine Tastaturvorschau und gibt nur bestimmte `KeyPress`-Ereignisse an seine Steuerelemente weiter. Herausgefilterte Zeichen erscheinen in der Titelzeile des Formulars.

FormBildlauf (ab Seite 457)

Einfache Bildanzeige zur Demonstration der automatischen Bildlaufleisten eines Formulars. Die Anwendung erfragt den Namen einer *.jpg*-Datei und liest deren Inhalt als Hintergrundbild des Formulars ein. Der Anwender kann den Bildlauf sowohl über die Bildlaufleisten als auch per Mausklick abwickeln.

Farben (ab Seite 487)

Das umfangreiche Beispielprojekt stellt die verschiedene .NET-Klassen vor, die mit Farben und Farbnamen zu tun haben. Das Formular ermöglicht es, per Menü die Farben der Klassen `Color`, `SystemColors`, `KnownColor` sowie verschiedene kontinuierliche Farbverläufe auszugeben und die Farbnamen bzw. ARGB-Werte in Erfahrung zu bringen. Demonstriert den praktischen Einsatz der `Tooltip`-Komponente.

HatchBrush-Demo (ab Seite 501)

Demonstriert den Umgang mit Schraffurpinseln. Das Programm zeichnet schlicht alle Schraffuren als Kästchen in den Clientbereich des Fensters und zeigt die Namen der Schraffurstile über eine `ToolTip`-Komponente an.

Mandelbrot (ab Seite 516)

Das optisch ungemein eindrucksvolle und programmiertechnisch sehr anspruchsvolle Beispiel demonstriert anhand einer Fahrt durch die Mandelbrotmenge verschiedene Techniken, die mit der komplexe Grafikausgabe im Zusammenhang stehen. Die einzelnen Techniken sind:

- Aufwändiges Rendering mit Einsatz der `DoEvents()`-Methode.
- Pufferung der Grafikausgabe in Bitmaps
- Umgang mit `Bitmap`-Objekten
- Gummibandanimation zur Bereichsauswahl per Maus (Zoomfunktion)
- Konservierung mehrerer Puffer in einem Stack des Typs `ArrayList`

Blocksatz (ab Seite 532)

Das anspruchsvolle Codebeispiel zeigt den grundsätzlichen Weg auf, der zum Blocksatz führt und deckt dabei gleichzeitig auch die Schwächen der Methode `MeasureString()` auf.

Taschenrechner (ab Seite 556)

Recht einfach gehaltenes Codebeispiel, das die Implementierung eines Taschenrechners zeigt, der alle vier Grundrechenarten beherrscht und auch über ein M-Register zum Speichern und Abrufen von Zwischenergebnissen verfügt.

RadioGroup (ab Seite 568)

Benutzerdefiniertes Steuerelement, das eine Gruppierung von `RadioButton`-Steuerelementen vornimmt.

ListBoxDemo (ab Seite 575)

Demonstriert die wichtigsten Funktionalitäten der Steuerelemente `ListBox` und `ComboBox` anhand einer Auflistung der Systemschriften – darunter auch die Technik des *owner draw*.

Bildlaufleisten (ab Seite 581)

Stellt die Implementierung einer Bildlaufleistenfunktionalität vor, wie man sie sich ähnlich für viele Zeichenprogramme wünschen würde:

- Verschieben der Bildlaufmarke verschiebt Ansicht und Bildlaufmarke
- Mausklicks in die Bereiche zwischen Bildlaufmarke und Pfeilschaltfläche (`LargeXxx`) vergrößern/verkleinern den Zoomfaktor
- Pfeilschaltflächen verschieben die Ansicht, *nicht* jedoch die Bildlaufmarke (`SmallXxx`)

PanelDemo (ab Seite 587)

Demonstriert den Umgang mit den Steuerelementen `Panel` und `Splitter`.

TreeViewDemo (ab Seite 594)

Interessante Implementation, die den Umgang mit dem `TreeView`-Steuerelement demonstriert Das Codebeispiel ist die Implementierung eines Pfadauswahl-Dialogs, wofür sich (leider) kein Steuerelement in der TOOLBOX findet. Um es übersichtlich zu halten, beschränkt es sich auf das lokale Dateisystem des Computers und gibt sich auch keine Mühe, unterschiedliche Laufwerksarten symbolisch zu unterscheiden.

MenüDemo (ab Seite 602)

Einfacher Viewer für Bilder. Demonstriert verschiedene Techniken für die Pflege und den Umbau eines Menüs zur Laufzeit.

MenüDemoOwnerDraw (ab Seite 609)

Erweitert den Funktionsumfang der *MenüDemo*-Anwendung dahingehend, dass die Menüeinträge für die zuletzt geöffneten Dateien vor dem Dateinamen noch eine Miniatur des jeweiligen Bildes zeigen.

Leisten (ab Seite 626)

Das Codebeispiel ist speziell darauf zugeschnitten, den Umgang mit Statusleisten und Symbolleisten zu demonstrieren. Zeigt auch Drag&Drop für `ToolbarButton`-Steuerelemente.

Diaprojektor1, Diaprojektor2 und DiaprojektorOLE (ab Seite 613)

Weitere Codebeispiele für vollständige Anwendungen mit/ohne Menü, Kontextmenü, Tastaturschnittstelle und Vollbildansicht. Die drei Anwendungen sind Variationen eines Viewers für Bitmaps, der – wie ein Diaprojektor – Bilder in passender Skalierung mit Timerautomatik anzeigt. *DiaprojektorOLE* enthält, wie der Name andeutet, eine Unterstützung für Drag&Drop.

Knob (ab Seite 642)

Zeigt den Steuerelemententwurf von »der Pike auf«. Es geht um ein Drehknopf-Steuerelement, das einen Wert aus einem freidefinierbaren Wertebereich als Winkelstellung eines Drehknopfs darstellt und sich mit der Maus oder der Tastatur bedienen lässt – kurzum das Gegenstück zum `TrackBar`-Steuerelement in rund. Es verfügt über eine Beschriftung, zeigt eine Skala mit einer freidefinierbaren Anzahl von Skalenstrichen an, und lässt sich im Gegensatz zum `TrackBar`-Steuerelement wahlweise in beide Richtungen (gegen oder mit dem Uhrzeigersinn) orientieren, bei freidefinierbaren Endstellungen.

MemTextBox (ab Seite 658)

Konzeption einer abgeleiteten Steuerelementklasse, deren Implementierung den Funktionsumfang eines Textfelds dahingehend erweitert, dass es sich die letzten *n* Einträge merkt und während der Eingabe dezent vorschlägt.

Standarddialoge (ab Seite 680)

Demonstriert den prinzipiellen Umgang mit den sechs Standarddialogen.

SuchenErsetzen (ab Seite 699)

Zeigt die Implementierung eines moduslosen Suchen/Ersetzen-Dialogs.

MyDialog (ab Seite 714)

Demonstriert die Implementierung eines modalen Eigenschaftendialogs.

MyDerivedDialog (ab Seite 718)

Demonstriert die Prinzipien der Formularverwebung anhand eines per Ableitung erweiterten Eigenschaftendialogs mit `TabControl`-Steuerelement.

Zwischenablage (ab Seite 725)

Anspruchsvolles Codebeispiel, das die Nutzung der Zwischenablage vorstellt.

DragDrop (ab Seite 736)

Codebeispiel, das die Implementierung von Drag&Drop-Funktionalität und das Benutzerfeedback über die Mauscursorform zeigt.

D.2.4 Teil 4 – Verteilte Programme und Web-Anwendungen

XMLIntro (ab Seite 793)

Zeigt das Transformieren von XML-Code mit einem Stylesheet unter Verwendung der entsprechenden XML-Klassen.

XmlValidierung (ab Seite 801)

Zeigt anhand verschiedener XML-Dokumente die Möglichkeiten für die Validierung von XML-Code.

IEHosting (ab Seite 816)

Zeigt, wie man den Internet Explorer als Steuerelement in ein Formular einbindet und damit einen Webbrowser implementiert.

DNS (ab Seite 820)

Demonstriert DNS-Abfragen und Rückwärtsabfragen unter Verwendung der `DNS`-Klasse.

WHOIS (ab Seite 824)

Demonstriert Whois-Abfragen für verschiedene Whois-Server.

Mail (ab Seite 828)

Demonstriert den Versand einer E-Mail auf Basis eines `MailMessage`-Objekts.

Pop3 (ab Seite 834)

Stellt das automatisierte Abfragen eines Mail-Kontos durch die `TcpClient`-Klasse vor. Das Beispielprogramm kann sich mit einem POP3-Server verbinden, die Betreffzeilen der vorhandenen Mails auflisten und die Verbindung wieder beenden.

ScreenScraper (ab Seite 841)

Zeigt die gezielte Extraktion von Informationen aus HTTP-Seiten. Sammelt Daten von den Webseiten zweier Wintersportorte, sucht die Schneehöhen heraus und zeigt sie in einem Textfeld. Im Sommer können stattdessen die Windstärken an zwei Segel- bzw. Surfrevieren abgefragt werden.

AutoSuche (ab Seite 846)

Automatisierte Suche nach einem Auto auf der Basis einer bestehenden Webseite mit Datenbankfunktionalität. Die Implementation arbeitet mit der Klasse WebClient und demonstriert den Einsatz von HTTP GET und HTTP POST-Anfragen.

AutoSucheAsync (ab Seite 852)

Asynchrone Fassung von *AutoSuche*.

PlugProt (ab Seite 864)

Demonstriert das Konzept der austauschbaren Protokolle (Pluggable Protocols) mit den Klassen `WebRequest` und `WebResponse`.

HtmlStatisch (ab Seite 884)

Codebeispiel für statisches HTML.

HtmlForm (ab Seite 891)

Besteht aus drei Formularen und einer CSS-Datei. Das erste Formular, *Anmeldung.aspx*, implementiert eine Benutzeranmeldung, verwendet Script-Code zur Validierung der Eingaben und demonstriert die Probleme der Zustandslosigkeit. Das zweite Formular, *AnmeldungCli.aspx*, zeigt die Positionierung von HTML-Steuerelementen, das dritte, *Resultat.aspx*, der Vollständigkeit halber eine Zusammenfassung der Eingaben.

WfFormHtml (ab Seite 900)

Codebeispiel für die Implementation von Web Forms-Seiten mit HTML-Serversteuerelementen.

WfFormWeb (ab Seite 910)

Codebeispiel für die Implementation von Web Forms-Seiten mit Webserver-Steuerelementen.

WfData (ab Seite 916)

Anwendung, die das Binden von Web Forms-Seiten und Serversteuerelementen an Daten demonstriert.

WfDataGrid (ab Seite 919)

Codebeispiel für das Paginieren und Sortieren mit dem `DataGrid`-Steuerelement.

WfRepeater (ab Seite 924)

Zeigt, wie man eine Zebrastreifentabelle mit dem `Repeater`-Steuerelement generiert.

WfDataList (ab Seite 928)

Zeigt, wie man das `DataList`-Steuerelement verwendet, um Listeneinträge auf mehrere Spalten zu verteilen.

WsHelloWorld (ab Seite 943)

Codebeispiel, das in Webdienste einführt.

WebServiceAsync (ab Seite 955)

Demonstriert den asynchronen Aufruf von Methoden und den Einsatz von Timeouts zum Abbruch von Webdienst-Anforderungen.

WsCachingUndState (ab Seite 961)

Stellt eine Technik für die Zustandskonservierung von Webdiensten auf der Basis eines `Session`-Objekts vor.

WsCaching (ab Seite 972)

Demonstriert fortgeschrittene Techniken für das Caching im Zusammenhang mit Webdiensten.

WsGrafikSpender (ab Seite 985)

Implementiert einen »kostenpflichtigen Grafikspender« auf der Basis eines Webdienstes. Der Benutzer gibt einen Text und Design-Festlegungen an, wie die zu verwendende Schriftart, und das Programm liefert eine entsprechende Grafik.

D.2.5 Teil 5 – Weiterführendes

ThreadwithoutClass (ab Seite 1003)

Codebeispiel für den Aufruf von Threads ohne eigene Klasse.

ThreadIAsyncResult (ab Seite 1014)

Demonstriert die Technik des asynchronen Aufrufs von Methoden.

ThreadSyncInCB (ab Seite 1014)

Codebeispiel zur Synchronisation von Threads.

InteractiveThread (ab Seite 1020)

Praxisnahe Implementation einer Anwendung mit »Spracherkenner-Thread«, die Lösungen für die Interaktion mit Threads vorstellt.

InteractiveThread2 (ab Seite 1030)

Erweiterte Fassung von *InteractiveThread*, in der Erkenner-Thread erkannte Befehle in einer Liste speichert.

RCWPhotoshopImport (ab Seite 1047)

Demonstriert die Integration eines COM-Objekts (Adobe Photoshop) in ein .NET-Programm.

MSAcalImport (ab Seite 1055)

Demonstriert im Wesentlichen die Projektstruktur für den Einsatz eines ActiveX-Steuerelements in einer .NET-Anwendung.

Net_for_COMDemo (ab Seite 1061)

Demonstriert den Einsatz einer NET-DLL als COM-Komponente.

DllPath (ab Seite 1073)

Demonstriert den Import einer Dll-Routine in eine .NET-Anwendung.

DLLLoadTime (ab Seite 1074)

Deckt anhand einer in Delphi geschriebenen DLL auf, zu welchem Zeitpunkt eine DLL vom .NET-Framework geladen und wann sie wieder aus dem Speicher hinausbefördert wird.

DllCheck (ab Seite 1076)

Stellt eine Technik vor, wie man feststellt, ob eine bestimmte DLL auf dem System verfügbar ist.

CompleteDecl (ab Seite 1078)

Zeigt die Konvertierung verschiedener Stringformate

GetUserName (ab Seite 1079)

Demonstriert den Umgang mit API-Funktionen, die Ergebnisse in aufruferseitig bereitgestellten Stringpuffern frei wählbarer Größe hinterlassen.

uniondemo (ab Seite 1085)

Zeigt den Einsatz des `StructLayout`-Attributs für die Definition einer Union, die als Puffer für den Aufruf von API-Routinen verwendbar ist.

FindFirst (ab Seite 1086)

Demonstriert den Einsatz von API-Routinen, die mit Strukturen operieren. Basiert auf einem leicht nachkorrigierten Beispiel von Microsoft, das die Dateien im aktuellen Verzeichnis auflistet und dazu die API-Funktionen `FindFirst()`/`FindNext()` verwendet.

EnumWindows (ab Seite 1091)

Demonstriert den Umgang mit API-Routinen, die eine Rückrufroutine erwarten. Das Programm zählt alle aktuell existierenden Fenster auf.

ShutdownWindows (ab Seite 1093)

API-Aufruf für einen Plattformdienst von Windows, den .NET selbst nicht zur Verfügung stellt: Zeigt das Herunterfahren von Windows und das damit verbundene Hantieren mit Sicherheitsattributen auf Systemebene.

Stichwortverzeichnis

!
#define 238, 727
#endif 725
#if 725
#region-Präprozessordirektive 414, 646
% (Operator) 193
&& (Operator) 193
&, Definition von Alt-Tastenkürzel 550
***/** 128
++ (Operator) 195
-- (Operator) 195
! (Operator) 193
!= (Operator) 194
.ani 420
.asmx 870
.aspx 870
.cs-Datei 43, 51
.csproj 51
.csproj.user 51
.cur 420
.dll 87
.h 87
.hpp 87
.ico 51
.idl 1043
.lib 87
  Metadaten 1043
.NET
  als neues Fundament für Anwendungsentwicklung 31
  -Ausnahmeklassen 349
  -Basisklassen 118
  C++, verwaltete Erweiterung 91
  Dateizugriff 390
  -Framework 82
  Hüllklassen für COM 1047
  Installation 1107
  Multithreading 999
  Objektmodell 33
    – Struktur 37
  Philosophie 29
  Portabilität 32
  Rolle von C# als Sprache für 87, 91
  Schichtung 82
  Service Packs 1108
  -Speicherverwaltung 362
  Systemdienste 91
  Typmanagement 90
  -Typverwaltung 43
  und COM 1041
  vs. COM 85
.NET Framework
  SDK 1139
  Service Pack2 1139
.NET-Framework
  Einführung in 32
.NET-Klassenhierarchie
  als gemeinsame Typbibliothek 40
.NET-Laufzeitumgebung
  Installation 1112
.NET-Sicherheitskonzept 34
.NET-Sprachen 32
.ocx 1053
.resx 51
.sln 51
.snk 1063
.suo 52
.tlb 89
  Delphi-Problem mit 1067
  Projektverweis hinzufügen 1047
/* 128
/checked 142
/unsafe (Compilerschalter) 196
== (Operator) 194
@ (Schlüsselwörter als Bezeichner kennzeichnen) 105, 106, 174
\-Escape-Sequenzen 170
_, Problem bei Konsolenausgabe 370
_Entkoppelung 1019
_Threads
  Entkoppelung 1019
    – Beispiel 1026
|| (Operator) 193

