PC Magazin SPEZIAL 29

Österreich € 11,50 / Belgien € 11,50 / Niederlande € 11,50 / Luxemburg € 11,50 / sfr 19,80 € 10,–

.NET- und C# Praxis

Professionell & effektiv programmieren

AUF CD

.NET UND C# -PRAXIS-EINSTIEG

Komplettes .NET Software Development Kit

- **Mit C#- und VB-Compiler**
- **SDK-Beispiele**
- Dokumentation
- **.NET-Laufzeitumgebung** zum Weitergeben
- Compiler auch unter Windows 9.x nutzbar

Freie Entwicklungsumgebung

SharpDevelop

Gleich losjegen und selbst programmieren

Beispielprogramme zur eigenen Erweiterung

- **Tray-Anwendungen**
-
-
-
-
-

Optimal umsteigen & starten

- **Endlich genau erklärt: Das ist .NET**
- **SDK und Framework verstehen**
- **Step by Step einsteigen**

Gratis mit Win 9.x

- **.NET** programmieren
- Eigenen Win **9.x-C#-**Compiler **entwickeln**
- **C#-Programme** unter Win 9.x kompilieren

Profiprogrammierung mit C#

- Script-Engine **für** eigene Programme
- Tipps zur optimalen Performance
- Google per Webservice in XML abfragen

Praxis-Workshops

- **temüberwachung mit Kompo-ten aus der C#-Toolbox**
- **perty-Grid: Compiler-Optionen ant setzen**
- **trols bauen & wiederverwenden**

4 194261 210004 29

.NET glasklar

„Das also ist .NET", werden Sie nach dem Durcharbeiten dieser Ausgabe sagen, „toll, endlich Klarheit!". Und: „Das ist ja wirklich das, was wir immer schon haben wollten!"

Sie müssen nur ins Heft schauen, Sie werden sehen, dass wir Recht haben, denn mit dieser Ausgabe von PC Magazin Spezial machen wir Schluss mit den ominösen Vermutungen, was denn .NET wirklich sei und werden konkret. Alles, was Microsoft bisher nicht verraten hat, (deshalb auch die Unsicherheit), steht in diesem Heft: .NET ist zusammen mit C# eine fantastische Fortentwicklung der Programmiertechnik für Entwickler und Anwender und ermöglicht zudem systemübergreifendes Programmieren. In den Libraries und den zu .NET gehörenden Programmiersprachen ist die Objektorientierung konsequent durchgehalten - und doch gibt es Abwärtskompatibilität zu klassischen C++-Konstrukten. In .NET kann Code zu Laufzeit kompiliert werden – und auf die Zielmaschine optimiert werden. In der .NET-Framwork Library gibt es Klassen ohne Ende. Die Framework Library ist die eigentliche Schnittstelle fiir Anwendungsprogramme. Sie ersetzt die Win32API. Aber „ersetzen" ist hier ein schwacher Ausdruck, denn mit .NET wird vieles, was in der API kniffelig und umständlich zu programmieren war, dank geeigneter Klassen und Objekte zum Einzeiler. Sie glauben es nicht? Wir beweisen es Ihnen in dieser Ausgabe anhand vieler Beispiele. Sie werden mit der Windows Forms Klasse arbeiten, Sie probieren das neue GDI+ aus, Sie absolvieren die Grundübung aller Programmierer, Daten laden und abspeichern, unter .NET aus, und Sie sehen, wie auf Datenbanken unter .NET zugegriffen wird.

Ulrich Rohde

Wir verraten fiir Profis und solche, die es werden wollen, was die .NET Toll-box liefert, wie man Fenster rund macht und vieles mehr.

Sie haben kein Visual Studio .NET, und konnen die Beispiele nicht nachvollziehen? Wir liefern auf der CD die Framework Library (mit Laufzeitsystem) und einen C#-Compiler, damit sie sich sofort von den Vorzügen der .NET-Programmier-Techniken überzeugen konnen. Sie finden auf der CD auch die freie Entwicklungsumgebung SharpDevelop. Sie bauen sich mit dieser Ausgabe sogar unter Win9.x eine kleine IDE fiir die .NET- Programmierung unter Win 9.x, und konnen dann gratis ausprobieren, was an .NET interessant und neu ist.

Sie werden staunen, das glauben wir!

U. Rohde

Ulrich Rohde
Chefredakteur

Das Heft und

DAS IST NET

Neues Programmieren 8

.NET hat einen recht weitgehenden Einfluss auf die Art und Weise, wie Software entwickelt, eingesetzt, verteilt und verwaltet wird.

VB und VC++ und COM: was ändert sich? 14

C# ist die eigentliche .NET-Sprache. Aber man kann für .NET auch mit Visual Basic und mit VC++ programmieren. Allerdings muss man einige Änderungen beachten.

DAS IST C#

O- die .NET Sprache 22

Die eigentliche .NET Sprache ist C# (C-sharp gesprochen). Und wer ein wenig Hintergrundwissen mit C oder C++ hat, wird sich schnell einarbeiten können.

Das erste .NET Programm 28

In diesem Beitrag werden Sie Ihr erstes eigenes .NET-Programm schreiben und dabei einige Klassen aus der .NET-Klassenbibliothek kennen lernen.

ENTWICKLUNGSMÖGLICHKEITEN

SharpDevelop 36

Wer für .NET programmieren möchte, aber vor den Kosten für Microsofts Visual Studio zurückschreckt, der steht nicht vor dem Aus. Es gibt eine Open-Source Alternative: SharpDevelop.

Commandozeilen-Compiler verwenden 40

Auch wer kein Visual Studio oder eine anderen Entwicklungsumgebung für .NET besitzt, kann Programme für .NET schreiben.

Gratis-C#-Compiler für alle 43

Ein kleines C#-Projekt, mit dem auch unter Windows 9.x compiliert werden kann. Ganz recht, hier bauen Sie Ihren eigenen Compiler.

EFFIZIENTE AUFGABENLÖSUNGEN

Windows Forms-Bibliothek nutzen 50

Mit den Windows Forms ist vieles einfach, wie Sie in diesem Beitrag erfahren werden.

Malen mit GDI+ 52

Bei .NET gibt es viel Neues: Das in die Tage gekommene GDI-Interface wurde aufgebohrt, und gemalt wird bei .NET nun mit GDI+.

Laden und Speichern 56

Egal, für welche Aufgabe eine Anwendung gedacht ist - eines muss sie fast immer tun: Daten vom Benutzer entgegen nehmen und speichern.

Datenbankzugriffe mit .NET 60

Daten sind wirklich lästig - in praktisch jeder Anwendung braucht man welche, und immer wollen die auch bearbeitet und gespeichert werden. Mit .NET wird das einfach.

COMPONENTEN UND CONTROLS

Gefunden in der Toolbox: Hilfreiche Komponenten 66

Die C#-Toolbox von Visual Studio enthält nicht nur die Kontrollelemente für Windows Forms, sondern auch noch einen Satz an interessanten Komponenten.

Eine eigene IDE für C# selbst gebaut 72

PropertyGrids und Serialisierung in .NET machen den Ausbau des Mini-C#-Compilers zum vollwertigen Instrument möglich.

Fenster rund machen 76

Mit .NET ist es extrem einfach, Fenster mit unregelmäßigen Formen zu erzeugen

Controls selber machen 81

UserControls zu entwerfen, ist mit C# ein reines Kinderspiel. Die Regeln zum Spiel erfahren Sie in diesem Beitrag.

FORTGESCHRITTENES

Die Script-Engine 84

Bei .NET gibt es eine interessante Möglichkeit, Programme zu übersetzen - mit der Scripting-Engine. Dieser Compiler kann als Scripting-Motor benutzt werden.

Öffentliche Objekte auf fremden Rechnern 90

XML-Webservices sind im Gespräch. Dummerweise weiß praktisch niemand wirklich, was ein Web-Service eigentlich ist.

Die Top-Tipps: C# optimieren 94

Der C#-Compiler ist ein optimierender Compiler, aber darauf sollte man sich nicht allein verlassen. Es ist sogar viel wichtiger, Algorithmen von Hand optimal einzusetzen.

SERVICE

Editorial 3

Helpline 71

Vorschau und Impressum 98

die CD

Komplettes Microsoft .NET Software Development Kit .NET SDK.

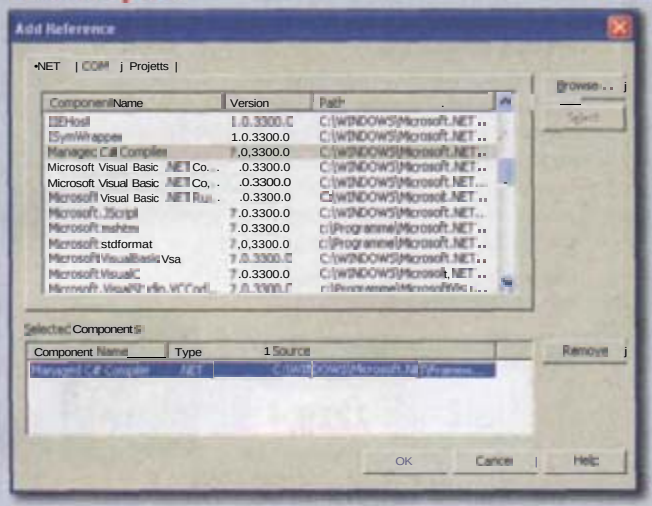
MIT C#- UND VB-COMPILER:

- .NET Redistributable Dateien
- .NET Software Development Kit
- .NET Service Pack 2 (beinhaltet auch SP1)
- .NET Laufzeitumgebung als weitergebbares Setup
- SDK-Beispiele
- Dokumentation

Projekte

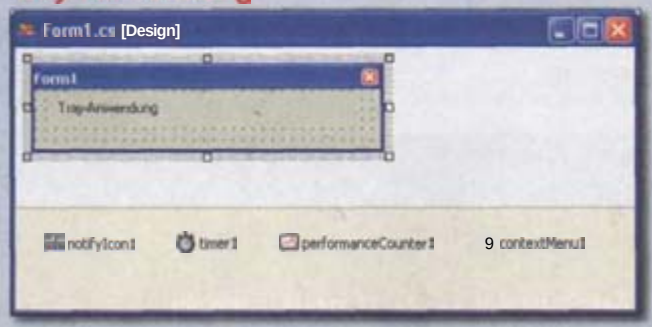
Die Listings zu allen Projekten sind auf der CD, damit Sie schnell loslegen können. Zum Beispiel:

C#-Compiler



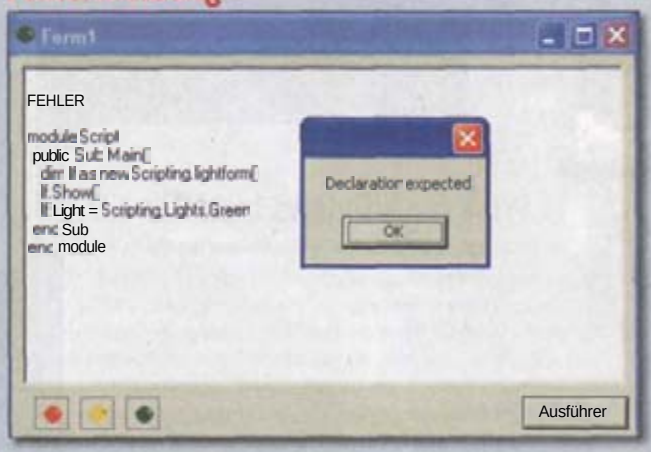
C#-COMPILER selbst gebaut und übesetzt; mit selbst gebauter GUI auch unter Windows 9.x nutzbar.

Tray-Anwendung



EIN REAKTIONSFÄHIGES TRAY-IKON in die Task-Leiste einbauen.

Fehlermeldung



EIGENBAU-COMPILER gibt Fehlermeldung aus.

Grundlegende Übung

DOUBLETEXTBOX-CONTROL entwerfen.

- **Änderungen für Programmierer**
- C# in Aktion - das erste Beispielprogramm
- Eine Tray-Anwendung mit Windows Forms
- Malen mit GD1+
- Scripting in der eigenen Anwendung
- Webdienste verwenden - Google abfragen
- Serialisierung mit .NET
- Datenbankzugriffe mit .NET
- .NET vom Pinguin: SharpDevelop
- C# Compiler für alle - auch unter Windows 98
- Gefunden in der Toolbox: Hilfreiche Komponenten
- Ein GUI für die eigene IDE
- Komische Formen - Fenster einmal anders
- Eigene Controls - Steuerelemente selber machen
- C# optimieren - die Top-Tipps
- Fenster im XP Stil - Ein Application Manifest

ALLE PROGRAMME zur eigenen Erweiterung.

Neues Programmieren

.NET in aller Munde

.NET hat einen recht weitgehenden Einfluss auf die Art und Weise, wie Software entwickelt, eingesetzt, verteilt und verwaltet wird. Und das ist der Grund dafür, dass bei .NET eine solche Verwirrung herrscht, denn .NET ist für praktisch jeden Personenkreis etwas anderes.

THOMAS WÖLFER

Plötzlich war es da, das erste .NET-Programm: Mit Overnet brachten die Programmierer des eDonkey 2000 File-Sharing-Programms eine Version ins Netz, für deren Betrieb man die '.NET Framework Runtime'-Umgebung herunterladen und installieren musste. Warum musste man das tun - was macht dieses .NET eigentlich? Diese Frage beantwortet der folgende Beitrag.

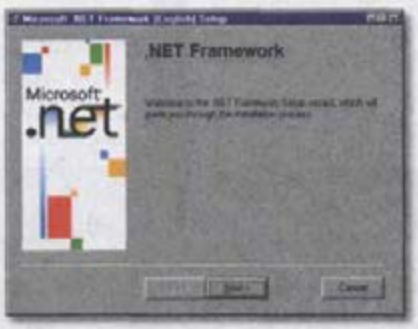
Es handelt es sich bei .NET um eine ganze Menge an Dingen - im Wesentlichen ist es aber so, dass .NET die zukünftige Basis für's Programmieren im Microsoft-Umfeld ist. Allerdings bleiben auch viele ältere Betriebssystemgrundlagen erhalten - sie werden in die neue Sicht der Dinge mit aufgenommen.

Diese neue Basis baut auf den vorhandenen Funktionen von Windows auf und stellt eine Menge an Erweiterungen dafür zur Verfügung. Die Konsequenz daraus ist, dass viele Programme und Programmarten einfacher zur programmieren sind als zuvor - zumindest mit Windows. Dazu zählen vor allem auch Programme, die in irgendeiner Form Netzwerke oder Netzwerkdienstleistungen verwenden oder zur Verfügung stellen sollen. Zum Beispiel also Internet-Clientprogramme, Client- und Server-Dienste: Dazu zählen Anwendungen, die auf einem Webserver ausgeführt werden, aber auch völlig eigenständige Serverprogramme.

Microsoft hatte im Rahmen der .NET-Veröffentlichung leider kein sonderlich gutes Händchen bewiesen: Alle möglichen Dinge, Dienste und Programme wurden seit Mitte 2001 plötzlich mit dem .NET-Label vermarktet - nicht zuletzt auch Windows XP – ohne dass ein Konzept sichtbar wurde. Denn viele dieser Dienste und Programme hatten im Kern zumindest zum Zeitpunkt ihrer Veröffentlichung gar nichts mit .NET zu tun (so wurde das ursprüngliche Windows XP zum Beispiel gar nicht mit der Laufzeitumgebung für .NET ausgeliefert).

• Wo .NET eine Rolle spielt

Das primär mit .NET belabelte Programm - und das mit der vermutlich größten Verbreitung - ist Windows XP: Und das hat, wie gesagt, mit .NET selbst nicht so viel zu tun. Aber das XP Service Pack kam bereits mit der ersten Variante der .NET-Laufzeitbibliothek und stellt damit zumindest für andere Anwendungen die .NET-Laufzeitumgebung zur Verfügung.



DAS .NET FRAMEWORK SETUP installiert nur den redistributable Teil von .NET. Das SDK ist ein separates Download, findet sich aber auch auf der Heft-CD.

Dann gibt es den Windows XP-Server. Der sollte ursprünglich mit Windows XP Professional und XP Home ausgeliefert werden, doch der ursprüngliche Zeitplan wurde geändert, das Programm verschoben. Gegen Ende 2002 kann man aber nun doch mit dem XP-Server rechnen, und zwar unter dem Namen .NET Server.

IM ZUGE DER INSTALLATION werden jede Menge neuer Komponenten auf dem Rechner installiert. Kein Wunder - schließlich handelt es sich um eine komplett neue Betriebssystem-API.

Es gibt verschiedene 'Web Services' genannte Pläne und Produkte – ein recht bekanntes ist zum Beispiel Microsoft Passport. Der Passport-Dienst soll als zentrale Anmeldestelle im Web verwendet werden können:

Einmal bei Passport angemeldet, kann man automatisch an allen zugehörigen Webseiten teilnehmen, und jedem Webseitenbetreiber ist es dabei überlassen, an Passport teilzunehmen.

Schließlich gibt es eine ganze Menge an Programmen, die die .NET-Laufzeitumgebung verwenden – und die Anzahl solcher Programme wird stetig zunehmen, so viel ist sicher. Schließlich ist noch anzumerken, dass es auch bei Li-

DAS .NET FRAMEWORK SDK umfasst über 130 MB. Sie müssen es nicht herunterladen, denn das SDK befindet sich auf der Heft-CD.

nux Projekte gibt, die sich mit .NET beschäftigen.

So kümmert sich das MONO-Projekt beispielsweise um eine Implementierung der .NET-Laufzeitbibliothek für das Open Source-System.

Die .NET Umgebung - eine Kurzbeschreibung

Bei .NET handelt es sich zunächst um ein Laufzeitsystem, das Anwendungsprogramme verwenden können, um vorgefertigte Funktionen daraus nutzen zu können. In der Praxis ist die Sache aber vielschichtiger, .NET ist noch mehr.

Tatsächlich setzt sich .NET aus einer Reihe von Schichten zusammen, die unter dem Begriff CLI (Common Language Infrastructure) zusammengefasst sind.

Zunächst einmal gibt es da die Common Language Runtime (CLR). Die CLR stellt das eigentliche Laufzeitsystem dar, es handelt sich also um die ausführende Instanz:

.NET Programme werden von der CLR - oder besser: innerhalb der CLR ausgeführt. Prinzipiell können CLR-Programme mit praktisch allen Programmiersprachen erzeugt werden, allerdings müssen sich diese natürlich an die Vorgaben der CLR halten.

Dann gibt es die .NET Framework Klassenbibliothek, die von Programmen innerhalb der CLR verwendet werden kann. Diese Bibliothek umfasst einige Tausend Klassen, die einen riesigen Funktionsumfang zur Verfügung stellen. Diese Klassenbibliothek stellt die eigentliche Programmierschnittstelle für Anwendungsprogramme innerhalb der CLR dar. Sie ersetzt mehr oder weniger die Win32 API.

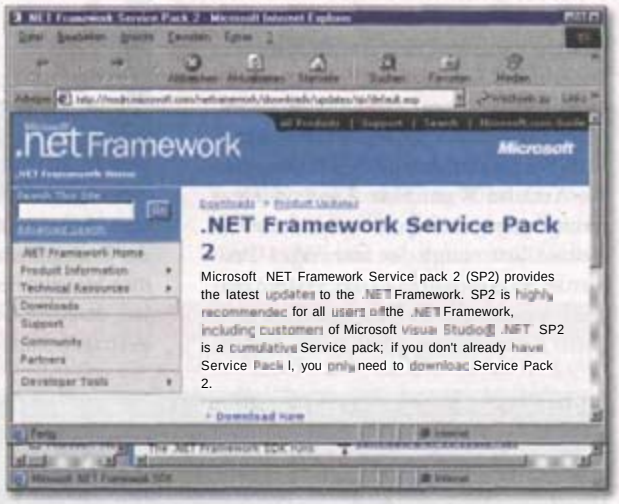
Im .NET-Framework gibt es auch fertige, mitgelieferte Programmiersprachen - und zwar einschließlich der zugehörigen Compiler für diese Sprachen. Von Haus aus werden ein C# Compiler, ein Visual Basic Compiler sowie ein JavaScript-System mitgeliefert.

Was nicht mitgeliefert wird, ist eine Entwicklungsumgebung - wer Software für .NET entwickeln möchte und wem die Kommandozeilen-Compiler nicht komfortabel genug sind, der braucht aber so etwas.

Das Visual Studio .NET von Microsoft enthält eine solche Entwicklungsumgebung. Auch Borland hat bereits eine IDE angekündigt. Ebenso gibt es in Form von ICSharpCode ein Open Source-Entwicklungssystem.

Wer einen C/C++ Compiler nutzen will, der findet ihn im im Visual Studio: Dort gibt es VC++ in Version 7.0: Beim .NET Laufzeitsystem selbst ist kein C++ Compiler dabei.

Ferner handelt es sich bei .NET auch um eine Sammlung an Spezifikationen



WIE DAS IMMER SO IST: Natürlich gibt es auch für's Framework Service-Packs.

FRAMEWORK REDISTRIBUTABLE UND FRAMEWORK SDK - DER UNTERSCHIED

Auf der Heft-CD - und natürlich auf dem Webserver von Microsoft - finden sich zwei unterschiedliche Downloads, die beide mit dem .NET Framework zu tun haben. Das ist zum einen das überschaubar große .NET Framework Redistributable und zum anderen das .NET Framework SDK, das mit etwa 131 MB schon größer ist.

Das Redistributable File wird immer dann benötigt, wenn .NET Framework-Anwendungen ausgeführt werden sollen. Man installiert damit die benötigten Komponenten, um Programme zu starten, die mit Hilfe des SDK entwickelt wurden.

Diese Umgebung kann auf allen Windows-Versionen ab (und einschließlich) Windows 98 installiert werden. Dabei unterstützt das .NET Framework aber nur unter Windows XP Professionell und Windows 2000 Server-Anwendungen, die ASP.NET verwenden - also solche, die einen Webserver zum Betrieb brauchen. Für normale Windows-Programme ist das gleichgültig, und das gilt auch für die meisten Beispielprogramme aus diesem Sonderheft.

Das SDK hingegen enthält zusätzlich alle Komponenten, die man benötigt, um Anwendungen für .NET zu schreiben. Dafür braucht man aber etwas modernere Plattformen: Das SDK kann unter Windows NT 4 mit dem Service Pack 6, unter Windows 2000 mit dem SP 2 und unter Windows XP Professional benutzt werden. Dafür enthält das SDK auch einen großen Satz an Beispielprogrammen und nicht zuletzt die Kommandozeilen-Compiler, mit denen man eigene Programme auch dann übersetzen kann, wenn man keine Entwicklungsumgebung besitzt, wie zum Beispiel Visual Studio.NET.

Auf der CD zu diesem Sonderheft finden Sie aber auch den Quellcode für einen einfache Kommandozeilen-Compiler, den Sie selbst leicht erweitern können - und der im Gegensatz zu den beim SDK mitgelieferten Compilern auch unter Windows 9.x und den anderen vom SDK nicht unterstützten Betriebssystemen läuft. Sie werden verblüfft sein, wie einfach die Implementierung ist.

für verschiedene Dienste und Protokolle - diese sind dafür gedacht, von Dritten implementiert zu werden, damit die Programme und Dienste dieser Drittanbieter einfacher zusammenarbeiten können. In wieweit das angenommen wird - und ob es dann auch wirklich funktioniert - ist eine Frage, die die Zukunft klären wird.

• .NET für Anwender

Im Großen und Ganzen ist es aus Sicht des Anwenders eigentlich völlig gleichgültig, mit welcher Sprache oder für welches System eine Software programmiert wurde. So praktisch eine zentrale Anmeldestelle im Web sein mag, so hilfreich verschiedenste Webdienste sein mögen - für den Anwender ist es hauptsächlich von Interesse, dass diese Dinge funktionieren, nicht aber, mit welchem Entwicklungssystem sie erstellt wurden.

Trotzdem wird .NET auch auf Anwender Einfluss haben - und zwar einfach deshalb, weil das System Entwickler in der Lage versetzt, effizienter zu arbeiten und zuvor nur schwer zu implementierende Dienste anzubieten.

.NET hat darüber hinaus Einfluss auf die Art und Weise, wie Anwender mit Computern arbeiten können, denn das System löst einige der bisherigen Probleme. So ermöglicht es das System beispielsweise mehrere Versionen der gleichen Anwendung parallel zu installieren, ohne dass sich diese Versionen gegenseitig stören. Dabei wird gleich auch noch mit dem, .DLL-Hell genannten, Problem aufgeräumt: Multiple Versionen der gleichen DLL können parallel installiert und verwendet sein. Das Phänomen, dass im Zuge der Installation einer Anwendung eine andere nicht mehr läuft, ist damit in Kürze nur noch Vergangenheit. Zumindest gilt das für Programme, die innerhalb der .NET Umgebung laufen.

Das normale Windows wird mit Erscheinen von .NET nicht einfach verschwinden - auf einem gängigen PC werden also Windows und .NET Programme einfach Seite an Seite betrieben. Der eine Teil davon mit den altbekannten Problemen, der andere hoffentlich nicht mit neuen.

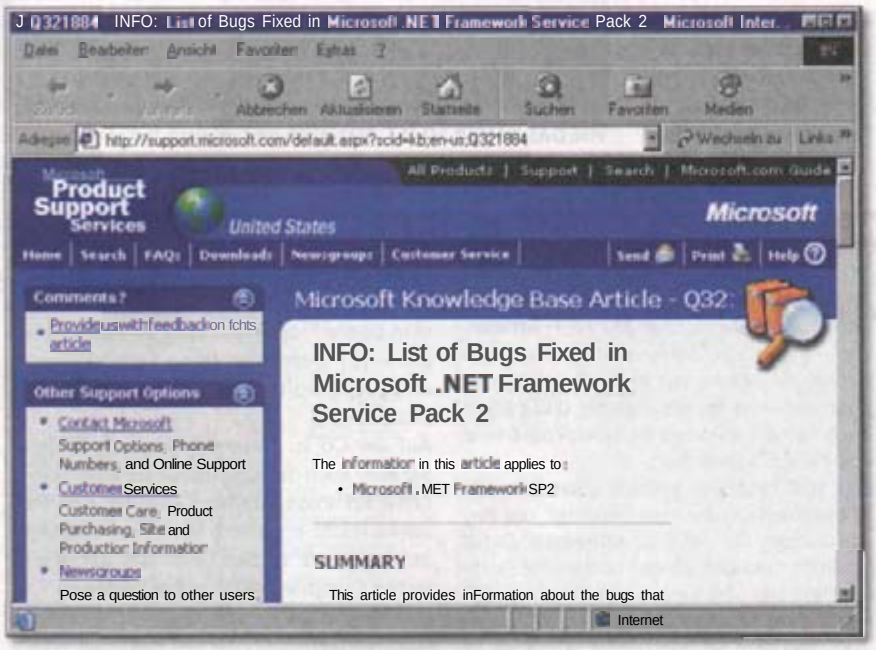
Der Parallelbetrieb läuft völlig transparent ab. Man kann einem :NET-Programm, das unter Windows betrieben wird, von außen nicht ohne weiteres ansehen, dass es ein solches Programm ist: Das Aussehen der Fenster, der Schaltknöpfe und alles übrige ist bei einem .NET Programm nicht anders als bei normalen Windows-Programmen.

DAS FRAMEWORK VERWENDET DIE 'INSTALL-TIME'-Variante der Codegenerierung. Auch wenn der Fortschrittsmelder sagt, man brauche noch '0' Sekunden, sollte man besser etwas warten, denn auf langsamen Rechnern dauert das doch gerne auch ein paar Minuten.

Durch die Installation der .NET-Umgebung auf einem Computer tut sich auf dem System zunächst überhaupt nichts.

Das einzige optische Merkmal für eine installierte .NET-Umgebung sind zwei neue Einträge im Startmenü. Bei Windows XP tauchen diese beiden Einträge zum Beispiel im Menü 'Verwaltung' auf. Dabei handelte sich um den .NET Framework-Assistenten und die .NET Framework-Konfiguration. Um zu verstehen, wofür diese beiden Anwendungen gedacht sind, braucht man ein bisschen Hintergrundwissen über einige Elemente von .NET.

Werden Programmteile mehrfach verwendet, dann liegen diese Teile meist in Form von .DLL-Dateien auf der Festplatte vor. Ganz früher einmal enthielten die Dateien auch tatsächlich direkt ausführbaren Code, der direkt von außen angesprungen wurde. Das ist noch immer möglich, hat sich aber im Laufe der Zeit nachhaltig verändert. Heute sind .DLL-Dateien häufig nur noch das Transportmedium für Objekte, die ihre Dienste und Funktionen zwar auch in Form von Programmcode in der DLL enthalten - die aber nicht direkt angesprungen werden können. Statt dessen erzeugt ein Programm heute ein Objekt vom gewünschten Typ (das geht mit Hilfe von Programmcode aus der DLL) und ersucht dieses dann, einen seiner Dienste oder Funktionen zur Verfügung zu stellen bzw. auszuführen.



NACHDEM IMMER WIEDER EINMAL FEHLER AUFTAUCHEN KÖNNEN, empfiehlt es sich, die passende Webseite bei Microsoft hin und wieder zu konsultieren um zu überprüfen, ob neue Patches vorhanden sind.

Das ist mehr oder minder die Bedeutung von COM, einem der technologischen Vorgänger und Wegbereiter für .NET. Bei .NET nun enthalten DLLs weiterhin eines oder mehrere Objekte, aber auch noch eine ganze Menge an weiteren Informationen. Dazu gehört zum Beispiel eine Information über die Abhängigkeiten der Objekte in der DLL: Welche anderen DLLs in welcher Version werden benötigt? Darüber hinaus ist es so, dass der Programmcode innerhalb einer DLL nicht alles darf, was er will - der Besitzer des Rechners hat Möglichkeiten, dem Code innerhalb der DLL bestimmte Schranken vorzugeben. (Wurde eine DLL nicht in ihrer Funktion eingeschränkt, dann nennt man das 'der DLL wird vertraut').

DIE .NET KONFIGURATIONSTOOLS werden im Verwaltungs-Teil der Systemsteuerung installiert.

Im Rahmen der Versionitis und der Sicherheit ist es ein besonderer Teil der Vertrauensstellung, in welcher Weise die DLLs anderen Programmcode ausführen dürfen, und in welcher Art und Weise sie Programmcode von anderen Orten kopieren können: Sei das nun aus dem Internet oder aber von einem anderen Rechner im LAN.

Ferner können diese .NET-DLLs auch Ressourcen (Texte, Bilder, Bitmaps, Menüs) enthalten und zwar gleich mehrere Versionen dieser Ressourcen für mehrere Sprachen.

Im Sprachgebrauch von .NET heißen solche DLLs, die verwaltet werden können, denen ein bestimmtes Maß an Vertrauen zugesprochen wird und die die anderen angesprochenen Informationen enthalten, 'Assemblies'.

Diese Assemblies können sich entweder ganz normal auf der Festplatte befinden, oder sie befinden sich an einem speziellen Speicherort, dem Assembly-Cache. Der Assembly-Cache hat vielfältige Aufgaben. Am wichtigsten dabei ist, dass Assemblies im Assembly-Cache schneller geladen werden können.

Die beiden Programme, die im Zuge der .NET Installation auf dem Rechner installiert werden, dienen der Verwaltung diese Assemblies und zugeordneten Funktionen, wie dem 'Reparieren' von Anwendungen, die von .NET verwaltet werden.

• .NET für Programmierer

Den größten Einfluss hat .NET zunächst auf jeden Fall auf die Programmierer - zumindest auf diejenigen, die sich darauf einlassen. Dafür gibt es eine Menge guter Gründe, und das ist nicht zuletzt die Verfügbarkeit einer riesigen Klassenbibliothek.

Sie löst nicht nur die bisherigen Windows-Funktionen ab, sondern stellt eine ganze Reihe an zusätzlicher Funktionalität zur Verfügung. Und das nicht in Form der über Jahre gewachsenen Win32 API mit all ihren zusätzlichen

DIE ENTWICKLUNG DER APIS: ASSEMBLER, C, C++, CLR

Das .NET-Framework ist eigentlich eine nur zeitgemäße Weiterentwicklung der Art und Weise, wie ein Anwendungsprogramm mit dem Betriebssystem kommuniziert. Zumindest dann, wenn man die Reihe der am weitesten verbreiteten Systeme betrachtet - und damit sind MS-DOS, Windows 3, Windows 95 und XP gemeint.

Unter MS-DOS bestand die Betriebssystem-API im Wesentlichen aus einem Satz an Assembler-Routinen, die so interessante Dinge tun konnten, wie ein Zeichen von der Tastatur zu lesen. Wer damals interessante Dinge programmieren wollte, der musste einfach Assembler nehmen - andere Möglichkeiten gab es nicht.

Mit der Zeit verbreitete sich aber recht schnell auch unter DOS die Sprache C. Die hatte den unter anderem den Vorteil, dass Sie mit einer standardisierten Funktionsbibliothek kam, die schon deutlich abstraktere Funktionen zur Verfügung stellte: Plötzlich konnte man Strings 'einfach so' verketten und aus Dateien lesen. Wer allerdings direkt in den Bildschirmspeicher schreiben wollte oder Grafikkarten zu verwenden hatte, der brauchte weiterhin Assembler - oder beschaffte sich von C aufrufbare, kommerziell erhältliche Bibliotheken.

Damit wurde C schnell zum Standard für Programme unter MS-DOS. Alle Versionen von MS-DOS blieben dabei von der API her aber gleich - egal welche neuen Funktionen das System zur Verfügung stellte, es handelte sich immer um per Assembly aufrufbare Funktionen.

Das änderte sich mit Windows - denn hier wurde plötzlich eine 'C'-API geboten: Die Windows-Funktionen waren plötzlich 'C'-Funktionen, die direkt von C aus aufgerufen werden konnten. Das blieb auch eine ganze Zeit so - eigentlich bis heute, denn die 'Original' C-APIs sind noch immer in Windows vorhanden. Allerdings verbesserte sich auch die Sprache C - und aus vielen C-Programmierern wurden C++ Programmierer. Von C++ aus war aber das Programmieren gegen die Windows 'C'-API eher unschön: Dieses Problem wurde mit Klassenbibliotheken gelöst, die die 'C'-API und C++-Klassen kapselten.

Daraufhin änderte sich aber Windows erneut und binäre 'Objekte' wurden eingeführt. Neue Funktionalität in Windows wurde seit Windows 95 zwar nicht flächendeckend, aber immer öfter in Form von COM-Objekten zur Verfügung gestellt. Mit ein wenig Unterstützung waren COM-Objekte von C++ aus relativ leicht zu Nutzen - Betriebssystem und Sprache waren wieder gleichauf. Das Problem: Im Gegensatz zu früher wurde es auch für Anwendungsprogrammierer immer wichtiger, selber binäre Objekte zu erzeugen, und das war (und ist) mit der COM Library eine Qual.

Bei der CLR ist die Objektorientierung nun auch in die Basis-API eingezogen: Alles ist mehr oder minder grundsätzlich ein Objekt - und kann auch sehr einfach von C++ aus benutzt werden. Andere Gründe der CLR Architektur machen C++ aber zu einer etwas problematischen Sprache - und so kann man als kommende 'Standardsprache' für Windows .NET Anwendungen wohl schon heute *Ci* identifizieren: Das wäre dann das erste Mal, dass das (MS)-Betriebssystem von Haus aus eine Sprache mitbringt die auch tatsächlich perfekt zur API des Systems passt.

und drangebastelten Elementen wie OpenGL, DirectX und dergleichen, sondern eben in Form einer einheitlich aufgebauten Bibliothek.

Dabei kann man als Programmierer entweder die gewohnte Sprache weiterverwenden (zunächst gilt das für Visual Basic und Visual C++, weitere werden aber folgen) oder man erlernt mit der Sprache C# eine neue, C++ und Java ähnliche Sprache. Das hat alles seine Vor- und Nachteile, was Sie an vielen Beispielprogrammen in dieser Ausgabe erkennen werden.

Zunächst aber noch ein paar Bemerkungen zu den grundlegenden Vorteilen und Neuigkeiten, die man durch .NET erhält:

- Riesige Klassenbibliothek
- GUI-Anwendungen, Web-Anwendungen, XML-Dienste, Windows Dienste und viele weitere Ziel-Projekte sind möglich
- Einfache Entwicklung von exportierbaren Klassen, leichte Wiederverwendung von selbst definierten Objekten
- Scripting-Support in der eigenen Anwendung
- Compiler-Support auf dem Zielsystem
- Optimierter Code für unterschiedliche Zielsysteme
- Vereinfachungen bei Setup und Deployment
- Cross-Language, Cross-Prozess-Debugging
- Memory Managment mit Garbage Collection
- Type-Checking, Range-Checking durch die Laufzeitumgebung

• Die Klassenbibliothek

Die Klassenbibliothek des .NET Frameworks ist das zunächst eindruckvollste Novum.

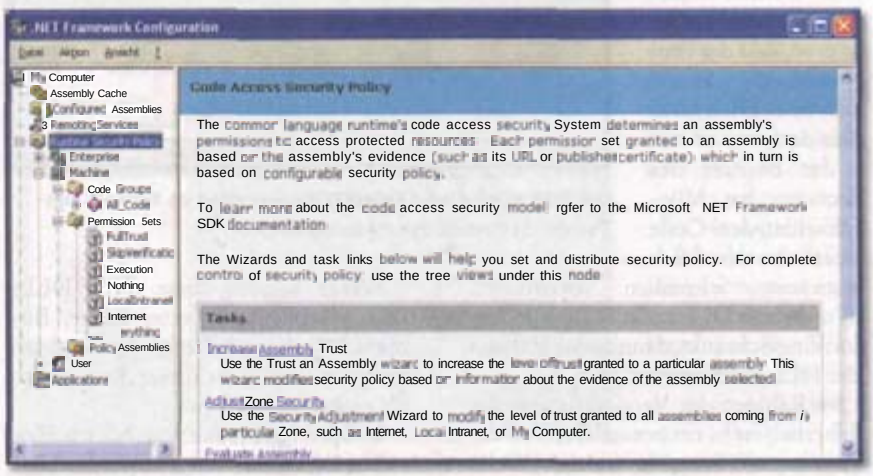
Einige Tausend Klassen bieten die neue, komplett objektorientierte API für Windows. Dabei enthält die Klassenbibliothek Objekte für ganz normale Aufgaben unter Windows - also zum Beispiel für's Öffnen von Fenstern oder dem Lesen und Schreiben von Dateien. Es finden sich aber auch weitere Klassen für die Kommunikation im Internet, für die Lokalisierung von Anwendungen für mehrere Sprachen und so weiter und so fort:

In Tausenden von Klassen kann man schon eine ganze Menge an Funktionalität unterbringen, wie Sie im Folgenden sehen.

• Anwendungstypen

Bisherige Klassenbibliotheken, wie zum Beispiel die MFC, deckten immer nur einen kleinen Teil der Funktionen ab, die unter Windows möglich sind. So war es mit der MFC beispielsweise nicht ohne weiteres möglich, einen Windows Service zu programmieren.

Mit der .NET Framework Klassenbibliothek können nicht nur alle 'klassischen' Windows-Anwendungen, wie eben Dienste, Kommandozeilen-Programme und GUI Anwendungen, entwickelt werden, sondern es werden obendrein noch eine ganze Reihe an zusätzlichen Anwendungstypen möglich gemacht. Dazu zählen zum Beispiel Web-Anwendungen oder XML-Dienste. Für die wichtigsten davon finden Sie Beispielprogramme in den Beiträgen dieses Sonderheftes.



MIT DER .NET FRAMEWORK CONFIGURATION lassen sich installierte Assemblies vom Administrator konfigurieren.

• Wiederverwendung

Wer bisher mit C oder C++ programmiert hat und versuchte, seine Klassen oder Funktionen an dritter Stelle weiter zu verwenden, der tat dies in Form von öffentlichen Header-Dateien und Bibliotheken. Dabei konnten die Bibliotheken entweder in Quellcodeform oder in Form vorkompilierter Libraries vorliegen. Das hat seinen Liebreiz - schließlich will man meist selbst seinen eigenen Code wiederverwenden und hat daher auch nur wenig Probleme dabei das 'richtige' Werkzeug, sprich: den richtigen Linker, zu verwenden.

Etwas modernere Varianten der Wiederverwendung fanden unter Windows bisher in Form von COM-Objekten statt. Dabei wurden die Objekte in ihrer binären Form wiederverwendet: Type-Libraries enthielten dabei die benötigten Informationen für Compiler und Entwicklungssystem. Dabei war die Programmierung eines COM-Objektes aber etwas problematisch: Ohne die Hilfe von Klassenbibliotheken - also rein auf den COM-Libraries basierende Objekte - waren nur sehr mühevoll zu programmieren.

Im Rahmen der Wiederverwertung von Code kann man bei .NET problemlos von der Fortsetzung von COM mit allen Mitteln sprechen. Jedes, wirklich jedes mit einer .NET-Sprache entwickelte Objekt, kann direkt wiederverwendet werden - und zwar in seiner binären Form. Dazu muss der Programmierer rein gar nichts tun: Alle benötigten Informationen werden von CLR Compilern automatisch erzeugt. Dabei ist auch die verwendete Sprache gleichgültig:

Verwendet der Compiler das CTS (Common Type System) – und das muss er, wenn er Code für die CLR erzeugen will - dann entstehen beim Übersetzen automatisch neue wiederverwendbare Komponenten. Diese Komponenten sind dann in jeder .NET-Sprache verwendbar: Egal, ob es sich dabei um VB.NET VC++.NET oder Cobol.NET handelt.

Das ist nicht nur äußerlich nützlich sondern auch extrem effizient: Nachdem zusätzliche Arbeiten zum Erzeugen neuer Komponenten einfach entfallen, ist die Wiederverwendung von Code ein Kinderspiel.

• Skripting

Nachdem man alle selbst programmierten Objekte so leicht wiederverwenden kann, ist es auch kein Wunder, dass es

extrem einfach ist, die eigene Anwendung um Skripting-Möglichkeiten zu erweitern. Dazu verwendet man einfach den bei .NET mitgelieferten Skripting-Mechanismus, der von Haus aus mit JavaScript und VB Programmen klarkommt.

Das versetzt den Programmierer in die Lage, seine eigene Anwendung so zu erweitern, dass Anwender einer Anwendung eigene Methoden und Programme innerhalb dieser Anwendung schreiben können.

Mit ein wenig zusätzlichem Code können die Anwender selbst dann ebenfalls die für das Programm implementierten Klassen verwenden: Erweiterbare Anwendungen sind damit so einfach zu haben wie noch nie. Das Beste daran: Anders als beim Automation-Mechanismus muss auch hier kein eigener Code geschrieben werden - die Skripte der Anwender benutzen einfach die Objekte, die der Programmierer der Anwendung auch verwendet hat.

• Compiler-Support auf dem Zielsystem

Die Sprachen VB.NET, C# und JavaScript sind Teil der .NET-Laufzeitumgebung. Das bedeutet, dass jeder Anwender, der diese Laufzeitumgebung installiert hat, auch über die entsprechenden Compiler verfügt. Zwar stehen diese nicht in Form von Kommandozeilen-Compilern bereit (die kann man aber in Form des .NET SDKs ebenfalls umsonst erhalten), sondern nur in Form von Compiler-Services, die per Programmcode angeworfen werden müssen.

Wer aber schon immer ein Interesse daran hatte, auf dem Zielsystem beim Anwender Code zu übersetzen, kann das mit .NET nun tun, denn auf dem Zielsystem befinden sich die gleichen Compiler, die auch für die Anwendungsentwicklung verwendet werden.

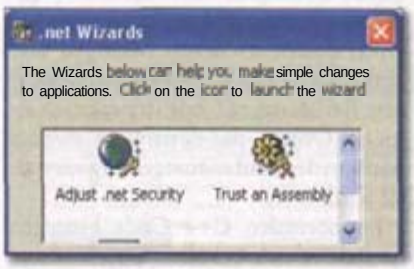
• Code für unterschiedliche Zielsysteme

Beim Übersetzen von .NET-Anwendungen wird kein nativer Code erzeugt - zumindest nicht, wenn man sich nicht ausdrücklich darum bemüht. Statt dessen erzeugen die Compiler MSIL-Code - und dieser wird erst auf dem Zielsystem beim Laden der Anwendung in nativen Code umgewandelt. In der Praxis bedeutet das, dass die Eigenschaften des Zielsystems deutlich besser ausgenutzt werden können:

Liegt eine Pentium II CPU vor, so können spezielle Optimierungen für diese CPU vorgenommen werden - ist es ein P4, dann kann die Optimierung entsprechend ausfallen.

Ein ebenso interessanter Nebeneffekt ist dabei die Tatsache, dass die Codegeneratoren beim Laden der Anwendung auch weitere Informationen über das Zielsystem berücksichtigen: Es können zum Beispiel unterschiedliche Optimierungen in Abhängigkeit von der vorhandenen Speichermenge durchgeführt werden, und auch die bereits geladenen Module können in die Optimierung eingehen. An dieser Stelle ist noch viel von zukünftigen Versionen des .NET Frameworks zu erwarten: Langsamer wird er Code dadurch sicherlich nicht.

Wer sich daran stört, dass die Übersetzung jedes Mal beim Laden der Anwendung stattfindet, der kann natürlich entweder weiterhin nativen Code für ein festes Zielsystem erzeugen oder aber



DIE .NET WIZARDS helfen bei allgemeinen Aufgaben im Zusammenhang mit .NET. von Haus aus wird man die Wizards aber nicht benötigen da .NET sinnvolle Default-Policies für den Einsatz von Anwendungen bereitstellt.

man verwendet ein Feature, das sich Setup-Compilation nennt. Dabei wird die Anwendung während der Installation in nativen Code umgewandet: Der Vorteil dabei ist, dass die Übersetzung bei jedem Startvorgang entfällt, und dennoch die Zielumgebung beim Erzeugen des Binary berücksichtigt werden kann.

• Vereinfachung bei Setup und Deployment

.NET-Programme sind von Haus aus autark: Sofern der Programmierer nichts Gegensteiliges unternimmt, kann eine .NET-Anwendung einfach per XCOPY installiert werden. Verzeichnis anlegen - Dateien reinkopieren, fertig.

Registrierungsmaßnahmen mit der Registry sind nicht notwendig und auch Konfigurationsdateien können (im Prinzip) einfach mitkopiert werden.

• Cross-Language, Cross Process Debugging

Die Software-Entwicklung als solche bietet ebenfalls deutlich mehr Möglichkeiten: Anwendungen können sich über mehrere Sprachen, mehrere Prozesse und mehrere Rechner verteilen: Der Source-Level-Debugger macht dabei alles mit: Egal ob ein Kontext-Switch zwischen Sprachen, Prozessen oder Rechnern stattfinden - es kann debuggt werden.

• Memory Managment mit Garbage Collection

Beim .NET-Framework wird die Speicherverwaltung durch die Laufzeitumgebung gehandhabt. Das wird alle C- und C++-Programmierer zunächst zu Tode erschrecken, aber der Schreck ist eher ungerechtfertigt. Die Laufzeitumgebung ist nicht nur sehr effizient, sondern in vielen Fällen sogar schneller als die nativen Memory Management Mechanismen, die in C und C++ Programmen verwendet werden.

Wer seine Zweifel hat, der sollte es ausprobieren: So schlimm, wie man befürchtet, ist der Garbage Collector nicht - und das schreibt ein Programmierer, der seit 15 Jahren mit C und C++ gearbeitet hat: Das deutlich angenehmere Programmieren, das der Garbage Collector ermöglicht, wiegt die Nachteile mehr als auf - und wer in speziellen Fällen trotzdem nicht auf eigene Mechanismen verzichten will, der wird nicht davon abgehalten, 'unmanaged' C++-Code an den passenden Stellen zu verwenden.

Alles in allem bietet .NET nicht nur eine extrem komfortable Klassenbibliothek und – dank des Garbage Collectors – ein neues Programmiergefühl, sondern es macht in der Praxis auch wirklich Spaß, damit zu arbeiten. © UR

Änderungen für Programmierer

VB und VC++ und COM im Blick

C# ist die eigentliche .NET-Sprache. Aber man kann für .NET auch mit Visual Basic und mit VC++ programmieren, damit nicht alle erarbeiteten Werte verloren gehen. Allerdings muss man einige Änderungen beachten.

THOMAS WÖLFER

Wer bisher mit VC++ programmiert hat, der muss mit der neuen Version von VC++ nicht unbedingt irgend etwas Neues lernen, denn VC++ Version 7 bietet sich in zwei Modi an: Entweder man programmiert für die CLR - dann schreibt man Managed Code - oder man lässt es bleiben. Im zweiten Fall ändert sich nichts, abgesehen davon, dass das neue VC++ bessere Kompatibilität zum Standard aufweist, dass es neue MFC-Klassen gibt (wenn auch nicht viele) und einige andere Details zu beachten sind. Im Wesentlichen ist 'Unmanaged Code' mit VC++ nichts anderes, als die logische Fortsetzung von VC++ 6.0 - nur eben ohne die CLR und ohne eine Möglichkeit, die .NET Klassenbibliothek zu verwenden.

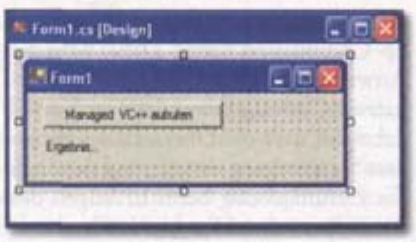
• Visual C++ zugemischt

Das wird schlagartig anders, wenn man Managed Code einsetzt: Hier bekommt man Zugriff auf die Klassenbibliothek von .NET mit ihren zahllosen Möglichkeiten und allen ihren Methoden und Controls. Der wesentliche Unterschied ist jetzt aber, dass man bei Managed Code auch in C++ innerhalb des Garbage Collectors von .NET operiert. Das bedeutet im Klartext: Objekte brauchen nicht länger freigegeben zu werden, denn das besorgt der Garbage Collector der Laufzeitumgebung.

Das ist zwar ganz praktisch, aber auf alte Projekte nicht ohne weiteres anzuwenden - denn die sind natürlich voll von Destruktor-Aufrufen und ähnlichen Dingen, die die CLR nur stören. Das war auch den Programmierern bei Microsoft klar, und so gibt es auch die Möglichkeit, Managed und Unmanaged Code zu mischen - sofern man auf einige Dinge achtet.

Das Wichtigste dabei ist das so genannte 'Boxing'. Dazu muss man die Unterschiede zwischen Managed und Unmanaged Code etwas besser verstehen. Bei Managed Code ist praktisch alles ein Objekt, das dynamisch erzeugt und von der Laufzeitumgebung verwaltet wird.

Im normalen C++ Code hingegen handelt es sich bei allen Objekten um Objekte, die vom Programmierer verwaltet werden - und außerdem gibt es jede Menge atomarer Typen, die nicht im Heap sondern auf dem Stack alloziert werden. Das widerspricht natürlich massiv der Arbeitsweise im Managed



DER AUFRUF VON C++-CODE durch andere Sprachen ist ganz einfach, wenn es sich beim C++ Code um Managed Code handelt, der innerhalb der CLR läuft.

Code – und braucht eine Lösung, wenn man beide Arten von Code zusammenbringen will.

Diese Lösung trägt den schönen Namen Boxing - und wird mit separaten neuen Schlüsselworten vom Compiler direkt unterstützt. Im Wesentlichen passiert dabei das Folgende:

Wird ein Objekte aus der CLR in Unmanaged Code überführt, dann wird dazu eine Kopie der Daten des Objekts erzeugt, die nicht von der CLR verwaltet werden. Damit kann der Unmanaged Code dann tun, was ihm beliebt. Wird ein Objekt hingegen aus Unmanaged Code an Managed Code übergeben, so bekommt es zuvor eine Hülle, die vom Managed Code aus verwaltet werden kann: In diesem Fall wird also eine Box um das Objekt erzeugt, im anderen Fall wird Sie zuvor entfernt.

DAS ERGEBNIS eines Funktionsaufrufs in eine C++-Klasse aus einer C#-Klasse.

Damit wäre das Wesentliche für den C++-Programmierer erläutert - wenn da nicht die bösen Details wären. Die betreffen vor allem die Art und Weise, wie man neue Anwendungen anlegt - und haben ihre Tücken.

Um es vorweg zu sagen: Wer hofft, in Zukunft weiter mit VC++ und MFC-Anwendungen programmieren zu können, der wird enttäuscht: Man kann zwar weiterhin MFC-Anwendungen erzeugen - sogar mit einigen neuen Features - aber nicht für die CLR.

Wer also weiterhin auf die MFC schwört, der bleibt beim alten Modell und bei der alten Plattform - Zugriff auf die Klassen, die Ressourcen-Verwaltung und andere Mechanismen der CLR gibt es nicht.

Das ist auch völlig klar, denn C++ als solches ist nun mal keine Sprache, die sich sonderlich gut mit einem Garbage Collector und einem gemeinsamen Typensystem verträgt - man denke dabei nur an die Verwendung von Templates, die alles andere als kompatibel zum gemeinsamen Typensystem sind.

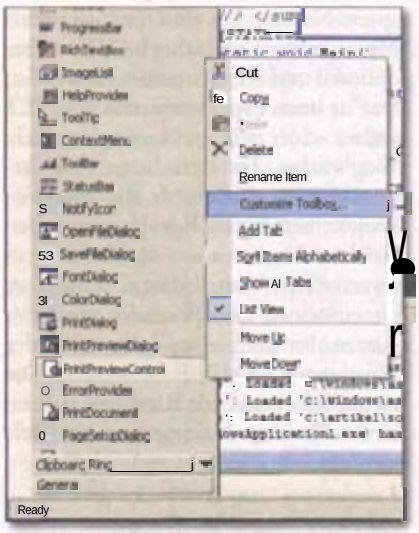
Wer allerdings weiterhin lokale Elemente und Libraries in VC++ implementieren will, hat Glück: COM-Objekte, die mit C++ implementiert wurden, können dank der InterOp-Funktionen der CLR mehr oder minder transparent innerhalb der CLR benutzt werden. Und Bibliotheken, wie etwa mathematische oder wissenschaftliche Funktionensammlungen, lassen sich mit dem Boxing-Mechanismus leicht weiter verwenden - alles was man tun muss, ist einen Satz an Wrapper-Klassen oder -Funktionen zu implementieren.

Für VC++ Programmierer, die bei der Sprache bleiben wollen, ist die Sache also durchwachsen: Zwar kann man mehr oder minder einfach so weitermachen wie bisher - nur fallen dann viele Möglichkeiten weg. Auf der anderen Seite gibt es die Sprache C#, mit der die Anwendungsentwicklung nicht wesentlich aufwändiger ist, als mit VC++ und MFC - und die obendrein für einen VC++-Programmierer extrem leicht zu erlernen ist. Der Umstieg wird einem von Microsoft also recht schmackhaft gemacht – zumindest, wenn es darum geht, neue Anwendung unter Verwendung alter Komponenten zu schreiben.

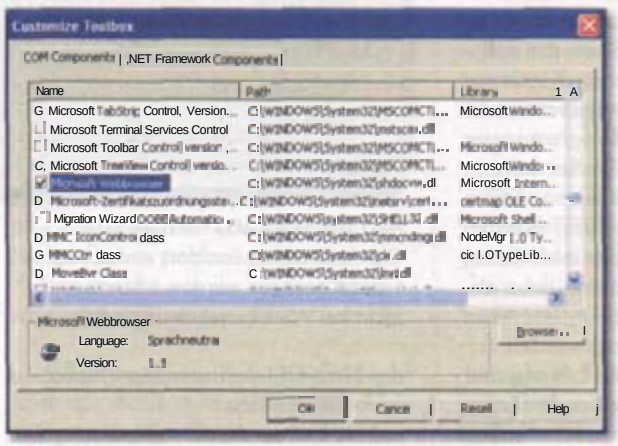
• Visual Basic

Bei Visual Basic stellt sich ein anderes Bild dar. Der Visual-Basic-Programmierer erhält praktisch die gleichen Möglichkeiten wie der C#-Programmierer in .NET - allerdings zu einem erheblichen Preis: Die Sprache als solche, die Art und Weise, wie Komponenten verwendet werden und so ungefähr alles andere, was mit der Entwicklung von VB-Anwendungen zusammenhängt, hat sich bei der in Visual Studio 7 enthaltenen VB-Version verändert. Man programmiert zwar weiterhin mit einer Sprache, die Visual Basic heißt, nur hat die eben sehr viele Eigenschaften verloren, während neue hinzugekommen sind. Massives Umgewöhnen ist angesagt.

Auf alle Details der Änderungen im Einzelnen einzugehen wäre Thema für ein eigenes VB-.NET-Heft. Daher soll die folgende Aussage zunächst ausreichen: Sie können auch für .NET weiterhin mit VB arbeiten, allerdings ist es von der Art und Weise, wie Sie bisher mit dieser Sprache gearbeitet haben abhängig, wie viel Sinn das macht. Unter Umständen ist die benötigte Umgewöhnungszeit nicht geringer, wenn Sie gleich den Umstieg von VB6 auf C# wagen, statt von VB6 auf VB7 umzusteigen.



COM-OBJEKTE kann man einfach dadurch benutzen, indem man die ToolBox der IDE um das gewünschte Objekt erweitert.



UM EIN COM OBJEKT zu verwenden, braucht man es nur in der Liste der zur Verfügung stehenden Objekte markieren: Die Entwicklungsumgebung erzeugt den kompletten benötigten Wrapper-Code automatisch.

• Ein Beispiel für Managed Code in VC++

Wie bereits erwähnt, kann man sehr wohl weiterhin in VC++ programmieren – und das auch für die CLR und unter Verwendung der CLR, einschließlich der angebotenen Klassen aus dem .NET-Framework. Eine solche Klasse ist zum Beispiel die String-Klasse aus dem Namespace System.

Im Folgenden finden Sie ein Beispielprogramm, das das Vorgehen beschreibt. Das Programm tut nur wenig Sinnvolles - zeigt dafür aber die Konzepte sehr deutlich auf. Es wird Folgendes implementiert: Das Beispielprogramm besteht aus zwei Projekten, wobei das eine Projekt eine normale C#-Anwendung ist, die im Wesentlichen aus einem Fenster mit einem Button und einem Label Control besteht. Das zweite Projekt enthält eine mit VC++ programmierte Klasse namens CSample. Diese Klasse enthält nur eine einzige Methode namens Get() - und die liefert einen String der im Konstruktor der Klasse alloziert wird. Dabei wird kein C++-String (wie zum Beispiel die MFC C-String-Klasse) verwendet, sondern das VC++-Programm benutzt die String-Klasse aus der CLR.

Wird nun im C#-Programm auf den Button gedrückt, dann erzeugt das Programm eine neue Instanz eines CSample Objekts (das in C++ implementiert wurde), erfragt bei der Instanz den Text und trägt selbigen im Label-Control ein. So einfach das klingen mag und so einfach das in reinem C# ist - bei einer Mischung von C# und C++ muss man da schon einige Dinge berücksichtigen.

Zunächst einmal erzeugen Sie ein ganz normales C#-Projekt vom Typ 'Windows Application'. Das Projekt enthält dann ganz automatisch eine Windows Forms-Klasse, die Sie mit dem GUI-Editor von C# bearbeiten können. Für diese Testanwendung reicht es aus, einen Button und ein Label-Control per Drag&Drop auf dem Fenster zu platzieren.

Das war auf Seiten von C# zunächst alles. Im nächsten Schritt legen Sie ein neues Projekt an, das Sie aber in der gleichen Solution (im gleichen Arbeitsblatt) wie das C#-Projekt speichern. Beim neuen Projekt wählen Sie als Projektart 'Managed C++ Class Library' - es wird also ein VC++ Projekt, das mit Managed Code arbeitet. Als Projektnamen geben Sie 'mc_vc' (für Managed Code, Visual C) an.

Ist das Projekt erzeugt, so enthält es bereits mehrere CPP-Dateien. Die einzige von Interesse ist dabei die Datei 'mc_vc.cpp'. In dieser Datei implementieren Sie die Csample-Klasse.

Hier schlagen nun direkt jede Menge neue oder zumindest ungewohnte Dinge zu.

Die normale Implementierung der beschriebenen CSample-Klasse würde folgendermaßen aussehen - wenn man keinen Managed Code und obendrein die MFC String-Klasse verwenden würde:

```
#include <stdafx.h>
class CSample
{
private:
  CString m_str;
public:
  CSample()
  {
    m_str ="hallo welt";
  }

  CString Get()
  {
    return m_str;
  }
};
```

Das geht so aber nicht mehr. Zunächst einmal muss man sich darüber klar werden, dass die Cstring-Klasse nicht die Klasse ist, die verwendet werden soll - statt dessen soll ja die String-Klasse aus der .NET-Klassenbibliothek verwendet werden. Das stellt den Programmierer aber vor ein Problem: Woher kommt dann die Klassen-Definition, denn schließlich gibt es für die .NET-Klassen keine Header-Files die man inkludieren könnte. Das ist etwas dumm - denn ohne einen Prototyp für die Klasse kann auch keine Instanz erzeugt werden.

Dafür gibt es aber eine Lösung und zwar die neue Präprozessor-Directive *#using*. Diese funktioniert recht ähnlich wie die *#import*-Direktive die mit VC++ 5.0 für die einfachere Verwendung von COM Objekten aus DLLs eingeführt wurde.

Bei *#using* gibt man den Namen einer .NET Assembly an - das ist meist eine DLL. Diese Assembly enthält Typinformationen, Metadaten und Code. Auf Basis dieser Informationen baut der Compiler dann den benötigten Prototyp automatisch. Die .NET String-Klasse befindet sich beispielsweise in der .NET system.dll Assembly. Man benötigt also ein Statement der Form:

```
ttusing<system.dll>
```

Danach können alle Typen aus dieser Assembly auch in VC++ verwendet werden - so auch die string-Klasse.

