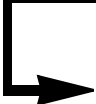


Klaus Prinz

Go To



VBA mit Excel 2000

 **ADDISON-WESLEY**

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist
bei der Deutschen Bibliothek erhältlich.

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

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

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

Umwelthinweis:

Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.

Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1
04 03 02 01 00

ISBN 3-8273-1525-5

© 2000 by Addison Wesley Verlag,
ein Imprint der Pearson Education Deutschland GmbH
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Alle Rechte vorbehalten

Einbandgestaltung: Barbara Thoben, Köln

Lektorat: Christina Mosler, cmosler@pearson.de

Korrektorat: Simone Burst, Großberghofen

Herstellung: TYPisch Müller, Gräfelfing

Satz: reemers publishing services gmbh, Krefeld

Druck und Verarbeitung: Schoder, Gersthofen

Printed in Germany



Inhaltsverzeichnis

	Vorwort.	11
	Die Icons in diesem Buch	14
1	Excel und Visual Basic for Applications.	15
	1.1 Objektkunst	16
	1.2 Auf ein Wort	19
	1.3 Übungsaufgabe	21
2	Die Entwicklungsumgebung.	23
	2.1 Projekt-Explorer	26
	2.1.1 Microsoft Excel Objekte	26
	2.1.2 Formulare	27
	2.1.3 Module	27
	2.1.4 Klassenmodule	27
	2.1.5 Das Kontextmenü des Projekt-Explorers	27
	2.2 Eigenschaftsfenster	29
	2.3 Codefenster	31
	2.3.1 Kontextmenü des Codefensters	33
	2.3.2 Kontextmenü des Codefensters im Haltemodus	39
	2.4 Direktfenster	40
	2.5 Lokal-Fenster	42
	2.6 Überwachungsfenster	43
	2.6.1 Bedeutung der Spalten	44
	2.6.2 Überwachungsausdrücke	45
	2.7 Objektkatalog	47
	2.8 Menüs	52
	2.8.1 Menü Datei	52
	2.8.2 Menü Bearbeiten	53
	2.8.3 Menü Ansicht	54
	2.8.4 Menü Einfügen	59

	2.8.5	Menü Format	61
	2.8.6	Menü Debuggen	61
	2.8.7	Menü Ausführen	63
	2.8.8	Menü Extras	64
	2.9	Symbolleisten	79
3	Einstieg	83	
	3.1	Lassen wir es piepen	84
	3.2	Etwas Zellzugriff	92
	3.2.1	Excel-Objekte – eine erste Annäherung	93
	3.2.2	Gezielter Zugriff auf eine Zelle	95
	3.2.3	Dynamischer Zugriff auf Zellen	103
	3.2.4	Ermittlung der letzten belegten Zeile	105
	3.2.5	Rückblick	108
4	Sprachelemente, die erste	111	
	4.1	Datentypen und Variablen	113
	4.1.1	Byte, Integer und Long	115
	4.1.2	Single und Double	115
	4.1.3	Currency	116
	4.1.4	Date	116
	4.1.5	Boolean	117
	4.1.6	String	117
	4.1.7	Object	117
	4.1.8	Variant	118
	4.1.9	Datenfelder	119
	4.1.10	Sonderzustände von Datentypen	120
	4.2	Modulooptionen	120
	4.2.1	Option Base 0 1	121
	4.2.2	Option Compare Binary Text	121
	4.2.3	Option Explicit	121
	4.2.4	Option Private Module	121
	4.3	Prozeduren	121
	4.3.1	Sub-Prozeduren	122
	4.3.2	Function-Prozeduren	123
	4.3.3	Übergabe von Argumenten	124
	4.4	Verzweigungen	126
	4.4.1	Verzweigungen mit If-Anweisungen	127
	4.4.2	Verzweigungen mit Select Case-Anweisungen	130
	4.5	Schleifen	132
	4.5.1	For-Next-Schleifen	132
	4.5.2	Die For-Each-Next-Schleife	133
	4.5.3	For-Next versus For-Each-Next	134
	4.5.4	Die Do-Loop-Schleife	135
	4.5.5	Verschachtelung von Schleifen	136

4.6	Vom Umgang mit Zeichenketten	137
4.6.1	Teilzeichenketten	138
4.6.2	Informationen über Zeichenketten	139
4.6.3	Split und Join	143
4.6.4	Zeichenketten verändern	144
4.6.5	Zeichenketten zusammenfügen	147
4.7	Formatierungen	148
4.7.1	Formatierung von Zeichenketten	148
4.7.2	Formatierung von Zahlen	149
4.7.3	Formatierung von Datums- und Zeitwerten	152
4.8	Numerische Funktionen	156
4.8.1	Konvertierungsfunktionen mit ganzzahligem Ergebnis	156
4.8.2	Konvertierungsfunktionen mit reellem Ergebnis	158
4.9	Datumsfunktionen	158
4.9.1	Standardfunktionen	158
4.9.2	Verzichtbare Datumsfunktionen	161
4.10	Ein- und Ausgabefunktionen	161
4.10.1	MsgBox-Funktion	161
4.10.2	InputBox-Funktion	165
5	Objekte	167
5.1	Objekte und ihre Elemente	169
5.1.1	Eigenschaften	170
5.1.2	Methoden	172
5.1.3	Ereignisse	174
5.2	Objekte und Auflistungen	176
5.2.1	Zugriff auf existierende Objektverweise	177
5.2.2	Erzeugen und Auflösen von Objektverweisen	179
5.2.3	Auflistungen	184
5.2.4	Kapselung, Polymorphie und Vererbung	187
5.3	Zugriff auf Zellen	189
5.3.1	Der Zugriff auf die Objekte	190
5.3.2	Die Kunst des Weglassens	191
5.3.3	Der Einsatz von Objektvariablen	192
5.3.4	Der Pflock in Zelle A1	194
6	Zugriff auf Excel-Objekte	195
6.1	Das Range-Objekt	197
6.1.1	Eigenschaften und Methoden im Überblick	198
6.1.2	Eigenschaften und Methoden unter der Lupe	201
6.2	Die Worksheets-Auflistung	237
6.3	Das Worksheet-Objekt	237
6.3.1	Eigenschaften und Methoden im Überblick	238
6.3.2	Eigenschaften und Methoden unter der Lupe	239
6.3.3	Ereignisse	251

6.4	Die Workbooks-Auflistung	262
6.4.1	Eigenschaften und Methoden im Überblick	262
6.4.2	Eigenschaften und Methoden unter der Lupe	262
6.5	Das Workbook-Objekt	270
6.5.1	Eigenschaften und Methoden im Überblick	270
6.5.2	Eigenschaften und Methoden unter der Lupe	270
6.5.3	Ereignisse	277
6.6	Das Application-Objekt	281
6.6.1	Eigenschaften und Methoden im Überblick	281
6.6.2	Eigenschaften und Methoden unter der Lupe	282
6.6.3	Ereignisse	291
6.6.4	Tabellenfunktionen in VBA	292
6.7	CommandBars und CommandBarControls	294
6.7.1	CommandBar erzeugen	296
6.7.2	FacelDs	297
6.7.3	Fazit	300

7 Sprachelemente, die zweite 301

7.1	Arrays	302
7.1.1	Array-Funktionen	303
7.1.2	Mehrdimensionale Arrays	305
7.1.3	Dynamische Arrays	306
7.1.4	Das Beispiel	308
7.2	Collections	314
7.3	Benutzerdefinierte Datentypen	317
7.4	Registry-Zugriffe	318
7.4.1	VB and VBA Program Settings	318
7.4.2	Sprachelemente für Registry-Zugriffe	321
7.4.3	Das Beispiel Registry.xls	323
7.5	Runden	330
7.6	Farben	332
7.6.1	Sprachelemente	332
7.6.2	Ein Beispiel	335
7.7	API	337
7.7.1	Der WinAPI-Viewer	337
7.7.2	Struktur einer API-Deklaration	339
7.7.3	Anwendername ermitteln	339
7.7.4	Pause im Programmablauf	340
7.7.5	Laufwerkstyp ermitteln	341
7.7.6	Ein Timer	343
7.7.7	Locale Settings	347
7.8	Dateioperationen	349
7.8.1	Operationen an Dateien	350
7.8.2	Ein Beispiel	353
7.8.3	Operationen in Dateien	356
7.8.4	Ein Beispiel	357

7.9	Rekursionen	359
7.9.1	Berechnung einer Fakultät	360
7.9.2	Bäume durchsuchen	361
7.10	Programmausführung unterbrechen	365
8	Dialoge.....	369
8.1	Funktionsweise von Dialogen	371
8.2	Login-Dialog	374
8.2.1	Erzeugen und Anordnen der Steuerelemente	376
8.2.2	Optische Korrekturen	378
8.2.3	Ergänzung des Formularfußes	380
8.2.4	Aktivierreihenfolge	382
8.2.5	Funktionale Aspekte des Login-Dialogs	383
8.2.6	Codierung des Login-Dialogs	386
8.2.7	Aufruf und Auswertung des Logins	390
8.3	Gemeinsamkeiten der Steuerelemente	391
8.3.1	Eigenschaften	391
8.3.2	Font-Objekt	397
8.3.3	Methoden	399
8.3.4	Ereignisse	403
8.4	UserForm	414
8.4.1	Eigenschaften	414
8.4.2	Methoden	414
8.4.3	Ereignisse	415
8.4.4	Eine kleine Spielerei mit einer UserForm	416
8.5	Label	422
8.6	TextBox	423
8.7	ComboBox	424
8.7.1	Eigenschaften	425
8.7.2	Methoden	427
8.7.3	Ereignisse	428
8.7.4	Besonderheit der ComboStyle-ComboBox	429
8.7.5	Wie sortiert man die Einträge einer ComboBox	430
8.8	ListBox	431
8.8.1	SingleSelect ListBoxes	431
8.8.2	MultiSelect ListBoxes	431
8.8.3	Anbindung an Tabellen	433
8.9	CheckBox	433
8.10	OptionButton	434
8.11	CommandButton und ToggleButton	434
8.12	SpinButton und ScrollBar	435
8.13	MultiPage und TabStrip	436
8.14	RefEdit	437
8.15	Steuerelemente in Tabellen	438

9	Fehlerbehandlung	441
9.1	Am Beispiel des Öffnens einer Arbeitsmappe	443
9.1.1	Fehlervermeidung	445
9.1.2	Methodiken der Fehlerbehandlung	448
9.1.3	Wie geht's bei uns weiter?	450
9.2	Noch etwas Theorie	456
9.2.1	Fehlerbehandlungshierarchie	456
9.2.2	Vorgehensweisen im ErrorHandler	458
9.2.3	Das Err-Objekt	459
9.2.4	Aktivieren von Fehlerbehandlungsroutinen	461
9.3	Typen von Fehlerbehandlungsroutinen	462
9.4	Zentrale Fehlerklasse clsError	466
9.4.1	Klasseninitialisierung	467
9.4.2	Fehlerausgabe	469
9.4.3	Fehlerlogbuch	470
9.4.4	Infoausgabe	472
9.4.5	Fazit	474
9.5	Fehlervermeidung	474
9.6	Gängige Fehler und ihre Behandlung	475
10	Klassenprogrammierung	485
10.1	Word.Application	486
10.2	Klassen	487
10.2.1	Klassenmodul erstellen	488
10.2.2	Elemente einer Klasse	490
10.3	Eine Datenklasse für den Login-Dialog	500
10.3.1	Die Tabelle User	501
10.3.2	Funktionale Überlegungen	501
10.3.3	Implementierung	502
10.3.4	Verwendung der Klasse	505
10.4	Fehlerbehandlung in Klassen	506
10.4.1	Ein Fehler im Initialize-Ereignis einer Klasse	507
10.4.2	Fehler im Terminate-Ereignis einer Klasse	509
10.4.3	Die Konstante vbObjectError	510
10.5	Das Rechnungsbeispiel	513
10.5.1	Die Aufgabenstellung	513
10.5.2	Die Tabellen	513
10.5.3	Klassenarchitektur	516
10.5.4	Klassencodierung	517
10.5.5	Codierung des Moduls shtInvoice	525
10.5.6	Zusammenfassung	528
10.6	DLLs – wann und wozu	529
10.6.1	Zur Abschreckung ein Beispiel	529

11	Applikationsdesign	533
11.1	Allgemeine Aspekte des Applikationsdesigns	535
11.2	Modularisierung und Auslagerung	535
11.3	Variablen	536
11.3.1	Datentypen und Präfixbildung	536
11.3.2	Objektvariablen	537
11.3.3	Namen von Standardvariablen	538
11.4	Kommentierung des Codes	539
11.4.1	Kommentarblock im Prozedurkopf	539
11.4.2	Kommentierung der Variablen und Konstanten	541
11.4.3	Kommentierung im Code	542
11.5	Anordnung von Prozeduren im Codemodul	542
11.6	Konstanten und Enumerationen	544
11.6.1	Spaltenzeiger	544
11.6.2	Enumerationen	545
11.7	Die Benutzerschnittstelle	546
11.7.1	Statusanzeige	547
11.7.2	ComboBoxes	548
11.7.3	ControlTipText	549
11.8	Einfache Parameterablage	549
11.9	Settings.xls	550
11.9.1	Die Klasse clsSettingsSheet	551
11.9.2	Die Klasse clsRegistry	554
11.9.3	Die Klasse clsNLS	558
11.9.4	Schlussbemerkungen zu Settings.xls	558
11.10	Mehrsprachige Applikationen	558
11.10.1	Die Beispielanwendung	560
11.10.2	Die Tabellen	561
11.10.3	Der Code	563
11.11	Tabellendesign	575
11.11.1	Worum geht es ? – Ein Beispiel	575
11.11.2	Normalisierung in der Theorie	576
11.11.3	Normalisierung in der (Excel-) Praxis	580
11.11.4	Skalierbarkeit	581
12	Excel und Datenbanken	583
12.1	Data Access Objects	586
12.1.1	Wege des Datenzugriffs	588
12.1.2	Das DAO-Objektmodell	589
12.1.3	Zugriff auf eine JET-Datenbank	590
12.1.4	Zugriff auf eine ODBC-Datenbank	592
12.1.5	Zugriff auf Recordsets	594
12.1.6	Zugriff auf Felder	595
12.2	ActiveX Data Objects	596
12.2.1	Das ADO-Objektmodell	597
12.2.2	Navigation in einem Recordset	599

12.3	ANSI-SQL	601
12.3.1	SELECT	603
12.3.2	UPDATE	609
12.3.3	INSERT	610
12.3.4	DELETE	611
12.4	Die Klasse clsUtil	611
12.5	DBRechnung.xls	614
12.5.1	Die neuen Datenklassen	614
12.5.2	Der Kundendialog	617
12.5.3	Zusammenfassung	624
13	Add-Ins	627
13.1	Komponenten von Add-Ins	628
13.2	Erstellen von Add-Ins	629
13.2.1	CommandBar erzeugen	629
13.2.2	Der Dummy-Dialog	632
13.2.3	Add-In erzeugen	633
14	Excel und Visual Basic	635
14.1	Von VBA nach VB	637
14.2	Voraussetzung für eine Portierung	640
14.2.1	Komponentenprogrammierung	640
14.2.2	Verweise auf Excel-Objekte im Excel-Code	640
14.2.3	Verweis auf die Excel-Bibliothek	641
14.2.4	Verweis auf Excel in Visual Basic	641
14.3	Excel-Objekte im Visual Basic Code	642
14.3.1	Zugriff auf Excel-Objekte im Code	642
14.3.2	Excel-Verweise auflösen	643
14.3.3	Tabellenblätter im OLE-Steuerelement	646
15	Lösungen	649
A	Anhang	653
A.1	Abkürzungen und Begriffe	654
A.2	KeyCode-Konstanten	656
	Stichwortverzeichnis	661



Vorwort

Im Dezember 1999 hielt ich ein 3-tägiges VBA-Seminar in den Räumen eines sehr großen deutschen Unternehmens. Dabei hatte ich mir vorgenommen, nie wieder zu versuchen, in drei Tagen Excel-VBA an den Mann zu bringen. Aus allen Ecken der Republik reisten zu diesem Seminar Mitarbeiter dieses Unternehmens an. Natürlich hatten sie und auch ihr berufliches Umfeld gewisse Erwartungen an dieses Seminar.

Da war zum Beispiel die Seminarbeschreibung aus dem Fortbildungskatalog dieses Unternehmens, die den Eindruck erweckte, dass in drei Tagen Excel-VBA (sprich VBA und das Excel-Objektmodell) anhand von Beispielen *erarbeitet* werden kann. Oder etwa der *Makro*-Rekorder, mit dem man so nützliche kleine *Makros* aufzeichnen kann. *Makros*? Das sind doch so kleine Aneinanderreihungen von Anweisungen, die immer wiederkehrende Aufgaben automatisieren, oder nicht?

In diesen drei Tagen hatten wir uns nicht mit Makros beschäftigt. Wir redeten über Datentypen und deren Gültigkeit, wir diskutierten Aspekte des Anwendungsdesigns, bauten Dialoge, durchquerten Kontrollstrukturen im gestreckten Galopp, entwarfen eine Fehlerbehandlung, kapselten das Öffnen einer Arbeitsmappe in einer ellenlangen Methode (statt sie einfach mit der Open-Methode zu öffnen), schrieben Klassen für Datenzugriff, legten Programmparameter wahlweise in einem Tabellenblatt oder der Registry ab, entwickelten Techniken im Umgang mit schlecht strukturierten Tabellen ...

»Gut«, sprach dann ein Teilnehmer (stellvertretend für alle) am dritten Tag, »bei *richtigen* Applikationen, die unternehmensweit eingesetzt werden, muss man das alles wissen. Aber wenn ich für mich ein kleines Makro « – wieder dieses entsetzlich Wort – » schreiben will, muss ich nicht prüfen, ob die Datei da ist. Ich muss auch keine ListBoxes gegeneinander verriegeln, Datenzugriffe kapseln oder ...«

Dann erzählte ich ihnen die Geschichte von der Sales-Controlling-Applikation, die ich irgendwann 1998 für ein ebenfalls sehr großes Unternehmen schrieb. Dieses Excel-VBA-Programm erstellte monatlich nationale Verkaufsübersichten, plante Budgets, berechnete die sehr komplizierten Rabatte und Boni für die unterschiedlichsten Kunden. Der Sales-Controller, mit dem zusammen dieses Tool entstand, war ein ausgesuchter Excel-Kenner, der sogar über gute VBA-Kenntnisse verfügte. Gegen meinen ausdrücklich und regelmäßig geäußerten Rat, etwas mehr Zeit in das Drumherum des Programms zu investieren, beschränkten wir uns auf reine Funktionalität.

Da die Zusammenhänge sehr komplex waren, entwickelten wir das Programm gemeinsam. Gemeinsam heißt, er saß neben mir und sagte, wo welche Information steht und wie es miteinander zu verknüpfen ist, und ich schrieb im Akkord die dazugehörigen Programmzeilen. Anstelle von Plausibilitätskontrollen erhielt das Programm Funktionalität und noch mal Funktionalität. »Wenn ich mit den Basisdateien fertig bin, stimmen sie.«, sagte er, »Das können wir uns also schenken.« Da ich zu Anfang die Strukturen nicht verstanden hatte, entstand Zeile um Zeile. Er sagte: »Das ist so und so.«, und ich verknotete zeitgleich die Daten. Das Ergebnis war zutiefst unmodular, Spaghetti-Code in Reinkultur. Trotz Range-Objekten, Workbooks und Worksheets.

Die Investitionen in dieses Programm amortisierten sich innerhalb weniger Wochen. Alle waren zufrieden, sieht man einmal von mir ab.

Dann kam das Unvermeidliche: Dieser Mitarbeiter wechselte Mitte 1999 zu einer renommierten Unternehmensberatung. Seine Nachfolger (Plural!) im ehemaligen Unternehmen kamen mit diesem hochspezialisierten Tool jedoch nicht zurecht, trotz geregelter Übergabe und Einarbeitung. Anfangs riefen sie noch täglich bei mir an, dann immer weniger, und nach zwei Monaten herrschte Funkstille. Das Programm, so wird inzwischen berichtet, ist nicht mehr im Einsatz. Man plant wieder von Hand, mit all den Problemen, die auch damals zu der Entscheidung führten, dieses Projekt überhaupt anzugehen.

Auch damals wusste ich, wie man Programme schreibt oder besser gesagt, wie man sie nicht schreibt. Mein Fehler war schlicht und ergreifend, dass ich mich nicht daran hielt.

Obwohl das Ganze kein Ruhmesblatt für mich darstellt, erzählte ich damals diesem Seminarteilnehmer und auch jetzt Ihnen diese Geschichte.

Eine Excel-VBA-Anwendung besteht nicht nur aus einer Reihe raffinierter, meterlanger Zugriffe auf Range-Objekte. Eine Excel-VBA-Anwendung basiert auf Datenstrukturen, die das Wort Struktur auch verdienen, kapselt

die Zugriffe auf diese Daten in Klassen, verfügt über eine ausgefeilte Fehlerbehandlung und ist in hohem Maße wartungsfreundlich. Und wenn dann irgendwann die Datenhaltung nicht mehr in Excel-Arbeitsmappen erfolgt, sondern Datenbanken zum Einsatz kommen, so werden ein oder zwei Klassenmodule angepasst, ohne dass dies die geringste Auswirkung auf die Programmlogik hat. Soll nun die Applikation sogar mehrsprachig mit dem Anwender in Kontakt treten, so ist auch das kein Problem.

Und genau davon handelt auch dieses Buch. Natürlich reden wir auch über Zugriffe auf das Range-Objekt, aber wir werden auch die anderen Aspekte besprechen, die eine professionelle Anwendung ausmachen. Denn mit Excel-VBA steht Ihnen ein Werkzeug zur Verfügung, mit dem man wirklich professionelle Anwendungen erstellen kann.

Wenn Sie weder mit VBA, noch mit den Excel-Objekten, noch mit Applikationsdesign oder Objektorientierung vertraut sind, steht ein langer und mitunter schwieriger Weg vor Ihnen. Lang und schwierig, aber die Mühe wird sich lohnen ...

Die Seiten, die vor Ihnen liegen, geben meine persönliche Sicht der Dinge wieder. Aus diesem Grund ist das Wort *ich* in seinen Deklinationsformen vielleicht das meistgebrauchte Wort dieses Buches. Es spricht keine geheimnisvolle Person zu Ihnen, die den Autor zu kennen scheint und ihn als *der Autor* reden lässt. *Ich* habe dieses Buch geschrieben und *ich* mache Fehler, und zwar mit schöner Regelmäßigkeit.

Außerdem entwickelt man sich weiter, auch oder vielleicht sogar als alter Hase. Als ich mit dem letzten Kapitel fertig war, hatte ich wieder Ideen, wie man die älteren Teile des Buches ändern könnte. Und so würde ich vermutlich in zwei Jahren noch an diesem Buch schreiben, wenn da nicht eine Allianz aus Verlag, Druckerei und Ehefrau auf dem Plan erschienen wäre. Der Verlag will verlegen, die Druckerei will drucken und die Ehefrau möchte wieder Ehefrau sein.

Wir werden auf der WebSite meines Unternehmens unter www.Sophtware.de eine Ecke einrichten, auf der Sie Korrekturen finden können. Wenn Beispieldateien geändert werden, so finden Sie auch diese unter der angegebenen Adresse. Sollten Sie auf Fehler stoßen oder Anregungen haben, so scheuen Sie sich bitte nicht, eine Mail an mich zu senden (Klaus.Prinz@Sophtware.de).

Die Icons in diesem Buch

Um Ihnen die Orientierung in diesem Buch zu erleichtern, haben wir den Text in bestimmte Funktionsabschnitte gegliedert. Folgende Icons finden Verwendung:



Manches ist von besonderer Bedeutung und verdient darum auch, besonders hervorgehoben zu werden. Solche *Hinweise* sind sehr nützlich, sie bringen einen geschwinder ans Ziel.



Dieses Icon weist auf Besonderheiten in *Excel 97* hin.



Auf Dinge, die Sie in *Excel 2000* berücksichtigen sollten, werden Sie mit diesem Icon hingewiesen.



Excel und Visual Basic for Applications

1

1.1	Objektkunst	16
1.2	Auf ein Wort	19
1.3	Übungsaufgabe	21



Visual Basic for Applications (VBA) ist ein modernes Entwicklungssystem auf Basis der Sprache Visual Basic und hat zuerst einmal mit Excel nichts zu tun. VBA ist zwar in einer Reihe von Applikationen innerhalb und außerhalb der Microsoft-Produktpalette implementiert und arbeitet mit den dortigen Objektmodellen zusammen, im Kern handelt es sich bei VBA in diesen Applikationen aber um eine Reihe von identischen Funktionsbibliotheken. VBA wird in dem Moment zu Excel-VBA, in dem es Kontakt zur Microsoft Excel 8.0 Object Library bekommt. Ebenso ist Word-VBA die Synthese aus VBA und der Microsoft Word 8.0 Object Library.

Unsere mit VBA aufgerüstete Applikation Excel erschließt uns also einen komfortablen programmatischen Zugriff auf alle in der gastgebenden Applikation implementierten Funktionalitäten. Allerdings sind wir hier an einer wesentlichen Einschränkung angelangt, denn VBA ist ohne den Gastgeber nicht lauffähig. Wir können in VBA somit keine eigenständig lauffähigen Programme erstellen und diese ohne unseren Gastgeber Excel zu irgendetwas bewegen. Excel-VBA-Anwendungen bestehen also immer aus einer Arbeitsmappe und den VBA-Komponenten.

Obwohl VBA und VB vieles gemeinsam haben, so ist VBA nur ein Teil dessen, was Visual Basic anzubieten hat. Zwar ist zu erwarten, dass sich die beiden immer ähnlicher werden, doch wird es in VBA auch in absehbarer Zukunft nicht möglich sein, eigenständig lauffähige Programme schreiben zu können. Auch das Entwickeln eigener ActiveX-Steuerelemente wird wohl VB vorbehalten bleiben. Doch darf man gespannt sein, wo die Reise hingehen wird, und zwar bei VB und bei VBA ...

Doch zurück zu Excel-VBA. Wenn wir uns nun dem Komplex Excel-VBA nähern, so liegen die beiden Komplexe Excel-Objektmodell und VBA vor uns, wobei Letzteres sich in der Praxis aus der reinen Sprache (VBA 5 Language Engine) und den Dialogen (Microsoft Forms 2.0 Object Library) zusammensetzt. Wenn diese Begriffe zu Anfang noch etwas verwirren sollten, so ist das nicht weiter tragisch, denn es handelt sich zum einen um reine Hintergrundinformationen, die für das Verständnis von VBA nicht lebensnotwendig sind. Zum anderen werden wir in späteren Kapiteln noch ausführlich darüber reden.

1.1 Objektkunst

Natürlich sind auch der Wolf und die sieben Geißlein Objekte. Der Wolf, nach seiner Standardeigenschaft gefragt, würde vermutlich hungrig antworten und als Standardmethode die Pirsch auf Geißlein angeben.

In einem VBA-Seminar für Mitarbeiter eines großen deutschen Bahnbetreibers näherten wir uns den Objekten, indem wir als Anschauungsobjekt unserer Objektbetrachtungen eine Lokomotive wählten. Für gestan-

dene Eisenbahner ist dies etwas konkreter als Excel-Objekte, zumal wir uns durch einen Blick aus dem Fenster jederzeit davon überzeugen konnten, dass diese Objekte existierten.

Wir fanden eine Reihe von Attributen (Eigenschaften), mit denen eine Lokomotive beschrieben werden kann:

Eigenschaft	Beschreibung	Wertbeispiel
Masse	Masse der Lokomotive in kg	Lokomotive.Masse = 50000
Länge	Länge der Lokomotive in m	Lokomotive.Länge = 12,34
Antrieb	Antriebsart der Lokomotive	Lokomotive.Antrieb = 1 (Dampf) Lokomotive.Antrieb = 2 (Diesel) Lokomotive.Antrieb = 3 (Elektro)

Die Eigenschaft Masse ist durch einen Punkt von dem voranstehenden Objekt Lokomotive getrennt. Die Wertzuweisung erfolgt durch = Gleichheitszeichen und den Wert *ohne* Einheit. Nach den Eigenschaften, quasi einer statischen Beschreibung des Zustands unseres Lokomotiven-Objekts, nun die dynamische Komponente, die Methoden.

Was kann eine Lokomotive tun? Nun, fahren zum Beispiel oder bremsen. Wenn ein Lokomotivführer den Auftrag bekommt, zu fahren, dann weiß er, was zu tun ist. Wenn wir aber eine allgemein gültige Beschreibung des Vorgangs Fahren erzeugen wollen, so reicht Fahren allein nicht aus, denn sie könnte zumindest vorwärts oder rückwärts losfahren:

Lokomotive.Fahren vorwärts

Reicht das? Kommt darauf an, wie die Anweisung (oder Methode) Fahren umgesetzt (oder implementiert) ist. Übergibt man der Antriebssteuerung eine Zielgeschwindigkeit oder ein Drehmoment?

Lokomotive.Fahren vorwärts, 120 (km/h)

oder

Lokomotive.Fahren vorwärts, 10000 (Nm)

Nun bremsen wir das Ungetüm noch ab. Genau wie bei den uns etwas vertrauteren Vehikeln können wir auch bei Lokomotiven die Bremswirkung beeinflussen. In pneumatischen Bremssystemen könnten wir den Bremsdruck angeben:

Lokomotive.Bremsen 8 (bar)

Die Elektrolokomotive wird noch einen weiteren Trick auf Lager haben, der etwas mit Wirbelstrombremse oder auch Gleichstrombremse zu tun hat:

Lokomotive.Wirbelstrombremse = Ein

Hier bremsen wir indirekt über eine Eigenschaft, indem wir einen elektromagnetischen Prozess einleiten.

Nach diesem kleinen Ausflug in die Welt der Hardware kehren wir wieder zurück in die Gefilde der Software.

Excel-Objekte

Excel ist voller Objekte; rund 120, sagt man. Eine Arbeitsmappe beispielsweise ist ein Objekt, aber auch eine Tabelle oder ein Zelle. Sie sind gewissermaßen die Objekte unserer Begierde in Excel VBA, denn wir wollen Arbeitsmappen öffnen, Tabellen auswählen und in deren Zellen schreiben oder sie auslesen.

Objekte verfügen in der Regel über Eigenschaften und Methoden, einige sogar über Ereignisse. Unsere Arbeitsmappe (Excel-Datei) hat auch Eigenschaften und Methoden. Als Eigenschaft könnte man beispielsweise den Namen anführen, als Methode Öffnen, Speichern oder Schließen.

Die nachstehende Tabelle zeigt einige Excel-Objekte mit typischen Eigenschaften:

Objekt	Eigenschaft	Erläuterung
Application	ScreenUpdating	Bildschirmaktualisierung eingeschaltet oder nicht
Workbook	Name	Name der Datei ohne Pfadangabe
	Path	Pfad der Datei ohne Dateinamen
Worksheet	Name	Name des Tabellenblattes
	Visible	Legt fest, ob die Tabelle sichtbar ist
Range	Value	Der Inhalt der Zelle
	NumberFormat	Zahlenformat der Zelle

Application.ScreenUpdating = True

Worksheet.Name = "Datentabelle"

Schauen wir uns demgegenüber ein Reihe typischer Methoden an:

Objekt	Methode	Erläuterung
Application	Calculate	Berechnet alle geöffneten Arbeitsmappen
Workbook	Save	Speichert die Arbeitsmappe
	Close	Schließt die Arbeitsmappe
Worksheet	Delete	Löscht die Tabelle
	PrintOut	Druckt die Tabelle
Range	ClearContents	Löscht die Inhalte von Zellen
	Sort	Sortiert einen Zellbereich

1.2 Auf ein Wort

Die meisten Menschen neigen in der einen oder anderen Form zur Gruppenbildung. Untereinander verständigen sich diese Gruppenmitglieder unter anderem über Sprache und meist entwickelt sich nach kurzer Zeit ein über die gesellschaftlichen Konventionen hinausgehender Gruppendialekt. Dieser dient gelegentlich der einfacheren Verständigung der Gruppenmitglieder untereinander, gelegentlich auch der Abgrenzung der Lateiner von den Nicht-Lateinern.

Auch die deutschsprachige Entwicklergemeinde verhält sich da nicht anders. Da gibt es Begriffe, die ihren Ursprung in der deutschen Sprache haben, vielleicht mit einer Spur Latein angereichert. Nehmen Sie den wunderschönen Ausdruck Methodensignatur. Oder Instantiierung, Objektorientierung, Population und Serialisierung.

Dann gibt es auch Begriffe, die in deutscher und englischer Schreib- und Hunderten von Sprechweisen existieren. Sollen wir uns mit den englischen Ausdrücken abgeben, wenn es deutsche Äquivalente gibt, die man ohne mitten im Wort Luft holen zu müssen auch verwenden kann?

Die Antwort ist ein ganz klares »Ja, aber«. *Ja*, denn das Wort Schaltfläche beispielsweise ist prägnant und weit verbreitet. *Aber* bei einem Kontrollkästchen denkt man schon eher an diese Blechbüchse in alten Filmen, in der Nachtwächter bei ihrer Rundgängen einen kleinen Schlüssel umdrehen. Da ist *noch* ein *aber* im Spiel, das jedoch einen kleinen Umweg erfordert.

Der erste Kontakt mit der VBA-Hilfe verursacht zumindest ein leichtes Erstaunen. Man fühlt sich an ein klassisches Theater erinnert, in dem Kulissen hin und her geschoben werden. Da drängt sich ein Fenster ruckweise von rechts auf die Bühne, verdeckt das andere zu zwei Dritteln, und wechselt man wieder zurück in die Ausgangskulisse, so hat diese ein Drittel ihrer ursprünglichen Breite eingebüßt.

Dieses Provinztheater VBA-Hilfe spricht deutsch, erinnert bezüglich seiner Wortgewaltigkeit jedoch an Pantomime. Doch wir haben gleich um die Ecke ein Staatstheater mit hervorragendem Programm und der Eintritt ist *frei!*

Where's the catch? It does not speak a word german!

Dieses Wunderwerk nennt sich MSDN (Microsoft Developer Network) und ist dank eines Umdenkens bei Microsoft nicht mehr mit Investitionen verbunden. Aktuelle Informationen erhalten Sie unter www.microsoft.com/germany/msdn.

Um Ihnen die Nase lang zu machen, hier ein ScreenShot des MSDN-Inhalts:

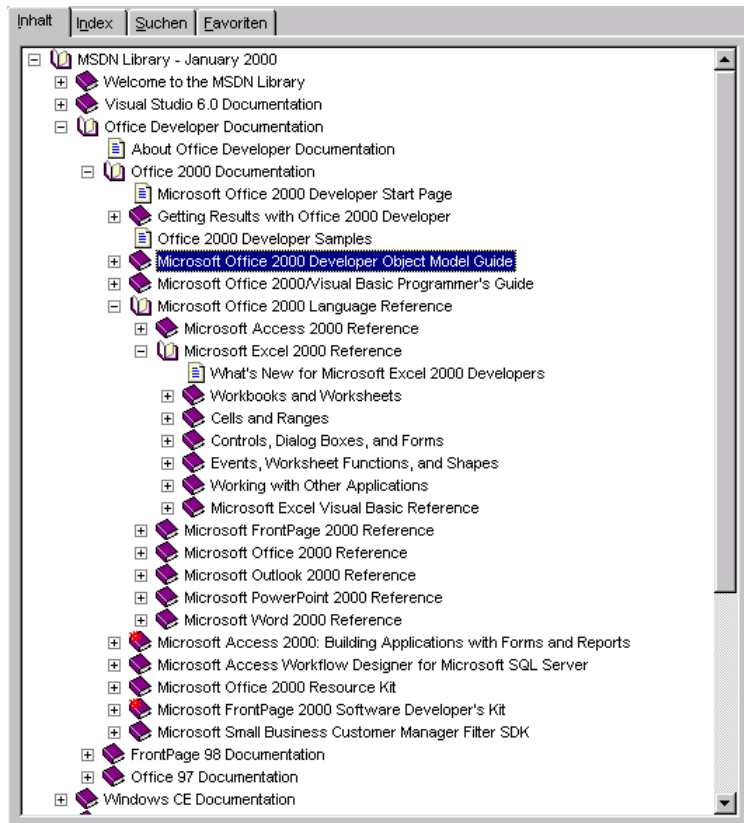


Abbildung 1.1: MSDN-Inhalt

Man könnte vermuten, dass die Hilfe zu Excel-VBA eine reine Übersetzung ist und im Übrigen tupfengleich der MSDN entspricht. Dem ist nicht so, in vielen Fällen hilft die Hilfe nämlich nicht, aber ein Blick in die MSDN bringt nach kurzem die gewünschte Information. Es ist sicherlich keine Lobhudelei, die MSDN als hervorragende Informationsquelle zu bezeichnen. Nicht nur, dass eine umfassende Dokumentation aller Entwicklungssysteme und Objektmodelle darauf zu finden ist, sie enthält auch eine riesige Zahl von Artikeln, die in der Knowledge Base zusammengefasst sind.

Die Knowledge Base enthält Tausende von Beiträgen, die nach dem Muster Symptome, Ursache, Lösung, Status und Verweise aufgebaut sind. Als ich darüber stolperte, dass die Datei »Pagefile.sys« in Zusammenhang mit der `GetAttr`-Funktion einen unerklärbaren Fehler brachte, zeigte ein Suche nach dem Begriff »Pagefile.sys« innerhalb von wenigen Sekunden die Lösung: Es handelte sich in der Tat um einen Bug. In Kapitel 7 wird dieser Fall auch erklärt.

Knowledge Base

Die MSDN enthält darüber hinaus neun vollständige Bücher, 22 mitunter größere Buchauszüge und ebenfalls 22 Zeitschriften, die jedoch nur in Auszügen dargestellt sind.

Überflüssig zu sagen, dass Sie hier das eine oder andere Megabyte Festplattenkapazität bereitstellen müssen. Es gibt zwar eine Minimalinstallation, bei der Sie jedoch Diskjockey (3 CDs) werden, aber die Vollinstallation mit mindestens einem Gigabyte ist wirklich sinnvoll.

Nach diesem mächtigen Bogen über die MSDN kommen wir zu der Frage, ob man Kontrollkästchen oder CheckBox sagen soll. Die Tooltips der Werkzeugsammlung reden von einem Kontrollkästchen, das Eigenschaftsfenster von der CheckBox und die VBA-Hilfe von beidem. Da Sie in der MSDN jedoch bei einem Kontrollkästchen nur Kopfschütteln ernten werden, scheint die Verwendung der englischen Ausdrücke ein Gebot der Stunde zu sein. Und so wollen wir es auch halten, mögen auch nicht alle glücklich darüber sein.

1.3 Übungsaufgabe

Wir setzen unsere Lokomotive mit der folgenden Methode in Bewegung:

Lokomotive.Fahren vorwärts, 120 (km/h)

Wäre es denkbar, diesen Effekt nur mit Eigenschaften zu erreichen? Wie müssten sie aussehen?



Die Entwicklungsumgebung

2

Kapitelüberblick

2.1	Projekt-Explorer	26
2.1.1	Microsoft Excel Objekte	26
2.1.2	Formulare	27
2.1.3	Module	27
2.1.4	Klassenmodule	27
2.1.5	Das Kontextmenü des Projekt-Explorers	27
2.2	Eigenschaftsfenster	29
2.3	Codefenster	31
2.3.1	Kontextmenü des Codefensters	33
2.3.2	Kontextmenü des Codefensters im Haltemodus	39
2.4	Direktfenster	40
2.5	Lokal-Fenster	42
2.6	Überwachungsfenster	43
2.6.1	Bedeutung der Spalten	44
2.6.2	Überwachungsausdrücke	45
2.7	Objektkatalog	47
2.8	Menüs	52
2.8.1	Menü Datei	52
2.8.2	Menü Bearbeiten	53
2.8.3	Menü Ansicht	54
2.8.4	Menü Einfügen	59
2.8.5	Menü Format	61
2.8.6	Menü Debuggen	61
2.8.7	Menü Ausführen	63
2.8.8	Menü Extras	64
2.9	Symbolleisten	79

**Integrated
Development
Environment**

Seit Excel '97 steht uns VBA-Entwicklern eine Entwicklungsumgebung zur Verfügung, in der das Arbeiten richtig Spaß macht. Somit ist eine Grundvoraussetzung für gute Arbeitsergebnisse erfüllt.

Dieser Komfort hat allerdings seinen Preis, denn man fühlt sich beim ersten Kontakt zu dieser IDE (Integrated Development Environment, der amerikanische Ausdruck für Entwicklungsumgebung) schon ein wenig erschlagen. Doch es ist wie mit allem Neuen: Es wird von Mal zu Mal vertrauter. Zudem gilt auch hier, dass man nicht alle Features von Anfang an beherrschen muss. Mit der Zeit wird sich auch eine Routine einstellen, die einen Großteil Funktionalität erübrigen wird. Wenn Sie Ihre einhundertste Befehlsschaltfläche in Ihren dreißigsten Dialog gesetzt haben, so werden Sie keine Unterstützung der IDE bei der Ausrichtung dieser Elemente mehr benötigen: Sie werden auf Anhieb an der richtigen Position platziert.

In diesem Kapitel werden alle Elemente der Entwicklungsumgebung im Einzelnen vorgestellt. Hierunter fallen die einzelnen Fenstertypen, die Symbolleisten, die Menüs, die Kontextmenüs und deren gemeinsame Dialoge. Wie es in guten Windows-Programmen nun mal üblich ist, erscheinen in Kontextmenüs jedoch keine originären Funktionalitäten, sondern nur auf den jeweiligen Kontext bezogene Einträge, die auch im eigentlichen Menü zu finden sind. Gleiches gilt übrigens für Symbolleisten.

Kontextmenüs stellen somit eine Teilmenge der in den eigentlichen Menüs enthaltenen Funktionen dar. Die Kontextmenüs wiederum sind an dem jeweiligen Bedienkontext ausgerichtet, daher der Name ;-). Das wiederum verleitete mich dazu, die Beschreibung genau von dieser Seite her aufzuzäumen: Wir beginnen also mit den Kontextmenüs der einzelnen Fenster. Danach werden die restlichen Menüfunktionen behandelt. Abschließend widmen wir uns dann den Symbolleisten, die ja per Definition keine neuen Elemente mehr beinhalten oder besser gesagt enthalten dürfen, denn zwei sind dennoch reingerutscht.

Der Umfang des folgenden Kapitels hat mich selbst überrascht. Nur das Wesentliche darzustellen habe ich schnell verworfen, da das *Wesentliche* eines solch mächtigen Werkzeugs viel mit Erfahrung und Neigung zu tun hat. Und Erfahrung und Neigung sind höchst subjektive Aspekte des Arbeitens mit VBA. Somit ist das Kapitel etwas größer geworden, zeigt aber nun mit Ausnahme der Add-Ins jede Funktion, die in der Entwicklungsumgebung enthalten ist.

Es ist also eine Mischung aus Leitfaden und Referenz geworden, mit all den daraus resultierenden Nachteilen, was Sie mir bitte nachsehen wollen.

Fast alles ist per Menü bedienbar, das Wichtigste davon auch per Kontextmenü und vieles per Symbolleiste oder ShortCuts, also für jeden etwas

...

Die folgende Abbildung 2.1 zeigt nun die Entwicklungsumgebung in einer durchaus üblichen Erscheinungsform. Es sind nicht alle möglichen, sondern alle in der Regel nötigen Komponenten zu sehen:

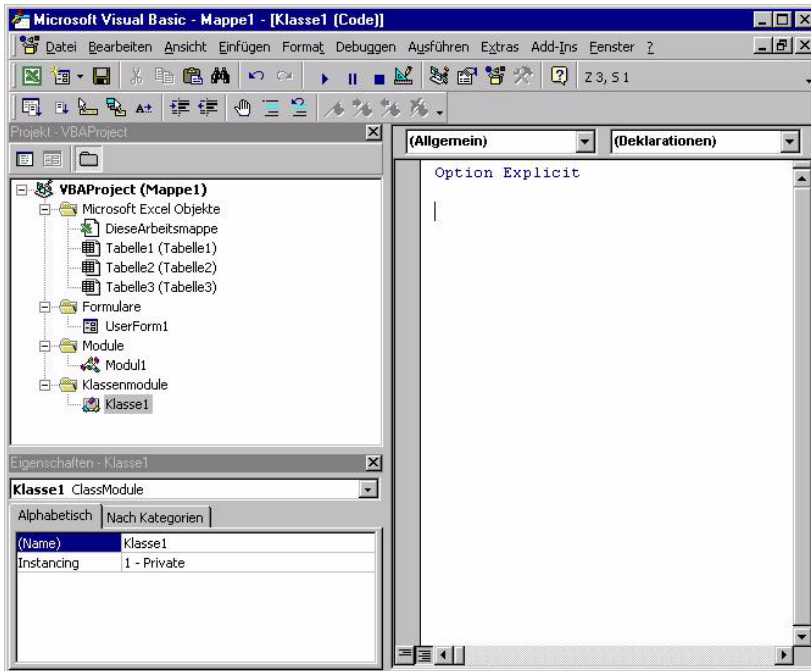


Abbildung 2.1: Entwicklungsumgebung mit üblichen Elementen

Neben der Menüleiste und den Symbolleisten sind in der vorangehenden Abb. 2.1 oben links der Projekt-Explorer, unten links das Eigenschaftsfenster und rechts das Codefenster zu sehen.

2.1 Projekt-Explorer

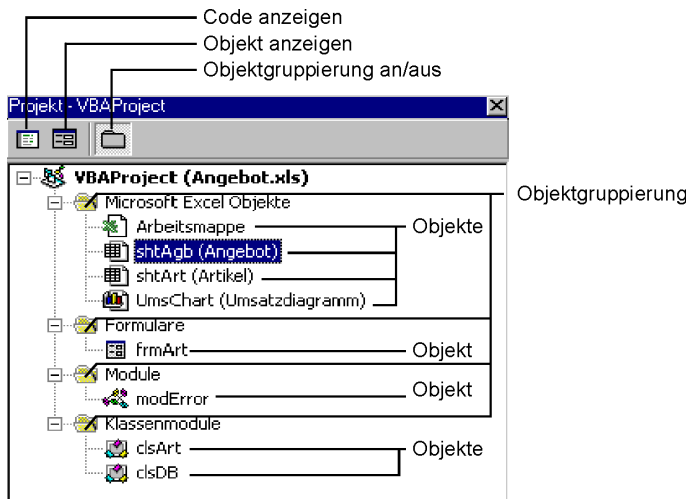


Abbildung 2.2: Projekt-Explorer mit allen verschiedenartigen Elementen

Der in diesem Fenster zu sehende TreeView, so heißt dieses aus dem Windows-Explorer bekannte Steuerelement, gibt die jeweiligen Strukturen der dargestellten Projekte wieder. Hierbei wird eine Datei einem Projekt gleichgestellt, was eine durchaus übliche Vorgehensweise ist. Ein Projekt kann folgende Gruppen enthalten:

2.1.1 Microsoft Excel Objekte

Diese immer vorhandene Gruppe enthält alle in der Arbeitsmappe enthaltenen Tabellen- und Diagrammblätter sowie einen die Arbeitsmappe selbst repräsentierenden Eintrag. Sollten Sie – aus welchem Grunde auch immer – noch Excel 5 Dialoge oder Makroblätter in Ihrer Arbeitsmappe enthalten haben, so werden diese nicht im Projekt-Explorer erscheinen. Denn genau genommen werden nur die Blätter (oder wollen wir sie vielleicht treffender Objekte nennen) aufgeführt, die Ereignisse produzieren können, was Excel 5 Dialogen und Makroblättern sinnvoller Weise verwehrt bleibt. Das genau ist auch der Grund, weshalb die Arbeitsmappe selbst erscheint, denn diese kann übergreifende Ereignisse erzeugen. Die zu Grunde liegenden Mechanismen werden bei der Vorstellung des Codefensters genauer gewürdigt.

Jedem hier aufgeführten Microsoft Excel Objekt ist ein eigenes, gebundenes Codemodul fest zugeordnet. Die beiden Befehlsschaltflächen *Code anzeigen* und *Objekt anzeigen* im Kopf des Projekt-Explorers dienen der Navigation zwischen dem Objekt selbst und seinem gebundenen Codemodul.

2.1.2 Formulare

In dieser Rubrik sind alle Formulare aufgeführt, die in dem Projekt enthalten sind. Der Begriff Formular steht gewissermaßen als Synonym für Dialog. Die Erstellung und Steuerung von Formularen oder Dialogen wird im Kapitel Dialog ausführlich vorgestellt.

2.1.3 Module

In Ergänzung zu den gebundenen Codemodulen kennt VBA auch noch freie oder ungebundene Codemodule. Üblicherweise wird darin projektübergreifender Code stehen, beispielsweise Fehlerbehandlung oder umfangreiche Routinen, die aus Gründen der Übersichtlichkeit aus gebundenen Codemodulen ausgelagert sind.

2.1.4 Klassenmodule

Klassenmodule sind den freien Codemodulen sehr ähnlich, erlauben aber das Erzeugen von Objekten. Diesem überaus spannenden Thema werden wir uns auch noch intensiv widmen.

2.1.5 Das Kontextmenü des Projekt-Explorers

Der Projekt-Explorer verfügt über ein umfangreiches Kontextmenü, welches Sie mittels der rechten Maustaste hervorlocken können.

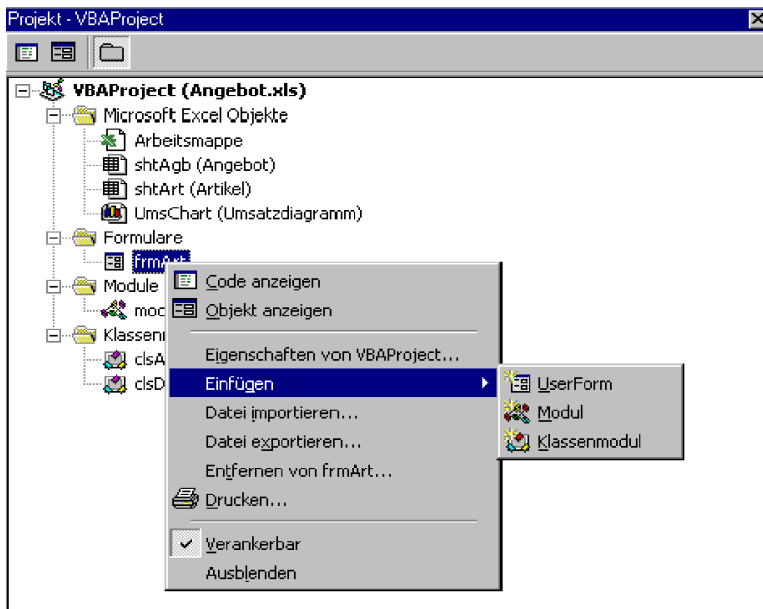


Abbildung 2.3: Kontextmenü des Projekt-Explorers

Code anzeigen

Dieser Menüpunkt führt zur Anzeige des Codemoduls des ausgewählten Eintrags.

Objekt anzeigen

Sofern Sie ein Element der Gruppe Microsoft Excel Objekte oder Formulare ausgewählt haben, wird das entsprechende Objekt angezeigt. Wählen Sie ein Objekt der Gruppe Microsoft Excel Objekte, so wechselt VBA zu Excel und zeigt die jeweilige Tabelle.

Haben Sie ein Modul oder ein Klassenmodul selektiert, so steht dieser Befehl nicht zur Verfügung.

Eigenschaften von VBA-Projekt ...

Wählen Sie diesen Eintrag aus, so wird der Dialog Projekteigenschaften angezeigt.

Einfügen

Dieser Menüpunkt blendet ein Untermenü ein, das die Einträge User-Form, Modul und Klassenmodul bereithält. Hiermit können Sie eines dieser Elemente zu ihrem aktuellen Projekt hinzufügen.

Datei importieren ...

Formulare, ungebundene Module und Klassenmodule können separat gespeichert werden. *Datei importieren ...* blendet den Datei-Öffnen-Dialog ein und erlaubt eine solche eigenständige Datei Ihrem aktuellen Projekt hinzuzufügen.

Datei exportieren ...

Wenn Formulare, ungebundene Module und Klassenmodule importiert werden können, so ist es hilfreich, wenn diese auch aus einem Projekt exportiert werden können. Und diese Möglichkeit bietet Ihnen diese Funktion.

Entfernen von Objektname ...

Sie können sich jederzeit auch wieder von Formularen, ungebundenen Modulen und Klassenmodulen mittels dieses Menüpunktes trennen. Generell wird die folgende MessageBox eingeblendet, die Ihnen ermöglicht, das betreffende Objekt vorher zu exportieren:

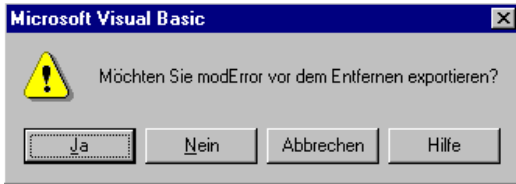


Abbildung 2.4: Sicherheitsabfrage vor Entfernen eines Objektes

Entscheiden Sie sich für Ja, so wird der von »Datei exportieren...« her bekannte Dialog erscheinen, bei Nein wird das Objekt endgültig entfernt.

Drucken ...

Diese Auswahl bringt den Drucken-Dialog auf den Bildschirm. Haben Sie ein Microsoft Excel Objekt oder ein Formular ausgewählt, so wird der Drucken-Dialog angezeigt.

Verankerbar

Der Bildschirmbereich der Entwicklungsumgebung ist *zu Anfang* in einen linken und rechten Bereich unterteilt. In diesen Bereichen lassen sich bis auf das Codefenster alle Fenster einrasten oder verankern. Im linken dieser beiden Bereiche befinden sich üblicherweise der Projekt-Explorer und das Eigenschaftsfenster. Verankerbar heißt nun, dass Sie ein Fenster in diesen Bereich hineinziehen können, woraufhin das Fenster automatisch rastet. Rasten Sie, so viel Sie wollen, aber achten Sie darauf, dass im linken Bereich immer mindestens ein Fenster verbleibt. Entfernen Sie alle Fenster aus dem linken Bereich, so erwartet Sie eine hübsche Zeit der Fummelei, bis dieser linke Bereich wieder als eigenständige Rastfläche zur Verfügung steht.

Wenn Sie etwas mehr Platz für Ihr Codefenster benötigen, so können sie den rechten Rand des Projekt-Explorers oder des Eigenschaftsfensters nach links verschieben, bis diese Fenster fast verschwunden sind. Aber lassen Sie bitte diese beiden Fenster eingeblendet und verankert!



Ausblenden

Dieser Kontextmenüeintrag schließt den Projekt-Explorer. Den gleichen Effekt mit einem Mausklick weniger erreichen Sie, wenn Sie auf das Kreuz in der rechten oberen Ecke des Projekt-Explorers klicken.

2.2 Eigenschaftsfenster

Das Eigenschaftsfenster zeigt die zur Entwurfszeit veränderbaren Eigenschaften des selektierten Objektes an. Stellvertretend für die Eigenschaften der Gruppe Microsoft Excel Objekte steht die folgende Abbildung:

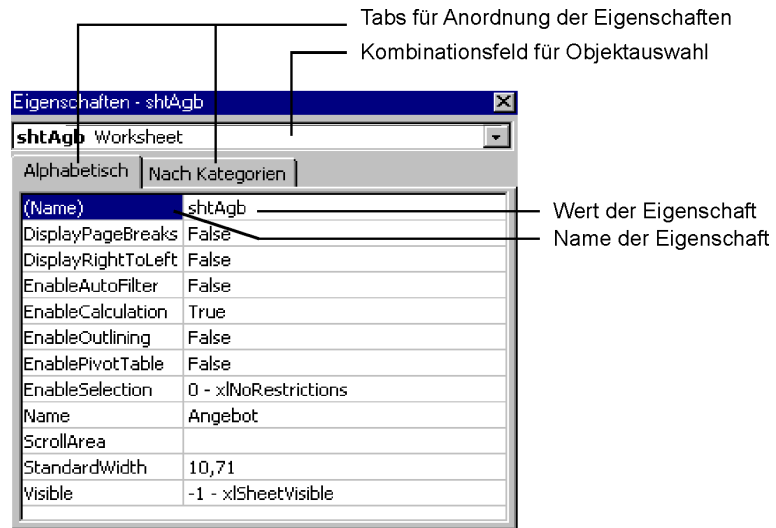


Abbildung 2.5: Eigenschaften einer Tabelle

Container Unterhalb der Titelleiste dieses Fensters erlaubt ein Kombinationsfeld (ComboBox) die Auswahl der verfügbaren Objekte, die je nach Art des ausgewählten Containerobjektes andere Einträge beinhalten, wobei das Containerobjekt selbst auch Teil dieser Auswahl ist. Das Objekt Arbeitsmappe beispielsweise ist Container für die darin enthaltenen Tabellen- und Diagrammblätter, das Objekt Tabelle für alle dortigen Steuerelemente und ein Formular für die dort platzierten Steuerelemente.

Über die beiden Tabs »Alphabetisch« und »Nach Kategorien« können Sie die Sortierung der Eigenschaftsnamen beeinflussen. Beim ersten Kontakt mit unbekannten Objekten ist eine kategorisierte Anordnung sicherlich hilfreich. Mit der Zeit jedoch werden Sie die Gesetzmäßigkeiten der Namensgebung erkennen und sich nahezu blind in der alphabetischen Sortierung bewegen können.

Die eingeklammerte (Name)-Eigenschaft des betreffenden Objekts beinhaltet einen Namen, unter dem das Objekt im Code angesprochen werden kann. Es existiert ein leider nicht einheitliches Regelwerk für die Namensbildung, auf das wir noch eingehen werden. Einige Vereinbarungen sind im Projekt-Explorer, in dem die Objekte übrigens alphabetisch nach diesen Namen sortiert erscheinen, zu erkennen. Tabellenblätter sind mit dem Präfix *sht* dargestellt, ungebundene Module beginnen mit *mod* und Klassenmodule mit *cls*. Da aber das Objekt, welches die Arbeitsmappe selbst repräsentiert, sinnvollerweise am Anfang der Liste erscheinen soll, so ergibt sich ein Konflikt mit dem Präfix, der eigentlich *wbk* lautet. Dies

hätte jedoch die Anordnung am unteren Ende der Liste zur Folge. Da sich jedoch bei der Referenzierung von Arbeitsmappen andere Möglichkeiten auftun, kann die Arbeitsmappe problemlos als »Arbeitsmappe« oder »Diese Arbeitsmappe« als Voreinstellung bezeichnet werden.

Die in diesem Fenster einstellbaren Eigenschaften können in folgende Gruppen untergliedert werden:

- ▼ Eigenschaften mit (mehr oder weniger) frei definierbaren Werten,
- ▼ Eigenschaften mit vordefinierten Werten und
- ▼ Eigenschaften, die über einen eigenen Dialog verfügen.

Bei Ersteren können Sie den Wert einfach eintragen. Existieren vordefinierte Werte wie zum Beispiel `True` oder `False`, so erscheint bei Aktivierung (Fokuserhalt) eine Auswahl Schaltfläche am rechten Rand der Eigenschaft. In diesem Falle können Sie sich per Doppelklick durch die einzelnen Auswahlwerte bewegen oder durchhangeln. Eigenschaften mit eigenem Dialog sind beispielsweise solche, die in einer Dateiauswahl münden oder Ihnen Einstellungen der Schrift ermöglichen. Diese sind daran erkennbar, dass am rechten Rand eine Schaltfläche mit drei Punkten auftaucht. Ein Klick hierauf produziert den betreffenden Dialog auf den Bildschirm.

Noch etwas zum Handling beim Verändern von frei definierbaren Eigenschaften. Natürlich können Sie den Cursor in das Feld platzieren und mittels Löschen- oder Rück-Taste Zeichen für Zeichen entfernen. Auch das Markieren mit der Maus funktioniert natürlich. Die einfachste Vorgehensweise ist jedoch, den Namen der Eigenschaft anzuklicken. Die Tabulator-taste bringt Sie anschließend in das Eigenschaftsfeld, wobei der komplette Inhalt dann bereits markiert ist und durch das nächste Zeichen überschrieben wird. Probieren Sie es einfach mal aus.

2.3 Codefenster

Obwohl uns die moderne Entwicklungsumgebung von VBA eine Menge Arbeit abnimmt und auch angesichts früherer Arbeitsbedingungen für uns Entwickler so manche Codezeile erspart, kommen wir jedoch nicht umhin, die eine oder andere Programmzeile selbst zu schreiben. Und hierfür stehen uns die gebundenen und freien Module sowie die Klassenmodule zur Verfügung. Für jedes der im Projekt verfügbaren Module wird auf Anforderung ein eigenes Codefenster mit dem entsprechenden Modul geöffnet.

Im Kontextmenü des Projekt-Explorers existieren hierzu der Eintrag »Code anzeigen« und eine eigene Schaltfläche in der linken oberen Ecke. Das bequemste ist jedoch der Doppelklick auf das betreffende Objekt im

Explorer, was bei den Microsoft Excel Objekten, den Modulen und den Klassenmodulen zum gewünschten Ergebnis führt. Lediglich bei Formularen bringt der Doppelklick nicht das Codefenster, sondern das Formular selbst in den Vordergrund. Ein Doppelklick auf das Formular wiederum erzeugt das gebundene Codefenster.

Den Anhängern von Tastaturkommandos (ShortCuts) sei die **[F7]**-Taste ans Herz gelegt, die bei allen Objekten des Projekt-Explorers das dazugehörige Codefenster öffnet.

Hier nun ein Beispiel eines Codefensters.

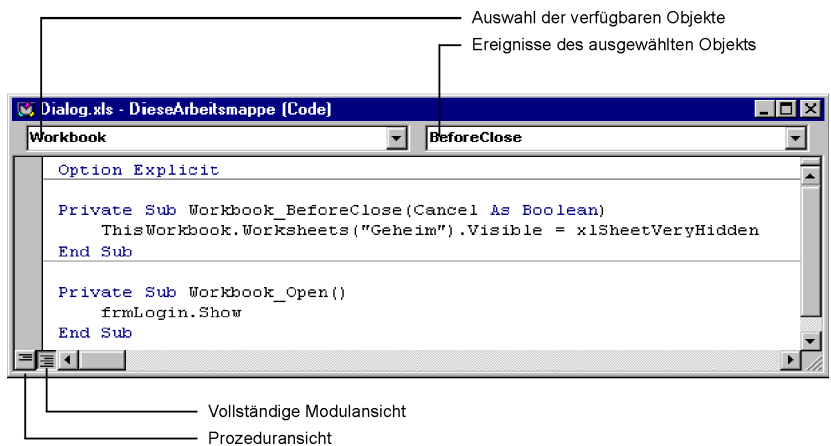


Abbildung 2.6: Codefenster

Unterhalb der Titelleiste befindet sich eine ComboBox, die den Eintrag »(Allgemein)« enthält sowie alle im Kontext enthaltenen Objekte, die Ereignisse produzieren können. In der daneben stehenden ComboBox sind alle Ereignisse des jeweiligen Objektes verfügbar.

In der linken unteren Ecke sind zwei Schaltflächen, mit denen die Darstellung innerhalb des Codefensters gesteuert werden kann. Zu sehen ist in der Abbildung die vollständige Modulansicht, durch die Sie sich wie in einem beliebigen Editor durch den Code bewegen können. Die Prozeduransicht hingegen zeigt jeweils nur die Prozedur, die durch die beiden ComboBoxes spezifiziert sind. Da man jedoch öfter etwas in anderen Prozeduren nachschauen muss, ist die vollständige Modulansicht die bessere Wahl.

2.3.1 Kontextmenü des Codefensters

Hier nun das Kontextmenü des Codefensters, wenn derzeit kein Code ausgeführt wird:

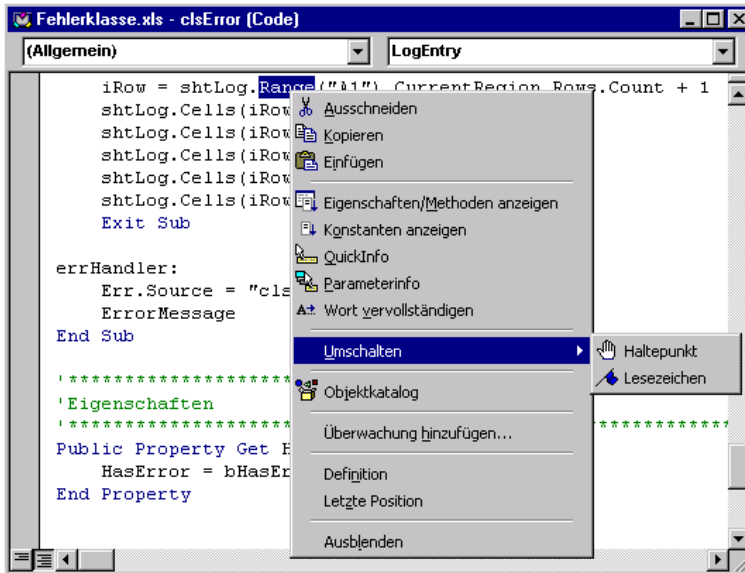


Abbildung 2.7: Kontextmenü des Codefensters

Ausschneiden Strg X

Schneidet die aktuelle Markierung aus und hinterlegt sie in der Zwischenablage.

Kopieren Strg C

Kopiert die aktuelle Markierung in die Zwischenablage.

Einfügen Strg V

Fügt an der betreffenden Stelle ein vorher in die Zwischenablage hinterlegtes Codefragment ein.

Eigenschaften/Methoden anzeigen Strg J

Befindet sich der Cursor an einer Stelle eines Ausdrucks, an dem Eigenschaften und Methoden zur Verfügung stehen, so werden diese in einem Kontextfenster angezeigt. Ein Doppelklick auf einen solchen Eintrag ersetzt den aktuellen Ausdruck durch den so ausgewählten. Die Tabulatortaste führt übrigens zum selben Ergebnis.

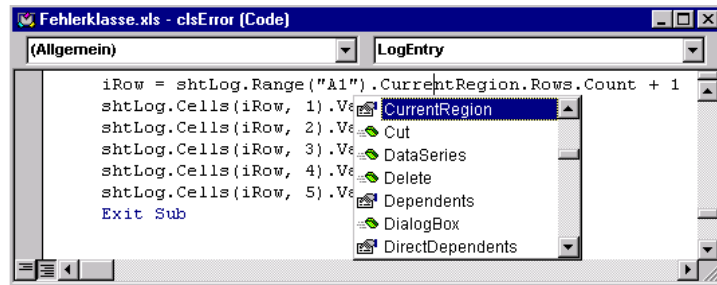


Abbildung 2.8: Eigenschaften/Methoden anzeigen

Konstanten anzeigen

Wenn ein ganzzahliger numerischer Ausdruck, also eine Variable, eine Eigenschaft oder ein Argument einer Methode des Datentyps Byte, Integer oder Long, nicht beliebige Werte enthalten kann, sondern nur einen Wert aus einer Gruppe vordefinierter Werte (Enumerationen), dann kann man sich diese definierten Werte mit diesem Befehl anzeigen lassen.

So darf eine logische Variable nur ein True oder False enthalten. Steht der Cursor rechts des Gleichheitszeichens dieser Zuweisung, so führt dieser Befehl dazu, dass die für diesen Ausdruck definierten Konstanten angezeigt werden:

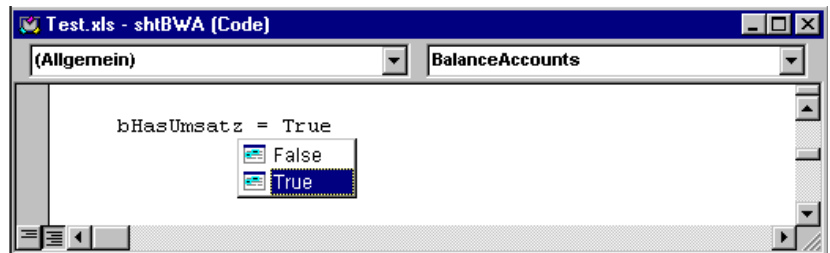


Abbildung 2.9: Konstanten eines logischen Ausdrucks

In der folgenden Abbildung werden die Konstanten des Paste-Arguments der PasteSpecial-Methode angezeigt:

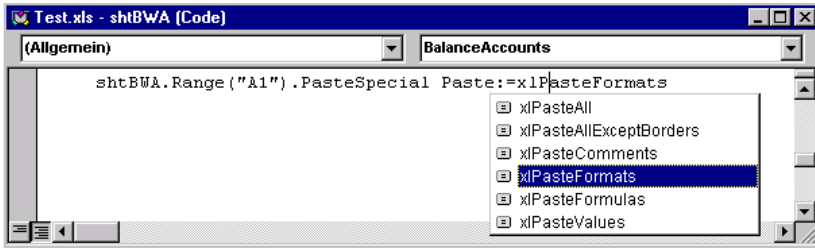


Abbildung 2.10: Konstanten des Paste-Arguments der PasteSpecial-Methode

Der Befehl »Konstanten anzeigen« korrespondiert mit der Option »Elemente automatisch auflisten« im Register Editor des Optionsdialogs. Ist diese Option aktiviert, so werden nach dem Schreiben des den Wert einleitenden Ausdrucks (`>=<` bei Variablen und Eigenschaften, `>:=<` oder `> <` bei Argumenten einer Methode) die Konstanten automatisch auf die oben gezeigte Art und Weise eingeblendet.

Da die Konstanten beim Schreiben des Codes automatisch vorgeblendet werden, dient dieser Befehl eigentlich nur dazu, diese Konstanten anzeigen zu lassen, wenn der betreffende Ausdruck bereits fertig ist, also beispielsweise, um sich die Alternativen noch einmal anzusehen.

Quickinfo Strg I

Je nachdem, wo sich der Cursor befindet, führt die Quickinfo zu verschiedenen Ergebnissen. Steht der Cursor auf einer Variablen oder einem Eigenschaftsnamen, so erscheint die Definition des Ausdrucks:

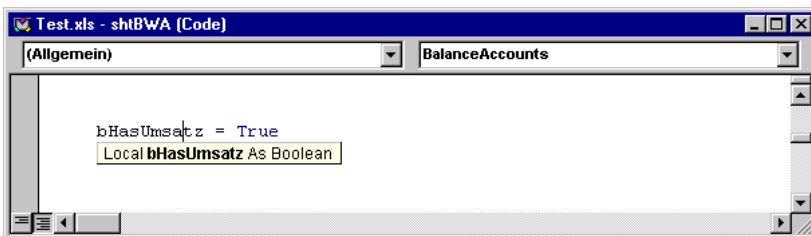


Abbildung 2.11: Quickinfo einer Variablen

Hier sehen wir also, dass es sich um eine lokale Variable des Typs `Boolean` handelt.

Befindet sich der Cursor auf einer Argumentkonstanten einer Methode, so erscheint der Wert der Argumentkonstanten:

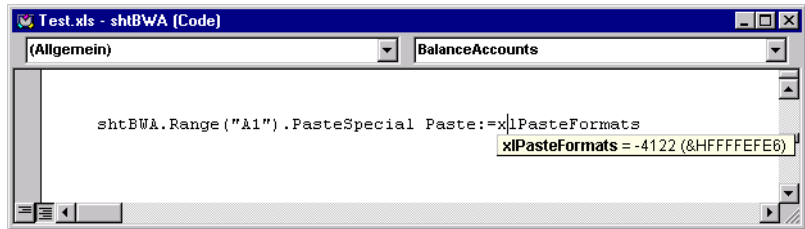


Abbildung 2.12: Quickinfo auf einer Argumentkonstanten

Eine vollständige Methodensignatur hingegen erhalten Sie, wenn sich der Cursor in einem Methodennamen oder dem Argumentnamen einer Methode befindet. Im zweiten Fall ist das entsprechende Argument fett hervorgehoben:

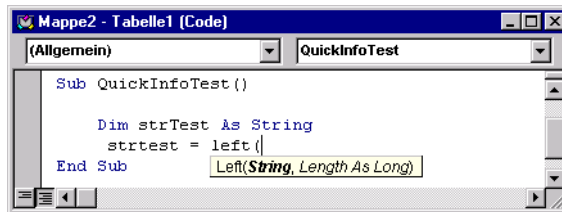


Abbildung 2.13: Quickinfo auf einem Argumentnamen einer Methode

Ob es sich bei der Methode um eine solche handelt, die Bestandteil eines Microsoft Objekts ist, oder eine selbst verfasste Methode (oder auch Funktion), ist hierbei ohne Bedeutung.

Parameterinfo

Die Parameterinfo verhält sich im Prinzip wie die Quickinfo auf dem Argumentnamen einer Methode. Der Punkt ist aber, dass Sie bei Argumenten den Argumentnamen *nicht* angeben *müssen*. Steht der Cursor nun auf der Argumentkonstanten, so sehen Sie eben nur den Wert der Konstanten, wie in Abbildung 2.12 zu sehen. Wählen Sie hingegen die Parameterinfo, so erscheint das Infowindow wie in Abbildung 2.13.

Wort vervollständigen oder

Ist die Option »Elemente automatisch auflisten« im Register Editor des Optionsdialogs aktiviert, so werden nach Eingabe des Gliederungspunktes in einer Objektkette alle verfügbaren Elemente, also Eigenschaften oder Methoden, dargestellt, die für das voranstehende Element möglich sind. Die Eingabe eines jeden weiteren Zeichens führt dazu, dass das erste Element markiert wird, dessen Anfangsbuchstabe mit dem eingetippten Fragment übereinstimmt. Ist dieses Fenster auf dem Bildschirm, so kön-

nen Sie auch mit den Cursortasten oder der Maus in dieser Auswahl navigieren:

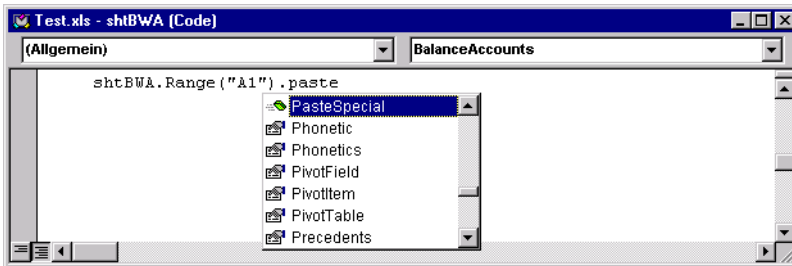


Abbildung 2.14: Wort vervollständigen

Die Tastenkombination **Strg** **Leerzeichen** vervollständigt das begonnene Element, ebenso übrigens die **Tab**-Taste.

Umschalten – Haltepunkt **F9**

Ein Haltepunkt in einer Programmzeile bewirkt, dass das Programm an dieser so markierten Stelle stehen bleiben wird, und zwar vor Ausführung der betreffenden Zeile.

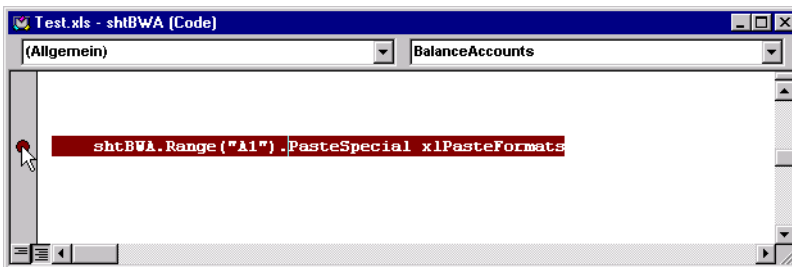


Abbildung 2.15: Haltepunkt

Sie setzen einen Haltepunkt mit diesem Befehl oder entfernen einen gesetzten Haltepunkt durch diesen Befehl. Durch einen Klick auf den linken Steg im Codefenster (siehe Abb. 2.15) können Sie einen Haltepunkt ebenfalls setzen bzw. zurücksetzen. Voraussetzung für die Mausvariante allerdings ist, dass Sie die CheckBox-Kennzeichenleiste im Register Editorformat des Optionsdialogs nicht deaktiviert haben.

Die farbliche Darstellung eines Haltepunktes können Sie im Register Editorformat des Optionsdialogs verändern, was aber nicht unbedingt sinnvoll ist.

Umschalten – Lesezeichen

Ein Lesezeichen ist eine Markierung, die Sie in einer beliebigen Zeile eines der im Projekt-Explorer verfügbaren Codemodule positionieren können. Die Befehle »Nächstes Lesezeichen« und »Vorheriges Lesezeichen« im Menü BEARBEITEN | LESEZEICHEN ermöglichen ein schnelles Wechseln zu den anderen durch Lesezeichen markierten Zeilen. Etwas schneller geht es allerdings mit den entsprechenden Schaltflächen der Symbolleiste Bearbeiten.

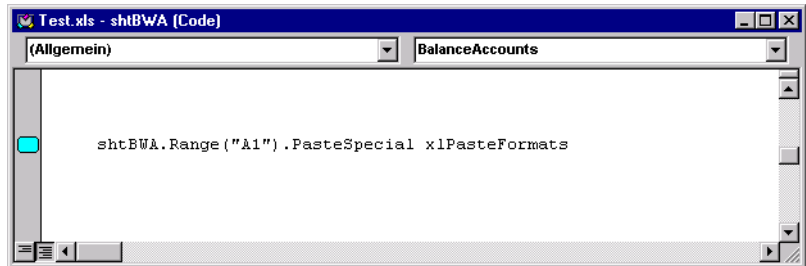


Abbildung 2.16: Lesezeichen

Objektkatalog F2

Dieser Befehl bringt den Objektkatalog auf den Bildschirm, der in Kapitel 2.7 ausführlich dargestellt ist.

Überwachung hinzufügen

Überwachungen und das Hinzufügen von Überwachungen sind in Kapitel 2.6 erklärt.

Definition Umschalt F2

Befindet sich der Cursor auf einer Variablen, so wechselt der Cursor des Codefensters zu der Programmzeile, in der die Definition der aktuellen Variablen erfolgt.

Haben Sie hingegen einen VBA-Befehl, ein Objekt, eine Eigenschaft oder eine Methode ausgewählt, so wird der Objektkatalog angezeigt und das betreffende Element selektiert.

Letzte Position

Dieser Befehl springt zu der letzten Zeile innerhalb des aktuellen Codemoduls, an der die betreffende Variable verwendet wird.

Ausblenden

Schließt das aktuelle Codefenster

2.3.2 Kontextmenü des Codefensters im Haltemodus

Befindet sich der Code im Haltemodus, so erscheint folgendes Kontextmenü:

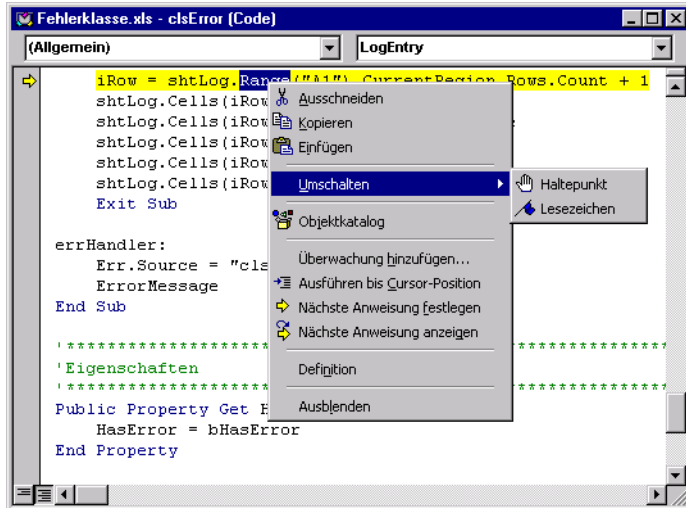


Abbildung 2.17: Kontextmenü des Codefensters im Haltemodus

In Abbildung 2.17 ist ein Prozedurteil zu sehen, bei dem die Ausführung des Codes vor der Abarbeitung der markierten Zeile unterbrochen wurde.

Hier nun die Erläuterungen zu den Befehlen, die nicht bereits unter 2.3.1 erklärt wurden.

Ausführung bis Cursor-Position Strg F8

Würde sich der Cursor in einer Codezeile unterhalb der aktiven Codezeile befinden, so würde der Code bis zu dieser Zeile weiter ausgeführt und würde vor der Bearbeitung dieser Zeile wieder in den Haltemodus wechseln.

Nächste Anweisung festlegen Strg F9

Befindet sich der Cursor in einer anderen als der aktiven Zeile, so wird diese Zeile zur Nächsten. Das heißt, es wird kein Code dazwischen ausgeführt, sondern diese neue Zeile wird als Nächstes abgearbeitet, sobald der Haltemodus beendet wird.

Von der aktiven Zeile aus gesehen nach unten macht dieser Befehl zu meist wenig Sinn. Anders verhält es sich, wenn eine bereits abgearbeitete Zeile auf diesem Weg noch einmal zur Ausführung kommt. Wenn Sie beispielsweise in einer bereits bearbeiteten Zeile den Inhalt einer Zelle einer

Tabelle eingelesen haben und dieser nicht Ihren Erwartungen entspricht, so können Sie, während sich das Programm im Haltemodus befindet, den Zellinhalt verändern und die diese Zeile einlesende Zeile noch einmal durchlaufen lassen. Diese Zeile kann in der Zwischenzeit sowohl in Excel selbst als auch durch eine entsprechende Anweisung im Direktfenster geändert worden sein.

Neben der beschriebenen Vorgehensweise per Menü- oder Kontextmenübefehl existiert noch eine Variante, die per Maus wesentlich schneller und eleganter funktioniert. Die aktive Zeile ist, wie Sie bereits sahen, gelb hinterlegt, und die Kennzeichenleiste zeigt einen gelben, in Richtung Code zeigenden Pfeil. Diesen wiederum können Sie mit gedrückter linker Maustaste auf dieser Kennzeichenleiste nach oben oder unten verschieben:

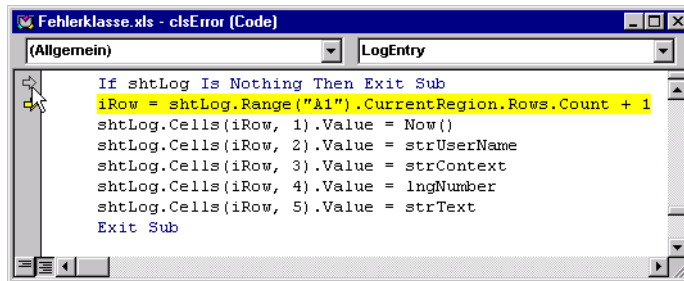


Abbildung 2.18: Nächste Anweisung per Maus festlegen

2.4 Direktfenster

Im Direktfenster können Sie jederzeit einzelne Anweisungen ausführen oder sich Rückgabewerte von VBA-Funktionen ansehen:

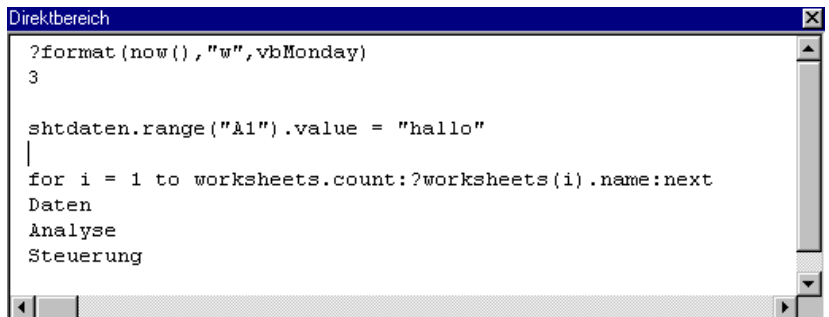


Abbildung 2.19: Direktfenster mit Beispielanweisungen

Schreiben Sie hierzu die Anweisung in eine neue Zeile und betätigen Sie die Return-Taste. Hierbei muss der Cursor übrigens nicht am Ende der Zeile stehen, denn Return bewirkt keinen Zeilenumbruch, sondern lediglich das Ausführen der Anweisungen der betreffenden Zeile.



Nun zu den Beispielen der Abbildung. Im ersten Fall wird die Nummer des Wochentages des aktuellen Datums ermittelt:

```
?format(now(),"w",vbMonday)
```

Das Fragezeichen am Anfang der Zeile beinhaltet die Aufforderung an VBA, das Ergebnis der nachfolgenden (in der nächsten Zeile) auszugeben.

Die anschließende Anweisung bewirkt, dass in Zelle A1 der durch `shtDaten` referenzierten Tabelle der Wert »hallo« eingetragen wird. Da hier keine Rückgabe gewünscht wird, fehlt auch das Fragezeichen. Wenn Sie vor diese Zeile dennoch ein Fragezeichen setzen und die Anweisung ausführen lassen, erscheint in der nächsten Zeile ein Wahr oder Falsch, je nachdem, ob die Aussage »In Zelle A1 steht "hallo"« wahr oder falsch ist.

Dass sogar mehrzeilige Anweisungen ausgeführt werden können, zeigt das nächste Beispiel:

```
for i = 1 to worksheets.count:?worksheets(i).name:next
```

Die entscheidenden Zeichen sind die beiden Doppelpunkte. Dieses ansonsten höchst überflüssige Relikt aus alten Basic-Tagen tut hier gute Dienste. Interessant und logisch ist hierbei übrigens, dass die Variable `i` nicht vorher deklariert werden muss, unabhängig davon, ob die betreffende Option »Variablendeklaration erforderlich« aktiviert ist oder nicht.

Diese Anweisung jedenfalls listet nacheinander die Namen aller Tabellenblätter auf, und zwar dem Index folgend von links nach rechts.

Das bisher zum Direktfenster Gesagte funktioniert unabhängig davon, ob gerade ein Programm *aktiv* ist oder nicht. Ist ein Programm aktiv, so muss sich dieses im Unterbrechungsmodus befinden, damit das Direktfenster genutzt werden kann. Unterbrechungsmodus bedeutet, das Programm wurde angehalten, verfügt aber über alle im Code vereinbarten Variablen und Konstanten. Wir werden darauf im nächsten Kapitel anhand eines konkreten Beispiels wieder zurückkommen.

Unterbrechungsmodus

Hier nun das Kontextmenü des Direktfensters:

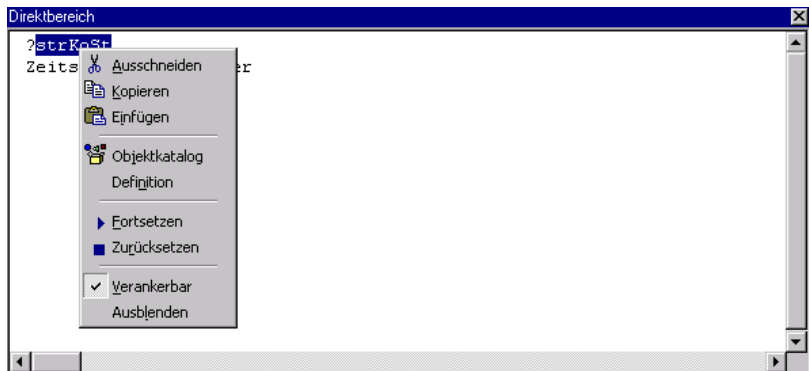


Abbildung 2.20: Kontextmenü des Direktfensters

Die beiden neuen Befehle dieses Kontextmenüs sind Fortsetzen und Zurücksetzen, die jedoch nur dann verfügbar sind, wenn sich das Programm im Haltemodus befindet. Fortsetzen setzt das Programm fort, Zurücksetzen bricht es ab.

2.5 Lokal-Fenster

Das Lokal-Fenster zeigt Ihnen alle lokalen Variablen, also die, die in den gerade im Unterbrechungsmodus befindlichen Routinen deklariert sind. Und nur die!

Das Lokal-Fenster ist nur eines von mehreren Möglichkeiten, wenn es darum geht, sich Auskunft über den Zustand der einen oder anderen Variablen einzuholen. Seine Stärke spielt es aus, wenn es darum geht, sich einen Überblick über die lokalen Variablen einer zumeist auch größeren Prozedur zu verschaffen:

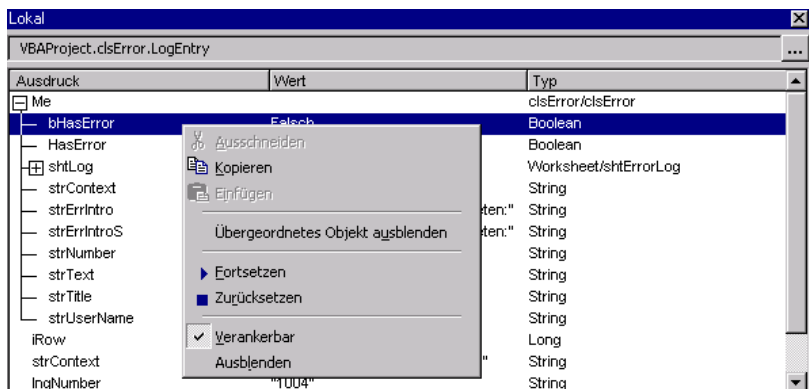


Abbildung 2.21: Lokal-Fenster

In der Abbildung sind alle lokalen Variablen der Prozedur `VerdichteUmsatze` zu sehen, die im gebundenen Modul `shtAW` des Projekts `VBAProject` enthalten sind. Hinter den mit einem Plus-Zeichen versehenen Objekten befindet sich eine Reihe von Eigenschaften, die den jeweiligen Objekten zugeordnet sind. Da diese hierarchische Sammlung auch Aufwärtsreferenzen enthält, können Sie sich im Prinzip durch alle Einstellungen des gesamten Projektes durchhangeln. Das mag vielleicht ganz interessant sein, um sich ein paar Abhängigkeiten vorführen zu lassen, wird aber auch leicht unübersichtlich. Daher ist es in der Regel einfacher, sich im Direktfenster den gewünschten Begriff gezielt anzeigen zu lassen.

Der einzige neue Befehl des Kontextmenüs ist [in der Hierarchie] »Übergeordnetes Objekt ausblenden«. Im aktuellen Beispiel würden die untergeordneten Elemente des `Me`-Knotens verschwinden:

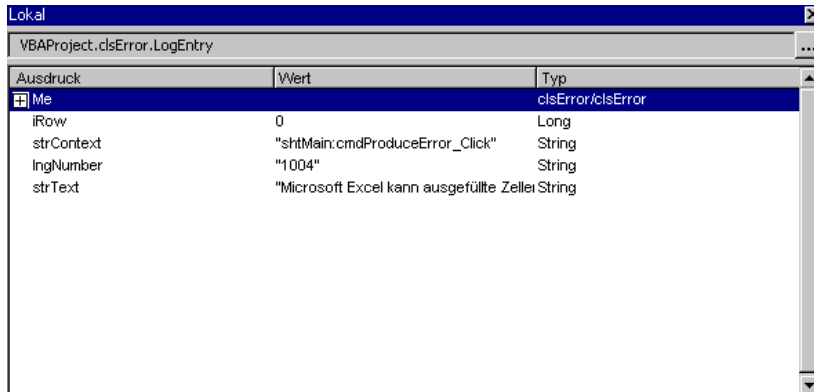


Abbildung 2.22: Lokal-Fenster, übergeordnetes Element ausgeblendet

2.6 Überwachungsfenster

Das Lokal-Fenster ermöglicht einen sofortigen Zugang zu allen lokalen Variablen, wie wir gesehen haben. Das Überwachungsfenster ist hinsichtlich seiner Bedienung etwas aufwendiger, bietet aber eine größere Variabilität.

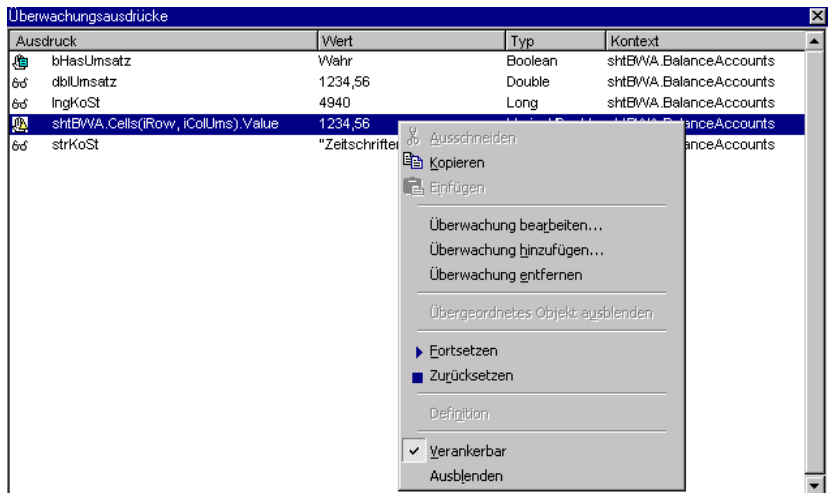


Abbildung 2.23: Überwachungsfenster

Schauen wir uns zunächst die Spalten und ihre Bedeutung an.

2.6.1 Bedeutung der Spalten

Ausdruck

Hiermit sind alle Ausdrücke gemeint, die einen Wert beinhalten können. `strKoSt` ist beispielsweise eine Variable, die natürlich einen Wert haben kann, aber auch `shtBWA.Cells(iRow, iColUms).Value` als ein Bezug zu einer Zelle kann mit einem Wert aufwarten, da es sich um einen Ausdruck handelt, der den Inhalt einer Tabellenzelle referenziert.

Wert

Unter Wert versteht man den Inhalt eines Ausdrucks. Die Variable `lngKoSt` hat in unserem Beispiel den Wert 4940, `strKoSt` den Wert »Bücher, Zeitschriften«. Haben Sie einen Überwachungsausdruck gewählt, auf den im aktuellen Kontext nicht zugegriffen werden kann, so erscheint in der Spalte Wert

<Ausdruck in diesem Kontext nicht definiert>

Typ

Diese Spalte enthält den Datentyp des Ausdrucks. Diese Datentypen werden im Kapitel 4 behandelt.

Kontext

Im Kontext ist der strukturelle Bezug des Überwachungsausdrucks dargestellt, der sich aus dem Namen des Codemoduls und dem Prozedurnamen

zusammensetzt. Im Beispiel handelt es sich um eine Prozedur namens `BalanceAccounts`, die sich in dem an die Tabelle `shtBWA` gebundenen Code-modul befindet.

2.6.2 Überwachungsausdrücke

Überwachung bearbeiten ... Strg W

Dieser Befehl bringt den folgenden Dialog auf den Bildschirm:

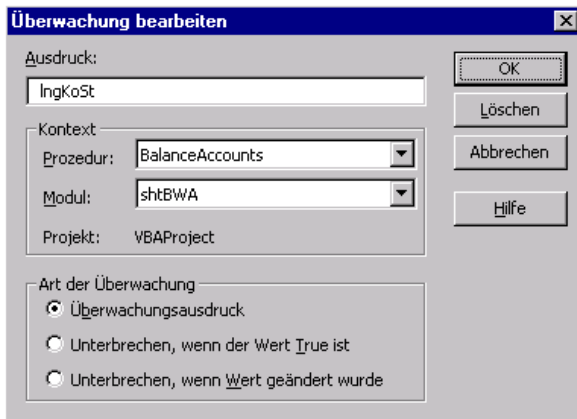


Abbildung 2.24: Überwachung bearbeiten

Im Feld `Ausdruck` steht der veränderbare Ausdruck. Sie können also in diese Textbox einen völlig anderen Ausdruck hineinschreiben.

In der `ComboBox-Prozedur` stehen alle Prozeduren des aktuellen Moduls sowie an oberster Stelle »(alle Prozeduren)« zur Auswahl. Wenn Sie Ausdrücke aus der aktuellen Prozedur hinzufügen, so werden sie auch mit dem aktuellen Prozedurkontext in die Überwachungsliste aufgenommen. Stehen dort jedoch noch »ältere« Einträge aus anderen Prozeduren, so ist dort dieser Prozedurkontext zu anzutreffen. Handelt es sich bei dem überwachten Ausdruck um einen `Private` oder `Public` definierten Wert, können Sie beispielsweise »(alle Prozeduren)« oder die aktuelle Prozedur zuweisen, um den Wert anzeigen zu lassen. Auf gleiche Weise ist auch die `ComboBox` des Modulkontexts zu verstehen, denn Sie haben damit Zugriff auf öffentliche (`Public`) Ausdrücke anderer Module.

Unter »Art der Überwachung« stehen die folgenden Optionsfelder zur Auswahl bereit:

- ▼ Überwachungsausdruck
- ▼ Unterbrechen, wenn der Wert `True` ist

▼ Unterbrechen, wenn Wert geändert wurde

Die oberste Option zeigt den Wert einfach an, wohingegen die beiden anderen eine Unterbrechung der Programmausführung zur Folge haben. Ersterer reagiert, wenn der Wert logisch `True` wird, was natürlich nur bei Ausdrücken des Datentyps `Boolean` oder `Variant` funktioniert. Die letzte Option hingegen unterbricht das Programm, sobald sich der zu überwachende Wert ändert. Diese beiden Optionen sind also sehr hilfreiche Features bei der Fehlersuche, wenn Sie der Stelle auf die Schliche kommen wollen, die den eigentlich erwarteten Wert durch einen anderen ersetzt.

Überwachung hinzufügen ...

Wenn Sie eine Überwachung hinzufügen, haben Sie in der Regel einen gültigen Überwachungsausdruck markiert, der dann automatisch in der Textbox Ausdruck erscheint. Die OK-Schaltfläche fabriziert den Ausdruck anschließend in das Überwachungsfenster. Bei 1-gliedrigen Ausdrücken, also Variablen, reicht es, den Cursor im Ausdruck stehen zu haben, ohne den ganzen Ausdruck markiert zu haben.

Es gibt jedoch zwei weitere Möglichkeiten, einen Ausdruck in der Überwachungsliste hinzuzufügen. Haben Sie einen Ausdruck markiert, so können Sie mit (Umschalt)(F9) den Dialog »Aktuellen Wert anzeigen« hervorlocken. Neben der Anzeige des Wertes steht die Schaltfläche Hinzufügen bereit, mit der dieser Ausdruck nun in die Überwachungsliste wandert.

Die nächste, zweifellos eleganteste Möglichkeit besteht darin, den Ausdruck im Code zu markieren und per Drag & Drop in das Überwachungsfenster zu ziehen. Hierzu muss das Überwachungsfenster jedoch sichtbar sein, was bei den anderen beiden Wegen nicht erforderlich ist.

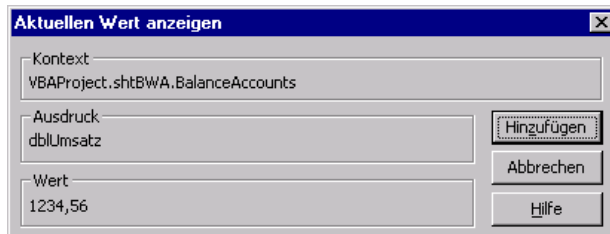


Abbildung 2.25: Dialog Aktuellen Wert anzeigen

Überwachung entfernen

»Überwachung löschen« entfernt die aktuellen Zeilen aus der Überwachungsliste, übrigens ohne Warnung. Die Funktion »Rückgängig« des Menüs Bearbeiten macht das Löschen auch nicht mehr ungeschehen.

2.7 Objektkatalog

Der Objektkatalog (siehe Abb. 2.26) ist eine Art Vorstufe der Hilfe. Er zeigt zwar keine Beschreibung des gesuchten Elements, wohl aber seine Stellung innerhalb des jeweiligen Objektbezugs, was vielfach ausreichend ist, wie wir noch sehen werden.