32-Bit-Technologie 84

A
Abbildung
indentische 464
lineare 464
Ablaufsteuerung 169
Ableitung 316
Steuerelemente, eigene 633
von Formular 708
– Implementierungsstrategien 710
– Verkapselung 711
von Schnittstellen 317
Abort 675
AbortRetryIgnore 674
about-Protokoll 818
abstract 102, 246, 334
Datenfelder als 250
-Klasse 319
-Methoden 319
vs. interface 322
Abstrakte Elemente 332
Abstrakte Klasse 331
als Ersatz für Schnittstelle 336
Implementierungszwang 335
in C# 331
in C++ 331
ohne abstrakte Elemente 334
Syntax 332
vs. Schnittstelle 322, 335
AbstrakteKlassen (Codebeispiel) 332
Abstrakter Datentyp 112
AcceptButton-Eigenschaft 693
AcceptsReturn-Eigenschaft 569
AcceptsTab-Eigenschaft 569
Access 86
Accessibility 741
Accessor, set- und get- 297
Activate() 431
Activated-Ereignis 448, 449
ActiveBorder-Eigenschaft 485
ActiveCaption-Eigenschaft 485
ActiveCaptionText-Eigenschaft 485
ActiveForm-Eigenschaft 431
ActiveX 82, 84
Codebeispiel 1055
Import 1054
in Visual Studio .NET 1053
Add() 376, 615
AddArc() 476
AddBezier() 476
AddBeziers() 476, 477
AddClosedCurve() 476, 478
AddCurve() 476, 478
AddEllipse() 476, 477
AddExtension-Eigenschaft 678
AddLine() 475
AddLines() 476
AddPie() 476
Address Resolution Protocol (ARP) 1133
AddString() 480, 482
AddStrip() 613, 615
umgehen 616
Adress Resolution Protocol 1133
AfterCheck-Ereignis 589, 591
AfterCollapse-Ereignis 589, 591
AfterExpand-Ereignis 589, 591
AfterLabelEdit-Ereignis 590, 591
AfterSelect-Eigenschaft 589
AfterSelect-Ereignis 591
Aktivierungsreihenfolge 550
Alignment-Eigenschaft 480, 497, 530, 704
AllowDrop-Eigenschaft 546, 554, 630, 737, 743
AllowSomePages-Eigenschaft 688
AllPaintingInWmPaint 645
Alpha-Blending 460
Alpha-Wert 484
AltDirectorySeperatorChar-Eigenschaft 386
Alt-Eigenschaft 433
Alt-Tastenkürzel 550
Anchor 547
AnchorStyles-Aufzählung 547
Ändern von Variablenwerten (Debugger) 72
Andocken, Steuerelement 587
AnsiBStr 1078
Antiviren-Software 1109
Anweisung 123
break 146, 148, 156
case 146
continue 148, 156
do 150
for 148
foreach 150
goto 147, 148, 156
if 144
Kontrollstrukturen 143
return 156
Sprung- (Überblick) 156
switch 145
throw 156
while 149

Anweisungsblock 125
untergeordneter 127
Anweisungsfolge 124
Strukturierung 125
Anwendung
anhalten 74
Startoptionen 59
übersetzen und starten 54
Anwendung, HTTP 871
Anwendungsübergreifender Datenaustausch 40
Anzeigestatus, siehe ViewState-Objekt 907
Apfelmännchen 523
AppDomain-Klasse
GetCurrentThreadId() 855
GetCurrentThreadId()-Methode 853
Appearance-Eigenschaft 565, 567, 621, 704
Append (MatrixOrder) 465
Append() 374
AppendFormat() 374
AppendText() 384, 570
Application Domain 46, 872
Application.DoEvents() 516
Application.Run() 413, 415, 440
formularloser Programmstart 722
ApplicationContext-Klasse 722
ApplicationException-Klasse 838
Ableitung, eigene 837
Application-Klasse
StartupPath-Eigenschaft 847, 853
Application-Objekt 964
Apply-Ereignis 685
AppWorkspace-Eigenschaft 485
ARGB-Farbraum 490, 506
ARGB-Modell 484
ArgumentException-Klasse 349, 354
ArgumentNullException-Klasse 349
ArgumentOutOfRangeException 973
Arithmetische Operatoren 193
ARP 1133
-Anfrage 1133
arp.exe 1133
Array
als Parameter für Methode 214
Eigenschaft als 302
Array.Clear() 221
Array.Copy() 214
ArrayList-Klasse 377, 796
Arrays 211
Durchsuchen 220
eindimensionale 214
Elementyp 215
Elementzugriff 214
Initialisierung 214
Instanziierung 213
Laufzeitfehler 216
mehrdimensionale 217, 218
Multi- 217, 218
polymorphe Initialisierungen 215
Rang 217
Rank-Eigenschaft 217
Sicherheit 213
Sortieren 220
Vereinbarung 213
Wertlisteninitialisierung 214, 215, 217
Zuweisung von 214
Arten der Vererbung 316
as (Operator) 196
ASCII-Code 170
ASCIIEncoding-Klasse 170
ASP 883, 889
ASP.NET 32, 869, 884
Anwendung im ASP-Stil implementiert 891
C# im -Block 890, 893
C# im script-Element 891
HTTP-Pipeline 871
Installationsvoraussetzungen 1113
JavaScript im script-Element 894
Page-Anweisung 890
VB.NET im -Block 890
Verbesserungen gegenüber ASP 870
Web Form 899
Webanwendung 881
ASP.NET Arbeitsprozess
Konfigurieren 877
AspNet_IsApi.Dll 871
AspNet_Wp.Exe 871
Debugger anfügen 952
HTTP Pipeline in 981
HTTP-Pipeline in 872
Konfigurieren 980
Starten 952
Assembly 93, 731
(Überblick) 43
Abschnitte 44
Cache 93
Einsprungpunkt 279
genutzte 45
ohne GUID 1062
Pools 93
private 44

Assembly Cache 666
Steuerelementbibliothek aufnehmen in 667
AssemblyInfo.cs
starker Name 1063
Assemblyinfo.cs 51
AsyncCallback-Delegate 855
AsyncCallback-Klasse 854, 855
asynchron
Ausführung 124
Dateioperationen 400
Dateizugriff 401
– Rückrufe 401
– Statusmeldung 404
Destruktion 287
Asynchronous Pluggable Protocol 819
AsyncState-Eigenschaft 402, 404
-Attribut 183
Attribut
Conditional 311
Flags 183, 433
-klassen 245
Attribute 643
Browsable 649
ClassInterfaceType.AutoDual 1065
DefaultEvent 645
DesignerSerializationVisibility 647, 649
Aufgabenliste 52, 128
Auflistungen 375
Auflistungsklassen 375
eigene
– Codebeispiel 378
– gangbare Wege für die Implementierung 377
Aufrufliste (Visual Studio .NET) 71
Aufrufliste von Delegaten 204
Aufrufmechanismen 261
Aufrufparameter einer Methode 261
Aufrufstapel 345
Aufzählungstypen 182
aufzählen 185
Basistyp 184
Ausführen bis Rücksprung (Visual Studio .NET) 73
Ausführung
synchron vs. asynchron 124
Ausführungsgeschwindigkeit 117
Ausführungssicherheit 34
Ausgabebereich, maskieren 479
Ausgabegerät 459
Ausgestaltung eines Dialogfelds 703
Ausnahme
-behandlung 339
– Mechanismus 340
– sprachübergreifende 341
– strukturierte 342
– Syntax 345
-klassen 344
– .NET- 349
– benutzerdefinierte 353
Regeln für richtige Dosierung 354
Ausnahmebehandlung 821, 825, 830, 836, 838, 840, 843, 848, 855, 862, 866, 959
eigene Ausnahmen-Klasse ableiten 837
AusnahmenDemo (Codebeispiel) 351
Ausnahmenhierarchie 838
Austauschbare Protokolle 863
Auswertungsreihenfolge, Kommaoperator 125
AutoCheck-Eigenschaft 565, 567
Automatisierungsstandard 89
AutoReset-Eigenschaft 428
AutoScroll-Eigenschaft 455
Statusleiste, Problem mit 625
AutoScrollMargin-Eigenschaft 456
AutoScrollMinSize-Eigenschaft 455
AutoScrollPosition-Eigenschaft 456
Besonderheiten 456
AutoSize-Eigenschaft 569, 621
AutoSuche (Codebeispiel) 846
AutoSucheAsync (Codebeispiel) 852
AXImp.exe 1054
Listings 1057
Speicherort 1055

B
BackColor-Eigenschaft 547
BackgroundImage-Eigenschaft 445, 489
Backup des Projektordners 641
base 232, 249, 274
Base64-Format 755
BaseStream-Eigenschaft 395
BASIC 86
Basisklasse 318
Formular als 708
Formular planen als 716
object als 317
Sicherheit der Implementierung 709
zum Begriff 316
Basisklassenkonstruktor 284
Baumstruktur
Darstellung in TreeView-Steuerelement 588
Füllen 596
Suchen in 596

bedingte Symbole 1116
Bedingte Verzweigung 144
Befehlstasten 431
BeforeCheck-Ereignis 589, 591
BeforeCollapse-Ereignis 589, 591
BeforeExpand-Ereignis 589, 591, 597
BeforeLabelEdit-Ereignis 590, 591
BeforeSelect-Eigenschaft 589
BeforeSelect-Ereignis 591
BeginContainer() 510
BeginEdit() 589, 593
BeginInvoke() 1015
BeginPrint-Ereignis 688
BeginRead() 393, 400, 401, 404
BeginUpdate() 574
BeginWrite() 393, 400
Begleit-CD 1139
Behandlungsmethode
Ereignis_ einfügen 417
hinzufügen 543
löschen 543
Standardereignis, ergänzen für 417
Beispielprojekte
Installation 1140
Übersicht über 1140
Benannte Farben 484, 486, 490
benanntes Literal 236
Benennungskonventionen 104
eigene 109
Schreibweise für Bezeichner 110
Strukturierung von Bezeichnern 111
ungarische Notation 109
benutzerdefinierte
Ausnahmen 353
Datentypen 111, 163, 182
Benutzerdefiniertes Steuerelement
Assistent 634
vs. Benutzersteuerelement 638
Benutzerkonto (Visual Studio .NET) 62
Benutzersteuerelement
_e in Steuerelementbibliothek zusammenfassen 665
Assistent 634
eigenes _ in Toolbox verfügbar machen 665
grafisches Layout 643
implementieren (Übersicht) 638
Testbett für 640
Vererbung von 669
– Ausgangssituation 670
– schrittweise 670
Benutzersteuerelement hinzufügen (Befehl) 639
BeOS 84
Besitzerzeichnung 608
Bezeichner
qualifizieren 97
Bezeichnerwahl 105
Benennungskonventionen 104
– ungarische Notation 109
CLS-konform 105
eigene Benennungssysteme 109
Namensraum 96
Schreibweise 110
Strukturierung 111
Bezierkurve 476, 511
Bibliothek
dynamische 87
sprachspezifische 87
statische 87
Bibliotheken
dynamische 1042
statische 1041
Bildlauf des Formulars, Eigenschaften und Methoden im Überblick 455
Bildlaufleisten 455
des Formulars verwenden 455
Bildlaufleisten (Codebeispiel) 581
Bildliste 613
Kacheln einfügen 615
Binärdatei 399
BinaryReader-Klasse 391, 392, 395, 399
BinarySearch() 220
BinaryWriter-Klasse 391, 392, 395
Bindung
frühe (COM) 89, 1044
– für .NET-Komponenten 1065
späte (COM) 89, 1044
– für .NET-Komponenten 1064
Bitmap, zeichnen in 515
Bitmap-Klasse 485, 518
Instanz als Ressource einlesen 740
Konstruktor 990
Save() 986
SetResolution() 986
Bitvektor 183
enum- 183
bitweise
Operationen 194
Operatoren 193
Blockkommentar 127, 128
Blocksatz (Codebeispiel) 532

Bold 482
Bold-Eigenschaft 529
bool 169
  Literal 169
Boolean 169
boolesche Operatoren 193
BorderStyle-Eigenschaft 621
Bottom-Eigenschaft 547
Bouncing (Codebeispiel) 254
Bounds-Eigenschaft 547
Boxing 181, 227
  struct 188
break-Anweisung 146, 148, 156
BringToFront() 448, 551, 587
Broadcast-Paket 1134
Browsable-Attribut 649
Browser 1137
Brushes-Klasse 500
Brush-Klasse 494, 500
BufferedStream-Klasse 392, 398
BuildAction-Eigenschaft 740
ButtonBase-Klasse 563
  Eigenschaften 563
ButtonClick-Ereignis 619, 623
  Behandlung 628
ButtonDropDown-Ereignis 624
Button-Eigenschaft 421
Button-Klasse 564
  Eigenschaften 564
Buttons-Eigenschaft 621, 623
ButtonSize-Eigenschaft 621
Button-Steuerelement 564
byte 166
Byte Order Mark 754

C

C# 331
  Abstammung 95
  als Sprache von .NET 87, 91
  als vollständig objektorientierte Sprache 116
  Dateizugriff 390
  grundlegende Konzepte 95
  Integrierte Datentypen 163
  Klassenkonzept 244
  Namensraum 98
  Objektbegriff von 242
  Operatorüberladung 304
  optionale Parameter 271
  Quellcode 43
    – Sicherungskopien 76
  reservierte Schlüsselwörter 106
  Schnittstelle 323
  Sprachkern 95
  Standardoperatoren 191
  Typumwandlung 222
  Zeigerkonzept 113
C#, warum eine neue Sprache 81
C++ 83
  Vererbung 316
  Verteilte Anwendungen mit 84
  verwaltete Erweiterung 91
  verwaltete Erweiterungen 33
CacheItemPriority-Klasse 973
Call-by-reference-Aufruf 261
  Laufzeitvorteile 265
  Verweistyp 264
Call-by-value-Aufruf 261
  Verweistyp 263
Cancel 675
CancelButton-Eigenschaft 693
Cancel-Eigenschaft 448
CanFocus-Eigenschaft 431, 547
CanRead-Eigenschaft 393
CanSeek-Eigenschaft 393
CanSelect-Eigenschaft 548
CanUndo-Eigenschaft 569
CanWrite-Eigenschaft 393
Capacity-Eigenschaft 373
Capture-Eigenschaft 547
Cascading Style Sheets 748, 883, 886
case-Zweig 145, 146
catch-Block 346
CausesValidation-Eigenschaft 548
CD-ROM 1139
Ceiling() 469
CenterColor-Eigenschaft 506
CenterParent 693
CenterPoint-Eigenschaft 506
CenterScreen 693
CGI 883
ChangeExtension() 387
char 169, 170
  Literal
    – Escape-Sequenzen 170
  Literale 170
Character 531
Character Reference 756
CharacterCasing-Eigenschaft 569
CheckAlign-Eigenschaft 565, 567
CheckBoxes-Eigenschaft 589, 590

CheckBox-Klasse 565
Eigenschaften 565
Ereignisse 566
CheckBox-Steuerelement 565
checked 142, 343
CheckedChanged-Ereignis 565, 566
Checked-Eigenschaft 565, 566, 567, 592, 602, 603
CheckFileExists-Eigenschaft 678
CheckPathExists-Eigenschaft 678
CheckStateChanged-Ereignis 565, 566
CheckState-Eigenschaft 565, 566
Checksummen (Codebeispiel) 396
class 245
-Klasse 244
– Konstruktor 284
-Vereinbarung, Aufbau der 247
Class-C-Netz 1131
ClassInterfaceType.AutoDual 1065
Clear() 376, 570, 615
ClearSelected() 574
ClearUndo() 570
Click-Ereignis 417, 425, 426
Clicks-Eigenschaft 421
Clientbereich
als Panel 586
ClientRectangle-Eigenschaft 548
ClientSize-Eigenschaft 548
Clipboard-Klasse 725, 727
GetDataObject() 729
SetDataObject() 728
Clippingbereich 473
ClipRectangle-Eigenschaft 461, 462, 467, 514, 518
Clone() 214, 219, 365, 593
CloneMenu() 607
Close() 290, 393, 398, 448
Closed-Ereignis 449
Close-Ereignis 442
CloseFigure() 475, 478
Closing-Ereignis 448, 449, 696, 700
CLR
Codeverwaltung 46
Typverwaltung 43
Überblick 41
CLS 38
konforme Bezeichnerwahl 105
-konformer Code 40
Regelsatz 39
Code
Wiederverwendung von 40
Code, in eine eigene Datei auslagern 658
Codeansicht 636, 637
Codebeispiel
(Übersicht) 1140
AbstrakteKlassen 332
AusnahmenDemo 351
AutoSuche 846
AutoSucheAsync 852
Bildlaufleisten 581
Blocksatz 532
Bouncing 254
Checksummen 396
CompleteDecl 1078
DelegateDemo 202
DisposeDemo 292
DllCheck 1076
DLLLoadTime 1074
DllPath 1073
DNS 820
Eigenschaften 297
EnumeratorBeispiel 328
EnumWindows 1091
ErstellZeit 383
Farben 487
FindFirst 1086
FormBildlauf 457
Formularvererbung 713
GetUserName 1079
Gummiband 421
HatchBrush-Demo 502
HtmlForm 891
HtmlStatisch 884
IEHosting 816
IGeneralEnumerable 327
Indexer 301
Indexer1 302
InteractiveThread 1020
InteractiveThread2 1030
Knob 642
Leisten 626
ListBoxDemo 575
Mail 828
Mandelbrot 516
MemTextBox 658
MenüDemo 602
MenüDemoOwnerDraw 609
MSAcalImport 1055
MultiCastDelegateDemo 206
MyDerivedDialog 713
MyDialog 714
Net_for_COMDemo 1061

OperatorExample 311
PanelDemo 587
Pfadauswahl per TreeView 594
PlugProt 864
Pop3 834
PrimNumbers 157
RadioGroup 568
RCWPhotoshopImport 1047
Schnittstellen 324
ScreenScraper 841
ShutdownWindows 1093
Standarddialoge 680
StringBuilder 374
StringFormat 371
StringOps 364
SuchenErsetzen 699
Taschenrechner 557
Tastaturanalyse 433
Tastaturvorschau 437
TextReader 176
ThreadIAsyncResult 1014
ThreadSyncInCB 1013
ThreadwithoutClass 1003
TreeViewDemo 594
Typumwandlung 224
uniondemo 1085
WebServiceAsync 955
WfData 916
WfDataGrid 919
WfDataList 928
WfFormHtml 900
WfFormWeb 910
WfRepeater 924
WHOIS 824
WsCaching 972
WsCachingUndState 962
WsGrafikSpender 985
WsHelloWorld 943
XMLIntro 793
XmlValidierung 801
Zeitanzeige 429
Zwischenablage 727

Codebeispiel RadioGroup 633
Codegerüst 439, 440, 636
erweitern 416
Windows-Anwendung 412
Codesicherheit 33
Codeverwaltung 46
Codierung 170
Collapse() 589, 593
CollapseAll() 591
Collect() 288
CollectionBase-Klasse 376
Collection-Klassen 375
Collections-Namensraum 155
ColorDepth-Eigenschaft 613, 614
ColorDialog-Klasse 685
Color-Klasse 486
FromName() 986
Color-Struktur 484, 490
ColumnWidth-Eigenschaft 572
COM 1041
.NET-Hüllklassen 1047
.NET-Komponenten in 1060
– Codebeispiel 1061
– Grenzen 1067
– GUID-Problem 1062
– späte Bindung 1064
– Voraussetzungen 1060
Benennungsschema in .NET 1048
Bestandteile des 84
DCOM 1046
Delegation als Ersatz für Vererbung 90
direkte Schnittstellendeklaration 1049
Fehlerprüfungen 1051
frühe Bindung 89
Grenzen 1044
GUID und Versionskonflikt 93
IDispatch-Schnittstelle 89
IUnknown-Schnittstelle 89
Klassenbibliotheken 1044
Ladezeitpunkte 1051
Metadaten 1043
– direkte Deklaration 1049
Microsoft Transaction Server 1046
-Objekte 88
ohne Stellvertreterklassen 1059
Parallelbetrieb unmöglich 1045
Registrieren von Komponenten 1044
Rolle der Registrierung 88, 1044
-Schnittstellen 322
späte Bindung 89
Syntaxhilfe für Hüllklasssen 1050
Typbibliothek importieren 1048
Typverwaltung 42
Typverwaltungsmechanismus 93
Überblick 1041
virtuelle Methoden 1045
vs CTS, als Objektmodell 36
vs. .NET 85