Als Nächstes benötigt man für die zu implementierende Klasse einen Namespace. Namespaces sind für C++ nichts Neues, waren aber bisher bei VC++ eher optional und wurden selten verwendet. Das ist beim Programmieren für .NET anders - hier geht an Namespaces kein Weg vorbei. Im einfachsten Fall verwendet man einfach für alle an einem Projekt beteiligten Komponenten den gleichen Namespace - dann hat man zwar keinen Vorteil davon, kann aber die einzelnen Komponenten einfacher nutzen. Im Beispielsprojekt wurde der Namespace SAMPLE verwendet. Die zu implementierende Klasse muss dann innerhalb des Namespaces definiert werden:

```
namespace SAMPLE
{
   // hier kommt die klasse
}
```

(Der Namespace innerhalb des C# Projekts wurde übrigens auch SAMPLE genannt - Sie können das anhand des auf der Heft-CD vorliegenden Quellcodes leicht überprüfen.)

NEU ZUR TOOLBOX hinzugefügte Objekte können von dort einfach auf Fenster gezogen werden: Allerdings nicht bei VC++-Projekten mit MFC.

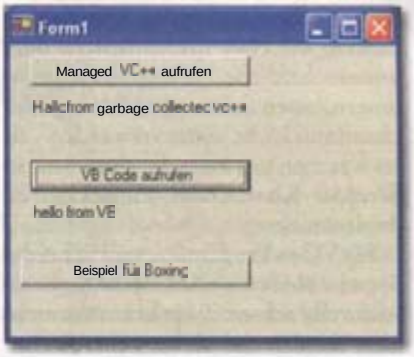
Nun kommt die dritte Neuerung: Die zu implementierende Klasse muss mit einem Schlüsselwort versehen werden, das die Sichtbarkeit der Klasse nach außen definiert. Nachdem die Klasse von außen - also vom C#-Code aus — vollständig sichtbar sein soll, braucht man in diesem Fall das Schlüsselwort *public*. Vor dem Klassennamen, wohlgemerkt:

```
public class CSample
{
   // die klasse kommt hier
};
```

Auch das ist noch nicht ausreichend: Die Klasse soll ja eine Klasse innerhalb des Managed Codes sein - das heißt, der Garbage Collector soll sich um Instanzen dieser Klasse bemühen. Diese Tatsache muss dem C++ Compiler ebenfalls mitgeteilt werden, und das geht mit dem neuen Schlüsselwort __gc. (Die beiden Underscores zeigen dabei an, dass es sich um eine Spracherweiterung handelt, die Buchstaben stehen dabei für garbage collected). Der Beginn der Klasse sieht dann folgendermaßen aus:

```
public __gc class CSample
{
   // die klasse kommt hier
};
```

Innerhalb der Klasse selbst passiert zunächst nichts Außergewöhnliches. Es gibt einen public-Teil mit dem Konstruktor und der Get()-Methode, sowie einen private-Teil für die Instanz des Strings.



AUCH VISUAL BASIC CODE kann man einfach mit ins Projekt aufnehmen: Darum hat die CLR ja den Namen 'Common Language Runtime'.

Dass sich hier nichts tut, scheint aber nur auf den ersten Blick der Fall zu sein. Die Member-Variable vom Typ String muss das folgende Aussehen haben:

```
String* m_pstr;
```

Dabei ist besonders der [*] zu berücksichtigen: Es handelt sich also nicht um eine Instanz vom Typ String sondern um einen Zeiger auf einen String. Das Wichtige dabei: Sie können keine Instanz vom Typ String definieren. Der Versuch, die Variable wie folgt zu definieren, wird vom Compiler mit einer Fehlermeldung quittiert:

```
String m_str;
```

Der Grund ist, dass es sich hier eben um Managed Code handelt - und im Rahmen von Managed Code müssen alle Objekte dynamisch alloziert werden: Sie arbeiten also grundsätzlich mit Zeigern auf Objekte.

Würde die String-Instanz nicht dynamisch alloziert, dann könnte sie auch nicht vom Garbage Collector verwaltet werden, und darum lässt der Compiler dies erst gar nicht zu.

Das ist allerdings sehr gewöhnungsbedürftig - und so gibt es extra eine spe-

zielle Fehlermeldung des Compilers: *Did you forget a *?*.

Die Tatsache, dass es sich bei *m_pstr* um einen Zeiger handelt, hat natürlich auch Konsequenzen für den Konstruktor. Der muss nun den String dynamisch erzeugen, statt ihn wie im ursprünglichen Beispiel einfach mit einem Wert zu belegen:

```
CSample()
{
  m_pstr = new String
  ➥("Hallo from garbage
  ➥collected vc++");
}
```

Und ebenso ist auch die Get()-Methode betroffen - die liefert schließlich ebenfalls einen Zeiger:

```
String* Get()
{
  return m_pstr;
}
```

Damit ist die GSample-Klasse abgeschlossen. Das sollte Sie aber eigentlich verblüffen - denn es gibt nirgends einen Destruktor, der *m_pstr* wieder freigibt. Nachdem der String dynamisch alloziert wurde, würde man normalerweise in etwa folgenden Code erwarten:

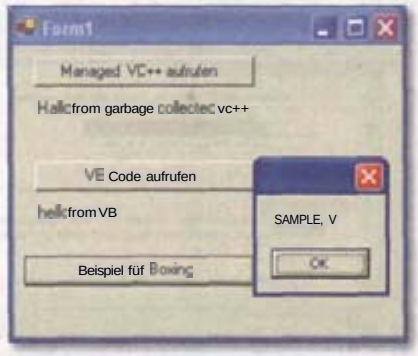
```
~CSample()
{
  delete m_pstr;
}
```

Das ist aber genau einer der Vorteile, mit denen Sie es bei Managed Code zu tun haben: Es ist schlicht und ergreifend nicht notwendig, den String wieder freizugeben. Der Garbage Collector der Laufzeitumgebung kümmert sich darum. Um genau zu sein: Es ist sogar falsch, den String zu zerstören und ein entsprechender Aufruf von *operator delete* wird deshalb auch vom Compiler bemängelt.

Die vollständige Implementierung der CSample Klasse sieht also folgendermaßen aus:

```
ttusing<system.dll>
namespace SAMPLE
{
  public __gc class CSample
  {
  private:
    String* m_pstr;
  public:
    CSample()
    {
      m_pstr = new String
      ➥("Hallo from garbage
      ➥collected vc++");
    }

    String* Get()
    {
      return m_pstr;
    }
  };
}
```

Sie können das VC++-Projekt nun übersetzen. Dabei passieren ebenfalls einige neue und interessante Dinge - die Details würden allerdings etwas zu weit führen. Im Wesentlichen ist es wichtig zu wissen, dass die Csample-Klasse deutlich einfach wiederverwendbar geworden ist, als sie das mit einer normalen C++-Implementierung wäre. Aufgrund der Tatsache, dass es sich um ein Managed Code-Projekt handelt, kann die CSample-Klasse ab sofort genau so weiterverwendet werden, wie das beider bereits verwendeten String-Klasse der Fall ist. Dazu braucht man nur noch die vom Projekt erzeugte DLL - ein Header-File oder eine separate Link-Library sind nicht notwendig. Das werden Sie gleich selbst ausprobieren. Dazu wechseln Sie zurück in das C#-Projekt.



HIER WIRD EIN 'GEBOXTES' VALUE-OBJEKT ANGEZEIGT: Um genau zu sein, handelt es sich um das Ergebnis der *ToString()*-Methode, die auf das Objekt angewendet wurde.

Hier öffnen Sie im Solution-Explorer dann den Ast *References* unter der *Windows Application 1*. Dort sehen Sie verschiedene Einträge, die beim Anlegen des Projekts automatisch vorgenommen worden. Einer diese Einträge hat den Namen *System* - und das ist genau die gleiche system.dll, die Sie schon im VC++-Projekt eingebunden haben, um den String-Typ verwenden zu können. Wie man sieht, erfolgt diese Einbindung beim C# Compiler anders als beim C++ Compiler: In C++ binden Sie die Referenzen auf Assemblies im Quellcode per *#using* ein, während das bei C# direkt in der Entwicklungsumgebung geht. (Die Entwicklungsumgebung ihrerseits erzeugt im Wesentlichen einen Kommandozeilen-Befehl für den C# Compiler - man kann Referenzen also auch ohne die IDE verwenden.)

Genau eine solche Referenz fügen Sie nun hinzu. Dazu klicken Sie mit der rechten Maustaste auf *References* und wählen dann den Befehl *Add Reference*. Der öffnet einen Dialog, auf dem Sie sich zwischen .NET-Komponenten, COM-Komponenten und Projek-

VALUE-OBJEKTE verhalten sich innerhalb der CLR so wie Variable, die in C auf dem Stack alloziert werden: Sie werden nicht von der Laufzeitumgebung verwaltet.

ten entscheiden müssen. Nachdem die CSample Klasse aus einem eigenen Projekt stammt, wählen Sie den Reiter *Projects* und erhalten eine Liste aller Projekte aus Ihrer Solution. Im Wesentlichen findet sich da das VC++-Projekt - denn andere, dritte Projekte existieren ja noch nicht.

Dieses Projekt wählen Sie aus und schließen den Dialog. Sie werden dann mit einem zusätzlichen Eintrag namens *mc_vc* in den Referenzen des C# Projekts belohnt.

Das war alles: Sie können die CSample Klasse nun verwenden. Um das auch tatsächlich zu tun, doppelklicken Sie auf den Button Ihres Fensters im C#-Programm. Die Entwicklungsumgebung fügt dann einen Button-Click Eventhandler für den Button ein. Der trägt den Namen *button1_click()*.

Hier erzeugen Sie nun einfach eine neue CSample Instanz und weisen dann dem 'Label' Control den Text der Instanz zu:

```
CSample B = new CSample();
this.label1.Text = s.Get();
```

Auch hier braucht es keinen Destruktor, und auch hier müssen Sie das Objekt nicht selbst wieder zerstören: Der Garbage Collector kümmert sich automatisch darum. Mit anderen Worten: Sie können nun auch das C#-Projekt übersetzen und ausprobieren:

Voila - Sie haben nun ein Programm das sich aus VC++ Code und C# Code zusammensetzt, wobei der C# Code ein

C++ Objekt einfach so benutzt, als wäre es schon immer vorhanden gewesen.

• Boxing - und unboxing

Nun ist das alles gut und schön - aber dummerweise will man innerhalb von C++ nicht immer mit Zeigern auf Objekte arbeiten. Bei komplexen Objekten wird das in normalem Code zwar schon immer der Fall gewesen sein, anders ist das aber bei Typen, die ihrer Art nach eher atomarer Natur sind: Das gilt zum Beispiel für einfache Punkte, die zum Beispiel wie folgt definiert sein können und für geometrische Operationen verwendet werden:

```
struct Point
{
   double x;
   double y;
};
```

Hier - und in praktisch allen anderen Fällen, bei denen structs zum Zuge kommen, will man nahezu immer, dass diese Elemente eben keine Objekte sind, die von der CLR verwaltet werden. Genau das ist auch möglich. Wie man derlei Dinge im C++ Code verwendet zeigt das folgende Beispiel.

Grundsätzlich ist es so das die CLR zwischen zwei Typen unterscheidet: Ein Element ist entweder ein object Type - dann wird es als Objekt verwaltet - oder ein value Type. Value Types werden nicht von der CLR verwaltet, sondern funktionieren wie Variable, die beim herkömmlichen C auf dem Stack abgelegt werden.

Damit man damit arbeiten kann, ist es aber notwendig, zwischen object Types und value Types konvertieren zu können: Man will einen value Type unter Umständen manchmal eben doch als Objekte betrachten können - in anderen Fällen will man das nicht.

Dazu dient das, boxing und unboxing, genannte Verfahren. Beim boxing wird ein value Type in einen object Type umgewandelt. Dazu wird ein Objekt erzeugt, und die Daten des value Types werden dann im Objekt abgelegt. Mit diesem Objekt kann dann weitergearbeitet werden. Beim unboxing wird das wieder rückgängig gemacht.

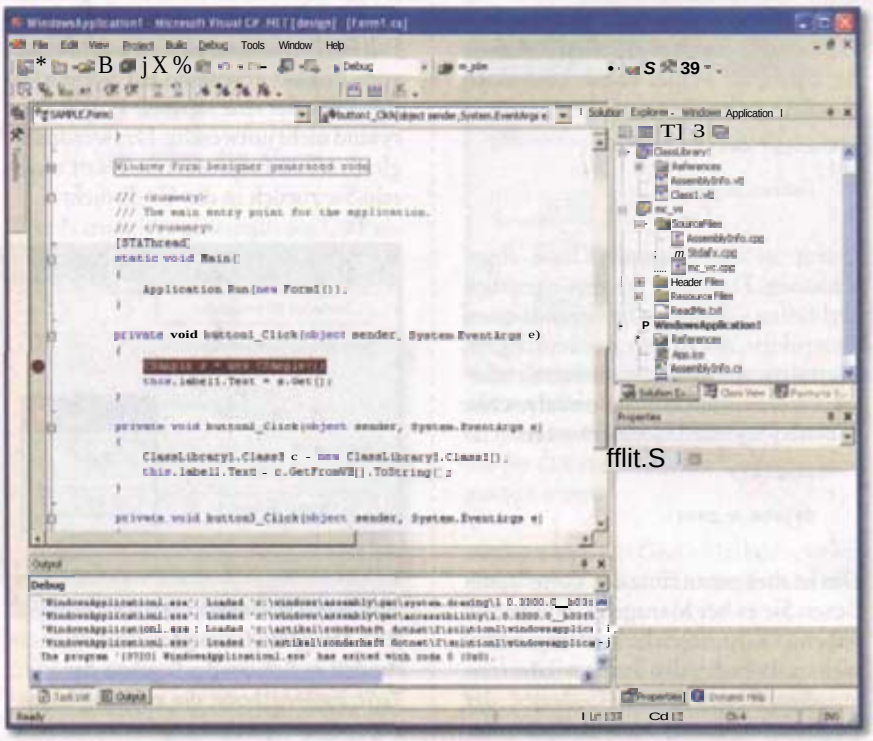
Im Beispiel wird hierzu im C++ Code zunächst ein value Type definiert. Dabei handelt es sich einfachum eine Struktur, die per Schlüsselwort *public*, öffentlich gemacht wird und per Schlüsselwort *__value* als value Type markiert ist:

```
public __value struct V
{
   double d;
};
```

Diese Stuktur soll nun einmal als value Type und ein anderes Mal al object Type betrachtet werden. Dazu wird eine Klasse definiert, mit der man eine solche Struktur manipulieren kann. Die Klasse initialisiert eine Variable V mit dem Wert 42 und bietet dann verschiedene Zugriffsmöglichkeiten an: Die Variable kann zum einen als value und zum anderen als Objekt erfragt werden, außerdem ist es möglich, die Variable als Object mit einem neuem Wert zu belegen. Mit anderen Worten: Die Behandlung davon ist sowohl als value Type wie auch als object Type möglich:

```
public __gc class CBoxSample
{
private:
V m_d;

public:
  CBoxSample()
  {
    m_d.d = 42;
  }

  Object *GetValueAsObject()
  {
    return __box( m_d);
  }

  double GetValue()
  {
    return m_d.d;
  }
  void Set( Object* o)
  {
    m_d = *dynamic_cast
    <__box V*>(o);
  }
};
```

Der Konstruktor sieht hier noch völlig normal aus: Das Member *m_d.d* wird mit dem Wert *42* belegt. Die Methode *GetValueAsObject()* ist hingegen schon etwas verwirrend. Die Methode liefert einen Zeiger auf Object zurück - dabei handelt es sich um die Object-Basisklasse der .NET-Framework-Klassenbibliothek. Dazu muss der Wert aber in ein Objekt konvertiert werden, denn schließlich wurde ja zuvor explizit gesagt, dass es sich bei *struct V* um einen value Typ handelt (*__value*). Das geht mit dem für VC++ neuen Schlüsselwort *__box*.



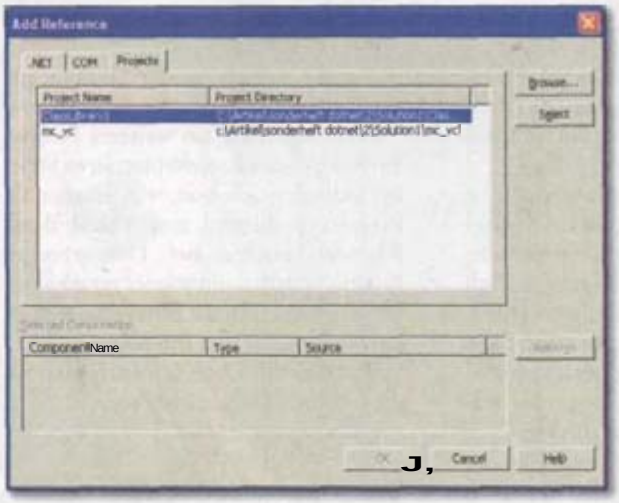
DAS KOMPLETTE BEISPIELPROJEKT besteht aus einer C#-Anwendung, die sowohl eine C++ Klasse als auch eine VB Klasse verwendet.

Die Methode *GetValue()* ist wieder völlig normal und sieht genau so aus, wie man das erwarten würde. Anders ist das wieder mit *Set()* - hier wird die Sache deutlich komplizierter.

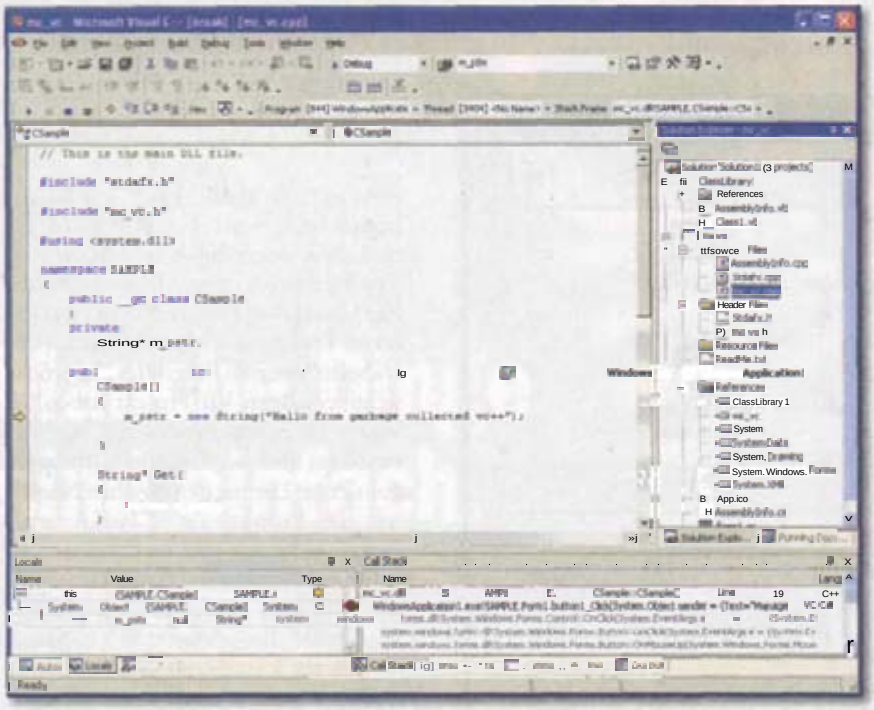
Die *Set()*-Methode bekommt einen Parameter vom Typ *Object** - sie kann also mit einem Object Type aufgerufen werden. Die Methode soll aber das *m_d* Member neu belegen, und das bedeutet, dass es notwendig ist, das Objekt in seine value-Repräsentation zurückzuwandeln. Nun würde man vielleicht ein *__unbox*-Schlüsselwort erwarten, das dies leistet: Das gibt es aber nicht.

Statt dessen verwendet man normale C++ Cast-Operatoren, um die Umwandlung durchzuführen. Dabei gelten die üblichen Regeln: Verwendet man den richtigen Cast, und ist der Cast nicht möglich, dann führt das zu einem Fehler. Im Beispiel wurde einfach ein dynamic_cast verwendet: Der Object* Zeiger wird zunächst in einen ___box V* Zeiger konvertiert und dann dereferenziert. Daraus erhält man dann den im Objekt übergebenen Wert.

Innerhalb des C#Codes kann man das ausprobieren. Im Beispiel wurde das mit einem dritten Button und einem entsprechenden Event-Handler gelöst. Der sieht wie Listing 1 aus und wird anschließend erläutert.



REFERENZEN MUSS MAN DAUERND HINZUFÜGEN: Für den C++-Programmierer ersetzt das in etwa die Verwendung von Header-Files und Libraries.



NATÜRLICH KANN MAN AUCH mit dem Debugger im Single-Step Modus durch die beteiligten Klassen hindurchsteppen: Dass sich dabei die verwendete Sprache ändert, ist völlig gleichgültig.

LISTING 1:

```
private void button3_Click(object sender, System.
➥EventArgs e)
{
  CBoxSample b = new CBoxSample();
  String B = b.GetValueAsObject() .ToString();
  MessageBox.Show( s);
  MessageBox.Show( b.GetValue().ToString());

  V v;
  v.d = 50.0;

  b.Set( v as object);
  s = b.GetValueAsObject().ToString();
  MessageBox.Show( s);
  MessageBox.Show( b.GetValue().ToString());
}
```

Zunächst wird ein passenden CboxSample-Objekt erzeugt. Dieses Objekt wird dann nach seinem Value gefragt - allerdings als Objekt. Selbiges Objekt wird dann in einen String konvertiert. Das ist eine grundsätzlich zur Verfügung stehende Methode bei allen von Object abgeleiteten Klassen. Die Default-Implementierung liefert dabei einen String, der den Klassen-Namen des Objekts enthält. Dieser wird dann mit *MessageBox.Show()* angezeigt - und im Beispiel führt das dazu, dass der Text 'SAMPLE.V dargestellt wird: Schließlich ist das der Name der Klasse, die man erhalten hat.

Der zweite Aufruf von *MessageBox.Show()* zeigt dann den Wert an: Dazu wird die GetValue()-Methode verwendet, die den Wert als value liefert. Dieser ist ein double, aber auch doubles können mit *ToString()* in einen Text umgewandelt werden: Deshalb erscheint dann eine Meldungsbox mit der passenden Zahl.

Im nächsten Schritt wird eine Instanz von *V* auf dem Stack angelegt: Hier ist zu beachten, dass es sich um das erste Mal im Beispiel handelt, bei dem eine Variable nicht mit *new* erzeugt wurde: Das geht nur deshalb, weil V explizit als value Type angelegt wurde.

V wird dann als object übergeben - und danach erneut angezeigt: Das Resultat ist, wie man erwarten würde, ein ähnliches: Zunächst wird der Name der Struktur, dann der neue Wert (also 50) angezeigt.

... und was ist mit COM?

Soweit so gut. Eine weitere Frage bleibt aber (noch) ungeklärt: Was tun, wenn Sie COM-Komponenten mit VC++ programmiert haben und diese weiterverwenden möchten? Auch das ist verhältnismäßig transparent möglich.

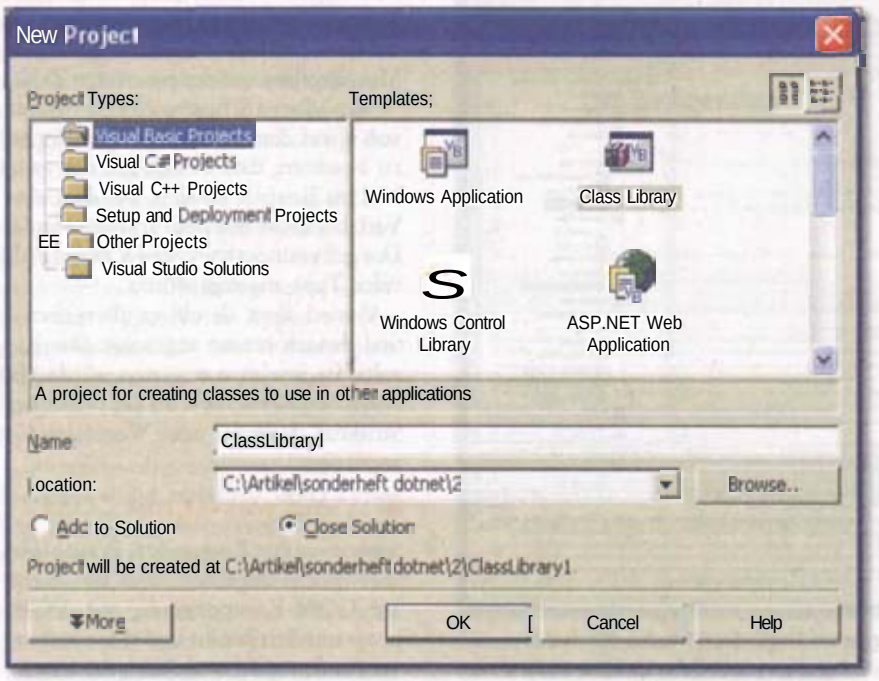
Am einfachsten können Sie COM-Objekte - und zwar sowohl selbst programmierte als auch solche die im System vorliegen - wiederverwenden, indem Sie diese Objekte über die Werkzeugleiste von Visual Studio zugänglich machen. Dazu klicken Sie mit der rechten Maustaste auf die *Toolbox* und wählen den Befehl *Customize Toolbox*. Daraufhin erscheint eine Dialogbox, die es Ihnen ermöglicht aus .NET- und COM-Komponenten auszuwählen.

Sie können nun zum Beispiel eine COM-Komponente wie den Microsoft Webbrowser auswählen - dazu müssen Sie einfach nur die Option neben der ge-

wünschten Komponente anklicken. Danach schließen Sie den Dialog: Die Komponente erscheint dann am unteren Rand der Toolbox.

Nun können Sie diese Komponenten einfach aus der Toolbox auf ein Fenster ziehen: Die IDE erzeugt dann passende Wrapper-Klassen die automatisch ins Projekt mit eingebunden werden. Im C#-Quellcode benutzen Sie dann einfach die neue Komponente: Im Fall des WebBrowsers etwa, indem Sie dessen *Navigate()*-Methode aufrufen.

Man kann also sagen, dass sich an der Verwendung von COM Objekten auch innerhalb von .NET nichts getan hat: Es ist weder schwieriger noch einfacher geworden, solche Objekte zu benutzen: Das wird alle Programmierer erleichtern, die bisher stark auf COM und verwandte Technologien gesetzt haben.



ES IST PROBLEMLOS MÖGLICH, mehrere Projekte mit unterschiedlichen Sprachen innerhalb einer 'Lösung' (Solution) anzulegen. Die Entwicklungsumgebung kümmert sich dabei um die auftretenden Abhängigkeiten.

• VB kommt ins Spiel

Abschließend soll das bisherige Beispielprojekt noch um eine Klasse erweitert werden, die in Visual Basic implementiert wird. Dabei wird auch nochmals auf die Namespaces eingegangen: Diesmal mit einem alternativen Verfahren.

Bei der Erweiterung des Programms durch Visual Basic Code passiert nichts grundlegend Neues: Das C# Window erhält zunächst einen weiteren Button und ein weiteres Label Control. Beide ziehen Sie einfach aus der Toolbox auf das Fenster.

Dann legen Sie ein weiteres Projekt an. Auch dieses Projekt platzieren Sie in der aktuellen Solution, wählen aber als Projekttyp diesmal eine Visual Basic Klassenbibliothek aus. Das erzeugte Projekt enthält dann wieder verschiedene Files - aber nur die Datei class1.vb ist davon zunächst von Interesse.

Wenn Sie die Datei öffnen, finden Sie die folgenden Statements vor:

```
Public Class Class1
End Class
```

Es liegt also zunächst eine Klasse namens Class1 vor: Den Namen der Klasse können Sie natürlich ändern - im Beispielprogramm passierte das aber nicht, sodass die Klasse auch im Quellcode auf der Heft-CD weiterhin den Namen Class1 trägt.

Diese VB-Klasse erweitern Sie nun einfach um eine Funktion namens GetFromVB mit dem folgenden Aussehen:

```
Public Function GetFromVB()
   GetFromVB = „hello from VB"
End Function
```

Sie müssen hier weder Namespaces beachten noch müssen irgendwelche Referenzen eingefügt werden: Anders als beim VC++ Projekt, sind die benötigten Referenzen bereits durch die Entwicklungsumgebung eingefügt worden. Der Namespace wird im Beispiel absichtlich nicht gesetzt, damit Sie die Auswirkungen davon sehen können.

Damit ist die Implementierung der zugegebenermaßen etwas simplen VB-Klasse abgeschlossen: Sie können das Visual Basic Projekt nun übersetzen.

Dann wechseln Sie wieder zurück ins C#-Projekt und gehen den bereits bekannten Weg: Mit Hilfe der rechten Maustaste fügen Sie eine neue Referenz auf das soeben übersetzte VC-Projekt zum C#-Projekt hinzu. Nun doppelklicken Sie auf den neuen Button und landen dadurch im Event-Handler für den entsprechenden Button-Click (Listing 2). Diese füllen Sie nun aus - und

LISTING 2:

```
private void button2_Click
➥(object sender, System.Even
➥tArgs e)
{
}
```

zwar auf eine ähnliche Art, wie Sie das bereits beim ersten Event-Handler getan haben: Sie erzeugen eine Instanz vom Typ der gewünschten Klasse – in diesem Fall Class1 – und tragen dann den gelieferten Text im neuen Label-Control ein. Dabei ist aber nun Folgendes zu berücksichtigen: Beim VB-Projekt haben Sie keinen Namespace verwendet, und darum kam dort der Default-Namespace zum Zuge. Der hat den gleichen Namen wie das Projekt, also ClassLibrary1. Nachdem Sie sich aber im Code des C#-Projekts befinden und dieses seinerseits innerhalb des Namespaces SAMPLE liegt, muss die gewünschte VB-Klasse mit Ihrem kompletten Namen angesprochen werden - und er enthält auch den Namesapce. Der benötigte Code sieht also wie Listing 3 aus:

LISTING 3:

```
ClassLibrary1.Class1 c = new
➥ClassLibrary1.Class1();
this.label2.Text = c.Get
➥FromVB().ToString();
```

Sie wissen nun, was beim Versuch des Wiederverwendens von Komponenten, die Sie mit Ihrer bisherigen Sprache entwickelt haben, erwartet - und welche neuen Möglichkeiten sich bieten; egal, ob Sie dabei mit managed VC++, mit VB, oder mit C# arbeiten: Die Nutzung der CLR und der zugehörigen Klassenbibliothek lohnt sich. © UR

C# - die .NET Sprache

Das scharfe C

Die eigentliche .NET Sprache ist C# (C-sharp gesprochen). Und wer ein wenig Hintergrundwissen mit C oder C++ hat, wird sich schnell einarbeiten können.

THOMAS WÖLFER

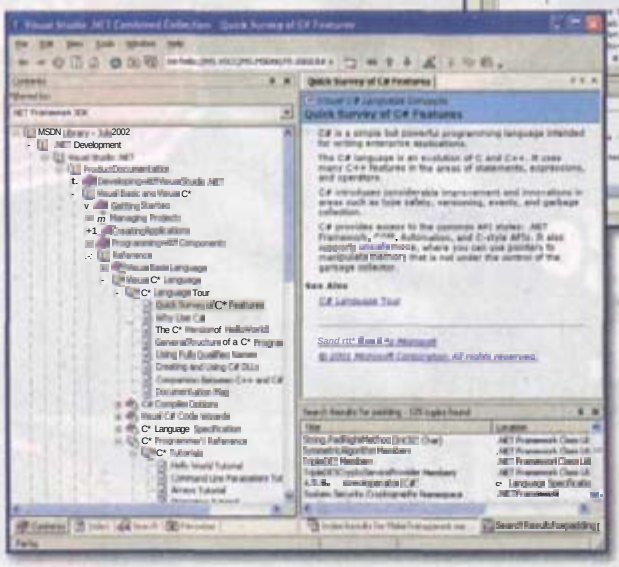
Grundsätzlich ist es fast mit jeder Sprache möglich, .NET Programme zu schreiben: Wie man schon unter Windows 3.1 Programme auch in Fortran und Programme für Win32 auch in Cobol schreiben konnte, ist das bei .NET nicht anders. Aber am effizientesten programmiert man für .NET in C#. Deshalb sind hier alle wichtigen Grundlagen rund um C# beschrieben.

C# ist eine einfache, aber mächtige Sprache, die aus C und C++ hervorgegangen ist. Die Sprache verwendet eine ganze Reihe an Statements, Ausdrücken und Operatoren, die auch in C und C++ üblich sind.

Darüber hinaus ist C# an verschiedenen Stellen typensicherer als seine Vorgänger, hat ein eigenes Modell für die Behandlung von Ereignissen und unterstützt die .NET Garbage Collection direkt im Sprachumfang.

Dabei bietet C# Zugriff auf verschiedene APIs. Sowohl die .NET-Klassenbibliothek, als auch COM, Automation und 'C' Interfaces werden unterstützt.

Darüber hinaus gibt es einen unsicheren (unsafe) Modus, in dem direkt Pointer verwendet werden können. Es kann also auch auf Speicher zugegriffen werden, der nicht vorn Garbage Collector kontrolliert wird. Existierender Code kann von C# aus aufgerufen und verwendet werden. Besonders einfach ist das bei COM-Objekten, aber auch ganz normale C-Funktionen, die in DLLs abgelegt sind, sind erreichbar.



DIE DOKUMENTATION VON VS enthält auch eine umfangreichen Teil, der die Spache C# beschreibt.

Als erstes Beispiel hier das 'Hello World'-Programm in der C# Variante:

LISTING 1:

```
// Ein Hello-World Programm
dass HelloWorld
{
  static void Main()
  {
    System.Console.writeLine
    ➥("Hello World from C#");
  }
}
```

In diesem kleinen Beispiel sind bereits einige der wichtigsten Elemente von C# zu erkennen.

C# - Kommentare

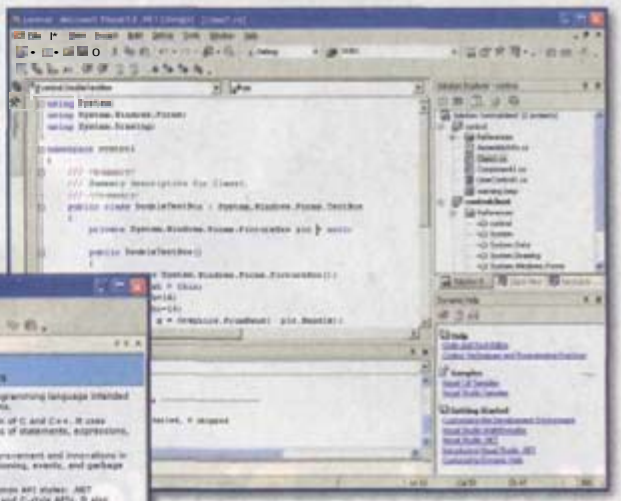
C# unterstützt zwei Arten von Kommentaren im Quellcode. Egal an welcher Stelle einer Zeile ein doppelter Slash eingesetzt wird - der komplette folgende Text ist ein Kommentar. Außerdem gibt es auch die 'C'-Kommentare. Dabei wird alles, was zwischen den Zeichen /* und */ platziert wird, als Kommentar angesehen.

Die Methode 'Main'

Jedes C# Programm muss die Methode Main() enthalten. Diese Methode ist der Ein- und Austrittspunkt jedes C#-Programms: Hier beginnt das Programm, und dort endet es auch. Die Methode Main ist eine statische Methode, die sich innerhalb einer Klasse (oder einer Struktur) befinden muss – im Falkes Beispiels befindet sich die Methode in der Klasse mit dem Namen 'HelloWorld'.

'Main' kann auf drei Arten definiert werden. Entweder ist die Funktion void, oder sie liefert einen Integer. Drittens ist es noch möglich, die Methode so zu definieren, dass sie Argumente übergeben bekommt - das sind dann die Argumente, die an das Programm beim Start übergeben wurden. (static int Main(string[] args))

Bei den Kommandozeilen-Argumenten gibt es in C# einen kleinen Unterschied zu C und C++: Die Argumente enthalten nicht den Namen des laufenden Programms. (Der ist natürlich trotzdem ermittelbar - nur verwendet man dazu in .NET andere Methoden als in C++.)



Ctt **IST DIE .NET-SPRACHE:** Man kann natürlich auch andere Sprachen verwenden - aber das ist eher eine Sache für Leute, die 16bit Windows-Anwendungen in Fortran programmiert haben.

Die Eingabe - und die Ausgabe

C# Programme verwenden für die Ein- und Ausgabe die Dienste der .NET-Laufzeitumgebung. Diese werden in Form der .NET-Klassenbibliothek zur Verfügung gestellt. Im Beispielprogramm sieht man das in der folgenden Zeile:

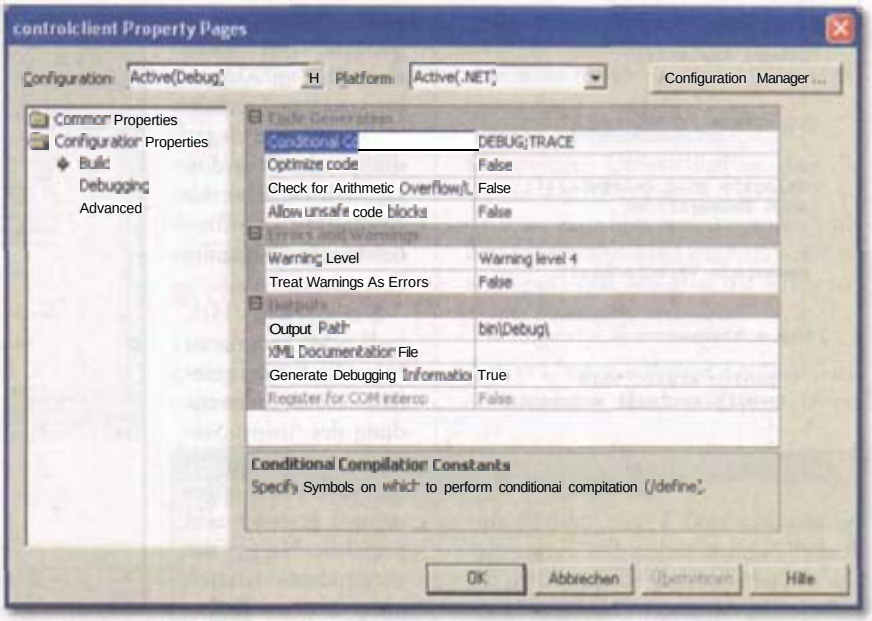
```
System.Console.WriteLine(
➥"Hello World from C#");
```

Hier wird die Methode WriteLine der 'Console'-Klasse verwendet. Diese Methode gibt den übergebenen String auf der Konsole aus. Die 'Console'-Klasse befindet sich im Namespace 'System'. Die komplette .NET-Klassenbibliothek ist in Namespaces aufgeteilt. Will man Objekte aus einem Namespace häufiger verwenden und daher den Namespace nicht immer mit angeben, so kann man das mit Hilfe des 'using'-Statements tun:

```
using System;
```

Nach der Verwendung dieses Statements - im Beispiel im Zusammenhang mit dem Namespace 'System' - kann man alle Klassen aus diesem Namespace direkt verwenden, ohne ihn extra spezifizieren zu müssen. Man kann also den folgenden Ausdruck hinschreiben:

```
Console.WriteLine(
➥"Hello World from C#");
```



DER C# COMPILER hat viel weniger Optionen, als man das von C++ Compilern gewohnt ist. Ein Präprozessor fehlt mehr oder minder ganz.

• Programmstrukturen in C#

Ein C# Programm besteht aus einer oder mehreren Dateien, wobei alle Dateien einen oder mehrere Namespaces enthalten können. Ein einzelner Namespace kann über mehrere Dateien verteilt werden.

Jeder Namespace kann Typen enthalten, wobei ein Typ entweder eine Klasse, eine Struktur, ein Interface, eine Enumeration oder ein Delegate sein kann. Außerdem kann ein Namespace andere Namespaces beinhalten - Namespaces können also verschachtelt werden.

Hier ein Beispiel eines C# Programms, das alle diese Elemente beinhaltet:

```
using System;
namespace Beispiel
{
   class Klassel
```

BEISPIELE MIT SHARPDEVELOP NACHVOLLZIEHEN

Auch wenn Sie das Visual Studio nicht besitzen, können Sie die Beispiele aus diesem Sonderheft leicht nachvollziehen - und zwar mit derfreien Entwicklungsumgebung 'SharpDevelop' die Sie auf der Heft-CD zu diesem Sonderheft finden.
Das Interessante daran: SharpDevelop sieht in vielerlei Hinsicht genau so aus wie das Visual Studio - und funktioniert auch weitestgehend so. Ein paar Kleinigkeiten müssen Sie allerdings doch von Hand ändern - denn diese werden zwar vom Visual Studio, nicht aber von SharpDevelop, geboten.
Wie Sie die Beispielprogramm mit SharpDevelop umsetzen, erfahren Sie hier - anhand des Beispielprogamms aus dem umgebenden Beitrag.

Projekt erzeugen

Zunächst einmal erzeugen Sie mit SharpDevelop ein neues Projekt vom Typ 'Windows-Forms' Projekt. Darin legen Sie dann eine neue XmlForm-Date an: Die können Sie dann in SharpDevelop genau so bearbeiten wie im Forms-Editor von Visual Studio. Das bedeutet, Sie ziehen auch hier einfach die benötigten Controls aus der Toolbar auf die Form. Geben Sie der Form auch einen sinnvollen Namen, zum Beispiel 'ImageForm'.

Form hinzufügen

Die fertige Form speichern Sie dann als .CS-Datei ab und fügen diese dann zu Ihrem Projekt hinzu.
Nun müssen Sie sich darum kümmern das diese Form auch beim Programmstart verwendet wird. Dazu ändern Sie die Zeile, die das Programm startet, so ab, dass statt der automatisch erzeugten MainForm ihre 'ImageForm' verwendet wird:

```
Application.Run(new
➥GeneratedForm.ImageForm());
```

Event-Handler einbauen

Den im Beitrag beschriebenen Quellcode können Sie in SharpDevelop genau so eingeben, wie das auch im Visual Studio der Fall ist Allerdings gibt es da Ausnahmen, und das sind alle Event-Handler.
SharpDevelop hat in der vorliegenden Version noch keine Möglichkeit eingebaut, Event-Handler direkt im Forms-Editor anzugeben, und darum müssen Sie das von Hand tun. (Der Quellcode für die Handler selbst ist natürlich völlig identisch.)

Wenn Sie also im Visual Studio auf den Button zum hinzufügen eines Paint-Event Handlers für das PictureBox-Control drücken, dann gehen Sie in Sharp Develop statt dessen in den Code zum initialisieren der Komponenten.
Um dort den gewünschten Eventhandler hinzuzufügen, würden Sie folgendes Statement verwenden:

```
pictureBox1.Paint += new
System.Windows.Forms.Paint
➥EventHandler(this.OnPaint);
```

Wenn Sie sich nicht sicher sind wie ein benötigtes Statement für einen Event-Handler aussehen muss:
Kein Problem - Sie können einfach den Beispiel-Quellcode von der Heft-CD verwenden, denn der ist natürlich auch mit SharpDevelop kompatibel.

```
    {
    }
    struct Struktur1
    {
    }
    interface Interface1
    {
    }
    delegate void Delegate1();
    enum Enumeration1
    {
    }
    namespace Verschachtelt
    {
    }
    class Klasse2
    {
       public static void
       Main(string[] arguments)
       {
       }
    }
}
```

Namespaces und Typen müssen eindeutige Namen haben. Die Eindeutigkeit entsteht bei den Namen durch die vollständige Angabe der Namens-Hierarchie. Man spricht dabei von 'fully qualified Names'. So ist zum Beispiel 'Beispiel.Klasse1' der fully qualified Name der Klasse 'Klasse1' aus dem Namespace 'Beispiel'.

Anders als bei C++-Programmen sind DLLs bei C# Programmen ein extrem wichtiger Mechanismus. DLLs werden zur Laufzeit des Programms gelinkt und können Code oder Ressourcen enthalten. Die Typen, die in einer DLL enthalten sind, werden dem Programm dabei ebenfalls zur Laufzeit bekannt gemacht.

• Sprachfeatures im Detail

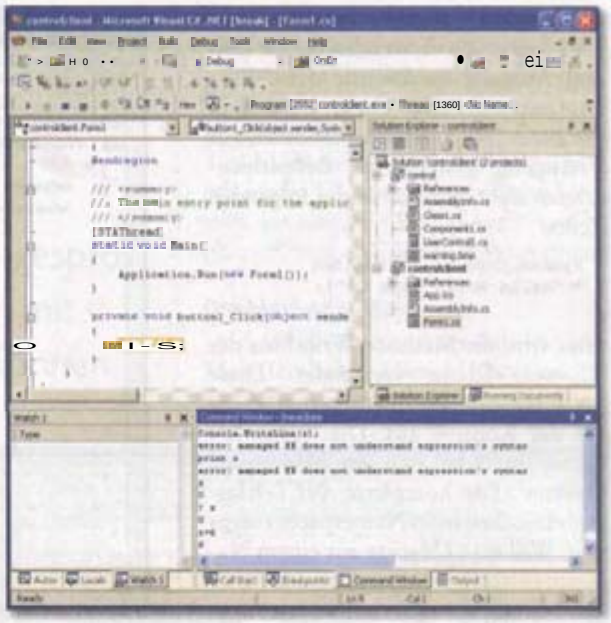
Mit C# kann man Programme für die Kommandozeile, normale Windows-Anwendungen, Win32 Services, und eine ganze Reihe von anderen Programmarten schreiben. Dazu zählen nicht zuletzt auch Clients für Web-Services oder Serveranwendungen. Das führt aber alles im Augenblick zu weit. Deshalb beginnt der Einstieg in die Sprache C# an dieser Stelle zunächst mit einem etwas ausführlicheren Ausflug in die Kommandozeile.

Das einfache 'Hello-World'-Programm haben Sie bereits gesehen. Unter Verwendung des 'using'-Statements kann man sich die Sache etwas angenehmer gestalten und muss die Namen der verwendeten Klassen nicht länger vollständig qualifizieren:

```
using System;
class HelloWorld
{
  static void
  Main()
  {
    Console.write
  Line(
      „Hello World
    from C#");
  }
}
```

Will man die Kommandozeilen-Parameter des Programms verwenden, dann muss man dazu nur die Signatur der Methode Main() verändern.

Im Beispiel sehen Sie dabei auch, wie Sie einen Wert an das Betriebssystem zurück-



IN DER ENTWICKLUNGSUMGEBUNG kann man mit Hilfe des Command-Windows direkt in den Ablauf eines C#-Programms eingreifen.

LISTING 1:

```
using System;
class HelloWorld
{
static int  Main( string[] arguments)
{
  for( int I=0; I<arguments.Legth; I++)
            {
               Console.WriteLine("{0}", arguments[i]);
            }

  return 1;
}
}
```

LISTING 3:

```
int[] ArrayAusIntegern = new int[3] { 1, 3, 5 };
int[] ArrayAusIntegern = new int[] { 1, 3, 5};
int[] ArrayAusIntegern = int[] { 1, 3, 5};
```

GARBAGE COLLECTION - WAS IST DAS?

Die .NET-Laufzeitumgebung beinhaltet einen Garbage Collector und der ist fü'rs Aufräumen des Speichers zuständig. Im Wesentlichen geht es bei Systemen, die Garbage Collection verwenden, darum, dass der Programmierer von einer lästigen Pflicht befreit wird: Er muss sich nicht länger darum kümmern zuvor angeforderten Speicher freizugeben.

In C/C++ wird Speicher gängigerweise per *mallog()* oder *new* angefordert. Als Resultat dieser Anforderung erhält man einen Zeiger auf einen Speicherbereich den man verwenden kann. Benötigt man diesen Speicher nicht mehr, muss er per Aufruf von *free()* oder *delete* wieder freigegeben werden.

Vergisst man aber dieses Freigeben, dann bleibt der Speicher blockiert - es entsteht ein Speicherleck. Passiert dies oft, dann geht dem Programm - und im schlimmsten Fall auch dem System - irgendwann der Speicher aus. Das ist ärgerlich und vor allem bei ereignisorientierten Programmen, wie Windows-Programme das im Allgemeinen sind, macht das Mitführen von Listen aus Zeigern auf Speicherbereiche eine Menge Arbeit.

Bei Systemen, die einen Garbage Collector zur Verfügung stellen, steht der ganze Speicher unter der Kontrolle dieses Garbage Collectors. Der Collector führt dabei Buch darüber, welcher Speicherbereich verwendet wird und welcher nicht. Hat ein Programm einen Speicherbereich angefordert, besitzt aber keinen Zeiger mehr auf diesen Bereich - wird der Bereich also nicht länger vom Programm referenziert - so gibt der Garbage Collector den Speicher an das Betriebssystem zurück.

Der Effekt ist der, dass man als Programmierer einfach immer Speicher anfordern kann, ohne sich um das lästige Aufräumen zu kümmern: Das System nimmt einem diese Aufgabe ab. Resultat: Es bieten sich völlig neue Wege für's Programmieren. Wege, die man aufgrund der aufwändigen Aufräum-Vorgänge mit Systemen ohne Garbage-Collector nicht beschritten hätte.

LISTING 4:

```
using System;
class Student
{
    private string name = "keiner";
    private int note = 0;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    public int Note
    {
        get
        {
            return note;
        }
        set
        {
            note = value;
        }
    }

    public static void Main()
    {
        Student s = new Student();
        s.Name = "Peter MÄller";
        s.Note = 3;
        Console.WriteLine( s.Note);
        s.Note += 1;
        Console.WriteLine(s.Note);
    }
}
```

liefern können: Auch dazu muss die Signatur von Main() verändert werden: Sie geben an, dass Sie einen 'int' zurückliefern möchten und tun das dann mit dem 'return'-Statement am Ende der Funktion.

• C# und Arrays

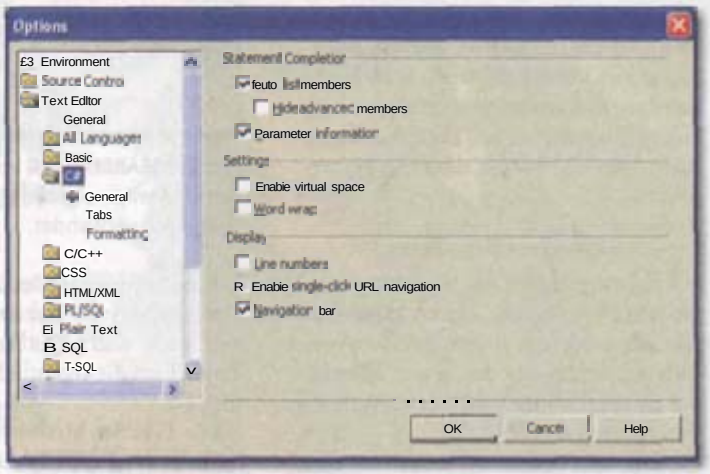
Der Array-Index beginnt in C# bei 0 - genau so, wie in vielen anderen Sprache und auch in C++. Anders als in C++ stehen die eckigen Klammern aber nicht hinter der Variable, sondern hinter der Typangabe:

```
int[] ArrayAuslntegern;
```

Ein weiterer Unterschied ist, dass die Größe des Arrays nicht Teil des Typs ist. Man kann also eine Array-Variable deklarieren und dann ein Array zuweisen - ganz egal, wie groß das Array ist:

```
int[] ArrayAuslntegern;
ArrayAuslntegern = new int[42];
```

Arrays können während der Deklaration initialisiert werden. Werden Arrays



WIE **FÜR DIE** ANDEREN unterstützten Sprachen kann man in VS auch für C# spezielle Einstellungen für den Quellcode-Editor vornehmen.

C# UND C++ - EINE KURZÜBERSICHT

Ci und C++ sind zwar sehr verwandt - aber eben doch nicht gleich. An vielen Stellen gibt es Unterschiede in den Sprachen. Hier die wichtigsten in der Kurzübersicht.

Vererbung: Eine Klasse kann in C# nur eine Basisklasse haben, es gibt also nur eine einfache Vererbung von Implementierungen. Ein Klasse (oder ein Interface) kann aber von mehreren Interfaces erben, beziehungsweise mehrere Interfaces implementieren.

Arrays: Arrays werden in *Ctt* anders deklariert als in C++. Die eckigen Klammern [] stehen in *Ctt* hinter dem Typen des Arrays, nicht hinter der Array-Variable.

Bool: Es gibt keine automatische Konvertierung von bool in andere Datentypen - auch nicht in den Typ 'int'.

Long: Der Datentyp 'long' hat in *Ci* 64 bits - beim VC++ und anderen Win32 C++ Compilern hat ein long nur 32 bits.

Structs: In *Ctt* sind Structs und Klassen semantisch unterschiedliche Dinge. Eine Klasse ist ein 'Reference' Typ, während ein Struct ein 'Value' Typ ist.

Das Switch-Statement: In C# ist der 'fall through'-Mechanismus bei Switch Statements nicht möglich. Jeder 'Case' muss abgeschlossen werden.

Delegates: Bei Delegates handelt es sich um einen Mechanismus, der in etwa Funktionszeigern entspricht. Allerdings sind Delegates typensicher und unterliegen auch den Sicherheits-Mechanismen von .NET.

Aufrufe der Basisklasse: Um in einer abgeleiteten Klasse innerhalb einer überschriebenen Funktion die Implementierung der Basis-Klasse aufzurufen, wird das 'base' Schlüsselwort verwendet. (base.Function())

Präprozessor: Es gibt eigentlich keinen. Zwar unterstützt *Ctt* einen Präprozessor, der ist aber ausschließlich für bedingtes Kompilieren gut. Macros oder Ähnliches können mit dem C#-Präprozessor nicht angelegt werden. *Ci* kennt keine Header-Dateien.

Operatoren: C# hat einige Operatoren, die es in C++ nicht gibt. Das sind zum Beispiel der 'is'- und der 'typeof'-Operator.

Extern und static: Beide Schlüsselworte haben eine leicht unterschiedliche Bedeutung im Vergleich mit C++.

Parameter als Referenz: Für die Übergabe von Parametern als Referenz (by reference) sind in *Ctt* die Schlüsselworte 'ref' und 'out' zu verwenden. Pointer sind zwar möglich - aber nur in einem speziellen Modus, dem 'unsafe'-Mode der Sprache, der außerhalb des Garbage Collectors operiert.

Globale Funktionen und Variable: ... sind in *Ctt* verboten. Jeder Funktion und jede Variable muss in einer Typendeklaration enthalten sein.

Initialisierung: Lokale Variable müssen in *Ctt* initialisiert worden sein, bevor sie verwendet werden können. Sind sie das nicht, so generiert das eine Fehlermeldung des Compilers.

Default-Werte: C# kennt keine Default-Werte für Parameter von Funktionen.

Mit Ausnahme des doch manchmal schmerzlich vermissten Präprozessors sind in *Ctt* im Wesentlichen Dinge weggelassen worden, die in C++ eher selten benutzt wurden - die Sprache als solches ist dadurch aber deutlich einfacher und übersichtlicher geworden. Einige Regeln der Sprache führen dazu, dass Unklarheiten, die in C++ zum Beispiel beim Überladen von Funktionen möglich waren, gar nicht erst entstehen können.

vom Code aus nicht initialisiert, so initialisiert sie der Compiler auf den Default-Wert des Array-Typen. Beim Initialisieren kann man verschiedene Dinge weglassen - so zum Beispiel die Größe des Arrays und auch das 'new'-Statement. Daher sind die folgenden Ausdrücke inhaltlich alle identisch.

Der Zugriff auf die einzelnen Elemente im Array erfolgt wie in C++. Der zweite Wert aus dem Beispielarray kann also wie folgt ausgelesen werden:

```
int x = ArrayAusIntegern[ 1];
```

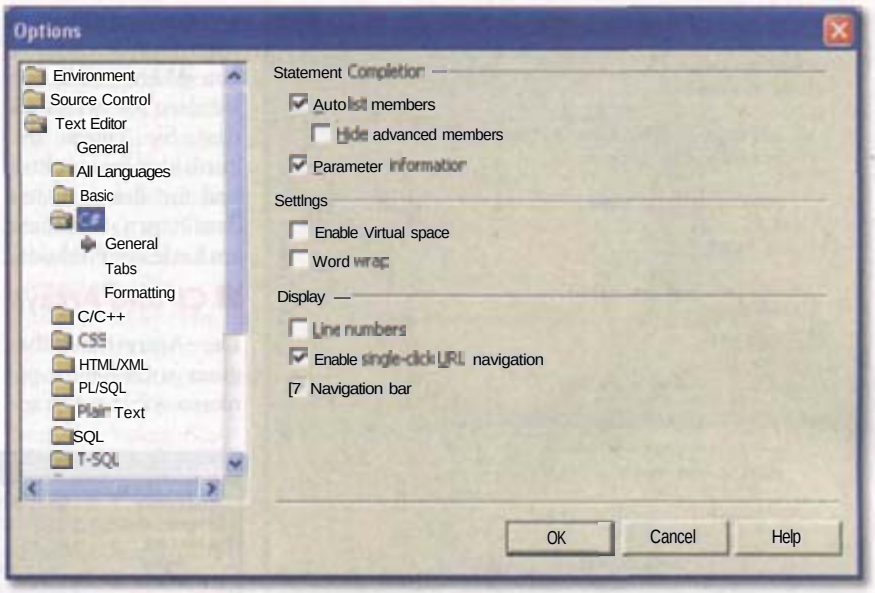
Allerdings sind Arrays in C# eigentlich Objekte - und vom Typ *System.Array* abgeleitet. Daher kann man alle Methoden und Eigenschaften, die die Basisklasse zur Verfügung stellt, auch in abgeleiteten Klassen verwenden. Eine solche Eigenschaft ist 'Length'. Damit kann man die Länge eines Arrays ermitteln:

```
int Anzahl =
    ArrayAusIntegern.Length;
```

C# stellt außerdem das 'foreach'-Statement zur Verfügung. Damit kann über alle Elemente aus einem Array (sowie über alle Elemente aus einem Objekt, das ein bestimmtes Interface implementiert) iteriert werden:

```
foreach(
    int x in ArrayAusIntegern)
{
    Console.WriteLine( x);
}
```

Wichtig: Die 'Eigenschaften' (properties). C# hat mit 'Properties' einen Sprachteil, der dafür sorgt, dass Objekte deutlich einfacher benuzt werden können als in C++. Eine 'Property' ist eine Methode, die lesenden und optional schreibenden Zugriff auf ein privates Datum einer Klasse bietet.
Properties sind also Zugriffsmethoden auf solche Eigenschaften einer Klasse, auf die der Programmierer der Klasse dem Nutzer der Klasse Zugriff geben möchte.



FÜR DIE BEARBEITUNG von Properties zur Laufzeit des Programms sorgt das .NET-Framework mit einer eigenen Komponente: Die wird von VS zum Beispiel zum Setzen der Einstellungen verwendet.

Eine Property-Methode hat einen Datentyp, einen Bezeichner sowie einen 'get'- und einen 'set'-Teil für den Zugriff auf die eigentlichen Daten (Listing 4).

Die Get/Set Methoden sind also innerhalb der Property-Deklaration einzufügen. Beide Methoden sind dabei optional: Man kann als auch eine Read-Only oder einer Write-Only Property definieren.

Nach der Deklaration der Properties können Sie so verwendet werden, als wären es Felder der Klasse. Das macht die Benutzung sehr einfach. In der 'Set'-Methode einer Property-Definition steht die besondere Variable mit dem Namen 'value' zur Verfügung. Diese enthält den Wert, der gesetzt wurde.

Anders als bei Zugriffsmethoden in C++ kann man mit den Properties von C# einige Dinge deutlich kompakter hinschreiben:

```
S.Note +=1;
```

Dieser Ausdruck würde in C++ in etwa folgendermaßen zu formulieren sein:

```
s.SetNote( s.GetNote() +1 );
```

• Überladenes

C# hat für das Überladen von Funktionen bestimmte Schlüsselworte reserviert. Dabei handelt es sich um die Worte 'new', 'virtual', und 'override'. Ist eine Funktion in einer Basisklasse dafür vorgesehen, überladen zu werden, so ist sie mit dem Schlüsselwort 'virtuaP zu versehen.

Die abgeleitete Klasse kann die Methode dann entweder mit dem 'override' Schlüsselwort überladen oder die Implementierung der Basisklasse mit dem 'new' Schlüsselwort verstecken (Listing 5).

In diesem Beitrag haben Sie die ersten Grundzüge der Sprache C# erlernt. Natürlich hat C# noch eine ganze Reihe an anderen Features zu bieten, doch für einen ersten Einblick in die Sprache sollte dieser Beitrag ausreichen. Wie das immer so ist: Trockenes Lernen allein reicht nicht aus - darum finden Sie in diesem Sonderheft auch viele Beispielprogramme, mit denen Sie Ihre Wissen über die Programmierung mit C# für .NET erweitern können. © uR

LISTING 5:

```
public class Basis
{
    public virtual string Methode ()
    {
        return "Basis";
    }
}

public class Derived1 : Basis
{
    override public string Methode()
    {
        return "Abgeleitet";
    }
}
```

Mit der Windows Forms Bibliothek:

Das erste .NET Programm

Im Zuge dieses Beitrages werden Sie Ihre erstes eigenes .NET-Programm schreiben und dabei einige Klassen aus der .NET-Klassenbibliothek kennen lernen.

THOMAS WÖLFER

Die .NET-Klassenbibliothek ist umfangreich - sehr umfangreich. Entsprechend einfach ist es recht leistungsfähige Programme mit relativ wenig Quellcode zu erstellen. Schritt für Schritt entwickeln Sie dabei ein Bildbetrachtungsprogramm mit C# und dem Visual Studio.

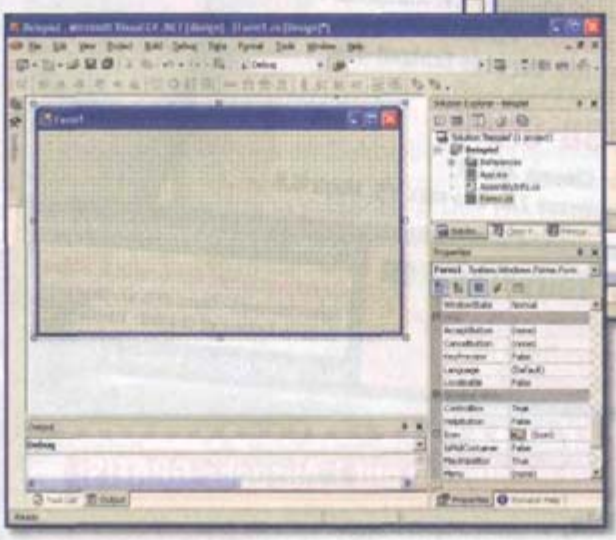
Das Programm wird ein Programm, das die Windows.-Forms-Bibliothek benutzt.