Die oberste ComboBox bietet, solange keine Verweise hinzugefügt oder entfernt wurden, die folgenden Bibliotheken zur Auswahl:

- ▼ **<Alle Bibliotheken>** als Vereinigungsmenge aller weiteren Bibliotheken
- ▼ **Excel**, die alle Excel-spezifischen Objekten und deren Elemente enthält
- ▼ **Office** mit Office-übergreifenden Objekten, wie etwa CommandBars
- ▼ **StdOLE**, die uns hier nicht interessiert
- ▼ **VBA** mit allen VBA-spezifischen Sprachelementen
- ▼ **VBAProject**, welches die Codestruktur des aktuellen Projekts wiedergibt

Je nach dem, welche Bibliothek Sie ausgewählt haben, erscheinen in der Listbox Klassen alle verfügbaren – nennen wir sie fürs Erste – Objekte der gewählten Bibliothek.

Hier nun der Objektkatalog:

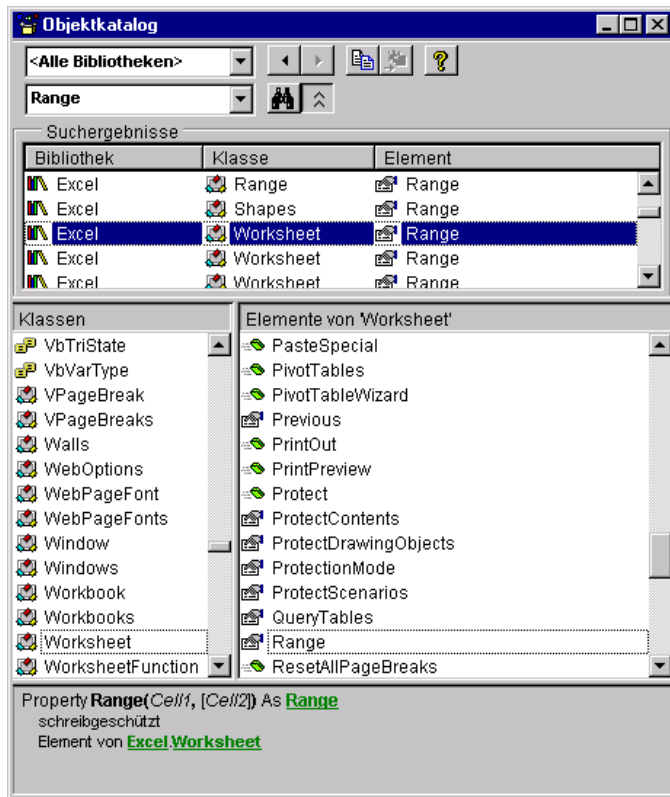


Abbildung 2.26: Objektkatalog

Die Abbildung 2.27 zeigt die Bedeutung der verwendeten Symbolik in diesem Dialog. Wichtig in dieser Übersicht sind unter der Rubrik Klassen die Begriffe Klasse, Modul und Aufzählung (Enumeration) und bei den Elementen die Begriffe Eigenschaft, Ereignis, Konstante und Methode.

- ▼ **Klasse** steht für Objekt oder Auflistung
- ▼ Unter **Modul** sind Anweisungen und Funktionen der VBA-Bibliothek thematisch geordnet
- ▼ Eine **Enumeration** enthält Konstanten für ein Argument einer Methode oder Funktion
- ▼ **Ereignisse** sind Prozeduraufrufe, die von Objekten angestoßen werden.
- ▼ Als **Eigenschaft** bezeichnet man einen speziellen Wert eines Objektes
- ▼ **Methoden** sind Prozeduren, die von Objekten ausgeführt werden.

Symbol:	Bedeutung:
	Eigenschaft
	Standardeigenschaft
	Methode
	Standardmethode
	Ereignis
	Konstante
	Modul
	Klasse
	Benutzerdefinierter Typ
	Global
	Bibliothek
	Projekt
	Integrierte Schlüsselwörter und Typen
	Aufzählung (Enum)

Abbildung 2.27: Objektkatalog, verwendete Symbolik

Abgesehen davon, dass der Objektkatalog alle Sprachelemente aller geladenen Bibliotheken aufzulisten vermag, gelingt es uns mit den beiden ListBoxes »Klassen« und »Elemente von ...«, die komplette Hierarchie eines umfangreichen Objektmodells darzustellen.

In Kapitel 5 werden die Begrifflichkeiten rund um Objekt, Ereignisse, Eigenschaften und Methoden gründlich behandelt. Sollte Ihnen an dieser Stelle noch das eine oder andere unklar sein, so ist das kein Problem. Es geht hier nur um die Mechanik des Objektkatalogs. In Kapitel 6 werden wir uns mit dem Excel-Objektmodell, der Excel-Bibliothek also, auseinander setzen.

Am Beispiel des Range-Objekts werden wir die komplette Objektkette in diesem Objektkatalog verfolgen. Das Einzige, was Sie zu Anfang wissen müssen, ist das oberste Objekt der Bibliothek, welches bei Excel das Application-Objekt ist.

Wählen Sie hierzu in der obersten ComboBox die Bibliothek Excel und danach in der linken Listbox die Klasse Application. In der rechten List-

box erscheinen danach alle Elemente, also Ereignisse, Eigenschaften und Methoden. Auf dem Weg vom Application-Objekt zum Range-Objekt ist das Element **Workbooks** der erste Schritt. Die rechte Listbox reagiert auf Tastatureingaben und wählt das Element aus, dessen Wortanfang den eingegebenen Zeichen entspricht. Abbildung 2.28 zeigt das Ergebnis unserer bisherigen Auswahl.

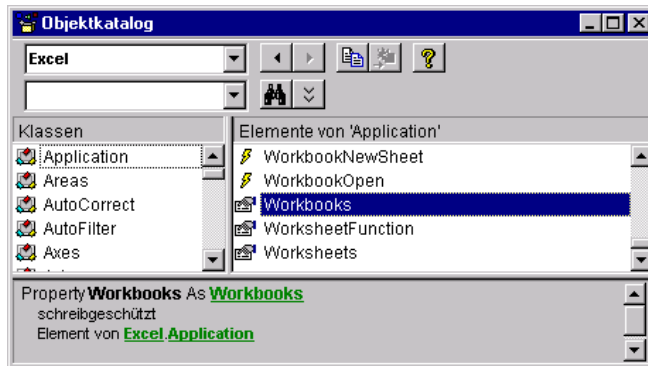


Abbildung 2.28: Objektkatalog, Application-Workbooks-Beziehung

Nun muss man wissen, dass die hier gezeigte **Workbooks**-Eigenschaft als Ergebnis ein einzelnes **Workbook**-Objekt zurückgibt. Diese Zusammenhänge werden in Kapitel 5 genauer beleuchtet.

Um jetzt die Kette weiter zu verfolgen, müssen Sie in der linken Listbox jetzt den Begriff **Workbook** auswählen, woraufhin die rechte Listbox wiederum alle Elemente des **Workbook**-Objekts auflistet. Der nächste Schritt in Richtung **Range**-Objekt ist das **Worksheet**-Objekt. Der Zugriff darauf erfolgt in Analogie zu dem letzten Schritt über die Eigenschaft **Worksheets**:

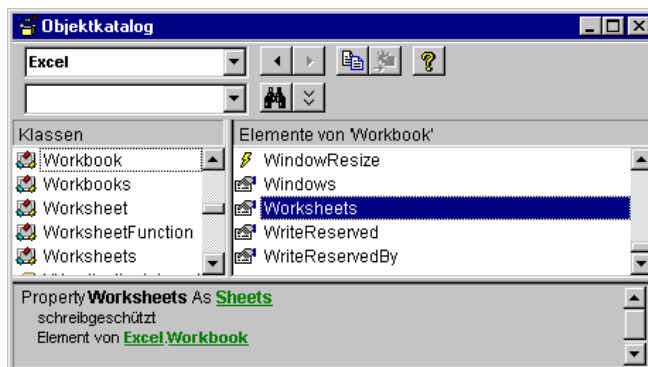


Abbildung 2.29: Objektkatalog, Workbook-Worksheets-Beziehung

Auch hier ist die Rückgabe der Worksheets-Eigenschaft ein einzelnes Objekt des Typs Worksheet, das, in der linken Listbox ausgewählt, wiederum rechts seine Elemente feilbietet:

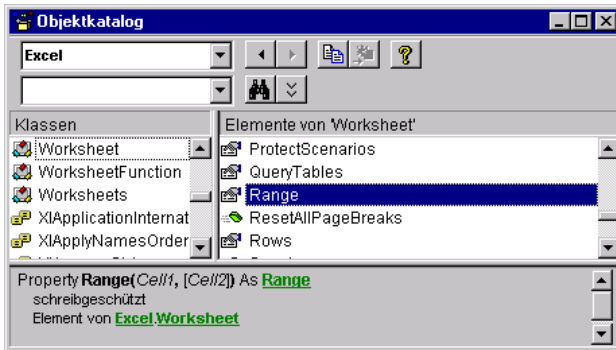


Abbildung 2.30: Worksheet-Range-Beziehung

Wählen Sie nun links das Range-Objekt aus, so zeigt die rechte Listbox alle Elemente des Range-Objekts und wir sind gewissermaßen am Ziel:

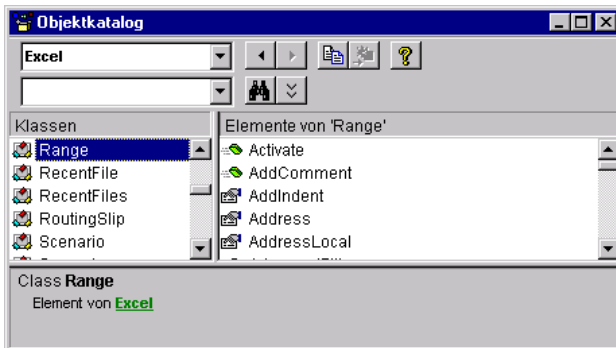


Abbildung 2.31: Das Range-Objekt mit seinen Elementen

Wählen Sie eines der rechten Elemente aus, so können Sie sich durch einen Klick auf das gelbe Fragezeichen die Hilfe zu dem ausgewählten Element anzeigen lassen.

In Abbildung 2.26 ist in der oberen Fensterhälfte ein Grid mit Suchergebnissen zu sehen, auf die der Suchbegriff Range zutrifft. Sie sehen links die entsprechenden Bibliotheksnamen, in der Mitte die Klassennamen und rechts die Elemente, also Ereignisse, Eigenschaften und Methoden. Die Dubletten, die in der Abbildung zu sehen sind, verschwinden, wenn Sie statt <alle Bibliotheken> nur die Excel-Bibliothek auswählen.

2.8 Menüs

Einer ehernen Regel zufolge sind in Programmen alle Funktionen über die Menüs zugänglich, die in Auszügen auf Symbolleisten und in Kontextmenüs vertreten sind. Bei der Behandlung der einzelnen Fenster in den vorangehenden Abschnitten ist uns schon ein Großteil der Menüfunktionen in Form der Kontextmenüs begegnet. So werden hier die Menüfunktionen erläutert, die bislang nicht auftauchten.

2.8.1 Menü Datei

COM Add-Ins, eine der großen Neuerungen in Office 2000, sind ebenso wie die klassischen Add-Ins Erweiterungen für die betreffenden Office-Produkte. Dahinter verbergen sich DLLs, die naturgemäß keinem proprietären Format mehr unterliegen, wie es bei den »alten« Add-Ins der Fall war. Ein in Excel-VBA geschriebenes COM Add-In wird auch in Word laufen.

In Kapitel 16 werden wir uns näher mit den alten und neuen Add-Ins auseinander setzen.

Neues Projekt

Dieser Befehl öffnet einen Dialog zur Auswahl des Projekttyps:

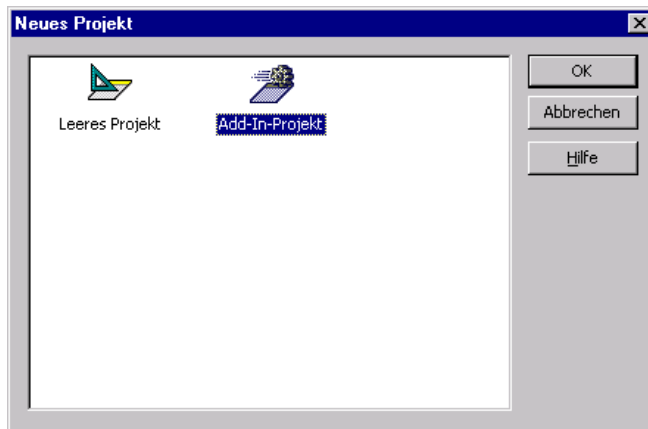


Abbildung 2.32: Dialog Neues Projekt

Projekt öffnen ...

Dahinter verbirgt sich der altbekannte Datei-Öffnen-Dialog, der allerdings mit dem Filter auf den neuen Dateityp *.vba eingestellt ist.

... erstellen

Mit dieser Funktion können Sie die DLL eines Add-Ins erstellen.

2.8.2 Menü Bearbeiten

Das Bearbeiten-Menü enthält viele Standards, wie Rückgängig oder Kopieren, Ausschneiden und Einfügen, über die wir keine Worte verlieren müssen.

Einzug vergrößern Tab

Hiermit wird in Code-Modulen ab der Cursorposition ein Einzug erzeugt, der dem in den Optionen unter Tab-Schrittweite im Register Editor entspricht. Es wird nicht die als Schrittweite vereinbarte Anzahl von Leerzeichen eingefügt, sondern so viele, wie zum Erreichen der nächsten Position erforderlich. Positionen errechnen sich aus: Anzahl der Tabulator x Schrittweite + 1, also 5, 9, 13 etc. Haben Sie mehrere Zeilen markiert, so erstreckt sich diese Funktion auf alle markierten Zeilen und fügt von links die entsprechende Anzahl von Leerzeichen ein, wie im vorherigen Absatz beschrieben.

Als Standard hat sich der auch in den Optionen voreingestellte Wert von 4 Zeichen etabliert. Über weitere Aspekte der Codegestaltung wird im Kapitel 11 – Applikationsdesign zu reden sein.

Einzug verkleinern Umschalt Tab

Hier passiert das Gegenteil von dem, was unter Einzug vergrößern beschrieben ist.

Lesezeichen

Lesezeichen sind eine tolle Erleichterung bei unserer Arbeit. Sie erlauben uns, an beliebigen Stellen in Codemodulen eine beliebige (?) Anzahl von Zeichen zu setzen, die wir mit den Befehlen Nächstes bzw. Vorheriges Lesezeichen bequem anspringen können.

Lesezeichen – Lesezeichen setzen/zurücksetzen

Hiermit setzen Sie in der aktuellen Zeile ein Lesezeichen, ein bereits vorhandenes wird entfernt.

Lesezeichen – Nächstes Lesezeichen

Dadurch wird das nächste Lesezeichen innerhalb des aktuellen Moduls in Richtung Modulende angesprungen. Weist das aktuelle Modul keine weiteren Lesezeichen mehr auf, wird das nächste Modul untersucht.

Lesezeichen – Vorheriges Lesezeichen

Dieser Befehl durchsucht das Modulblatt in Richtung Modulkopf und springt zur nächsten mit Lesezeichen versehenen Zeile. Ist der Modulkopf erreicht, so wird das vorstehende Modul untersucht.

Lesezeichen – Alle Lesezeichen löschen

Alle existierenden Lesezeichen werden hiermit aufgehoben.

2.8.3 Menü Ansicht

Die meisten der hier verfügbaren Befehle haben wir bereits in den Kontextmenüs abgehandelt.

Aufrufeliste [Strg] [L]

Ist Ihr Programm im Haltemodus, so können Sie über die Aufrufeliste den Weg des Programms bis zu der abgegebenen Stelle verfolgen. Das folgende Beispiel ist aus dem Leben gegriffen:

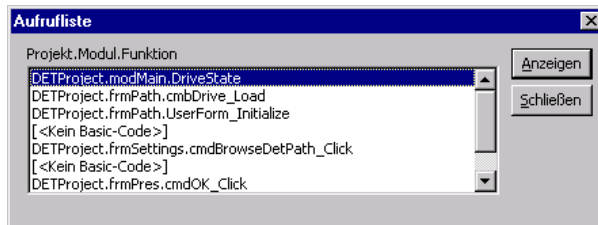


Abbildung 2.33: Aufrufeliste

Wenn Sie nun den zweiten Eintrag auswählen und auf die Schaltfläche Anzeigen klicken, so wird die betreffende Codestelle angezeigt, wobei die aufrufende Zeile durch ein grünes Dreieck gekennzeichnet ist:

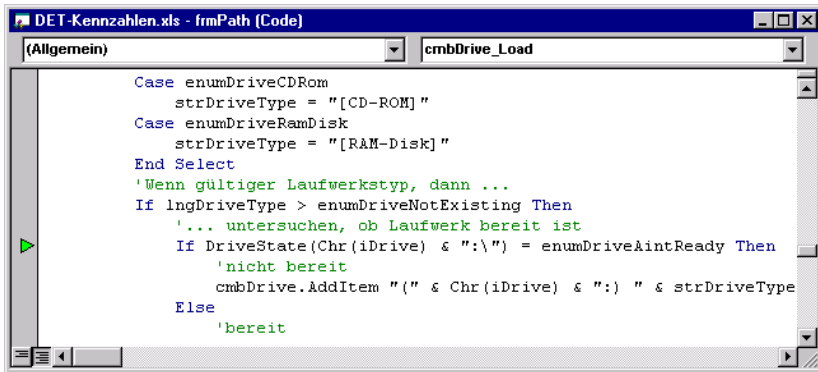


Abbildung 2.34: Aufrufeliste, Anzeigen einer Prozedur

Es kommt vielfach vor, dass Ihnen die fehlerhafte Programmzeile klar ist, Sie aber wissen möchten, von wo aus und unter Umständen mit welchen Argumenten diese Prozedur aufgerufen wird. In diesen Fällen erweist sich die Aufrufeliste als nützliches Hilfsmittel.

Werkzeugsammlung

Die Werkzeugsammlung enthält alle Steuerelemente, die derzeit in eine UserForm eingefügt werden können. Ist dieser Menüpunkt aktiviert, so wird die Werkzeugsammlung korrespondierend mit dem Fokus der UserForm ein- und ausgeblendet. Ist keine UserForm im aktiven Fenster, so kann die Werkzeugsammlung nicht eingeblendet werden.

UserForms und Steuerelemente werden in Kapitel 8 ausführlich behandelt.



Abbildung 2.35: Werkzeugsammlung

Die Werkzeugsammlung verfügt über ein Kontextmenü, mit dem die Konfiguration der Werkzeugsammlung verändert werden kann:

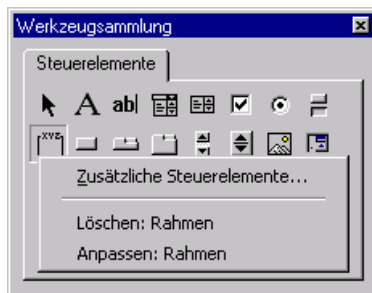


Abbildung 2.36: Werkzeugsammlung, Kontextmenü eines Steuerelements

Mit **Löschen** kann das aktuelle Steuerelement aus der Werkzeugsammlung entfernt werden. Der Punkt **Anpassen** führt zu einer Spielerei, die zu nichts nütze ist und deshalb hier nicht behandelt wird.

Zusätzliche Steuerelemente führt Sie zu einem Dialog, in dem alle Steuerelemente enthalten sind, die in UserForms verwendet werden können. Der Umfang dieser Steuerelemente richtet sich danach, welche Office-Version Sie installiert haben und ob andere Steuerelementbibliotheken auf Ihrem Rechner verfügbar und in der Registry eingetragen sind.

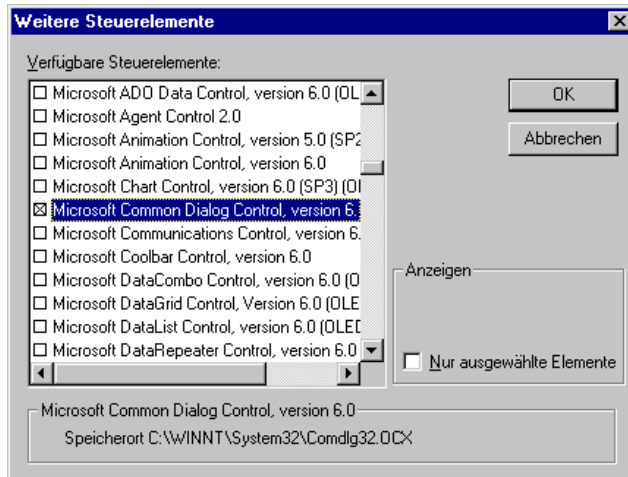


Abbildung 2.37: Dialog Weitere Steuerelemente

Wenn Sie ein weiteres Steuerelement auswählen wollen, klicken Sie in der betreffenden Zeile in die Checkbox.

Befindet sich Ihr Cursor hingegen auf dem Registernamen, so erscheint das folgende Kontextmenü:

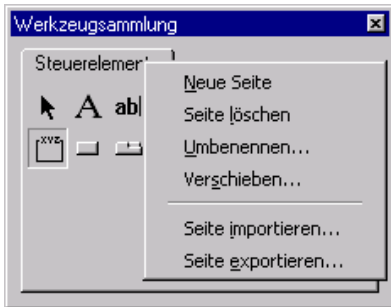


Abbildung 2.38: Werkzeugsammlung, Kontextmenü auf Register

Neue Seite erzeugt eine neue Seite, die mit **Seite löschen** gelöscht und mit **Umbenennen** umbenannt werden kann. Die Reihenfolge der Seiten ändern Sie mit **Verschieben**. **Export**- und **Import**möglichkeiten für ganze Seiten verbergen sich hinter den beiden entsprechenden Menüpunkten.

Die nächste Abbildung zeigt eine Werkzeugsammlung mit Unterscheidung in VBA und VB-Steuerelemente:



Abbildung 2.39: Werkzeugsammlung mit eigener Seite für VB-Steuerelemente

Aktivierreihenfolge

Innerhalb eines Dialoges kann man sich mittels Tabulatortaste von Steuerelement zu Steuerelement bewegen. Die Reihenfolge der Steuerelemente kann durch diesen Dialog festgelegt werden:

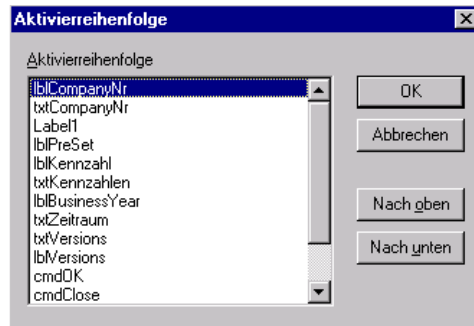


Abbildung 2.40: Aktivierreihenfolge

Wer weiß, wie man sich in Visual Basic mit der `TabIndex`-Eigenschaft herumquälen muss, wird auf den ersten Blick von dieser Variante begeistert sein. Allerdings ist es unverständlich, weshalb auch die Steuerelemente die Liste bevölkern, deren `TabStop`-Eigenschaft auf `False` gesetzt ist. Ebenso unverständlich wie die fehlende Möglichkeit, Steuerelemente so zu deaktivieren, dass sie auch nicht in der Objekt-ComboBox des gebundenen Codemoduls auftauchen.

Mit den beiden Schaltflächen »Nach oben« und »Nach unten« lässt sich die Reihenfolge des ausgewählten Steuerelements in der Liste verändern.

Symbolleisten

Hiermit können Sie die vier Symbolleisten der Entwicklungsumgebung ein- oder ausschalten. Der Menüpunkt `Anpassen` bringt den folgenden Dialog auf den Bildschirm, mit dem die Symbolleisten individuell und dauerhaft konfiguriert werden können:

Die Mechanismen dürften aus den Office-Produkten hinlänglich bekannt sein. Wenn Sie den Dialog aber schon einmal auf dem Bildschirm haben, so können Sie die Schaltfläche `Aufrufeliste` auf eine der anderen Symbolleisten ziehen, da man sie doch recht häufig benötigt.

Im Übrigen erreichen Sie die Funktionalität des Untermenüs `Symbolleisten` des `Bearbeiten`-Menüs etwas bequemer, indem Sie mit der rechten Maustaste auf eine Symbolleiste oder die Menüleiste klicken.

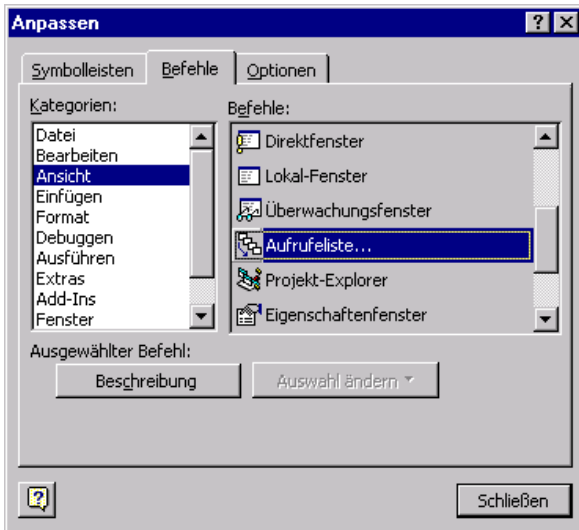


Abbildung 2.41: Dialog Symbolleisten anpassen

2.8.4 Menü Einfügen

Die Befehle Einfügen UserForm, Modul oder Klassenmodul fügen in die aktive Datei eben eines der gewählten Elemente ein. Also nicht weiter aufregend.

Prozedur ...

Frage: Wie fügt man eine Prozedur ein?

Antwort: Indem man sie einfach in das Modul schreibt.

Frage: Wozu ist dann dieser Menüpunkt da?

Antwort: Weiß ich nicht.

Komponenten ... Strg T

Dieser Menüpunkt führt zu einem Dialog, dessen erste Seite identisch ist mit dem unter 2.8.3 beschriebenen Weitere Steuerelemente. Die zweite Seite hingegen enthält Designer, die in ein VBA-Projekt eingefügt werden können:

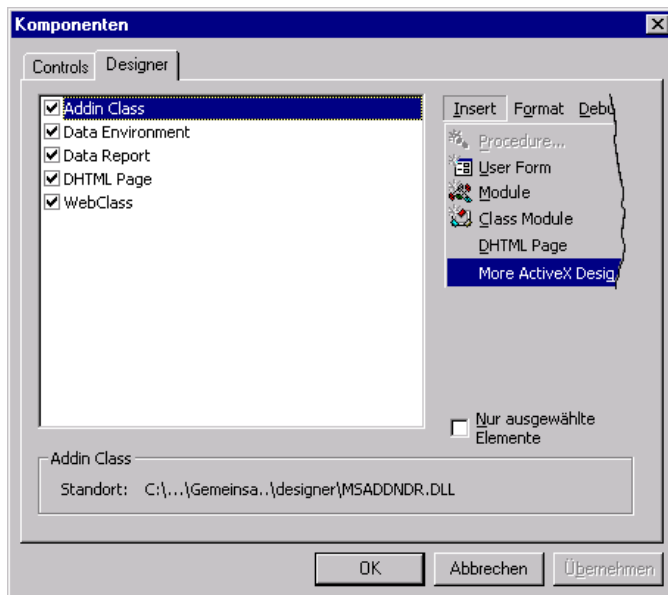


Abbildung 2.42: Dialog Komponenten Designer

Alle hier ausgewählten Komponenten stehen anschließend im Menü Einfügen und im Kontextmenü des Projekt-Explorers als zusätzliche Menüpunkte zur Verfügung. Eine Add-In Class erhalten Sie automatisch, sobald Sie im Menü Datei »Neues Projekt« aufrufen und dort Add-In-Projekt auswählen. Über die restlichen Punkte lässt sich trefflich streiten:

- ▼ Ein Data Environment ist einem in Datenklassen oder gar in DLL's gekapselten Datenzugriff weit unterlegen. Im Prinzip widerspricht es sogar dem neuen DNA-Konzept, (Distributed interNet Applications), von dem die Microsoft Publikationen inzwischen voll sind. Und wenn Sie die ADO im Griff und Ihre Datenschicht sauber strukturiert haben, so benötigen Sie dieses nette Spielzeug nicht.
- ▼ Excel ist für den, der damit umgehen kann, diesem Add-In in Sachen Reportgestaltung haushoch überlegen.
- ▼ So habe ich auch meine Zweifel, ob ein VBA-Projekt der richtige Container für WebClasses und DHTML-Designer ist.

Datei

Wenn Sie sich in einem Codemodul befinden, so können Sie aus einer Textdatei den Code importieren. Hierbei sind in erster Linie keine Textdateien, sondern *.bas und *.cls Dateien gemeint, also Module und Klassenmodule. Der normale Weg zum Import von Codeteilen wird aber wohl der bleiben, dass man wiederverwendbare Codemodule entwirft, die dann als Ganzes dem Projekt hinzugefügt werden. Der Import einer ganzen Datei in ein Modul hinein wird wohl eine seltene Ausnahme bleiben.

2.8.5 Menü Format

Das komplette Format-Menü betrifft nur UserForms oder, besser gesagt, die Steuerelemente auf UserForms. Die Funktionen dienen im Wesentlichen der Ausrichtung, Größenanpassung und Positionierung dieser Steuerelemente.

In meinen nunmehr fünf Jahren der fast ausschließlichen Beschäftigung mit VB und VBA, während der Hunderte von Dialogen entstanden sind, habe ich noch nie die Funktionen des Format-Menüs benutzt. Das Einzige, was man zum effizienten Bauen von Dialogen benötigt, ist eine Spur Gefühl für Ästhetik, einige Designregeln, etwas Routine und ein eingeschaltetes Raster. Unter diesen Voraussetzungen darf man sogar etwas zittern, ohne dass die Steuerelemente wirken, als sei die Form genau über dem Epizentrum eines schweren Erdbebens gelegen. Das Fundament hierzu werden wir in Kapitel 8 legen und vermutlich werden Sie das Format-Menü danach auch nicht mehr benötigen.

2.8.6 Menü Debuggen

Kompilieren von ...

VBA hat viele Stufen der Codeüberprüfung. Zum einen bewirkt das Verlassen einer Codezeile, dass diese syntaktisch überprüft und rot dargestellt wird, sobald einer der folgenden Fehler erkannt wird:

**Syntaxcheck beim
Verlassen einer
Codezeile**

- ▼ Tauchen Sub, Dim, Private, Public, As, Set oder andere Anweisungen oder Teile von Anweisungen auf, so wird die korrekte Syntax der Zeile überprüft.
- ▼ Die Klammerung der Argumente offensichtlicher Funktionen oder Methoden auf der rechten Seite eines Gleichheitszeichens wird überprüft. Ebenso werden alle Zeichen außer dem Komma als Listentrennzeichen moniert.

Hierbei werden alle darin enthaltenen Begriffe mit den Sprachelementen der per Verweis eingerichteten Bibliotheken und den in diesem Kontext gültigen Variablen abgeglichen. Im Falle der Übereinstimmung wird der

Begriff in die Schreibweise umgewandelt, in der dieser Begriff referenziert wurde.

Nicht erkannt werden falsche Funktionsnamen, undefinierte Variablen, falsche Datentypzuweisung oder falsche Methodensignaturen:

```
MsgBox Links("Hallo", 3)
```

geht durch, weil es eine gültige Funktion des Namens `Links` irgendwo im Code geben könnte.

```
Dim lngKoSt as Long
lngKoSt = "Hallo"
```

bleibt als Datentypfehler unbeanstandet.

```
Dim rngDaten As Range
rngDaten.Aktivate
```

wird ebenfalls nicht erkannt, weil die Methodensignaturen nicht anhand der Bibliothek überprüft werden.

```
Left("123", 1) = "5"
```

erregt auch keinen Verdacht, weil es wie ein Teil einer Objektmethode mit weggelassener Standardeigenschaft wirkt.

Codeprüfung vor Abarbeiten der Prozedur

Bevor die Bearbeitung einer Routine begonnen wird, erfolgt eine Überprüfung aller innerhalb der Routine verwendeten Funktionen, Anweisungen und Eigendefinitionen mit Methodencharakter. Hierbei bleibt die fehlerhafte `Links()`-Funktion im oberen Beispiel hängen. Die drei anderen Fehler werden vor der Abarbeitung der betreffenden Zeilen nicht erkannt.

Codeprüfung vor Abarbeiten der Zeile

Sie fallen aber auf, wenn die betreffende Zeile unmittelbar abgearbeitet werden soll.

Unsere drei fehlerhaften Zeilen haben keinen Argwohn erweckt, bis die Prozedur zur Abarbeitung anstand. Mit anderen Worten, eine Programmüberprüfung durchlaufen zu lassen bringt nur dann eventuelle Fehler zu Tage, wenn die betreffende Routine auch bearbeitet wird.

Was tut Kompilieren?

Was tut nun die Funktion Kompilieren von ... im Menü Debuggen? Kann sie uns bei der Überprüfung des vollständigen Codes helfen?

Sie kann, denn sie tut das, was der *Codeprüfung vor Abarbeiten* der Prozedur entspricht, und zwar in allen Routinen. Dabei wird all das übersehen, was auch dieser Prüfung durch die Lappen gehen würde.

Dass es auch anders geht, sind die VB-Entwickler gewöhnt, denn dort überprüft die so genannte vollständige Kompilierung alle Methodensi-

gnaturen. Von unseren Beispielen würde diese vollständige Kompilierung nur die fehlerhafte Datentypzuweisung nicht erkennen. Bleibt zu wünschen, dass die nächste Version mit einer vollständigen Kompilierung aufwarten kann.

Einzelschritt F8

Im Einzelschrittmodus können Sie durch wiederholtes Betätigen der F8-Taste das Programm Zeile für Zeile abarbeiten lassen. Die jeweilig anstehende Programmzeile wird dabei gelb markiert. Stößt VBA hierbei auf eine aufzurufende Routine, wird in diese gewechselt und dort ebenfalls Zeile für Zeile abgearbeitet.

Prozedurschritt Umschalt F8

Der Prozedurschrittmodus verhält sich so wie der Einzelschrittmodus, nur werden aufzurufende Routinen nicht zeilenweise abgearbeitet, sondern die komplette Aufrufzeile wird wie eine Programmzeile behandelt, die Prozedur gewissermaßen als ein Schritt interpretiert.

Prozedur abschließen Strg Umschalt F8

Dieser Befehl schließt die Bearbeitung der aktuellen Prozedur ab und bleibt vor der dem Aufruf dieser Routine folgenden Zeile stehen. Wurde die so abgeschlossene Routine von keiner anderen aufgerufen, so wird das Programm zu Ende bearbeitet.

Ausführen bis Cursorposition Strg F8

Das Programm wird bis zu der Zeile weiter abgearbeitet, in der sich der Cursor befindet. Diese Funktion steht auch im Kontextmenü des Codefensters zur Verfügung, die drei oberen nicht.

2.8.7 Menü Ausführen

Sub/UserForm ausführen (F5)

Ist eine UserForm aktiv, so wird diese gestartet. Befinden Sie sich jedoch in einer Sub-Prozedur (ohne Argumentübergabe) eines Moduls, so wird diese Routine gestartet.

Unterbrechen (Strg)(Unterbrechen)

Mit dieser Funktion wird die Abarbeitung des Programms unterbrochen, kann aber danach wieder fortgeführt werden. Variablen und Objekte behalten hierbei ihre Werte. Das ist beispielsweise dann sinnvoll, wenn sich Ihr Programm in einer Endlosschleife befindet.

Zurücksetzen

Zurücksetzen bricht das Programm ab.

Entwurfsmodus und Entwurfsmodus beenden

Das Einzige, was in der Entwicklungsumgebung Ähnlichkeit mit einem Entwurfsmodus hat, ist das Bearbeiten einer Form. Und die ist per Definition im Entwurfsmodus, wenn das Programm nicht gerade läuft. Also kann man mit dieser Funktion den Entwurfsmodus von Excel, also seiner Tabellen, gewissermaßen fernsteuern. Tolles Feature ;-)

Projekt ausführen

Haben Sie ein Projekt geladen, so wird dieses ausgeführt.

2.8.8 Menü Extras

Verweise ...

Windows ist bis unter den Desktop vollgepackt mit ActiveX-Servern. Word ist zum Beispiel ein solcher ActiveX-Server, Excel ebenso. Aber auch die Datenbankschnittstellen alter (DAO) oder neuer (ADO) Prägung. Wenn es Ihnen hilft, so setzen Sie ActiveX mit OLE2 gleich, denn einige der heutigen ActiveX-Server waren in ihrem früheren Leben OLE-Server, ohne dass sie eine Metamorphose durchmachen mussten.

Ein Server stellt Services, also Dienstleistungen, bereit. So erlaubt der ActiveX-Server Excel jedem Client, der mit ihm Kontakt aufnehmen kann, dessen umfangreichen statistischen Funktionen zu nutzen, Word bietet zum Beispiel die Rechtschreibprüfung feil und die Datenbankschnittstelle managt Datenbankzugriffe für Nicht-Datenbanker.

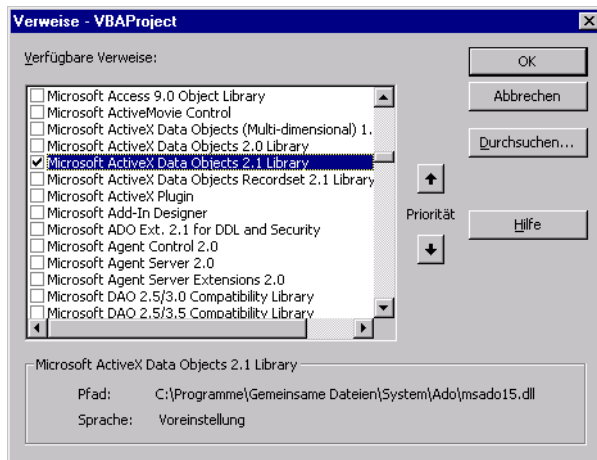


Abbildung 2.43: Dialog Verweise

Im Verweise-Dialog werden alle registrierten ActiveX-Server des Typs DLL, OLB und EXE angeboten. Durch einen Verweis auf eine solche Bibliothek steht uns in unserer Applikation der Funktionsumfang dieses Servers zur Verfügung.

Wenn wir den Eintrag Microsoft ActiveX Data Objects 2.1 Library auswählen, können wir anschließend im Code die Datenbankobjekte der ADO (so heißt das obige Wortungetüm kurz) verwenden und uns dadurch die Welt der kleinen (Access) und großen (SQL-Server, ORACLE) Datenbanken erschließen.

Ein Blick auf den Objektkatalog zeigt uns die neue Vielfalt:

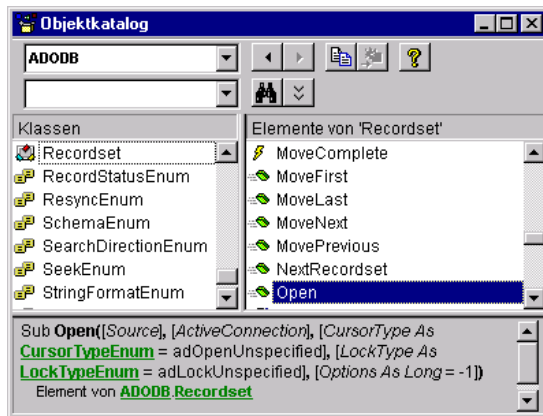


Abbildung 2.44: Objektkatalog mit ADODB-Bibliothek

Dass die ADO-Bibliothek sich hier unter dem Namen ADODB präsentiert, soll uns nicht weiter stören. ADODB ist der Klassenname in der Registry.

Wir sehen also auf der linken Seite ein Recordset-Objekt und auf der rechten Seite zum Beispiel die Open-Methode, mit der ein solches Recordset geöffnet wird. Und im Code können wir nun ebenfalls darauf zugreifen, wie in Abbildung 2.45 zu sehen ist.

Ganz so einfach ist die Sache letztendlich doch nicht. Soll unser Programm samt Verweis auf einem fremden Rechner laufen, so muss auf diesem Rechner eine kompatible Bibliothek des von uns gewählten Typs installiert sein. Bei Standardbibliotheken aus dem Office-Bereich, zu denen die ADODB letztlich gehört, können wir es voraussetzen. Schwieriger wird das Ganze bei exotischeren Bibliotheken. Es stehen uns zwar einige Mittel zur Verfügung, diese Bibliotheken auf Fremdsystemen verfügbar zu machen, aber Sie sollten im Hinterkopf behalten, dass die Bibliothek nicht notwendiger Weise vorhanden sein muss.

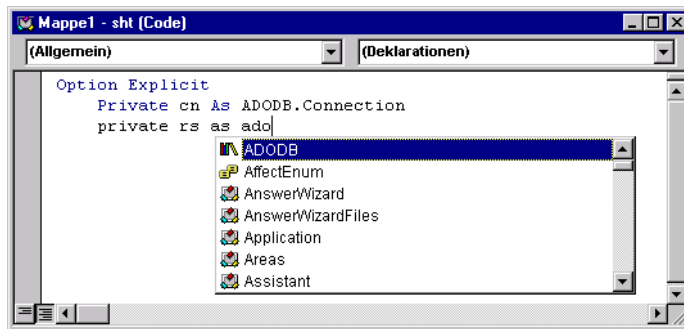


Abbildung 2.45: Die ADODB-Bibliothek im Code

Makros ...

Ich weiß nicht, wo sich in einem VBA-Projekt ein Makro befinden soll. VBA ist keine Makrosprache und Microsoft richtet mit diesem Begriff viel Unheil an. Ich denke mit einer gehörigen Portion Unwohlsein an die vielen Seminarteilnehmer zurück, die sich in VBA-Seminaren einfanden und etwas über Makroprogrammierung erfahren wollten. Als ich ihnen dann VBA in voller Schönheit präsentierte, glaubten sie sich im falschen Film.

Der Dialog, der sich hinter diesem Menüpunkt verbirgt, zeigt uns alle ausführbaren, also öffentlichen, Sub-Prozeduren des Projekts. Es gibt große Projekte, die vollständig auf privaten Ereignisroutinen basieren. Bei diesen wäre dieser Dialog leer.

Es handelt sich um ein Relikt aus VBA 2 Tagen (Excel 5), das heute schlicht und ergreifend überflüssig ist.

Optionen ...

Die vier Register des Optionsdialogs, die Ihnen nun präsentiert werden, enthalten die Einstellungen, die sich bewährt haben

Optionsdialog – Register Editor

Die **Automatische Syntaxüberprüfung** ist höchst überflüssig, denn eine Syntaxüberprüfung findet, wie wir bei der Diskussion des Befehls Kompilieren im Debuggen-Menü sahen, ohnehin beim Verlassen der Zeile statt. Und ein Mehr an Überprüfung findet auch nicht statt, wenn dieser Punkt aktiviert ist. Lediglich eine nervende MessageBox macht uns zusätzlich zu der ohnehin durchgeführten Rotfärbung darauf aufmerksam, dass da ein Fehler ist. Oft aber verlässt man eine unvollständige und somit fehlerhafte Zeile absichtlich, um aus einer anderen Zeile einen Ausdruck zu kopieren, was aber prompt durch diese MessageBox quittiert wird, die wir dann auch noch bestätigen müssen ...

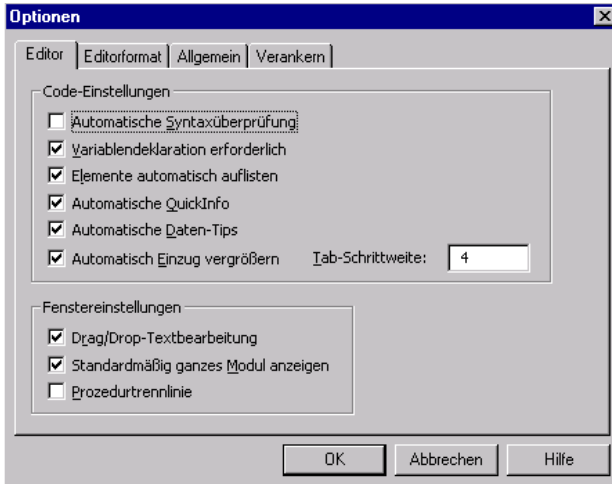


Abbildung 2.46: Optionsdialog – Register Editor

Variablen vor der Verwendung zu deklarieren, ist keine Frage der Etikette. **Variablendeklaration** ermöglicht die Erstellung stabiler und weniger speicherintensiver Applikationen und liegt somit in unserem ureigensten Interesse. Also bitte eine Häkchen reinmachen.

Elemente automatisch auflisten führt dazu, dass nach dem Schlüsselwort `As` in der Variablendeklaration und nach jedem `».«` in einer Objektkette ein kleines Fenster mit den nun möglichen Elementen erscheint. Also eine nützliche Sache, die im Gegensatz zu den beiden vorangehenden Optionen bereits richtig eingestellt ist.

Die **Automatische QuickInfo** zeigt uns die Parametrierung von Funktionen, wie folgendes Beispiel zeigt:

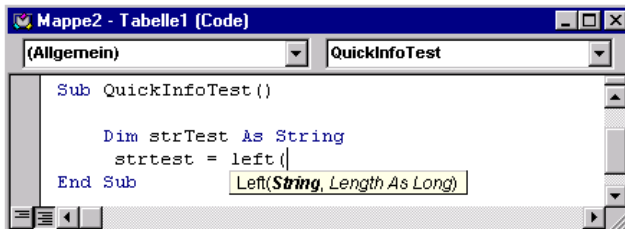


Abbildung 2.47: Codefenster mit QuickInfo

Bei aktivierten **Automatischen Daten-Tipps** können Sie im Haltemodus den Cursor einfach über eine Variable halten und nach kurzer Zeit erscheint ein kleines, gelbes Fenster mit dem aktuellen Wert der Variablen.

Den Cursor durch die linke Maustaste in die Variable zu positionieren ist hierbei nicht erforderlich, schadet aber auch nicht.

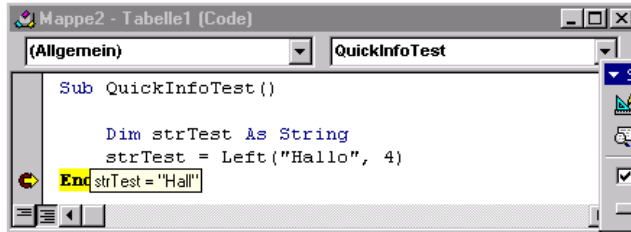


Abbildung 2.48: Codefenster mit automatischem Daten-Tipp

Die Option **Einzug automatisch vergrößern** bewirkt, dass ein Return am Ende einer (per Tabulator) eingezogenen Zeile den Cursor in der nun eingefügten Zeile an die Anfangsposition der vorangehenden Zeile setzt. Anders herum ausgedrückt heißt das, dass der Cursor an den Anfang der Zeile springt, wenn diese Option nicht eingestellt ist. Die rechts daneben stehende **Tab-Schrittweite** bestimmt das Intervall der Einzugspositionen.

Drag & Drop Textbearbeitung ermöglicht das Ziehen eines markierten Codefragments per Maus zu einer anderen Stelle. Bei gedrückter Steuerungstaste wird das Codefragment kopiert.

Ist die Option **Standardmäßig ganzes Modul anzeigen** aktiviert, so ist im aktuellen Codefenster so viel Code zu sehen, wie darzustellen ist, egal, aus wie vielen Prozeduren dieser stammt. Ist diese Option hingegen deaktiviert, erscheint jeweils nur die aktuelle Prozedur im Codefenster; die Navigation zu anderen Prozeduren muss mühsam über die beiden Combo-Boxes im Modulkopf erfolgen.

Die **Prozedurtrennlinie** ist eine graue, durchgezogene Linie, mit der Prozeduren optisch ein wenig voneinander abgehoben werden. Geschmacksache, ich habe sie ausgeschaltet.

Optionsdialog – Register Editor

Die Konfigurierbarkeit der farblichen Erscheinung des Codes beeinflussen zu können, mag den Individualisten vielleicht erfreuen. Dem Entwickler hingegen, der sich den Code anderer ansieht und dessen Code von anderen angesehen wird, bringt dies nichts. Im Gegenteil, denn in der Farbdarstellung steckt eine Information. In unserem Unternehmen ist dieses Register tabu, zum anfänglichen Entsetzen des einen oder anderen Studenten.

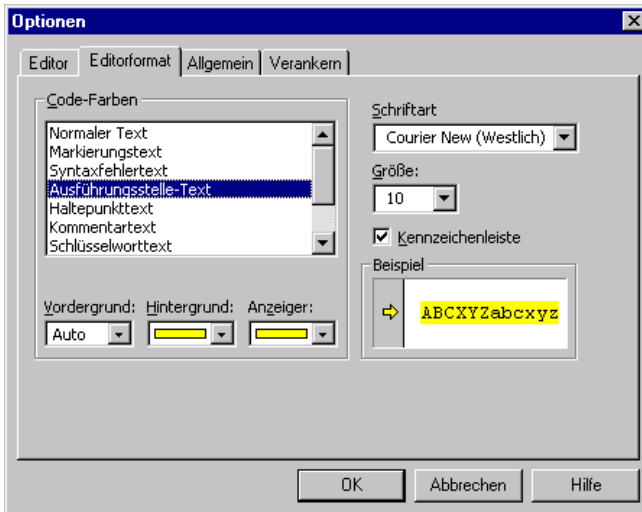


Abbildung 2.49: Optionsdialog – Register Editorformat

Optionsdialog – Register Allgemein

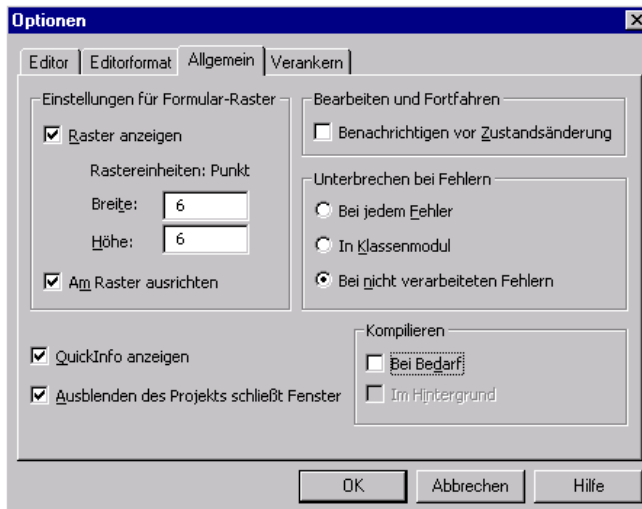


Abbildung 2.50: Optionsdialog – Register Allgemein

In Visual Basic hat sich das Formularmaß Twip etabliert, das ebenso wie Punkt oder Pixel ein zölliges Maß ist. Ein Punkt ist ein Zweiundsiebzigstel Zoll und ein Twip wiederum ein Zwanzigstel Punkt. Es steht zu erwarten, dass auch in VBA irgendwann das Maß Twip Einzug halten wird, späte-

Twip und Punkt

stens dann, wenn die derzeit in VBA vorherrschende Forms-2-Bibliothek ihren Abschied nehmen wird.

Die **Einstellungen für Formularraster** sollten Sie so belassen, wie sie sind. Im Kapitel über Dialoge werden wir noch darauf zurückkommen.

Benachrichtigung vor Zustandsänderung bedeutet laut Hilfe, dass eine Benachrichtigung erfolgt, wenn eine Aktion alle Variablen auf Modulebene zurücksetzt. Mir ist keine Aktion gelungen, die nur diese Benachrichtigung produzierte, ohne zugleich das ganze Projekt zurückzusetzen. Sollten Sie in Erfahrung bringen, wozu diese Option dient, so schreiben Sie mir bitte eine Mail.

Unterbrechen bei Fehlern hat folgende Einstellungen:

- ▼ Bei jedem Fehler
- ▼ In Klassenmodul (Default-Einstellung)
- ▼ Bei nicht verarbeiteten Fehlern

In Kapitel 9 werden wir das Thema Fehlerbehandlung gründlich untersuchen, was uns aber nicht hindern sollte, kurz auf diese Optionen einzugehen.

Bei jedem Fehler die Ausführung des Code unterbrechen zu lassen, ist dann sinnvoll, wenn Sie der fehlerhaften Zeile innerhalb einer Prozedur auf die Spur kommen wollen.

In Klassenmodulen zu unterbrechen ist eine Option, die ich nicht verstanden habe, denn qualitativ besteht kein Unterschied zwischen einem Fehler innerhalb oder außerhalb eines Klassenmoduls. Beide müssen wir so weit bringen, dass sie im Regelbetrieb fehlerfrei arbeiten und beide nur bei unerwarteten Problemen über ErrorHandler in einer lückenlosen Aufwärtskette zu einem gesteuerten Abschluss kommen.

Bei nicht verarbeiteten Fehlern zu unterbrechen, sollte Ihre Standardeinstellung werden, denn zum einen können Sie sich auf Ihre ErrorHandler verlassen und zum anderen kommen Sie durch diese Einstellung auf Dauer jedem vergessenen ErrorHandler auf die Schliche. Spätestens, wenn Sie Ihre Konstruktion anderen überlassen, sollte diese Option aktiviert sein.

QuickInfo anzeigen betrifft nur die QuickInfo der Entwicklungsumgebung. Wahrscheinlich ist diese Checkbox der optischen Ausgewogenheit wegen ergänzt worden: da war in der Ecke noch etwas Platz.

Ausblenden des Projekts schließt Fenster betrifft nun die so genannten VBA-Projekte, nicht aber Projekte im Sinne einer Arbeitsmappe. Die Einstellung schließt und öffnet die beteiligten Fenster, wenn das Projekt über

das Plus-Minus-Zeichen vor dem Projekteintrag im Projekt-Explorer ein- oder ausgeblendet wird. Es macht wohl Sinn, diese Option aktiviert zu haben, wenn Sie ein VBA-Projekt bearbeiten. Allerdings wäre es auch schön, wenn diese Reaktion bei allen gruppierenden Elementen des Projekt-Explorers funktionieren würde.

Ist Kompilieren **Bei Bedarf** aktiviert, so wird eine Prozedur nur dann überprüft (siehe Codeüberprüfung vor Abarbeiten der Prozedur unter 2.8.6), wenn die Prozedur auch durchlaufen werden muss. Wenn diese Option nicht aktiviert ist, werden alle Prozeduren aller Module kompiliert, wenn das Programm gestartet wird. Es ist also durchaus sinnvoll, diese standardmäßig aktivierte Option auszuschalten.

Kompilieren **Im Hintergrund** kann nur aktiviert werden, wenn auch Kompilieren *Bei Bedarf* eingeschaltet ist. Diese Einstellung überzeugt mich nicht so richtig, *da* die Kompilierung *Bei Bedarf* hierzu eingeschaltet werden muss.

Optionsdialog – Register Verankern

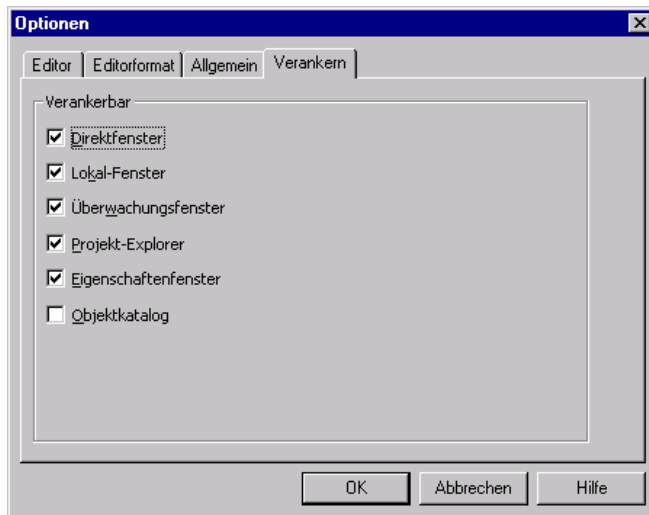


Abbildung 2.51: Optionsdialog – Register Verankern

Das Verankern eines der hier aufgeführten Fenster hat zur Folge, dass sich das betreffende Fenster neben und gleichzeitig mit dem Code auf dem Bildschirm präsentiert und mit seinen Kanten an den benachbarten Fenstern anschließt. Das heißt noch nicht, dass es auch angezeigt wird, denn man kann das Direktfenster über das Ansicht-Menü zu- und abschalten. Es bedeutet nur, dass es sich so verhält, wenn es sichtbar ist.

Es sind in der Regel also alle Fenster verankert, die gleichzeitig mit einem Codemodul oder einer UserForm benötigt werden können. Befindet sich die Codeausführung im Unterbrechungsmodus, so macht es Sinn, das Direkt- Lokal- oder Überwachungsfenster neben dem Codemodul einsehbar zu haben. Anders hingegen der Objektkatalog, der in der Regel nur für eine kurze Auskunft zurate gezogen wird und somit nicht ständig sichtbar sein muss.

Projekteigenschaften – Register Allgemein

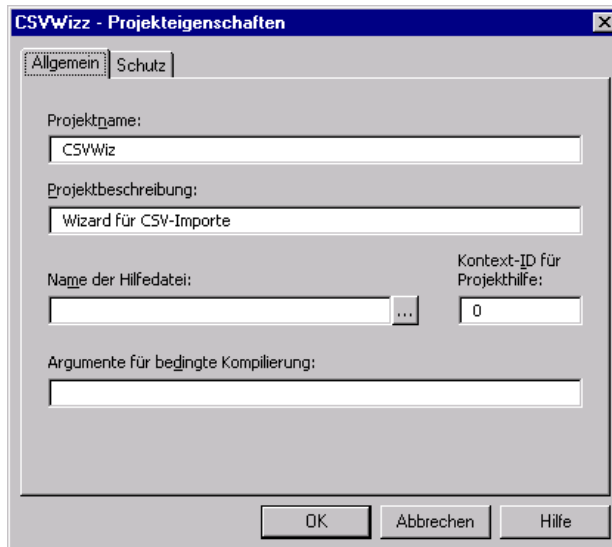


Abbildung 2.52: Dialog Projekteigenschaften – Register Allgemein

Unter **Projektname** können Sie den Namen angeben, den das Projekt zukünftig tragen soll. Dadurch wird der standardmäßige Ausdruck VBAProjekt ein wenig aussagekräftiger. Dieser neue Name wird auch im Projekt-Explorer angezeigt und muss nicht notwendigerweise mit dem Dateinamen übereinstimmen.

Die hier angegebene **Projektbeschreibung** wird im Objektkatalog angezeigt.

Sofern Ihr Projekt über eine eigene Hilfe verfügen soll, können Sie hier den **Namen der Hilfedatei** angeben, die von dem Programm verwendet werden soll. Unterstützt werden nicht nur die alten hlp-Dateien, sondern auch die moderneren, kompilierten HTML-Hilfedateien, die chm als Dateierweiterung tragen. Hierzu benötigen Sie allerdings den HTML Help Workshop, der sich auf der Developer-CD befindet. Auf der folgenden

Seite finden Sie aktuelle Informationen zu diesem Programm sowie einen Download-Link:

»<http://msdn.microsoft.com/workshop/author/htmlhelp/workshop.asp>«

Die **Kontext-ID für Projekthilfe** ist die ID, unter der die Informationen stehen sollten, die im Objektkatalog angezeigt werden sollen, sofern dort Hilfe zu der Bibliothek angeboten werden soll.

Bei **Argumente für bedingte Kompilierung** können Sie Ausdrücke angeben, die in Verzweigungen für bedingte Kompilierungen geprüft werden können. Was dahinter steckt, wird vielleicht am besten anhand eines Beispiels klar.

Ihr Programm verfügt über einen Login-Dialog, in dem Name und Passwort des Anwenders geprüft werden sollen. Den Dialog haben Sie fertiggestellt und getestet. Während der Entwicklung wollen Sie jedoch nicht bei jedem Programmstart den Login-Dialog ausfüllen, nur um das Programm zu testen. Sie können nun als Argument für bedingte Kompilierung `bDebug = -1` eingeben. Da hier nur numerische Argumente akzeptiert werden, funktioniert `fDebug = True` leider nicht. Aber `-1` entspricht dem logischen `True`, was letztendlich nichts anderes als ein Integer ist.

Im Code nun können Sie diesen Wert abfragen und ihn zur Entscheidung, ob der Login-Dialog angezeigt werden soll, heranziehen:

```
Private Sub Workbook_Open()
    ...
    #If Not fDebug Then
        frmLogin.Show
    #End If
    ...
End Sub
```

Der Haken daran ist, dass Sie dieses Argument im Eigenschaftsdialog verändern oder entfernen müssen, bevor Sie Ihre Anwendung auf die Menschheit loslassen. Andernfalls würde der Login-Dialog auch bei Ihren Anwendern nicht erscheinen.

**der Haken der
bedingten Kompilierung**

Was also ebenso gut funktionieren würde, wäre das Auskommentieren des Dialogaufrufs, denn auch daran müssen Sie sich erinnern, bevor Sie die Datei verteilen:

```
Private Sub Workbook_Open()
    ...
    'frmLogin.Show
```

```
...
End Sub
```

Ich hätte da einen anderen Vorschlag. Legen Sie unter C:\ eine Datei mit irgendeinem verrückten Namen an, zum Beispiel Q2F57LXW.357. Im Code prüfen Sie mit der Dir-Funktion, ob diese Datei da ist, und entscheiden damit, ob der Dialog angezeigt werden soll:

```
Private Sub Workbook_Open()

    If Dir("C:\Q2F57LXW.357") = "" Then
        frmLogin.Show
    End If

End Sub
```

Damit haben Sie eine Lösung, die immer funktioniert und Ihr Gedächtnis nicht belastet. Ich kann mich lebhaft an Situationen erinnern, bei denen mal wieder auf die Schnelle eine kleine Änderung per Mail rausgehen musste, ohne dass ich daran dachte, irgendein Debug-Merkmal auszubauen.

Projekteigenschaften – Register Schutz

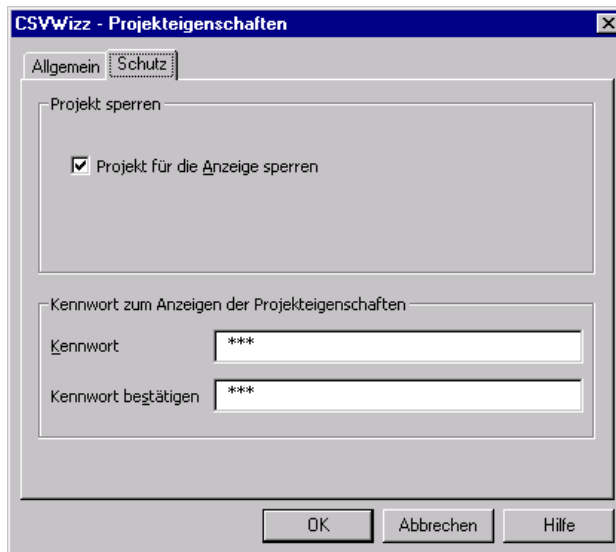


Abbildung 2.53: Projekteigenschaften – Register Schutz

Das VBAProjekt, also letztendlich den Code, vor unerwünschtem Zugriff zu sichern, kann durchaus sinnvoll sein. Zum einen natürlich, weil Sie das Programm schlichtweg für genial halten und verhindern wollen, dass andere damit groß rauskommen. Andererseits ist ein geschütztes Projekt aber auch manipulationssicher.

Das Argument, dass ein Passwortschutz verhindert, dass der folgende Dialog erscheint, zieht meines Erachtens nicht, denn Programme werden bei korrekt implementierter Fehlerbehandlung diesen Dialog nie anzeigen:



Abbildung 2.54: Dialog Laufzeitfehler

Ist Ihr Projekt mit einem Passwort geschützt, so führt ein Klick auf das Plus-Minus-Zeichen links neben dem Projektnamen zu diesem Dialog:

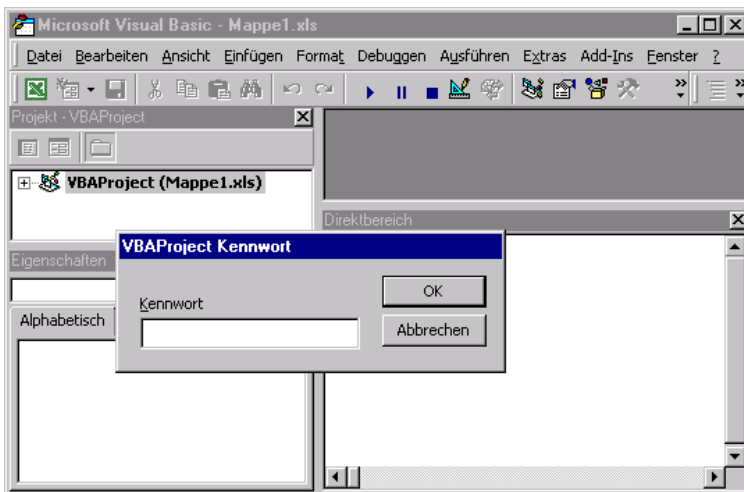


Abbildung 2.55: Entwicklungsumgebung mit Passwortschutz

Digitale Signatur ...

In Zeiten, in denen ein so genannter Makrovirus ein Unternehmen lahm legen kann, sind Mechanismen gefragt, die Unternehmen etwas mehr schützen, als es Dialoge wie der folgende tun:

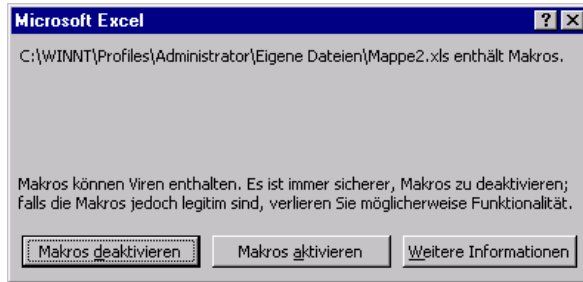


Abbildung 2.56: Digitale Signatur – Sicherheitshinweis beim Öffnen einer Datei

Das Kraut, das dagegen gewachsen ist, heißt *Digitale Signatur*. Eine Signatur ist ein digitaler Schlüssel, der in Code enthaltene Dateien integriert werden kann. Jede Anwendung, die Authenticode interpretieren kann, hat nun die Möglichkeit die in der Datei enthaltene Signatur mit den im System als vertrauenswürdige eingestuft Quellen zu vergleichen und entsprechend zu handeln.

Es gibt drei Möglichkeiten, eine solche Signatur zu erzeugen:

- ▼ weltweit authentifizierte Signatur durch ein anerkanntes Zertifizierungsunternehmen
- ▼ firmenübergreifende Signatur per Microsoft Certificate Server
- ▼ private Signatur per SelfCert.exe

weltweit
authentifizierte
Signatur

Informationen über weltweit authentifizierte Signaturen und die sie ausstellende Unternehmen erhalten Sie unter folgender Web-Adresse:

»<http://officeupdate.microsoft.com/office/redirect/fromOffice9/cert.htm>«

Diese Website enthält jedoch im Schwerpunkt amerikanische Unternehmen. Aber auch unser Bundesamt für Sicherheit in der Informationstechnik hat sich des Themas angenommen und bietet Informationen unter:

»<http://www.bsi.bund.de/aufgaben/projekte/pbdigsig/start.htm>«

firmenübergreifende
Signatur

Firmenübergreifende Signaturen können Sie vergeben und verwalten, wenn Sie zum Beispiel den Microsoft Certificate Server im Einsatz haben.

Lediglich zu Testzwecken empfiehlt Microsoft den Einsatz privater Signaturen. Office 2000 enthält eine Installationsoption namens »Digitale Signatur für VBA-Projekte« unter »Office-Tools«. Danach steht in Ihrem Office-Verzeichnis die SelfCert.exe, mit der Sie eine private Signatur erzeugen können. Sie geben einfach Ihren Namen an, das Programm erzeugt diese Signatur und beendet sich wieder mit einer kleinen Erfolgsmeldung. Danach steht Ihnen diese Signatur in der Entwicklungsumgebung zur Verfügung. Wählen Sie hierzu den Punkt Digitale Signatur ... im Menü Extras:



Abbildung 2.57: Dialog Digitale Signatur

Die Schaltfläche Wählen bringt Sie zu dem folgenden Dialog:

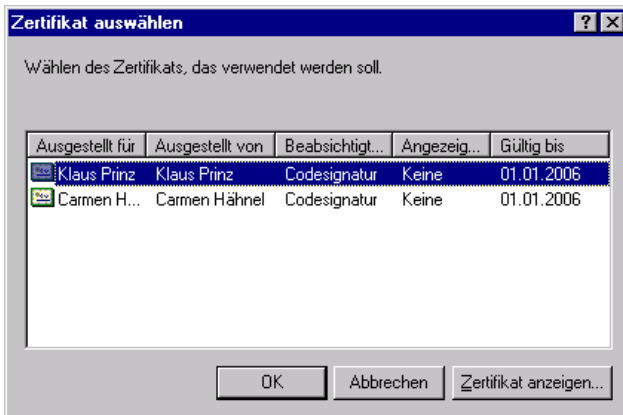


Abbildung 2.58: Digitale Signatur – Dialog Zertifikat auswählen

Die Schaltfläche **Zertifikat anzeigen ...** produziert einen Dialog mit Informationen zu dem Zertifikat.

Nachdem Sie Ihrem VBA-Projekt ein Zertifikat hinzugefügt haben, sollten Sie die aktuelle Datei schließen. Im Menü **EXTRAS | MAKRO | SICHERHEIT ...** sollten Sie mittlere oder hohe Sicherheit einstellen.

Öffnen Sie danach die soeben mit Signatur versehene Datei, so werden Sie mit folgendem Dialog begrüßt:

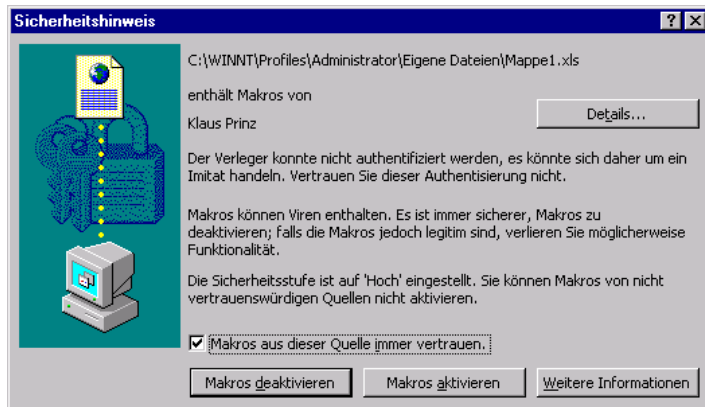


Abbildung 2.59: Excel-Dialog Sicherheitshinweis

Wenn Sie die Checkbox »Makros aus dieser Quelle immer vertrauen« aktivieren, wird die Schaltfläche »Makros aktivieren« bedienbar und Sie können den Dialog damit schließen.

Nun sind alle mit dieser Signatur versehenen Arbeitsmappen als sicher eingestuft und werden mit aktiviertem Code geöffnet, sofern die Sicherheit auf mittel oder hoch eingestellt ist.

Warum erzähle ich Ihnen das alles?

Als VBA-Entwickler wissen Sie, dass es digitale Signaturen gibt, und Sie wissen, wie man sie einsetzt. Vielleicht sollten Sie das Thema Sicherheit in Ihrer Firma einmal mit den Verantwortlichen diskutieren und ggf. die Informationen aus MSDN und dem Web zur Verfügung stellen, denn Sie sind ja der Fachmann dafür.

Add-Ins

Sofern Sie noch keine Add-Ins aktiviert haben, so steht nur der Befehl **Add-In-Manager ...** zur Verfügung, der den Dialog in Abb. 2.60 startet.

Im Add-In-Manager können Sie nun für jedes der angezeigten Add-Ins über die beiden Checkboxes festlegen, ob sie jetzt und eventuell sogar bei jedem zukünftigen Start geladen werden sollen.

Über Sinn und Zweck der einzelnen Add-Ins zu diskutieren erspare ich uns. Die Palette reicht von überaus nützlich bis ... na ja, lassen wir das. Das einzige dieser Tools, welches den Status unverzichtbar verdient, ist der API-Viewer. Aber richtig nutzbar wird er erst, wenn Dan Appleman's Visual Basic Programmer's Guide to the Win32 API in Griffweite liegt.

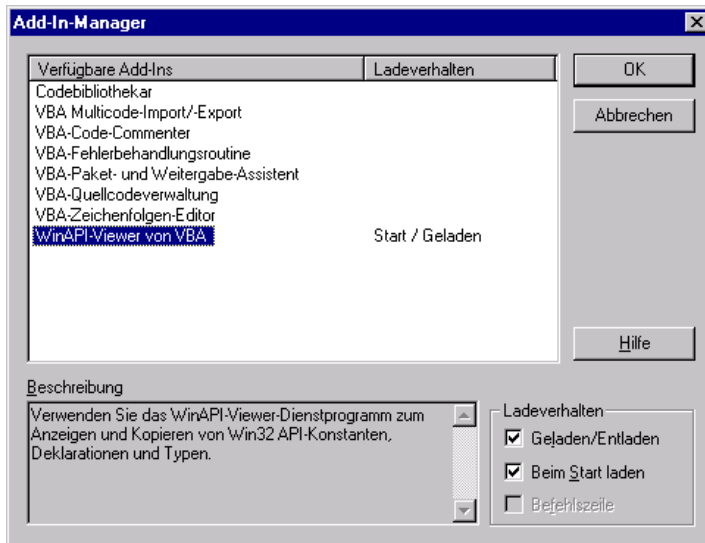


Abbildung 2.60: Dialog Add-In-Manager

2.9 Symbolleisten

Zum Abschluss dieses Kapitels möchte ich Ihnen noch die vier Symbolleisten der Entwicklungsumgebung zeigen, von denen ich nur die Symbolleisten Bearbeiten und Voreinstellung auf dem Bildschirm habe. Das einzige Symbol, das dann noch fehlt, ist die Aufrufeliste.

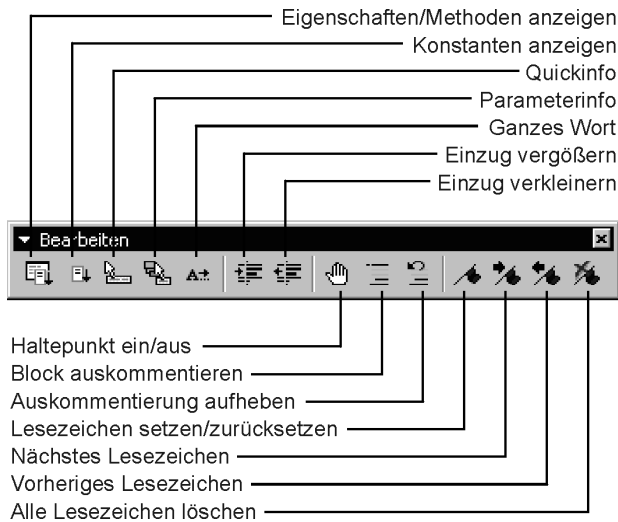


Abbildung 2.61: Symbolleiste Bearbeiten

Die Symbolleiste Bearbeiten enthält zwei Symbole, die wichtig sind und deren Funktionalität in keinem Menü enthalten sind (?).

Mit **Block auskommentieren** können Sie an den Zeilenanfang aller markierten Zeilen ein Kommentarzeichen ergänzen. **Auskommentierung aufheben** entfernt für alle markierten Zeilen je ein Kommentarzeichen. Dieses zu entfernende muss wiederum nicht am Zeilenanfang stehen.

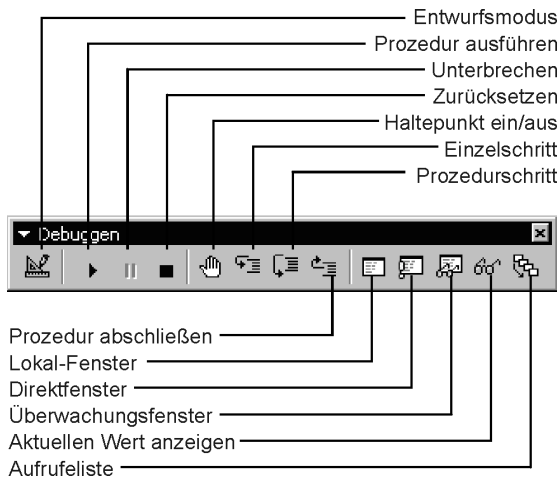


Abbildung 2.62: Symbolleiste Debuggen

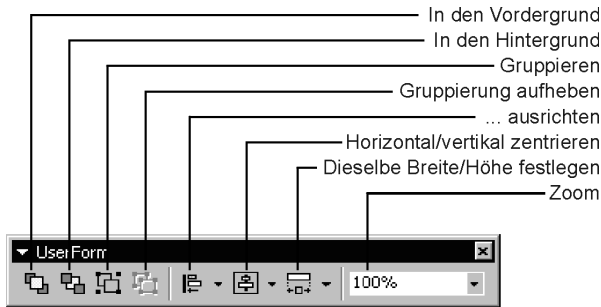


Abbildung 2.63: Symbolleiste UserForm

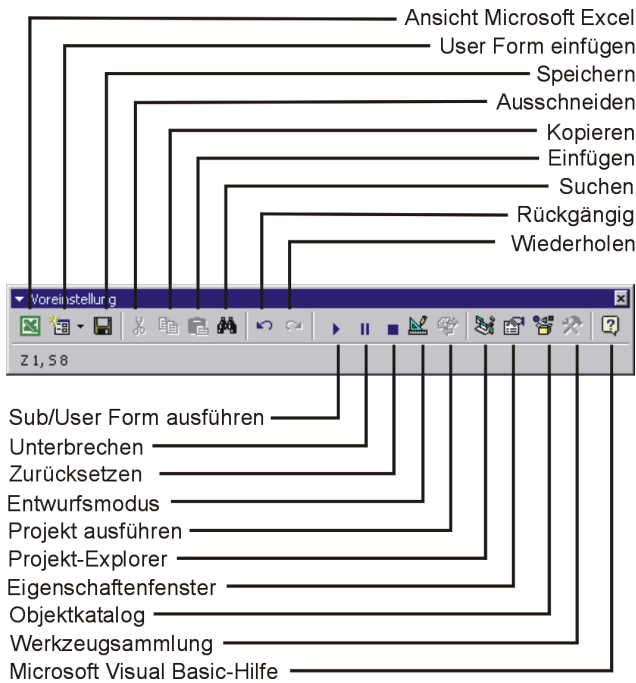


Abbildung 2.64: Symbolleiste Voreinstellung



Einstieg

3

Kapitelüberblick

3.1	Lassen wir es piepen	84
3.2	Etwas Zellzugriff	92
3.2.1	Excel-Objekte – eine erste Annäherung	93
3.2.2	Gezielter Zugriff auf eine Zelle	95
3.2.3	Dynamischer Zugriff auf Zellen	103
3.2.4	Ermittlung der letzten belegten Zeile	105
3.2.5	Rückblick	108

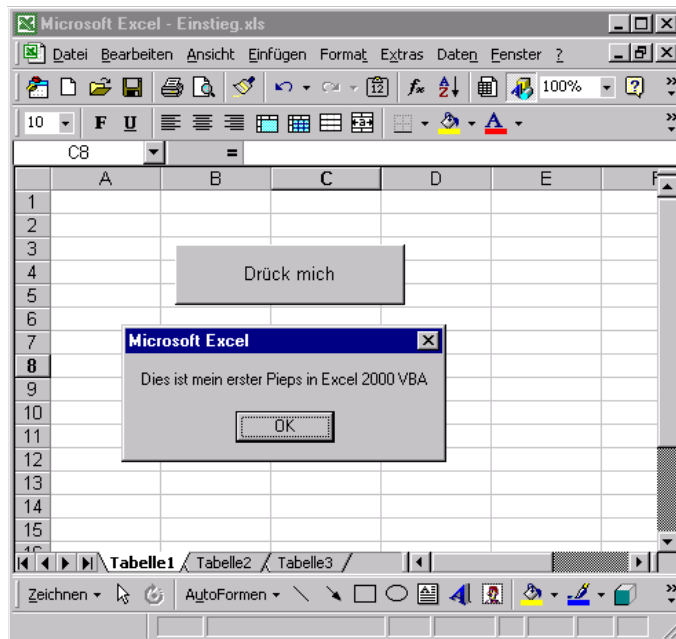


Abbildung 3.1: Der erste Pieps

Neues ist ihm scheinbar auch eingefallen. Aber immerhin hat er anstelle des obligatorischen »Hello World« etwas mit »Pieps« hervorgebracht. Na ja. Recht haben Sie. Und irgendwo muss man eben anfangen. Warum also nicht mit einem Pieps.

Sprachelemente werden in den Kapiteln 4 und 7 besprochen, in Kapitel 5 schauen wir uns Objekte sowie deren ganzes Drumherum an und in Kapitel 6 geht es dann um die Excel-Objekte. Aber ohne ein paar Sprachelemente und einige Häppchen Excel-Objekte werden wir hier nicht auskommen. Sie werden also ein wenig Excel-VBA nachplappern und in den folgenden 4 Kapiteln Vokabular und Syntax gründlich erlernen.

3.1 Lassen wir es piepen

Doch nun zurück zu unserem Pieps aus Abbildung 3.1.

Die Entwicklungsumgebung erreichen Sie über das Menü EXTRAS | MAKRO | VISUAL BASIC EDITOR, die Tastenkombination **Alt + F11** oder die Schaltfläche, die Sie in Abb. 3.1 ganz links in der obersten Symbolleiste sehen können. Diese Schaltfläche ist standardmäßig nicht in der Symbolleiste enthalten, Sie können sie aber über den Dialog Symbolleisten Anpassen an einer beliebigen Position einer Symbolleiste einfügen.

Die Entwicklungsumgebung dürfte sich inzwischen in einem separaten Fenster präsentiert haben. Wir stehen nun vor der Entscheidung, wohin das Progrämmchen zum Erzeugen des Pieps denn nun geschrieben werden soll. Die Regel ist im Prinzip ganz einfach:

Schreiben Sie den Code in das Modulblatt, in dem Sie später vermutlich auch danach suchen werden.

Die Schaltfläche, die den Pieps hervorgebracht hat, befindet sich in der Tabelle mit dem Namen Tabelle1. Also schreiben wir unsere Programmzeilen auch genau dort hin. Machen Sie einen Doppelklick auf dem Eintrag Tabelle1(Tabelle1) im Projektexplorer, woraufhin sich das dazugehörige Modulblatt öffnet. Wenn Sie die empfohlenen Optionen des Kapitels 2 eingestellt haben, so müsste das Modulblatt etwa so aussehen:

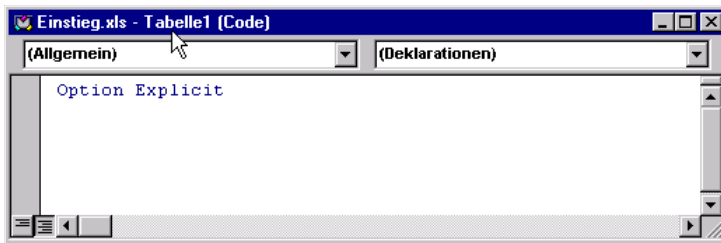


Abbildung 3.2: Modulblatt ohne Code

Damit die Prozedur, die wir nun schreiben wollen, außerhalb unseres Modulblatts sichtbar ist, muss sie als öffentliche Prozedur angelegt werden. Das Schlüsselwort für öffentlich lautet `Public` und die Bezeichnung einer – nennen wir sie eine gewöhnliche – Prozedur lautet `Sub`. Fehlt noch ein Name für unsere Prozedur.

Zwar dürfen wir in VBA Namen aus Umlauten bilden, aber dem einen oder anderen Zeitgenossen jagt ein Prozedurname wie `SchließeDatei`, `ÖffneDatenbank` oder `MachePieps` immer noch eine Gänsehaut über den Rücken. Die Sprache der Softwareentwicklung ist und bleibt vermutlich auch Englisch und außerdem sind englische Begriffe meist deutlich kürzer als die deutschen Pendanten. Möglich wären also `MyFirstProgramm`, `MakeCheep` oder schlicht `Cheep`. Also schreiben wir an die Stelle, an der unser Cursor vor sich hinblinkt:

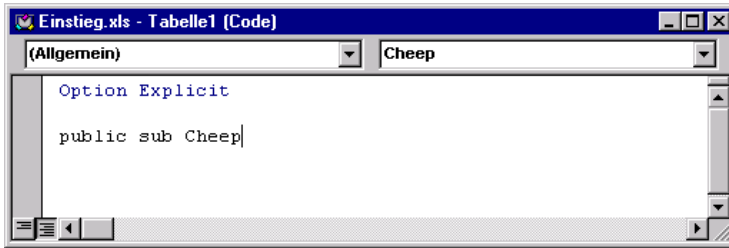


Abbildung 3.3: Codefenster mit Sub-Zeile

Die Schlüsselwörter `Public` und `Sub` sind kleingeschrieben, das Wort `Cheep` hingegen beginnt mit einem großen Anfangsbuchstaben. Das hat auch seinen Grund, denn der Editor ändert die Schreibweise erkannter Schlüsselwörter so ab, wie diese in der Sprachbibliothek hinterlegt sind. Wenn wir gleich die Zeile mit einem beherzten `Return` abschließen, so werden die beiden Begriffe abgeändert. Mit der Zeit werden Sie diesen Vorgang unbewusst in den Augenwinkeln wahrnehmen und vor allem bemerken, dass die Veränderung gelegentlich ausbleibt. Und dann können Sie sicher sein, dass Sie sich verschrieben haben. Wie wir in Kapitel 2 im Rahmen der Kompilierung feststellten, ist die Umwandlung allein noch keine Garantie für eine korrekte Zeile, aber das Ausbleiben ein sicherer Hinweis auf einen Fehler.

`Cheep` hingegen ist ein von uns geprägter Begriff, worunter nicht nur Prozedurnamen, sondern auch die Namen von Variablen, Konstanten, Modulen oder auch Steuerelementen fallen. Der Editor übernimmt aber im gesamten weiteren Projekt die Schreibweise, mit der dieses Element quasi veröffentlicht worden ist. Und die Stelle, an der Prozedurnamen verbindlich vereinbart werden, ist nun mal die Zeile, in der sie hinter den Schlüsselwörtern `Sub`, `Function` oder `Property` geschrieben werden.

Nun wird's aber Zeit für das die Zeile abschließende `Return`.

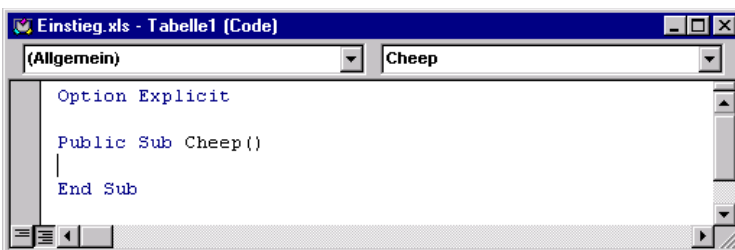


Abbildung 3.4: Codefenster mit Prozedurrahmen

Wie Abb. 3.4 zeigt, wurden die Schlüsselwörter korrekt umgewandelt, blau eingefärbt. Hinter unserem Prozedurnamen wurde ein Klammernpaar gesetzt und zwei Zeilen tiefer die mit der Sub-Zeile korrespondierende End Sub-Zeile ergänzt.

Es hat sich bei den meisten Entwicklern eingebürgert, alle Zeilen zwischen der Sub- und der End Sub-Zeile um eine Tabulatorweite einzurücken. Das macht den Code einfacher lesbar.

Das Nachrichtenfenster, eine so genannte MessageBox, die in Abb. 3.1 zu sehen ist, kann mit einer Anweisung erzeugt werden. Die Funktion dazu lautet `MsgBox`. Sobald Sie dahinter ein Leerzeichen schreiben, erscheint ein Hinweis mit der so genannten QuickInfo (siehe Abb. 3.5), die generell dann Hilfestellung bei der Parametrierung bieten, wenn ein Sprachelement solche Parameter (Argumente) erwartet oder unterstützt:

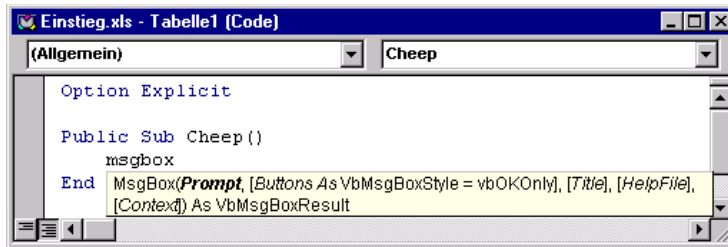


Abbildung 3.5: Codefenster mit QuickInfo zu `MsgBox`

Diese QuickInfo mag auf den ersten Blick voller Hieroglyphen sein, weshalb wir die einzelnen Komponenten nun sauberlich zerlegen werden. Das Erste, was auffällt, ist, dass unmittelbar hinter dem letzten Zeichen der `MsgBox`-Funktion eine Klammer geöffnet wird, die erst in der zweiten Zeile wieder geschlossen wird. Der Grund dafür ist, dass eine MessageBox eine Rückgabe liefern kann. Wenn wir daran interessiert sind, so schreiben wir die Funktion nebst Argumenten auf die rechte Seite eines Gleichheitszeichens und die Funktion gibt ein Ergebnis an die linke Seite des Gleichheitszeichens zurück. Da aber die Rückgabe die Regel beim Einsatz der Funktion ist, werden die Argumente der Funktion eben mit diesen Gleichheitszeichen dargestellt. Außerdem kann so die hinter der abschließenden Klammer stehende Rückgabe besser von den Argumenten abgegrenzt werden.

Somit hätten wir bereits die zweite Komponente der Quickinfo entschlüsselt, nämlich die Rückgabe, die dort als `As VbMsgBoxResult` gekennzeichnet ist. `VbMsgBoxResult` ist übrigens der Name einer Liste von Konstanten, die verschiedene benannte, numerische Werte enthält. Diese Details sollten uns hier jedoch nicht weiter belasten.

Bleibt uns noch der Ausdruck zwischen den Klammern, der durch Komma getrennt die einzelnen Argumente enthält. Argumente ohne Klammern müssen angegeben werden, solche mit Klammern sind optional. In unserem Beispiel muss das Argument Prompt angegeben werden, die anderen dürfen wir weglassen. Als Prompt erwartet die MsgBox den Text, den sie darstellen soll.

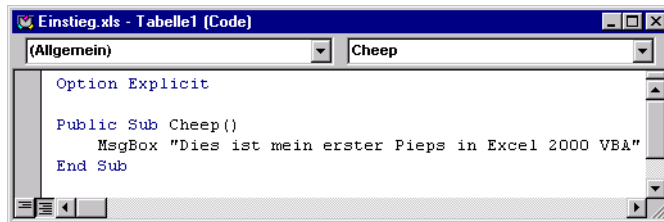


Abbildung 3.6: Codefenster mit MsgBox-Anweisung

Wenn Sie die Zeile mit einem Return verlassen haben, so wurde eine Zeile zwischen der MessageBox und der End Sub-Zeile eingefügt. Diese neue Zeile können Sie verhindern, wenn Sie sich mit einer Cursor-Taste aus der Zeile herausbewegen.

Solange wir uns in der Entwicklungsumgebung befinden, können wir eine Prozedur über das Menü AUSFÜHREN | SUB/USERFORM AUSFÜHREN starten, per ShortCut [F5] oder über das entsprechende Symbol der Symbolleiste Voreinstellung oder Debuggen. Danach wird zu Excel gewechselt und die MessageBox wird, wie in Abb. 3.1 zu sehen, auf dem Bildschirm erscheinen.

Nun werden wir noch die Schaltfläche in die Tabelle platzieren und wir sind so gut wie fertig.

Unter den Excel-Symbolleisten befinden sich gleich zwei, die mit Steuerelementen bestückt sind, wie die beiden folgenden Abbildungen zeigen:



Abbildung 3.7: Symbolleiste Formular



Abbildung 3.8: Symbolleiste Steuerelement-Toolbox

Der Cursor befindet sich in beiden ScreenShots über dem Steuerelement Schaltfläche (CommandButton), das auch in unserem Beispiel Verwen-

derung findet. Wozu zwei verschiedene Symbolleisten mit im Prinzip identischen Steuerelementen?

Die **Symbolleiste Formular** enthält die Elemente, die in Excel 5 und '95 verwendet werden mussten. Diese Steuerelemente sind von recht einfachem Zuschnitt und unterstützen nicht den ActiveX-Standard (ehemals OLE). Sie müssen manuell mit einer `Public Sub` verbunden werden, sind also nicht in der Lage, richtige Ereignisse zu unterstützen.

Symbolleiste FORMULAR

Bei den Steuerelementen der **Symbolleiste Steuerelement-Toolbox** handelt es sich hingegen um ActiveX-Controls, die mit allen entsprechenden Containern kooperieren können. Sie sind in der Microsoft Forms 2.0 Object Library (FM20.DLL) zusammengefasst. Die Verbindung zum VBA-Code wird über jeweils eine Vielzahl von Ereignisroutinen hergestellt.

**Symbolleiste STEUER-
ELEMENT-TOOLBOX**

Da der Umgang mit den Elementen der Steuerelement-Toolbox etwas komplizierter ist, verzichten wir an dieser Stelle darauf und nehmen stattdessen eine Schaltfläche aus der Formular-Symbolleiste. Die Steuerelement-Toolbox wird in Kapitel 8 – Dialoge näher untersucht.

Steuerelemente werden wie alle anderen graphischen Objekte in eine Tabelle platziert, indem Sie auf die entsprechende Schaltfläche klicken und die linke Maustaste gleich wieder loslassen. Der Cursor zeigt nun das Fadenkreuzsymbol. Positionieren Sie dieses Fadenkreuz an die Stelle, an der die obere linke Ecke der Schaltfläche liegen soll. Ziehen Sie nun die Maus bei gedrückter linker Maustaste bis zu der Stelle, an der die untere, rechte Ecke der Schaltfläche liegen soll. Sobald Sie die linke Maustaste loslassen, erscheint der folgende Dialog:

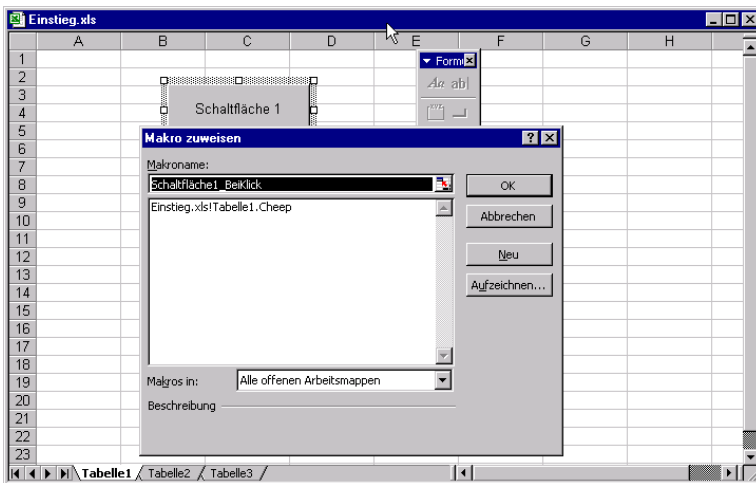


Abbildung 3.9: Dialog Makro zuweisen

Die Relikte aus Excel 5 und '95 sind hier noch gut daran zu erkennen, dass als Vorschlag für den Prozedurnamen `Schaltfläche1_BeiClick` erscheint, denn dieses alte VBA (VBA 2) ermöglichte die Codierung in deutscher Sprache. Damals sind die meisten Autoren darauf hereingefallen und veröffentlichten Bücher über die Entwicklung von VBA-Programmen in Deutsch.

In dem Listenfeld (ListBox) darunter sehen Sie alle `Public Subs` im gesamten Projekt, an die diese Schaltfläche gebunden werden kann. Da wir nur eine geschrieben haben, ist darin auch nur unsere `Cheep` enthalten, die mit ihrem vollständigen Kontext dargestellt ist. Sie befindet sich in dem gebundenen Modulblatt `Tabelle1`, das wiederum in der Arbeitsmappe `Einstieg.xls` enthalten ist:

`Einstieg.xls!Tabelle1.Cheep`

Der Ausdruck `Tabelle1` nimmt übrigens nicht Bezug auf den Tabellenblatt-namen, sondern den Objektnamen der Tabelle, der in der Entwicklungsumgebung eingestellt werden kann. Diese beiden Namen sind zu Anfang identisch.

Klicken Sie bitte auf den Eintrag in der ListBox und danach auf die OK-Schaltfläche. Noch hat Ihre Schaltfläche den gemusterten Rand mit den acht Anfassern, eine Art Bearbeitungsmodus. Ein Klick in eine beliebige Zelle beendet diesen Modus und die Schaltfläche ist bereit. Bewegen Sie Ihren Cursor nun über die Schaltfläche, nimmt der Cursor die Form einer Hand mit gestrecktem Zeigefinger an, was er nur bei der Schaltfläche aus der Symbolleiste Formular kann.

Klicken Sie nun mit der linken Maustaste auf die Schaltfläche, so erscheint unsere `MessageBox`.

Bearbeitungsmodus

Sie kennen es sicherlich von anderen Shapes (der Oberbegriff für alle on-top von Tabellen platzierbaren Objekten), dass über das Kontextmenü eines solchen Objekts oder den obersten Eintrag des Format-Menüs ein Dialog erreicht werden kann, der Formatierung des betreffenden Objekts ermöglicht. Üblicherweise reicht das Anklicken aus, und die Befehle stehen zur Verfügung. Klicken wir unsere bereits zugewiesene Schaltfläche hingegen mit der linken Maustaste an, so läuft ja unser Code los. Also müssen wir die rechte Maustaste benutzen, die dann auch prompt das Kontextmenü hervorzaubert (siehe Abb. 3.11).

Der Menüeintrag `Steuerelement formatieren ...` bietet zwar eine Reihe von Formatierungsmöglichkeiten, jedoch keine Möglichkeit, die Aufschrift der Schaltfläche zu ändern. Der Klick mit der rechten Maustaste hat zwar das Kontextmenü hervorgehoben, aber auch gleichzeitig die Schaltfläche in

den Bearbeitungsmodus versetzt. Deshalb können Sie nun mit der linken Maustaste die von Excel vergebene Aufschrift »Schaltfläche 1« wie gewohnt markieren und überschreiben (siehe Abb. 3.12).

Die Symbolleiste Zeichnen enthält eine Schaltfläche namens Objekte markieren (siehe Abb. 3.10), mit der Sie ein Objekt ebenfalls in den Bearbeitungsmodus überführen können, ohne die rechte Maustaste benutzen zu müssen.