COM+ 90, 1046
COM+ Services 813
Combine() 387
CombineMode-Aufzählung 479
ComboBox-Klasse 571
  Eigenschaften 572
  Ereignisse 575
  erweitern 664
  Methoden 574
  SelectedIndex 990
ComboBox-Steuerelement 571
commentreport.css 134
Common Language Runtime 41
Common Language Specification 38
  Regelsatz 39
Common Object Model 1041
Common Type System
  Überblick 35
CommonDialog-Klasse 676
CompareTo() 117, 365
Compilerschalter
  /checked 142
CompleteDecl (Codebeispiel) 1078
Component Object Model 84
CompositingMode-Eigenschaft 510
CompostingQuality-Eigenschaft 510
Console.Write()
  Formatierungszeichenfolgen 368
Console.WriteLine() 268
  variable Parameteranzahl 266
const 102, 230, 238, 247
Container 708
ContainerControl-Klasse 443
Container-Eigenschaft 548
Container-Objekt 918
  DataItem 918
Container-Steuerelemente 585
Contains() 376, 472, 551, 615
ContainsFocus-Eigenschaft 431, 548
ContentAlignment-Aufzählung 550
ContextMenu-Eigenschaft 548, 607
  NotifyIcon 722
ContextMenu-Klasse
  Symbolschaltflächen 620
ContextMenu-Steuerelement 607
continue-Anweisung 148, 156
Control.MouseButtons-Eigenschaft 419
Control.MousePosition-Eigenschaft 419
ControlCollection-Klasse 375, 447
ControlDarkDark-Eigenschaft 486
ControlDark-Eigenschaft 485
Control-Eigenschaft 433, 485
Control-Klasse
  als Basisklasse für Steuerelemente 540
  als Vererbungsbasis 443
  Eigenschaften 546
  Ereignisse 554
  Invoke() 1012
  InvokeRequired() 1013
  Methoden 551
  Thread-Synchronisation 1012
  vererbte Eigenschaften 546
  vererbte Methoden 551
ControlLight-Eigenschaft 486
ControlLightLight-Eigenschaft 486
ControlPaint-Klasse 644
  DrawReversibleFrame() 423
ControlRemoved-Ereignis 554
Controls-Auflistung 417, 447, 548
  Steuerelemente einfügen 557
Controls-Eigenschaft 548
ControlStyles
  AllPaintingInWmPaint 645
  DoubleBuffer 645
  UserPaint 645
ControlText-Eigenschaft 486
Convert() 170
Convert-Klasse 222
CoordinateSpace-Aufzählung 467
Copy() 214, 384, 725
Copy-Operation (Drag&Drop) 736
Copyrightvermerk 44
CopyTo() 173, 214, 216, 219, 365, 376
CR 174
Create() 385
CreateControl() 551
Created-Eigenschaft 548
CreateDirectory() 382
CreateGraphics() 510, 551
CreateSubKey() 714
CreateText() 385
CreationTime-Eigenschaft 389
CryptoStream-Klasse 392
CSS 883, 886
  Eingebettet 888
  Extern 888
  Inline 888
CSS-Editor 907
CTS 90
  alle Datentypen als Klassen 117

Überblick 35
vs. COM, als Objektmodell 36
CulturInfo-Klasse 369
CurrentCulturInfo 369
Cursor.Current-Eigenschaft 420
Cursor.Draw() 420
Cursor.Hide() 420
Cursor.Point-Eigenschaft 420
Cursor.Position-Eigenschaft 419
Cursor.Show() 420
Cursor-Eigenschaft 419, 420, 548
Cursor-Klasse 420
Eigenschaften 420
Instanz als Ressource einlesen 740
CursorKlasse
Methoden 420
CustomColors-Eigenschaft 685
CustomEndCap-Eigenschaft 499
CustomLineCap-Klasse 498
CustomStartCap-Eigenschaft 498
Cut() 570, 725

D
Darstellung
Baumstruktur 588
DashCap-Eigenschaft 498
DashPattern-Eigenschaft 498
DashStyle-Eigenschaft 498
DataBind()-Methode 916
DataBinder-Klasse
Eval() 918
Data-Eigenschaft 473, 630
DataFormats-Klasse 631, 729
Konstanten 729
DataGrid-Webserver-Steuerelement 917, 919
An Datenmenge binden 919
DataBind() 919
DataSource 919
PageIndexChanged 922
Paging 922
SortCommand 923
Sortierung 922
DataListItem-Klasse
FindControl() 936, 938
ItemType 938
DataList-Webserver-Steuerelement 917, 927
An Datenmenge binden 919
Client-seitiges Skript einsetzen 937
DataKeyField 930
DataKeys 931
Detailanzeige in einer Tabelle 929
EditItemIndex 936
EditItemTemplate 934
FooterStyle 934
FooterTemplate 934
HeaderStyle 934
HeaderTemplate 934
LinkButton-Webserver-Steuerelement einsetzen 929
Listeneinträge auf mehrere Spalten verteilen 928
OnCancelCommand 934
OnDeleteCommand 937
OnEditCommand 934
OnItemCreated 937
OnSelectedIndexChanged 930
OnUpdateCommand 934
SelectedIndex 931
SelectedIndexChanged 934
SelectedItemTemplate 932
DataMember-Eigenschaft 917
DataObject-Klasse 729, 730, 733, 734
DataSource-Eigenschaft 571, 572
DataTextField-Eigenschaft 917
Datei Speichern unter (Dialog) 682
Dateien, Liste der zuletzt geöffneten implementieren 604
Dateinamenerweiterung registrieren 877
Dateistruktur von Projekten 50
Dateizugriff 390
.NET-Klassen für 390
asynchroner 401
– Rückrufe 401
– Statusmeldung 404
indexsequenzieller 395
synchroner vs. asynchroner 400
Datenbindung
Web Forms 915
Datenfelder
Geltungsbereich 231
Gültigkeitsbereich 296
Instanz- 228
Instanz- vs. statische 249
Integrität 296
Lebensdauer 233
Lesesperren 296
readonly- 239
Schreibsperren 296
statische 229, 249
– Initialisierung 230
– Lebensdauer 233
struct_, initialisieren 188

Überschreibung von 249
Datenformate für Zwischenablage und Drag&Drop 729
Datenstrukturen 186
Datentransfer
Zwischenablage, Drag&Drop 725
Datentypen
abstrakte 112
allgemein 163
allgemeines über 95
als Klassen 116
Aufzählungstypen als 182
integrierte vs. benutzerdefinierte/komplexe 111, 163
komplexe 182
string 172
Stringrepräsentation 367
struct-Klassen 186
verschachtelte 250
DateTimeFormatInfo-Klasse 369
DCOM 88, 90, 1046, 1138
DCOM-Protokoll 814
Deactivate-Ereignis 448, 449
Deadlock 1007
Beispiel 1018
Debuggen
Webdienst 952
Debuggen beenden (Visual Studio .NET) 73
Debugger
Benutzersteuerelementdesign 640
Grundfunktionen 1115
Überwachungsfenster 729
Debugging 70
Debug-Klasse 1116
Debug-Modus 74, 1115
DEBUG-Symbol 1116
decimal 168
Literal 168
Decimal-Klasse
ToInt32() 990
ToSingle 990
DefaultEvent-Attribut 645
DefaultExt-Eigenschaft 678
DefaultPageSettings-Eigenschaft 687, 689
default-Zweig 145, 146
Deklaration, Variablen 163
Dekrement 195
Delegat 199
als Klasse 199
Aufruf 201, 205
Aufrufliste 204
Einsatzmöglichkeiten 201
Ereignisbehandlung 201
Instanzmethoden, für 201
Multicast- 203
out-, ref-Parameter 205
ref-Parameter 205
DelegateDemo (Codebeispiel) 202
Delegate-Klasse 199
Thread-Rückrufe 1011
Delegaten
DLL-Rückrufe 1091
Delegation als Ersatz für Vererbung unter COM 90
delegierte Konstruktion 282
Delete() 382, 385
Delphi
Extended 168
Fehlersuche 1120
inherited 274
Initialisierung statischer Instanzen 1076
Problem mit mscorlib.tlb 1067
raise 346
Regstrierungs-Assistent 1067
uses-Klauseln 98
Visual Studio .NET, Parallelinstallation 1111
Delta-Eigenschaft 421
DereferenceLinks-Eigenschaft 678
Description-Attribut 645, 661
Deserialisierung 708
Designer
Arbeitsweise 637
außer Tritt geraten, was tun? 641
Hauptmenü anlegen 601
wie kommen die Vorgabewerte für Text- und Name-Eigenschaften zustande 650
Designer, Größenfestlegung 53
DesignerSerializationVisibility-Attribut 647, 649
Designmaße 481
DesignMode-Eigenschaft 638
Desktop-Eigenschaft 486
Destruktor 258, 282, 286
287
asynchrone Destruktion 287
Close() 290
Dispose() 290, 291
Finalize() 288
IDisposable 291
Diagnostics-Namensraum 1000, 1116
Dialog 673
als alleinstehende Anwendung 717
als direkte Instanz der Form-Klasse 440
als schwebendes Steuerelement 673

Ausgestaltung 703
Datei Speichern unter 682
Farben 681
modaler 673, 674, 677
– Übernehmen-Schaltfläche 696
nicht-modaler 698
Öffnen 680
Pfadauswahl (Codebeispiel) 594
Schriftart 681
Seite einrichten 683
zentrieren 693
DialogResult-Aufzählung
Elemente 675
DialogResult-Eigenschaft 564, 692
DialogResults-Aufzählung 674
DictionaryEntry-Klasse 972
Die 713
Directory-Eigenschaft 389
DirectoryInfo-Klasse 389
Directory-Klasse 381
Methoden, Überblick 382
DirectoryName-Eigenschaft 389
DirectorySeparatorChar-Eigenschaft 386
DISCO 943
Diskettenlaufwerk 596
Display (Einheit) 468
DisplayMemberChanged-Ereignis 575
DisplayMember-Eigenschaft 571, 572
Dispose Pattern 859
Dispose() 290, 414, 551, 859
IDisposable-Schnittstelle, und 291
DisposeDemo (Codebeispiel) 292
Disposing-Eigenschaft 549
Divider-Eigenschaft 621
DLL
Aufrufe 1069
Fehlerprüfungen 1076
Fehlersuche in 1120
Funktionsergebnisse 1089
keine Metadaten 1042
Ladezeitpunkte 1074
Marshalling 1078
Metadaten 1081
Pfade 1073
Rückrufe 1090
struct-Parameter 1081
Versionskonflikte bei Update 92
DLL (d 665
DllCheck (Codebeispiel) 1076
DLL-Hölle 92
DllImportAttribute 1070
DLLLoadTime (Codebeispiel) 1074
DllPath (Codebeispiel) 1073
DNS 1130
DNS (Codebeispiel) 820
DNS-Abfrage 820
DNS-Klasse 820
do-Anweisung 150
Dock-Eigenschaft 549, 582
DockStyle-Aufzählung 549, 587
Document (Einheit) 468
Document Object Model 749
Document Type Definition 749
Document-Eigenschaft 687
DocumentName-Eigenschaft 688
DoDragDrop() 551, 736, 742
DoEvents() 516, 518
Dokumentationskommentar 127, 131
Handhabung 132
Kommentarwebseiten erstellen 132
Nachteile 131
Vorteile 131
XML-Tags für 134
Dokumentationskommentare
vs. Kurzbeschreibung für Eigenschaften in Eigenschaften-Fenster 645, 661
dokumentenzentrierter Ansatz 84
Dokumentvorlage 709
DOM 749
Domain Name System 1130
Doppelpufferung 645
Dosierung von Ausnahmen 354
double 167
DoubleBuffer 645
DoubleClick-Ereignis 426, 453
DoubleClickSize-Eigenschaft 419
DoubleClickTime-Eigenschaft 419, 453
DpiX-Eigenschaft 468
DpiY-Eigenschaft 468
Drag&Drop 725, 736
Ablegen des Objekts 741
Beispielcode für FileDrop 613
Datenformate 729
Datentypen, eigene, anwendungsübergreifend transferieren 731
Einladung zum 737
Feedback an den Benutzer 738
frühe Wertbindung 728
Kommunikation Quell-/Zielobjekt 737
Mehrere Datenformate/typen en bloc 733

Operationen, mögliche 736
Referenztypen und eigene Datentypen 730
späte Wertbindung 728
– Vorteile 732
Symbolschaltflächen in Symbolleiste per Drag&Drop einfügen (Codebeispiel) 630
Verschiebeoperation, mißlingen einer 742
DragAction-Aufzählung 741
DragDropEffects 630
DragDropEffects-Aufzählung 736
DragDrop-Ereignis 427, 449, 554, 630, 742, 743
behandeln 742
DragEnter-Ereignis 427, 449, 554, 630, 737, 743
behandeln 737
DragEventArgs-Klasse 737
DragLeave-Ereignis 427, 449, 554, 737
DragOver-Ereignis 427, 450, 554, 737
behandeln 737
Draw() 615
DrawArc() 511
DrawBackground() 612
DrawBezier() 511
DrawBeziers() 511
DrawCloseCurve() 511
DrawCurve() 512
DrawEllipse() 512
DrawIcon() 494, 512
DrawIconUnstretched() 512
DrawImage() 494, 512, 514
Maskierung 479
DrawImageUnscaled() 512, 514
DrawItem-Ereignis 509, 571, 575, 609, 705, 706
DrawItemEventArgs-Klasse 609
falsche Hintergrundfarbe bei Menüs 612
DrawLine() 512
DrawLines() 512
DrawMode-Eigenschaft 571, 572, 705
DrawPath() 474, 512
DrawPie() 512
DrawPolygon() 513
DrawRectangle() 513
DrawRectangles() 513
DrawReversibleFrame() 423
DrawString() 494, 513, 527, 690, 691
Drehknopf-Steuerelement 642
Dreischichtsystem 811
DropDownArrows-Eigenschaft 621
DropDownButton-Eigenschaft 620, 624, 626
DropDown-Eigenschaft 620, 624
DropDownMenu-Eigenschaft 622
DropDown-Schaltfläche 624
DropDownStyle-Eigenschaft 572
DropDownWidth-Eigenschaft 573
Druckausgabe
Seitenaufbau 689
Drucken 687
Standardkoordinatensystem für 689
Drucken (Dialog) 683, 690
Drucker. Strichstärke 495
Druckseite 689
Druckvorgang 689
Abwicklung durch PrintDocument-Objekt 688
Parameter für 688
Druckvorgang (Codebeispiel) 686
DTD 749
dynamische Bibliothek 87
dynamische Bibliotheken 1042
dynamischer Standardkonstruktor 282

E
ebXML 943
ECMAScript 883
Effect-Eigenschaft 736, 738
Eigenschaft
als Array 302
Kurzbeschreibung per Attribut für Eigenschaften-Fenster definieren 645, 661
SystemInformation-Klasse 419
Eigenschaften 258, 295
Bildlauf des Formulars 455
ButtonBase-Klasse 563
Button-Klasse 564
CheckBox-Klasse 565
ComboBox-Klasse 572
Control-Klasse 546
Cursor-Klasse 420
Exception-Klasse 344
Fokus, in Zusammenhang mit 431
Font-Klasse 529
Gebrauch von 297
ImageList-Klasse 614
Indexer, Zugriff auf per 299
KeyEventArgs-Klasse 433
KeyPressEventArgs-Klasse 432
ListBox-Klasse 572
Mauszustand, für 419
MouseEventArgs-Klasse 421
OpenFileDialog-Klasse 678
PointF-Struktur 469
Point-Struktur 469

RadioButton-Klasse 567
RectangleF-Struktur 472
Rectangle-Struktur 472
Serialisierung von Eigenschaften durch den Designer 647
SizeF-Struktur 471
Size-Struktur 471
Stream-Klasse 393
StringCollection-Klasse 373
Syntax 297
SystemColor-Klasse 485
TabControl-Klasse 704
TextBox-Klasse 569
Timer-Klasse 428
ToolBarButton-Klasse 622
ToolBar-Klasse 621
TreeNode-Klasse 592
TreeView-Klasse 590
von Steuerelementen 546
Eigenschaften (Codebeispiel) 297
Eigenschaften (Visual Studio .NET) 52
alle Konfigurationen 76
Eigenschaften-Fenster 637
Hilfetexte für Eigenschaften und Ereignisse 542
Kurzbeschreibung für Eigenschaft per Attribut definieren 645, 661
Serialisierung von Eigenschaften durch den Designer 647
Eigenschaftsdialog 694
Möglichkeiten der Initialisierung 694
Eigenschaftsseiten (Visual Studio .NET) 59
Einfache Verweistypen 172
Einfache Werttypen 165
Einfachvererbung 317
Einfügereihenfolge, Auswirkungen der _ von Steuerelementen 448
Eingabefokus 430
Eingebettete Ressource 740
Einsprungpunkt
Threads 1000
Einzelnes Startprojekt 280
Einzelschritt (Visual Studio .NET) 73
Elapsed-Ereignis 428
ElapsedEventArgs-Klasse 428
Elements-Eigenschaft 464
Elementzugriff auf Array 214
EllipsisCharacter 531
EllipsisPath 531
EllipsisWord 531
else if 145
else-Anweisung 144
Em-Höhe 481
Enabled-Eigenschaft 549, 603, 622
Encoding-Klasse 170, 790, 794, 836, 843
EndCap-Eigenschaft 499
EndContainer() 510
EndEdit() 589, 593
EndInvoke() 1015
Endlosschleifen abbrechen (Visual Studio .NET) 74
EndPrint-Ereignis 688
EndRead() 394, 400, 401, 404
EndsWith() 173, 366
EndUpdate() 574
EndWrite() 394, 400
EnsureCapacity() 374
EnsureVisible() 593
Enter-Ereignis 448, 450, 554
Entity Reference 755
Entwurfsansicht 636, 637
Enum.Parse() 716
enum-Aufzählung 182
auflisten 491
aufzählen 185
Basistyp 184
Bitoperatoren 184
Konstanten exportieren 184
ToString() 184
Typumwandlung 184
Enumeration 182
EnumeratorBeispiel (Codebeispiel) 328
Enum-Klasse
GetNames() 986
EnumWindows (Codebeispiel) 1091
Environment-Klasse
TickCount 855
TickCount-Eigenschaft 853
Equals() 194
Eratosthenes, Sieb des 157
Ereignis 553
Behandlungsmethode einfügen für 417
Behandlungsmethoden vs. OnXxx()-Überschreibung 451
Formular- (Überblick) 448
im Designer automatisch verdrahten 539
Maus- (Überblick) 420, 425
Mechanismus, als 208
sekundäres generieren 453
Signalisierungsschema 208
vs. Multicast-Delegaten 208
Ereignisbehandlung
Abfolge der 515