Diese Bibliothek ist für normale Windows Programme mit Fenstern, Menüs und normalen Kontrollelementen gedacht. Sie wird vom Visual Studio besonders unterstützt: Dort gibt es einen grafischen Editor der das Zusammenbauen von Fenstern aus verschiedenen Komponenten ermöglicht. So wie man früher in VC++ Dialogboxen editieren konnte, kann man mit dem Forms Editor komplette Fenster anlegen.

• Der erste Schritt

Ein neues Projekt legt man in Visual Studio mit dem passenden Befehl aus dem Datei-Menü an. Dabei wird eine Dialogbox geöffnet, in der die Entwicklungsumgebung verschiedene Projektarten zur Auswahl anbietet. Für das Beispielprogramm benötigen Sie ein *Windows Forms* Projekt.

Wird die Auswahl abgeschlossen, dann legt Visual Studio das komplette Projekt an. Dazu gehört auch eine vordefinierte Form. Forms sind die Hüllobjekte zur Aufnahme von Kontrollelementen - es handelt sich dabei also um die normalen 'Fenster' eines Windows-Programms.



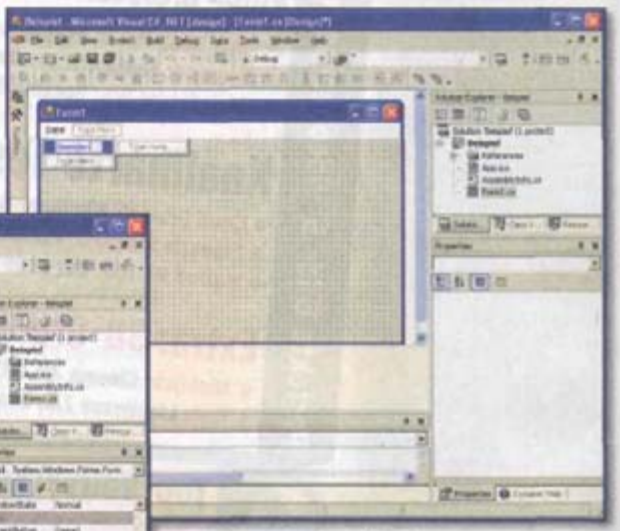
DIE ERSTE FORM, ein ganz normales Windowsfenster.

Das erzeugte Projekt enthält drei Dateien. Die eine Datei ist die Icon-Datei für das Programm und die zweite Datei hat den Namen *AssemblyInfo.cs*. Die AssemblyInfo ist nur eine Hilfsdatei mit einigen Metainformationen über das Programm. Diese Datei können Sie zunächst ignorieren. Die dritte Datei hat den Namen *Form1.cs*. Das ist die Datei, die das erzeugte Fenster enthält, und dies ist auch die Datei, mit der Sie hauptsächlich arbeiten werden.

Wenn Sie im Visual Studio auf diese Datei doppelklicken, wird sie in den Forms-Editor geladen. Im Forms-Editor können Sie das Fenster mit grafischen Werkzeugen bearbeiten. Wenn Sie hingegen im Quellcode der Datei arbeiten wollen, dann können Sie die Datei mit einem Rechtsklick in den Quellcode-Editor laden.

• Menü anlegen

Zunächst laden Sie die Datei aber in den Forms-Editor, und der zeigt dann ein einfaches, noch völlig leeres Fenster an. Das Programm soll ein Bildbetrachter werden. Und wie jeder gute Bildbetrachter soll es natürlich auch über ein Menü verfügen. Dieses Menü legen Sie nun als Erstes an.



DAS ERSTE MENÜ anlegen.

Bei geöffnetem Forms-Editor klicken Sie dazu auf die *Toolbox* von Visual Studio. Diese öffnet sich und zeigt verschiedene Reiter an. Einer davon ist der Reiter *Forms Controls*, und darin befinden sich - wie überraschend - die Kontroll-Elemente, die Sie auf Forms verwenden können.

Eines dieser Elemente ist das Element vom Typ *MainMenu*. Das sind Menüs, die am oberen Rand des Fensters angezeigt werden. Genau so ein Kontroll-Element ziehen Sie nun mit der Maus aus der Toolbox auf die Form. Das Menü erscheint dann innerhalb der Form, am oberen Rand der Form wird zeitgleich ein leeres Menü eingeblendet. Dort geben Sie zunächst den Text *Datei* ein, und

unterhalb dieses 'neuen' Menüs tragen Sie dann den Text *Beenden* ein: Das wird später zur Laufzeit des Programms das Menü des Bildbetrachters.

• Panels verwenden

Ist das Menü einmal angelegt, so kann der Rest der Arbeitsfläche des Programms gestaltet werden. Der Bildbetrachter soll in zwei Teile aufgeteilt sein.

Der linke Teil des Programmfensters wird einen Browser für das Dateisystem enthalten, der dem Explorer-Baum ähnelt. Der rechte Teil des Fensters wird eine Zeichenfläche enthalten, in der das links ausgewählte Bild angezeigt wird.

Windows Forms verfügt über verschiedene zusammenhängende Konzepte, die das Layouten von Fenstern bereffen. Zum einen ist es so, dass Fenster einschließlich ihres Inhalts in ihrer Größe verändert werden können. Dabei wird die Größe der verschiedenen Elemente im Fenster automatisch angepasst, wenn man zuvor die Art und Weise festlegt, in der sich die Elemente aufeinander oder auf das Hauptfenster beziehen.

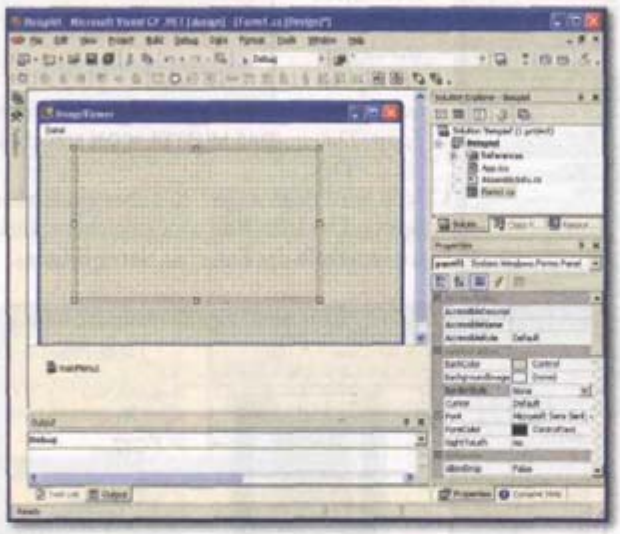
Dazu gibt es so genannte *Panel*-Controls. Diese Panels können mit einem festen Abstand, bezogen auf die Elemente, die um das Panel herum angeordnet sind, definiert werden. Oder man befestigt das Panel fest an einer bestimmten Seite des Fenster.

Das Befestigen eines Controls an einem Rand nennt man 'docken'. Als Zustand für das Docken gibt es die Werte *none*, die vier Seiten - also *top*, *left*, *bottom* und *right* - sowie den Zustand *Fill*. Bei letzterem füllt das Kontrollelement den gesamten Platz innerhalb des Controls auf, in dem es sich befindet und der noch nicht von anderen Elementen belegt wird.

Zum Trennen zwischen Elementen, die innerhalb eines Panels liegen, gibt es so genannte *Splitter*-Elemente. Befindet sich ein Splitter zwischen zwei Controls, dann kann er dazu verwendet werden, die Position der Controls gegeneinander zu verschieben.

Wird der Splitter also nach rechts bewegt, dann erhält das linke Kontrollelement mehr Platz während der Raum für das rechte Element kleiner wird. Dabei kann man Panels auch verschachteln. Panels können sich also auch innerhalb von anderen Panels befinden.

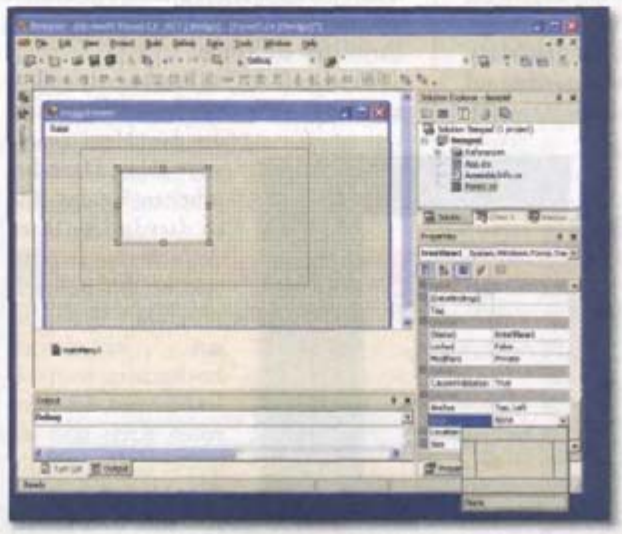
Zunächst ziehen Sie nun ein Panel aus der Toolbox auf die Form.



EIN PANEL aus der Toolbox ziehen.

Das Panel wird später alle Elemente der Arbeitsfläche des Programms beinhalten. Das erste Element ist dabei eine *TreeView*. Eine TreeView ist ein Kontroll-Element, das man für die Anzeige von hierarchisch geordneten Daten verwenden kann - also zum Beispiel für die Anzeige des Dateisystems.

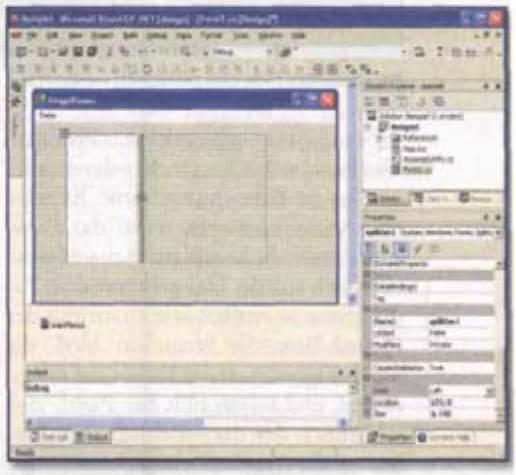
Ziehen Sie zunächst das TreeView-Element aus der Toolbox auf die Form und zwar in einen Bereich innerhalb des Panels.



DAS TREEVIEW-ELEMENT für hierarchisch geordnete Daten.

Dann wechseln Sie ins *Eigenschaften*-Fenster von Visual Studio und suchen dort die Eigenschaft *Dock* aus dem Bereich *Layout*. Dort stellen Sie als *Dock*-Verhalten ein, dass TreeView links im Panel angedockt sein soll. Im Panel können Sie dann sofort erkennen, dass das Element an den linken Rand des Panels verschoben wird. Den rechten Rand von TreeView können Sie dann noch an die Position ziehen, die Ihnen für die Größe des Dateisystem-Browsers vorschwebt. (Zur Laufzeit des Programms wird das aber auch möglich sein - Sie legen hier nur die initiale Größe von TreeView fest.)

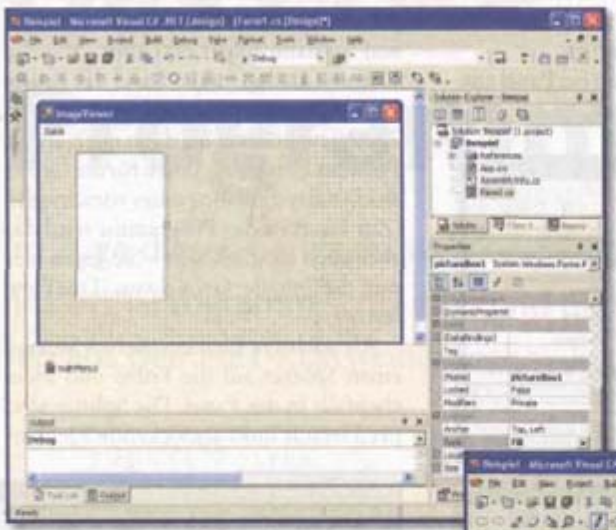
Als nächstes Element ziehen Sie nun einen *Splitter* auf die Form und zwar ebenfalls in das Panel. Die Splitter sind per Default links andockende Elemente, und so wird der Splitter auch automatisch am rechten Rand der TreeView angezeigt. Beide Elemente docken also am linken Rand an - die Reihenfolge der Elemente von links nach rechts ist dabei davon abhängig, welches der Elemente zuerst erzeugt wurde. (Sie können diese Reihenfolge aber später im Visual Studio beeinflussen.)



DER SPLITTER dockt an TreeView an.

Als viertes und vorläufig letztes Element ziehen Sie nun ein Kontroll-Element vom Typ *PictureBox* in den freien Bereich des Panels. Für dieses Element legen Sie dann als Dock-Zu-

stand aber nicht den linken Rand fest. Statt dessen stellen Sie dessen Dock-Eigenschaft auf *Fill*. Die Picture-Box füllt den kompletten freien Bereich des Panels aus.

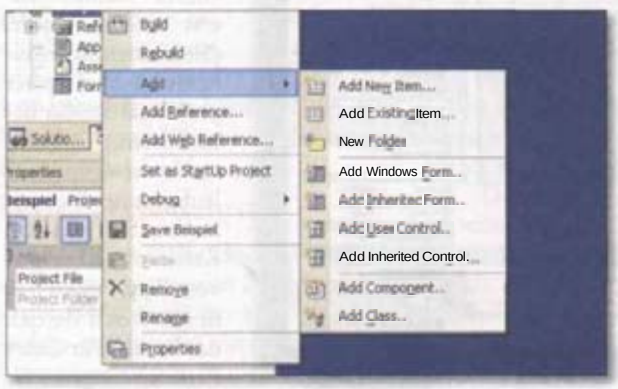


PICTUREBOX füllt den restlichen freien Platz.

Nun gibt es noch ein kleines Problem: Das Panel selbst liegt mehr oder minder planlos im Bereich der Form. Es wäre aber wünschenswert, wenn das Panel statt dessen die komplette Arbeitsfläche ausfüllen würde. Das geht mit dem Panel genau so einfach wie zuvor mit der Picture-Box: Sie brauchen bloß die *Dock*-Eigenschaft des Panels auf *Fill* zu stellen, und schon füllt das Panel, einschließlich der darin befindlichen Elemente, die Arbeitsfläche aus.

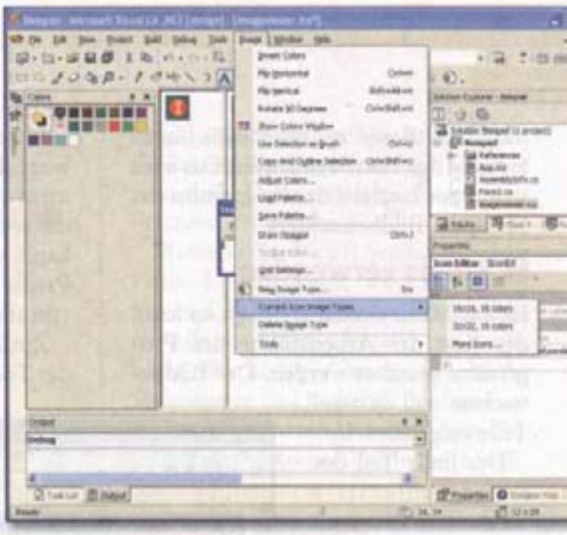
Damit ist das Hauptfenster fertig gestellt. Allerdings hat das Programm noch ein relativ hässliches Icon. Wie bei jedem guten Programm ist das Programm-Icon aber natürlich das wichtigste überhaupt, und darum ändern Sie das Icon nun nach Ihren Wünschen.

Dazu wird zunächst ein Icon angelegt. Sie klicken mit der rechten Maustaste auf den Projektnamen in der Projektansicht.



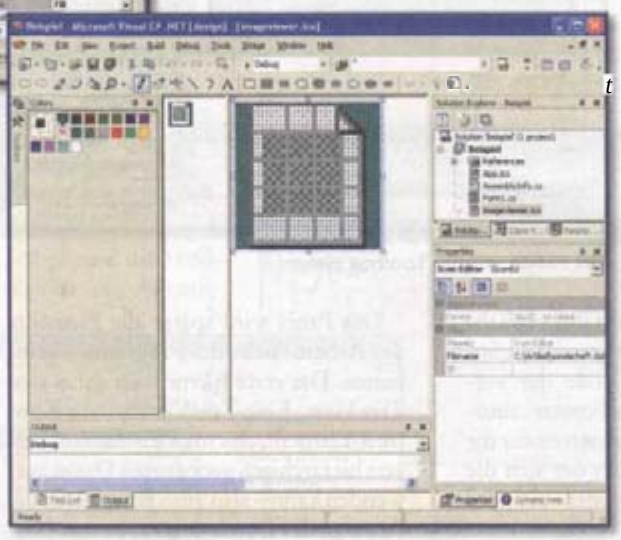
MIT ADD NEW ITEM den Icon-Bau starten.

Visual Studio öffnet dann ein Menü aus dem verschiedene Befehle ausgewählt werden können.



DAS MUSTER-ICON schlicht und einfach.

Anschließend wählen Sie hier den Befehl *Add New Item* und als neu anzulegendes Element eine *Icon-Datei*. Das Icon wird dann angelegt und im Icon-Editor innerhalb der Entwicklungsumgebung geladen. Icons sind eigentlich nur Bitmaps, allerdings kann eine Icon-Datei mehrere Bilder mit unterschiedlichen Auflösungen und Farbtiefen enthalten.



ARBEITEN mit dem Icon-Editor.

Welches Bild Sie bearbeiten wollen, können Sie mit dem passenden Befehl aus dem Menü *Image* auswählen. Dann bearbeiten Sie das Bild so, dass das Icon Ihren Anforderungen entspricht. Im Beispielprogramm finden Sie ein künstlerisch hochgradig wertvolles Element, das einen roten Kreis und ein gelbes [I] enthält. Wenn Sie ein besserer Künstler sind als der Autor, werden Sie sicher ein schöneres Icon erfinden. Dann speichern Sie das Icon und schließen den Icon-Editor.

Wenn der Forms-Editor nicht noch aktiv ist, dann öffnen Sie die Forms-Datei erneut durch einen Doppelklick auf die zugehörige Datei.

In den Eigenschaften der Form suchen Sie dann das Feld fürs *Icon*. Dort finden Sie einen Button, der einen Dialog zur Auswahl von Icon-Dateien öffnet. Mit diesem Dialog wählen Sie dann das zuvor erzeugte Icon aus.

Das Bild wird dann im Eigenschafts-Fenster angezeigt, und auch die Form wird mit dem neuen Icon in der Titelleiste angezeigt.

Damit sind die visuellen Tätigkeiten zunächst beendet: Sie können das Programm nun übersetzen und das erste Mal starten.

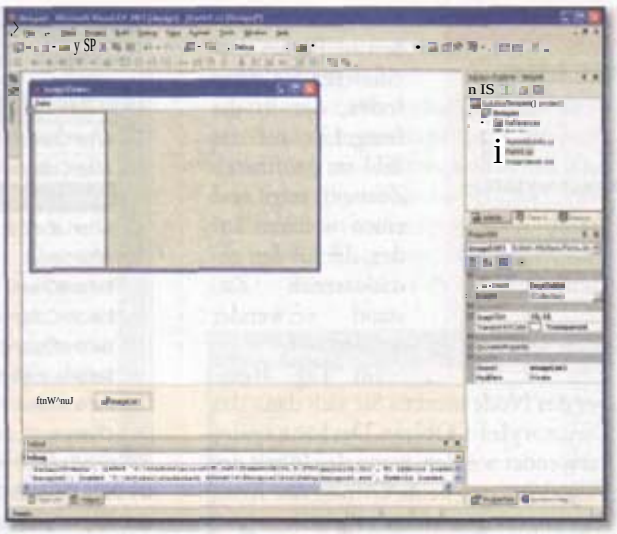
DER IMAGE-VIEWER im Leerlauf.

Ihr Programm hat nun schon eine ganze Menge an Funktionalität. Sie können das Fenster in der Größe verändern und auch den Platz der beiden Elemen-

te innerhalb des Arbeitsbereiches des Fensters durch das Verschieben des Splitters anpassen. Außerdem hat das Programm auch ein Hauptmenü - auch wenn dieses selbst nur einen Befehl enthält, der nichts tut.

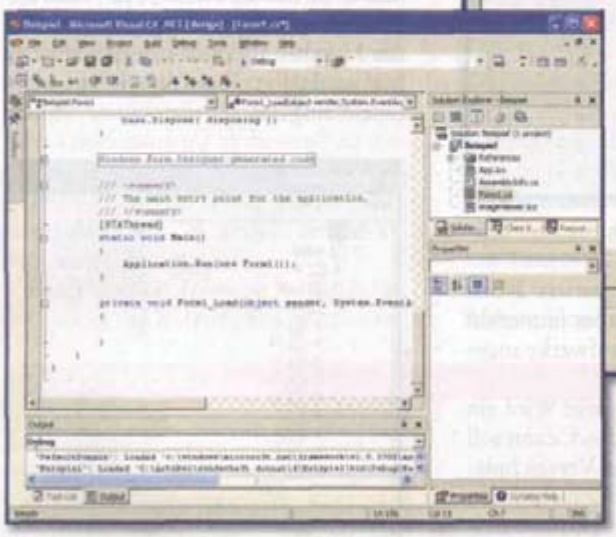
Programmcode

Nun kommt der eigentliche Programmcode an die Reihe. Das Programm soll beim Starten automatisch den Inhalt des Dateisystems in TreeView anzeigen. Das bedeutet, Sie brauchen einen Zeitpunkt zum Start des Programms, bei dem TreeView bereits existiert und mit Daten gefüllt werden kann. Dafür gibt es bei Windows.Forms ein spezielles Ereignis und zwar das Load() Event der Form.

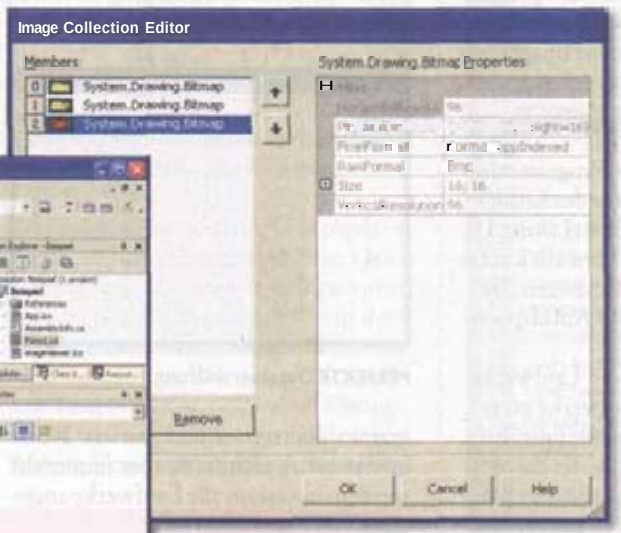


MIT IMAGELIST die Dateisystem-Symbole verwalten.

IMAGELIST muss TreeView zugewiesen werden.



DAS LOAD()-EVENT muss **ausprogrammiert** werden.



DREI SYMBOLE FÜR DIE BILD-DATEI-VERWALTUNG: Ordner geschlossen, Ordner offen und Bilddatei-Symbol.

Um Code zu generieren, der für dieses Ereignis ausgeführt wird, brauchen Sie einen Event-Handler, und den erzeugen Sie am einfachsten durch einen Doppelklick auf die Form. Visual Studio erzeugt dann einen leeren Handler, öffnet den Quellcode-Editor und platziert den Cursor an der zugehörigen richtigen Stelle.

Für die Anzeige des Dateisystems in der TreeView brauchen Sie ein paar Bitmaps: Eines für die Darstellung eines geöffneten Ordners, eines für die Darstellung eines geschlossenen Ordners und eines für die Darstellung von Bilddateien.

Bitmaps werden in Forms-Programmen mit *ImageListen* verwaltet. Dazu gibt es ein spezielles ImageList Kontroll-Element in der Toolbox. Ein solches Element ziehen Sie nun aus der Toolbox auf die Form.

Nun müssen die Bitmaps, die in der ImageList enthalten sein sollen, zur Liste hinzugefügt werden. Dazu legen Sie zunächst die drei Bitmaps mit dem Bitmap-Editor an, oder Sie verwenden einfach die aus dem Beispielprogramm.

Um die Bilder zur Liste hinzuzufügen, klicken Sie in der Eigenschafts-Ansicht der ImageList auf den Button *Collection*.

Daraufhin öffnet sich der Editor zum Bearbeiten von Bild-Sammlungen, der Image-Collection Editor. Hier können Sie mit Hilfe des *Add*-Buttons die drei Bitmaps zur Liste hinzufügen. Dann können Sie den Editor beenden, die Liste ist vollständig definiert.

Damit Sie die Bilder auch in TreeView verwenden können, müssen Sie nun die

LISTING 1:

```
private void Form1_Load(object sender, System.EventArgs e)
{
  string[] drives = Environment.GetLogicalDrives();
  foreach( string drive in drives)
  {
  Directoryinfo di = new DirectoryInfo( drive /*"c:\\"*/);
  TreeNode tn = new TreeNode( drive /*"c:\\"*/, 0, 1);
    tn.Tag = di;
    this.treeView1.Nodes.Add(tn);
  }
}
```

LISTING 2:

```
private void OnAfterExpand(object sender, System.
➥Windows.Forms.TreeViewEventArgs e)
{
  if( e.Node.Tag != null)
  {
    DirectoryInfo di = e.Node.Tag as DirectoryInfo;
    e.Node.Tag = null;
    e.Node.Nodes.Clear();
    Directories( di, e.Node);
    Files( di, e.Node);
  }
}
```

ImageList noch zuweisen. Das geht ebenfalls m der Eigenschafts-Ansicht.

Zunächst klicken Sie dazu im Forms-Editor auf *Tree View*. Werden dessen Eigenschaften angezeigt, dann suchen Sie darin nach dem Feld *ImageList*. Dort können Sie die zu verwendende ImageList aus einer Combo-Box auswählen. Nachdem es nur eine einzelne ImageList gibt, fällt hier die Auswahl nicht schwer.

• Endlich Programmieren

Jetzt geht es endlich ans Programmieren. Der Event-Handler für das Load()-Ereignis wird nun aufgefüllt (Listing 1). VonHausaussollTreeViewalle Laufwerke im lokalen System anzeigen. Deren Inhalt wird dann beim Aufklappen ermittelt und angezeigt.

Zunächst gilt es also, die Laufwerke zu ermitteln. Um die Laufwerke zu ermitteln, verwenden Sie in .NET die 'Environment'-Klasse. Diese bietet die Methode *GetLogicalDrives* die die Bezeichnung aller Laufwerke als Array von Strings liefert.

Dieses Array durchlaufen Sie dann mit *foreach*. In jedem Durchlauf wird also ein Laufwerksbuchstabe behandelt.

Informationen über Verzeichnisse ermitteln Sie bei .NET mit der Directory-Info-Klasse. Davon legen Sie eine Instanz an.

Ist das erledigt, geht es um die Aufnahme eines neuen Knotens in Tree-View. Knoten werden dabei durch Tree-Node-Elemente verwirklicht. Als Text für den Knoten geben Sie einfach den Laufwerksbuchstaben an. Ferner spezifizieren Sie einen Index, der in die ImageList auf das Bild im geöffneten Zustand zeigt und einen weiteren Index, der für den geschlossenen Zustand verwendet werden soll.

Im 'Tag' *Member* des Node merken Sie sich dann das DirectoryInfo-Objekt. Das kann später verwendet werden, wenn der Inhalt des Verzeichnisses beim Öffnen des Knotens angezeigt werden soll.

Schließlich fügen Sie den neuen TreeNode der Knoten-Sammlung in TreeView hinzu. Nun können Sie das Programm übersetzen und starten. Noch immer tut es nicht viel, aber immerhin werden nun schon alle Laufwerke angezeigt.

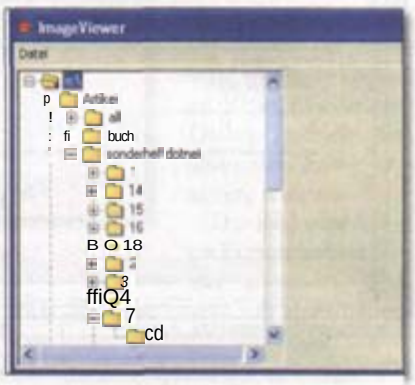
PERFEKTE Dateiverwaltung.

Jetzt kann es weiter gehen: Wird ein Knoten in Tree View geöffnet, dann soll der Inhalt des zugehörigen Verzeichnisses angezeigt werden. Sie brauchen also wieder einen Event-Handler, diesmal einen für den 'Expand' Event. Diesen Handler erzeugen Sie am besten in der *Events*-Ansicht des Eigenschafts-Fensters, wenn es die Eigenschafen von Tree-View anzeigt (Listing *2*).

Damit die Verzeichnisse nicht jedes Mal ausgelesen werden, wenn sich ein Ordner öffnet, wenden Sie einen Trick an: Im 'Tag' *Member* der Knoten haben Sie sich ja die DirectoryInfo-Objekte gemerkt.

IN DEN EIGENSCHAFTEN von TreeView das Expand-Ereignis instrumentieren.

Was Sie nun tun, ist das Folgende: Sie überprüfen ob dieses Member gesetzt ist und wenn das der Fall ist, dann lesen Sie das Verzeichnis aus. Danach setzen Sie das Member auf *Null* - beim nächsten Mal wird dann dieses Verzeichnis nicht erneut ausgelesen.



DIE DATEI-HIERACHIE einwandfrei dargestellt.

Für das Verzeichnis legen Sie dann ebenfalls eine neues TreeNode-Element an. Mit einer Hilfefunktion füllen Sie dann diesen Knoten mit den Informationen über das Verzeichnis auf.

LISTING 3:

```
private void Directories( DirectoryInfo di,
➥TreeNode tn)
{
  foreach( DirectoryInfo dir in di.GetDirectories())
  {
    TreeNode n = new TreeNode( dir.Name, 0, 1);
    n.Tag = dir;
    tn.Nodes.Add( n);
  }
}
```

LISTING 4:

```
foreach( FileInfo fi in di.GetFiles())
{
  if( IsImage( fi.Extension))
  {
    TreeNode n = new TreeNode( fi.Name, 2, 2);
    n.Tag = fi;
    tn.Nodes.Add( n);
  }
}
```

Zunächst lesen Sie alle Verzeichnisse aus, die im aktuellen Verzeichnis enthalten sind (Listing 3).

Der Code für's Auffüllen des Knotens wird Ihnen bekannt vorkommen, denn er entspricht im Wesentlichen dem Code, den Sie schon für die Laufwerke verwendet haben.

Nun lesen Sie noch alle Dateien aus dem Verzeichnis aus. Dabei überprüfen Sie, ob es sich um eine Datei mit einem bekannten Bildformat handelt. Natürlich kommt .NET mit den Windows typischen Bildformaten wie *.ico*, *.bmp* und *.ffiw*/klar. Zusätzlich sind aber auch die gängigen Internet-Formate wie *.gif* und *.png* kein Problem und auch *.tif*-Dateien und andere können von .NET direkt gelesen werden.

Im Beispielprogramm werden nicht alle von .NET unterstützten Dateitypen auch verwendet, es ist aber kein Problem wenn Sie das selbst erweitern möchten (Listing 4).

Für jeden vom ImageViewer unterstützten Dateityp erzeugen Sie ein neues TreeNode-Objekt. Das Objekt wird dann den Knoten von TreeView hinzugefügt. Ähnlich wie bei den Directory-Info-Objekten wird in diesem Fall das FileInfo-Objekt im Tag-Member des Knotens gemerkt. Diese Information wird später für's Anzeigen der Bilder verwendet.

Die Funktion *IsImage()* wird dafür verwendet um festzustellen, ob die Dateierweiterung der Datei zu einem bekannten Bilddateiformat gehört oder nicht (Listing 5).

Bei den Dateien wird allerdings ein anderer Index in die ImageList angegeben, als bei den Verzeichnissen: Hier soll statt des Ordner-Symbols das Symbol für eine Bilddatei verwendet werden. Wenn Sie nun das Programm übersetzen und starten, dann bietet sich ein neues Bild:

DIE BILDDATEIEN werden sichtbar.

Es werden nicht nur die Order sondern auch alle Bilddateien innerhalb der Ordner angezeigt. Zumindest dann, wenn es sich um eine Bilddatei handelt mit dem das Programm klarkommt.

So weit, so gut. Allerdings fehlt für einen Image-Viewer noch eine Kleinigkeit: Das Programm zeigt noch keine Bilder an, und das ist natürlich ein bisschen unbefriedigend für ein Programm, dessen zentrale Aufgabe das Anzeigen von Bilder ist.

• Bilder in der Picture Box anzeigen

Dieser Teil des Programms folgt aber nun- und wie Sie schnell feststellen werden, ist das viel einfacher, als man sich vorstellen würde.

Jedesmal, wenn ein Bild in TreeView ausgewählt wird, soll es in der PictureBox, die Sie zuvor auf der Arbeitsfläche platziert haben, angezeigt werden. Die

EIN EVENT-HANDLER für die Bild-Anwahl muss programmiert werden.

Informationen über die Bilddateien sind ja im Tag-Member der TreeNode-Elemente gespeichert.

Alles was man also braucht, ist ein Event-Handler, der sich um das Anklicken der TreeNodes kümmert. Dazu gehen Sie wieder in das Eigenschaften-Fenster und dort in die Event-Ansicht.

Das Ereignis, das behandelt werden soll ist das 'AferSelect'-Ereignis: Nachdem ein TreeNode ausgewählt wurde, wird es ausgelöst und folgt Ihren Befehlen (Listing 6). Bei diesem Ereignis untersuchen Sie zunächst, ob das Tag-

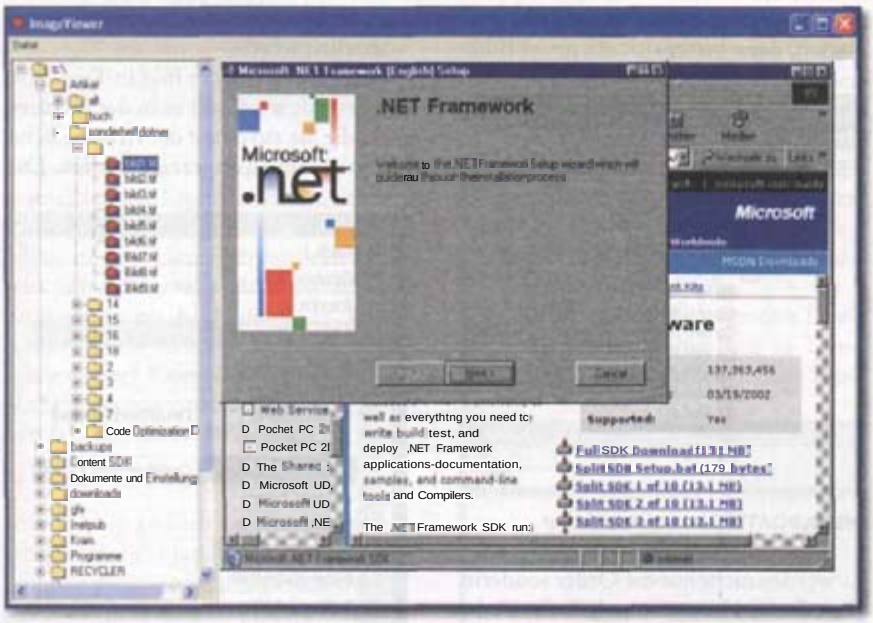
LISTING 5:

```
private bool IsImage( string ex)
{
  if( ex.Length > 2)
  {
    ex = ex.ToLower().Substring( 1);
    if ( ex == „bmp" || ex == „jpg" || ex == „ico" || ex == „tif"
    ➥|| ex == „gif" || ex == „png")
    {
      return true;
    }
  }
  return false;
}
```

LISTING 6:

```
private void OnAfterSelect(object sender, System.Windows.Forms.
➥TreeViewEventArgs e)
{
  if( e.Node.Tag != null)
  {
    if( e.Node.Tag is FileInfo)
    {
      FileInfo fi = e.Node.Tag as FileInfo;
      Display( fi);
    }
  }
}
```

Member vom ausgewählten TreeNode gesetzt ist. Ist das der Fall, muss noch sichergestellt werden, ob es sich um ein FileInfo-Objekt handelt. Schließlich könnte es ja auch sein, dass ein Verzeichnis ausgewählt wurde. Handelt es sich aber um ein FileInfo-Objekt, dann



DAS ANGEWÄHLTE BILD WIRD ANGEZEIGT. Hier ein Screenshot aus dem ersten Beitrag in dieser Ausgabe.

bemühen Sie eine private Funktion namens *Display*, um das Bild anzuzeigen (Listing 7).

Für die Anzeige des Bildes brauchen Sie zunächst ein 'Graphics'-Objekt. Dieses Objekt kapselt alle Funktionen, die zum Malen von Elementen und zum Anzeigen von Bildern verwendet werden können.

Dann erzeugen Sie ein Image-Objekt mit der FromFile()-Methode der Image-Klasse. Diese kümmert sich um all die lästigen Details, wie zum Beispiel um das Dekodieren der verschiedenen Dateiformate. Danach liegt das Bild vor und kann verwendet werden.

Dazu wird zunächst der Hintergrund der Imagebox übermalt. Das stellt sicher, dass das neue Bild auf einer 'frischen' Zeichenfläche ausgegeben wird. Schließlich verwenden Sie die Methode *DrawImageUnscaled()*, um das Bild

LISTING 7:

```
private void Display( FileInfo file)
{
  Graphics g = Graphics.FromHwnd( this.pictureBox1.Handle);
  Image image = Image.FromFile( file.FullName);
  g.Clear( this.pictureBox1.BackColor);
  g.DrawImageUnscaled( image, new Point( 1,1));

  string s = file.FullName + " " + file.Length + "(byte) "
  ➥+ image.Size.ToString() + " " + image.PixelFormat.ToString();
  this.statusBar1.Text = s;..
}
```

LISTING 8:

```
ttprivate void OnPaint(object sender, System.Windows.Forms.PaintEventArgs e)
{
  if( current != null)
  {
    Image image = Image.FromFile( current.FullName);
    e.Graphics.Clear( this.pictureBox1.BackColor);
    e.Graphics.DrawImageUnscaled( image, new Point( 1,1));
  }
}
```

DIE STATUSZEILE wird angelegt.

auszugeben. Es wird einfach ohne irgendwelche Bearbeitungen oder Skalierungen angezeigt. Gleichzeizig werden ein paar Informationen an die noch nicht vorhandene Statuszeile abgesandt.

Jetzt ist das ganze Programm schon deutlich funktionaler. Allerdings: Es fehlt noch die Statuszeile, in der die Bildinformationen angezeigt werden. Dazu gehen Sie in den Forms Editor und platzieren eine Statuszeile aus der Toolbox in der Form.

Danach kann das Programm wieder übersetzt werden. Jetzt gibt es nur noch ein Problem:

Wenn Sie das Fenster des Programms in der Größe verändern, dann wird das Bild nicht neu angezeigt und das erzeugt ein paar hässliche weiße Ränder.

Das ist aber ebenfalls leicht geklärt. Alles was Sie brauchen, ist ein Event-Handler für den Paint-Event der Picturebox. Dort tragen Sie fast den gleichen Code ein, den Sie auch für das Malen des Bildes zuvor verwendet haben.

Auf ein Update der Statuszeile können Sie in diesem Zusammenhang verzichten (Listing 8).

In diesem Beitrag haben Sie erfahren, wie Sie ein vollwertiges .NET Windows-Programm erzeugen und dabei die Windows.Forms Bibliothek verwenden.

Windows.Forms brauchen Sie jedes Mal, wenn Sie ein Client Programm für .NET schreiben wollen. Und mit dem Forms Editor können Sie Windows Programme deutlich einfacher zusammensetzen, als das bisher möglich war: Bereichern Sie den Bildbetrachter um Ihre eigene Funktionen, es ist ganz einfach.

SharpDevelop

Eine freie .NET Entwicklungsumgebung

Wer für .NET programmieren möchte, aber vor den Kosten für Microsofts Visual Studio zurückschreckt, der steht nicht vor dem Aus. Zum einen kann man natürlich einfach die Kommandozeilen-Compiler von .NET verwenden, zum anderen gibt es aber auch eine OpenSource Alternative: SharpDevelop.

THOMAS WÖLFER

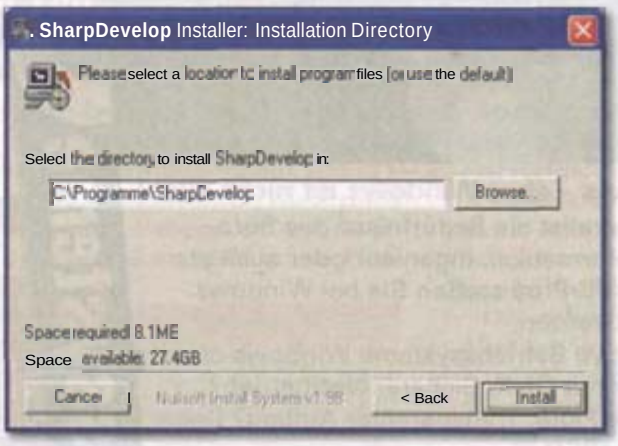
Grundsätzlich ist eines zu SharpDevelop zu sagen: Das Open-Source-Projekt verändert und verbessert sich praktisch laufend weiter. Wenn Ihnen die auf der Heft-CD mitgelieferte Version gefällt, sollten Sie nicht davon zurückschrecken, sich die aktuellste Version von http://www.icsharpcode.net herunterzuladen - aufgrund der Geschwindigkeit, mit der neue Features hinzukommen, wird sich bis zum Erscheinen dieses Heftes mit Sicherheit einiges getan haben.

SharpDevelop ist ein klassisches Open-Source Projekt, allerdings eines, das nicht Programme für Linux herstellt, sondern eine Entwicklungsumgebung für .NET, die unter Windows betrieben werden kann: im Wesentlichen also ein direkter Konkurrent zu Microsofts Visual Studio. Allerdings unterstützt SharpDevelop nur die Entwicklung von C# und VB.NET Programmen - C++-Programmierer bleiben außen vor.

Das wird in erster Linie damit zusammenhängen, dass .NET von sich aus keinen C++ Compiler in der Laufzeitumgebung bietet, denn SharpDevelop greift natürlich auf die mitgelieferten .NET Compiler zurück.

Optisch ist SharpDevelop dem Konkurrenten Microsoft nachempfunden, allerdings haben ja alle Entwicklungsumgebungen heute ein mehr oder minder ähnliches Look&Feel.

SharpDevelop unterstützt nicht nur die Entwicklung von Kommandozeilen-Programmen, sondern hat auch einen grafischen Editor für die Erzeugung von Forms. Sie können also, genau wie im Visual Studio, einfach ein Fenster öffnen und Buttons, Listboxen und andere Kontroll-Elemente darauf platzieren. Anders, als bei Visual Studio, werden die Forms aber nicht direkt in C#-Quellcode abgebildet, sondern liegen in einer Zwischenstufe als XML-Dateien vor, aus denen heraus



DER SHARPDEVELOP INSTALLER MACHT DIE INSTALLATION EINFACH: Optionen sind nur wenige auszuwählen.

SHARPDEVELOP: DIE FEATURE-ÜBERSICHT

SharpDevelop bietet eine erstaunliche Anzahl an Features. Hier die wichtigsten im Überblick:

- Vollständig in C# programmiert, Open-Source mit GPL-Lizenz: Sie können nicht nur das Programm selbst, sondern auch den Quellcode dazu erhalten und untersuchen.
Das Programm lässt sich mit dem normalen C# Compiler von .NET übersetzen.
- Code Completion: Die Entwicklungsumgebung kennt die am Projekt beteiligten Typen und öffnet während des Tippens von Quellcode praktische Hilfefenster, mit denen man zum Beispiel Klassennamen automatisch vervollständigen lassen kann.
- Mehrsprachigkeit: SharpDevelop liegt in mehreren Sprachen vor - Wenn Sie es vorziehen, in einer anderen Sprache als Deutsch zu arbeiten: kein Problem.
- Support für C#, ASP.NET ADO.NET, XML und HTML-Code: Jeweils mit Syntax-Highlighting für die bessere Übersicht im Editor.

dann der C#-Code oder der für VB.NET für die tatsächliche Form erzeugt wird.

Dazu gibt es ein Designer-Fenster mit verschiedenen Reitern: In einem Reiter kann die Form ganz normal mit Drag&Drop-Operationen zusammengesetzt werden, auf den anderen Reitern werden die alternativen Repräsentationen (C#-Code, XML, VB-Code) angezeigt.

Der Forms-Designer ist allerdings eines der neueren Features in SharpDevelop - daher sind in diesem Bereich einige Veränderungen zu erwarten.

• Ein ersten C#-Projekt mit SharpDevelop

Bei SharpDevelop beginnt die Programmierung eines neuen Projekts nicht anders, als im VisualStudio. Der einzige Unterschied ist, dass man sich bei SharpDevelop für eine etwas andere Namensgebung entschieden hat: Ein Project bzw. eine Solution trägt hier die Bezeichnung Project oder Combine - der Unterschied zwischen den Varianten ist in der Online-Hilfe erläutert.

Um ein neues Projekt zu beginnen, wählen Sie den entsprechenden Menüpunkt aus dem Datei-Menü. Daraufhin öffnet sich eine Dialogbox, aus der Sie den gewünschten Projekt-Typen auswählen können. Für einen ersten Testlauf empfiehlt sich ein Windows Forms Projekt - also eines, bei dem eine ganz normale Windows-Anwendung hergestellt werden soll. SharpDevelop legt dann eine Ordnerstruktur für das Projekt an und erzeugt eine Reihe von Dateien - ganz ähnlich, wie das beim Visual Studio der Fall ist. Das Projekt ist direkt danach bereits in Quellcode überführt und kann gestartet werden. Das erreichen Sie mit dem Button, auf dem ein grüner Pfeil abgebildet ist. Der Quellcode wird übersetzt, und Ihre erste mit SharpDevelop erzeugte Form erscheint am Bildschirm.

Viel mehr als dieses Fenster anzuzeigen, kann das Programm noch nicht. Jetzt müssen Sie das Projekt um eine Form mit etwas mehr Funktionalität erweitern. Die neue Form soll mit einigen Kontroll-Elementen aufgefüllt werden. Dazu klicken Sie mit der rechten Maustaste in das Projekt-Fenster und wählen den Befehl *Hinzufügen*. Im daraufhin angezeigten Dialog wählen Sie eine *XmlForm* aus der Kategorie *XmlForm* aus und klicken auf *Erstellen*.

SharpDevelop öffnet dann ein Fenster, in dem die Form grafisch bearbeitet werden kann. In diesem Fenster sehen Sie am unteren Rand die bereits erwähnten Reiter zum Umschalten der Repräsentation der Form. Darüber befindet sich der normale Arbeitsbereich, in dem die Form so dargestellt wird, wie sie später zur Laufzeit des Programms auch angezeigt werden soll.

Danach klicken Sie einmal in das Fenster mit der Form - daraufhin erscheinen die zur Verfügung stehenden Komponenten, die auf der Form platziert werden können, automatisch in der Toolbox. Dabei handelt es sich um die gleichen Komponenten, wie sie auch beim Visual Studio zur Verfügung stehen - es sind eben die in .NET vorhandenen Komponenten.

SHARPDEVELOP AUF DER HEFT-CD

Auf der Heft-CD zu diesem Sonderheft finden Sie eine komplette Version von Sharpevelop mit der Sie sofort loslegen können. Zusätzlich gibt es noch zwei Beispielprojekte. Das kleinere davon enthält genau die im Beitrag angesprochene Beispiel-Form, das größere enthält eine etwas ausführlichere Form auf der verschiedenen Kontroll-Elemente demonstriert werden.

Natürlich ist nicht alles, was man zu Sharp Develop sagen kann, mit diesem Beitrag, der Online-Hilfe und den Texten auf der Homepage von SharpDevelop erläutert. Darum gibt es zusätzlich ein Support-Forum in dem Sie Ihre Fragen und Kommentare loswerden können. Das Forum finden Sie unter http://www.icsharpcode.net/OpenSource/SD/Forum/.

DAMIT DAS 'CODECOMPLETION'-FEATURE zur Verfügung steht, muss SharpDevelop einmal eine Datenbank mit den benötigten Informationen anlegen.

LISTING 1:

```
public class CreatedObject0 : System.Windows.Forms.Form {

  private System.Windows.Forms.Button button;

  public CreatedObject0() {
    //  Must be called for initialization
    this.InitializeComponents();
}

  /// <summary>
  ///     This method was autogenerated - do not change the Contents
➥manually
  /// </summary>
  private void InitializeComponents() {
    //
    //  Set up generated class CreatedObject0
    //
    this.SuspendLayout();
    this.Name = "CreatedObject0";
    this.Size = new System.Drawing.Size(168, 120);

    //
    //  Set up member button
    //
    button = new System.Windows.Forms.Button();
    button.TabIndex = 0;
    button.Name = "button";
    button.Text = "button";
    button.Location = new System.Drawing.Point(32, 24);
    this.Controls.Add(button);
    this.ResumeLayout(false);
    }
}
```

Um nun zum Beispiel einen Button auf der Form zu platzieren, kann einfach das Button-Elemente aus der Toolbox auf die Form gezogen werden.

Die Eigenschaften des Buttons (und der Form) können dann einfach in der Eigenschaften-Ansicht von SharpDevelop verändert bzw. eingestellt werden.

Was SharpDevelop an dieser Stelle fehlt - zumindest im Vergleich zu VisualStudio – ist eine Möglichkeit, Event-Handler direkt im Forms-Designer anzugeben. Dazu müssen Sie bei SharpDevelop den ganz normalen Quellcode-Editor verwenden.

Die erzeugte Form erhält dabei übrigens einen automatisch erzeugten Namen, und unter diesem Namen können Sie die Form dann im restlichen Programm verwenden. Ein Muster dazu finden Sie im Beispielprojekt zu diesem Beitrag auf der Heft-CD.

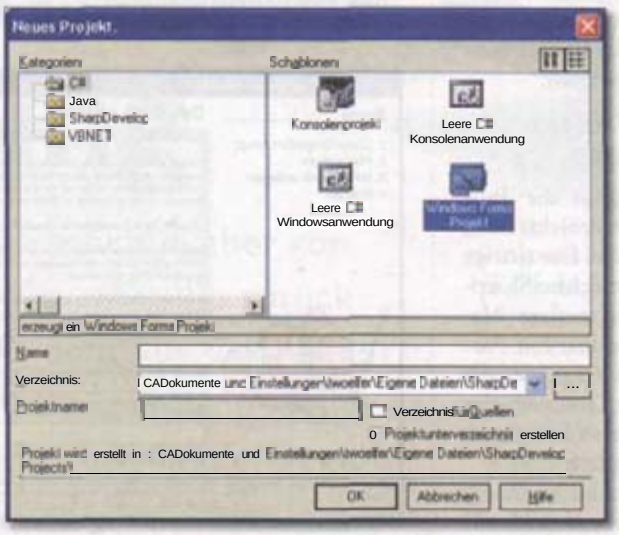
Der C#-Quellcode, der für die Form erzeugt wird ,entspricht weitestgehend dem, der auch von Visual Studio erzeugt wird (Listing 1).

Wie man am Quellcode erkennen kann, wird zunächst eine neue Klasse unter dem Namen 'CreatedObjectO' angelegt. Diesen Namen können Sie natürlich verändern, indem Sie die Form anklicken und dann in das Eigenschafts-Fenster wechseln. Dort suchen Sie nach der Eigenschaft *Name* und geben den gewünschten Namen ein. Dieser findet sich dann sowohl im zugehörigen XML-Code als auch im C#- bzw. im VB.NET-Code wieder.

Der Konstruktor des Objekts ruft dann die Methode *InitializeComponent()* auf, und dort werden die einzelnen Eigenschaften zu denen auch die auf der Form platzierten Elemente zählen, festgelegt.

Diesen Quellcode zu verändern - zum Beispiel, um die Größe der Form auf einen anderen Wert festzulegen - bringt nichts: Lassen Sie die Hände davon. Der komplette Code innerhalb von *Initializecomponent()* wird immer wieder neu automatisch generiert und zwar anhand der in der Eigenschaften-Ansicht eingestellten Werte. Mit anderen Worten: Wenn Sie eine Eigenschaft der Form verändern wollen, dann sollten Sie das im 'Eigenschaften'-Fenster tun und nicht innerhalb von 'InitializeComponent()'.

Wenn Sie Ihre Form mit einem Icon ausstatten wollen, werden Sie feststellen, dass SharpDevelop auch an dieser Stelle ein Werkzeug vermissen lässt: Es gibt weder einen Bitmap- noch einen Icon-Editor. Um solche Dateien zu erstellen, müssen Sie also an anderer Stelle nach geeigneten Werkzeugen suchen und diese dann notfalls als 'Externe Werkzeuge' in SharpDevelop integrieren. Sie können sich natürlich auch aufraffen und einen eigenen Bitmap-Editor mit SharpDevelop programmieren, um diesen dann dem Projekt zur Verfügung zu stellen: Die anderen SharpDevelop Anwender werden es Ihnen danken!



MIT EINEM WINDOWS-FORMS PROJEKT erzeugen Sie ein normales Windows-Programm, können aber den Forms-Designer zum Gestalten des Fensters verwenden.

• Andere Werkzeuge in SharpDevelop

SharpDevelop stellt dafür aber eine ganze Reihe an anderen Werkzeugen und Ansichten zur Verfügung. So gibt es neben der Projekt-Ansicht auch eine Ansicht, in der die am aktuellen Projekt beteiligten Klassen dargestellt werden - und zwar komplett mit allen Methoden und Eigenschaften. Ferner gibt es eine Explorer-Ansicht, die Einsicht in das Dateisystem bietet, die Eigenschaften-Ansicht, die Sie schon kennen, und eine Hilfe-Ansicht, mit der Sie die integrierte Hilfe lesen können. Es ist allerdings zu empfehlen, auf jeden Fall einmal die Walk-Throughs und die Feature-Beschreibungen auf der Homepage von SharpDevelop durchzulesen, denn nicht alles was das Programm kann, findet sich auch in der Hilfe wieder.

Darüber hinaus kommt SharpDevelop mit einem eingebauten Dokumentations-Generator namens 'Ndoc' und einem System für's Testen Ihrer Anwendung. Dabei können Test-Skripts erstellt und wiederverwendet werden, so dass Regressions-Tests einfach sind. Allerdings sind die aktuellen Versionen sowohl vom Test- als auch vom Dokumentationsprogramm noch etwas wackelig auf den Beinen: Dieser Zustand wird sich aber sicherlich bald ändern.

Ein besonders interessantes Detail findet sich bei SharpDevelop unter den Projekt-Optionen. Hier können Sie nämlich nicht nur die einzelnen Compiler-Optionen einstellen, Sie haben hier auch die Möglichkeit, das Zielsystem Ihres Projekts festzulegen. Von Haus aus könnte man natürlich davon ausgehen, dass das Zielsystem von vorneherein klar ist, schließlich soll das Programm unter Microsofts .NET-Umgebung betrieben werden.

Das ist aber nicht die einzige Umgebung, die SharpDevelop versorgt. Statt dessen kann auch Mono als Zielsystem ausgewählt werden. Bei Mono handelt es sich um eine andere OpenSource-Implementierung im .NET Bereich - und zwar um eine OpenSource-Version von .NET, die zum Beispiel auch unter Linux eingesetzt werden kann. Mit anderen Worten: Mit SharpDevelop schreiben Sie nicht nur Programme für Windows mit .NET - Sie können mit dem gleichen Quellcode auch ein Programm erzeugen, das mit Hilfe von Mono unter Linux betrieben werden kann: Das VisualStudio von Microsoft kann das - wie überraschend - nicht.

Was man bei SharpDevelop schmerzlich vermisst, ist ein eingebauter Source-Level-Debugger, mit dem das eigene Programm auf Fehler untersucht werden kann. Doch was nicht ist, kann ja noch werden - die Entwicklung von SharpDevelop geht, wie erwähnt, ja relativ zügig voran. Natürlich kann trotzdem debuggt werden: Dazu verwendet man den Gui-Debugger aus dem .NET SDK *DbgClr.exe* - nur ist der eben nicht so angenehm in den SharpDevelop Editor eingebaut, wie das bei den Debuggern von anderen Entwicklungsumgebungen der Fall ist. ® UR

Die harte Tour

Kommandozeilen-Compiler verwenden

Auch wer kein Visual Studio oder eine anderen Entwicklungsumgebung für .NET besitzt, kann Programme für .NET schreiben: Und zwar mit C#, mit VB.NET und mit JavaScript.

THOMAS WÖLFER

Man kann ohne große Kosten für .NET programmieren, denn die Compiler für C#, VB.NET und Java sind mit allem, was dazu gehört, beim kostenlosen .NET Framework SDK dabei: Man muss es nur herunterladen und installieren.

Das SDK umfasst aber über 130 Megabyte - PC Magazin-Sonderheft-Leser können das SDK einfach direkt von der Heft-CD aus installieren. Die Installation des .NET Frameworks - selbst wenn Sie nur die redistributable Komponente, also nicht das ganze SDK installieren - dauert eine ganze Weile. Dabei ist der letzte Schritt im Fortschrittsmelder besonders listig: Setup zeigt dabei nämlich an, dass noch '0' Sekunden fürs Installieren benötigt werden, braucht dann aber doch einige Minuten, zumindest auf nicht gar so schnellen Rechnern. Dabei wird eine Meldung angezeigt, die besagt das 'Code' erzeugt werden würde.

Das ist ein Merkmal von .NET, das besonderer Erwähnung bedarf. Bei .NET ist es nämlich möglich, ausgelieferte Programme erst auf dem Zielsystem zu übersetzen. Dabei kann dann das komplette Zielsystem, einschließlich seiner Speicher-Ausstattung, der gelaDenen Komponenten und der CPU-Architektur beim Optimieren des Codes berücksichtigt werden. Ziel der Sache ist, dass möglichst optimaler native Code in allen Fällen herauskommt, und der sieht bei einer Pentium 4-CPU deutlich anders aus als bei einer Pentium2-CPU.

Lange Rede kurzer Sinn: Wenn Sie mit den .NET-Compilern ein Programm für .NET übersetzen, dann übersetzen Sie es im Normalfall nicht in ausführbaren X86 Code. Statt dessen erzeugen die Compiler eine Art Byte-Code, der erst auf dem eigentlichen Zielsystem in native code umgewandelt wird. Dieser Byte-Code hat auch einen Namen. Es handelt sich um MSIL (Microsoft Intermediate Language). Die MSIL ist gut dokumentiert, und es existieren sowohl Assembler wie auch Disassembler dafür im SDK.

Doch darauf soll an dieser Stelle nicht weiter eingegangen werden, denn für den normalen Bedarf verhält sich ein mit den .NET Compilern übersetztes Programm genau so, wie man erwarten würde: Ruft man es auf, dann wird es ganz

DAS .NET FRAMEWORK wird im Windows-Ordner installiert. Ein bisschen suchen muss man schon, wenn man den kompletten Pfad wissen will.

C# COMPILER-OPTIONEN

Anders als beim VB Compiler zeigt der C# Compiler seine Optionen nicht an, wenn man ihn ohne Parameter startet. Hier muss man statt dessen die Option */help* verwenden, um eine Liste der unterstützten Parameter und Optionen zu erhalten.

Auch beim C# Compiler gibt es die Optionen */out* und */target* - Die Beschreibung dazu finden Sie im Kasten „Visual Basic kompilieren", denn der C# Compiler funktioniert an dieser Stelle identisch. Auch bei der Codeerzeugung stimmen die C#- mit den VB-Optionen mehr oder minder überein. So schaltet */optimize+* den Optimizer ein und */debug-* die Erzeugung von Debug-Informationen aus.

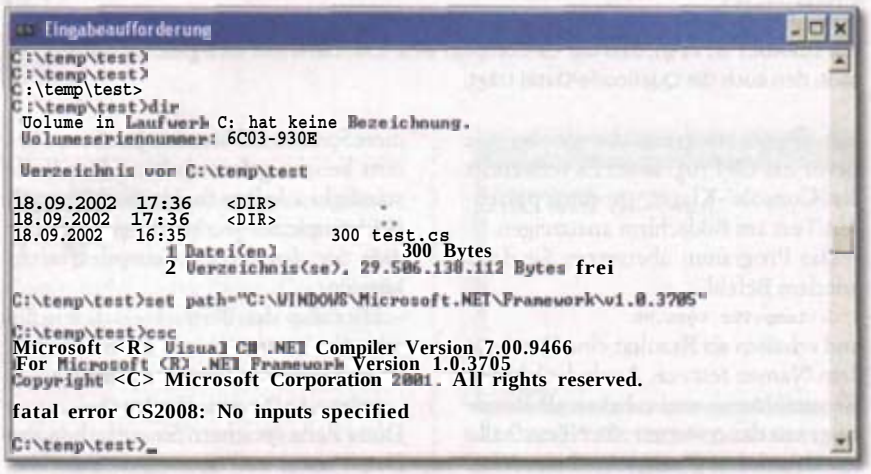
normal gestartet. Dass .NET im Hintergrund noch nativen Code erzeugt, ist dafür nicht weiter von Belang - und man wird das im Normalfall auch gar nicht bemerken.

Nachdem Sie das SDK installiert haben, sollten Sie noch nach eventuell zwischenzeitlich aufgetauchten Service Packs Ausschau halten. Auf der Heft-CD finden Sie Service Pack 1, es ist aber durchaus möglich das mittlerweile weitere Service Packs zur Verfügung stehen.

Ist das SDK installiert, können Sie gleich loslegen. Allerdings müssen Sie noch ein paar Dinge überprüfen - am wichtigsten ist dabei, dass der Pfad auf die SDK-Programme auch gesetzt ist. Das SDK wird normalerweise nach

```
\Windows\Microsoft.NET\
➥Framework\VERSION
```

installiert, wobei VERSION natürlich die Version des Frameworks angibt. Dort befinden sich auch die Kommandozeilencompiler, wie zum Beispiel der css.exe, der für's Übersetzen von C#-Programmen benutzt wird. Dieses Verzeichnis muss in Ihrem Pfad liegen, sonst macht das Übersetzen Probleme.