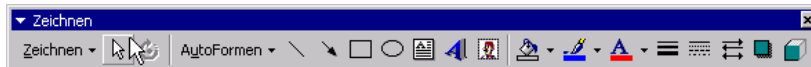


Abbildung 3.10: Symbolleiste Zeichnen, Schaltfläche Objekte markieren

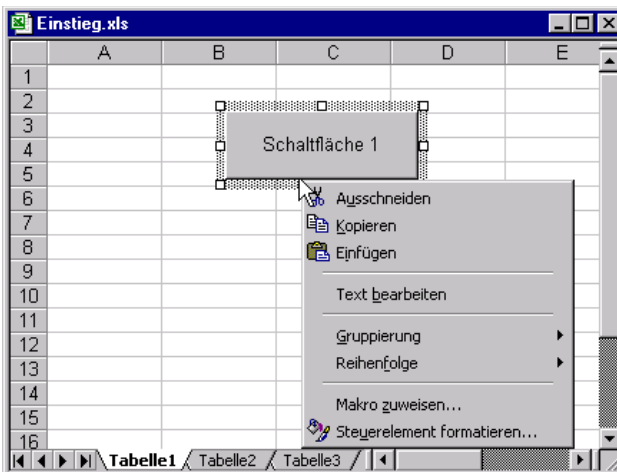


Abbildung 3.11: Kontextmenü einer Schaltfläche

Unser Anfangsbeispiel ist zu Ende. Zuerst haben wir also etwas Code geschrieben und danach dafür gesorgt, dass der Code auch ausgeführt wird. In diesem Beispiel übernahm eine Schaltfläche aus dem etwas antiquierten Reservoir Formular diese Aufgabe, die jedoch den unbestreitbaren Vorteil hat, dass sie nach einfachen Mustern funktioniert. Außerdem kann es nicht schaden, wenn man sie einmal gesehen und angewandt hat.

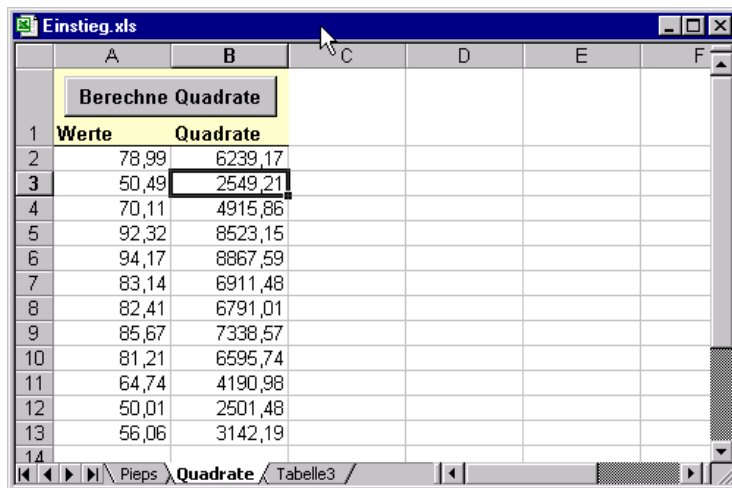
Für den auf die Schnelle geschriebenen Fünfzeiler, der mir meine eigene Arbeit mit Excel erleichtern soll, verwende ich sie heute noch. Es ist jedoch zu vermuten, dass sie früher oder später nicht mehr unterstützt werden, was vermutlich dann der Fall sein wird, wenn von Excel 5 und '95 wirklich niemand mehr spricht.

Wir werden das überleben, denn die Steuerelemente aus der Steuerelement-Toolbox sind um einiges leistungsfähiger, und die Zeiten, als sie, in Tabellen eingesetzt, gelegentlich den Dr. Watson zu Rate zogen, gehören auch der Vergangenheit an. Und in den Dialogen waren sie schon bei Excel '97 das einzig Sinnvolle.

3.2 Etwas Zellzugriff

Die meisten in Excel-VBA geschriebenen Anwendungen drehen sich irgendwie um Tabellenzellen, denn Tabellen sind das eigentlich herausragende Merkmal von Excel. In Kapitel 6 – Zugriff auf Excel-Objekte werden Sie gründlich mit dem Umgang mit Zellzugriffen vertraut gemacht. Aber ein wenig vorgreifen müssen wir doch. Wir werden jedoch versuchen, den Ball flach zu halten.

In Abbildung 3.12 ist das Ziel unserer Bemühungen zu sehen. In einem zusammenhängenden Zellbereich in Spalte A unserer zweiten Tabelle der Arbeitsmappe stehen Zufallswerte, die in den korrespondierenden Zellen der Spalte B ins Quadrat gesetzt werden sollen. Die Tabelle selbst wurde in »Quadrate« umbenannt.



	A	B	C	D	E	F
	Berechne Quadrate					
1	Werte	Quadrate				
2	78,99	6239,17				
3	50,49	2549,21				
4	70,11	4915,86				
5	92,32	8523,15				
6	94,17	8867,59				
7	83,14	6911,48				
8	82,41	6791,01				
9	85,67	7338,57				
10	81,21	6595,74				
11	64,74	4190,98				
12	50,01	2501,48				
13	56,06	3142,19				
14						

Abbildung 3.12: Tabelle Quadrate, wie sie einmal aussehen soll

Programme wie Excel bestehen aus zwei wesentlichen Komponenten:

- ▼ Einem Objektmodell mit allen Fähigkeiten, die zur Erledigung der Aufgaben des Programmes erforderlich sind.

- ▼ Einer Oberfläche mit Menüs und Symbolleisten, die als Benutzerschnittstelle zwischen Anwender und Objektmodell angesiedelt ist.

Das Excel, welches sich dem normalen Anwender präsentiert, ist genau diese Oberfläche, diese Benutzerschnittstelle. Dort sind Auszüge aus dem Funktionsumfang der Bibliothek in ansprechender Form durch Tastatur- und Mausektionen zugreifbar. Manches sind einzelne Funktionalitäten, die auch als einzelne Elemente des Objektmodells existieren, andere dem Benutzer zur Verfügung stehende Befehle beinhalten wiederum Metabefehle, die aus einer Fülle einzelner Aktionen am Objektmodell zusammengesetzt sind.

Noch drastischer tritt diese Zweiteilung bei Access zu Tage, denn viele Programme nutzen Access-Datenbanken, ohne Access irgendwie daran zu beteiligen. Vielleicht sollte man sagen, sie nutzen dieselben Datenbanken, die auch Access benutzt. Access ist also eine Bedienoberfläche für (genau genommen ISAM-)Datenbanken. Auch wir werden noch auf ISAM-Datenbanken zugreifen, ohne Access zu behelligen.

3.2.1 Excel-Objekte – eine erste Annäherung

Schauen wir uns mal eine grobe Darstellung einiger Excel-Objekte an:

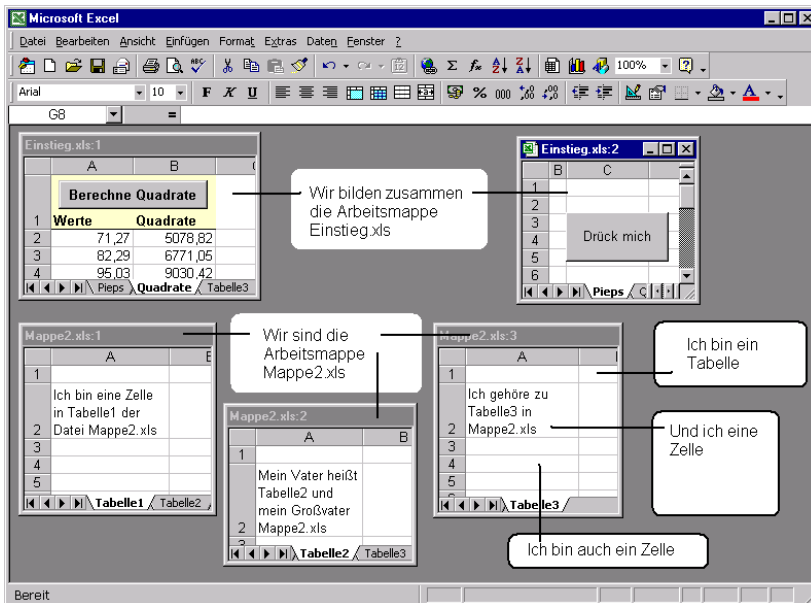


Abbildung 3.13: Excel-Objekte

Wir sehen also, dass Excel hier ein Container ist, in dem aktuell die beiden Arbeitsmappen Einstieg.xls und Mappe2.xls enthalten sind. Jede dieser Arbeitsmappen verfügt über mehrere Tabellen, und jede dieser Tabellen über 265 x 65536, also 16.777.216 Zellen.

Nun benötigen wir noch einen Oberbegriff für Arbeitsmappe, Tabellenblätter und Zellen, und dieser Begriff ist Objekt oder Klasse. Den Unterschied werden wir in Kapitel 5 – Objekte noch genauer untersuchen. Für unsere Zwecke ist Objekt hier genau genug.

**Workbook,
Worksheet,
Range und Cell**

Wenn wir gerade bei Begrifflichkeiten sind, so können wir uns auch die englischen Objektnamen angewöhnen, da wir mit den deutschen Begriffen in VBA nicht weit kommen. Eine Arbeitsmappe nennt man *Workbook* und eine Tabelle *Worksheet*. Bei Zellen ist es nicht ganz so einfach, denn eine Zelle wird als kleinste Menge eines Zellbereichs zu einem *Range*, taucht aber gelegentlich auch als *Cell* auf, wenn gezielt von einer Zelle eines Zellbereichs die Rede ist.

Schauen wir uns die hierarchische Struktur dieser Objekte in einer Grafik an:

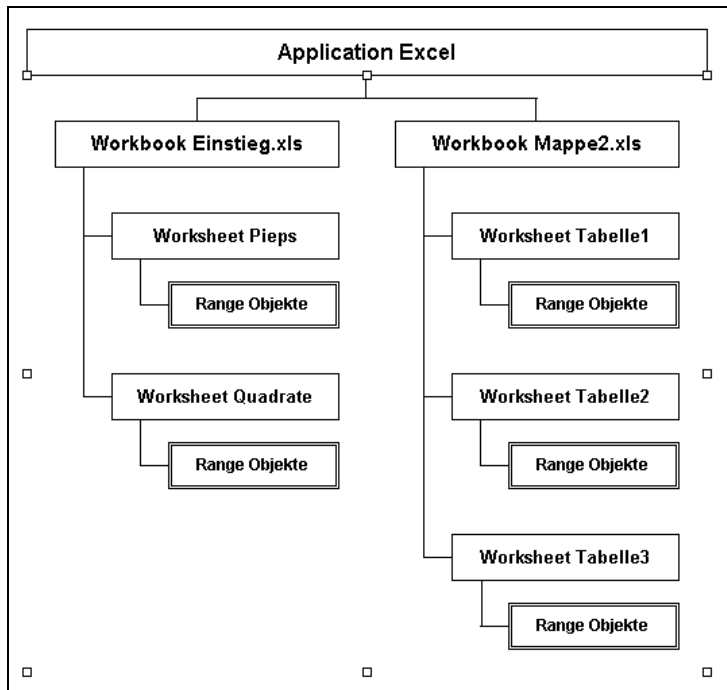


Abbildung 3.14: Excel-Objekthierarchie

Solche zusammengehörigen Objekte werden unter dem Begriff Objektmodell zusammengefasst. Auch das Wort Bibliothek ist gebräuchlich, wobei es meist verwendet wird, wenn von der Datei die Rede ist, in der das Objektmodell abgebildet ist.

Zu unseren **Workbooks**, **Worksheets** und den **Range**-Objekten hat sich ein Weiteres gesellt, denn die **Workbooks** hängen nicht in der Luft, sondern sind quasi unter einem anderen Objekt aufgehängt, das sich **Application** nennt. Dieses **Application**-Objekt thront auch in anderen Objektmodellen über den ersten eigentlichen Objekten des jeweiligen Objektmodells. Bei Word beispielsweise sind es die **Document**-Objekte, aus denen die Ebene unter dem **Application**-Objekt gebildet wird.

Wenn VBA-Programme auf Zellen zugreifen, dann greifen sie gewissermaßen von außen in das Objektmodell und erwarten einen Zugriff auf die gewünschte Zelle, um anschließend beispielsweise den Inhalt oder die Formatierung dieser Zelle zu ändern. Sind in unserer aktuellen Instanz von Excel aber mehrere Arbeitsmappen geöffnet, die jeweils über mehrere Tabellenblätter verfügen, so können Sie sich die Verwirrung vorstellen.

3.2.2 Gezielter Zugriff auf eine Zelle

Was tun Sie in einer Excel-Zelle, wenn Sie auf eine Zelle einer fremden Arbeitsmappe zugreifen?

Sie geben den vollständigen Bezug inklusive Arbeitsmappe, Tabellenblatt und Zelladresse ein:

```
=[ExterneDatei.xls]Tabelle1!A1
```

Bleibt der Bezug innerhalb der Datei, so wird daraus:

```
=Tabelle1!A1
```

und innerhalb derselben Tabelle ein schlichtes

```
=A1
```

Dieser Zellbezug steht aber in der Tabelle selbst, wodurch ein eindeutiger Bezug zu dieser Tabelle hergestellt werden kann. Dies ist in der Entwicklungsumgebung nicht der Fall! Wir befinden uns (nicht nur in der Taskleiste) außerhalb des aktiven Excels. Zwar verfügt jedes laufende Excel, in der mindestens eine Arbeitsmappe geöffnet ist, über eine aktive Arbeitsmappe und diese über ein aktives Blatt, aber wir sollten diesen mit Annahmen gespickten Weg niemals gehen. Ersetzen Sie die *Annahme*, dass es sich hierbei um unser gewünschtes Ziel handelt, durch die *Sicherheit*, auf das richtige Tabellenblatt zuzugreifen.

Wir haben hier gleich zwei Möglichkeiten, die ich Ihnen beide vorstellen möchte. Die erste besteht darin, per Code nacheinander die gewünschte Arbeitsmappe und die Tabelle zu aktivieren, um anschließend in einer dritten Zeile die gewünschten Operationen durchzuführen. Dies ist weit verbreitet und ein sicheres Zeichen dafür, dass der Betreffende entweder Objekte im Allgemeinen, Excelobjekte im Besonderen oder beides nicht verstanden hat. Da es das ausgemachte Ziel dieses Buches ist, beides zu vermitteln, werden wir uns nun dem zweiten Weg zuwenden.

ThisWorkbook Der Code, den wir in der Entwicklungsumgebung zum Besten geben, ist natürlich immer einer Datei zugeordnet und in dieser später gespeichert. In der Regel werden Sie den Code in die Modulblätter einer Arbeitsmappe schreiben, die in unserer Diktion ein `Workbook` ist. Für die Arbeitsmappe, in der sich der aktuelle Code befindet, gibt es einen besonderen Ausdruck, der sich `ThisWorkbook` nennt. Durch dieses Ausdruck `ThisWorkbook` haben wir bereits einen Bezug zu der gewünschten Arbeitsmappe, der nun lediglich noch durch die Auswahl des gewünschten Tabellenblatts ergänzt werden muss, in unserem Fall die Tabelle »Quadrante«. Und das sieht so aus:

```
ThisWorkbook.Worksheets("Quadrante")
```

Auf deutsch: in dieser Arbeitsmappe wähle aus den Tabellenblättern das aus, welches den Namen »Quadrante« trägt. Der Punkt, der zwischen `ThisWorkbook` und `Worksheets(..)` steht, wird generell zwischen die einzelnen Elemente eines Objektausdrucks geschrieben.

Fehlt noch die Zelle selbst, wozu wir uns des Objekts `Range` bedienen. Die Zelle B2 kann mit `Range(»B2«)` angesprochen werden, wodurch unser Ausdruck jetzt so aussieht:

```
ThisWorkbook.Worksheets("Quadrante").Range("B2")
```

Nun ist aber zwischen einer Zelle als `Range`-Objekt und ihrem Inhalt ein gewaltiger Unterschied. Bislang haben wir lediglich das `Range`-Objekt. Um an den Inhalt zu gelangen, müssen wir dem Objektausdruck noch mitteilen, dass wir den Inhalt, oder auch Wert, wie der korrekte Ausdruck lautet, verändern wollen:

```
ThisWorkbook.Worksheets("Quadrante").Range("B2").Value
```

Nun ist der Ausdruck fertig. Was fehlt, ist noch der Wert, der dieser Zelle zugeordnet werden soll:

```
ThisWorkbook.Worksheets("Quadrante").Range("B2").Value = 2
```


Wechseln Sie bitte in Ihre Entwicklungsumgebung und rufen Sie dort das Direktfenster auf, das sich im Menü Ansicht verbirgt. Die Anhänger der ShortCuts mögen es bitte mit (Strg)(G) versuchen. Dort schreiben Sie nun diesen Ausdruck in eine neue Zeile:

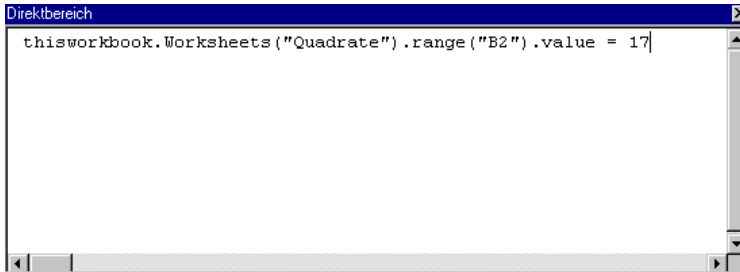


Abbildung 3.15: Zugriff auf Range-Objekt im Direktfenster

Schließen Sie die Zeile bitte mit Return ab. Ein Blick in die Tabelle Quadrat müsste nun ergeben, dass dort in Zelle B2 die Zahl 17 steht.

Es dürfte Ihnen auch aufgefallen sein, dass die Entwicklungsumgebung Sie beim Schreiben dieser Zeile unterstützt hat. So erschienen nach dem ».« hinter `ThisWorkbook` in einem eigenen Fenster alle Begriffe, die als Eigenschaften oder Methoden eines `Workbook`-Objekts möglich sind. Das Handwerkliche dazu ist ausführlich in Kapitel 2 – Die Entwicklungsumgebung dargestellt. Wie Sie auch bemerkt haben, versagte dieser Mechanismus nach dem Punkt hinter dem zweiten Punkt, was eine alte und gleichermaßen unerfreuliche Erscheinung ist. Der Punkt hinter dem `Range`-Objekt veranlasste VBA auch nicht dazu, die Elemente des `Range`-Objektes aufzulisten. Das stört uns aber nicht weiter, weil es zu unserer bisherigen Vorgehensweise eine Erleichterung gibt, bei der es wieder funktioniert.

Hierzu müssen wir uns das Eigenschaftsfenster in Abb. 3.16 unten links näher ansehen:

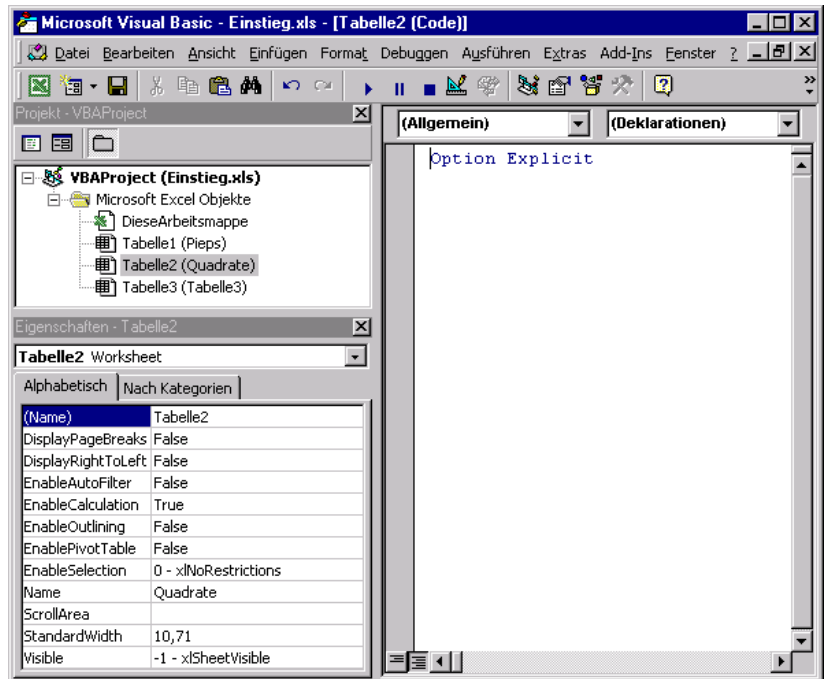


Abbildung 3.16: Entwicklungsumgebung vor Änderung des Worksheet-Objekts

Wir sehen eine Name-Eigenschaft an viertletzter Stelle, der rechts daneben Quadrate zugeordnet ist. Hierbei handelt es sich um den Namen des Blattes, wie er in Excel selbst definiert wurde. Sie können diesen Werte spaßeshalber ändern und in Excel nachsehen, was sich tut.

An erster Stelle befindet sich der Begriff Name erneut, aber diesmal in Klammern. Der dort stehenden Begriff `Tabelle2` ist der Name, unter dem wir die Tabelle im Code direkt ansprechen können. Probieren Sie es aus. Schreiben Sie den folgenden Ausdruck ins Direktfenster:

```
Tabelle2.Range("B3").Value = 18
```

In Zelle B3 der Tabelle Quadrate steht nun die Zahl 18.

Wie Sie feststellen konnten, funktioniert nun auch das automatische Auflisten der Elemente nach `Tabelle2` und nach `Range("B3")`. Wenn nicht, so sollten Sie die Option Elemente automatisch auflisten im Register Editor des Optionsdialogs aktivieren.

Wenn wir in einem Programm auf mehrere Tabellen zugreifen, so leuchtet ein, dass ein Objektname wie `Tabelle2` kein Ausdruck ist, mit dem wir sonderlich viel anfangen können, wenn er uns begegnet. Ein Objektname sollte generell auf den ersten Blick offenbaren, auf welche Tabelle er sich bezieht. Unsere Tabelle trägt den Namen `Quadrate`, somit sollte der Objektname irgendeinen Bezug dazu herstellen und nach Möglichkeit kurz sein. Wenn der Ausdruck später auch noch erkennen lässt, dass der Ausdruck eine Tabelle darstellt, so ist der Name perfekt. Für die letzte dieser Anforderungen verwendet man einen Präfix, also einen (zumeist) dreistelligen Bezeichner, der den Namensrumpf einleitet. Der Präfix für ein Worksheet ist `sht`. Als Namensrumpf käme zum Beispiel `Squares` in Frage. Tragen Sie also rechts des Begriffs (Name) im Eigenschaftsfenster `shtSquares` ein. Der Großbuchstabe dient zum einen der optischen Trennung des Präfix vom Rumpf, hat zum anderen aber auch den bereits in Kapitel 2 besprochenen Vorteil, dass der Codeeditor zukünftig genau auf diese Schreibweise zurückgreifen wird und alle unsere `shtsquares` in `shtSquares` umwandelt. An dieser Umwandlung werden wir erkennen, dass der Begriff korrekt geschrieben wurde.

Ein kleiner Test im Direktfenster wird Ihnen zeigen, dass wir diesen neuen Objektnamen zukünftig verwenden können:

```
shtSquares.Range("B4").Value = 19
```

Zurück zu unserer Aufgabenstellung. Wir wollten in der Tabelle `Quadrate` in Spalte B die Quadrate der jeweils in Spalte A stehenden Werte bilden.

Diesmal wollten wir das Programm nicht über eine Schaltfläche der Symbolleiste Formular starten, sondern die Symbolleiste Steuerelement-Toolbox benutzen. Hierzu müssen wir die Schaltfläche zuerst erzeugen, denn sie lässt sich nicht einfach einer Prozedur zuweisen. Sie hat ihre eigenen Vorstellungen davon, wo die Prozedur steht und wie sie heißen soll.

Also zurück in die Tabelle. Aktivieren Sie dort bitte die Symbolleiste Steuerelement-Toolbox und platzieren Sie eine Schaltfläche an der gewünschten Stelle. Das Prozedere ist genau wie bei der anderen Schaltfläche.

Klicken Sie nun bitte auf die Schaltfläche Eigenschaften der Symbolleiste Steuerelement-Toolbox. Das nun erscheinende Eigenschaftsfenster ist uns bereits aus der Entwicklungsumgebung bekannt.

Abbildung 3.17 zeigt nun das Ensemble, das sich Ihnen präsentiert.



Was zur (Name)-Eigenschaft des Worksheets gesagt wurde, gilt natürlich auch für den `CommandButton`: Das Kind braucht einen Namen. Der Präfix lautet `cmd` und der Namensrumpf sollte etwas mit der Aufgabe zu tun haben, die der Schaltfläche zugedacht ist. Tragen Sie hier bitte `cmdCompute` ein. Die Aufschrift der Schaltfläche verbirgt sich hinter der Eigenschaft `Caption`. Sie können beispielsweise *Quadrate berechnen* eintragen oder auch irgendetwas anderes, denn diese Eigenschaft hat keinerlei Auswirkungen auf den Code.

Nun zu der hinterlistigen Eigenschaft, die den Namen `TakeFocusOnClick` trägt. Der Name ist Programm, denn sie raubt ihrem Container, also der Tabelle, den Fokus, die somit ihre aktive Zelle einbüßt. Und dies hindert `Range`-Objekte am Zugriff auf die Zellen. In Excel 97 sollten Sie diese Eigenschaft unbedingt auf `False` setzen.



Im Prinzip ist unsere Arbeit in der Tabelle abgeschlossen und das Eigenschaftsfenster kann geschlossen werden. Auf der Symbolleiste Steuerelement-Toolbox befindet sich eine Schaltfläche `CODE ANZEIGEN`. Wenn Sie darauf klicken, wechselt Excel in die Entwicklungsumgebung und zeigt das Codefenster der Tabelle an. Und darin erwartet uns bereits ein fertiger Prozedurrahmen:

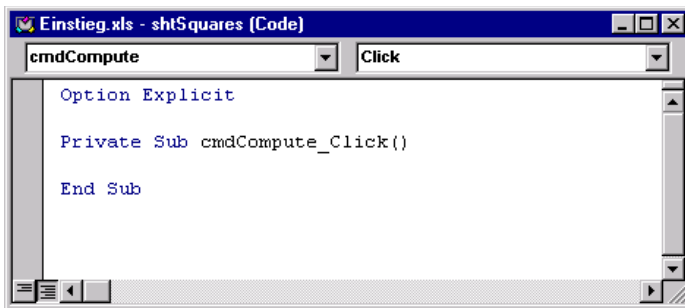


Abbildung 3.18: Codefenster mit Prozedurrahmen `cmdCompute_Click`

In der ComboBox links oben startet uns bereits der Objektname unserer Schaltfläche `cmdCompute` entgegen und in der rechten ComboBox sehen wir den Ausdruck `Click`. Klappen Sie diese rechte ComboBox auf, so sehen Sie alle Ereignisse alphabetisch aufgelistet, die diese Schaltfläche beherrschen. Klicken Sie dort beispielsweise auf `GotFocus`, so legt der Editor eine neue Routine des Namens `cmdCompute_GotFocus` an.

Das, was uns in dieser rechten ComboBox an Einträgen erwartet, sind die so genannten Ereignisse, die eine Schaltfläche verarbeiten und weiter melden kann. Erschreckend, nicht wahr. Aber trösten Sie sich, Sie werden bei Schaltflächen selten mehr als das `Click`-Ereignis benötigen, und das ist das Standardereignis einer Schaltfläche.

Was sind Ereignisse?

Ereignisse

Ereignisse sind die Fähigkeit von ActiveX-Objekten, automatisch andere Prozeduren aufzurufen, und ein Ereignis ist der Aufruf einer solchen Prozedur durch ein ActiveX-Objekt.

Damit das (ActiveX-)Objekt die aufzurufende Prozedur auch findet, muss diese den Namenskonventionen folgen (und die Methodensignatur über-

einstimmen). Sie müssen sich jedoch mit diesen Details nicht auseinander setzen, solange Sie nicht selbst ActiveX-Objekte entwerfen, und bis dahin haben wir noch ein paar hundert Seiten Zeit.

Sie wissen, dass sich in einem Codemodul alle damit assoziierten Objekte dadurch bemerkbar machen, dass sie die linke obere ComboBox bevölkern. Und dort können Sie ein solches Objekt auswählen, woraufhin der Editor automatisch die Ereignisprozedur des Standardereignisses anlegt. Möchten Sie ein anderes Ereignis, so wählen Sie es einfach in der rechten ComboBox aus und der Editor wird es für Sie anlegen. So einfach ist das.

Nun ist es Zeit für ein bisschen Code. Bevor wir uns der Aufgabe zuwenden, die Quadrate in die Tabelle zu transportieren, machen wir einen kurzen Test mit einer einfachen MessageBox. Schreiben Sie die folgende Anweisung in unsere Ereignisprozedur:

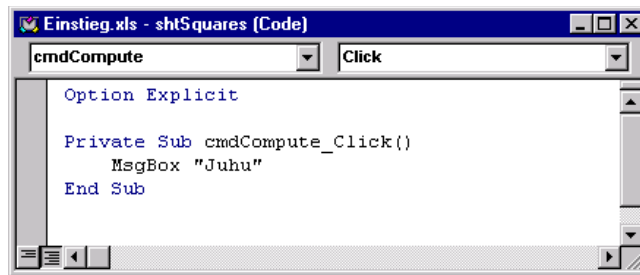


Abbildung 3.19: Test der Ereignisroutine mit einer MessageBox

Wenn Sie nun zu Excel wechseln, wird die Tabelle vermutlich noch im Entwurfsmodus sein, den Sie mit der Schaltfläche ENTWURFSMODUS BEENDEN der Symbolleiste Steuerelement-Toolbox abschließen können.

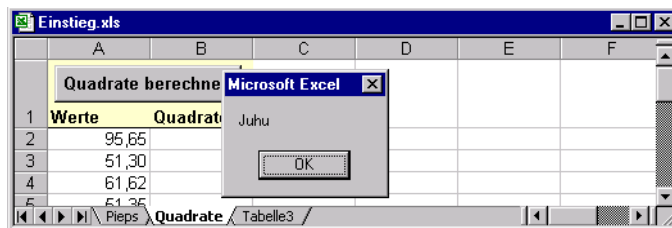


Abbildung 3.20: Tabelle mit Test-MessageBox

Ein Klick auf die Schaltfläche müsste ein ähnliches Bild wie in Abbildung 3.20 hervorbringen. Soweit funktioniert unsere bisherige Konstruktion also. Nun stellen wir uns dem Problem, die Quadrate zu berechnen und einzutragen.

3.2.3 Dynamischer Zugriff auf Zellen

Bislang griffen wir auf die Zellen zu, indem wir absolute Zellbezüge in unseren Ausdrücken verwendeten. Unsere Aufgabe aber, eine Reihe von Werten in einer Tabelle einzutragen, kann mit absoluten Zugriffen wohl kaum zufrieden stellend gelöst werden. Hätten wir 200 Werte zu berechnen, benötigten wir 200 Programmzeilen.

Wir wissen lediglich, dass die Werte ab der zweiten Zeile einzutragen sind. Wir kennen die letzte Zeile und nehmen als Wert dafür 12 an.

Sie erinnern sich sicherlich noch an unser `Worksheet`-Objekt mit dem Namen `shtSquares`. Ein `Worksheet`-Objekt verfügt über die Methode `Cells`, die über zwei ganzzahlig numerische Argumente verfügt. Das erste ist eine Zeilennummer und das zweite eine Spaltennummer. Das Ergebnis dieser Methode ist ein `Range`-Objekt, also das, was wir benötigen.

```
shtSquares.Cells(3, 2)
```

ergibt ein `Range`-Objekt in der dritten Zeile und zweiten Spalte. Wollen wir etwas in eine Zelle hineinschreiben, so sollten wir auch die entsprechende Eigenschaft hinter das `Range`-Objekt schreiben, die den Wert der Zelle darstellt:

```
shtSquares.Cells(3, 2).Value
```

Um alle zu bearbeitenden Werte zu quadrieren, müssen wir nacheinander die Zeilen 2 bis 12 erreichen. Zu diesem Zweck gibt es so genannte Variablen. Nach Brockhaus ist eine Variable ein benannter Speicherplatz. Für uns ist es ein Ausdruck, der sprichwörtlich variabel ist, um nacheinander die Werte 2 bis 12 einzunehmen.

Variablen

Bevor man Variablen verwendet kann, sollten sie deklariert werden. Innerhalb einer Prozedur erfolgt dies mit der Anweisung `Dim`, gefolgt von dem Namen der Variablen und dem Typ. Diese Variablendeklaration erfolgt üblicherweise im Kopf einer Prozedur. Schreiben Sie bitte in die erste Zeile unserer Prozedur:

```
Dim iRow As Long
```

`i` ist ein in der Mathematik üblicher Präfix für einen Zeiger, und nichts anderes ist unsere Zeilenvariable. Der Namensrumpf `Row` sagt uns, dass es sich um einen Zeilenzeiger handelt. Das Schlüsselwort `As` leitet die Typdeklaration ein, die `Long` als Typ enthält. Der `Long`-Datentyp ist ein ganzzahliger Wert, der sich grob zwischen -2 Milliarden und $+2$ Milliarden bewegt. Wer's genauer wissen will: zwischen -2^{31} und $+2^{31}-1$. In dieser Zeile müssen Sie nur das `R` aus `iRow` groß schreiben, weil das die Stelle ist, welche der Editor für zukünftige Schreibweisen heranzieht. Über das Thema

Variablen und Datentypen gibt Kapitel 4 – Sprachelemente, die erste erschöpfende Auskunft. Für unsere Zwecke genügt das, was bisher dazu gesagt wurde.

Die so geschaffene Variable können wir nun in der `Cells`-Methode an der Position des Zeilenzeigers einsetzen:

```
shtSquares.Cells(iRow, 2).Value
```

Was wir nun benötigen, ist eine Konstruktion, die unsere Variablen nacheinander mit den gewünschten Werten versorgt:

```
For iRow = 2 To 12 Step 1
```

```
Next
```

Die `For`-Zeile dieser `For-Next`-Schleife wird beim ersten Mal der Variablen `iRow` den Wert 2 zuweisen, die sich dann mit diesem in den Parcours zwischen `For`- und `Next`-Zeile begibt. Bei `Next` angelangt, wird sie zurückgeschickt, wo ihr dann der Wert der Schrittweite hinter dem Schlüsselwort `Step` hinzuaddiert wird. Aus dem vormaligen 2 und der Schrittweite 1 wird 3 und die Variable marschiert mit diesem neuen Wert los, bis sie von der `Next`-Anweisung erneut nach oben geschickt wird. Wie Sie richtig vermuten, wird dieser Vorgang so lange wiederholt, bis `iRow` einen Wert aufweist, der größer ist als die mit 12 angegebene obere Grenze. Danach wird die der `Next`-Anweisung folgende Zeile abgearbeitet.

Unsere Prozedur sieht nun so aus:

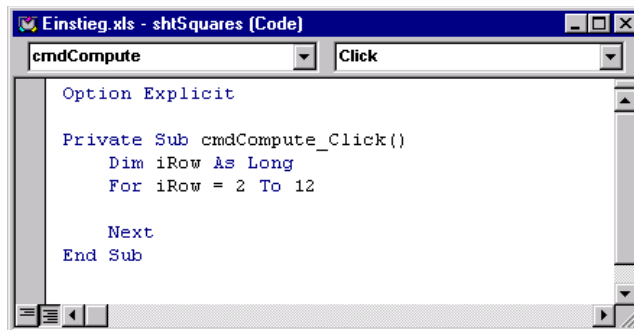


Abbildung 3.21: `cmdCompute_Click` mit `For-Next`-Schleife

Es fehlt nun nur noch die Zeile, in der die eigentliche Berechnung stattfindet. In VBA werden Zuweisung generell so geschrieben:

```
Ziel = Quelle
```


Das Ziel in unserem Beispiel ist die Zelle der 2. Spalte in der durch `iRow` bezeichneten Zeile:

```
shtSquares.Cells(iRow, 2).Value
```

Und die Quelle ist das Quadrat des Wertes der 1. Spalte in der Zeile `iRow`:

```
shtSquares.Cells(iRow, 1).Value ^ 2
```

Das Caret-Zeichen `^` dient ebenso wie in Excel der Potenzierung. Aus Platzgründen wird die vollständige Zeile hier umgebrochen. Hierzu muss an der gewünschten Stelle ein Leerzeichen gefolgt von einem Tiefstrich `_` stehen. Danach können Sie mit der Return-Taste einen Zeilenumbruch erzeugen und in der nächsten Zeile weiterschreiben:

```
shtSquares.Cells(iRow, 1).Value = _  
    shtSquares.Cells(iRow, 1).Value ^ 2
```

In den Folgezeilen einer umgebrochenen Befehlszeile rücke ich üblicherweise signifikant ein, damit der Umbruch wirklich ins Auge fällt. Wenn Sie nicht gerade einen Miniaturmonitor vor sich haben, müssen Sie vermutlich nicht umbrechen.

Hier noch einmal unsere Prozedur zur Kontrolle:

```
Private Sub cmdCompute_Click()  
    Dim iRow As Long  
    For iRow = 2 To 12 Step 1  
        shtSquares.Cells(iRow, 1).Value = _  
            shtSquares.Cells(iRow, 1).Value ^ 2  
    Next  
End Sub
```

Dann lassen Sie das gute Stück mal laufen. Zurück nach Excel und ein mutiger Klick auf die Schaltfläche.

3.2.4 Ermittlung der letzten belegten Zeile

VBA und insbesondere gerade Excel-VBA ist die Plattform der tausend Möglichkeiten. Auch hier stehen uns verschiedene Wege offen, um die letzte zu berechnende Zeile zu ermitteln.

So könnten wir zum Beispiel prüfen, ob in Spalte A der aktuellen Zeile überhaupt noch etwas steht. Das würde uns aber vor das Problem stellen, dass wir die obere Grenze der `For`-Anweisung mächtig nach oben schrauben müssten. Im Grunde müssten wir einen anderen Schleifentyp wählen.

Wir haben eine andere Möglichkeit, die Sie dem Grunde nach vielleicht sogar von Excel her kennen. Zur Verdeutlichung wurden vor unserem Zellbereich noch eine Zeile und eine Spalte eingefügt. Wenn Sie die Aktion nachvollziehen wollen, so vergessen Sie danach bitte nicht, die Zeile und die Spalte wieder zu löschen.

Hier nun die Tabelle:

	A	B	C	D	E	F
1						
2		Quadrate berechnen				
3		Werte	Quadrate			
4		50,37	2537,14			
5		96,84	9377,99			
6		94,75	8977,56			
7		61,30	3757,69			
8		79,67	6347,31			
9		55,68	3100,26			
10		85,43	7298,28			
11		54,15	2932,22			
12		65,68	4313,86			
13		94,04	8843,52			

Abbildung 3.22: Tabelle ohne markierten aktuellen Bereich

Setzen Sie Ihren Cursor bitte in eine beliebige Zelle des belegten Tabellenbereichs. Danach benutzen Sie mit gedrückter Steuerungstaste den Multiplikationsoperator im numerischen Tastaturblock. Ihre Markierung in der Tabelle müsste nun so aussehen:

	A	B	C	D	E	F
1						
2		Quadrate berechnen				
3		Werte	Quadrate			
4		50,37	2537,14			
5		96,84	9377,99			
6		94,75	8977,56			
7		61,30	3757,69			
8		79,67	6347,31			
9		55,68	3100,26			
10		85,43	7298,28			
11		54,15	2932,22			
12		65,68	4313,86			
13		94,04	8843,52			

Abbildung 3.23: Tabelle mit markiertem aktuellen Bereich

Dieser nette und nützliche Effekt nennt sich aktueller Bereich. Und da sich die Oberfläche Excel letztlich der Excel-Bibliothek bedient, muss es etwas Ähnliches auch als Methode eines Excel-Objekts geben. Das Objekt heißt `Range` und die Methode `CurrentRegion`. Was die `CurrentRegion` genau bewirkt, sehen wir noch in Kapitel 6 im Detail.

Uns genügt hier, dass sie ein `Range`-Objekt erzeugt, das in Zeilen- und Spaltenrichtung vollständig von leeren Zellen umgeben ist. Nun besitzt aber jedes `Range`-Objekt wiederum eine Eigenschaft, die uns die von diesem `Range`-Objekt berührten Zeilen zurückgibt. Und diese Eigenschaft heißt `Rows`. Eine zweite Eigenschaft namens `Columns` gibt entsprechend die beteiligten Spalten zurück. Ergänzt durch die `Count`-Eigenschaft erhalten wir die betroffenen Zeilen als Zahl.

Nachdem wir den aktuellen Bereich erzeugt haben, müssen wir ihn also lediglich nach seiner `Rows`-Eigenschaft befragen und haben die obere Grenze für unsere Schleife.

Zu diesem Zweck bauen wir eine zweite Variable, die diese Zeilenzahl für uns aufbewahren soll:

```
Dim nRows As Long
```

`n` ist ebenfalls in der Mathematik für eine unbestimmte Zahl gebräuchlich. Da sie die Anzahl der Zeilen (`Rows`) aufnehmen soll, liegt `nRows` auf der Hand. Hier nun die Zuweisung an diese Variable:

```
nRows = shtSquares.Cells(1, 1).CurrentRegion.Rows.Count
```

Diese `nRows` muss noch an Stelle der Zahl 12 in die `For`-Zeile integriert werden, wonach unsere Prozedur so aussieht:

```
Private Sub cmdCompute_Click()
    Dim iRow As Long
    Dim nRows As Long
    nRows = shtSquares.Cells(1,
1).CurrentRegion.Rows.Count
    For iRow = 2 To nRows Step 1
        shtSquares.Cells(iRow, 1).Value = _
            shtSquares.Cells(iRow, 1).Value ^ 2
    Next
End Sub
```

3.2.5 Rückblick

Dieser Einstieg begann recht banal mit dem Pieps und wuchs im zweiten Beispiel zu einer Prozedur an, die zwar nicht sonderlich groß wurde, aber dennoch ein paar Hürden aufwies.

Es beginnt schon mit der Namensvergabe. Solange es keine bindende Unternehmensrichtlinie bei Ihnen gibt, können Sie Ihrer Fantasie freien Lauf lassen. Es kommt darauf an, das Sie während des Schreibens des Codes wissen, was welche Variable tut. Und wenn Sie nach Wochen anhand des Namens auch sofort erkennen, was die Aufgabe der Variablen oder der Prozedur ist, dann ist der Name offensichtlich gut gewählt. Mit der Zeit werden Sie Ihren eigenen Stil entwickeln.

Wie wird man jedoch mit der riesigen Menge an Sprachelementen fertig? Das ist schon ein ernsteres Problem. Zu Anfang werden Sie schon eine Menge neuer Begriffe erlernen müssen, doch bei weitem nicht in dem Umfang, wie in diesem Buch dargestellt. Denn Tausende von VBA-Entwicklern kamen über die Runden, bevor mit die mit VB 6 und VBA 6 neu eingeführten `Split`- und `Join`-Funktionen zur Verfügung standen. Natürlich ist es heute einfacher, eine mit Trennzeichen versehene Zeichenkette in ein Datenfeld zu verwandeln. Doch als diese nicht zur Verfügung standen, schrieb man sich eben eine kleine Funktion, die genau das erledigte. Auch die neue `InStrRev`-Funktion, die das erste Auftauchen eines Zeichens in einer Zeichenkette von der *rechten* Seite her ermittelt, ermöglicht prinzipiell keine neue Funktionalität. Ohne diese Funktion hängelte man sich einfach zeichenweise von rechts nach links und stieß auch früher oder später auf dieses Zeichen. Eine solche Funktionalität benötigt man beispielsweise, um den reinen Dateinamen aus einem vollständigen Namen nebst Pfad zu ermitteln, denn dieser Name beginnt hinter dem letzten »\«. Wer das zum zweiten Mal benötigt, wird sich ohnehin eine Funktion schreiben und diese immer dann aufrufen, wenn er sie benötigt. Ab da geht das Codieren ohne die `InStrRev` genauso schnell wie mit ihr. Das Programm wird ein paar Millisekunden länger laufen, doch das fällt sicherlich nicht ins Gewicht.

Das ist eine lange Vorrede für eine Binsenweisheit:

Sie müssen nicht den vollständigen Sprachumfang von VBA im Repertoire haben, um gute Programme zu schreiben. Gute Programme zeichnen sich durch Datensicherheit, Stabilität und ein gewisses Maß an Bedienkomfort aus. Das interessiert die Anwender, nicht Ihre Akrobatik oder die phänomenale Eleganz der Anwendung.

Selbst nach Jahren intensiver Beschäftigung mit VBA wird man regelmäßig von neuen Erkenntnissen überrascht. Manchmal sind sie einem bewusst, oftmals aber stellt man erst bei einem mehr zufälligen Blick in älteren Code mit Erstaunen fest, wie sich doch in letzter Zeit der eigene Stil verändert hat. Man hätte aber damals Stein und Bein geschworen, *das* Programm geschrieben zu haben, über das die Nachwelt noch reden wird.



Sprachelemente, die erste

4

4.1	Datentypen und Variablen	113
4.1.1	Byte, Integer und Long	115
4.1.2	Single und Double	115
4.1.3	Currency	116
4.1.4	Date	116
4.1.5	Boolean	117
4.1.6	String	117
4.1.7	Object	117
4.1.8	Variant	118
4.1.9	Datenfelder	119
4.1.10	Sonderzustände von Datentypen	120
4.2	Modulooptionen	120
4.2.1	Option Base 0 1	121
4.2.2	Option Compare Binary Text	121
4.2.3	Option Explicit	121
4.2.4	Option Private Module	121
4.3	Prozeduren	121
4.3.1	Sub-Prozeduren	122
4.3.2	Function-Prozeduren	123
4.3.3	Übergabe von Argumenten	124
4.4	Verzweigungen	126
4.4.1	Verzweigungen mit If-Anweisungen	127
4.4.2	Verzweigungen mit Select Case-Anweisungen	130
4.5	Schleifen	132
4.5.1	For-Next-Schleifen	132
4.5.2	Die For-Each-Next-Schleife	133
4.5.3	For-Next versus For-Each-Next	134

4.5.4	Die Do-Loop-Schleife	135
4.5.5	Verschachtelung von Schleifen	136
4.6	Vom Umgang mit Zeichenketten	137
4.6.1	Teilzeichenketten	138
4.6.2	Informationen über Zeichenketten	139
4.6.3	Split und Join	143
4.6.4	Zeichenketten verändern	144
4.6.5	Zeichenketten zusammenfügen	147
4.7	Formatierungen	148
4.7.1	Formatierung von Zeichenketten	148
4.7.2	Formatierung von Zahlen	149
4.7.3	Formatierung von Datums- und Zeitwerten	152
4.8	Numerische Funktionen	156
4.8.1	Konvertierungsfunktionen mit ganzzahligem Ergebnis	156
4.8.2	Konvertierungsfunktionen mit reellem Ergebnis	158
4.9	Datumsfunktionen	158
4.9.1	Standardfunktionen	158
4.9.2	Verzichtbare Datumsfunktionen	161
4.10	Ein- und Ausgabefunktionen	161
4.10.1	MsgBox-Funktion	161
4.10.2	InputDialog-Funktion	165

Wenn wir nach Italien oder Frankreich reisen und versuchen uns, radebrechend verständlich zu machen, so werden wir – je nach Gesichtsausdruck – auch mehr oder weniger verstanden. Ob wir nun das Adverb vor oder hinter das Verb positioniert haben, ist letztendlich egal. Der unsere Konstruktionen interpretierende italienische oder französische Zeitgenosse wird unsere Nachricht verstehen.

Auch die VBA zugrunde liegende Sprache Visual Basic ist eine Art Sprache. Allerdings weist sie keine 300.000 Begriffe auf, sondern in der Ausprägung des Excel-Objektmodells nur etwa 2.000, zählen wir die benannten Argumente hinzu, so wird daraus auch leicht das Doppelte. Allerdings sind wir auch da nicht am Ende angelangt, denn Excel VBA schreitet gewissermaßen nach Datenbanken und sonstigen externen Objektmodellen, die wiederum eine Fülle von Begriffen und Begrifflichkeiten mit sich bringen.

Obwohl der Sprachumfang gegenüber einer herkömmlichen Sprache doch etwas zurücksteht, haben wir auf der anderen Seite eine stringente Syntax und Semantik, die uns jeden Patzer ankreidet. Wir werden Sprachkonstruktionen begegnen, die in Länge und Verschachtelungstiefe durchaus Sätzen aus Traktaten des vorigen Jahrhunderts gleichkommen. Angesichts dieser Komplexität sollten wir von dem Begriff der Makrosprache Abstand nehmen. Abschied nehmen sollten wir auch von der Aussicht, VBA so nebenbei zu erlernen.

Hier steckt nicht die Absicht dahinter, Ihnen von Anfang an den Mut zu rauben. Das Gegenteil ist richtig.

In diesem ersten Teil der Behandlung von Sprachelementen werden wir uns ein Grundgerüst erarbeiten. Hierbei werden wir nur ein bisschen schnorcheln. Im Kapitel »Sprachelemente – die Zweite« binden wir uns eine Pressluftflasche um und beginnen den einen oder anderen Abstieg in tiefere Gewässer, wo uns dann auch wieder Elemente begegnen werden, die in diesem ersten Teil schon besprochen sind.

4.1 Datentypen und Variablen

Verschiedene Datentypen gehören zuerst einmal nicht zur Grundausstattung des Homo Sapiens. »Die Umsatzsteuer wird um zwei Prozentpunkte auf 18 % angehoben.« In dieser Drohung stecken zwei Zahlen, die wir trotz verschiedener Schreibweise gleich interpretieren. Deutlicher wird dies, wenn wir den Satz nicht lesen, sondern nur hören. Im Laufe unserer Ausbildung werden uns dann verschiedene Verhaltensweisen antrainiert, die uns den Unterschied zwischen Zahlen und Nichtzahlen verinnerlichen. Wir würden uns dann schon ein wenig wundern, dem Term $234 + 127$ in Form von zweihundertvierunddreißig plus einhundertsiebenundzwanzig zu begegnen.

In der EDV käme man mit den beiden Datentypen Zahlen und Nichtzahlen auch aus, weitere Untergliederung des Datentyps Nichtzahl hat aber erhebliche Vorteile im Bereich der Performance und auch des Speicherplatzes. Auch wir nutzen diesen Mechanismus, wenn wir große Zahlen in Zehnerpotenzen ausdrücken. Wenn wir die Bandbreite einer Farbe mit je 256 Rot-, Grün- und Blauanteilen berechnen, rechnen wir auch nicht $256 \times 256 \times 256$, sondern $2 (8 + 8 + 8)$, also $2 (24)$, und kommen auf rund 1 Million ($2 (20)$) mal 16 ($2 (4)$), und somit auf 16 Millionen.

Es folgt nun eine Tabelle mit den Datentypen in VBA. Zu dieser Übersicht möchte ich noch anmerken, dass die angegebenen Wertebereiche genau dem entsprechen, was ich mir auch gemerkt habe. Denn dass die Zahl 2 hoch 31 rund 2 Milliarden oder genau 2.147.483.648 ausmacht, ist Verschwendung geistiger Kapazitäten. Wenn Sie den Verdacht hegen, auch nur in die Nähe dieses Wertes zu geraten, *müssen* Sie einen anderen Datentyp verwenden.

Präfix	Datentyp	Wertebereich	Größe	Genauigkeit
byte	Byte	0 bis 255	1 Byte	
int	Integer	-2^{15} bis $2^{15}-1$	2 Byte	
lng	Long	-2^{31} bis $2^{31}-1$	4 Byte	
sng	Single	$\pm 10^{38}$ bis $\pm 10^{-45}$	4 Byte	8stellig
dbl	Double	$\pm 10^{308}$ bis $\pm 10^{-324}$	8 Byte	16stellig
cur	Currency	$\pm 9 \times 10^{14}$	8 Byte	19stellig
(date)	Date	01.01.0100 bis 31.12.9999	8 Byte	
b	Boolean	True und False	2 Byte	
str	String	0 bis 2^{31} Zeichen	1 Byte je Zeichen	
	Object	alle verfügbaren Objekttypen	4 Byte	
var	Variant	alle voranstehenden	mindestens 16 Byte	
dec	Decimal	$\pm 8 \times 10$ hoch 28	14 Byte	29stellig

Tabelle 4.1: Datentypen in VBA

Anmerkungen

Ein **Präfix** ist ein zumeist dreistelliger Bezeichner, der zur späteren Identifizierung des Datentyps im Code sehr nützlich ist, wie wir noch sehen werden. Hierzu gibt es eine unüberschaubare Menge von Vorschlägen. Ich

hätte mich auch gerne der Lesart des Hauses Microsoft angepasst, wenn ich wüsste, welcher.

Die angegebene **Genauigkeit** beinhaltet eine Stelle für das Dezimaltrennzeichen, der Datentyp `Double` hat dem gemäß als Gleitkommawert eine Stelle vor dem Komma, die Kommastelle und vierzehn Nachkommastellen.

4.1.1 Byte, Integer und Long

Microsoft schreibt im Visual Basic 6 Programmierhandbuch, Seite 792:

... Verwenden Sie Long-Variablen, wo immer Sie können, besonders in Schleifen. Der Datentyp Long ist der systemeigene Datentyp von 32-Bit-CPUs ...

Somit können wir uns viele Worte sparen und uns auf den `Long` beschränken, der sogar noch der schnellste der drei genannten sein soll.

Es gibt viele Einsatzgebiete für den `Long`-Datentyp. Zum einen kann er natürlich Werte dieses Datentyps aufnehmen, etwa eine Anzahl wiedergeben. Gerade in Excel-VBA dient der `Long`-Datentyp aber vielfach als Zeiger. Wenn Sie einen Zellbereich zeilenweise und/oder spaltenweise abarbeiten wollen, benötigen Sie einen solchen Zeiger:

```
Dim iRowDat as Long
Dim iColDat as Long
For iRowDat = 2 To ...
    For iColDat = T to ...
        ...
    Next
Next
```

Hier werden zwei Variablen mittels der `Dim`-Anweisung erzeugt, die als Zeilen- und Spaltenzeiger eines Tabellenblattes dienen. Innerhalb der beiden Schleifen dienen sie dann dem Zugriff auf die Zelle der durch die jeweilige Variable angegebenen Zeile oder Spalte.

4.1.2 Single und Double

Hier fällt die Wahl leicht, denn sieht man einmal von Anwendungsfällen im Bereich des Mikro- oder Makrokosmos mit sehr kleinen oder sehr großen Zahlen ab, so reicht der Wertebereich einer Variablen des Typs `Single` wohl in den meisten Fällen völlig aus.

Wertebereich

Das Kriterium ist hingegen in erster Linie die unterschiedliche Genauigkeit der beiden. Der Gleitkommawert `Single` verfügt lediglich über sechs Nachkommastellen, der `Double` jedoch über 14. Zudem ist der `Double`-Datentyp der Standarddatentyp einer Zelle numerischen Inhalts.

Genauigkeit

Eine einfache Rechenkette mit Steuersätzen lässt sich durchaus mit dem schnelleren, weil kleineren `Single`-Datentyp realisieren:

```
Dim sngUSt As Single
Dim sngBruttoUms As Single
Dim sngNettoUms As Single
...
sngNettoUms = sngBruttoUms / (1 + sngUSt)
```

Sind die Rechenketten hingegen länger, so reicht die Genauigkeit des `Single`-Datentyps möglicherweise nicht mehr aus, und Sie müssen auf den genaueren, aber langsameren `Double` zurückgreifen.

4.1.3 Currency

Der `Currency`-Datentyp ist mit 8 Byte genauso groß wie der `Double`, weist aber eine höhere Genauigkeit bei reduziertem Wertebereich auf. Intern wird er als 64-Bit-Ganzzahl gespeichert und durch 10.000 geteilt. Teilt man diesen Wert durch 2, so erhält man den vorzeichenbehafteten positiven und negativen Maximalwert von $9,22 \times 10^{14}$.

Das Einsatzgebiet liegt wohl bei Geldbeträgen in volkswirtschaftlichen Größenordnungen.

4.1.4 Date

Datumswerte in Excel werden als `Double`-Datentyp dargestellt, in dessen Vorkommawert das Datum und Nachkommawert die jeweilige Uhrzeit enthalten ist. Allerdings beschränkt sich Excel auf Datumswerte zwischen dem 01.01.1900 00:00 und dem 31.12.9999 23:59:59,999. Der Grund für die untere Grenze des Jahres 100 liegt in dem Umstand, dass VBA zweistellige Jahreszahlen dem aktuellen Jahrhundert anpasst. Aus dem 17.05.57 würde durch diesen Algorithmus in unseren Tagen zwangsläufig der 17.05.1957. Ein kleines Centennium-Problem, gewissermaßen, das uns aber sicherlich nicht stört.

Der Vorteil dieser internen Darstellung des `Date`-Datentyps als `Double` liegt in der einfachen Berechnung von Differenzen zweier solcher Werte. Die Zuweisung erfolgt über Zeichenketten, die allerdings ein gültiges Datum zum Inhalt haben müssen:

```
actDate = "1.2.1999"
actDate = "01.02.1999"
actDate = "1. Februar 1999"
actDate = "1. Feb 1999"
actDate = "1.2.1999 20:15"
actDate = "20:15"
```

Das Datum der letzten Zeile ist übrigens der 31.12.1899, da die fehlende Datumsangabe als Vorkommawert 0 interpretiert wird; Excel selbst macht daraus den 0.1.1900.

4.1.5 Boolean

Der Boolean ist ein logischer Datentyp in VBA, der uns die Werte `True` und `False` bereithält. Intern wird `True` in diesem eigentlichen Integer-Datentyp als `-1` und `False` als `0` dargestellt. Binär besteht `True` aus 16 Einsen und `False` eben aus 16 Nullen, weil das den einzigen Weg darstellt, zu den 16 Einsen eine 1 hinzuzuzählen und durch Wegfall der 17. Stelle 16 Nullen zu erhalten.

Speichern wir ein `True` (ohne Hinzunahme der API) in der Registry, so wird daraus »-1« und aus `False` »0«, wie wir noch sehen werden.

4.1.6 String

Dieser Datentyp kann alle 256 Zeichen des Windows zugrunde liegenden ANSI-Zeichensatzes aufnehmen, und zwar bis zu einer Gesamtlänge von 2 hoch 31 Zeichen. Da auch die Sonderzeichen 1 bis 31 (0 nur als letztes Zeichen) enthalten sein können, darunter auch Tabulatoren, Carriage Return und Line Feed, können Sie im Prinzip komplette Textdateien in einer `String`-Variablen ablegen. Wie Ihr Arbeitsspeicher allerdings mit einer 2 GByte großen Variablen zurechtkommt, können Sie ja mal ausprobieren.

Zumeist werden Sie kürzere Texte darin ablegen:

```
strWoTag = "Montag"  
strKunde = "Meier GmbH"
```

4.1.7 Object

Der Datentyp `Object` enthält einen 4 Byte großen Verweis auf ein im Speicher befindliches Objekt. Dem Grunde nach ist dieser Datentyp `Object` ein Meta-Datentyp, also ein Oberbegriff ohne ein real existierendes Element, ein Gattungsbegriff ähnlich dem Begriff Pflanze. Eine Rose ist eine Pflanze, eine mikroskopische Alge oder aber ein 6 Tonnen schwerer Riesenkaktus oder ein Mammutbaum. Aus diesen lassen sich verschiedene Merkmale des Meta-Begriffes Pflanze ableiten, aber umgekehrt lässt sich aus dem Meta-Begriff Pflanze nicht das Individuum ableiten.

VBA stellt eine Handvoll Objekte bereit, Excel rund 130, die alle verschiedene Strukturen zum Inhalt haben. Auf eine Reihe dieser Objekte, also Implementierung der Meta-Klasse Objekt, werden wir noch eingehen.

Eine Besonderheit sei hier erwähnt. Alle anderen Datentypen werden zugewiesen durch

```
Variable = Wert
```

Bei der Zuweisung von Objekten hingegen wird generell das Wort `Set` vorangestellt:

```
Set rngDat = shtDat.Range("A1")
```

4.1.8 Variant

Ein `Variant` kann in beliebigem Wechsel alle anderen Datentypen annehmen. Tolle Sache, möchte man meinen. Doch schauen wir uns kurz an, wie Ralf Morgenstern seinen Artikel »Kein einfacher Typ – inside Variants« einleitet, erschienen im VBA-Magazin 2/98, Computerfachverlag Cordula Lochmann:

Die `Variants` sind das Chamäleon unter den Datentypen. Je nach zugewiesenem Wert wandeln sie ihren Typ, ohne dass ein Übergang erkennbar wäre. Und so, wie man das Chamäleon häufig gar nicht oder erst spät erblickt, machen sich auch die Randerscheinungen des `Variants` oft erst spät bemerkbar.

Wo so viel Licht ist, muss auch viel Schatten sein.

Er verbraucht mehr Speicher als die zugrunde liegenden Originale und ist damit auch langsamer als diese. Der wichtigste Aspekt jedoch ist, dass er Fehler verschleppt und an Stellen auftauchen lässt, wo sie gar nicht verursacht werden:

```
1 Dim varWert As Variant
2 Dim varErgebnis As Variant
3 varWert = shtDat.Cells(iRowDat, iColDat).Value
  ...
35 varErgebnis = varWert * 2
```

In Zeile 3 wird der Variablen `varWert` der Inhalt der durch `iRowDat` und `iColDat` bestimmten Zelle der Tabelle `shtDat` zugewiesen. Steht dort kein numerischer Wert, so ficht das die `Variant`-Variable nicht an. Aber in Zeile 35 steht dann die Katastrophe ins Haus, sofern die Variable eben nicht numerisch ist.

Es gibt jedoch die eine oder andere Situation, in der ein `Variant` erforderlich ist, wie wir noch sehen werden.

Untertypen des Variant

Alle bisher besprochenen originären Datentypen können als Untertyp des `Variants` auftreten:

```
Dim x As Variant
x = 123.45                                'Untertyp Double
```

```
x = Now() 'Untertyp Date
x = "Mann, bin ich variant." 'Untertyp String
```

In Form des Untertyps `Decimal` kann der `Variant` aber einen Datentyp verwalten, der nicht als originärer Datentyp in Erscheinung treten kann; er existiert nur als Untertyp des `Variant`.

Für ganzzahlige Werte steht der $\pm 79.228.162.514.264.337.593.543.950.335$ zur Verfügung. Rationale Zahlen, also Dezimalzahlen, hingegen bewegen sich im Bereich von $\pm 7,9228162514264337593543950335$. Dazwischen arbeitet der `Decimal` im Wertebereich des `Double`:

```
Dim x As Variant
x = CDec(3.14/1717171717) 'ergibt
0,00000000182858823547698
```

Das Ergebnis dieser Division weist 25 Nachkommastellen auf. Die kleinste darstellbare Zahl > 0 ist 10^{-28} , also 0,00000000000000000000000001.

4.1.9 Datenfelder

Von den bisherigen Datentypen könnte man in Anlehnung an die Geometrie sagen, dass sie, einem Punkt vergleichbar, keine Ausdehnung besitzen. Nun bieten uns VBA die Möglichkeit, diesen gewissermaßen dimensionslosen Variablen eine oder mehrere Dimensionen hinzuzufügen.

Die nächste Dimensionsgröße nach dem Punkt ist eine eindimensionale Linie, danach folgt die zweidimensionale Fläche und die dreidimensionalen Körper, womit unsere Vorstellungskraft in der Regel ausgereizt ist. VBA geht da ein Stück weiter und erlaubt uns volle 60 Dimensionen. Doch zurück zu überschaubaren Größenordnungen.

Hinterlegen wir einen Monatsnamen in einer Variablen, so könnte das so aussehen:

```
Dim strMonat As String
strMonat = "Mai"
```

Wenn wir hingegen alle zwölf Monatsnamen benötigen, so macht das Ablegen in zwölf einzelnen Variablen keinen Sinn mehr. Hier bietet sich ein `String-Datenfeld (Array)` an:

```
Dim strMonat(12) As String
strMonat(1) = "Januar"
strMonat(2) = "Februar"
...
strMonat(12) = "Dezember"
```

Die erste Zeile erzeugt eine eindimensionale Variable mit zwölf Elementen in dieser ersten und einzigen Dimension. Diese Vorgehensweise ist ein wenig unprofessionell und lässt sich durch eine kleine Schleife mit der `Format()`-Funktion um einiges eleganter lösen (siehe For-Next-Schleifen).

Der Zugriff auf ein einzelnes Element dieses Datenfeldes erfolgt nun dadurch, dass im Klammersymbol des Variablennamens der gewünschte Index angegeben wird. Mit

```
strMonat(5) = "Mai"
```

wird dem 5. Element der Wert »Mai« zugewiesen und mit

```
... = strMonat(5)
```

wieder ausgelesen.

Das mag an dieser Stelle genügen. Im Kapitel »Sprachelemente, die zweite« werden wir uns gründlich mit Datenfeldern auseinandersetzen.

4.1.10 Sonderzustände von Datentypen

Obwohl es hier noch ein bisschen früh dafür ist, seien der Vollständigkeit halber noch ein paar Sonderzustände von Datentypen erwähnt, auf die an anderer Stelle noch detailliert eingegangen wird.

Empty

Empty tritt dann auf, wenn ein Variant-Datentyp nicht initialisiert ist. Das gilt nicht nur für Variablen des Datentyps `Variant`, sondern auch für das eine oder andere Objekt, wie zum Beispiel eine Zelle in einer Excel-Tabelle.

Nothing

Nicht initialisierte Objekte weisen den Zustand `Nothing` auf.

Null

Der Datentyp `Null` (hat nichts mit dem numerischen Wert 0 oder der als Nullstring gekannten leeren Zeichenkette »« zu tun) signalisiert einen ungültigen Inhalt eines Datentyps. Wir werden diesem unbeliebten Kandidaten vor allem im Zusammenhang mit Datenbanken begegnen.

4.2 Modulooptionen

Auf Modulebene lassen sich über die Option Anweisung insgesamt vier Einstellungen festlegen, die für das jeweilige Modulblatt Geltung haben. Die Anweisungen müssen vor der ersten Prozedur im Modulkopf stehen.

4.2.1 Option Base 0 | 1

Mit `Option Base` und der nachfolgenden Zahl 0 oder 1 legen Sie fest, welches die untere Grenze für Datenfelder in diesem Modulblatt sein soll. Der Standardwert ist 0. Somit ist die untere Grenze 0, wenn Sie `Option Base` nicht angeben.

4.2.2 Option Compare Binary | Text

Hiermit legen Sie die Regeln für Zeichenkettenvergleiche fest. Beide Varianten basieren auf der Code-Seite von Windows. Bei `Option Compare Binary` hält sich VBA strikt an den ANSI-Code der Zeichen, wonach »a« nach »Z« folgt. `Option Compare Text` hingegen stellt die Zeichen in Klein- und Großschreibung gleich, auch bei Umlauten und anderen Sonderzeichen europäischer Sprachen, bei denen je ein Zeichen für Klein- und Großschreibung existiert: é und É oder î oder Î.

Der Standardwert ist `Option Compare Binary`.

4.2.3 Option Explicit

Diese Anweisung bedeutet, dass in dem betreffenden Modulblatt alle Variablen explizit deklariert werden müssen, bevor sie verwendet werden. `Option Explicit` wird von der Entwicklungsumgebung selbständig in jedes Modulblatt eingefügt, wenn die Option »Variablendeklaration erforderlich« im Register Editor des Optionsdialogs aktiviert ist.

`Option Explicit` sollte in keinem Modul fehlen.

4.2.4 Option Private Module

`Option Private Module` begrenzt die Gültigkeit von öffentlichen (Public) Variablen, Objekten oder Prozeduren auf das aktuelle Projekt.

4.3 Prozeduren

Nach diesem kurzen Überblick über Datentypen und die Modulooptionen widmen wir uns den Strukturen, in denen VBA-Sprachelemente integriert werden.

Prozeduren sind gewissermaßen Container, in denen die eigentlichen Programmanweisungen Platz finden. Hiervon ausgenommen sind die `Option`-Anweisungen und einzelne Variablendeklarationen, die im Modulkopf außerhalb von Prozeduren stehen.

Sehen wir einmal von Property-Prozeduren ab, die uns noch im Rahmen der Behandlung von Klassen begegnen, existieren die beiden Typen Sub-Prozeduren und Function-Prozeduren. Der Unterschied ist ganz einfach:

▼ Eine `Function` kann einen Wert zurückgeben.

▼ Eine `Sub` kann keinen Wert zurückgeben.

Nehmen wir zur Verdeutlichung eine Anleihe bei Excel selbst. Hinter dem Menüpunkt Sortieren des Daten-Menüs verbirgt sich ein Programmteil, der den betreffenden Tabellenteil unter Berücksichtigung der dort vorgenommenen Einstellungen schlicht und ergreifend sortiert. Das Sortieren der Tabelle könnten wir über eine `Sub` realisieren. Anders wiederum das Suchen einer Zelle im Menü Bearbeiten. Hier wird die gefundene Zelle zurückgegeben (und markiert). Für das Zurückgeben der mit den Kriterien übereinstimmenden Zelle müssten wir eine `Function` zur Hand nehmen. Auch die Tabellenfunktionen, `SUMME()`, `MITTELWERT()` etc., sind als Funktionen realisiert, denn sie geben die jeweiligen Ergebnisse ihrer Berechnungen zurück.

4.3.1 Sub-Prozeduren

Sub-Prozeduren werden durch die Sub-Anweisung eingeleitet, die den Namen und eventuelle zusätzliche Argumente deklariert:

```
[Private | Public] [Static] Sub Name [(Argumente)]
... Code
End Sub
```

Abgeschlossen wird eine `Sub` immer durch `End Sub`. Doch nun zu den Ausdrücken in den eckigen Klammern.

Private, Public Durch `Private` bzw. `Public` definieren Sie den Gültigkeitsbereich der Prozedur. Ersteres bedeutet, dass die Prozedur nur innerhalb des sie beherbergenden Modulblatts angesprochen werden kann. Für andere Module oder gar Projekte ist diese Prozedur schlicht nicht existent. Zweiteres macht die Prozedur für andere Module hingegen sichtbar, was nichts anderes bedeutet, als dass diese darauf zugreifen können. Geben Sie keines von beiden an, so ist die Prozedur automatisch `Public`. Tun Sie sich und anderen aber den Gefallen, und schreiben Sie bei einer `Public`-Prozedur das Wort `Public` auch hin, denn darin steckt eine Information.

Static Das optionale Schlüsselwort `Static` deklariert automatisch alle lokalen Variablen der Prozedur zu statischen Variablen. Sie behalten nach Abarbeiten der Prozedur ihren Wert und erinnern sich beim nächsten Aufruf auch wunderbarerweise wieder daran. Eine nette Sache, die ich jedoch nicht verwende. Wenn eine oder mehrere Variablen einer Prozedur statisch sein sollen, so schreibe ich dieses Schlüsselwort `Static` vor die betreffende Variable. Im Kopf der Prozedur kann man das leicht übersehen, in der Zeile, in der die Variable hingegen erzeugt wird, kaum.

Eine Sub-Prozedur kann vorzeitig mit der Anweisung `Exit Sub` vorzeitig verlassen werden.

Einer Prozedur können **Argumente** mit auf den Weg gegeben werden, die in der Prozedur weiterverarbeitet werden. Da dieses Thema aber gleichermaßen für Prozeduren und Funktionen gilt, wird es in Kapitel Übergabe von Argumenten separat gewürdigt.

Argumente

4.3.2 Function-Prozeduren

Function-Prozeduren werden durch die Function-Anweisung eingeleitet, die den Namen, eventuelle zusätzliche Argumente sowie den Datentyp des Rückgabewertes der Funktion selbst deklariert:

```
[Private | Public] [Static] Function Name [(Arg.)) [As
Typ]
... Code
End Function
```

Eine Funktion wird durch `End Function` abgeschlossen. Die Schlüsselwörter `Private`, `Public` und `Static` sind bereits im voranstehenden Abschnitt behandelt worden. Durch `Exit Function` kann eine Function-Prozedur vorzeitig verlassen werden.

Anders hingegen `As Typ`, worunter sich die Angabe des Datentyps der Funktion selbst verbirgt. Soll eine Funktion zum Beispiel eine Variable des Typs `Date` in eine Form umwandeln, in der sie in einen SQL-String (Datenbankabfrage) integriert werden kann, so muss sie etwa wie `#12/31/1999#` aussehen. Dieser Ausdruck ist allerdings ein `String`, denn er genügt weder dem Datentyp `Date` noch sonst einem. Die Funktion, auf die wir im Kapitel Datenbanken noch zurückkommen, könnte etwa so aussehen:

Datentyp der Funktion

```
Private Function getSQLDate(actDate As Date) As String
    getSQLDate = "#" & Month(actDate) & "/" etc.
End Function
```

Der Name der Funktion `getSQLDate` wird also innerhalb der Funktion zu einer Variablen, und die sollte man bekanntermaßen explizit deklarieren, sonst werden sie automatisch zu einem `Variant`, die wir ja vermeiden wollen. Der Aufruf dieser Funktion könnte folgendermaßen aussehen:

```
strSQL = "... WHERE Datum = " & getSQLDate(dateRech) &
".."
```

Hier wird eine `String`-Variable des Namens `strSQL` aufgebaut, mit der später eine Datenbankoperation ausgeführt werden soll. Die Variable `dateRech`, Rechnungsdatum, wird aber in dieser Form von dem nachgeschalteten SQL-Interpreter nicht akzeptiert werden, der sie ja beispiels-

weise als #12/31/1999# erwartet. Deshalb wird in der Zeile, die den SQL-String zusammensetzt, die Funktion `getSQLDate` kurzerhand integriert. Gerät nun VBA an diese Zeile, so wird zuerst der rechts des Gleichheitszeichens stehende Teil abgearbeitet und die Funktion `getSQLDate` mit dem Argument `dateRech` aufgerufen. Die von dieser Funktion zurückgegebene Zeichenkette, die nun der korrekten Form entspricht, wird in die Zeichenkette integriert und am Ende der Variablen `strSQL` zugewiesen.

4.3.3 Übergabe von Argumenten

Dieser Funktion `getSQLDate` wurde also beim Aufruf eine Variable übergeben, die das in eine Zeichenkette zu konvertierende Rechnungsdatum enthielt. Sie tunnelt gewissermaßen von der aufrufenden zur aufgerufenen Sub oder Function, ohne dass sie außerhalb der beiden sichtbar wäre, hat also einen rein privaten Charakter.

Der Vorteil dieser Argumentübergabe liegt in der Einsparung von Variablen.

Übergabemodi

Es gibt zwei Arten der Übergabe:

▼ als Referenz (ByRef)

▼ als Wert (ByVal)

Übergabe als Wert

Bei der Übergabe als Wert (ByVal) wird eine Kopie der Variablen erstellt, die in der aufgerufenen Prozedur genauso viel Speicherplatz einnimmt wie die Originalvariable, also ab einem Byte aufwärts. In der Regel benötigt diese Variante jedoch mehr als die vergleichbaren vier Bytes für die Übergabe als Referenz.

```
Private Function getSQLDate(ByVal actDate As Date) As Str.
```

Übergabe als Referenz

Bei der Übergabe als Referenz (ByRef) wird ein Zeiger auf die Variable an die aufgerufene Prozedur übergeben wird. Im Adressraum der Prozedur fallen hierfür genau 4 Bytes (Long-Datentyp) für den Verweis auf die Originalvariable an. Die Übergabe als Referenz ist der Standardmodus und kommt somit auch dann zum tragen, wenn der Modus nicht angegeben wird:

```
Private Function getSQLDate(actDate As Date) As String
```

ist gleichbedeutend mit

```
Private Function getSQLDate(ByRef actDate As Date) As Str.
```

Da ja lediglich ein Zeiger übergeben wird – die Variable selbst also deklariert und somit auch als Typ bekannt ist, kann man sich sogar die Angabe des Datentyps sparen, wie folgendes Beispiel zeigt. Dort werden einem `Double` und einem `Variant` jeweils der Wert `PI` zugewiesen. Die aufgerufene Sub `Slave` nimmt die beiden Variablen als Referenz ohne Angaben des Typs entgegen:

```
Sub Master()
    Dim dbl As Double
    Dim var As Variant
    dbl = Application.Pi
    var = Application.Pi
    Slave dbl, var
End Sub

Sub Slave(dblLocal, varLocal)
    varLocal = "Hallo"
    dblLocal = "Hallo"
End Sub
```

Die Anweisung `dblLocal = "Hallo"` produziert den Fehler »Typen unverträglich«, womit bewiesen ist, dass `dblLocal` kein `Variant`, sondern ein `Double` ist. Bei einer sauberen Präfixbildung geht auch keine Information verloren.

Wann setzt man nun die beiden Varianten ein?

Wenn die aufgerufene Prozedur die Variable nicht verändern kann (und das haben wir ja im Griff), dann kann sie in der zumeist effizienteren Form als Referenz übergeben werden.

Die folgende Function-Prozedur berechnet die Fakultät der übergebenen Variablen `lngWert`. Hierbei verändert sie den Wert der Variablen und nimmt diesen `ByVal` entgegen.

```
Public Function Faculty (ByVal lngWert As Integer) As Long
    lngWert = lngWert - 1
    If lngWert = 0 Then
        Faculty = 1
        Exit Function
    End If
    Faculty = Faculty(lngWert) * (lngWert + 1)
End Function
```

Ändern Sie in dieser Function-Prozedur den Übergabemodus in `ByRef`, so werden Sie eine Überraschung erleben, denn das Ergebnis wird immer 1

Rekursion

sein. Der Grund liegt darin, dass in der letzten Zeile vor `End Function` die Function-Prozedur erneut eine Instanz von sich selbst aufruft. Diese sogenannte Rekursion erfolgt, bevor die Variable `Faculty` berechnet wird. Bei jedem Aufruf einer solchen rekursiven Function-Prozedur wird ein separater Adressraum (eine Instanz der Function) angelegt, in dem auch alle lokalen Variablen hinterlegt sind. Verweisen jedoch alle Instanzen über eine Referenz auf dieselbe Variable, die ständig dekrementiert wird, so muss das Ergebnis 1 sein, denn die letzte Berechnung erfolgt mit `lngWert = 0` und lautet als `Faculty = 0 * (0 + 1)`.



Nehmen Sie also in rekursiven Function-Prozeduren generell das veränderliche Argument als Wert (ByVal) entgegen!

Mit einem weiteren Kandidaten für eine Übergabe als Wert haben wir es dann zu tun, wenn Programmteile gekapselt werden sollen, was wir im Kapitel Klassen und DLLs diskutieren werden.

4.4 Verzweigungen

Als versierter Excel-Anwender sind Ihnen Verzweigungen längst geläufig, denn wenn Sie in einer Excel-Tabelle eine `WENN()`-Funktion oder eine `WAHL()`-Funktion einsetzen, schaffen Sie sich eine Verzweigung.

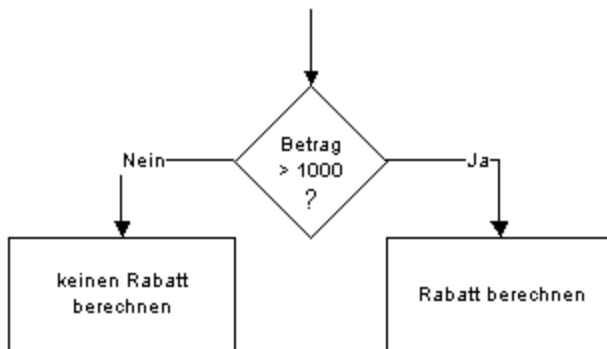


Abbildung 4.1: Schema einer Verzweigung

In VBA stehen uns die Verzweigungen in Form von

- ▼ If-Anweisungen *und*
- ▼ Select Case-Anweisungen

zur Verfügung. Es gibt zwar noch die `Choose`-Funktion, die `Switch`-Funktion und die `If()`-Funktion, denen jedoch im Prinzip keinerlei praktische Bedeutung zukommt.

4.4.1 Verzweigungen mit If-Anweisungen

Einfache If-Anweisungen

Eine einfache Verzweigung mit der If-Anweisung sieht so aus:

```
If Bedingung Then
```

```
...
```

```
End If
```

Als Bedingung kann jeder Ausdruck stehen, der im Ergebnis ein `True` oder `False` hervorbringen kann.

Zur Konstruktion einer solchen Bedingung können wir uns einer Reihe von Vergleichsoperatoren bedienen, die da sind:

Vergleichsoperatoren

Operator	Bedeutung
=	gleich
>	größer als
>=	größer oder gleich
<	kleiner als
<=	kleiner gleich
<>	ungleich

Tabelle 4.2: Vergleichsoperatoren

Im folgenden Beispiel wird beim Überschreiten eines Schwellwertes eine Rabattberechnung vorgenommen:

```
Sub Rabatt()
    Dim sngRabatt As Single      'Rabattbetrag
    Dim sngRechng As Single      'Rechnungsbetrag
    Dim lngRabSchw As Long       'Rabattschwellwert
    sngRechng = ...
    lngRabSchw = ...
    If sngRechng >= lngRabSchw Then
        sngRabatt = sngRechng * 0.1
    End If
    ...
End Sub
```

Ist der Rechnungsbetrag `sngRechng` größer oder gleich der Rabattschwellwert `lngRabSchw`, so beträgt der Rabattbetrag `sngRabatt` 10 % des Rechnungsbetrags. Interessant in diesem kleinen Beispiel ist übrigens noch, dass die Variable für den Schwellwert als `Long`-Datentyp gebildet wurde,

weil davon auszugehen ist, dass dieser Schwellwert als voller DM-Betrag ohne Nachkommastellen auskommen wird.

Diese einfache Verzweigung ist von dem Merkmal gekennzeichnet, dass keine weiteren `ElseIf`- oder `Else`-Komponenten folgen und dass die Anweisung aus einer einzelnen Zeile besteht. In diesem Falle lassen sich diese drei Zeilen noch weiter verkürzen:

```
If sngRechnung >= lngRabSchw Then sngRabatt = sngRechnung * 0.1
```

Auch wenn man dadurch zwei Zeilen einspart, so birgt diese Variante dennoch die Gefahr in sich, dass sie im Code leichter übersehen werden kann.

Noch ein paar Worte zu Bedingungen

Die bislang gezeigten Bedingungen bestanden aus zwei numerischen Werten und einem Vergleichsoperator. Dieser Ausdruck ergab einen logischen Wert, also `True` oder `False`. Nun interessiert es VBA nicht, ob ein solcher logischer Wert selbst wiederum aus einzelnen logischen Werten zusammengesetzt ist.

zusammengesetzte logische Ausdrücke

Die einzelnen Teilkomponenten eines zusammengesetzten Ausdrucks können mit folgenden logischen Operatoren verbunden werden:

Operator	Bedeutung	Syntax	Ergebnis True	Ergebnis False
And	Konjunktion	A1 And A2	A1 und A2 = True	A1 oder A2 = False
Eqv	Äquivalenz	A1 Eqv A2	A1 = A2	A1 ≠ A2
Imp	Implikation	A1 Imp A2	A1 = False oder A1 und A2 = True	A1 = True und A2 = False
Not	Negation	Not A1	A1 = False	A1 = True
Or	Disjunktion	A1 Or A2	A1 und/oder A2 = True	A1 und A2 = False
Xor	Exklusion	A1 Xor A2	A1 oder A2 = True	A1 = A2

Tabelle 4.3: Logische Operatoren

Im Grunde kommt man mit `And`, `Not` und `Or` bequem über den Winter, die drei anderen sind eher selten anzutreffen. Die Verknüpfungslogik dieser Operatoren unterliegt im übrigen den Regeln der Booleschen Algebra. Für unsere Erfordernisse in diesem Kapitel reicht ein kleiner Auszug aus dem Regelwerk:

▼ And hat Vorrang vor Or

▼ Not betrifft den unmittelbar folgenden Ausdruck

Sollen diese Regeln durchbrochen werden, so müssen Klammern gesetzt werden. Das ist ähnlich der Arithmetikregel, derzufolge Punktrechnung (* oder /) vor Strichrechnung (- oder +) geht. Bezüglich Rangordnung ist And also der Multiplikation vergleichbar und Or der Addition. Wollen Sie die Summe aus a und b mit c multiplizieren, müssen Sie auch Klammern setzen:

$(a + b) * c$

In unsere Boolesche Welt übertragen hieße das:

$(a \text{ Or } b) \text{ And } c$

Ist also mindestens einer der beiden Werte a oder b gleich True und c ebenfalls True, so ist der gesamte Ausdruck True. Ein Beispiel:

```
If sngRechnung >= 1000 And strMonat = "Mai" Then
    'Berechne Rabatt
End If
```

Rabatt gibt's also nur im Mai ab einem Rechnungsbetrag von 1000 DM. Hier nun bekommt ein guter Kunde generell Rabatt, auch außerhalb des Wonnemonats:

```
If sngRechnung >= 1000 And strMonat = "Mai" Or bTopKunde
Then
```

Komplexere If-Anweisungen

If-Verzweigungen können jedoch komplexer aufgebaut sein, als uns dieses kleine Beispiel suggeriert, wie das folgende Schema zeigt:

```
If Bedingung Then
    ...
ElseIf Bedingung-1 Then
    ...
ElseIf Bedingung-n Then
    ...
Else
    ...
End If
```

Im folgenden Beispiel ist eine Function-Prozedur zu bestaunen, die der Datei XLS-ToDB.xls entnommen ist. Diese Function nimmt einen Spaltentitel entgegen und konvertiert diesen in ein Format, der als Name eines

Datenbankfeldes dienen kann. `strOK` enthält das internationale Alphabet in Großbuchstaben sowie die zehn Ziffern und den Tiefstrich "_", `strSonderD` beinhaltet die Umlaute und das »ß«, `strSonderNeu` hält die Entsprechungen von `strSonderD` bereit.

Auch wenn in diesem Beispiel noch einige böhmische Dörfer enthalten sein sollten, so hat es dennoch den unbestreitbaren Vorteil, dass es aus dem Leben gegriffen ist. Die fett dargestellten Zeilen zeigen die If-Verzweigung.

```
Private Function CreateFieldName(strTitle) As String
    Dim i As Long                'Zeiger in strTitle
    Dim strChar As String        'aktuelles Zeichen
    Dim strOK As String          'erlaubte Zeichen
    Dim strSonderD As String     'deutsche Sonderzeichen
    Dim strSonderNeu As String   'Ersatz für Sonderzeichen
    strOK = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_"
    strSonderD = "äöüßÄÖÜ"
    strSonderNeu = "aeoeuessAeOeUe"
    For i = 1 To Len(strTitle)
        strChar = Mid(strTitle, i, 1)
        If InStr(1, strOK, UCase(strChar)) > 0 Then
            'erlaubte Zeichen
            CreateFieldName = CreateFieldName & strChar
        ElseIf InStr(1, strSonderD, strChar) > 0 Then
            'Umlaute oder "ß"
            CreateFieldName = CreateFieldName & _
                Mid(strSonderNeu, InStr(1, strSonderD, strChar) _
                    * 2 - 1, 2)
        Else
            'alle anderen Zeichen durch "_" ersetzen, sofern
            'vorangehendes Zeichen nicht bereits "_" ist.
            If Right(CreateFieldName, 1) <> "_" Then
                CreateFieldName = CreateFieldName & "_"
            End If
        End If
    Next
End Function
```

Im Else-Zweig ist übrigens eine weitere If-Verzweigung enthalten, die sicherstellt, dass keine zwei »_« hintereinander folgen.

4.4.2 Verzweigungen mit Select Case-Anweisungen

Der wesentliche Unterschied zwischen Verzweigungen mit If und Select Case liegt darin, dass die Bedingungen, die in den If- bzw. ElseIf-Zeilen erscheinen können, völlig unabhängig voneinander sind.

```

If strMonat = "Mai" Then
    sngRabatt = sngRechnung * 0.2
ElseIf sngRechnung >= 1000 Then
    sngRabatt = sngRechnung * 0.1
Else
    sngRabatt = sngRechnung * 0.05
End If

```

Im Mai beträgt der Rabatt 20 %, in den übrigen Monaten ab 1000 DM 10 % und in den anderen Fällen 5 %.

Mit einer Select Case-Anweisung wäre dieser (zugegebenermaßen) reichlich schwachsinnige Fall nicht durchführbar, da nur ein Wert überwacht werden kann. Um bei den Rabatten zu bleiben, könnten wir eine komplette Rabattstaffel gestalten:

```

Select Case sngRechnung
Case Is >= 10000      'alle übrigen Beträge ab 10.000 DM
    sngRabatt = 0.2
Case Is >= 5000      'Beträge ab 5.000 DM
    sngRabatt = 0.15
Case Is >= 2000      'Beträge ab 2.000 DM
    sngRabatt = 0.1
Case Else            'Beträge kleiner als 2.000 DM
    sngRabatt = 0.05
End Select

```

Muss eigentlich nicht mehr vertieft werden, oder?

Das Schlüsselwort Is wird von VBA übrigens selbständig ergänzt. Nun gibt es noch ein paar Besonderheit, die am besten in einem (ebenfalls reichlich konstruierten) Beispiel bewundert werden können:

```

Select Case strKunde
Case Is = "Meier"
    sngRabatt = 0.15
Case "Müller"
    sngRabatt = 0.15
Case "B" To "Bzz"
    sngRabatt = 0.1
Case "Schmid", Is = "Schmidt", "Schmitt"
    sngRabatt = 0.05
End Select

```

Kunde Meier erhält also ebenfalls wie Kunde Müller 15 % Rabatt. Sie sehen hier, dass das Schlüsselwort Is dann erforderlich wird, wenn ein Ver-

gleichsoperator auftaucht. Alle Kunden, deren Namen mit »B« beginnt, bekommen 10 % eingeräumt. Bzz steht für den alphabetisch letzt denkbaren mit »B« beginnenden Namen. Aber auch eine Liste mit Werten ist möglich, wie der letzte Fall zeigt. Die letzte Case-Anweisung hält schließlich eine Liste von Vergleichswerten bereit. Spaßeshalber wurde der Name Schmidt mit dem Operator »=« und dem nun obligatorischen Schlüsselwort Is versehen.

4.5 Schleifen

Schleifen dienen dazu, Teile einer Prozedur mehrfach zu durchlaufen. Unterschieden wird hierbei u. a. in unbedingte und bedingte Schleifen. Bei unbedingten steht bereits vor dem ersten Durchlauf fest, wie oft sie durchlaufen werden müssen. Bedingte Schleifen hingegen sind an eine logische Abbruchbedingung gebunden.

Zur Konstruktion von Schleifen stehen uns die Typen For-Next und Do-Loop zur Verfügung, die nachfolgend erläutert werden. Auf einen weiteren Vertreter des namens While-Wend wird nicht weiter eingegangen, da er funktional einen Teil der Do-Loop-Variante bietet und somit keine Vorteile aufweist, aber den Nachteil des eingeschränkten Funktionsumfangs mit sich bringt. Er ist schlichtweg überflüssig.

4.5.1 For-Next-Schleifen

For-Next-Schleifen, die dem Typus der unbedingten Schleifen angehören, sind folgendermaßen aufgebaut:

```
For Zähler = Anfang To Ende [Step Schrittweite]
    ...
Next
```

oder an einem Beispiel festgemacht:

```
For iMonat = 1 To 12 Step 1
```

Der Variablen iMonat (Zähler) wird beginnend mit 1 (Anfang) nacheinander ein um 1 (Schrittweite) größerer Wert zugewiesen, bis die obere Grenze von 12 (Ende) erreicht ist. Die übrigens nicht notwendigerweise ganzzahlige Schrittweite darf im Falle mit +1 weggelassen werden.

Sie erinnern sich noch an unser Datenfeld strMonate(12), was unter Datenfelder behandelt wurde. Dort versprach ich Ihnen eine elegantere Variante. Hierfür benötigen wir zwei Funktionen, die es auch als Zellfunktionen gibt: TEXT() und DATUM(). Der folgende Ausdruck

```
=TEXT(DATUM(1999;1;1);"MMMM")
```

liefert als Ergebnis »Januar«, weil die Funktion DATUM() die als Argumente übergebenen Zahlen in ein gültiges Datum (01.01.1999) umwandelt, was wiederum durch die TEXT()-Funktion als ausgeschriebener Monatsname zurückgegeben wird. In der folgenden For-Next-Schleife passiert dasselbe mit VBA-Funktionen:

```
Dim iMonat As Long           'Monatszähler
Dim strMonat(12) As String   'Datenfeld für die 12 Monate
For iMonat = 1 To 12
    strMonat(iMonat) = _
        Format(DateSerial(1999, iMonat, 1), "MMMM")
Next
```

Die DateSerial-Funktion erzeugt aus dem Jahr 1999, dem die Werte 1 bis 12 durchlaufenden Monatszähler und der Zahl 1 (wir hätten auch die 17 nehmen können) den 01.01.1999 als Datumswert. Dieser wiederum wird durch die Format-Funktion in den jeweils ausgeschriebenen Monatsnamen umgewandelt. Danach steht uns das mit den Monatsnamen gefüllten Datenfeld zur Verfügung.

Eine For-Next-Schleife kann mit Exit For vorzeitig verlassen werden.

Die For-Next-Schleife wird uns noch viele Male begegnen, wenn es darum geht, einen Tabellenbereich zeilen- und/oder spaltenweise abzuarbeiten.

Übriges, wie groß ist iMonat nach Beendigung der Schleife? Wenn Sie aufgrund der Fragestellung vermuten, dass iMonat nicht 12 ist, dann liegen Sie richtig. Wird im 12. Durchlauf die Next-Anweisung erreicht, so wird iMonat dennoch inkrementiert und erreicht somit den Wert 13. Danach stellt VBA jedoch fest, dass die obere Grenze von 12 überschritten ist, und das Programm wird mit der ersten Zeile nach Next fortgeführt.

4.5.2 Die For-Each-Next-Schleife

Diese Variante der For-Next-Schleife verwendet anstelle eines Zeigers innerhalb zweier Grenzen eine Auflistung oder ein Datenfeld. Die allgemeine Schreibweise lautet:

```
For Each Element In Gruppe
    ...
Next [Element]
```

Gruppe kann hierbei eine beliebige Auflistung, also eine Gruppe gleichartiger Objekte, sein. Dann übernimmt *Element* nacheinander die Rolle eines jeden Mitglieds der Auflistung.

Im folgenden Beispiel wird die Tabellenblattauflistung der Arbeitsmappe, in der sich der Code befindet, durchlaufen und die Inhalte aller Tabellen gelöscht:

```
Dim shtAct As Worksheet
For Each shtAct In ThisWorkbook.Worksheets
    shtAct.UsedRange.ClearContents
Next
```

In der zweiten Variante ist *Gruppe* ein Datenfeld beliebigen Typs und *Element* eine Variable des Typs Variant:

```
Dim varMonat As Variant
Dim strMonate(12) As String
strMonate(1) = "Januar"
strMonate(2) = "Februar"
...
strMonate(12) = "Dezember"
For Each varMonat In strMonate()
    strMessage = strMessage & varMonat & vbCRLF
Next
MsgBox strMessage
```

Auch hier übernimmt *varMonat* nacheinander die Rolle jedes Elements des Datenfelds *strMonate()*, ist beim ersten Durchlauf also »Januar« und beim letzten »Dezember«.

4.5.3 For-Next versus For-Each-Next

Es gibt keine zwingende Notwendigkeit, For-Each-Next-Schleifen einzusetzen, denn sie können ebenso durch For-Next-Schleifen nachgebildet werden. Das erste Beispiel funktioniert auch so:

```
Dim iSheet As Long
For iSheet = 1 to ThisWorkbook.Worksheets.Count

    ThisWorkbook.Worksheest(iSheet).UsedRange.ClearContents
Next
```

Es gibt dennoch Fälle, in denen der Einsatz der For-Each-Next-Schleifen sinnvoll ist, nämlich dann, wenn Sie innerhalb der Schleife mehrfach auf dasselbe Element zugreifen müssen. Den nicht gerade kurzen Ausdruck *ThisWorkbook.Worksheets(iSheet)* müsste man in diesem Fall für jeden Zugriff voranstellen. Hier würde sich dann gewissermaßen anbieten, eine Objektvariable für diesen Ausdruck zu verwenden, und genau dies tut die For-Each-Next-Schleife: sie stellt uns ohne separate Zuwei-

sung eine Objektvariable zur Verfügung. Neben einem übersichtlicheren Code bringt dies auch einen Geschwindigkeitsvorteil.

4.5.4 Die Do-Loop-Schleife

Bei einer Schleife dieser Machart steht in der Regel nicht von Anfang an fest, wie oft sie durchlaufen wird. Die folgende Schleife wird so oft durchlaufen, bis Sie den PC ausschalten:

```
Do
    Anweisungen
Loop
```

Da dies in der Regel wenig sinnvoll ist, gibt es mehrere Möglichkeiten, das Durchlaufen der Schleife abubrechen:

▼ Abbruchbedingung in Do- oder in Loop-Zeile

▼ Verlassen der Schleife per `Exit Do`

Zur Formulierung einer Abbruchbedingung stehen uns die beiden komplementären Begriffe `Until` (solange bis) und `While` (solange) zur Verfügung, die von einem logischen Ausdruck gefolgt sein müssen. Die folgende mit

```
Do Until rs.EOF
    Anweisungen
Loop
```

eingeleitete Schleife wird solange durchlaufen, bis End of File des Recordsets `rs` erreicht ist. Sie hätte funktional identisch auch so gestaltet sein können:

```
Do While Not rs.EOF
    Anweisungen
Loop
```

Die Abbruchbedingung kann auch in der Fußzeile untergebracht sein:

```
Do
    Anweisungen
Loop Until rs.EOF
```

bzw.

```
Do
    Anweisungen
Loop While Not rs.EOF
```

Im Unterschied zur Abbruchbedingung in der Kopfzeile wird die fußgesteuerte Schleife mindestens einmal durchlaufen. Es ergibt sich also aus der Programmlogik, ob Sie eine fußgesteuerte Schleife verwenden dürfen oder nicht.

Ebenso wie die For-Next-Schleife können Sie eine Do-Loop-Schleife jederzeit durch ein **Exit Do** vorzeitig verlassen. Hier ein Beispiel aus der Fehlerbehandlungsroutine Function-Prozedur CheckPath:

```
Do
    strMessage = "Bitte überprüfen Sie den Datenträger ..."
    lngReturn = MsgBox(strMessage, vbOKCancel)
    If lngReturn = vbOK Then
        Resume
    Else
        strMessage = "Der Datenträger ist nicht bereit."
        Exit Do
    End If
Loop
```

Solange die MessageBox mit der OK-Taste quittiert wird, wird mit Resume die Laufwerksüberprüfung fortgesetzt. Wird jedoch die Schaltfläche Abbrechen angeklickt, so wird die Schleife verlassen.

Sie sollten beim Einsatz der Do-Loop-Schleife ein gerüttelt Maß Sorgfalt bei der Formulierung der Abbruchbedingung walten lassen, denn eine Endlosschleife ist schnell geschrieben.

4.5.5 Verschachtelung von Schleifen

Beide Schleifentypen lassen sich ineinander verschachteln. So, wie dieser Begriff von dem Ineinanderstellen von Schachteln abgeleitet ist, muss sich eine innere Schleife auch vollständig innerhalb der äußeren befinden.

Wenn Sie zum Beispiel eine Tabelle spalten- und zeilenweise abarbeiten wollen, so bieten sich zwei verschachtelte For-Next-Schleifen an. Die erste und äußere Schleife bewegt sich beginnend mit Zeile 2 über alle Zeilen hinweg, wohingegen die zweite, innere beginnend mit 1 nacheinander alle Spalten umfasst:

```
For iRow = 2 To nRows
    [Anweisungen]
    For iCol = 1 To nCols
        Anweisungen
    Next iCol
```


[Anweisungen]

Next

Sie dürfen in der jeweilige Next-Zeile die dazugehörige Variable mit anführen, wie im obigen Beispiel in der inneren Schleife mit `Next iCol` gezeigt. Wenn Sie jedoch konsequent mit dem Tabulator arbeiten, ist durch die identische Spaltenposition auf einen Blick erkennbar, welches Next zu welchem For gehört.