Steuerelement identifizieren 557
Zehntelsekundenregel 515
Ereignisfolge für ein Formular 448
Ereignisorientierung
Zeichnen auf Anforderung 460
Ereignisse
als Multicast-Delegaten 208
CheckBox-Klasse 566
Control-Klasse 554
Form-Klasse 449
ListBox-Klasse 575
Mausereignisse 426
ToolBar-Klasse 622
TreeView-Klasse 591
ErstellZeit (Codebeispiel) 383
EscapePressed-Eigenschaft 741
Escape-Sequenzen für string/char-Notation 170
event 102, 208
event bubbling 918
Events 1034
Excel 86
Exception-Klasse 344, 349
Eigenschaften 344
Exists() 382, 385
ExitThread() 722
ExitWindowsEx 1093
Expand() 589, 591, 593
ExpandAll() 591, 593
explicit 309
Explizite Typumwandlung 222, 223
Export von Konstanten 248
Extended (Datentyp) 168
Extensible Stylesheet Language Transformations 748
extern 102
externe Hilfe, Visual Studio 819
externe Routinen 1070

F

f 754
Fallunterscheidung
enum-Konstanten 183
switch-Struktur 145
Farbe
Bekannte Systemfarben ermitteln 986
Farben 484
benannte, aufzählen 490
Farben (Codebeispiel) 487
Farben (Dialog) 681
Farbraum 489
Farbschema 485
Farbverlauf 460
komplex 506
linear 505
Fat Client 811
Fehlerbedingung, signalisieren 341
Fehlerbehandlung, siehe Ausnahmebehandlung 339
Fehlercodes 342
Fehlersuche 70, 1115
Ändern von Variablen 72
Befehle 73
Delphi 1120
in DLLs 1120
MSVC 1123
Visual Studio 1123
Fensterstil, für Dialog 693
FieldOffset 1085
Figuren 475
FileInfo-Klasse 389
FileName-Eigenschaft 678
FileNames-Eigenschaft 679
FileNotFoundException 991
FileOk-Ereignis 677, 684
file-Protokoll 818
FileStream-Klasse 793, 848, 855, 862
FileSystemInfo-Klasse 389
FileWebRequest-Klasse 863
FillClosedCurve() 513
FillEllipse() 513
FillPath() 474, 478, 513
FillPie() 513
FillPolygon() 513
FillRectangle() 513
FillRectangles() 513
FillRegion() 513
FillToRight 706
Filter 680
Filter-Eigenschaft 679
FilterIndex-Eigenschaft 679
Finalize() 288
Finalizer 859
finally-Block 347
FindFirst (Codebeispiel) 1086
FindForm() 551
FindString() 574
FindStringExact() 574
Firewall 1134
FirstNode-Eigenschaft 592
Fixed 706
Fixed3D 444, 445
FixedDialog 694

FixedToolWindow 445
183
Flags-Attribut 183, 433
FlatStyle 563
FlatStyle-Eigenschaft 564
float 167
Flush() 394
Focus() 431, 551
Focused-Eigenschaft 431, 549
Font
GetHeight() 645
FontChanged-Ereignis 648
FontDialog-Klasse 676, 685
Font-Eigenschaft
Serialisierung 648
FontFamily.GetFamilies() 480, 528
FontFamily-Eigenschaft 529
FontFamily-Klasse 480, 528
Families 986
Font-Klasse 528
Eigenschaften 529
Konstruktor 986
FontStyle-Aufzählung 482, 529
for-Anweisung 148
foreach-Anweisung 150
Typsicherheit 153
foreach-Schleife 147
ForeColor-Eigenschaft 549
Format() 367, 370
FormBildlauf (Codebeispiel) 457
FormBorderStyle-Aufzählung 444
FormBorderStyle-Eigenschaft 444
Form-Klasse
als Vererbungsbasis 443
direkte Instanziierung 440
Ereignisse 449
Formular
als Objekt einer Formularklasse 442
als Schablone für die weitere Ableitung 709
Ausstattung eines _ 443
Bildlaufleisten des 455
Erbmasse 439
Ereignisfolge bei Aufruf und Abbau 448
Vererbung von 708
Vererbung, schrittweise Anleitung 718
Formular (Überblick) 439
Formularentwurf
Steuerelemente platzieren in 539
Tipps und Tricks 541
Formularentwurf/Quelltext-Analogie 415
Formularschablonen 709
Formularvererbung 708
Codebeispiel 713
Implementierungsstrategien 710
Verkapselung 711
Freispeicherverwaltung 91, 282
anstoßen 288
Destruktor 286
Lebensdauer und 232
FromArgb() 484
FromKnownColor() 486
FromLTRB() 471
FromName() 486
frühe Bindung 89, 1044
FTP-Protokoll 818
Füllbereich 473
Füllfigur 474
FullName-Eigenschaft 389
FullPath-Eigenschaft 590, 592, 597
Problem mit 599
Funktionale Programmierung 765
Funktionen 258
Funktionszeiger 199
Instanzmethoden, für 201

G
GAC 45
gacutil.exe 1064
Ganzzahlliterale 167
Ganzzahltypen 166
Garbage Collector 48
Gateway, Standard- 1131
GC 48
GC.Collect() 288
GC-Klasse
SuppressFinalize() 859
GDI (G 459
GDI+
(Überblick) 459
neue Features gegenüber GDI 459
GDI-Objekte, Abbau von 701
Geerbtes Formular hinzufügen (Befehl) 670
Geerbtes Steuerelement hinzufügen (Befehl) 670
Geltungsbereich 231
Gemeinsam genutzte Assemblies 45
Gemischtsprachliche Wiederverwendung von Code 40
Gerätekoordinatensystem 468
get-Accessor 297
GetAttributes() 385
GetCellAscent() 482

GetCellDescent() 482
GetChildAtPoint() 551
GetChildIndex() 448, 557
GetContainerControl() 552
GetCreationTime() 382, 385
GetCurrentDirectory() 382
GetCurrent-Eigenschaft 153
GetData() 631, 727
GetDataObject() 728, 729
GetDataPresent() 630, 631, 727, 730
GetDirectories() 382, 597
GetDirectoryName() 387
GetDirectoryRoot() 382
GetEmHeight() 482
GetEnumerator() 152, 153, 157, 327, 366
GetExtension() 387
GetFamilies() 480, 528
GetFileName() 387
GetFileNameWithoutExtension() 387
GetFiles() 382
GetFileSystemEntries() 382
GetFormat() 369
GetFormats() 727, 729, 730
  Ergebnis im Überwachungsfenster begutachten 729
GetFullPath() 387
GetGetMethod() 197
GetHeight() 482, 530, 645
GetItemHeight() 575
GetItemRectangle() 575
GetItemText() 575
GetLastAccessTime() 382, 385
GetLastWriteTime() 383, 385
GetLength() 217
GetLineSpacing() 482
GetLogicalDrives() 383, 596
GetLowerBound() 217
GetNextControl() 552
GetNodeAt() 591
GetNodeCount() 591, 594
GetParent() 383
GetPathRoot() 387
GetPixel() 485
GetProperties() 197
GetRegionData() 473
GetRegionScan() 474
GetTabRect()-Eigenschaft 706
GetTempFileName() 387
GetTempPath() 387
GetThumbnailImage() 589
GetTypeCode() 117
GetUpperBound() 217
GetUserName (Codebeispiel) 1079
GetValue() 197, 214, 714, 716
GetValues() 185, 197, 491
Gevierthöhe 481, 482
Gitter, magnetisches, für Splitter 588
GiveFeedback-Ereignis 450, 554, 739
Glättungsfaktor 511
Gleitkommaliterale 167
Gleitkommatypen 167
Global Assembly Cache
  (Übersicht) 45
  Installation in 1064
Global.asax 873
Global.asax.cs 874
Globale Seiteneinstellungen für Drucker 687
Globale Transformation 464
globale Variablen 258
Globaler Assembly Cache 666
  Steuerelementbibliothek aufnehmen in 667
Globales Koordinatensystem 464
Globally Unique Identifier 93
GotFocus-Ereignis 450, 554
goto-Anweisung 147, 148, 156
Grafikausgabe 459
Grafikausgabe puffern 514
Grafikformate, Überstützung durch GDI+ 460
Grafikkontext 459, 462, 509, 689, 690
  gepuffert zeichnen 514
  Zeichnen in Bitmap 515
Grafikpfad 474
  Figuren 475
  Schriftzüge, in 480
  transformieren 474
GraphicPath-Klasse
  als Elementtyp von Region-Klasse 473
Graphics Device Interface 459
Graphics.FromImage() 515
Graphics-Eigenschaft 462, 463, 509
Graphics-Klasse 463, 510
  DrawString() 986
  FromImage() 986
  MeasureString() 986
  Methoden für globale Transformation 467
  Transform() 986
  Zeichenoperationen im Überblick 511
GraphicsPath-Klasse 459, 474
  Linienenden für Pen-Objekte definieren 498
GraphicsUnit-Aufzählung 467, 468, 529
GraphicsUnit-Eigenschaft 468, 689

GrayText-Eigenschaft 486
GroupBox-Klasse 585
GroupBox-Steuerelement 585, 586
GUID 93, 322
Guid-Attributklasse 1062
GUI-Thread 1013
Gültigkeitsbereich für Datenfelder implementieren 296
Gummiband (Codebeispiel) 421

H
Haarlinie 495
Haltepunkte 70, 1117
  Debugger-Konflikt 1123
  Debugger-Konflikte 1120
  Threads 1000
Handled-Eigenschaft 432, 433, 435
Handle-Eigenschaft 549
HasChildren-Eigenschaft 549
HasExtension() 387
HasMorePages-Eigenschaft 690, 691
HatchBrush-Demo (Beispielprojekt) 502
HatchBrush-Klasse 501
HatchStyles-Aufzählung 501
Hauptformular 442
Headers-Auflistung
  Auswerten 994
Heap 114
  Stringverwaltung 362
Height-Eigenschaft 529
Hello World-Programm 58
HelpLink-Eigenschaft 344
Herstellerinformation 44
Hide() 552
Highlight-Eigenschaft 486
HighlightText-Eigenschaft 486
Hilfe
  externe für Visual Studio 819
HorizontalExtent-Eigenschaft 573
HorizontalScrollbar-Eigenschaft 573
HotTrack-Eigenschaft 486, 705
HScrollBar-Klasse 578
  Eigenschaften 579
HScrollBar-Steuerelement 578
HScroll-Eigenschaft 456
HTML 881
  CSS einsetzen 886
  Dynamisches 889
  Formular 891, 892
  Formular aufrufen 893
  Formular mit Client-seitigem Script validieren 894
  Statisches 884
  Steuerelemente im Formular mit Tabelle positionieren 896
  Verstecktes input-Steuerelement einsetzen 897
HTML-Dokument
  Aufbau 885
HtmlForm (Codebeispiel) 891
HTML-Serversteuerelement 899, 900
  Auf Wert zugreifen 902
  Ausführung auf dem Server festlegen 901
  Gruppe von Optionsschaltflächen bilden 905
  id 905
  id-Eigenschaft festlegen 901
  Label als Überschrift formatieren 905
  Label durch Stylesheet formatieren 908
  Label-Text durch InnerHtml-Elgenschaft setzen 909
  Liste für Kombinationslistenfeld eingeben 905
  Mit Client-seitigem Skript kombinieren 902
  name 905
  Stil für Label festlegen 901
  Text für Label zuweisen 905
  Text in Label eingeben 901
  Wert aus Kombinationslistenfeld auslesen 907
  Wert aus Optionsschaltfläche auslesen 907
HtmlStatisch (Codebeispiel) 884
HTTP 882, 1125
  -Anwendung 871
  -Handler 873, 874, 875
    – verfügbar machen 875
  -Modul 873, 874, 875
    – verfügbar machen 876
  -Pipeline 871, 872
    – Konfiguration 874
  -Protokoll 814, 818
  Redirect 893
HttpApplication-Klasse 874
HttpContext-Klasse 873, 874
HTTPS 1135, 1138
HttpWebRequest-Klasse 863
Hyperlink 881
Hypertext 881
HyperText Markup Language 748

I
I/O-Operationen
  stream-orientierte 390
IANA 1130
IAsyncResult 959
  Auswerten 959
  IsCompleted 959

-Schnittstelle 855
IAsyncResult-Klasse 1015
IAsyncResult-Schnittstelle 401, 402, 404
ICANN 1130
ICloneable-Schnittstelle 336
ICMP 1137
ICollection-Schnittstelle 155, 376
Icon-Eigenschaft 624
Icon-Klasse
Instanz als Ressource einlesen 740
ICustomFormatter-Schnittstelle 370
IDataObject-Schnittstelle 630, 729, 730, 734
Methoden 727
Identische Abbildung 464
IDictionary-Schnittstelle 155
IDispatch 1060
IDispatch.Invoke() 1044
IDispatch-Schnittstelle 89, 322
IDisposable 291
IDisposable-Schnittstelle 858, 859
IE
-Hosting 815
Kontextmenü unterdrücken 819
-Steuerelement 816
– Navigate()-Methode 818
IEHosting (Codebeispiel) 816
IEnumerable-Schnittstelle 154, 326, 376
IEnumerator-Schnittstelle 154, 157, 326
if-Anweisung 144
IFormatProvider-Schnittstelle 369
Ignore 675
IIS
Benutzerkonto VS Developers 62
Konfiguration 877
IL 33, 35, 123
ILDASM 1058
Speicherort 1058
IList-Schnittstelle 155, 376
Datenquelle, als 571
Image.FromFile() 445
ImageAlign-Eigenschaft 564
ImageAttributes-Klasse 504
Image-Auflistungs-Editor, Probleme mit 617
ImageCollection-Klasse 375
Image-Eigenschaft 563
ImageIndex-Eigenschaft 564, 590, 592, 620, 622
ImageList-Eigenschaft 564, 590, 621, 705
ImageList-Klasse 613
Eigenschaften 614
ImageList-Komponente 613, 619
Kacheln einfügen 615
ImageList-Steuerelement 594, 609
Images-Eigenschaft 613, 614
ImageSize-Eigenschaft 613, 614, 619, 621
Image-Webserver-Steuerelement 928
imperative Programmierung vs. objektorientierte 243
Implementierung
polymorph vs. nicht-polymorph 317
polymorphe 275, 319
– Streams als Beipiel für 391
Implementierungsvererbung 198, 316, 318
Einfach- vs. Mehrfachvererbung 316
partielle 319
struct-Klassen 190
implicit 309
Implizite Typumwandlung 222, 223
in 1080
InactiveBorder-Eigenschaft 486
InactiveCaption-Eigenschaft 486
InactiveCaptionText-Eigenschaft 486
Inch (Einheit) 468
Index-Eigenschaft 592
Indexer 299
Codebeispiel für 157
Schnittstelle, in 304
Vererbung 303
Indexer (Codebeispiel) 301
Indexer1 (Codebeispiel) 302
Indexfehler, behandeln 339
IndexFromPoint() 575
IndexOf() 366, 376, 615, 623, 624
IndexOfAny() 366
IndexOutOfRangeException-Klasse 349, 354
Indexsequenzieller Dateizugriff 395
indizierbare Eigenschaften 302
Infixnotation 305
Inflate() 471, 472
Info-Eigenschaft 486
InfoText-Eigenschaft 486
Initialisierung
Arrays 214
lokale Variablen 165
statischer Elemente 253
von Werttypen 115
– struct 188
Initialisierungscode, eigener 415
Initialisierungsvereinbarung 230
InitializeComponent() 414, 415, 416
Serialisierung von Eigenschaften 647
Inkrement 195

Inline-Hilfe 409, 542
InnerException-Eigenschaft 344
Insert() 366, 374, 376
Installation 1107
.NET-Laufzeitumgebung 1112
Visual Studio .NET 1107
von .NET 1107
Instanz 243
Begriff 243
-felder 228
– Geltungsbereich 231
– Initialisierung 230
– Lebensdauer 233
– vs. statische Datenfelder 249
-konstruktor 282
-kontext 244
-methode 258, 281
statische 253, 277
unbenannte 285
Instanziierung 251
anonyme 252
Arrays 213
int 166
Int64 166
IntegralHeight-Eigenschaft 573
Integrierte Datentypen 111, 163
Integrität für Datenfelder implementieren 296
InteractiveThread (Codebeispiel) 1020
InteractiveThread2 (Codebeispiel) 1030
interface 318, 323, 331
vs. abstract 322
Interface Description Language 1043
Intermediate Language 33, 35, 123
internal 101, 102
Geltungsbereich, Auswirkung auf 231
internal protected 102
internal public
Geltungsbereich, Auswirkung auf 231
Internet 882
Protokoll 1125
Internet Assigned Numbers Authority 1130
Internet Control Messaging Protocol 1137
Internet Corporation for Assigned Names and Numbers 1130
Internet Protocol 1130
Internet Service Provider 1130
Internet-Programmierung 814
InterpolationMode-Eigenschaft 510
Interrupts 1005
Debugger-Konflikte 1120
Intersect() 471, 473
IntersectsWidth() 473
Interval-Eigenschaft 428
Invalidate() 462, 552
partielles Rendering 518
vs Refresh() 462
WM_ERASEBKGND 518
Invalidated-Ereignis 450, 554
InvalidOperationException-Klasse 349, 354
InvalidPathChars-Eigenschaft 386
Invoke() 1012
IDispatch 1044
InvokeMember() 1059
InvokeRequired() 1013
IOException-Klasse 349
IP 1130
-Adresse 1130
– lokale 1131
-Header 1132
Subnetzmaske 1131
IPAddress-Klasse 821
IPHostEntry-Klasse 821
IPv6 1131
IPX/SPX 1137
is 195
IsDefault 564
IsEditing-Eigenschaft 589, 592
IsExpanded-Eigenschaft 589
IsExpand-Eigenschaft 592
IsFixedSize-Eigenschaft 376
IsHandleCreated-Eigenschaft 549
ISO-8859-1 754
ISP 1130
IsPathRooted() 387
IsPostBack 903
IsReadOnly-Eigenschaft 376
IsSelected-Eigenschaft 589, 592
IsSynchronized-Eigenschaft 376
IsVisible-Eigenschaft 592
Italic 482
Italic-Eigenschaft 529
ItemCheck-Ereignis
Behandlung 629
ItemDrag-Ereignis 592
ItemHeight-Eigenschaft 571, 573, 611
Items-Eigenschaft 573
ItemSize-Eigenschaft 705
Items-Klasse
AddRange() 990
ItemWidth-Eigenschaft 611