DAMIT DIE .NET-COMPILER von der Kommandozeile aus benutzbar werden, muss man den Pfad zu diesen Werkzeugen angeben. Danach kann man die Compiler aufrufen.

Ist das sichergestellt, dann können Sie einen ersten Test wagen. Dazu brauchen Sie natürlich ein kleines C#-Programm, und das sieht wie Listing 1 aus.

Mit diesem Quellcode lernen Sie eigentlich auch schon alles kennen, was

LISTING 1:

```
using System;
dass HelloWorld
{
  public static int
➥Main(String[] args)
  {
    Console.WriteLine
➥("Hello World from CSharp");
    return 0;
  }
}
```

man an grundlegenden Dingen über C# wissen muss: Ein C#-Programm, das für die Konsole gedacht ist, braucht mindestens eine Klasse (deren Name ist egal) und eine Main()-Funktion. Die muss 'static' und public sein - es handelt sich um den Eintrittspunkt für das Programm.

Den abgebildeten Quellcode geben Sie nun einfach in Ihrem Editor ein - dazu reicht auch notepad - und speichern ihn auf der Platte. Zum Beispiel unter dem Namen *test.cs*. Dann werfen Sie den C#-Kommandozeilen-Compiler an:

```
C:\temp>csc test.cs
```

INSTALL-TIME COMPILATION, LOAD-TIME COMPILATION, NATIVE CODE

Von Haus aus generieren .NET-Compiler keinen nativen X86 Code, sondern nur einen Byte-Code in der MSIL (Microsoft Intermediate Language). Dieser Byte-Code wird dann zur Ladezeit der Anwendung von der .NET-Laufzeit in nativen Code übersetzt. Das Programm besteht also erst dann aus echtem Maschinencode, wenn es für die Ausführung geladen wird. Der Mechanismus ist also sehr ähnlich wie das von Java bekannte 'Just-In-time Compilation'.

Dieses Vorgehen hat verschiedene Vor- und Nachteile. Der größte Vorteil ist ganz klar bei der Optimierung zu finden. Wird der native Code erst auf der Plattform erzeugt, für die er gedacht ist, dann kann diese Plattform vollständig in die Optimierungsüberlegungen eingehen. So kann man das gleiche Programm auf Rechner (A) der mit 128 MB RAM und einem Pentium 4 ausgestattet ist, eben anders optimieren als auf Rechner (B) mit nur 64 MB RAM und einer anderen CPU.

Der größte Nachteil ist natürlich, dass die Ladezeit des Programms deutlich länger wird: Schließlich muss der Programmcode ja erst erzeugt werden. Je größer das Programm nun ist, um so länger wird die Wartezeit des Benutzers, während das Programm geladen wird.

Man kann dies bei .NET nun auf zwei Arten umgehen: Zum einen kann man die Compiler anweisen, keinen MSIL-Code, sondern direkt nativen Code zu erzeugen. Das hat natürlich den Nachteil, dass der erzeugte Code keinerlei Optimierungen für die verschiedenen möglichen Zielplattformen enthalten darf. Das bedeutet letzthin nichts anderes, als dass Code für relativ alte CPUs erzeugt werden muss - und dass die Ausstattung des Zielsystems nicht berücksichtigt werden kann: Man muss also von einem Minimal-System ausgehen.

Besser ist es da, die dritte Variante zu verwenden. Dabei wird das Programm während der Setup-Phase kompiliert. So geht beispielsweise auch das Installationsprogramm für das .NET-Framework selbst vor. Dabei erzeugt der Compiler zur Compilierzeit zwar MSIL-Code, beim Installieren des Programms auf der Zielplattform wird dann aber Native Code für eben diese Zielplattform generiert. Das führt dann dazu, dass die Ladezeit des Programms auf der Zielplattform völlig normal ist, aber trotzdem Code vorliegt, der für die Zielplattform optimiert wurde.

Damit erkauft man sich aber ein anderes Problem: Ändert sich die Ausstattung der Zielplattform (mehr RAM, ...), so wird diese Änderung natürlich nicht berücksichtigt - denn es liegt ja bereits nativer Code vor. Genauso verhält es sich, wenn das Programm vom Anwender auf einen anderen Rechner kopiert wird. Hier können sogar deutlich dramatischere Probleme auftauchen. Zum Beispiel dann, wenn der optimierte Code Sequenzen für die auf dem einen Rechner vorliegende CPU enthält, die von der CPU auf dem anderen Rechner gar nicht unterstützt wird. Letzteres Problem kann man aber natürlich einfach dadurch umgehen, dass man die Installation per Setup-Programm auf die ein- oder andere Art erzwingt. Dadurch verliert man aber natürlich die Möglichkeit des 'Xcopy deployments' - das Programm muss eben doch wieder per Installationsprogramm auf den Zielrechner transportiert werden.

Der Compiler meldet sich mit einem kurzen Text und beginnt dann den Quellcode zu übersetzen. Das geht relativ schnell - und als Resultat haben Sie danach eine zweite Datei mit dem Namen *test.exe*. Dabei handelt es sich schon um das übersetzte Programm. Dass es auch funktioniert, können Sie einfach verifizieren, indem Sie das Programm ausführen: Es sollte den Text 'Hallo World from Csharp' ausgeben.

Den CSC können Sie mit einer recht großen Anzahl an Parametern füttern. Die wichtigsten davon finden Sie im Kasten 'C#Compiler-Optionen' erläutert.

• Mehrsprachigkeit

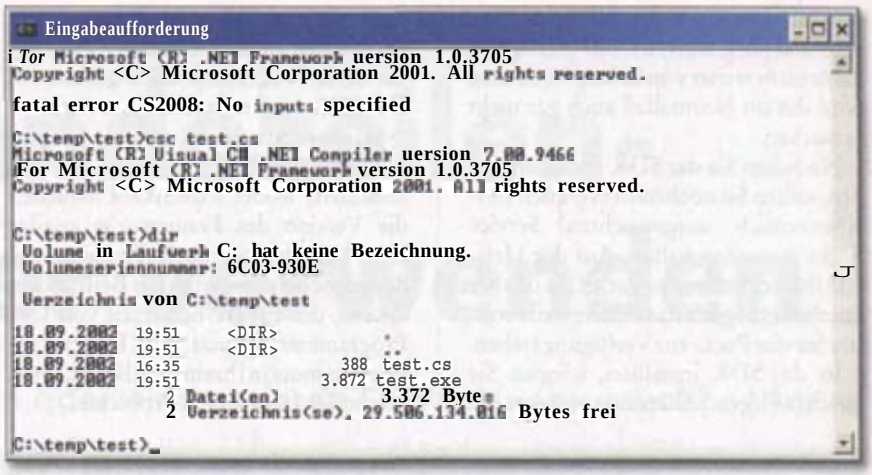
Das .NET Laufzeitsystem ist mit mehreren Compilern ausgestattet. Es gibt einen Compiler für Jscript - einer JavaScript-Abart– und Visual Basic. Auch diese sollten Sie ausprobieren, und darum finden Sie im Folgenden noch ein paar kurze Beispielprogramme zum Testen der beiden anderen Übersetzer. Begonnen wird mit Visual Basic.

Das VB-Programm hat recht große Ähnlichkeit mit dem C#-Programm und sieht wie in Listing 2 aus:

Im Wesentlichen besteht das Ganze also aus einem 'Module', das seinerseits das Unterprogramm 'sub Main()' enthält. Dieses tut genau das gleiche, wie zuvor das C# Programm: Es verwendet die 'Console'-Klasse, um einen passenden Text am Bildschirm anzuzeigen.

LISTING 2:

```
Module Module1
    Sub Main()
        System.Console.WriteLine
➥("hallo aus vb...")
    End Sub
End Module
```



PER DEFAULT ist es so, dass der *Ctt* Compiler eine .EXE-Datei mit dem gleichen Namen baut, den auch die Quellcode-Date trägt.

Das Programm übersetzen Sie dann mit dem Befehl

```
C:\temp>vbc test.vb
```

und erhalten als Resultat eine Datei mit dem Namen *test.exe*. Auch die können Sie ausführen - und erhalten als Resultat genau das erwartete: Der Text 'hallo aus vb' wird angezeigt. Auch der VBC kann mit einer Vielzahl an Parametern bestückt werden. Die wichtigsten finden Sie im Kasten: 'VisualBasic kompilieren'

Schließlich enthält das SDK noch einen Compiler für eine Sprache, die man ansonsten als interpretierte Sprache kennt: Jscript. Wenn Sie sich schon einmal mit der Programmierung von Webseiten befasst haben, dann werden Sie diese Sprache in Form von JavaScript bereits kennen gelernt haben. Der Vollständigkeit halber finden Sie hier noch ein komplettes Jscript Programm, mit dem Sie den Jscript-Compiler testen können.

Hier also das Beispiel - das, wie Sie schnell sehen werden, deutlich einfacher ist als die beiden vorgegangenen:

```
print("hallo von jscript");
```

Diese Zeile speichern Sie einfach in der Dateit *test.js* und übersetzen diese Daten mit

```
C:\tem>jsc test.js
```

Wiederum erhalten Sie als Resultat eine text.exe - und wiederum führt das Ausführen davon zum erwarteten Resultat: Der Text 'hallo von jscript' wird angezeigt.

Damit sind Kommandozeilen-Compiler erklärt. © U R

VISUAL BASIC KOMPILIEREN

Visual Basic Quellcodes zu übersetzen, ist ganz einfach - im einfachsten Fall lautet der Aufruf des Compilers:

```
vbc NameDerQuellcodeData.vb
```

In diesem Fall geht der Compiler davon aus, dass eine-.EXE Datei erzeugt werden soll. Man kann aber eine ganze Menge an Parametern beim Kompilieren angeben. Die Anzahl an möglichen Parameter ist zwar nicht ansatzweise so umfangreich wie die bei C++-Programmen - trotzdem gibt es eine ganze Reihe davon.

Wer sich schnell einen Überblick verschaffen will, der gibt einfach den Namen des Compiler-Programms ein und erhält dann eine Anzeige aller möglichen Parameter.

Die wichtigsten davon sind die folgenden:

```
/out:DateiName
```

Mit diesem Parameter kann man bestimmen, wie die erzeugte Datei heißen soll. Per Vorgabe erzeugt der Compiler eine .EXE-Datei mit dem gleichen Namen, den auch die Quellcode-Datei trägt. Hat der Quellcode also den Namen *test.vb,* dann erzeugt der Compiler die Datei *test.exe.*

```
/target:TARGET
```

Mit diesem Schalter kann man angeben, was für ein Programm erzeugt werden soll. Dabei ist TARGET ein String, der das Zielprogramm beschreibt.

TARGET kann die Werte 'exe', 'winexe', 'library' und 'module' annehmen. Mit 'exe' gibt man an, dass eine Konsolen-Anwendung erzeugt werden soll, und 'winexe' spezifiziert, dass das erzeugte Programm ein Windows-Programm mit Fenstern sein soll. Mit dem Wert 'library' erzeugt der Compiler eine Assembly, und 'module' erzeugt ein Modul, das nachträglich zu einer Assembly hinzugefügt werden kann.

Eine weitere wichtige Option ist die Option

```
/reference:Dateien.
```

Damit gibt man an, welche Referenzen auf andere Assemblies verwendet werden sollen. Will man beispielsweise auch Objekte aus Windows.Forms verwenden, dann muss das dem Compiler per */reference*-Option mitgeteilt werden.

Fürs Generieren von Code gibt es die wichtigen Optionen

```
optimize+
```

beziehungsweise

```
optimize-
```

Damit kann man den Code-Optimizer ein-, beziehungsweise ausschalten. Ebenso schaltet man mit

```
debug+
```

beziehungsweise

```
debug-
```

die Erzeugung von Debug-Informationen ein- oder aus.

Auch unter Windows 9.x

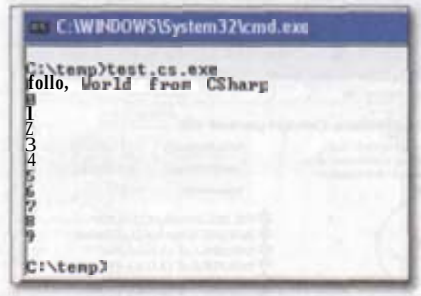
C# Compiler für alle

Die kostenlosen Kommandozeilen-Compiler des .NET-Frameworks stehen nur dann zur Verfügung, wenn man das SDK installiert hat. Windows 9.x-Programmierer bleiben also zunächst außen vor. Grund genug ein kleines C#-Projekt zu starten, mit dem auch unter Windows 9.x kompiliert werden kann. Ganz recht, hier bauen Sie Ihren eigenen Compiler.

THOMAS WÖLFER

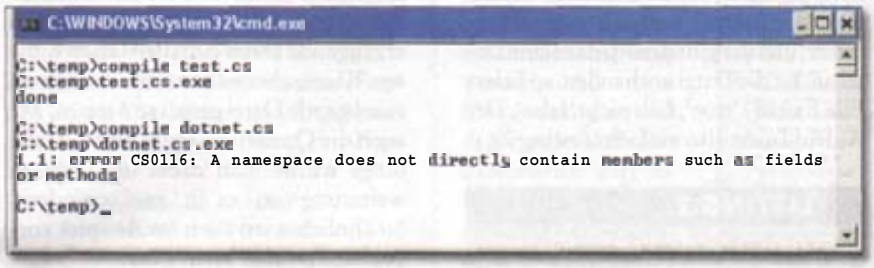
Natürlich ist es etwas übertrieben zu behaupten, im Artikel würde ein kompletter C#-Compiler inklusive Parser, Code Generator und allem, was dazu gehört implementiert. Das wäre genau genommen sogar ziemlich unsinnig - denn der C#-Compiler ist ja Teil des Frameworks, und das steht auch unter Windows 09.x zur Verfügung.

Wenn Sie also unter Windows 9.x arbeiten und über keinen Zugriff auf die Kommandozeilen-Compiler des SDK verfügen, stehen Sie also mit Hilfe dieses Beitrags nicht vor dem Nichts, denn der hier entwickelte Compiler versetzt Sie sehr wohl in die Lage, die ersten Schritte mit C# zu wagen.



DAS ÜBERSETZTETEST-PROGRAMM läuft sofort und klaglos: Damit kann man also auch unter Windows 9.x C#-Code ans Laufen bringen.

Dabei werden gleich zwei Versionen des Compilers entwickelt. Die eine stellt eine einfache Kommandozeilen-Variante dar, die eine einzelne Quellcode-Datei in eine ausführbare Datei umwandelt. Die zweite Variante ist eine GUI-Variante, die ebenfalls ein Executable erzeugt, aber die auch einen kleinen Editor, ein Fenster für Fehlermeldungen und die Möglichkeit zum Laden und Speichern von Quellcode-Dateien anbietet. Diese zweite Variante wird anderer Stelle in diesem Sonderheft noch weiterentwickelt: Dort kommen dann Compiler-Optionen und zugehörige Elemente hinzu, die das Programm zu einer einfachen aber doch funktionsfähigen Entwicklungssoftware ausbauen.



DER SELBSTGEBAUTE KOMMANDOZEILEN-COMPILER kann C#-Quellcodes übersetzen und zeigt auch eventuell notwendige Fehlermeldungen an.

• Der Mini-Compiler

Für den ersten Mini Kommandozeilen-Compiler benötigen Sie zunächst ein neues C#-Kommandozeilen-Projekt. Das legen Sie zum Beispiel mit dem Visual Studio mit Wizard-Unterstützung an, oder Sie verwenden einfach den Quellcode von der Heft-CD.

Im Zuge der Entwicklung werden Sie natürlich den Framework C#-Compiler benötigen, und darum sollten Sie als Erstes eine Referenz zu diesem Compiler anlegen. Der Compiler befindet sich im Modul 'cscompmgd', das Sie einfach Ihrer Liste an Referenzen hinzufügen.

Im nächsten Schritt verwenden Sie ganz zu Beginn des Programms das 'using'-Keyword, um sich später etwas Schreibarbeit zu sparen.

Mit *using* legen Sie in C# fest, welche Namespaces Sie verwenden wollen. Wenn Sie zum Beispiel *using System* angeben, dann können Sie die 'string'-Klasse aus diesem Namespace direkt verwenden, ohne dass Sie erst mühevoll den Namespace mit angeben müssen. Es darf dann also im Quellcode

```
string B = „test"
```

statt

```
System.string = „test"
```

lauten.

Für das Compiler-Beispiel brauchen Sie die Namespaces 'System', 'Microsoft.Csharp' und 'System.CodeDom.Compiler'. Außerdem sollen ja Dateien auf der Kommandozeile übersetzt werden . Dazu muss man diese lesen. Die IO-Funktionen befinden sich im Namespace System.IO, und auch dieser wird also benutzt. Die ersten Zeilen im Quellcode lauten daher:

```
using System;
using Microsoft.CSharp;
using System.CodeDom.Compiler;
using System.IO;
```

Das vom Wizard erzeugte Projekt beinhaltet bereits eine Main() Funktion, die auch schon mit den richtigen Attributen ausgestattet ist.

```
[STAThread]
static void Main (string[] args)
```

Hier beginnt das Programm. Main() bekommt die Kommandozeilen-Parameter des Programms in einem Array aus Strings übergeben, und diese Kommandozeilenparamter sollten den Pfad zur übersetzenden Quellcode-Datei enthalten. Der spätere Aufruf des Programms lautet also:

```
C: >compile
➥NameDerQuellcodeDatei.cs
```

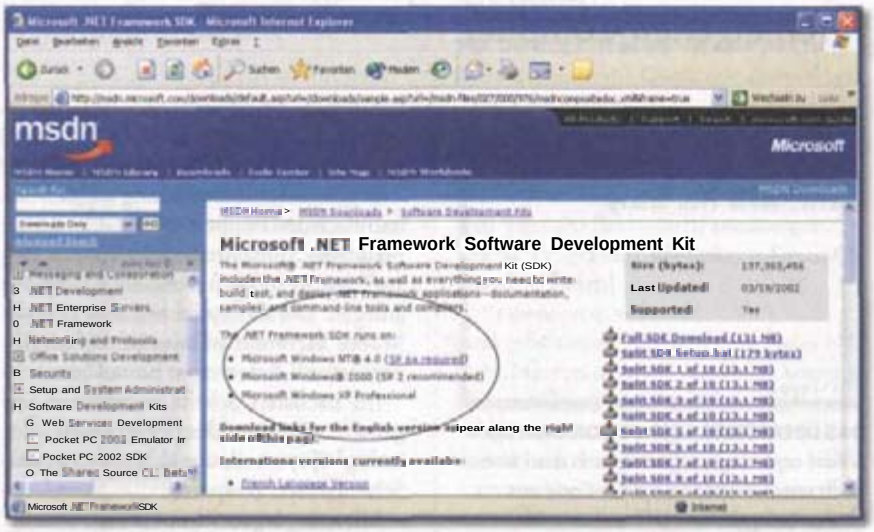
Zunächst ist also sicherzustellen, dass die Datei angegeben wurde und dass die angegebene Datei auch existiert. Das geht wie folgt:

Sie sorgen zuerst dafür, dass das an Main() übergebene Array auch etwas enthält. Ist das nicht der Fall, dann wurde kein Parameter beim Aufruf des Programms angegeben und sie können einfach eine Fehlermeldung ausgeben und das Programm beenden (Listing 1).

COMPILERPARAMETER IM DETAIL

Beim Übersetzen eines Programms mit den .NET-Compilern übergeben Sie dem ICompile Interface nicht nur das zu übersetzende Programm, sondern auch einen Satz an Parametern. Das erfolgt mit Hilfe der CompilerParameters-Klasse. Die CompilerParameters-Klasse kapselt im Wesentlichen genau die Parameter, die Sie auch an die Kommandozeilen-Compiler des SDK weitergeben können.

Einige Parameter werden aber schlicht und ergreifend nicht mit Eigenschaften einer Instanz dieser Klasse abgedeckt - dafür gibt es aber den Familien-Universalbenutzer-Parameter 'CompilerOptions'. Hier können Sie einfach einen String zusammensetzen, der genau die Parameter enthält, die Sie auch auf der Kommandozeile übergeben würden. Dieser String wird dann von der Implementierung des ICompiler Interfaces ausgewertet: Ganz so, als wären die Optionen auf der Kommandozeile übergeben worden.



DAS PROBLEM: Das .NET SDK läuft nicht unter Windows 9.x: Hier muss man also selber Hand anlegen, wenn man einen kostenlosen Compiler haben möchte.

LISTING 1:

```
if( args.Length == 0)
{
   Console.WriteLine("Usage:
   ➥ compile FileName.cs
   ➥(only compiles csharp");
   return;
}
```

Die Länge des Arrays ist einfach eine Eigenschaft, die direkt per 'Length' abgefragt werden kann. Ist die Länge '0', dann enthält das Array nichts - es wurde also kein Dateiname angegeben. Diese Fall quittieren Sie, wie abgebildet, mit einer Fehlermeldung.

Im nächsten Schritt muss nun sichergestellt werden, dass die angegebene Datei auch existiert. Das kann man mit einer (statischen) Methode der File-Klasse tun. File.Exists() erwartet einen Parameter, und der gibt den Pfad auf eine Datei an. Ist die Datei vorhanden, so liefert File.ExistsQ 'true', falls nicht 'false'. Der Aufruf lautet also einfach (Listing 2):

LISTING 2:

```
if ( !File.Exists( args[0]))
{
   Console.WriteLine ("The file
   ➥'" + args[0] + "' does
   ➥not exist");
   return;
}
```

Nun ist also bekannt, dass die angegebene Datei existiert. Damit die Datei übersetzt werden kann, muss der in der Datei befindliche Quellcode ausgelesen werden. Zum Lesen von Dateien gibt es im .NET-Framework eine ganze Menge Methoden.

Hat man den Namen der Datei als String vorliegen, so verwendet man am einfachsten einen StreamReader. Die StreamReader-Klasse kann aus Textdateien lesen und den Inhalt einer Textdatei als String zurückliefern. Genau das wird benötigt, sodass Sie folgenden Quellcode verwenden können um die Datei zu lesen (Listing 3):

LISTING 3:

```
StreamReader sr = new
➥StreamReader( args[0]);
string source = sr.ReadToEnd();
```

Jetzt liegt der Inhalt der Quellcode-Datei im String 'source' vor. Langsam wird es also spannend, denn die Implementierung nähert sich offensichtlich mit großen Schritten dem Übersetzungsteil.

Dafür ist aber noch Vorarbeit zu leisten. Diese Vorarbeit besteht im Wesentlichen darin, dass Sie sich noch einen passenden Namen für die zu erzeugende Datei einfallen lassen müssen. Klassischerweise würde man die zu erzeugende Datei genau so nennen, wie auch die Quellcode-Datei heißt. Allerdings würde man dabei die Dateierweiterung von '.cs' in '.exe' verändern. So ähnlich wird auch im Beispiel vorgegangen - aber nicht ganz.

Um die Sache möglichst einfach zu halten, wurde im Beispiel nämlich auf Schnickschack verzichtet - und darum wird als Name der erzeugten Datei einfach der Original-Name einschließlich Dateierweiterung verwendet, und dieser Name wird schlicht und ergreifend um die Erweiterung .EXE erweitert. Die Quellcode-Datei 'test.cs' wird also in die ausführbare Datei 'test.cs.exe' übersetzt. Nicht schön, aber praktikabel. Zu diesem Zweck müssen erst einmal der Name und Pfad der Quellcode-

Datei ermittelt werden. Das ist aber ganz praktisch, denn bei der Gelegenheit können Sie gleich noch überprüfen, ob es sich bei der auf der Kommandozeile angegebenen Datei um eine C#-Datei handelt oder nicht. C#-Dateien haben per Konvention die Erweiterung .cs, und das prüfen Sie gleich noch nach.

Informationen wie Datei-Namen, Pfade und Erweiterungen kann man in .NET am einfachsten mit dem FileInfo-Objekt ermitteln. Dieses Objekt erhält im Konstruktor den Pfad auf eine Datei und kann danach über die verschiedenen Eigenschaften dieser Datei befragt werden.

```
Fileinfo fi =
➥ new FileInfo( args[0]);
```

Zunächst einmal prüfen Sie also, ob es sich bei der Datei um eine .cs-Datei handelt. Falls nicht, gibt es erneut eine Fehlermeldung:

```
if( fi.Extension!=".cs")
   Console.Write("Kann nur
   ➥.cs Dateien kompilieren...");
```

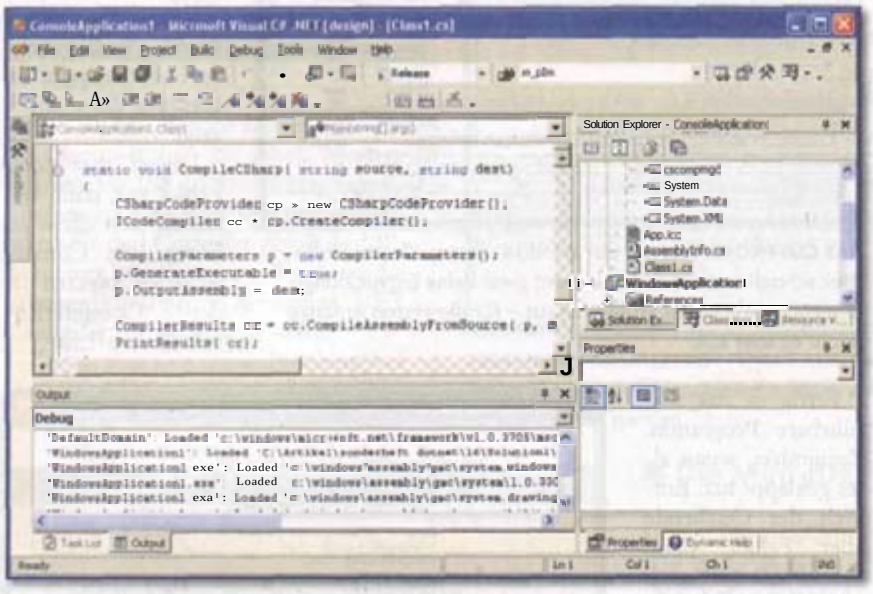
Ist es aber doch eine .cs-Datei, dann bauen Sie den Namen der Ziel-Datei zusammen. Der setzt sich aus dem Ursprungspfad, dem Ursprungsnamen und der Erweiterung .EXE zusammen:

```
string sDest = fi.Directory
➥.FullName + "\\" + fi.Name +
➥ ".exe";
```

Jetzt sind alle Informationen da – und das Beispielprogramm ruft deshalb nun eine separate Methode zum Übersetzen von .cs-Dateien auf. Und zwar deshalb, weil man den Kommandozeilen-Compiler dadurch leicht so erweitern kann, dass er auch andere Quellcode-Dateien übersetzen kann. Der Aufruf lautet wie folgt:

```
CompileCSharp( source, sDest);
```

Die CompileSharp()-Funktion ist nun für verschiedene Dinge zuständig. Zum einen muss sie den übergebenen Quellcode natürlich übersetzen. Dann muss der erzeugte Code in der ebenfalls angegebenen ausführbaren Datei gespeichert werden. Konnte der Quellcode hingegen nicht übersetzt werden, dann müssen natürlich entsprechende Fehlermeldungen angezeigt werden. All diese Dinge sind aber mit dem durch das .NET-Framework angebotenen Klassen sehr einfach zu erledigen.



DAS BEISPIELPROJEKT enthält den Kommandozeilen-Compiler und die GUI-Variante. Beides gibt es natürlich auch in vorkompilierter Form.

• Compiler erzeugen

Zunächst einmal brauchen Sie einen C#-Compiler. Wenn Sie nun an Backus-Nauer-Syntax oder das Dragon-Book denken, dann werden Sie von den nächsten beiden Zeilen Quellcode sehr überrascht sein. Einen C-Sharp-Compiler erzeugt man nämlich mit Hilfe von .NET wie in Listing 4:

LISTING 4:

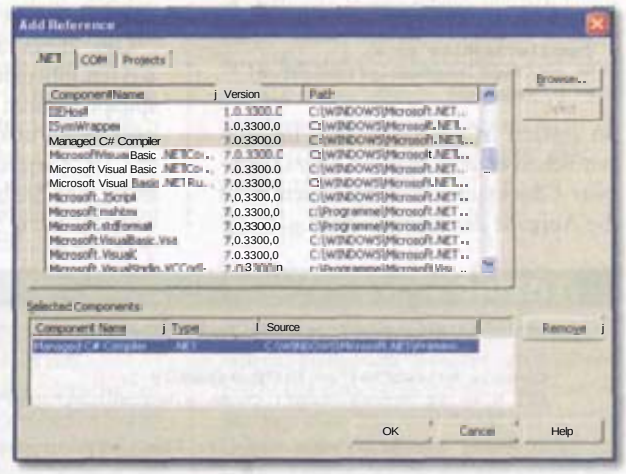
```
CSharpCodeProvider cp =
➥new CSharpCodeProvider();
ICodeCompiler cc =
➥cp.CreateCompiler();
```

Das war's. Der Compiler ist fertig. Alles, was man also tun muss, ist einen CSharpCodeProvider zu erzeugen. Den kann man dann darum bitten, eine neue Instanz eines Compilers zu erzeugen. (Das ist so notwendig, weil der CodeProvider sich unter anderem auch darum kümmert, dass die benötigten Threads richtig organisiert werden. Wenn Sie versuchen, einen CSharp-Compiler direkt zu erzeugen, dann bemängelt das Framework falsche Thread-Einstellungen.)

Wunderbar - einen Compiler haben Sie also nun. Stellt sich die Frage, wie man diesen dazu bringt, auch etwas zu übersetzen. Dazu muss man dem Compiler natürlich verschiedene Parameter mitgeben - und wie nicht anders zu erwarten, existiert auch dafür im .NET-Framework eine passende Klasse. Die trägt den Namen CompilerParameters. Davon brauchen Sie nun einen Instanz.

```
CompilerParameters p =
➥ new CompilerParameters();
```

Nun kann man eine ganze Menge an Parametern angeben. Welche das sind, erfahren Sie im Beitrag 'Kommandozeilen-Compiler verwenden' in diesem Sonderheft.



FÜR BEIDE PROJEKTE brauchen Sie eine Referenz auf den 'Managed C# Compiler' des .NET Frameworks.

Für dieses Beispiel brauchen Sie nur sehr wenige Parameter. Zum einen müssen Sie angeben, dass der Compiler eine ausführbare Datei anlegen soll. Das ist schließlich der Sinn der ganzen Übung,

Außerdem müssen Sie spezifizieren, wo diese Datei auf der Festplatte abgelegt werden soll. Den dazu benötigten Pfad haben Sie ja bereits weiter oben im Quellcode zusammengesetzt.

Diese beiden Angaben gelingen einfach über die Zuweisung an die passenden Eigenschaften des CompilerParameters-Objekts.

```
p.GenerateExecutable = true;
p.OutputAssembly = dest;
```

Das war also nicht sonderlich schwierig. Und damit ist die Sache auch schon so gut wie ausgestanden, nur noch das Übersetzen fehlt. Dazu gibt es im ICodeCompiler Interface gleich eine ganze Reihe an Funktionen. In diesem speziellen Fall soll der Compiler aber eine Assembly in Form einer ausführbaren Datei auf der Festplatte anlegen. Und zwar auf Basis von Quellcode, der in einer string-Variable steht. Das geht mit der Methode

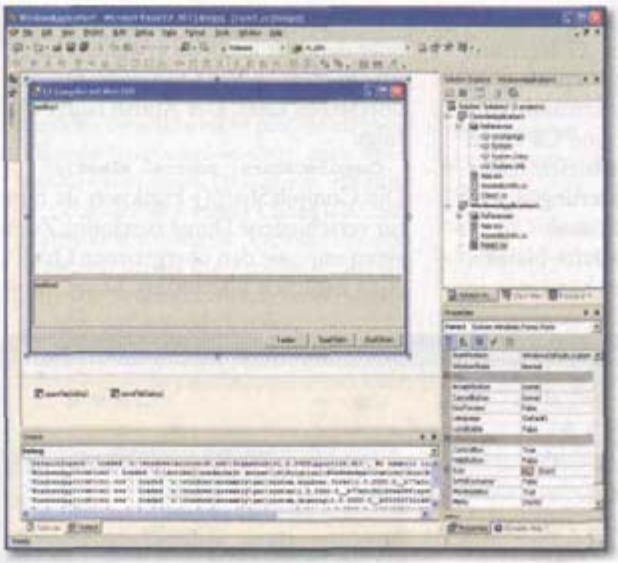
```
CompileAssemblyFromSource().
```

• Fehlermeldungen bearbeiten

Nun kann man sich leicht vorstellen, dass es neben dem erzeugten Kompilat noch weitere Ergebnisse eines Übersetzungsvorganges gibt. So können zum Beispiel Fehlermeldungen anfallen. Diese müssen natürlich von CompileAssemblyFromSource() geliefert werden, und auch dafür gibt es wieder eine Klasse. Die trägt - welche Überraschung - den Namen 'CompilerResults'. Der Aufruf zum Kompilieren sieht also wie folgt aus:

```
CompilerResults cr =
➥ cc.CompileAssemblyFromSource(
➥ p, source);
```

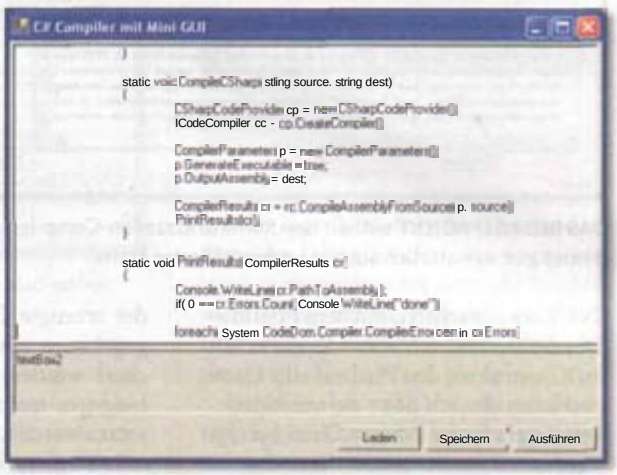
So einfach ist die Sache also. Im Wesentlichen erzeugt man ein neues Compiler-Objekt und überträgt diesem dann die Aufgabe des Kompilierens.



DAS GUI-PROJEKT ist mit der Windows.Forms-Klassenbibliothek schnell erstellt. Dabei kommt zwar keine Entwicklungsumgebung wie Visual Studio heraus - für die ersten Anfänge reicht es aber aus.

Fertig ist das ausführbare Programm. Zumindest, wenn alles geklappt hat. Enthielt der Quellcode aber Fehler, dann müssen Sie sich noch um die Ausgabe kümmern. Die Fehler - so es welche gibt - befinden sich in der Instanz von CompilerResults, die CompileAssemblyFromSource() zurückgeliefert hat.

Um diese auszuwerten, gibt es im Beispielprogramm erneut eine Funktion - das ist übrigens die letzte hier. Sie hat den Namen PrintResults() (Listing 5).



DIE MINI-GUI IN AKTION. Zwar reicht diese Variante noch nicht aus, um sich selbst zu übersetzen, aber ein 'HelloWorld' Programm ist durchaus möglich.

PrintResults() gibt zunächst einmal den Namen der erzeugten Assembly aus.

Der ist praktischerweise ebenfalls Bestandteil von CompilerResults und steht daher direkt zur Verfügung. Als Nächstes überprüft PrintResultsQ, ob Fehler anliegen - ist das nicht der Fall, dann wird die Meldung 'done' angezeigt.

Liegen aber doch Fehler vor, dann befinden sich diese in Form von CompileError-Objekten in der CompilerResults.Errors-Liste.

Darüber können Sie einfach mit *foreach* iterieren. Bei jedem Iterationsschritt arbeiten Sie dann mit einem der CompilerError-Objekte und werten es aus. Das geschieht im Wesentlichen dadurch, dass die im Objekt enthaltenen Informationen auf der Konsole angezeigt werden.

Das komplette Beispielprojekt umfasst übrigens gerade mal knapp 90 Zeilen - und das einschließlich von Kommentaren. Das ist nicht schlecht für einen voll funktionsfähigen Kommandozeilencompiler: Wer eine mächtige Klassenbibliothek benutzt, darf sich eben darüber freuen, dass doch sehr umfangreiche Mengen an Funktionalität auf sehr einfache Weise zur Verfügung steht.

LISTING 5:

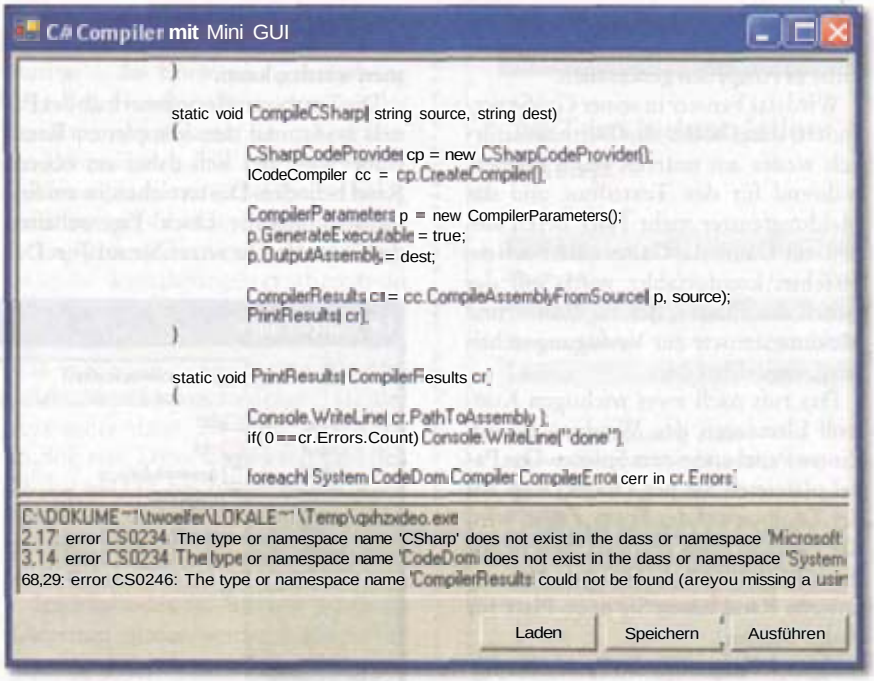
```
static void PrintResults( CompilerResults cr)
{
   Console.WriteLine( cr.PathToAssembly );
   if( 0 == cr.Errors.Count) Console.WriteLine("done");

   foreach( System.CodeDom.Compiler.CompilerError cerr in cr.Errors)
   {
      string t = "";
      if( cerr.IsWarning) t = "warning ";
      else t = "error ";

      Console.WriteLine( cerr.Line.ToString() + "," +
        cerr.Column.ToString() + ": " + t +
        cerr.ErrorNumber + " : " + cerr.ErrorText );
   }
}
```

Natürlich ist der Compiler in seiner bisherigen Form noch nicht komplett. So kann man weder Compiler-Optionen übergeben, noch ist es möglich, Referenzen auf andere .NET-Objekte zu spezifizieren. Trotzdem: Für ein einfaches 'Hello Word'-Testprogramm in C# ist der Compiler bereits geeignet. Sich selbst kann er aber noch nicht übersetzen. Das wird aber eine an anderer Stelle im Sonderheft implementierte Version korrigieren.

• Mini-Compiler auf Heft-CD

Bevor Sie den eigenen Kommandozeilen-Compiler nun benutzen können, müssen Sie Ihn natürlich erst einmal übersetzen. Das ist etwas unhandlich, wenn Sie den Compiler genau deshalb brauchen, weil Sie unter Windows 9.x arbeiten - und darum gar keinen .NET Compiler zur Verfügung haben.

Das ist aber kein echtes Problem: Auf der Heft-CD finden sie nämlich den Mini-Compiler in vorkompilierter Form.



TRETEN BEIM ÜBERSETZEN FEHLER AUF, dann werden diese im Fehler-Bereich angezeigt. Fehler die wegen fehlender Referenzen auftreten, können aber in diesem Stadium noch nicht behoben werden.

LISTING 6:

```
using System;
class HelloWorld
{
  public static int Main(String[] args)
  {
    Console.WriteLine("Hello World from CSharp");
    string s = Console.ReadLine();
    return 0;
  }
}
```

Um das Programm zu testen, brauchen Sie also nur den vorkompilierten Compiler aus dem Beispielprojekt der Heft-CD in ein temporäres Verzeichnis zu kopieren. Und dann brauchen Sie noch ein kleines C#-Beispielprogramm, das Sie übersetzen können. Nehmen Sie das in Listing 6.

Wenn Sie dieses Programm zum Beispiel in der Datei *test.cs* gespeichert haben, dann können Sie es mit dem eigenen Kommandozeilen-Compiler auf folgende Art übersetzen:

```
C:\temp>compile test.cs
```

Als Resultat erhalten Sie eine Datei mit dem Namen *test.cs.exe*. Diese kann direkt ausgeführt werden. Wenn Sie das tun, zeigt das Programm zunächst den Text 'Hello World from CSharp' an und wartet dann darauf, dass Sie einmal *[Return]* drücken. Damit ist der Beweis erbracht: C#-Quellcode können Sie auch unter Windows 9x übersetzen - wenn auch nicht mit den Compiler aus dem nicht installierbaren SDK.

• Die Mini-IDE

So nett der erste Ausflug in die Tiefen der .NET-Compiler-Dienste auch sein mag: Wirklich komfortabel ist das bisher vorliegende Programm noch nicht.

Schön wäre es, wenn man statt dessen auch eine kleine IDE hätte. Zumindest ein Textfeld, ein Feld für Fehlermeldungen, für das Laden und Speichern von Quellcode-Dateien und einen Button zum Übersetzen des geladenen Quellcodes.

Wie Sie im ersten Windows.Forms-Programm an anderer Stelle in diesem Sonderheft erfahren haben, ist das Programmieren von Windows.Forms-Anwendungen wirklich extrem angenehm. Dazu trägt nicht zuletzt der GUI-Designer aus dem Visual Studio bei.

Wenn Sie dieses Programm nicht besitzen, dann sollten Sie sich, bevor Sie an dieser Stelle weiterarbeiten, vielleicht mit der OpenSource-Entwicklungsumgebung SharpDevelop vertraut machen. Dieses Programm wird laufend weiterentwickelt, ist kostenlos, auf der Heft-CD enthalten, und bietet ebenfalls einen GUI-Designer - wenn auch einen noch nicht ganz so komfortablen wie der in Visual Studio eingebaute.

Das kann und wird sich im Laufe der Zeit aber sicher ändern: Es lohnt sich ganz bestimmt, der entsprechenden Webseite hin und wieder einen Besuch abzustatten.

Die Entwicklungsumgebung, die Sie im Folgenden implementieren, kann natürlich weder mit CSharpCode noch mit dem Visual Studio mithalten. Trotzdem erfahren Sie im Zuge dieser Implementierung eine Menge über das Entwickeln von Programmen für .NET und haben hinterher eine einfache Umgebung für das Programmieren von kleinen C#-Programmen. Wenn sie das auf den Geschmack gebracht hat, dann ist der Wechsel zu einer 'richtigen' IDE ja immer noch möglich.

• Die IDE-Form

Zunächst einmal legen Sie ein neues Windows-Forms-Projekt an. Nachdem die komplette Umgebung hinterher mit einem einzelnen Fenster auskommt, reicht der dabei von Visual Studio automatisch erzeugt Code schon fast aus: Sie haben eine Form, auf der Sie die gewünschten Kontroll-Elemente platzieren können – es fehlen also nur noch diese Elemente.

Das Fenster soll im fertigen Zustand unten einige Buttons enthalten, mit denen die verschiedenen Funktion aufgerufen werden können. Darüber soll ein Areal für die Anzeige von Meldungs-

texten stehen, und der größte Teil des Fensters wird von einem kleinen Texteditor in Anspruch genommen.

Wird das Fenster in seiner Größe verändert, dann sollen die Buttons natürlich weiter am unteren Rand bleiben, während für den Texteditor und das Meldungfenster mehr Platz bereit stehen soll. Damit das Ganze auch noch ein bisschen komfortabler wird, soll der Anteil des Platzes, der für Editor und Meldungsfenster zur Verfügung stehen wird, einstellbar sein.

Das ruft nach zwei wichtigen Kontroll-Elementen der Windows.Forms: Einem Panel und einem Splitter. Das Panel platzieren Sie per Drag&Drop aus der Toolbox auf der Form. Dabei wird das Panel so groß gezogen, dass es fast die komplette Form bedeckt. Nur am unteren Rand lassen Sie noch Platz für einige Buttons.

Danach fixieren Sie das Panel über seine 'Anchor'-Eigenschaften. Mit den 'Anchor'-Eigenschaften legen Sie fest, zu welchen Seiten des Container-Elements ein gegebenes Element einen festen Abstand haben soll. Mit anderen Worten: Ankert ein Panel am rechten Rand einer Form, dann verändert sich der Abstand zwischen dem rechten Rand des Panels und dem rechten Rand der Form auch dann nicht, wenn die Form in ihrer Größe verändert wird. Dabei ist es völlig gleichgültig, wie groß dieser Abstand ursprünglich eingestellt wurde.

Für die Mini-Entwicklungsumgebung soll das Panel später den Texteditor und den Meldungsbereich aufnehmen. Beide zusammen sollen aber immer möglichst viel des zur Verfügung stehenden Raums in Anspruch nehmen. In letzter Konsequenz bedeutet das, dass Sie das Panel ganz an den rechten, linken und oberen Rand der Form platzieren und dann die 'Anchor'Eigenschaft für alle vier Seiten auf *true* setzen. Das Panel füllt dann, mit Ausnahme des Platzes für die Buttons am unteren Rand, immer die komplette Form aus - gleichgültig, wie groß dieses gerade ist.

Im zweiten Schritt ziehen Sie nun ein Text-Feld in das Panel. Von Haus aus sind diese Felder so angelegt, dass sie nur eine Zeile Text aufnehmen können: Das kennen Sie vielleicht schon von den entsprechenden Win32 Controls.

Diese Tatsache können Sie aber leicht über die Eigenschaften des Text-Controls verändern. Schalten Sie einfach die Eigenschaft 'Multiline' von *false* auf *true* - schon wird die Textbox etwas größer. Das soll symbolisieren, dass nun mehr als eine einzelne Zeile Text aufgenommen werden kann.

Die Textbox soll nun innerhalb des Panels horizontal den kompletten Raum einnehmen und sich dabei am oberen Rand befinden. Das erreichen Sie am einfachsten über die 'Dock'-Eigenschaften des Controls. Die setzen Sie auf *Top*. Danach füllt das Textfenster bereits im GUI-Editor den gewünschten Raum aus: Nur den unteren Rand müssen Sie noch nach eigenem Geschmack anpassen.

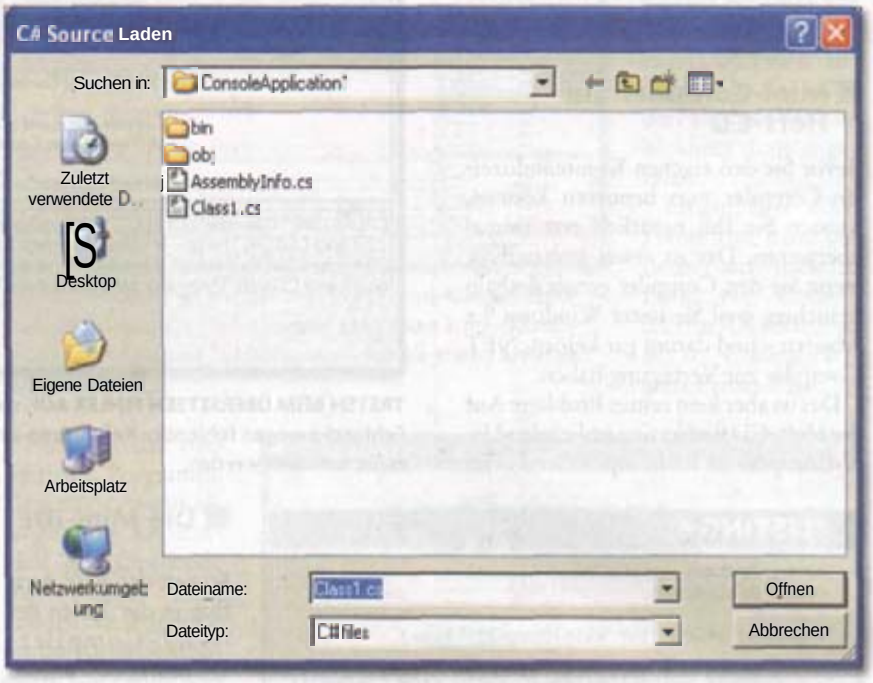
Nachdem der vertikal zur Verfügung stehende Raum im Panel vom Texteditor und vom Meldungsfenster gleichzeitig verwendet werden soll - und nachdem Sie zur Laufzeit vermutlich einstellen wollen, welches dieser beiden Fenster wie viel Platz in vertikaler Richtung beanspruchen darf, brauchen Sie nun ein weiteres Element im Panel: einen Splitter.

• Trennende Splitter

Splitter sind Elemente, die Sie innerhalb eines Fensters einsetzen können, um zwei andere Elemente voneinander zu trennen. Der Splitter kann entweder horizontal oder vertikal zwischen zwei Elenenten angebracht sein. Verläuft er horizontal, so kann er in vertikaler Richtung bewegt werden. Die Elemente ober- und unterhalb des Splitters werden dann in der Größe entsprechend angepasst.

Analoges gilt für vertikale Splitter und Elemente, die links und rechts davon angebracht sind.

Im vorliegenden Fall brauchen Sie einen horizontalen Splitter. Die Toolbox unterscheidet aber nicht zwischen horizontalen und vertikalen Splittern: Das ist eine Eigenschaft des Splitters selbst. Wenn Sie also einen Splitter aus der Toolbox in den Panel ziehen, dann wird dieser zunächst als 'vertikaler' Splitter angezeigt. Gehen Sie dann in dessen Eigenschafts-Anzeige, und verändern Sie den Splitter so, dass daraus ein horizontaler Splitter wird. Danach setzen Sie dessen Dock-Verhalten noch auf *Top*. Der Splitter 'klebt' dann unter dem Textfeld, das Sie schon platziert haben.



DEN DIALOG ZUM ÖFFNEN von Quellcode-Dateien kann man mit der 'FileOpenDialog' Komponente sehr einfach erzeugen.

Schließlich brauchen Sie noch ein Control für die Ausgabe der Fehlermeldungen des Compilers. Dazu verwenden Sie eine weitere Textbox die Sie in den restlichen verbleibenden Platz des Panels ziehen. In den Eigenschaften dieser zweiten Box setzen Sie nun das 'ReadOnly' Feld auf 'true' und die 'Dock' Eigenschaft auf 'Fill'. Das zweite Control nimmt dann den kompletten noch freien Platz innerhalb der Form für sich in Anspruch.

Damit ist ein Großteil des GUIs schon fertig. Sie können das Projekt nun übersetzen und die Form testen. Dabei werden Sie feststellen, dass Sie die Form beliebig in Ihrer Größe verändern können:

Die beiden Textfelder beanspruchen immer den kompletten Platz für sich und lassen nur unten einen kleinen Streifen frei. Außerdem können Sie die Maus über den Splitter bewegen und diesen dann bei gedrückter Taste bewegen: Der den beiden Textfenstern zugewiesene Raum lässt sich auf diese Weise beeinflussen.

Soweit - so gut. Nun fehlen noch drei Buttons im unteren, bisher nicht genutzten Raum der Form. Dort platzieren Sie die Buttons mit Hilfe der Toolbox. Den ersten (linken) Button versehen Sie mit dem Text 'Laden', den zweiten mit dem Text 'Speichern' und den dritten mit dem Text 'Ausführen'. Welche Funktionen diese Buttons später ausführen sollen ist dabei wohl einigermaßen klar ersichtlich.

• Quellcodes laden und speichern

Zum Laden und Speichern von Quellcodes bietet .NET ebenfalls Controls. Um genau zu sein, sind die gebotenen Controls nicht zum Laden und Speichern gedacht, sondern für die Anzeige des 'Datei laden'- bzw. 'Datei speichern'-Dialoges. Diese Controls haben aber auf der eigentlichen Form keine optische Repräsentation. Daher muss man Sie auch nicht auf die Form selbst, sondern auf den weißen, die Form einschließenden Bereich ziehen. Genau das tun Sie als Nächstes und zwar einmal mit dem 'FileOpenDialog'- und ein weiteres Mal mit dem 'FileSaveDialog'-Element.

Damit ist der größte Teil der Arbeit schon erledigt, es fehlt allerdings noch ein bisschen Glue-Code, der die verschiedenen Elemente miteinander verbindet. Im besondern fehlt der Event-Handler, der ausgeführt werden soll, wenn der 'Laden'- Button gedrückt wurde.

Der ist aber leicht eingefügt. Doppelklicken Sie dazu einfach auf diesen Button - die Entwicklungsumgebung fügt den Handler dann an passender Stelle ein und platziert sie im Quellcode-Editor innerhalb dieses Handlers.

Hier muss nun ein 'FileOpen'-Dialog geöffnet und die ausgewählte Datei geladen werden. Die Anzeige des Dialogs ist trivial, denn dafür gibt es ja bereits ein passendes Element. Das müssen Sie nur zur Anzeige des Dialogs veranlassen, was wiederum durch Aufruf der Methode *ShowDialog()* erfolgt. Hat der Anwender dann auf dem geöffneten Dialog eine Datei ausgewählt und den Dialog mit *OK* geschlossen, dann liefert der Aufruf einen Wert vom Typ *DialogResults.OK* zurück. Der komplette Aufruf sieht also wie in Listing 8 aus.

Innerhalb des 'if'-Blockes muss die Datei nun geladen werden. Das geht bereits mit dem schon bekannten StreamReader-Objekt.

Selbiges kann man nicht nur mit dem Pfad zu einer Datei als string konstruieren, man kann auch einen geöffneten Stream dazu verwenden. Einen geöffneten Stream erhalten Sie aber an dieser Stelle am einfachsten dadurch, das Sie den openFileDialog darum bitten:

```
StreamReader sr =
➥ new StreamReader (
➥ this.openFileDialog1.OpenFile()
➥);
```

Mit diesem StreamReader können Sie nun den Text aus der Quellcode-Datei einlesen und in das 'Text'-Feld des Editor-Controls schreiben: Der Text wird dann im Editor angezeigt.

```
string source = sr.ReadToEnd();
this.textBox1.Text = source;
```

Damit ist der zum Laden einer Datei benötigte Quellcode vollständig. Für das Speichern der Datei können Sie im Prinzip genauso vorgehen: Nur eben umgekehrt und unter Verwendung eines StreamWriter-Objekts. Im Beispiel ist das aber nicht zu Ende geführt. Statt dessen wird noch der Eventhandler für den 'Übersetzen'-Button gezeigt, denn der ist deutlich interessanter als eine weitere File-IO-Operation.

Der Code dafür wird Ihnen aber sehr bekannt vorkommen, schließlich ist es fast der gleiche Code, den Sie schon beim Kompilieren von der Kommandozeile verwendet haben. Die einzige Ausnahme beim Kompilieren ist, dass der Quelltext, der zu übersetzen ist, nicht aus einer Datei sondern eben aus dem Textfeld ausgelesen wird:

```
CompilerResults cr =
➥ cc.CompileAssemblyFromSource(
➥ p, this.textBox1.Text );
```

Eine ähnlich minimale Änderung gibt es auch in der PrintResults() Funktion: Hier wird er anzuzeigende Text, anders als beim Konsolen-Beispiel, nicht auf die Konsole ausgegeben, sondern eben im Meldungsfenster angezeigt.

Nachdem beim Meldungsfenster keine 'WriteLine()'-Methode, wie auf der Konsole, zur Verfügung steht, müssen Sie aber hier die Zeilenumbrüche manuell einfügen. Das geht einfach dadurch, das Sie \r\n (Carriage-Return, New-Line) am Ende jeder Zeile anfügen.

So einfach ist es, die Compiler-Dienste von .NET zu nutzen und auch unter Windows 9.x zu einem kostenlosen .NET-Compiler zu kommen. © UR

LISTING 7:

```
private void button1_Click(object Sender, System.EventArgs e)
{
}
```

LISTING 8:

```
if( DialogResult.OK == this.openFileDialog1.ShowDialog())
{
}
```

LISTING 9:

```
this.textBox2.Text = "";
this.textBox2.Text += cr.PathToAssembly;
if( 0 == cr.Errors.Count) this.textBox2.Text += " done";
```

Windows Forms-Bibliothek nutzen

Eine Tray-Anwendung

Unter Win32 war es zwar ganz einfach, ein Icon in den System Tray zu bekommen - das zugehörige Programmfenster war aber nur relativ schlecht zu verstecken. Mit den Windows Forms ist die Sache viel einfacher, wie Sie im diesem Beitrag erfahren werden.

THOMAS WÖLFER

Eine Tray-Anwendung ist ein Programm, das kein normalerweise sichtbares Fenster hat und das auch nicht in der Liste der laufenden Programme auftaucht, wenn man diese mit [ALT+TAB] öffnet. Natürlich soll auch kein Button in der Windows Taskleiste auftauchen: Die einzige Stelle, an der sich das Programm bemerkbar macht, ist der Tray, der kleine Bereich in derTaskbar in der auch die Uhr ist.

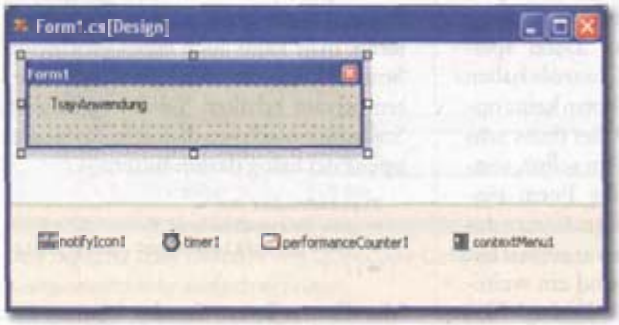
So eine Anwendung kann man für alles Mögliche gebrauchen. Ein kleines Tool, das den aktuellen Zustand eines entfernten Rechners in regelmäßigen Abständen testet, ein Werkzeug, das die eigene Online-Zeit mitführt und viele andere sinnvolle Einsatzgebiete sind denkbar. Immer, wenn man ein längerfristig benötigtes Werkzeug, das nicht viel Platz beanspruchen darf, programmieren möchte, ist eine Tray-Anwendung der richtige Weg.

In diesem Beispiel implementieren Sie genau so eine Anwendung. Es wird dafür ein Programm verwendet, das den momentan verfügbaren Speicher des eigenen Rechners in regelmäßigen Zeitabständen überprüft. Das Programm erscheint dabei nur als Icon im Tray. Hält man die Maus über das Icon, dann wird der verfügbare Speicher in einem Tool-Tipp eingeblendet. Außerdem wird man mit der Maus auf das Icon klicken können, das öffnet dann ein Objektmenü.

• Schritt eins

Als erstes brauchen Sie dafür zunächst eine ganz normale Windows Forms-Anwendung. Die von Haus aus erzeugte Form ist für diesen Bedarf völlig ausreichend. Bevor Sie nun aus dieser Form eine 'echte' Tray-Anwendung machen implementieren Sie aber zunächst einmal die gewünschte Funktionalität.

Der momentan verfügbare Speicher kann am einfachsten über einen Performance-Counter ermittelt werden. Für solche Performance Counter gibt es extra eine spezielle Komponente. Der Zugriff darauf erfolgt aber nicht über die Toolbox des Visual Studios - denn Performance-Counter werden vom Visual Studio als Server-Objekte betrachtet und daher so ähnlich behandelt, wie zum Beispiel Datenbanken. Mit anderen Worten: Die auf dem Rechner verfügbaren Performance Counter werden im Server-Explorer angezeigt. Dabei handelt es sich um das Fenster, das sich den Platz mit der Toolbox teilt.



FAST DAS GANZE PROGRAMM kann innerhalb des Forms-Editors zusammengesetzt werden. Echte Code-Zeilen gibt es nur sehr wenige.

Wenn Sie den Server-Explorer öffnen, bietet das Visual Studio alle Datenbankverbindungen, die EventLogs und MessageQueues, die Dienste und weitere Server-Elemente zur Auswahl an. Dazu gehören auch die Performance-Counter des lokalen Systems. Sie sind aber nicht auf das lokale System beschränkt. Mit einem rechten Mausklick auf das Server-Symbol können Sie auch eine Verbindung zu einem weiteren Rechner herstellen und dann auch dessen Performace-Counter sowie die anderen Informationsquellen benutzen.

Für dieses Beispiel ist der lokale Performance-Counter völlig ausreichend. Für das Beispiel-Programm benötigen Sie einen Counter aus dem Bereich 'Speicher' - und zwar den Counter mit der Beschreibung 'Verfügbare MB'.

Dabei lohnt es sich, auch einmal einen Blick auf die anderen Counter zu werfen, denn es gibt jede Menge davon - und die stellen viele Informationen über das lokale System zur Verfügung, die vor der Einführung des Performance-Counter-Objektes nur schwer zu erreichen waren. Sie können mit den Countern beispielsweise sehr leicht ein Programm zusammenstellen, das eine Übersicht über den Zustand des lokalen SMTP-Servers gibt: Wie viele Mails wurden nicht zugestellt, welche Gründe gab es dafür - das sind nur wenige Beispiele für die zur Verfügung stehenden Informationen.

• Schritt zwei

Öffnen Sie zunächst den 'Speicher'-Ast und ziehen den gewünschten Counter auf die Form. Visual Studio erzeugt nun ein passendes Counter-Objekt, das Sie im Programm verwenden können.

Mit diesem Objekt können Sie nun leicht ermitteln, wie viel Speicher noch zur Verfügung steht - nur muss das natürlich regelmäßig passieren, denn die Menge an verfügbarem Speicher ändert sich ja laufend. Auch das geht recht

leicht. Alles was Sie dafür brauchen, ist ein regelmäßiges Ereignis, und das kann ein Timer-Objekt liefern. Dazu öffnen Sie nun die Toolbox und wählen darin den Timer aus den *Components* aus. Den ziehen Sie dann ebenfalls auf die Form.