Ein Ineinanderschachteln von Do-Loop-Schleifen folgt demselben Muster:

```
Do Until rs.EOF
  [Anweisungen]
  Do
    Anweisungen
  Loop Until strLine = ""
  [Anweisungen]
```

Loop

Alle bislang besprochenen Verzweigungs- und Schleifenstrukturen lassen sich natürlich mischen, wobei allerdings darauf zu achten ist, dass die einzelnen Strukturen auch tatsächlich **ineinander** geschachtelt werden. Folgendes ist nicht erlaubt:

```
If Bedingung Then
  Do
  Endif
Loop
```

Dieser Konstruktion kann man irgendwie bereits ansehen, dass sie nicht funktionieren kann, nicht wahr?

4.6 Vom Umgang mit Zeichenketten

VBA stellt eine recht ansehnliche Sammlung von Werkzeugen zur Manipulation von Zeichenketten zur Verfügung. Sie können beispielsweise aus einer Zeichenkette einen Teil ausschneiden, Teilzeichenketten zusammenfügen, umwandeln, Sie können ...

Obwohl Speicherplatz heutzutage nahezu keine Rolle mehr spielt, kann es in umfangreicheren VBA-Anwendungen zu dem sehr unangenehmen Fehler »Nicht genügend Speicher« (Fehler 7) kommen. Verursacht wird er in einem 64 KB großen Speichersegment, das wohl gelegentlich an seine Grenzen geraten kann. Irgendwie weckt die Zahl 64 KB, also 2^{16} , Erinnerungen an längst vergangen geglaubte Zeiten. Wie dem auch sei ...

\$-Zeichen in Funktionsnamen

Der Ausdruck »Hallo« als `String` benötigt 5 Bytes, wohingegen derselbe Ausdruck als `Variant` 21 Bytes benötigt, denn ein `Variant` besteht aus 16 Bytes plus einem Byte je Zeichen. Viele der im folgenden gezeigten Funktionen geben einen `Variant`-Datentyp des Untertyps `String` zurück, verfügen jedoch auch über eine Variante, die einen reinen `String`-Datentyp zurückgeben. Hierbei wird generell zwischen Funktionsnamen und der ersten Klammer ein `$`-Zeichen eingefügt.

`Left(strWert, 1)` erzeugt einen `Variant` mit dem Inhalt "A"
`Left$(strWert, 1)` erzeugt einen `String` mit dem Inhalt "A"

Wenn eine Funktion in der `String`-Variante existiert, so ist das `$`-Zeichen in der Funktionsübersicht enthalten.

Um ein durchgängiges Beispiel für die nächsten Funktionen zur Verfügung zu haben, konstruieren wir eine Artikelnummer, die aus vier Abschnitten besteht. Hierbei steht A für eine alphabetische Stelle und N für eine numerische:

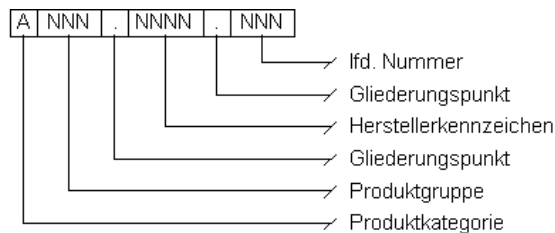


Abbildung 4.2: Artikelnummernstruktur für Beispiele

Beispiel: B100.1234.001, nachfolgend als `strArtNr` referenziert.

4.6.1 Teilzeichenketten

Die häufigsten Operationen betreffen wohl Teilzeichenketten: die linken drei Zeichen, die rechten fünf oder zwei irgendwo aus der Mitte.

Left\$(String, nZeichen)

String	zu bearbeitende Zeichenkette
nZeichen	Anzahl der Zeichen, die von links entnommen werden sollen

Die `Left`-Funktion gibt eine Zeichenkette zurück, die von links `nZeichen` der übergebenen Zeichenkette `String` umfasst.

Beispiele:

Left(strArtNr, 1) ergibt "B"
 Left(strArtNr, 4) ergibt "B100"

Right\$(String, nZeichen)

String	zu bearbeitende Zeichenkette
nZeichen	Anzahl der Zeichen, die von rechts entnommen werden sollen

Die Right-Funktion gibt eine Zeichenkette zurück, die von rechts nZeichen der übergebenen Zeichenkette String umfasst.

Beispiele:

Right(strArtNr, 3) ergibt "001"
 Right(strArtNr, 8) ergibt "1234.001"

Mid\$(String, erstesZeichen, nZeichen)

String	zu bearbeitende Zeichenkette
erstesZeichen	Position des Zeichens, ab dem entnommen werden soll
nZeichen	Anzahl der Zeichen, die entnommen werden sollen

Die Mid-Funktion gibt eine Zeichenkette zurück, die beginnend mit erstesZeichen nZeichen der übergebenen Zeichenkette umfasst.

Beispiele:

Mid(strArtNr, 2, 3) ergibt "100"
 Mid(strArtNr, 6, 4) ergibt "1234"

Das Argument erstesZeichen kann aber auch 1 sein. Dann verhält sich die Mid-Funktion wie die Left-Funktion:

Mid(strArtNr, 1, 4) ist identisch mit Left(strArtNr, 4).

Siehe hierzu auch das Beispiel zur Len-Funktion weiter unten.

4.6.2 Informationen über Zeichenketten

Len(String)

String	zu überprüfende Zeichenkette

Die Len-Funktion gibt die Anzahl der Zeichen zurück, die in String enthalten sind:

Len(strArtNr) ergibt 13



Wenn Sie mit VBA 6 (Office 2000) arbeiten, bitte ich Sie, das nächste Beispiel geflissentlich zu übersehen. Denn Ihnen steht durch die weiter unten vorgestellte InStrRev-Funktion eine weitaus elegantere Vorgehensweise zu Verfügung.

Die folgende Funktion ExtractFile ermittelt den Dateinamen, der in dem übergebenen Ausdruck enthalten ist:

```
Public Function ExtractFile(ByVal strFile As String) _
    As String
    Dim i As Integer    'Zeiger in strFile
    For i = Len(strFile) To 1 Step -1
        If Mid(strFile, i, 1) = "\" Then Exit For
    Next
    ExtractFile = Right(strFile, Len(strFile) - i)
End Function
```

In der For-Next-Schleife wird ein *i* gebildet, welches sich beginnend mit der Länge der Zeichenkette *strFile* dekrementiv bis 1 bewegt. Ist *strFile* = »C:\Test\Test.xls«, so bewegt sich *i* ganzzahlig von 16 bis 1. Dann wird geprüft, ob das *i*-te Zeichen von *strFile* = »\« ist und im Falle der Übereinstimmung die Schleife verlassen. In unserem Beispiel wäre *i* = 8, denn an der achten Stelle steht der erste Backslash von rechts. Abschließend wird dem Funktionsnamen *ExtractFile* die Teilzeichenkette zugewiesen, die von rechts aus gesehen die Differenz der Länge von *strFile* minus der Position des gefundenen Backslash beträgt: 16 – 8 ergibt 8, und die rechten 8 Zeichen lauten »Test.xls«.

Diese überaus nützliche Prozedur, die sich in *modUtil* befindet, hat in Form der *ExtractPath* ein Geschwister, welches eben den linken Teil von *strFile* bis zum letzten Backslash zurückgibt – den Pfad also. Die For-Next-Schleifen beider Prozeduren sind identisch, der Unterschied liegt in der letzten Zeile. Zum direkten Vergleich sind hier die betreffenden Zeilen aus beiden Prozeduren aufgeführt:

```
ExtractFile = Right(strFile, Len(strFile) - i)
ExtractPath = Left(strFile, i - 1)
```

InStr(Start, String1, String2, Vergleich)

<i>Start</i>	(optional) die Position des Zeichens, ab der <i>String1</i> untersucht werden soll, fehlt dieses Argument, wird die Suche beim ersten Zeichen begonnen
<i>String1</i>	zu durchsuchende Zeichenkette
<i>String2</i>	gesuchte Zeichenkette
<i>Vergleich</i>	(optional) Art des Vergleichs als Konstante: vbBinaryCompare (0) unterscheidet zwischen Groß- und Kleinschreibung vbTextCompare (1) unterscheidet nicht zwischen Groß- und Kleinschreibung vbDatabaseCompare (2) nur in Access relevant Wird Vergleich nicht angegeben, so wird die Einstellung von Option Compare übernommen.

Die InStr-Funktion gibt die Position innerhalb von String1 als Long-Wert zurück, an der String2 gefunden wurde. Ist String2 nicht in String1 enthalten, so gibt sie 0 zurück.

Beispiele:

```
InStr(1, "B100", "1") ergibt 2
InStr(1, "B100", "B") ergibt 1
InStr(1, "B100", "b") ergibt 0, weil gleichbedeutend mit
InStr(1, "B100", "b", vbBinaryCompare)
InStr(1, "B100", "b", vbTextCompare) ergibt 1
```

Ist jedoch Option Compare Text im Modulkopf definiert, so folgt daraus

```
InStr(1, "B100", "b") ergibt 1, weil gleichbedeutend mit
InStr(1, "B100", "b", vbTextCompare)
```

In dem folgenden Beispiel transformiert die Prozedur SQL_Decimal einen Double-Wert, der möglicherweise ein Komma als Dezimaltrennzeichen beinhaltet, zu einem SQL-verträglichen Format mit einem Dezimalpunkt: also aus »3,14« wird »3.14«.

```
Public Function SQL_Decimal(ByVal dblWert As Double) _
    As String
    SQL_Decimal = CStr(dblWert)
    If InStr(1, SQL_Decimal, ",") > 0 Then
        Mid(SQL_Decimal, InStr(1, SQL_Decimal, ","), 1) =
        "."
    End If
End Function
```



InStrRev(*String1*, *String2*, *Start*, *Vergleich*)

<i>String1</i>	zu durchsuchende Zeichenkette
<i>String2</i>	gesuchte Zeichenkette
<i>Start</i>	(optional) fehlt dieses Argument, wird -1 angenommen und die Suche beginnt beim letzten Zeichen
<i>Vergleich</i>	(optional) Art des Vergleichs als Konstante: vbBinaryCompare (0) unterscheidet zwischen Groß- und Kleinschreibung vbTextCompare (1) unterscheidet nicht zwischen Groß- und Kleinschreibung vbDatabaseCompare (2) nur in Access relevant Wird Vergleich nicht angegeben, so wird die Einstellung von Option Compare übernommen.

Bitte achten Sie darauf, dass die Reihenfolge der Argumente dieser Funktion von der Reihenfolge der InStrRev-Funktion abweicht, unnötiger Weise, möchte ich meinen.

Im folgenden Beispiel werden aus dem vollständigen Dateinamen strFullName der Pfad strPath und der Dateiname strFile extrahiert und in einer MessageBox ausgegeben:

```
Dim strFullName As String
Dim strFile As String
Dim strPath As String
strFullName = "C:\Test\Test.xls"
strFile = Right(strFullName, Len(strFullName) - _
    InStrRev(strFullName, "\"))
strPath = Left(strFullName, InStrRev(strFullName, "\") -
    1)
MsgBox "Pfad: " & vbTab & strPath & vbCrLf & _
    "Datei: " & vbTab & strFile
```

StrComp(*String1*, *String2*, *Vergleich*)

<i>String1</i>	erste Zeichenkette
<i>String2</i>	zweite Zeichenkette
<i>Vergleich</i>	(optional) Art des Vergleichs als Konstante: vbBinaryCompare (0) unterscheidet zwischen Groß- und Kleinschreibung vbTextCompare (1) unterscheidet nicht zwischen Groß- und Kleinschreibung vbDatabaseCompare (2) nur in Access relevant Wird Vergleich nicht angegeben, so wird die Einstellung von Option Compare übernommen.

StrComp signalisiert durch den Rückgabewert, ob *String1* gleich *String2* (Rückgabe 0) oder *String1* alphabetisch vor (Rückgabe -1) oder nach (Rückgabe 1) *String2* einzuordnen ist.

Diese Funktion hat in Excel-VBA fast keine praktische Bedeutung, da uns andere Möglichkeiten zum Sortieren von Zeichenketten zur Verfügung stehen.

4.6.3 Split und Join

Diese beiden Funktionen sind neu in VBA 6 und ersparen so manchen Umweg, der in der Vergangenheit notwendig war.



Split(*String*, *Trennzeichen*, *Anzahl*, *Vergleich*)

<i>String</i>	zu bearbeitende Zeichenkette
<i>Trennzeichen</i>	(optional) das Trennzeichen, anhand dessen die Aufteilung von <i>String</i> vorgenommen werden soll
<i>Anzahl</i>	(optional) die Anzahl der zu erzeugenden Elemente: -1 bedeutet »soviel wie erforderlich« 0 geht, macht aber keinen Sinn > 0 erzeugt die angegebene Anzahl von Elementen, aber (natürlich) höchstens so viele wie erforderlich
<i>Vergleich</i>	(optional) das Vergleichsmuster: vbUseCompareOption folgt der Einstellung von Option Compare vbBinaryCompare führt einen binären Vergleich durch, der zwischen Groß- und Kleinschreibung unterscheidet. vbTextCompare führt einen Textvergleich durch, der nicht zwischen Groß- und Kleinschreibung unterscheidet.

Die Split-Funktion erzeugt ein Datenfeld des Typs String, das die unter *Anzahl* festgelegte Anzahl von Elementen beinhaltet. Ein jedes Element enthält eine Teilzeichenkette, die zwischen Anfang und dem ersten Trennzeichen, zwischen zwei aufeinanderfolgenden Trennzeichen oder dem letzten Trennzeichen und dem Ende der Zeichenkette liegt.

Ein Beispiel:

```
Dim strTeil() As String
strTeil() = Split(strArtNr, ".", -1, vbTextCompare)
```

Danach enthält das Datenfeld strTeil() die folgenden Elemente:

```
strArtNr = " B100.1234.001" (zur Erinnerung)
strTeil(0) enthält "B100"
strTeil(1) enthält "1234"
strTeil(2) enthält "001"
```

Zu beachten ist, dass das erzeugte Datenfeld immer mit `Feld(0)` beginnt, unabhängig davon, was bei `Option Base` angegeben wird.

```
strTeil() = Split(strArtNr, ".", 2, vbTextCompare)
```

ergibt übrigens

```
strTeil(0) enthält "B100"
```

```
strTeil(1) enthält "1234.001"
```

Join(Datenfeld, Trennzeichen)

<i>Datenfeld</i>	enthält die zusammenzufügenden Teilzeichenketten
<i>Trennzeichen</i>	(optional) das Trennzeichen, das zwischen die Teilzeichenketten des <i>Datenfelds</i> eingefügt werden soll

Die ebenfalls neue Funktion `Split()` fügt ein Datenfeld wieder zu einer Zeichenkette zusammen, also quasi das Gegenteil von `Split`.

Nehmen wir an, Sie kreieren Artikelnummern algorithmisch aus Datenfeldern, deren Elemente einzeln hinzugefügt werden:

```
Dim strArtNr As String
Dim strArtNrC(3) As String
strArtNrC(1) = "B100"
strArtNrC(2) = "1234"
strArtNrC(3) = "001"
strArtNr = Join(strArtNrC(), ".") 'ergibt "B100.1234.001"
```

Anders als die `Split`-Funktion hält sich `Join` an die Vorgabe der `Option Base`-Anweisung.

4.6.4 Zeichenketten verändern

Beginnen wir gleich mit einem alten Bekannten, der `Mid`-Funktion.

Mid(String, erstesZeichen, nZeichen) = neuerString

<i>String</i>	zu bearbeitende Zeichenkette
<i>erstesZeichen</i>	Position des Zeichens, ab dem entnommen werden soll
<i>nZeichen</i>	Anzahl der Zeichen, die entnommen werden sollen
<i>neuerString</i>	an der angegebenen Stelle einzufügende Zeichenkette

Die `Mid`-Funktion ersetzt in *String* die durch *erstesZeichen* und *nZeichen* spezifizierte Teilzeichenkette durch *neuerString*.

Nach `Mid(strArtNr, 6, 4) = "1235"` wird unsere Variable `strArtNr` zu `B100.1235.001`. Die `Left`- und die `Right`-Funktionen vermögen dies übrigens nicht, was uns aber nicht weiter stören soll, denn die `Mid`-Funktion deckt die beiden mit ab.

Chr(ANSI-Code)

ANSI-Code

ANSI-Code des Zeichens

Die `Chr`-Funktion gibt das dem *ANSI-Code* entsprechende Zeichen zurück.

Sie fragen sich vielleicht, wozu diese Funktion denn gut sei. Man kann das betreffende Zeichen direkt angeben, also statt `Chr(65)` gleich »A« hinschreiben. Für die meisten Fälle trifft dies auch zu, aber es gibt Ausnahmen.

Sie kennen sicherlich die benutzerdefinierten Zahlenformate, die Sie im Menü `Format Zellen` einstellen können. Möchten Sie z. B., dass Zahlen in der Einheit »Stck« in der Zeile erscheinen, so geben Sie folgendes Format ein:

```
0 "Stck"
```

Die doppelten Anführungsstriche sind allerdings obligatorisch, müssen also mit übergeben werden. Nun ist aber das ganze Zahlenformat in VBA eine Zeichenkette, die mit dem Zeichen `"` eingeleitet und abgeschlossen werden. Die folgenden Formatierungen produzieren einen Fehler:

```
"0 "Stck"
```

```
"0 "Stck""
```

Wir müssen also die Anführungsstriche, die den Ausdruck `Stck` einkleiden, vor VBA verstecken. Und genau dafür bietet sich die `Chr`-Funktion an:

```
Range.NumberFormat = "0 " & Chr(34) & "Stck" & Chr(34)
```

Um etwa einen Tabulator, ein CarriageReturn `CR` und/oder ein LineFeed zu erzeugen, benötigen wir die `Chr`-Funktion nicht, denn dafür steht uns eine Reihe von Konstanten wie `vbTab`, `vbCR`, `vbLF`, `vbCRLF` und einige mehr zur Verfügung. Den Umgang damit werden wir in diesem Kapitel bei der Durchsprache der `MsgBox`-Funktion noch kennenlernen.

LCase(String)

String

umzuwandelnde Zeichenkette

LCase (Lower Case) wandelt *String* in Kleinbuchstaben um.

LCase("Hallo") ergibt "hallo"

UCase(*String*)

<i>String</i>	umzuwandelnde Zeichenkette

UCase (Upper Case) wandelt *String* in Großbuchstaben um.

UCase("Hallo") ergibt "HALLO"

Nicht konvertierbare Zeichen werden von beiden Funktionen ignoriert.

Space\$(*n*)

<i>n</i>	Anzahl der zu erzeugenden Blanks

Space erzeugt eine Zeichenkette, die aus *n* Blanks besteht.

Nehmen wir an, Sie müssen eine Exportdatei z.B. für SAP erstellen. SAP unterstützt ein Textformat mit konstanten Zeilenlängen:

B100.1234.001Laserdrucker	1859.00
B100.1235.023Tintenstrahldrucker	499.00

Wir müssen also die Variable *strName*, die den Begriff »Laserdrucker« enthält, auf insgesamt 40 Zeichen erweitert, hinter die Artikelnummer in eine Zelle schreiben:

```
rng.Cells(iRow, 1).value = strArtNr & _
    strName & Space$(40 - Len(strName))
```

Das Hinzufügen des konvertierten Preises schenken wir uns hier, denn das Prinzip ist das Gleiche. Im übrigen werden diese Techniken in Kapitel 6 ausführlich behandelt.

String\$(*n*, *Zeichen*)

<i>n</i>	Anzahl der Wiederholungen von <i>Zeichen</i>
<i>Zeichen</i>	zu erzeugendes Zeichen

String erzeugt eine Zeichenkette, die *n* mal *Zeichen* enthält:

```
String$(5, "#") ergibt "#####"
```

```
String$(3, "ABC") ergibt "AAA"
```

Trim\$(String)*String*

umzuwandelnde Zeichenkette

Die Trim-Funktion entfernt führende und nachfolgende Blanks der Zeichenkette *String*.

Trim\$(" Hallo ") ergibt "Hallo"

LTrim\$(String)*String*

umzuwandelnde Zeichenkette

LTrim entfernt führende Blanks der Zeichenkette *String*:

LTrim\$(" Hallo ") ergibt "Hallo "

Rtrim\$(String)*String*

umzuwandelnde Zeichenkette

RTrim entfernt nachfolgende Blanks der Zeichenkette *String*:

RTrim\$(" Hallo ") ergibt " Hallo"

Blanks, die zwischen anderen Zeichen angesiedelt sind, bleiben bei allen Trim-Funktionen unangetastet:

Trim\$("Hallo, ich bin's") bleibt "Hallo, ich bin's"

CStr(Ausdruck)*Ausdruck*

umzuwandelnder Wert

CStr wandelt einen konvertierbaren (!) Ausdruck in eine Zeichenkette um. Hierunter fallen die Datentypen Boolean, Date, Empty, Error sowie alle numerischen Datentypen. Die restlichen Datentypen haben einen Fehler zur Folge.

4.6.5 Zeichenketten zusammenfügen

Implizit ging es bereits aus einigen der voranstehenden Beispielen hervor: Zeichenketten können mittels des &-Operators zu einer größeren Zeichenkette zusammengefügt werden.

```
strGesamt = strTeil1 & strTeil2 & ... & strTeiln
```

Alternativ kann auch das Pluszeichen + verwendet werden. Verwenden Sie konsequent das Pluszeichen für arithmetische Operationen und das &-Zeichen für Zeichenketten, so ist Ihr Code zweifellos einfacher zu interpretieren.

4.7 Formatierungen

Wenn Sie sich in Excel das Register Zahlen im Dialog Format Zellen ansehen, so sehen Sie eine Vielfalt an Formatierungsmöglichkeiten für Zellen. Darauf werden wir im nächsten Kapitel noch zu sprechen kommen.

Eine Reihe dieser Formatierungsmöglichkeiten stecken auch so in der Format-Funktion, die in Visual Basic entstanden ist. Aber es handelt sich um zwei verschiedene Entwicklungen, die zwar Gemeinsamkeiten aufweisen, aber auch Unterschiede.

Grundsätzlich verarbeitet die Format-Funktion Zeichenketten, numerische Datentypen und Datumswerte, wobei das Ergebnis einer solchen Formatierung ein Variant Untertyp String oder eine Zeichenkette sein kann – je nachdem, ob Sie Format() oder Format\$() verwenden.

4.7.1 Formatierung von Zeichenketten

Format\$(String, Format)

<i>String</i>	zu formatierende Zeichenkette
<i>Format</i>	Formatausdruck

Folgende Zeichen können im Formatausdruck Verwendung finden:

@	Dieser Platzhalter für ein Zeichen zeigt ein vorhandenes Zeichen an. Ein nicht vorhandenes Zeichen wird durch ein Leerzeichen dargestellt. Die Platzhalter werden von rechts nach links aus dem darzustellenden Ausdruck aufgefüllt.
&	Dieser Platzhalter für ein Zeichen zeigt ein vorhandenes Zeichen ebenfalls an. Ein nicht vorhandenes Zeichen wird allerdings auch nicht dargestellt. Die Platzhalter werden von rechts nach links aus dem darzustellenden Ausdruck aufgefüllt.
>	Alle Zeichen werden in Großbuchstaben umgewandelt.
<	Alle Zeichen werden in Kleinbuchstaben umgewandelt.
!	Keht die Auffüllrichtung für die Platzhalter @ und & um, so dass diese nunmehr von links nach rechts aufgefüllt werden.

Beispiele:

```
Format$("abcd", "@@.@")   ergibt "ab.cd"
Format$("abcde", "@@.@")  ergibt "abc.de"
```

Letzteres ist logisch, da die Platzhalter von rechts aufgefüllt werden. Im nächsten Beispiel werden sie bedingt durch das !-Zeichen von links aufgefüllt:

```
Format$("abcde", "!@@.@@") ergibt "ab.cde"
```

In den nächsten Beispielen werden weniger Zeichen übergeben, als Platzhalter verfügbar sind:

```
Format$("abc", "@@.@@") ergibt " b.cd"
```

```
Format$("abc", "!@@.@@") ergibt "ab.c "
```

Nun wird der Platzhalter @ gegen & ausgetauscht:

```
Format$("abc", "&&.&&") ergibt "b.cd"
```

```
Format$("abc", ">!&&.&&") ergibt "AB.C"
```

Im letzten Beispiel wurde das >-Zeichen ergänzt, was zur Umwandlung der Zeichen in Großbuchstaben bewirkte. Es ist im übrigen scheinbar völlig einerlei, an welcher Stelle dieser Operator steht.

Im Beispiel zur Space-Funktion ergänzten wir die Variable strName durch Leerzeichen auf eine konstante Länge von 30 Zeichen:

```
strName & Space$(40 - Len(strName))
```

Zum selben Ergebnis kämen wir u. a. auch durch folgende Konstruktion:

```
Format$(strName, "!" & String$(30, "@"))
```

Damit die ursprüngliche Zeichenkette auch links beginnt und rechts von Leerzeichen begrenzt wird (und nicht umgekehrt), wird das !-Zeichen mit der Rückgabe der String-Funktion verkettet.

4.7.2 Formatierung von Zahlen

Wobei die Formatierung von Zeichenketten eher die Ausnahme darstellt, sind Zahlenformatierungen schon häufiger anzutreffen. Da uns in Excel jedoch ausgefeilte Formatierungen in Zellen zur Verfügung stehen, wird die Format-Funktion seltener zusammen mit Zellen eingesetzt. Nicht zuletzt auch deshalb, weil die Format-Funktion eine Zeichenkette produziert, wohingegen eine formatierte Zelle numerischen Inhalts nach wie vor eine Zahl ausweist, was gerade in einer Tabellenkalkulation ein unbestreitbarer Vorteil ist.

Format\$(Zahl, Format)

<i>Zahl</i>	zu formatierende Zahl
<i>Format</i>	Formatausdruck

Benannte Formate

Bei der Formatierung von Zahlen können wir auf eine überschaubare Zahl von benannten Formaten zurückgreifen, die nebst ihrem jeweiligen Ergebnis in der folgenden Übersicht aufgeführt sind:

```
Format$(123456.7, "General Number") wird zu 123456,7
Format$(123456.7, "Currency")       wird zu 12.3456,70 DM
Format$(123456.7, "Fixed")           wird zu 123456,70
Format$(123456.7, "Standard")        wird zu 123.456,70
Format$(123456.7, "Percent")         wird zu 12345670,00%
Format$(123456.7, "Scientific")      wird zu 1,23E+05
```

Die nächsten drei Kandidaten prüfen nur, ob es sich um eine Zahl handelt.

```
Format$(123456.7, "Yes/No")         wird zu Ja
Format$(123456.7, "True/False")     wird zu Wahr
Format$(123456.7, "On/Off")         wird zu Ein
```

Es handelt sich bei diesen benannten Formaten übrigens nicht um benannte Argumente, die in VB(A) üblicherweise flächendeckend anzutreffen sind. Benannte Argumente sind im Gegensatz zu diesen benannten Formaten Long-Konstanten.



Noch einmal zur Verdeutlichung: Die Ergebnisse der oberen sechs Operationen sind Zeichenketten!

Benutzerdefinierte Formate

Schauen wir uns zuerst die Elemente der benutzerdefinierten Zahlenformate an. Einige Elemente, die in der Online-Hilfe aufgeführt sind, fehlen in dieser Liste. Die Ergebnisse des Einsatzes dieser Platzhalter sind allerdings durch den Einsatz der Chr-Funktion zu erreichen.

0	Dieser Platzhalter für eine Ziffer zeigt die betreffende Stelle unabhängig davon an, ob die Stelle zur Wiedergabe des Wertes erforderlich ist oder nicht. Sind hingegen mehr Stellen besetzt als mit diesem Platzhalter versehen, so werden die Stellen vor dem Komma natürlich angezeigt. Im Nachkommabereich wird jedoch auf die letzte Stelle gerundet.
#	Dieser Platzhalter zeigt die betreffende Stelle nur dann an, wenn die Stelle auch zur korrekten Wiedergabe des Wertes erforderlich ist. Klar? In Wirklichkeit dient dieser Platzhalter nur zur Positionierung eines Tausendertrennzeichens.

.	Der Punkt ist der Platzhalter für das Dezimaltrennzeichen. Welches Zeichen nun tatsächlich als Dezimaltrennzeichen ausgegeben wird, hängt von den Ländereinstellungen ab. Bei uns ist es in der Regel das Komma.
,	Das Komma stellt das Tausendertrennzeichen dar. Bezüglich des tatsächlich dargestellten Zeichens gilt obiges.
%	Das %-Zeichen führt zur Multiplikation des Wertes mit 100. Das %-Zeichen selbst wird an der Stelle dargestellt, an der es auch im Format steht.
E+ e+ E- e-	Diese Platzhalter erzwingen eine Darstellung der Zahl in Exponentialschreibweise. Die Unterschiede dieser vier Varianten sind in Beispielen besser darzustellen als in unleserlichen Sätzen. Deshalb bitte ich um einen Moment Geduld.

Beispiele:

`Format$(12.3, "000.000")` ergibt `"012,300"`

Der Platzhalter »0« erzwingt die Darstellung der betreffenden Stelle, auch wenn diese zur korrekten Wiedergabe der Zahl nicht erforderlich wäre. So zum Beispiel die Positionen 10^2 , 10^{-2} und 10^{-3} .

Ersetzen wir nun die Null durch die Raute, so werden keine zusätzliche Stellen angezeigt:

`Format$(12.3, "###.###")` ergibt `"12,3"`

Lediglich der Punkt führte zur Darstellung des Dezimaltrennzeichens. Sinnvoll werden die Rauten allerdings, wenn Sie mit dem Tausendertrennzeichen kombiniert werden. Das folgende Format kennen Sie bereits aus Excel:

`Format$(1234.5, "#.##0.00")` ergibt `"1.233,40"`

Dieses Resultat erzielen wir auch mit dem benannten Format »Standard«. Um das benannte Format »Percent« nachzubilden, benötigen wir folgendes Format:

`Format$(123456.7, "0.00%")` ergibt `"12345670,00%"`

Beim Einsatz der Exponentialschreibweise von Zahlen wird »E« als »E« und »e« als »e« angezeigt. Das nachgestellte Plus- oder Minuszeichen hingegen beeinflusst nur die Darstellung positiver Exponenten; negative Exponenten werden (natürlich) generell mit einem Minuszeichen vor dem Exponenten versehen:

`Format$(1234, "0.00E+0")` ergibt `"1,23E+3"`

`Format$(1234, "0.00E-0")` ergibt `"1,23E3"`

```
Format$(0.00003,"0.00E+0")      ergibt "3,00E-5"
Format$(0.00003,"0.00E-0")      ergibt "3,00E-5"

Format$(0.00003,"0.00e-0")      ergibt "3,00e-5"
```

Literale Werfen wir zum Abschluss noch einen Blick auf die Darstellung sog. Literale. Ist einem Zeichen keine symbolische Bedeutung zugeordnet, so spricht man von einem Literal. Insbesondere die Formatierung von Datumswerten kennt viele Symbole, wie wir noch sehen werden. Aber auch wir begegneten bereits den Zeichen @, &, !, 0 oder #.

Möchten Sie ein Zahl mit Literalen garnieren, so empfiehlt es sich generell, diese in doppelte Anführungszeichen einzuschließen, um einem eventuellen Symbolkonflikt aus dem Wege zu gehen. Und wie man so was macht, haben wir schon bei der Durchsprache der **Chr-Funktion** gesehen:

```
Format$(1, Chr$(34) & "etwa " & Chr$(34) & "0" & Chr$(34)
-
      & "%" & Chr$(34))
```

ergibt "etwa 1%".

Würde das %-Zeichen nicht durch die beiden Chr\$(34) eingeschlossen, so wäre es kein Literal, sondern ein Steuerzeichen, das zur Multiplikation des Wertes mit 100 führte:

```
Format$(1, Chr$(34) & "etwa " & Chr$(34) & "0%")
```

würde zu "etwa 100%".

4.7.3 Formatierung von Datums- und Zeitwerten

Die Format-Funktion bietet eine Fülle von Formatierungsmöglichkeiten für Datums- und Zeitwerte. Vieles ist bereits von den Zahlenformaten der Zellen her bekannt. Darüber hinaus kann die Format-Funktion aber auch die Kalenderwoche eines Datums ermitteln. Die einschlägige Fachpresse und das Internet sind voll von mehr oder weniger eleganten Algorithmen zur Ermittlung der Kalenderwoche mittels VB oder VBA. Die Format-Funktion erspart uns jedoch all diese geistigen Kimmzüge, wie wir noch sehen werden.

Format\$(Datum, Format, firstdayofweek, firstweekofyear)

<i>Datum</i>	zu formatierendes Datum
<i>Format</i>	Formatausdruck

<i>firstdayofweek</i>	(optional) Konstante für den Tag, mit dem die Woche beginnt
<i>firstweekofyear</i>	(optional) Konstante zur Steuerung der ersten Woche des Jahres

Für *firstdayofweek* stehen die folgenden Konstanten bereit, wobei NLS für National Language Support steht:

<i>vbUseSystem</i>	Einstellung der NLS API
<i>vbSunday</i>	Woche beginnt mit Sonntag (Voreinstellung)
<i>vbMonday</i>	Woche beginnt mit Montag

Die Liste geht noch weiter bis *vbSaturday*, doch das können wir uns schenken.

Hier nun die Konstanten für *firstweekofyear*:

<i>vbUseSystem</i>	Einstellung der NLS API
<i>vbFirstJan1</i>	1. Januar liegt in der 1. Woche (Voreinstellung)
<i>vbFirstFourDays</i>	1. Woche mit mindestens vier Tagen ist Woche 1
<i>vbFirstFullWeek</i>	1. vollständige Woche ist Woche 1

Nach DIN 28601 liegt der erste Donnerstag des Jahres in der ersten Woche. Oder anders ausgedrückt, die erste Woche enthält mindestens vier Januartage. Da die Woche sieben Tage hat, überwiegen also in der ersten Woche die Januartage gegenüber den Dezembertagen. Somit ergibt sich auch automatisch der Montag als erster Wochentag. Für uns bedeutet das, dass wir im Argument *firstdayofweek* die Konstante *vbMonday* einsetzen müssen und als *firstweekofyear* die Konstante *vbFirstFourDays*, sofern wir auf die Kalenderwoche eines Datums werts abzielen. In allen anderen Fällen lassen wir die Argumente einfach weg.

Erste Kalenderwoche

Benannte Formate

Auch für die Formatierung von Datums- und Zeitwerten verfügt die *Format*-Funktion über eine Reihe von benannten Argumenten, wie die folgende Übersicht zeigt:

```
Format$(Now(),"General Date") ergibt "17.05.99 15:01:38"
Format$(Now(),"Long Date")   ergibt "Montag,
                               17. Mai 1999"
Format$(Now(),"Medium Date") ergibt "17. Mai. 99"
Format$(Now(),"Short Date")  ergibt "17.05.99"
Format$(Now(),"Long Time")   ergibt "15:01:38"
Format$(Now(),"Medium Time") ergibt "03:01 PM"
Format$(Now(),"Short Time")  ergibt "15:01"
```

Ländereinstellungen
der Systemsteuerung

Die Erscheinungsform der Formate "Long Date", "Short Date" und "Long Time" sind in den Ländereinstellungen der Systemsteuerung einstellbar. "General Date" wiederum setzt sich aus "Short Date" und "Long Time" zusammen. Die Platzhalter in den Ländereinstellungen sind identisch mit denen der Zahlenformate in Excel, z. B. »TT.MM.JJJJ«.

AM/PM Kennung

Die AM/PM Kennung des Formats "Medium Time" wird nur dann zurückgegeben, wenn in den Ländereinstellungen der Systemsteuerung die Symbole für Vormittag und Nachmittag definiert sind. Diese auf acht Stellen begrenzten Texte könnten somit auch »vorm.« und »nachm.« lauten. Fehlen diese Texte, erscheint logischerweise lediglich ein verträumtes Leerzeichen hinter der letzten Minutenstelle.

Benutzerdefinierte Formate

Die nachstehende Tabelle enthält alle für Datums- und Zeitformate definierten Platzhalter. Bitte führen Sie sich vor Augen, dass sämtliche Rückgaben der Format-Funktion Zeichenketten sind, auch wenn in der folgenden Übersicht die Rede ist von einer Zahl: z. B. »5«. Diese lassen sich allerdings über entsprechende Konvertierungsfunktionen, z. B. CLng, in echte Zahlen umwandeln.

/	Datumstrennzeichen, trennt Tag-, Monat- und Jahresabschnitt
.	funktioniert auch als Datumstrennzeichen
:	Zeittrennzeichen, trennt Stunden-, Minuten- und Sekundenabschnitt
c	Datum und Uhrzeit gemäß "General Date" (siehe »Benannte Formate«)
d	Tag ohne führende Null: 1 bis 31
dd	Tag mit führender Null: 01 bis 31
ddd	zweistellige Abkürzung des Wochentags: Mo bis So
dddd	Wochentag: Montag bis Sonntag
dddddd	Datumsanzeige gemäß "Short Date", das den Einstellungen für Kurzes Datum der Ländereinstellungen der Systemsteuerung folgt.
ddddddd	Datumsanzeige gemäß "Long Date", das den Einstellungen für Langes Datum der Ländereinstellungen der Systemsteuerung folgt.
h	Stunde ohne führende Null: 0 bis 23
hh	Stunde mit führender Null: 00 bis 23
m	Monat ohne führende Null: 1 bis 12 Steht m hinter h ; so steht m für Minute (siehe n)
mm	Monat mit führender Null: 01 bis 12 Steht mm hinter h ; so steht mm für Minute (siehe nn)
mmm	dreistellige Abkürzung des Monats: Jan bis Dez
mmmm	Monatsname: Januar bis Dezember

n	Minute ohne führende Null: 0 bis 59
nn	Minute mit führender Null: 00 bis 59
q	Quartal: 1 bis 4
s	Sekunde ohne führende Null: 0 bis 59
ss	Sekunde mit führender Null: 00 bis 59
tttt	Uhrzeit gemäß "Long Time" (siehe »Benannte Formate«)
w	Wochentag als Zahl (1 bis 7). Anders als in der Online-Hilfe dargestellt, folgt es dabei dem Argument <code>firstdayofweek</code> . Ist dies <code>vbMonday</code> , so ergibt der Freitag eine 5. Fehlt es hingegen, so wird für Freitag eine 6 ausgegeben, da die Woche dann mit Sonntag beginnend gerechnet wird.
ww	Kalenderwoche als Zahl. Ohne Angabe des Arguments <code>firstweek-of-year</code> lautet die Rückgabe für den 17.05.1999 fälschlicherweise 21. Erst durch die Konstante <code>vbFirstFourDays</code> wird die korrekte Zahl 20 zurückgegeben: <code>Format\$(Now(), "ww", vbMonday, vbFirstFourDays)</code> Achtung: Wenn Sie <code>vbMonday</code> nicht angeben, erhalten Sie alle bis 1992 und ab 2004 alle 16 Jahre falsche Wochennummern. Die Unstetigkeitsstelle im Intervall zwischen 1992 und 2004 ergibt sich durch die Ausnahmebehandlung des Jahres 2000 als Schaltjahr.
y	Kalendertag als Zahl: 1 bis 366
yy	zweistellige Jahreszahl: 00 bis 99
yyyy	vollständige, nicht notwendigerweise vierstellige Jahreszahl
AM/PM	erzeugt den Zusatz AM oder PM
Am/pm	erzeugt den Zusatz am oder pm
A/P	erzeugt den Zusatz A oder P
A/p	erzeugt den Zusatz a oder p
AMPM	erzeugt den Zusatz entsprechend den Ländereinstellungen der Systemsteuerung (siehe AM/PM Kennung weiter oben)

Auch Datums- und Zeitformate lassen sich um Literale erweitern. Wie bereits bei den benutzerdefinierten Zahlenformaten beschrieben, müssen wir hierzu diesen Literal in `Chr(34)` einkleiden:

```
Format$(Now(), "dddd" & Chr(34) & " Morgen" & Chr(34))
```

Der vorstehende Ausdruck ergibt »Freitag Morgen«.

Im übrigen wird hier auf weitere Beispiel verzichtet, denn die Formulierung benutzerdefinierter Formate, die über die oben gezeigten Möglichkeiten hinausgehen, sind wohl kaum erforderlich.

4.8 Numerische Funktionen

Alle nachfolgend vorgestellten Funktionen folgen dem Muster

```
Ergebnis = Funktion(Wert)
```

Hierbei kann Wert eine beliebige Zahl oder eine in eine Zahl konvertierbare Zeichenkette sein. Zum Prüfen, ob ein Wert numerisch oder in eine Zahl konvertierbar ist, steht uns die Funktion `IsNumeric()` zur Verfügung:

`IsNumeric(Wert)`

Wert	zu überprüfender Wert

Die `IsNumeric`-Funktion gibt ein `True` zurück, wenn es sich bei Wert um eine Zahl oder um einen in eine Zahl konvertierbaren Wert handelt:

```
IsNumeric(3.14)    ergibt True
```

```
IsNumeric("3,14")  ergibt True
```

4.8.1 Konvertierungsfunktionen mit ganzzahligem Ergebnis

`CByte(Wert)`

Wert	zu konvertierender Wert

`CByte()` gibt einen ganzzahligen Wert vom Typ `Byte` innerhalb dessen Wertebereichs zurück. Ist das Argument eine rationale Zahl, so rundet `CByte()`.

`CInt(Wert)`

Wert	zu konvertierender Wert

`CInt()` gibt einen ganzzahligen Wert vom Typ `Integer` zurück. Ist das Argument eine rationale Zahl, so rundet `CInt()`.

`CLng(Wert)`

Wert	zu konvertierender Wert

`CLng()` gibt einen ganzzahligen Wert vom Typ `Long` zurück. Ist das Argument eine rationale Zahl, so rundet `CLng()`.

Int(Wert)

Wert	zu konvertierender Wert

`Int()` gibt den ganzzahligen Anteil von *Wert* zurück, wobei der Datentyp von *Wert* beibehalten wird. Bei negativen Werten gibt `Int()` den nächsten ganzzahligen Wert zurück, der kleiner oder gleich *Wert* ist.

Fix(Wert)

Wert	zu konvertierender Wert

`Fix()` gibt den ganzzahligen Anteil von *Wert* zurück, wobei der Datentyp von *Wert* beibehalten wird. Bei negativen Werten gibt `Fix()` den nächsten ganzzahligen Wert zurück, der größer oder gleich *Wert* ist.

Val(Wert)

Wert	zu konvertierender Wert

`Val()` extrahiert aus *Wert* von links beginnend eine Zahl des Datentyps `Double`. Führende Leerzeichen, Tabulatoren oder Returns werden ignoriert. Kann keine Zahl extrahiert werden, so gibt `Val` 0 zurück.

Beispiel:

```
Val(" 123 DM")    ergibt 123
Val(" 123.45 DM") ergibt 123,45
Val("Hallo")      ergibt 0
```

Eine Besonderheit der `Val()`-Funktion soll hier nicht unerwähnt bleiben. Gewissermaßen als Pendant zur `Hex()`-Funktion ermöglicht sie die Umwandlung von maximal 8-stelligen Hexadezimalzahlen in Dezimalzahlen:

```
Val("&HA")          ergibt 10
Val("&H12345678")    ergibt 305419896
Val("&H123456789")   erzeugt einen Überlauf-Fehler
```

Die folgende Tabelle gibt die Rückgabe und den Datentyp der in diesem Abschnitt behandelten Funktion für die Zahlen 3,14, -3,14 und 3,77 wieder:

	3,14	Typ	-3,14	Typ	-3,77	Typ
<code>CByte()</code>	3	Byte	Fehler 6	N/a	Fehler 6	n/a
<code>CInt()</code>	3	Integer	-3	Integer	-4	Integer

	3,14	Typ	-3,14	Typ	-3,77	Typ
CLng()	3	Long	-3	Long	-4	Long
Int()	3	Double	-4	Double	-4	Double
Fix()	3	Double	-3	Double	-3	Double
Val()	3	Double	-3	Double	-3	Double

4.8.2 Konvertierungsfunktionen mit reellem Ergebnis

CCur(Wert)

Wert	zu konvertierender Wert

CCur() konvertiert Wert in den Datentyp Currency. Werte kleiner 10^{-4} werden auf der Stelle 10^{-4} gerundet.

CDbl(Wert)

Wert	zu konvertierender Wert

CDbl() konvertiert Wert in den Datentyp Double.

CDec(Wert)

Wert	zu konvertierender Wert

CDec() konvertiert Wert in den Untertyp Decimal des Datentyps Variant.

CSng(Wert)

Wert	zu konvertierender Wert

CSng() konvertiert Wert in den Datentyp Single.

4.9 Datumsfunktionen

4.9.1 Standardfunktionen

Die folgenden zehn Datumsfunktionen kennen Sie vermutlich bereits als Tabellenfunktionen aus Excel. Zur Orientierung sind die Tabellenfunktionen hinter dem Funktionsnamen aufgeführt.

Cdate(Wert) =DATWERT()

Wert	zu konvertierender Wert

Wert ist hierbei ein beliebiger, in einen Datumswert konvertierbarer Ausdruck, zum Beispiel »17.5.1999«, »17. Mai 1999« oder 36297.

Date() **=HEUTE()**

Date() liefert das Tagesdatum als Datumswert.

DateSerial(Jahr, Monat, Tag) **=DATUM()**

<i>Jahr</i>	das Jahr als Zahl
<i>Monat</i>	der Monat als Zahl
<i>Tag</i>	der Tag als Zahl

DateSerial() erzeugt aus den drei Komponenten einen entsprechenden Datumswert:

DateSerial(1999, 5, 17) ergibt 17.05.1999 als Datumswert

Bei der ersten Durchsprache der Datenfelder erzeugten wir strMonat durch die etwas umständliche und unschöne Zuweisung strMonat(1) = "Januar". Eleganter geht es mit der DateSerial()-Funktion:

```
Dim i As Long
Dim strMonat(12) As String
For i = 1 To 12
    strMonat(i) = Format(DateSerial(1999, i, 1), "mmm")
Next
```

Es ist im übrigen völlig egal, was Sie hierbei an Jahres- und Tageszahl übergeben. Wichtig ist der Monat i.

Now() **=JETZT()**

Now() liefert das Tagesdatum inklusive der aktuellen Uhrzeit als Datumswert.

Year(Wert) **=JAHR()**

<i>Wert</i>	ein gültiger Datumswert

Year() liefert den Jahresanteil des Datumswerts als Zahl.

Month(Wert) **=MONAT()**

<i>Wert</i>	ein gültiger Datumswert

Month() liefert den Monatsanteil des Datumswerts als Zahl.

Zeitraum In der Praxis kommt es häufig vor, dass man aus einem Datumswert einen Zeitraum bestimmen muss, um zum Beispiel monatsweise irgendwelche Werte zu aggregieren. Für solche Fälle verwende ich einen Zeitraum nach dem Muster JJJJMM, also 199905 für Mai 1999. Wie erzeugt man nun einen solchen Wert?

Die Variable lngZeitraum (so nenne ich die immer) wird aus dem angenommenen Rechnungsdatum dateRech mittels der beiden Funktionen Year() und Month() ermittelt:

$\text{lngZeitraum} = \text{Year}(\text{dateRech}) * 100 + \text{Month}(\text{dateRech})$

In Tabellen erreichen Sie dies durch $\text{=JAHR(B2)*100+MONAT(B2)}$, wobei angenommener Weise die Zelle B2 das Datum enthält.

Day(Wert) =TAG()

Wert	ein gültiger Datumswert

Day() liefert den Tagesanteil des Datumswerts als Zahl.

Hour(Wert) =Stunde()

Wert	ein gültiger Datumswert

Hour() gibt die Anzahl der angebrochenen Stunden des Zeitraumausdrucks zurück.

Minute(Wert) =MINUTE()

Wert	ein gültiger Datumswert

Minute() gibt die Anzahl der angebrochenen Minuten der aktuellen Stunde zurück. Der Wert bewegt sich zwischen 0 und 59.

Second(Wert) =SEKUNDE()

Wert	ein gültiger Datumswert

Second() gibt die Anzahl der angebrochenen Sekunden der aktuellen Minute zurück. Der Wert bewegt sich zwischen 0 und 59.

4.9.2 Verzichtbare Datumsfunktionen

Dieses Buch ist weder eine vollständige Referenz von VBA noch eine des Excel-Objektmodells. Beides finden Sie in der Hilfe, die nach wie vor die erste Informationsquelle bezüglich dieser Themen darstellt.

Dennoch möchte ich ein paar Funktionen nicht unerwähnt lassen, die Ihnen vielleicht in der Hilfe begegnen. Hierunter fallen die Funktionen `WeekDay()`, `DateAdd()`, `DateDiff()` und `DatePart()`. Diese Funktionen sind keineswegs überflüssig, sondern lediglich verzichtbar. `WeekDay()` und `DatePart()` lassen sich durch die `Format()`-Funktion ersetzen, `DateAdd()` und `DateDiff()` durch direkte Datumsberechnungen.

Natürlich ist es von Vorteil, wenn Sie diese Funktionen beherrschen. Aber es war schon immer besser, wenige grundlegende und leistungsfähige Funktionen gründlich zu kennen, als eine wesentlich größere Funktionsmenge mit hoher Spezialisierung auswendig zu lernen, und darauf läuft es hinaus.

4.10 Ein- und Ausgabefunktionen

Die Mensch-Maschine-Schnittstelle kann in Excel-VBA auf vielerlei Arten realisiert werden. Zum einen stehen uns natürlich die Tabellenzellen selbst zur Verfügung. Wem das zu profan ist, der kann komplexe Dialoge gestalten, die dann so aussehen wie der Optionen-Dialog in Excel. Natürlich geht das auch ein paar Nummern kleiner.

Aber es gibt auch Situationen, in denen Sie nur eine kurze Entscheidung vom Anwender benötigen. Etwa, ob die Tabelle neu aufgebaut werden soll. Oder aber Sie benötigen einen Dateinamen, unter dem eine Datei gespeichert werden soll. Hier stehen uns in Form der `MsgBox()`- und `InputBox()`-Funktion zwei leistungsfähige Kandidaten zur Verfügung.

4.10.1 MsgBox-Funktion

Wenn Sie eine umfangreiche Berechnung durchgeführt haben, so macht es durchaus Sinn, den Anwender darüber in Kenntnis zu setzen, dass das Werk vollbracht ist. Dies könnte zu Beispiel folgende `MessageBox` für Sie tun:

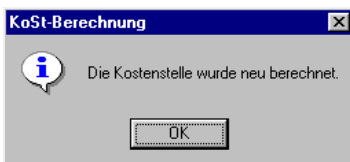


Abbildung 4.3: Beispiel einer Information Ihres Anwenders

Oder aber Sie möchten vom Anwender wissen, ob die Kostenstelle neu berechnet werden soll:

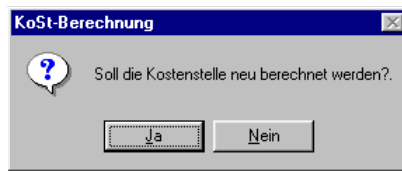


Abbildung 4.4: Beispiel einer Entscheidung, die der Anwender treffen muss

Solch einfache Benutzerschnittstellen lassen sich mittels der `MsgBox()`-Funktion realisieren, die folgendermaßen aufgebaut ist:

`MsgBox(prompt, buttons, title, helpFile, context)`

<i>prompt</i>	Zeichenkette, die den anzuzeigenden Text enthält. In unserem ersten Beispiel die Information: Die Kostenstelle wurde neu berechnet.
<i>buttons</i>	(optional) Ein numerischer Ausdruck, der Informationen über die darzustellenden Schaltflächen, deren Default-Schaltfläche, den Typ des Symbols sowie die Art der Bindung des Dialogs enthält. Wird dieses Argument nicht angegeben, so wird 0 angenommen. Die Konstanten dieses Arguments werden weiter unten erläutert.
<i>title</i>	(optional) Zeichenkette, die im Dialogtitel angezeigt wird. Wenn dieses Argument weggelassen wird, so erscheint der Text »Microsoft Excel« im Dialogtitel.
<i>helpfile</i>	(optional) Der Name der Hilfedatei. Wenn dieses Argument angegeben wird, so ist auch das nachfolgende Argument <i>context</i> anzugeben.
<i>context</i>	(optional) Die Help-Context-ID, unter der die Information in der Hilfe abgelegt ist.

Nun zu den Konstanten, die für das Argument *buttons* definiert sind. In der Beschreibung des Arguments ist zu lesen, dass 0 angenommen wird, wenn das Argument fehlt. Wie so oft ist hier 0 ungleich nichts. Die folgende Übersicht enthält alle Konstanten mit dem Wert 0:

Konstante	Wert	Beschreibung
<code>vbOKOnly</code>	0	Nur die Schaltfläche OK wird angezeigt.
<code>vbDefaultButton1</code>	0	Die erste Schaltfläche ist Default-Schaltfläche (und kann somit mit der Return-Taste angesprochen werden).
<code>vbApplicationModal</code>	0	Die Applikation (Excel) kann erst fortgeführt werden, wenn die MessageBox geschlossen wird.

Alle diese Einstellungen machen sich in der MessageBox bemerkbar, wenn kein Argument angegeben wurde oder der Wert des Arguments gleich 0 ist.

Die folgenden Konstanten betreffen die anzuzeigenden Schaltflächen, wobei aus jeder Tabelle nur ein Wert angegeben werden darf:

Konstante	Wert	Beschreibung
vbOKCancel	1	Die Schaltflächen OK und Abbrechen werden angezeigt.
vbAbortRetryIgnore	2	Die Schaltflächen Abbrechen, Wiederholen und Ignorieren werden angezeigt.
vbYesNoCancel	3	Die Schaltflächen Ja, Nein und Abbrechen werden angezeigt.
vbYesNo	4	Die Schaltflächen Ja und Nein werden angezeigt.
vbRetryCancel	5	Die Schaltflächen Ignorieren und Abbrechen werden angezeigt.

Mit den folgenden Konstanten wird festgelegt, welche Schaltfläche zur Default-Schaltfläche wird und sich somit durch die Return-Taste angesprochen fühlt:

Konstante	Wert	Beschreibung
vbDefaultButton2	256	Die zweite Schaltfläche ist Default-Schaltfläche.
vbDefaultButton3	512	Die dritte Schaltfläche ist Default-Schaltfläche.
vbDefaultButton4	768	Die vierte Schaltfläche ist Default-Schaltfläche.

`vbDefaultButton4` macht nur dann Sinn, wenn Sie auch einen *helpfile* und einen *context* angegeben haben.

Die nächste Übersicht enthält die Symbol-Konstanten:

Konstante	Wert	Beschreibung
vbCritical	16	Das Stop-Symbol wird angezeigt.
vbQuestion	32	Das Fragezeichen-Symbol wird angezeigt.
vbExclamation	48	Das Ausrufezeichen-Symbol wird angezeigt.
vbInformation	64	Das Informations-Symbol wird angezeigt.

Bleibt noch `vbSystemModal` übrig, das Pendant zu `vbApplicationModal`. Diese Konstante mit dem Wert 4096 hat zur Folge, dass alle Applikationen ruhen, bis die MessageBox geschlossen wird. Dies liegt in der Nähe einer Nötigung und ist somit verwerflich.

Beispiele:

Die folgende Anweisung erzeugt die erste der vorstehenden MessageBoxes mit benannten Argumenten:

```
MsgBox prompt:="Die Kostenstelle wurde neu berechnet.", _  
        buttons:=vbInformation, title:="KoSt-Berechnung"
```

Und hier die zweite, aber diesmal ohne benannte Argumente:

```
MsgBox "Soll die Kostenstelle neu berechnet werden?", _  
        vbQuestion + vbYesNo, "KoSt-Berechnung"
```

Hier sieht man, dass die beiden Konstanten `vbQuestion` und `vbYesNo` einfach zusammengezählt werden.

An dieser Stelle möchte ich noch ein paar persönliche Worte zu VBA-Programmen und Hilfedateien zum Besten geben. Wenn in VBA-Programmen eine Online-Hilfe erforderlich wird, so ist eine Dimension erreicht, in der ich zu dem hierfür wesentlich besser geeigneten Visual Basic wechseln würde. Erschwerend kommt hinzu, dass die Tools zur Hilfeeinstellung aus dem Hause Microsoft keine Preise gewinnen würden. Hier wären also Produkte von Drittanbietern gefragt, die aber zuerst einmal Geld kosten und auch keineswegs einfach in der Handhabung sind (z. B. HelpMagician oder RoboHelp). Eine Alternative sind HTML-Hilfedateien, die jedoch einen Internet Explorer ab Version 4 erfordern.

Die Rückgabe der MsgBox

Die MessageBox ist nicht nur eine vielseitige, einfach zu nutzende Möglichkeit der Informationsausgabe, sie erzählt uns auch, mit welcher Schaltfläche der Anwender den Dialog quittiert. Die `MsgBox`-Funktion verfügt über eine Rückgabe des Datentyps `Long`, für die folgende Konstanten definiert sind:

Konstante	Wert	Beschreibung
<code>vbOK</code>	1	OK-Schaltfläche
<code>vbCancel</code>	2	Abbrechen-Schaltfläche
<code>vbAbort</code>	3	Abbrechen-Schaltfläche
<code>vbRetry</code>	4	Wiederholen-Schaltfläche
<code>vbIgnore</code>	5	Ignorieren-Schaltfläche
<code>vbYes</code>	6	Ja-Schaltfläche
<code>vbNo</code>	7	Nein-Schaltfläche

Immer wenn Sie von einer Funktion eine Rückgabe erwarten, müssen die Argumente in Klammern eingeschlossen werden. Die folgende Anweisung erzeugt nur die Ausgabe:

```
MsgBox "Geht's gut?", vbYesNo
```

Wollen wir das Ergebnis auswerten, so sieht das Ganze so aus:

```
If MsgBox("Geht's gut?", vbYesNo) = vbYes Then  
    MsgBox "Freut mich."  
Else  
    MsgBox "Schade."  
End If
```

4.10.2 InputBox-Funktion

Wenn der Anwender unsere besorgte Frage, ob's gut geht, mit nein beantwortet, können wir mittels einer InputBox hinterfragen, woran es liegt. Der Else-Zweig könnte dann so aussehen:

```
strReturn = InputBox("Schade. Woran liegt's?")  
If strReturn = "" Then  
    MsgBox "OK, Du magst nicht darüber reden."  
Else  
    MsgBox "Hmm, " & strReturn & ", verstehe."  
End If
```

Wird die InputBox bei einem leeren Textfeld mit der OK-Schaltfläche geschlossen oder die Abbrechen-Schaltfläche geklickt, so ist der Rückgabewert ein leerer String "".

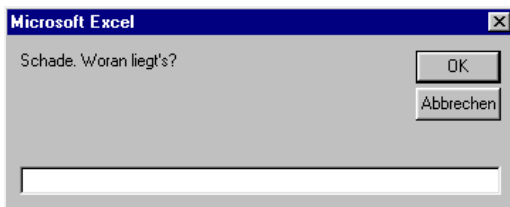


Abbildung 4.5: Beispiel einer InputBox



Objekte

5

Kapitelüberblick

5.1	Objekte und ihre Elemente	169
5.1.1	Eigenschaften	170
5.1.2	Methoden	172
5.1.3	Ereignisse	174
5.2	Objekte und Auflistungen	176
5.2.1	Zugriff auf existierende Objektverweise	177
5.2.2	Erzeugen und Auflösen von Objektverweisen	179
5.2.3	Auflistungen	184
5.2.4	Kapselung, Polymorphie und Vererbung	187
5.3	Zugriff auf Zellen	189
5.3.1	Der Zugriff auf die Objekte	190
5.3.2	Die Kunst des Weglassens	191
5.3.3	Der Einsatz von Objektvariablen	192
5.3.4	Der Pflock in Zelle A1	194

Offen gesagt bereitet mir die Definition des Begriffes Objekt Schwierigkeiten. Und das, obwohl ich seit Jahren damit arbeite.

Zu Anfang glaubte ich, dass Microsoft mit dem Schüler Fritz, der in der Klasse 1 der Schule A sein Dasein fristet, eine gute Metapher gefunden hat. Denn Fritz als Objekt ist Mitglied der Auflistung Schüler in der Auflistung Klassen der Auflistung Schulen in dem Kontext »UnsereStadt«. Immerhin hat Fritz die Eigenschaften Größe, Alter und Geschlecht. Außerdem verfügt er über die Methoden Aufstehen und Singen. Toll, nicht wahr?

Dann schrieb ich irgendwann meine erste DLL und definierte fortan ein Objekt als das, was es ist: eine Instanz einer Klasse. Endlich *wusste* ich, was ein Objekt ist, denn ich hatte ja immerhin ein eigenes gebaut. In Wirklichkeit nutzte ich lediglich eine Reihe überaus raffinierter Features von Visual Basic, die den Kern der Objekte, nämlich COM, geschickt vor uns VBlern verbergen.

Diesen Begriff Component Object Modell (COM) brachte mir 1996 ein Buch von David Chapell näher. Zweieinhalb Jahre später las ich COM- und MTS-Programmierung mit Visual Basic von Ted Pattison. Und obwohl ich bis zu diesem Zeitpunkt wirklich zu wissen glaubte, was ein Objekt ist, entdeckte ich in diesem sehr erbaulichen Werk doch eine Reihe von Informationen, die mein Bild von Objekten teils untermauerten, teils aber auch revidierten. Auch will ich nicht verleugnen, dass dieses Buch eine Reihe von Fragezeichen in meinem Kopf hinterließ. Ein Zitat aus dem Vorwort dieses Buches von Don Box:

Bis heute werden Sie kaum zwei Menschen finden, die eine vollständig übereinstimmende Meinung darüber besitzen, was COM ist.

Wie tröstlich.

Dann las ich 1999 Dan Applemans Werk Com/ActiveX-komponenten mit Visual Basic 6 entwickeln (genau genommen lese ich seit einem halben Jahr in diesem Buch wie ein Priester in seinem Brevier) und lernte schon wieder eine ganze Menge über Objekte und ActiveX-Komponenten.

Weshalb erzähle ich Ihnen diese Geschichte? Ich möchte einem Gefühl entgegenwirken, das Sie möglicherweise bei Ihren ersten Kontakten mit Objekten beschleicht: »Das kapiere ich nie!« könnte eine Erscheinungsform dieses Gefühles sein. Vergessen Sie's. Mit jeder Programmzeile, die einen erfolgreichen Objektzugriff enthält, steigt Ihre Sicherheit im Umgang mit Objekten. Sie werden mit der Zeit eine Routine erreichen und die Objekte werden das tun, was Sie von ihnen erwarten.

Am Ende dieses Kapitels werden wir die Begriffe untersuchen, die zum Teil bereits fielen. Am Ende deshalb, weil es leichter fällt, einem Phänomen,

das man schon irgendwo gesehen hat, einen Namen zuzuordnen als sich mit einer Begriffsdefinition herumzuschlagen, die abstrakter kaum sein könnte. In Kapitel 10 – Klassen und DLLs werden wir auch selbst Klassen bauen, die wir als Objekte zum Leben erwecken.

Schauen Sie sich an, wie erfolgreich Kinder eine oder gar mehrere Sprachen lernen, ohne dass sie überhaupt wissen, was die Begriffe Konjugation und Deklination, Semantik oder Syntax bedeuten. Wenn sie in die Schule kommen, verfügen diese Knirpse und Knirpsinnen über einen erstaunlichen Sprachschatz, den sie sicher beherrschen.

Wenn Ihr persönlicher Lerntypus diesem effizienten Typus der Kinder sehr ähnlich ist, dann lassen Sie bitte ein paar Erklärungsversuche in diesem Kapitel über sich ergehen. Danach widmen wir uns unverzüglich dem experimentellen Stadium, dem des Begreifens. In diesem Wort steckt das Verb greifen. Wir werden in die Tasten greifen und Objekte kennen lernen. Und ich werde Ihnen die Wahrheit über Objekte erzählen, zu erklären versuchen, wie sie funktionieren, und Ihnen keine (oder besser gesagt, fast keine) mehr oder weniger sinnvollen biologischen Analogien präsentieren, die letztendlich doch alle ihre Schwächen haben.

5.1 Objekte und ihre Elemente

Vor längerer Zeit saß in einem 5-tägigen Excel-VBA-Seminar eine blitzgescheite junge Frau, die am ersten Tag mächtige Probleme mit dem Begriff Objekt hatte. Auch am zweiten Tag gelang uns der Durchbruch nicht, sie verließ sich allerdings auf meine Aussage, dass der Groschen gelegentlich nicht am Stück rutscht, sondern möglicherweise auch molekülweise. Doch irgendwann rutscht er. Am fünften Tag platzte sie früh morgens in den Seminarraum und erklärte ganz aufgeregt, dass sie nun wisse, was ein Objekt sei:

Ein Objekt ist ein Ding, mit dem man etwas machen kann.

Sie werden möglicherweise staunen, aber das ist eine der treffendsten Definitionen des Begriffs Objekt, die mir bislang zu Ohren kam.

Nun überlegen Sie mal, mit welchem Ding Sie als einfacher Anwender in Excel etwas machen können.

Haben Sie schon einmal die Zahl 7 in die Zelle B4 einer Tabelle geschrieben? Wenn ja, dann haben Sie die Eigenschaft *Value* des Objekts *Range*(»B4«) des betreffenden *Worksheets* verändert.

Haben Sie schon einmal den Namen der Tabelle »Tabelle1« in »GuV 1999« geändert? Wenn ja, dann haben Sie die Eigenschaft *Name* des Objekts *Tabelle1* in »GuV 1999« geändert.

Haben Sie schon einmal eine Tabelle nach Spalte A sortiert? Wenn ja, dann haben Sie die Methode *Sort* des Objektes *Range.CurrentRegion* angewandt und den zugrunde liegenden Zellbereich indirekt verändert.

Haben Sie schon einmal in Zelle A4 die Formel `=SUMME(A1:A3)` berechnet? Wenn ja, so haben Sie die Methode *Sum* des Objekts *Application* aufgerufen, ihr das Objekt *Range(»A1:A3«)* übergeben und das Ergebnis dieser Berechnung in das Objekt *Range(»A4«)* eingetragen.

Haben Sie schon einmal eine Arbeitsmappe geschlossen? Wenn ja, so wendeten Sie die Methode *Close* des Objekts *Workbook* an.

Also haben Sie vermutlich in Ihrem Leben 10ⁿ Objekte in Excel manipuliert, ohne sich dessen unmittelbar bewusst gewesen zu sein. Genaugenommen ist also Excel eine Oberfläche, die der Manipulation von Objekten dient. Unter dieser Oberfläche sitzt für den Anwender unsichtbar eine Objektbibliothek, in der die eigentliche Arbeit passiert. Und genau dort können wir uns als VBA-Entwickler einklinken und die mächtigen Fähigkeiten dieser Bibliothek nutzen.

Hier können wir also festhalten, dass der normale Anwender schon eine Reihe von Objekten manipuliert hat:

- ▼ eine Zelle oder auch ein Range-Objekt
- ▼ eine Tabelle oder auch ein Worksheet-Objekt
- ▼ eine Arbeitsmappe oder auch ein Workbook-Objekt

5.1.1 Eigenschaften

Schauen wir nun, auf welche Art und Weise sich ein Objekt manipulieren lässt. Es war die Rede von Eigenschaften und Methoden, so zum Beispiel von der Eigenschaft *Value* (Wert) des Range-Objekts (der Zelle) und der Eigenschaft *Name* des Worksheet-Objekts (der Tabelle). Durch Verändern einer Eigenschaft wird das Objekt verändert: das Objekt Zelle hat einen anderen Inhalt und das Objekt Tabelle einen anderen Namen.

Die einzig zuverlässige Möglichkeit, Eigenschaften und Methoden in der Praxis zu unterscheiden, liegt in dem Merkmal, dass eine Eigenschaft ihren neuen Wert immer durch eine Zuweisung erhält, in der ein Gleichheitsoperator zu finden ist:

```
Object.Value = "Hallo"
```

Hier wird also die *Value*-Eigenschaft des davor stehenden Objekts *Object* verändert. Der Punkt `».«` zwischen dem Objekt und der Eigenschaft trennt die beiden Begriffe voneinander.

Eigenschaften unter der Lupe

Eine Variable kann generell gelesen und geschrieben werden:

```
Dim strName As String  
strName = "Scrooge McDuck"  
MsgBox "Der Name lautet " & strName
```

Die Anweisung `strName = "Scrooge McDuck"` beschreibt die Variable, wohingegen sie durch `MsgBox "Der Name lautet " & strName` gelesen wird. Soweit ist das nichts Neues.

In Form der Konstanten haben wir aber auch Komponenten in VBA, die nur das Lesen eines Wertes erlauben. Diese Konstanten können entweder vom System, also von Bibliotheken, oder von uns erzeugt werden. Hier ein Beispiel einer selbst erzeugten Konstanten:

```
Const iColDate As Long = 3  
rng.Cells(3, iColDate).Value = Date
```

Die folgende Zeile würde aber einen Kompilierfehler produzieren, da Konstanten schreibgeschützt sind:

```
iColDate = 17
```

Diese Mechanismen sind auch auf Eigenschaften übertragbar, denn auch da gibt es Eigenschaften, die gelesen und geschrieben werden können, aber auch solche, bei denen ein Schreibzugriff nicht erlaubt ist. Auch wenn nicht jede Eigenschaft geschrieben werden kann, so erzwingt es die Logik, dass alle Eigenschaften gelesen werden können.

Die `Name`-Eigenschaft einer Tabelle (Worksheet-Objekt) zum Beispiel kann in beiden Richtungen verwendet werden, denn wir dürfen den Namen einer Tabelle verändern. Anders verhält es sich mit der Eigenschaft einer Arbeitsmappe, die uns die Anzahl der darin enthaltenen Tabellen mitteilt. Diese kann natürlich gelesen werden, aber wir können eine Arbeitsmappe mit drei Tabellen auf diesem Wege nicht davon überzeugen, nun über vier Tabellen zu verfügen. Somit kann diese Eigenschaft nur gelesen werden.

Implementiert sind Eigenschaften von Objekten immer mit so genannten Eigenschaftsprozeduren, die wir in Kapitel 10 noch näher untersuchen werden. Für jede Transportrichtung, also Lesen und Schreiben, existiert eine eigene Prozedur. Die eine gibt den Wert der Eigenschaft zurück, die andere, sofern sie existiert, nimmt ihn entgegen.

Abbildung 5.1 zeigt dies am Beispiel der `Name`-Eigenschaft eines Worksheet-Objekts, also einer Tabelle. In der Anweisung `shtX.Name = "User"`

wird die Schnittstelle der Name-Eigenschaft aufgerufen und ihr die Zeichenkette "User" übergeben. In der nächsten Zeile wird mit `MsgBox shtX.Name` diese Eigenschaft ausgelesen und ihr Wert der `MsgBox` zur Anzeige übergeben.

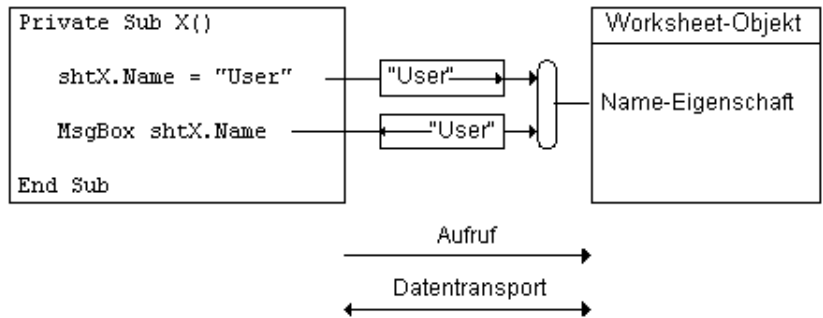


Abbildung 5.1: Eigenschaften

Es ist wichtig, dass Sie hierbei zwischen der Richtung des Aufrufs der Schnittstelle und des Datentransports unterscheiden. Unser Code in der Sub `X` rüttelt an der Schnittstelle des `Worksheet`-Objekts, auch wenn es unser Anliegen ist, dass das `Worksheet`-Objekt uns den Inhalt seiner `Name`-Eigenschaft zurückgibt. Wir rufen diese Schnittstelle auf.

5.1.2 Methoden

Methoden haben zumeist etwas mit größeren, dynamischen Zustandsänderungen des zugrunde liegenden Objektes zu tun. Denken Sie beispielsweise an das Sortieren eines Zellbereichs. Dieser Prozess besteht gleich aus mehreren Komponenten:

- ▼ Der Prozess benötigt Informationen, um die Sortierung steuern zu können. Dazu gehören Angaben darüber, nach welchen Spalten und, innerhalb der jeweiligen Spalte, nach welcher Sortierreihenfolge sortiert werden soll.
- ▼ Danach werden die Originaldaten an irgendeiner Stelle in der gewünschten Sortierung aufgebaut.
- ▼ Abschließend werden die sortierten Daten in die Ausgangstabelle übertragen.

Allein schon die Anzahl an Informationen verhindert, diese Zustandsänderung durch eine Eigenschaft hervorrufen zu lassen, denn eine Eigenschaft vermag immer nur einen Wert entgegenzunehmen.

Methoden sind auch generell dann erforderlich, wenn ein weiteres zu einer Reihe von existierenden Elementen hinzugefügt werden soll. So zum Beispiel das Hinzufügen einer neuen Tabelle in eine Arbeitsmappe oder das Erzeugen einer neuen, leeren Arbeitsmappe. Weiter oben war zu lesen, dass das Gleichheitszeichen ein Kennzeichen einer Eigenschaft ist. Das heißt im Umkehrschluss, dass das Fehlen dieses Gleichheitszeichens eindeutig auf eine Methode hinweist:

```
Object.Sort Argument1, Argument2, ..., Argumentn
```

Hier ist `Sort` eine Methode des voranstehenden Objekts `Object`.

Methoden unter der Lupe

Auch Methoden werden ebenfalls unabhängig von einem eventuellen Datentransport von uns aufgerufen. Das erste Beispiel der Abbildung 5.2 zeigt die Methode `Calculate`. Es handelt sich bei `Calculate` um eine Methode mit der einfachst möglichen Signatur, denn sie benötigt weder Informationen darüber, wie denn nun zu berechnen sei, noch gibt sie uns das Ergebnis der Berechnung zurück, denn `Calculate` bezieht sich auf den Zellbereich, der durch das mit `rngX` bezeichnete Objekt repräsentiert ist.

Das zweite Beispiel in Abbildung 5.2 zeigt die `Find`-Methode, die in einem Zellbereich nach einer Zelle sucht, die einen gewissen Inhalt hat. Damit diese Methode arbeiten kann, muss sie den Suchbegriff (und noch eine Reihe mehr) kennen, nach dem sie suchen soll. In diesem Beispiel wird ihr der Suchbegriff »Hallo« übergeben. Eine solche zusätzliche Information, die einer Methode übergeben wird, nennt man `Argument`.

Nun gehört aber die `Find`-Methode zur Gruppe der Methoden, die das Ergebnis – in diesem Falle des Suchprozesses – auch zurückliefern. Das tut sie, indem sie einen Verweis auf die gefundene Zelle übergibt, der in der Abbildung mit `Range` bezeichnet ist.

Auch bei Methoden verläuft die Richtung des Aufrufs von unserem Code zum Objekt. Der Transport zusätzlicher Informationen als Argumente und eine eventuelle Rückgabe ändern nichts daran.

Nachdem wir bei Eigenschaften und Methoden einen solch großen Wert auf die Richtung des Aufrufs legten, können Sie davon ausgehen, dass sich diese Richtung auch ändern kann. Und genau das wird bei den nun behandelten Ereignissen der Fall sein.

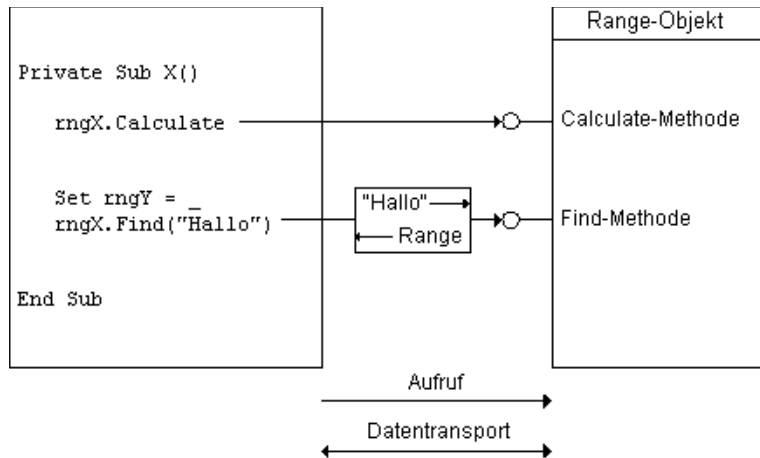


Abbildung 5.2: Methoden

5.1.3 Ereignisse

Bevor Objekte Einzug hielten in die Programmierung, hatten wir Entwickler die Steuerung des Programmes völlig in der Hand. Schien es uns passend, so boten wir dem Anwender ein Menü an, in dem er unter einer Reihe von uns definierten Punkten etwas auswählen durfte. Im Übrigen wurde er nicht gefragt. Die Steuerung des Programmes erfolgt vollständig von innen heraus. Und ... das Programm lief ständig. Selbst wenn ein Menü auf dem Bildschirm erschien, befand sich das Programm in irgendeiner Menüschleife, von der aus diverse Ausgänge herausführten.

Bildlich gesprochen hatte das Programm alle naslang den Tastaturpuffer gefragt, ob eine Nachricht vorläge. Vernunftbegabte Wesen würden nach kurzer Zeit zu einem anderen Modus übergehen und den Spieß umdrehen. Der modus operandi hieße dann vermutlich: »Jetzt halt mal die Füße still. Ich werde dir Bescheid sagen, wenn sich etwas tut.« Auf diese Idee wäre die Software alleine nie gekommen.

Die daraus resultierende Änderung des Mechanismus ist ein wesentlicher Baustein moderner Betriebssysteme. Dadurch ist es merklich ruhiger geworden im Innern eines Rechners, denn dort verbringen die Programme die meiste Zeit mit Warten, lassen auch anderen Programmen Raum und haben (zumindest die meisten) aufgehört sich gegenseitig bei der Vergabe von Interrupt-Prioritäten auszutricksen. Programme warten aber nicht auf Godot, Programme warten auf Benutzeraktionen und Systemmeldungen. Die Programme haben nicht mehr das Heft in der Hand, sondern äußere Einflüsse bestimmen zunehmend das Geschehen, und sie reagieren nur noch auf diese Einflüsse, die wir Ereignisse nennen.

Ein VBA-Programm wird in der Regel durch ein Ereignis gestartet. Das kann sein:

- ▼ Eine Befehlsschaltfläche wird angeklickt.
- ▼ Eine Tabelle wird ausgewählt.
- ▼ Der Wert einer Tabellenzelle wird verändert.
- ▼ Eine Arbeitsmappe wird geöffnet oder geschlossen.
- ▼ Eine Tabelle wird ausgedruckt.

Wenn aber, wie wir feststellten, nicht unser Programm laufend und reihum alle möglichen Objekte nach irgendwelchen Zustandsänderungen befragen kann, muss also eine Möglichkeit gefunden werden, wie diese Objekte (oder auch das Betriebssystem) uns benachrichtigen können.

In Abbildung 5.3 ist ein solches Szenarium aufgebaut. Dort wird die `Calculate`-Methode des `Worksheet`-Objekts aufgerufen. Das `Worksheet`-Objekt verfügt aber auch über ein Ereignis namens `Calculate`. Wenn wir dem `Worksheet`-Objekt nun unsererseits eine Schnittstelle zur Verfügung stellen, so ruft es diese auf. Das funktioniert deshalb, weil auch unser Codemodul ein COM-Objekt ist und COM-Objekte nun mal über Schnittstellen verfügen. Das `Calculate`-Ereignis des `Worksheet`-Objekts wird immer ausgelöst und fragt eine besondere Schnittstelle unseres Codemoduls, ob es dort die Prozedur `Worksheet_Calculate` gibt. Wenn ja, so ruft das `Worksheet`-Objekt diese Prozedur auf.

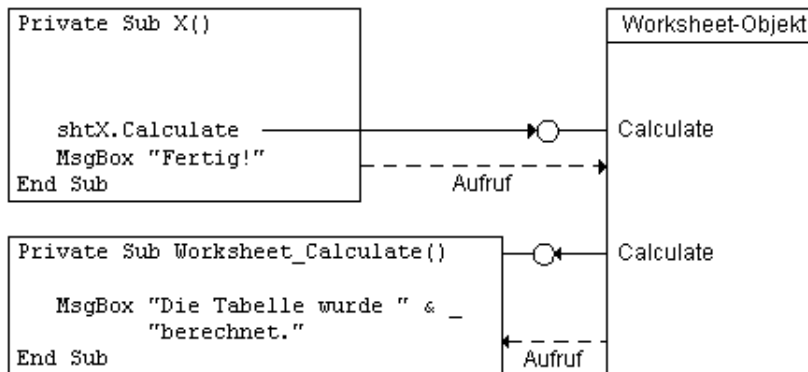


Abbildung 5.3: Ereignisse

Hier haben wir also endlich die Aufrufrichtung umgekehrt, und unser Code wird von einem Objekt aufgerufen. Wenn wir wollen, dass unser Objekt uns ein Ereignis sendet, so müssen wir lediglich die passende Ereignisprozedur bereitstellen. Diese können Sie Zeichen für Zeichen von Hand schreiben, aber auch vom Codefenster erzeugen lassen.

Natürlich ist es reichlich unsinnig, sich von dem `Worksheet`-Objekt eine Nachricht zukommen zu lassen, dass dieses gerade berechnet wurde, wenn man unmittelbar vorher einen entsprechenden Auftrag vergeben hat.

Nun die Frage, auf die Sie sicherlich alle gewartet haben: Welche Message-Box kommt zuerst? »Fertig!« oder »Die Tabelle wurde berechnet.«?

5.2 Objekte und Auflistungen

Ein Objekt ist ein irgendwo im Hauptspeicher liegendes Stück Software mit Schnittstellen für Eigenschaften und Methoden und gelegentlich auch mit Rückrufmechanismen, die wir Ereignisse genannt haben. Sobald Sie eine Arbeitsmappe öffnen, werden die Objekte dieser Arbeitsmappe geladen. Dazu gehören neben dem `Application`-Objekt (Excel selbst) ein `Workbook`-Objekt und eine entsprechenden Anzahl von `Worksheet`-Objekten, eins für jede Tabelle. Enthält die Arbeitsmappe Diagramme, so wird auch eine entsprechende Anzahl von `Chart`-Objekten erzeugt. Diese Objekte werden nicht (nur) erzeugt, damit wir per Entwicklungsumgebung darauf zugreifen können, sondern auch eifrig von Excel selbst verwendet. Die Programmierbarkeit von Excel oder Word durch VBA ist nicht das Ergebnis einer gesonderten Anstrengung von Microsoft, sondern Basis der mit OLE 2 ins Leben gerufenen Automation (einst OLE-Automation genannt), die 1994 in die Microsoft Office-Produkte Einzug hielt.

Objektverweis

Wenn wir mit einem Excel-Objekt arbeiten wollen, so benötigen wir einen *Verweis* auf dieses bereits im Speicher befindliche Objekt. Ein solcher Verweis ist im Grunde genommen ein `Long`-Datentyp, der auf die Speicheradresse des Objekts *verweist*. Schreiben wir also `Objekt.Eigenschaft = "Hallo"`, so weiß VBA durch den Objektverweis, an welcher Speicheradresse das Objekt zu finden ist, mit dem es sich über die verwendeten Eigenschaften oder Methoden unterhalten muss. Im Grunde genommen ist das auch *der* Mechanismus (Dispatch-Interface), der uns die Elemente eines Objektes auflistet, sobald wir einen Punkt hinter einen Ausdruck schreiben.

5.2.1 Zugriff auf existierende Objektverweise

Einige Objektinstanzen existieren bereits, wenn wir die Entwicklungsumgebung starten. Zum einen ist es das `Application`-Objekt, wohinter sich Excel selbst verbirgt (genau genommen die laufende Excel-Instanz). Auch ohne eine Arbeitsmappe geöffnet zu haben, funktioniert das `Application`-Objekt (soweit sich die Eigenschaft oder die Methode nicht auf eine Arbeitsmappe bezieht), wie Abbildung 5.4 zeigt.

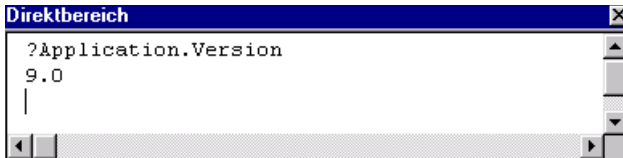


Abbildung 5.4: Application-Objekt im Direktfenster

Zum anderen gesellt sich eine Reihe weiterer, existierender Objektverweise hinzu, sobald Sie eine Arbeitsmappe geöffnet haben.

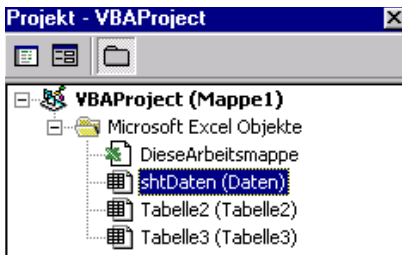


Abbildung 5.5: Excel-Objekte im Projekt-Explorer

Die Namen dieser Objekte sind in Abbildung 5.5 zu sehen. Es handelt sich hierbei um `VBAProject`, `DieseArbeitsmappe`, `shtDaten`, `Tabelle2` und `Tabelle3`. In Abbildung 5.6 ist wieder ein Direktfenster zu sehen, in dem diese Objekte zum Einsatz kommen, indem jeweils deren `Name`-Eigenschaft abgefragt wurde.

Unter den vier genannten Objektnamen `DieseArbeitsmappe`, `shtDaten`, `Tabelle2` und `Tabelle3` verbergen sich also die Objektverweise zur Arbeitsmappe sowie deren drei Tabellen. Daneben sind die beiden Ausdrücke `ThisWorkbook` und `ActiveWorkbook` zu sehen.

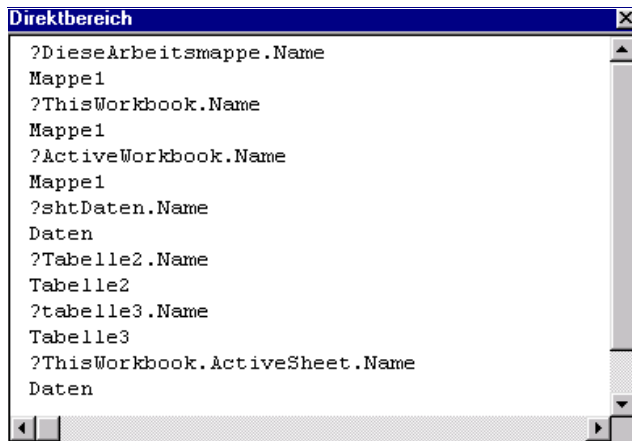


Abbildung 5.6: Projekt-Objekte im Direktfenster

ThisWorkbook ThisWorkbook ist die Arbeitsmappe, die im Projekt-Explorer den Fokus hat. Codemodule sind immer an eine Arbeitsmappe gebunden und unter ThisWorkbook können Sie im Code die Arbeitsmappe ansprechen, die das jeweilige Codemodul beherbergt. Hier ist die Beziehung also eindeutig. Sind Sie hingegen im Direktfenster, so haben Sie mit ThisWorkbook einen Verweis zu der Arbeitsmappe mit dem Fokus im Projekt-Explorer.

ActiveWorkbook Unter ActiveWorkbook verbirgt sich *immer* die Arbeitsmappe, die gerade den Fokus hat. Und das macht diesen Ausdruck so gefährlich. Lediglich wenn Sie beispielsweise in Form eines Add-Ins Funktionen bereit stellen, die quasi überall verwendet werden können, sind Verweise auf die aktive Arbeitsmappe sinnvoll. Im folgenden Kapitel 5.2.2 werden wir sehen, wie man der mit diesem Ausdruck verbundenen Gefahr weiträumig aus dem Wege geht.

ActiveSheet Hinter dem Ausdruck ActiveSheet, der auf jede Arbeitsmappe angewendet werden kann, verbirgt sich die Tabelle, die gerade den Fokus besitzt. In Excel selbst ist es die, der Sie ins Antlitz sehen oder, wenn Sie mehrere Fenster nebeneinander angeordnet haben, deren Titelleiste die Titelleisten zugeordnete Farbe hat (Systemsteuerung). Auch hier gilt das voll inhaltlich, was auch zu dem Begriff ActiveWorkbook ausgeführt wurde: dieser Ausdruck ist gefährlich und selten wirklich nötig. Auch darüber werden wir in Kapitel 5.2.2 zu reden haben.

Neben diesen explizit verfügbaren Objekten gibt es auch eine Reihe anderer, impliziter Objekte, die sich aus diesen ableiten. Ein Worksheet-Objekt hat über Methode Zugriff zu Zellbereichen und ein Zellbereich wiederum ist ein Range-Objekt usw. Doch diese impliziten Objekte lassen wir hier außer Betracht.

5.2.2 Erzeugen und Auflösen von Objektverweisen

In Kapitel 5.2.3 werden uns die für komplexe Anwendungen wie Excel, Word oder PowerPoint üblichen Objektketten noch begegnen. Hier ein Beispiel:

```
ThisWorkbook.Worksheets("GuV 1999").Range("A1").Value
```

Die Objektkette in dieser Beispielzeile ist kein sonderlich großer Vertreter der Gattung, man hätte sie im Gegenteil auch auf drei Zeilen ausdehnen können, ohne sich dem Vorwurf der Konstruktion schuldig zu machen.

Abgesehen davon, dass solche Ausdrücke unnötig schwer lesbar sind, kosten sie auch Rechenzeit, da sich der Interpreter beim Bearbeiten eines solchen Ausdrucks von Punkt zu Punkt zu immer neuen Objektadressen durchfragen muss. `ThisWorkbook` ist ein Objekt oder besser ein Objektverweis, hinter dem die Speicheradresse der Arbeitsmappe steckt. Der Aufruf der `Worksheets`-Eigenschaft mit dem Namen der gewünschten Tabelle gibt wiederum ein `Worksheet`-Objekt zurück – genau genommen eine weitere Speicheradresse zu dem `Worksheet`-Objekt, wie wir ja inzwischen wissen. Nun wendet sich der Interpreter an diese neue Adresse und wird wegen der `Range`-Eigenschaft vorstellig. Und was erhält er? Vermutlich eine neue Adresse, an die er sich wenden kann, um dort etwas über eine gewisse `Value`-Eigenschaft in Erfahrung zu bringen (siehe Abbildung 5.7)

Nun hat so ein Interpreter ja keinen Verstand, sonst würde er spätestens beim zweiten Mal erkennen, dass er bei diesem `Range`-Objekt schon mal war, und sich für alle Fälle die Adresse notieren. Tut er aber nicht.

Aber wir können die Adresse in Erfahrung bringen und einem Objektverweis zuordnen. Und das geschieht folgendermaßen:

```
'Objektverweis erzeugen ...  
Dim rng As Range  
'... und nun zuweisen  
Set rng = ThisWorkbook.Worksheets("GuV 1999").Range("A1")
```

Wir haben einen neuen Objektverweis erzeugt und die direkte Adresse des `Range A1` der Tabelle `GuV 1999` in dieser Arbeitsmappe dem Objekt `rng` zugewiesen. Und wenn wir jetzt auf den `Value` dieses `Range`-Objekts zugreifen, wendet sich dieser Ausdruck direkt an die Adresse des `Range`-Objekts und fragt dort nach der `Value`-Eigenschaft.

Wie das Beispiel zeigte, werden Objektvariablen immer mit `Set` zugewiesen.

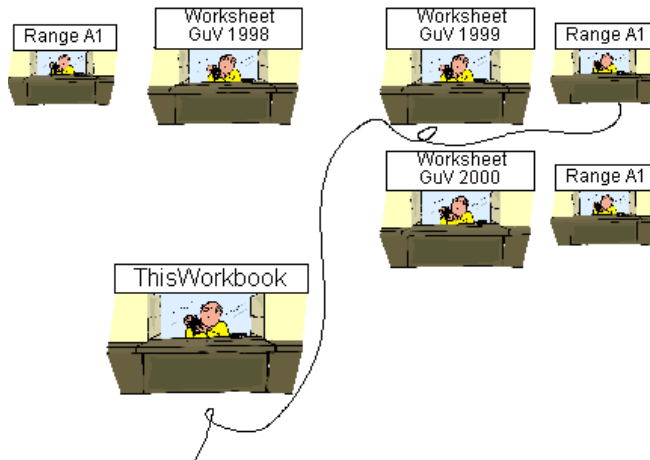


Abbildung 5.7: Der lange Weg

Wenn Sie einmal auf ein Objekt innerhalb einer Objektkette zugreifen müssen, lohnt sich ein Objektverweis in der Regel nicht. Erfolgt der Zugriff aber öfters, so sparen Sie eine Menge Zeit, wenn Sie ein Objekt auf die gewünschte Stelle verweisen.

Was tun wir, wenn ein Objekt nicht mehr benötigt wird? Kommt darauf an.

Existiert das Objekt bereits und wir haben nur eine zusätzliche Verbindung zu diesem Objekt per Objektvariable hergestellt, so müssen wir nichts tun. VBA kümmert sich gemeinsam mit der hinter dieser ganzen Objektkiste steckenden COM-Technologie automatisch um die Auflösung solcher Verweise.

```
Private Sub A()  
    Call B  
    ...  
End Sub  
  
Private Sub B()  
    Dim rng As Range  
    Set rng = ThisWorkbook.Worksheets("Daten").Range("A1")  
    ...  
End Sub
```

Routine A ruft B auf, wo eine lokale Objektvariable auf einen Range verweist. Wird die End Sub-Anweisung in Sub B erreicht, wird die lokale Referenz auf das Range-Objekt automatisch aufgelöst. Wir haben dieses Range-Objekt nicht erzeugt, denn es existierte bereits irgendwo in einer Ecke des Objekts ThisWorkbook.

Aber es gibt auch Objekte, die wir erzeugen, indem wir sie überhaupt erst in den Speicher laden. Nehmen wir als Beispiel eine Arbeitsmappe, die wir per Code öffnen:

```
Private Sub OpenWorkbook()  
    Workbooks.Open "D:\Test\TestMappe.xls"  
End Sub
```

Danach ist die Arbeitsmappe zwar geöffnet und somit in den Speicher geladen worden, aber wir können erst mal nichts damit anfangen, da wir keinen Verweis auf das Objekt besitzen. Wir könnten uns an Excel wenden und es bitten, uns mit der Arbeitsmappe TestMappe.xls verbinden, um dann irgendetwas damit anzustellen:

```
Workbooks("TestMappe.xls").Worksheets("Daten").Name =  
"abc"
```

Doch damit haben wir wieder unsere Ausgangssituation geschaffen, die in Abbildung 5.7 wiedergegeben ist. Die Lösung liegt natürlich in einer Objektvariablen, die auf das Workbook verweist:

```
Private Sub OpenWorkbook()  
    Dim wbkTest As Workbook  
    Workbooks.Open "D:\Test\TestMappe.xls"  
    Set wbkTest = Workbooks("TestMappe.xls")  
    wbkTest.Worksheets("Daten").Name = "abc"  
End Sub
```

Doch das Öffnen und das Zuweisen der Variablen geht bei den meisten Objekten in einem Schritt:

```
Private Sub OpenWorkbook()  
    Dim wbkTest As Workbook  
    Set wbkTest = Workbooks.Open("D:\Test\TestMappe.xls")  
    wbkTest.Worksheets("Daten").Name = "abc"  
End Sub
```

Wenn Sie nun öfters auf das Worksheet Daten zugreifen müssen, lohnt sich natürlich auch eine entsprechende Objektvariable:

```
Private Sub OpenWorkbook()  
    Dim wbkTest As Workbook  
    Dim shtDaten As Worksheet  
    Set wbkTest = Workbooks.Open("D:\Test\TestMappe.xls")  
    Set shtDaten = wbkTest.Worksheets("Daten")  
    shtDaten.Name = "abc"  
End Sub
```

Um die Variable `shtDaten` müssen wir uns nicht kümmern, denn sie wird automatisch mit dem `Workbook`, zu dem sie ja gehört, aufgelöst, sobald das `Workbook` aus dem Speicher entfernt wird. Doch was machen wir mit der `Workbook`-Variablen `wbkTest` und wie schließen wir die Arbeitsmappe überhaupt?

Auch hier übernimmt VBA zusammen mit dem bereits zitierten COM-Mechanismus das Auflösen aller Objektvariablen, sobald das Objekt aus dem Speicher entfernt wird. Wir müssen uns also nur noch darum kümmern, die Arbeitsmappe zu schließen. Und dafür stellt uns jedes `Workbook`-Objekt eine Methode namens `Close` zur Verfügung:

```
Private Sub OpenWorkbook()  
    Dim wbkTest As Workbook  
    Set wbkTest = Workbooks.Open("D:\Test\TestMappe.xls")  
    ...  
    wbkTest.Close  
End Sub
```

Sobald nun das Ende des Gültigkeitsbereichs der Variablen `wbkTest` erreicht wird, löst VBA die Variable automatisch auf.

Es stellt sich nun die Frage, ob es überhaupt die Notwendigkeit zum Auflösen einer Objektvariablen gibt.



In VBA müssen wir uns nur in Ausnahmefällen um die Auflösung von Objektverweisen kümmern. Und überall dort, wo es erforderlich scheint, kann man zumindest von einer Designschwäche, wenn nicht gar von einem Bug ausgehen.

Und hier die sehnlichst erwartete Antwort auf die Frage, wie man denn einen Objektverweis auflöst, so es denn einmal erforderlich sein sollte:

```
Set wbkTest = Nothing
```

Objektverweis mit With – End With

Ein Verweis auf ein Objekt mittels Variable ist also ein effizienter Weg bei wiederholten Zugriffen auf ein Objekt innerhalb einer Objektkette. Als nachteilig mag gelten, dass die Variable deklariert und zugewiesen werden muss. Wenn es sich bei dem gewünschten Zugriff auf eine örtlich beschränkte Angelegenheit handelt, so bietet sich uns über die Konstruktion `With` und `End With` eine interessante Alternative zur Objektvariablen.

Im folgenden Codefragment wird die Titelzeile einer Tabelle formatiert:

```
Dim rngTitel As Range
Dim fntTitel As Font
Set rngTitel = shtTest.Range("A1:B1")
Set fntTitel = rngTitel.Font
rngTitel.Value = "Titel 1"
rngTitel.Cells(1, 2).Value = "Titel 2"
fntTitel.Size = 12
fntTitel.Bold = True
```

Mit einer With-Konstruktion anstelle einer Variablen sieht sie so aus:

```
With shtTest.Range("A1:B1")
    .Value = "Titel 1"
    .Cells(1, 2).Value = "Titel 2"
    With .Font
        .Size = 12
        .Bold = True
    End With
End With
```

Als Vorteile werden von Anhängern dieser Technik angeführt, dass sie einen direkten Verweis zur Speicheradresse der Objektdaten einrichtet und somit schneller ist als die Wiederholung eines längeren Objektverweises. Tolle Sache, aber genau das tut eine Objektvariable auch. Man spart aber Tipperei, sagen sie. Stimmt, aber man muss ständig im Hinterkopf die With-Zeile mit sich herumschleppen, damit man weiß, worauf sich ein .Size = 12 bezieht. Ich verwende With-Konstruktionen nicht und hätte das obige Beispiel so realisiert:

```
Dim rngTitel As Range
Set rngTitel = shtTest.Range("A1:B1")
rngTitel.Value = "Titel 1"
rngTitel.Cells(1, 2).Value = "Titel 2"
rngTitel.Font.Size = 12
rngTitel.Font.Bold = True
```

Der einzig wirklich sinnvolle Einsatz einer solchen Konstruktion finden Sie in Artikeln in Fachzeitschriften, in deren mittlerweile nur noch daunenbreiten Spalten kein Objektbezug ohne Zeilenumbruch darstellbar ist.