IUnknown-Schnittstelle 89, 322
IWebRequestCreate-Schnittstelle 864

J
J++ 31
J++-Affäre 81
Java 30, 81
JavaScript 883
JIT-Compiler 46, 123
JIT-Compiler, Ladezeitpunkte von DLLs 1076
Join() 1002
JScript .NET 32
just-in-time-Kompilierung 46

K
Kardinaler Spline 460, 477, 511, 513
KeyChar-Eigenschaft 432
KeyCode-Eigenschaft 433
KeyData-Eigenschaft 433
KeyDown-Ereignis 432
  behandeln 660
KeyEventArgs-Klasse 432, 433
  Eigenschaften 433
KeyPress-Ereignis 432
KeyPressEventArgs-Klasse 432
  Eigenschaften 432
KeyPreview-Eigenschaft 434, 629
Keys-Aufzählung 184, 433
KeyState-Eigenschaft 738, 741
  Bitmasken 738
KeyStates-Aufzählung 738
KeyUp-Ereignis 432
KeyValue-Eigenschaft 433
Klasse
  abstrakte 331
    – in C# 331
    – in C++ 331
    – vs. Schnittstelle 322, 335
  Datentypen definieren, in 250
  Konstanten 247
  Sichtbarkeit von _elementen 247
  statische Elemente 252
    – Initialisierung 253
  statischer vs. dynamischer Bereich 244
  statisches Datenfeld vs. Instanzfeld 249
  Vererbungsmodifizierer 246
  vs. Objekt 242, 243
  vs. Schnittstelle 322
  Zugriffsmodifizierer 246
Klassen
  -begriff 241
  -bibliothek
    – MFC als 87
  -elemente
    – Zugriffsmodifizierer 247
  -feld 229
  -hierarchie
    – Ausnahmeklassen 349
  -kontext 244
  -konzept 244
Klassenansicht 52
Klassenbibliothek
  eigene 731
Knob (Codebeispiel) 642
Knob-Steuerelement
  Spezifikation 642
KnownColor-Aufzählung 486, 487
kombinierte Zuweisung 195
Kommandozeilenparameter 59
  Windows-Anwendungen 60
Kommandozeilensitzung 667
Kommaoperator 125
Kommentar 127
  Aufgabenliste 128
  Block 128
  Dokumentations- 131
  TODO- 128
  Token- 128
  -webseiten erstellen 132
  Zeilen- 128
Kompilierungseinheit 43
Komplexe
  Datentypen 111, 163, 182
  Verweistypen 198
  Werttypen 182
Komplexe Farbverläufe 506
Komponente 414
  Eigenschaften-Fenster 542
  ImageList 613
    – Kacheln einfügen 615
  NotifyIcon 722
  ToolTip 492
Komponentenarchitektur 84
  Objektmodell für 86
Konfiguration, aktive 75
Konfigurationen (Visual Studio .NET) 74
Konfigurations-Manager 74, 1115
Konsolenanwendungen 56
Konsolenfenster offenhalten 58

Konstante 236
const- 247
enum- 183
Export von 238, 248
readonly- 239, 248
Regeln für die Vereinbarung 237
Konstanten
nicht für COM 1061
Konstruktor 251, 281
aufrufen 285
– überladenen Konstruktor 285
der Basisklasse aufrufen 284
Regeln für die Konzeption von 286
statischer 249, 282, 284
statischer vs. Main() 279
this-Delegation 282
Konstruktore 258
kontextbezogene Typverwaltung 91
Kontextmenü 607
für NotifyIcon 722
von Symbolschaltfläche 620
Kontextmenü im IE unterdrücken 819
Kontextsensitivität 607
Kontrollkästchen 565
Kontrollstrukturen 143
Konventionen
Benennung 104
– ungarische Notation 109
eigene Benennungssysteme 109
Schreibweise für Bezeichner 110
Strukturierung von Bezeichnern 111
Koordinatensysteme (Überblick) 463
Koordinatensystemtransformation 460
Grundoperationen 464
Kulturinformation 44
Kurven 476

L
LabelEdit-Eigenschaft 590
Label-Element
Als Überschrift formatieren 914
Label-Klasse 568
Label-Steuerelement 568
LargeChange-Eigenschaft 579, 583
LastAccessTime-Eigenschaft 389
LastIndexOf() 366
LastIndexOfAny() 366
LastNode-Eigenschaft 592
LastWriteTime-Eigenschaft 389
Laufwerk 596
Laufzeitansicht 636
Laufzeitinstanz, der Klasse vs. Objekt 244
Layout-Ereignis 450, 554
LayoutKind.Sequential 1082
Leave-Ereignis 448, 450, 554
behandeln 660
Lebensdauer 232
Datenfelder 233
Variablen 233
Legacy Code 1069
Leider 754
Length-Eigenschaft 373, 393
Lesesperren für Datenfelder implementieren 296
LF 174
LineAlignment-Eigenschaft 480, 530
Lineare Abbildung 464
Lineare Farbverläufe 505
LinearGradientBrush-Klasse 490, 505
als Unterlage für Pen-Klasse 496
LineCap-Aufzählung 498
LineJoin-Eigenschaft 497
Lines-Eigenschaft 569
Linienenden 498
selbst definieren 498
Linienstile 497
selbst definieren 498
Linienstöße 497
LinkButton-Webserver-Steuerelement 930, 934
Client-seitiges Skript einsetzen 937
ListBoxDemo (Codebeispiel) 575
ListBox-Klasse 571
Eigenschaften 572
Ereignisse 575
Methoden 574
ListBox-Steuerelement 571
listengebundenes Webserver-Steuerelement 917
Auslösbare Ereignisse 918
Datenzugriff 918
Event bubbling 918
Vorlage 918
ListItemType 938
Literal 236
bool 169
char 170
decimal- 168
Ganzzahl- 167
Gleitkomma- 167
String- 174
literale Konstante 236
vs. Wertkonstante 236

Load-Ereignis 450
vs Konstruktor 716
LoadLibrary() 1077
localhostbin 62
lock() 1005, 1033
Deadlock 1007
Werttypen 1006
zu enge Klammerung 1008
lock()-Methode 857
Locked-Eigenschaft 549
Logische Operatoren 193
lokale Variablen 228
Initialisierung 165
Lebensdauer 233
long 166
LostFocus-Ereignis 555

M
MAC-Adresse 1132
Machine.config 874, 875, 876, 877
Mail (Codebeispiel) 828
MailMessage-Klasse 830
Main() 278
als Einsprungpunkt 279
Programmstart, ohne Formular 722
MainMenu-Klasse 601
MainMenu-Steuerelement 601
Mandelbrot (Codebeispiel) 516
Manifest 44
Anzeige mit ILDASM 1058
GUIDs 1062
Margins-Eigenschaft 687
MarshalAs 1078
Marshalling 46, 1078
Funktionsergebnisse 1089
Maske (IP) 1131
Maskieren von Ausgabebereichen 479
Match-Klasse 838, 840, 843
Matrix-Klasse 464
Shear() 986
MatrixOrder-Aufzählung 465
Maus 418
Mausereignisse
primär 420
sekundär 425
Mausereignisse, Überblick 426
MaxCapacity-Eigenschaft 373
MaxDropDownItems-Eigenschaft 573
Maximum-Eigenschaft 579
MaxLength-Eigenschaft 569
MDI-Anwendung 447
MdiChildActivated-Ereignis 450
MeasureItem-Ereignis 571, 575, 609
behandeln 611
MeasureString() 531, 532, 611, 691
Ärger mit 535
mehrdimensionale Arrays 217, 218
Mehrere Startprojekte 280
Mehrfachvererbung 316, 318
Member 246
MemoryStream-Klasse
Codebeispiel für 398
Write() 855
MemTextBox (Codebeispiel) 658
Menü 600
Arten von _ 600
Behandlungsmethoden der Befehl als Einsprungpunkte der Anwendung 603
Hauptmenü 601
Kontextmenü 607
Rolle von Checked und Enabled 603
Trennlinie 604
Zoom 603
zur Laufzeit erweitern 604
MenüDemo (Codebeispiel) 602
MenüDemoOwnerDraw (Codebeispiel) 609
Menu-Eigenschaft 486
MenuItem-Klasse 602, 604
Objekte in mehrere Menüs eingliedern 607
MenuItems-Eigenschaft 604
Menültem-Objekte in mehreren Menüs verwenden 607
Menüleiste 600
MenuText-Eigenschaft 486
MessageBeep() importieren 857
MessageBox.Show() 674
in Threads 1000
MessageBoxIcons-Klasse 675
MessageDialogResults-Aufzählung 674
Elemente 674
Message-Eigenschaft 344
Metadaten 44, 1042
.lib-Dateien 1043
.ocx vs. COM-DLLs 1053
COM 1043
DLL-Aufrufe 1081
GUIDs 1062
Untersuchen mit ILDASM 1058
Methode 257
Allgemeines 258
Aufruf überladener 269
Aufrufmechanismen 261

Begriff 243
Destruktor 286
Eigenschaften 295
Konstruktoren 281
Objekt- 258, 281
Parameteranzahl, variable 266
Parameterübergabe 260
ref- und out-Parameter 261
Rückgabetyp 259
Signatur 260
statische 258, 260, 277
statische Variablen 229
Syntax 259
überladen 268
Überschreiben 272
Vererbungsmodifizierer 259
virtuelle 275, 320
– COM 1045
Methoden
Bildlauf des Formulars 455
ComboBox-Klasse 574
Control-Klasse 551
CursorKlasse 420
Designmaße für 482
Directory-Klasse 382
Fokus, in Zusammenhang mit 431
Graphics-Klasse, für globale Transformation 467
Graphics-Klasse, Zeichenoperationen 511
IDataObject-Schnittstelle 727
ImageList-Klasse 615
ListBox-Klasse 574
OpenFileDialog-Klasse 679
PointF-Struktur 469
Point-Struktur 469
RectangleF-Struktur 471
Rectangle-Struktur 471
SizeF-Struktur 471
Size-Struktur 471
StringCollection-Klasse 374
System.String-Klasse 363, 365
TabControl-Klasse 706
TextBox-Klasse 570
TreeNode-Klasse 593
TreeView-Klasse 591
von Steuerelementen 551
MFC (Klassenbibliothek) 87
MFC-Klassenbibliothek 83
Microsoft Foundation Classes Library 87
Microsoft Transaction Server 1046
Millimeter (Einheit) 468
Minimum-Eigenschaft 579
Mitglieder 246
Modaler Dialog 673, 674, 677
eigener 692
Übernehmen-Schaltfläche 696
Modifiers-Eigenschaft 433
Modifizierer 101, 230
für Klasse 246
für Methoden 259
für Schnittstelle 323
Modulo-Operator 193
Modusloser Dialog 673, 698
Monitor-Klasse 1039
MouseButtons-Aufzählung 421
MouseButtons-Eigenschaft 419
MouseButtonsSwapped-Eigenschaft 419
MouseDown-Ereignis 420, 425
MouseEnter-Ereignis 425, 426
MouseEventArgs-Klasse 421
Eigenschaften 421
MouseHover-Ereignis 425, 426
MouseLeave-Ereignis 426
MouseMove-Ereignis 420
MousePosition-Eigenschaft 424
MousePresent-Eigenschaft 419
MouseUp-Ereignis 420, 425, 453
MouseWheel-Ereignis 420
MouseWheelPresent-Eigenschaft 419
MouseWheelScrollLines-Eigenschaft 419
Move() 383, 385
Move-Ereignis 445
MoveNext() 153
Move-Operation (Drag&Drop) 736
abschließen 742
MSACALImport (Codebeispiel) 1055
ms-help-Protokoll 819
Muench-Methode 775
Multiarrays 217, 218
Wertlisteninitialisierung 218
Multicast-Delegat 203
Aufruf 205
Ereignis 208
Ereignisse 208
MultiCastDelegateDemo (Codebeispiel) 206
MulticastDelegate-Klasse 199
MultiColumn-Eigenschaft 571, 573
Multiline-Eigenschaft 569, 705
Multiple Dokument Interface 447
MultiplyTransform() 467
Multiselect-Eigenschaft 679

Multithreading 999
asynchrone Methoden 1014
Deadlock 1007
Entkoppelung 1019
Interaktion 1016
– Codebeispiel 1020
lock() 1005
Rückrufe 1010
– Delegaten 1011
– nachträglich synchronisiert 1012
Synchronisation 1004
Synchronisationsobjekte 1030
Umschaltung 1005
Wettrennen 1008
Mutex 1006
MyDerivedDialog (Codebeispiel) 713
MyDialog (Codebeispiel) 714

N
Nachrichtenschleife 413
Name-Eigenschaft 389, 546
wie kommt der Vorgabewert zustande 650
Namensauflösung
bei Überladung 269
verschachtelte Namensräume 99
Namensraum 96
Alias für 99
Bezeichner qualifizieren 97
bezeichnerloser 100
Diagnostics 999, 1116
für Typbibliotheken 1048
Regeln für 100
Runtime.InteropServices 1059
System.Text 1091
Threading 999
tlbimp.exe 1048
umbenennen 99
Verschachtelung von 98
Namensräume 43
NameValueCollection-Klasse 847, 853, 860
Net_for_COMDemo (Codebeispiel) 1061
NETComTest.vbs 1064
netstat.exe 1138
Network News Transport Protocol 1137
NetworkStream-Klasse 825, 835, 840
Netzwerkmonitor 849, 1126
Neues Element hinzufügen (Dialog) 634
new (Modifizierer) 103, 230, 259, 273
Konstantenvereinbarung, mit 237
new (Operator) 251, 281
NewValue-Eigenschaft 583
NeXT 84
NextNode-Eigenschaft 592
NextVisibleNode-Eigenschaft 592
Nicht gesicherter Code 1097
Nicht-modaler Dialog 698
Wertübergabe/-nahme 694
Wertübernahme 699
NNTP 1137
No 675
NodeFont-Eigenschaft 592
Nodes-Eigenschaft 590, 593
None 675
Norton Antivirus 1109
NotifyIcon-Komponente 722
Kontextmenü 722
n-Schichtsystem 812
NullReferenceException-Klasse 349, 354
NumberFormatInfo-Klasse 369
NumericUpDown-Klasse 664
NumericUpDown-Steuerelement 727

O
object-Klasse 172
allgemein 178
Boxing/Unboxing 226
Objekt
-begriff 241, 242
– von C++ 84
-feld 228
Konstruktion eines 251
-methoden 258, 281
-modell, sprachunabhängiges vs. sprachabhängiges 86
-variable 243
vs. Klasse 242, 243
Objekt/Verb-Reihenfolge 111
Objektmodell
CTS vs. COM 36
Struktur von .NET- 37
Objektmodell von .NET 33
objektorientierte Programmierung
vs. imperative 243
Objektvariable 243
Office-Suite 84
Öffnen (Dialog) 680
Offset() 473
OK 674, 675
OKCancel 674
OLE 1.0 84
OLE 2.0 84, 88

OLE-Automatisierung 86, 89
OLE-Technologie 83
OnKeyDown()
überschreiben, Probleme bei Befehlstasten 654
OnPaint() 461
Überschreiben (Codebeispiel) 651
Überschreibung vs. Paint-Behandlung 452, 461
OnPaintBackground() 518
OnXxx()-Methoden 452
Open() 385
Close() 290
OpenFile() 679
OpenFileDialog-Klasse 676, 677, 684
Eigenschaften 678
Filter 680
Methoden 679
OpenFileDialog-Steuerelement 830
OpenRead() 385
OpenText() 385
OpenWrite() 385
Operand 304
operator _() 315
operator_+() 315
Operatoren 258, 304
als spezialisierte Methoden 304
arithmetische 193
Auflösung von, Regeln für die implizite Typumwandlung 310
logische 193
Position 305
Priorität 305
Typumwandlung für 309
Überblick über 191
überladen 306
– Einschränkungen 306
Wertigkeit 305
OperatorExample (Codebeispiel) 311
optionale Parameter 271
Orientation-Eigenschaft 579
OSI-Modell 1136
out 261, 1080
out-Parameter
Delegaten, und 205
OutputDebugString 1116
overload 258, 272
override 103, 258, 259, 272, 273, 276, 277, 331
OwnerDraw-Eigenschaft 609
OwnerDrawFixed 571, 705
OwnerDrawVariable 571
Owner-Eigenschaft 699, 700

P
Padding-Eigenschaft 705
PadLeft() 366
PadRight() 366
Page-Klasse 873
Page-Objekt
IsPostback 903
Load 903
PageScale-Eigenschaft 467
PageSettings-Eigenschaft 687
PageSettings-Klasse 689
PageSetupDialog-Klasse 676, 680, 683, 687
Probleme mit 687
Workaround für 687
PageUnit-Eigenschaft 467, 468
Strichstärke, Auswirkung auf 495
Paint-Ereignis 451, 461, 509, 555
Behandlung vs. OnPaint()-Überschreibung 452, 461
PaintEventArgs-Klasse 461, 462, 509
Paket
beschädigtes 1129
PanelClick-Ereignis 624
PanelDemo (Codebeispiel) 587
Panel-Klasse 585
Instanzen der _ in Statusleiste 624
Panels-Auflistung 624
Panels-Eigenschaft 624
Panel-Steuerelement 582, 585, 586
Parameter
Array als 214
-arten 261
-liste, einer Methode 259
optionale 271
-übergabe, an Methoden 260
variable Anzahl von _ 266
params 266, 268
param-Tag 138
Parent-Eigenschaft 389, 549, 593
Parse() 716
PartialPush-Eigenschaft 622
Partielle Implementierungsvererbung 319
PasswordChar-Eigenschaft 569
Paste() 570, 725
PathGradientBrush-Klasse 490, 506
PathSeparator-Eigenschaft 386, 590
Pausentaste (Visual Studio .NET) 74
PenAlignment-Aufzählung 497
Pen-Klasse 494
Brush-Objekte als Grundlage 496
Konstruktor 496