In der Eigenschafts-Ansicht können Sie nun das Intervall festlegen, in dem der Timer aktiv werden soll. Für dieses Beispiel wird eine Periode von einer Sekunde ausreichen, Sie geben also den Wert 1000 an, da die Timer-Perioden in Millisekunden angegeben werden.

Damit Sie auf den Timer reagieren können, brauchen Sie noch einen Event-Handler für das 'Elapsed'-Ereignis des Timers. Den nennen Sie zum Beispiel 'OnTimer'. Dann füllen Sie den Event-Handler mit Code.

Dabei soll für den ersten Test einfach nur der Text in der Titelzeile des Programms verändert werden. Er soll regelmäßig verfügbaren Speicher anzeigen. Das geht, indem Sie den Performance-Counter nach seinem 'Raw-Value' Wert fragen. Dabei handelt es sich um eine long Integer, die in diesem Fall einfach die Menge an verfügbarem Speicher in Megabyte darstellt. Den weisen Sie dann der Titelzeile der Form zu:

LISTING 1:

```
private void OnTimer(object sender, System.Timers.ElapsedEventArgs e)
{
  long l = this.performanceCounter1.RawValue;
  this.Text = l + " MB";
}
```

Sie können das Programm jetzt übersetzen und kurz testen: Einmal pro Sekunde zeigt es nun den verfügbaren Speicher in seiner Titelzeile an.

• Schritt drei

Im nächsten Schritt fügen Sie nun das Icon für den Tray-Bereich hinzu. Dazu zeichnen Sie zunächst einmal ein Icon mit dem Icon-Editor. Dann ziehen Sie ein 'notifyIcon'-Objekt aus der Toolbox auf die Form. Dieses Objekt kümmert sich um die Anzeige Ihres Icons, allerdings müssen Sie dazu noch die 'Icon'-Eigenschaft des Notify-Icon-Objektes auf das zuvor erstellte Icon setzen. Damit das Notify-Icon den verfügbaren Speicher anzeigen kann, müssen Sie die OnTimer()-Methode noch ein bisschen aufbohren. Dabei weisen Sie den Text, den Sie zuvor nur der Titelleiste der Form zugewiesen haben, auch dem Notify-Icon zu

Wenn Sie das Programm nun erneut übersetzen und starten, dann sehen Sie sofort Ihr Icon im Tray-Bereich. Wenn Sie die Maus darüber halten, dann erscheint auch der gewünschte ToolTip mit der Information über den freien Speicher.

DER SERVER-EXPLORER bietet extrem leichten Zugriff auf Server-Objekte. Dazu zählen auch die Performance-Counter.

LISTING 2:

```
private void OnTimer(object sender, System.Timers.ElapsedEventArgs e)
{
  long l = this.performanceCounter1.RawValue;
  this.Text = l + " MB";
  this.notifyIcon1.Text = l + " MB frei";
}
```

• Schritt vier

Bei einem Klick auf das Icon passiert aber noch nichts. Das ist kein Wunder, denn diesen Fall haben Sie ja im Programm noch gar nicht behandelt.

Erwünscht ist, dass man mit der rechten Maustaste auf das Icon klicken kann. Das soll dann ein Kontext-Menü öffnen, das weitere Befehle anbietet. Im diesem Beispiel wird das einfach nur der Befehl zum Schließen der Anwendung sein.

Dazu ziehen Sie zunächst ein Kontext-Menü aus der Toolbox auf die Form und erweitern das Menü dann um den Befehl *Beenden.* Dafür definieren Sie einen Event-Handler, der das Programm beendet:

```
private void menuItem1_Click(
    object sender,
    System.EventArgs e)
{
  this.Close();
}
```

Damit ist funktional schon alles geklärt - nur das Fenster muss noch aus der [Alt+TAB]-Liste verschwinden – und auch sonst muss es unsichtbar werden.

Auch das geht aber extrem einfach und kann vollständig über die Eigenschafts-Ansicht der Form geklärt werden. Die beiden Eigenschaften, die Sie benötigen, sind 'FormBorderStyle' und 'WindowState'. Den BorderStyle setzen Sie auf FixedToolWindow und den 'WindowState' auf *Minimized.* Schon ist das Fenster verschwunden und nur noch über sein Icon im Tray zu finden. Jetzt sind Erweiterungen denkbar: Sie können zusätzliche Befehle in das Programm einbauen und weitere Performance-Counter für zusätzliche Informationen anzapfen. Natürlich wäre auch ein Handler, der einen einfache Mausklick oder einen Doppelklick auf das Icon behandelt, wünschenswert - all diese Funktionen sind abermit dem Visual Studio leicht implementiert.

DIE FERTIGE TRAY-ANWENDUNG in Aktion. Das Programm zeigt den verfügbaren Speicher in Megabyte an.

Nun wissen Sie, wie einfach man eine Tray-Anwendung programmieren kann und wie leicht zugänglich die Performance-Counter Ihres Systems für eigene Anwendungen sind: Dem eigenen Ersatz für den Performance Monitor oder den Taskmanager steht nichts mehr im Wege. © UR

Keine Kunst

Malen mit GDI+

Bei .NET gibt es viel Neues: Das in die Tage gekommene GDI-Interface wurde aufgebohrt, und gemalt wird bei .NET nun mit GDI+.

THOMAS WÖLFER

GDI+ ist eigentlich ein Teil von Windows XP, das aber in Form einer DLL auch auf älteren Windows-Versionen eingesetzt werden kann. GDI+ stellt dem Programmierer Möglichkeiten der zweidimensionalen Vektorgrafi, der Bildbearbeitung und der Typografie zur Verfügung. Dabei baut GDI+ auf dem alten Windows GDI auf, hat aber in vielen Bereichen bessere und weiter ausgebaute Funktionen.

Bei .NET existiert ein Satz an Klassen, die GDI+ kapseln und mit denen GDI+ direkt innerhalb von .NET verwendet werden kann – aber eben auf Basis dieser Klassen und nicht auf Basis der GDI+ API. Genau wie bei dem alten GDI handelt es sich auch bei GDI+ um ein Device Interface, das den Programmierer in die Lage versetzt, Informationen am Bildschirm oder einem anderen Ausgabegerät auszugeben, ohne sich darum kümmern zu müssen, um welches Gerät es sich dabei konkret handelt.

• Die GDI+ Dreiteilung:

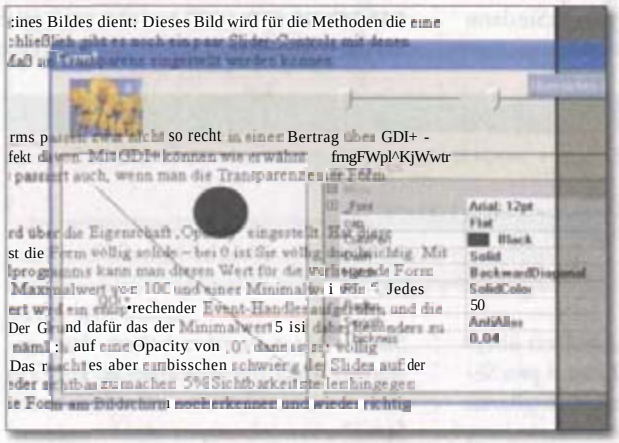
GDI+ ist in drei Teile geteilt: 2d Vektor-Grafik, Bildverarbeitung, Typografie. Bei *2d- Vektorgraphik* geht es um das Zeichnen von Primitiven wie zum Beispiel Linien und Kurven. Diese Primitive werden durch einen Satz an Punkten definiert, die sich in einem Koordinatensystem befinden. GDI+ stellt dafür verschiedene Elemente bereit.

Bei diesen Elementen handelt es sich um Klassen, die spezifizieren, wie die Primitive gemalten werden, Klassen, die die Primitive beschreiben, und Klassen die das tatsächliche Malen besorgen. So speichert beispielsweise die Point()-Klasse einen Punkt, der für die Anfangs- oder Endkoordinate einer Linie zuständig ist. Die Pen()-Klasse speichert Informationen über die Linienart, die Strichstärke und die Farbe, während die Graphics Klasse Methoden zum Ausgeben von Linie zur Verfügung stellt.

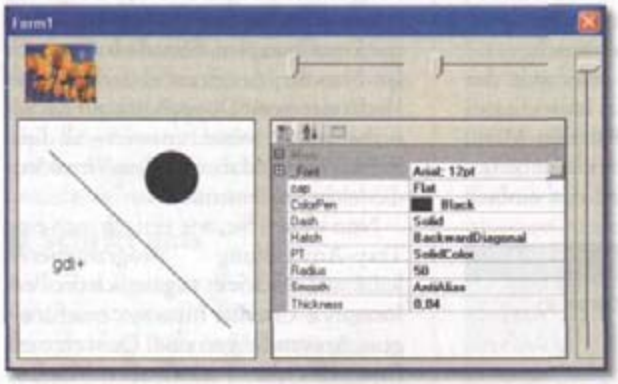
Darüber hinaus gibt es auch noch die Brush()-Klasse. Diese ist für geschlossene Figuren zuständig und spezifiziert, wie solche Figuren gefüllt werden sollen.

Bei der *Bildverarbeitung* (Imaging) geht es auch ums Malen - aber um einen anderen Typ der Malerei. Bestimmte Arten von Grafiken sind mit reiner Vektor-Grafik schlecht zu erzeugen. Ein digitales Bild kann zwar in der Theorie auch mit Vektorgrafik wiedergegeben werden, es ist aber leichter, solche Bilder in Form von Bitmaps zu speichern und zu manipulieren. Auch dafür stellt GDI+ eine Vielzahl an Funktionen zur Verfügung. Dazu gehören auch Funktionen, mit denen man Bilddaten direkt laden kann. Es werden die gängigen Windows-Formate, aber auch andere Formate wie PNG, GIF oder TIF unterstützt.

Typografie beschäftigt sich mit der Anzeige von Texten in unterschiedlichen Schriftarten, -großen und -schnitten. Auch dafür stellt GDI+ Klassen zur Verfügung. GDI+ kapselt all diese Funktionalität in etwa 60 Klassen, 50 Enumerationen und einigen Strukturen. Kern der Sache ist dabei die Graphics-Klasse, die letztendlich immer bemüht werden muss wenn eine Grafik tatsächlich angezeigt werden soll.



VORHER UND NACHHER: Ein fast transparentes Fenster ist nicht wirklich gut zu erkennen. Unter bestimmten Bedingungen kann das aber hilfreich sein.



DAS BEISPIELPROGRAMM in Aktion. Gemalt wird nicht viel - aber was gemalt wird, kann vollständig konfiguriert werden.

• GDI+: Was ist neu?

Eine sehr prominente neue Möglichkeit in GDI+ besteht aus den Gradient Brushes. Dabei handelt es sich um Brushes, die zur Ausgabe von Farbübergängen verwendet werden können. Wird eine Fläche mit einem Gradienten gefüllt, erhält man einen Farbverlauf.

Auf diesen Verlauf kann man bei den Gradient Brushes auf verschiedenste Arten Einfluss nehmen. So ist es zum Beispiel möglich, einen 'klassischen'

Farbverlauf zu erzeugen, bei dem ein normaler Farbübergang von links nach rechts erfolgt.

Es ist aber auch möglich, einen Farbverlauf zu erzeugen, bei dem bestimmte Punkte einer Fläche die eine Farbe, die komplette Außenkante aber die andere Farbe annimmt.

Mit Gradient Brushes kann man daher nicht nur sehr schöne, sondern auch sehr praktische Effekte erzielen.

So ist es etwa nicht das geringste Problem, einen räumlichen Eindruck einer Kugel zu erzeugen, indem man einen passenden Farbverlauf innerhalb einer Ellipse platziert.

Splines

GDI+ unterstützt von Haus aus Splines. Bei Splines handelt es sich um Kurven, die durch Stützpunkte definiert werden. Die Kurve verläuft dabei durch alle Stützpunkte - es entstehen keine Sprünge im Verlauf der Steigung: Man erhält also immer sanfte Rundungen in allen Stützpunkten.

Unabhängige Pfade

Beim GDI-System existierten Path-Elemente, die Zeichnungsteile enthalten konnten. Allerdings gehörte ein solches Element zum Device-Kontext und ging zusammen mit diesem verloren.

In GDI+ kann man beliebig viele GraphicsPath-Objekte erzeugen, die ebenfalls zusammengehörende Zeichenoperationen kapseln – und die nicht verloren gehen.

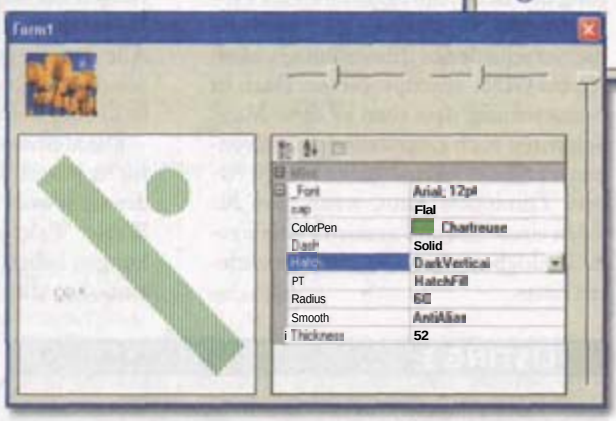
Alpha-Blending

Ebenfalls neu bei GDI+ *ist Alpha-Blending*. Damit kann man den Grad der Transparenz einer Füllfarbe angeben. Das bedeutet, dass Teile einer Grafik durch andere Teile der gleichen Grafik hindurchscheinen können.

Wie durchscheinend eine Fläche sein soll, kann angegeben werden - es ist also auch möglich, unterschiedliche Stufen von durchscheinenden Flächen zu erhalten. Das findet sich übrigens auch bei Windows-Forms wieder, denn dort ist es mit .NET ebenfalls möglich, den Grad der Transparenz einzustellen.

Komplett anderer Programmierstil

Aufgrund der Kapselung in Klassen ändert sich die Art und Weise, wie mit GDI+ gemalt wird im Vergleich zu klassischem GDI völlig. Dazu ein Beispiel: Es soll eine Linie ausgegeben werden. Mit klassischem GDI würde diese Aufgabe wie in Listing 1 erledigt werden.



SCHRAFFUREN sind mit GDI+ ganz einfach. Eine Vielzahl an vordefinierten Schraffuren stehen zur Verfügung.

Man besorgte sich zunächst einen Device-Context Handle, erzeugte dann einen Pen und selektierte diesen in den Device-Context. Dann startete man die Mal-Operation mit der Moveto()-Funktion und zog die Linie mit *LineTo()* an den Endpunkt. Schließlich wurde das Zeichnen mit *EndPain()* beendet.

Bei GDI+ sieht das anderes aus - und zwar wie in Listing 2. Nachdem alle Methoden bei C# in einer Klasse enthalten sind, ist das hier im Beispiel auch so beibehalten worden. Die Klasse als solches hat aber eigentlich nichts mit dem Malen zu tun.

Wie man sieht, werden alle Attribute einer Zeichenoperation direkt mit übergeben - der Fall, dass man zunächst irgend ein Objekt in den Device-Context selektieren muss und dann erst damit arbeiten kann, existiert bei GDI+ nicht.

MIT EINER 'LINEARGRADIENTBRUSH' kann man hübsche Effekte erzeugen. GDI+ kann obendrein auch komplexere Gradienten.

Was sich ebenfalls verändert hat, ist die Art und Weise, wie Flächen gefüllt werden. Bei GDI wurden eine Fläche mit einem Funktionsaufruf sowohl gezeichnet als auch gefüllt. Bei GDI+ gibt es für das Zeichnen einer Fläche eine Funktion und für das Füllen einer Fläche eine andere. So viel zu den allgemeinen Grundlagen.

Die MSDN Libaray enthält aber auch sehr viel weitergehende Informationen - sowohl im öffentlichen zugänglichen Bereich online als auch auf der nur per Abo erhältlichen CD.

• Zeit für ein Beispiel

Das Beispielprogramm zu diesem Beitrag zeigt eine ganze Reihe der von GDI+ gebotenen Möglichkeiten auf. Es handelt sich um eine kleine Anwendung, mit der eine gefüllte Fläche, eine

LISTING 1:

```
HDC         hdc;
PAINTSTRUCT PS;
HPEN        hPen;

hdc = BeginPaint(hWnd, &ps);
   hPen = CreatePen(PS_SOLID, 3, RGB(255, 0, 0));
   SelectObject(hdc, hPen);
   MoveToEx(hdc, 20, 10, NULL);
   LineTo(hdc, 200, 100);
EndPaint(hWnd, &ps);
```

LISTING 2:

```
class PlainForm : Form
{
   protected override void OnPaint
(PaintEventArgs e)
   {
      Pen royPen = new Pen(Color.Red, 3);
      Graphics myGraphics = e.Graphics;
      myGraphics.DrawLine(myPen, 20, 10, 200, 100);
   }
}
```

einfache Linie und ein Text angezeigt werden. Dabei kann die Größe der Fläche, die Strichstärke, die Schriftart und -große verändert werden. Ebenso ist es möglich, die verwendete Farbe, die Linienart und die Art der Linienenden, sowie viele andere Attribute zu verändern.

Die Form des Projekts enthält eine PictureBox zur Ausgabe der Graphik und ein PropertyGrid zum Einstellen der verschiedenen Attribute.

Außerdem gibt es eine zweite PictureBox, die der Aufnahme eines Bilds dient: Dieses Bild wird für die Methoden verwendet, die eine Textur benötigen. Schließlich gibt es noch ein paar Slider-Controls, mit denen verschiedene Größen und das Maß an Transparenz eingestellt werden können.

• Transparenz zuerst

Die transparenten Windows Forms passen zwar nicht so recht in einen Beitrag über GDI+ - sind aber trotzdem auch ein Effekt davon. Mit GDI+ können transparente Farben definiert werden und genau das passiert auch, wenn man die Transparenz einer Form verändert.

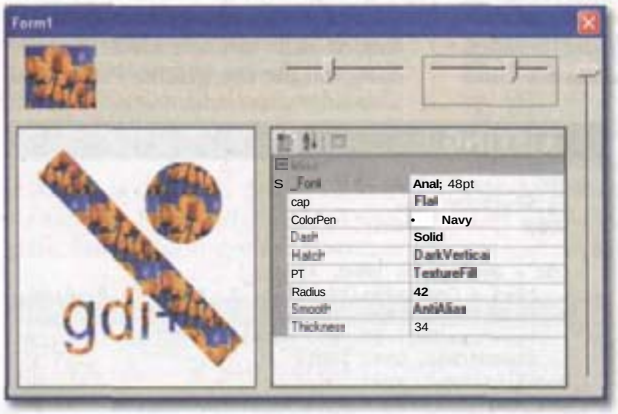
Die Transparenz von Forms wird über die Eigenschaft *Opacity* eingestellt. Hat diese Eigenschaft den Wert 100, so ist die Form völlig solide, bei 0 ist Sie völlig durchsichtig. Mit dem rechten Slider des Beispielprogramms kann man diesen Wert für die vorliegende Form einstellen. Der Slider hat einen Maximalwert von 100 und einen Minimalwert von 5. Jedesmal, wenn sich dieser Wert ändert, wird ein entsprechender Event-Handler aufgerufen und die Opacity der Form neu gesetzt. Der Grund dafür, dass der Minimalwert dabei 5 ist, liegt darin, dass eine Form mit einer Opacity von '0' völlig durchsichtig - also unsichtbar wird. Das würde es etwas schwierig machen, den Slider auf der Form zu finden und selbige wieder sichtbar zu machen. Fünf Prozent Sichtbarkeit stellen einigermaßen sicher, dass man die Form am Bildschirm noch erkennen und wieder richtig sichtbar machen kann.

Vor den echten Malfunktionen noch ein kleiner Hinweis auf die Strukturierung des Beispielprogramms: Das Programm soll Sie ja in die Lage versetzen, die verschiedenen Einstellmöglichkeiten bei GDI+ auszuprobieren. Dazu ist es notwendig, dass eben all diese Möglichkeiten auch eingestellt werden können - und das ist eine ganze Menge Arbeit. Zumindest dann, wenn man für jeden einstellbaren Parameter eine eigene Dialogbox von Hand implementieren muss.

LISTING 3:

```
private void OnValueChanged(object Sender,
➥System.EventArgs e)
{
  this.Opacity = ((double)this.trackBar1.Value / 100.0);
}
```

Bei .NET gibt es aber bereits eine sehr praktische Möglichkeit, Eigenschaften von Objekten zu verändern: Das PropertyGrid. Dieses Grid ist an anderer Stelle in diesem Sonderheft ausführlich erläutert. Es ist allerdings wichtig zu wissen, dass das Grid ein .NET-Objekt benötigt. Das ist das Objekt, dessen Eigenschaften man mit dem Grid einstellen kann. Darum wurde im Beispielprogramm eine private Klasse definiert, die alle einstellbaren Parameter für die Darstellung der Objekte aufnimmt.



MIT EINER TEXTUR und dem passenden Brush bekommt man ein wenig Frühling sogar auf den Windows-Bildschirm.

Von dieser Klasse wird beim Start des Programms eine Kopie erzeugt, und diese Kopie wird für die gesamte Laufzeit des Programms bei allen Maloperationen nach den zu verwendenden Parametern befragt. Dieses Objekt wird außerdem in das PropertyGrid gestellt: Das hat den Vorteil, dass man auf einen Schlag alle Parameter ändern kann, ohne dass eigens eigene Einstellmöglichkeiten geschaffen werden müssen.

Für die Größe des anzuzeigenden Kreises und die Dicke der anzuzeigenden Linie wurden zusätzlich TrackBar Controls auf der Form platziert: Damit kann man die Größe besser einstellen, als das im PropertyGrid der Fall ist.

Die Set-Accessors der Privaten Klassen kümmern sich dabei auch gleich noch um das Invalidieren der Zeichenfläche: Das hat den Vorteil, dass Änderungen der Eigenschaften im Grid direkt auch visuelle Auswirkungen haben. Alle Accessors haben dabei mehr oder minder den gleichen Aufbau - daher hier in Listing 4 einer zum Verständnis.

Die Methode zum Ausgeben der Grafik ist immer die gleiche - nur verwendet sie jeweils die aktuell eingestellten Werte. Welche davon welche Auswirkungen haben, erfahren Sie gleich, hier zunächst die Paint-Methode an sich (Listing 5): Zunächst wird der SmoothingMode des Graphics-Objekts eingestellt. Der SmoothingMode legt fest, wie die ausgegebenen Zeichnungen geglättet werden sollen. Dabei könnte zum Beispiel Anti-Aliasing zum Einsatz kommen. Die Auswirkungen einer Veränderung des SmoothingMode erkennen Sie im Beispielprogramm am besten dann, wenn Sie die gefüllten Kreis vergrößern. Ohne Anti-Aliasing verläuft Rand des Kreises dann treppenförmig. Schalten Sie nun zwischen den verschiedenen Smoothing-Modes um – im PropertyGrid geht das mit dem Feld *Smooth* - so können Sie die Verbesserung der Bildqualität (oder natürlich deren Verschlechterung) sofort erkennen.

LISTING 4:

```
public Color ColorPen
{
  get
  {
    return m_colpen;
  }

  set
  {
    m_colpen = value;
    m_pp.Invalidate();
  }
}
```

Um etwas zeichnen zu können, braucht man fast immer einen Brush. Brushes werden verwendet, um Flächen zu füllen, und dabei können eine ganze

LISTING 5:

```
private void OnPaint(object sender, System.Windows.Forms.PaintEventArgs e)
{
  e.Graphics.SmoothingMode = paintProps.Smooth;

  Brush b = null;
  switch(paintProps.PT)
  {
    case PenType.HatchFill:
      b = new HatchBrush(paintProps.Hatch, Color.AliceBlue,
paintProps.ColorPen);
      break;
    case PenType.LinearGradient:
      b = new LinearGradientBrush(new Rectangle(0,0,210,210),
➥Color.AliceBlue,this.paintProps.ColorPen,45f,false);
      break;
    case PenType.SolidColor:
      b = new SolidBrush(this.paintProps.ColorPen);
      break;
    case PenType.TextureFill:
      b = new TextureBrush(new Bitmap(this.pictureBox2.Image));
      break;
  }

  if(b == null) return;

  Pen p = new Pen(b, this.paintProps.Thickness);
  p.DashStyle = paintProps.Dash;

  p.StartCap = p.EndCap = paintProps.cap;
  e.Graphics.DrawLine(p, new Point(30, 30), new Point(200, 200));

  e.Graphics.DrawString("gdi+", paintProps._Font, b, 30, 130);

  e.Graphics.FillEllipse(b, 120, 30, paintProps.Radius, paintProps.Radius);
}
```

LISTING6:

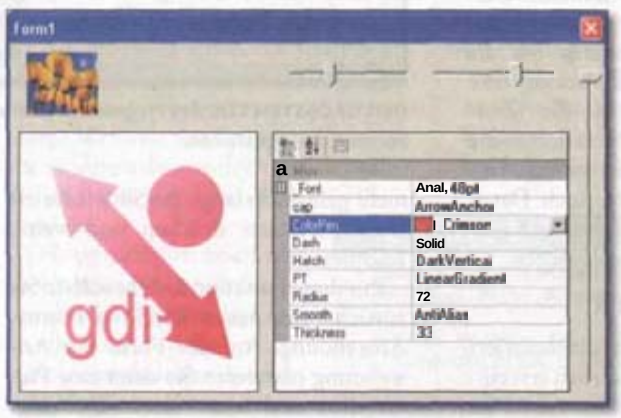
```
Pen p = new Pen(b, this.paintProps.Thickness);
p.DashStyle = paintProps.Dash;
p.StartCap = p.EndCap = paintProps.cap;
```

Menge an unterschiedlichen Brushes zum Einsatz kommen. Im Beispielprogramm können Sie vier unterschiedliche Brushes auswählen. Dabei handelt es sich um:

- HatchBrush
- LinearGradientBrush
- SolidBrush
- TextureBrush

Wie der Name schon sagt, gibt HatchBrush Schraffuren aus. Dabei gibt es bei GDI+ eine Vielzahl an vordefinierten Schraffur-Arten. Wenn Sie im Beispielprogramm für die Eigenschaft *PT* (PenType) *HatchFill* ausgewählt haben, dann wird ein HatchBrush für die Anzeige verwendet. Welche Schraffur dann von diesem Brush verwendet wird, das können Sie im Beispielprogamm unter der Eigenschaft *Hatch* auswählen.

Ein LinearGradientBrush wird verwendet, wenn Sie als PenType *LinearGradient* ausgewählt haben. Ein solcher Brush wird für Gradienten - und zwar lineare - verwendet. Dabei gibt man die Richtung des Verlaufs bei der Konstruktion des Brush mit an. Im Beispielprogramm verläuft der Gradient einfach von oben links nach unten rechts. Die Brushes haben übrigens auch einen Einfluss auf die ausgegebene Schrift: Das können Sie am einfachsten überprüfen, wenn Sie eine sehr große Schriftart auswählen.



DIE ART UND WEISE, wie der Anfang und das Ende einer Linie gezeichnet werden sollen, kann ebenfalls vorgegeben werden. Damit lassen sich sehr leicht Pfeile und andere Symbole malen.

Der SolidBrush ist die langweiligste : Hier wird einfach eine ganz normale Farbe ausgegeben. Als Farbe verwendet das Beispielprogramm dabei die unter *PenColor* eingestellte - wenn Sie als PenType *SolidColor* ausgewählt haben.

Ein TextureBrush füllt Flächen mit Texturen. Dabei liegt die Textur einfach als Bitmap vor und kann auf verschiedene Arten in der Fläche aufgebracht werden. Das Beispielprogramm verwendet den kleinen Foto-Ausschnitt von der Form als Textur. Interessante Effekte erhält man dabei dann, wenn man die Breite des Striches verändert.

• Ein Pen wird erzeugt

Pens werden für das Ausgeben von Elementen verwendet, die eben nicht gefüllt werden. In diesem Fall wird der Pen mit Hilfe des Brushs erzeugt: Das führt dazu, dass die Einstellungen des Brushs auch Auswirkungen auf die Elemente haben, die mit dem Pen ausgegeben wurden. Alternativ kann man Pens aber auch einfach nur mit einer Farbe initialisieren. Die Breite des Pens bestimmt die Strichstärke. Wird diese größer, dann wird der Strich breiter. Pens haben noch eine ganze Reihe weiterer Eigenschaften. So können Sie im Programm den *DashStyle* verändern. Der legt fest, ob es sich um einen duchgezogenen Strich handelt - oder ob der Strich anderweitig gemalt wird. Die Eigenschaften *StartCap* und *EndCap* legen fest, wie der Anfang und das Ende eines Striches gemalt werden. Damit können Sie ganz einfach Pfeile erzeugen. Im Beispielprogramm werden noch eine Linie gezogen, ein Text ausgegeben und der gefüllte Kreis gemalt: jeweils mit den Eigenschaften, die über das UI des Programms gesetzt wurden (Listing 7).

LISTING 7:

```
e.Graphics.DrawLine(p, new Point(30, 30), new Point(200, 200));
e.Graphics.DrawString("gdi+", paintProps._Font, b, 30, 130);
e.Graphics.FillEllipse(b, 120, 30, paintProps.Radius, paintProps.Radius);
```

Serialisierung in .NET

Laden und Speichern

Egal, für welche Aufgabe eine Anwendung gedacht ist - eines muss sie fast immer tun: Daten vom Benutzer entgegen nehmen und speichern. Und zwar so, dass diese Daten später auch wieder geladen werden können.

THOMAS WÖLFER

Laden und Speichern war schon immer mit ein wenig Aufwand verbunden, denn schließlich müssen die Daten aus dem Speicher ja irgendwie auf die Festplatte gebracht werden. Der Aufwand ist aber mit .NET verschwunden - dank der eingebauten Serialisierung.

Eng verwandt damit sind die 'neuen' Common Dialogs und die Verwendung des Clipboards fürs Cut&Paste.

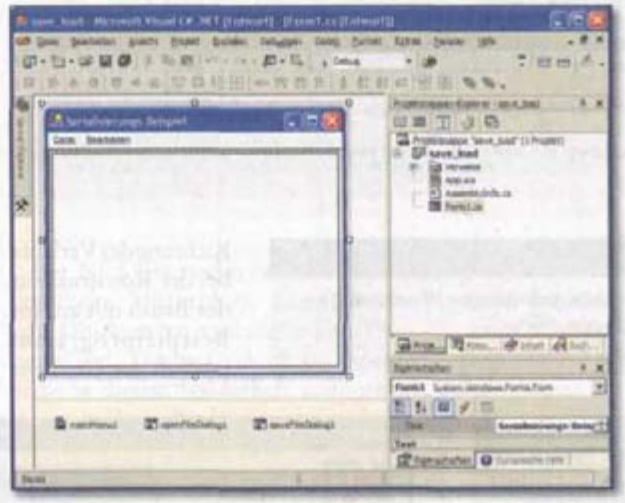
Als Beispielprogramm zu diesen Techniken implementieren Sie ein kleines Malprogramm. Die am Bildschirm erstellten Zeichnungen können Sie dann auf der Festplatte speichern oder von der Festplatte laden. Dazu verwenden Sie die Unterstützung der Common Dialogs für diese Zwecke.

Außerdem implementieren Sie die Unterstützung für's Clipboard: Ihre Zeichnungen können in die Zwischenablage kopiert und von dort in die Zeichnung zurück kopiert werden. Damit können Sie natürlich auch Daten zwischen zwei parallel laufenden Kopien des Malprogramms austauschen.

• Das Zeichenblatt

Damit überhaupt etwas demonstriert werden kann, muss es natürlich erst einmal ein kleines Malprogramm geben. Sie werden überrascht sein, wie leicht das zu programmieren ist.

Das Programm soll ein ganz einfaches Pixel-orientiertes Malen ermöglichen. Dabei bietet das Programm von Haus aus eine leere Zeichenfläche an. Bewegt man die Maus bei gedrückter linker Maustaste über dieser Zeichenfläche, dann werden Linien zwischen den verschiedenen Mauspositionen gezogen. Lässt man die Maustaste los, werden keine weiteren Linien mehr gezeichnet. Das Ganze ist also ein Art Mini-Skizzenblock, wobei keine besonderen Modi existieren: Sie können direkt loszeichnen, ohne erst einen Befehl auswählen zu müssen, und wenn Sie den Stift (die Maus) absetzen, dann wird eben nicht mehr gemalt. So lange, bis Sie wieder die linke Maustaste drücken und weiter zeichnen.



DEN GRÖSSTEN TEIL des Programms können Sie direkt im Forms-Editor gestalten.

Für diese Funktionalität brauchen Sie zunächst eine neuen Windows-Forms-Anwendung. Auf der Form zur Anwendung platzieren Sie dann eine Picture-Box und setzen deren Docking-Verhalten auf *Fill* (Füllen). Im Beispielprogramm kommt dabei auch noch ein Panel ins Spiel - der ist aber nur für die Optik notwendig so dass Sie das Panel zunächst ruhig weglassen können.

Ist die PictureBox platziert, brauchen Sie zwei Event-Handler. Einen für das MouseMove-Ereignis der PictureBox, einen zweiten für das MouseUp Ereignis des gleichen Kontrollelementes.

MouseMove wird immer dann ausgelöst, wenn die Maus über dem Fenster des Controls bewegt wird, während *MouseUp* dann ausgelöst wird, wenn die Maus vom gedrückten in den nicht gedrückten Zustand übergeht.

Das wird im Programm als Signal zum absetzen des 'Stiftes' interpretiert. Gezeichnet wird dabei nur dann, wenn die linke Maustaste gedrückt ist. Wird die Maus also über das Fenster bewegt ohne das diese Taste gedrückt wird, dann erzeugt das auch keine Striche. Nachdem die einzelnen Striche auch später wieder ausgegeben werden müssen - zum Beispiel, wenn sich das Fenster in seiner Größe ändert - müssen dieper MouseMove-Behandlung eingesammelten Punkte natürlich gespeichert werden. Das geht am einfachsten in einer ArrayList. Davon legen Sie als zuerst ein privates Member in der Form an:

```
private ArrayList list =
➥ new ArrayList();
```

Weiterhin brauchen Sie noch einen Ausgangspunkt fürs Zeichnen. Wird der erste Punkte ausgewählt, dann ist das ja für das Zeichnen einer Linie noch nicht ausreichend: Es werden immer zwei Punkte benötigt.

Aus diesem Grund erzeugen Sie noch ein weiteres privates Member in der Form. Dieses Member ist vom Typ Point und nimmt immer den zuletzt ausgewählten Punkt auf. Ist noch kein Punkt ausgewählt worden, dann enthält dieses Member den Wert *Point.Empty*. Auf diesen Wert initialisieren Sie den Punkt auch:

```
private Point last = Point.Empty;
```

Nun ist es an der Zeit, den MouseMove Event zu behandeln. Dazu aktivieren Sie zunächst die PictureBox in der Form, damit Sie in der Eigenschafts-Ansicht von Visual Studio die Eventhandler definieren können. Der, den Sie definieren müssen ist der Handler für das MouseMove Ereignis (Listing 1).

LISTING 1:

```
private void OnMove(object sender, System.Windows.Forms
➥.MouseEventArgs e)
{
  if( e.Button == MouseButtons.Left)
  {
    Graphics g = Graphics.FromHwnd
    ➥(this.pictureBox1.Handle);
    Point p = new Point( e.x, e.Y);
    list.Add( p);
    if( last != Point.Empty)
    {
      g.DrawLine( new Pen( Color.Blue, 1f), last, p);
    }
    last = p;
  }
}
```

Der Event-Handler bekommt als zweites Argument eine Instanz eines MouseEventArgs-Objekts. Dieses Objekt enthält alle Informationen, die Sie brauchen, um das Ereignis nach Wunsch zu behandeln. Zunächst überprüfen Sie, ob die linke Maustaste gedrückt ist. Ist das nicht der Fall, dann ist das Ereignis nicht weiter von Interesse – Es soll ja nur dann gemalt werden, wenn diese Taste gedrückt ist.

Ist die Taste gedrückt, gilt es, eine Linie zu zeichnen - zumindest dann, wenn es schon einen Startpunkt dafür gibt. Dazu erzeugen Sie zunächst ein Graphics Objekt auf Basis der PictureBox. Das Graphics Objekt ist das Objekt, das alle Methoden enthält, mit denen tatsächlich gemalt werden kann.

Als Nächstes erzeugen Sie einen neuen 'Point' anhand der Koordinaten der aktuellen Mausposition. Die erhalten Sie genau wie die gedrückte Maustaste aus den MouseEventArgs. Diesen Punkt merken Sie sich in der ArrayList, damit Sie ihn auch später nochmals malen können.

Dann überprüfen Sie, ob bereits ein Startpunkt anliegt. Dieser Punkt stammt aus einer vorherigen MouseMove Behandlung. Wenn es also noch keinen MouseMove Event gab, dann liegt auch noch kein Startpunkt an, und es kann nichts gezeichnet werden.

Liegt hingegen schon ein Startpunkt an - ist also 'last' ungleich einem leeren Punkt - dann verwenden Sie die DrawLine()-Methode des Graphics Objekts, um zwischen dem vorigen und dem aktuellen Punkt eine Linie auszugeben.

Das ist natürlich noch ein bisschen langweilig: Sie können aber Einfluss auf die Farbe, die Form um die Strichstärke nehmen, indem Sie den zum Zeichnen verwendeten Pen anders als im Beispiel erzeugen. Dazu finden Sie Details in diesem Sonderheft.

Schließlich merken Sie sich noch den Punkt in 'last', damit beim nächsten MouseMove Event die folgende Linie gezeichnet werden kann.

Zu guter Letzt ist es noch notwendig, das Malen auch irgendwann zu unterbrechen. Das passiert einfach dadurch, dass der MouseButton wieder losgelassen wird. Mit anderen Worten, Sie müssen das MouseUp Event behandeln. Der Handler muss dazu aber nicht viel tun.

Es ist völlig ausreichend, den Startpunkt auf einen leeren Punkt zu setzen, denn dann wird bei der nächsten MouseMove-Behandlung nichts mehr gezeichnet (Listing 2).

Damit haben Sie schon ein ganz brauchbares interaktives Zeichenprogramm. Nur leidet das Programm noch unter einer kleinen Einschränkung,

LISTING 2:

```
private void OnUp(object sender,
➥System.Windows.Forms.
➥MouseEventArgs e)
{
  last = Point.Empty;
  list.Add( last);
}
```

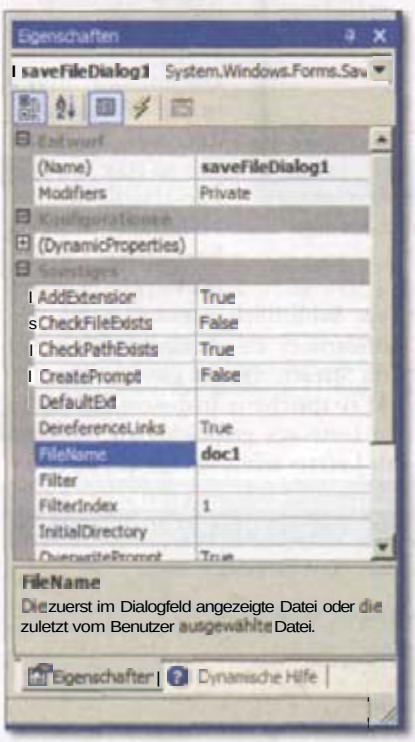
denn wenn Sie das Fenster in seiner Größe verändern, dann werden die zuvor gezeichneten Linien nicht nochmals ausgegeben und das Bild geht verloren.

Dazu haben Sie aber zuvor alle Punkte in dem ArrayList-Objekt gespeichert. Sie brauchen also nur noch das richtige Ereignis zu behandeln und dabei die Liste der Punkte zum Zeichnen verwenden. Das passenden Ereignis ist das Paint-Event der PictureBox. Im Event-Handler dieses Ereignisses iterieren Sie einfach über alle gespeicherten Punkte und verwenden Sie zum Zeichnen der Linien. (Listing 3).

Damit ist das Programm mehr oder minder komplett - zumindest für die Belange, die Sie für den Rest des Beispiels brauchen:

LISTING 3:

```
private void OnPaint (object sender, System.
➥Windows.Forms.PaintEventArgse)
{
  Point pLast = Point.Empty;
  foreach( Point p in list)
  {
    if( p != Point.Empty)
    {
      if( pLast != Point.Empty)
      {
        e.Graphics.DrawLine( new Pen
        ➥( Color.Blue, 1f), pLast, p);
      }
    }
  pLast = p;
  }
}
```



DIE ART UND WEISE, wie der fileSave-Dialog funktionieren soll, können Sie in der Eigenschafts-Ansicht einstellen.

Sie können nun interaktiv Daten erzeugen, und das ist eine prima Grundlage, um die Serialisierungsfunktionen von .NET zu erkunden.

• Laden und Speichern: Der .NET-Weg

Für's Laden und Speichern von Daten brauchen Sie zunächst einmal die passenden Befehle in einem Menü - das bedeutet, Sie brauchen zunächst einmal ein Menü. Dazu ziehen Sie ein *MainMenu* aus der Toolbox auf die Form. Visual Studio zeigt dann oberhalb der Form den Menü-Editor an, in dem Sie den Befehl zum Laden und zum Speichern eintragen. Bei der Gelegenheit können Sie gleich auch noch einen Befehl zum Beenden des Programms einfügen.

Beim Laden und Speichern von Dateien empfiehlt es sich, die von Windows bzw. .NET vorgesehenen 'Common'-Dialogs zu verwenden, denn dann hat

DAS MENÜ DES PROGRAMMS können Sie im Menü-Editor erzeugen. Dazu tippen Sie die gewünschten Texte für die Menü-Befehle einfach ein.

Ihr 'Datei laden'-Dialog das gleiche Aussehen, wie das bei allen anderen Windows-Programmen der Fall ist. Auch für diesen Zweck gibt es eine Komponente in der Toolbox. Sie suchen also den 'saveFile'- und den 'openFile'-Dialog in der Toolbox und ziehen davon jeweils eine Kopie auf die Form.

Dann doppelklicken Sie auf den *Laden*- bzw. den *Speichern*-Befehl im Menü-Editor. Visual Studio erzeugt dann passende Event-Handler, die Sie nur noch auffüllen müssen. Dabei verwenden Sie das 'fileSave'- bzw. das 'fileOpen'-Objekt, um dem Anwender eine Möglichkeit zu geben, die gewünschte Datei auszuwählen.

Hat der Anwender diesen Dialog mit dem 'OK'-Button beendet, dann liefert der Dialog den Wert *DialogResult.OK* zurück. Nur in diesem Fall sind weitere Aktionen notwendig, und dabei geht es entweder um Speichern oder um Laden der Dateien.

Zum Laden und Speichern von Daten bietet .NET einen eigenen Mechanismus an: Die Serialisierung. Beim Serialisieren werden ganze Objekt-Hierarchien auf einen Schlag auf die Festplatte gespeichert. Wenn Sie also komplex zusammengesetzte Objekt haben, dann ist das Speichern genau so leicht, als wenn Sie nur ein einfaches Objekt hätten. Alles was Sie dazu tun müssen, ist der Serialisierung das Top-Level Objekt zu übergeben. Alle darin enthaltenen Objekte und Daten werden auf die Platte geschrieben, ganz ohne weiteres Zutun.

Die Art und Weise, wie die Daten auf der Platte abgelegt werden, ist dabei von einem Formatierung-Objekt abhängig. .NET bietet dazu zwei unterschiedliche Geschmacksrichtungen an: einen binären Formatierer und einen XML-Formatierer. Der eine schreibt die Daten in binärer Form auf die Platte, der andere in Form einer XML-Datei. Beides hat seine Vorteile und seine Nachteile.

Der binäre Formatierer ist deutlich schneller als der XML-Formatierer, dafür sind die Daten des XML Formatierers leicht von Menschen zu lesen und

DER DIALOG ZUM SPEICHERN VON DATEN stammt aus einer Toolbox-Komponente.

können auch von XML-fähigen Programmen gelesen werden. Eine von .NET per XML-Formatierer erzeugte Datei kann zum Beispiel einfach im Internet Explorer geladen werden. Der Explorer zeigt dann die Objekt-Hierar-

DAS MALPROGRAMM IN AKTION: Freihandzeichnungen sind damit leicht möglich.

chie innerhalb der Datei in Form eines XML-Baums an. Im Beispiel wird einfach der binäre Formatierer verwendet. Wenn Sie statt dessen den XML-Formatierer verwenden wollen, dann geht das einfach dadurch, dass Sie im Quelltext den 'BinaryFormatter' durch einen 'XMLFormatter' ersetzen.

Zum Speichern der Daten brauchen Sie einen Pfad - den der Datei, in der gespeichert werden soll. Diesen Pfad erhalten Sie aus dem fileSave-Dialog.

Dann müssen Sie ein Stream Objekt erzeugen. Ein Stream ist im Wesentlichen das, was Sie unter Win32 oder C++ als File-Handle kennen. Nun erzeugen Sie ein Formatierer-Objekt vom gewünschten Typ. Schließlich verwenden Sie den Formatierer im Zusammenhang mit dem Stream, um das gewünschte Objekt zu speichern. In diesem Fall ist das die Liste aus gespeicherten Punkten. Als Letztes schließen Sie den Stream -

LISTING 4:

```
private void menuItem2_Click(object sender, System.EventArgs e)
{
  if( DialogResult.OK == this.saveFileDialog1.ShowDialog() )
  {
    string path = this.saveFileDialog1.FileName;
    Stream file = File.Open( path, FileMode.Create);
    IFormatter formatter = new BinaryFormatter();
    formatter.Serialize(file, Hat);
    file.Close();
  }
}
```

die Daten liegen dann auf der Festplatte (Listing 4).

Beim Laden der Daten gehen Sie praktisch genauso vor - nur werden die Daten dabei nicht serialisiert, sondern eben deserialisiert. Außerdem invalidieren Sie nach dem Laden die PictureBox, damit die Daten auch sofort angezeigt werden (Listing 5).

Nun kann Ihr Programm die eingegebenen Daten auch speichern und wieder laden. Allerdings sind die Dialoge zum Laden und Speichern noch ein wenig dürftig - es fehlt zum Beispiel noch eine Dateierweiterung, eine Vorgabename und einige anderer hilfreiche Parameter. All diese Parameter können Sie aber direkt in der Eigenschafts-Ansicht der entsprechenden Komponenten bearbeiten, daher wird in diesem Beispiel nicht weiter darauf eingegangen.

Das Serialisieren und Deserialisieren von Daten können Sie aber nicht nur für das Ablegen von Daten auf der Festplatte verwenden, Sie können sie auch

DAS ZEICHENPROGRAMM unterstützt auch die Zwischenablage.

für die Clipboard-Unterstützung Ihres Programms verwenden.

Die Zwischenablage ist in .NET extrem einfach zu verwenden. Eigentlich funktioniert sie genau so, wie auch das Serialisieren von Objekten funktioniert. Der einzige Unterschied besteht darin, dass Sie nicht in eine Datei Serialisieren, sondern in einen Speicherbereich im RAM. Diesen Speicherbereich können Sie dann mehr oder minder direkt ins Clipboard kopieren, wobei allerdings noch eine eindeutige Beschreibung mit gespeichert wird. Im Beispiel wird dazu einfach eine E-Mail-Adresse verwendet.

Natürlich brauchen Sie dazu auch noch die passenden 'Copy'- und 'Paste'-Befehle im Menü. Die legen Sie genau so an, wie zuvor die Befehle zum Speichern und Laden. Der Handler für den Copy-Befehl hat das Aussehen wie in Listing 6. Es kommen also der gleiche Formatter und die gleiche Serialisierung ins Spiel, die Sie schon kennen. Anders als beim Speichern verwenden Sie als Ziel der Speicherung aber keinen Stream, sondern einen MemoryStream.

Dann brauchen Sie noch ein DataObjekt. Dieses DataObjekt ist der Container für die Daten, die Sie in die Zwischenablage kopieren wollen. Das DataObjekt wird dann mit den Daten und dem eindeutigen Bezeichner versehen und dann mit *Clipboard.SetData-Objekt* ins Clipboard kopiert.

Der Paste-Befehl funktioniert in genau umgekehrter Richtung. Allerdings ist es im Gegensatz zum Laden der Daten so, dass die per Zwischenablage eingefügten Objekte zu den bereits vorhandenen hinzugefügt werden sollen. Darum können Sie nicht einfach direkt in die Liste deserialisieren. Statt dessen verwenden Sie dazu eine temporäre andere Liste, aus der Sie die Daten in die eigentliche Liste kopieren (Listing 7).

Das Programm unterstützt nun auch die Zwischenablage. Das können Sie einfach testen, indem Sie zwei Kopien Ihres Programms laden und dann in der einen Kopie ein Bild malen. Ist das Bild fertig, betätigen Sie den *Copy*-Befehl und wechseln in die zweite Kopie des Programms. Wenn Sie dort den *Paste*-Befehl verwenden, werden die Daten in dessen Grafik eingefügt. © U R.

LISTING 5:

```
private void menuItem3_Click(object sender, System.EventArgs e)
{
  if( DialogResult.OK == this.openFileDialog1.ShowDialog())
  {
    string path = this.openFileDialog1.FileName;
    Stream file = File.Open( path, FileMode.Open);
    IFormatter formatter = new BinaryFormatter();
    Hat = formatter.Deserialize(file) as ArrayList;
    file.Close();
    this.pictureBox1.Invalidate();
  }
}
```

LISTING 6:

```
private void menuItem8_Click(object sender, System.EventArgs e)
{
  DataObject data = new DataObject();
  MemoryStream file = new MemoryStream();
  IFormatter formatter = new BinaryFormatter();
  formatter.Serialize(file, list);
  data.SetData( "samples@woelfer.com", false, file);
  Clipboard.SetDataObject( data, true);
}
```

LISTING 7:

```
private void menuItem9_Click(object sender, System.EventArgs e)
{
  IDataObject d = Clipboard.GetDataObject();
  object objcb = d.GetData( "samples@woelfer.com");
  MemoryStream file = objcb as MemoryStream;
  IFormatter formatter = new BinaryFormatter();
  ArrayList ar = formatter.Deserialize( file) as ArrayList;
  foreach( Point p in ar)
  {
    list.Add( p);
  }
  this.pictureBox1.Invalidate();
}
```

Alles ganz einfach

Datenbank-zugriffe mit .NET

Daten sind wirklich lästig - in praktisch jeder Anwendung braucht man welche, und immer wollen die auch bearbeitet und gespeichert werden. Das machte bisher große Mühe. Mit .NET wird das einfach.

THOMAS WÖLFER

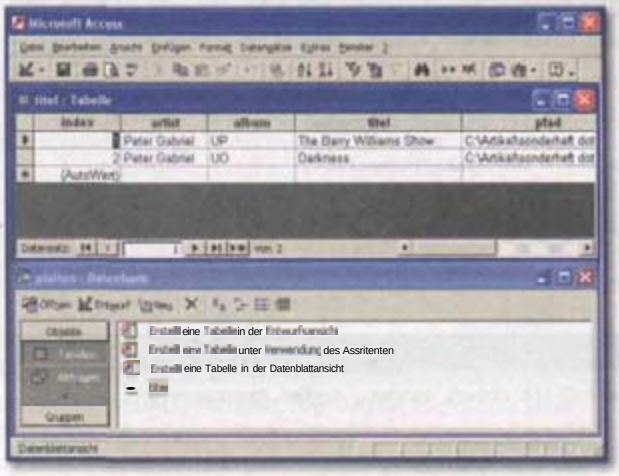
Das .NET Framework hat die Unterstützung für Datenbankzugriffe von Haus aus fest eingebaut. Dabei kommen die ADO.NET-Klassen zum Zuge, die in Form der ADO Api teilweise auch unter Win32 schon verfügbar waren. Jetzt als .NET-Klassen aber in dramatisch einfacher verfügbarer Form: ADO.NET bietet Zugriff auf verschiedenste Datenbanken - und zwar entweder auf 'echte' Datenbank-Serveranwendungen wie den SQL-Server - oder auf Datenbank-Dateien wie zum Beispiel auf die .MDB-Dateien von Access.

Der Zugriff auf die Daten erfolgt dabei in einer einheitlichen Form und immer mit den gleichen Objekten: Ob die Daten also in einem SQL-Server liegen oder aber in einer MDB-Datei verpackt wurden, ist völlig gleichgültig.

Und wie nicht anders zu erwarten war, gibt es auch einige fertige Komponenten, die man 'einfach so' per Drag & Drop verwenden kann, um auf die Datenbanken zuzugreifen. Ähnlich elegant, wie das an anderer Stelle im Heft besprochene PropertyGrid eine visuelle Möglichkeit der Bearbeitung von .NET-Objekten schafft, schafft Ado-.NET in Zusammenarbeit mit dem DataGrid-Objekt einfachsten Datenbankzugriff per Drag & Drop.

Zusätzliche Unterstützung erhält man dabei von einem Assistenten, der die grundlegend notwendigen Elemente einer Windows Form, die mit einer Datenbank zusammenarbeiten soll für den Programmierer richtig zusammensetzt.

Hier werden Sie die Drag&Drop-Methode der Datenbankzugriffe bei .NET kennen lernen. Es wird aber auch die Funktionsweise der beteiligten Klassen im Detail geklärt, denn man will ja nicht immer alles aus der Toolbox zusammenklicken und -ziehen.



DIE **DATENBANK-TABELLE** wird ganz normal mit Access erzeugt. Man kann aber Tabellen mit ADO.NET auch programmatisch erzeugen.

Ganz nebenher bauen Sie dabei eine kleine Datenbank-basierte Software für das Verwalten und Abspielen von Multimediatiteln zusammen und lernen dabei die einfachste Methode kennen, Multimedia-Dateien mit Hilfe der 'Process'-Klasse abzuspielen.

• Daten - woher nehmen und nicht stehlen ?

ADO.NET ist es völlig gleichgültig, wo die Daten herkommen: Egal, ob es sich bei der Quelle um eine ODBC-Treiber, um den JET-Treiber für .MDB oder um einen SQL-Server handelt: Der Zugriff erfolgt immer auf die gleiche Art und Weise. Programmiert man aber an einem normalen Anwendungsprogramm und nicht an einer Lösung für ein Unternehmen, dann werden die Daten nahezu nie in einem SQL-Server sondern fast immer in einer Access-Datenbank in Form einer .MDB-Datei vorliegen. Aus diesem Grund wird das in diesem Beispiel so vorausgesetzt - die Datenbank wird initial einmal mit Access angelegt und kann dann später mit dem fertigen Beispielprogramm bearbeitet werden.

Die Beispieldatenbank ist wirklich sehr einfach: Die Datenbank enthält eine einzige Tabelle, und die enthält Spalten, in denen die Informationen über die Multimedia-Titel gespeichert werden. Das sind der Name des Künstlers, das Album, der Titel des Stücks und der Pfad zur Datei auf der lokalen Festplatte. Dass damit keine besonders elegante Verwaltung von Titeln möglich ist, ist klar- aber es handelt sich ja auch nur um ein Beispiel, das möglichst übersichtlich gehalten werden soll.

Ist die Tabelle angelegt - oder von der Heft-CD kopiert - legen Sie ein neues C# 'WindowsApplication'-Projekt an. Das besteht, wie immer bei solchen Projekten, zunächst aus einer einzelnen lee-

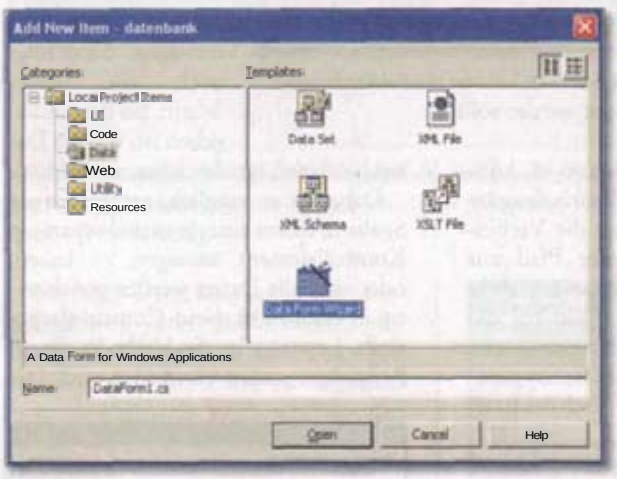
ren Form. Die wird später verwendet, um das eigentliche Front-End des Multimedia-Players darzustellen. Zunächst ist diese Form aber nicht von Interesse.

• Datenzugriff mit der DataForm

Der einfachste Weg, eine Verbindung mit einer Datenbank aufzunehmen, läuft über den DataForm Wizard. Den finden Sie mit dem Befehl *AddItem*, wenn Sie dann aus der vorgeschlagenen Liste der hinzufügbaren Elemente die *DataForm* auswählen.

Nach der üblichen Wizard-Begrüßung, in der einem versichert wird, dass nun alles ganz einfach und unproblematisch abläuft, kommt es im nächsten Schritt sofort zu einem kleinen Problem. Der Wizard hätte gerne gewusst, wie das zu verwendende DataSet heißen soll. Das Problem dabei ist, dass man, so man zum ersten Mal mit Datenbanken konfrontiert ist, vermutlich nicht weiß, was ein DataSet überhaupt ist.

ZUNÄCHST BRAUCHT MAN ein neues Dataset-Objekt. Das DataSet-Objekt kapselt die Daten, die man verwalten möchte.



DER EINFACHSTE WEG zur Datenbank erfolgt über den *DataForm Wizard.*

Daher ist an dieser Stelle ein kurzer Seitenblick notwendig. Tiefergehende Informationen finden Sie im Kasten unten.

Bei einem DataSet handelt es sich um einen im Speicher befindlichen Cache von Daten, die aus einer Datenbank entnommen wurden. Jedes DataSet besteht dabei aus einer Sammlung von DataTables die ihrerseits Tabellen einer Datenbank abbilden und die tatsächlichen Daten beinhalten. Damit ein solches Objekt erzeugt werden kann, muss natürlich bekannt sein, wie die Tabellen in der Datenbank aussehen und welche Daten sie enthalten. Diese Informationen speichert das Visual Studio später in einer XSD-Datei (XML Schema Definition Language), die dann im Prinzip eine neue Klasse definiert und praktisch genauso verwendet werden kann wie eine C# Klasse.

Der Wizard will an dieser Stelle noch gar nichts über die Daten selbst wissen - alles, was Sie hier angeben, ist der Klassenname der Klasse, die Sie später verwenden wollen. Mit anderen Worten: Alles was Sie tun müssen, ist sich einen passenden Namen für Ihre Datensammlung auszusuchen. Also zum Beispiel *songs* - denn das beschreibt die zu verwaltenden Daten aus dem Beispielprogramm recht gut.

Nun hätte der Wizard gerne gewusst, mit welcher Datenbank Sie zusammenarbeiten möchten. Dabei geht er unpraktischerweise immer davon aus, dass es sich zumindest um eine SQL-Server-Datenbank handelt.

Das erkennt man beim Wizard aber nicht auf den ersten Blick, denn der öffnet von Haus aus den Reiter *Verbindung*, auf dem nur noch die Verbindungsart festgelegt werden kann: Der Typ der Datenquelle ist schon als SQL-Server vorausgewählt.

UM ZUGRIFF AUF DIE DATENBANK zu erhalten braucht man eine Verbindung zur Datenbank. Dabei muss die Zugriffsmethode spezifiziert werden. Im Falle einer .MDB Datei ist der *Jet 4.0*-Provider der richtige.

Für den Zugriff auf eine .MDB Datei müssen Sie zunächst auf den Reiter *Provider* wechseln und dort von dem SQL-Server Provider auf den *Jet4 OLE DB*-Provider wechseln. Dann schalten Sie

DIE WICHTIGSTEN ADO.NET-OBJEKTE.

Bei der Arbeit mit Datenbanken arbeiten eine ganze Reihe von ADO.NET und normalen .NET-Elementen Hand in Hand zusammen.
Hier eine Kurzübersicht:

Die XSD-Datei
Diese Datei enthält die XSD-Beschreibung eines DataSet Objekts. Diese beschreibt die Form der Daten, die das DataSet Objekt aufnehmen kann.

Das DataGrid
Hierbei handelt es sich um eine visuelle Komponente, die der Bearbeitung von Datenbank-Daten dient.
Mit dem DataGrid können Sie Daten genau so bearbeiten, wie das zum Beispiel auch in Access oder dem SQL-Server Manager der Fall ist.

Die OleDBConnection
Dabei handelt es sich um ein Objekt das die Verbindung zur Datenbank kapselt. Die wesentliche Information, die dieses Objekt beinhaltet, ist der Connection-String

Der OleDbDataAdaper
Der DataAdapter kümmert sich um das Mapping von Datenbank-Tabelle und Inhalten auf solche des DataSets.
Dadurch wird es zum Beispiel möglich, im DataSet andere Bezeichner zur verwenden, als die ursprünglich in der Datenbank verwendeten.
Die OleDB*Commands
Der [*] steht dabei für die verschiedenen Kommando-Arten. Es gibt beispielsweise ein SelectCommand, das für die Auswahl von Daten zuständig ist, aber auch ein DeleteCommand, das für das Löschen von Daten gebraucht wird.

zurück auf den Reiter *Verbindung* - und der sieht dann deutlich verständlicher aus, als zuvor, denn nun kann man dort den Pfad zu einer Datenbank angeben.

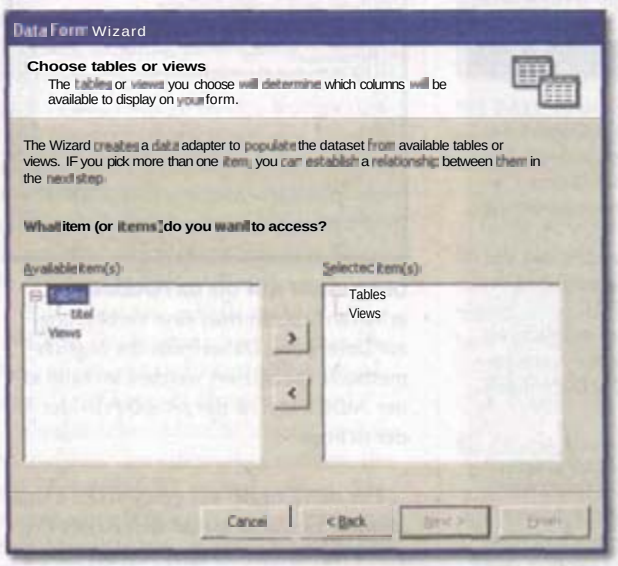
Diesen Pfad wählen Sie aus und geben bei Bedarf auch noch Anmeldeinformationen für die Datenbank an. Im Falle der Beispieldatenbank ist das nicht notwendig, denn die ist in keiner Weise geschützt.