ActiveWorkbook

Wir sprachen bereits davon, dass der Ausdruck `ActiveWorkbook` gefährlich ist. Sie wissen nun auch, dass man einen Objektverweis auf jede beliebige Arbeitsmappe herstellen kann. Und das zusammen macht die Verwendung von `ActiveWorkbook` überflüssig.

Wir haben zwei Szenarien zu berücksichtigen.

Ist die Arbeitsmappe bereits geöffnet, so erzeugen wir einen Verweis auf die geöffnete Arbeitsmappe mit:

```
Set wbk = Workbooks("TestMappe.xls")
```

Ist die Arbeitsmappe hingegen noch nicht geöffnet, so können wir das Öffnen und Zuweisen in einer Zeile erledigen:

```
Set wbkTest = Workbooks.Open("D:\Test\TestMappe.xls")
```

ActiveSheet

Dieser Ausdruck verweist auf das aktuelle Blatt der aktuellen Arbeitsmappe. Außer bei Add-Ins fällt mir keine Situation ein, bei der wir weder den Namen der Arbeitsmappe noch den der Tabelle- oder des Diagrammblatts kennen. Somit spricht absolut alles für die Verwendung einer Objektvariablen mit einem Verweis auf die gewünschte Tabelle oder das Diagrammblatt:

```
Sub MakeSalesChart()
    Dim shtSales As Worksheet
    Dim chartSales As Chart
    Set chartSales = wbkSales.Charts("Umsatz-Chart")
    Set shtSales = wbkSales.Worksheets("Umsatz-Daten")
End Sub
```

5.2.3 Auflistungen

Eine Auflistung (Collection) ist eine Gruppe gleichartiger Dinge. Ein Obstkorb enthält eine Früchteauflistung und eine VBA-Auflistung enthält eben Objekte. Gemeinsam ist beiden Auflistungen, dass gelegentlich der Wurm drin ist.

Obwohl ich Ihnen versprochen hatte, Sie mit biologischen Analogien zu verschonen, erlauben Sie mir bitte, dass wir noch kurz bei dem Obstkorb verweilen.

Eine Auflistung in unserem Sinne wäre sowohl eine Sammlung verschiedenster Früchte, aber auch eine Gruppe genetisch identischer Äpfel oder Birnen.

In unserer VBA-Welt gibt es Vertreter beider Modelle. Die einfachsten und auch gebräuchlichsten Auflistungen enthalten Objekte identischer Struktur oder besser Signatur. Hierzu gehören die *Workbooks*-, die *Worksheets*- oder die *Cells*-Auflistung. Die Tabellen einer Arbeitsmappe werden wohl kaum identisch sein und sicherlich nicht den selben Namen haben, aber sie haben alle eine *Name*-Eigenschaft, die den Namen der Tabelle enthält, und verfügen alle über 265 Spalten und 65536 Zeilen. Man sagt hier auch, sie gehören zur selben Klasse, sind Implementierungen oder auch Instanzen dieser Klasse (siehe Abbildung 5.8).

Die Klasse (oder der Typ, wenn Sie so wollen) *Worksheet* bestimmt über Eigenschaften, Methoden und Ereignisse den Aufbau und das Verhalten der nach diesem Muster gebauten Objekte. Aber die Klasse sagt nichts über die Inhalte, die Ausprägung des Objektes. Somit gehören die Tabellen einer Arbeitsmappe zu der nach dem Muster der *Worksheet*-Klasse gebauten *Worksheets*-Auflistung, aber sie sind nicht identisch, da sie unterschiedliche Inhalte aufweisen.

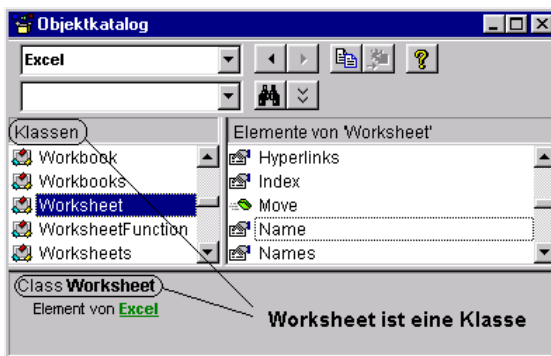


Abbildung 5.8: Der Klassenbegriff im Objektkatalog

Einige typische Auflistungen, die Objekte derselben Klasse enthalten, sehen Sie in der folgenden Übersicht:

Auflistung	Inhalt
Cells	alle Zellen einer Arbeitsmappe oder eines Zellbereichs
ChartObjects	alle Diagramme in einer Tabelle
Charts	alle Diagrammblätter einer Arbeitsmappe
Comments	alle Zellkommentare einer Tabelle
Workbooks	alle derzeit geöffneten Arbeitsmappen
Worksheets	alle Tabellenblätter einer Arbeitsmappe

Es gibt aber noch ein paar Auflistungen, die Elemente verschiedener Klassen enthalten können:

Auflistung	Inhalt
Controls	alle Steuerelemente in einem Container
Shapes	alle graphischen Objekte in oder besser auf einer Tabelle
Sheets	alle Diagramm- und Tabellenblätter einer Arbeitsmappe

Die Aufgabe von Auflistungen ist die Verwaltung ihrer Elemente und die Rückgabe eines Mitglieds auf Anforderung. Gerne würde ich jetzt schreiben: Alle Auflistungen sind mit einem identischen Stamm an Methoden und Eigenschaften ausgestattet. Aber leider stimmt das so nicht, denn es gibt aus der `Count`-Eigenschaft (deren Name eher an eine Methode erinnert) und dem `Item`-Dingsbums, das mal als Eigenschaft und mal als Methode einherkommt, keinen Stamm an Eigenschaften und Methoden oder auch Ereignissen, obwohl es eine Reihe sehr interessanter Aufsätze zu diesem Thema in der MSDN gibt, in denen ein festes Regelwerk für den Aufbau von Auflistungen entworfen wird, an die wir (nicht bei Microsoft arbeitenden Entwickler) uns halten sollen.

Count-Eigenschaft

Diese `Count`-Eigenschaft, die nur gelesen werden kann, gibt die Anzahl der in der Auflistung enthaltenen Elemente zurück:

```
lngSheetNumber = wbkX.Worksheets.Count
```

Item-Methode

Die `Item`-Methode gibt ein Element aus der Auflistung zurück:

```
Set shtAct = ThisWorkbook.Worksheets.Item("GuV 2000")
```

Dieser Ausdruck gibt das Tabellenobjekt der Tabelle GuV 2000 zurück, die natürlich in der `Worksheets`-Auflistung enthalten sein muss. Der folgende Ausdruck gibt die zweite Tabelle des Workbooks zurück.

```
Set shtAct = ThisWorkbook.Worksheets.Item(2)
```

Die `Item`-Methode ist zweifellos die häufigste Methode im Umgang mit Auflistungen. Dass sie Ihnen womöglich kaum begegnen wird, liegt daran, dass sie als Standardmethode weggelassen werden kann:

```
Set shtAct = ThisWorkbook.Worksheets("GuV 2000")
Set shtAct = ThisWorkbook.Worksheets(2)
```

Jedes Objekt kann über eine Standardeigenschaft oder Standardmethode verfügen, muss aber nicht. Im Grunde ist dies, abgesehen von der `Item`-Methode, mehr Fluch als Segen, denn sie verleitet dazu, Eigenschaften im Code wie Objekte aussehen zu lassen. Und das geht so weit, bis der Unterschied im Kopf auch nicht mehr erkennbar ist.

NewEnum-Methode

Diese nicht sichtbare Methode ist dafür verantwortlich, dass man mittels `For-Each`-Schleifen sich alle Objekte der Auflistung einzeln und nacheinander zurückgeben lassen kann:

```
For Each shtAct in ThisWorkbook.Worksheets
    If shtAct.Name = ...
Next
```

Den Ausdruck `NewEnum` können Sie nun getrost wieder vergessen.

Hinzufügen und Entfernen von Objekten in Auflistungen

Hierzu wird Ihnen im Excel-Objektmodell alles begegnet, was man sich so ausdenken kann. Die meisten halten sich jedoch an das Microsoft-eigene Regelwerk und erlauben uns zumindest das Hinzufügen einzelner Objekte zur Auflistung nach folgendem Muster:

```
ThisWorkbook.Worksheets.Add
Workbooks.Add
```

Das Entfernen einer Tabelle wird beispielsweise so vorgenommen:

```
ThisWorkbook.Worksheets(3).Delete
```

In Kapitel 6 – Zugriff auf Excel-Objekte werden diese Methoden im Zusammenhang mit dem jeweiligen Objekt vorgestellt.

Im folgenden Abschnitt 5.3 werden wir uns den Zugriff auf Zellen aus einem anderen Blickwinkel vornehmen.

5.2.4 Kapselung, Polymorphie und Vererbung

Ein Hinweis zum Genuss dieses Abschnitts. Die Ausführungen zum Thema Vererbung sind eigentlich mehr ein Plädoyer und haben als solches einiges mit Meinung zu tun. Sie können 5.2.4 komplett übergehen, ohne einen größeren Schaden zu erleiden.

Aber wenn von Objekten die Rede ist, dürfen diese drei Begriffe nicht fehlen. Sie stellen die drei zentralen Charakteristiken der Objektorientierung dar.

Kapselung (Encapsulation)

Ein Worksheet-Objekt ist als Objekt ein klassischer Vertreter einer gekapselten Software, denn zur Beeinflussung der Daten oder des Verhaltens des Objekts stehen uns *nur* Eigenschaften und Methoden zur Verfügung. Diese stellen eine Art Kanal dar, durch den unsere Daten hin und her transportiert werden. Hierbei werden sie auf Gültigkeit und Typkonformität geprüft.

Mit der Kapselung ist aber nicht gemeint, dass die Excel-Objekte sauber gekapselt sind. Vielmehr verbirgt sich dahinter die Fähigkeit von VBA seit Version 5 (Office 97), selbst Klassen zu schreiben und somit unsere Programmkomponenten zu kapseln. Die Kapselung von Code durch Klassen und deren klare Schnittstelle ist eine wesentliche Komponente stabilen Codes. In den Kapiteln 10 und 11 werden wir uns damit auseinander setzen.

Hier können wir getrost 99 Punkte vergeben, denn mittlerweile funktionieren auch Enumerationen. Die Deklaration einer Prozedur als Standard-eigenschaft oder Standardmethode würde den letzten Punkt erhalten.

Polymorphie (Polymorphism)

Hinter diesem Begriff, der mit Vielgestaltigkeit übersetzt werden kann, verbirgt sich die Technik für dem Grunde nach identische Vorgänge. Wenn ein Element zu einer Auflistung hinzugefügt wird, so wird die betreffende Methode einheitlich `Add` genannt. Statt

```
Workbooks.AddWorkbook
```

und

```
Worksheets.AddWorksheet
```

heißt es einheitlich

```
Worksheets.Add
```

```
Workbooks.Add
```

Hinter dem einheitlichen Ausdruck `Add` verbirgt sich also eine vielgestaltige Implementierung, die vom jeweiligen Objekttyp abhängt.

Bezogen auf die Excel-Objekte würde ich vorsichtige 90 Punkte vergeben. Und was unsere Prozedurnamen angeht, so haben wir im Prinzip freie Hand, da wir als Methoden- und Eigenschaftsnamen fast alles vergeben können, wonach uns der Sinn steht. Also 100 Punkte.

Vererbung (Inheritance)

Vererbung lässt sich am besten an einem Beispiel erklären. Nehmen wir an, wir wollten eine Applikation schreiben, die uns die Projektierung und Kalkulation eines modularen, auf Komponenten basierenden Systems ermöglicht, zum Beispiel ein Antrieb. Ein Antrieb besteht immer aus dem eigentlichen Motor und einem Getriebeteil und erscheint je nach Getriebeteil in den Basistypen Stirnrad-, Flach-, Kegelrad- und Schneckengetriebe. Es macht Sinn, für jeden dieser Basistypen eine eigene Klasse zu schreiben, die mit ihren Methoden und Eigenschaften die Besonderheiten der Basistypen repräsentiert. Aber in jedem Antriebsbasistyp ist ein Motor enthalten.

Bei einer Implementierungsvererbung würde nun eine zentrale Motorklasse geschrieben werden, die dann in jeder Getriebebasisklasse verwendet werden kann, so, als wäre der Code der Motorklasse in jeder Getriebebasisklasse enthalten.

Und das funktioniert in VBA nicht, wir sollten vielleicht sagen, noch nicht. Das werden Ihnen zumindest alle Puristen sagen und Sie mit einem mehr oder weniger mitleidigen Blick bedenken. Also null Punkte. Oder doch nicht?

Unter dem Begriff Aggregation steht uns ein Verfahren zur Verfügung, das uns in die Lage versetzt, trotzdem auf die Vorzüge der Vererbung nicht verzichten zu müssen. Der tiefere Sinn dieser Vererbung ist, dass man den Code für den Motor nur einmal schreiben muss und in jedem Antrieb damit arbeiten kann. Und vor allem, dass wir bei einer Änderung der Motorklasse sofort in allen diese Klasse beerbenden Codekomponenten ohne die geringste Änderung weiterarbeiten können. Und genau das können wir, indem wir in den partizipierenden Codekomponenten eine Instanz dieser Klasse erzeugen.

Aggregation

Doch nun wieder zurück zur Erde.

5.3 Zugriff auf Zellen

Um das Thema Objekte und deren Zugriff nicht völlig im luftleeren Raum zu diskutieren, haben wir natürlich ein wenig mit Excel-Objekten experimentiert. Hierbei gelang uns der eine oder andere Einblick in das Excel-Objektmodell.

Am Beispiel von Zugriffen auf Zellen wollen wir die beteiligten Objekte und Auflistungen ein wenig näher unter die Lupe nehmen. Abbildung 5.9 zeigt die Stellung des Range-Objekts innerhalb des Excel-Objektmodells.

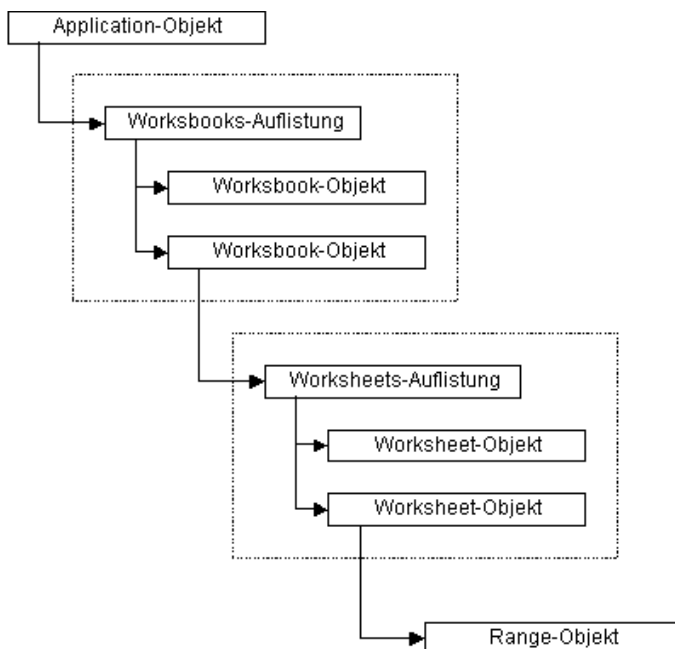


Abbildung 5.9: Range-Objekt, Stellung innerhalb des Objektmodells

5.3.1 Der Zugriff auf die Objekte

Das **Range-Objekt** repräsentiert einen Zellbereich; im einfachsten Fall eine einzelne Zelle, kann aber auch alle Zellen eines Tabellenblattes umfassen. Innerhalb des Objektmodells von Excel nimmt es folgende Position ein:

Das **Application-Objekt** stellt Excel selbst dar. Ihm zugeordnete Eigenschaften und Methoden sind globaler Natur, etwa die Tabellenfunktionen (=SUMME() etc.) oder Teile der Optionen, die im Extras-Menü anzutreffen sind, wie etwa der Berechnungsmodus u. a.

In Form der **Workbooks-Auflistung** hat das **Application-Objekt** Zugriff auf alle geöffneten Arbeitsmappen. Die **Workbooks-Auflistung** verfügt auch über eine **Open-Methode**, mittels der noch nicht geöffnete Arbeitsmappen zu der Auflistung hinzugefügt werden können. Neben der **Open-Methode**, die nur bei einigen Auflistungen implementiert ist, ist die **Add-Methode** im Prinzip bei allen Auflistungen implementiert. Durch die **Add-Methode** wird meist ein Standardobjekt erzeugt, beispielsweise ein leeres **Workbook** oder ein leeres **Worksheet**.

Der Zugriff auf ein einzelnes Objekt innerhalb einer Auflistung ist über die `Item`-Methode möglich, die jedoch in der Praxis kaum anzutreffen ist, da sie als Standardeigenschaft einer jeden Auflistung einfach weggelassen wird, wie wir noch sehen werden.

Der folgende Ausdruck, der aus der Liste der geöffneten Arbeitsmappen die »GuV 1998.xls« auswählt,

```
Application.Workbooks.Item("GuV 1998.xls")
```

wird schlicht und ergreifend zu

```
Application.Workbooks("GuV 1998.xls")
```

Beachten Sie bitte, dass Sie beim Öffnen der Datei den vollständigen Pfad angeben müssen, wohingegen beim Zugriff auf die geöffnete Arbeitsmappe nur der reine Dateiname angegeben wird – so, wie er im Menü Fenster zu sehen ist.

```
Application.Workbooks.Open("D:\Controlling\GuV 1998.xls")
```

Wie dem auch sei, nach jedem dieser Ausdrücke haben Sie ein `Workbook`-Objekt in der Hand. Der nun anstehende Zugriff auf eines der existierenden Tabellenblätter erfolgt durch die `Item`-Methode

```
...Worksheets.Item("Kosten")
```

oder ohne `Item`

```
...Worksheets("Kosten")
```

Nun fehlt nur noch der Zugriff auf die Zelle selbst, also auf das `Range`-Objekt. Hier nun der Zugriff auf die Zelle A1 vollständig:

```
Application.Workbooks("GuV 1998.xls"). _  
Worksheets("Kosten").Range("A1")
```

Natürlich müssen Sie nicht bei jedem Zugriff auf eine Zelle solch einen Roman schreiben. Hier werden uns Objektvariablen noch gute Dienste leisten. Doch bevor wir uns diesem Punkt nähern, soll ein wichtiger Aspekt noch beleuchtet werden, der Erleichterung oder aber Verdruss bringt.

5.3.2 Die Kunst des Weglassens

In VBA dürfen Sie viel weglassen. Sie sollten sich aber dessen bewusst sein, dass VBA dann gelegentlich anstelle der weggelassenen Teile (Objekte) Annahmen setzt. Und genau die sollten Sie kennen.

Generell dürfen Sie in der oben besprochenen Objektkette alles weglassen, was eindeutig ist. Zum Beispiel das `Application`-Objekt, weil die darunter liegende `Workbooks`-Auflistung nur einen Parent kennt: eben das `Application`-Objekt. Darüber hinaus regiert der Leichtsinn, denn der Ausdruck

```
Worksheets("Kosten").Range("A1")
```

wird von VBA so interpretiert:

```
Application.ActiveWorkbook.Worksheets("Kosten").Range("A1")
```

Und `ActiveWorkbook` ist immer die Arbeitsmappe, die gerade den Fokus hat. Es wird in fast allen Fällen gut gehen, aber tun Sie sich den Gefallen und schreiben Sie das dazugehörige `Workbook` davor. Es erleichtert zudem das Lesen eines Programmes, das mit mehreren Dateien umgeht. Denn Ihr sich schnell entwickelnder VBA-Interpreter irgendwo auf der Großhirnrinde wird bei einer solch unvollständigen Zeile – wenn auch kaum merklich – ins Stocken geraten und überlegen, zu welchem `Workbook` dieses `Worksheet` "Kosten" gehört.

Brandgefährlich hingegen sind Ausdrücke, die sogar das `Worksheet` einsparen, denn dies wird interpretiert als

```
Application.ActiveWorkbook.ActiveSheet.Range("A1")
```

Und dieses `ActiveSheet` ist wie ein Stück Seife, welches nur durch Vorstellen von

```
Worksheets("Kosten").Select
```

gefahrlos verwendet werden kann. Und genau diese Verwendung von `Select`-Methoden in VBA ist umständlich und zeugt von schlechtem Stil. Denn genau hier setzen die Objektvariablen an, deren Einsatz die vollen Möglichkeiten des Excel-Objektmodells erschließen und ein Programm erst übersichtlich machen.

5.3.3 Der Einsatz von Objektvariablen

In unserem Beispiel setzten wir vier verschiedene Objekttypen ein: `Application`, `Workbook`, `Worksheet` und `Range`, wobei die letzten drei Typen, allen voran `Range`, durch Variablen repräsentiert werden.

Präfixbildung

Nun ist es in größeren Applikation mitunter nervtötend, für alle die nötigen Variablen auch noch Namen mit Wiedererkennungswert und gebotener Kürze zu erfinden. Doch hier gibt es Abhilfe in Form sinnvoller Präfixbildung und einiger leicht erlernbarer Regeln.

Zur Präfixbildung haben sich folgende Konventionen als sinnvoll erwiesen:

- ▼ wbk für Workbook
- ▼ sht für Worksheet
- ▼ rng für Range

Dem Präfix bei Workbook und bei Worksheet folgen sprechende Kurzformen des jeweiligen Objektnamens, also etwa wbkGuV für das Workbook »GuV 1998.xls« und shtKost oder shtKosten für das Tabellenblatt »Kosten«. Sollten Sie gleichzeitig auch noch in der GuV des Vorjahres operieren, müssen auch für die nunmehr zwei Workbooks eindeutige Namen vergeben werden, etwa wbkGuV1998 oder wbkGuV98 je nach ästhetischer Grundstimmung. Wir werden noch ein Beispiel für eine überaus sinnvolle Klasse aufgreifen, die das Öffnen eines Workbooks mittels einer Methode übernimmt und im Falle eines Fehlers eine sinnvolle Fehlermeldung in Form einer Eigenschaft dieser Klasse produziert.

Doch zurück zu unserem Range-Objekt. Eine Regel vorweg: Ein Range-Objekt stellt einen Pflock dar, der in eine Tabelle geschlagen wird. Ausgehend von diesem Pflock erfolgen alle weiteren Zugriffe auf beliebige Zellen dieser Tabelle über die Methode `Cells(iRow, iColumn)`, die uns noch eine Weile begleiten wird.

Da wir also bis auf sehr wenige und wohlbegründete Ausnahmen nur ein Range-Objekt je Tabelle benötigen, ist eigentlich schon klar, wie die Objektvariable heißen muss. Klar, sie heißt so, wie die Variable des dazugehörigen Worksheets heißen würde, nur der Präfix `rng` tritt an Stelle von `sht`: `rngKost` bzw. `rngKosten`. Wenn Sie irgendwo in Ihrem Code eine Variable des Namens `Kosten` sehen, muss Ihr interner VBA-Interpreter wieder interpretieren, also überlegen, ob es eine Variable für ein Worksheet oder einen Range ist. VBA ist so komplex, dass es für das Hirn lohnendere Aufgaben gibt, als über Variablentypen nachzudenken.

Nehmen wir an, unser Programm befände sich bereits in dem Workbook »GuV 1998.xls«. Aus Sicht des Programmes wird dieses Workbook als `ThisWorkbook` referenziert. Wenn wir nun im weiteren Programmverlauf Variablen für Worksheet und Range benötigten, könnte die Erzeugung und Zuweisung der Variablen folgendermaßen aussehen:

```
Dim shtKost As Worksheet
```

```
Dim rngKost As Range
```

```
Set shtKost = ThisWorkbook.Worksheets("Kosten")
```

```
Set rngKost = shtKost.Range("A1")
```

Wie Sie sehen, können wir bei der Zuweisung des Range-Objekts das bereits existierende Worksheet-Objekt verwenden.

5.3.4 Der Pflock in Zelle A1

Es hindert Sie niemand daran, ein Range-Objekt irgendwo in einer Tabelle zu positionieren. Doch es gibt ein Argument, welches zwingend für die Zelle A1 spricht, denn nur dadurch sind aufgrund einer Besonderheit der Methode `Cells(iRow, iColumn)` diese beiden Argumente `iRow` und `iColumn` identisch mit den Werten der `Column`- und `Row`-Eigenschaften der ganzen Tabelle. Im Klartext: Die relativen und absoluten Zeilen- und Spaltenzeiger sind identisch und das vereinfacht die Entwicklung von Programmen mitunter erheblich.



Zugriff auf Excel-Objekte

6

Kapitelüberblick

6.1	Das Range-Objekt	197
6.1.1	Eigenschaften und Methoden im Überblick	198
6.1.2	Eigenschaften und Methoden unter der Lupe	201
6.2	Die Worksheets-Auflistung	237
6.3	Das Worksheet-Objekt	237
6.3.1	Eigenschaften und Methoden im Überblick	238
6.3.2	Eigenschaften und Methoden unter der Lupe	239
6.3.3	Ereignisse	251
6.4	Die Workbooks-Auflistung	262
6.4.1	Eigenschaften und Methoden im Überblick	262
6.4.2	Eigenschaften und Methoden unter der Lupe	262
6.5	Das Workbook-Objekt	270
6.5.1	Eigenschaften und Methoden im Überblick	270
6.5.2	Eigenschaften und Methoden unter der Lupe	270
6.5.3	Ereignisse	277
6.6	Das Application-Objekt	281
6.6.1	Eigenschaften und Methoden im Überblick	281
6.6.2	Eigenschaften und Methoden unter der Lupe	282
6.6.3	Ereignisse	291
6.6.4	Tabellenfunktionen in VBA	292
6.7	CommandBars und CommandBarControls	294
6.7.1	CommandBar erzeugen	296
6.7.2	FacelDs	297
6.7.3	Fazit	300

Quält man nun den Leser durch eine x-seitige Behandlung der wichtigsten Objekte nebst Eigenschaften und Methoden oder unterbreitet man stattdessen den in letzter Zeit so arg strapazierten *intuitiven* Einstieg? Diese Frage beschäftigt mich nicht erst seit der Arbeit an diesem Buch.

Ein Argument spricht jedoch für den klassischen Einstieg der Behandlung der wichtigsten Objekte nebst ihren Eigenschaften und Methoden:



Der professionelle Umgang mit den Excel-Objekten erfordert eine solide Kenntnis der Objekte.

Wenn Sie sich mit der Elektronik auseinander setzen, werden Sie auch nicht damit beginnen, einen HiFi-Verstärker zu bauen. Es wird losgehen mit der Physik von PN-Übergängen, mit dem guten alten Fermi und seinen Bändern, dann kommen irgendwann Dioden und Transistoren. Und wenn Sie sich die benötigten Komponenten hinreichend verinnerlicht haben, beginnen Sie mit dem Bau ganzer Schaltungen. Und mit Visual Basic und Excel haben Sie es mit Themen zu tun, die in punkto Komplexität an die Anwendungselektronik heranreichen.

Excel ist ein Tabellenkalkulationsprogramm und als solches sicherlich das Beste, was man für Geld kriegen kann. Tabellenkalkulation wiederum riecht stark nach Zahlen. Was liegt also näher, als Excel auch automatisiert zur Verarbeitung und Analyse von Zahlen heranzuziehen.

Natürlich ist das nicht alles, was Excel kann. On Top von Tabellen lassen sich beliebige ActiveX Komponenten platzieren – solche, die Excel selbst produziert, aber auch andere. Sieht man einmal von Diagrammen ab, so lässt sich allerdings über Sinn und Zweck eines solchen Einsatzes von Excel trefflich streiten.

Dieser Diskussion möchte ich mich hier fast völlig entziehen und mich auf das beschränken, was unbestreitbar die Kernaufgabe von Excel darstellt: nämlich das Arbeiten mit Zahlen. Und gerade hier wuchert Excel mit seinen Pfunden.

Dieses Kapitel wird sich gründlich mit den Objekten innerhalb des Excel-Objektmodells auseinander setzen, die zum 1x1 der Zellmanipulation gehören. Das beinhaltet die Objekte

- ▼ Application
- ▼ Workbook
- ▼ Worksheet
- ▼ Range

Im Zusammenhang mit diesen Objekten werden natürlich auch einige Randthemen beleuchtet, ohne dass ihnen allerdings die Bedeutung dieser vier Objekte zuteil wird. So werden auch einige Unterelemente der Objekte `Worksheet` und `Range` hinreichend gewürdigt. Aber das Hauptaugenmerk liegt auf diesen vier Objekten.

6.1 Das Range-Objekt

In diesem Kapitel stellen wir die wichtigsten Eigenschaften und Methoden des `Range`-Objekts vor. Hierbei handelt es sich allein um über 150 Elemente, zählt man jedoch noch die dem `Range`-Objekt untergeordneten Objekte und Auflistungen und wiederum deren Eigenschaften und Methoden hinzu, so dürfte die Zahl von 200 überschritten sein. Alles in allem ein hinreichender Grund, um die Elemente, die man wirklich benötigt, näher zu beleuchten.

Tabellen, die wir bereits zur Entwicklungszeit zur Verfügung haben, lassen sich recht einfach über die Eigenschaften der Tabelle referenzieren. Wählen Sie hierzu in der Entwicklungsumgebung die betreffende Tabelle im Projektextplorer (hier Tabelle `Report`) aus und ändern Sie danach die Eigenschaft `(Name)` ab. Diese `(Name)`-Eigenschaft stellt den öffentlichen Objektnamen dar, unter dem Sie zukünftig das `Worksheet`-Objekt der Tabelle ansprechen können. Diese `(Name)`-Eigenschaft ist nicht identisch mit der `Name`-Eigenschaft, die den Namen des Tabellenblattes wiedergibt:

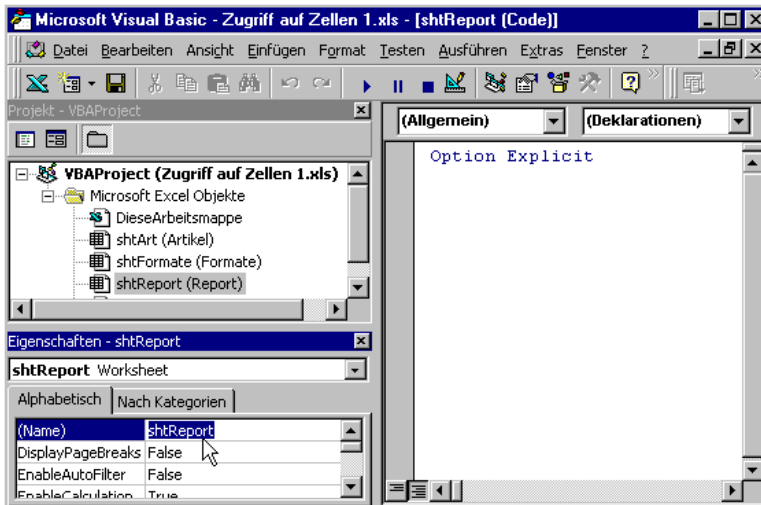


Abbildung 6.1: Erzeugen des Objekts `shtReport` über die Eigenschaften

6.1.1 Eigenschaften und Methoden im Überblick

Bisher besprochen wir die Rolle des `Range`-Objekts innerhalb des Excel-Objektmodells und den Einsatz von Objektvariablen. Eine Referenz auf die Zelle A1 der Tabelle `Kosten` wurde definiert als:

```
Dim rngRep as Range  
Set rngRep = Thisworkbook.Worksheets("Report").Range("A1")
```

Oder unter Verwendung des `Worksheet`-Objektes (siehe Abb. 6.1):

```
Set rngRep = shtRep.Range("A1")
```

Danach steht uns über diese Objektvariable `rngKost` und der nachgeordneten `Cells`-Auflistung das komplette Tabellenblatt mit seinen über 16 Millionen Zellen zur Verfügung. Der weitere Zugriff auf die Tabellenzeilen erfolgt zumeist über die `Cells`-Auflistung, die in drei Varianten den Zugriff entweder auf die Zellen innerhalb oder auch außerhalb des Ranges erlaubt. Die Zellen innerhalb und außerhalb eines Ranges – sprich Zellbereich – ergeben nun mal alle Zellen.

Mit dieser überaus wichtigen `Cells`-Auflistung werden wir uns nachfolgend noch auseinander setzen. Doch nicht nur mit ihr, sondern mit rund 40 weiteren Elementen, also Eigenschaften, Methoden und Auflistungen. Das `Range`-Objekt verfügt aber über mehr als 150 Elemente. Wieso sollen wir uns mit einem knappen Drittel davon begnügen? Die Beschränkung erfolgt aus rein pragmatischen Gründen, denn diese Auswahl reicht erfahrungsgemäß aus fast alles mit dem `Range`-Objekt zu veranstalten, was in der Praxis erforderlich ist.

Bevor wir uns den Elementen nun im Detail widmen, sollten wir noch einen Blick auf die Rückgabe riskieren, die in der nachfolgenden Tabelle mit einer eigenen Spalte versehen ist. Excel zeichnet sich wie kaum ein anderes Objektmodell durch mitunter recht lange Objektbegriffe aus, wenn ein `Range`-Objekt darin verwickelt ist. Das liegt darin begründet, dass einige Methoden, Eigenschaften und auch Auflistungen selbst wieder ein `Range`-Objekt erzeugen.

`rngKosten` sei ein `Range`-Objekt in Tabelle »Kosten« in Zelle A1. Möchten Sie davon ausgehend die Spalte C vollständig erfassen, so bietet sich folgende Konstruktion an:

```
rngKosten.Cells(1, 3).EntireColumn
```

Alle drei Elemente stellen einen `Range` dar:

```
rngKosten
```

die Zelle A1,

```
rngKosten.Cells(1, 3)
```

die Zelle C1 und der vollständige Ausdruck

```
rngKosten.Cells(1, 3).EntireColumn
```

eben die Spalte C. Also stehen Ihnen nach jedem Teil dieses obigen Ausdrucks erneut alle Elemente des Range-Objekts zur Verfügung.

Dies ist im Übrigen kein konstruiertes Beispiel, vielmehr etwas ganz Normales. Solche langen, teils auch verschachtelten Ausdrücke stellen vielleicht die größte Hürde dar, die Ihnen im Umgang mit dem Range-Objekt oder auch Excel im Allgemeinen bevorsteht. Also mein Tipp: Wenn Sie sich einzelne Elemente der folgenden Tabelle nebst ihren Parametern einprägen, so vergessen Sie bitte nicht die Rückgabewerte. Sie werden sich vielleicht fragen, wozu der Aufwand, da die Entwicklungsumgebung doch über eine Option verfügt, die da lautet »Elemente automatisch auflisten«. Die zugrunde liegende Typelibrary verliert leider gelegentlich selbst den Überblick und das beliebte kleine Fenster bleibt aus. Auch hier gilt wieder, dass es gut ist, wenn Automatismen zur Verfügung stehen, aber noch besser, wenn man die Regeln selbst beherrscht.

Die folgende Tabelle zeigt die behandelten Elemente im Überblick. Einige sind mit einem Satz hinreichend beschrieben, andere erfordern eine intensivere Behandlung.

	Element	Beschreibung	Rückgabe
M	Activate	macht die Zelle zur aktiven Zelle	
M	AddComment	fügt einen Kommentar zur aktiven Zelle hinzu	Comment
E	Address	gibt die Adresse des Ranges zurück	String
M	AutoFilter	erzeugt einen Autofilter im Range	
M	AutoFit	wendet die optimale Spaltenbreite an	
A	Borders(i)	gibt ein Border-Objekt (Rahmen) zurück	Border
M	BordersAround	erzeugt einen Rahmen um den Range herum	
A	Cells	Collection mit allen Zellen des Tabellenblattes	Range
E	Characters	erzeugt ein Characters-Objekt	Characters
M	Clear	löscht Inhalte, Formate und Kommentare	
M	ClearComments	löscht die Kommentare	
M	ClearContents	löscht die Inhalte	
M	ClearFormats	löscht die Formate	
E	Column	absolute Spaltennummer	Long

	Element	Beschreibung	Rückgabe
A	Columns	Collection mit allen im Range enthaltenen Spalten	Range
E	ColumnWidth	Spaltenbreite	Double
E	Comment	Kommentar der Zelle	Comment
M	Copy	kopiert den Range in Zwischenablage oder zu dem anzugebenden Ziel	
M	CurrentRegion	erzeugt aktuellen Bereich, der von leeren Zellen oder dem Tabellenrand umgeben ist	Range
M	Delete	entfernt die Zellen (üblicherweise Zeile oder Spalte)	
E	EntireColumn	erzeugt die ganze Spalte	Range
E	EntireRow	erzeugt die ganze Zeile	Range
M	Find	gibt die Zelle zurück, in der »what« gefunden wurde	Range
E	Font	gibt das Font-Objekt zurück	Font
E	FormulaLocal	enthält die Formel der Zelle	String
E	FormulaR1C1Local	enthält die Formel in Z1S1-Schreibweise	String
E	HasFormula	True, wenn Zelle Formel enthält et vice versa	Boolean
E	Height	Höhe der Zelle bzw. Zeile in Punkt	Double
E	Hidden	True, wenn Zelle (Spalte oder Zeile) nicht sichtbar	Boolean
E	Interior	gibt das Interior-Objekt (Hintergrund) zurück	Interior
E	Left	Position des linken Zellrandes in Punkt	Double
E	Locked	legt fest, ob die Zelle nach Aktivieren des Blatt-schutzes gesperrt sein soll	Boolean
E	MergeArea	gibt den kompletten Range eines verbundenen Zellbereichs zurück	Range
E	MergeCells	legt fest, ob die Zellen des voranstehenden Ranges verbunden sein sollen oder nicht	Boolean
E	NumberFormat	Zahlenformat des Ranges	String
M	Offset	erzeugt einen um rows Zeilen und cols Spalten verschobenen Range gleicher Größe	Range
E	Parent	gibt das übergeordnete Worksheet zurück	Worksheet
M	PasteSpecial	fügt Inhalt der Zwischenablage ein	
M	Range(cell1, cell2)	erzeugt einen Range zwischen Cell1 und Cell2	Range
M	Resize(rows, cols)	redimensioniert den Range mit rows Zeilen und cols Spalten	Range
E	Row	absolute Zeilennummer	Long

	Element	Beschreibung	Rückgabe
A	Rows	Collection mit allen im Range enthaltenen Zeilen	Range
M	Sort	sortiert den Range	
M	SpecialCells	eine Untermenge des angegebenen Ranges, z.B. leere oder sichtbare Zellen	Range
M	TextToColumns	wandelt ein- in mehrspaltige Werte um	
E	Top	Position des oberen Zellrandes in Punkt	Double
E	Validation	gibt das dem Range zugeordnete Validation-Objekt zurück	Validation
E	Value	Wert der Zelle	Variant
E	Width	Breite der Zelle (Spalte) in Punkt	Double

M = Methode, E = Eigenschaft, A = Auflistung

6.1.2 Eigenschaften und Methoden unter der Lupe

Activate-Methode

Die `Activate`-Methode macht die voranstehende Zelle zur aktiven Zelle. Diese ist sichtbar und verfügt über den für die aktive Zelle eines Tabellenblattes (um genau zu sein: des Fensters) typischen schwarzen Rand.

Sind mehrere Zellen per Maus oder Tastatur selektiert worden, so ist dennoch nur eine Zelle die aktive Zelle. Der Hintergrund selektierter Zellen wird in der Komplementärfarbe dargestellt, die aktive Zelle als Teil der selektierten Zellen erscheint in der Ursprungsfarbe (oder, wenn Sie so wollen, wird die Zellfarbe zweimal komplementiert). Den Zugriff auf die selektierten Zellen wird zusammen mit der `Cells`-Auflistung dargestellt.

In schlechten Programmen kommt die `Activate`-Methode häufig vor. In guten Programmen hingegen zumeist als eine der letzten Zeilen, um eine Zelle im oberen Teil einer Tabelle zur aktiven Zelle zu machen.

Nehmen wir an, Sie haben eine Umsatzauswertung aufgebaut, wie sie in Abb. 6.2 zu sehen ist. Die zuletzt bearbeitete Zelle ist Zeile 38 gewesen. Würden Sie Ihr Programm nun beenden, wären möglicherweise die ersten Datenzeilen der Tabelle ausgeblendet und die aktive Zelle wäre I38. Es macht jedoch Sinn, dem Anwender die obersten Zeilen der Tabelle zu präsentieren:

```
...
Next
shtReport.Range("C6").Activate
End Sub
```

Zugriff auf Zellen 1.xls									
	A	B	C	D	E	F	G	H	I
1	Niederlassung 1						XYZ		
2	Oktober 1998			Warengruppenumsätze			GmbH & Co. KG		
3									
4	Tag	Wochen-tag	Warengruppen prozentual			Warengruppen absolut			
			1	2	3	1	2	3	Summe
6	1	Do	35,28 %	25,44 %	39,27 %	72.665,95	52.400,21	80.888,47	205.954,63
7	2	Fr	30,16 %	25,64 %	44,20 %	62.263,82	52.921,83	91.233,62	206.419,27
8	3	Sa	31,77 %	25,96 %	42,27 %	66.457,07	54.297,74	88.398,89	209.153,70
9	Summe:		32,40 %	25,68 %	41,92 %	201.386,64	159.619,78	260.520,98	621.527,60
10	5	Mo	32,71 %	23,31 %	43,98 %	69.741,24	49.696,28	93.774,36	213.211,88
11	6	Di	31,10 %	22,24 %	46,66 %	62.193,44	44.469,69	93.321,41	199.984,54
12	7	Mi	33,88 %	20,75 %	45,37 %	73.780,95	45.184,90	98.819,86	217.785,71
13	8	Do	33,90 %	24,39 %	41,72 %	74.079,37	53.294,49	91.174,34	218.548,20
14	9	Fr	34,70 %	21,21 %	44,09 %	67.392,05	41.189,90	85.610,91	194.192,86
15	10	Sa	32,18 %	24,76 %	43,06 %	69.004,50	53.083,81	92.313,12	214.401,43
16	Summe:		33,08 %	22,81 %	44,11 %	416.191,55	286.919,07	555.014,00	1.258.124,62
17	12	Mo	33,69 %	23,70 %	42,61 %	71.411,20	50.236,69	90.311,75	211.959,64
18	13	Di	37,61 %	22,55 %	39,83 %	75.921,31	45.525,62	80.408,24	201.855,17
19	14	Mi	34,78 %	25,06 %	40,16 %	73.553,92	52.980,91	84.922,73	211.457,56
20	15	Do	34,23 %	24,51 %	41,27 %	70.333,94	50.366,93	84.805,34	205.496,21

Abbildung 6.2: Die Activate-Methode aktiviert abschließend die Zelle C6.

AddComment-Methode

Die AddComment-Methode erzeugt ein der Zelle zugeordnetes Comment-Objekt, also ein Zellkommentar. Als eigenständiges Objekt verfügt Comment natürlich über Eigenschaften und Methoden. Hiervon verwende ich nur die Text-Methode, die den Inhalt des Comment-Objekts, also den Kommentar, repräsentiert. Zusammen mit der AddComment-Methode und der ClearComments-Methode, die dem Range-Objekt zugeordnet sind, reicht das aus.

Als Beispiel hierzu bietet sich ein Auszug aus der EURO-Konvertierung an, die im Kapitel Add-Ins näher behandelt wird. Hier nur so viel: Um eine in EURO konvertierte Zelle zu kennzeichnen, wird der betreffenden Zelle ein Kommentar hinzugefügt, aus dem hervorgeht, dass der Zellwert in EURO konvertiert wurde, von wem und wann:

```
...
strComment = "EURO-konvertiert" & vbCrLf & _
Application.UserName & vbCrLf & Now()
rngAct.AddComment strComment
rngAct.Value = rngAct.Value / sngKurs
```

rngAct beinhaltet jeweils eine der selektierten Zellen. Der Wechselkurs ist als Konstante sngKurs angenommen worden.

Address-Eigenschaft

Die Address-Eigenschaft ohne weitere Parametrierung gibt die Adresse der aktiven Zelle eines Ranges als String zurück.

Sie wird beispielsweise in Ereignisprozeduren des `Worksheet`-Objekts eingesetzt, die ein `Range`-Objekt übergeben. Ein Beispiel:

In einer Tabellenvorlage eines Steuerbüros soll in Zelle C7 die Mandantennummer eingetragen werden. Sobald dies erfolgt ist, soll das Programm aus einer Access-Datenbank den dazugehörigen Mandantennamen, die Steuernummer und den Namen des zuständigen Finanzamtes ermitteln und in den Tabellenkopf eintragen.

```
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
    If Target.Address <> "$C$7" Then Exit Sub
    ... Anweisungen
End Sub
```

In diesem Beispiel werden die Anweisungen nur ausgeführt, wenn die das `Change`-Ereignis auslösende Änderung in Zelle C7 erfolgte; andernfalls wird über `Exit Sub` die Prozedur verlassen.

Autofilter-Methode

Diese Methode filtert die Daten an Ort und Stelle, so, als würde man einen `AutoFilter` in einer Tabellenspalte manuell konfigurieren. Sie würden jedoch kaum diese Methode einsetzen, um einen `AutoFilter` zu aktivieren. Interessant wird die Sache, wenn man nun eine komplette Spalte per Programm so bearbeiten möchte, wie es uns die `TEILERGEBNIS()`-Tabellenfunktion ermöglicht, denn hier stehen uns plötzlich Teile einer Spalte zur Verfügung, nämlich die sichtbaren Teile (Zellen). Vergleicht man es mit dem Zugriff auf eine Datenbank, so spielt die `Autofilter`-Methode die Rolle der `WHERE`-Klausel in einer SQL-Abfrage.

Der klassische Fall ähnelt dem Anwendungsgebiet für eine Pivot-Tabelle. Eine Datentabelle (`rngDat`) enthält neben anderen auch die Spalten `Projekt`, `Zeitraum` und `Stunden`. In einer Projekttable (`rngPro`) werden nun die Stunden je Zeitraum und Projekt verdichtet. Hierin befinden sich in Zeile 1 beginnend mit B1 bereits die in Frage kommenden Zeiträume und in Spalte A ab A2 die Projekte.

```
Public Sub VerdichteProjekte()
    Dim rngDat As Range
    Dim iColDatZtr As Integer
    Dim iColDatPro As Integer
    Dim iColDatStd As Integer
    Dim rngPro As Range
    Dim iRowPro As Long
    Dim iColPro As Integer
    Dim lngZtr As Long
```

```

Dim strPro As String
Set rngDat = shtDat.Range("A1")
Set rngPro = shtPro.Range("A1")
iColDatZtr = rngDat.EntireRow.Find(what:="Zeitraum", _
    lookat:=xlWhole).Column
iColDatPro = rngDat.EntireRow.Find(what:"Projekt", _
    lookat:=xlWhole).Column
iColDatStd = rngDat.EntireRow.Find(what:"Stunden", _
    lookat:=xlWhole).Column
For iRowPro = 2 To rngPro.CurrentRegion.Rows.Count
    For iColPro = 2 To
rngPro.CurrentRegion.Columns.Count
        lngZtr = rngPro.Cells(1, iColPro).Value
        strPro = rngPro.Cells(iRowPro, 1).Value
        rngDat.AutoFilter field:=iColDatZtr, _
            Criterial:=lngZtr
        rngDat.AutoFilter field:=iColDatPro, _
            Criterial:=strPro
        rngPro.Cells(iRowPro, iColPro).Value = _
            Application.Sum(rngDat.Cells(1, iColDatStd). _
                EntireColumn.SpecialCells(xlVisible))
    Next
Next
rngDat.AutoFilter
End Sub

```

Zugegeben, das Beispiel hätte kompakter sein können. Andererseits geht Kürze vielfach einher mit Komplexität und Leichtsinn. Nehmen wir die drei Spaltenzeiger `iColZtr`, `iColPro` und `iColStd` in der Tabelle Daten. Natürlich könnten wir stattdessen in den beiden `AutoFilter`-Methoden feste Werte einsetzen, die genau so lange funktionieren, bis jemand die Tabelle umbaut. Wir werden im Verlauf der Behandlung der `Find`-Methode noch darauf eingehen, dass auch hier ein Stück Leichtsinn drinsteckt. Doch zurück zu unserem Beispiel.

Bis zu den `For`-`Next`-Schleifen passiert neben Variablendimensionierung und-zuweisung nichts Aufregendes. Die äußere von beiden erzeugt den Zeilenzeiger `iRowPro`, die innere den Spaltenzeiger `iColPro`, beide beginnend mit der ersten und bis zur letzten reichend. Danach werden die beiden Variablen für Zeitraum und Projektname zugewiesen.

Nun die Szene mit dem `AutoFilter`. Das Argument `Field` beinhaltet die laufende Nummer der Spalte innerhalb des aktuellen Bereichs (siehe `CurrentRegion`-Methode), die wir ja bereits den betreffenden Variablen zu-

gewiesen haben. Das Argument `Criteria1` erhält das alleinige Kriterium unseres Beispiels. Ebenso wie der `AutoFilter` im Menü Daten können wir auch ein zweites Kriterium (`Criteria2`) angeben und mit einem Operatoren (`Operator`) verknüpfen. Der folgende Ausdruck zeigt die Verwendung der `AutoFilter`-Methode unter Verwendung aller Argumente.

```
rngDat.AutoFilter field:=iColDatPro, Criteria1:=">=" & _
    strPro, Operator:=xlAnd, Criteria2:="<=" & strPro
```

Dieser Ausdruck ist übrigens funktional identisch mit dem in unserem Beispiel. Wie Sie sehen, sind den beiden `Criteria`-Argumente die Operatoren `>=` und `<=` als String verpackt vorangestellt. Der Operator `=` darf weggelassen werden, alle anderen müssen enthalten sein.

Einen Wermutstropfen gibt es noch: Die Verwendung vieler `AutoFilter`-Methoden wirkt sich merklich auf die Verarbeitungszeit aus. In diesem Falle ist der umgekehrte Weg anzuraten: zeilenweises Abarbeiten der Tabelle Daten und selektives Schreiben in Tabelle Projekte.

AutoFit-Methode

Diese Methode wendet die optimale Breite auf die betreffende Zeile oder Spalte an. Sie erwartet eine oder mehrere vollständige Zeilen oder Spalten als vorangehendes Range-Objekt:

```
For iColDat = 1 to rngDat.CurrentRegion.Columns.Count
    shtDat.Columns(iColDat).AutoFit
next
```

Borders-Eigenschaft

Die `Borders`-Eigenschaft, zumeist mit einer der Konstanten `xlEdgeTop`, `xlEdgeBottom`, `xlEdgeLeft` oder `xlEdgeRight` (es gibt noch 3 weitere), erzeugt ein `Border`-Objekt. Dieses `Border`-Objekt kann, nachdem es definiert ist, auf unterschiedliche Art und Weise angesprochen werden. Ich habe mich für die `LineStyle`-Eigenschaft entschieden:

```
rngDaten.Borders(xlEdgeBottom).LineStyle = xlContinuous
```

Übrigens, die alten Konstanten aus Excel 5 und 7 `xlTop`, `xlBottom`, `xlLeft` und `xlRight` funktionieren auch (noch).

Durch das Festlegen der `LineStyle`-Eigenschaft auf einen gültigen Wert wird das `Border`-Objekt sichtbar, wobei die Linienstärke automatisch zu `xlThin` wird. Benötigen Sie eine hiervon abweichende Linienstärke, so müssen Sie mit der `Weight`-Eigenschaft des `Border`-Objekts nachhelfen:

```
rngDaten.Borders(xlEdgeBottom).Weight = xlMedium
```

Die Erläuterungen zur Copy-Methode enthalten ein Beispiel für Zellformatierungen, die ohne das Border-Objekt auskommen.

BorderAround-Methode

Im Gegensatz zu der voranstehenden Borders-Eigenschaft, die in allen Zellen des Range-Objekts den gewählten Rahmen darstellt, wird die BorderAround-Methode verwendet, wenn um einen (aus zwangsläufig mehreren Zellen bestehenden) Range ein äußerer Rahmen erzeugt werden soll.

```
rngDaten.CurrentRegion.BorderAround _
    LineStyle:=xlContinuous, Weight:=xlThin,
```

Natürlich greift auch diese Methode auf die Implementierung der Borders-Auflistung zurück, wodurch uns auch dieselben Argumente zur Verfügung stehen, die da sind LineStyle, Weight, ColorIndex und Color. Hierbei ist ColorIndex ein Integer, der die betreffende Farbe wiedergibt, Color hingegen ein RGB-Array. In Excel 5 und 7 konnte man in den Palette-Schaltflächen für Zellhintergrund- und Schriftfarbe noch den Wert der gewünschten Farbkonstanten von oben links nach unten rechts abzählen; die erste war 1 und die letzte 56. In Excel 8 hat irgendein Witzbold bei Microsoft die Farben dieser Paletten sortiert ..., und die Konstanten können nur noch über die Hilfe oder das Direktfenster ermittelt werden. Das Merken der wichtigsten Werte (anhand einer älteren Excel-Version) ist durchaus von Nutzen, denn die Visual-Basic-Farbkonstanten (vbBlack, vbBlue, vbCyan, vbGreen, vbMagenta, vbRed, vbWhite und vbYellow) funktionieren hier bedauerlicherweise nicht.

Cells-Collection

Cells wird bei Microsoft kurzerhand als Eigenschaft geführt, worüber man trefflich streiten kann. Fest steht, dass eine Zelle ein Objekt ist und darüber hinaus Member (Mitglied) der Cells-Auflistung. Wie sonst könnte Cells.Count oder Cells.Item(i) oder kürzer Cells(i) implementiert sein? Da nun aber Item die Standardmethode der Cells-Auflistung ist, wird die Einordnung als Eigenschaft zunehmend unverständlich.

Cells: Der Cells-Auflistung sind wir im Rahmen der Behandlung der AutoFilter-Methode bereits begegnet:

```
For Each rngAct In rngSel.Cells
```

Cells(i): rngAct war ein Range-Objekt, das in dieser Schleife nacheinander die Rolle einer jeden in rngSel enthaltenen Zelle übernahm. Hier sehen wir auch wieder die ambivalente Rolle des Range-Objekts: einmal

Zelle, einmal `Cells`-Auflistung. Wir hätten das Beispiel auch anders realisieren können: Statt des `Range`-Objekts `rngAct` in einer `For-Each`-Schleife wäre eine `For-Next`-Schleife mit einer Laufvariablen `iCell` möglich, die uns sogar `rngAct` erspart hätte. Anstelle von `rngAct` müssten wir überall `rngSel.Cells(iCell)` einsetzen:

```
Public Sub DMtoEURO()
    Dim rngAct As Range
    Dim strComment As String
    Const sngKurs As Single = 1.95583
    ...
    For Each rngAct In ActiveWindow.Selection.Cells
        If IsNumeric(rngAct.Value) And Not _
            rngAct.Value = "" Then
            If Not rngAct.Comment Is Nothing Then
                strComment = rngAct.Comment.Text
            Else
                strComment = ""
            End If
            If InStr(1, strComment, "EURO-konvertiert") = 0 _
                Then
                If Not rngAct.Comment Is Nothing Then
                    rngAct.ClearComments
                End If
                strComment = "EURO-konvertiert" & vbCrLf & _
                    Application.UserName & vbCrLf & Now()
                rngAct.AddComment strComment
                rngAct.Value = rngAct.Value / sngKurs
            End If
        End If
    Next
End Sub
```

Es ist offensichtlich, dass `rngSel.Cells(iCell)` mehr Platz und Übersicht fordert als die Variante mit `rngAct` und der `For-Each`-Schleife.

Cells(iRow, iCol): Sie werden bemerkt haben, dass ich bezüglich Variablennamen recht eigen bin. Bequem ist vielleicht der bessere Ausdruck, denn ich habe kein Interesse, beim Einsatz eines `Range`-Objekts zu überlegen, wie die dazugehörigen Zeilen- und Spaltenzeiger heißen. Also lautet der Beschluss, dass dies vereinfacht werden muss. Der Präfix `i` steht für einen Zeiger, `Row` und `Col` für Zeilen- und Spaltenzeiger und der Rest bildet sich aus dem Stamm der `Range`-Variablen. `iColDaten` steht für einen Spaltenzeiger in `rngDaten`. Spreche ich darüber hinaus eine spezielle

Spalte an, etwa die Spalte, die den Kundennamen bereithält, so lautet der Zeiger `iColDatenKunde`. Zugegeben, die Namen werden länger als `i`, `j` oder `x1`. Aber sie erleichtern das Schreiben und Lesen von Programmen erheblich.

Da die `Cells`-Methode recht einfach ist und auch in nahezu jedem Beispiel auftaucht, sei hier auf weitere Ausführungen verzichtet.

Übrigens, haben Sie schon öfter überlegt, welche Spaltennummer die Spalte `L` hat?

TIPP: Merken Sie sich die Fünferschritte der Spalten 5, 10, 15, 20 und 25 in Form von E, J, O, T und Y und Sie werden die Spaltenzuordnung ohne Zuhilfenahme der Finger oder der Option `Z1S1` Bezugsart recht schnell ermitteln. Wenn Sie eine Eselsbrücke in Form eines Satzes mit diesen Anfangsbuchstaben ausknobeln, so bitte ich höflichst um Nachricht. Vorschlag unserer Korrektorin: Emils junger Onkel trägt Yuppieklamotten.

Characters-Eigenschaft

Mit der `Characters`-Eigenschaft erhält man Zugriff auf einen Substring (Teilzeichenkette) eines Zellinhalts, üblicherweise zum Zwecke der Textformatierung. Der folgende Ausdruck

```
rngAct.Characters(2,3).Font.Bold = True
```

formatiert beginnend mit dem zweiten Zeichen die drei folgenden in fetter Schrift. `Characters` wird also vornehmlich zusammen mit dem `Font`-Objekt zum Einsatz kommen.

Die Clear-Methoden

Die Methoden `ClearContents`, `ClearComments`, `ClearFormats` entfernen Inhalte, Kommentare und Formate des jeweils vorangestellten Range-Objekts; die `Clear`-Methode löscht alle vorstehenden Teile sowie `Notes` und `Outlines`. `Clear` löscht nicht die Zellen, sondern nur Inhalte und die angegebenen Attribute! Verwenden Sie zum Löschen von Zellen die `Delete`-Methode.

Column-Eigenschaft

Die `Column`-Eigenschaft gibt die absolute Spaltennummer eines Range-Objekts zurück. Im Beispiel zur `Autofilter`-Methode wurde die Nummer der Spalte ermittelt, in der die Überschrift „Zeitraum“ stand:

```
iColDatZtr = rngDat.EntireRow.Find(what:="Zeitraum", _  
lookat:=xlWhole).Column
```


Die `Find`-Methode erzeugt (oder auch nicht) ein `Range`-Objekt in der Zelle, in der der Wert Zeitraum gefunden wurde. Die nachstehende `Column`-Eigenschaft weist `iColDatZtr` die Spaltennummer zu, also 10 für Spalte J.

Columns-Eigenschaft

Zuerst wird's kompliziert:

Die `Columns`-Eigenschaft enthält eine `Cells`-Auflistung, die je eine Zelle aller durch das `Range`-Objekt betroffenen Spalten enthält. Diese Zellen befinden sich in der Zeile, in der die aktive Zelle des der `Columns`-Eigenschaft voranstehenden `Range`-Objekts steht. `Range("B3:C10").Columns` wird zu `Range("B3:C3")`.

Doch jetzt wird's einfach:

Da diese `Cells`-Auflistung von jeder betroffenen Spalte genau eine Zelle beinhaltet, kann man die jeder Auflistung innewohnende `Count`-Methode verwenden, um festzustellen, über wie viele Spalten sich ein `Range` erstreckt. Wir hatten das bereits, denn in

```
For iColPro = 2 To rngPro.CurrentRegion.Columns.Count
```

im Beispiel zur `AutoFilter`-Methode ermittelt der Ausdruck hinter `To` die Gesamtzahl der durch den Tabellenbereich (`CurrentRegion`) belegten Spalten. Doch Vorsicht! Der Spaltenzeiger `iColPro` ist nur dann unbedenklich, wenn Ihr `Range`-Objekt in Zelle A1 positioniert wurde; andernfalls müssen Sie rechnen. ;-)

ColumnWidth-Eigenschaft

Die `ColumnWidth`-Eigenschaft ist das Maß für die Spaltenbreite, die der Breite eines Zeichens in der Formatvorlage Standard entspricht. Bei Proportionalschriftarten, was ja die Regel in Excel ist (Arial), wird die Breite des Zeichens 0 verwendet.

Comment-Eigenschaft

Die `Comment`-Eigenschaft gibt das zur Zelle gehörende `Comment`-Objekt zurück. Ist der Zelle kein Kommentar zugeordnet, so ist das Objekt `Nothing`. Nachdem wir in der vorangehenden Ausgabe im Beispiel zur `AddComment`-Methode den Kommentar nutzten, um zu hinterlegen, wer wann den Zellwert in EURO konvertiert hat, wird das folgende Programmbeispiel den Zellkommentar nutzen, um SQL-Anweisungen zu hinterlegen.

Der Beispielcode ist einer Projektverwaltung eines großen Marketingunternehmens entnommen. Eine Reihe von Mitarbeitern arbeitet parallel an

statistischen Marketinganalysen. Jeden Montag werden an das Controlling verschiedene Zahlen bezüglich Auftragseingang und Abrechnungsvolumina geliefert. Eine feste Codierung der Abfragemechanik erschien mir so wenig angebracht, dass ich die SQL-Abfragen in die Kommentare der einzelnen Zellen hinterlegte. So werden in einer Schleife der belegte Tabellenbereich nach Zellkommentaren untersucht, die darin enthaltenen SQL-Abfragen ausgeführt und die Ergebnisse in die betreffenden Zellen hineingeschrieben.

```
Dim rs As Recordset
Dim strSQL As String
Dim rngAct As Range
Dim shtMontag As Worksheet
Set shtMontag = ThisWorkbook.Worksheets("Montagszahlen")
For Each rngAct In shtMontag.UsedRange.Cells
    If Not rngAct.Comment Is Nothing Then
        strSQL = rngAct.Comment.Text
        Set rs = db.OpenRecordset(Name:=strSQL, _
            Type:=dbOpenSnapshot)
        rngAct.Value = rs.Fields(0).Value
    End If
Next
```

Formeln oder Formelfragmente zu hinterlegen, die in VBA direkt ausgeführt werden können, funktioniert so natürlich nicht. Doch da liegt gerade die Würze des Einsatzes von Excel als Front-End für Datenbanken.

Copy-Methode

Kopieren bedeutet, dass danach irgendetwas doppelt vorhanden sein muss, und das ist zumeist nicht wünschenswert. Wenn ein Wert von einer zur nächsten Stelle übertragen wird, so handelt es sich in der Regel um eine Aggregation, sprich Summen-, Mittelwertbildung oder Ähnliches, wofür man sinnvollerweise nicht kopiert. Was bleibt also in der Praxis? Das Kopieren von vorformatierten Tabellenblättern beispielsweise. Nehmen wir an, Sie wollen Daten erfassen, die je Kostenstelle portioniert werden. Ihre Arbeitsmappe enthält also für alle bekannten Kostenstellen (oder Projekte) je eine Tabelle. Soll nun eine neue Kostenstelle ergänzt werden, so duplizieren Sie eine vorformatierte Mastertabelle und ändern den Blattnamen und ggf. die Beschriftung des Blattes irgendwo im Kopf. Richtig?

Falsch. Das ist der klassische Designfehler in Excel-Anwendungen. Der bessere Weg liegt darin, alle Daten in einer Datentabelle zu sammeln und die Analyse und Ausgabe dynamisch je Kostenstelle in einer speziellen

Ausgabetablelle vorzunehmen. Das ehemalige Tabellenblattkriterium, also Kostenstelle in unserem Beispiel, wird zu einer separaten Spalte. Sind die Daten zu umfangreich, um in einer Tabelle hinterlegt zu werden, so haben Sie wahrscheinlich den Punkt erreicht, an dem die Daten ohnehin in einer Datenbanktabelle besser aufgehoben wären.

Was also in der Praxis tatsächlich bleibt, ist das Kopieren und Einfügen von Zellformaten. Betrachten Sie hierzu bitte die Abb. 6.2 am Anfang des Kapitels. Wird ein neuer Bericht aufgebaut, so werden alle Zeilen unterhalb der Zeile 5 inklusive ihrer Formate gelöscht.

```
shtReport.UsedRange.Offset(5, 0).Clear
```

Der `UsedRange` (der gesamte belegte Tabellenbereich) gehört übrigens nicht zum `Range`-Objekt, sondern zum `Worksheet`-Objekt.

Danach wird der Report des gewünschten Zeitraums zeilenweise in einer Schleife aufgebaut, die hier nur angedeutet ist:

```
Do While <Bedingung>
    iRowReport as iRowReport + 1
    shtFormate.Rows(2).Copy
    shtReport.Rows(iRowReport).PasteSpecial xlPasteFormats
    'Daten eintragen
Loop
shtFormate.Rows(6).Copy
shtReport.Rows(iRowReport).PasteSpecial xlPasteFormats
Application.CutCopyMode = False
'Summenwerte errechnen und eintragen
```

Zuerst wird der Zeilenzeiger `iRowReport` inkrementiert. Bevor die Daten nun eingetragen werden, wird das Zeilenformat für die Datenzeilen aus der separaten Formate-Tabelle übertragen. Dies erfolgt über die `Copy`-Methode und die `PasteSpecial`-Methode, die als Argument `xlPasteFormats` erhält, wodurch nur die Formatierungen eingefügt werden. Nachdem die Datenzeilen abgearbeitet sind, folgt die Summenzeile, wobei auch hier zuerst die Formate übertragen werden. Bleiben noch die Werte der Summenzeile, und der Report ist fertig. Hier nun noch die dazugehörige Formate-Tabelle:

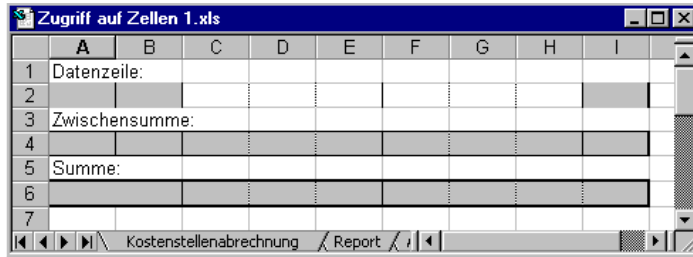


Abbildung 6.3: Tabelle mit vollständig vorformatierten Zeilen

Neben Zellhintergrund und Rahmen enthalten diese Zellen natürlich auch die Zeichen- und Zahlenformate. Abgesehen davon, dass Formatierungen per Code recht aufwendig sind, bietet diese Vorgehensweise auch noch den Vorteil, dass der Anwender jederzeit die Zellformate des Reports ändern kann, ohne dass wir Entwickler in Erscheinung treten müssen.

CurrentRegion-Methode

Die `CurrentRegion` gibt den aktuellen Bereich zurück, wie das so schön heißt. Sie erzeugt ausgehend von der aktiven Zelle einen zusammenhängenden, rechteckigen Zellbereich, der auf allen Seiten entweder vom Tabellenrand oder von leeren Zellen begrenzt ist. Probieren Sie es am besten selbst aus, denn die `CurrentRegion` ist mittels des Shortcuts `[Strg]+[*]` (das Multiplikationszeichen auf dem numerischen Tastaturblock) erreichbar.

Gerade zum Abarbeiten von Datentabellen klassischen Zuschnitts, also eine Überschriftzeile und viele Datenzeilen, leistet sie gute Dienste. In diesem Falle ist sie ideal dazu geeignet, in `For-Next`-Schleifen die obere Grenze festzulegen.

```
Dim rngDat As Range
Dim iRowDat As Long
Set rngDat = ThisWorkbook.Worksheets("Daten").Range("A1")
For iRowDat = 2 To rngDat.CurrentRegion.Rows.Count
    Anweisungen
Next
```

Delete-Methode

Die `Delete`-Methode löscht den Zellbereich, der durch das voranstehende `Range`-Objekt belegt ist. Mir ist kein Fall bekannt, in dem das Löschen einer einzelnen Zelle oder eines Zellbereichs erforderlich war. Was gelegentlich jedoch nötig wird, ist das Löschen einer ganzen Zeile oder einer ganzen Spalte.

Die `Delete`-Methode stellt ein optionales Argument bereit, mit dem durch `xlShiftToLeft` oder `xlShiftUp` festgelegt werden kann, in welche Richtung die anderen Zellen nachgeschoben werden sollen. Ist der Zellbereich hingegen eindeutig, kann auf dieses Argument verzichtet werden. Denn wenn Sie eine ganze Zeile löschen, ist es klar, wie Excel zu verfahren hat. Im folgenden Beispiel wird die Zeile 7 der Tabelle Daten gelöscht.

```
shtDat.Rows(7).Delete
```

EntireColumn-Eigenschaft

`EntireColumn` gibt ausgehend von der aktiven Zelle des Ranges die komplette Spalte zurück. Die nachstehende Programmzeile formatiert die durch `iColDat` bezeichnete Spalte kursiv.

```
rngDat.Cells(1, iColDat).EntireColumn.Font.Italic = True
```

Dieser Ausdruck ist gleichbedeutend mit

```
ShtDat.Columns(iColDat).Font.Italic = True
```

EntireRow-Eigenschaft

`EntireRow` gibt ausgehend von der aktiven Zelle des Ranges die komplette Zeile zurück. Im Beispiel zur `Delete`-Methode wurde mit dem `Worksheet`-Objekt und der `Rows`-Methode eine Zeile gelöscht. Hat man gerade kein `Worksheet`-Objekt zur Hand, geht's auch so:

```
rngDat.Cells(7, 1).EntireRow.Delete
```

Auch hier führt die Verwendung des `Worksheet`-Objekts zu einem kürzeren Ausdruck:

```
shtDat.Rows(7).Delete
```

Find-Methode

Die `Find`-Methode kann eine große Hilfe sein, wenn man sie im Griff hat, aber auch genauso gut eine große Plage, wenn man sie nicht im Griff hat. Sie dient dazu, eine Zelle in einem Zellbereich aufzufinden, deren Inhalt teilweise oder vollständig mit dem Suchbegriff übereinstimmt. Sie hat folgende Argumente:

- ▼ **What** (erforderlich)
der Suchbegriff üblicherweise als `String`, als numerischer oder Datumsdatentyp
- ▼ **After** (optional)
die Zelle, nach der die Suche begonnen werden soll, als `Range`-Objekt

- ▼ **LookIn** (optional)
eine der Konstanten `xlValues`, `xlFormulas` oder `xlNotes` für Zellinhalt, Formel oder Notiz
- ▼ **LookAt** (optional)
eine der Konstanten `xlPart` oder `xlWhole` für teilweise oder vollständige Übereinstimmung
- ▼ **SearchOrder** (optional)
eine der Konstanten `xlByRows` oder `xlByColumns` für zeilenweises oder spaltenweises Suchen
- ▼ **SearchDirection** (optional)
eine der Konstanten `xlNext` oder `xlPrevious` für nächste oder vorangehende Zelle
- ▼ **MatchCase** (optional)
eine der Konstanten `True` oder `False` für Unterscheiden oder Nicht-Unterscheiden in Groß- und Kleinbuchstaben
- ▼ **MatchByte** (optional)
wird nur in Ostasien verwendet

Eine wichtige Eigenart der `Find`-Methode ist, dass sie sich bis auf das Argument `What` die zuletzt verwendeten Parameter merkt; übrigens unabhängig davon, ob sie per Programm oder per Menübefehl Suchen im Menü Bearbeiten verwendet wurde. Deshalb ist es zwingend erforderlich, die wirklich wichtigen Argumente immer mit anzugeben. Dies sind neben dem obligatorischen `What` die Argumente `LookIn` und `LookAt`. Haben Sie das vergessen, sucht Excel Ihr `What` möglicherweise in den Zellkommentaren, und Sie wundern sich, weshalb Ihr Programm mal läuft und mal nicht.

Eine weitere Eigenart der `Find`-Methode ist der Umstand, dass sie erst nach der aktiven Zelle zu suchen beginnt. Steht Ihr Suchbegriff aber genau in dieser Zelle, findet die `Find`-Methode ihn nicht. Erwähnt werden sollte auch, dass die `Find`-Methode ein `Range`-Objekt zurückgibt, wenn sie den Suchbegriff gefunden hat, aber `Nothing`, wenn nicht.

Die Einschränkung, dass die `Find`-Methode erst nach, also (zumeist) unterhalb der aktiven Zelle ihre Suche aufnimmt, wirkt sich dann nicht aus, wenn der durchsuchte Zellbereich (meist eine ganze Spalte) durch eine Überschrift eingeleitet wird. Und das ist in Datentabellen wohl die Regel. Gegen die Persistenz der zuletzt verwendeten Argumente ist auch ein Kraut gewachsen: Man gibt sie an. Und die letzte Hürde, dass die `Find`-Methode eben `Nothing` zurückgibt, wenn sie keine Übereinstimmung

findet, kann dadurch genommen werden, dass genau dieser Fall abgeprüft wird, bevor wir uns auf das zurückgegebene Range-Objekt stürzen.

Ein Beispiel. In einer Artikeltabelle stehen unter der Überschriftzeile alle Artikel unseres Hauses, wobei die Spalte A die Artikelnummern enthält. Auf einem anderen Tabellenblatt stoßen wir auf eine Artikelnummer und wollen uns aus der Artikeltabelle den dazugehörigen Preis ermitteln.

```
Dim rngAng As Range      'Range-Objekt in Tabelle Angebot
Dim rngArt As Range      'Range-Objekt in Tabelle Artikel
Dim iRowAng As Long      'Zeilenzeiger in Tabelle Angebot
Dim iRowArt As Long      'Zeilenzeiger in Tabelle Artikel
Dim strArtNr As String   'gesuchte Artikelnummer
For iRowAng = 2 To rngAng.CurrentRegion.Rows.Count
    strArtNr = rngAng.Cells(iRowAng, 1).Value
    If Not rngArt.EntireColumn.Find(What:=strArtNr, _
        LookIn:=xlValues, LookAt:=xlWhole) Is Nothing
Then
    iRowArt = rngArt.EntireColumn.Find(What:=strArtNr, _
        LookIn:=xlValues, LookAt:=xlWhole).Row
    rngAng.Cells(iRowAng, 2).Value = _
        rngArt.Cells(iRowArt, 2).Value
Else
    rngAng.Cells(iRowAng, 2).Value = "n.v."
End If
Next
```

Innerhalb der For-Next-Schleife wird der Variablen `strArtNr` die Artikelnummer des zu suchenden Artikels zugewiesen. Danach wird in der If-Zeile geprüft, ob die Artikelnummer gefunden wird, also nicht `Nothing` ist. Wenn ja, wird die `Row`-Eigenschaft des von der `Find`-Methode zurückgegebenen Range-Objekts in dem Zeilenzeiger `iRowArt` abgelegt. Anschließend wird in `rngAng.Cells(iRowAng, 2)` der `Value` von `rngArt.Cells(iRowArt, 2)` eingetragen. Im Else-Zweig erhält die Zelle den Eintrag `n. v.` für nicht vorhanden.

Gelegentlich ist eine Variante anzutreffen, die eine weitere Range-Variablen verwendet. Das ist etwas übersichtlicher, erfordert aber einen weiteren `Dim` sowie eine weitere Programmzeile (Änderungen und Ergänzung sind fett dargestellt):

```
...
Dim rngFind As Range
...
```

```

For iRowAng = 2 To rngAng.CurrentRegion.Rows.Count
    strArtNr = rngAng.Cells(iRowAng, 1).Value
    Set rngFind = rngArt.EntireColumn.Find(What:=strArtNr,
-
        LookIn:=xlValues, LookAt:=xlWhole)
    If Not rngFind Is Nothing Then
        iRowArt = rngFind.Row
        rngAng.Cells(iRowAng, 2).Value = _
            rngArt.Cells(iRowArt, 2).Value
    Else
        rngAng.Cells(iRowAng, 2).Value = "n.v."
    End If
Next

```

Was nun, wenn der zu suchende Begriff in der Tabelle mehrfach vertreten ist? Zuerst zur Theorie. Es gibt eine Methode des Namens `FindNext`, die den zuletzt ausgeführten `Find` fortführt. In dem Argument `After` wird mittels eines `Range`-Objekts angegeben, nach welcher Zelle die Suche fortgeführt wird. Ist sie erfolglos am Ende der Spalte angekommen, beginnt `FindNext` wieder am Anfang. Sie untersucht alle Zellen des angegebenen Bereichs bis auf die durch `After` spezifizierte. Also eigentlich ganz praktisch, aber sie merkt sich nicht, ob sie eine Zelle bereits zurückgegeben hat, d. h. sie durchläuft den `Range` wie ein Hamster in seinem Rad, den es auch nicht interessiert, ob er immer wieder auf denselben Holm tritt.

Ein Beispiel. Die Spalte A der zu durchsuchenden Tabelle enthält Nachnamen. Außerhalb der Schleife wird das Ergebnis der `Find`-Methode dem Objekt `rngFind` zugewiesen sowie das erste Array-Element hinzugefügt. Die Schleife wird nun so lange durchlaufen, bis die Rückgabe `Nothing` ist. In der Schleife wird in das letzte Array-Element der in Spalte B stehende Vorname eingetragen. Danach erhält `iRowAlt` die Zeile des letzten Treffers. Nach der Erhöhung der oberen Grenze des Arrays wird mit `FindNext` der nächste ermittelt. Ist die Zeilennummer dieses Treffers kleiner oder gleich `iRowAlt`, so haben wir einen alten Bekannten getroffen und können die Suche abbrechen.

```

Dim rngSuch As Range      'zu durchsuchender Range
Dim rngFind As Range      'Rückgabe-Range
Dim strVorname() As String 'Datenfeld zum Speichern der
                           'gefundenen Begriffe
Dim iRowAlt As Long       'Zeiger auf den vorgehenden
                           'rngFind

```



```

Set rngSuch = ThisWorkbook.Worksheets("Personen"). _
    Range("A1")

Set rngFind = rng.EntireColumn.Find(what:="Meier", _
    LookIn:=xlValues, lookat:=xlWhole)

ReDim strVorname(1)
Do Until rngFind Is Nothing
    strVorname(UBound(strVorname())) = _
        rngFind.Cells(1, 2).Value
    iRowAlt = rngFind.Row
    ReDim Preserve strVorname(UBound(strVorname()) + 1)
    Set rngFind = rng.EntireColumn.FindNext(rngFind)
    If rngFind.Row <= iRowAlt Then Exit Do
Loop
ReDim Preserve strVorname(UBound(strVorname()) - 1)

```

Wie Sie sehen, ist das also nicht ganz so einfach. Um es kurz zu machen, ich setze diese Vorgehensweise nicht ein. Denn mittels `AutoFilter`- und `SpecialCells`-Methode erhalte ich eine `Cells`-Auflistung, die alle gesuchten Zellen, und zwar *nur diese*, enthält. Diese Lösung kommt mir handlicher vor. Wir werden anlässlich der Besprechung der `SpecialCells`-Methode darauf zurückkommen.

Font-Eigenschaft

Die `Font`-Eigenschaft des `Range`-Objekts gibt das `Font`-Objekt zurück, das wiederum alle Informationen enthält, die dem kompletten Zellinhalt zugeordnet sind. Im Unterschied hierzu beinhaltet die `Font`-Eigenschaft des `Characters`-Objekts nur die Formatierungen eines Teils des Zellinhalts.

Hinter dem `Font`-Objekt verbergen sich Eigenschaften wie `Bold`, `Italic`, `Name`, `Size` sowie einige andere. Wenn Sie die Techniken verwenden, die unter `Copy`-Methode beschrieben sind, so werden Sie dem `Font`-Objekt selten begegnen, sieht man von sporadischen `Font.Bold = True` einmal ab. Und wenn Sie zum Beispiel negative Werte in roter Farbe darstellen wollen, so können Sie dies auch dem `Zahlenformat` anvertrauen und benötigen hierfür keinen Zugriff auf das `Font`-Objekt.

FormulaLocal-Eigenschaft

Die `FormulaLocal`-Eigenschaft bietet Ihnen die Möglichkeiten, Formeln in der A1-Schreibweise in eine Zelle zu schreiben. Im Gegensatz zur `Formula`-Eigenschaft, die eine amerikanische Syntax zugrunde legt, können die Formeln bei Verwendung der `FormulaLocal`-Eigenschaft gemäß den Ländereinstellungen mit deutschen Funktionsnamen gebildet werden. Die folgenden Ausdrücke sind funktional identisch:

```
rng.Value = "=AVERAGE(B1,C1,3.14)"
rng.Formula = "=AVERAGE(B1,C1,3.14)"
rng.FormulaLocal = "=MITTELWERT(B1;C1;3,14;3,14)"
```

Da man schon genug damit zu tun hat, sich die deutschen Funktionsnamen zu merken, macht es wenig Sinn, sich auch noch mit über 250 englischen Funktionsnamen zu belasten, wodurch die `FormulaLocal`-Eigenschaft der `Formula`-Eigenschaft vorzuziehen ist.

FormulaR1C1Local-Eigenschaft

`Formula` und `FormulaLocal` arbeiten in der A1-Schreibweise, wie wir oben gesehen haben. Zwei Situationen könnten dazu verleiten, stattdessen Z1S1-Schreibweise zu verwenden:

- ▼ Sie wollen explizit eine relative Adressierung.
- ▼ Sie wollen einen Spaltenzeiger in der Formel verwenden.

Bei der Z1S1-Schreibweise bedeutet der Wert hinter Z bzw. S ohne Klammern, dass Sie in exakt die Zeile oder Spalte wollen, die durch die Zahl angegeben wurde. C5 wird zu R5C3, also Row 5 und Column 3. Schreiben Sie Werte hingegen in Klammern, so geben Sie damit an, wie viele Zeilen oder Spalten zwischen der aktiven Zelle und der Zielzelle liegen.

Nehmen wir an, Sie wollen in A1 die Summe der absoluten Bezüge `$B$1` und `$C$1` bilden. Die Formel sähe dann so aus:

```
rng.FormulaR1C1Local = "=SUMME(Z1S2;Z1S3)"
```

Unter der Verwendung der relativen Adressierung für B1 und C1 ausgehend von A1 wird hieraus:

```
rng.FormulaR1C1Local = "=SUMME(ZS(1);ZS(2))"
```

Natürlich kann man das auch mit der `FormulaLocal`-Eigenschaft erreichen, denn `Chr(64 + iCol)` ergibt auch den gewünschten Spaltenbezeichner. Aber nur bis zur 26. Spalte, denn danach ist unser Alphabet zu Ende und die Ermittlung der weiteren Spaltenbezeichner erfordert unnötige Klammern.

HasFormula-Eigenschaft

Die `HasFormula`-Eigenschaft teilt uns mit, ob die betreffende Zelle eine Formel aufweist.

Zum Einsatz kommt sie beispielsweise, wenn Sie in einer Tabelle schreiben müssen, die teilweise mit Formeln versehen ist. In der Beispieltabelle sind diese angenommenen Formelzellen fett dargestellt und mit einer Hinter-

grundfarbe versehen. Es geht hier nicht um das Tabellendesign, sondern um die Lage von Daten- und Formelzellen.

	A	B	C	D	E	F	G	H	I	J
1										
2	Kostenträger	121	124	120	132	134	137	138	130	Gesamt
3	11	100	100	200	100	100	100	100	400	600
4	12			0					0	0
5	10	100	100	200	100	100	100	100	400	600
6	21			0					0	0
7	22			0					0	0
8	23			0					0	0
9	24			0					0	0
10	20	0	0	0	0	0	0	0	0	0
11	Gesamt	100	100	200	100	100	100	100	400	600
12										

Abbildung 6.4: Tabelle mit Formeln

Es geht also darum, von Zelle B3 (denn oberhalb und links davon stehen Überschriften) bis zur letzten Zelle des aktuellen Bereichs, also der `CurrentRegion`, alle Zellen zu bearbeiten, die keine Formeln aufweisen (`Not Range.HasFormula`). Ist eine solche Zelle gefunden, werden die betreffende Kostenstelle aus Zeile 2 und der Kostenträger aus Spalte A in zwei Variablen hinterlegt, die dann in eine SQL-Abfrage integriert werden. Da es sich um eine Aggregatfunktion handelt, steht das Ergebnis in Feld 0 des Recordsets und wird von dort in das Range-Objekt übertragen.

```
Public Sub CalcKoStSheet()
    Dim db As Database           'Datenbankvariable
    Dim rs As Recordset         'Recordsetvariable
    Dim rngKoSt As Range        'Range-Objekt in KoSt-Rechnung
    Dim iRowKoSt As Integer     'Zeilenzeiger in rngKoSt
    Dim iColKoSt As Integer     'Spaltenzeiger in rngKoSt
    Dim lngKoSt As Long         'KostenstellenvARIABLE
    Dim lngKoTr As Long         'KostenträgervARIABLE
    Dim strSQL As String        'SQL-Abfrage

    Set db = _
    DBEngine.OpenDatabase(ThisWorkbook.Path & "\KoSt.mdb")
    Set rngKoSt = shtKoSt.Range("A1")
    For iRowKoSt = 3 To rngKoSt.CurrentRegion.Rows.Count
        For iColKoSt = 2 To _
            rngKoSt.CurrentRegion.Columns.Count
            If Not rngKoSt.Cells(iRowKoSt, iColKoSt). _
                HasFormula Then
```

```

lngKoSt = rngKoSt.Cells(2, iColKoSt).Value
lngKoTr = rngKoSt.Cells(iRowKoSt, 1).Value
strSQL = "SELECT Sum(Betrag) FROM " & _
        "tbKostenstellen WHERE KoSt = " & _
        lngKoSt & " AND KoTr = " & lngKoTr & ";"
Set rs = db.OpenRecordset(strSQL)
rngKoSt.Cells(iRowKoSt, iColKoSt).Value = _
        rs.Fields(0).Value
End If
Next
Next
db.Close
End Sub

```

Eine Anmerkung noch: Datenbanken öffne ich nicht so wie in dem Beispiel, sondern mittels einer Klasse, die mir eine Methode `GetDB` bereitstellt, die alle möglichen Fehlerfälle prüft und mir die Datenbank dann geöffnet zurückgibt. Im Fehlerfall erzeugt die Klasse eine Fehlermeldung, die in einer Fehlerbehandlungsroutine aufgefangen wird. Eine ähnliche Klasse verwende ich zum Öffnen eines Workbooks. Beide Klassen werden noch vorgestellt.

Height-, Left-, Top- und Width-Eigenschaft

Dass Excel sich bestens zur Manipulation von Zahlen eignet, ist hinlänglich bekannt. Eine Excel-Tabelle kann aber auch Container für eine Vielzahl von graphischen Objekten sein. So zum Beispiel GANTT-Diagramme, die auch als Balkendiagramme bekannt sind.

Die hier vorgestellte Lösung ist kein Ersatz für professionelle PM-Software wie MS-Project oder andere. Doch manchmal geht es nur um die Visualisierung von ein paar zeit- oder terminrelevanten Daten, wofür der Einsatz einer solchen Software in keinem Bezug zur Aufgabe stehen würde. Abb. 6.5 zeigt die Visualisierung von vier einfachen Vorgängen, die in Abb. 6.6 in Tabellenform dargestellt sind.

Nach der Variablendeklaration werden in einer `For-Next`-Schleife eventuell vorhandene Shapes gelöscht. Shapes stellen eine Art Metaklasse dar, unter die alle graphischen Objekte eines Tabellenblattes fallen. Diese Schleife muss übrigens rückwärts durchlaufen werden, weil der Index (für den ich kurzerhand `iRowGANTT` gewählt habe, um eine weitere Variable einzusparen) sonst Elemente ansprechen würde, die bereits gelöscht sind.

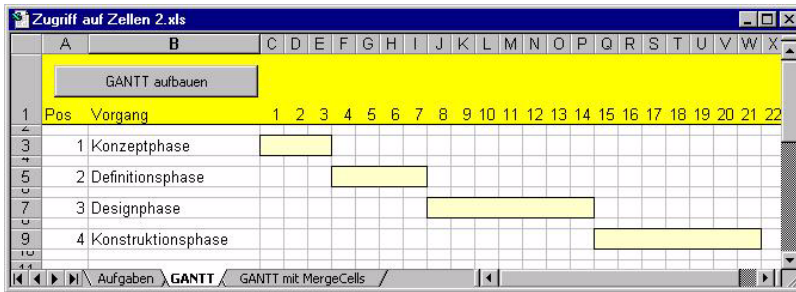


Abbildung 6.5: Visualisierung terminbezogener Aufgaben

Pos	Aufgabe	Vorgänger	Dauer
1	Konzeptphase		3
2	Definitionsphase	1	4
3	Designphase	2	7
4	Konstruktionsphase	3	7

Abbildung 6.6: dazugehörige Aufgabentabelle

Danach wird der Spaltenzeiger für die erste Spalte auf 3 gesetzt, da (per Definition) der erste Balken am linken Rand der Spalte C beginnen soll. Nun wird in einer For-Next-Schleife der Zeilenzeiger `iRowTasks` für die Aufgabentabelle gebildet. Der Zeilenzeiger `iRowGantt` wird bestimmt durch das Tabellendesign aus `iRowTasks` generiert. Jetzt geht es darum, das Ende des jeweiligen Balkens zu finden, was durch

```
iColGANTT2 = iColGANTT1 + _
    rngTasks.Cells(iRowTasks, 4).Value
```

geschieht. Nun werden die Koordinaten und Größen der zukünftigen Textbox gebildet. `sngWidth` müsste eigentlich aus dem rechten Rand der vorangehenden Zelle gebildet werden, der jedoch mit dem linken Rand der darauffolgenden Zelle übereinstimmt, was die Sache vereinfacht.

Nachdem alle Parameter bekannt sind, wird das neue Shape mit der `Add-Textbox-Methode` der `Shapes`-Auflistung erzeugt. Leider funktioniert es nicht, das neue Objekt in derselben Zeile einer `Shape`-Variablen zuzuweisen nach dem Muster:

```
ShpACT = shtGANTT.Shapes.AddTextbox(...)
```

weshalb die `With`-Konstruktion erforderlich ist. Abschließend wird die ehemalige rechte Zelle zur neuen linken Zelle des nächsten Vorgangs.

```

Private Sub btnMakeGANTT_Click()
    Dim rngTasks As Range
    Dim rngGANTT As Range
    Dim iRowTasks As Integer
    Dim iRowGANTT As Integer
    Dim iColGANTT1 As Integer
    Dim iColGANTT2 As Integer
    Dim sngTop As Single
    Dim sngLeft As Single
    Dim sngWidth As Single
    Dim sngHeight As Single
    Set rngTasks = shtTasks.Range("A1")
    Set rngGANTT = shtGANTT.Range("A1")
    'Löschen aller vorhandener Shapes bis auf CommandButton
    For iRowGANTT = shtGANTT.Shapes.Count To 1 Step -1
        If shtGANTT.Shapes(iRowGANTT).Name <> _
            "cmdMakeGANTT" Then
            shtGANTT.Shapes(iRowGANTT).Delete
        End If
    Next
    'Anfangswert für 1. Spalte
    iColGANTT1 = 3
    'Schleife für alle Einträge in rngTasks
    For iRowTasks = 2 To rngTasks.CurrentRegion.Rows.Count
        iRowGANTT = -1 + iRowTasks * 2
        iColGANTT2 = iColGANTT1 + _
            rngTasks.Cells(iRowTasks, 4).Value
        'Pos und Aufgabe aus rngTasks übertragen
        rngGANTT.Cells(iRowGANTT, 1).Value = _
            rngTasks.Cells(iRowTasks, 1).Value
        rngGANTT.Cells(iRowGANTT, 2).Value = _
            rngTasks.Cells(iRowTasks, 2).Value
        'Koordinaten und Größe des Shape festlegen
        sngTop = rngGANTT.Cells(iRowGANTT, iColGANTT1).Top
        sngLeft = rngGANTT.Cells(iRowGANTT, iColGANTT1).Left
        sngWidth = rngGANTT.Cells(iRowGANTT, _
            iColGANTT2).Left - sngLeft
        sngHeight = rngGANTT.Cells(iRowGANTT, _
            iColGANTT1).Height
        'Shape erzeugen
        With shtGANTT.Shapes.AddTextbox( _
            msoTextOrientationHorizontal, _
            sngLeft, sngTop, _

```

```

        sngWidth, sngHeight)
    'Hintergrundfarbe festlegen
    .Fill.ForeColor.SchemeColor = 26
End With
'neuer Anfang gleich altes Ende
iColGANTT1 = iColGANTT2
Next
End Sub

```

Dieses Beispiel ist zugegebenermaßen recht spartanisch. So weist die Zeitachse keine absoluten Datumswerte auf, auch finden Wochenenden oder Feiertage keine Berücksichtigung.

Hidden-Eigenschaft

Die Hidden-Eigenschaft, die nur auf ganze Zeile oder Spalten angewendet werden kann, blendet mit dem Argument True oder False den betreffenden Zellbereich aus oder ein:

```
rng.Cells(iRow, 1).EntireRow.Hidden = True
```

oder mit einem Worksheet-Objekt

```
sht.Rows(iRow).Hidden = True
```

Interior-Objekt

Das Interior-Objekt stellt den Zellhintergrund dar und stellt alle Formatierungen bereit, die im Menü FORMAT | ZELLEN im Register Muster angeboten werden. Die Formatierungen, die Sie mit dem Interior-Objekt vornehmen, beziehen sich im Übrigen auf alle Zellen des links davon stehenden Range-Objekts.

Von den insgesamt neun Eigenschaften des Interior-Objekts ist zum Glück nur die ColorIndex-Eigenschaft von praktischer Bedeutung. Die jeweilige Farbe kann über eine der 56 Farbindizes sowie die Konstante xlColorIndexNone (-4124) zugewiesen werden. Wie an anderer Stelle bereits erwähnt wurde, war es bis zur Version 7 von Excel möglich, den Wert der Konstanten in der Palette-Schaltfläche „Füllfarbe“ abzulesen, da die einzelnen Farben in der aufsteigenden Reihenfolge ihrer Farbindizes angeordnet waren. Unglücklicherweise fiel diese Hilfestellung für uns Entwickler ab der Version 8 weg, da dort die Farben eine andere, allerdings farblich bestimmte Sortierung erfuhren.

Wie kommt man nun zu den Farbindizes? Formatieren Sie in einer leeren Tabelle einige Zellen der Spalte A mit den gewünschten Farben. Schreiben Sie dann die folgenden Zeilen in ein Modulblatt:

```
Sub GetColors
    Dim i as Integer
    For i = 1 To 10
        ActiveSheet.Range("B" & i).Value = _
            ActiveSheet.Range("A" & i).Interior.ColorIndex
    Next
End Sub
```

Nachdem Sie das Programmchen laufen lassen, stehen in der Spalte B die Farbindizes.

Es stehen uns nun drei Varianten zur Farbzuzuweisung zur Verfügung. Ab Office 2000 aber werden diese um eine vierte ergänzt, die hier vorgestellt werden soll.

Zuweisung über Farbindizes: Sie lernen die Farbindizes Ihrer Lieblingsfarben auswendig und wenden diese unter Angabe der jeweiligen Zahl fallweise an:

```
Range.Interior.Colorindex = 4 'hellgrün
```

Zuweisung über farborientierte Konstanten: Sie legen sich ein separates Modulblatt an und deklarieren dort eine Reihe von Farbkonstanten nach folgendem Muster:

```
Public Const xlColorBlack As Integer = 1
Public Const xlColorRed As Integer = 3
Public Const xlColorOrange As Integer = 46
Public Const xlColorGreenLight As Integer = 4
Public Const xlColorYellow As Integer = 6
Public Const xlColorYellowMedium As Integer = 36
Public Const xlColorYellowLight As Integer = 19
Public Const xlColorYellowBrown As Integer = 40
Public Const xlColorGray50 As Integer = 16
Public Const xlColorGray40 As Integer = 48
Public Const xlColorGray25 As Integer = 15
```

Die Zuweisung erfolgt dann über „sprechende“ Konstanten:

```
Range.Interior.Colorindex = xlColorGreenLight
```


Zuweisung über aufgabenorientierte Konstanten: Eine größere Flexibilität erhalten Sie, wenn Sie neben diesen Zuweisungen eine weitere Gruppe von Konstanten erzeugen, die aufgabenorientiert sind. Anstelle des obigen Anweisungsblocks könnte dann Folgendes stehen:

```
Private Const xlColorGray50 As Integer = 16
Private Const xlColorGray25 As Integer = 15
Public Const xlColorTitle As Integer = xlColorGray40
Public Const xlColorData As Integer = xlColorGray25
Public Const xlColorSummary As Integer = xlColorGray40
```

Die Farbzweisung würde dann so aussehen:

```
Range.Interior.Colorindex = xlColorTitle
```

Der Vorteil liegt darin, dass Sie sich bei der Codierung keine konkreten Gedanken über die endgültige Farbgestaltung Ihrer Applikation machen müssen. Wenn Sie in Ihre blaue Periode übergehen, so weisen Sie Ihren `xlColorTitle` und `xlColorData` die entsprechenden Farbkonstanten der blauen Farbtöne zu.

Zuweisung über Klasse mit Enumeration: In den Deklarationsteil eines Klassenmoduls des Namens `clsRange` kommen folgende Zeilen:

```
Public Enum xlColor
    xlColorBlack = 1
    xlColorRed = 3
    xlColorOrange = 46
End Enum
```

Danach erzeugen wir die Methode `InteriorColor`, in deren Argumententeil die oben erzeugte Enumeration `xlColor` als Datentyp angegeben wird:

```
Public Sub InteriorColor(ByVal Range As Range, ByVal
ColorIndex As xlColor)
    Range.Interior.ColorIndex = ColorIndex
End Sub
```

Nun können wir in allen Modulen dieses Projekts auf diese Enum-Konstanten durch Anwenden der Methode `InteriorColor` unserer Klasse zugreifen. Und jetzt wird's spannend: Sobald das Argument `ColorIndex` aktiv wird, werden die bei `xlColor` definierten Konstanten als Elemente automatisch vorgeblendet.