Linienenden 498
Linienstile 497
Linienstöße 497
Strichstärke 495
Systemfarben 495
Pens-Klasse 495
PenType-Aufzählung 497
PenType-Eigenschaft 497
PerformLayout() 552
Pfadauswahl per TreeView (Codebeispiel) 594
Ping 1137
Pinsel 494
HatchBrush 501
LinearGradientBrush 505
PathGradientBrush 506
SolidBrush 501
TextureBrush 503
PInvoke 1069
Pixel (Einheit) 468
PixelOffsetMode-Eigenschaft 510
Platform Invokation Services 1069
Plattformunabhängigkeit 82
Pluggable Protocols 863
PlugProt (Codebeispiel) 864
Point (Einheit) 468
Point3D-Klasse 311
PointF-Struktur 468, 469
Eigenschaften 469
Methoden 469
Point-Struktur 468, 469
Eigenschaften 469
Methoden 469
PointToScreen() 424, 552
polymorphe
Implementierung 319
– Streams als Beipiel für 391
Initialisierung von Arrays 215
Polymorphie 179, 275, 316, 319
POP3 833, 1137
Pop3 (Codebeispiel) 834
Port 1129, 1137
Portabilität von .NET 32
Portnummer
Aufteilung in Bereiche 1137
Position-Eigenschaft 393, 398
Post Office Protocol 1137
Postfixnotation 305
PostMessage() 1019
Präfixnotation 305
PreferredHeight-Eigenschaft 570
Prepend (MatrixOrder) 465
PrevNode-Eigenschaft 593
PrevVisibleNode-Eigenschaft 593
Primäre Mausereignisse 420
PrimNumbers (Codebeispiel) 157
Primzahlalgorithmen 157
Print() 688, 689, 690
PrintDialog-Klasse 676, 680, 683, 687, 688
PrintDocument-Klasse 680, 687, 688
Abwicklung des Druckvorgangs 688
PrinterSettings-Eigenschaft 688
PrintEventArgs-Klasse 688
PrintPage-Ereignis 688, 689, 690
Seitenaufbau 689
PrintPageEventArgs-Klasse 690
Grafikkontext 690
Priorität
Operator 305
private 101, 103
Geltungsbereich, Auswirkung auf 231
Private Assemblies 44
ProcessCmdKey() 431, 654
ProcessKeyEventArgs() 431, 654
Produktinformation 44
Programm, verteiltes 809
Programmgerüst 50
Programmstart, ohne Formular 722
ProgressBar
Eigenschaften 579
ProgressBar-Klasse 578
ProgressBar-Steuerelement 578
Projekt
Dateistruktur 50
Eigenschaftsseiten 59
startfähiges 278
Verweise
– Webverweis hinzufügen 67
Projekt, neues anlegen 412
Projekt, Verweis hinzufügen 1047
Projekteigenschaften
Debug-Modus 1115
Projektmappen-Explorer 52
PropertyInfo-Klasse 197
protected 101, 103
Geltungsbereich, Auswirkung auf 231
Protokoll 1125
ARP 1133
HTTP 1125
HTTPS 1138

ICMP 1137
IP 1130
IPX/SPX 1137
NNTP 1137
POP3 1137
TCP 1128
UDP 1137
Protokollstapel 1136
ProxyTrace benutzen 953
Prozedurschritt (Visual Studio .NET) 73
public 101, 103
Schreibweise für Bezeichner 110
Pufferung 514
Pulse() 1039
Punktgröße 480
Punkt-zu-Punkt-Farbverlauf 506
Pushed-Eigenschaft 623, 626

Q
qualifizieren 97
qualifizierter Zugriff auf Datenfelder 250
Quellcode, C#- 43
Quellobjekt 725
Quelltext/Formularentwurf-Analogie 415
QueryContinueDrag-Ereignis 451, 555, 741
QueryPageSettings-Ereignis 688, 689, 690
QueryPageSettingsEventArgs-Klasse 689

R
RadioButton-Klasse 566
Eigenschaften 567
RadioButton-Steuerelement 566
RadioGroup (Codebeispiel) 568, 633
Rahmenstil 694
Rank-Eigenschaft 217
RCW 1047
Anlegen 1047
ohne 1059
RCWPhotoshopImport (Codebeispiel) 1047
Read() 394
ReadByte() 394
readonly 103, 230, 239, 248
-Vereinbarung 296
ReadOnlyChecked-Eigenschaft 679
Readonly-Eigenschaft 570
RectangleF-Struktur 468, 471
Eigenschaften 472
Methoden 471
Rectangle-Struktur 468, 471
Eigenschaften 472
Methoden 471
RectangleToClient() 552
RectangleToScreen() 552
Reentranz 153, 157
ref 261, 1080
vs. In, Out 1088
Referenzparameter
DLL-Aufrufe 1079
Marshalling 1088
Referenzzählungsmechanismus 85
Reflexion
Eigenschaften aufzählen 197
enum-Typen aufzählen 185
ref-Parameter
Delegaten, und 205
Refresh() 462, 553
vs Invalidate() 462
REG_DWORD 714, 716
REG_SZ 714, 716
RegAsm.exe 1061
GUID-Erzeugung 1062
vs. tlbexp.exe 1066
Regex-Klasse 838, 840, 841
RegionData-Klasse 473
Region-Eigenschaft 479, 549
Regionen 460
Region-Klasse 473
(De-)serialisierung 473
registered ports 1137
Registerkarte 704, 707
in abgeleiterer Formularklasse hinzufügen 720
zur Laufzeit ausstatten 708
Registerkarte hinzufügen (Befehl) 707
Registrierung 322
.NET-Klassen für COM 1062
– GUIDs 1062
Änderungen beobachten 723
COM, Rolle für 88
Rolle bei COM 1044
schreiben in 714
Registrierungseditor 723
RegistryKey-Klasse 714
Registry-Klasse 714
regsrv32.exe 1044
Regular 482
Regulärer Ausdruck 844
Rekursion 157
Release-Modus 74
Remoting 1138
Remove() 366, 374, 376, 594, 615

RemoveAt() 376, 615
Rendering 518
Rendering, aufwändiges 514
RenderingOrigin-Eigenschaft 501, 510
Repeater-Webserver-Steuerelement 917, 924
AlternatingItemTemplate 924
An Datenmenge binden 919
FooterTemplate 924
HeaderTemplate 924
ItemTemplate 924
SeparatorTemplate 924, 927
Replace() 367, 374
Request
Form 903
-Objekt 897
QueryString 898, 909, 914
Request-Objekt
QueryString 897
Reset() 679
ResetTransform() 467
Resize-Ereignis 445, 451, 552, 555
Probleme mit 525
Response
-Objekt 897
– Write() 897
Redirect() 903, 907, 912
Write() 898
Response.Redirect 914
res-Protokoll 818
Ressourcedateien 740
Ressourcen 43
.resx-Datei 51
Bitmap-, Icon- und Cursor-Instanzen einlesen von 740
Probleme beim Einlesen von 99
RestoreDirectory-Eigenschaft 679
Restwertdivision 193
ResumeLayout() 417, 554
Retry 675
RetryCancel 674
return-Anweisung 156
RichTextBox
Zwischenablage 725
RichTextBox-Klasse, Zwischenablage 725
Richtungstasten, Interpretation der 431
Right-Eigenschaft 549
rollenbasierte Sicherheit 34
Rotate() 465
RotateFlip() 616
RotateTransform() 465, 467, 504
Rotation 464
Round() 469
Router 1129
RowCount-Eigenschaft 705
Rückgabetyp, einer Methode 259
Rückrufe 401
Delegaten 1011
nachträglich synchronisiert 1012
Rückrufmethode 851, 855
Run() 415
formularloser Programmstart 722
Runtime Callable Wrapper 1047
Runtime.InteropServices-Namensraum 1059

S
Sandbox 31
Sandboxkonzept 81
SaveFileDialog-Klasse 676, 682, 686
SAX 749
sbyte 166
Scale() 465, 553
ScaleTransform() 465, 467, 504
Schaltfläche
Aktion zuordnen 53
Einfügen 52
Größe 53
Schaltfläche Übernehmen 681
Schichtenaufteilung 809
Schichtung 82
Schleife 147
do 150
for 148
foreach 150
Kriterium für 148
while 149
Schließen-Befehl 448
Schlüsselwörter
als Bezeichner 98
für C# reservierte 106
für VB.NET reservierte 106
Schnellüberwachung (Visual Studio .NET) 73
Schnittstelle
(Übersicht) 335
Ableitung von 317
Ableitung von Schnittstelle 324
abstrakte Klasse als Ersatz für 336
als Klasse 323
als Konzept 322
als Vertrag 323
C#- 322
COM- 322

Designrichtlinien 336
ICloneable 336
ICollection 155, 376
ICustomFormatter 370
IDictionary 155
IEnumerable 154, 326, 376
IEnumerator 154, 157, 326
IFormatProvider 369
IList 155, 376
Implementierung durch struct-Klasse 190
in .NET-Klassenbibliothek 326
Indexer deklarieren 304
Modifizierer für 323
Vererbung als 709
vs. abstrakte Klasse 322, 335
vs. Klasse 322
Schnittstellen (Codebeispiel) 324
Schnittstellenvererbung 242, 316, 318, 336
Schraffuren 501
Schraffurpinsel 501
Schreibschutz für Datenfelder 239, 296
Schrift 480
Designmaße 481
Schrift (Dialog) 685
Schriftart 528
Vorhandene ermitteln 986
Schriftart (Dialog) 681
Schriftgröße 482, 529
Schriftschnitt 529, 685
Schriftzüge 480
ScreenScraper (Codebeispiel) 841
Script Blocking 1109
ScrollableControl-Klasse 455
ScrollBar-Eigenschaft 486
ScrollBars-Eigenschaft 570
ScrollControlIntoView() 456
Scroll-Ereignis 580, 645
ScrollEventType-Aufzählung
Elemente 580
ScrollToCaret() 570
sealed 104, 117, 246, 276, 334
Secure Sockets Layer 1135, 1138
Seek() 394, 398
Seite einrichten 683
Seite einrichten (Dialog) 683
Seiteneinstellung
für Druckvorgang 687
Seitenkoordinatensystem 464, 467
Seitentransformation 467
Seitenumbruch 690
Sekundäre Ereignisse 453
Sekundäre Mausereignisse 425
Select() 553, 570
SelectAll() 570
SelectedImage-Eigenschaft 589
SelectedImageIndex-Eigenschaft 590, 593
SelectedIndexChanged-Ereignis 575
SelectedIndex-Eigenschaft 571, 573, 706
SelectedIndices-Eigenschaft 574
SelectedNode-Eigenschaft 589, 590
SelectedTabChanged-Ereignis 706
SelectedTab-Eigenschaft 706
SelectedText-Eigenschaft 570
SelectedValueChanged-Ereignis 575
SelectionLength-Eigenschaft 570, 662
SelectionMode-Eigenschaft 571, 574
SelectionStart-Eigenschaft 570, 662
SelectNextControl() 553
Semaphoren 1037
eigene Klasse 1038
Send() 629, 726
SendKeys.Send() 629, 726
SendMessage() 1009
SendToBack() 448, 553, 587
Separator 626
Serialisierung 708
Serialisierung von Eigenschaften durch den Designer 647
Serializable-Attribut 732
Server-Explorer (Visual Studio .NET) 63
Service Packs 1108
Session-Objekt 919
set-Accessor 297
SetAttributes() 385
SetBounds() 553
SetChildIndex() 448, 587
SetClip() 479
SetCreationTime() 383, 386
SetCurrentDirectory() 383
SetData() 727
SetDataObject() 728, 733
frühe Wertbindung 728
späte Wertbindung 728
– Vorteile 732
Strings 728
SetLastAccessTime() 383, 386
SetLastWriteTime() 383, 386
SetLength() 395
SetPixel() 485, 518
SetSelected() 575

SetStyle() 645
SetValue() 214, 714
shfusion.dll 1064
Shift-Eigenschaft 433
short 166
ShortCut-Eigenschaft 601
Show() 445, 553, 674, 698
  nicht-modaler Dialog 673
ShowApply-Eigenschaft 685
ShowDialog() 673, 677, 679, 684, 692
  Aufruf in Main() 717
ShowLines-Eigenschaft 589, 590
ShowPlusMinus-Eigenschaft 589, 591
ShowReadOnly-Eigenschaft 679
ShowRootLines-Eigenschaft 591
ShowShortCut-Eigenschaft 601
ShowToolTips-Eigenschaft 620, 621
ShutdownWindows (Codebeispiel) 1093
Sicherheitsloch 711
Sicherheitsphilosophie 81
Sicherungskopien von Quelltexten 76
Sieb des Eratosthenes 157
Signatur 269
  einer Methode 260
  Rückgabetyp 269
Simple API for XML 749
SimpleHandlerFactory-Klasse 876
Simulieren, Tasteneingaben 726
Sizable 444
SizableToolWindow 444, 445
SizeConst 1084
SizeF-Struktur 468, 470
  Eigenschaften 471
  Methoden 471
SizeInPoints-Eigenschaft 529
SizeMode-Eigenschaft 706
sizeof (Operator) 196
sizeof-Operator 1103
Size-Struktur 468, 470
  Eigenschaften 471
  Methoden 471
skalares Produkt 315
Skalarprodukt 315
Skalierung 464
SmallChange-Eigenschaft 579
SmoothingMode-Eigenschaft 510
SmtpMail-Klasse 826, 830
SOAP 942
  -Erweiterung 981
    – Auf Methode anwenden 984, 992
    – Auf Webdienst anwenden 983, 992
    – DLL in bin-Verzeichnis kopieren 992
  -Header 984
    – Auf Methode anwenden 994
    – Auswerten 995
    – Auswirkung auf Proxy-Code 995
    – Headers-Auflistung auswerten 994
    – Wert im Client setzen 996
SoapDocumentMethod-Attribut 970
SoapExtensionAttribute-Klasse 984
  Eigene Klasse ableiten 990
SoapExtension-Klasse 982
  ChainStream() 983
  Eigene Klasse ableiten 991
  GetInitializer() 983
  Initialize() 983
  ProcessMessage() 983
SoapHeaderAttribute-Klasse 985
SoapHeader-Klasse 984
  Eigene Klasse ableiten 993
SoapMessage-Klasse
  GetInParameterValue() 991
  MethodInfo 993
SoapMessageStage-Enumeration
  AfterDeserialize 991
Socket 1137
SolidBrush-Klasse 501
SolidBrush-Klasse Konstruktor 986
Sort() 220
Sorted-Eigenschaft 571, 574
Sortieren von Arrays 220
Source-Eigenschaft 344
späte Bindung 89, 1044
  .NET-Komponenten in COM 1064
Speicherauszüge 1118
Speicherverwaltung 48
Spline 511, 513
Splines, kardinale 477
Split() 367
SplitterEventArgs 588
Splitter-Klasse 586
SplitterMoved-Ereignis 588
SplitterMoving-Ereignis 588
Splitter-Steuerelement 576, 586, 587
  magnetisches Gitter für 588
Sprachübergreifende Fehlerbehandlung 341
sprachunabhängige Objekte 85
Sprunganweisung, unbedingte 156
SSL 1135, 1138
Stack 114

als Speicherort für Werttypen 165
Platz reservieren 1100
-Rahmen 127
stackalloc 1100
stackalloc-Operator 1103
StackTrace-Eigenschaft 344, 345
Standarddialoge 676
Standarddialoge (Codebeispiel) 680
Standardereignis, Behandlungsmethode ergänzen für 417
Standard-Gateway 1131
Standardkonstruktor 252, 282
Standardnamensraum 99
Standardnamespace-Eigenschaft 99, 545, 740
Standardoperatoren 191
überladbare 306
starke Typisierung 95
starker Name 45, 1063
Start() 428
StartCap-Eigenschaft 498
startfähiges Projekt 278
StartFigure() 475
Startoptionen
für Anwendungen 59
Visual Studio .NET 76
Startprojekt
mehrere 280
StartsWith() 173, 367, 663
STAThread-Attribut 246
static 104, 229, 249
-Methode 277
statische
Bibliothek 87
Datenfelder 229
– Lebensdauer 233
– vs. Instanzfelder 249
Elemente, einer Klasse 252
– Initialisierung 253
Instanz 244, 253, 277
Konstruktoren 249
– class-Klasse 284
– struct-Klasse 282
– vs. Main() 279
Methoden 258, 260, 277
Standardkonstruktoren 282
Variablen 229
statische Bibliotheken 1041
statische Elemente
nicht für COM 1061
statische Instanzen 1076
StatusBar-Klasse 624
StatusBarPanel-Auflistungs-Editor 624
StatusBarPanelClickEventArgs-Klasse 624
StatusBarPanel-Eigenschaft 624
StatusBar-Steuerelement 624
Statusleiste 624
Problem mit AutoScroll-Eigenschaft des Formulars 625
Steuerelement
als Container 585
andocken 587
Button 564
CheckBox 565
ComboBox 571
ContextMenu 607
eigenes _ in Toolbox verfügbar machen 665
eigenes, dauerhaft in Toolbox übernehmen 668
eigenes, verfügbar machen 665
Eigenschaften, Überblick über 546
Eigenschaften-Fenster 542
geerbtes hinzufügen 670
GroupBox 585, 586
identifizieren 557
Invoke()-Methode 856
Koordinatensystem 555
ListBox 571
MainMenu 601
Methoden, Überblick über 551
modifizieren, bestehendes 656
NumericUpDown 727
Panel 585, 586
platzieren in Entwurf 539
RadioButton 566
Reihenfolge beim Zeichnen 447
selbst ableiten 633
Splitter 576, 586
StatusBar 624
TabControl 704
TextBox 568
ToolBar 618
Toolbox, _e der 563
TrackBar 578
TreeView 588
Überblick über _ 539
Vererbung für 656
VScrollBar 578
was passiert beim Platzieren 637
Z-Reihenfolge 447
zur Laufzeit generieren 556
Steuerelementbibliothek 635
(Benutzer-)Steuerelemente zusammenfassen in 665