IM FALLE EINER ACCESS-DATEI gibt man den Pfad zur Datei an. Der Wizard erzeugt dann den passenden Connection-String.

Die eingegebenen Daten verwendet der Wizard im Wesentlichen dazu, um den Verbindungs-String für die Aufnahme der Verbindung zur Datenbank durch den Jet-Treiber vorzunehmen.

Ist der Pfad angegeben, sollte der Kontaktaufnahme nichts mehr im Wege stehen. Ob das auch wirklich so ist, können Sie mit dem Button *Verbindung testen* ausprobieren. Schlägt die Verbindung fehl, so werden der Pfad zur .MDB-Datei oder das Passwort nicht stimmen. Ein anderer Grund für das Fehlschlagen der Verbindung kann sein, dass die Zugriffsrechte auf Dateisystem-Ebene nicht richtig sind - auch das ist sicherzustellen.

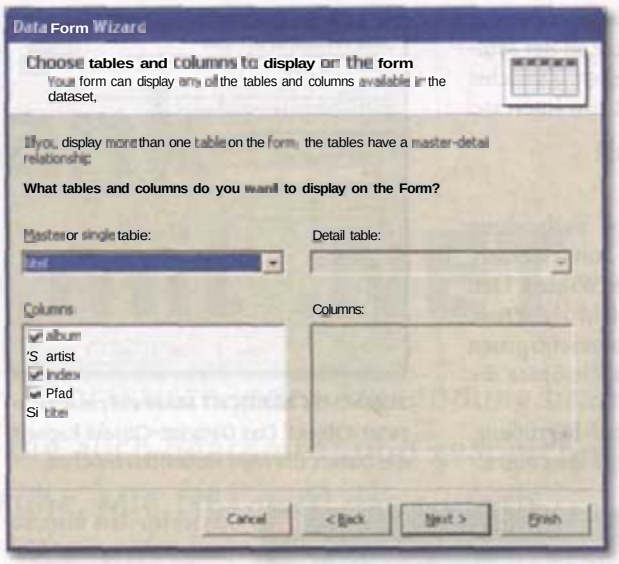
Hat der Verbindungstest geklappt, gelangen Sie zum nächsten Dialog des Wizards. Hier ist anzugeben, welche der Elemente aus der Datenbank verwendet werden sollen. Eine Datenbank kann ja mehrere Tabellen, gespeicherte Prozeduren und auch weitere Elemente, wie zum Beispiel Ansichten, enthalten.



IST DIE VERBINDUNG zur Datenbank hergestellt, legt man fest, auf welche Elemente daraus zugegriffen werden soll.

Für das Beispielprogramm ist die Sache einfach, denn es befindet sich nur eine einzelne Tabelle in der Datenbank. Das ist logischerweise auch die Tabelle, die verwendet werden soll. Dazu wählt man in der linken Liste der verfügbaren Elemente die Tabelle aus und klickt dann auf den Button '>'. Dadurch taucht die Tabelle rechts in der Liste auf.

Die Tabelle ihrerseits enthält ja nun einige Spalten – und unter Umständen will man nicht alle davon

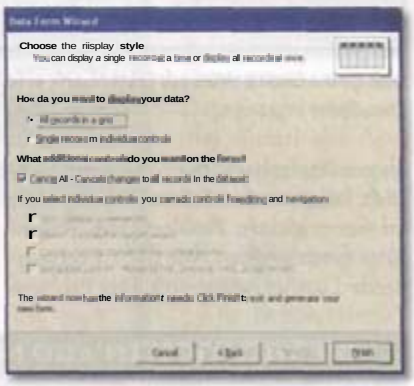


AUS DER AUSGEWÄHLTEN TABELLE können die Felder ausgewählt werden, die verwendet werden sollen.

im Programm anbieten. Dazu dient der folgende Dialog des Wizards. Hier legt man fest, welche der Spalten verwendet werden sollen und welche nicht. Für das Beispielprogramm werden einfach alle Spalten aktiviert und dann der 'Next'-Button betätigt.

Damit ist definiert, welche Daten konkret verwendet werden sollen. Der Wizard kann jetzt das DataSet erzeugen. Nun folgt noch ein letzter Schritt, bei dem anzugeben ist, wie die Daten bearbeitet werden können sollen.

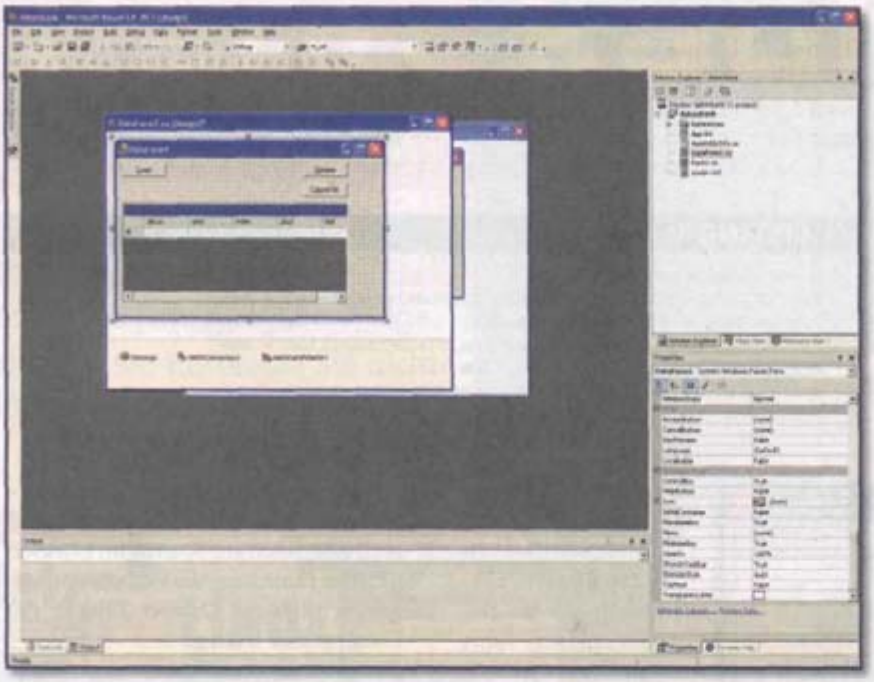
Dabei ist es möglich, entweder jede Spalte in einem einzelnen und separaten Kontrollelement anzeigen zu lassen, oder aber alle Daten werden gleichzeitig in einem DataGrid-Control dargestellt. Letzteres ist die Methode, die im Beispielprogramm verwendet wird. Das



DER WIZARD erlaubt es, die Daten entweder in einem Grid oder in einzelnen Kontrollelementen darzustellen.

DataGrid ist im Zusammenhang mit den ADO.NET-Klassen ein ähnlich mächtiges Werkzeug wie das PropertyGrid im Zusammenhang mit .NET Objekten – man kann es sich am einfachsten als eine Art Access ohne Access-GUI vorstellen.

Wird dieser Schritt des Wizards – und damit der Wizard selbst - beendet, passieren zwei Dinge. Zum einen erzeugt der Wizard eine XSD-Datei, die das DataSet beschreibt, zum anderen erzeugt der Wizard eine Windows-Form, auf der ein DataGrid und einige zusätzliche Buttons angebracht sind.



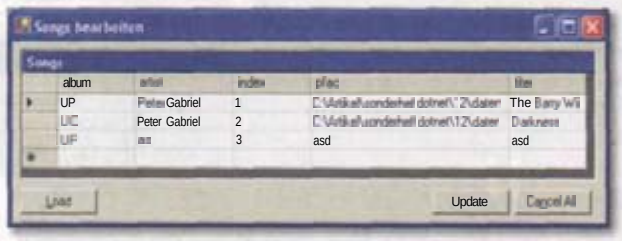
DIE FERTIGE FORM ist zwar nicht hübsch - aber funktional. Optische Änderungen sind jedoch ein Kinderspiel.

Die XSD-Datei können Sie direkt innerhalb von Visual Studio betrachten - der XML-Editor kann diese Datei prima anzeigen. Die Form ist eine ganz normale Form, nur ist eben bereits alles darauf enthalten, um die Datenbank zu bearbeiten.

Um die Form nun zu testen, muss nicht viel passieren: Sie legen einfach auf der ersten Form einen Button an und definieren für diesen einen Click() Handler der die DataForm anzeigt:

```
DataForm1 f = new DataForm1();
f.Show();
```

Die Form können Sie jetzt direkt verwenden, um die Datenbank zu bearbeiten. Allerdings bedarf sie noch ein wenig der optischen Nachbearbeitung, denn wirklich ansprechend ist das vom Wizard erzeugte Layout nicht.



MIT EIN WENIG NACHARBEITUNG sieht die Form ganz passabel aus.

Das Nachbearbeiten geht mit den ganz normalen Werkzeugen im Visual Studio, daher wird an dieser Stelle nicht weiter darauf eingegangen.

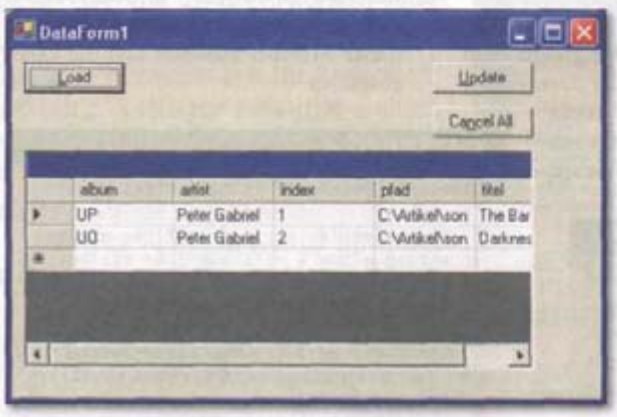
Die DataForm verfügt über vier Kontroll-Elemente. Das erste, das Sie davon benötigen, ist der 'Load' Button. Dieser lädt die Daten aus der Datenbank in ein passendes DataSet-Objekt und stellt diese dann im DataGrid dar. In diesem Grid können Sie die Daten ganz so bearbeiten, als würden Sie innerhalb von Access (oder dem SQL Server) arbeiten.

Sind alle Änderungen vorgenommen worden, können Sie mit dem *Update*-Button dafür sorgen, dass diese Änderungen aus dem lokalen DataSet in die zugrunde liegende Datenbank übertragen werden. Beim nächsten Load werden dann bereits die veränderten Daten verwendet. Der *Cancel All*-Button verwirft alle gemachten Veränderungen - die Originaldatenbank wird dann in keiner Weise angetastet.

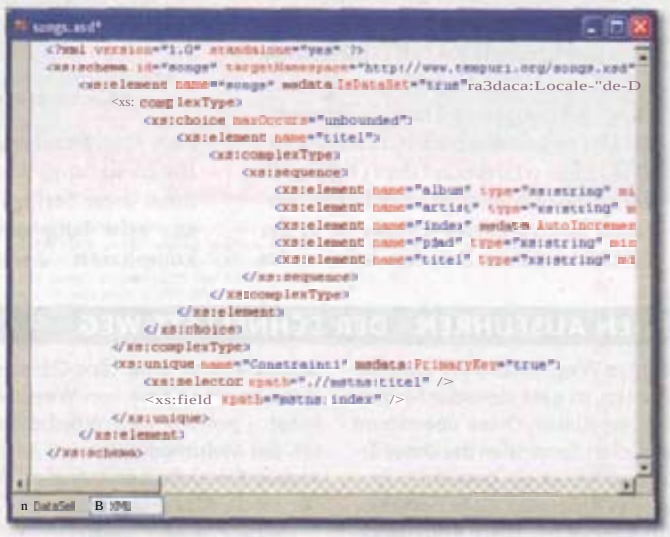
• Wer tut was – ohne Drag & Drop

Nun ist das sicherlich alles sehr praktisch - aber auch ein wenig undurchsichtig. Nachdem alle grundlegenden Operationen vom Wizard in Code gegossen werden, ist nicht so ganz klar, an welcher Stelle welche Klasse eigentlich welche Aufgabe übernimmt.

Das klärt sich aber, wenn Sie das Front-End des Multimedia-Players implementieren. In ihm können die Daten angezeigt und per Doppelklick abgespielt werden. Für die Anzeige der Daten verwenden Sie aber diesmal kein DataGrid, sondern eine normale ListView. Das hat den Vorteil,



DIREKT NACH DEM ABLAUF des Wizards hat man bereits eine Form, mit der die Daten bearbeitet werden können.



DIE BESCHREIBUNG DES DATASETS liegt in Form von XSD in XML vor.

dass die 'automatisch' ablaufenden Vorgänge von Hand nachgebaut werden müssen und darum deutlich verständlicher werden.

Übrigens: Beim Beispielprogramm wurde der Pfad zur Datenbank fest im Programm - also im Connection-String hineincodiert. Wenn Sie das bei sich ausprobieren möchten, dann müssen Sie diesen Pfad an Ihre lokalen Gegebenheiten anpassen.

Die Daten sollen auf der ersten Form des Projektes also in einer ListView angezeigt werden. Das bedeutet, dass Sie zunächst per Drag & Drop eine solche

DAS FERTIGE PROGRAMM hat das abgebildete Aussehen - ein einfacher, aber effektiver Player, der auf Datenbank-Daten zurückgreift.

View auf der Form platzieren müssen. Das ist hier die letzte Drag & Drop-Operation - ab jetzt folgt nur noch Quellcode. Den platzieren Sie im Load() Event-Handler der Form.

Dabei ist in zwei Schritten vorzugehen. Im ersten Schritt muss ein DataSet Objekt aufgefüllt werden, im zweiten müssen die Daten aus diesem Objekt in die ListView übertragen werden.

Für's Auffüllen eines DataSet-Objekts brauchen Sie insgesamt gleich vier Objekte aus der ADO.NET-Bibliothek: Ein Kommando-Objekt, ein Verbindungs-Objekt, einen Data-Adapter und das DataSet-Objekt selbst. Das DataSet-Objekt basiert dabei auf dem Objekt, für das Sie einen Namen ganz zu Beginn vergeben hatten. Dieses Objekt ist in der .XSD-Datei ihres Projekts definiert. Im Beispiel hat die Klasse den Namen 'songs' (Listing 1). Diese Objekte sind nun zu initialisieren. Am einfachsten ist das beim Kommando-Objekt; hier müssen eigentlich nur die zu verwendende Verbindung und das passende SQL-Query angegeben werden (Listing 2).

Dann initialisieren Sie das DataSet Objekt mit seinem Bezeichner (der führt dazu, dass die passende Beschreibung in der XSD Datei gefunden wird) und mit einem 'temporären' namespace, denn alle solche Objekte benötigen diese Angabe (Listing 3).

Nun müssen Sie den String für die Verbindungsaufnahmen des Connection-Objektes initialisieren:

```
oleDbConnection.
➥ConnectionString = ...
```

Den kopieren Sie am besten aus dem Initialisierungs-Code der DataForm, denn diese Strings sind nicht nur sehr lang sondern auch kompliziert - und der Wizard baut sie ganz wunderbar und vor allem auch korrekt zusammen.

Schließlich ist noch das DataAdapter-Objekt zu initialisieren. In diesem Fall braucht dieses Objekt ein 'SelectCommand' - also ein Kommando zum Auswählen von Daten aus der Datenbank - und eine Tabelle für's Mappen von Datenbankspalten auf Spalten im DataSet-Objekt. Dann kann die Verbindung aufgebaut und das DataSet Objekt mit Daten gefüllt werden:

```
oleDbConnection.Open();
oleDbDataAdapter.Fill( data);
```

Damit ist der erste Schritt erledigt: Sie haben eine lokale Kopie der Daten aus der Datenbank in Ihrem DataSet Objekt vorliegen. Über diese Daten können Sie nun iterieren und selbige dann in der ListView eintragen (Listing 4).

Den Pfad zur Multimedia-Datei merken Sie sich dabei im 'Tag'-Member der ListView. Wird später in der ListView eine Zeile angeklickt, dann können Sie dieses Member auslesen und die Datei abspielen. © UR

LISTING 1:

```
songs data = new songs();
System.Data.OleDb.OleDbCommand oleDbSelectCommand = new
System.Data.OleDb.OleDbCommand();
System.Data.OleDb.OleDbConnection oleDbConnection = new
System.Data.OleDb.OleDbConnection();
System.Data.OleDb.OleDbDataAdapter oleDbDataAdapter= new
System.Data.OleDb.OleDbDataAdapter();
```

LISTING 2:

```
oleDbSelectCommand.CommandText = "SELECT album, artist, pfad,
➥titel FROM titel";
oleDbSelectCommand.Connection = oleDbConnection;
```

LISTING 3:

```
data.DataSetName = "songs";
data.Namespace = "http://www.tempuri.org/songs.xsd";
```

LISTING 4:

```
foreach( DataRow r in data.titel.Rows)
{
  string artist = r["artist"] as string;
  string album = r["album"] as string;
  string titel = r["titel"] as string;
  string pfad = r["pfad"] as string;

  ListViewItem lvi = new ListViewItem
➥( new string[] { artist,album,titel});
  lvi.Tag = pfad;
  this.listView1.Items.Add( lvi);
}
```

DATEIEN AUSFÜHREN - DER SCHNELLSTE WEG

)er einfachste Weg, unter .NET eine Dai auszuführen, ist eine statische Methoie der Process-Klasse. Diese übernimmt das unhandliche Überprüfen der Datei-Erweiterung und findet ein passendes Pro;ramm für das Abspielen einer bestimmen Datei. Übergibt man dem Kommando beispielsweise eine .doc-Datei, so wird diese an eine Kopie von Word weitergeleitet - notfalls wird Word dazu gestartet. Bei Multimedia-Dateien ist das nicht anders: Ein einfacher Aufruf reicht - und der Media-Player legt los:

```
Process.Start( path);
```

Gefunden in der Toolbox:

Hilfreiche Komponenten

Die C# Toolbox des Visual Studio enthält nicht nur die Kontrollelemente für Windows Forms, sondern auch noch einen Satz an Komponenten. Die befinden sich im Toolbox-Reiter *Components* und liefern interessante Dienste.

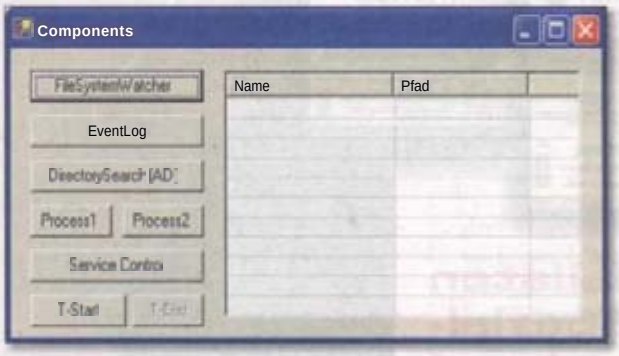
THOMAS WÖLFER

Wie Sie bereits an anderer Stelle in diesem Sonderheft erfahren haben, wird bei .NET zwischen Komponenten (Components) und Kontrollelementen (Controls) unterschieden. Der wesentliche Unterschied ist, dass eine Komponente kein eigenes Userinterface besitzt und daher nicht mit Hilfe von Designern platziert werden kann. Trotzdem können Komponenten natürlich in Windows-Forms-Programmen benutzt werden.

Dazu zieht man die Komponente genau wie ein Control aus der Toolbox auf die Form. Der Forms Designer legt dann ein neue Variable vom passenden Typ innerhalb der Form an und zeigt die Komponente symbolisch unterhalb der Form an. Klickt man dann auf die Komponente, kann man deren Eigenschaften ganz normal in der Eigenschafts-Ansicht einstellen.

Komponenten können Ereignisse auslösen, die von anderen Objekten behandelt werden können. So wäre beispielsweise eine 'Wecker'-Komponente denkbar, die zu einem fest eingestellten Zeitpunkt ein Ereignis auslöst, das dann von einer anderen Klasse behandelt werden könnte. Diese andere Klasse würde, beispielsweise ein Soundfile abspielen oder eine entsprechende Nachricht am Bildschirm anzeigen.

Von Haus aus enthält die Visual Studio Toolbox eine Handvoll fertiger Komponenten, die hautpsächlich der Interaktion mit dem lokalen System dienen. Man kann zum Beispiel Einträge im EventLog vornehmen, Teile des Dateisystems beobachten oder mit dem ActiveDirectory interagieren. Letzteres natürlich nur dann, wenn man sich innerhalb einer ActiveDirectory Domain befindet. Auch für die Prozess-, Services- und Timer-Steuerung sind Komponenten vorhanden. Was diese im einzelnen tun und wie Sie diese einsetzen können, erfahren Sie ausführlich in diesem Beitrag.



DAS BEISPIELPROGRAMM vereint viele der Komponenten in einem übersichtlicher Test-Beispiel

• Der FileSystemWatcher

Die FileSystemWatcher-Komponente kann verwendet werden, um auf Veränderungen in bestimmten Verzeichnissen zu 'horchen'. Dabei können sowohl Änderungen in den Ordnern selbst als auch Veränderungen in Unterordnern betrachtet werden. Die Komponente ist dabei nicht nur auf den lokalen Computer beschränkt, sondern kann auch Veränderungen auf entfernten Rechner im LAN beobachten.

Der FileSystemWatcher verwendet dazu Teile des Win32-Subsystems, die nur unter NT, Windows 2000 und XP verfügbar sind. Daraus resultiert, dass diese Komponente auch nur unter diesen Systemen verwendet werden kann.

Wenn Sie ein ganzes Verzeichnis betrachten wollen, dann verwenden Sie als Filter *.* - ansonsten können Sie auch einen Dateinamen angeben. Wird eine neue Datei in einem betrachteten Verzeichnis angelegt, wird eine Datei verändert oder gelöscht - dann kann die Komponente ein Ereignis auslösen, das von einer anderen Klasse behandelt werden kann.

Im Beispielprogramm werden die vier wichtigsten Ereignisse behandelt: das Verändern einer Datei, das Löschen einer Datei, das Umbenennen einer Datei und das Anlegen einer neuen Datei.

Konkret hat man sich das so vorzustellen, als würde jemand ununterbrochen auf ein Verzeichnis starren und jedes Mal laut schreien, wenn sich eine Veränderung ergibt. Das ist praktisch, falls man zum Beispiel bestimmte Datendateien unter Beobachtung halten möchte oder wenn man den Verlauf einer Download-Sitzung unter Beobachtung halten möchte. Natürlich kann man mit dieser Funktionalität auch überwachen, dass ein bestimmtes Verzeichnis nicht überläuft.

Im Beispielprogramm können Sie zunächst ein Verzeichnis zur Beobachtung auswählen. Dabei wird eine normale File-Open Dialogbox verwendet, so dass Sie in der Praxis eine Datei aus

dem gewünschten Verzeichnis auswählen müssen. Anhand der ausgewählten Datei wird dann der Verzeichnispfad ermittelt. Dazu wird das sehr handliche FileInfo()-Objekt verwendet.

Danach wird der FileSystemWatcher angeworfen, und es wird auf die von ihm erzeugten Ereignisse gewartet. Dazu gibt es in der Form einfach passende Methoden, die auf diese Ereignisse durch die Anzeige einer Dialogbox hinweisen.

LISTING 1:

```
private void button1_Click(object sender, System.EventArgs e)
{
  if( this.openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    FileInfo fi = new FileInfo( this.openFileDialog1.FileName);

    this.fileSystemWatcher1.EnableRaisingEvents=true;
    this.fileSystemWatcher1.Path = fi.DirectoryName;
  }
}

private void OnFileChanged(object sender, System.IO.FileSystemEventArgs e)
{
  MessageBox.Show( "Datei wurde verändert:" + e.Name ) ;
}
```

Bei dem Event-Handler kommt dabei ein Objekt vom Typ FileSystemEventArgs an, das alle relevanten Informationen über die Änderung im Dateisystem beinhaltet. Dazu gehört zum Beispiel auch der Name der betroffenen Datei.

Um die Komponente mit dem Beispielprogramm zu testen, klicken Sie zunächst auf den passenden Button und wählen dann eine Datei in dem Verzeichnis aus, das Sie untersuchen wollen. Das aktiviert den FileSystemWatcher für dieses Verzeichnis.

Danach gehen Sie einfach in den passenden Ordner und legen zum Beispiel eine neue Datei an oder löschen eine vorhandene: Daraufhin wird die entsprechende Meldung vom Testprogramm angezeigt.

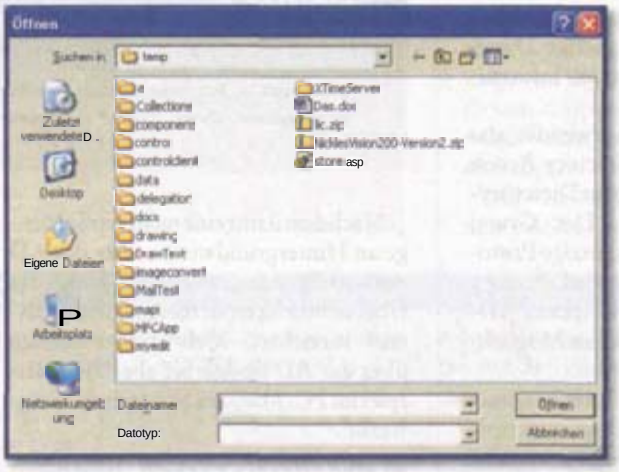
• EventLog: Protokollieren und betrachten

Win32 hat schon seit Windows NT 3.1 einen Teil, der sich EventLog nennt. Dabei handelt es sich um einem systemweiten Speicherbereich auf der Festplatte, in der das System, Dienste oder Anwendungsprogramme Informationen mitprotokollieren kann. Stürzt beispielsweise unter Windows 2000 ein Dienst ab, dann verursacht das einen entsprechenden Eintrag im EventLog.

Das EventLog steht auch unter Windows XP zur Verfügung. Von Hand können Sie sich die Einträge im EventLog mit dem Ereignisbetrachter aus den administrativen Werkzeugen anzeigen lassen.

Die EventLog-Komponente ermöglicht es Ihnen, auf Systemen, die einen EventLog unterstützen (ältere Windows-Versionen wie Windows 98 tun das nicht), selber Einträge darin vorzunehmen. Außerdem können Sie, ähnlich wie mit dem FileSystemWatcher, EventLogs beobachten. Wird ein neuer Eintrag im betrachteten Log vorgenommen, so erzeugt die Komponente ein Ereignis, das von interessierten Objekten behandelt werden kann.

Beim Schreiben m ein EventLog muss angegeben werden, welches EventLog verwendet werden soll. Existiert dieses EventLog nicht, so erzeugt die Komponente ein 'Custom' Log und verwendet dieses zum Protokollieren. Ihre Anwendung wird dabei als Quelle für Events in diesem Log registriert.



FÜR DEN TEST des FileSystemWatchers müssen Sie eine Datei im zu untersuchenden Verzeichnis auswählen.

Mit der Komponente können Sie auch aus einem EventLog Einträge auslesen. Dazu müssen Sie das gewünschte Log sowie den Namen des Rechners angeben, auf dem sich das Log befindet. Um den lokalen Rechner zu verwenden, können Sie als Namen auch einfach einen Punkt [.] angeben.

Beim Schreiben in ein EventLog können Sie zusätzlich zur eigentlich protokollierten Information verschiedene zusätzliche Angaben machen. Dazu gehört die Kategorie des protokollierten Ereignisses, der Typ des Ereignisses und weitere. Als Kategorie können Sie entweder aus einer Liste der vorhandenen und vordefinierten Kategegorien auswählen oder Sie legen eine eigenen Kategorie an. Als Typ des Ereignisses können Sie angeben, ob es sich um einen Fehler, eine Warnung, eine Information oder einen anderen der Typen aus der zugehörenden Enumeration handelt.

Sofern Sie die Rechte dazu haben, können Sie obendrein die statischen Member der EventLog-Komponente verwenden, um vorhandene Logs zu löschen oder neue anzulegen.

Sie können auch nach Quellen für EventLog-Einträge suchen, um sicherzustellen, dass Sie beim Vergeben des Names für die eigene Quelle keinen bereits vorhandenen Namen verwenden. Das ist wichtig, da alle Namen von Quellen für Ereignisse auf einem Rechner eindeutig sein müssen.

Beim Protokollieren von Ereignissen sollten Sie sicherstellen, dass Sie nicht zu viel Information protokollieren. Der Mechanismus ist dafür gedacht, selten auftretende Ereignisse - also zum Beispiel Fehler - zu protokollieren: Anwendungs-Traces sind an anderer Stelle besser aufgehoben und sollten nicht im EventLog landen.

Im Beispielprogramm wird zunächst ein Event-Handler definiert, der sich darum kümmert, dass ein neuer Eintrag im 'Application' EventLog vorgenommen wurde. Dann wird die EventLog-Komponente verwendet, um einen solchen Eintrag anzulegen. Das Ergebnis ist eine entsprechende Nachricht am Bildschirm, die angezeigt wird, wenn sie den EventLog-Button des Testprogramms betätigen.

LISTING 2:

```
private void OnEntry(object sender, System.Diagnostics.
➥EntryWrittenEventArgs e)
{
  MessageBox.Show( "Neuer EventLog-Eintrag:" + e.Entry.ToString() ) ;
}

private void button2_Click(object sender, System.EventArgs e)
{
  this.eventLog1.WriteEntry( "Ein neuer Eintrag",
  ➥EventLogEntryType.Information ) ;
}
```

Der EventHandler erhält einen `EntryWrittenEventArgs` als Parameter. Dieses Objekt enthält unter anderem die protokollierte Nachricht, der Kategorie und alle anderen Informationen über das Ereignis. Sowohl der Typ *EntryWrittenEventArgs* als auch der Typ *EventLogEntryType* stammen aus dem Namespace *System.Diagnostics*. Wenn Sie sich also Schreibarbeit sparen wollen, dann verwenden Sie am Anfang des Programms ein

```
using System.Diagnostics;
```

Statement, um nicht immer den voll qualifizierten Namen der Klasse angeben zu müssen.

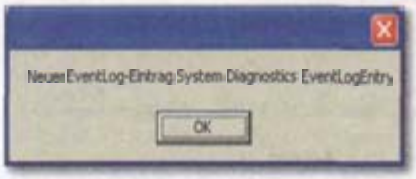
• Arbeiten mit dem ActiveDirectory

Wenn sich Ihr Rechner in einer Windows 2000 (oder XP Server) ActiveDirectory-Domäne befindet, dann ist der DirectorySearcher eine sehr hilfreiche Komponente. Diese ermöglicht Ihnen das Durchsuchen eines Active Directory nach zu spezifizierenden Informationen.

Die Komponenten verwendet das LDAP (Leightweight Directory Access Protocol), um auf eine ActiveDirectory-Hierarchie zuzugreifen. Der Grund dafür ist, dass LDAP das einzige Protokoll ist, mit dem ein ActiveDirectory durchsucht werden kann: Andere ADSI Provider stellen dazu keine Möglichkeit zur Verfügung.

Zusätzlich zur Suche im ActiveDirectory kann ein Administrator auch neue Einträge im ActiveDirectory vornehmen oder vorhandene löschen oder bearbeiten. Wer sich mit dem ActiveDirectoy auskennt, der wird viele der zusätzlichen Parameter des DirectorySearchers sehr hilfreich finden. Eine ausführliche Übersicht darüber erhält man in der MSDN Dokumentation zum *System.DirectoryServices Namespace*.

Das Beispielprogramm geht nicht sonderlich weit auf den DirectorySearcher ein: Statt eine gezielte Suche durchzuführen, wird einfach das komplette AD ausgelesen und alle Einträge werden mit ihrem Namen und ihrem Pfad im AD in einer Listview angezeigt. Das mag für eine allgemeine Übersicht hilfreich ein - im Normalfall wird man aber wohl eher eine gezielte Suche nach bestimmten Usern, Rechnern, Druckern oder Diensten durchführen wollen.



DIE EVENTLOG-KOMPONENTE meldet sich mit einem entsprechenden Hinweis, wenn ein neuer Eintrag zum EventLog hinzugefügt wird.

Nachdem dafür eine recht große Menge an Hintergrundwissen über das AD notwendig wäre, wurde im Sinne der Übersichtlichkeit in diesem Beitrag darauf verzichtet. Mehr Informationen über das AD finden Sie aber zum Beispiel im PC-Magazin Sonderheft 'Netzwerke'.

• Prozesse starten, beenden, kontrollieren

Die Process-Komponente gibt Ihnen Zugriff auf lokal laufenden Prozesse. Im einfachsten Fall ist unter einem Prozess eine laufende Anwendung zu verstehen, aber auch Windows Services sind Prozesse - genau wie der 'Idle'-Prozess des Systems. Mit der Process Komponente können Sie neue Prozesse starten und vorhandene anhalten oder kontrollieren. Dazu bietet die Process-Komponente unter anderen eine Methode, mit der Sie alle laufenden Prozesse auf dem lokalen System ermitteln können. Für jeden Prozess können Sie dann die relevanten Eigenschaften erfragen.

Dazu gehört zum Beispiel die Working-Set-Größe, die Menge an virtuellem Speicher, die dem Prozess zugesprochen wurde, oder auch dessen Priorität.

Einige diese Eigenschaften können Sie auch mit Hilfe der Process-Komponente zur Laufzeit verändern. Dazu gehört zum Beispiel die Prioritätsklasse der Komponente.

Interessanterweise sind Informationen über Prozesse teilweise persistent - sie überleben auch das Ende des Prozesses. Daher können Sie zum Beispiel den Exit-Code eines Prozesses auch nach seinem Ende erfragen. Das ist auch notwendig - denn vor dem Ende des Prozesses wäre es ein wenig schwierig, den Exit-Code zu ermitteln, denn dann hat der Prozess ja noch gar keinen Exit-Code liefern können.

Es gibt hier zwei Beispiele für die Verwendung der Process-Komponente: ein einfaches Beispiel, das eine atomare Information über den aktuellen Prozess anzeigt: die Größe des zugehörigen virtuellen Speichers.

LISTING 3:

```
private void button4_Click(object sender, System.EventArgs e)
{
  process1 = Process.GetCurrentProcess();
  MessageBox.Show( "VM: "+ process1.VirtualMemorySize);
}
```

Das zweite - etwas komplexere - Beispiel steht in Form einer weiteren Dialogbox zur Verfügung. Diese Dialogbox zeigt alle momentan laufenden Prozesse des Systems in einer Listbox an. Dabei wird einfach über alle Prozesse iteriert, und die einzelnen Prozess-Objek-

LISTING 4:

```
private void Refresh2()
{
  this.listBox1.Items.Clear();

  Process[] ps = Process.
  ➥GetProcesses();
  foreach( Process p in ps)
  {
    this.listBox1.Items.Add( p);
  }
}
```

te werden dabei in die Listbox gestellt. Diese zeigt daraufhin die String-Repräsentation der Prozesse an (Listing 4)

Wird ein Prozess aus dieser Liste ausgewählt, so werden alle seine Eigenschaften in einem Property-Grid angezeigt. Dort können Sie die Eigenschaften des Prozesses, die verändert werden können, auch zur Laufzeit bearbeiten. Dank der leichten Wiederverwendbarkeit des PropertyGrids macht das keinen großen Aufwand: Der Prozess wird einfach als 'SelectedObject' zugewiesen und das Grid übernimmt den Rest der Arbeit (Listing 5).

Ferner können Sie mit dem 'Kill'-Button auch den in der Listbox ausgewählten Prozess beenden. Das geschieht genau so wie der Name des Buttons vermuten lässt: Der Prozess wird einfach abrupt - also unsauber - aus dem Speicher geworden. Dazu kann man die Kill()-Funktion der Process-Komponente verwenden (Listing 6).

Nach dem Beenden eines Prozesses mit *Kill()* stimmt die Liste der in der Listbox angezeigten Prozesse natürlich nicht mehr. Aus diesem Grund gibt es noch einen Refresh-Button, mit dem Sie die Liste erneuern können. Dabei wird die gleiche Funktion aufgerufen, die schon beim Öffnen der Form zum Auffüllen der Liste verwendet wurde (Listing 7).

An dieser Stelle wäre eine Komponente, die erst etwas später im Beitrag erwähnt wird, hilfreich gewesen: Die Timer-Komponente. Damit wäre es nämlich leicht möglich gewesen, die 'Refresh()'-Funktion einfach regelmäßig aufzurufen, und man hätte sich den Refresh-Button sparen können: Die Liste wäre dann immer automatisch auf dem neuesten Stand gewesen.

• Der Service Controller: Dienste kontrollieren

Mit der Service Controller-Komponente können Sie Dienste kontrollieren – so, wie es der Name nahe legt. Dazu muss man aber zunächst einmal wissen, was ein Dienst überhaupt ist.

Ein Windows Service ist ein Prozess, der sich deutlich von normalen Windows-Prozessen unterscheidet. Dienste können beim Systemstart gestartet werden und können somit auch dann laufen, wenn kein Anwender am System angemeldet ist. Dienste sind darauf ausgelegt, vom Start des Rechners bis zum Ausschalten des Rechners betrieben zu werden - und sie haben kein eigenes Userinterface.

Um mit Diensten zu kommunizieren, verwenden Sie im Normalfall den Service Control Manager oder die 'Dienste'-Anwendung aus den administrativen Werkzeugen. Damit kann man Dienste starten, anhalten oder fortsetzen, und außerdem ist es damit auch möglich, bestimmte Befehle an einen oder mehrere Dienste zu senden.

Der Mail-Transport mit dem SMTP bei Windows 2000 und XP Pro ist zum Beispiel als Dienst implementiert, genau wie der Internet InformationServer und andere recht grundlegende Funktionen des Betriebssystems.

Die Dienste waren vom Programmierer immer ein wenig schwierig zu erreichen - das änderte sich erst, als sie im Zuge von WMI erreichbar wurden: Seit diesem Zeitpunkt war es sogar Skripten relativ leicht möglich, einen Dienst zu erreichen.

Mit der ServiceController-Komponente sind Dienste nun auch in norma-

LISTING 5:

```
private void OnSelChanged(object sender, System.EventArgs e)
{
  Process p = this.listBox1.SelectedItem as Process;
  this.propertyGrid1.SelectedObject = p;
}
```

LISTING 6:

```
private void button1_Click(object sender, System.EventArgs e)
{
  // DIE! Process - DIE!
  Process p = this.listBox1.SelectedItem as Process;
  p.Kill();
}
```

LISTING 7:

```
private void button2_Click(object sender, System.EventArgs e)
{
  Refresh2();
}
```

NAMESPACE - WHERE ARE YOU?

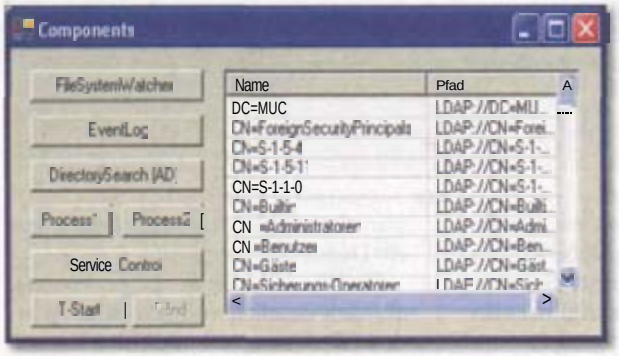
Alle Klassen der .NET-Klassenbibliothek sind in Namespaces unterteilt. Die elementaren Klassen, wie zum Beispiel der 'String', befinden sich dabei im Namespace 'system'. Andere Klassen sind in Namespaces innerhalb des 'System'-Namespaces untergebracht - wieder andere befinden sich im Namespace 'Microsoft'.
So finden Sie zum Beispiel die Process-Klasse im Namespace 'System.Diagnostics', die ArrayList-Klasse stammen ebenfalls aus System, ebenso Collections und die Klassen für Datei I/O.
Nun ist es aber sehr hinderlich, immer wieder den komplett qualifizierten Namen einer Klasse hinschreiben zu müssen, wie das im folgenden Ausdruck getan wird:

```
System.Diagnostics.Process p =
➥new
➥System.Diagnostics.Process();
```

Zu diesem Zweck kann man C# dazu bringen, die Namespaces etwas weniger eng zu sehen und bei einige Klassen auf die vollständige Angabe der voll qualifizierten Namen zu verzichten. Dazu dient das 'using'-Statement. Damit teilen Sie dem C#-Compiler mit, welche Namespace(s) Sie verwenden wollen - und bei allen Klassen aus diesen Namespaces können Sie dann auf die voll qualifizierten Namen verzichten. Aus dem voranstehenden Beispiel wird dann also

```
using System.Diagnostics;
Process p = new Process();
```

Das ist schon deutlich angenehmer hinzuschreiben und auch viel leichter zu lesen - und geht dankbarerweise auch mit mehreren Namespaces gleichzeitig.
Alles, was Sie dazu tun müssen, ist mehrere Using-Statements einzufügen: Jeder Namespace, den Sie verwenden wollen, erhält dazu einen eigenen.



DER DIRECTORYSEARCHER listet einfach alle Eintrage aus dem lokalen ActiveDirectory in einer ListView auf.

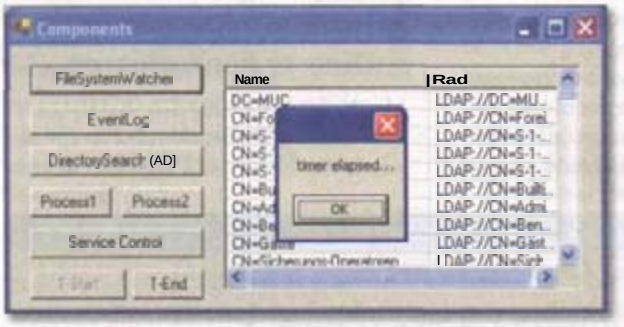
len .NET-Programmen leicht zu erreichen. (Das WMI Interface ist prima für Skripte geeignet, in C/C++ oder C+ Programmen führt es aber zu recht unhandlichem Code da praktisch die komplette Kommunikation über konstante Strings stattfindet.)

Die ServiceController Komponente erlaubt das Starten und Stoppen von Diensten und bietet auch Informationen über diese. So kann man beispielsweise erfragen, ob ein Dienst von anderen Diensten abhängig ist, ob ein Dienst momentan beendet werden kann und dergleichen.

Dabei können die Komponenten sowohl mit Diensten, die auf dem lokalen Rechner laufen, kommunizieren, aber auch mit Diensten, die auf einem entfernten Rechner im Netz betrieben werden.

Das Beispielprogramm überprüft, ob der Alert-Dienst in einem Zustand ist, der es erlaubt, diesen Dienst zu beenden - und wenn das der Fall, beendet es diesen Dienst.

Mit der ServiceController-Komponente können Sie immer nur einen Dienst gleichzeitig steuern. Welcher das ist, legen Sie am einfachsten in der Eigenschafts-Ansicht der Komponente im Visual Studio fest. Sie können diese Eigenschaft aber natürlich auch zur Laufzeit des Programms festlegen, wie Sie in Listing 7 sehen können.



DER TIMER MELDET SICH alle fünf Sekunden mit einer entsprechenden Dialogbox. Um das abzuschalten brauchen Sie den T-End Button im Beispielprogramm.

• Wem die Stunde schlägt: die Time-Komponente

Als letzte der besprochenen Komponenten aus der Toolbox geht es im restlichen Teil dieses Beitrags um die Time-Komponente. Die Time-Komponente kapselt Windows-Timer. Bei diesen Timern handelt es sich, wie der Name bereits vermuten lässt, um zeitgebende Einheiten.

Unter Win32 erzeugen Sie einen Timer mit *CreateTimer()* und identifizieren diesen durch einen eindeutigen Identifizierer. Den können Sie sich bei .NET sparen: Timer sind einfach Objekte, die instanziert und dann aktiviert werden können.

Dabei hat ein Timer eine Zeitspanne in Millisekunden als wichtigste Eigenschaft. Wird der Timer aktiviert und läuft diese Zeitspanne ab, dann löst der Timer ein Ereignis aus, und das kann von einer interessierten Klasse behandelt werden. Im Wesentlichen bedeutet das, dass Sie in der Lage sind, immer wieder kehrende Operationen ganz einfach zeitbasiert zu steuern: Sie werfen einen Timer an und warten dann darauf, dass die zuvor spezifizierte Methode aufgerufen wird.

Im Beispielprogramm passiert das mit zwei Buttons: Der eine aktiviert den Timer und disabled dann sich selbst. Außerdem wird der zweite Button aktiviert. Der Timer selbst meldet sich aufgrund der vorgenommenen Einstellung alle fünf Sekunden. Will man diese Meldungen nicht länger sehen, kann man den Timer mit dem zweiten Button beenden. Der disabled dann sich selbst und schaltet den ersten Button wieder an: Damit kann der Vorhang wieder von vorne begonnen werden (Listing 8).

In diesem Beitrag haben Sie erfahren, wie Sie die zur Verfügung stehenden Komponenten aus der Visual Studio Toolbox richtig einsetzen - und was Sie damit alles anstellen können. Die vermutlich wichtigste Komponente ist dabei auch die einfachste: Der Timer. Der funktioniert nämlich, im Gegensatz zu vielen anderen Komponenten, auf Plattformen und bietet einen Dienst, den man in praktisch jedem Programm gut gebrauchen kann. © UR

LISTING 8:

```
private void button5_Click(object sender, System.EventArgs e)
{
  if( this.serviceController1.CanStop)
  {
    this.serviceController1.Stop();
    MessageBox.Show("Alerter-Service angehalten");
  }
}
```

LISTING 9:

```
private void button6_Click(object sender, System.EventArgs e)
{
  this.button6.Enabled = false;
  this.button7.Enabled = true;
  this.timer1.Enabled = true;
}private void OnTimerElapsed(object sender, System.Timers.
➥ElapsedEventArgs e)
{
  MessageBox.Show("timer elapsed...");
}

private void button7_Click(object sender, System.EventArgs e)
{
  this.button6.Enabled = true;
  this.button7.Enabled = false;
  this.timer1.Enabled = false;
}
```

Fenster mal anders

Individuelle Form

Fenster sind eckig, und Kontrollelemente sind das auch - das stimmt nicht ganz. Bei .NET ist es sogar extrem einfach, Fenster mit unregelmäßigen Formen zu erzeugen

THOMAS WÖLFER

Für Forms gibt es in .NET zwei Möglichkeiten, dem Rechteck zu entfliehen. Und sogar Kontrols haben eine Möglichkeit, rund auszusehen. Nach dem ewigen Kampf der Programmierer mit durchsichtigen und nicht rechteckigen Fenstern in älteren Windows-Versionen - beides war zwar im Prinzip möglich, aber nur mit sehr viel Aufwand - hat sich Microsoft bei .NET endlich erbarmt und macht die Gestaltung interessanterer Fenster leicht.

Ganz besonders einfach ist die Sache mit Fenstern geworden, die keine eckige Form haben sollen. Hier ist jede beliebige Form ganz ohne eine einzige Zeile Quellcode möglich. Es braucht dann allerdings ein bisschen unterstützenden Code, wenn das Fenster verschoben werden soll.

Es ist sogar ohne Mehraufwand möglich ein Fenster zu definieren, das sich aus mehreren nicht zusammenhängen Teilen zusammensetzt. Der Trick besteht darin, dass man bei einem Fenster einen 'Transparency Key' angeben kann. Dabei handelt es sich um einen Farbwert. Bei der Anzeige der Form werden dann alle Bereiche der Form, die diese Farbe haben, durchsichtig dargestellt. Man kann sich das als eine Art Filter vorstellen: Vor der Anzeige eines Fensters werden alle Bereiche, die eine vorgegebene Farbe darstellen, einfach ausgefiltert. Dadurch bleibt der unter diesen Bereichen befindliche Teil des Windows-Desktops oder der darauf befindlichen Fenster sichtbar.

• Hintergrundbild wird Farbfilter

Dabei stellt sich nur die Frage, wie man die Farbe 'richtig' auf dem Fenster verteilt. Das ist aber extrem einfach, denn eine Windows-Form hat eine spezielle Eigenschaft, die in diesem Zusammenhang sehr hilfreich ist: das Hintergrundbild. Ist ein Hintergrundbild gesetzt, so stellt dieses den Farbfilter für das unsichtbar Machen der Form dar.

Sie müssen also zunächst erst einmal eine Bitmap erzeugen. Die Bitmap besteht dabei im einfachsten Fall aus zwei Farben. Die eine Farbe füllt genau die Teile aus, die das Aussehen der Form bestimmen sollen. Die andere Farbe füllt die Flächen, die quasi aus der Form herausgeschnitten werden sollen.

ZUNÄCHST BRAUCHEN SIE eine Bitmap, die die gewünschte Form enthält Dabei merken Sie sich die Farbe, die ausgestanzt werden soll.

Wollen Sie also beispielsweise eine runde Form definieren, dann erzeugen Sie eine Bitmap, die einen gefüllten Kreis enthält. Wenn Sie eine Form wünschen, die sich aus mehreren nicht zusammenhängen Teilen zusammensetzt, dann füllen Sie einfach auch noch andere Flächen innerhalb der Bitmap in der Farbe des Kreises aus.

Sie können aber auch mehr tun: Nachdem zur Laufzeit des Programms nur der Bereich sichtbar ist, der die 'nicht' unsichtbare Farbe enthält, ist diese Farbe, bzw. der Inhalt dieser Flächen, effektiv der Hintergrund ihrer Form. Sie können hier also auch Texturen oder Farbverläufe unterbringen - die Form hat dann zur Laufzeit eben einen Farbverlauf als Hintergrund. Genau das wurde auch im Beispielprogramm getan.

Das Bild können Sie mit einem beliebigen Zeichenprogramm erzeugen, das in der Lage ist, Windows Bitmaps abzuspeichern. Also zum Beispiel mit dem Bitmap-Editor von Visual Studio, mit Windows Paint oder auch mit einem etwas aufwändigeren Bildbearbeitungsprogramm wie PaintShop Pro. Letzteres wurde für das Bitmap aus dem Beispielprogramm verwendet, da die Erzeugung von Farbverläufen in diesem Programm besonders einfach ist.

DEN BEREICH, der nicht ausgeschnitten werden soll, können Sie beliebig anders färben: So sieht dann hinterher Ihre Form aus.

Dann legen Sie ein neues C# Projekt vom Typ 'Windows Application' an. Die automatisch mit erzeugte Form ist für's Beispiel völlig ausreichend: Aus dieser Form wird dann mit Hilfe der zuvor erzeugten Bitmap der nicht erwünschte Teil herausgeschnitten. Sie brauchen die Bitmap übrigens nicht extra zum Projekt hinzuzufügen - das geschieht automa-

tisch, sobald Sie die Bitmap als Hintergrundbild spezifizieren.

Genau das passiert im nächsten Schritt. Sie gehen in die Eigenschafts-Ansicht der Form und suchen dort die Eigenschaft 'Background-Image' (Hintergrundbild.)

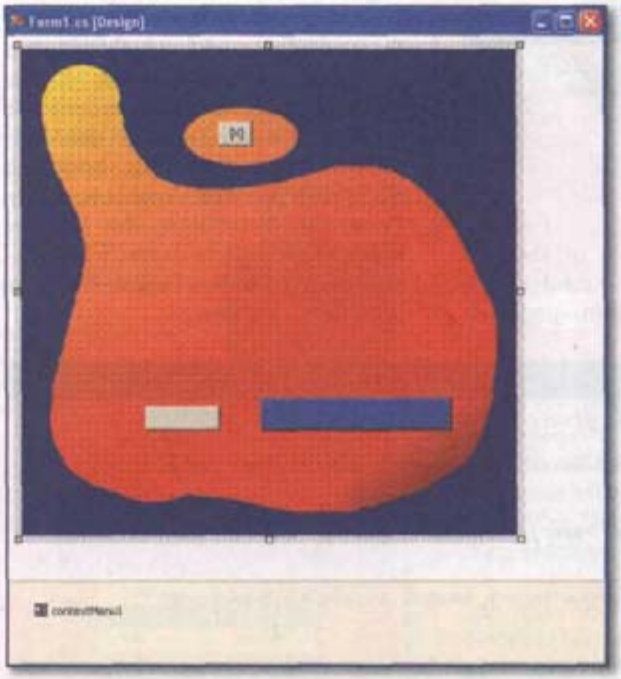
Dort finden Sie eine Möglichkeit, das Bild anzugeben. Die Form wird danach sofort mit dem ausgewählten Bild als Hintergrund angezeigt.

Nun müssen Sie den Transparency-Key für die Form setzen. Auch das geschieht in der Eigenschafts-Ansicht. Sie können dort eine Farbe auswählen - das bedeutet, dass Sie wissen müssen, welche Farbe die Flächen haben, die ausgeschnitten werden sollen. Daher ist es am einfachsten, wenn Sie für die ursprüngliche Bitmap nur eine kleine Palette verwenden - das macht es leichter, diese Farbe zu identifizieren.

Ist der Key für die Transparenz gesetzt, dann ist die Form nahezu fertig. Allerdings fehlt noch ein wichtiges Detail - denn die Form hat noch immer eine Rahmen und eine Titelleiste. Das führt aber zur Laufzeit des Programms zu einem nicht gerade wünschenswerten Effekt:

Wenn Sie das Projekt nämlich jetzt so übersetzen und starten, stellen Sie fest, dass die gewünschten Bereiche des Fensters zwar durchsichtig sind, nur sind eben der Rahmen und die Titelleiste noch immer sichtbar.

Dieses kleine Problem ist schnell behoben: Sie müssen nur die Eigenschaft 'BorderStyle', mit der die Art des Fensterrahmens festgelegt werden kann, auf *none* stellen.



IST DIE BITMAP als Hintergrundbild gesetzt, so ist das auch sofort im Forms-Designer zu erkennen.

Dann hat das Fenster keine Rahmen und keine Titelleiste mehr - und der Effekt ist: Die störenden Elemente sind verschwunden.

Wenn Sie das Programm nun erneut starten, dann stellen Sie schnell ein neues Problem fest: Das Programmfenster hat nun keine Titelzeile mehr und kann deswegen auch nicht mehr verschoben oder per Menüauswahl geschlossen werden. Der einzige Weg das Fenster zu schließen besteht darin, dass man die Tastenkombination [Alt+F4] betätigt. Verschieben kann man das Fenster gar nicht.

• Fenster verschieben

Dieses Probleme sind aber nicht mehr allein im Forms Designer zu klären - es muss Quellcode her, der sich der Probleme annimmt.

Zunächst wird sich aber erst einmal um das konventionelle Schließen des Fensters gekümmert. Das ist allerdings trivial: Ziehen Sie einen Button aus der Toolbox auf den 'sichtbaren' Bereich der Form und legen Sie dafür einen Event-Handler für das Click-Ereignis an.

EIN OBJEKTMENÜ

Nachdem das im Beispielprogramm implementierte Fenster keine Titel leiste hat, bietet es auch keinen Platz für ein Menü. Das bedeutet aber nicht, dass Sie kein Menü auf Ihrem Fenster unterbringen können. Start eines normalen Menüs können Sie ja einfach ein Objekt-Menü definieren.

Diese Menüs werden angezeigt, wenn mit der rechten Maustaste auf das Fenster geklickt wird – und dieser Vorgang ist schon von vorneherein im Forms-Designer vorgesehen. Um also ein Objekt-Menü für den rechten Mausklick zu definieren, muss man sich nicht sonderlich viel Arbeit machen.

Als ersten Schritt zieht man ein Objekt-Menü aus der Toolbox auf die Form. Das Menü erscheint dann am unteren Rand des Form-Designers.

Wenn Sie darauf klicken, dann öffnet sich der normale Menü-Editor oberhalb der Form. Dort können Sie das Menü einfach dadurch definieren, dass Sie die einzelnen Befehle, die angezeigt werden sollen, direkt eintippen.

Alles, was Sie dann noch brauchen, sind die Handler für die einzelnen Menü-Befehte. Die legen Sie einfach durch einen Doppelklick auf den Menüeintrag an: Der benötigte Code wird vom Visual Studio erzeugt, und Sie brauchen die entsprechende Funktion nur noch aufzufüllen.

Damit das Menü bei einem rechten Mausklick auch tatsächlich angezeigt wird, müssen Sie die Eigenschaft 'ContextMenü' der Form so setzen, dass das soeben definierte Menü verwendet wird. Dazu klicken Sie in der Eigenschafts-Ansicht auf die Combo-Box-Button - die Entwicklungsumgebung bietet dann alle auf der Form verfügbaren Objekt-Menüs zur Auswahl an: also eines.

Danach müssen Sie das Projekt nochmals übersetzen und können das Fenster dann zusätzlich über ein Menü steuern, das aufklappt, wenn Sie mit der rechten Taste auf das Fenster klicken.

Beim Definieren der Event-Handler für das Menü können Sie übrigens zuvor definierte Event-Handler, die die gleiche Signatur haben, wiederverwenden. So haben Sie zu Anfang des Beispielprogramms einen Button zum Schließen des Fensters auf die Form gezogen und diesen mit einem Click-Event Handler ausgestattet. Dieser Handler hat zum Beispiel die gleiche Signatur wie ein Click-Handler für's Objektmenü.

Wenn Sie also im Objektmenü einen *Schliessen*-Befehl eingebaut haben, dann können Sie als Event-Handler dafür einfach den gleichen verwenden, den Sie bereits für den Button definiert haben.

Dazu klicken Sie einmal auf den entsprechenden Eintrag im Menü und wechseln dann in die Eigenschaften-Ansicht. Dort aktivieren Sie die Ansicht für die Events und suchen den Click-Event.

Dort finden Sie auch wieder eine Combo-Box, die alle definierten Handler anzeigt Alles was Sie dann noch tun müssen, ist den Handler für den Close-Button aus der Liste auszuwählen - dieser Handler wird dann auch bei einem Click auf den Menü-Befehl ausgeführt.

Da das Click-Ereignis das Default-Ereignis für Buttons ist, reicht dazu ein Doppelklick auf den Button im Forms-Editor. Der Handler selbst sieht dann wie folgt aus:

LISTING 1:

```
private void button1_Click
➥(object sender, System.
➥EventArgs e)
{
  this.Close();
}
```

Alles, was also passiert ist, dass die Close()-Methode der Form aufgerufen wird. Das schließt die Form - und da es sich dabei auch um den Eintrittspunkt des Programms handelt, beendet diese Vorgang auch das Programm.

Das war einfach - aber auch das Verschieben den Form ist nicht wesentlich schwieriger. Dazu müssen Sie allerdings gleich zwei Ereignisse behandeln: Zum einen das MouseDown-Ereignis und zum anderen das MouseMove-Ereignis. Ersteres tritt ein, wenn die Form von einem Mausklick getroffen wird, letzteres tritt ein, wenn die die Maus bewegt wird, während sie sich über der Form befindet.

Die Idee der Sache ist, dass Sie sich die initiale Position der Maus im MouseDown Event merken. Im MouseMove Handler überprüfen Sie dann, ob die linke Maustaste gedrückt ist. Ist das der Fall, dann verwenden Sie die aktuelle Mausposition und die ursprüngliche Position der Maus, um damit die neue Position für das Fenster zu berechnen: Das Fenster kann dann also dadurch verschoben werden, dass der Anwender in irgend einen Bereich des Fensters klickt und dann die Maus bei gedrückter Maustaste bewegt.

Welche Maustaste gedrückt ist und an welcher Stelle die Maus sich momentan befindet, das sind Informationen, die der Event-Handler mit übergeben bekommt. Sie können also direkt darauf zugreifen. Die Maustasten können dabei einfach mit den möglichen Maustasten aus der Enumeration der Maustasten verglichen werden:

Die ursprüngliche Position der Maus - also die, die zum Zeitpunkt des MouseDown Events aktuell war - speichern Sie in einer privaten Variable vom Typ *Point*. Das Verschieben des Fensters können Sie einfach dadurch bewerkstelligen, das Sie dem Fenster eine neue 'Location' zuweisen.

LISTING 2:

```
private Point mouse_offset;

private void OnMouseDown(object sender, System.Windows.Forms.
➥MouseEventArgs e)
{
..mouse_offset = new Point(-e.X, -e.Y);
}

private void OnMouseMove(object sender, System.Windows.Forms.
➥MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
  {
    Point mousePos = Control.MousePosition;
    mousePos.Offset(mouse_offset.X, mouse_offset.Y);
    Location = mousePos;
  }
}
```

Damit ist der erste Teil erledigt: Sie können nun Fenster mit beliebigen Formen definieren und es ist auch sichergestellt, dass diese Fenster dennoch geschlossen und verschoben werden können. Alles zusammen geht mit gerade mal sechs Zeilen Code, die man selbst eintippen muss - nicht schlecht für eine Aufgabe, die mit älteren Windows- Versionen durchaus eher in mehreren Stunden gelöst werden musste.

• Controls: Jetzt auch in Farbe

Controls haben leider keinen Transparency Key, und daher ist es dort auch ein wenig aufwändiger, beliebige Formen zu erzeugen. Dabei ist aber 'wenig' genau die richtige Umschreibung, denn wesentlich mehr Arbeit macht die Sache bei Controls auch nicht. Übrigens ist der im Folgenden vorgestellte Weg auch für Forms möglich - das ist die eingangs erwähnte zweite Methode, Fenster mit beliebigen Formen zu erzeugen.

Für das Erzeugen von eigenen Formen für Controls muss man ein wenig Hintergrundwissen über zwei Objekte aus dem Drawing Namespace haben. Dabei handelt es sich um den *GraphicsPath* und die *Regions*.

Ein GraphicsPath ist ein Objekt, das ein Anwendungsprogramm verwenden kann, um Outlines von Formen zu zeichnen, um Formen zu füllen und um Clipping-Regionen zu spezifizieren.

Dabei kann sich ein Pfad aus einer beliebigen Anzahl von Figuren zusammensetzen. Jede Figur setzt sich dabei entweder aus einer Sequenz aus miteinander verbundenen Linien zusammen, oder aber es handelt es sich einfach nur um eine geometrische Primitive. Dabei sind auch Texte als Pfade zulässig - die Graphik-Engine von Windows berechnet dabei die Outlines der Buchstaben und erzeugt daraus Figuren.

Eine Region beschreibt das innere einer grafischen Form, die sich aus Rechtecken und GraphicsPathes zusammensetzt. Das bedeutet, dass eine Region mit einem GraphicsPath als Parameter erzeugt werden kann.