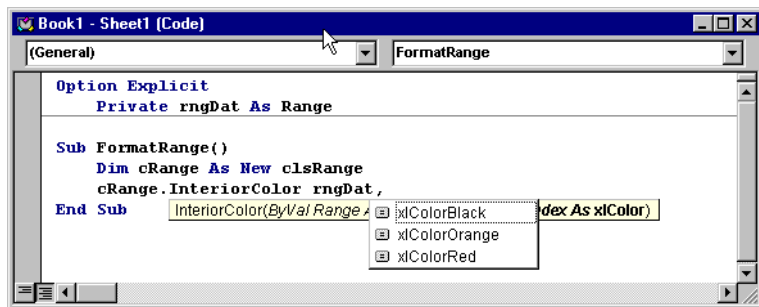


Abbildung 6.7: Einsatz der Enumeration

Locked-Eigenschaft

Das Sperren von Zellen vollzieht sich in zwei Stufen. Mit der `Locked`-Eigenschaft eines `Range`-Objekts können Sie für einzelne Zellen oder Zellbereiche festlegen, ob diese gesperrt sein sollen, wenn – und das ist der zweite Schritt – das Tabellenblatt mit der `Protect`-Methode geschützt wird.

```
rngTest.Locked = True
shtTest.Protect
```

Wenn beim Versuch, die `Locked`-Eigenschaft eines `Range`-Objekts zu verändern, der Fehler 1004 auftritt, so ist dieses `Range`-Objekt möglicherweise Teil eines verbundenen Zellbereichs. In diesem Fall hilft Ihnen die nachfolgend beschriebene `MergeArea`-Eigenschaft weiter.

MergeArea-Eigenschaft

Ist eine Zelle Bestandteil eines verbundenen Zellbereichs, so gibt die `MergeArea`-Eigenschaft den kompletten verbundenen Zellbereich zurück:

```
rngTest(rngTest.MergeArea.Address).Locked=true
```

MergeCells-Eigenschaft

Mit dem Wert dieser Eigenschaft (`True` oder `False`) wird festgelegt, ob die Zellen des davor stehenden `Range`-Objekts verbunden werden sollen oder nicht. Diese im Grunde recht triviale Geschichte ist auch durchaus geeignet, das Beispiel zur `Height`-Eigenschaft zu vereinfachen.

In dem dortigen Beispiel mussten wir einige Anstrengungen unternehmen, um die Koordination und Dimensionen der `TextBox` zu ermitteln, bevor sie über `Shapes.AddTextbox` erzeugt werden konnte. Folgender Code würde mittels verbundener Zellen „quasi“ zum selben Ergebnis führen (geänderte und neue Zeilen sind fett dargestellt):

```

Private Sub btnMakeGANTT_Click()
    Dim rngTasks As Range
    Dim rngGANTT As Range
    Dim rngMerge As Range
    Dim iRowTasks As Integer
    Dim iRowGANTT As Integer
    Dim iColGANTT1 As Integer
    Dim iColGANTT2 As Integer
    Set rngTasks = shtTasks.Range("A1")
    Set rngGANTT = shtGANTT.Range("A1")
    iColGANTT1 = 3
    For iRowTasks = 2 To rngTasks.CurrentRegion.Rows.Count
        iRowGANTT = -1 + iRowTasks * 2
        iColGANTT2 = iColGANTT1 + rngTasks.Cells(iRowTasks,
4).Value - 1
        rngGANTT.Cells(iRowGANTT, iColGANTT1).EntireRow.Clear
        rngGANTT.Cells(iRowGANTT, 1).Value =
rngTasks.Cells(iRowTasks, 1).Value
        rngGANTT.Cells(iRowGANTT, 2).Value =
rngTasks.Cells(iRowTasks, 2).Value
        Set rngMerge = rngGANTT.Range( _
            rngGANTT.Cells(iRowGANTT, iColGANTT1), _
            rngGANTT.Cells(iRowGANTT, iColGANTT2))
        rngMerge.MergeCells = True
        rngMerge.Interior.ColorIndex = 6
        rngMerge.BorderAround Weight:=xlMedium
        iColGANTT1 = iColGANTT2 + 1
    Next
End Sub

```

Die Berechnung der Spalte iColGANTT2 ändert sich, da es nicht mehr darum geht, die Koordinate des linken Randes der Zelle zu ermitteln, sondern diese komplett in unseren Zellbereich zu integrieren. Mit

```
Set rngMerge = rngGantt.Range(...)
```

wird ein neues Range-Objekt erzeugt, das sich über alle zusammenzufassenden Zellen erstreckt. Dieses neue Range-Objekt dient nunmehr als Basis für MergeCells, Interior.ColorIndex und BorderAround.

Wozu dient also noch die reichlich umständliche Vorgehensweise, die bei der Diskussion der Height-, Left-, Top- und Width-Eigenschaften vorgestellt wurde? Sie ist zum Beispiel zum einen dann erforderlich, wenn Ihr rechter oder linker Rand nicht mit Spaltengrenzen übereinstimmt. Haben Sie also anstelle von Tagen im Diagramm die Einheit Wochen gewählt,

wollen aber dennoch angebrochene Wochen darstellen, endet oder beginnt der Balken zwangsläufig zwischen den Spaltengrenzen, was mit `MergeCells` nicht darstellbar ist.

Wollen Sie hingegen eine Art Preisliste oder Angebot mit Bildern (`Shapes.AddPicture`) darstellen, was übrigens eine interessante Aufgabe ist, so müssen Sie in jedem Falle diese Berechnungen durchführen. Also: `MergeCells` hilft Ihnen bei einem Sonderfall von graphischen Elementen, ersetzt diese aber nicht.

NumberFormat- und NumberFormatLocal-Eigenschaft

Mit diesen beiden Eigenschaften sind wir in der Lage, Zahlenformate per Programm zuzuweisen. Ähnlich wie bei `Formula` und `FormulaLocal` können die internationalen (amerikanischen) Formatierungskonventionen oder aber die lokalen gemäß der Ländereinstellungen der Systemsteuerung zur Anwendung kommen. Die `NumberFormat`-Eigenschaft folgt hier den Regeln der `Format()`-Funktion aus VB und VBA.

So würden die Datumsformate für 02.02.1999 16:31:24 in beiden Varianten aussehen:

```
Range.NumberFormat = "dd.mm.yyyy hh:mm:ss"  
Range.NumberFormatLocal = "TT.MM.JJJJ hh:mm:ss"
```

Und so für die Zahl 1.234,56 als Zahlenformat:

```
Range.NumberFormat = "#,##0.00"  
Range.NumberFormatLocal = "#.##0,00"
```

Die vordefinierten Formate, wie sie die `Format()`-Funktion zulässt, können allerdings nicht angewendet werden.

Offset-Methode

Sie kennen sicherlich die Situation, dass Sie wiederholt eine Tabelle unbekannter Ausmaße aufbauen müssen. Das Beispiel in Abbildung 6.8 stellt eine variierende Anzahl von Projekten über einen Zeitraum von sechs Monaten dar.

Wenn nun sichergestellt ist, dass die zugrunde liegenden Daten genau einen Wert je Projekt und Monat vorweisen können, müssen wir die Daten der Auswertungstabelle vor dem Neuaufbau nicht löschen. Ist dies jedoch nicht der Fall, muss eine Löschschleife her – oder eben die `Offset`-Methode.

	A	B	C	D	E	F	G	H
1		Jan	Feb	Mrz	Apr	Mai	Jun	
2	Projekt 1	89,25 h	70,10 h	58,20 h	98,80 h	60,33 h	68,89 h	
3	Projekt 2	58,28 h	74,30 h	52,61 h	50,15 h	91,25 h	96,63 h	
4	Projekt 3	55,06 h	71,24 h	76,86 h	59,12 h	56,99 h	96,49 h	
5	Projekt 4	55,34 h	68,04 h	60,72 h	89,73 h	92,35 h	95,92 h	
6	...	83,42 h	54,08 h	98,32 h	56,35 h	54,91 h	86,14 h	
7	Projekt n	59,63 h	98,40 h	55,19 h	61,15 h	80,23 h	60,38 h	
8								
9								
10								

Abbildung 6.8: Beispiel einer Tabelle mit einer unbekannten Anzahl von Zeilen

Die auf ein Range-Objekt anzuwendende Methode verschiebt diesen Range um eine anzugebende Anzahl von Zeilen und/oder Spalten. Der Basis-Range, auf den die Offset-Methode angewendet wird, ist unsere altbekannte und bewährte CurrentRegion:

	A	B	C	D	E	F	G	H
1		Jan	Feb	Mrz	Apr	Mai	Jun	
2	Projekt 1	89,25 h	70,10 h	58,20 h	98,80 h	60,33 h	68,89 h	
3	Projekt 2	58,28 h	74,30 h	52,61 h	50,15 h	91,25 h	96,63 h	
4	Projekt 3	55,06 h	71,24 h	76,86 h	59,12 h	56,99 h	96,49 h	
5	Projekt 4	55,34 h	68,04 h	60,72 h	89,73 h	92,35 h	95,92 h	
6	...	83,42 h	54,08 h	98,32 h	56,35 h	54,91 h	86,14 h	
7	Projekt n	59,63 h	98,40 h	55,19 h	61,15 h	80,23 h	60,38 h	
8								
9								
10								

Abbildung 6.9: Durch CurrentRegion erzeugter Range

Da dieser Range die Zeilen- und Spaltentitel umfasst, verschieben wir die Markierung einfach um eine Spalte nach rechts und eine Zeile nach unten und erhalten einen neuen Range, wie in Abbildung 6.10 dargestellt. Dieser kann nun bequem durch ClearContents gelöscht werden.

Der Code:

```
rngDat.CurrentRegion.Offset( 1, 1).ClearContents
```

Dass dieser neue Range über unseren Ursprungs-Range hinausragt, stört uns hier nicht weiter ... Was aber, wenn die Tabelle eine Summenzeile und -spalte umfasst, die wir nicht gelöscht haben wollen?

	A	B	C	D	E	F	G	H
1		Jan	Feb	Mrz	Apr	Mai	Jun	
2	Projekt 1	89,25 h	70,10 h	58,20 h	98,80 h	60,33 h	68,89 h	
3	Projekt 2	58,28 h	74,30 h	52,61 h	50,15 h	91,25 h	96,63 h	
4	Projekt 3	55,06 h	71,24 h	76,86 h	59,12 h	56,99 h	96,49 h	
5	Projekt 4	55,34 h	68,04 h	60,72 h	89,73 h	92,35 h	95,92 h	
6	...	83,42 h	54,08 h	98,32 h	56,35 h	54,91 h	86,14 h	
7	Projekt n	59,63 h	98,40 h	55,19 h	61,15 h	80,23 h	60,38 h	
8								
9								
10								

Abbildung 6.10: Durch Offset verschobener Range

Resize-Methode

Da `Resize` oft zusammen mit `Offset` eingesetzt wird, verlassen wir unsere alphabetische Ordnung und ziehen diese Methode vor.

Jeder Range besteht aus einer aktiven und einer oder mehreren selektierten Zellen. Nach Anwendung der `CurrentRegion` war A1 die aktive Zelle. Die Verschiebung des Range durch die `Offset`-Methode machte Zelle B2 zur aktiven Zelle. `Resize` wiederum dimensioniert ausgehend von dieser aktiven Zelle einen Range, der `nRows` Zeilen und `nCols` Spalten umfasst.

```
rngDat.CurrentRegion.Offset(1, 1). _
    Resize(nRows, nCols).ClearContents
```

Fehlt also nur noch die Zuweisung der Variablen `nRows` und `nCols`:

```
nRows = rngDat.CurrentRegion.Rows.Count - 2
nCols = rngDat.CurrentRegion.Columns.Count - 2
```

Der Subtrahend 2 ergibt sich aus der Anzahl der auszulassenden Kopf- und Fußzeilen, also 1 Kopfzeile plus eine Fußzeile.

Wenn `nRows` und `nCols` nicht noch an einer anderen Stelle benötigt werden, würde ich auf die Variablen verzichten und in der `Resize`-Klammer die Werte direkt ermitteln. Dieser Ausdruck wird dann 132 Zeichen umfassen, deshalb verzichten wir an dieser Stelle auf seine Darstellung.

Parent-Eigenschaft

Die `Parent`-Eigenschaft, die generell das dem aktuellen unmittelbar übergeordnete Objekt referenziert, gibt beim `Range`-Objekt das `Worksheet`-Objekt zurück. Das war ein schwerer Satz, doch nun kommt ein leichter Satz: Sie ist völlig überflüssig!

Seit VBA 5 besteht die Möglichkeit, jedem Worksheet im Eigenschaftsfenster der Entwicklungsumgebung einen Namen zuzuordnen (Abb. 6.11), der eigentlich CodeName heißt.

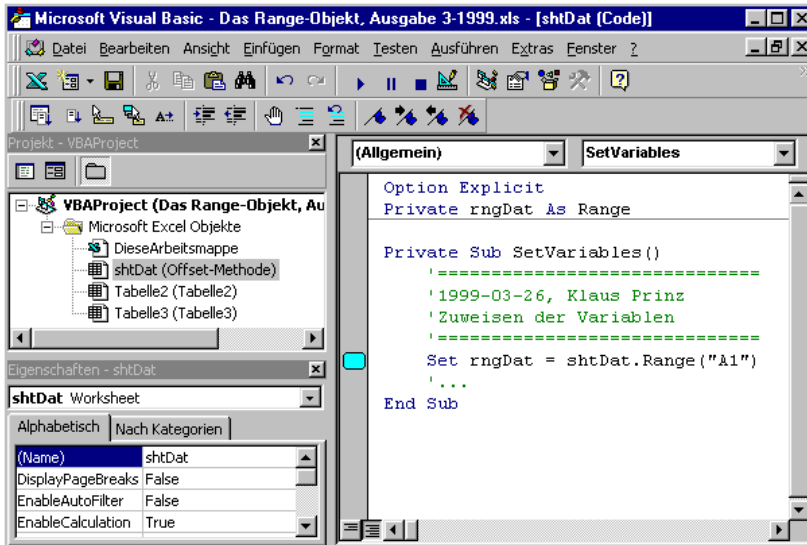


Abbildung 6.11: Verwendung der (Name)-Eigenschaft des Worksheets

Dieser CodeName steht ohne eine Zeile Code als Public-Variable des Typs Worksheet im gesamten Projekt zur Verfügung. Hierdurch reduziert sich die Zuweisung des Range-Objekts üblicherweise auf

```
Set rngDat = shtDat.Range("A1")
```

PasteSpecial-Methode

Diese Methode fügt ein Range-Objekt aus der Zwischenablage in einen Zellbereich ein. Das allein ist noch nichts Besonderes, aber diese Methode verfügt über eine Reihe von Argumenten, die festlegen, was eingefügt werden soll und auch wie.

Wenn Sie sich anschauen wollen, was die PasteSpecial-Methode alles kann, so kopieren Sie bitte eine Zelle in die Zwischenablage und rufen Sie den Menüpunkt Inhalte einfügen ... aus dem Bearbeiten-Menü auf.

Ich setze diese Methode lediglich dazu ein, Formate zu übertragen. Wenn Tabellen dynamisch aufgebaut werden, so füge ich zeilen- oder spaltenweise Formatierungen aus einer separaten Formate-Tabelle ein. Diese Formatzeile oder Spalte beinhaltet Zahlenformate, Hintergründe und Rahmen:

```
shtFormate.Rows(2).Copy
```

```
shtDaten.Rows(iRowDaten).PasteSpecial(xlPasteFormats)
```

Somit werden in lediglich zwei Programmzeilen umfangreiche Formatierungen erledigt.

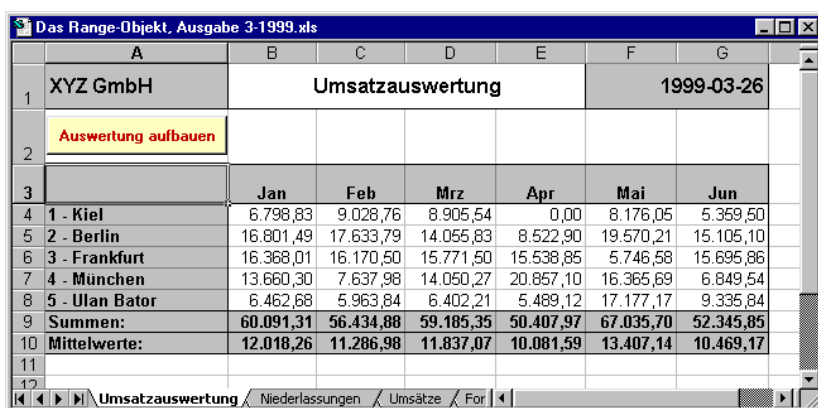
Range-Methode

Die Range-Methode des Range-Objekts treibt uns VBA-Entwickler scharenweise in die Hände der Monitorfabrikanten, denn Programmzeilen mit dieser Methode werden in der Regel sehr lang, wodurch das Verlangen nach 19-Zöllern enorm steigt.

Die Range-Methode erlaubt uns die Definition eines rechteckigen Zellbereichs durch die Angabe der linken oberen und rechten unteren Zelle. Hier das Prinzip:

```
rngDat.Range(Range1, Range2)
```

Der typische Fall, der auch diesem Beispiel zugrunde liegt, ist die Ermittlung arithmetischer oder statistischer Gesamtwerte in einer dynamischen Tabelle (siehe Abb. 10.12). Eine der letzten Aktionen dieses auf der CD beiliegenden Programms ist die Ermittlung der Mittelwerte in Zeile 10. Die Niederlassungen selbst werden dynamisch aufgebaut, wodurch wir also die Zeile 10 für die Mittelwerte nicht fest verdrahten können. Wir wissen nur, dass es die übernächste Zeile nach der letzten Datenzeile ist. Da auch die Summenwerte bereits eingetragen sind, können wir nicht die komplette Spalte berechnen, zumal wir mögliche Zahlen aus der Überschrift mit berechnen würden.



	A	B	C	D	E	F	G
1	XYZ GmbH	Umsatzauswertung				1999-03-26	
2	Auswertung aufbauen						
3		Jan	Feb	Mrz	Apr	Mai	Jun
4	1 - Kiel	6.798,83	9.028,76	8.905,54	0,00	8.176,05	5.359,50
5	2 - Berlin	16.801,49	17.633,79	14.055,83	8.522,90	19.570,21	15.105,10
6	3 - Frankfurt	16.368,01	16.170,50	15.771,50	15.538,85	5.746,58	15.695,86
7	4 - München	13.660,30	7.637,98	14.050,27	20.857,10	16.365,69	6.849,54
8	5 - Ulan Bator	6.462,68	5.963,84	6.402,21	5.489,12	17.177,17	9.335,84
9	Summen:	60.091,31	56.434,88	59.185,35	50.407,97	67.035,70	52.345,85
10	Mittelwerte:	12.018,26	11.286,98	11.837,07	10.081,59	13.407,14	10.469,17
11							
12							

Abbildung 6.12: Die fertige Umsatzauswertung

Um nun den Mittelwert aller Niederlassungen im März in Zelle D10 zu ermitteln, benötigen wir den Range D4:D8, etwas formaler ausgedrückt

```
D10 = Application.Sum(rngUAW.Range(D4, D8))
```

Die Ausdrücke D10, D4 und D8 stehen aber jeweils für ein vollständiges Range-Objekt des Zuschnitts

```
rngUAW.Cells(iRow, iCol)
```

und schon wird's lang:

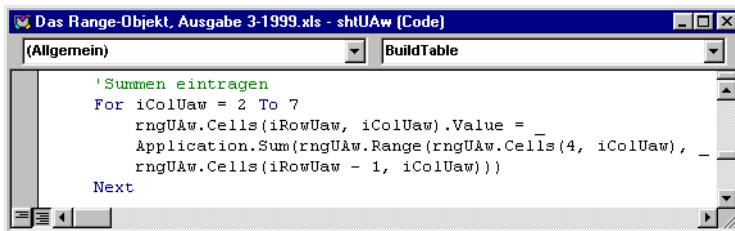


Abbildung 6.13: Die Verwendung der Range-Methode

Wenn Ihnen danach ist, können Sie eine solche Konstruktion durch die Verwendung von `Offset` und `Resize` vermeiden, aber es wird weder einfacher noch kürzer.

Row-Eigenschaft

Die `Row`-Eigenschaft gibt die absolute Zeile der aktiven Zelle des Range-Objekts zurück. Der Ausdruck

```
shtDat.Range("A3").Row
```

ergibt natürlich 3, aber auch der folgende:

```
shtDat.Range("A3").CurrentRegion.Row
```

denn nach der Anwendung der `CurrentRegion` ist A3 nach wie vor die aktive Zelle.

Rows-Auflistung

Die `Rows`-Auflistung gibt eine `Cells`-Auflistung zurück, die je eine Zelle aller durch das Range-Objekt betroffenen Zeilen enthält. Diese Zellen befinden sich in der Spalte, in der die aktive Zelle des der `Rows`-Auflistung voranstehenden Range-Objekts steht. `Range("B3:C10").Rows` wird insgeheim zu `Range("B3:B10")`.

Da diese Cells-Auflistung nun mal von jeder betroffenen Zeile genau eine Zelle beinhaltet, kann man die jeder Auflistung innewohnende Count-Methode verwenden, um festzustellen, über wie viele Zeilen sich der Range erstreckt. Diese dem beiliegenden Beispiel entnommene Zeile steuert die For-Next-Schleife von der zweiten zur letzten belegten Zeile:

```
For iRow = 2 To rngNd1.CurrentRegion.Rows.Count
```

Sort-Methode

Die Sort-Methode ist wie einige andere vollständig in Excel implementiert (Abb. 6.14). Alles, was Sie dort sehen, können Sie einstellen, müssen aber zum Glück nicht, denn es wird einem schon schwindlig, wenn man das erste Mal den 15 Argumenten ausgesetzt wird.

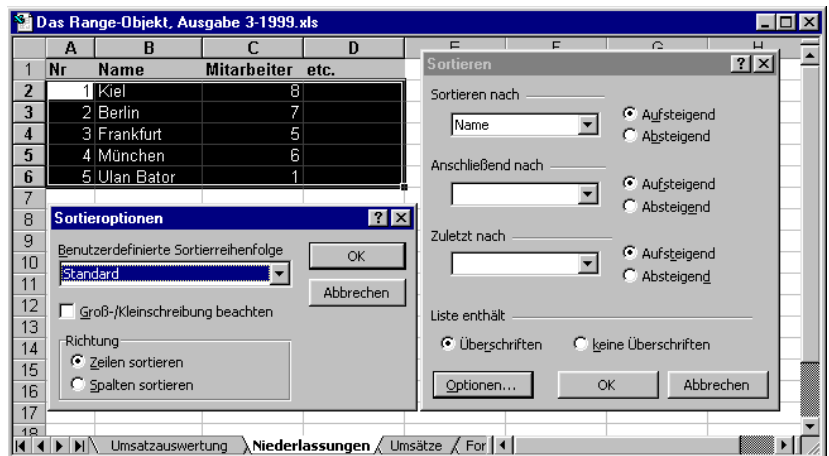


Abbildung 6.14: Die Implementierung der Sort-Methode in Excel

Fangen wir mit der von Microsoft fein säuberlich konstruierten Falle an. Wenn Sie in Excel die Funktion Sortieren im Menü Daten aufrufen, untersucht Excel den Tabellenbereich (genau genommen die `CurrentRegion` der aktiven Zelle) daraufhin, ob der Zellbereich eine Überschrift aufweist. Und das mit einem ausgefeilten Algorithmus mit einer ziemlich hohen Trefferquote. Diese Funktionalität findet sich in dem Argument `Header` wieder, die mit den Konstanten `x1Yes` für Ja, `x1No` für nein und `x1Guess` für »schau selber nach«. Sie ahnen's schon, der Standardwert für dieses Argument ist nicht `x1Guess` so wie in Excel, sondern `x1No`. Also, immer `x1Yes` angeben, wenn die Tabelle einen nicht mitzusortierenden, einzeiligen Kopf hat. In dem Wort einzeilig steckt die zweite Falle.

Sie haben eine Tabelle mit einer Überschriftzeile und wollen diese nach der ersten Spalte sortieren:

```
rngDat.Sort key1:=rngDat, header:=xlYes
```

Es gibt drei Key- und drei Order-Argumente, da wir nach maximal drei Spalten sortieren können. Der Standardwert für die Order ist `xlAscending`, was wir also weglassen dürfen. Wollen Sie hingegen nach einer Spalte absteigend sortieren, müssen Sie das betreffende Order-Argument angeben:

```
rngDat.Sort Key1:= rngDat, _  
            Key2:=rngDat.cells(1,2), _  
            Key3:= rngDat.cells(1,4), _  
            Order3:=xlDescending, _  
            Header:=xlYes
```

Interessant ist die Möglichkeit, selbst definierte AutoAusfüll-Reihen zugrunde zu legen (Register AutoAusfüllen der Optionen). Hierfür ist das Argument `OrderCustum` zuständig, das die laufende Nummer der Liste erwartet.

SpecialCells-Methode

Die `SpecialCells`-Methode, auf ein Range-Objekt angewandt, gibt einen Teil dieses Ranges zurück. Die Regeln, welcher Teil zurückgegeben wird und welcher nicht, stecken in der Konstante `Type`:

- ▼ `xlCellTypeNotes` (Zellen mit Notizen)
- ▼ `xlCellTypeConstants` (Zellen mit Konstanten)
- ▼ `xlCellTypeFormulas` (Zellen mit Formeln)
- ▼ `xlCellTypeBlank` (leere Zellen)
- ▼ `xlCellTypeLastCell` (letzte Zelle)
- ▼ `xlCellTypeVisible` (sichtbare Zellen)

Die ersten fünf Typen setze ich kaum ein, die letzte allerdings taucht in nahezu jedem Programm auf. Auch das Beispiel auf CD verwendet diesen Typ in einer Funktion, die in der Umsatztable die jeweiligen Teilumsätze je Niederlassung und Monat ermittelt und zurückgibt:

Listing 1: Funktion zur Ermittlung der Teilumsätze je Niederlassung und Monat

```
Private Function getUmsatz(lngNd1, lngMonat) As Double
    rngUms.AutoFilter field:=2, Criteria1:=lngNd1
    rngUms.AutoFilter field:=4, Criteria1:=lngMonat
    getUmsatz = Application.Sum(shtUms.Columns(3). _
        SpecialCells(xlCellTypeVisible))
End Function
```

Die beiden Konstanten `lngNd1` und `lngMonat` werden als Argumente des `AutoFilters` eingesetzt. Die Spalte 3 wird nun auf die sichtbaren Zellen reduziert der `Sum`-Methode übergeben.

Text-Eigenschaft

Die `Text`-Eigenschaft gibt den Inhalt der Zelle zurück, und zwar so, wie sie sich dem Auge des Betrachters präsentiert: mit allen Formatierungen. Ideal beispielsweise, um in einer `MsgBox` dem Anwender präsentiert zu werden.

Validation-Eigenschaft

Das von dieser Eigenschaft zurückgegebene `Validation`-Objekt entspricht dem, was unter dem Punkt Gültigkeit des Menüs Daten mit einem Zellbereich veranstaltet werden kann. Nachfolgend sehen Sie ein Code-Fragment, das einer Beispielprozedur entnommen ist, die bei der Behandlung des `Worksheet_Activate`-Ereignisses vorgestellt wird.

```
iLastValRow = _
    shtDaten.Range("G1").CurrentRegion.Rows.Count
shtDaten.Columns(3).Validation.Delete
shtDaten.Columns(3).Validation.Add _
    Type:=xlValidateList, _
    Formula1:="=$G$1:$G$" & iLastValRow
```

Das Löschen und Erzeugen eines `Validation`-Objekts ist einfacher als die nachträgliche Parametrierung, weshalb ich Ersteres vorziehe.

Value-Eigenschaft

Diese Standardeigenschaft des `Range`-Objekts gibt den Inhalt der Zelle zurück. Sie darf, wie jede Standardeigenschaft, weggelassen werden. Darf, sollte aber nicht.

6.2 Die Worksheets-Auflistung

Als Auflistung stellt Worksheets eine Art Container dar, in der alle Worksheet-Objekte der betreffenden Arbeitsmappe enthalten sind. Einige Eigenschaften und Methoden des Worksheet-Objekts sind auch in der Auflistung enthalten und gelten somit für alle Tabellen der betreffenden Arbeitsmappe. Hierunter fallen Copy, Move, Delete, Printout und Ähnliche. Bei Move, Delete und auch bei Visible gibt es zwangsläufig eine funktionale Einschränkung, denn wir dürfen einer Arbeitsmappe nicht die letzte Tabelle entziehen oder ausblenden. Befindet sich beispielsweise ein Diagrammblatt in der Arbeitsmappe, so werden die beiden problemlos funktionieren, denn dieses Diagrammblatt wird die Arbeitsmappe über den Verlust der Tabellen hinwegtrösten. Mit Count und Item stehen zwei Eigenschaften bereit, die jeder Worksheets-Auflistung eigen sind. Sie werden im Folgenden kurz vorgestellt.

Count-Eigenschaft

Count teilt uns mit, wie viele Elemente, also Worksheets, in der Worksheets-Auflistung enthalten sind.

```
MsgBox "Die Mappe enthält " & _
    ThisWorkbook.Worksheets.Count & " Tabellen."
```

Item-Eigenschaft

Diese Eigenschaft ermöglicht den Zugriff auf ein einzelnes Worksheet der Worksheets-Auflistung. Als Standardeigenschaft darf sie weggelassen werden:

```
ThisWorkbook.Worksheets.Item("Daten")
ThisWorkbook.Worksheets("Daten")
```

Die Auflistung kann in gewohnter Weise auch über einen Zeiger auf eine einzelne Tabelle durchlaufen werden, wobei hier insgeheim die Item-Eigenschaft zugange ist:

```
For iSheet = 1 to ThisWorkbook.Worksheets.Count
    MsgBox ThisWorkbook.Worksheets(iSheet).Name
Next
```

6.3 Das Worksheet-Objekt

Das Worksheet-Objekt ist Mitglied der Worksheets-Auflistung, die alle Tabellenblätter einer Arbeitsmappe, also eines Workbooks, umfasst. Das Worksheet-Objekt selber ist Container für eine Reihe von Objekten:

- ▼ ChartObjects (Diagramme)
- ▼ OLEObjects (Buttons, Listboxes etc.)

- ▼ Outline (Gliederung)
- ▼ PageSetup (Seiteneinstellungen)
- ▼ PivotTables (Pivot-Tabellen)
- ▼ QueryTables (Datenbankabfragen)
- ▼ Range (Range-Objekt)
- ▼ Scenarios (Szenarien)
- ▼ Shapes (Superklasse für alle on top von Tabellen platzierten (nicht nur OLE- bzw. ActiveX-) Elemente)

6.3.1 Eigenschaften und Methoden im Überblick

Typ	Element	Beschreibung	Rückgabe
M	Activate	Macht die Tabelle zur aktiven Tabelle der Arbeitsmappe	
P	Cells	Gibt alle Zellen der Tabelle zurück	Range
E	Columns	Erlaubt den Zugriff auf eine oder mehrere Spalten	Range
C	Comments	Beinhaltet alle Comment-Objekte der Tabelle	Comments
M	Copy	Kopiert die Tabelle	
M	Delete	Löscht die Tabelle	
M	Move	Verschiebt eine Tabelle	
E	Name	Name des Tabellenblattes	String
O	PageSetup	Zugriff auf druck-spezifische Einstellung	PageSetup
C	PivotTables	Enthält alle in der Tabelle befindlichen Pivot-Tabellen	PivotTables
E	Parent	Gibt das Workbook-Objekt zurück, in dem das Tabellenblatt liegt.	Workbook
M	PrintOut	Druckt das Tabellenblatt	
M	PrintPreview	Zeigt die Druckvorschau der Tabelle	
M	Protect	Schützt die Tabelle	
C	QueryTables	Enthält alle QueryTables (Datenbank-Auswahl-abfragen)	Query Tables
E	Range	Ermöglicht den Zugriff auf eine und (i. d. R.) mehrere Zellen	Range
C	Rows	Ermöglicht den Zugriff auf eine oder mehrere Spalten	Range
M	SaveAs	Speichern der Tabelle in einer Datei	

Typ	Element	Beschreibung	Rückgabe
C	Shapes	Zugriff auf alle Elemente, die in der Tabelle platziert wurden	Shapes
M	UnProtect	Hebt den Blattschutz auf	
E	UsedRange	Gibt den belegten Tabellenbereich zurück	Range
P	Visible	Steuert die Sichtbarkeit der Tabelle	Boolean

C = Auflistung, M = Methode, O = Objekt, P = Eigenschaft

6.3.2 Eigenschaften und Methoden unter der Lupe

Eigenschaften zur Entwurfszeit

Im Projektexplorer der Excel-Entwicklungsumgebung erscheinen unter der Kategorie Microsoft Excel Objekte alle die Komponenten, die Ereignisse produzieren können. Dazu gehören die Arbeitsmappe selbst sowie alle Tabellenblätter und separaten Diagramme. Wählen Sie aus dieser Liste eine Tabelle aus, so erscheinen im Eigenschaftsfenster die Eigenschaften, die zur Entwurfszeit verändert werden können (siehe Abb. 6.11).

(Name)- und Name-Eigenschaft

Die (Name)-Eigenschaft beinhaltet in VB und in VBA generell den Namen des Objekts, unter dem das Objekt direkt im Code angesprochen werden kann. Die Klammern dienen vermutlich nur dazu, diese Eigenschaft an den Anfang der sortierten Eigenschaftsliste zu rangieren. Objektnamen mit einem einheitlichen Präfix zu versehen, dürfte Allgemeingut geworden sein. Ich verwende `sht` als Präfix für Worksheet-Objekte, gefolgt von einem kurzen und aussagekräftigen Bezeichner. Lautet der Tabellenblattname beispielsweise »Statistik«, so würde das Worksheet-Objekt `shtStat` als (Name)-Eigenschaft erhalten. Nur um sicherzugehen: Sie benötigen weder einen `Dim` noch eine `Set`-Zuweisung, um mit der Objektvariablen arbeiten zu können. Sie steht sofort zur Verfügung! Mit einer kleinen Einschränkung: Der problemlose Zugriff auf die Tabelle setzt voraus, dass sie sich in derselben Arbeitsmappe befindet, in der auch der Code steht. Wollen Sie also auf ein Worksheet-Objekt einer anderen Arbeitsmappe zugreifen, so führt kein Weg an der klassischen Vorgehensweise vorbei:

```
Dim shtFremd As Worksheet
Set shtFremd = wbkFremd.Worksheets("Fremde Tabelle")
```

Dieses kleine Beispiel baut auf einer existierenden `Workbook`-Variablen auf. Das Öffnen eines Workbooks werden wir uns demnächst einmal vornehmen.

Weiter unten im Eigenschaftsfenster steht die Name-Eigenschaft, die den Namen des Tabellenblattes beinhaltet. Er kann dort verändert werden, ebenso wie in der Tabelle durch einen Doppelklick auf den Tabellenblatt-namen.

Zugriff mit Me

In dem gebundenen Modulblatt einer Tabelle steht uns durch das Schlüsselwort `Me` ein noch kürzerer Weg zum Zugriff auf das Tabellenblattobjekt zu, dem dieses Modulblatt gewissermaßen gehört. Heißt unser `Worksheet-Objekt` beispielsweise `Statistik (shtStat)`, so sind folgende Zeilen im gebundenen Modulblatt gleichbedeutend:

```
ThisWorkbook.Worksheets("Statistik").  
shtStat.  
Me.
```

Kennen Sie das? Das Telefon klingelt, Sie melden sich mit Ihrem Namen und vom anderen Ende her ertönt es »Ich bin's.« Nun setzt ein hektischer Stimmenvergleich ein, der dann meist zum gewünschten Ergebnis führt: »Ach, du bist's.«

Wenn Sie es nur mit einer per Code referenzierten Tabelle in Ihrem Projekt zu tun haben, dann ist der Verweis mit `Me` auf dieses Objekt in seinem gebundenen Modulblatt völlig ausreichend. Sind jedoch mehrere dieser Tabellen im Spiel, die möglicherweise in ihren Modulblättern wiederum Code aufweisen, so könnte das `Me` genauso schwierig zu interpretieren sein wie das »Ich bin's« in einem sehr großen Bekanntenkreis.

Visible-Eigenschaft

In der Liste der Eigenschaften des Eigenschaftsfensters verdient `Visible` noch Beachtung. Die beiden ersten Enumerationskonstanten `xlSheetVisible` gleich `-1` und `xlSheetHidden` gleich `0` entsprechen den Werten `True (-1)` und `False (0)` älterer Excel-Versionen. Sie stehen auch im Excel-Menü `Format` über `Ausblenden` und `Einblenden` zur Verfügung. Der dritte Wert `xlSheetVeryHidden` gleich `2` jedoch bedeutet, dass dieses Tabellenblatt auch nicht mehr in der Liste der einblendbaren Tabellen auftaucht, also tatsächlich »sehr versteckt« ist. Eine solchermaßen verborgene Tabelle lässt sich nur noch über VBA oder die Entwicklungsumgebung ans Licht bringen. Gegen die Veränderung dieser Eigenschaft von einem fremden VBA-Projekt aus ist kein Kraut gewachsen; wenn Sie jedoch das VBA-Projekt per Passwort geschützt haben, kommt zumindest der »normale« Anwender nicht mehr an diese Tabelle heran, sofern er von ihrer Existenz nichts weiß. Kennt er jedoch den Namen dieser versteckten Tabelle, so kann er jederzeit über `=VersteckteTabelle!A1` darauf zugreifen. Wollen Sie Informationen wirklich sicher vor Anwenderzugriffen schützen, so bieten sich hierfür die Registry oder passwort-geschützte Datenbanken an.

Worksheet und Range

Es gibt viele Methodiken des Zellzugriffs, bei denen es letztendlich egal ist, ob man ihnen ein Range-Objekt voranstellt oder ein Worksheet-Objekt. Ihnen ist die Cells-Eigenschaft sicherlich noch geläufig, die wir bei der Behandlung des Range-Objekts vorstellten. Sie ermöglicht einen Zugriff auf eine einzelne Zelle, die an einer bestimmbar, relative Position zum vorangestellten Range-Objekt steht. Es stellt sich nun die Frage, an welcher Position das Range-Objekt angenommen wird, zu dem die relative Position festgelegt wird. Es ist logischerweise die Zelle A1 :

```
Dim rngStat As Range
Set rngStat = shtStat.Range("A1")
rngStat.Cells(4, 3).Value = "Hallo"
```

Der folgende Ausdruck erzeugt dieselbe Wirkung:

```
shtStat.Cells(4, 3).Value = "Hallo"
```

Da die Cells-Eigenschaft immer ein Range-Objekt erzeugt, steht Ihnen also auch ohne die explizite Verwendung eines Range-Objekts die komplette Bandbreite dieses Objektes dadurch zur Verfügung, dass Sie die Cells-Eigenschaft auf das Worksheet-Objekt anwenden.

Neben der Cells-Eigenschaft können die Elemente Rows(), Columns(), Range(), und Comments() mit demselben Resultat einem Worksheet-Objekt folgen.

Activate-Methode

Es zeugt von gutem Stil, am Ende eines Programmes die Ergebnistabelle in den Vordergrund zu bringen und nach Möglichkeit auch den Cursor in die erste Datenzeile zu positionieren. Ist die gewünschte Tabelle aktiviert, so können Sie hierzu einfach das betreffende Range-Objekt mittels der Activate-Methode ansprechen. Ist die zugehörige Tabelle jedoch nicht aktiv, muss sie zuvor aktiviert werden. Das folgende Beispiel produziert ohne die zweite Zeile einen Fehler:

```
Workbooks("Mappel.xls").Activate
shtStat.Activate
shtStat.Range("B2").Activate
```

Cells-Eigenschaft

Genau genommen gibt die Cells-Eigenschaft auf das Worksheet-Objekt angewendet alle 2^{24} (2^{16} Zeilen x 2^8 Spalten) Zellen zurück. Wir verwenden Sie jedoch in der Regel in einer der beiden folgenden Syntaxvarianten:

```
shtstat.Cells(2, 1).Value = "Variante 1"
```

oder

```
shtstat.Cells(257).Value = "Variante 2"
```

Beide Codezeilen schreiben wir übrigens in Zelle B1. Aus der in beiden Fällen erzeugten `Cells`-Auflistung wird nun mittels der `Item`-Methode jeweils eine Zelle entnommen. Eigentlich müssten beide Zeilen so geschrieben sein:

```
shtstat.Cells.Item(2, 1).Value = "Variante 1"
```

bzw.

```
shtstat.Cells.Item(257).Value = "Variante 2"
```

Im ersten Fall werden je ein Zeilen- und ein Spaltenindex angegeben, der Zellen der zweiten Zeile und der ersten Spalte spezifiziert, also B1. Der zweite Ausdruck entnimmt die 257. Zelle der `Cells`-Auflistung. Da die Tabelle zeilenweise organisiert ist und über 256 Spalten verfügt, ist eben die 257. Zelle die erste Zelle in der zweiten Zeile, also B1. Nun ist aber jedes Element der `Cells`-Auflistung ein `Range`-Objekt und die `Item`-Methode die Standard-Methode des `Range`-Objekts. Somit darf das Schlüsselwort `Item` nebst dem vorangestellten Punkt weggelassen werden.

Es ist ein großer Vorteil von VBA, dass man recht viel weglassen darf. Man sollte aber jederzeit wissen, welche Annahme VBA dann an Stelle des Weggelassenen trifft.

Columns-Eigenschaft

Die `Columns`-Eigenschaft gibt 256 `Range`-Objekte mit jeweils 65536 Zellen zurück. Ahnen Sie es schon? Wenn Sie die `Columns`-Eigenschaft einsetzen, so ist zumeist auch die dann weggelassene `Item`-Methode des `Range`-Objekts zugange, denn

```
shtStat.Columns(1).ClearContents
```

wird intern als

```
shtStat.Columns.Item(1).ClearContents
```

interpretiert. Auch die beiden folgenden Anweisungen sind identisch:

```
shtStat.Columns.Item("A:B").ClearContents
```

```
shtStat.Columns("A:B").ClearContents
```

denn die `Item`-Methode ist (wie flächendeckend üblich in VB und VBA) in der Lage, mit einem Index und einem Key zu arbeiten. Bei `Columns(1)` tritt der Index zutage, bei `Columns("A:B")` ein Algorithmus, der einem Key nahe kommt.

In welche Zelle schreibt folgender Ausdruck:

```
shtStat.Columns(2).Cells(2, 2).Value = "Such mich"
```

Oder vollständig:

```
shtStat.Columns.Item(2).Cells.Item(2,2).Value = "Such mich"
```

Hier wirkt noch ein anderer Mechanismus im Hintergrund mit. Jeder Zellbereich besteht aus einer aktiven und einer oder mehreren selektierten Zellen. Einige Eigenschaften und Methoden wirken auf alle selektierten Zellen, z. B. `ClearContents`, `Interior` oder `Value`. Die Methode `Item(2, 2)` interessiert sich nicht mehr für die selektierten Zellen, sondern adressiert ausgehend von der aktiven Zelle der Spalte B (`Columns.Item(2)`, also B1, die Zelle C2).

Wenn Ihnen das alles klar ist, dann haben Sie das Prinzip der Verwaltung von Collections verstanden und somit auch die `Range`-Klasse und ihre `Collection`-Klasse `Worksheet`. Excel ist dann ein offenes Buch für Sie.

Comments-Auflistung

`shtStat.Comments` enthält alle `Comment`-Objekte, die in der Tabelle `shtStat` enthalten sind. Ein `Comment`-Objekt repräsentiert den Kommentar einer einzelnen Zelle. Zu dem `Comment`-Objekt gibt es nicht sonderlich viel zu sagen, außer vielleicht, dass es seinen eigentlichen Inhalt nicht in Form einer Eigenschaft, sondern einer Methode verwaltet. Eigentlich ist das recht geschickt, weil über eine Eigenschaft nur ein Wert übergeben werden könnte, eben der Text. Die Methode hält jedoch noch die optionalen Argumente `Start` und `Overwrite` bereit, mit der einige Möglichkeiten mehr bereitstehen als nur den Text zu überschreiben, was beim Weglassen der Argumente passiert. Nachteilig ist nur, dass dies die einzig mir bekannte Methoden-Implementierung eines Textes ist, über die man regelmäßig stolpert.

Copy- und Move-Methoden

`Copy` und `Move` ermöglichen das Kopieren und Verschieben einer Tabelle innerhalb der Arbeitsmappe oder in eine neue Arbeitsmappe. Die folgende Anweisung kopiert die Tabelle `shtStat` in eine neue Arbeitsmappe.

```
shtStat.Copy
```

Sowohl `Copy` als auch `Move` verfügen über die beiden optionalen Argumente `before` und `after`, denen jeweils ein `Workbook`-Objekt übergeben werden kann. Die folgende Anweisung erstellt eine Kopie der Tabelle `shtStat`, die vor der Tabelle `shtStat` eingefügt wird:

```
shtStat.Copy before:=shtStat
```

Logischerweise kann nur eines der beiden Argumente `before` und `after` angegeben werden.

Ein typischer Anwendungsfall liegt übrigens im Erzeugen von zu mailenden Exportdateien, in denen nur eine oder zwei Tabellen enthalten sind. Wenn aber die `Copy`-Methode das Tabellenblatt jeweils in eine neue Datei kopiert, so stellt sich die Frage, wie man denn zwei Tabellenblätter in eine Datei hineinfabriziert? So:

```
ThisWorkbook.Worksheets(Array("Tabelle1", "Tabelle2")).Copy
```

Sie sehen also, die `Worksheets`-Auflistung verfügt ebenfalls über eine `Copy`- (und eine `Move`-)Methode.

Delete-Methode

Die `Delete`-Methode löscht das voranstehende `Worksheet`-Objekt. Auch hier können Sie eine Gruppe von `Worksheets` nach folgendem Muster löschen:

```
ThisWorkbook.Worksheets(Array("Table1", "Table2")).Delete
```

Denken Sie bitte daran, folgende Zeile voranzustellen, wenn Sie nicht wollen, dass der Anwender mit der üblichen Warnmeldung von Excel konfrontiert werden soll:

```
Application.DisplayAlerts = False
```

PageSetup-Objekt

Dieses Objekt bietet Ihnen den Zugang zu allen Einstellungen, die unter Seite einrichten des Datei-Menüs angeboten werden. Damit ist es beispielsweise möglich, den vollständigen Namen der Arbeitsmappe mit Pfad in die Fußzeile zu schreiben, was für den VBA-unkundigen Anwender nicht möglich ist.

Es wäre wünschenswert, wenn `PageSetup` ein sauber strukturiertes Objektmodell erhielte. Dann erwartete uns vielleicht eine Konstruktion, die so aussehen könnte:

```
shtStat.PageSetup(xlLeftFooter).Font.Bold = True
```

```
shtStat.PageSetup(xlLeftFooter).Text = _
    ThisWorkbook.Path & "\" & ThisWorkkook.Name
```

Werfen wir nun einen Blick auf die Wirklichkeit. `LeftFooter` existiert zwar, aber nicht als Objekt, sondern als Eigenschaft des `PageSetup`-Objekts. Pfad und Dateiname können wir somit erzeugen:

```
shtStat.PageSetup.LeftFooter = _
    ThisWorkbook.Path & "\" & ThisWorkkook.Name
```

Doch nun zur Formatierung. Die Hilfe beschreibt eine Reihe von Formatcodes, die der `LeftFooter`-Eigenschaft übergeben werden. So zum Beispiel `&F` für fett oder `&K` für kursiv. Somit könnten wir also dieser Eigenschaft den String `»&F« & ThisWorkbook.Path` etc. zuweisen. Als Ergebnis sehen Sie allerdings den Dateinamen vor dem Pfad: `»Test.xlsC:\Test\Test.xls«`. Ahnen Sie es schon? `&F` steht nicht für fett, sondern für File, womit eine fette Formatierung logischerweise mit `&B` für bold (das englische Synonym für fett) eingeleitet werden muss. Unsere fette Fußzeile mit vollständigem Dateinamen sieht also so aus:

```
shtStat.PageSetup.LeftFooter = _
    "&B" & ThisWorkbook.Path & "\" & ThisWorkkook.Name
```

Was wird aus `»Bits&Bytes«`? `»Bitsytes«`, da das B von Bytes als Teil des Formatcodes aufgefasst wird. OK, was ist mit `»Bits & Bytes«`? Das dem `&` folgende Leerzeichen hat (zum Glück) keine Bedeutung, die Seitenansicht zeigt aber `»Bits Bytes«` an. Sie müssen also zwei `&` hintereinander schreiben, damit eines durchkommt: `»Bits && Bytes«` wird zu `»Bits & Bytes«`.

PivotTables-Auflistung

Pivot-Tabellen sind eine vielseitige und komplexe Funktionalität von Excel. Zugriff auf eine einzelne Pivot-Tabelle haben Sie über den Index, der sowohl einen Index als Long-Datentyp entgegennimmt, als auch den Namen der Pivot-Tabelle akzeptiert, so wie es generell bei Auflistungen üblich ist:

```
shtStat.PivotTables(1)
```

Oder

```
shtStat.PivotTables("PT1")
```

Ob es sinnvoll ist, eine komplexe Pivot-Tabelle per Code zu generieren, muss letztendlich jeder für sich beantworten. Ich persönlich ziehe es vor, die Pivot-Tabelle manuell zu erstellen und lediglich die `SourceData`-Eigenschaft auf möglicherweise geänderte Ausgangsdaten anzupassen:

```
shtStat.PivotTables(1).SourceData = "Daten!A1:C75"
```

Den Zellbereich können Sie jederzeit unter Zuhilfenahme der `CurrentRegion` selbst erzeugen:

```
strSourceData = "Daten!" & shtDat.CurrentRegion.Address
```

Das wohl wichtigste Betätigungsfeld für VBA bei Pivot-Tabellen liegt aber zweifellos in der Aktualisierung der Tabelle. Hierzu kann uns das `Activate`-Ereignis des Worksheets gute Dienste leisten. Nehmen wir an, Ihre Pivot-Tabelle befände sich in Tabelle `shtPivot`. Wählen Sie in dem gebundenen Modulblatt dieser Tabelle aus der linken `ComboBox` den Eintrag `Worksheet` und in der rechten `ComboBox` »Activate«. Die Entwicklungsumgebung erzeugt nun den Prozedurrahmen, den Sie wie folgt ergänzen:

```
Private Sub Worksheet_Activate()  
    shtPivot.PivotTables(1).RefreshTable  
End Sub
```

Damit haben Sie den größten Mangel der Pivot-Tabellen ausgebügelt.

PrintOut- und PrintPreview-Methoden

Die Zeile

```
shtPivot.PrintPreview
```

zeigt die Seitenansicht Ihrer Tabelle `shtPivot`. Der Anwender kann danach selbst entscheiden, ob er drucken will oder nicht.

Die `PrintOut`-Methode bietet eine Reihe von benannten Argumenten, die im Prinzip selbst erklärend sind:

```
shtPivot.PrintOut From:=1, To:=3, Copies:=2, _  
    ActivePrinter:="\\NTDC\HP LaserJet 5L"
```

Die Seiten 1 bis 3 der Tabelle `shtPivot` werden je zweimal auf dem angegebenen Drucker ausgedruckt.

Protect- und UnProtect-Methoden

Diese beiden Methoden bilden den aus dem Extras-Menü bekannten Blattschutz nach. Seit Service Release 2 von Office 97 ist auch der Passwortschutz »dicht«.

```
shtstat.Protect Password:="Sicher"
```

sichert die Tabelle `shtStat` und

```
shtstat.UnProtect Password:="Sicher"
```

hebt diesen Schutz wieder auf.

QueryTables-Auflistung

Hinter dem QueryTable-Objekt steckt der MS-Query, ein alter Weggefährte unseres Excel, ein bisschen in die Jahre gekommen. Für Anwender mit wenig Datenbankkenntnissen ist er eine gute Hilfe. Ob er aber ein gutes Werkzeug für versierte und ambitionierte VBA-Entwickler darstellt, ist eine andere Frage.

Dennoch ermöglicht er mit wenigen Programmzeilen komplette Datenbankabfragen mitsamt Überschriften in eine Excel-Tabelle einzulesen:

```
Sub AddQueryTable()
    Dim ws As Workspace
    Dim db As Database
    Dim rs As Recordset
    Dim strSQL As String
    Dim strDBName As String
    strSQL = "SELECT DISTINCT Zeitraum FROM tbPersDaten"
    Set ws = DBEngine.Workspaces(0)
    Set db = ws.OpenDatabase("D:\Test\Epos.mdb")
    Set rs = db.OpenRecordset(strSQL, dbOpenSnapshot)
    shtStat.QueryTables.Add Connection:=rs, _
        Destination:=shtStat.Range("A1")
        shtStat.QueryTables(1).Refresh
End Sub
```

Diese kleine Routine ist unprofessionell, dient aber der Demonstration des QueryTable-Objekts. Anstelle des Recordset-Objekts rs als Connection-Argument kann auch ein ODBC-Connect-String angegeben werden. Da dies aber in der Regel recht unhandlich ist, wurde hier ein Recordset-Objekt verwendet.

Wichtig ist übrigens auch die Refresh-Methode, die den Datentransfer erst initiiert; ohne sie bliebe die Tabelle leer.

Range-Eigenschaft

Die Range-Eigenschaft erzeugt ein Range-Objekt in den folgenden beiden Syntaxvarianten:

```
shtStat.Range("A1:B20")
```

oder gleichbedeutend

```
shtStat.Range(shtStat.Cells(1, 1), shtStat.Cells(20, 2))
```

Im ersten Falle steht ein Zellbezug, wie er vielfach in Excel zur Anwendung kommt. Das zweite Beispiel übergibt an die `Range`-Eigenschaft zwei `Range`-Objekte, die die obere linke und untere rechte Ecke des zukünftigen Ranges beschreiben. »A1:B20« ist dann natürlich nahe liegend, wenn der Zellbereich bereits zur Entwurfszeit bekannt ist und möglichst unverändert bleibt. Ist dieser Bereich jedoch dynamisch, ist Variante 2 attraktiver, da die `Row`- und `Column`-Parameter der beiden `Cells`-Eigenschaften über `Long`-Zeiger gebildet werden können. Natürlich wäre dies selbst mit einem dynamischen Zellbereich in Variante 1 möglich, denn der Spaltenbezeichner »B« kann auch aus `Chr(64 + 2)` gebildet werden, doch ab Spalte AA wird es stark algorithmisch.

Rows-Eigenschaft

Über die `Rows`-Eigenschaft haben Sie Zugriff auf eine bis alle Zeilen einer Tabelle, die in Form von `Range`-Objekten zu je 256 Zellen zurückgegeben werden. Im Übrigen gelten die Ausführungen, die unter `Columns` zu finden sind.

SaveAs-Methode

Die `SaveAs`-Methode des `Worksheet`-Objekts ermöglicht Ihnen die betreffende Tabelle in einem fremden Dateiformat abzuspeichern. Vielfältige Argumente stehen für Passwortschutz, Nur-Lese-Empfehlung oder Zugriffsmodus zur Verfügung.

Wichtig aber sind die beiden ersten `Filename` und `FileFormat`. Ersteres erwartet den vollständigen Dateinamen nebst Pfad. Mit den `FileFormat`-Konstanten legen Sie fest, in welchem Format die Datei gespeichert werden soll. In der Regel werden CSV-Dateien erzeugt, die beispielsweise in Buchhaltung oder Materialwirtschaft Import werden können. Mit `xlCSV` erzeugen Sie eine `Comma Separated Values (CSV)`-Datei, deren Spalten mit Komma getrennt sind, `xlTestWindows` erzeugt eine Datei mit Tabulatoren als Trennzeichen. In beiden Fällen werden eventuell vorhandene Dezimalkommata in Dezimalpunkte umgewandelt:

```
ShtStat.SaveAs Filename:="D:\Test\Test.csv", _
               FileFormat:=xlCSV
```

Shapes-Auflistung

Über die `Shapes`-Auflistung haben Sie Zugang zu allem, was nicht zu einem leeren Tabellenblatt gehört: `Textboxes`, graphische Elemente wie Rechtecke, Kreis oder Linien, alle in eine Tabelle platzierbaren `ActiveX`-Controls, also `CommandButtons`, `CheckBoxes` etc, aber auch Diagramme und natürlich auch die alten Steuerelemente aus der Formular-Symbolleiste.

Ein Beispiel. Sie haben eine Tabelle `shtStat`, in der eine Statistik aufgebaut wird. Die Tabelle selbst verfügt über zwei `CommandButtons`, ersterer zum Aufbau der Statistik und der zweite, um die Tabelle in eine zu mailende Exportdatei zu kopieren. Beim Kopieren der Tabelle werden die beiden `CommandButtons` ebenfalls in die Exportdatei verfrachtet. Mit einer kleinen Schleife können wir diese in der Exportdatei natürlich funktionslosen Schaltflächen entfernen:

```
Sub RemoveButtons()
    Dim i As Long
    For i = sht.Shapes.Count To 1 Step -1
        sht.Shapes(1).Delete
    Next
End Sub
```

Beachten Sie bitte, dass die Schleife rückwärts durchlaufen werden muss, denn wenn Sie das Element 1 löschen, wird Element 2 zu 1. Danach inkrementiert der Zeiger `i` auf 2 und das ehemalige zweite Element ist Ihnen prompt durch die Lappen gegangen. Im Übrigen läuft die Routine nach der Hälfte der ursprünglichen Elemente auf einen Fehler.

UsedRange-Eigenschaft

Sie erinnern sich sicherlich noch an die `CurrentRegion` des `Range`-Objekts, die ausgehend von der aktiven Zelle des `Range` den zusammenhängend belegten Zellbereich selektiert. Das Wort zusammenhängend beschreibt auch die Grenze dieser `CurrentRegion`, denn sie vernachlässigt alle Zellen, die von diesem Bereich durch eine durchgehende Reihe von leeren Zellen begrenzt ist.

	A	B	C	D
1	Spalte 1	Spalte 2	Spalte 3	Spalte 4
2	Daten	Daten	Daten	Daten
3	Daten	Daten	Daten	Daten
4	Daten	Daten	Daten	Daten
5	Daten	Daten	Daten	Daten
6	Daten	Daten	Daten	Daten
7	Daten	Daten	Daten	Daten
8	Daten	Daten	Daten	Daten
9	Daten	Daten	Daten	Daten
10	Daten	Daten	Daten	Daten
11	Daten	Daten	Daten	Daten
12	Daten	Daten	Daten	Daten
13	Daten	Daten	Daten	Daten
14	Daten	Daten	Daten	Daten
15				
16	Fußzeile in der Tabelle			
17				

Abbildung 6.15: Wirkungsbereich der `CurrentRegion`

In Abb. 6.15 ist in einer Beispieltabelle der Bereich dargestellt, der von der `CurrentRegion` erfasst wird. Die stilisierte Fußzeile bleibt ausgeschlossen. Der `UsedRange` des Worksheets hingegen beinhaltet auch die abgesetzte Fußzeile, wie in Abb. 6.16 zu sehen ist.

	A	B	C	D
1	Spalte 1	Spalte 2	Spalte 3	Spalte 4
2	Daten	Daten	Daten	Daten
3	Daten	Daten	Daten	Daten
4	Daten	Daten	Daten	Daten
5	Daten	Daten	Daten	Daten
6	Daten	Daten	Daten	Daten
7	Daten	Daten	Daten	Daten
8	Daten	Daten	Daten	Daten
9	Daten	Daten	Daten	Daten
10	Daten	Daten	Daten	Daten
11	Daten	Daten	Daten	Daten
12	Daten	Daten	Daten	Daten
13	Daten	Daten	Daten	Daten
14	Daten	Daten	Daten	Daten
15				
16	Fußzeile in der Tabelle			
17				

Abbildung 6.16: Wirkungsbereich des `UsedRange`

So können Sie beispielsweise vor Beginn des Aufbaus einer Tabelle die »alten« Daten vollständig löschen:

```
shtStat.UsedRange.Clear
```

Wenn in Zeile 1 des Tabellenblattes eine Zelle belegt ist, so liefert Ihnen

```
ILastRow = shtStat.UsedRange.Rows.Count
```

sowohl die Anzahl der belegten Zeilen als auch die Nummer der letzten Zeile. Sind im Tabellenkopf hingegen freie Zeilen und Sie wollen die letzte belegte Zeile dennoch ermitteln, haben Sie zwei Möglichkeiten. Sie können beispielsweise in die Zelle A1 ein Blank schreiben, bevor Sie den `UsedRange` einsetzen. Die etwas komplizierter aussehende Variante besteht darin, aus dem `UsedRange` die letzte belegte Zelle mittels `SpecialCells` zu ermitteln:

```
iLastRow = _
```

```
shtStat.UsedRange.SpecialCells(xlCellTypeLastCell).Row
```

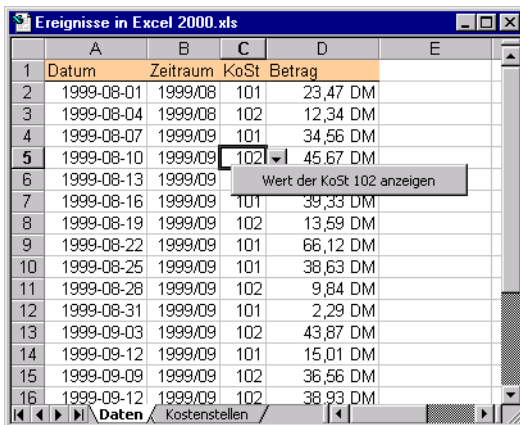
Anstelle von `UsedRange` ginge übrigens auch `Cells`, womit alle 16 Millionen Zellen eingeschlossen sind.

6.3.3 Ereignisse

Das Worksheet-Objekt besitzt die äußerst nützliche Fähigkeit, mit uns Entwicklern in Kontakt zu treten, um uns über eine Reihe ausgewählter Aktionen des Anwenders in Kenntnis zu setzen:

Ereignis	Beschreibung
Activate	tritt auf, wenn die Tabelle zur aktiven Tabelle wird
BeforeDoubleClick	tritt auf, wenn ein Doppelklick auf der Tabelle erfolgt
BeforeRightClick	tritt auf, wenn die rechte Maustaste gedrückt wird
Calculate	tritt auf, wenn die Tabelle berechnet wird
Change	tritt auf, wenn eine eben veränderte Zelle verlassen wird
Deactivate	tritt auf, wenn die Tabelle nicht mehr die aktive Tabelle der Arbeitsmappe ist
FollowHyperlink	tritt auf, wenn ein Hyperlink in der Tabelle angeklickt wird
SelectionChange	tritt auf, wenn sich die Zellauswahl der Tabelle ändert

Auf der beiliegenden CD finden Sie die Beispieldateien »Ereignisse in Excel 2000.xls«, die eine recht einfache Anwendung beinhaltet. Gegenstand dieser Anwendung ist eine Kostenstellenerfassung, die aus den Tabellen Daten und Kostenstellen besteht. In ersterer (siehe Abb. 6.17) werden Datenzeilen mit Datum, Kostenstellennummer und zu belastendem Umsatz erfasst.



	A	B	C	D	E
1	Datum	Zeitraum	KoSt	Betrag	
2	1999-08-01	1999/08	101	23,47 DM	
3	1999-08-04	1999/08	102	12,34 DM	
4	1999-08-07	1999/09	101	34,56 DM	
5	1999-08-10	1999/09	102	45,67 DM	
6	1999-08-13	1999/09			
7	1999-08-16	1999/09	101	39,33 DM	
8	1999-08-19	1999/09	102	13,59 DM	
9	1999-08-22	1999/09	101	66,12 DM	
10	1999-08-25	1999/09	101	38,63 DM	
11	1999-08-28	1999/09	102	9,84 DM	
12	1999-08-31	1999/09	101	2,29 DM	
13	1999-09-03	1999/09	102	43,87 DM	
14	1999-09-12	1999/09	101	15,01 DM	
15	1999-09-09	1999/09	102	36,56 DM	
16	1999-09-12	1999/09	102	38,93 DM	

Abbildung 6.17: Tabelle Daten

In der Tabelle Kostenstellen (siehe Abb. 6.18) werden diese Umsätze je Kostenstelle verdichtet. Außerdem dient diese Tabelle zur Definition der gültigen Kostenstellen.

	A	B	C	D
1	KoSt	Name	Umsatz	
2	101	Kostenstelle 101	271,28 DM	
3	102	Kostenstelle 102	200,80 DM	
4	103	Kostenstelle 103	78,96 DM	
5				

Abbildung 6.18: Tabelle Kostenstellen

Zur Vorgehensweise

Beim Programmstart wird die Tabelle Daten aktiviert und in dieser wiederum in die nächste freie Zeile die Zelle der Spalte A. In dieser Zelle wird ein Datum eingegeben. Eine Ereignisroutine errechnet daraus einen Zeitraum, der in Spalte B eingetragen wird.

Der Cursor wird anschließend in Spalte C positioniert, in der alle definierten Kostenstellen aus der Tabelle Kostenstellen über eine Zell-DropDown-Liste abgerufen werden können. Dahinter verbirgt sich der Mechanismus, der als Gültigkeit im Menü Daten zur Verfügung steht. Dieser Mechanismus vermag allerdings keine Quellbezüge außerhalb seiner eigenen Tabelle zu verarbeiten, weshalb ihm über VBA ein wenig auf die Sprünge geholfen wird. Die Spalte C enthält außerdem ein eigenes Kontextmenü, das in Abb. 6.17 die Aufschrift »Wert der KoSt 102 anzeigen« trägt. Wird dieser Menüpunkt angeklickt, wird der Kostenstellenumsatz vor Beginn der Änderung in einer MessageBox angezeigt (siehe Abb. 6.19).

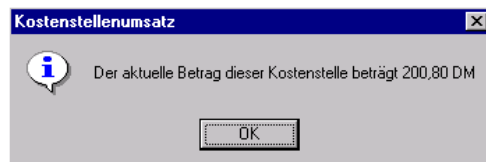


Abbildung 6.19: MessageBox nach Anklicken des Kontextmenüs

In Spalte D der Tabelle Daten werden letztlich die betreffenden Kostenstellenumsätze eingetragen.

Wird die Tabelle Daten verlassen, werden die in Tabelle Kostenstellen gezeigten aktuellen Kostenstellenumsätze neu berechnet.

Wie gesagt, nichts Großartiges, aber ein ideales Betätigungsfeld für die verschiedensten Ereignisse, die uns das Worksheet-Objekt zur Verfügung stellt.

Realisierung

Ein kleine Anleihe müssen wir beim Workbook-Objekt machen, da uns dieses über das Open-Ereignis davon in Kenntnis setzen kann, dass die Arbeitsmappe geöffnet wurde. Denn da müssen wir einhaken, um das Kontextmenü zu erzeugen und die Tabelle Daten zu aktivieren.

```
Private Sub Workbook_Open()
    Dim mnu As CommandBar           'Menü-Objekt
    Dim ctl As CommandBarControl    'Menüeintrag
    On Error GoTo errHandler
    Set mnu = Application.CommandBars.Add( _
        Name:="mnuPopUpKoSt", _
        Position:=msoBarPopup, _
        MenuBar:=False, _
        temporary:=True)
    Set ctl = mnu.Controls.Add(Type:=msoControlButton)
    Exit Sub

errHandler:
    Err.Source = "Workbook_BeforeClose"
    cError.ErrorMessage
End Sub
```

Es werden also zuerst die beiden Objektvariablen erzeugt. Danach wird über die Add-Methode der CommandBars-Auflistung ein neues PopUp-Menü des Namens mnuPopUpKoSt erzeugt, dem anschließend mit der Add-Methode der Controls-Auflistung ein einzelner Eintrag angefügt wird. Da Excel 97 keine Variablenerzeugung mit WithEvents- also: erzeuge eine Objektinstanz und stelle mir die Ereignisse dieser Klasse zur Verfügung, -zustande bringt, muss in Excel 97 vor der Exit Sub Anweisung noch eine Zeile eingefügt werden, in der diesem Control eine Prozedur zugewiesen wird:

```
ctl.OnAction = "shtDaten.ShowKoStValue"
```

In Excel 2000 geht das inzwischen eine Spur eleganter.

Wenn wir beim Öffnen der Datei eine CommandBar erzeugen, so müssen wir diese beim Schließen der Arbeitsmappe auch wieder löschen. Andernfalls liefе logischerweise das Erzeugen der CommandBar beim nächsten Öffnen auf einen Fehler hinaus:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    On Error GoTo errHandler
    Application.CommandBars("mnuPopUpKoSt").Delete
    Exit Sub
```

```
errHandler:
    Err.Source = "Workbook_BeforeClose"
    cError.ErrorMessage
End Sub
```

Die Tabelle **Daten** erhält den Namen `shtDaten`, die Tabelle **Kostenstellen** `shtKoSt`. Hieraus leiten sich die Namen für die Range-Objekte als `rngDaten` und `rngKoSt` bereits zwangsläufig ab.

Nun zu dem Code im gebundenen Modulblatt der Tabelle `shtDaten`:

```
Option Explicit
Private bForgetIt As Boolean
Private rngKoSt As Range
Private rngDaten As Range
Private WithEvents mnuPopUpKoSt As CommandBarButton
```

Neben den beiden Range-Objekten kommt noch die boolesche Variable `bForgetIt` zum Einsatz, die ich immer dann verwende, wenn Ereignisprozeduren in gewissen Fällen unterdrückt werden müssen. Weiter unten werden Sie noch sehen, wie das funktioniert.

In der letzten Zeile des Modulkopfs wird das Objekt `mnuPopUpKoSt` deklariert, welches in der Prozedur des nachfolgenden `Activate`-Ereignisses instantiiert wird.

Activate-Ereignis

Dieses Ereignis tritt auf, wenn die Tabelle aktiviert wird, übrigens unabhängig davon, ob sie durch eine Anwenderaktion oder per Code aktiviert wurde.

Wir nutzen das Ereignis, den Menüeintrag des Kontextmenüs der Variablen `mnuPopUpKoSt` zuzuweisen. Danach werden die derzeit definierten Kostenstellen aus der Tabelle **Kostenstellen** in die vorher gelöschte Spalte **G** der Tabelle **Daten** eingetragen. Diese Spalte ist übrigens ausgeblendet, da sie den Anwender eigentlich nichts angeht.

Hier nun die Ereignisprozedur `Worksheet_Activate` in voller Schönheit.

```
Private Sub Worksheet_Activate()
    Dim rngValSource As Range _
        'Range-Objekt für Validation-Objekt in Spalte G
    Dim iLastValRow As Long _
        'Zeile mit letztem Validation-Eintrag
    On Error GoTo errHandler
    'Range-Objekte zuweisen
    Set rngDaten = shtDaten.Range("A1")
    Set rngKoSt = shtKoSt.Range("A1")
    Set rngValSource = shtDaten.Range("G1")
    'CommandBar-Control-Objekt zuweisen
    'Set mnuPopUpKoSt = _
        CommandBars("mnuPopUpKoSt").Controls(1)
    'Validation-Range löschen und Daten aus
    'Tabelle >Kostenstellen< neu übertragen
    rngValSource.CurrentRegion.ClearContents
    rngKoSt.CurrentRegion.Offset(1, 0). _
        Resize(rngKoSt.CurrentRegion.Rows.Count - 1, 1).Copy
    rngValSource.PasteSpecial xlPasteValues
    Application.CutCopyMode = False
    'letzte Zeile ermitteln und Validation-Objekt erzeugen
    iLastValRow = _
        shtDaten.Range("G1").CurrentRegion.Rows.Count
    shtDaten.Columns(3).Validation.Delete
    shtDaten.Columns(3).Validation.Add _
        Type:=xlValidateList, _
        Formula1:="=$G$1:$G$" & iLastValRow
    'aktive Zelle in neuer Zeile positionieren
    rngDaten.Cells(rngDaten.CurrentRegion.Rows.Count + 1, _
        1).Activate
Exit Sub

errHandler:
    Err.Source = "shtDaten:Worksheet_Activate"
    cError.ErrorMessage
End Sub
```

Der Code enthält keine Besonderheiten, sieht man von dem `Validation`-Objekt einmal ab. Ein bestehendes `Validation`-Objekt umzuparametrieren ist nicht ganz einfach (warum eigentlich?), weswegen das bestehende Objekt mit den Kostenstellen gelöscht und einfach ein neues erzeugt wird.

BeforeDoubleClick-Ereignis

BeforeDoubleClick tritt – wie der Name schon sagt – auf, bevor der Doppelklick an Excel weitergemeldet wird. Before-Ereignisse verfügen in der Regel über einen **Cancel**-Parameter. Wenn Sie wünschen, dass das Ereignis in Excel nicht ausgeführt wird, so können Sie diesen **Cancel**-Parameter einfach auf **True** setzen und das betreffende Ereignis wird nicht weitergemeldet.

In unserem kleinen Beispiel wird genau dies getan:

```
Private Sub Worksheet_BeforeDoubleClick( _
    ByVal Target As Excel.Range, Cancel As Boolean)
    Cancel = True
End Sub
```

BeforeRightClick-Ereignis

BeforeRightClick verhält sich ähnlich wie **BeforeDoubleClick**, tritt aber zusammen mit der rechten Maustaste auf. In unserem Beispiel wird es dazu genutzt, das in **Workbook_Open** erzeugte Kontextmenü dann anzuzeigen, wenn sich die aktive Zelle in der Kostenstellenspalte C befindet.

In der **BeforeRightClick**-Prozedur wird nun erst geprüft, ob sich die im Argument **Target** enthaltene aktive Zelle in der Spalte C befindet und der Inhalt dieser Zelle auch numerisch ist. Sind diese Bedingungen erfüllt, so wird die **Caption** des einzigen **Control**-Objekts unserer **CommandBar** **mnuPopUpKoSt** mit der in **Target.Value** enthaltenen Kostenstelle gebildet und angezeigt.

```
Private Sub Worksheet_BeforeRightClick( _
    ByVal Target As Excel.Range, Cancel As Boolean)
    On Error GoTo errHandler
    If Target.Column = 3 And IsNumeric(Target.Value) Then
        Application.CommandBars("mnuPopUpKoSt"). _
            Controls(1).Caption = _
                "Wert der KoSt " & Target.Value & " anzeigen"
        Application.CommandBars("mnuPopUpKoSt").ShowPopup
    End If
    Cancel = True
    Exit Sub

errHandler:
    Err.Source = "shtDaten:Worksheet_BeforeRightClick"
    cError.ErrorMessage
End Sub
```


Wird dieser Eintrag des Kontextmenüs angeklickt, so wird die Ereignisprozedur `mnuPopUpKoSt_Click` ausgeführt:

```
Private Sub mnuPopUpKoSt_Click()
    Dim strValue As String          'formatierter Umsatz
    On Error GoTo errHandler
    'Umsatz über SUMMEWENN()-Funktion ermitteln ...
    strValue = Format(Application. _
        SumIf(shtKoSt.Columns(1), _
        ActiveWindow.ActiveCell.Value, _
        shtKoSt.Columns(3)), "Currency")
    '... und ausgeben
    MsgBox "Der aktuelle Betrag dieser Kostenstelle " & _
        "beträgt " & strValue, _
        vbInformation, "Kostenstellenumsatz"
Exit Sub
errHandler:
    Err.Source = "shtDaten:ShowKoStValue"
    cError.ErrorMessage
End Sub
```

Worksheet_Change-Ereignis

In unserem Beispiel wird hier auf Eingaben in den verschiedenen Spalten der Tabelle Daten reagiert, denn dieses Ereignis tritt auf, nachdem die Bearbeitung einer Zelle abgeschlossen wurde.

Wurde ein Wert in Spalte A verändert, so wird geprüft, ob es sich bei dem neuen Wert um ein gültiges Datum handelt. Wenn ja, wird der Zeitraum errechnet und in Spalte B eingetragen. Danach wird die Zelle in Spalte C aktiviert, weil dort schließlich die nächste Eingabe zu erfolgen hat. Wurde kein gültiges Datum eingegeben, so wird der Zelleninhalt gelöscht, die Zelle wieder aktiviert und ein interner Fehler erzeugt.

Erfolgte die das Ereignis auslösende Veränderung in Spalte C, wird die Zelle in Spalte D aktiviert (Achtung! Relative Adressierung).

War es hingegen in Spalte D, so wird die Zelle in Spalte A der nächsten Zeile zur aktiven Zelle. Auch hier ist die Adressierung relativ.

```
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
    On Error GoTo errHandler
    If Target.Row = 1 Or bForgetIt Then Exit Sub
    Select Case Target.Column

        Case 1 'Datumsspalte
```

```

'Wenn Zelle kein gültiges Datum enthält, wird die
'Zelle geleert und eine Meldung angezeigt.
'bForgetIt unterdrückt die erneute
'Ereignisbehandlung beim Leeren.
If Not IsDate(Target.Value) Then
    bForgetIt = True
    Target.Value = ""
    bForgetIt = False
    Target.Activate
    Err.Raise 9999
End If
'Berechnen des Zeitraums in Spalte B und Zelle in
'Spalte C aktivieren
Target.Cells(1, 2).Value = _
    Year(Target.Cells(1, 1).Value) * 100 + _
    Month(Target.Cells(1, 1).Value)
Target.Cells(1, 3).Activate
Case 3 'Kostenstelle
    'Zelle in Spalte D aktivieren
    Target.Cells(1, 2).Activate
Case 4 'Buchungsbetrag
    'Zelle in Spalte A der nächsten Zeile aktivieren
    Target.Cells(2, -2).Activate
End Select
Exit Sub

```

```

errHandler:
Err.Source = "shtDaten:Worksheet_Change"
If Err.Number = 9999 Then
    Err.Description = "Die Zelle " & Target.Address &
        " muss ein gültiges Datum enthalten."
End If
cError.ErrorMessage
End Sub

```

Als Besonderheit sei hier erwähnt, dass im Falle eines ungültiges Datums in Spalte A mit `Err.Raise` ein Fehler erzeugt wird. In diesem Falle wird sofort der `errHandler` angesprungen und eine passende Fehlerbeschreibung erzeugt.

Worksheet_Deactivate-Ereignis

`Worksheet_Deactivate` bedeutet für uns, dass der Anwender seine Eingaben abgeschlossen und die Tabelle Daten verlassen hat. Dies ist die Ge-

legenheit, um die Eingaben in der Tabelle Kostenstellen zu den jeweiligen Kostenstellen zu verdichten. Das hätte man natürlich mit Formeln machen können, werden Sie vielleicht einwenden. Im Prinzip ja, wenn man den Umfang der Kostenstellen kennt. In unserem Beispiel ist es einerlei, wie viele Kostenstellen angelegt werden, denn wir verdichten alle, die das sind. Und wenn es Tausende sind.

Hierzu werden zuerst die alten Kostenstellenumsätze gelöscht. Anschließend arbeitet eine Schleife alle Datenzeilen ab und ermittelt mittels der Find-Methode die entsprechende Zeile in der Tabelle Kostenstellen, wo dann der jeweilige Umsatz dem dortigen Wert hinzuaddiert wird. Wurde eine Kostenstelle nicht gefunden, so wird die betreffende Kostenstelle in der Tabelle Daten rot formatiert und der Zähler lngNotFound inkrementiert. Ist ein solcher Fall aufgetreten, wird eine entsprechende Meldung aufgebaut und per MsgBox angezeigt.

```
Private Sub Worksheet_Deactivate()
    Dim rngFind As Range      'Range-Objekt für Find-Methode
    Dim iRowDaten As Long     'Zeilenzeiger in rngDaten
    Dim iRowKoSt As Long      'Zeilenzeiger in rngKoSt
    Dim lngKoSt As Long       'aktuelle Kostenstelle
    Dim lngNotFound As Long   'Anz.der nicht gefundenen KoSt
    Dim strMessage As String  'Ausgabe bei fehlerhaften KoSt
    On Error GoTo errHandler
    'Löschen der Kostenstellenumsätze in shtKoSt
    rngKoSt.CurrentRegion.Offset(1, 2).Resize _
    (rngKoSt.CurrentRegion.Rows.Count - 1, 1).ClearContents
    'Alle Datenzeilen in shtDaten durchlaufen
    For iRowDaten = 2 To rngDaten.CurrentRegion.Rows.Count
        lngKoSt = rngDaten.Cells(iRowDaten, 3).Value
        'Kostenstelle in Spalte A der shtKoSt suchen
        Set rngFind = shtKoSt.Columns(1).Find( _
            what:=lngKoSt, LookIn:=xlValues, lookat:=xlWhole)
        If rngFind Is Nothing Then
            lngNotFound = lngNotFound + 1
            rngDaten.Cells(iRowDaten, 3).Font.ColorIndex = 3

        Else
            rngKoSt.Cells(rngFind.Row, 3).Value = _
                rngKoSt.Cells(rngFind.Row, 3).Value + _
                rngDaten.Cells(iRowDaten, 4).Value
        End If
    Next
    If lngNotFound = 1 Then
```

```

        strMessage = "Eine Kostenstellen ist nicht " & _
                    "definiert." & vbCrLf & vbCrLf & _
                    "Sie ist rot gekennzeichnet."
    Else
        strMessage = lngNotFound & " Kostenstellen sind " & _
                    "nicht definiert." & vbCrLf & vbCrLf & _
                    "Die betreffenden Kostenstellen " & _
                    "sind rot gekennzeichnet."
    End If
    If strMessage <> "" Then
        MsgBox strMessage, vbExclamation, "Kostenstellen"
        shtDaten.Activate
    End If
    Application.StatusBar = False
    Exit Sub

errHandler:
    Err.Source = "shtDaten:Worksheet_Deactivate"
    cError.ErrorMessage
End Sub

```

Worksheet_SelectionChange-Ereignis

SelectionChange tritt auf, wenn die Zellauswahl geändert wurde: per Maus, per Cursor-, Tabulator- oder Return-Taste oder mittels VBA.

In unserem Beispiel werden in der StatusBar Ausfüllhinweise zu den einzelnen Spalten ausgegeben, also nichts Aufregendes. Das ist auch der Grund für die Zeile

```
Application.StatusBar = False
```

im Worksheet_Deactivate-Ereignis, die diese StatusBar wieder an Excel zurückgibt.

```

Private Sub Worksheet_SelectionChange( _
    ByVal Target As Excel.Range)
    On Error GoTo errHandler
    Select Case Target.Column
    Case 1
        Application.StatusBar = "Bitte geben Sie in " & _
            "dieser Spalte das Datum der " & _
            "Kostenstellenbelastung an."
    Case 2
        Application.StatusBar = "Dieser Zeitraum wird " & _
            "automatisch berechnet."
    End Select

```

```

Case 3
    Application.StatusBar = "Bitte wählen Sie eine " & _
        "Kostenstelle aus."
Case 4
    Application.StatusBar = "Bitte geben Sie in " & _
        "dieser Spalte den Betrag an, mit dem Sie " & _
        "die Kostenstellen belasten wollen."
Case Else
    Application.StatusBar = False
End Select
Exit Sub

```

```

errHandler:
    Err.Source = "shtDaten:Worksheet_Deactivate"
    cError.ErrorMessage
End Sub

```

Worksheet_Calculate-Ereignis

Hierzu ist mir in dem ansonsten geschlossenen Beispiel wirklich nichts eingefallen. Es wäre jedoch denkbar, eine umfangreiche Kalkulation einer einzelnen Kostenstelle in einem gesonderten Blatt durchzuführen und das Ergebnis anschließend irgendwo weiterzuverwenden. Doch auch hierzu würde man sich vermutlich des VBAs bedienen, um die zu berechnenden Werte in diese Tabelle zu übertragen. Ist dies geschehen, würde ich die Tabelle mit der Calculate-Methode durchrechnen und in der folgenden Programmzeile irgendetwas mit dem Ergebnis anstellen. Auch hier wäre ich nicht auf das Calculate-Ereignis angewiesen.

Wie dem auch sei. Wenn Sie an dieser Stelle angelangt sind, werden Sie mit dem Calculate-Ereignis vermutlich alleine fertig werden.

Worksheet_FollowHyperlink-Ereignis

Dieses in Excel 2000 neue Ereignis wird aufgerufen, wenn Ihre Tabelle einen Hyperlink aufweist und dieser angeklickt wurde. Als Argument wird Ihnen ein Hyperlink-Objekt übergeben, welches Ihnen ermöglicht den Hyperlink umzuleiten, Dokumente zu erzeugen oder beliebige, andere Aktionen auszuführen.

Wir haben nun an einem lebenden Beispiel 6 der 8 Ereignisse des Worksheet-Objekts näher untersucht. Sie sind alle auch als Ereignisse des Workbook-Objekts verfügbar, wobei ihnen dort der Begriff Sheet vorangestellt wurde, also beispielsweise Workbook_SheetActivate. Auch wird das jeweilige Worksheet-Objekt als Argument übergeben. Mit diesen Ereignissen können Sie gewissermaßen zentral auf die Ereignisse der ein-

zelen Worksheets reagieren, weil Sie zur Entwurfszeit zum Beispiel noch gar nicht wissen, wie viele und welche Tabellen in der Arbeitsmappe einmal enthalten sein werden. Als Beispiel mag eine Datei dienen, in der die Abschreibung von Anlagegütern kalkuliert wird, wobei für jedes Anlagenobjekt (PC, Auto, Werkzeugmaschine) eine eigene Tabelle verwendet wird.

6.4 Die Workbooks-Auflistung

Die Workbooks-Auflistung enthält alle derzeit geöffneten Arbeitsmappen.

6.4.1 Eigenschaften und Methoden im Überblick

	Element	Beschreibung	Rückgabe
M	Add	Erzeugt eine neue Arbeitsmappe	Workbook
M	Close	Schließt alle geöffneten Arbeitsmappen	
E	Count	Gibt die Anzahl der geöffneten Arbeitsmappen zurück	Integer
E	Item	Gibt eine Arbeitsmappe aus der Liste der geöffneten Arbeitsmappe zurück	Workbook
M	Open	Öffnet eine Arbeitsmappe	Workbook
M	OpenText	Öffnet eine Textdatei als Arbeitsmappe mit einer Tabelle	

M = Methode, E = Eigenschaft, A = Auflistung

6.4.2 Eigenschaften und Methoden unter der Lupe

Add-Methode

Die Add-Methode erzeugt eine neue Arbeitsmappe und gibt, wenn der Ausdruck auf der rechten Seite des Gleichheitszeichens steht, einen Objektverweis auf diese Arbeitsmappe zurück. Sie kann mit dem optionalen Argument *Template* versehen werden. Enthält *Template* eine der folgenden Konstanten, so enthält die neue Arbeitsmappe genau ein Blatt des angegebenen Typs:

- ▼ `xlWBATChart`,
- ▼ `xlWBATExcel4IntlMacroSheet`,
- ▼ `xlWBATExcel4MacroSheet` oder
- ▼ `xlBATWorksheet`

Sie können jedoch auch den Namen einer Datei übergeben, die übrigens keine Vorlage (*Template*) sein muss, eine ganz gewöhnliche Excel-Arbeitsmappe tut's auch. Die Online-Hilfe meint übrigens, man *müsse* keinen

Pfad angeben, man *könne* allenfalls. Versuchen Sie es einmal spaßeshalber ohne ...

Wird das Argument `Template` weggelassen, so enthält die neue Arbeitsmappe die Anzahl an leeren Tabellenblättern, die im Register Allgemein der Optionen unter »Blätter in neuer Arbeitsmappe:« angegeben sind. Diese Zahl können Sie über die Eigenschaft `Application.SheetsInNewWorkbook` auch per Code ermitteln.

Im ersten Beispiel wird eine Datei geöffnet, ohne sie einer Objektvariablen zuzuweisen. Ein Anwendungsfall könnte sein, dass Sie eine Kopie einer Datei erzeugen wollen, die danach manuell bearbeitet werden soll. In diesem Falle ist eine Zuweisung an eine Objektvariable überflüssig:

```
Workbooks.Add "C:\Projekte\Kalkulation.xls"
```

Wollen Sie hingegen damit weiter arbeiten, so ist die folgende Variante vorteilhafter:

```
Dim wbkNew as Workbook  
Set wbkNew = Workbooks.Add("C:\Projekte\Kalkulation.xls")
```

Close-Methode

Wird die `Close`-Methode auf die `Workbooks`-Auflistung angewandt, werden alle geöffneten Arbeitsmappen geschlossen. Excel wird bei jeder veränderten Arbeitsmappe den Anwender fragen, ob die Änderungen gespeichert werden sollen:

```
Workbooks.Close
```

Das gezielte Schließen jeder einzelnen Arbeitsmappe ist allerdings der bessere Weg.

Count-Eigenschaft

Diese Eigenschaft gibt die Anzahl der geöffneten Arbeitsmappen zurück:

```
MsgBox Workbooks.Count
```

Item-Eigenschaft

Die `Item`-Eigenschaft ist im Prinzip die Standardeigenschaft jeder Auflistung und gibt auch bei der `Workbooks`-Auflistung eine einzelne Arbeitsmappe zurück. Als Standardeigenschaft darf sie weggelassen werden. Die folgenden Ausdrücke geben aus der Liste der geöffneten Arbeitsmappen die zurück, deren Name `TestMappe.xls` lautet:

```
Workbooks.Item("TestMappe.xls")  
Workbooks("TestMappe.xls")
```

Open-Methode

Die `Open`-Methode ermöglicht das Öffnen einer Arbeitsmappe. Anders als die `Add`-Methode weist sie allerdings eine ganze Reihe von Argumenten auf:

Argumentname	Beschreibung
<code>FileName</code>	Pfad und Name der Arbeitsmappe
<code>UpdateLinks</code>	(optional) Existiert ein Fernbezug oder externer Bezug und wird das Argument nicht angegeben, so wird der Anwender über einen Dialog gefragt, ob und wie die Aktualisierung erfolgen soll. 0 – keine Aktualisierung von Bezügen 1 – externe Bezüge werden aktualisiert, jedoch keine Fernbezüge 2 – Fernbezüge werden aktualisiert, jedoch keine externen Bezüge 3 – beide Arten werden aktualisiert
<code>ReadOnly</code>	(optional) Wenn <code>True</code> , wird die Datei schreibgeschützt geöffnet.
<code>Format</code>	(optional) Wenn Sie eine Textdatei öffnen, definieren Sie mit diesem Argument das zu erwartende Trennzeichen: 1 – Tabstopps 2 – Kommas 3 – Leerzeichen 4 – Semikolons 5 – keine 6 – in <code>Delimiters</code> -Argument anzugebendes Trennzeichen
<code>Password</code>	(optional) Das Lese- und Schreibschutzkennwort
<code>WriteResPassword</code>	(optional) Das Schreibschutzkennwort
<code>IgnoreReadOnlyRecommended</code>	(optional) Sind dieses Argument und <code>ReadOnly True</code> , so wird die Schreibschutzempfehlung nicht ausgegeben.
<code>Origin</code>	(optional) Wenn Sie eine Textdatei öffnen, so können Sie über die Konstanten <code>xlMacintosh</code> , <code>xlWindows</code> oder <code>xlMS-DOS</code> festlegen, welche Codeseiten der Datei zugrunde gelegt sind.

Argumentname	Beschreibung
Delimiter	(optional) Öffnen Sie eine Textdatei mit dem Format-Argument = 6, so können Sie hier das verwendete Trennzeichen angeben.
Editable	(optional) Betrifft Excel 4 und 5 AddIns und soll uns hier nicht weiter belasten.
Notify	(optional) Kann die zu öffnende Datei nicht mit Schreibzugriff geöffnet werden, so wird die Datei schreibgeschützt geöffnet und weiterhin versucht, sie mit Schreibzugriff zu öffnen, wenn dieses Argument True ist. In diesem Fall erfolgt eine Benachrichtigung, wenn Öffnen mit Schreibzugriff möglich ist. Ist dieses Argument False oder nicht angegeben, so wird die Datei nicht geöffnet, sofern sie nicht mit Schreibzugriff geöffnet werden kann.
Converter	(optional) Der Index des Dateikonverters, der zuerst verwendet werden soll. Misslingt der Versuch, werden alle anderen Konvertierungsserver in der Reihenfolge ihrer Installation gestartet.
AddToMRU	(optional) Ist dieses Argument True, wird die Datei der Liste der Recent Files List hinzugefügt.

Diese Mammutliste macht das Öffnen einer Arbeitsmappe zu einem größeren Problem, als es eigentlich ist. Außer (natürlich) `FileName` und gelegentlich `Password` oder `WriteResPassword` bestand zumindest für mich bislang keine Veranlassung weitere Argumente einzusetzen. Und genau um diese Fälle werden wir uns jetzt kümmern.

Nur in Ausnahmefällen ist es damit getan, eine Arbeitsmappe nur zu öffnen. Dies mag der Fall sein, wenn wir per Programm eine Arbeitsmappe erstellt oder bearbeitet haben und sie danach dem Anwender zur manuellen Bearbeitung zur Verfügung stellen. Zumeist ist es unser Begehren, per Code in einer zu öffnenden Datei Veränderungen vorzunehmen. Und hierzu benötigen wir nicht nur die geöffnete Arbeitsmappe, sondern einen Objektverweis auf diese.

Das folgende Codefragment der Funktion `GetWBK` wird Ihnen in den folgenden Kapiteln noch begegnen. Im fertigen Zustand kommt sie als Methode einer Klasse namens `c1sWBK` zum Einsatz.

```

Public Function getWBK(ByVal strPath As String, _
    ByVal strFile As String, _
    Optional ByVal strWritePassword As Variant, _
    Optional ByVal strReadWritePassword As Variant) _
    As Workbook
    'Argumente:
    '   strPath:           Pfad der Arbeitsmappe
    '   strFile:           Name der Arbeitsmappe
    '   strWritePassword:   Kennwort für Schreibschutz
    '   strReadWritePassword: Kennwort für Lese- und
                            'Schreibschutz
    ...
    If Not IsMissing(strWritePassword) And Not _
        IsMissing(strReadWritePassword) Then
        'Fehler erzeugen
    End If
    If IsMissing(strWritePassword) And _
        IsMissing(strReadWritePassword) Then
        'normal
        Set getWBK = Workbooks.Open( _
            Filename:=strPath & strFile)
    ElseIf Not IsMissing(strWritePassword) And _
        IsMissing(strReadWritePassword) Then
        'mit Schreibschutz
        Set getWBK = Workbooks.Open( _
            Filename:=strPath & strFile, _
            WriteResPassword:=strWritePassword)
    ElseIf IsMissing(strWritePassword) And Not _
        IsMissing(strReadWritePassword) Then
        'mit Lese- und Schreibschutz
        Set getWBK = Workbooks.Open( _
            Filename:=strPath & strFile, _
            Password:=strReadWritePassword)
    End If
End Function

```

Die folgende Tabelle gibt eine Übersicht über die Passwortoptionen, wobei WPW für das Argument WriteResPassword (Schreibschutz) und RWPW für Password (Lese- und Schreibschutz) steht:

Aktion	Schreibschutz	Lese- und Schreibschutz
IsMissing(WPW) And IsMissing(RWPW)	ohne	ohne
Not IsMissing(WPW) And IsMissing(RWPW)	mit	ohne
IsMissing(WPW) And Not IsMissing(RWPW)	ohne	mit
macht keinen Sinn, weshalb ein Fehler erzeugt wird	mit	mit

Sollten Sie Schwierigkeiten mit den Begrifflichkeiten dieser beiden Schutzarten haben, so sind Sie in bester Gesellschaft, denn bei Microsoft scheinen sie auch vorzuherrschen, wie ein Blick auf den Dialog der Passwortbestätigung zeigt. Dank der Funktion `GetWBK` sind Sie diese Sorgen jedoch zukünftig los.

OpenText-Methode

Die `OpenText`-Methode erzeugt eine neue Arbeitsmappe mit einem Tabellenblatt und lädt die angegebene Textdatei in diese Tabelle. Ebenso wie die `Open`-Methode ist auch sie ein Ungetüm, wie die folgende Argumentliste nachdrücklich zeigt:

Argumentname	Beschreibung
Filename	Pfad und Name der zu öffnenden Textdatei
Origin	(optional) Beschreibt die Herkunft der Textdatei in Form einer der folgenden Konstanten: xlMacintosh – Mac-Umgebung xlWindows – Windows-Umgebung xlMSDOS – MSDOS-Umgebung Wird dieses Argument weggelassen, so wird die letzte Einstellung (!) übernommen. Sind Sie verlässlich in einem homogenen Umfeld, so können Sie dieses Argument weglassen. Ist dies nicht sichergestellt und Sie lassen das Argument weg, fangen Sie sich potenziell Probleme mit Code-Pages und Datumswerten ein.
StartRow	(optional) Die Zeile, ab der eingelesen werden soll. Wird dieses Argument weggelassen, so wird ab der ersten Zeile eingelesen. Hiermit können Sie das Einlesen eventueller Überschriftzeilen verhindern.

Argumentname	Beschreibung
DataType	(optional) Hiermit geben Sie an, ob die einzelnen Werte einer Zeile über Trennzeichen voneinander getrennt sind oder ob sie eine feste Breite aufweisen: <code>x1Delimited</code> weist auf Trennzeichen hin. <code>x1FixedWidth</code> kennzeichnet Felder (und somit auch Zeilen) konstanter Länge.
TextQualifier	(optional) Hiermit können wir Texterkennungszeichen beim Import unterdrücken. Die Konstanten hierfür lauten: 1 – <code>x1TextQualifierDoubleQuote</code> 2 – <code>x1TextQualifierSingleQuote</code> -4142 – <code>x1TextQualifierNone</code> Interessanterweise lassen sich die beiden ersten Konstanten zu 3 aufaddieren, ohne dass allerdings beide Qualifier erkannt werden. Im Gegenteil, denn es wird keiner von beiden berücksichtigt. Wird das Argument nicht angegeben, so werden die doppelten Anführungszeichen unterdrückt (<code>x1TextQualifierDoubleQuote</code>).
ConsecutiveDelimiter	(optional) Mit <code>True</code> bei diesem Argument steuern Sie, ob aufeinander folgende Trennzeichen als ein Trennzeichen interpretiert werden sollen; <code>False</code> ist Voreinstellung. Ein <code>True</code> ist hier nicht ganz ungefährlich, denn leere Felder werden von Programmen, die diese Art Dateien erstellen, oft durch zwei aufeinander folgende Trennzeichen exportiert. Durch <code>True</code> werden diese leeren Felder übersprungen, wodurch die nachfolgenden Felder quasi aufrücken.
Tab	(optional) Ist <code>Tab</code> gleich <code>True</code> , wird ein Tabulatorzeichen als Trennzeichen erkannt. Die Voreinstellung ist <code>False</code> .
Semicolon	(optional) Ist <code>Semicolon</code> gleich <code>True</code> , wird ein Semikolon als Trennzeichen erkannt. Die Voreinstellung ist <code>False</code> .
Comma	(optional) Ist <code>Comma</code> gleich <code>True</code> , wird ein Komma als Trennzeichen erkannt. Die Voreinstellung ist <code>False</code> .
Space	(optional) Ist <code>Space</code> gleich <code>True</code> , wird ein Tabulatorzeichen als Trennzeichen erkannt. Die Voreinstellung ist <code>False</code> .
Other	(optional) Ist <code>Other</code> gleich <code>True</code> , so wird das unter <code>OtherChar</code> angegebene Zeichen als Trennzeichen erkannt. Die Voreinstellung ist <code>False</code> .

Argumentname	Beschreibung
OtherChar	(optional) Dieses Zeichen wird als Trennzeichen interpretiert, sofern Other gleich True ist.
FieldInfo	(optional) Mit diesem Argument, das ein Array erwartet, können Sie Angaben zur Formatierung einzelner Spalten machen. Da es wenig Sinn macht, diese vielfältigen Möglichkeiten hier noch einmal aufzuführen, sei auf die Hilfe verwiesen.
DecimalSeparator	(optional) Dieses Argument bestimmt das zu verwendende Dezimaltrennzeichen. Wird es nicht angegeben, so wird das Trennzeichen der Ländereinstellungen verwendet.
ThousandsSeparator	(optional) Dieses Argument bestimmt das zu verwendende Tausendertrennzeichen. Wird es nicht angegeben, so wird das Trennzeichen der Ländereinstellungen verwendet.

Die Felder Tab, Semicolon, Comma, Space und OtherChar können miteinander kombiniert werden, wodurch alle diese Zeichen als Trennzeichen interpretiert werden.

Auf so etwas kommt man normalerweise nicht, aber das Experimentieren mit diesen Argumenten brachte zutage, dass sich Excel die in der *aktuellen Instanz* zuletzt verwendeten Einstellungen merkt und klammheimlich weiter verwendet, wenn sie denn plötzlich weggelassen werden. Somit müssen Sie in der voranstehenden Tabelle überall dort, wo von der Voreinstellung die Rede ist, in Gedanken hinzufügen, sofern niemand in der Zwischenzeit etwas anderes vereinbart hat.



Dass Excel nachtragend ist und optionale Einstellungen der letzten Parametrierung weiter verwendet, kennen wir bereits, beispielsweise von der Find-Methode des Range-Objekts. Dort bekommen wir das jedoch durch die Angabe zweier weiterer Argumente gut in den Griff.

Zusammen mit der doch recht umfangreichen Parametrierung dieser Methode und dem Rest Unsicherheit, was eventuelle Left-overs angeht, kann ich der Verwendung dieser Methode nicht vollends überzeugt das Wort reden. Sie werden in Kapitel 7 die Sprachelemente kennen lernen, mit denen Sie eine Textdatei nach bewährtem Muster öffnen und zeilen- respektive feldweise verarbeiten können. Entscheiden Sie über Ihr künftiges Vorgehen, wenn Sie mit diesen Techniken vertraut sind.

6.5 Das Workbook-Objekt

Das `Workbook`-Objekt ermöglicht den Zugriff auf eine geöffnete Arbeitsmappe und repräsentiert als Objekt die Arbeitsmappe inklusive aller ihrer Elemente, also Tabellen- oder Diagrammblätter, aber auch die Codekomponenten.

6.5.1 Eigenschaften und Methoden im Überblick

Auch hier werden wir aus der Fülle der uns zu Gebote stehenden Eigenschaften und Methoden einige näher beleuchten.

	Element	Beschreibung	Rückgabe
M	Close	Schließt die Arbeitsmappe	
E	FullName	Beinhaltet den Namen der Arbeitsmappe inklusive des vollständigen Pfads	String
E	Name	Der Name der Arbeitsmappe	String
A	Names	Beinhaltet die Names-Auflistung der Arbeitsmappe	Names
M	Save	Speichert die Arbeitsmappe	
M	SaveAs	Speichert die Arbeitsmappe unter einem anderen Namen	
M	SaveCopyAs	Speichert die Arbeitsmappe zusätzlich unter einem anderen Namen	
A	Sheets	Beinhaltet alle Blätter (Tabellen- und Diagrammblätter) der Arbeitsmappe	Sheets
E	VBProject	Gibt das VBProject der Arbeitsmappe zurück, in dem alle VBA-Komponenten enthalten sind	VBProject
A	Windows	Gibt alle Fenster zurück, die der Arbeitsmappe zugeordnet sind	
A	Worksheets	Gibt die Auflistung aller in der Arbeitsmappe enthaltenen Tabellenblätter zurück	Worksheet

M = Methode, E = Eigenschaft, A = Auflistung

6.5.2 Eigenschaften und Methoden unter der Lupe

Close-Methode

Wir sprachen bei der `Close`-Methode der `Workbooks`-Auflistung bereits davon, dass das gezielte Schließen der beteiligten Arbeitsmappen dem Rundumschlag `Workbooks.Close` vorzuziehen ist. Die optionalen Argumente `SaveChanges`, `FileName` und `RouteWorkbook` bieten weitere Steuerungsmöglichkeiten.

Mit dem logischen Argument `SaveChanges` legen Sie fest, ob eventuelle Änderungen in der Arbeitsmappe gespeichert werden sollen. Wurde die Arbeitsmappe seit dem Öffnen oder dem letzten Speichern verändert und dieses Argument beim Schließen nicht angegeben, so wird der Anwender gefragt, wie nun zu verfahren sei.