Steuerlemente gleich entwickeln in 666
Verweis auf einfügen 668
Steuerelementdesign 635
Ansichten 636
Steuerelemente
Thread-Synchronisation 1009
zur Thread-Synchronisierung 1012
Steuerelemententwurf
Steuerelemente platzieren in 539
Stift 494
Strichstärke 495
STL 213
Stop() 428
Stream 390
Binärdatei lesen 399
-Operationen 391
-Operationen, rohe 391
schließen 398
zeichenweise lesen 398
Stream-Klasse 390
BeginRead() 854, 855
Close() 398
Eigenschaften 393
EndRead() 855
StreamReader-Klasse 391, 392, 825, 835, 839, 840, 862, 866
StreamWriter-Klasse 391, 392, 825, 860, 865
Write() 686
Strichstärke 495, 497
Drucker vs. Bildschirm 495
Strikeout 482
StrikeOut-Eigenschaft 530
strikte 118
strikte Typisierung 118
String 534
string 172, 372
Call-by-reference-Übergabe 264
Call-by-value-Übergabe 263
Escape-Sequenzen 170
Formatierung von 367
Hintergründe zu 361
Methoden 173
Nachteile 175
Werttyp oder Verweistyp 172
String.Format() 367
StringAlignment-Aufzählung 480, 530
StringBuilder (Codebeispiel) 374
StringBuilder-Klasse 175, 363, 372, 839, 860
Fenstertitel 1091
für API-Aufrufe 1079
StringCollection-Klasse 375, 604
Eigenschaften 373
Methoden 374
StringFormat (Codebeispiel) 371
StringFormat-Klasse 480, 530, 531, 532
Stringliterale 174
Stringmanipulation 175, 363
StringOps (Codebeispiel) 364
Stringpool 362
StringReader-Klasse 391
Strings
DLL-Aufrufe 1079
SizeConst in Strukturen 1084
StringTrimming-Aufzählung 530
Elemente 531
Stringvergleich 362
StringWriter-Klasse 391
struct
allgemein 186
als Klasse 187
Boxing/Unboxing 188, 191
DLL-Aufrufe 1081
Initialisierung 188
Konstruktur 188
LayoutKind 1082
Marshalling als Funktionsergebnis 1090
parameterloser Konstruktor 189
– statischer 189
ValueType als Basisklasse 190
virtuelle Methoden 190
vs. class 190, 1083
struct-Klasse
Konstruktor 282
Schnittstellenimplementierung 190
StructLayout 1081
StructLayout.Sequential
Debugger 1119
strukturierte
Fehlerbehandlung 342
Namensraumbezeichner 99
Style-Eigenschaft 530, 623
Stylesheet 895
Web Forms-Seite zuweisen 908
Subnetz 1131
SubString() 367
Suchdialog 699
SuchenErsetzen (Codebeispiel) 699
summary-Tag 134
Sun Systems 81
SurroundColors-Eigenschaft 507

SuspendLayout() 417, 554
switch-Anweisung 145
fall through 146
Symbolleiste 618
im Designer zusammenstellen 619
Transparenzfarbe, Problem mit 619
Zwischenablagenfunktionalität implementieren 725
Symbolschaltflächen 618
Arten von 620
DropDown- 624
hinzufügen, im ToolBarButton-Auflistungs-Editor 619
per Drag&Drop hinzufügen 631
synchrone
Ausführung 124
Dateioperationen 400
Synchronisation
Events 1034
Gründe 1004
lock() 1005
Mutex 1006
Rückrufe 1010
Semaphoren 1037
Steuerelemente 1009
Synchronisationsobjekte 1030
SyncRoot-Eigenschaft 376
System.Array 155
System.Boolean 169
System.Byte 166
System.Char 170
System.Collections 155, 326
System.Decimal 168
System.Double 167
System.Enum 185, 197
System.GC.Collect() 288
System.Int16 166
System.Int32 166
System.Int64 166
System.Object 178
System.SByte 166
System.Single 167
System.String 173
Hintergründe zu 361
System.String-Klasse
Methoden 363, 365
System.UInt16 166
System.UInt32 166
System.UInt64 166
System.ValueType 226
SystemColorsChanged-Ereignis 489
SystemColors-Klasse 485, 490
Eigenschaften 485
Systemdienste von .NET 91
SystemException-Klasse 349, 838
Systemfarbe
Bekannte ermitteln 986
Systemfarben 485
SystemInformation-Klasse 418
Eigenschaften 419
Systemmenü (M 693

T
Tab entfernen (Befehl) 707
TabAlignment-Aufzählung 704
TabAppearance-Aufzählung 704
TabControl-Klasse 704, 714
Eigenschaften 704
Methoden 706
TabControl-Steuerelement 704
(Codebeispiel) 714
TabCount-Eigenschaft 706
TabDrawMode-Aufzählung 705
Tabelle
Zebrastreifen mit XSLT generieren 773
TabPage-Auflistungs-Editor 707, 720
TabPage-Klasse 707
TabPages-Eigenschaft 706, 707
TabStop-Eigenschaft 550
Tabulator-Reihenfolge 550, 586
-Tag 134
-Tag 138
Tag-Eigenschaft 550, 557, 593
zur Identifikation von Symbolschaltflächen 624
TargetSite-Eigenschaft 344
Tastatur 430
Tastaturanalyse (Codebeispiel) 433
Tastatureingabe simulieren 726
Tastaturereignis 431
Überblick 432
Vorschau durch Formular 434
Wege eines 434
Tastaturschnittstelle 434, 557, 613, 644, 654
Eingaben simulieren 726
für Zwischenablage 725
implementieren 663
Tastaturvorschau (Codebeispiel) 437
Tastenkürzel 550
TCP 1128
-Header 1129
-Paket 1129
Port 1129

TCP/IP 882
TCP/IP-Stapel 1137
TcpClient-Klasse 825, 834, 835, 840
Technologien für Mehrschichtsysteme unter .NET 813
Template 918
TextAlign-Eigenschaft 550, 622
Textausgabe 527
TextBox
  Ableitung von 658
  Zwischenablage 725
TextBoxBase-Klasse 725
TextBox-Klasse 568
  Eigenschaften 569
  Ereignisse behandeln 660
  Methoden 570
  Zwischenablage 725
TextBox-Steuerelement 568
TextChanged-Ereignis 592
  behandeln 660
Texteditor, Einstellungen 76
Text-Eigenschaft 550, 574, 593, 623, 649
  Vorgabewert des Designers für Benutzersteuerelemente nutzen 650
  wie kommt der Vorgabewert zustande 650
Textfeld
  Ableitung von 658
Textkodierung 753
TextReader (Codebeispiel) 176
TextReader-Klasse 391
TextureBrush-Klasse 503
Texturpinsel 503
TextWriter-Klasse 391
this
  als Bezeichner für Indexer 300
  Delegaten bei DLL-Rückrufen 1091
  DLL-Rückrufe 1090
  Konstruktor 282
  Qualifizierung mit, bei Verdeckung 235
Thread
  Anweisungsfolge als 124
  -Pool, des Systems 400
Thread.Join() 1002
Thread.Start() 1001
ThreadIAsyncResult (Codebeispiel) 1014
Threading-Namensraum 999
Thread-Klasse 999
  Überblick 1002
  versiegelt 1001
ThreadPriority-Eigenschaft 1002
Threads 999
  asynchrone Methoden 1014
  Deadlock 1007
  Dokumentation 999
  eigene Klasse 1001
  Einsprungpunkt 1000
  Entkoppelung
    – durch andere Threads 1020
  Events 1034
  gemeinsame Ressourcen 1024
  GUI-Thread 1013
  Haltepunkte 1000
  IasyncResult-Klasse 1015
  Interaktion 1016
    – Codebeispiel 1020
  lock() 1005
  Monitor-Klasse 1039
  Mutex 1006
  Namensräume 999
  ohne eigene Klasse 1003
  Parameterwechsel 1017
  Programm- vs. Thread-Ende 1026
  Programmende 1018
  Recognizer-Beispielklasse 1017
  Rückrufe 1010
    – Delegaten 1011
    – nachträglich synchronisiert 1012
  Semaphoren 1037
    – eigene Klasse 1038
  Synchronisation 1004
    – mehrerer Instanzen 1022
    – Steuerelemente, InvokeRequired() 1013
  Synchronisationsobjekte 1030
  Trace.WriteLine() 1000
  Umschaltung 1005
  WaitOne() 1036
  Wettrennen 1008
  Work-Liste 1030
ThreadState-Eigenschaft 1002
ThreadSyncInCB (Codebeispiel) 1013
ThreadwithoutClass (Codebeispiel) 1003
ThreeState-Eigenschaft 565, 566
throw-Anweisung 156, 345
Tick-Ereignis 428
TickFrequency-Eigenschaft 579
TickStyle-Eigenschaft 580
Time to Live 1132
Timer 427
Timer-Klasse 427
  Eigenschaften 428
  Thread-Entkoppelung 1019

Timer-Komponente 416
TimeSpan-Klasse
FromSeconds() 973
Title-Eigenschaft 679
tlbexp.exe 1065
vs. RegAsm.exe 1066
tlbimp.exe 1048
Speicherort 1049
TLS 1.0 1138
ToCharArray() 367
TODO-Kommentare 128
Toggle() 594
ToggleButton 626
Tokenkommentare 128
ToLower() 367
ToolBarButton-Auflistungs-Editor 619
ToolBarButtonClickEventArgs-Klasse 623
ToolBarButton-Klasse 618, 619, 621
Eigenschaften 622
Tag-Eigenschaft, Verwendung der 624
ToolBarButtonStyle-Aufzählung 620
ToolBar-Klasse 618, 623
DropDown-Schaltflächen 624
Eigenschaften 621
Ereignisse 622
Schaltflächen verdrahten 623
ToolBar-Steuerelement 618
Toolbox 52, 658
abgeleitetes Steuerelement anzeigen, Problem mit 657
eigene (Benutzer-)Steuerelemente verfügbar machen 665
Erscheinen einer Steuerelementklasse in 658
Leiste verschieben 541
was passiert beim Platzieren von Steuerelement-Instanzen 637
Toolbox anpassen 668
ToolTip-Komponente 492, 502
ToolTipText-Eigenschaft 623, 624
TopLevelControl-Eigenschaft 550
ToString() 179, 367
Aufzählklassen mit Flags-Attribut 433
enum-Typ 184
– Wirkung des Flags-Attributs auf 183
ToUpper() 367
Trace.WriteLine() 1000
Trace-Klasse 311, 1000, 1116
Tracert.Exe 1134
TRACE-Symbol 1116
TrackBar-Klasse 578
Eigenschaften 579
Knob-Steuerelement als Gegenstück 642
TrackBar-Steuerelement 578
Transform() 465, 474
Transform-Eigenschaft 504
TransformPoints() 467
TranslateClip() 467
TranslateTransform() 465, 467, 504
Translation 464
TransparentColor-Eigenschaft 613, 614
Probleme mit 617
Transparenz 484, 485
Transparenzfarbe
ändern 618
für Symbolleiste 619
TreeNode-Klasse 589
Eigenschaften 592
Methoden 593
TreeViewDemo (Codebeispiel) 594
TreeView-Eigenschaft 593
TreeView-Klasse 588
Eigenschaften 590
Ereignisse 591
Methoden 591
TreeView-Steuerelement 588
Trim() 367
TrimEnd() 367
Trimming-Eigenschaft 530
TrimStart() 367
TripleClick-Ereignis 453
Truncate() 469
try-Block 345
Typauflösung 43, 45
Typbezeichner
qualifizieren 97
Typbibliothek (COM) 89
Typbibliothek (COM) importieren 1048
Typbibliotheken 118
Typisierung 118
Typmanagement 90, 119
Typografie 480, 481
Typreflexion 196
Eigenschaften aufzählen 197
enum-Typen aufzählen 185
Typsicherheit 33, 199, 260
Ausnahmebehandlung 341
bei foreach-Iteration 151, 153, 154
Parameterübergabe an Methoden 260
Typsystem, Entwicklung unter Windows 87
Typüberprüfung
statische vs. dynamische 34

Typumwandlung 222
Auflösung von Operatoren 310
Boxing 227
Boxing/Unboxing 181
explizite 222, 223
implizite 222, 223
Methodenparameter für 260
Unboxing 227
Vererbung, bei 225
Werttypen 166
Typumwandlung (Codebeispiel) 224
Typumwandlungsoperatoren 309
Typverletzungen, behandeln 339
Typverwaltung 42
.NET 43
für COM 42

U
Überlademechanismus 269
Überladen 258
Hauptanwendungsbereiche 271
Methoden 268
Namensauflösung 269
Standardoperatoren 306
– Einschränkungen 306
Vererbung und 315
wann sinnvoll 270
überladene Methoden
und COM 1061
Überlaufkontrolle 142
ein-/ausschalten 342
Übernehmen-Schaltfläche 681, 696
Überprüfungssteuerelement 912
RequiredFieldValidator einsetzen 912
überschreiben 258, 272
Ausgraben verdeckter Methoden 274
Basisklassenmethoden 270
Codebeispiel 272
durch override-Deklaration 276
statische Methoden 274
Vererbung und 315
Überschreibungslinie 276
Überwachen (Visual Studio .NET) 72
Überwachungsfenster 729
UDDI 943
UDP 1137
uint 166
UInt64 166
ulong 166
Umbenennen
EXE-Datei 546
Formular 545
Namensbereich 545
Namensraum, Probleme bei 99
Projektmappe 545
Projektname 545
vom Designer generierte Bezeichner 544
Umrissfigur 474
Unbedingte Verzweigung 155
Unboxing 181, 226, 227
struct 188
unchecked 142, 343
Underline 482
Underline-Eigenschaft 530
ungarische Notation 109
Unicode 170, 754
UnicodeEncoding-Klasse 170
Union() 471, 473
uniondemo (Codebeispiel) 1085
Unions 1084
Unit-Eigenschaft 530
Unix 84
UnmanagedType 1078
UnmanagedType-Klasse 1078
unsafe 104, 114, 196
Destruktor 287
Klassen mit unsafe-Methoden 1101
Möglichkeiten und Grenzen 1101
Operatoren für 191
Voraussetzungen 1098
vs. unverwaltet 1099
Unsichere Codeblöcke zulassen 196
Unterlänge 482
Update
Versionskonflikte 92
UseDefaultCursors 739
User Datagram Protocol 1137
UserControl-Klasse 633, 638
Text-Eigenschaft sichtbar machen 649
Toolbox, Erscheinen einer Steuerelementklasse in 658
UserPaint 645
ushort 166
vs. char 169
using (Anweisung) 291
using-Anweisung vs. -Direktive 99
using-Direktive 97
untergeordnete Namensräume 100
UTF
-16 754
-8 754

UTF7Encoding-Klasse 170
UTF-8 753
UTF8Encoding-Klasse 170
UUID, vs. GUID 1043

V
Validated-Ereignis 555
ValidateNames-Eigenschaft 679
Validating-Ereignis 555
value 297
ValueChanged-Ereignis 580, 645
Value-Eigenschaft 580
ValueMemberChanged-Ereignis 575
ValueMember-Eigenschaft 574
ValueMember-Elemente 571
value-Tag 138
ValueType-Klasse 117
als Basisklasse für struct 190
variable Parameteranzahl 266
Variablen 228
Anzeige in Visual Studio .NET 71
Geltungsbereich 231
Instanzfelder, als 228
lokale 228
– Lebensdauer 233
Verdeckung von 235
VB.NET
Kompatibilität 105
reservierte Schlüsselwörter 106
VBA 86
VBByRefStr 1078
VBScript 883, 889
Vektorprodukt 315
Verb/Objekt-Reihenfolge 111
Verdecken von Variablen 235
Verdrahten
Symbolschaltflächen 623
Vereinbarung
Arrays 213
Initialisierung bei 115
Variablen 163
Vererbung 241, 657
als wohldefinierte Schnittstelle 709
Arten der 316
auf Codebasis 91
Einfachvererbung 317
Formular
– Implementierungsstrategien 710
Implementierungsvererbung 318
– partielle 319
Indexer 303
Mehrfachvererbung 318
Modifizierer für 246
polymorphe 319
Schnittstellen- 336
Schnittstellenvererbung 318
Typumwandlung bei
– 225
Überblick über 315
Verkapselung 711
von Benutzersteuerelement 669
– Ausgangsituation 670
– schrittweise 670
von Formular 708
– schrittweise 718
Vererbungsmodifizierer 259
Vererbungsschnittstelle 711
Verkapselung 711
Versalhöhe 482
verschachtelte Datentypen 250
Verschachtelung von Namensräumen 98
Verschiebeoperation, Drag&Drop 742
Versionsinformation 44
Aufbau 44
Versionskontrolle 92
Versionsverwaltung 91
verteilte Anwendungen
allgemein, DCOM, COM+ 90
verteiltes Programm 809
verwaltete Erweiterung für C++ 91
verwaltete Erweiterungen, C++ 33
Verweis 44
Verweis hinzufügen 635
Verweistyp 116
als Heapwert 114
Boxing 227
einfacher 172
komplexer 198
object 178
Unboxing 227
vs. Werttyp 112
Verzweigung
unbedingte 155
ViewState-Objekt 907, 914
Virenschutzprogramme 1109
virtual 104, 259, 275, 277, 320, 331
Datenfelder als 250
virtuelle Java-Maschine 31
virtuelle Methode 275, 320
Prinzip der 179
struct 190
vs. COM 1045