ALS STIL FÜR DEN RAND der Form geben Sie None an. Das Fenster hat dann weder einen Rand noch eine Titelleiste.

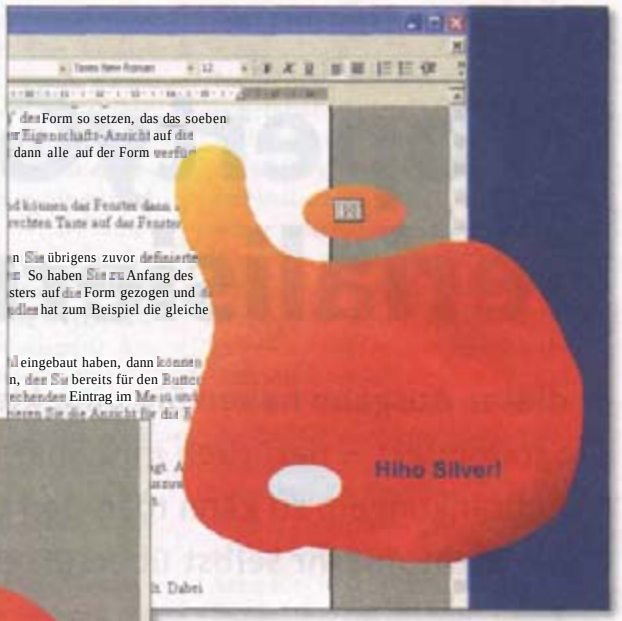
Regions sind skalierbar und können als Ausgabebereich für grafische Operationen verwendet werden. Das Interessanteste daran ist aber, dass alle Controls eine Region als Eigenschaft haben

- und diese Region beschreibt, wo und wie das Control gezeichnet wird.

Den letzten Satz haben Sie richtig gelesen: Die Region bestimmt das Aussehen des Controls. Und da die Region eine Eigenschaft ist, die zur Laufzeit zugewiesen werden kann, bedeutet das auch, dass man das Aussehen aller Controls einfach dadurch beeinflussen kann, dass man ihm ein neue Region zuweist.

Wie zuvor erwähnt, können Regions durch einen GraphicsPath erzeugt werden, und ein GraphicsPath kann sich auch aus Texten zusammensetzen. Genau das macht sich das folgende Beispiel zu Nutze: Es wird dabei ein GraphicsPath erzeugt, der sich aus einem Text zusammensetzt. Dieser Pfad wird dann als Grundlage für das Erzeugen einer Region verwendet. Und diese Region wird dann einem Button zugewiesen. Als Resultat erhält man einen Button der seine Form aus der Outline eines Textes erhält.

Um zu erreichen, dass der Button in der neuen, statt in seiner normalen Form, gemalt wird, ist nicht viel zu tun: Sie müssen nur einen Event-Handler für das Paint-Event programmieren. Dazu ziehen Sie zunächst einen Button auf den 'sichtbaren' Teil der Form und suchen dann in dessen Eigenschaften nach dem Paint-Event. Für dieses geben Sie den Namen eines Handlers an und drücken *Return*. Visual Studio erzeugt dann den passenden Code und zeigt Ihnen im Quellcode-Editor das Listing an dieser Stelle. Der Inhalt der Funktion sieht dann wie Listing 3 aus. Zunächst erzeugen Sie ein neues GraphicsPath-Objekt. Das ist der GraphicsPath, der später das Aussehen des Buttons bestimmen wird. Dann spezifizieren Sie die einzelnen Parameter für den auszugebenden Text. Das ist zunächst einmal der Text selbst, dessen Schriftart und Schnitt sowie dessen Größe. Diese Größe ist letztlich auch dafür verantwortlich, wie groß der Button hinterher wird.



FENSTER MIT LUSTIGEN FORMEN sind leichter zu haben, als man das erwarten würde. Der Text 'Hiho Silver' in der Abbildung ist übrigens ein Button.

UM DIE FORM mit einem Objektmenü auszustatten, ist ebenfalls nicht viel zu tun. Sie können das direkt im Forms-Editor erledigen.

Dann fügen Sie den Text mit der AddString()-Methode zum GraphicsPath hinzu - und das war's im Prinzip: Sie müssen nun nur noch eine neue Region erzeugen, und die erzeugen Sie mit Hilfe des GraphicsPaths.

Wenn Sie die Anwendung erneut übersetzen und starten, dann finden Sie einen Button in Form des Textes 'hiho Silver' auf Ihrem Fenster.

Um zu testen, ob der Button auch tatsächlich anklickbar ist, können Sie zusätzlich noch einen Event-Handler für den Click-Event erzeugen.

Der Button ist dabei übrigens überall innerhalb des Pfades anklickbar: Wenn Sie außerhalb der Buchstaben klicken, dann fühlt er sich nicht getroffen.

In diesem Beitrag haben Sie erfahren, wie man ein Fenster mit beliebiger Form erzeugt und wie man das gleiche auch mit Kontrollelemente machen kann. Die zugegebenermaßen künstlerisch nicht besonders wertvollen Formen warten darauf, dass Sie sie mit eigener Kreativität in hübsche Elemente umwandeln.

© UR

LISTING 3:

```
private void OnTextButtonPaint(object Sender, System.Windows.Forms.PaintEventArgs e)
 {
   System.Drawing.Drawing2D.GraphicsPath myGraphicsPath  = new
                  System.Drawing.Drawing2D.GraphicsPath();

   string stringText = "Hiho Silver!";
   FontFamily family = new FontFamily("Arial");
   int fontstyle = (int)FontStyle.Bold;
   int emSize = 24;
   PointF origin = new PointF(0,  0);
   StringFormat format = new StringFormat(StringFormat.GenericDefault);
   myGraphicsPath.AddString(stringText, family, fontstyle, emSize, origin, format);
   textbutton.Region = new Region(myGraphicsPath);
 }
```

Eine eigene IDE für C#:

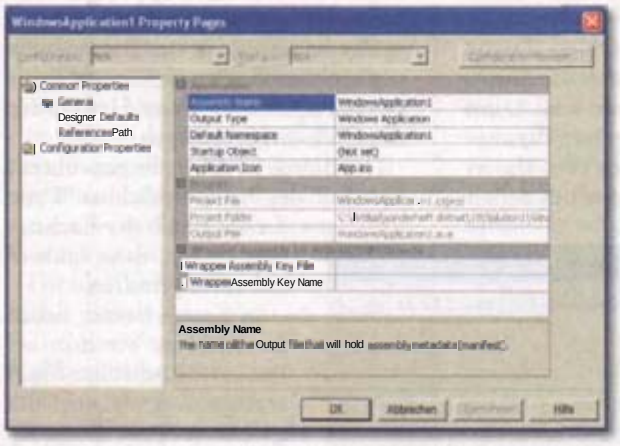
PropertyGrids und Serialisierung in .NET

In dieser Ausgabe haben Sie eine kleine Entwicklungsumgebung für C# programmiert - natürlich in C Sharp. Die leidet allerdings unter ein paar Einschränkungen, so kann man den Quellcode dieser Umgebung beispielsweise nicht mit ihr selbst übersetzen. Das ändert sich jetzt.

THOMAS WÖLFER

Die bisher zur Verfügung stehende Entwicklungsumgebung leidet unter zwei größeren Problemen, die aber eng miteinander verwandt sind. Zum einen ist es in der IDE nicht möglich, irgendwelche Compiler-Optionen anzugeben, zum anderen können Sie dem Compiler nicht mitteilen, welche Assembly-Referenzen er verwenden soll.

Ohne eine Einstellmöglichkeit für die Compiler-Optionen könnte man ja noch leben - aber ohne eine Möglichkeit, Assemblies zu spezifizieren, ist das schon schwieriger. Stellt man keine Compiler-Optionen ein, dann wird eben einfach immer mit den Compiler-Defaults kompiliert. Kann man aber keine Referenzen angeben, dann ist es unter Umständen nicht möglich, den Quellcode überhaupt zu übersetzen. Um zu verstehen, warum das so ist, muss man ein wenig über die Art und Weise wissen, wie .NET funktioniert und wozu Referenzen überhaupt gut sind.

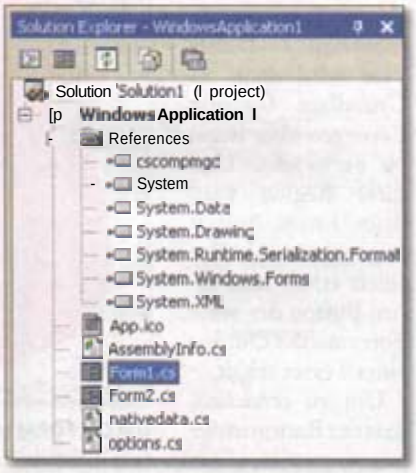


DER OPTIONS-DIALOG für den C#-Compiler in Visual Studio: So etwas in der Art braucht die eigene IDE auch.

• Referenzen-Grundlagen

Beim normalen Programmieren mit C++ kann man auch Objekte verwenden -um genau zu sein, ist das der primäre Grund, weshalb Programmierer mit C++ statt mit C arbeiten. Dabei ist es meist so, dass eine Headerdatei die Objektbeschreibung, also dessen Interface, enthält, während die Implementierung des Objekts in einer C++-Datei vorliegt. Die C++-Datei wird nun übersetzt, und das Resultat wird dann in einer Library abgelegt.

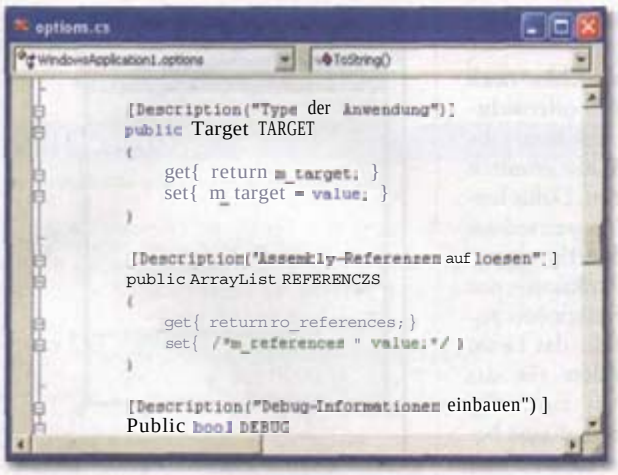
Will man das Objekt später verwenden, so wird das zum Objekt gehörende Headerfile an passender Stelle inkludiert und die Library mitgelinkt. Der Compiler kann mit Hilfe der Informationen aus dem Headerfile sicherstellen, dass das Objekt korrekt benutzt wird, und der Linker stellt durch das Hinzufügen des Object-Codes des Objekts sicher, dass der benötigte Programmcode zur Laufzeit auch vorhanden ist.



IN VISUAL STUDIO gibt es einen eigenen Editor für die Auswahl von Referenzen auf Assemblies.

Bei .NET funktioniert diese Sache aber deutlich anders. Hier sind die Informationen über ein Objekt in der gleichen Datei abgelegt wie das Objekt selbst - und zwar in einer vorkompilierten Form. Zu diesen Informationen gehören alle das Objekt beschreibenden Informationen, also zum Beispiel eine Aufzählung der unterstützten Interfaces, eine Liste der vom Objekt zur Verfügung gestellten Methoden und deren Typen und natürlich auch alle Ressourcen, die der Programmierer dem Objekt mitgeben will.

Alles zusammen findet sich dann (meist) in einer DLL wieder. Eine solche DLL kann aber auch mehr als nur ein Objekt beinhalten - der Menge der Objekte sind da keine Grenzen gesetzt. Eine solche DLL nennt man unter .NET



MIT PROPERTY-METHODEN und passenden Attributen versorgen Sie nicht nur sich selbst, sondern auch das PropertyrGrid mit hilfreichen Informationen.

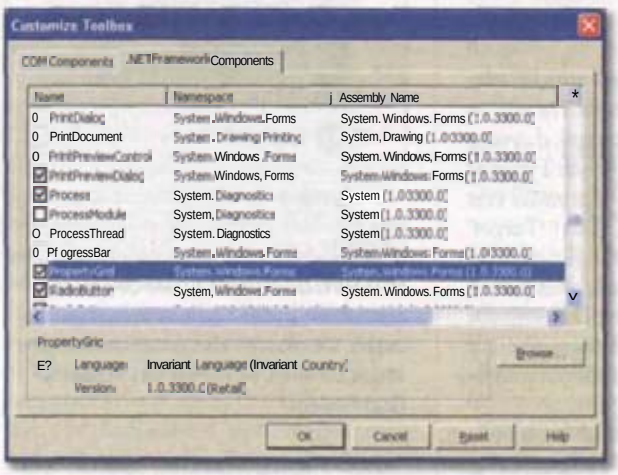
'Assembly' - also etwa '(An)Sammlung'. Dass diese Dateien die Erweiterung .DLL haben, ist dabei mehr oder minder Zufall, denn Assembly DLLs sind nun mal per Definition eine Sammlung an Code mit deutlich mehr Funktionalität als das bei normalen DLLs der Fall ist.

Um nun so ein Objekt aus einer Assembly im Quellcode eines Programms nutzen zu können, muss der Compiler, der den Quellcode des Programms übersetzt, natürlich wissen, wie das Objekt definiert ist. Es muss also bekannt sein, welche Interfaces das Objekt unterstützt, welche Methoden es hat und so weiter: Das sind aber genau die Informationen, die zusammen mit dem Code in der Assembly DLL enthalten sind. Diese Informationen nennt man Metadaten - also 'Daten über Daten'.

Der Compiler kann aber schlecht raten, in welcher Assembly ein bestimmtes Objekt enthalten ist: Er kann nur während des Übersetzens des Quellcodes bemerken, dass ein Objekte aus einem bestimmten Namespace mit einem bestimmten Namen verwendet werden soll. Ist dem Compiler das Objekt nicht bekannt, dann resultiert das in einer Fehlermeldung: Man kann den Quellcode nicht übersetzen.

Also muss es eine Methode geben, dem Compiler mitzuteilen, in welchen Assemblies er nach den verwendeten Objekten suchen soll. Genau das tun 'Referenzen' - sie geben an, welche Assemblies durch die Benutzung eines Objekts daraus referenziert werden. Wenn man so will, ist das auch nichts anderes, als die Angabe einer Link-Library beim Zusammenbauen eines C/C++-Programms durch den Linker. Der Unterschied besteht darin, dass die Assembly nicht in die ausführbare Datei mit eingebunden wird und dass das .NET-Framework beim Laden der ausführbaren Datei nochmals versucht, die benötigte(n) Assemblies auf dem aktuellen System zu laden. Dabei werden natürlich auch so interessante Dinge, wie die den einzelnen Assemblies zugestandenen Zugriffsfreiheiten und die Versionsnummern der Assemblies berücksichtigt.

Nur sind die vielen Klassen des .NET-Frameworks in einer großen Anzahl an unterschiedlichen Assemblies untergebracht. Die 'string'-Klasse liegt beispielsweise in der 'system.dll' Assembly, die 'Form'-Klasse befindet sich in 'System.Windows.Forms' etc.



BEVOR SIE DAS PROPERTYGRIO benutzen können, müssen Sie es zur Toolbox hinzufügen.

UM EINEN DIALOG zum Öffnen von Dateien verwenden zu können, ziehen Sie einfach die passenden Komponenten auf die Form.

• Referenzen verwenden

Nun könnte man dem Compiler natürlich einfach immer alle auf einem System vorhandenen Assemblies zum Untersuchen übergeben, aber das würde dummerweise dazu führen, dass das Übersetzen von Dateien deutlich länger in Anspruch nehmen würde als unbedingt notwendig. Das will man aber vermeiden, und daher ist es üblich nur diejenigen Assemblies zu nennen, die man auch tatsächlich benötigt.

Für's Übersetzen der ersten Version der C#-IDE aus diesem Sonderheft wird zum Beispiel die Assembly mit dem 'Managed C# Compiler' und die Assembly System Windows.Forms benötigt. Dem Compiler aus dieser IDE kann aber in der bisherigen Ausbaustufe gar keine Assembly-Referenz übergeben werden, und darum kann man die Entwicklungsumgebung auch nicht mit sich selbst übersetzen. Man braucht also einen Weg, den Compiler über Referenzen zu informieren, und das geht mit Hilfe eines Compiler-Parameters, dem */reference*-Parameter.

Der wird in der DIE-Version aus diesem Beitrag unterstützt. Das passiert zwar nur auf eine etwas dürftige Tour - es sollte aber kein Problem sein, mit den hier beschriebenen Methoden auch eine bessere Variante zu bauen. Nachdem die alte IDE mit der neuen IDE übersetzbar ist, kann man also auch die alte IDE um alle benötigten Features erweitern, wenn man keinen Kommandozeilen-Compiler aus dem SDK einsetzen kann.

• Optionen, Optionen

Letzten Endes sind die zu übergebenden Referenzen aber einfach nur ein Teil der Optionen, die dem Compiler übergeben werden können. Zu diesen Optionen zählt auch zum Beispiel der Schalter, der bestimmt, ob der Compiler Debug-Informationen im Programm unterbringen soll oder nicht, und genauso auch der Schalter, mit dem der Code-Optimierer angeworfen werden kann. Eine weitere Option ist, welche Referenzen verwendet werden sollen.

Es bietet sich nun an, für die Sammlung an Optionen einfach eine Klasse zu definieren, die für die Aufnahme der aktuell eingestellten Optionen zuständig ist. Im Beispiel werden nicht alle Compiler-Optionen unterstützt, es ist aber ein Leichtes, die Klasse entsprechend zu erweitern.

Für eine Optionen-Klasse, die zum Beispiel die Referenzen, den Zieltype der Anwendung und den Debug-Schalter aufnehmen kann, benötigt man zum Beispiel die folgenden Klasse (Listingl):

LISTING 1:

```
public class options
{
   private Target m_target;
   private ÄrrayList m_references;
   private bool m_debug;

   public options()
   {
     m_target = Target.exe;
     m_references =
     ➥new ÄrrayList ();
     m_debug = true;
   }
}
```

Dieses kurze Fragment enthält nun schon einige neue Elemente. Da gibt es das private Member 'm_target', das vom Typ 'Target' ist. Einen solchen Typ gibt es aber in .NET nicht - der muss erst noch definiert werden. Das 'Target' Member soll angeben, was für ein Ziel das Übersetzen des Quellcodes haben soll: Wird eine ausführbare Datei für die Konsole, ein Windows-Programm mit Fenstern oder eines der anderen möglichen Ziele gebaut?

Dazu bietet es sich an, diese möglichen Ziele in einer Aufzählung aufzulisten, und die sieht im Beispielcode folgendermaßen aus:

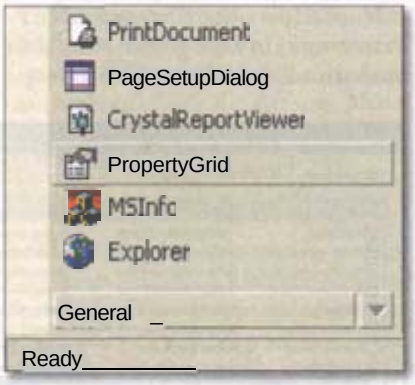
```
public enum Target
{
 exe,
 winexe,
 library,
 module
};
```

Die Enumeration Target kann also vier Werte annehmen – und im Konstruktor von options() wird der Member m_target dann auf den Wert Target.exe gesetzt.

Der Member m_reference ist vom Typ ÄrrayList. Bei einer ÄrrayList handelt es sich um eine der .NET Collection-Klassen, die genau das kapselt, was der Name verspricht: Man kann diese Klasse sowohl als Array als auch als Liste verwenden.

• Property-Methoden

Damit ist aber die Options Klasse noch nicht ganz fertig: Es fehlen offensichtlich Zugriffsfunktionen, mit denen die einzelnen Member der Klasse ermittelt und gesetzt werden können. Dafür bieten sich in diesem Fall 'Properties' an. Eine 'Property' (Eigenschaft) ist eigentlich eine ganz normale Funktion - nur setzt sie sich aus zwei Funktionen zusammen. Eine davon ist für das Lesen der Eigenschaft, die andere für das Schreiben der Eigenschaft zuständig. Solche Methoden werden auf eine bestimmte Art ausformuliert. Anders als normale Funktionen gibt es dabei keine Klammern hinter dem Namen der Funktion. Die beiden Bereiche fürs Lesen und Schreiben werden mit den Schlüsselworten 'get' und 'set' markiert.



NEU HINZUGEFÜGTE KOMPONENTEN erscheinen in der Toolbox immer ganz unten: Sie müssen also unter Umständen etwas scrollen, bevor Sie das Property-Grid finden.

Außerdem gibt es eine 'vordefinierte' Variable, die durch das Schlüsselwort 'value' verwendet werden kann: Das stellt bei der 'set'-Variante den Wert des Eingangsparameters dar. Die Property-Methode für das Member m_target sieht dann folgendermaßen aus:

```
public Target TARGET
{
 ..get{ return
   ➥m_target; }
  set{ m_target =
   ➥value; }
}
```

Die Methode hat also den Namen TARGET, ist vom Typ 'Target' und unterstützt sowohl einen 'get' als auch einen 'set' Zugriff. (Das ist optional - es müssen nicht beide Wege angeboten werden.) Die übrigen Methoden können

DAS IN DIE FORM eingebaute Property-Grid: Man kann schon erkennen, dass es sich dabei um die gleiche Komponente handelt, mit der Sie auch innerhalb von Visual Studio die Eigenschaften von Objekten bearbeiten.

dann analog implementiert werden. Den Grund dafür, weshalb sich hier Property-Methoden aufdrängen, erfahren Sie, wenn es darum geht, die eingestellten Optionen zu verändern.

• Compiler-Optionen setzen

Wenn Sie sich noch an die Art und Weise erinnern, wie der Compiler im Code anzuwerfen war, dann wissen Sie, dass dabei ein Objekt vom Typ Compiler-Parameter im Spiel war (Listing 2).

Dieses Objekt hat ein bestimmtes Feld -die Eigenschaft 'CompilerOptions'. In diesem Feld können die Compiler-Optionen als String übergeben werden. Damit man dies tun kann, ist es also hilfreich, wenn die 'options'-Klasse in der Lage ist, eine eigene Repräsentation in String-Form zu liefern. Dazu wird die 'ToString()'-Funktion implementiert Sie hat den Aufbau wie in Listing 3. In

LISTING 2:

```
CSharpCodeProvider cp = new CSharpCodeProvider();
ICodeCompiler cc = cp.CreateCompiler();
Compilerparameters p = new CompilerParameters();
p.GenerateExecutable = true;
// p.CompilerOptions = ...
```

der Beispiel-Implementierung wird dabei allerdings ein bisschen geschummelt, was man aber hier im abgebildeten Code nicht sehen kann. Im Beispiel wurde nämlich keine Funktionalität implementiert, mit der die Referenzen tatsächlich angegeben werden können. Statt

dessen wurden einfach die für das Übersetzen der Entwicklungsumgebung benötigten Referenzen fest eincodiert.

LISTING 3:

```
override public string ToString()
{
  string B = "/target:" + m_target.ToString()
    + " /debug" + (m_debug? "+" : "-")
    + " /optimize" + (m_optimize? "+" : "-")
    + " /checked" + (m_checked? "+" : "-");

  B += " /reference:";
  foreach( string sref in m_references)
  {
    B += sref + ",";
  }
}
```

Interessant ist hier im abgebildeten Code vor allem die Stelle, bei der das Ziel der Übersetzung genannt wird: „/target:" + m_target.ToString()

Wie Sie zuvor gesehen haben, handelt es sich bei 'Target' - das ist der Typ von m_target - um eine Enumeration. Interessanterweise kann man trotzdem unter Verwendung einer m_target-Instanz eine Methode aufrufen - nämlich

DAS PROPERTYGRID bietet die Möglichkeit, Objekt-Eigenschaften extrem einfach zu bearbeiten. Eigenen Code für diese Aufgabe brauchen Sie in Zukunft nicht mehr zu schreiben.

ToString(). Der Grund dafür ist, dass unter .NET bei C# auch eine Enumeration eine Klasse ist, und dass diese Klasse von Haus aus eben eine ToString()-Methode hat.

Wird die Methode aufgerufen, dann liefert Sie eine String-Repräsentation des ausgewählten Enumerations-Wertes. Und das ist praktischerweise einfach der symbolische Name aus der Enumeration als String. Der Wert Target.exe wird dadurch also zum String 'exe' - das ist auch der Grund, weshalb die Mitglieder der Enumeration so heißen wie sie heißen, denn die Namen sind einfach die Strings die man ohnehin dem Compiler übergeben muss. Das bedeutet, dass man im Verlauf des Programms die Enumeration wie eine solche verwenden kann.

Braucht man aber den zugehörigen Text, kann dieser einfach mit *ToStringO* aus einer Instanz gebildet werden.

Dieses ist ein wirklich extrem nützliches Feature in C#.

• Daten speichern

Soweit - so gut. Später sollen aber die gesetzten Optionen auch noch gespeichert werden - und zwar im Rahmen des Beispiels zusammen mit dem Quelltext, der in der IDE eingegeben wurde. Das macht zwar nicht viel Sinn, kann aber ein anderes Feature von .NET schön demonstrieren, zu dem Sie später im Beitrag mehr erfahren.

Im Beispiel werden also der Quelltext des eingegebenen Programms und die zugehörigen Optionen als die Daten angesehen, die von der IDE als 'native' Daten betrachtet werden. Darum liegt nichts näher, als eine weitere Klasse zu definieren, die diese Daten aufnehmen kann. Der Name dieser Klasse lautet 'nativedata'. Von der Logik her entspricht der Aufbau dieser Klasse genau dem Aufbau von 'options', nur enthält sie eben einen String und einen Member vom Typ 'options' (Listing 4).

Nun kann in der Form der Anwendung ein Member vom Typ 'nativedata' eingebettet werden und die Sache nimmt langsam Form an. Die Compiler-Optionen können dann vor dem Kompilieren leicht gesetzt werden:

```
p.CompilerOptions =
  m_data.
  Options.ToString();
```

Alles was nun noch fehlt, ist eine Möglichkeit, die Optionen zu setzen, und in diesem Zusammenhang lernen Sie eine etwas verstecktes aber extrem hilfreiches Control aus dem .NET Framework kennen. Dabei handelt es sich um das PropertyGrid.

Egal, ob sie mit Visual C++, Visual Basic oder anderen Programmen gearbeitet haben - das PropertyGrid wird Ihnen irgendwie bekannt vorkommen. Dabei handelt es sich um ein Control, in dem links der Name einer Eigenschaft steht, während rechts der aktuelle Wert dieser Eigenschaft angezeigt wird. Hat die Eigenschaft den Typ 'string', dann kann der Text im Feld, das den aktuellen Text anzeigt, auch editiert werden.

Handelt es sich hingegen um einen boolschen Wert, dann erscheint entweder eine Checkbox oder eine Combobox, mit der zwischen 'ja' und 'nein' umgeschaltet werden kann. Für andere Datentypen gibt es andere Repräsentationen. Genau das Beschriebene tut das PropertyGrid:

Es zeigt die Namen der Eigenschaften der Instanz einer Klasse an und ermöglicht es gleichzeitig, diese Eigenschaften zu bearbeiten. Und das Beste daran ist, man muss nichts dafür tun.

• Das Property-Grid

Das Property-Grid ist eine der Komponenten in .NET, bei deren Anwendung die Mächtigkeit des Frameworks geradezu in die Augen springt. Trotzdem hat Microsoft dieses Grid in der normalen Visual Studio IDE ein bisschen versteckt - warum, das wissen wohl nur die Geheimprogrammierer von Microsoft.

Beim Property-Grid handelt es sich um eine allgemeingültig verwendbare Komponente, mit der die 'Properties'

LISTING 4:

```
public class nativedata
{
 private string m_source;
 private options m_options;
 public nativedata()
 {
    m_options = new options ();
    m_source = "// ein leeres file";
 }
 public string Source
 {
    get { return m_source; }
    set { m_source = value; }
 }
 public options Options
 {
    get { return m_options; }
    set { m_options = value; }
 }
}
```

von Objekten bearbeitet und angezeigt werden können. Alle Eigenschaften eines Objekts, die mit der Propert-Get- bzw. Propert-Set-Methode erreichbar sind, können ohne weiteres Zutun mit dem Grid bearbeitet werden. Alles, was man dazu tun muss, ist das zu bearbeitende Objekte an das Grid zuzuweisen. Dabei können auch mehrere Objekte gleichzeitig bearbeitet werden, und natürlich sind die von Haus aus ins Grid eingebauten Editoren veränderbar. Gleiches gilt für die beschreibenden Texte, die Sortierreihenfolge und die Unterteilung der Eigenschaften in Kategorien. Das würde aber alles ein bisschen weit führen - für diesen ersten Kontakt mit dem PropertyGrid reicht aber auch die nicht veränderte Basisfunktionalität völlig aus.

Wie erwähnt, ist das PropertyGrid in der IDE ein bisschen versteckt worden. Bevor Sie es also im Folgenden nutzen können, muss diese Komponente ans Licht gezogen werden. Das Grid soll auf einer eigenen Form zum Einsatz kommen, und diese Form wird dann dem Einstellen von Kommandozeilen-Parametern für den Compiler dienen. Dort werden also die Eigenschaften des CompilerOptions-Objekts bearbeitet.

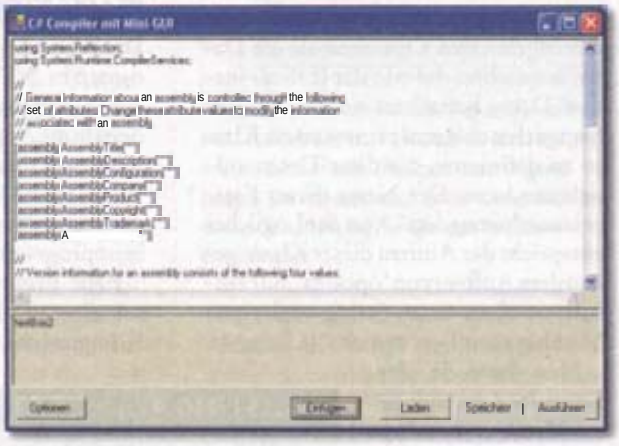
Zu diesem Zweck erzeugen Sie zunächst eine neue leere Form. Ist diese am Bildschirm angezeigt, dann öffnen Sie die Toolbox der Entwicklungsumgebung. Ist diese geöffnet, klicken Sie mit der rechten Maustaste auf einen freien Bereich der Toolbox und wählen dann den Befehl *Customize Toolbox*. Daraufhin öffnet sich ein entsprechender Dialog mit zwei Reitern. Auf dem einen Reiter werden die auf ihrem System vorliegenden COM-Komponenten angezeigt, der andere Reiter zeigt die vorliegenden .NET Framework-Komponenten an.

Dort suchen Sie dann nach einer Komponente mit dem Namen *PropertyGrid* und schalten dessen Options-Box ein. Wenn Sie den Dialog dann mit *OK* beenden, erscheint das PropertyGrid innerhalb der Toolbox: Sie haben eine neue Komponente, die Sie verwenden können. (Hinweis: Die neu zur Toolbox hinzugefügten Komponenten landen immer am unteren Rand der Toolbox, Sie müssen also unter Umständen ein bisschen scrollen bis das PropertyGrid sichtbar wird.) Nun können Sie einfach ein PropertyGrid aus der Toolbox auf die zuvor angelegte Form ziehen. Das Grid wird Ihnen dann sofort bekannt vorkommen - es handelt sich nämlich um die gleiche Komponente, mit der Sie auch die Eigenschaften der Forms oder die des C#-Compilers innerhalb von Visual Studio bearbeiten.

Man soll es nicht für möglich halten - aber die meiste Arbeit, die für's Bearbeiten von Objekt-Eigenschaften erledigt werden muss, ist damit schon erledigt. Das Einzige, was noch fehlt, ist eine Möglichkeit, das zu bearbeitende Objekt zuzuweisen. Dazu erweitern Sie die Form mit dem PropertyGrid um eine Methode names Set() die als Parameter einen Wert vom Typ 'options' erhält:

```
public void Set( options o)
```

Diese Methode erhält dann eine einzige Zeile Quellcode. In dieser Zeile weisen Sie das options-Objekt an das PropertyGrid zu: Fertig ist der Editor für die Compiler-Einstellungen (Listing 5).



DIE EIGENE ENTWICKLUNGSUMGEBUNG IN AKTION:
Nicht komfortabel aber funktional.

Nun muss nur noch ein Interface für die Benutzung der neuen Form geschaffen werden. Dazu legen Sie einfach einen Button auf der ursprünglichen Form an und geben diesem Button einen Event-Handler für den Click-Event mit folgendem Aussehen (Listing 6).

Wird der Button zur Laufzeit des Programms gedrückt, dann öffnet sich der Dialog (die Form), und die Optionen können bearbeitet werden: Einfach so, ohne dass weiterer Quellcode geschrieben werden muss. Wesentlich einfacher kann das Bearbeiten von Objekt-Eigenschaften nicht mehr werden.

LISTING 5:

```
public void Set( options o)
{
  this.propertyGrid1.
➥SelectedObject = o;
}
```

LISTING 6:

```
private void button4_Click
➥(object sender, System.
➥EventArgs e)
{
  Form2 f = new Form2();
  f.Set( m_data.Options );
  f.ShowDialog();
}
```

Das PropertyGrid ist dabei extrem flexibel. Sie können grundsätzliche alle Eigenschaften aller Objekte - egal, ob solche aus dem .NET-Framework oder selbst definierte - bearbeiten. Zumindest dann, wenn eine Set()- und eine Get()-Methode implementiert wurden. Wurde nur eine Set()-Methode implementiert, dann zeigt das PropertyGrid die entsprechende Eigenschaft nur an. Wenn Sie einmal Eigenschaften in einem Objekt haben, die nicht vom Grid angezeigt werden sollen, dann können Sie dazu das [Browsable()]-Attribut verwenden und selbiges mit dem Parameter 'false' ausstatten. Eine Property mit dem Attribut [Browsable(false)] wird also nicht angezeigt.

Darüber hinaus kann das PropertyGrid auch mit einer ganzen Reihe an weiteren Attributen umgehen. So existieren Attribute, mit denen Sie die Eigenschaften in verschiedenen Kategorien unterbringen können, und es existieren auch solche, mit denen Sie die Hilfetexte für die Properties festlegen können.

Auch sonst ist mehr oder minder alles im PropertyGrid konfigurierbar. Das geht soweit, dass Sie sogar die für die einzelnen Properties angezeigten Texte verändern können; tun Sie das nicht, so werden einfach die Namen der Variable aus Ihrem Quellcode verwendet. Herkömmlichen Code für das Bearbeiten von Objekt-Eigenschaften brauchen Sie in Zukunft dank des PropertyGrids mit .NET nie wieder zu programmieren.

Nun wissen Sie, wie Sie, aufbauend auf die Kommandozeilen-Compiler in .NET, eine eigenen kleine Entwicklungsumgebung bauen und wie Sie mit dem PropertyGrid arbeiten.

UserControls mit C#

Controls selber machen

UserControls zu entwerfen, ist mit C# ein reines Kinderspiel. Die Regeln zum Spiel erfahren Sie in diesem Beitrag.

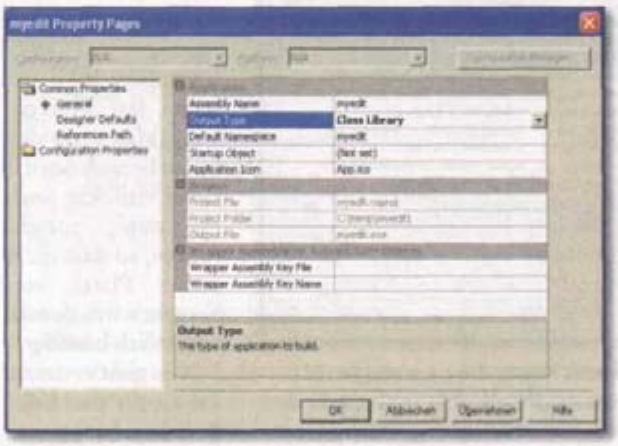
THOMAS WÖLFER

Die Möglichkeit, eigene Kontrollelemente zu entwerfen, gibt es unter Windows schon lange. Ganz früher waren das einmal 'CustomControls', und die entwickelten sich im Laufe der Zeit zu ActiveX Controls, bei denen es sich im Wesentlichen um spezielle COM-Objekte handelte. Eines hatten aber alle Geschmacksrichtungen gemeinsam: Sie waren doch eher aufwändig zu implementieren.

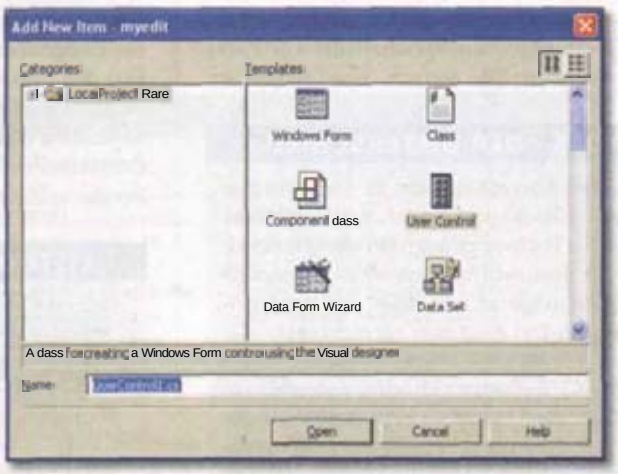
Wie Sie an anderer Stelle in diesem Sonderheft erfahren haben, können Sie Typen, die innerhalb der Regeln der CLR definiert wurden, in allen Programmen, die innerhalb der CLR laufen, einfach weiterverwenden.

Das gilt natürlich auch für Kontrollelemente, die visuelle Interfaces besitzen, also Buttons, Listboxen und dergleichen. Natürlich können Sie auch eigene Kontrollelemente erzeugen und diese dann weiterverwenden.

Das ist sehr viel einfacher, als man meinen möchte. Besonders, wenn man ein wenig erfahren mit der Programmierung von ActiveX Controls oder COM Objekten ist, wird man die neue Einfachheit schnell schätzen lernen. Prinzipiell folgt die Programmierung und Nutzung eigener Elemente diesen Schritten:



WENN SIE SICH BEIM ANLEGEN des DLL Projektes vertan haben: Kein Problem - der Projekttyp kann auch nachträglich verändert werden.



DER ERSTE SCHRITT zum eigenen Kontrollelement besteht aus der Erzeugung eines UserControls.

- Ein neues Projekt vom Typ 'Windows DLL' wird angelegt.
- In diesem Projekt werden neue Typen definiert. Soll es sich dabei um Kontrollelemente handeln, dann werden diese Typen von UserControl abgeleitet.
- Der oder die Typen werden dann implementiert, und die DLL wird übersetzt.
- Zum Testen wird dann ein weiteres Projekt angelegt, diesmal vom Typ 'Windows Forms Application'.
- Bei diesem Projekt wird in den Forms Editor gewechselt. Von dort aus kann dann die Visual Studio Toolbox erweitert werden.
- Dabei wird das zuvor programmierte Steuerelement zur Toolbox hinzugefügt.
- Das neue Element kann nun genau so auf Windows Forms verwendet werden, wie das bei den mitgelieferten .NET-Elementen der Fall ist.

Im folgenden Beispiel wird ein solches Kontrollelement implementiert. Das Control setzt sich dabei aus zwei vorhandenen Elementen zusammen: Eine Textbox dient der Eingabe von Text, und eine Combobox bietet die letzten Eingaben dieser Textbox zur Auswahl an. Die Combobox merkt sich also alle Eingaben der Textbox.

Um die Sache übersichtlich zu halten, wurde dabei auf optische Spielereien, das Ein- und Ausblenden von Teilen des Controls und dergleichen, verzichtet: Das fertige Control ist aber ein vollwertiges Kontrollelement und kann sehr wohl als Ausgangspunkt für ein sehr praktisches Element dienen.

• Der erste Schritt: Die DLL

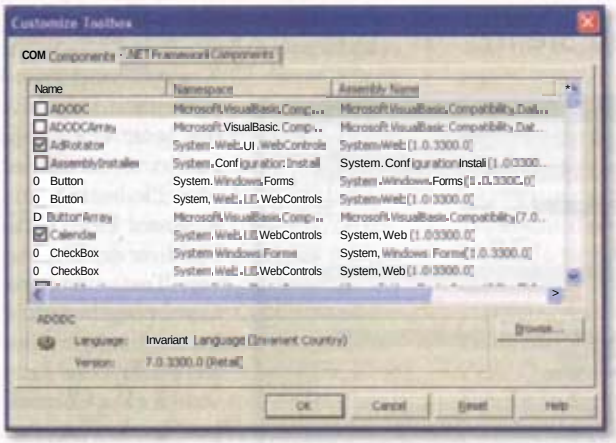
Für dieses Beispiel brauchen Sie zwei Projekte: Das eine erzeugt eine Windows Forms-Anwendung. Das zweite Projekt erzeugt eine DLL, die das selbst definierte Kontrollelement enthält. Das Element wird dann in der Anwendung verwendet. Um dabei nicht die Übersicht zu verlieren, sollten Sie zunächst beide Projekte in einer Visual Studio Solution belassen.

Eine Visual Studio 'Solution' kann beliebig viele Projekte aufnehmen, wobei ein einzelnes Projekt jeweils eine bestimmtes Build-Target hat:

Ein Projekt erzeugt also eine DLL, ein Setup-Programm, ein Executable oder eines der anderen möglichen Ziele. Eine Solution fasst mehrere solcher Projekte zusammen.

Legen Sie also zunächst ein Projekt vom Typ 'DLL' an. Visual Studio setzt dannalleCompiler-Switchesrichtigund erzeugt ein leeres Projekt. Dieses Projekt erweitern Sie dann per rechtem Mausklick um ein 'UserControl'.

Ein UserControl ist eine Klasse, die von *System.Windows.Forms.UserControl* abgeleitet ist. (Man kann eigene Controls auch direkt von Controls aus den Windows Forms ableiten - im Beispiel wird aber der 'normale' Weg per UserControl gewählt.) Das neue User-



UM DIE TOOLBOX zu erweitern, müssen Sie zur erzeugten DLL navigieren. Doch nur dann, wenn sich diese nicht im GAC befindet.

Control wird dabei auch gleich im Forms-Editor angezeigt. Es besteht im Wesentlichenauseinerleeren Fläche,auf der neue Elemente platziert werden können - fast wie eine normale Form.

Das zu entwickelnde Element soll sich aus zwei Teilen zusammensetzen: Es soll eine normale Textbox für die Eingabe von Text geben und eine ComboBox, die die bisher eingegebenen Texte zusätzlich zur Auswahl anbietet.

Beide Elemente werden einfach in einem Panel untergebracht und durch einen Splitter getrennt. Damit sich das Elemente beim Vergrößern und Verkleinern im Wesentlichen so verhält, wie eine normale Textbox, dockt man die Combox und den Splitter rechts und setzt das Docking-Verhalten der Textbox auf *Fill*. Wird das Panel dann später vergrößert, vergrößert sich nur die Text-Box, und der ComboBox-Teil bleibt in seinen Ausmaßen erhalten.

Optisch ist damit schon fast alles geklärt: Man sollte den freien Bereich um das Panel vielleicht noch ein wenig zurecht rücken, so dass nicht mehr Platz verbraucht wird, als man tatsächlich benötigt.

Nun geht es daran, die Logik des Kontrollelements zu entwickeln. Zum einen braucht man eine Möglichkeit, den Text der Textbox zu speichern, zum anderen muss darauf reagiert werden, wenn der Anwender einen Eintrag aus der ComboBox ausgewählt hat. Zum Speichern der Texte verwenden Sie einfach eine ArrayList. ArrayListen stammen aus dem *System.Collections.Namespace* und stellen einfach eine Listen-Implementierung dar. Dabei können Elemente vom Typ 'object' zur Liste hinzugefügt und aus der Liste entfernt werden. Nachdem strings vom .NET-Datentyp 'string' auch 'objects' sind, kann man die ArrayList auch für diese verwenden. Und wie es der Zufall will, wird der Text, der in einer TextBox eingegeben wird als 'string' gespeichert. Was man also nun braucht, ist ein Member vom Typ ArrayList im Control:

```
private ArrayList    m_recent;
```

Die Liste muss aber auch erzeugt werden. Das machen Sie am besten im Konstruktor des Kontrollelementes:

```
public UserControl1()
{
..InitializeComponent();
  m_recent = new ArrayList();
}
```

Platz zum Speichern ist nun da- es muss nur noch gespeichert werden. Was gespeichert werden soll, ist der Text, der in der TextBox eingegeben wurde. Und der steht immer dann fest, wenn die TextBox den Focus verliert. Nichts liegt also näher, als einen Event-Handler für die TextBox zu implementieren, der das LostFocus Event behandelt.

Dazu klicken Sie zunächst auf die Textbox und wechseln dann in die Eigenschafts-Ansicht des Visual Studio. Dort aktivieren Sie die Ansicht der Events und suchen nach dem Leave Event: Das ist der 'offizielle' Name für das Ereignis, das eintritt wenn die Text-Box den Focus verliert.

Die Implementierung dieses Handlers ist denkbar einfach: Sie speichern einfach den aktuellen Text aus der TextBox in der ArrayList *m_recent* (Listing 1):

LISTING 1:

```
private void OnLeave(object sender,
➥System.EventArgs e)
{
  m_recent.Add( this.textBox1.Text);
}
```

Nun müssen Sie noch sicherstellen, dass die Combox-Box die gespeicherten Texte auch anzeigt. Das geht am einfachsten, wenn Sie die Combo-Box mit den gespeicherten Werten auffüllen, wann immer dieses Element den Focus gewinnt. Dazu muss das GotFocus Ereignis behandelt werden (Listing 2).

Bevor Sie die zwischengespeicherten Texte einfügen, löschen Sie alle Ele-

KOMPONENTEN UND CONTROLS - DER UNTERSCHIED

Bei .NET wird genau wie bei älteren Windows-Umgebungen zwischen 'Components' (Komponenten) und 'Controls' unterschieden. Dabei ist der Unterschied aber eigentlich gar nicht so groß, denn ein Control ist auch immer eine Komponente.

Im Wesentlichen ist eine Komponente ein Objekt, das mit anderen Objekten interagieren kann und das einfach in mehreren Programmen wiederverwendet werden kann. Genau das tut ein Control auch - aber es bietet zusätzlich noch ein optisches Interface, so dass es im visuellen Designer, wie dem Forms-Editor, direkt manipuliert werden kann: So ist es bei einem Control beispielsweise möglich, die Größe und Form zu verändern. Bei einer einfachen Komponente geht das nicht - denn die hat ja gar kein Userinterface.

Wie Sie in diesem Beispielprogramm erfahren haben, ist es relativ einfach, ein eigenes Kontrollelement zu implementieren. Allerdings ist die Art, wie das Element in die Toolbox gelangt, ein wenig unhandlich, denn man muss immer zur passenden DLL navigieren und diese von Hand eintragen. Das muss aber gar nicht sein - dazu gibt es den Global Assembly Cache (GAC). Befindet sich eine Assembly im GAC, dann taucht sie einfach direkt in der Liste der Assemblies auf, und Objekte daraus können direkt ausgewählt werden: Die Navigation entfällt.

Um Assemblies im GAC zu platzieren, gibt es verschiedene Tools, die sich im .NET SDK befinden, und auch Setup-Programme für .NET sind in der Lage, Assemblies im GAC abzulegen. Weitere Details dazu finden Sie in der Dokumentation des .NET SDKs unter dem Begriff 'Global Assembly Cache'.

mente aus der ComboBox: Das stellt sicher, dass keine Einträge doppelt vorkommen.

LISTING 2:

```
private void GetFocus(object sender, System.EventArgs e)
{
  this.comboBox1.Items.Clear();
  foreach( string s in m_recent)
    this.comboBox1.Items.Add( s);
}
```

Nun könnten man sich die ArrayList m_recent natürlich auch ganz sparen und die Texte direkt in der Sammlung aus 'Items' in der ComboBox speichern. Der vorgeschlagene Weg hat aber seine Vorteile - und dabei geht es im Wesentlich enum eine einfacheErweiterbarkeit. Nachdem das Auffüllen der Combo-Box in einer separaten Funktion besorgt wird, bietet sich diese Funktion auch für spätere Erweiterungen an. Unter Umständen wollen Sie bestimmte Texte eben nicht speichern und herausfiltern, oder Sie wollen die Texte in einer bestimmten Sortierreihenfolge in die ComboBox einfügen: Alle dies wäre in *GetFocus()* gut aufgehoben.

Bleibt nur noch die Implementierung, die sich damit beschäftigt, dass eine Auswahl aus der Combo-Box in der Text-Box eingetragen wird. Dazu muss man wissen, welchen Text der Benutzer aus der Combo-Box ausgewählt hat. Auch das ist aber einfach zu ermitteln. Wird ein Text aus der ComboBox ausgewählt und diese dann geschlossen, dann löst das das *SelectionChangeCommit*-Ereignis aus. Das müssen Sie behandeln. Im Zuge der Behandlung ermitteln Sie einfach das momentan ausgewählte ElementderComboBox und wandeln es mit der ToString()-Methode, die jedes .NET-Objekt unterstützt, in einen Text um. Nachdem es sich bei den Einträgen in dieser ComboBox ohnehin um Texte handelt, gibt es hier nicht viel umzuwandeln - und der ausgewählte Text erscheint in der TextBox. Damit ist das eigene UserControl fertig. Sie können das Projekt nun übersetzen.

DAS EIGENE CONTROL IN AKTION: Eingegebene Texte werden in der Combox-Box zur Auswahl angeboten.

• UserControls verwenden

Im nächsten Schritt soll das Control natürlich auch getestet werden. Dazu legen Sie innerhalb der aktuellen Solution ein neues Projekt vom Typ 'Windows FormsApplication' an. Visual Studio erzeugt dafür eine fertige leere Form. Diese Form ist zum Testen schon ausreichend. Damit Sie Ihre Control verwenden können, müssen Sie sich aber noch um deren Erreichbarkeit kümmern.

LISTING 3:

```
private void OnCommit(object sender, System.EventArgs e)
{
  if( this.comboBox1.SelectedItem != null)
  {
    this.textBox1.Text = this.comboBox1.SelectedItem.ToString();
  }
}
```

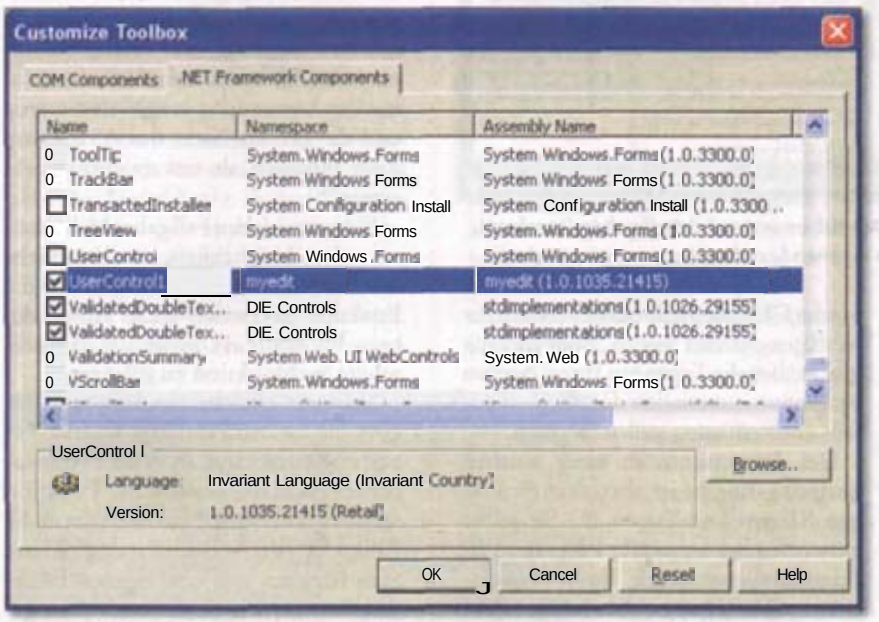
Schließlich soll das eigene Control genau so leicht zu verwenden sein wie die mitgelieferten TextBoxen.

Auch das ist aber kein Problem. Sie klicken dazu mit der rechten Maustaste auf die Toolbox und wählen dann den Befehl *Customize Toolbox*. Es öffnet sich ein Dialog, aus dem Sie .NET- oder COM-Komponenten auswählen können. Dabei gibt es auch einen 'Browser-Button, mit dem Sie nach Dateien, die Komponenten implementieren, suchen können. Damit navigieren Sie zu der DLL, die Sie soeben übersetzt haben.

Wenn Sie diese DLL ausgewählt haben, erscheint das soeben programmierte Control unter der Bezeichnung *UserControl1* in der Toolbox. Der Name ist dabei vielleicht etwas ungünstig - er resultiert aber einfach aus dem Namen der Klasse, die das Control implementiert. Wenn Sie diesen Namen in Ihrer DLL ändern, dann taucht auch ein anderer Name in der Toolbox auf.

Dieses Kontrollelement können Sie nun ganz normal auf die Form ziehen. Wenn Sie jetzt das Projekt übersetzen unddas Programm starten, haben Sie bereits ein fertige Beispiel für ein Testprojekt, das Ihr eigenes Control verwendet.

Nun wissen sie, wie einfach Sie eigene Kontroll-Elemente mit .NET erstellen - und wie einfach sie sich in die vorhandene Entwicklungsumgebung einpassen. © UR



EINMAL AUSGEWÄHLT, erscheint das eigene Control in der Liste der verfügbaren Objekte. Dort markiert, erscheint es in der Toolbox.

Die Skript-Engine

Eigene Objekte wieder verwenden

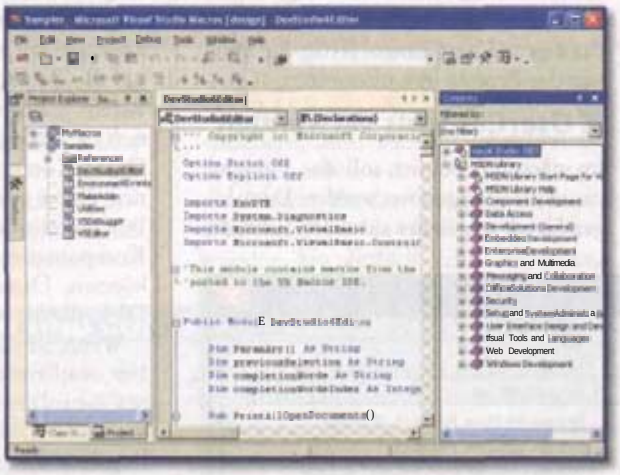
Bei .NET gibt es eine interesante Möglichkeit, Programme zu übersetzen - mit der Scripting-Engine. Dabei handelt es sich im Wesentlichen zwar auch um einen Compiler, aber der kann relativ einfach in das eigenen Programm eingebaut und dort als Scripting-Motor benutzt werden.

THOMAS WÖLFER

Der Unterschied zwischen Skripten und kompilierten Programmen ist in .NET nicht mehr sonderlich groß. Tatsächlich ist es so, dass auch Skripte einfach bei Bedarf kompiliert werden. Das bedeutet auch, dass Sie Skripte problemlos von Haus aus in den .NET-Sprachen VB und C# schreiben können. Das Schreiben von Skripten als solches ist aber gar nicht so interessant - warum ein Skript in einer Sprache schreiben, wenn man in der gleichen Sprache auch ein fertiges Kompilat herstellen kann?

Der tatsächliche Sinn von Skripten ist es, das eigene Programm mit einer Engine auszustatten, die es dem Anwender ermöglicht, das Programm um eigene Funktionalität zu erweitern. Mit anderen Worten: Das, was mit Visual Basic für Applications in Office und einigen anderen Programmen möglich war, kann ab sofort auch in der eigenen Anwendung eingebaut werden. Und das mit .NET ohne großen Aufwand.

Der Trick beim Skripting ist dabei der, dass die dem Skript zur Verfügung stehenden Typen dynamisch erweitert werden können. Der Benutzer der Scripting-Engine - also der Anwender - sieht davon nichts: Für ihn sind diese Typen fest eingebaute Typen aus der Scripting-Engine der Anwendung. Insgesamt setzen sich also die zur Verfügung stehenden Typen für den Anwender aus zwei Gruppen zusammen. Zum einen gibt es da die Typen aus der CLR, die Sie ihren Anwendern zur Verfügung stellen wollen. Zum anderen gibt es eben die Typen aus Ihrem eigenen Programm - zumindest die, die sie ebenfalls zur Verfügung stellen möchten.



MICROSOFTS MACRO-IDE basiert auf den gleichen Interfaces, die Sie auch für den eigenen **Scripting-Support** verwenden.

Das Interessante an einer solchen Skripting-Engine ist also, dass diejenigen Klassen und Typen, die Sie selbst entworfen haben, um Ihr Programm zu Programmieren, auch Ihren Anwendern zur Verfügung stehen können. Um das etwas verständlicher zu machen, sei ein kleines Beispiel gezeigt.

• Ein Beispiel

Angenommen, Sie haben in Ihrem Programm eine Klasse namens 'Buch' und eine Klasse namens 'BuchKatalog' implementiert. Der Buchkatalog dient zum iterieren über alle Bücher, während eine einzelne Instanz vom Typ Buch die Informationen über ein spezielles Buch liefert. Also etwa die ISBN-Nummer, den Titel und so weiter.

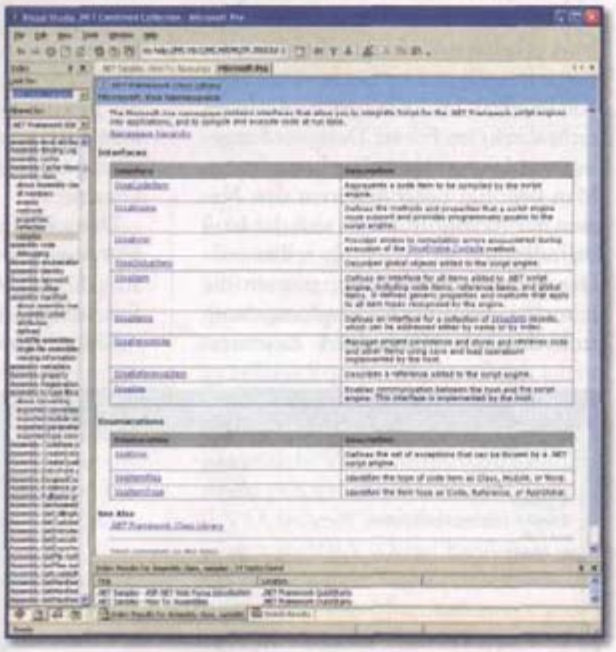
Nun haben Sie in Ihrer Anwendung eine Methode zum Hinzufügen und zum Löschen von Büchern aus dem Katalog implementiert - aber keine Methode, um nach einem bestimmten Buch zu suchen. (Zugegeben: Das ist ein bisschen unwahrscheinlich. Es ist aber nun mal eine Tatsache, dass Anwender immer eine Funktion 'erfinden', die nicht mit der Anwendung mitgeliefert wurde und die sich aus Sicht des Anwenders dann plötzlich als unverzichtbar herausstellt.)

In einem solchen Fall gab es bisher keine andere Möglichkeit, als die tatsächlich ausgelieferte Anwendung um die Funktion zu erweitern: Der Anwender brauchte dann ein Update, um an die ersehnte Suchfunktion zu gelangen.

Mit einer eingebauten Scripting-Engine, die auf Anwendungs-interne Typen zugreifen kann, ist es dem Anwender möglich, die erwünschte Funktion selbst zu schreiben. Dazu benutzt er die Fähigkeit der 'BuchKatalog'-Klasse zum Iterieren und eine eigenen Funktionalität für die Suche nach dem gewünschten Buch. Das Skript könnte in etwa so aussehen wie in Listing 1.

Die Sprachkonstrukte, wie 'foreach', stellen dabei die im Skript verwendete Sprache zur Verfügung, die MessageBox-Klasse stammt aus dem .NET-Framework und die Typen 'Buch' und 'Buchkatalog' stammen aus Ihrer Anwendung.

Ein wesentlich mächtigeres Werkzeug für die Erweiterung einer Anwendung kann man dem Anwender wohl kaum bieten: Er kann alles tun, was auch der Programmierer der Anwendung selbst tun kann - und dabei auf alle Objekte zurückgreifen, die der Programmierer selbst auch verwendet hat. Der Programmierer leistet also einen großen Teil an Vorarbeit, die sich in Klassen niederschlägt - und diese liegen dann nicht wie bisher unproduktiv herum, sondern können vom Anwender wiederverwendet werden.



DER NAMESPACE MICROSOFT.VSA enthält alles was das Herz begehrt. Zumindest dann, wenn man die eigenen Anwendung um Scripting-Fähigkeiten erweitern möchte.

• Scripting mit .NET

Um Funktionalität und Objekte per Scripting verfügbar zu machen, braucht man natürlich zunächst einmal eines: die Funktionalität. Es gilt also, zunächst einmal etwas zu implementieren, das man dann später zur Verfügung stellen kann.

Genau das tut das Beispielsprogramm: Es stellt ein Art Ampel zur Verfügung. Diese Ampel ist einfach als Windows Form implementiert und zeigt das aktuelle Ampel-Licht in der Caption-Bar an. Dazu gibt es drei Icons: Überraschenderweise sind diese Icons rot, gelb und grün. Die Ampel-Klasse befindet sich dabei in einer eigenen DLL. Zusätzlich gibt es noch ein Testprogramm, das die Ampel-Klasse verwendet und mit dem die Ampel Klasse ganz 'klassisch' getestet werden kann.(Der komplette Quellcode zu den Projekten befindet sich auf der Heft-CD.)

Die Ampel selbst befindet sich in der Form 'lightform', die ihrerseits in der Datei 'lightform.cs' definiert ist. Im gleichen Projekt befinden sich außerdem die drei farbigen Icons. Das 'Ampel-Verhalten' wird dabei einfach dadurch realisiert, dass das Icon in der Form zur Laufzeit zwischen den drei vorhandenen Icons umgeschaltet wird.

EINE 'AMPEL': Dieses Objekt wird für Skript verfügbar gemacht und kann dann auch von Skript aus gesteuert werden.