```
ThisWorkbook.Close SaveChanges:=True
```

Das zweite Argument `FileName` legt den Namen fest, unter dem die Datei gespeichert werden soll. Gibt es im Verzeichnis der Datei `Mappe1.xls` nach diesen Codezeilen eine Datei namens `Test.xls`?

```
Dim wbk As Workbook
Set wbk = Workbooks("Mappe1.xls")
wbk.Save
wbk.Close SaveChanges:=True, _
    FileName:=wbk.Path & "\Test.xls"
```

Nein, denn die Datei wurde seit dem letzten Speichern nicht mehr verändert, und gespeichert wird in der Zeile vor dem Schließen. Wenn Sie also eine Datei unter einem anderen Namen oder demselben Namen an einem anderen Ort gespeichert haben wollen, so ist dies nicht der ideale Weg. Hier ist den weiter unten vorgestellten Methoden `SaveAs` oder `SaveCopyAs` der Vorzug zu geben.

Mit dem Argument `RouteWorkbook` legen Sie fest, ob die Arbeitsmappe nach dem Schließen weitergeleitet werden soll.

FullName- und Name-Eigenschaften

Die `Name`-Eigenschaft gibt den Dateinamen der Arbeitsmappe zurück. Dies ist auch der `Name`, mit dem die `Item`-Methode gefüttert werden kann. Die folgenden, zugegebenermaßen reichlich unsinnigen Zeilen geben dieses Verhältnis wieder:

```
Dim wbk As Workbook
Set wbk = Workbooks(ThisWorkbook.Name)
```

`FullName` wiederum enthält den vollständigen Namen inklusive Laufwerk:

```
MsgBox ThisWorkbook.Name & vbCrLf & ThisWorkbook.FullName
```

Names-Auflistung und Name-Objekt

Die Einsatzmöglichkeiten eines `Name`-Objekts bei `Worksheet` oder `Range` halten sich in Grenzen. Anders hingegen bringt die `Names`-Auflistung des `Workbook`-Objekts schon Vorteile, denn dabei handelt es sich um die klassischen Namen, die wir vom Umgang mit Excel-Zellen bereits kennen.

Die Names-Auflistung und das Name-Objekt bieten hinreichend Stoff für ein ganzes Kapitel. Allerdings lichtet sich der Nebel erheblich, wenn man sich darauf beschränkt, mit existierenden Namen zu operieren.

Die Item-Methode ermöglicht den Zugriff auf ein Name-Objekt innerhalb der Auflistung. Als Standard-Methode dürfen wir die Methode nach gewohntem Muster wieder weglassen:

```
ThisWorkbook.Names.Item("Vorname") 'oder kurz
ThisWorkbook.Names("Vorname")
```

Beide Ausdrücke geben ein Name-Objekt zurück, was nicht gleichbedeutend ist mit dem Ziel unserer Begierde, dem eigentlichen Range. Natürlich können wir die Names-Auflistung auch mit einer For-Each-Schleife durchlaufen oder auch enumerieren, wie man so schön sagt:

```
Private Sub ShowNames()
    Dim actName As Name          'Name-Objekt für Schleife
    Dim strMessage As String     'Ausgabetext
    strMessage = "Die Arbeitsmappe enthält folgende " & _
        "Namen:" & vbCrLf & vbCrLf
    For Each actName In ThisWorkbook.Names
        strMessage = strMessage & actName.Name & vbCrLf
    Next
    MsgBox strMessage, vbInformation
End Sub
```

Um zu prüfen, ob ein Name existiert, könnten Sie die Names-Auflistung auf diese Weise durchlaufen und die Schleife bei Übereinstimmung verlassen. Das vorzeitige Verlassen belässt actName auf einem gültigen Name-Objekt, was am Ende geprüft und als Ergebnis der Funktion selbst als Rückgabe mit auf den Weg gegeben wird:

```
Private Function CheckName(ByVal strName as String) _
    As Boolean
    Dim actName As Name          'Name-Objekt für Schleife
    For Each actName In ThisWorkbook.Names
        If actName.Name = strName Then Exit For
    Next
    CheckName = Not actName Is Nothing
End Function
```


Etwas in Sachen Geschwindigkeit optimiert sähe die Funktion so aus:

```
Private Function CheckName(...) As Boolean
    Dim actName As Name          'Name-Objekt für Schleife
    CheckName = True
    For Each actName In ThisWorkbook.Names
        If actName.Name = strName Then Exit Function
    Next
    CheckName = False
End Function
```

Hart gesottene Zeitgenossen führen einen Testzugriff auf das Name-Objekt aus und reagieren auf den Fehler 9, der sich zwangsläufig dann zu Wort meldet, wenn das gesuchte Element nicht existiert. Doch irgendwie überzeugt mich eine Funktion obigen Zuschnitts mehr.

Nachdem wir uns solchermäßen vergewissert haben, dass unser Name-Objekt existiert, können wir getrost darauf zugreifen und zum Beispiel den Inhalt auslesen. Zu diesem Zwecke steht uns die Eigenschaft `RefersToRange` zur Verfügung, die das Range-Objekt zurückgibt, dem dieser Name zugeordnet ist:

```
MsgBox ThisWorkbook.Names("Vorname").RefersToRange.Value
```

Als Besonderheit sei noch erwähnt, dass Sie auch mehrere Zellen mit einem Namen versehen können. Logischerweise führt dann ein Zugriff auf das dahinter stehende Range-Objekt zu einem Fehler, wenn Sie eine Eigenschaft oder eine Methode verwenden, die eine einzelne Zelle erwartet. Aber ein Range-Objekt mit mehreren Zellen verfügt über eine `Cells`-Auflistung, die uns der Zugriff auf eine einzelne Zelle erlaubt:

```
ThisWorkbook.Names("Accounts").RefersToRange.Cells(1).Value
```

Benannte Zellen sind dann eine sinnvolle Angelegenheit, wenn dem Anwender die Möglichkeit eingeräumt wird die Tabelle durch Einfügen oder Löschen einzelner Zellen oder gar Zellverbände strukturell zu verändern. In solchen Fällen beschleicht mich immer ein ungutes Gefühl, denn wer hindert ihn letztlich daran, auch strukturell wichtige Zellen zu löschen? Hier sind wir wieder an einem Punkt angelangt, der sich wie ein roter Faden durch VBA- und speziell Excel-VBA-Anwendungen zieht, denn selbst die besten Programme benötigen letztlich kompetente und verlässliche Anwender. Software ersetzt keinen Verstand, sie unterstützt ihn.

Save-, SaveAs- und SaveCopyAs-Methoden

Allen drei Methoden ist gemeinsam, dass sie das vor der Methode stehende Workbook speichern. `SaveAs` und `SaveCopyAs` speichern die Arbeitsmappe unter einem anderen Namen, was soweit auch noch einleuchtet. Bei `SaveAs` wird jedoch die Arbeitsmappe unter dem neuen Namen geöffnet und die alte geschlossen, während bei `SaveCopyAs` nach wie vor die alte Arbeitsmappe geöffnet bleibt.

Ein Beispiel mit einer Frage: Welche der beiden Dateien im folgenden Code ist aktueller? `Mappe1.xls` oder `Mappe2.xls`?

```
Private Sub WorkbookSave()  
    Dim wbk As Workbook  
    Dim strWBK1 As String  
    Dim strWBK2 As String  
    strWBK1 = ThisWorkbook.Path & "\Mappe1.xls"  
    strWBK2 = ThisWorkbook.Path & "\Mappe2.xls"  
    Set wbk = Workbooks.Open(strWBK1)  
    wbk.SaveAs strWBK2  
    wbk.Save  
    wbk.Close  
End Sub
```

Es ist zwangsläufig `Mappe2.xls`, da `Mappe1.xls` als `Mappe2.xls` gespeichert wird, wobei `Mappe2` geschlossen wird und unsere `Workbook`-Variable auf die neue Datei verweist. Die nachfolgende `Save`-Methode speichert `Mappe2` erneut.

Doch nun dasselbe Beispiel mit der `SaveCopyAs`-Methode:

```
Private Sub WorkbookSave()  
    ...  
    wbk.SaveCopyAs strWBK2  
    wbk.Save  
    wbk.Close  
End Sub
```

Nun wird `Mappe1` aktueller sein als `Mappe2`, da `Mappe1` nach Abarbeiten der `SaveCopyAs`-Methode geöffnet bleibt und die nachfolgende `Save`-Methode `Mappe1.xls` zwangsläufig noch einmal speichert.

Das ist ein kleiner, aber mitunter entscheidender Unterschied.

Wenn Sie sich die Parametrierung dieser beiden Methoden noch ansehen, werden Sie feststellen, dass `SaveAs` über eine Reihe weiterer Argumente verfügt, während sich `SaveCopyAs` mit dem Namen zufrieden gibt. Sa -

veAs bietet die ganze Palette von Dateiformat über Passworte bis hin zum Hinzufügen zur Liste der zuletzt geöffneten Dateien.

Sheets- und Worksheets-Auflistungen

Die `Worksheets`-Auflistung ist eine 1-basierte Collection mit allen Tabellenblättern der betreffenden Arbeitsmappe, mit der wir uns bereits in Kapitel 6.2 auseinander gesetzt haben. Die `Sheets`-Auflistung enthält darüber hinaus auch alle anderen Arten von Blättern, die in einer Arbeitsmappe auftauchen können. Die `Sheets`-Auflistung ist also gewissermaßen ein Art Meta-Auflistung.

VBProject-Eigenschaft

Hiermit erhalten Sie einen Verweis auf die VBA-Elemente der Arbeitsmappe sowie auf die Entwicklungsumgebung selbst. Obwohl sich durch dieses `VBProject`-Objekt schon enorme Möglichkeiten ergeben, ist der praktische Nutzen eher als gering einzustufen.

Ein potenzieller Anwendungsfall ist auf den ersten Blick eine Routine, die uns ermöglicht, in anderen Arbeitsmappen Code auszutauschen, also gewissermaßen ein Programm-Update durchzuführen. In diese Verlegenheit geriet ich auch einmal, aber der Grund dafür war, dass ich den Code nicht sauber von den Daten getrennt hatte. Also existierten Dutzende von Excel-Dateien in Dutzenden von Niederlassungen eines Unternehmens, die alle irgendwelche Daten enthielten. Und dann wollte der Kunde eine Änderung des Programmes, die er noch Wochen zuvor strikt ausgeschlossen hatte. Als alter Hase hätte ich diesen Beteuerungen kein Vertrauen schenken dürfen und den Code in einer separaten Datei ablegen müssen, egal was er sagt. Und wenn Sie sich an diese Regel halten, werden Sie kaum in die Verlegenheit kommen, auf Codekomponenten per Programm zugreifen zu *müssen*. Denn Sie werden schon einige Energie in das `VBProject`-Objekt und seine Strukturen stecken müssen, ehe Sie sichere Erfolge aufweisen können.

Windows-Auflistung und Window-Objekt

Excel ist eine so genannte MDI-Anwendung (Multiple Document Interface), was bedeutet, dass innerhalb einer Instanz der Anwendung gleichzeitig mehrere Dokumente geöffnet und auch angezeigt werden können. Die MDI-Anwendung spendiert hierbei jedem geöffneten Dokument ein eigenes Fenster. Diese Fenster tummeln sich in der `Windows`-Auflistung des Programmes, also letztlich des `Application`-Objektes. Haben Sie zwei Arbeitsmappen ohne weitere Manipulationen geöffnet, so wird der Ausdruck `Application.Windows.Count` vermutlich zwei ergeben.

Eigentümer eines jeden dieser beiden Fenster ist allerdings die jeweilige Arbeitsmappe. Der Ausdruck `Workbooks(i).Windows.Count` gibt die Zahl eins zurück. Sofern Sie kein weiteres Fenster öffnen, teilen sich alle Tabellenblätter das Fenster der Arbeitsmappe, zumindest theoretisch. Lassen Sie uns einen kleinen Umweg machen, bevor wir wieder auf diesen kleinen Unterschied zu sprechen kommen.

Die Frage, die sich nämlich hier erst einmal stellt, ist, wozu man das Window-Objekt überhaupt benötigt. Die Antwort sehen Sie in Abbildung 6.20, in der sich gleich zehn Ansichtsoptionen als Eigenschaften des betreffenden Window-Objekts darstellen. Wollen Sie also die Gitternetzlinien einer Tabelle ausschalten, so müssen Sie die `DisplayGridlines`-Eigenschaft des Window-Objekts auf `False` setzen. Wenn Sie diese Eigenschaft im Optionsdialog verändern, werden Sie feststellen, dass sich die Veränderung nur auf die aktive Tabelle bezieht, womit wir bei dem Begriff theoretisch des letzten Absatzes angelangt sind. Und wenn Sie nun noch ein zweites Fenster in dieser Arbeitsmappe öffnen, sehen Sie, dass die Veränderung tatsächlich nur die Darstellung der Tabelle in einem Fenster verändert hat. Ich weiß natürlich nicht, was sich Microsoft bei dieser Geschichte gedacht hat; ich hätte diese Eigenschaften der jeweiligen Tabelle zugeordnet, aber das hilft uns nicht weiter.

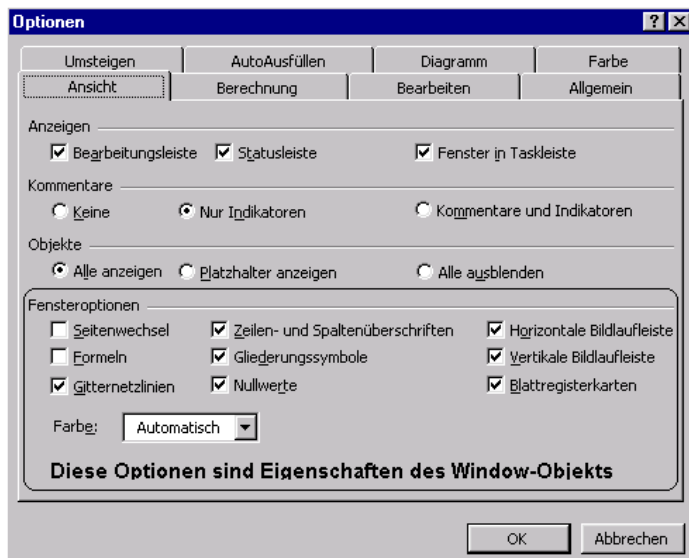


Abbildung 6.20: Optionsdialog, Eigenschaften des Window-Objekts

Zwar halten die Windows-Auflistung und das Window-Objekt noch einiges mehr an Möglichkeiten bereit, doch wir wollen uns hier auf diese Eigenschaften beschränken. In der folgenden Übersicht sind den Optionsnamen des Dialogs die Eigenschaftsnamen zugeordnet:

Option	Eigenschaft
Formeln	DisplayFormulas
Gitternetzlinie	DisplayGridlines
Farbe (der Gitternetzlinien)	GridlineColor oder GridlineColorIndex
Zeilen- und Spaltenüberschriften	DisplayHeadings
Gliederungssymbole	DisplayOutline
Nullwerte	DisplayZeros
Horizontale Bildlaufleiste	DisplayHorizontalScrollbar
Vertikale Bildlaufleiste	DisplayVerticalScrollbar
Blattregisterkarten	DisplayWorkbookTabs

Haben Sie den Seitenwechsel vermisst? Der steckt interessanterweise in der Eigenschaft `DisplayPageBreaks`, die dem jeweiligen Worksheet-Objekt zugeordnet ist.

Schauen wir uns nun den Zugriff auf eine solche Eigenschaft an. Im ersten Beispiel werden die Gitternetzlinien für die Tabelle Daten (`shtDat`) eingeschaltet und in einem Grauton formatiert:

```
shtDat.Activate
ThisWorkbook.Windows(1).DisplayGridlines = False
ThisWorkbook.Windows(1).GridlineColorIndex = 15
```

Damit wir auch die richtige Tabelle erwischen, wird diese vorher aktiviert.

6.5.3 Ereignisse

Mittels neun Ereignissen vermag ein `Workbook`-Objekt mit uns zu kommunizieren. Hinzu kommen acht Ereignisse, die wir schon vom `Worksheet`-Objekt kennen. Abgerundet wird dieses Mitteilungsbedürfnis durch drei weitere, die allerdings von dem `Window`-Objekt hervorgerufen werden.

Activate-Ereignis

Dies ist im Prinzip ein alter Bekannter, der sich so verhält, wie wir das vom `Worksheet`-Objekt her kennen. Wird die Arbeitsmappe, in der dieser Code steht, aktiviert, so tritt dieses Ereignis auf:

```
Private Sub Workbook_Activate()
End Sub
```

AddinInstall-Ereignis

AddinInstall meldet sich, wenn die Arbeitsmappe als Addin installiert wird. Diese Routine kann dazu genutzt werden, Steuerelemente oder gar Symbolleisten zu erzeugen, um eine Bedienschnittstelle zu schaffen:

```
Private Sub Workbook_AddinInstall()  
  
End Sub
```

Beachten Sie bitte auch die Bemerkungen zum Open-Ereignis.

AddinUninstall-Ereignis

Dieses Ereignis tritt auf, wenn die Arbeitsmappe als Addin deinstalliert wird, und kann dazu dienen, erzeugte Steuerelemente wieder zu entfernen:

```
Private Sub Workbook_AddinUninstall()  
  
End Sub
```

BeforeClose-Ereignis

Unmittelbar bevor die Arbeitsmappe geschlossen wird, tritt dieses abbrechbare Ereignis auf. Es kann dazu genutzt werden, gewisse Aufräumarbeiten anzustoßen. Ist irgendeine wichtige Bedingung Ihrer Einschätzung nach nicht erfüllt, so haben Sie die Möglichkeit, den Anwender zumindest zu fragen, ob er seiner Sache sicher ist. Ist er es nicht, so können Sie den Prozess abbrechen, indem Sie das Argument Cancel auf True setzen:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)  
    If MsgBox("Sind Sie..., vbYesNo) = vbNo Then  
        Cancel = True  
    Endif  
End Sub
```



Achten Sie jedoch darauf, dass Sie sich nicht mit einer Bedingung selbst umzingeln, denn wenn dieses Cancel gleich True ist, können Sie Excel nur noch per Task-Manager schließen, wodurch natürlich sämtliche Änderungen aller geöffneten Dateien verloren sind.

BeforePrint-Ereignis

BeforePrint tritt auf, bevor der Druckprozess angestoßen wird. Sie können dieses Ereignis dazu nutzen, die Arbeitsmappe neu zu berechnen oder wie in dem folgenden Beispiel den Druck auf den Farbdrucker umzuleiten, wenn die Tabelle shtDat gedruckt wird. Hierzu wird eine zumindest private Variable bPrintData benötigt, um das BeforePrint-Ereignis dann

abzubrechen, wenn der Druckvorgang gerade per Code eingeleitet wurde, denn auch dann tritt das Ereignis auf.

```
Private Sub Workbook_BeforePrint(Cancel As Boolean)
    If bPrintData Then Exit Sub
    If ActiveWindow.ActiveSheet.Name = shtDat.Name Then
        shtDat.PrintOut ActivePrinter:="ColorPrinter"
        Cancel = True
    Else
        bPrintData = True
        ActiveWindow.ActiveSheet.PrintOut _
            ActivePrinter:="MonochromePrinter"
        bPrintData = False
    End If
End Sub
```

Auch hier, letztlich wie bei allen Before-Ereignissen, kann der jeweilige Prozess durch `Cancel = True` abgebrochen werden.

BeforeSave-Ereignis

Unmittelbar vor dem Speichern wird diese Ereignisprozedur aufgerufen. Neben dem bekannten `Cancel`-Argument wird in Form von `SaveAsUI` ein logisches Argument übergeben, das uns durch ein `True` signalisiert, dass der Anwender versucht die Datei über den Dialog Speichern unter zu schließen. Wenn er das aus irgendeinem Grunde nicht soll, können Sie ihm das mitteilen und das Speichern unter abbrechen, und zwar *bevor* der Dialog erscheint:

```
Private Sub Workbook_BeforeSave(ByVal SaveAsUI As _
    Boolean, Cancel As Boolean)
    If SaveAsUI Then
        MsgBox "Die Datei sollte nicht ..."
        Cancel = True
    End If
End Sub
```

Deactivate-Ereignis

Verliert Ihre Arbeitsmappe den Fokus an eine andere Arbeitsmappe, so wird dieses Ereignis ausgelöst.

```
Private Sub Workbook_Deactivate()
End Sub
```

NewSheet-Ereignis

Nachdem ein neues Blatt, also Worksheet oder Chart, eingefügt wurde, werden wir durch dieses Ereignis davon in Kenntnis gesetzt. Das Argument `Sh` enthält einen Verweis auf das Objekt.

```
Private Sub Workbook_NewSheet(ByVal Sh As Object)

End Sub
```

Open-Ereignis

Bei dem `Open`-Ereignis handelt es sich wohl um das am häufigsten einsetzbare und damit vielleicht auch wichtigste Ereignis, das uns die Arbeitsmappe zu bieten hat.

Wenn Sie beispielsweise Datenbankzugriffe in Ihrer Applikation verwenden, ist das die passende Gelegenheit, das Öffnen der Datenbank einzuleiten oder die Verbindung zu der Client/Server-Datenbank aufzubauen, was üblicherweise das eine oder andere Sekündchen dauert. Dadurch ist sichergestellt, dass Ihre Applikation *betriebsbereit* ist, wenn die Arbeitsmappe geöffnet vor dem Anwender liegt. Denn man ist eher bereit, einen längeren Öffnungsprozess zu akzeptieren, als geraume Zeit auf die Ausführung der ersten Operation zu warten. Aber auch dann, wenn Ihre Applikation andere Arbeitsmappen benötigt, kann das Öffnen dieser hier eingeleitet werden.

Da man weder Datenbanken noch Datenbankverbindungen oder Arbeitsmappen mit einem Einzeiler öffnen sollte, verzichte ich hier auf ein anschauliches Beispiel.

```
Private Sub Workbook_Open()

End Sub
```

Auf ein Wort zu `Add-Ins` und dem bereits behandelten `AddinInstall`-Ereignis. Leichtfertig sprach ich dort davon, dass man `AddinInstall` dazu verwenden kann, Steuerelemente zu erzeugen und auch zuzuweisen, denn darum geht es ja. Solche Operationen verlagere ich stattdessen in das `Open`-Ereignis, denn dieses wird im `Addin` auch dann ausgelöst, wenn die Excel-Instanz startet und dann das `Add-In` geladen wird. Dieses wird uns von dem `AddinInstall`-Ereignis nämlich nicht gemeldet. Dadurch können Sie verhindern, dass eine versehentlich zwischenzeitlich gelöschte Symbolleiste nur durch erneute Aktivierung des `Add-Ins` im `Add-In-Manager` hervorgeholt werden kann.

Sheet*-Ereignisse

Diese Ereignisse entsprechen den bereits in 6.3.3 vorgestellten Worksheet-Ereignissen. Um jedoch unterscheiden zu können, welche Tabelle oder welches sonstige Blatt das Ereignis auslöste, wird ein Verweis auf das betreffende Objekt in Form der Variablen `Sh` übergeben, wie folgendes Beispiel zeigt:

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
```

```
End Sub
```

Window*-Ereignisse

Mit `WindowActivate`, `WindowDeactivate` und `WindowResize` haben wir die Chance auf entsprechende Ereignisse eines Fensters zu reagieren. Sie sind jedoch nur dann sinnvoll, wenn eine Arbeitsmappe gleichzeitig in mehreren Fenstern abgebildet ist.

6.6 Das Application-Objekt

Das `Application`-Objekt steht für Excel selbst. Dahinter verbirgt sich die Funktionalität, die mehr globaler Natur ist, also gewissermaßen Gültigkeit hat für alle unterhalb dieses Objekts angesiedelten Strukturen. Als Beispiel mögen die Symbolleisten oder aber die Tabellenfunktionen dienen, die gleichermaßen für alle geöffneten Arbeitsmappen gelten.

6.6.1 Eigenschaften und Methoden im Überblick

Wie nicht anders zu erwarten, hält dieses Objekt eine entmutigende Fülle von Eigenschaften und Methoden bereit. Deshalb wollen wir den Rauch ein wenig abziehen lassen, um einen freien Blick auf eine Auswahl an Eigenschaften und Methoden für den täglichen Bedarf zu ermöglichen.

Das Thema Symbolleisten werden wir separat in Kapitel 6.7 behandeln.

	Element	Beschreibung	Rückgabe
M	AddCustomList	Fügt eine benutzerdefinierte Liste hinzu	
A	AddIns	Enthält die AddIns-Auflistung	AddIns
E	AltStartupPath	Alternatives Startverzeichnis	String
M	Calculate	Berechnet alle geöffneten Arbeitsmappen	
E	Calculation	Berechnungsmodus	Long
A	CommandBars	Beinhaltet alle CommandBars, auf die innerhalb der Arbeitsmappe zugegriffen werden kann	CommandBars

	Element	Beschreibung	Rückgabe
E	CustomListCount	Anzahl der benutzerdefinierten Listen	Long
E	CutCopyMode	Ausschneide- und Kopiermodus	Long
M	DeleteCustomList	Löscht eine benutzerdefinierte Liste	
E	DisplayCommentIndicator	Darstellung der Kommentaranzeige	Long
M	FindFile	Öffnet einen Dialog zum Öffnen einer Arbeitsmappe	Boolean
M	GetCustomListContents	Gibt eine benutzerdefinierte Liste zurück	String-Array
M	GetCustomListNum	Gibt die Nummer einer benutzerdefinierten Liste zurück	Long
M	GetOpenFilename	Zeigt den Öffnen-Dialog und gibt den Dateinamen zurück, ohne die Datei zu öffnen	Variant
M	GetSaveAsFilename	Zeigt den Speichern-unter-Dialog und gibt den Dateinamen zurück, ohne die Datei zu speichern	Variant
E	LibraryPath	Bibliotheksverzeichnis	String
M	OnKey	Verbindet eine Prozedur mit einer Tastenkombination	
M	Quit	Beendet Excel	
E	ScreenUpdating	Schaltet die Bildschirmaktualisierung ein und aus	Boolean
E	StartupPath	Startverzeichnis	String
E	StatusBar	Ändert den Inhalt der StatusBar	Variant
E	TemplatesPath	Vorlagenverzeichnis	String

M = Methode, E = Eigenschaft, A = Auflistung

6.6.2 Eigenschaften und Methoden unter der Lupe

AddCustomList-Methode

Mit dieser Methode können Sie eine benutzerdefinierte Liste hinzufügen. Sie verfügt über die Argumente `ListArray` und `ByRow`. `ListArray` kann ein String-Array sein, aber auch ein Range-Objekt, wobei Sie dann mit `ByRow` bei unklaren Verhältnissen festlegen können, ob die Inhalte der ersten Spalte (`True`) oder der ersten Zeile (`False`) als Inhalte übernommen werden sollen. Ich empfehle jedoch, klare Verhältnisse zu schaffen und nur die Zellen als Range zu übergeben, die auch die tatsächlichen Werte

enthalten. Damit umgehen Sie rechteckige Bereiche, bei denen sich Excel aussuchen kann, ob die Werte aus den Spalten- oder Zeilentiteln gebildet werden sollen.

Im ersten Beispiel wird eine benutzerdefinierte Liste mit einem Array erzeugt:

```
Application.AddCustomList Array("A", "C", "B")
```

Die folgenden beiden Zeilen erzeugen je eine Liste aus den Spalten- und Zeilentiteln:

```
Application.AddCustomList shtDat.Range("A1:F1")  
Application.AddCustomList shtDat.Range("A1:A6")
```

Befinden sich leere Zellen in dem übergebenen Range-Objekt, so werden diese ignoriert, ein netter Zug von Excel.

Da jedoch diese benutzerdefinierten Listen in aller Regel mit ihrem Index verwendet werden, ist es mit einer solchen Zeile wie oben zu sehen nicht getan. Bei der Erläuterung der `GetCustomListNum`-Eigenschaft finden Sie ein Beispiel, wie man dieses Problem lösen kann.

AddIns-Auflistung

Über die `AddIns`-Auflistung haben wir Zugriff auf alle im Add-In-Manger des Extras-Menüs aufgeführten Add-Ins. Wir können zum Beispiel prüfen, ob ein Add-In vorhanden und installiert ist, können es installieren und deinstallieren oder neue Add-Ins hinzufügen. Da die `AddIns`-Auflistung nur als Element des `Application`-Objekts existiert, dürfen wir den Ausdruck `Application` ohne Gefahr weglassen.

Wie bei jeder Auflistung können wir auch hier auf die `Item`- und `Count`-Eigenschaften der Auflistung vertrauen. `Count` gibt die Anzahl der installierten Add-Ins zurück:

```
MsgBox AddIns.Count
```

Die `Item`-Methode erlaubt den Zugriff auf ein installiertes Add-In über einen eins-basierten Index oder den Namen des Add-Ins. Als Besonderheit sei erwähnt, dass sich hinter dem Namen des Add-Ins nicht der Dateiname verbirgt, sondern der Titel, der in den Datei-Eigenschaften eingetragen wurde. Dieser Titel wird auch als Text im Add-Ins-Manager angezeigt:

```
AddIns(10)  
AddIns("Das ist der Add-In-Titel")
```

Um also zu prüfen, ob ein Add-In installiert ist, könnten wir die `AddIns`-Auflistung durchlaufen und jedes Element mit unserem Suchbegriff ver-

gleichen. Allerdings verhält sich die Auflistung sehr großzügig, wenn wir ein bereits installiertes Add-In erneut hinzufügen. Somit können wir mit der `Add`-Methode der `AddIns`-Auflistung das gewünschte Add-In einfach hinzufügen und anschließend mittels der `Installed`-Eigenschaft installieren:

```
strAddIn = Application.LibraryPath & "\TestAddIn.xla"  
AddIns.Add(strAddIn).Installed = True
```

Wie man hier sehen kann, gibt die `Add`-Methode der Auflistung das Add-In als Objekt zurück, wodurch wir die Eigenschaftszuweisung unmittelbar folgen lassen können. Die Deinstallation kann so aussehen:

```
AddIns(10).Installed = False  
AddIns("Das ist der Add-In-Titel").Installed = False
```

Mehr muss man im Umgang mit fertigen Add-Ins kaum wissen.

AltStartupPath-Eigenschaft

Diese Eigenschaft repräsentiert das alternative Startverzeichnis, das als zusätzlicher Startordner im Register Allgemein des Optionsdialogs zu finden ist:

```
Application.AltStartupPath = "D:\Test"
```

Calculate-Methode

Mit `Calculate` werden alle geöffneten Arbeitsmappen neu berechnet:

```
Application.Calculate
```

Calculation-Eigenschaft

Hinter dieser Eigenschaft verbirgt sich der Berechnungsmodus, wie er auch im Register Berechnung des Optionsdialogs zu finden ist. Folgende Konstanten sind definiert:

Wert und Konstante	Bedeutung
-4135 – <code>xlCalculationManual</code>	manuell
2 – <code>xlCalculationSemiautomatic</code>	automatisch außer bei Mehrfachoperationen
-4105 – <code>xlCalculationAutomatic</code>	automatisch

In Arbeitsmappen, die sehr viele Formeln enthalten, bringt das Ausschalten der automatischen Berechnung schon einiges an Zeitgewinn, denn jede Zellmanipulation führt bei Automatik zur erneuten Berechnung der Mappe:

```
Application.Calculation = xlCalculationManual
'Zellmanipulationen
Application.Calculation = xlCalculationAutomatic
```

CommandBars-Auflistung

Über `Application.CommandBars` haben wir Zugriff auf alle Symbol- und Menüleisten sowie deren Steuerelemente. Die hierzu erforderlichen Techniken werden wir uns in Kapitel 6.7 gesondert ansehen.

CustomListCount-Eigenschaft

Die `CustomListCount`-Eigenschaft gibt die Anzahl der benutzerdefinierten Listen zurück:

```
MsgBox = Application.CustomListCount
```

Ein Beispiel finden Sie bei der `GetCustomListNum`-Eigenschaft.

CutCopyMode-Eigenschaft

Diese Eigenschaft stellt den Kopiermodus von Excel mit folgenden Konstanten dar:

Wert und Konstante	Bedeutung
0 – False	Kopieren oder Ausschneiden beendet
1 – xlCopy	Kopiermodus
2 – xlCut	Ausschneidemodus

Der wichtigste Wert ist `False`, denn damit können Sie den Laufrahmen zur Ruhe bringen, der bei Kopier- oder Ausschneideoperationen per VBA aktiv bleibt und den Anwender nur unnötigerweise irritiert:

```
rngCopy.Copy
rngPaste.PasteSpecial xlPasteFormats
Application.CutCopyMode = False
```

DeleteCustomList-Methode

Mit dieser Methode können Sie eine benutzerdefinierte Liste entfernen. Das ist immer dann sinnvoll, wenn Sie eine solche Liste zu Sortierzwecken per Code erzeugt haben:

```
Application.DeleteCustomList iList
```

`DeleteCustomList` benötigt den Zeiger auf die Liste, den man sich aber beim Erzeugen merken kann. Oder man entfernt eben die letzte Liste:

```
Application.DeleteCustomList Application.CustomListCount
```

DisplayCommentIndicator-Eigenschaft

Diese Eigenschaft entspricht den Einstellungen der Kommentardarstellung, wie sie auch in Abbildung 6.20 zu sehen ist. Die folgenden Konstanten stehen zur Verfügung:

Wert und Konstante	Bedeutung
0 – xlNoIndicator	keine
-1 – xlCommentIndicatorOnly	nur Indikatoren
1 – xlCommentAndIndicator	Kommentare und Indikatoren

Ein interessantes Beispiel zu dieser Eigenschaft ist unter der Beschreibung der OnKey-Methode zu finden.

FindFile-Methode

Diese Methode bringt den Datei-öffnen-Dialog auf den Bildschirm und teilt uns über die Rückgabe True oder False mit, ob der Anwender nun eine Datei geöffnet hat oder den Dialog mit der Abbrechen-Schaltfläche geschlossen hat.

Im folgenden Beispiel wird die neu geöffnete Datei der Workbook-Variablen des Funktionsnamens OpenNewWorkbook zugewiesen:

```
Private Function OpenNewWorkbook() As Workbook
    If Application.FindFile Then
        Set OpenNewWorkbook = Workbooks(Workbooks.Count)
    End If
End Sub
```

Der Trick liegt darin, dass die neue Datei den größten Index haben muss, den wir über die Count-Eigenschaft der Workbooks-Auflistung ermitteln können.

GetCustomListContents-Eigenschaft

Diese Methode ermöglicht uns, den Inhalt einer benutzerdefinierten Liste auszulesen. Die Methode benötigt hierzu den Index der gewünschten Liste und gibt uns ein String-Array zurück:

```
Private Sub ShowCustomList()
    Dim varList As Variant      'Array für Rückgabe
    Dim iList As Long           'Zeiger in varList
    Dim strMessage As String    'Textausgabe
    strMessage = "Die Liste enthält ..." & vbCrLf
```

```

varList = Application.GetCustomListContents(7)
For iList = 1 To UBound(varList)
    strMessage = strMessage & varList(iList) & vbCrLf
Next
MsgBox strMessage
End Sub

```

GetCustomListNum-Eigenschaften

Diese Eigenschaft gibt die Nummer einer benutzerdefinierten Liste zurück, die ihr als Argument übergeben wird. Existiert die Liste nicht, gibt sie null zurück.

Sie kann dazu verwendet werden, zu ermitteln, ob eine Liste existiert, um sie gegebenenfalls zu erzeugen, sollte sie nicht existieren:

```

Sub SortWithCostumList()
    Dim iList As Long
    iList = Application.GetCustomListNum( _
        Array("A", "C", "B")) + 1
    If iList < 2 Then
        Application.AddCustomList Array("A", "C", "B")
        iList = Application.CustomListCount + 1
    End If
    Me.Columns(1).Sort Key1:=Me.Range("A1"), _
        Header:=xlYes, OrderCustom:=iList
End Sub

```

Da wir als OrderCustom-Argument der Sort-Methode den Index (oder Offset, wie Microsoft das nennt) benötigen, müssen wir diesen etwas umständlichen Weg beschreiten. Sie sehen hier auch, dass die Sort-Methode einen um eins größeren Wert als den tatsächlichen Index erwartet, was doch ein wenig irritiert. Vielleicht heißt dieser Wert auch deshalb Offset und nicht Index.

GetOpenFilename

GetOpenFilename gaukelt dem Anwender vor, dass er eine Datei öffnet, gibt uns aber in Wirklichkeit den Namen der ausgewählten Datei zurück, ohne diese zu öffnen. Folgende Argumente stehen bereit:

Argument	Bedeutung
FileFilter	(optional) Filtertext sowie durch Komma getrennt die Filtermaske Beispiel 1: Excel-Arbeitsmappen »Excel-Dateien (*.xls), *.xls« Beispiel 2: Excel-Arbeitsmappen und Textdateien in einem Filter »Importdateien *.xls *.txt, *.xls; *.txt« Beispiel 3: Excel-Arbeitsmappen und Textdateien in zwei Filtern »Excel-Dateien *.xls, *.xls, Text-Dateien *.txt, *.txt«
FilterIndex	(optional) Ein Zeiger auf den gewünschten Filter (siehe Beispiel 3).
Title	(optional) Titel des Dialogs
ButtonText	(optional) (nur bei MACs)
MultiSelect	(optional) Wenn True, dürfen mehrere Dateien ausgewählt werden. In diesem Falle ist die Rückgabe ein eins-basiertes String-Array, das Sie allerdings als Variant deklarieren müssen.

Beim FileFilter ist also entscheidend, ob die Werte per Komma oder Semikolon getrennt werden. Was ich nicht ganz begreife, ist, dass das für solche Fälle übliche Pipe-Zeichen |, Chr(124), nicht zur Trennung von Text und Maske verwendet wurde.

```
strFileFilter = "Excel-Arbeitsmappen *.xls, *.xls"
strFile = Application.GetOpenFilename(strFileFilter)
```

In einer neuen Excel-Instanz wird das Standard-Arbeitsverzeichnis verwendet. Verändert der Anwender in diesem oder dem regulären Datei-öffnen-Dialog den Pfad, so wird dieser neue Pfad zum Initialisierungsverzeichnis des nächsten Aufrufs des Dialogs. Es gelang mir nicht, diesen Wert zu beeinflussen. Dies ist höchst bedauerlich, denn es zeugt von gutem Stil, den Anwender gezielt in das Verzeichnis zu lenken, in dem auch die zu öffnenden Dateien liegen und, ihn nicht in komplexen Strukturen herumsuchen zu lassen.

GetSaveAsFilename

Diese Methode verhält sich ähnlich wie die GetOpenFilename-Methode, nur dass ihr alle mit MultiSelect zusammenhängenden Fähigkeiten fehlen. Somit fehlt auch das letzte Argument MultiSelect. Auch die Rückgabe ist ein wenig anders, denn diese Methode gibt entweder den vollständigen Namen der Datei zurück oder eben False, wenn der Anwender den Dialog mit der Abbrechen-Schaltfläche geschlossen hat.


```
strFile = Application.GetSaveAsFilename
```

LibraryPath

Diese schreibgeschützte Eigenschaft gibt das Bibliotheksverzeichnis zurück, in der Regel das Verzeichnis Makro hinter dem Verzeichnis, in dem die Excel.exe liegt. Es kann dazu verwendet werden, das Speicherverzeichnis für Add-Ins festzulegen.

```
FileCopy "X:\..", Application.LibraryPath & "\MyAddIn.xla"
```

OnKey-Methode

Diese Methode präsentiert sich nach außen nicht in dem Gewand der Objektorientierung und Ereignissteuerung, sonst gäbe es ein entsprechendes Key-Ereignis oder gar die übliche Kombination von KeyDown-, KeyPress- und KeyUp-Ereignissen. Stattdessen mutet sie ein wenig wie die spätneolithischen Sprachelemente aus VBA 2 (Excel 5) an. Was tut OnKey?

Mit der OnKey-Methode können wir Tasten oder Tastenkombinationen einer Prozedur zuweisen und diese Zuweisung auch wieder aufheben. Die Parametrierung der Tasten entspricht hierbei der, die auch bei der SendKeys-Anweisung Verwendung findet. Sie verfügt über das Key-Argumente und das optionale Procedure-Argument. Mit Key geben Sie die gewünschte Tastenkombination als String an, wobei Sie sowohl Zeichen (a, A oder beispielsweise 1) zuordnen können als auch Sonder-Funktions-tasten wie {F1}, {HOME} oder {ENTER}. Letztere werden in den geschweiften Klammern übergeben. Die Umschalt-, Steuerungs- und Alt-Tasten können durch Voranstellen der Zeichen +, ^ und % berücksichtigt werden:

"A"	entspricht dem großen A
"%A"	entspricht Alt-A
"{HOME}"	entspricht der Pos1-Taste
"^{RETURN}"	entspricht Strg-Return

Mit dem zweiten, optionalen Procedure-Argument stellen Sie die Verbindung zu einer Public Sub her. Fehlt dieses Argument, wird eine eventuell vorhandene Verbindung aufgelöst und der Tastaturbefehl wird wie gewohnt von Excel interpretiert.

Das folgende Beispiel findet in unserem Haus Verwendung, wo in Monatstabellen Termine für alle einsehbar gepflegt werden, wobei von Zellkommentaren reger Gebrauch gemacht wird. Die (F1)-Taste wird zweckentfremdet und schaltet die Kommentaranzeige ein und aus.

Das Open-Ereignis der Arbeitsmappe dient dazu, diese Verbindung herzustellen, die Zellkommentare anzuzeigen und einen Hinweis auf die Verwendung der (F1)-Taste in der StatusBar auszugeben:

```
Private Sub Workbook_Open()
    Application.OnKey "{F1}", "ToggleCommentDisplayMode"
    Application.DisplayCommentIndicator = _
        xlCommentAndIndicator
    Application.StatusBar = "Benutzen Sie die F1-Taste ..."
End Sub
```

Das BeforeClose-Ereignis der Arbeitsmappe wird genutzt, um die Verbindung zu entfernen und die StatusBar zurückzusetzen:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    Application.OnKey "{F1}"
    Application.StatusBar = False
End Sub
```

Bleibt noch die Prozedur ToggleCommentDisplayMode, die wie folgt aussieht:

```
Public Sub ToggleCommentDisplayMode()
    Application.DisplayCommentIndicator = _
        Application.DisplayCommentIndicator * -1
    Application.StatusBar = False
End Sub
```

Der aktuelle DisplayCommentIndicator wird negiert und die StatusBar zurückgesetzt. Da die Funktion ja gerade genutzt wurde, können wir den Hinweis aus der StatusBar getrost entfernen.

Quit

Diese Methode kann dazu verwendet werden, Excel zu beenden:

```
Application.Quit
```

ScreenUpdating

Mit der ScreenUpdating-Eigenschaft können Sie die Bildschirmaktualisierung in Excel ein- und ausschalten. Es ist zwar schön, wenn man sieht, dass zumindest einer arbeitet, aber ständig beobachtbare Wechsel zwischen Tabellen und Arbeitsmappen gehören nicht zum guten Ton. Wenn der Anwender also den Startschuss gegeben hat, schalten Sie die Bildschirmaktualisierung aus und am Ende Ihrer Bearbeitung wieder ein:

```
Private Sub cmdGo_Click()
    '...
    Application.ScreenUpdating = False
    'Programmaufrufe
    Application.ScreenUpdating = True
End Sub
```

StartupPath

Mit dieser Eigenschaft haben Sie Zugriff auf das Startverzeichnis:

```
MsgBox Application.StartupPath
```

StatusBar

Die StatusBar-Eigenschaft ermöglicht uns das Ausgeben von (Status-)Texten in der Statusleiste von Excel:

```
Application.StatusBar = "Datei xxx wird geöffnet..."
```

Dieser Text bleiben so lange stehen, bis er überschrieben wird oder Excel mit der folgenden Anweisung die Kontrolle über die Statusleiste zurückerhält:

```
Application.StatusBar = False
```

TemplatesPath

Diese schreibgeschützte Eigenschaft gibt das Vorlagenverzeichnis des aktuellen Anwenders zurück:

```
strPath = Application.TemplatesPath
```

6.6.3 Ereignisse

Das Application-Objekt verfügt über Ereignisse, die ihm allerdings erst entlockt werden müssen. Die hierzu nötige Technik erfordert jedoch die Verwendung eines Klassenmoduls, womit wir uns erst in Kapitel 11 gründlich auseinander setzen werden. Dennoch werden wir es hier vorstellen, weil es von Amts wegen hierhin gehört.

Sie benötigen ein Klassenmodul, in dessen Kopf Sie folgende Deklaration unterbringen:

```
Private WithEvents appExcel As Application
```

Nun müssen Sie appExcel noch mit Excel selbst verbinden, wozu Sie das Initialize-Ereignis der Klasse verwenden können:

```
Private Sub Class_Initialize()
    Set appExcel = Application
End Sub
```

Danach stehen Ihnen in diesem Klassenmodul alle Ereignisse des `Application`-Objekts zur Verfügung. Es unterstützt alle Ereignisse, die auch das `Workbook`-Objekt produzieren kann. Diese sind bereits vorgestellt worden, sodass wir uns hier nur das neue Ereignis ansehen müssen.

NewWorkbook-Ereignis

Wenn Sie sich das `NewSheet`-Ereignis der Arbeitsmappe vor Augen führen, dann erkennen Sie das Muster dieses Ereignisses sofort. Sobald eine neue Arbeitsmappe in der laufenden Instanz erzeugt wird, kommt dieses Ereignis zum Zuge und übergibt uns ein `Workbook`-Objekt, mit dem wir auf die neu erzeugte Arbeitsmappe zugreifen können:

```
Private Sub appExcel_NewWorkbook(ByVal Wb As Workbook)
End Sub
```

6.6.4 Tabellenfunktionen in VBA

Was Excel so mächtig macht, sind nicht zuletzt die ausgefeilten Tabellenfunktionen. In Excel 5 und 7 waren Sie Methoden des `Application`-Objekts, ab Excel 97 hat sich ein Objekt namens `WorksheetFunction` zwischen das Objekt `Application` und die Methoden eingeschmuggelt. Aus

```
Application.Sum(rngTest)
```

wurde

```
Application.WorksheetFunction.Sum(rngTest)
```

Allerdings hat das kaum jemand zur Kenntnis genommen, denn die Methodenaufrufe funktionieren immer noch ohne `WorksheetFunction`. Es stellt sich allerdings die Frage, wie lange noch.

VBAListe.xls

Ein richtiger Veteran unter den Komponenten einer Excel-Installation ist die Datei `VBAListe.xls`, denn sie ist seit Excel 5 dabei. Sie enthält zum einen die Schlüsselwörter des VBA 2 in Deutsch und Englisch. Nicht wenige sind auf diesen Blödsinn hereingefallen und schrieben in Excel 5 und 7 deutsche VBA-Programme. Excel 97 bereitete diesem Spuk mit VBA 5 ein Ende, wodurch dieser Tabelle wohl keine Bedeutung mehr zukommt.

Die zweite Tabelle hingegen enthält die Tabellenfunktionen in deutscher und englischer Sprache. Dass aus der `SUMME`-Funktion ein `Sum`-Methode wird, könnte man ja noch erraten, aber wer weiß schon, dass die Fakultät als abgekürzte `Faculty` zur `Fact`-Methode wird?

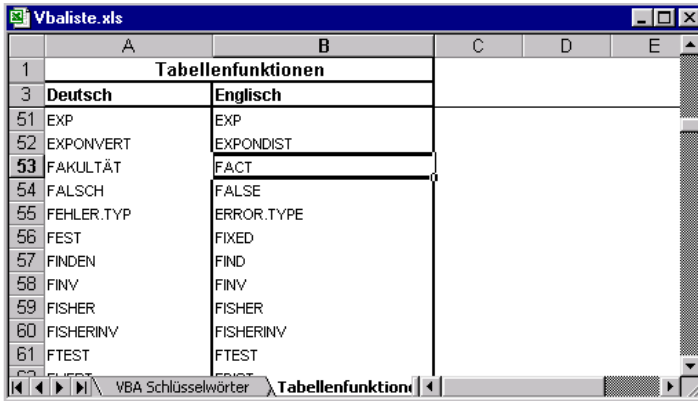


Abbildung 6.21: VBAListe.xls, Tabellenfunktionen

Die VBAListe.xls ist auf ihre alten Tage umgezogen und befindet sich nun nicht mehr in dem Verzeichnis, in dem auch die Excel.exe liegt, sondern in dem hinter Office liegenden nationalisierten Pfad, der durch eine Nummer gekennzeichnet ist (1031 für Deutsch Deutschland).

Parametrierung der Tabellenfunktionen

In Parametrierung und Datenformaten entsprechen die Methoden den Tabellenfunktionen, wie die Abbildung 6.22 bezüglich der Parametrierung zeigt.

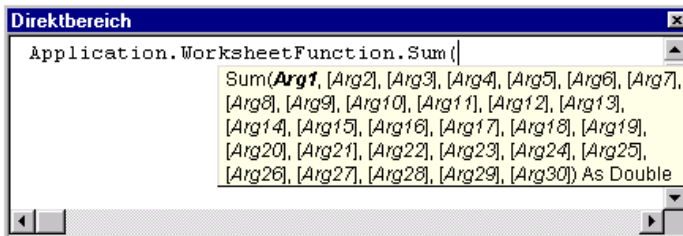


Abbildung 6.22: Parametrierung der Sum-Methode

Und überall dort, wo Sie in Excel Zellen angeben konnten, dürfen wir in VBA auch ein Range-Objekt verfüttern:

```
MsgBox Application.WorksheetFunction.Sum(rngTest)
```

Dieser übergebene Range darf auch abhängig von dem Argumententyp aus mehreren Zellen bestehen. Und dort, wo er aus mehreren Zellen zusammengesetzt sein kann, dürfen wir auch numerische Arrays übergeben:

```
Dim lngTest(1) As Long
lngTest(0) = 1
lngTest(1) = 2
MsgBox Application.WorksheetFunction.Sum(lngTest)
```

Obwohl ich in Excel reichlich Gebrauch mache von verschiedensten Tabellenfunktionen, reduziert sich der Umfang auf einige wenige Funktionen, die von Amts wegen in der Sprache VBA ihre Berechtigung hätten:

- ▼ Sum (SUMME)
- ▼ Min (MIN)
- ▼ Max (MAX)
- ▼ Average (MITTELWERT)

Aber auch die Funktionen CountIf (ZÄHLENWENN) und SumIf (SUMMEWENN) sind gelegentlich von Nutzen, wie die folgende Zeile zeigt:

```
MsgBox Application.WorksheetFunction.Sumif( _
    sht1.Columns(1), "A", sht1.Columns(2))
```

6.7 CommandBars und CommandBarControls

Das eigentlich Neue an diesen CommandBars ist, dass sie die klassischen Symbolleisten und Menüs nebst Kontextmenüs in einem gemeinsamen Objekt zusammenführen. Wenn man ein wenig darüber nachdenkt, so klingt das sehr schlüssig. In Abbildung 6.23 sehen Sie ein Menü inklusive Untermenü, das in der Symbolleiste *Formatting* untergebracht wurde. In der Menüleiste wiederum strahlt uns zwischen dem Datei- und dem Bearbeiten-Menü Smiley entgegen.

Der alte Heraklit von Ephesos muss das alles vorausgesehen haben, als er sein legendäres *pánta rhei* (alles fließt) formulierte.

Dem Anwender ist es ohnehin egal, worauf er herumklickt. Menüs sind eben am oberen Rand und Symbolleisten etwas tiefer und meist etwas bunter. Und wenn man ein Element anklickt, muss etwas passieren.

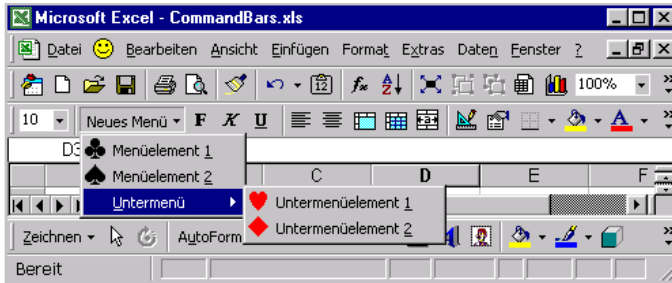


Abbildung 6.23: CommandBars, Menü in Symbolleiste

Beginnen wir mit einer leichten Fingerübung und listen in einer Tabelle alle verfügbaren CommandBars mit ihren Systemnamen, den lokalen Namen, unter dem sie im Dialog Anpassen erscheinen, und der Anzahl der jeweiligen Steuerelemente der CommandBar auf:

```
Private Sub cmdShowCommandBars_Click()
    Dim cb As CommandBar
    Dim iRow As Long
    iRow = 1
    For Each cb In Application.CommandBars
        iRow = iRow + 1
        Me.Cells(iRow, 1).Value = cb.Name
        Me.Cells(iRow, 2).Value = cb.NameLocal
        Me.Cells(iRow, 3).Value = cb.Controls.Count
    Next
End Sub
```

Das Ergebnis sehen Sie in Abbildung 6.24.

	A	B	C
	Systemname	Lokaler Name	Anzahl der Steuerelemente
1	Worksheet Menu Bar	Arbeitsblatt-Menüleiste	11
2	Chart Menu Bar	Diagrammmenüleiste	10
3	Standard	Standard	28
4	Formatting	Format	27
5	PivotTable	PivotTable	33
6	Chart	Diagramm	9
7	Reviewing	Überarbeiten	9
8	Forms	Formular	16
9	Stop Recording	Aufzeichnung beenden	2
10	External Data	Externe Daten	7
11	Auditing	Detektiv	9
12			

Abbildung 6.24: CommandBars, Liste der verfügbaren Symbolleisten

Obwohl Excel es, vermutlich noch aus Gründen der Kompatibilität zu früheren Versionen, noch eine gewisse Zeit unterstützen wird, ist das gewohnte Anbinden einer benutzerdefinierten Symbolleiste an eine Arbeitsmappe nicht mehr zeitgemäß. Und wenn Sie zu anderen Produkten der Office-Familie wechseln, werden Sie diese Funktionalität vergeblich suchen. Es ist also höchste Zeit für eine Beschäftigung mit diesen neuen Befehlsleisten, wie sie in der Übersetzung heißen.

6.7.1 CommandBar erzeugen

Wenn Sie Ihre Anwendung mit einer Befehlsleiste ausstatten wollen, so muss diese per Code erzeugt und dynamisch mit dem Code verbunden werden. Der ideale Einstieg dafür ist das `Open`-Ereignis der Arbeitsmappe, in der wir mit dieser Befehlsleiste arbeiten wollen. Wir werden eine Symbolleiste erzeugen, drei `CommandBarButton`s darauf platzieren und diese mit drei `Public`-Prozeduren verbinden, die sich in einem separaten Modul befinden.

Da dies möglicherweise nicht die einzige Aktion ist, die Sie in dieser Prozedur einleiten wollen, bietet sich an, eine separate Prozedur zu schreiben und diese in der Ereignisroutine aufzurufen:

```
Private Sub Workbook_Open()  
    Call CreateCommandBar  
End Sub
```

In der Prozedur `CreateCommandBar` benötigen wir ein `CommandBar`-Objekt (`cb`) und ein `CommandBarButton`-Objekt (`ccbDummy`). Zuerst wird geprüft, ob die Befehlsleiste schon existiert. Wird sie nicht gefunden, so wird sie erzeugt und erhält die drei `CommandBarButton`s:

```
Private Sub CreateCommandBar()  
    Dim cb As CommandBar                'CommandBar-Objekt  
    Dim ccbDummy As CommandBarButton    'Dummy-Button  
    'Prüfen, ob CommandBar bereits existiert  
    For Each cb In Application.CommandBars  
        If cb.Name = "GoTo" Then Exit For  
    Next  
    If cb Is Nothing Then  
        'CommandBar erzeugen  
        Set cb = Application.CommandBars.Add(Name:="GoTo")  
        'Button 1 erzeugen  
        Set ccbDummy = cb.Controls.Add(msoControlButton)  
        ccbDummy.OnAction = "Button1_Click"  
        ccbDummy.Caption = "Button 1"  
        ccbDummy.FaceId = 71  
    End If  
End Sub
```



```

    ccbDummy.Style = msoButtonIconAndCaption
    'Button 2 erzeugen
    Set ccbDummy = cb.Controls.Add(msoControlButton)
    ccbDummy.OnAction = "Button2_Click"
    ccbDummy.Caption = "Button 2"
    ccbDummy.FaceId = 72
    ccbDummy.Style = msoButtonIconAndCaption
    'Button 3 erzeugen
    Set ccbDummy = cb.Controls.Add(msoControlButton)
    ccbDummy.OnAction = "Button3_Click"
    ccbDummy.Caption = "Button 3"
    ccbDummy.FaceId = 73
    ccbDummy.Style = msoButtonIconAndCaption
End If
'CommandBar aktivieren
cb.Enabled = True
cb.Visible = True
End Sub

```

Zieht man die Zeilen des zweiten und des dritten `CommandBarButton`s ab, so haben wir es in der Tat nur mit sechs Zeilen zu tun. In der ersten wird die `CommandBar` erzeugt und gleichzeitig der Variablen `cb` zugewiesen. Anschließend wird der `CommandBarButton` erzeugt und ebenfalls in derselben Zeile der Variablen `ccbDummy` übergeben. Mit der `OnAction`-Eigenschaft erfolgt die Zuordnung zu unserer Prozedur, die `Caption`-Eigenschaft nimmt die Aufschrift entgegen, `FaceID` legt das Icon fest, das angezeigt werden soll, und über die Konstante `msoButtonIconAndCaption` der `Style`-Eigenschaft wird abschließend festgelegt, dass der `CommandBarButton` Text und Bild anzeigen soll. Damit haben Sie im Prinzip alles erledigt.



Abbildung 6.25: CommandBars, Befehlsleiste GoTo

6.7.2 FaceIDs

Sie werden sich nun vielleicht fragen, woher man weiß, welche `FaceID` welches Bild darstellt. In der Datei `CommandBars.xls` auf der Buch-CD befindet sich eine Tabelle, in die eine kleine Routine jeweils 300 Icons nebst dazugehöriger `FaceID` einträgt, wie Abbildung 6.26 zeigt:

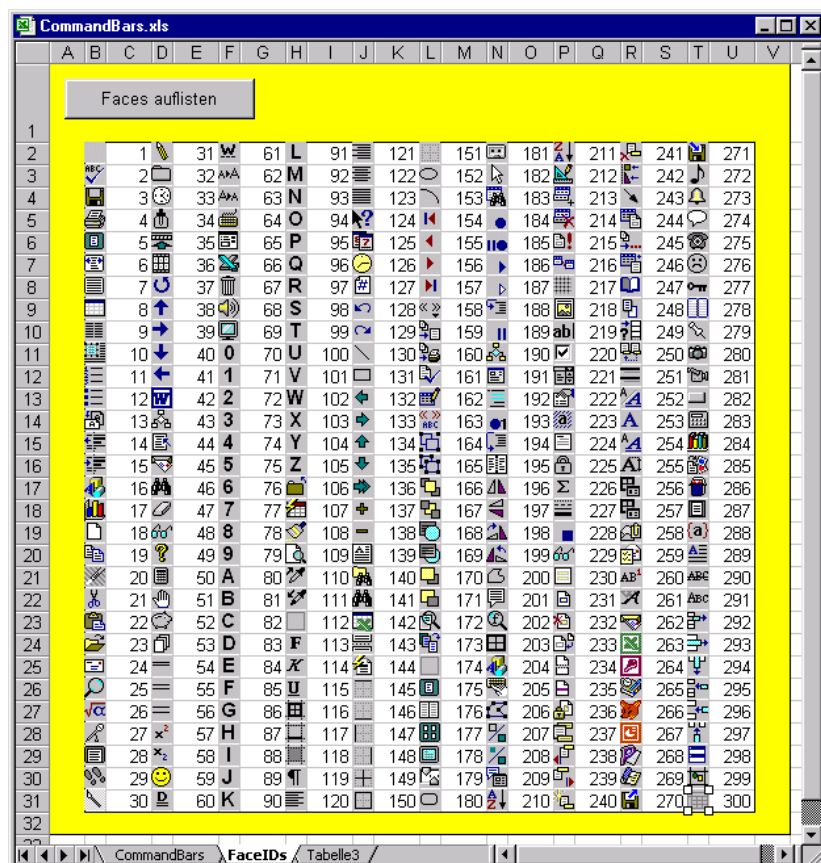


Abbildung 6.26: CommandBars, Tabelle FaceIDs

Das ist im Prinzip die einzige Chance, eine größere Gruppe von FaceIDs in einer Gegenüberstellung zu sehen, denn die Hilfe ist hier keine echte Hilfe.

```
Private Sub cmdShowFaces_Click()  
    Dim shpFace As Shape                'Button-Shape  
    Dim cb As CommandBar                'CommandBar-Objekt  
    Dim cbbDummy As CommandBarButton    'Dummy-Button  
    Dim iFaceID As Long                 'Face-ID  
    Dim iRow As Long                   'Zeilenzeiger  
    Dim iCol As Long                   'Spaltenzeiger  
    'Prüfen, ob CommandBar bereits existiert
```

```
For Each cb In Application.CommandBars
```

```

    If cb.Name = "Test" Then Exit For
Next
If cb Is Nothing Then
    'CommandBar erzeugen
    Set cb = Application.CommandBars.Add(Name:="Test")
    'Button erzeugen
    Set cbbDummy = cb.Controls.Add(msoControlButton)
    cbbDummy.Style = msoButtonIcon
Else
    'CommandBarButton-Objekt zuweisen
    Set cbbDummy = _
Application.CommandBars("Test").Controls(1)
End If
' Tabelle leeren und alte Shapes löschen
shtFace.UsedRange.ClearContents
For iFaceID = shtFace.Shapes.Count To 1 Step -1
    If shtFace.Shapes(iFaceID).Name <> "cmdShowFace" Then
        shtFace.Shapes(iFaceID).Delete
    End If
Next
'Faces eintragen
Do
    iFaceID = iFaceID + 1
    iRow = ((iFaceID - 1) Mod 30) + 2
    iCol = ((iFaceID - 1) \ 30) * 2 + 2
    cbbDummy.FaceId = iFaceID
    cbbDummy.CopyFace
    shtFace.Paste
    Set shpFace = shtFace.Shapes(shtFace.Shapes.Count)
    shpFace.Top = shtFace.Cells(iRow, iCol).Top
    shpFace.Left = shtFace.Cells(iRow, iCol).Left
    shtFace.Cells(iRow, iCol + 1).Value = iFaceID
Loop Until iFaceID = 300
cbbDummy.Parent.Delete
End Sub

```

Der hervorgehobene Code betrifft den Umgang mit dem CommandBar- und dem CommandBarButton-Objekt und weist außer der CopyFace-Methode nichts Neues auf. Unserem Dummy-Button ccbDummy wird reihum die FaceID 1 bis 300 zugewiesen und das Bild über diese Methode CopyFace in die Zwischenablage transportiert.

Shape-Objekt erzeugen

Neu hingegen ist der Rest innerhalb der Do-Loop-Schleife, denn dort wird per Paste-Methode des Worksheet-Objekts einfach ein so genanntes Shape erzeugt. Unter Shapes fallen alle die Objekte, die in Tabellen platziert werden können. Die Zeile

```
Set shpFace = shtFace.Shapes(shtFace.Shapes.Count)
```

geht einfach davon aus, dass dieses neue Shape den höchsten Index trägt, der durch `shtFace.Shapes.Count` ermittelt wird. Nun haben wir einen Objektverweis auf dieses Shape (ohne genau zu wissen, was sich dahinter verbirgt) und können mittels der `Top`- und der `Left`-Eigenschaft seine Position innerhalb der Tabelle festlegen. Bei dieser Positionierung verlassen wir uns wiederum ganz auf das Range-Objekt, dessen `Top`- und `Left`-Eigenschaften wir kurzerhand an das Shape-Objekt weiterreichen:

```
shpFace.Top = shtFace.Cells(iRow, iCol).Top  
shpFace.Left = shtFace.Cells(iRow, iCol).Left
```

Nun wird noch die `FaceID` in die benachbarte Zelle eingetragen, damit wir das gewünschte Bild der `FaceID` zuweisen können. Etwas außergewöhnlich ist vielleicht noch die Erzeugung des Zeilenzeigers `iRow` und des Spaltenzeigers `iCol`. Basis beider Zuweisungen ist die Variable `iFaceID`.

Der Zeilenzeiger

Die Anordnung der Bilder erfolgt in Blöcken à 30 Zeilen. Da in der Tabelle eine Art Kopf existiert, haben wir einen Versatz von eins zwischen `FaceID` und Zeilenzeiger (`iFaceID - 1`). Die Modulo-Division mit 30 gibt uns den Rest der Division als ganze Zahl, also beispielsweise $3 \text{ Mod } 30 = 3$. Zu diesem Wert wird eine Basis von zwei hinzugezählt, weil wir bei der Modulo-Division eins abzogen, damit die 30 bei dieser Modulo-Division nicht zu 0 wird.

```
iRow = ((iFaceID - 1) Mod 30) + 2
```

Der Spaltenzeiger

Der Spaltenzeiger wird ähnlich gebildet, nur wird nicht der Rest der Division weiter verarbeitet, sondern deren ganzzahliger Anteil.

```
iCol = ((iFaceID - 1) \ 30) * 2 + 2
```

6.7.3 Fazit

Sie haben es bei Excel mit einem sehr komplexen Objektmodell zu tun, das ich Ihnen hier vorzustellen versuchte. Es dauert erfahrungsgemäß eine gewisse Zeit, bis man damit sicher umgehen kann. Doch dann haben Sie ein unglaublich mächtiges Werkzeug in der Hand.



Sprachelemente, die zweite

7

Kapitelüberblick

7.1	Arrays	302
7.1.1	Array-Funktionen	303
7.1.2	Mehrdimensionale Arrays	305
7.1.3	Dynamische Arrays	306
7.1.4	Das Beispiel	308
7.2	Collections	314
7.3	Benutzerdefinierte Datentypen	317
7.4	Registry-Zugriffe	318
7.4.1	VB and VBA Program Settings	318
7.4.2	Sprachelemente für Registry-Zugriffe	321
7.4.3	Das Beispiel Registry.xls	323
7.5	Runden	330
7.6	Farben	332
7.6.1	Sprachelemente	332
7.6.2	Ein Beispiel	335
7.7	API	337
7.7.1	Der WinAPI-Viewer	337
7.7.2	Struktur einer API-Deklaration	339
7.7.3	Anwendername ermitteln	339
7.7.4	Pause im Programmablauf	340
7.7.5	Laufwerkstyp ermitteln	341
7.7.6	Ein Timer	343
7.7.7	Locale Settings	347
7.8	Dateioperationen	349
7.8.1	Operationen an Dateien	350
7.8.2	Ein Beispiel	353
7.8.3	Operationen in Dateien	356
7.8.4	Ein Beispiel	357
7.9	Rekursionen	359
7.9.1	Berechnung einer Fakultät	360
7.9.2	Bäume durchsuchen	361
7.10	Programmausführung unterbrechen	365

In diesem Kapitel werden Sprachelemente und Techniken beleuchtet, die vielleicht in der Fortgeschrittenenecke anzusiedeln sind, gleichwohl aber zum Handwerkszeug eines Entwicklers gehören.

7.1 Arrays

Eine Variable ist ein benannter Speicherplatz und ein Array eine Gruppe von Variablen, die unter einem Namen zusammengefasst sind. Zur Unterscheidung der einzelnen Variablen wird ein Index verwendet.

Ähnliches kennen wir bereits aus Auflistungen. Ebenso wie die zweite Tabelle einer Arbeitsmappe mit `Worksheets(2)` beschrieben wird, können wir den Monat Mai als fünften Monat eines Monats-Arrays mit `strMonate(5)` ansprechen.

Mit einer kleinen Anleihe bei der Geometrie könnte man eine normale Variable mit einem Punkt gleichsetzen und ein eindimensionales Datenfeld mit einer Geraden. Auch für zweidimensionale Datenfelder als Flächen und dreidimensionale als Kuben hält der Vergleich noch Stand. VBA bietet uns maximal 60 (in Worten: sechzig) Dimensionen, die allerdings nur von theoretischem Interesse sind, da der Speicherbedarf bei wachsenden Dimensionen eine Potenzreihe ist.

Ein Array, oft auch Datenfeld genannt, wird ebenso wie punktförmige Variablen mit `Dim`, `Private` oder `Public` erzeugt. Allerdings erhält sie am Ende ihres Namens ein Klammernpaar. Ist dieses Klammernpaar leer, so heißt das: Mach bitte ein Array. Wie groß es sein soll, weiß ich noch nicht, aber ich lasse es dich wissen.

```
Dim lngKoSt() As Long
```

Dieses Array `lngKoSt` ist im Prinzip dynamisch und mit diesen Typen werden wir uns im weiteren Verlauf auseinander setzen. Kehren wir zurück zu den statischen Arrays.

Hier ein Beispiel für ein String-Array, dessen Größe bekannt ist:

```
Dim strMonate(12) As String
```

Wie viele Elemente hat dieses Array? 12 werden Sie vielleicht sagen. Kann sein, würde ich antworten. Die Zahl 12 sagt lediglich aus, dass das oberste Elemente dieses Arrays den Index 12 hat. Die Zahl der Elemente können Sie aber erst dann bestimmen, wenn Sie wissen, welchen Index das Kleinste hat. In Kapitel 4 war bereits die Rede von der Option Base-Anweisung, und diese legt die untere Grenze aller Arrays fest, die in dem Modul erzeugt werden. Stünde also im Modulkopf die Anweisung `Option Base 0`, so böte dieses Array Platz für 13 Monate, bei `Option Base 1` hingegen

nur für 12. Übrigens, fehlt eine Option Base-Anweisung in dem betreffenden Modul, so wird 0 als untere Grenze angenommen. Sofern nicht ausdrücklich erwähnt, unterstellen wir eine Option Base von 0.

Wenn Sie auf diese Art und Weise ein Monats-Array erzeugen, so haben wir eigentlich ein Argument bereits ausgelassen, denn Arrays werden wie folgt definiert:

```
Dim VarName([Untergrenze To ]Obergrenze) As Typ
```

Was in den []-Klammern steht, dürfen wir bekanntlich weglassen. Wollen wir also unabhängig von der Option Base 12 Monate mit den Indizes 1 bis 12 haben, so müssen wir schreiben:

```
Dim strMonate(1 To 12) As String
```

Doch wer tut das schon?

Nun wollen wir dieses Array mit den 12 Monaten füllen:

```
Private Sub FillMonthArray()
    Dim strMonate(12) As String
    Dim iMonat As Long
    For iMonat = 1 To 12
        strMonate(iMonat) = _
            Format(DateSerial(2000, iMonat, 1),
                "mmm")
    Next
End Sub
```

In diesem kleinen Beispiel ist auch zu ersehen, wie auf ein Element eines Arrays zugegriffen wird. Geben Sie einfach in dem Klammerspaar den gewünschten Index an:

```
strMonate(5) 'ergibt den Wonnemonat Mai.
```

7.1.1 Array-Funktionen

Array-Funktion

Die Array-Funktion erzeugt ein Variant-Array aus der übergebenen Argumentliste:

```
Dim varAcc As Variant
varAcc = Array("Fahrzeugkosten", "Bürokosten")
```

LBound-Funktion

Mit dieser Funktion können Sie die untere (Lower Boundary) Grenze eines Arrays oder besser gesagt einer Dimension eines Arrays ermitteln:

```
LBound(ArrayName[, Dimension])
```

Immer wenn Sie in VBA etwas weglassen dürfen, ersetzt VBA dies durch einen Wert. In diesem Falle ist es 1 für die erste Dimension. Im ersten Beispiel gibt die LBound-Funktion als untere Grenze 5 zurück.

```
Dim lngMyArray(5 To 8)
MsgBox LBound(lngMyArray) 'ergibt 5
```

Betrachten wir nun die beiden unteren Grenzen eines zweidimensionalen Arrays:

```
Dim lngMyArray(1 To 7, 0 To 2)
MsgBox LBound(lngMyArray, 1) 'ergibt 1
MsgBox LBound(lngMyArray, 2) 'ergibt 0
```

Da die untere(n) Grenze(n) üblicherweise bei null liegen, ist die LBound-Funktion seltener anzutreffen als die nun folgende UBound-Funktion.

```
Dim lngMyArray(1 To 7, 0 To 2)
MsgBox UBound(lngMyArray, 1) 'ergibt 7
MsgBox UBound(lngMyArray, 2) 'ergibt 2
```

UBound-Funktionen

Die UBound-Funktion arbeitet nach dem Muster der LBound-Funktion und gibt die obere Grenze (Upper Boundary) einer Dimension zurück:

```
LBound(ArrayName[, Dimension])
```

Wird die Dimension nicht angegeben, so nimmt auch die UBound-Funktion 1 an und gibt die obere Grenze der ersten Dimension zurück.

Filter-Funktion

Mit dieser Funktion kann ein eindimensionales String-Array dahingehend untersucht werden, ob ein anzugebender Text in den Array-Elementen enthalten oder nicht enthalten ist. Die Rückgabe ist ein nullbasiertes String-Array mit den so gefilterten Elementen.

Filter(SourceArray, Match[, Include[, Compare]])

SourceArray	eindimensionales String-Array
Match	einzuschließender oder auszuschließender Text

Include	(optional) Ist Include gleich True oder wird dieses Argument weggelassen, so werden die Array-Elemente zurückgegeben, die den unter Match angegebenen Text enthalten. Ist Include gleich False, so werden die Array-Elemente zurückgegeben, die den unter Match angegebenen Text nicht enthalten.
Compare	(optional) 1 – vbUseCompareOption führt einen Vergleich unter Verwendung der Option Compare-Anweisung durch. 0 – vbBinaryCompare führt einen binären Vergleich durch. 1 – vbTextCompare führt einen Textvergleich durch.

Im folgenden Beispielcode werden alle Elemente des Arrays `strAcc()` zurückgegeben, die den Ausdruck »kosten« enthalten:

```
Private Sub FilterTest()
    Dim strAcc(4) As String
    Dim strCostAcc() As String
    Dim iAcc As Long
    Dim strMessage As String
    strAcc(0) = "Fahrzeugkosten"
    strAcc(1) = "Bürokosten"
    strAcc(2) = "Projekterlöse"
    strAcc(3) = "Verkaufserlöse"
    strAcc(4) = "Lohnkosten"
    strCostAcc = Filter(strAcc, "kosten", True)
    For iAcc = 0 To UBound(strCostAcc)
        strMessage = strMessage & strCostAcc(iAcc) & vbCrLf
    Next
    MsgBox strMessage
End Sub
```

Frage: Wie müsste diese Prozedur abgeändert werden, damit das Array `strAcc` mittels der Array-Funktion gefüllt wird? Die Antwort finden Sie wie immer im Kapitel Lösungen am Ende des Buchs.

7.1.2 Mehrdimensionale Arrays

Wie bereits erwähnt wurde, sind bis zu 60 Dimensionen in einem Array möglich. Doch dies ist eine eher theoretische Größe. Aber durchaus üblich sind zweidimensionale Datenfelder.

Mehrdimensionale Arrays werden nach folgendem Muster erzeugt:

```
Dim VarName(Index1[, Index2[, ...Indexn]]) As Typ
```

Hierbei steht Index für die Konstruktion [Untergrenze To]Obergrenze.

Die folgenden Beispiele erzeugen jeweils ein zweidimensionales, null-basiertes Array mit 8 x 8 Elementen:

```
Dim lngChess(7, 7)
Dim lngChess(0 To 7, 0 To 7)
```

Nun erzeugen wir ein eins-basiertes Array mit ebenfalls 8 x 8 Elementen:

```
Dim lngChess(1 To 8, 1 To 8)
```

Frage: Welche Aussagen lassen sich über folgendes Array machen?

```
Dim lngChess(8, 8)
```

Bei der unterstellten Option Base 0 verfügt dieses Array über 9 x 9 Elemente, denn es ist gleichbedeutend mit

```
Dim lngChess(0 To 8, 0 To 8)
```

Dass es ganz nützlich sein kann, sich die Monatsnamen oder die Namen von Niederlassungen in einem Array zu speichern, leuchtet ein. Aber ein mehrdimensionales Array?

Das nahezu ausschließliche Einsatzgebiet mehrdimensionaler Arrays ist der Transport von Informationen ganz allgemein zwischen Prozeduren. Üblicherweise handelt es sich bei diesen Arrays um dynamische Vertreter ihrer Gattung, um die es sich im folgenden Abschnitt dreht. Dort werden wir auch einem passenden Beispiel begegnen.

7.1.3 Dynamische Arrays

Bei dynamischen Arrays ist übrigens nur die letzte Dimension dynamisch. Hat ein Array nur eine Dimension, ist diese erste auch gleichzeitig die letzte und somit in ihrer Größe veränderbar.

Wir benötigen ein dynamisches Array, wenn die Anzahl der Elemente noch nicht bekannt ist oder im Zuge der weiteren Bearbeitung variieren wird. Das Array wird also mit einem leeren Klammernpaar deklariert und erhält später mit der ReDim-Anweisung den oberen und vielleicht auch noch den unteren, optionalen Index.

```
Dim strSheets() As String
ReDim strSheets(ThisWorkbook.Worksheets.Count)
```

Wenn Sie dies in einer Zeile probieren, erhalten Sie von VBA den Hinweis »Konstanter Ausdruck erforderlich«.

Nachdem Sie ein Array ohne Indexangabe erzeugt haben, können Sie es jederzeit mittels ReDim neu dimensionieren. Hier stehen zwei Varianten zur Verfügung:

```
ReDim VarName(NeuerIndex)
```

und

```
ReDim Preserve VarName(NeuerIndex)
```

Wohingegen im ersten Fall das Array seinen Inhalt verliert, bleibt er bei Verwendung des Schlüsselworts `Preserve` vorhanden. Mit einer Einschränkung, denn Elemente jenseits der neuen oberen Grenze sind natürlich verschwunden, wie im folgenden Beispiel das Konto 8001:

```
Dim lngKonten() As Long
ReDim lngKosten(1)
lngKonten(0) = 8000
lngKonten(1) = 8001
ReDim Preserve lngKonten(0)
```

Die Aussage »Erhöhe die obere Grenze um ein Element.« hat aber auf den zweiten Blick so ihre Tücken. Wie groß ist die derzeitige obere Grenze?

Das sieht nach einer typischen Aufgabe für die `UBound`-Funktion aus, doch der folgende Code produziert in der `ReDim`-Zeile den Fehler 9 – Index außerhalb des gültigen Bereichs:

```
Private Function ThisDoesNotWork() As Variant
    Dim strNumbers() As Long
    Dim i As Long
    For i = 1 To 10
        If i Mod 2 = 0 Then
            ReDim Preserve strNumbers(UBound(strNumbers)
                                     + 1)
            strNumbers(UBound(strNumbers)) = i
        End If
    Next
    ThisDoesNotWork = strNumbers
End Function
```

Die Ursache liegt darin, dass die `UBound`-Funktion nichts mit einem leeren Array anfangen kann. Wir könnten nun mit einer `If`-Konstruktion diesen Sonderfall abfangen. Dagegen ist absolut nichts einzuwenden.

Ich habe mir jedoch angewöhnt, das Element null außerhalb der Schleife zuzuweisen, sodass ich in der Schleife auf das oberste Element zugreifen kann. Danach wird Platz geschaffen für den nächsten Eintrag, indem die obere Grenze inkrementiert wird. Dadurch befindet sich am Ende dieses Prozesses ein Element mehr im Array, als tatsächlich verwendet wird. Und dieses gilt es dann noch zu entfernen:

```

Private Function ButThisWorks() As Variant
    Dim strNumbers() As Long
    Dim i As Long
    ReDim strNumbers(0)
    For i = 1 To 10
        If i Mod 2 = 0 Then
            strNumbers(UBound(strNumbers)) = i
            ReDim Preserve strNumbers(UBound(strNumbers)
                                + 1)
        End If
    Next
    ReDim Preserve strNumbers(UBound(strNumbers) - 1)
    ButThisWorks = strNumbers
End Sub

```

Der Nachteil dieser Vorgehensweise liegt ganz klar darin, dass unser Array generell ein leeres Element vor sich herschiebt, das am Ende entfernt werden muss. Wenn es möglich sein kann, dass das Array am Ende der Schleife leer ist, sollten Sie, um vor Überraschungen sicher zu sein, diesen Fall abfangen und ein Empty zuweisen.

7.1.4 Das Beispiel

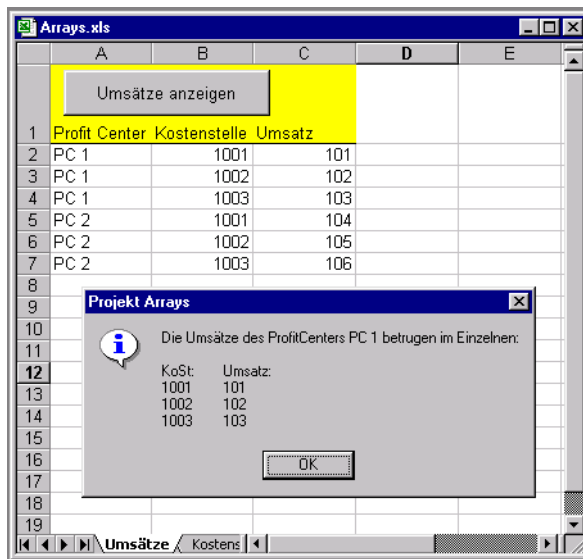


Abbildung 7.1: Arrays, Tabelle Umsätze mit InfoMessage

Abbildung 7.1 zeigt die zugrunde liegende Datentabelle und auch das Ergebnis in Form einer MessageBox. In der Tabelle stehen verschiedene Profit Centers mit je einer Gruppe von Kostenstellen und Umsätzen. Wir wissen weder, wie viele Profit Centers enthalten sind, noch kennen wir die jeweilige Zahl an Kostenstellen je Profit Center.

Selbstredend könnte man diese kleine Anwendung in einer Prozedur unterbringen, die jedoch eine Reihe verschachtelter Schleifen beherbergen würde und reichlich unübersichtlich wäre. Das Ermitteln der beteiligten Profit Centers und der Kostenstellen und Umsätze je Profit Center in jeweils eigenen Funktionen ist sicherlich der bessere Weg.

Abbildung 7.2 zeigt die beteiligten Prozeduren und den Datenfluss zwischen den einzelnen Prozeduren.

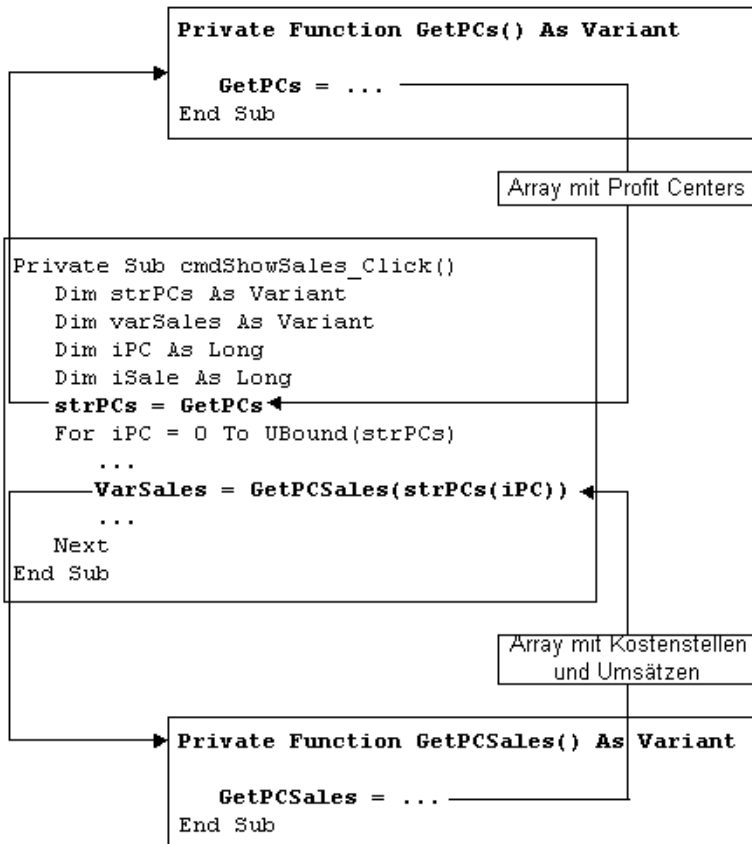


Abbildung 7.2: Datenfluss zwischen den Prozeduren

Beginnen wir mit den beiden Funktionen.

Die Funktion GetPCs

GetPCs hat die Aufgabe, die verschiedenartigen Inhalte der Spalte A in ein Array zu verpacken und dieses an die aufrufende Stelle zurückzutransportieren. Besteht die CurrentRegion aus nur einer Zeile (die hat sogar ein leeren Zellbereich), wird Empty zurückgegeben. Dieses Empty kostet uns den im Übrigen verwendbaren Datentyp String, weshalb GetPCs selbst ein Variant sein muss. Doch das ist mir eine klare Information wert, denn die Rückgabe Empty kann an der aufrufenden Stelle einfach und interpretationsfrei geprüft werden. Der Rest der Funktion enthält nichts Neues. Hat sich der Inhalt der Spalte A gegenüber dem vorherigen Wert verändert, wird dieser neue Wert in das oberste Element eingetragen und die Grenze erhöht.

Private Function GetPCs() As Variant

```
'ProfitCenters in Array zurück geben
Dim rngSales As Range           'Range-Objekt
Dim strPCs() As String         'Array mit Profitcenters
Dim strPC As String             'aktuelles Profitcenter
Dim iRowSales As Long           'Zeilenzeiger in rngSales
On Error GoTo errHandler
Set rngSales = shtSales.Range("A1")
'Tabelle sortieren
rngSales.Sort key1:=rngSales, header:=xlYes
'Abbruch, wenn keine Daten
If rngSales.CurrentRegion.Rows.Count = 1 Then
    GetPCs = Empty
    Exit Function
End If
'erstes Element hinzufügen
ReDim strPCs(0)
'Tabelle zeilenweise abarbeiten
For iRowSales = 2 To rngSales.CurrentRegion.Rows.Count
    'Prüfen, ob sich ProfitCenter geändert hat
    If strPC <> rngSales.Cells(iRowSales, 1).Value Then
        'ProfitCenter in oberstes Array-Element schreiben
        strPC = rngSales.Cells(iRowSales, 1).Value
        strPCs(UBound(strPCs)) = strPC
        'obere Array-Grenze erhöhen
        ReDim Preserve strPCs(UBound(strPCs) + 1)
    End If
Next
```

```

'letztes Array-Element entfernen ...
ReDim Preserve strPCs(UBound(strPCs) - 1)
'... und Array zuweisen
GetPCs = strPCs
Exit Function

```

```

errHandler:
    Err.Raise Err.Number, "shtSales:GetPCs"
End Function

```

Die Funktion GetPCSales

Diese Funktion erhält das gewünschte Profit Center als Argument und gibt alle Kostenstellen und Umsätze in einem Array zurück, die diesem Profit Center entsprechen.

```

Private Function GetPCSales(ByVal strPC As String) _
    As Variant
    'Kostenstellen und Umsätze des übergebenen ProfitCenters
    'in Array zurückgeben
    Dim rngSales As Range          'Range-Objekt
    Dim varValues As Variant       'Zwischen-Array
    Dim iRowSales As Long          'Zeilenzeiger in rngSales
    On Error GoTo errHandler
    Set rngSales = shtSales.Range("A1")
    'erstes Elementepaar hinzufügen
    ReDim varValues(1, 0)
    'Tabelle zeilenweise abarbeiten
    For iRowSales = 2 To rngSales.CurrentRegion.Rows.Count
        'Prüfen, ob gesuchtes ProfitCenter
        If rngSales.Cells(iRowSales, 1).Value = strPC Then
            'KoSt und Umsatz in letztes Elementepaar
            'eintragen
            varValues(0, UBound(varValues, 2)) = _
                CLng(rngSales.Cells(iRowSales,
                    2).Value)
            varValues(1, UBound(varValues, 2)) = _
                rngSales.Cells(iRowSales, 3).Value
            'obere Array-Grenze erhöhen
            ReDim Preserve _
                varValues(1, UBound(varValues, 2) + 1)
        End If
    Next
    'Empty zurückgeben, wenn obere Grenze immer noch 0
    If UBound(varValues, 2) = 0 Then

```

```

    GetPCSales = Empty
Else
    'letztes Array-Element entfernen ...
    ReDim Preserve varValues(1, UBound(varValues, 2) - 1)
    '... und Array zuweisen
    GetPCSales = varValues
End If
Exit Function

```

errHandler:

```

    Err.Raise Err.Number, "shtSales:GetPCSales"
End Function

```

Der wesentliche Unterschied zu der Funktion `GetPCs` liegt darin, dass diese Funktion ein zweidimensionales Array erzeugt. In der ersten Dimension liegt die inhaltliche Gliederung in Kostenstelle und Umsatz, wohingegen die zweite Dimension die Datensätze aufweist.

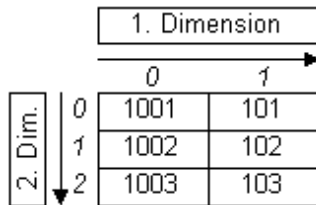


Abbildung 7.3: Das PCSales-Array

Da lediglich die letzte Dimension eines Arrays dynamisch ist, muss das PCSales-Array so aufgebaut werden, wie in Abbildung 7.3 zu sehen ist. Die Inkrementierung der oberen Grenze in der zweiten, dynamischen Dimension erfolgt dann auf diese Weise:

```

ReDim Preserve varValues(1, UBound(varValues, 2) + 1)

```

Die Größe der ersten Dimension wird also wiederholt (`varValues(1,)`), die der zweiten Dimension ermittelt (`UBound(varValues, 2)`) und um eins erhöht (`+ 1`).

Die Prozedur `cmdShowSales_Click`

Hier können wir die Mühen der beiden Funktionen ernten und uns auf die beiden dort erzeugten Arrays verlassen.

```

Private Sub cmdShowSales_Click()
    'Kostenstellen und Umsätze je ProfitCenter in einer
    'MessageBox anzeigen

```



```

Dim strPCs As Variant      'Array mit Profit Centers
Dim varSales As Variant    'Array mit Kost und Umsätzen
Dim iPC As Long            'Zeiger in strPCs()
Dim iSale As Long          'Zeiger in varSales
Dim strMessage As String   'Text der MessageBox
On Error GoTo errHandler
'vorhandene ProfitCenters ermitteln
strPCs = GetPCs
If IsEmpty(strPCs) Then
    cError.InfoMessage "Die Tabelle enthält keine Daten."
    Exit Sub
End If
'alle ProfitCenters abarbeiten
For iPC = 0 To UBound(strPCs)
    'Titel erzeugen
    strMessage = "Die Umsätze des ProfitCenters " & _
        "strPCs(iPC) & " betrug im Einzelnen:" & _
        vbCrLf & vbCrLf & "KoSt:" & vbTab & _
        "Umsatz:" & vbCrLf
    'Kostenstellen und Umsätze zurückgeben lassen, ...
    varSales = GetPCSales(strPCs(iPC))
    '... in Schleife abarbeiten ...
    For iSale = 0 To UBound(varSales, 2)
        '... und dem Message-String zuweisen
        strMessage = strMessage & varSales(0, iSale) & _
            vbTab & varSales(1, iSale) & vbCrLf
    Next
    'Message ausgeben
    cError.InfoMessage strMessage
Next
Exit Sub

errHandler:
If InStr(1, Err.Source, ":") = 0 Then
    Err.Source = "shtSales:cmdShowSales_Click"
End If
cError.ErrorMessage
End Sub

```

Das Profit-Center-Array strPCs ist als Variant erzeugt worden, aber dennoch mit dem Typpräfix str versehen worden. Was soll das?

`varSales` ist ein echter `Variant`, da er eine Kostenstelle als `Long`-Wert und einen Umsatz als `Double` enthält. `strPCs` ist jedoch ausschließlich mit `Strings` gefüllt und der `Variant` wird nur deshalb benötigt, weil wir den Wert `Empty` transportieren müssen, wenn keine Profit Centers gefunden wurden. Und der Präfix soll mich im Code darüber informieren, mit welchem Datentyp hier zu rechnen ist, und das ist nun mal ein `String`.

Schlussbemerkungen zu diesem Beispiel

Mit Kommentaren und `ErrorHandler` haben wir 120 Zeilen zur Lösung dieses doch recht kleinen Problems benötigt, gute 60 Zeilen werden es wohl netto sein.

Natürlich ließe sich das Ganze verkürzen und sogar in einer Prozedur unterbringen, aber auf Kosten der Übersichtlichkeit. Die Aufteilung der Lösung in drei Teile wiederum ist zwingend geboten, denn dadurch haben wir eine Art Kapselung geschaffen, die uns gewaltige Vorteile verschafft, sollte sich irgendetwas an der zugrunde liegenden Datenstruktur ändern. In diesem Fall müssen wir uns lediglich um die beiden Funktionen kümmern. Auf die Hauptroutine werden sich diese Änderungen dann nicht auswirken, man kann sogar sagen, nicht auswirken *können*!

7.2 Collections

Eine `Collection` ist im Prinzip eine Objekt-Auflistung, die *wir selbst* erzeugen können. In Kapitel 5 wurden Auflistungen theoretisch betrachtet. Auf existierende `Collections` haben wir bereits in Kapitel 7 zugegriffen, zum Beispiel die `Workbooks-Collection` oder die `Worksheets-Collection`. In Kapitel 8 wird uns auch eine `Controls`-Auflistung begegnen, in der alle Steuerelemente eines Containers enthalten sind.

Uns steht ein Datentyp namens `Collection` zur Verfügung, der in der Lage ist, Objekte per Code aufzunehmen und auf Anforderung wieder herauszurücken. Diese `Collection` weiß auch, wie viele Objekte sie beherbergt. Und sie ermöglicht uns auch, einzelne Objekte wieder aus der `Collection` zu entfernen. Folgende Methoden und Eigenschaften bietet sie uns an:

Add-Methode

Mit der `Add`-Methode wird ein Objekt zur `Collection` hinzugefügt.

Add Item, [Key[, Before[, After]]]

<i>Item</i>	das hinzuzufügende Objekt
<i>Key</i>	(optional) Alphanumerischer Key (Schlüssel), unter dem das Objekt später in der Collection angesprochen werden kann.
<i>Before</i>	(optional) Key oder Index des Objekts, <i>vor</i> dem das neue Objekt eingefügt werden soll.
<i>After</i>	(optional) Key oder Index des Objekts, <i>nach</i> dem das neue Objekt eingefügt werden soll.

Nur eines der Argumente `Before` und `After` darf übergeben werden. Wird keines der beiden Argumente angegeben, wird das Objekt der `Collection` angefügt. Und wenn Sie eines der beiden Argumente als `Key` angeben, dürfen darin keine nationalen Sonderzeichen enthalten sein, was in gutem Code ohnehin nicht der Fall ist!

Im folgenden Beispiel wird einer `Collection` ein `Worksheet`-Objekt unter seinem Namen vor der derzeitigen Position 1 hinzugefügt:

```
colSheets.Add shtSales, shtSales.Name, 1
```

Wenn Sie keinen `Key` angeben, so können wir dennoch über den 1-basierten Index auf ein einzelnes Objekt der `Collection` zugreifen.

Count-Eigenschaft

Die `Count`-Eigenschaft gibt die Anzahl der in der `Collection` enthaltenen Objekt zurück. Da der Index von `Collections` 1-basiert ist, entspricht der Wert der `Count`-Eigenschaft der oberen Indexgrenze.