Virtuelles Verzeichnis anlegen 878
Visible-Eigenschaft 550
Visual Basic
Objektorientierung 86
Visual Basic .NET 32
Visual Basic for Applications 86
Visual Studio .NET 70
ActiveX-Steuerelemente 1053
– Import 1054
Anwendung starten 54
Debug und Release 74
Debugging 1115
Delphi 6, Parallelinstallation 1111
Designer-Rahmencode 53
Eingabeaufforderung 1049, 1055, 1061
Einrichtungen zur Erleichterung des Code-Designs 409
Erste Schritte 49
EXE-Dateien, Speicherort 56
Fenster 52
Größe visueller Komponenten 53
GUID erzeugen 1063
Haltepunkte 1117
Installation
– Konflikte mit anderen Programmen 1109
– Service Packs 1108
– Voraussetzungen 1107
– Zeitbedarf, Umfang 1109
Kommandozeilenparameter 59
Konfigurationen 74
Konsolenanwendungen 56
Projektverweise 1047, 1055
– manueller Nachtrag 1056
Server-Explorer 63
Sicherungskopien von Quelltexten 76
Speicherauszüge 1118
Startoptionen 76
Syntaxhilfe
– für COM-Hüllklassen 1050
Texteditor, Einstellungen 76
Toolbox
– ActiveX-Steuerelemente 1053
Trial-/Beta-Version, Upgrade von 1110
unsicherer Code 1098
VS 6, Parallelinstallation 1110
Webdienst 61
– Client 65
Windows-Anwendungen 49
Visual Studio 6, und Visual Studio .NET 1110
Visual Studion .NET
Installation 1107
VolumeSeparatorChar-Eigenschaft 386
von 226
Vordefinierte Datentypen 118
Vorhandenes Element hinzufügen (Befehl) 641, 665
Vorlage 918
Vorschau von Tastasturereignissen 434
Vorsicht mit
Internet Information Server 62
VS.NET
Einrichtungen zur Erleichterung des Code-Designs 409
VScrollBar-Klasse 578
Eigenschaften 579
VScrollBar-Steuerelement 578
VScroll-Eigenschaft 456
vsdisco 875

W
Wait() 1039
WaitOne() 1036
Web Form 899, 900
Web Forms-Seite 873
Datenbindung 915
Eigenschaften einstellen 901
h1-Element anlegen 901
Mit Client-seitigem Skript validieren 902
Startseite festlegen 903
Stylesheet zuweisen 908
ViewState-Objekt benutzen 907
Webserver-Steuerelement 910
Wert des action-Attributs wird ignoriert 903
Web.config 874, 875, 876
Webanwendung 869, 881, 1138
Webbrowser mit dem IE-Steuerelement 816
WebClient-Klasse 841, 848, 854
Webdienst 61, 869, 941, 1138
Abort() 959
Akronyme 942
Application-Objekt benutzen 964
asmx-Datei im IE öffnen 946
Aufruf 67
Beteiligte Dateien 950
C#-Hüllklassse 68
Caching benutzen 965
Client 65
Code für asynchrone Methode im Proxy 956
CookieContainer-Klasse benutzen 964
Debuggen 952
DISCO verwenden 952
Erweitern mit SOAP-Erweiterung und SOAP-Header 981
Hallo Welt 943

Header VsDebuggerCausalityData 953
Implementation 63
Inhalt einer SOAP-Nachricht 946
Mehrere in einem Projekt implementieren 968
Methode asynchron aufrufen 955
Muster asynchroner Methoden-Aufrufe 956
Proxy instanziieren 950
Proxy-Code 945
ProxyTrace benutzen 953
Session-Objekt benutzen 962
Test-Client als Startprojekt der Projektmappe festlegen 950
Test-Client hinzufügen 944
Timeout 955
URL 70
Von WebService ableiten 965
Webverweis zu Client hinzufügen 944
Wsdl.Exe 946
WSDL-Datei 947
Zustand halten 961
WebException 959
WebForms 66
WebMethod-Attribut 63
CacheDuration-Parameter 966
EnableSession-Parameter 962
WebRequest-Klasse 860, 862
Ableitung eigener Klasse 864
RegisterPrefix() im Beispiel 866
WebResponse-Klasse 862
Ableitung eigener Klasse 865
Webserver-Steuerelement 910
Kombinationslistenfeld 913
– auslesen 914
Label formatieren 911
Listengebundenes 917
Optionsschaltflächengruppe 913
– auslesen 914
RequiredFieldValidator einsetzen 912
Textfeld als Passwort-Feld einsetzen 911
Überprüfungssteuerelement 912
WebService
Cache-Objekt benutzen 971
WebServiceAsync (Codebeispiel) 955
WebService-Attribut 970
Description 970
WebServiceBinding-Attribut 970
Webverweis hinzufügen 67
Weiter (Visual Studio .NET) 73
well known ports 1137
Weltkoordinatensystem 464
Wertebereich
decimal, von 168
Ganzzahltypen von 166
Gleitkommatypen, von 167
Wertebereichsüberschreitung
behandeln 339
prüfen 142
Wertigkeit, Operatoren von 305
Wertkonstante 237
vs. literale Konstante 236
Wertlisteninitialisierung 214, 215, 217, 218
Werttyp 115
als Stackwert 114
Boxing 227
Boxing/Unboxing 226
decimal 168
einfacher 165
Ganzzahltypen 166
Gleitkommatypen 167
komplexer 182
Typumwandlung 166
Unboxing 227
Vereinbarung und Initialisierung 165
vs. Verweistyp 112
Wertzuweisung
Stringzuweisung als 362
Wettrennen bei Threads 1008
WfData (Codebeispiel) 916
WfDataGrid (Codebeispiel) 919
WfDataList (Codebeispiel) 928
WfFormHtml (Codebeispiel) 900
WfFormWeb (Codebeispiel) 910
WfRepeater (Codebeispiel) 924
while-Anweisung 149
WHOIS (Codebeispiel) 824
Whois-Abfrage 822
Wiederverwendung von Code 40, 316
Konzepte 1041
Win32-API, vs. .NET 29
Window-Eigenschaft 486
WindowFrame-Eigenschaft 486
Windows Component Update 1111
Windows Form Designer generated code 414
Windows herunterfahren 1093
Windows-Anwendung
Codegerüst 412
Windows-Anwendungen 408
Windows-Anwendungen erstellen 49
Windows-Anwendungen, Kommandozeile 60
Windows-API 1069

Codebeispiel FindFirst 1086
Marshalling 1078
Rückrufe 1090
Strings 1079
Windows herunterfahren 1093
Windows-Farbschema 485
Windows-Registrierung 43
Windows-Steuerelementbibliothek 635
WindowText-Eigenschaft 486
with-Anweisung, Delphi, VBA 100
WM_ERASEBKGND 518
WM_EXITSIZEMOVE 525, 527
WM_PAINT-Nachrichten 461
WndProc() 526
Word 86, 531
WordWrap-Eigenschaft 570
World (Einheit) 468
World Wide Web 881
WrapMode-Eigenschaft 504
Wrappable-Eigenschaft 622
Write() 395, 686
Formatierungszeichenfolgen 368
WriteByte() 395
WriteLine() 58, 266, 268, 271
WsCaching (Codebeispiel) 972
WsCachingUndState (Codebeispiel) 962
WSDL 943
Wsdl.Exe 946
WsGrafikSpender (Codebeispiel) 985
WsHelloWorld (Codebeispiel) 943
WWW 881

X
XDR 749
XE 1128
X-Eigenschaft 421
XLink 750
XML 747
Akronyme, verwandte 748
Attribut 751
CDATA-Abschnitt 755
Character Reference 756
Dokument-Element 751
Element 750
encoding-Attribut 754
Entity Reference 755
Ergebnisbaum 756, 759
Erzeugen mit .NET-Klassen 784
Fehler 'Switch from current encoding to specified encoding not supported.' 755
gültig 801
im Webdienst 942
Informationsquellen 747
ISO-8859-1 754
Klassen in .NET 783
Knoten 751
Kommentar 752
Namensraum 752, 758
Prozessor des Internet Explorers verwenden 756
Sortieren mit ArrayList-Klasse 796
spezielles Zeichen in 755
Syntax 750
Textkodierung 753
Transformieren mit .NET-Klassen 784
Transformieren mit XslTransform-Klasse 792, 793
Transformieren, siehe auch XSLT 756
Unicode 754
UTF-16, UTF-8 754
UTF-8 753
Validieren 801
wohlgeformt 801
Wurzel-Element 751
Zeichensatz 753
XML Data Reduced 749
XML DOM 783
XML Schema Definition Language 749, 804
XmlAttribute-Klasse
Value 991
XmlAttributes-Klasse
Append() 991
XML-Deklaration 754
XmlDocument-Klasse 783, 788, 791, 793, 796, 804
AppendChild() 788, 791
CreateAttribute() 991
CreateComment() 788
CreateElement() 991
ImportNode() 791
Knoten erzeugen 788
Load() 807, 991
LoadXml() 788, 991
PrependChild() 788
Save() 991
SelectNodes() 796
SelectSingleNode() 788
Wert eines Attributs setzen 788
Wert eines Elements setzen 788
XML-Dokumentationsdatei-Eigenschaft 132
XmlElement
GetAttribute() 794

XmlElement-Klasse
AppendChild() 991
InnerText() 991
XmlNode-Klasse
InnerText 788
InnerXml 788, 791
OuterXml 788
XmlNodeList-Klasse 796
foreach 798
XML-RPC 942
XML-Spezifikation
Zeilenumbruch besteht aus Linefeed 979
XML-Tags
für Dokumentationskommentare 134
XmlText-Klasse 788
XmlTextReader-Klasse 784, 804, 806
ReadInnerXml() 791
XML-Formatierung mit 791
XmlTextWriter-Klasse 784, 788, 796
WriteAttributeString() 790
WriteElementString() 790
WriteEndAttribute() 790
WriteEndDocument() 790
WriteEndElement() 790
WriteStartAttribute() 790
WriteStartDocument() 790
WriteStartElement() 790
XML formatieren mit 791
XmlValidatingReader-Klasse 806
XmlValidierung (Codebeispiel) 801
XPath 750, 756
/ 763
@ 759
ancestor::node() 764
boolean() 801
concat() 801
count() 764
number() 800
position() 769, 773
round() 800
string() 801
substring() 764
sum() 780
XPathDocument-Klasse 784, 798
XPathExpression-Klasse 799
AddSort() 799
XPathNavigator-Klasse 783, 798, 799
Compile() 799
Evaluate() 800
Select() 799
XPathNodeIterator-Klasse 800
XPointer 750
XSD 749, 804
XSD.EXE 804
xsl
apply-templates 761, 764, 767, 770
attribute 774
call-template 764
choose 771
copy 796
copy-of 796
element 774
for-each 759, 777
if 770
key 776
number 769
otherwise 772
output
– funktioniert nicht mit XslTransform-Klasse 796
param 764
preserve-space 759
sort 767, 796
strip-space 759
stylesheet 760
template 760, 764
text 759, 774
value-of 759, 764, 780
when 772
with-param 764
XSL Formatting Objects 749
XSL-FO 749
XSLT 756
Attribut einfügen 773
Bedingung 769
Benannte Vorlage 762
Einrückungen entsprechend der Element-Hierarchie ausgeben 762
Element einfügen 773
Elemente kopieren 796
Ergebnisbaum 759
format-number() 769
Funktionale Programmierung 765
generate-id() 777
Gruppieren 775
key() 777, 780
Muench-Methode 775
NaN 782
Nummerieren 768
Parameter 762
Prädikat 769

Rekursion 775, 780
Sortieren 765, 796
Template 760
Variable 762
Variable, Sichtbarkeit 764
Vorlage 760
Zebrastreifen-Tabelle 773
Zwischensumme durch Rekursion bilden 775
XSLT (E 748
XslTransform-Klasse 784, 793, 794

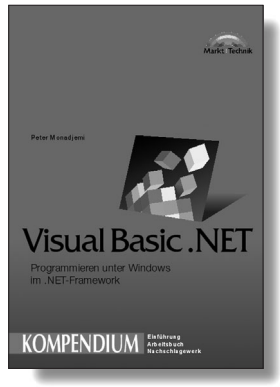
Y
Y-Eigenschaft 421
Yes 675
YesNo 674
YesNoCancel 674

Z
Zebrastreifen-Tabelle 773
Zehntelsekundenregel 515
Zeichenketten 172
Zeichenkodierung 391
Zeichenoperation 459, 463
Koordinatensystem 463
Zeichensatz 753
Zeichnen 461, 509
in Bitmap 515
Zeichnen, gepuffert 514
Zeiger
in unsicherem Code 1101
Zeigerkonzept 113
Zeilenhöhe für Schrift 529
Zeilenkommentar 127, 128
Zeilenumbruch
bei der Druckausgabe 690
Zeitanzeige (Codebeispiel) 429
Zeitscheibe für Threads 1005
Zeitsteuerung 427
Zielobjekt 725
Z-Reihenfolge 447
Zugriffsmodifizierer 101, 231, 247
für Klasse 246
für Klassenelemente 247
Zustandsklasse 858
Zustands-Objekt 851
Zuweisung
Arrays 214
kombinierte 195
Zweischichtsystem 810
Zwischenablage 627, 725
Bestehende Inhalte erhalten 734
Daten einlagern 728
Datenformat in, testen 730
Datenformate 729
Datentypen, eigene, anwendungsübergreifend transferieren 731
direkt ansprechen 727
frühe Wertbindung 728
GetDataObject() 729
Mehrere Datenformate/typen en bloc 733
Referenztypen und eigene Datentypen 730
SetDataObject() 728
späte Wertbindung 728
– Vorteile 732
Tastaturschnittstelle 725
Zwischenablage (Codebeispiel) 727
Zwischensprache 123

Markt+Technik

Programmierung

KOMPENDIUM

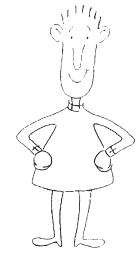


Peter Monadjemi
Visual Basic.net

ISBN 3-827**2-6273**-9, 1 CD-ROM
ca. 950 Seiten
€ 49,95 [D] / € 51,40 [A]

Fortgeschrittene / Profis

In diesem Kompendium erfahren Sie alles sowohl über die VB.NET-Programmierung als auch über die Softwareentwicklung unter dem .NET-Framework. Von der Entwicklungsumgebung Visual Studio .NET über OOP, Debugging, GUI-Erstellung, Datenbankanbindung mit ADO.NET bis hin zur ActiveX-Programmierung - das Wissen für den Ein- oder Umstieg ist kompakt, unterhaltsam und gut strukturiert aufbereitet.



Sie suchen ein professionelles Handbuch zu allen wichtigen Programmen oder Sprachen? Das Kompendium ist Einführung, Arbeitsbuch und Nachschlagewerk in einem. Ausführlich und praxisorientiert.

Unter **www.mut.de** finden Sie das Angebot von Markt+Technik.

InformIT™
CHECK IT

... aktuelles Fachwissen rund
um die Uhr – zum Probelesen,
Downloaden oder auch auf Papier.

www.InformIT.de

InformIT.de, Partner von **Markt+Technik**, ist unsere Antwort auf alle Fragen der IT-Branche.

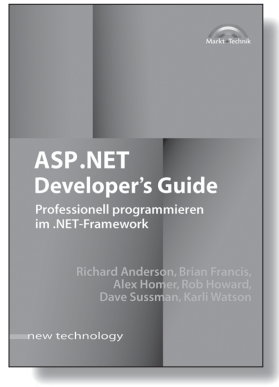
In Zusammenarbeit mit den Top-Autoren von Markt+Technik, absoluten Spezialisten ihres Fachgebiets, bieten wir Ihnen ständig hochinteressante, brandaktuelle Informationen und kompetente Lösungen zu nahezu allen IT-Themen.



wenn Sie mehr wissen wollen ... **www.InformIT.de**

Markt+Technik

Programmierung
.NET

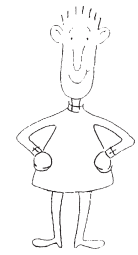


Richard Anderson et. al
ASP.NET Developer's Guide

ISBN 3-827**2-6280**-1, ca. 1320 Seiten
€ 79,95 [D] / € 82,50 [A]

Profi

Dieses umfangreiche Handbuch für professionelle Programmierer enthält viele praxisorientierte Code-Beispiele und umfassende Fallstudien. Sie lernen ASP.NET zu meistern und dynamische, erfolgreiche Web-Anwendungen für Ihr Unternehmen zu erstellen. Sie erfahren mehr über ASP.NET und das .NET Framework, wie man ASP.NET-Seiten erstellt und mit Server-Steuerelementen arbeitet. Das Buch widmet sich ebenso der Konfiguration und Weitergabe und befasst sich auch mit ADO.NET, XML und Web Services.

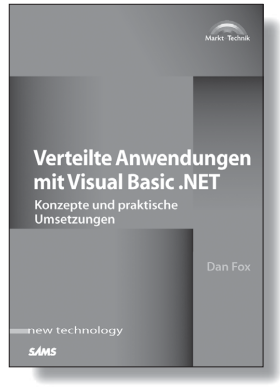


Ob Multimedia oder Datenbanken, ob Visual Basic oder C++, ob Einsteiger oder Profi – hier finden Sie Ihr Buch zum Thema Programmierung.

Unter **www.mut.de** finden Sie das Angebot von Markt+Technik.

Markt+Technik

Programmierung
.NET

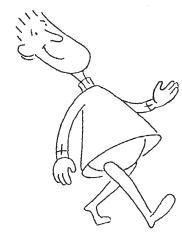


Dan Fox
Verteilte Anwendungen
mit Visual Basic.NET entwickeln

ISBN 3-827**2-6319**-0, ca. 630 Seiten
€ 59,95 [D] / € 61,70 [A]

Profi

Was nützt .NET, wenn man die Inhalte nicht ins Web bringt? Wenn Sie sich das auch schon gedacht haben (der Gedanke liegt auch nicht in allzu großer Ferne), sollten Sie sich als VB-Programmierer mit verteilten Anwendungen anfreunden. Sie werden erstaunt feststellen, dass das nun um Einiges leichter geworden ist. Und sie werden merken, dass Ihnen auch die praktische Anwendung leicht von der Hand geht. Dan Fox zeigt Ihnen wie.



Ob Multimedia oder Datenbanken, ob Visual Basic oder C++, ob Einsteiger oder Profi – hier finden Sie Ihr Buch zum Thema Programmierung.

Unter **www.mut.de** finden Sie das Angebot von Markt+Technik.



Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt.

Dieses eBook stellen wir lediglich als **Einzelplatz-Lizenz** zur Verfügung!

Jede andere Verwendung dieses eBooks und zugehöriger Materialien und Informationen, einschliesslich der Reproduktion, der Weitergabe, des Weitervertriebs, der Plazierung auf anderen Websites, der Veränderung und der Veröffentlichung bedarf der schriftlichen Genehmigung des Verlags.

Bei Fragen zu diesem Thema wenden Sie sich bitte an:

mailto:info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website



(http://www.informit.de)
herunterladen