Das initial anzuzeigende Icon kann dabei ganz einfach mit dem Forms Editor gesetzt werden. Die Ampel soll aber einen 'Dienst' - nämlich den Ampel-Dienst - zur Verfügung stellen, der von anderer Stelle genutzt werden kann. Dieser Dienst besteht im Wesentlichen daraus, dass man die Ampel-Farbe setzen kann.

Dazu genügt einfach nur eine Eigenschaft, die gesetzt werden kann. Im Beispiel ist dies die Eigenschaft 'Light'. Dieser Eigenschaft kann ein Wert vom Typ 'Lights' zugewiesen werden, und

DIE AMPEL meldet sich in der Taskbar. Zunächst kann man sie mit dem Client-Programm testen.

'Lights' ist seinerseits einfach eine Enumeration, die die gültigen Werte für die Ampel aufzählt:

```
public enum Lights
{
    Red,
    Green,
    Yellow
}
```

Um dann auf einen der Werte zuzugreifen, kann man einfach *Lights.Red*, *Lights.Green* oder *Lights.Yellow* schreiben. Die Enumeration befindet sich in der Datei, in der auch die 'lightsform' selbstdefiniert ist. An dieser Stelle ein kleiner, wichtiger Sicherheitshinweis zu sehr ärgerlichen Fehlern in Visual Studio .NET, der alle C# Programmierer betrifft.

• Ein kleiner, aber ärgerlicher Fehler

Normalerweise würde man sicherlich schon aus Gründen der Übersichtlichkeit die 'Lights'-Enumeration oberhalb der Lightsform-Klasse niederschreiben. Bei C# ist es zwar egal, an welcher Stelle in einem Projekt eine bestimmte Definition steht - trotzdem ist es für viele Programmierer vermutlich schon gefühlsmäßig so, dass der richtige Platz für's Definieren einer Enumeration vor der Stelle der ersten Verwendung dieser Enumeration liegt. Mit anderen Worten: Normalerweise würde die Datei 'lightsform.cs' in etwa den Aufbau wie in Listing 2 haben.

Das geht leider nicht so. Zwar bemängeln weder der Compiler noch die Entwicklungsumgebung irgend etwas an dieser Reihenfolge - und die sollte auch

LISTING 1:

```
foreach( Buch b in BuchKatalog)
{
   if( b.Titel == strSuchtTitel)
   {
      MessageBox.Show("Buch gefunden. ISBN ist:
      ➥" + b.ISBN);
   }
}
```

tatsächlich keinerlei Ärger machen - aber wenn man später versucht, das Programm auszuführen, stürzt es an einer

LISTING 2:

```
namespace lights
{
  public enum Lights
  {
  Red,
  Green,
  Yellow
  }

    // jetzt kommt die
    ➥klasse ...

   public class lightsform {
    // und so weiter

}
```

vom Forms Designer erzeugten Stelle ab. Die Zeile ist dabei die Zeile, bei der das initiale Icon gesetzt wird:

```
this.Icon = (
➥System.Drawing.Icon)
➥resources.GetObject
➥("$this.Icon")));
```

Ganz egal, was man vor der eigentlichen Klasse für die Form hingeschrieben hat: Eine andere Klassendefinition, eine Enumeration oder sonst ein gültiges Statement: Handelt es sich dabei nicht um einen Kommentar, wird die Icon-Ressource beim Laden des Programms nicht gefunden, und man erhält einen Laufzeitfehler. Das kann auf Dauer sehr lästig werden. Besonders deshalb, weil man der fehlschlagenden Zeile keine Probleme ansehen kann (schließlich ist sie ja auch völlig korrekt.). Hatte man zuvor eine funktionierende Form - einschließlich Icon - wird es eine ganze Weile dauern, bis man auf die Idee kommt, dass die Definition der Enumeration vor der Klassendefinition das Problem auslöst.

Man muss sich also merken, dass man selbiges besser sein lässt; die Ressourcenverwaltung kommt sonst durcheinander. Im Beispiel steht die Enumeration aus diesem Grund unterhalb der Klassendefinition. (Vermutlich wäre sie sogar als Teil der Ampel-Klasse besser aufgehoben, für dieses Beispiel ist das 'richtige' Design aber eher gleichgültig.)

• Eigenschaften der Ampel-Klasse

Die Ampelklasse hat, wie bereits erwähnt, eine einzige Eigenschaft vom Typ 'Lights'. Weist man dieser Eigenschaft einen neuen Wert zu, wird ein Icon geladen und als aktuelles Icon der Form festgelegt. Der Code zum Laden von Icons aus DLLs ist - sofern die Icons nicht direkt im Forms Designer festgelegt werden - ein bisschen kompliziert. Man benötigt unter anderem den Namen der Assembly, in der sich das Icon befindet, und die Assembly selbst.
Damit das Laden gelingt, müssen die Icons in der Entwicklungsumgebung außerdem als 'Embedded Resource' markiert werden. Das geht in der Eigenschafts-Ansicht für die Icons. Die komplette Methode zum Setzen der Lichter sieht dann aus wie in Listing 3.

Der Name der .ico-Datei wird dabei aus dem Wert der übergebene Enumeration synthetisiert. Die Enumeration hat ja die Member 'red', 'yellow' und 'green'. Die .ico-Dateien haben die Namen 'red.ico', 'yellow.ico' und 'green.ico'.

Verwendet man also den Wert der Enumeration und konvertiert diesen in einen String - das geht mit dem von Haus aus vorhandenen Typen-Konverter - dann erhält man den Basisnamen der gewünschten .ico-Datei. Alles was man noch tun muss, ist die Dateierweiterung '.ico' anzuhängen: Fertig ist der Dateiname.

Danach kann das Icon per *GetManifestresourceStream()* aus der momentan ausgeführten Assembly [Assembly.GetExecutingAssembly()] geladen werden.

Schließlich wird das Icon noch als das für die Form zu verwendente Icon gesetzt: fertig.

MIT DEM 'AUSFÜHREN'-BUTTON kann man die Scripting-Fähigkeiten testen: Das Beispielskript zur Ampel-Steuerung befindet sich auf der Heft-CD.

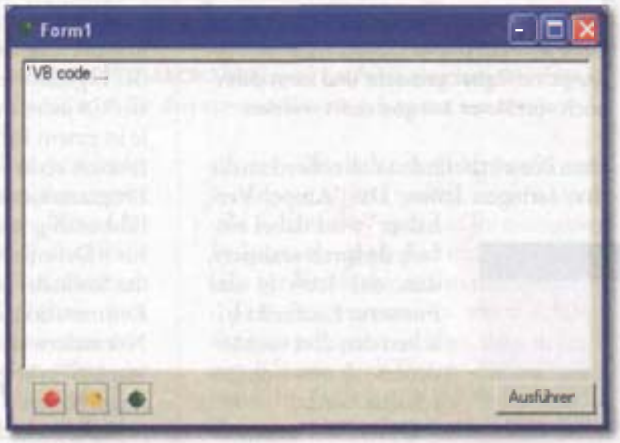
• Der erste Client

Im nächsten Schritt gibt es einen Client, der die Ampel auch tatsächlich verwendet. Dabei handelt es sich um Ihr 'Anwendungsprogramm' - wenn auch um ein sehr kleines: Das einzige was es tut, ist eine Ampel anzuzeigen. Für's Beispiel ist das aber ausreichend - denn hinterher soll ja die AmpeP-Funktionalität auch noch per Skript erreichbar gemacht werden.

Im Beispiel wird der Ampel-Client einfach als ein .EXEProjekt, das Teil der VS Solution ist, hergestellt. Im Wesentlichen wird die vom Wizard erzeugte Form um drei Buttons erweitert. Die Buttons haben - natürlich - die passenden Farben: rot, grün und gelb. Dazu werden einfach Buttons auf der Form platziert, diese erhalten dann die entsprechenden Icons als 'Hintergrundbild'. Das geht direkt im Forms Designer und bedarf keinerlei Programmierung.

Der 'rote' Button soll nun Ampel dazu veranlassen, das rote Icon anzuzeigen, der gelbe Button zeigt das gelbe Icon an und so weiter.

Dazu müssen drei Event-Handler definiert werden. Die definiert man am einfachsten durch je einen Doppelklick auf die drei Buttons. Die Entwicklungsumgebung fügt dann passende 'Click'-Handler in die Form ein. Bevor diese Handler aber ir-



DAS CLIENT-PROGRAMM ist wenig spektakulär. Das muss es auch nicht sein, denn es handelt sich ja nur um ein möglichst einfaches Beispiel.

LISTING 3:

```
public Lights Light
{
  set
  {
  string strAN = Assembly.GetExecutingAssembly().GetName().Name;
  string strName = value.ToString().ToLower() + ".ico";
  string name = strAN + "." + strName;
  Stream s = Assembly.GetExecutingAssembly().
  ➥GetManifestResourceStream( name);
  Icon icon = new Icon( s, 16, 16);
  this.Icon = icon;

  }
}
```

gendwas behandeln können, wird natürlich auch noch eine Ampel-Form gebraucht: Sonst ist ja niemand da, der die Icons anzeigen könnte. Der Einfachheit halber wird die Ampel-Form im Konstruktor des Clients angelegt und angezeigt, wie dies in Listing 4 zu sehen ist.

Die Handler verwenden dann einfach den Set-Accessor zum Setzen der gewünschten Farbe. Der Handler fürs Setzen des roten Icons sieht also beispielsweise wie Listing 5 aus.

Wenn Sie dieses Beispiel nun übersetzen, haben Sie bereits alles zur Hand, was im Folgenden benötigt wird: ein Anwendungsprogramm, das die Dienste einer Klasse als Client in Anspruch nimmt. Die Ampel kann direkt vom Anwendungsprogramm umgestellt werden. Das ist natürlich nicht sonderlich

LISTING 4:

```
private lightform lf = new lightform();
public Form1()
{
  InitializeComponent();
  lf.Show();
}
```

spektakulär - denn schließlich ist dies genau der Funktionsumfang der auch bisher möglich war. Jetzt wird die Sache etwas spektakulärer, denn es geht ans Skripting.

• Skripte ausführen: Microsoft.VSA

Wie das immer so ist bei Microsoft, es existieren jede Menge Softwareschichten. Jedes Mal, wenn man sich tief genug hineingebohrt hat, findet man ganz unweigerlich irgendwann eine Schicht, die extra dafür gedacht ist, zusätzliche Einnahmen zu verschaffen. Das ist ja grundsätzlich nichts Negatives - auch wenn es immer wieder erstaunt, an welchen Stellen derlei Dinge auftauchen. Man kann es sich leicht denken: Auch beim Scripting in .NET ist das nicht anders - der Knackpunkt nennt sich 'Visual Studio for Applications' (VSA).

VSA ist im Wesentlichen die Fortsetzung von VBA. Dabei handelt es sich um eine lizensierbare Komponente, die im Auftrag von Microsoft unter anderem von Summit Software vertrieben wird. Mit dieser Komponente kann man seine eigene Anwendung relativ einfach mit einer Scripting-Entwicklungsumgebung ausstatten.

Genau die Skript-Entwicklungsumgebung (inklusive Debugger, IntelliSense und allen anderen Annehmlichkeiten), die auch in Visual Studio eingebaut ist, ist zu haben. Allerdings für einen nicht gerade geringen Preis.

Man muss diese Umgebung aber nicht einsetzen, wenn man selbst Scripting anbieten will. Nur muss man dann eben auf die Entwicklungsumgebung verzichten. Wer aber selbst eine Entwicklungsumgebung zusammenbaut – und im einfachsten Fall reicht dafür ja ein TextBox Control - kommt auch in den Genuss von Scripting, denn die benötigte Laufzeitumgebung ist komplett vorhanden. Allerdings ist sie ein bisschen schwierig einzusetzen und nicht sonderlich gut dokumentiert. Das macht aber nichts - denn Sie haben ja

LISTING 5:

```
private void redButton_Click(object Sender, System.EventArgs e)
{
  lf.Light = Lights.Red;
}
```

den Rest dieses Beitrags, um auf den richtigen Weg zu kommen.

Scripting wird von Haus aus für VB.Net und JavaScript unterstützt, es wäre aber auch möglich, weitere Sprachen zu verwenden, wenn man einen entsprechend höheren Aufwand treiben würde.

Die benötigten Komponenten befinden sich in Microsoft.VSA und Microsoft.VisualBasic.Vsa. Diese müssen Sie also zuerst zu den Referenzen des Anwendungsprogramms hinzufügen. Alle Typen und Klassen, die für's Scripting benötigt werden, stehen dann zur Verfügung. (Die Beispiel-Implementierung finden Sie in der Datei 'executescript.cs' in Form der Klasse CExecuteScript.

• Das Backbone: IVsaSite

Dreh- und Angelpunkt des Scripting ist das IVsaSite Interface. Dieses Interface muss für den reibungslosen Scripting-Betrieb implementiert werden. Wenn Sie einmal einen Blick in die Dokumentation zu IVsaSite werfen, werden Sie eines schnell feststellen: Das Interface ist alles andere als leicht verständlich oder leicht zu durchschauen.

Interessanterweise braucht man für den einfachsten Scripting-Fall aber bei der Implementierung von IVsaSite gar nicht viel zu tun: Zwar sind Methoden zu implementieren, aber sonderlich viel tun müssen diese nicht. Im Wesentlichen geht es um die fünf Methoden, wie in Listing *6* zu sehen.

Bei *GetCompiledState()* geht es um das Speichern einer bereits übersetzen Assembly. Für das Beispiel ist das ohne Belang - die Skripte werden einfach jedes Mal neu übersetzt. (Wenn Sie sich dafür interessieren, wie man kompilierten Quellcode in Form von Assemblies auf der Festplatte oder sonst wo ablegt, dann sollten Sie sich die Beiträge über .NET-Compiler an anderer Stelle in diesem Sonderheft durchlesen - dort wird das genauer beschrieben.)

Die Methode *GetGlobalInstance()* wird von der Engine aufgerufen, wenn der Skript-Programmierer in seinem Skript eine 'globale' Variable verwendet hat. Das ist zum Beispiel für den Zugriff auf das Objektmodell Ihrer Anwendung praktisch.

Angenommen, Sie haben ein ganz einfaches Objektmodell. Es gibt darin ein Anwendungsobjekt. Dieses Anwendungsobjekt enthält das bereits bekannte Buchkatalog-Objekt und dieses wiederum enthält die einzelnen Bücher. Nun ist es für Skripte sehr hilfreich wenn man

dort einfach *Anwendung.Katalog* schreiben kann, um per 'Anwendung' auf das Katalogobjekt zugreifen zu können. Dabei stellt sich aber natürlich die Frage: Woher kommt die Variable 'Anwendung', die eine Referenz auf das Anwendungs-Objekt darstellt? Genau: aus *GetGlobalInstance()*.

Die Engine ruft diese Methode auf, um einen im Skript verwendeten 'globalen' Namen gegen eine in der Anwendung bekannte Instanz aufzulösen. Im Beispielprogramm wird das aber aus Gründen der Einfachheit nicht gemacht – und darum liefert *GetGlobalInstance()* dort auch immer 'null': Es gibt keine 'globalen' Instanzen.

GetEventSourceInstance() ist ähnlich geartet wie *GetGlobalInstance()* - nur werden hier eben Objekte aufgelöst, die Ereignisse ausgelöst haben. Auch das ist im Beispiel nicht vorgesehen - und also wird auch hier immer 'null' geliefert.

Mit der Notify()-Methode kann die Engine dem Script-Host (also der Anwendung) Nachrichten übermitteln. Das kann man aber für's erste getrost ignorieren, und so tut auch diese Methode rein gar nichts.

Schließlich ist das noch *OnCompilerError()*. Hier bietet es sich allerdings an, ein bisschen Arbeit zu investieren. Die Methode wird aufgerufen, wenn beim Übersetzen des Skripts ein Fehler aufgetreten ist.

Was die Methode mit dieser Information tut, ist ihre Sache. Ihr Rückgabewert teilt der Engine mit, ob Interesse an weiteren Compiler-Fehlern besteht oder nicht. Dabei ist ein Compiler-Fehler ein Fehler, den der Compiler im Quellcode entdeckt hat und nicht etwa ein Fehler im Compiler - das versteht sich aber vermutlich von selbst.

Im Beispielprogramm wird die Fehlermeldung einfach per *MessageBox.Show()* angezeigt. Der Skript-Programmierer weiß dann, dass etwas nicht in Ordnung ist und kann seinen Quellcode entsprechend ausbessern.

Soweit zu IVsaSite - bleibt die Frage: Wie führt man nun das Skript aus? Das ist Aufgabe des Script-Hosts - und wird im Beispielprogramm in der Methode Exec() besorgt.

ExecQ erhält einfach einen 'string' als Parameter. Dieser String enthält den auszuführenden Script-Code.

Bevor der Code ausgeführt werden kann, müssen aber einige Dinge (vorzugsweise richtig) initialisiert werden. Im Wesentlichen handelt es sich dabei um eine IVsaEngine – denn die wird sich letzten Endes tatsächlich darum kümmern, dass Quellcode übersetzt und ausgeführt wird.

Alle Skripts werden dabei in einem bestimmten Kontext ausgeführt, und der benötigt einen eindeutigen Namen. Im Beispiel wird dieser Namen in einer string-Variable abgelegt, wie in Listing 7 zu sehen ist.

Hier wird zunächst der *Root Namespace* festgelegt und dann eine neue Instanz der Script-Engine erzeugt. Die will zusätzlich zum *Root Namespace* auch noch eine eindeutige URL haben - dazu kann man zum Beispiel einfach den Namen der Anwendung verwenden.

Im nächsten Schritt benötigt die Engine eine Referenz auf eine IVsaSite-Implementierung. Die haben Sie in Form der zuvor besprochenen Funktionen bereits vorliegen - es kann also einfach *this* übergeben werden.

Dann wird die Engine initialisiert. Das passiert durch den Aufruf von *InitNew()*.

LISTING 6:

```
void IVsaSite.GetCompiledState(out Byte[] pe, out Byte[]
➥debugInfo)
{
  pe = debugInfo = null;
}

object IVsaSite.GetGlobalInstance(string name)
{
  return null;
}

object IVsaSite.GetEventSourceInstance
➥(string itemName, string eventSourceName)
{
  return null;
}

void IVsaSite.Notify(string notify, object info)
{
}

bool IVsaSite.OnCompilerError(IVsaError e)
{
  MessageBox.Show( e.Description );
  return false; // keine weiteren fehler mehr
}
```

LISTING7:

```
private IVsaEngine m_engine;
private string m_strRootNameSpace;

Zunächst ist der IVsaEngine zu initialisieren:

m_strRootNameSpace = "samplescript";
m_engine = new Microsoft.VisualBasic.Vsa.VsaEngine();
m_engine.RootMoniker = "TW://script";
m_engine.Site = this;
m_engine.InitNew();
m_engine.RootNamespace = m_strRootNameSpace;
m_engine.GenerateDebugInfo = true;
```

LISTING8:

```
// system.dll
refitem = (IVsaReferenceItem)items.CreateItem("system.dll",
➥VsaItemType.Reference,VsaItemFlag.None);
refItem.AssemblyName = "system.dll";
```

• Assemblies laden

Damit der Skript-Programmierer später Zugriff auf die benötigten Datentypen aus .NET hat, muss die Engine über diese informiert werden. Das geschieht dadurch, dass Sie so genannte *IVsaReferenceItems* auf die gewünschten DLLs erzeugen. Diese Items werden dann in die Collection der dem Engine bekannten Items gestellt:

```
IVsaItems items = m_engine.Items;
IVsaReferenceItem refitem;
```

Um zum Beispiel Zugriff auf die 'string'-Klasse zu gewähren, muss die 'System.dll' referenziert werden können.

Dazu erzeugen Sie ein entsprechendes IVsaReferenceItem(Listing 8). Wenn Sie nun beispielsweise auch noch den Zugriff auf die Klassen aus der system.drawing.dll ermöglichen wollen, ist auch diese DLL hinzuzufügen (Listing 9).

Das muss für alle Assemblies passieren, die der Skript Programmierer hinterher benutzen können soll.

Es bietet sich also an, die eignen Script IDE mit einem Mechanismus zu versehen, der es erlaubt, Referenzen hinzuzufügen. Wie Sie das tun, ist aber Ihrer Kreativitätüberlassen.

Das Beispiel geht einfachden Weg, einige wenige DLLs fest zu verdrahten und bietet keine Möglichkeiten des Zugriffs auf weitere DLLs - zumindest nicht, ohne das Beispielprogramm neu zu übersetzen.

• Eigene Typen zur Verfügung stellen

Schließlich ist der wichtige Moment gekommen: Die eigenen Typen sollen ebenfalls der Engine zur Verfügung gestellt werden. Das geht aber nicht wesentlich anders, als bei den Microsoftschen: Alles, was man benötigt, ist die Assembly, die diese Typen enthält. Eine Referenz auf diese Assembly erhält man beispielsweise, indem man *Assembly. Get-Assembly()* verwendet - und einen Typ aus der gewünschten Assembly als Parameter übergibt. Danach wird die Assembly genau so zur Engine hinzugefügt, wie Sie das schon von den vorigen Beispielen kennen (Listing 10). Nun muss die Engine noch wissen, welchen Scriptcode sie übersetzen soll. Dazu braucht man ein IVsaCodeItem. Dieses wird mit dem String, der den Quellcode enthält initialisiert (siehe dazu Listing 11).

LISTING 9:

```
// system.drawing.dll
refitem = (IVsaReferenceItem)items.CreateItem("system.drawing.dll",
➥VsaItemType.Reference, VsaItemFlag.None);
refItem.AssemblyName = "system.drawing.dll";
```

LISTING 10:

```
Assembly asm = Assembly.GetAssembly( typeof(Lights));
refitem = (IVsaReferenceItem)items.CreateItem( asm.Location,
➥VsaItemType.Reference, VsaItemFlag.None);
refItem.AssemblyName = asm.Location;
```

LISTING 11:

```
IVsaCodeItem codeItem = null;
codeItem = (IVsaCodeItem)items.CreateItem("Script",
   VsaItemType.Code, VsaItemFlag.None);
codeItem.SourceText = strCode;
```

LISTING 12:

```
if(! m_engine.Compile()) return false;
m_engine.Run();
```

LISTING 13:

```
Assembly   assem = m_engine.Assembly;
Type       type = assem.GetType( m_strRootNameSpace + ".Script") ;
if( type == null)
{
  MessageBox.Show( "Module named 'Script' is missing. ");
  return false;
}
```

LISTING 14:

```
Methodinfo method = type.GetMethod("Main");
if( method == null)
{ // fehler...
```

LISTING 15:

```
module Script
 public Sub Main()
   dim lf as new Scripting.lightform()
   lf.Show()
   lf.Light = Scripting.Lights.Green
 end Sub
end module
```

Und das war's eigentlich schon. Nun kann es ans Ausführen des Skripts gehen. Vor das Ausführen hat die IVsa-Engine aber das Übersetzen gestellt, doch das ist leicht zu haben (Listing 12).

Nun ist der Code übersetzt und wartet darauf, ausgeführt zu werden. Dazu brauchen Sie einen Einstiegspunkt ins Programm. Beim Beispiel ist das so gelöst, dass der Eintrittspunkt grundsätzlich die Methode 'Main' im Modul 'Script' ist. Das muss nicht so sein - macht es aber für's Beispiel deutlich einfacher.

Zunächst wird deshalb das Modul Script im übersetzten Code gesucht (Listing 13). Enthält das Script das benötigte Modul, kann nach dem wirklichen Eintrittspunkt in Form der Funktion Main gesucht werden (Listing 14).

Auch hier gibt das Beispiel eine Fehlermeldung aus, so diese Methode nicht vorhanden ist. Ist sie das aber, dann steht dem Ausführen des Skriptes nichts mehr im Wege:

```
method.Invoke(null, null);
```

Und das war alles - Ab sofort kann geskriptet werden. Ein Beispiel-Skript, das zunächst eine Instanz der Ampel-Klasse erzeugt und dann die Ampel auf Grün setzt, sieht aus wie in Listing 15 dargestellt.

Im Beispielprogramm finden Sie übrigens noch eine kleine Textbox, mit der Sie das Skript auch eingeben können: Etwas Luxus muss ja schließlich sein.

In diesem Beitrag haben Sie erfahren wie Sie Ihre eigene Anwendung um einen Scripting-Engine erweitern und wie Sie diesen Engine in die Lage versetzen auch die Klassen zu verwenden, die Sie selbst definiert haben.

Ganz nebenher haben sie eine rudimentär brauchbare Ampel-Klasse programmiert. Aber nicht vergessen: Immer nur bei 'Grün' fahren. © U R

Öffentliche Methoden auf fremden Rechnern

Webservices verstehen und nutzen

XML-Webdienste sind und waren eines der Schlagworte mit denen .NET durch die Presse geistert. Dummerweise ist es so, dass praktisch niemand wirklich weiss, was ein Web-Service eigentlich ist. Der Grund dafür ist einfach - es gibt nicht sonderlich viele davon, und die sind auch noch gut versteckt.

THOMAS WÖLFER

In diesem Beitrag werden Sie einen eigenen Client für einen doch ziemlich hilfreichen Web-Service schreiben: Die Google-Search API. Zunächst einmal muss mit einem grundlegenden Missverständnis aufgeräumt werden, das von der etwas ungeschickten Namensgebung 'Web-Service' herrührt: Web-Services haben eigentlich mit dem Web nicht sonderlich viel zu tun. Der Grund für die Bezeichnung Web-Service ist einzig und allein die Tatsache, dass die Kommunikation zwischen dem WebService Client und dem WebService-Anbieter über HTTP stattfindet.

Dabei kommt aber kein Browser ins Spiel, und auch die Auslieferung erfolgt nicht wirklich über einen Web-Server. Das HTTP-Protokoll wird nur deshalb verwendet, weil man damit relativ leicht Daten durch Firewalls durchreichen kann. Letzten Endes geht es bei WebServices nämlich darum, dass ein Client-Programm Funktionen aufruft, die auf einem (entfernten) Server-Rechner ausgeführt werden, mit dem das Client-Programm per TCP/IP verbunden ist. Das entspricht im Wesentlichen dem ganz normalen unter Windows bekannten RPC-Aufrufen - sei es über OLE, über Automation oder über normales RPC. Diese Aufrufe haben aber immer dann Probleme, wenn sich die beteiligten Rechner hinter Firewalls befinden, denn sie nutzen mehr oder minder proprietäre Protokolle und auch immer eigene TCP-Ports.

DIE SEARCH-API ist bei Google auf einer eigenen Seite beschrieben. Allerdings braucht man diese Beschreibung nicht wirklich, denn das Visual Studio besorgt sich alle Informationen mehr oder minder automatisch.

So etwas lassen Netzwerkadministratoren aber ungern zu - je mehr Dienste an irgendwelchen Ports horchen und je mehr proprietäre Protokolle und Anwendungen dabei im Spiel sind, um so unsicherer wird das Netz. Am liebsten haben es die Administratoren, wenn einfach alles zu ist - nur macht das die Vernetzung halt ein bisschen sinnlos.

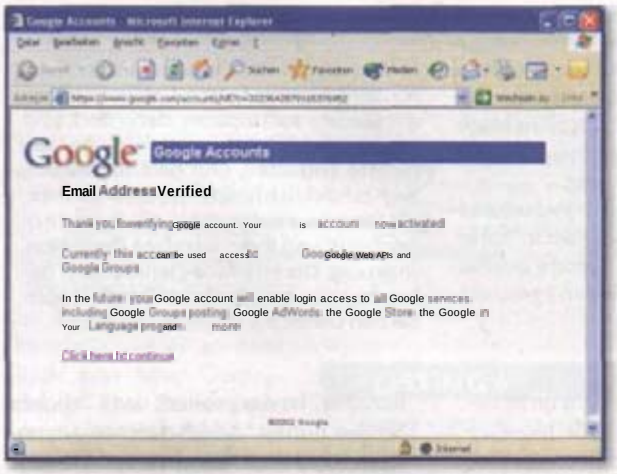
Eine Verbindungsart kann aber praktisch unter keinen Umständen vom Administrator abgekappt werden, und das sind http-Verbindungen über Port 80. Das ist genau das, was sich die Webservices zu Nutzen machen.

Dabei wird die Anfrage auf dem Client in XML umgewandelt. Die XML-Daten beschreiben, welcher Dienst angefragt werden soll und wie die Parameter der zum Dienst gehörenden Funktion aussehen (also deren Typ, deren Anzahl und deren Werte). Diese Daten werden dann per http an den Server übertragen, auf dem der Dienst läuft.

Der Dienst nimmt dann die Anfrage an, dekodiert die XML-Daten und führt seine Arbeit aus. Daraus resultieren dann auch irgendwelche Daten. Der Server nimmt dann diese Daten und verpackt sie ebenfalls in XML. Dieses XML wird dann dem Client - erneut per http - übermittelt. Dort werden dann die XML-Daten ausgelesen, und der Client hat seine Antwort.

• Google - ein Beispiel-Service

Ein Beispiel für einen solchen Dienst ist die Search-Engine Google. Google bietet nicht nur die normale Webseite zum Suchen an, man kann den Suchmotor auch per Web-Servcice befragen. Das bedeutet, dass man einen eigenen Client schreiben kann und dass dieser dann ver-



ZUM SUCHEN PER WEBDIENST braucht man einen Google-Account. Der ist aber kostenlos.

schiedene Google-Funktionen benutzen kann, ganz ohne dass man dafür einen Webbrowser braucht.

Nachdem eine dieser Funktionen auch der Google Spell-Checker ist, kann man auf diese Art zum Beispiel auch eine Rechtschreibkorrektur in einer Anwendung implementieren. Das Beispielprogramm tut das aber nicht. Statt dessen implementiert es eine einfache Suche mit einem eingebauten Browser, der direkt die Suchresultate anzeigen kann.

• XML oder was ?

Dabei stellen sich natürlich recht schnell eine ganze Menge Fragen: Wie erzeugt man denn das notwendige XML? Wie wird der Dienst konkret aufgerufen, und woher kennt man die richtigen Parameter? Wie kodiert man die - und wie dekodiert man die Antworten richtig? Schließlich ist das alles nicht sonderlich einfach, immerhin müssen die unterschiedlichsten möglichen Encodings und Sprachen berücksichtigt werden, das Format der gelieferten XML-Datei ist - von der Tatsache das es sich um XML handelt abgesehen — eigentlich unbekannt und möglicherweise gibt es vielleicht sogar Byte-Ordering Probleme zwischen dem Client und dem Server. Trotz der Problematik ist die Sache ganz einfach:

Man muss sich nämlich um nichts dergleichen kümmern. Der Aufruf eines Webservices sieht in C# nicht anders aus als der Aufruf einer öffentlichen Methode aus einer bekannten Klasse.

Und bekannt gemacht wird der Webdienst ebenfalls mit bekannten Methoden - man fügt eine Referenz zum Projekt hinzu und hat dann eine Klasse, die man instanzieren und verwenden kann. Wer sich noch daran erinnern kann, wie unglaublich aufwändig es war, einen RPC-Aufruf mit VC++ 6 oder 5 durchzuführen - gar nicht zu reden von dem benötigten MIDL-Code - und wie fehleranfällig die Sache aufgrund der vielen Handarbeit war, der wird bei WebServices mit C# vermutlich seine helle Freude haben.

XML sieht man keines, MIDL Code gibt's nicht - aber dafür gibt es IntelliSense in der IDE auch für die Klassen aus dem Webservice: Ganz egal, dass der auf einem völlig anderem Rechner stammt. Die Hilfe beim Tippen, die Parameterlisten und die Tooltips zu den Methoden sind bei WebServices nicht von denen bei Diensten normaler lokaler Klassen zu unterscheiden. Das Publizieren eines WebServices ist übrigens ähnlich einfach.

DAS SUCHPROGRAMM IN AKTION: Suchworte, Suchresultate und die gefundene Seite - alles in einem Interface.

• Zeit für's Programmieren

Zunächst braucht man ein ganz normales C#-Projekt vom Typ 'Windows Application'. Die automatisch erzeugte Form bekommt dann ein paar Kontroll-Elemente und zwar die folgenden:

- Eine Textbox für die (kostenlose) Google-Lizenz.
- Eine Textbox für den gesuchten Text und einen 'go'-Button der die Suche startet
- Eine ListView mit zwei Spalten für die Suchergebnisse. Die erste Spalte nimmt den Titel des Suchergebnisses auf, die zweite Spalte wird den beschreibenden Text, den Google liefert, aufnehmen.
- Und schließlich noch ein Explorer-Control. Das müssen Sie unter Umständen zuvor erst zur Tool-

LIZENZ ZUM SUCHEN

Google bietet seit einiger Zeit nicht nur die Webseite zum Suchen an: Sie können über WebServices den Google-Suchmotor, sowie einige andere Google-Funktionen auch, programmatisch verwenden. Dafür gibt es momentan eine recht einfache Lizenzpolitik: Sie benötigen eine gültige Lizenz und können damit bis zu 1000 Anfragen pro Tag durchführen.

Diese Lizenz ist für den eigenen Gebrauch momentan kostenlos und kann auf Googles Webseite angefordert werden. Dort gibt es auch noch zusätzliche Informationen über den WebService und was er alles kann.

Um die Lizenz zu erhalten, gehen Sie auf Googles Webserver unter **HYPERLINK „http://www.google.com/apis**

Dort finden Sie drei Schritte, die Sie nach und nach durchführen können. Der erste Schritt ist optional - Ob Sie das Entwickler-Kit herunterladen oder nicht, spielt keine Rolle, denn alle Informationen, die Sie für die Arbeit mit dem Dienst innerhalb von Visual Studio brauchen, besorgt sich das Visual Studio automatisch. Der zweite Schritt ist aber nicht optional, und dem sollten Sie nun folgen. Jetzt müssen Sie einen Google Account anlegen. Dazu müssen Sie im Wesentlichen eine E-Mail-Adresse angeben. An diese Adresse sendet Google dann prompt den Lizenz-Schlüssel, und dieser Schlüssel ist einfach nur ein kurzer String. Im Programm geben Sie diesen String dann in der dafür vorgesehenen Textbox an - und schon kann die Suche losgehen.

box hinzufügen, denn es handelt sich dabei um ein COM Objekt das nicht von Haus aus in der Toolbox enthalten ist. Wie Sie die Toolbox erweitern, ist ebenfalls in diesem Sonderheft ausführlich beschrieben.

Damit Sie Googles WebService überhaupt benutzen können, brauchen Sie einen gültigen Lizenz-Schlüssel. Dieser ist kostenlos und ermöglicht es Ihnen, jeden Tag bis zu 1000 Suchanfragen auf Google loszulassen. Wie Sie diesen Schlüssel erhalten, erfahren Sie im Kasten 'Lizenz zum Suchen'. Den sollten Sie nun einmal kurz durchlesen, denn dann könnten Sie den Schlüssel schon in Ihrer Inbox liegen haben, wenn Sie das Beispielprogramm in Kürze zum ersten Mal übersetzen.

Wie bereits erwähnt, werden Dienste von WebServices Servern genau so referenziert, wie Sie auch andere Klassen referenzieren: Sie fügen einfach eine Referenz zu Ihrem Projekt hinzu. Einen kleinen Unterschied gibt es aber doch - die Referenz landet nicht im normalen 'References'-Ordner sondern in einem eigenen 'WebReferences'-Order. Der wird von Visual Studio bei Bedarf angelegt.

Dazu klicken Sie zunächst mit der rechten Maustaste auf das Projekt und wählen dann den Befehl *Add Web References* (Web Referenz hinzufügen). Visual Studio öffnet den Dialog zum Hinzufügen solcher Referenzen. Damit sie eine Referenz zu einem WebService hinzufügen können, müssen Informationen über diesen Service herunter geladen werden - Sie müssen also Online sein. Natürlich nicht für die gesamte Entwicklungszeit, aber um die benötigten Informationen herunterzuladen.

Um Web-Dienste zu finden, können Sie das UDDI-Verzeichnis verwenden, das im Dialog angezeigt wird. Dabei gestaltet sich die Suche aber ein wenig kryptisch. Man kann wirklich nicht behaupten dass das vorliegende Interface auch nur ansatzweise leicht verständlich gestaltet ist. Wenn man aber die URL zu den Dienst-Dateien schon kennt, dann wird die Sache einfacher. Und Sie kennen die URL gleich - denn die ist hier einfach abgedruckt: HYPERLINK „http://api.google.com/GoogleSearch.wsdl"

Diese URL tragen Sie in der Adressleiste des Dialogs ein- Visual Studio lädt dann die zugehörigen Informationen herunter. Ist das passiert, dann drücken Sie den Button 'Add Reference' am unteren Rand der Dialogbox und die neue Web-Refenrenz wird im passenden Ordner innerhalb des Projekt-Fensters angezeigt.

EINESCHLICHTE OBERFLÄCHE ZUM SUCHEN: Die Ergebnisse werden in der Liste angezeigt. Das Explorer-Fenster zeigt dann die in der Liste ausgewählte Seite an.

Dabei hat sie aber einen etwas merkwürdigen Namen - den können Sie einfach ändern. Nennen Sie das Element zum Beispiel 'google', dann schreibt sich der Rest einfacher hin. Das war im Prinzip schon alles: Sie haben nun eine Klasse namens 'google', die Sie ganz normal innerhalb Ihres Programms verwenden können. Dazu noch ein kleines Beispiel.

Erzeugen Sie zunächst einen Event-Handler für das Click()-Ereignis auf dem 'Go'-Button. Hier starten Sie dann die Suche.

Dazu brauchen Sie zunächst ein Objekt vom Typ 'GoogleSearchService'. Das legen Sie an und lesen bei der Gelegenheit direkt noch die Texte aus den bei-

DIE WEB SERVICE DIRECTORIES

Der Microsoftsche UDDI Dienst stellt eine Sammlung von Webdiensten dar. Dabei sind die Dienste in Kategorien eingeteilt und stellen die unterschiedlichsten Informationen zur Verfügung.
Leider ist das Interface ein wenig unhandlich: Immer wieder landet man in 'toten' Bereichen, die gar keine Dienste beinhalten, und immer wieder fehlen irgendwelche Informationen.

Am besten eignet sich noch die Sammlung 'VS Web Service Search Categorization', um Dienste aufzuspüren, denn dort sind auch tatsächlich Informationen über die Dienste enthalten, und dort finden sich auch tatsächlich funktionierende Dienste. Die Dienste werden dabei mit Ihrem 'AccessPoint' und Ihrer 'Interface Definition' angezeigt. Die Interface-Definition ist dabei das, was man braucht: Damit binden Sie den Dienst ins Visual Studio ein

GOOGLE-DIENSTE IN DER ÜBERSICHT

Der GoogleSearchService bietet noch eine ganze Menge an weiteren Funktionen an, als die einfache Suche. So können Sie zum Beispiel auch den Google Spell-Checker befragen, Statistiken über Sites ermitteln, oder die Größe einer zwischengespeicherten Seite in Bytes erfragen. Stellt sich die Frage: Wie erfährt man, was der Dienst alles anbietet?
Dafür gibt es zwei Möglichkeiten: Eine pragmatische und eine spassige. Die pragmatische Lösung ist logischerweise die, dass man einfach die zum Development Kit gehörende Dokumentation liest, denn da sind alle Möglichkeiten beschrieben.
Die etwas andere Methode ist die, Visual Studios IntelliSense-Technik zu verwenden. Dazu brauchen Sie nur eine Variable vom Typ GoogleSearchService, tippen hinter dem Variablen-Namen einen Punkt und warten ganz kurz: Visualstudio öffnet dann eine Liste aller zur Verfügung stehenden Funktionen einschließlich derer Parameter. Zwar haben diese Funktionen dann keine Beschreibung, aber schon anhand der Namen und der Parameter ist es leicht zu ermitteln, was welche Funktion tut. In vielen Fällen kann das durchaus schneller sein als die Dokumentation zu lesen - was man natürlich trotzdem irgendwann tun sollte. Wer will kann aber außerdem auch die .WSDL Datei aus dem Projekt-Fenster anklicken. Visual Studio öffnet dann den XML-Editor und zeigt diese Datei an. Darin sind alle Informationen über den Dienst, dessen Funktionen und deren Parameter enthalten. Nachdem das ganze in Form von XML vorliegt, kann man die Sache auch recht einfach lesen und erhält so einen Überblick über die angebotenen Services.

den Feldern für den Lizenz-Schlüssel und den Suchbegriff aus (Listing 1).

Den GoogleSearchService fragen Sie nun einfach nach dem Suchbegriff – und der Service liefert daraufhin ein GoogleSearchResult-Objekt zurück (Listing 2).

Wenn Sie das Programm nun übersetzen und die Suche starten, gibt es aber gleich eine böse Überraschung: Die Suchanfrage wirft eine Exception. Es empfiehlt sich also, die Suchanfrage in einen Exception-Handler einzubetten, der sich darum kümmern kann, die Informationen über die Exception anzuzeigen (Listing 3).

Nochmal übersetzt und gestartet, gibt Ihnen die Message-Box Auskunft über das Problem: Sie haben keinen Lizenz-Schlüssel! Wenn Sie den Kasten 'Lizenz zum Suchen' durchgelesen und entsprechend gehandelt haben, dann befindet sich dieser Schlüssel mittlerweile in Ihrer E-Mail-Inbox. Von dort kopieren Sie den Schlüssel also einfach in das vorgesehene Feld auf der Form – und die nächste Suchanfrage läuft problemlos durch: Herzlichen Glückwunsch - Sie haben Ihren ersten WebService benutzt.

Jetzt bleibt Ihnen noch, die Anzeige der Resultate zu programmieren. Dazu iterieren Sie einfach über die einzelnen Elemente im von Google gelieferten SearchResult und zeigen diese in der ListView an (Listing 4).

Die zum Suchresultat gehörende URL speichern Sie im 'Tag' Member des Listview Items. Das können Sie dann in einem passenden Handler auslesen und das Explorer-Control dann dazu bewegen, zur passenden Seite im Web zu navigieren (Listing 5).

Eine Sache bleibt aber noch anzumerken: Die Resultate, die Google liefert, enthalten natürlich auch HTML-Elemente. Die eigenen sich aber nicht sonderlich gut für die Anzeige in einer Listview und müssen darum vorher rausgefiltert werden.

Im Beispiel tut das die Funktion StrpHTML(). Die verwendet dazu einfach das ungemein praktische Regular-Expression-Objekt aus der .NET-Klassenbibliothek (Listing 6).

In diesem Beitrag haben Sie erfahren, wie ungeheuer einfach die Verwendung von WebServices mit dem Visual Studio ist: Ein WebService wird einfach genau so verwendet, als wäre es eine Klasse aus der .NET-Klassenbibliothek oder eine eigene Klasse.

Das UDDI-Verzeichnis bietet dabei noch eine ganze Reihe an weiteren Web-Diensten an: Auch wenn das Interface dafür reichlich schauderhaft ist, lassen Sie sich nicht abschrecken – es lohnt sich, einmal darin herumzublättern.

© UR

LISTING 5:

```
private void listView1_SelectedIndexChanged(object sender, System.EventArgs e)
{
  if( this.listView1.SelectedItems.Count > 0)
  {
    object url = this.listView1.SelectedItems[0].Tag;
    object dummy = null;
    Cursor.Current = Cursors.WaitCursor;
    this.axWebBrowser1.Navigate2( ref url, ref dummy, ref dummy, ref dummy, ref dummy);
    Cursor.Current = Cursors.Default;
  }
}
```

LISTING 6:

```
private string StripHTML( string src)
{
 ..Regex rStrip = new Regex("<([^!>]([^>]|\n)*)>", RegexOptions.IgnoreCase);
return rStrip.Replace(src, "");
}
```

LISTING 1:

```
Google.GoogleSearchService s = new Google.GoogleSearchService();
string strKey = this.textBox1.Text;
string strQuery = this.textBox2.Text;
```

LISTING 2:

```
Google.GoogleSearchResult r =
s.doGoogleSearch ( strKey, strQuery, 0, 10,
➥false, "", false, "", "", "");
```

LISTING 3:

```
Google.GoogleSearchResult r = null;

try
{
  r = s.doGoogleSearch( strKey, strQuery, 0, 10, false, "",
  ➥false, "", "", "");
}
catch( Exception ex)
{
  MessageBox.Show( ex.Message, "Fehler...");
  return;
}
```

LISTING 4:

```
foreach( Google.ResultElement re in r.resultElements)
{
  string sTitle = StripHTML( re.title);
  string sUrl = re.URL;
  string sSummary = StripHTML( re.summary + " " +re.snippet);

  ListViewItem i = new ListViewItem( new string[] { sTitle, sSummary});
  this.listView1.Items.Add( i);
  i.Tag=sUrl;
}
```

Die Top-Tipps

C# optimieren

Der C#-Compiler ist ein optimierender Compiler, aber darauf sollte man sich nicht allein verlassen. Es ist sogar viel wichtiger, Algorithmen richtig einzusetzen und die vorhandenen Sprachfeatures zu verstehen, denn durch eine richtige Verwendung der Sprache lassen sich oft deutlich größere Performance-Verbesserungen herausholen, als das mit dem Optimizer möglich ist.

THOMAS WÖLFER

Wie auch in anderen Sprachen, gibt es einige Dinge, die man in C# zwar hinschreiben kann, die sich aber im laufenden Programm dann ganz fürchterlich rächen. In C++ fallen viele Programmierer damit auf die Nase, dass Sie aufgrund von copy-Konstruktoren immer wieder neue Kopien von Objekten anlegen - obwohl das gar nicht erwünscht ist. Um genau zu sein, gibt es noch jede Menge weiterer derartiger Probleme in C++ - eine Liste der häufigsten Fallstricke finden Sie im PC-Magazin Sonderheft 'C++programmieren'.

Bei C# sieht die Sache ein bisschen anderes aus. Das liegt primär daran, dass die Sprache schlicht und ergreifend einfacher ist als C++: Man hat nicht gar so viele Möglichkeiten, sich selbst ein Bein zu stellen wie beim älteren Bruder. Trotzdem geht das natürlich - ansonsten würde die Sache ja auch keinen Spaß machen. Damit Sie diese Fallstricke vermeiden und immer effizienten Code produzieren, gibt es diesen Beitrag.

• Boxing und Un-Boxing

Bei C# - oder, um genau zu sein, in der .NET-Laufzeitumgebung, gibt es das Konzept des Boxing und des Unboxing. Das brauchen Sie zum Beispiel dann, wenn Sie mit Code kommunizieren wollen, der nicht dem Garbage Collector unterliegt.

In solchen Fällen wird man meist sehr genau wissen, was passiert - es gibt aber auch Fälle, wo implizites Boxing vorgenommen wird und das hat teilweise recht heftige Folgen.

Angenommen, Sie haben eine Funktion, die im Wesentlichen den folgenden Aufbau hat:

```
void foo()
{
  int i = 123;
  i = 456;
  int j = i;
}
```

Dabei wird einer Integer ein Wert zugewiesen, dann wird der Wert verändert, und schließlich wird der Wert der Integer einer Integer-Variablen - also einem kompatiblen Typ - zugewiesen. So weit, so gut. Nun ist es aber so, dass man einen Integer-Wert unter Umständen gern als Objekt behandeln möchte - das geht ja schließlich auch ohne Probleme. Man könnte die Funktion also auch folgendermaßen formulieren:

```
void foo()
{
  int i = 123;
  object o = i;
  i = 456;
  int j = (int)o;
}
```

Auch hier wird zunächst der Integer initialisiert. Dann wird er aber einem 'object' zugewiesen - und das löst implizites Boxing aus: Schließlich ist die Integer ein Value-Type während ein 'object' eben ein Referenz-Type ist. Die Box ist also notwendig und wird auch bereitwillig vom Compiler erzeugt, ohne dass man das auf den ersten Blick sieht.

Dann wird Wert der Integer verändert, schließlich wird ein unboxing durchgeführt: Das geschieht über den Typecast in der letzten Zeile. (Dieser Cast würde übrigens eine Exception werfen, falls die Umwandlung nicht möglich wäre.)

Auch das sieht nicht weiter bedrohlich aus - und beide Funktionen führen formal genau die gleichen Operationen durch. Allerdings: Die zweite Variante braucht dafür fast die doppelte Zeit der ersten Variante.

Daraus ergibt sich eine einfache Regel: (Implizites) Boxing sollte man immer vermeiden, wenn man auf Performance bedacht ist. Das Speichern von

EINE HOCHAUFLÖSENDE ZEITSPANNE

Wenn Sie eigenen Testprogramme zum Messen der Performance Ihres Programms schreiben wollen, dann brauchen Sie zwei Dinge: einen hochauflösenden Timer, der auch Zeiten im Millisekunden-Bereich auflösen kann und eine Klasse, die Zeitunterschiede in dieser kleinen Größenordnung richtig darstellen kann. Beides gibt es unter Win32 - aber leider nicht in .NET. Darum müssen Sie die Interop-Services von .NET verwenden, um die Win32 Funktionalität verwenden zu können. Oder aber, Sie verwenden einfach die hier abgebildeten Klassen, die genau das tun.

Value-Typen in Referenz-Typen hat zwar seine Vorteile und Eleganz - aber eben auch dramatische Auswirkungen auf die Performance.

• Schleifen mit foreach und ohne

C# hat für das Iterieren über Collections das recht praktische *foreach*-Schlüsselwort. Wer aber glaubt, das sei nur so ein komisches Visual-Basic-artiges Konstrukt, das man halt optional verwenden kann, der irrt. Denn dieses komische Konstrukt ist zwar sehr praktisch - kostet aber Performance. Die 'natürliche' Schreibweise hat dabei eben auch ihre Probleme.

Angenommen, Sie haben eine Liste aus Integern in einer ArrayList und wollen darüber iterieren. Das können Sie auf drei leicht unterschiedliche Arten tun. Die erste Art ist die unter Verwendung von 'foreach':

```
void TestLoop1(ArrayList List)
{
  int id = 0;
  foreach (int i in List)
  {
    id = i;
  }
}
```

Hier wird ganz normal per 'foreach' iteriert, und bei jedem Durchlauf der Schleife wird der Wert 'i' aus der Liste der Variablen id zugewiesen. Das ist leicht zu lesen und sieht sehr praktisch aus. Allerdings, die folgende Variante ist deutlich schneller:

LISTING 1:

```
void TestLoop2(ArrayList List)
{
  int id = 0;

  for (int i = 0; i < List.
➥Count; i++)
  {
    id = (int)List[i];
  }
}
```

Auch bei dieser Variante wird iteriert, und auch hier wird der Wert ausgelesen und zugewiesen. Diese Variante der Funktion braucht aber weniger als die Hälfte der Laufzeit der ersten Variante. Auf einem Testrechner benötigte TestLoop1() bei 10.000 Durchläufen 1.63 ms, während die zweite Variante nur 0.77 ms für die gleiche Anzahl an Durchläufen benötigt. Wie man sieht, ist *foreach* zwar recht praktisch - aber bei Performance-orientierten Aufgaben lässt man besser die Finger davon. Noch schneller wird es natürlich mit einer dritten Variante, bei der die Anzahl der Elemente in der Liste nur noch einmal ausgelesen wird. Allerdings ist da der Performance-Gewinn nur noch gering:

LISTING 2:

```
// Variante mit einer Struktur
Structobject s = new StructObject();
s.x = 1;
inx x = TestStructObject(s);
int TestStructObject(StructObject s)
{
   int x = s.x;
   return x;
}

// Variante mit einer Klase
ClassObject c = new ClassObject();
c.x = 1;
int x = TestClassObject(c);
int TestClassObject(ClassObject c)
{
  int x = c.x;
  return x;
}
```

Die dritte TestLoop()-Funktion braucht 0.707 Sekunden und ist damit nur um Zehntel schneller als TestLoop2()

```
void TestLoop2 (ArrayList List)
{
  int id = 0;
  int count = List.Count;

  for (int i = 0; i < count; i++)
  {
    id = (int)List[i];
  }
}
```

Grundsätzlich ist es dabei natürlich so, dass die eigentliche Aufgabe in der Schleife unter Umständen deutlich aufwändiger ist als der Durchlauf selbst - ob die Handoptimierung durch das Entfernen von foreach wirklich Sinn macht, ist daher vom Einzelfall abhängig.

Was man daraus lernen kann ist, dass es sehr wohl einen Unterschied macht, welche Schreibweise man beim Iterieren verwendet: Das sollte einem bewusst und im Notfall als Optimierungsmöglichkeit zur Hand sein.

• Collections beschleunigen - Größen angeben

Die Collections-Klassen aus der .NET-Klassenbibliothek sind praktisch. Es gibt unter anderem Listen, Arrays, Stacks und Hashtables, teilweise auch Spezialisierungen davon, die für bestimmte Zwecke eingesetzt werden. Will man ein neues Objekt zum Beispiel vom Typ Hashtable verwenden, braucht man sich bloß eines zu erzeugen:

```
using System.Collections;
// code entfernt ...
void foo()
{
    Hashtable ht =
    ➥new Hashtable();
    // irgendwas damit tun
}
```

LISTING 3:

```
public class HighResolutionTimeSpan
{
  [System.Runtime.InteropServices.DllImport("KERNEL32")]
  private static extern bool QueryPerformanceFrequency
  ➥(ref long lpFrequency);

  public HighResolutionTimeSpan()
  {
    QueryPerformanceFrequency(ref m_Frequency);
  }

  public HighResolutionTimeSpan(long Ticks)
  {
    QueryPerformanceFrequency(ref m_Frequency);

    m_Miliseconds = 1000 * ((float)Ticks / (float)m_Frequency);
  }

  public override string ToString()
  {
  return String.Format("{0}", m_Miliseconds);
  }

      private long m_Frequency = 0;
      public long Frequency
      {
              get {return m_Frequency;}
      }

      float m_Miliseconds = 0;
      public float Miliseconds
      {
              get {return m_Miliseconds;}
      }
}
```

Hat man die Hashtable erst einmal, dann kann man sie beliebig erweitern - es gibt keine relevante Größenbeschränkung, denn die Klasse alloziert ganz nach Bedarf automatisch neuen Speicher, wenn sie welchen benötigt. Das ist praktisch - hat aber auch seine Nachteile und wie so oft, liegen auch die in der Performance. Daher gibt es verschiedene überladene Operatoren für die Collections, so dass meist auch eine Anzahl an Elementen bei der Konstruktion mit angegeben werden kann. Angenommen, Sie brauchen eine Hashtable aus Objekten und füllen diese auf:

```
for( int i=0; i<count; i++)
{
    ht.Add( i, new ___
}
```

Wenn Sie die Hashtable nun ohne Parameter - also ohne Größenangabe - erzeugt haben, dann gelingt das zwar, ist aber etwa 25 Prozent langsamer, als wenn Sie die Größe von vorneherein mit spezifizieren. Der Performance-Gewinn tritt dabei eigentlich in allen Collection-Klassen ein.

Als Regel lässt sich daraus ableiten, dass es grundsätzlich Sinn macht, die Anzahl der Elemente, die in einer Collection aufgenommen werden sol-

LISTING 4:

```
ein hochauflösender Timer mit den Interop-Diensten
public class PerformanceTimer
{
 [System.Runtime.InteropServices.DllImport("KERNEL32") ]
      private static extern bool QueryPerformanceCounter
➥(reflonglpPerformanceCount) ;

       [System.Runtime.InteropServices.DllImport("KERNEL32")]
      private static extern bool QueryPerformanceFrequency
➥(reflong lpFrequency) ;

      private long m_StartCount = 0;
      private long m_StopCount = 0;
      private long m ElapsedCount = 0;

      publicPerformanceTimer()
      {
            m_Frequency = 0;
            QueryPerformanceFrequency(refm_Frequency) ;
      }

      public void Start()
      {
            m_StartCount = 0;
            QueryPerformanceCounter(refm_StartCount) ;
      }

      public void Stop()
      {
            m_StopCount = 0;
            QueryPerformanceCounter(refm_StopCount) ;
            m ElapsedCount = m_StopCount - m_StartCount;
      }

      public void Reset()
      {
            m_StartCount = 0;
            m_StopCount = 0;
            m_ElapsedCount = 0;
      }

      public long Elapsed
      {
            get {return m_ElapsedCount;}
      }

      public float Seconds
      {
            get {return((float)m_ElapsedCount / (float)m_Frequency);}
      }

      private long m_Frequency = 0;
      public long Frequency
      {
            get {return m_Frequency;}
      }

      public HighResolutionTimeSpan ElapsedTime
      {
            get {return new HighResolutionTimeSpan(m_ElapsedCount);}
      }
}
```

len, bei der Erzeugung des Collection-Objektes mit anzugeben. Nachdem die Elemente ja oft aus einer anderen Sammlung kopiert werden, ist deren Anzahl bekannt - und daher ist die Größenangabe bei der Erzeugung kein Problem. Besonders nicht unter dem Eindruck der Tatsache, dass die Sache dadurch deutlich schneller wird.

• Exceptions: die Performance-Killer

Wie schon Win32, hat auch .NET ein eingebautes Modell für die Behandlung von Ausnahmefällen: Man kann Exceptions 'werfen' und einmal geworfene 'fangen' und behandeln. Das ist ungeheuer praktisch und macht einem das Leben an vielen Stellen leichter. Trotzdem sollte man aufpassen, wo man Exceptions verwendet - und wo nicht.

Vor allem Programmierer mit einem breiten Hintergrund in C++ werden relativ leicht verleitet sein, Exceptions als Standard-Mechanismus für die Fehlerbehandlung einzusetzen. Dabei liefern alle Funktionen keinerlei Return-Werte mehr, sondern werfen nur noch Exceptions, die dann mit zusätzlichen Informationen ausgestattet sein können. Praktisch - aber in dieser Anwendung grausam für die Performance eines Programms.

Wer statt Returncodes immer Exceptions verwendet, der sollte sich über eines im Klaren sein: Der Overhead einer geworfenen Ausnahme ist dramatisch und zwar sehr dramatisch.

Eine geworfene, gefangene und behandelte Exception braucht etwa 1000 (!) Mal so lange, wie das Liefern eines Fehlercodes.

Das ist ein sehr deutliches Zeichen für den richtigen Einsatzort von Exceptions. Der ist eigentlich schon anhand des Namens klar geregelt – man benutzt diese Methodik eben in Ausnahmesituationen, aber nicht für die normale Steuerung eines Programms, denn die 1000-fache Performance-Verschlechterung sollte einen wirklich unter allen Umständen davon abhalten.

• Strukturen versus Klassen: schwere Entscheidung

In C# können Sie sowohl Klassen als auch Strukturen verwenden – ganz ähnlich wie in C++ macht das bei simplen Elementen eigentlich keinen Unterschied. Anders als in C++ sind Strukturen und Klassen in C# aber semantisch deutlich unterschiedlich, denn Strukturen sind 'Value' Types, während Klassen Reference-Types sind.

Im Wesentlichen bedeutet das, dass Klassen auf dem Heap alloziert werden und Strukturen auf dem Stack. Das hat verschiedene Folgen, die sowohl die Performance beim Allozieren als auch die beim Initialisieren und Verwenden der Objekte beeinflussen.

Angenommen, Sie haben ein einfaches Element, das sowohl als Klasse als auch als Struktur ausgedrückt werden kann:

```
dass ClassObject
{
public int x;
}

struct StructObject
{
public int x;
}
```

Diese Objekte allozieren, initialisieren und verwenden Sie dann wie in Listing 2 beschrieben.

Funktional unterscheiden sich beide Varianten nicht - wenn Sie das aber oft durchführen, dann bildet sich schnell das folgende Bild: Das Allozieren der Klassen dauert deutlich länger, als das bei den Strukturen der Fall ist, und zwar fünf bis zehn Mal so lange! Die Initialisierung dauert in beiden Fällen gleich lang - aber die Verwendung der Klasse braucht gerade mal ein Drittel der Zeit, die für's Verwenden der Struktur benötigt wird.

Je öfter die Klasse also verwendet wird, um so stärker macht sich das bemerkbar und um so schneller wird der Overhead beim Allozieren unwichtig.

Daraus lässt sich eines leicht ableiten: Strukturen sollte man immer dann verwenden, wenn es darum geht, temporäre Objekte zu erzeugen, die nur für eine kurze Zeit verwendet werden.

Soll ein Objekt aber eine längere Lebenszeit haben und wird es oft verwendet - dann ist eine Klasse aus Sicht der Performance der richtige Weg der Implementierung.

• Strings richtig verwendet

Strings sind eine praktische Sache. Man braucht sie nahezu immer und in jedem Programm. Daher gibt es bei .NET auch eigens eine Klasse in der .NET Klassenbibliothek, die Strings kapselt. Die Klasse ist praktisch und tut mehr oder minder alles, was man mit Strings tun möchte - sie hat aber auch einen Nachteil.

Der liegt - wie sollte es anders sein - in der Performance. Dabei geht es im Wesentlichen nicht um normale String-Operationen, sondern vor allem um das Zusammensetzen von langen Strings. Jedes Mal, wenn im Code ein Statement wie das folgende auftaucht, bleibt dem Laufzeitsystem nicht viel anders übrig, als den String zu nehmen, ihn in einen entsprechend größeren Speicherbereich zu kopieren und dann zurückzuliefern:

```
string s = „andada"
s += „1234";
```

Im Normalfall tut das nicht weiter weh - geht es aber um lange Strings, dann schon. Deshalb gibt es bei .NET eigens eine StringBuilder genannte Klasse. Die ist dafür zuständig, lange Strings aus kürzeren zusammenzusetzen - und das macht der StringBuilder deutlich besser als die String-Klasse.

Wenn Sie beispielsweise den Text 'hello' 10000 Mal mit einem String verketten, dann dauert das etwa 800 Mal so lange, wie mit dem String-Builder. Wer also viel mit Texten arbeitet, der sollte auf jeden Fall einen Blick auf die String-Builder riskieren, denn durch deren Verwendung ist eine ganze Menge an Performance aus einem vorhandenen Programm herauszuholen.

In diesem Beitrag haben Sie eine erste Übersicht über verschiedene Optimierungsmöglichkeiten in C#-Programmen erhalten. Dabei ist eines klar geworden: Auch C# nimmt es einem nicht ab, darüber nachzudenken, welche Methode zu welchem Zeitpunkt die effizientere ist. Im Zweifelsfall ist es immer besser, kurz ein kleines Testprogramm zu schreiben und die zur Verfügung stehenden Möglichkeiten gegeneinander auszumessen. © UR