Item-Methode

Mit dieser Methode können wir auf ein einzelnes, in der `Collection` enthaltenes Objekt zugreifen, als Argument sind der Index und der `Key` des Objekts möglich. Die `Item`-Methode ist die Standard-Methode der `Collection` und darf somit weggelassen werden:

```
MsgBox colSheets.Item(1).Name
MsgBox colSheets(1).Name
MsgBox colSheets.Item("Sales").Name
MsgBox colSheets("Sales").Name
```

Die Rückgabe der `Item`-Methode ist das Objekt selbst, welches naturgemäß über seine individuellen Eigenschaften und Methoden verfügt. Die Option `Elemente` automatisch auflisten greift nach Eingabe des Punkts hinter der `Item`-Klammer natürlich nicht, da der Editor nicht wissen kann, welcher Art das dahinter stehende Objekt ist. Haben Sie eine ungültige Eigenschaft oder Methode angegeben, so wird das zur Laufzeit mit dem Fehler 438 quittiert.

Remove-Methode

Die `Remove`-Methode dient dem Entfernen eines Objekts aus der `Collection`. Auch sie arbeitet mit dem Index oder dem Key des Objekts:

```
colSheets.Remove 1
colSheets.Remove "Sales"
```

Ein Beispiel

Im folgenden Beispiel wird eine `Collection` mit allen Eigenschaften und Methoden zusammenhängend dargestellt:

```
Private Sub SomeCollectionExperiments()
    Dim strSheets(1 To 2) As String
    Dim colSheets As Collection
    Dim iSheet As Long
    Set colSheets = New Collection
    strSheets(1) = "Sales"
    strSheets(2) = "Costs"
    colSheets.Add ThisWorkbook.Worksheets(strSheets(1)), _
        strSheets(1)
    colSheets.Add ThisWorkbook.Worksheets(strSheets(2)), _
        strSheets(2)
    For iSheet = colSheets.Count To 1 Step -1
        MsgBox colSheets.Item(iSheet).Name
        'MsgBox colSheets(iSheet).Name
        colSheets.Remove strSheets(iSheet)
    Next
End Sub
```

Die einzelnen Objekte einer `Collection` können auch über eine `ForEach`-Schleife einer Objektvariablen zugewiesen werden:

```
For Each shtAct in colSheets
    shtAct...
Next
```

Wozu benötigt man eigene Collections?

Um existierende Excel-Objekte zu verwalten sind sie eigentlich nicht zwingend erforderlich. Eine Worksheet-Collection könnten wir auch über ein String-Array mit dem Namen der einzelnen Tabellen verwalten.

Interessant sind sie eher im Zusammenhang mit der Verwaltung eigener Objekte in einer eigenen Collection-Klasse. Doch da macht sich negativ bemerkbar, dass wir keine Standard-Methoden deklarieren können.

7.3 Benutzerdefinierte Datentypen

Dieser Begriff ist vielleicht etwas irreführend, denn wir können keine Datentypen erzeugen, die über die VBA-Typen hinausgehen, also in Ergänzung von Single und Double einen Triple-Datentyp schaffen oder etwas Ähnliches. Aber wir können aus den bestehenden Datentypen einen mehrgliedrigen Begriff schaffen. Ein Beispiel:

```
Option Explicit
Private Type udtPerson
    Firstname As String
    Name As String
    Age As Long
End Type

Private Sub UserDataType()
    Dim udtDuck As udtPerson
    udtDuck.Name = "Scrooge"
    udtDuck.Firstname = "McDuck"
    udtDuck.Age = 85
    MsgBox udtDuck.Firstname & " " & udtDuck.Name & ", " _
        udtDuck.Age
End Sub
```

Auf Modulebene (und nur dort ist das möglich) werden zwischen den Anweisungen Type und End Type die Teilkomponenten dieses Datentyps in Form der bekannten Variablendeklaration angegeben. Auch Arrays können als Variablen Teil eines benutzerdefinierten Datentyps sein.

Im Code selbst wird die Variable udtDuck als udtPerson deklariert, womit sie deren Struktur übernimmt. Die Variable udtDuck verhält sich nun im Code wie ein Objekt, wie Abbildung 7.4 zeigt. Nach Eingabe des Punkts nach dem Variablennamen udtDuck präsentieren sich die Elemente dieses Datentyps wie Eigenschaften.

In diesem Objekt-ähnlichen Verhalten liegen die Vor- und die Nachteile dieser User-Defined Types, wie sie auch genannt werden. Als Vorteil lässt sich sicherlich verbuchen, dass man komplexe Strukturen damit viel übersichtlicher abbilden kann, als das eine Reihe einzelner Variablen vermag. Der Nachteil liegt aber darin, dass sie *echten* Objekten klar darin unterlegen sind.

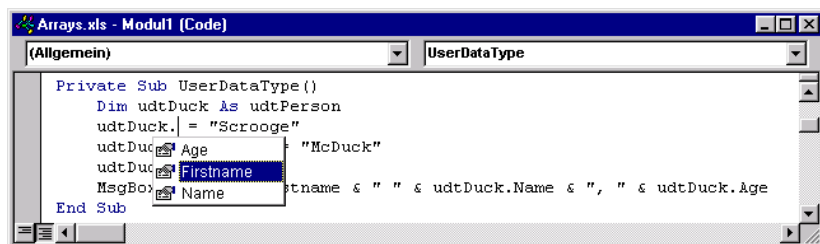


Abbildung 7.4: Benutzerdefinierte Datentypen im Code

Echte Objekte werden wir in Kapitel 11 zusammenbauen. Wenn Sie diese kennen und, so hoffe ich, schätzen gelernt haben, werden Sie schon wissen, wann Sie Klassenmodule schreiben oder benutzerdefinierte Datentypen einsetzen.

7.4 Registry-Zugriffe

7.4.1 VB and VBA Program Settings

Die Registry enthält die zentralen Konfigurationsdaten der Hardware, des Betriebssystems und der Anwendungsprogramme sowie die Profile der an diesem Rechner registrierten Anwender.

In Abbildung 7.5 sehen Sie die Registry, die Sie mit der Regedit.exe in Ihrem Windows-Verzeichnis starten können.

Üblicherweise ist der Zugriff auf beliebige Einträge in der Registry eine haarige Angelegenheit, die ohne API-Zugriffe nicht zu bewerkstelligen ist. Wenn Sie sich damit auseinander setzen wollen, empfehle ich Ihnen Dan Appleman's Visual Basic Guide to the Win32 API. Suchen Sie sich bitte einen Rechner zum Experimentieren, den Sie nicht für produktive Zwecke benötigen. Je nach Sorgfalt, die Sie dabei an den Tag legen, wird Ihr Rechner möglicherweise behaupten gar kein Rechner mehr zu sein.

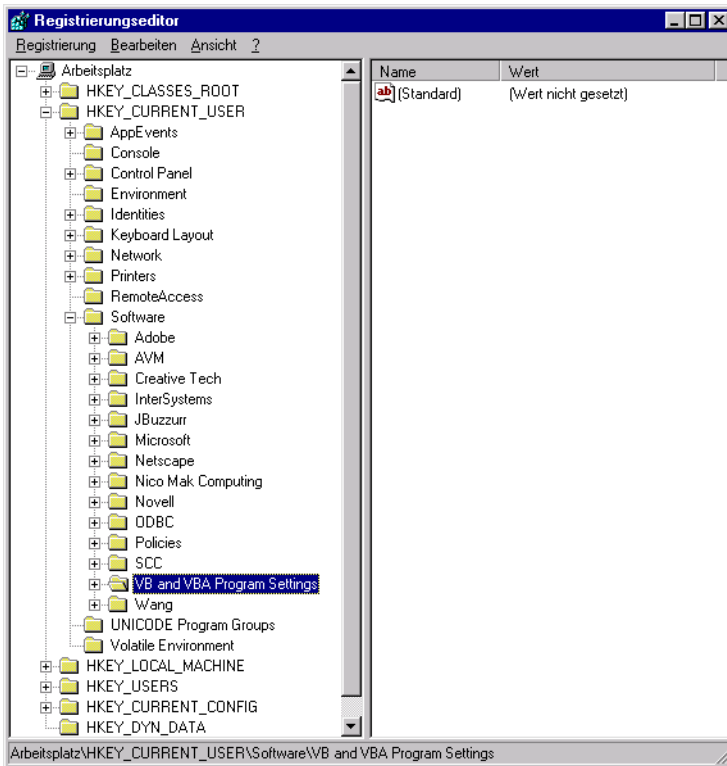


Abbildung 7.5: Die Registry

Microsoft hat uns aber für Zugriffe aus VB oder VBA eine Ecke eingerichtet, die uns einen sehr einfachen Zugang zur Registry ermöglicht. Dort haben wir es auch nur mit dem Datentyp `String` zu tun, wohingegen die restliche Registry voller Datentypen ist, die uns in VBA (und auch VB) *direkt* gar nicht zur Verfügung stehen.

Die für VBA eingerichtete Stelle finden Sie unter

HKEY_CURRENT_USER\Software\VB and VBA Program Settings

Die Buch-CD enthält eine Datei namens `Registry.xls`, in der sich die in Abbildung 7.6 dargestellte Tabelle befindet. Dort finden Sie drei Beispieleinträge sowie Schaltflächen zum Schreiben, Lesen und Löschen von Registry-Einträgen.

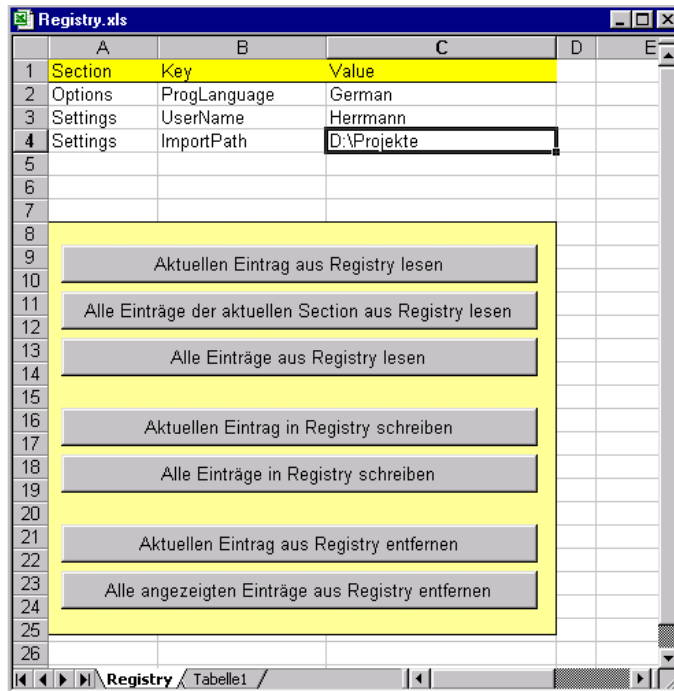


Abbildung 7.6: Tabelle Registry in Registry.xls

Abbildung 7.6 gibt auch die Struktur unserer Registry-Einträge wieder, die sich in die folgenden frei belegbaren, alphanumerischen Begriffe unterteilt. Einzig *AppName* taucht nicht in der Abbildung auf, da es sich hierbei um eine Konstante handelt, die sich auf alle Werte unserer Anwendung bezieht. Sie ist im Code vereinbart.

- ▼ **AppName** enthält als erste Gliederung und dem Schlüssel »VB and VBA Program Settings« den Namen unseres Programmes.
- ▼ **Section** ist eine Untergliederung, in die wir unsere eigentlichen Werte thematisch gruppieren können.
- ▼ **Key** ist der Schlüssel, mit dem wir unseren zu speichernden Wert innerhalb von *Section* identifizieren.
- ▼ **Value** ist der alphanumerische Wert, den wir speichern wollen.

Abbildung 7.7 zeigt die Einträge der Section Settings unseres Beispiels.

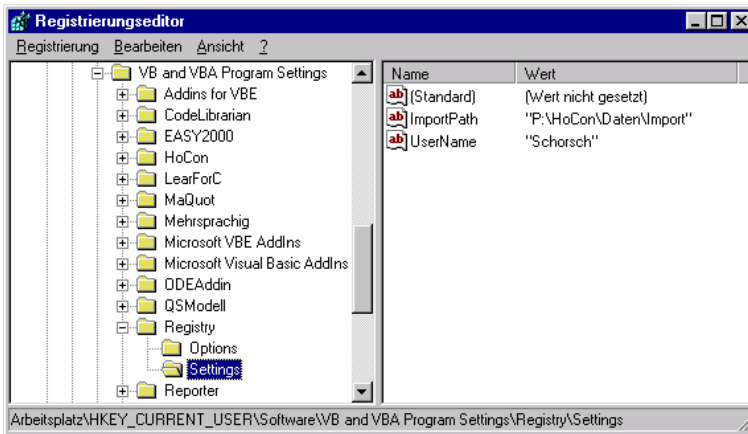


Abbildung 7.7: Keys und Values der Section Settings

Das in Registry.xls enthaltene Beispielprogramm zeigt den Einsatz der Sprachelemente, die uns im Zusammenhang mit der Registry zur Verfügung stehen.

Doch nun zu den beiden Funktionen und den Anweisungen, die uns auf recht einfache Weise Registry-Zugriffe ermöglichen.

7.4.2 Sprachelemente für Registry-Zugriffe

GetSetting-Funktion

Mit der GetSetting-Funktion können wir einen existierenden Eintrag auslesen. Existiert dieser nicht, erhalten wir einen NullString oder den Default-Wert, sofern dieser angegeben wurde.

GetSetting(AppName, Section, Key[, Default])

<i>AppName</i>	Name unseres Programmes
<i>Section</i>	Gruppierungsmöglichkeit der Einträge
<i>Key</i>	Name des Eintrags
<i>Default</i>	(optional) Der Wert, der zurückgegeben werden soll, wenn der Schlüssel nicht existiert.

Im folgenden Beispiel wird der Wert des in der Section »Settings« des Programmes »X« befindliche Key »ImportPath« der Variablen `strImportPath` zugewiesen. Für den Fall, dass dieser Schlüssel nicht existiert, soll das Verzeichnis Import hinter dem aktuellen Workbook-Verzeichnis zurückgegeben werden. Existiert das Verzeichnis nicht, so wird es kurzerhand angelegt.

```

strImportPath = GetSetting("X", "Settings", "ImportPath", _
                        ThisWorkbook.Path & "\Import")
If Dir(strImportPath, vbDirectory) = "" Then
    Mkdir strImportPath
End If

```

GetAllSettings-Funktion

Mit der GetAllSettings-Funktion können wir uns alle Einträge einer Section in einem String-Array zurückgeben lassen. Existiert kein Eintrag in der angegebenen Section, wird Empty zurückgegeben.

GetAllSettings(*AppName*, *Section*)

<i>AppName</i>	Name unseres Programmes
<i>Section</i>	Gruppierungsmöglichkeit der Einträge

Im nun folgenden Beispiel werden alle Keys und Values der Section »Settings« des Programmes »X« in das String-Array strSettings() eingelesen:

```

Dim strSettings() As String
strSettings = GetAllSettings("X", "Settings")

```

Ein ausführlicheres Beispiel wird weiter unten anhand des Codes der Registry.xls vorgestellt.

SaveSetting-Anweisung

SaveSetting schreibt einen Wert unter dem angegebenen Key, der Section und dem Programmnamen in die Registry. Existiert der Eintrag noch nicht, so wird er angelegt.

SaveSetting *AppName*, *Section*, *Key*, *Value*

<i>AppName</i>	Name unseres Programmes
<i>Section</i>	Gruppierungsmöglichkeit der Einträge
<i>Key</i>	Name des Eintrags
<i>Value</i>	einzutragender Wert

Das nächste Beispiel schreibt das hinter dem Workbook-Verzeichnis stehende Import-Verzeichnis unter dem Key ImportPath und der Section Settings des Programmes »X« in die Registry:

```
SaveSetting "X", "Settings", "ImportPath", _
    ThisWorkbook.Path & "\Import"
```

DeleteSetting-Anweisung

Hiermit kann ein Eintrag, eine Section oder ein kompletter Programmeintrag gelöscht werden. Existiert der entsprechende Eintrag nicht (mehr), so wird der Fehler 5 – Ungültiger Prozeduraufruf oder ungültiges Argument erzeugt.

DeleteSetting *AppName[, Section[, Key]]*

<i>AppName</i>	Name unseres Programmes
<i>Section</i>	(optional) Section Wird Section nicht angegeben, so wird der Programmeintrag nebst aller Sections und Einträge gelöscht. Der Hinweis, dass Section optional ist, ging in der Hilfe irgendwie verloren.
<i>Key</i>	(optional) Name des Eintrags Wird Key nicht angegeben, werden alle Einträge der Section gelöscht.

Die erste der folgenden Beispielzeilen löscht den Key ImportPath der Section Settings unserer Anwendung »X«. In der zweiten Zeile wird die ganze Section Options des Programmes X gelöscht und in Zeile drei der komplette Programmeintrag »X«:

```
DeleteSetting "X", "Settings", "ImportPath"
DeleteSetting "X", "Options"
DeleteSetting "X"
```

7.4.3 Das Beispiel Registry.xls

Die auf der Buch-CD befindliche Beispieldatei Registry.xls enthält folgende Funktionalitäten (siehe auch Abbildung 7.6):

- ▼ Aktuellen Eintrag aus Registry lesen
- ▼ Alle Einträge der aktuellen Section aus Registry lesen
- ▼ Alle Einträge aus Registry lesen
- ▼ Aktuellen Eintrag in Registry schreiben
- ▼ Alle Einträge in Registry schreiben
- ▼ Aktuellen Eintrag aus Registry entfernen
- ▼ Alle angezeigten Einträge aus Registry entfernen

Lediglich zur Realisierung der Prozedur, die alle Einträge aus der Registry liest, wurde ein wenig getrickst, denn eine Methode namens `EnumVBSections` der enthaltenen Klasse `clsRegistry` bedient sich der Unterstützung des Windows-API, um die vorhandenen `Sections` zu ermitteln. Der Rest aber ist mit Bordmitteln realisiert.

Alle Routinen enthalten Fehlerbehandlungsroutinen, die wir jedoch aus der nachfolgenden Betrachtung weitgehend ausklammern. Wenn Sie jedoch das Kapitel 9 – Fehlerbehandlung durchgearbeitet haben, können Sie sich die Prozeduren unter diesem Aspekt noch einmal zu Gemüte führen.

Bevor wir uns den Code jedoch im Detail ansehen, sollten wir einen Blick auf die Variablen und Konstanten dieses Moduls werfen:

```
Option Explicit
Private rngReg As Range      'Range-Objekt in Zelle A1
Private iRow As Long        'Zeilenzeiger in shtReg
Private strSection As String 'Section des Eintrags
Private strKey As String     'Key des Eintrags
Private strValue As String   'Wert des Eintrags
'Spaltenzeiger
Private Const iColSection As Long = 1
Private Const iColKey As Long = 2
Private Const iColValue As Long = 3
'Applikationsname
Private Const strAppName As String = "Registry"
```

Übergreifende Routinen

Die einzelnen Registry-Routinen verwenden ein `Range`-Objekt zur Bestimmung der `CurrentRegion` der Tabelle, um einem Zeilenumbruch in den Listings aus dem Wege zu gehen, den uns das `Worksheet`-Objekt eingehandelt hätte:

```
Private Sub Worksheet_Activate()
    'Wert der aktuellen Zeile aus Registry lesen
    On Error GoTo errHandler
    Set rngReg = shtReg.Range("A1")
    Exit Sub

errHandler:
    Err.Source = "shtReg:Worksheet_Activate"
    cError.ErrorMessage
End Sub
```

Drei Prozeduren benötigen die Zeilennummer, in der sich der Cursor derzeit befindet, da sie den aktuellen Eintrag bearbeiten. Hierzu sollte aber sichergestellt sein, dass sich dieser in einer gültigen Zeile befindet. Und genau das hat die folgende Funktion als Aufgabe:

Private Function GetValidatedRow() As Long

```
'Überprüfen, ob Cursor im aktuellen Zeilenbereich steht
On Error GoTo errHandler
iRow = ActiveWindow.ActiveCell.Row
If iRow > rngReg.CurrentRegion.Rows.Count Then
    Err.Raise 9999, , "Bitte setzen Sie den Cursor " & _
        "in eine Zelle innerhalb der Settings."
End If
GetValidatedRow = iRow
Exit Function
```

errHandler:

```
Err.Raise Err.Number, "shtReg:GetValidatedRow"
```

End Function

Befindet sich der Cursor außerhalb der *CurrentRegion* des *Range*-Objekts *rngReg*, wird der Fehler 9999 erzeugt, der in den aufrufenden Routinen entsprechend behandelt werden muss.

Aktuellen Eintrag aus Registry lesen

Hierzu wird zuerst die aktuelle Zeile über *GetValidatedRow* ermittelt. Dieser Zeiger wird anschließend verwendet, um die *Section* und den *Key* zu ermitteln und die Rückgabe aus der Registry in die Zielzelle zu schreiben.

Private Sub cmdGetCurrentSetting_Click()

```
'Wert der aktuellen Zeile aus Registry lesen
On Error GoTo errHandler
iRow = GetValidatedRow()
strSection = rngReg.Cells(iRow, iColSection).Value
strKey = rngReg.Cells(iRow, iColKey).Value
strValue = GetSetting(strAppName, strSection, strKey)
shtReg.Cells(iRow, iColValue).Value = strValue
Exit Sub
```

errHandler:

```
If Err.Number = 9999 Then
    cError.InfoMessage Err.Description
    Exit Sub
End If
```

```

If InStr(1, Err.Source, ":") = 0 Then
    Err.Source = "shtReg:cmdGetCurrentSetting_Click"
End If
cError.ErrorMessage
End Sub

```

Alle Einträge der aktuellen Section aus Registry lesen

Auch hier wird zuerst der Zeilenzeiger `iRow` durch `GetValidatedRow` ermittelt. Anschließend wird die aktuelle Section ermittelt und die Schleife rückwärts durchlaufen, um alle Zeilen zu löschen, die diese Section aufweisen. Diese Schleife muss rückwärts durchlaufen werden, da uns sonst Zeilen durch die Lappen gehen würden. Danach wird der Zeilenzeiger auf die nächste freie Zeile positioniert und das String-Array eingelesen, das uns die Einträge dieser Section zurückgeben soll. Da die Rückgabe jedoch Empty sein kann, wenn kein Eintrag unter der angeforderten Section enthalten ist, muss `strStettings` als Variant deklariert sein. Ist dies der Fall, wird eine `InfoMessage` ausgegeben und die Prozedur verlassen.

Danach wird dieses Array in die Tabelle eingetragen. Erwähnenswert ist vielleicht noch, dass der Zeilenzeiger über `iRow` plus den Settings-Zeiger in `strSettings` gebildet wird.

```

Private Sub cmdGetCurrentSectionSettings_Click()
    'Alle Keys der aktuellen Section aus Registry lesen
    Dim strSettings As Variant          'Alle Keys und Werte
                                        'einer Section
    Dim iSetting As Long                'Zeiger in strSettings
    On Error GoTo errHandler
    iRow = GetValidatedRow()
    strSection = shtReg.Cells(iRow, iColSection).Value
    For iRow = rngReg.CurrentRegion.Rows.Count To 2 Step -1
        If shtReg.Cells(iRow, iColSection).Value = _
            strSection Then
            shtReg.Rows(iRow).Delete
        End If
    Next
    iRow = shtReg.Range("A1").CurrentRegion.Rows.Count + 1
    strSettings = GetAllSettings(strAppName, strSection)
    If IsEmpty(strSettings) Then
        cError.InfoMessage "In Section " & strSection & _
            " existiert keine Eintrag in der Registry."
    Exit Sub
End If

```

```

For iSetting = 0 To UBound(strSettings, 1)
    shtReg.Cells(iRow + iSetting, iColSection).Value = _
        strSection
    shtReg.Cells(iRow + iSetting, iColKey).Value = _
        strSettings(iSetting, 0)
    shtReg.Cells(iRow + iSetting, iColValue).Value = _
        strSettings(iSetting, 1)
Next
Exit Sub

errHandler:
    If Err.Number = 9999 Then
        cError.InfoMessage Err.Description
        Exit Sub
    End If
    If InStr(1, Err.Source, ":") = 0 Then Err.Source = _
        "shtReg:cmdGetAllCurrentSectionSettings_Click"
    cError.ErrorMessage
End Sub

```

Alle Einträge aus Registry lesen

Gegenüber der voranstehenden Prozedur ist hier eigentlich nichts Neues zu beobachten. Die Methode `cReg.EnumVBSections` der beiliegenden Klasse `clsRegistry` betrachten Sie bitte als Bonbon; wir wollen hier nicht näher darauf eingehen.

```

Private Sub cmdGetAllSettings_Click()
    'Alle Sections und Keys aus Registry lesen
    Dim cReg As clsRegistry      'Instanz auf clsRegistry
    Dim strSections() As String 'Alle Sections des
Programms
    Dim strSettings() As String 'Alle Keys und Werte einer
                                'Section
    Dim iSection As Long        'Zeiger in strSections()
    Dim iSetting As Long        'Zeiger in strSettings()
    On Error GoTo errHandler
    Set cReg = New clsRegistry
    strSections = cReg.EnumVBSections(strAppName)
    rngReg.CurrentRegion.Offset(1, 0).ClearContents
    For iSection = 0 To UBound(strSections)
        iRow = rngReg.CurrentRegion.Rows.Count + 1
        strSettings = GetAllSettings(strAppName, _
            strSections(iSection))
        For iSetting = 0 To UBound(strSettings, 1)

```

```

        shtReg.Cells(iRow + iSetting, iColSection).Value _
            = strSections(iSection)
        shtReg.Cells(iRow + iSetting, iColKey).Value = _
            strSettings(iSetting, 0)
        shtReg.Cells(iRow + iSetting, iColValue).Value = _
            strSettings(iSetting, 1)

    Next
Next
Exit Sub

errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "shtReg:cmdGetAllSettings_Click"
    End If
    cError.ErrorMessage
End Sub

```

Aktuellen Eintrag in Registry schreiben

Auch diese Routine ist reichlich unspektakulär. Nachdem der Zeilenzeiger `iRow` ermittelt ist, werden `Section`, `Key` und `Value` ausgelesen und per `SaveSetting` an die Registry geschickt.

```

Private Sub cmdSaveCurrentSetting_Click()
    'Den Wert des aktuellen Eintrags in Registry schreiben
    On Error GoTo errHandler
    iRow = GetValidatedRow()
    strSection = rngReg.Cells(iRow, iColSection).Value
    strKey = rngReg.Cells(iRow, iColKey).Value
    strValue = shtReg.Cells(iRow, iColValue).Value
    SaveSetting strAppName, strSection, strKey, strValue
Exit Sub

errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "shtReg:cmdSaveCurrentSetting_Click"
    End If
    cError.ErrorMessage
End Sub

```


Alle Einträge in Registry schreiben

Im Vergleich zur voranstehenden Prozedur wird hier kein Zeilenzeiger benötigt. Stattdessen wird der Zellbereich zeilenweise abgearbeitet und die Werte werden an die Registry übergeben.

```
Private Sub cmdSaveAllSettings_Click()
    'Alle Einträge in Registry speichern
    On Error GoTo errHandler
    For iRow = 2 To rngReg.CurrentRegion.Rows.Count
        strSection = rngReg.Cells(iRow, iColSection).Value
        strKey = rngReg.Cells(iRow, iColKey).Value
        strValue = shtReg.Cells(iRow, iColValue).Value
        SaveSetting strAppName, strSection, strKey, strValue
    Next
    Exit Sub

errHandler:
    Err.Source = "shtReg:cmdSaveAllSettings_Click"
    cError.ErrorMessage
End Sub
```

Aktuellen Eintrag aus Registry entfernen

Auch hier wird die GetValidatedRow bemüht, um den Zeilenzeiger zu ermitteln. Anschließend werden Section und Key ermittelt und an die Anweisung DeleteSetting verfüttert.

Im ErrorHandler wird der Fehler noch abgefangen, den die DeleteSetting dann produziert, wenn der Schlüssel nicht mehr in der Registry vorhanden ist.

```
Private Sub cmdDeleteCurrentSetting_Click()
    'Aktuellen Eintrag aus Registry löschen
    On Error GoTo errHandler
    iRow = GetValidatedRow()
    strSection = rngReg.Cells(iRow, iColSection).Value
    strKey = rngReg.Cells(iRow, iColKey).Value
    DeleteSetting strAppName, strSection, strKey
    Exit Sub

errHandler:
    If Err.Number = 5 Then
        cError.InfoMessage "Der Key " & strKey & _
            " in Section " & strSection & " konnte " & _
            "nicht gelöscht werden.", "Vermutlich " & _
            "existiert er nicht mehr."
```

```

        Exit Sub
    End If
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "shtReg:cmdDeleteCurrentSetting_Click"
    End If
    cError.ErrorMessage
End Sub

```

Alle angezeigten Einträge aus Registry entfernen

Hier sind Sie gefordert. Wenn Sie wollen, können Sie diese Routine selbst codieren. Die Lösung finden Sie im letzten Kapitel.

7.5 Runden

Eigentlich könnte man das Runden von Werten in Kapitel 4 erwarten, denn immerhin gibt es ja eine Funktion des Namens Round:

Round(*Ausdruck* [, *AnzahlAnDezimalpunktn*])

<i>Ausdruck</i>	der zu rundende Ausdruck
<i>AnzahlAnDezimalpunktn</i>	(optional) Die Stelle rechts des Kommas, auf die <i>Ausdruck</i> gerundet werden soll.

Schauen wir uns einmal die Ergebnisse dieser Funktion etwas näher an:

```

Round(12345.123456)    ergibt 12345
Round(12345.123456, 0) ergibt 12345
Round(12345.123456, 1) ergibt 12345,1
Round(12345.123456, 2) ergibt 12345,12
Round(12345.123456, 3) ergibt 12345,123
Round(12345.123456, 4) ergibt 12345,1235

```

Soweit, so gut, doch

```
Round(12345.123450, 4) ergibt 12345,1234
```

Hier kommt doch ein leichter Zweifel auf. Was soll's, wir sind ja in Excel-VBA, also verwenden wir das Äquivalent der Tabellenfunktion RUNDEN(), die uns als Methode des Application-Objekt zur Verfügung steht:

```
Application.Round(12345.123450,4) ergibt 12345,1235
```

Abgesehen von dieser Ungereimtheit, für die es sicherlich eine Erklärung gibt, stört mich an der Round-Funktion, dass sie keinerlei Rundung an

Stellen größer 10^0 ermöglicht, also etwa an der Zehner- oder Hunderterstelle.

Solange wir in Excel-VBA sind, könnten wir ja auf die Round-Methode zugreifen. Doch was machen wir, wenn wir in Word-VBA arbeiten oder in Visual Basic? Eine Instanz von Excel erzeugen, diese einen Wert runden lassen und dann Excel wieder schließen?

Wir sind seit frühester Jugend daran gewöhnt, die zweite Stelle nach dem Komma als 10 hoch minus 2 und die zweite vor dem Komma als 10 hoch plus 2 zu bezeichnen. Und weshalb wurde statt dieser gebräuchlichen Notation einfach das Vorzeichen vertauscht?

Eine gewisse Zeit habe ich die Round-Methode des Application-Objekts durchaus verwendet, bis ich mal wieder, ohne mich der besonderen Notation zu erinnern, das falsche Vorzeichen übergab. Seitdem tut eine kleine Funktion namens RoundValue ihren Dienst:

```
Public Function RoundValue(ByVal varValue As Variant, _
                          lngExponent As Long) As Variant

'=====
'   Rückgabe des exponential gerundeten Wertes
'-----
'Argumente:
'   varValue:      zu rundender Wert
'   lngExponent:   zu rundende Stelle als Zehnerexponent

'=====
Dim dblDivisor As Double 'Divisor aus lngExponent
On Error GoTo errHandler
'Fehler erzeugen, wenn varValue nicht numerisch ist
If Not IsNumeric(varValue) Then
    Err.Raise 13, , "Der übergebene Wert " & varValue & _
        " ist keine Zahl."
End If
dblDivisor = 10 ^ lngExponent
RoundValue = Round(varValue / dblDivisor) * dblDivisor
Exit Function

errHandler:
    Err.Raise Err.Number, "clsUtil:RoundValue"
End Function
```

Diese Funktion wird nun mit der korrekten, exponentiellen Notation aufgerufen:

```
RoundValue(12345.123456, 2)  ergibt 12300
```

```
RoundValue(12345.123456, -2) ergibt 12345,12
```

Sie steckt in einem eigenen Modul (eigentlich eine Klasse) zusammen mit anderen nützlichen Funktionen. Und ein neues Projekt beginne ich generell damit, zuerst einmal eine Reihe von Modulen einzufügen, darunter natürlich auch dieses Modul.

7.6 Farben

Farben ziehen sich quer durch Excel und VBA. In Excel können wir Zellen anmalen, Schriftfarben verändern und beliebige Objekte einfärben. Als VBA-Komponenten bieten sich einige Dialogelemente an, die wir farbig gestalten können.

7.6.1 Sprachelemente

Bei Farben sind im Wesentlichen die beiden Eigenschaften `ColorIndex` und `Color` im Spiel.

ColorIndex-Eigenschaft

Hier können wir über die Werte 1 bis 56 auf vordefinierte Farben zurückgreifen. Diese Farben können im Register Farben des Options-Dialogs von Excel einzeln verändert werden. Außerdem stehen uns die beiden folgenden Konstanten zur Verfügung:

- ▼ 4105 - `xlColorIndexAutomatic` für die Schriftfarbe des Bildelements Fenster der Systemsteuerung.
- ▼ 4142 - `xlColorIndexNone` für die Hintergrundfarbe des Bildelements Fenster der Systemsteuerung.

Color-Eigenschaft

Hinter der Color-Eigenschaft verbirgt sich ein `Long`-Wert, der die Farbkomponenten Rot, Grün und Blau in einem so genannten RGB-Wert zusammenfasst. Abbildung 7.8 zeigt den Aufbau einer solchen RGB-Farbe.

Aus dieser Abbildung ergibt sich der Wertebereich einer RGB-Farbe, der zwischen 0 und `&HFFFFFF` oder dezimal 16.777.215 liegen muss. Daraus ergibt sich auch die maximale Farbtiefe von 24 Bit.

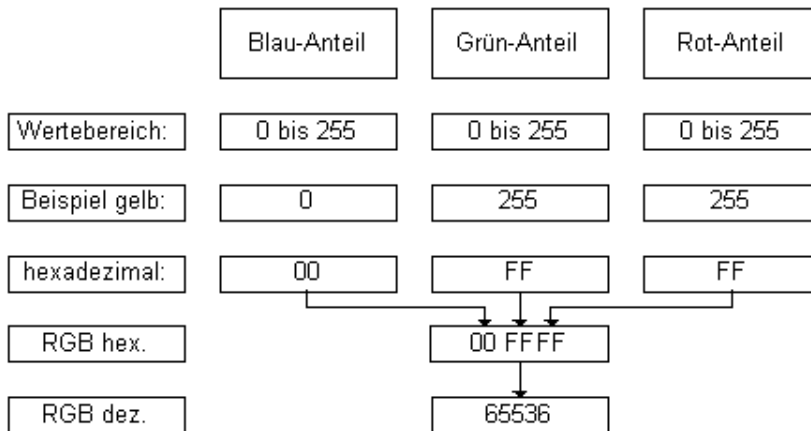


Abbildung 7.8: Farben, Aufbau einer RGB-Farbe

RGB-Funktion

Die RGB-Funktion baut uns aus den drei Komponenten Rot, Grün und Blau den dezimalen Farbwert zusammen:

```
Range.Interior.Color = RGB(255, 255, 0) 'ergibt 65536
```

Farbkomponente aus RGB-Farbe ermitteln

Hier bietet uns VBA keine Funktion an, weshalb wir uns selbst behelfen müssen.

Unter Excel 2000 würde ich eine Funktion verwenden, in der die Entscheidung, welche Farbkomponente nun extrahiert werden soll, über eine Enumeration fällt:



```
Public Enum enumRGBColor
    enumRGBColorRed = 0
    enumRGBColorGreen = 1
    enumRGBColorBlue = 2
End Enum
```

In der Zuweisung der Werte zu den einzelnen Konstanten liegt auch eine tiefere Bedeutung, wie wir noch sehen werden. Die eigentliche Funktion sieht so aus:

```
Public Function GetRGBColor(ByVal RGBColor As Long, _
    ByVal lngColor As enumRGBColor) As Long
    'Ermittelt den Farb-Anteil des übergebenen Farbwerts
    On Error GoTo errHandler
    If RGBColor < 0 Or RGBColor > 16777215 Then
```

```

        Err.Raise 6, , "Der übergebene RGB-Farbwert ist " & _
            "außerhalb des gültigen Bereichs."
    Else
        GetRGBColor = RGBColor \ (256 ^ lngColor) And 255
    End If
    Exit Function

errHandler:
    Err.Raise Err.Number, "modColor:GetRGBColor"
End Function

```

Das einzig Spannende dieser Funktion ist die hervorgehobene Zeile, in der zweierlei passiert. Zum einen wird aus dem übergebenen Farbwert mit einer ganzzahligen Division durch 1 bei Rot (&H1), 256 bei Grün (&H100) und 35536 (&H10000) bei Blau.

Die anschließende Maskierung mit 255 (&HFF) ergibt dann den gewünschten Farbanteil (siehe Abbildung 7.9).



Da wir in Excel 97 keine Enumerationen anwenden können, schlage ich vor, aus dieser einen, universellen Funktion drei angepasste Versionen für die einzelnen Farbkomponenten zu erstellen, die sich nur im Divisor unterscheiden. Mit Enumeration meldet sich das Argument `lngColor` mit seinen Mitgliedern, was eine komfortable und sichere(!) Parametrierung ermöglicht. In Excel 97 müssten wir uns die Werte 0, 1 und 2 für Rot, Grün und Blau merken. Und das ist überflüssig und unsicher.

Diese drei Einzelroutinen finden Sie in der Datei `Farben.xls` auf der Buch-CD.

Ganzzahlige Division

Rot-Anteil:	<code>&H99CCFF</code>	<code>\</code>	<code>&H1</code>	<code>=</code>	<code>&H99CCFF</code>
Grün-Anteil:	<code>&H99CCFF</code>	<code>\</code>	<code>&H100</code>	<code>=</code>	<code>&H99CC</code>
Blau-Anteil:	<code>&H99CCFF</code>	<code>\</code>	<code>&H10000</code>	<code>=</code>	<code>&H99</code>

Maskierung mit &HFF

<code>&H99CCFF</code>	<code>And</code>	<code>&HFF</code>	<code>=</code>	<code>&HFF</code>	<code>=</code>	255
<code>&H99CC</code>	<code>And</code>	<code>&HFF</code>	<code>=</code>	<code>&HCC</code>	<code>=</code>	204
<code>&H99</code>	<code>And</code>	<code>&HFF</code>	<code>=</code>	<code>&H99</code>	<code>=</code>	153

Abbildung 7.9: Farben, Zerlegen einer RGB-Farbe

7.6.2 Ein Beispiel

Die Datei Farben.xls enthält eine Tabelle, in der einige Experimente mit Farben vorgenommen werden (siehe Abbildung 7.10).

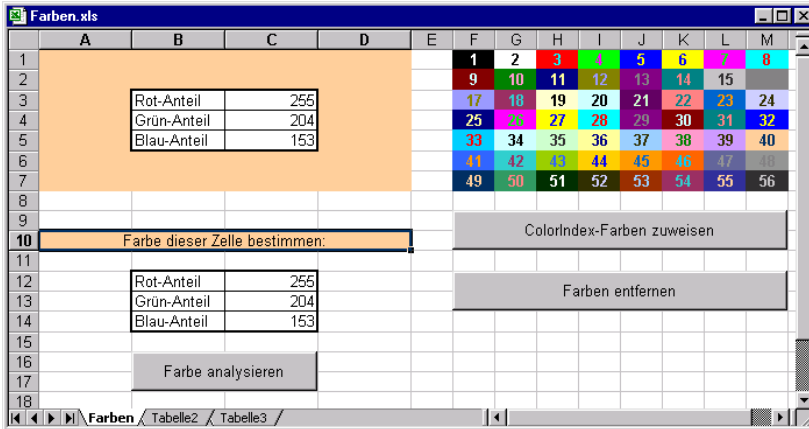


Abbildung 7.10: Farben.xls, Tabelle Farben

Farbzuweisung

In der oberen linken Ecke der Tabelle befinden sich drei Zellen, in denen die Farbanteile für den Bereich um diese Zellen herum angegeben sind. Die Codierung erfolgt im Change-Ereignis des Worksheet-Moduls:

```
Private Sub Worksheet_Change(ByVal Target As Range)
    Me.Range("A1:D7").Interior.Color = RGB( _
        Me.Range("C3").Value, _
        Me.Range("C4").Value, _
        Me.Range("C5").Value)
    Me.Range("B3:C5").Interior.ColorIndex = 2
End Sub
```

Farbanalyse

Die Farbkomponenten der Zelle A10 werden über die Ereignisroutine der Schaltfläche ermittelt:

```
Private Sub cmdAnalyzeColor_Click()
    Dim lngColor As Long 'Color-Eigenschaft der Zelle A10
    lngColor = Me.Range("A10").Interior.Color
    Me.Range("C12").Value = GetRGBColor(lngColor, _
        enumRGBColorRed)
    Me.Range("C13").Value = GetRGBColor(lngColor, _
        enumRGBColorGreen)
```

```
Me.Range("C14").Value = GetRGBColor(lngColor, _
                                enumRGBColorBlue)
```

```
End Sub
```

Die hier verwendete Funktion `GetRGBColor` wurde weiter oben bereits vorgestellt.

ColorIndex-Farben zuweisen

Im oberen rechten Tabellenbereich befinden sich 56 Zellen mit den Farbkonstanten 1 bis 56. Die Ereignisroutine der oberen Schaltfläche weist den Zellen die Farben zu, die dem jeweiligen `ColorIndex` entsprechen. Die Schriftfarbe wird als Komplementärfarbe zur Hintergrundfarbe ermittelt und zugewiesen:

```
Private Sub cmdAssignColorIndex_Click()
    Dim iRow As Long           'Zeilenzeiger
    Dim iCol As Long           'Spaltenzeiger
    Dim lngColor As Long       'RGB-Wert der Zelle
    Dim lngRC As Long          'Rot-Komplementär
    Dim lngGC As Long          'Grün-Komplementär
    Dim lngBC As Long          'Blau-Komplementär
    For iRow = 1 To 7
        For iCol = 6 To 13
            Me.Cells(iRow, iCol).Interior.ColorIndex = _
                Me.Cells(iRow, iCol).Value
            lngColor = Me.Cells(iRow, iCol).Interior.Color
            lngRC = 255 - GetRGBColor(lngColor, _
                                    enumRGBColorRed)
            lngGC = 255 - GetRGBColor(lngColor, _
                                    enumRGBColorGreen)
            lngBC = 255 - GetRGBColor(lngColor, _
                                    enumRGBColorBlue)
            Me.Cells(iRow, iCol).Font.Color = _
                RGB(lngRC, lngGC, lngBC)
        Next
    Next
End Sub
```

Die Schaltfläche **Farben entfernen** setzt die Farbzueweisung wieder zurück:

```
Private Sub cmdResetColorIndex_Click()
    Me.Range("F1:M7").Interior.ColorIndex = _
        xlColorIndexNone
    Me.Range("F1:M7").Font.ColorIndex = _
        xlColorIndexAutomatic
End Sub
```


7.7 API

API steht für Application Programming Interface und stellt die Schnittstelle zu Windows dar. Das API gibt uns Zugang zu elementaren Windows-Funktionen, wobei die Mechanismen von VBA und der Entwicklungsumgebung umgangen werden. Das ist das Wohl und Wehe des API, denn die Entwicklungsumgebung kapselt unseren Code und tut eine ganze Menge im Hintergrund, erstellt Referenzen, löst sie auf und fängt Fehler für uns ab. Die Entwicklungsumgebung ist unser Panzer, der uns vor Windows und seinen Gemeinheiten schützt. Wir schützen unsere Kinder auch, indem wir Gegenstände, die eine Gefahr für das Kind darstellen können, außer Reichweite aufbewahren. Und das API ist gewissermaßen eine Art Klappleiter für Kinder, mit der sie dann überall hinkommen und unsere gut gemeinten Warnungen mit ihren kleinen Füßen treten können.

Und wer ist so unvernünftig und gibt uns VBA-Entwicklern diese Klappleiter in die Hand? Microsoft natürlich, denn VBA unterstützt alle erforderlichen Mechanismen, um in die Welt des API einzudringen. So, als würden wir sagen: »Du darfst da nicht hin, mein Kleiner. Aber da hinten steht die Leiter...«

API und Appleman werden üblicherweise in der Welt von VB und VBA in einem Atemzug genannt. Gemeint ist damit Dan Appleman, ein höchst kompetenter Entwickler, Autor und Redner, der jährlich in Deutschland auf dem VB-Event BASTA life und in Farbe zu bewundern ist. Diese BASTA wird vom Steingraber Verlag, dem Herausgeber der BasicPro, veranstaltet.

Appleman schrieb unter anderem die beiden Standardwerke:

- ▼ Visual Basic x Programmer's Guide to the WIN32 API, ein 1500 Seiten starkes Werk über das Windows API, das leider nicht übersetzt wurde
- ▼ COM/Activex-Komponenten mit Visual Basic 6 entwickeln, ein deutschsprachiges, höchst erbauliches Buch, dessen Titel ein wenig in die falsche Richtung zu weisen scheint. Aber in dem Moment, in dem wir ein Klassenmodul erzeugen, bauen wir eine ActiveX-Komponente, die auf der COM-Technologie basiert. Das Aha-Buch schlechthin.

Nach dieser kurzen Einleitung wollen wir uns den WinAPI-Viewer etwas näher ansehen und uns danach einigen sinnvollen Funktionen widmen, die Ihre Applikationen bereichern werden.

7.7.1 Der WinAPI-Viewer

Wenn Sie im Menü Add-Ins den WinAPI-Viewer nicht finden, müssen Sie ihn über den Add-In-Manager der Entwicklungsumgebung zuerst aktivieren. Danach präsentiert er sich, wie in Abbildung 7.11 zu sehen ist. Zuerst

müssen Sie jedoch eine Textdatei laden. Wählen Sie hierzu im Menü Datei den Eintrag Textdatei laden ... und wählen Sie dort bitte die Win32API.txt.

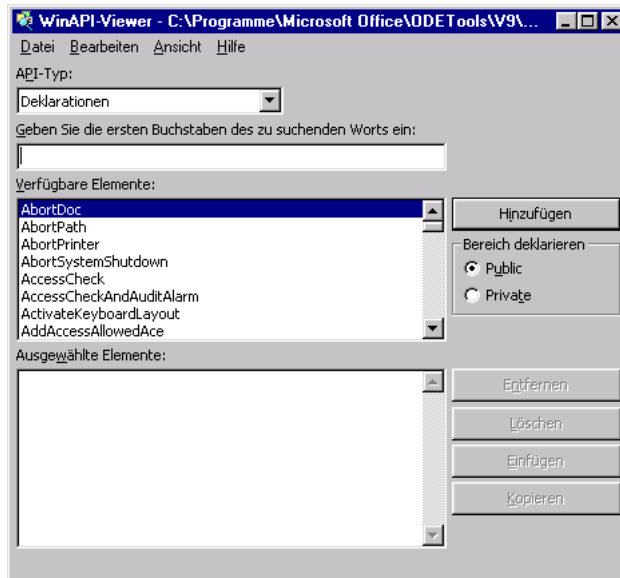


Abbildung 7.11: inAPI-Viewer mit geladener Win32API.txt

Sie können in der TextBox die Anfangsbuchstaben des gewünschten Eintrags eingeben, und die ListBox wird zu dem ersten passenden Element springen. Durch die Hinzufügen-Schaltfläche oder einen Doppelklick befördern Sie den Eintrag in die untere TextBox. Dort können Sie den Cursor in eine der Zeilen positionieren und den Eintrag über die Schaltfläche Kopieren in die Zwischenablage transportieren, von wo aus er über Einfügen [Strg]-[V] in den Code gelangt. Üben Sie ein wenig, und Sie werden das Handling schnell beherrschen. Abbildung 7.12 zeigt die GetUserName im Codefenster.

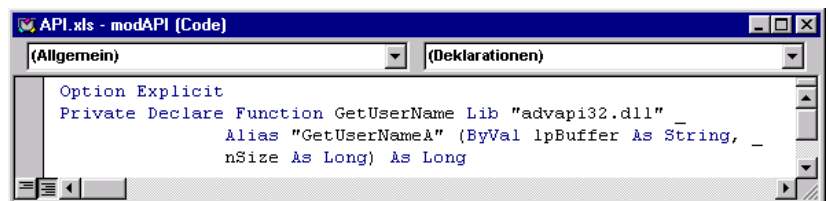


Abbildung 7.12: GetUserName im Codefenster

Deklarationen und auch Konstanten dürfen nur im Modulkopf eingefügt werden; andernfalls werden Sie einen entsprechenden Hinweis des Editors erhalten.

7.7.2 Struktur einer API-Deklaration

Folgende Elemente können in einer Deklaration auftauchen:

- ▼ `Private` | `Public` gibt den Gültigkeitsbereich der Deklaration an.
- ▼ `Declare` kennzeichnet die Deklaration eines Aufrufs in einer DLL.
- ▼ `Sub` | `Function` sagt aus, ob es sich um eine Prozedur oder eine Funktion mit Rückgabe handelt. Hinter `Sub` oder `Function` folgt der Name, unter dem wir die Prozedur im Code verwenden können.
- ▼ `Lib` leitet den Namen der DLL ein.
- ▼ `Alias` leitet den Namen ein, unter dem die Prozedur in der DLL steht. Dieses Argument ist nicht immer verfügbar und interessiert uns im Grunde auch nicht.
- ▼ Die Argumentliste enthält alle an die Prozedur zu übergebenden Variablen in einer Syntax, wie wir sie aus unseren Argumentlisten auch kennen.
- ▼ Bei Funktionen erfolgt eine Angabe, welchen Datentyps die Rückgabe ist. In der Regel handelt es sich um einen `Long`-Wert.

Schauen wir uns nun einige der Funktionen aus der Nähe an. Da wir sie bereits in das Modul kopierten, wollen wir mit der `GetUserName` beginnen.

7.7.3 Anwendername ermitteln

```
Private Declare Function GetUserName Lib "advapi32.dll" _
    Alias "GetUserNameA" ( _
        ByVal lpBuffer As String, _
        nSize As Long) As Long
```

Diese Funktion gibt uns den Anmeldenamen des aktuellen Anwenders zurück. Mit `lpBuffer` und `nSize` verfügt sie über zwei Argumente, die oft unter diesen oder ähnlichen Namen paarweise auftreten. Die erste der beiden muss eine Zeichenkette in ausreichender Länge sein, damit die aufgerufene Funktion das Ergebnis hineinschreiben kann. Das zweite Argument muss eine Zahl sein, die mindestens so groß ist wie die Anzahl der Zeichen in der ersten Variablen, also die Anzahl der erwarteten Zeichen.

Dies alles mutet etwas vorsintflutlich an, ist aber einfach so. Schauen wir uns das Ganze anhand eines Beispiels an:

```
Private Sub LoginName()
    Dim strName As String 'Variable für Anmeldenamen
    Dim nSize As Long     'Variable für Länge des Strings
    Dim lngResult As Long 'Dummy für Rückgabe
    nSize = 100
    strName = Space(100) 'mit 100 Blanks vorbelegen
    lngResult = GetUserName(strName, nSize)
    MsgBox Left(strName, nSize)
End Sub
```

Die beiden Argumente werden von der Funktion, mit Werten gefüllt, wieder zurückgegeben. Für den Anmeldenamen *Administrator* werden die ersten 14 Blanks durch den Namen *Administrator* und ein nachfolgendes Chr(0) ersetzt, und nSize beinhaltet mit 13 die Nettobreite der Rückgabe. Deswegen kann mit

```
MsgBox Left(strName, nSize)
```

der Anmeldename aus der Rückgabe herausgeschnitten werden.

Eine wesentliche Erkenntnis steckt noch in diesen Aufrufen, denn sie transportieren Argumente teils in beide Richtungen. Diesen Mechanismus kennen wir nur bei als Referenz übergebenen Variablen.

7.7.4 Pause im Programmablauf

```
Declare Sub Sleep Lib "kernel32" Alias "Sleep" ( _
    ByVal dwMilliseconds As Long)
```

Gelegentlich ist es erforderlich, im Programmablauf eine kleine Pause einzulegen, ohne dass deren Ende durch eine Anwenderaktion eingeleitet werden muss.

Eine ebenso gebräuchliche wie schlechte Vorgehensweise ist die For-Next-Schleife, die in aller Stille von 1 bis 100.000 zählt. Wenn wir schon eine Pause einlegen, so können wir diese Zeitscheibe auch Windows zur Verfügung stellen:

```
Private Sub PauseProgram()
    '...
    Beep
    Sleep 1000
    Beep
```

```
' ...
End Sub
```

Diese Routine legt zwischen den beiden Beeps eine Pause von 1000 Millisekunden ein.

7.7.5 Laufwerkstyp ermitteln

```
Declare Function GetDriveType Lib "kernel32" _
    Alias "GetDriveTypeA" ( _
        ByVal nDrive As String) As Long
```

Die `GetDriveType` offenbart uns das Ergebnis ihrer Auswertung in der Rückgabe, die folgende Bedeutung hat:

Wert	Bedeutung
0	Laufwerk kann nicht identifiziert werden
1	Laufwerk nicht vorhanden
2	Wechsellaufwerk (Diskette, ZIP)
3	festes Laufwerk (Festplatte)
4	Netzlaufwerk
5	CD-ROM-Laufwerk
6	RAM-Disk

`nDrive` erwartet das Root-Verzeichnis des Laufwerks, womit z. B. »C:« eigentlich nicht korrekt ist und eigentlich durch »C:\« ersetzt werden müsste. Interessanter Weise funktioniert »C:« auf meinen Rechnern, übersah aber bei einem Kunden (US-Versionen von NT und Office) die lokale Festplatte, wohingegen Disketten-, CD- und Netzlaufwerke erkannt wurden. Also, Backslash nicht vergessen.

Es ist sinnvoll, die Rückgabe dieser Funktion in einer Enumeration abzubilden:

```
'Enumeration für DriveType
Public Enum enumDriveType
    enumDriveNotIdentified = 0
    enumDriveNotExisting = 1
    enumDriveRemoveable = 2
    enumDriveFixed = 3
    enumDriveRemote = 4
    enumDriveCDRom = 5
    enumDriveRamDisk = 6
End Enum
```

In Kapitel 11 wird ein Beispiel namens Settings.xls verwendet, dem folgende Funktion entlehnt ist, die als Rückgabe obige Enumeration verwendet:

```
Public Function DriveType(ByVal strDriveLetter As String) _
    As enumDriveType
    'Rückgabe Laufwerkstyps
    On Error GoTo errHandler
    DriveType = GetDriveType(strDriveLetter)
    Exit Function

errHandler:
    Err.Raise Err.Number, "modUtil:DriveType"
End Function
```

Das folgende Beispiel zeigt die Verwendung der Funktion DriveType in Auszügen. In dem Beispiel wird eine ComboBox mit den verfügbaren Laufwerken nebst Laufwerkstyp geladen.

Im Original auf der Buch-CD wird auch noch der Zustand des Laufwerks durch Testzugriff untersucht, was wir jedoch hier aus Gründen der Übersichtlichkeit weglassen. Sie können dieses Beispiel jedoch auf CD näher untersuchen:

```
Private Sub cmbDrive_Load()
    'Laden der existierenden Laufwerke mit Typ und Status
    Dim iDrive As Long 'Zeiger in ANSI-Code-Großbuchstaben
    Dim strDriveType As String 'Laufwerktyp im Klartext
    Dim lngDriveType As enumDriveType 'Laufwerktyp als
    Enumeration

    cmbDrive.Clear
    'ANSI-Großbuchstaben durchlaufen
    For iDrive = 65 To 90
        'Laufwerkstyp ermitteln und Texte erzeugen
        lngDriveType = DriveType(Chr(iDrive) & ":\")
        Select Case lngDriveType
            Case enumDriveRemoveable
                strDriveType = "[Wechseldatenträger]"
            Case enumDriveFixed
                strDriveType = "[Festplatte]"
            Case enumDriveRemote
                strDriveType = "[Netzlaufwerk]"
            Case enumDriveCDRom
                strDriveType = "[CD-ROM]"
        End Select
    Next iDrive
End Sub
```

```

Case enumDriveRamDisk
    strDriveType = "[RAM-Disk]"
End Select
'...
cmbDrive.AddItem "(" & Chr(iDrive) & ":)" & _
                strDriveType
'...
Next
'...
End Sub

```

Es existiert zwar noch eine API-Funktion, mit der man die existierenden Laufwerke ermitteln und somit unnötige Zugriffe vermeiden kann, doch das Durchprobieren mit den Großbuchstaben bereitet keine Probleme.

7.7.6 Ein Timer

In Visual Basic steht uns ein Timer-Steuerelement zur Verfügung, das nach einem in Millisekunden anzugebenden Intervall ein Ereignis auslöst. Wenn Sie über dieses Steuerelement verfügen, können Sie es in Ihrer Applikation einsetzen. Aber es muss auf allen Rechnern installiert sein, auf denen Ihre Applikation laufen soll. Und das ist eine Hürde, die ich in VBA-Programmen nicht zu nehmen bereit bin. Aber das API stellt uns zwei Funktionen zur Verfügung, mit denen wir einen Timer bauen können.

```

Declare Function SetTimer Lib "user32" Alias "SetTimer" _
    (ByVal hWnd As Long, _
    ByVal nIDEvent As Long, _
    ByVal uElapse As Long, _
    ByVal lpTimerFunc As Long) As Long

```

Die ersten beiden Argumente `hWnd` und `nIDEvent` interessieren uns hier nicht, wir versorgen sie mit 0. Das dritte Argument `uElapse` gibt die Zeitdauer in Millisekunden an, nach deren Verstreichen die unter `lpTimerFunc` angegebene Funktion aufgerufen werden soll.

Na, stutzig geworden? Aber sicher, denn seit wann rufen wir Funktionen oder allgemein Prozeduren unter einer Long-ID auf? Wir verwenden Prozeduraufrufe ausschließlich unter ihrem Namen:

```

Call BalanceSheets
cAccount.BalanceAccount 1234

```

Wenn das API aber einen so genannten `CallBack` durchführen soll, so benötigt die Funktion einen `Long`-Zeiger auf die Funktion. Zu diesem Zwecke existiert der `AddressOf`-Operator, der diesen `Long`-Zeiger aus dem ihm folgenden Prozedurnamen erzeugt:

```
AddressOf ProzedurName
```

Es handelt sich hierbei nicht um eine Funktion, die nach dem Muster `Ergebnis = AddressOf(Name)` aufgebaut ist, sondern um einen Operator. Eine unerfreuliche Einschränkung hierbei ist der Umstand, dass die Prozedur in einem ungebundenen Modul stehen muss, wodurch eine Kapselung in einer Klasse nicht vollständig möglich ist.

Die Rückgabe dieser `SetTimer`-Funktion ist die neu geschaffene ID des Timers, die zum Löschen des Timers benötigt wird. Kam es zu einem Fehler, so ist die Rückgabe der Funktion 0.

Wir benötigen also noch eine weitere Funktion zum Auflösen des Timers:

```
Declare Function KillTimer Lib "user32" Alias "KillTimer"
    _
        (ByVal hWnd As Long, ByVal nIDEvent As Long) _
        As Long
```

Das Argument `hWnd` setzen wir auch hier auf 0, da wir keinen Fenster-Handle haben. Als `nIDEvent` übergeben wir die `TimerID`, die von der `SetTimer`-Funktion zurückgegeben wurde.

Ein Beispiel. Wir bauen eine Klasse namens `clsTimer`, die über eine Start-Methode aufgerufen wird und nach Ablauf der einzustellenden Zeit ein Ereignis auslöst:

```
Option Explicit
'Variablen
    Private lngTimerID As Long        'ID des Timers
    Private bKillTimer As Boolean     'Wenn True, wird der
                                     'Timer bei seinem ersten
                                     'Callback wieder gekillt
'Events:
    Public Event Timer()              'Deklaration des Timer-
                                     'Ereignisses
'APIs:
    Private Declare Function KillTimer ...
    Private Declare Function SetTimer ...

Private Sub Class_Terminate()
```



```

'=====
'    Timer auflösen, wenn ID ungleich 0
'=====

On Error Resume Next
If lngTimerID <> 0 Then
    KillTimer 0, lngTimerID
End If
End Sub

Public Sub Start(ByVal lngMS As Long, _
    Optional ByVal bKill As Boolean = True)

'=====
'    Start des Timers
'-----
'Argumente:
'    lngMS:  Dauer in Millisekunden
'    bKill:  Wenn True, wird der Timer im ersten Ereignis
'            zerstört
'=====

On Error Resume Next
bKillTimer = bKill
lngTimerID = SetTimer(0, 0, lngMS, _
    AddressOf TimerCallback)
If lngTimerID = 0 Then bKillTimer = False
End Sub

Public Sub TimerEvent()

'=====
'    Ereignis auslösen und evt. vorher Timer auflösen
'    Wird von TimmerCallback in modMain aufgerufen
'=====

On Error Resume Next
If bKillTimer Then
    KillTimer 0, lngTimerID
End If
RaiseEvent Timer
End Sub

```

In modMain müssen wir die hinter der AddressOf-Operator angegebene Prozedur TimerCallback schreiben:

```
Public Sub TimerCallback(hWnd As Long, uMsg As Long, _
                        idEvent As Long, dwTime As Long)

'=====
'   Methode TimerEvent der Timer-Klasse aufrufen
'=====

    On Error Resume Next
    frmMain.cTimer.TimerEvent
End Sub
```

Die Prozedur benötigt diese Signatur, denn die SetTimer will vier Long-Argumente übergeben.

Jetzt benötigen wir noch eine UserForm mit einer Schaltfläche, und wir können den Timer einsetzen:

```
Public WithEvents cTimer As clsTimer
```

```
Private Sub cmdTimer_Click()
```

```
'=====
'   Timer mit 1000 ms starten
```

```
'=====
    cTimer.Start 1000, True
End Sub
```

```
Private Sub cTimer_Timer()
```

```
'=====
'   von Klasse ausgelöstes Ereignis
```

```
'=====
    MsgBox "bingo"
End Sub
```

```
Private Sub UserForm_Initialize()
```

```
'=====
'   Instanz der Timer-Klasse erzeugen
```

```
'=====
```

```
Set cTimer = New clsTimer
End Sub
```

Ein Beispiel für die Integration eines Timers in eine Applikation finden Sie im Hasenspieler, das in Kapitel 8 – Dialoge vorgestellt wird.

7.7.7 Locale Settings

```
Declare Function GetLocaleInfo Lib "kernel32" _
    Alias "GetLocaleInfoA" ( _
        ByVal Locale As Long, _
        ByVal LCType As Long, _
        ByVal lpLCData As String, _
        ByVal cchData As Long) As Long
```

Mit dieser Funktion halten Sie eine wahre Fundgrube an Informationen in den Händen. Nicht nur vielfältige Informationen zur eingestellten Sprache erhalten Sie hiermit, auch alles rund um Zahlen-, Datums- und Uhrzeitformate, selbst die Wochentags- oder Monatsnamen in der eingestellten Sprache. Viele dieser Zugriffe erledigt die Format-Funktion für uns, die sich vermutlich auch dieser API bedient.

In Kapitel 10 werden wir einer Klasse begegnen, die uns einige dieser Informationen als Eigenschaften zur Verfügung stellt. Mit der Funktionsweise der Funktion werden wir uns aber hier auseinander setzen und mit den Argumenten anfangen:

<i>Locale</i>	Eine Konstante, die zwischen system- und anwender-bezogenen Einstellungen unterscheidet: LOCALE_SYSTEM_DEFAULT = &H800 LOCALE_USER_DEFAULT = &H400
<i>LCType</i>	Diese Konstante gibt an, welche Information zurückgegeben werden soll (siehe Beispiele und MSDN). Es existieren knapp 100 Konstanten!
<i>lpLCData</i>	Der Zeichenpuffer für die Rückgabe.
<i>cchData</i>	Die Länge der Rückgabe in <i>lpLCData</i> . Ist dieses Argument gleich 0, enthält die Rückgabe der Funktion die Anzahl der erforderlichen Zeichen inklusive des abschließenden <code>Chr(0)</code> .

Im folgende Beispiel wird ein Codeauszug zur Abfrage des lokalen Namens der eingestellten Sprache präsentiert.

Beginnen wir mit dem Code im (Klassen-)Modul. Dort gibt es eine private Funktion des Namens `GetLocaleValue`, in der die eigentliche API-Funktion aufgerufen wird. Da ich Modulköpfe mit Dutzenden von Zeilen an Konstantendeklarationen entsetzlich finde, nimmt die Funktion `GetLo-`

caleValue die Konstante als Argument entgegen. Wir müssen also später in der diese Funktion aufrufenden Prozedur dafür sorgen, dass wir den erforderlichen Wert übergeben, was die Angelegenheit wesentlich übersichtlicher gestaltet. Lediglich die Konstante `LOCALE_USER_DEFAULT` ist im Modulkopf deklariert.

Option Explicit

'Globale Konstanten

Private Const `LOCALE_USER_DEFAULT` = &H400

'Verwendete APIs

Private Declare Function `GetLocaleInfo` Lib "kernel32" _
Alias "`GetLocaleInfoA`" (ByVal `Locale` As Long, _
ByVal `LCType` As Long, ByVal `lpLCData` As String, _
ByVal `cchData` As Long) As Long

Private Function `GetLocaleValue`(ByVal `lngLCType` As Long) _
As String

'Ermittelt die Locale-Einstellung der übergebenen

'LCType-Konstanten

Dim `nSize` As Long 'Länge des Rückgabe

Dim `strBuffer` As String 'Rückgabe

Dim `lngReturn` As Long 'Dummy für Rückgabe

'Länge der `strBuffer` ermitteln

`nSize` = `GetLocaleInfo`(`LOCALE_USER_DEFAULT`, `lngLCType`, _
`strBuffer`, 0)

'erforderliche Größe einstellen ...

`strBuffer` = `Space`(`nSize`)

'... und Wert ermitteln

`lngReturn` = `GetLocaleInfo`(`LOCALE_USER_DEFAULT`, _
`lngLCType`, `strBuffer`, `nSize`)

'Rückgabe zuschneiden

`GetLocaleValue` = `Left`(`strBuffer`, `nSize` - 1)

End Function

Die Funktion `GetLocaleValue` gibt also alles das zurück, was man ihr aufträgt. In einer anderen, nun öffentlichen Prozedur, wird `GetLocaleValue` mit dem Wert der gewünschten `LCType`-Konstanten aufgerufen:

Public Property `GetCountryLocal`() As String

Const `LOCALE_SCOUNTRY` = &H6 'localized name of country

`CountryLocal` = `GetLocaleValue`(`LOCALE_SCOUNTRY`)

End Property

Diese Eigenschaftsprozedur wird im Code wie folgt verwendet:

```

Private Sub GetLocalCountryName()
    Dim cNLS As clsNLS
    Set cNLS = New clsNLS
    MsgBox cNLS.CountryLocal
End Sub

```

Die folgende Übersicht enthält einen Auszug aus der riesigen Zahl an LC-Type-Konstanten:

Konstantenname	Wert	Bedeutung (üblicher Wert für Deutschland)
LOCALE_ICOUNTRY	&H5	Landescode (49)
LOCALE_SENGCOUNTRY	&H1002	Englischer Name des Landes (Germany)
LOCALE_SCOUNTRY	&H6	Lokaler Name des Landes (Deutschland)
LOCALE_SINTLSYMBOL	&H15	Internationales Währungssymbol (DEM)
LOCALE_SCURRENCY	&H14	Lokales Währungssymbol (DM)
LOCALE_SDECIMAL	&HE	Dezimaltrennzeichen (,)
LOCALE_SENGLANGUAGE	&H1001	Englischer Name der Landessprache (German)
LOCALE_SNATIVELANGNAME	&H4	Lokaler Name der Landessprache (Deutsch)
LOCALE_SLIST	&HC	Listentrennzeichen (;)
	55 + i	Lokaler Name des i-ten Monats (Januar bei i = 1)

Experimentieren Sie ein wenig mit dieser Funktion. Sie ist einfach zu handhaben, wenn Sie dem Beispiel folgen, und sehr ergiebig.

7.8 Dateioperationen

Die in Kapitel 4 ausgeklammerten Dateioperationen gehören nicht notwendigerweise zum Basissprachschatz, wiewohl sie gelegentlich sehr nützlich sein können. Vor Excel 2000 war man noch häufiger darauf angewiesen, da die OpenText-Methode der Workbooks-Collection nicht in der Lage war, sich auf von den Ländereinstellungen abweichende Dezimaltrennzeichen einzustellen. Somit war man gezwungen, Textexporte aus SAP und ähnlichen Systemen per Code zeilenweise einzulesen. Im

Prinzip gestaltet sich das nicht schwierig, und wenn Sie damit rechnen müssen, dass Ihre Anwendung auf Excel 97 laufen könnte, so sind Sie auch heute noch gezwungen so vorzugehen. Aus diesem Grund werden wir uns noch mit diesen Techniken auseinander setzen müssen.

Im ersten Abschnitt beschäftigen wir uns mit Operationen *an* Dateien. Nach einem Beispiel geht es dann weiter mit Operationen *in* Dateien, was ebenfalls mit einem eigenen Beispiel abgeschlossen wird.

7.8.1 Operationen an Dateien

Dir-Funktion

Die `Dir`-Funktion wird in zwei Varianten verwendet. Die erste

`Dir(Pfad [, Attribute])`

<i>Pfad</i>	Suchverzeichnis, auch mit Suchmaske
<i>Attribute</i>	(optional) Integer als Summe aller gewünschten Einzelattribute

dient der Initialisierung der `Dir`-Funktion. In der zweiten Variante wird sie ohne Argumente aufgerufen und handelt sich hierbei von Eintrag zu Eintrag innerhalb des durch das `Pfad`-Argument angegebenen Verzeichnisses. Bevor wir uns mit den Argumenten dieser `Dir`-Funktion auseinander setzen, werfen wir einen Blick auf die generelle Funktionsweise:

```
Dim strFile As String
strFile = Dir("C:\Test\")
Do Until strFile = ""
    '...
    strFile = Dir
Loop
```

Die `Dir`-Funktion wird mit einem `Pfad`-Argument quasi initialisiert und gibt den reinen Namen des ersten Eintrags zurück, der in dem angegebenen `Pfad` gefunden wurde. Danach wird die `Dir`-Funktion ohne Argumente aufgerufen und gibt solange den jeweils nächsten Eintrag im angegebenen Verzeichnis zurück, bis sie nichts mehr findet. Die Rückgabe ist eine leere Zeichenkette, die im Kopf der `Do-Loop`-Schleife abgefangen wird.

Der Initialisierungsaufwurf kann auch Dateimuster enthalten, wie Sie es von der guten alten DOS-Variante kennen:

```
strFile = Dir("C:\Test\*.xls")
```

oder

```
strFile = Dir("C:\Test\*.xl*")
```

Suchen Sie nach zwei verschiedenen Dateierweiterungen, die sich nicht über die Wildcards »*« oder »?« verbinden lassen, so müssen Sie mit »*.« initialisieren und die Dateierweiterung gezielt, beispielsweise mit der Right-Funktion, überprüfen:

```
strFile = Dir("C:\Test\*.")
Do Until strFile = ""
    If UCase(Right(strFile, 4)) = ".DOC" Or _
        UCase(Right(strFile, 4)) = ".TXT" Then
        ' ...
End If
```

Halten Sie es für eine Marotte, aber ich gebe das Suchmuster generell mit an, auch wenn »C:\Test\« gleichbedeutend ist mit »C:\Test*.«.

Nun wenden wir uns dem optionalen Attribute-Argument zu, das die arithmetische Summe aller gewünschten Einzelattribute enthält:

Konstante	Wert	Bedeutung
vbNormal	0	normal, also weder 1 noch 2, 4, 16 oder 32
vbReadOnly	1	schreibgeschützt
vbHidden	2	versteckt
vbSystem	4	Systemdatei
vbDirectory	16	Verzeichnis
vbArchive	32	Datei wurde seit dem letzten Sichern geändert. Dieses Attribut ist unwirksam!

Was auf den ersten Blick nicht einleuchtet, ist, dass die Dir()-Funktion mit vbDirectory (oder etwas anderem) als Attribut diesen Eintragstyp *zusätzlich zu Dateien ohne Attribute* berücksichtigt. Mit anderen Worten:

In der Rückgabe von Dir(»C:*.«, vbDirectory) sind nicht nur Verzeichnisse, sondern zusätzlich auch alle Dateien ohne Attribute enthalten.

Wir werden mit der GetAttr-Funktion das Kraut kennen lernen, das dagegen gewachsen ist. Doch zuerst schauen wir uns die Verwendung dieses Arguments an Beispielen an:

Der folgende Aufruf veranlasst die Dir-Funktion, auch Verzeichnisse zurückzugeben:

```
strFile = Dir("C:\Test\*.*", vbDirectory)
```

Im nächsten Beispiel werden sowohl schreibgeschützte als auch versteckte Dateien berücksichtigt:

```
strFile = Dir("C:\Test\*.*", vbReadOnly + vbHidden)
```

Die Auswertung, ob nun Dateien ausgewertet werden sollen, die schreibgeschützt oder versteckt sind oder schreibgeschützt sind und gleichzeitig versteckt sind, müssen Sie ebenfalls mit der `GetAttr`-Funktion überprüfen, indem Sie die Präsenz der einzelnen Merkmale mit `Or` bzw. `And` überprüfen.

FileDateTime-Funktion

Diese Funktion gibt Datum und Uhrzeit der letzten Änderung der Datei zurück, die mit vollständigem Namen angegeben werden muss:

```
MsgBox FileDateTime(strPath & strFile)
```

FileLen-Funktion

`FileLen` gibt die Größe der Datei in Bytes zurück. Auch hier muss der vollständige Name übergeben werden:

```
MsgBox FileLen(strPath & strFile)
```

Wollen Sie die Größe in Kilobytes oder Megabytes, so müssen Sie die Rückgabe dieser Funktion durch 1024 bzw. $1024 * 1024$ teilen:

```
MsgBox FileLen(strPath & strFile) / 1024 & " KB"  
MsgBox FileLen(strPath & strFile) / 2 ^ 20 & " MB"
```

GetAttr-Funktion

Wie wir bei der Vorstellung der `Dir`-Funktion sahen, gibt sie auch mit oder ohne weitere Argumente alle Dateien mit dem Attribut `Normal` zurück. Und Dateien ohne Attribute sind nun mal die Regel. Wollen Sie gezielt nach Einträgen mit einem Attribut suchen, müssen Sie die `GetAttr`-Funktion verwenden.

Die Rückgabe dieser Funktion ist die Summe aller dem Argument `Pfadname` zugeordneten Attribute:

Das schreibgeschützte, versteckte Verzeichnis »E:\Test« ergibt folglich 19 als Summe von 1, 2 und 16. Wenn der zurückgegebene Wert also die Summe der vorhandenen Attribute ist, so stellt sich die Frage, wie man ein einzelnes Attribut aus dieser Zahl heraussummeln kann. Die Lösung wurde im Abschnitt `Farben` bereits behandelt, wo wir aus einem `Long`-Wert die Bytes 0, 1 oder 2 durch Maskierung erhielten. Wollen wir also

wissen, ob in dem Integer, den `GetAttr` liefert, die Zahl 16 enthalten ist, so führen wir eine binäre UND-Verknüpfung an der Stelle 2^4 durch und prüfen, ob das Ergebnis 2^4 ist:

```
If (GetAttr("E:\Test") And vbDirectory) = vbDirectory Then
```

Wenn Sie unter NT folgende Zeile laufen lassen, ernten Sie interessanterweise einen Laufzeitfehler:

**Laufzeitfehler bei
pagefile.sys unter NT**

```
strPath = "C:\pagefile.sys"
```

```
If (GetAttr(strPath) And vbDirectory) = vbDirectory Then
```

Die Ursache liegt nach MSDN darin, dass diese Datei »on first encounter«, wie es dort so schön heißt, als Verzeichnis interpretiert wird. Die Überprüfung durch die `GetAttr()`-Funktion wiederum produziert den Laufzeitfehler 5 »Ungültiger Prozeduraufruf oder ungültiges Argument«, da es sich eben doch nicht um ein Verzeichnis handelt. Sie müssen also mit einem vorangeschalteten `If` vermeiden, dass die `GetAttr()`-Funktion diesen Schwindler in die Fänge bekommt:

```
...
```

```
If InStr(1, Ucase(strPath), "PAGEFILE.SYS") = 0 Then
```

```
...
```

7.8.2 Ein Beispiel

Im folgenden Beispiel werden die Funktionen dieses Abschnitts mit Ausnahme der `FreeFile`-Funktion in einem zusammenhängenden Beispiel vorgestellt. `FreeFile` findet Berücksichtigung im nächsten Abschnitt.

Die Arbeitsmappe `Dateioperationen.xls` auf der Buch-CD enthält unter anderem eine Tabelle, in der ein Verzeichnis ausgelesen wird und die Informationen ähnlich dem Explorer aufgelistet werden, wie Abbildung 7.13 zeigt.

Der Modulkopf

Im Modul werden die vier Spaltenzeiger vereinbart, die für die Ausgabespalten verwendet werden:

```
Option Explicit
```

```
Private Const iColName As Long = 3
```

```
Private Const iColFileLen As Long = 4
```

```
Private Const iColDateTime As Long = 5
```

```
Private Const iColAttrib As Long = 6
```

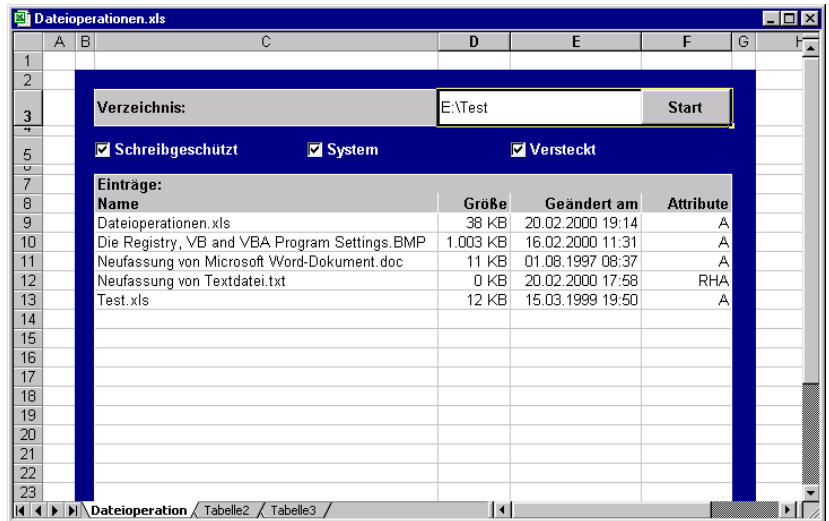


Abbildung 7.13: Dateioperationen, Ausgabetable

Funktionen

Um die eigentliche Prozedur etwas übersichtlicher zu gestalten, kommen zwei Funktionen zum Einsatz. Die erste sammelt die per CheckBox eingestellten Attribute ein und gibt sie als Long-Wert zurück:

Private Function GetAttributes() As Long

```
'eingestellte Attribute aufsummieren
If chkReadOnly.Value Then
    GetAttributes = GetAttributes + vbReadOnly
End If
If chkSystem.Value Then
    GetAttributes = GetAttributes + vbSystem
End If
If chkHidden.Value Then
    GetAttributes = GetAttributes + vbHidden
End If
```

End Function

Die nächste Funktion prüft das Vorhandensein eines Attributes einer Datei:

Private Function CheckAttrib(strFile As String, _ lngAttrib As Long) As Boolean

```
'das Attribut lngAttrib in strFile prüfen
CheckAttrib = _
    ((GetAttr(strFile) And lngAttrib) = lngAttrib)
```

End Function

Die cmdStart_Click

Dies ist die zentrale Routine, in der Dateien eingelesen und auch in der Zieltabelle ausgegeben werden:

```
Private Sub cmdStart_Click()
    'Dateien einlesen und ausgeben
    Dim iRow As Long           'Zeilenzeiger
    Dim lngAttrib As Long      'summierte Attribute
    Dim strFile As String      'Rückgabe aus Dir-Funktion
    Dim strPath As String      'zu durchsuchendes Verzeichnis
    Dim strAttrib As String    'Attribute zur Ausgabe
    iRow = 8
    'Datenbereich löschen
    Me.Cells(iRow, iColName).CurrentRegion.Offset(2, _
        0).ClearContents

    'Attribute einlesen
    lngAttrib = GetAttributes
    'Startverzeichnis festlegen
    strPath = _
        ThisWorkbook.Names("InitPath").RefersToRange.Value
    If Right(strPath, 1) <> "\" Then strPath = strPath & "\"
    'Dir initialisieren und erste Datei einlesen
    strFile = Dir(strPath & "**.*", lngAttrib)
    'Abbruch, wenn keine Dateien gefunden
    If strFile = "" Then
        Me.Cells(iRow + 1, iColName).Value = "Keine D..."
        Exit Sub
    End If
    'Einträge abarbeiten
    Do Until strFile = ""
        'Sonderfall abfangen
        If UCase(strFile) <> "PAGEFILE.SYS" Then
            'Zeilenzeiger inkrementieren
            iRow = iRow + 1
            'Attribute-String zurücksetzen
            strAttrib = ""
            'Attribute-String zusammensetzen
            If CheckAttrib(strPath & strFile, vbReadOnly) Then
                strAttrib = strAttrib & "R"
            End If
            If CheckAttrib(strPath & strFile, vbHidden) Then
                strAttrib = strAttrib & "H"
            End If
        End If
    End Do
End Sub
```

```

        If CheckAttrib(strPath & strFile, vbSystem) Then
            strAttrib = strAttrib & "S"
        End If
        If CheckAttrib(strPath & strFile, vbArchive) Then
            strAttrib = strAttrib & "A"
        End If
        'Ausgabe
        Me.Cells(iRow, iColName).Value = strFile
        Me.Cells(iRow, iColFileLen).Value = _
            Format(FileLen(strPath & strFile) / 1024, _
                "#,##0") & " KB"
        Me.Cells(iRow, iColDateTime).Value = _
            FileDateTime(strPath & strFile)
        Me.Cells(iRow, iColAttrib).Value = strAttrib
    End If
    'nächste Datei einlesen
    strFile = Dir
Loop
End Sub

```

7.8.3 Operationen in Dateien

Eine vollständige Darstellung des Themas wäre zwar sinnvoll um die teils recht verwirrenden Informationen der Hilfe und der MSDN ein wenig ins rechte Licht zu rücken. Andererseits ist es ein Anliegen dieses Buches, das riesige Sprachvolumen von Excel und VBA ein wenig auf das zu reduzieren, was tatsächlich erforderlich ist. Nach reiflicher Überlegung werden ich mich darauf beschränken, was gerade im Kontext von Excel auch sinnvoll ist. Diesem Ansatz fallen vor allem die vielfältigen Modi zum Opfer, in denen Dateien geöffnet werden können. In der Praxis reduzieren sie sich auf ein reines Einlesen von Textdateien. Zwar öffne ich auch gelegentlich Dateien im binären Modus, aber nur um sicherzustellen, dass ich über exklusive Rechte verfüge. Doch das ist wirklich die Ausnahme. Gleiches gilt für schreibende Zugriffe. In Excel können wir die Inhalte in der benötigten Reihenfolge in eine Tabelle schreiben und diese durch Speichern der Arbeitsmappe als Textdatei ablegen. Schreibende Zugriffe per Put, Write oder Print sind somit schlechterdings überflüssig.

Diese Reduzierung des Themas erspart uns zwanzig Seiten, die womöglich den Blick auf das Wesentliche verstellt hätten.

In diesem Abschnitt werden wir Textdateien öffnen und die Inhalte verarbeiten. Bevor wir in einem Beispiel die Verwendung dieser Sprachelemente zeigen, schauen wir uns zunächst die Sprachelemente selbst an.

FreeFile-Funktion

Die FreeFile-Funktion gibt eine Dateinummer zurück, die für Operationen mit Datei-Handle verwendet werden kann:

```
Dim hFile As Long  
hFile = FreeFile
```

Open-Anweisung

Mit der Open-Anweisung wird eine Datei geöffnet. Wir verwenden sie ausschließlich zum Einlesen der Daten:

```
Open strFile For Input As #hFile
```

Hier benötigen wir also den Datei-Handle, den uns die FreeFile-Funktion liefert.

Close-Anweisung

Mit Close #hFile wird die unter #hFile geöffnete Datei geschlossen. Lassen Sie das Argument #hFile weg, so werden alle per Open geöffneten Datei geschlossen.

Line Input-Anweisung

Diese Anweisung liest eine Zeile der Datei ein:

```
Line Input #hFile, strLine
```

Auch Line Input benötigt den Datei-Handle und außerdem eine String-Variable, an die sie die Textzeile übergeben kann.

7.8.4 Ein Beispiel

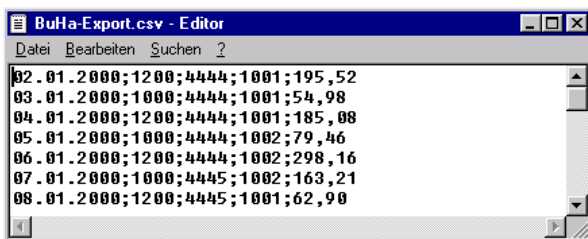


Abbildung 7.14: Inhalt der BuHa-Export.csv

Die zweite Tabelle der Dateioperationen.xls beschäftigt sich mit dem Einlesen von Daten aus einer Textdatei namens BuHa-Export.csv (siehe Abbildung 7.14). Das Ergebnis soll in eine Tabelle eingetragen werden und anschließend so aussehen, wie in Abbildung 7.15 zu sehen.

	A	B	C	D	E
1	Dateiname: BuHa-Export.csv Datei einlesen				
2					
3					
4					
5					
6	Datum	Kto	Gkto	KoSt	Umsatz
7	02.01.2000	1200	4444	1001	95,52
8	03.01.2000	1000	4444	1001	54,98
9	04.01.2000	1200	4444	1001	85,08
10	05.01.2000	1000	4444	1002	79,46
11	06.01.2000	1200	4444	1002	98,16
12	07.01.2000	1000	4445	1002	63,21
13	08.01.2000	1200	4445	1001	62,90
14	09.01.2000	1000	4445	1001	70,25
15	10.01.2000	1200	4445	1001	85,50

Abbildung 7.15: Tabelle nach Import der Textdatei

Der Code

```
Private Sub cmdReadFile_Click()
    'Datei einlesen und ausgeben
    Dim hFile As Long           'Datei-Handle
    Dim strFile As String       'Dateiname
    Dim strLine As String       'eingeliesene Zeile
    Dim varValues As Variant     'Array der Werte
    Dim iValue As Long          'Zeiger in varValues
    Dim iRow As Long            'Zeilenzeiger
    'Dateiname zusammensetzen und prüfen
    strFile = ThisWorkbook.Path & "\" & _
        ThisWorkbook.Names("FileName").RefersToRange.Value
    If Dir(strFile) = "" Then
        Beep
        MsgBox "Die Datei existiert nicht."
        Exit Sub
    End If
    iRow = 6
    Me.Cells(iRow, 1).CurrentRegion.Offset(1).ClearContents
    'Datei-Handle ermitteln
    hFile = FreeFile
    'Datei öffnen und erste Zeile einlesen
```

```

Open strFile For Input As #hFile
Line Input #hFile, strLine
'Schleife, bis strLine = ""
Do Until strLine = ""
    iRow = iRow + 1
    'Array aus strLine bilden und eintragen
    varValues = Split(strLine, ";")
    For iValue = 0 To UBound(varValues)
        If iValue = 4 Then
            Me.Cells(iRow, 1 + iValue).Value = _
                varValues(iValue) * 1
        Else
            Me.Cells(iRow, 1 + iValue).Value = _
                varValues(iValue)
        End If
    Next
    'Abbruch, wenn EOF
    If EOF(hFile) Then Exit Do
    'nächste Zeile einlesen
    Line Input #hFile, strLine
Loop
'Datei(en) schließen
Close
End Sub

```

Auf zwei Besonderheiten müssen wir noch eingehen.

Excel-Zellen erkennen Werte der Datentypen Single oder Double nicht sauber, was man an der linksbündigen Darstellung in der Zelle erkennen kann. Solche Werte multipliziere ich in der Zeile, in der sie in die Zelle eingetragen werden, kurzerhand mit 1 und der Spuk ist vorüber.

Die Textdateien können nach dem Return-Zeichen der letzten Zeile zu Ende sein, können aber auch noch eine Geisterzeile aufweisen. Die obige Variante wird mit beiden Fällen fertig. Wenn Sie es nicht so machen, wie dort zu sehen, kann es Ihnen passieren, dass der Fehler 62 – *Einlesen hinter Dateifende* auftritt oder dass bis an das Tabellenende der letzten Datensatz wiederholt wird.

7.9 Rekursionen

Wenn eine Aufgabe mehrfach durchgeführt werden muss, so stehen uns dafür die bekannten Schleifenkonstruktionen mit For-Next oder Do-Loop zur Verfügung. Das ist zweifellos ein sicherer und verlässlicher Weg, wenngleich diese Schleifen leicht unübersichtlich werden können.

7.9.1 Berechnung einer Fakultät

Eine Fakultät lässt sich in einer Routine nach folgendem Muster berechnen:

```
Sub ComputeFaculty_1()
    Dim i As Long
    Dim x As Long
    Dim strMessage As String
    x = 5
    strMessage = x & "! = "
    For i = x - 1 To 2 Step -1
        x = x * i
    Next
    MsgBox strMessage & x
End Sub
```

Die elegantere Version ruft eine Funktion namens `Faculty` auf und übergibt ihr den zu berechnenden Wert.

```
Private Sub ComputeFaculty_2()
    Dim x As Long
    x = 5
    MsgBox x & "! = " & Faculty(x)
End Sub
```

Und diese Funktion `Faculty` ruft sich so lange selbst auf, bis ihr Aufrufargument 1 ist:

```
Private Function Faculty(ByVal lngValue As Long) As Variant
    Faculty = 1
    If lngValue > 1 Then
        Faculty = lngValue * Faculty(lngValue - 1)
    Endif
End Function
```

Ruft eine Prozedur sich selbst wieder auf, so handelt es sich um eine Rekursion. In der Aufrufeliste lässt sich das gut beobachten, wie die folgende Abbildung 7.16 zeigt.

Die Vorteile der Rekursion springen in diesem Beispiel noch nicht ins Auge. Trotzdem mag ich diese kleine Funktion, weil man damit so schön die Qualitäten der Starentwickler bei Vorstellungsgesprächen testen kann. Sie sagt mehr aus als all die tollen Zeugnisse und Diplome.

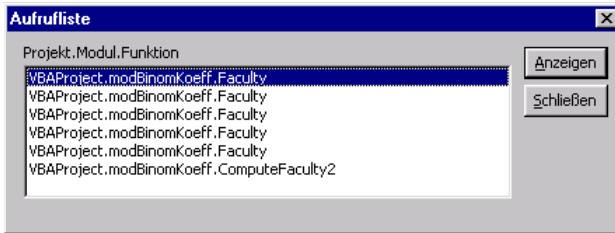


Abbildung 7.16: Rekursion, Aufrufeliste

Wenden wir uns einem weiteren Einsatzgebiet der Fakultät zu. Diese Fakultätsberechnung wird beispielsweise in der Kombinatorik eingesetzt, um die Wahrscheinlichkeit für einen Sechser im Lotto zu berechnen. Dieser Binomialkoeffizient n über k wird so berechnet:

Binomialkoeffizient

```
Private Function BinomKoeff(n, k) As Long
    BinomKoeff = Faculty(n) / (Faculty(n - k) * Faculty(k))
End Function
```

Wenn Sie diese Funktion mit $n = 49$ und $k = 6$ füttern, erhalten Sie die berühmten 13.983.816 Kombinationen einer Ziehung von 6 aus 49 ohne Wiederholung (ohne Wiederholung meint, dass eine gezogene Kugel nicht wieder in die Trommel zurückgeworfen wird).

7.9.2 Bäume durchsuchen

Die Fakultät und selbst der Binomialkoeffizient ließen sich auch herkömmlich berechnen. Schwieriger wird es beim Durchsuchen von Bäumen. Im folgenden Beispiel wird eine Datei in einem Verzeichnis(-baum) gesucht:

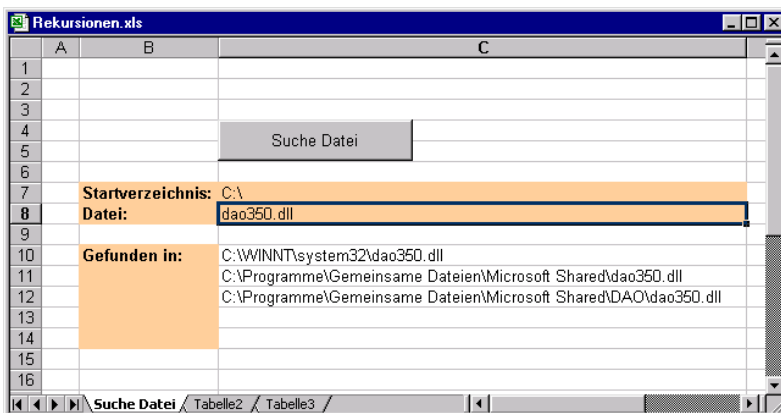


Abbildung 7.17: Rekursionen, Dateisuche

Zu Anfang sieht die Aufgabe recht einfach aus. Wir übergeben das Startverzeichnis an die `Dir`-Funktion und lassen uns die normalen Dateien sowie die Verzeichnisse zurückgeben:

```
strPath = Dir(strActPath & *.* , vbDirectory)
Do Until srPath = ""
    If ...
        strPath = Dir
Loop
```

Was tun wir jedoch mit neuen Verzeichnissen, die uns auf der Suche nach der gewünschten Datei über den Weg laufen? In jedem Falle müssen wir das aktuell durchsuchte Verzeichnis vollständig abarbeiten, da wir nicht mehrere Instanzen der `Dir`-Funktion bilden können. Sie wird sich immer nur eine Position merken können.

Hier zeichnet sich ein Problem ab, denn wir müssen eventuell beim Durchsuchen aufgetauchte weitere Verzeichnisse auf Halde legen und uns später darum kümmern. Wir benötigen eine dynamische Speicherkomponente, die als Halde dient. Hierfür bietet sich ein Array nachgerade an. Wenn wir das aktuelle, gerade untersuchte Verzeichnis ebenfalls in diesem Array ablegen können, sind alle unsere Probleme gelöst.

Abbildung 7.18 zeigt das dynamische Verhalten dieses Arrays, welches in dem Beispiel `strPaths()` genannt wird.

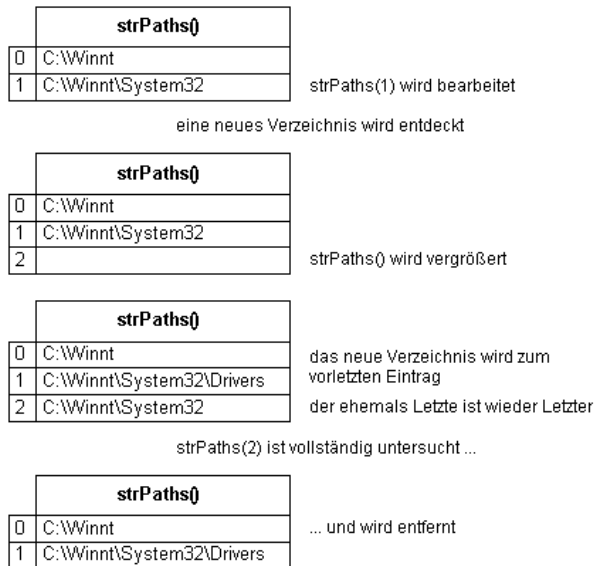


Abbildung 7.18: Rekursionen, Dynamik des Verzeichnis-Arrays

Um die gefundenen Pfade aufzubewahren, bietet sich ein zweites Array an, dessen Handling allerdings wesentlich einfacher ist, denn neue Fundstellen werden einfach hinten angefügt.

Zur Lösung dieser Aufgabe werden folgende Modulvariablen verwendet:

```
Private strPaths() As String 'zu durchsuchende Pfade
Private strFound() As String 'gefundene Dateien
Private strFile As String    'gesuchte Datei
```

Die Prozedur FindFile

Diese Prozedur untersucht das per `strActPath` übergebene Verzeichnis. Findet sie ein Verzeichnis, wird dieses in der beschriebenen Art und Weise in das Array `strPaths()` hineinmanövriert. Stößt sie hingegen auf die gesuchte Datei, so wird das aktuelle Verzeichnis nebst dem Dateinamen in das zweite Array `strFound()` eingetragen.

Am Ende der Untersuchung des aktuellen Verzeichnisses wird dieses aus dem Array entfernt. Dies stellt insofern kein Problem dar, da sich das aktuelle Verzeichnis am Ende des Arrays aufhält. Ein `ReDim` mit `-1` erledigt das für uns. Sind danach noch weitere Verzeichnisse in dem Array `strPaths()`, so wird die Prozedur `FindFile` erneut aufgerufen und ihr der aktuell letzte Eintrag übergeben.

```
Private Sub FindFile(ByVal strActPath As String)
    'Abarbeiten des aktuellen Verzeichnisses
    Dim strPath As String 'Rückgabe der Dir-Funktion
    On Error GoTo errHandler
    'Statusbar aktualisieren
    Application.StatusBar = "Verzeichnis: " & strActPath
    'ggf. Backslash anhängen
    If Right(strActPath, 1) <> "\" Then
        strActPath = strActPath & "\"
    End If
    'ersten Eintrag einlesen
    strPath = Dir(strActPath & " *.*", vbDirectory)
    Do Until strPath = ""
        'Sonderfälle abfangen
        If UCase(strPath) <> "PAGEFILE.SYS" And _
            strPath <> "." And strPath <> ".." And _
            strPath <> "?" Then
            'Prüfen, ob Eintrag ein Verzeichnis ist
            If (GetAttr(strActPath & strPath) And _
                vbDirectory) = vbDirectory Then
                'Array-Grenze erhöhen und aktuelles
```

```

        'Verzeichnis als Vorletztes eintragen
        ReDim Preserve strPaths(UBound(strPaths) + 1)
        strPaths(UBound(strPaths)) = _
            strPaths(UBound(strPaths) - 1)
        strPaths(UBound(strPaths) - 1) = _
            strActPath & strPath
        'Prüfen, ob Dateiname identisch mit Suchbegriff
        ElseIf UCase(strPath) = UCase(strFile) Then
            'aktuelles Verzeichnis anhängen
            strFound(UBound(strFound)) = strActPath & _
                strFile
            ReDim Preserve strFound(UBound(strFound) + 1)
        End If
    End If
    'neuen Eintrag einlesen
    strPath = Dir
Loop
'Prüfen, ob Array noch Elemente enthält
If UBound(strPaths) > 0 Then
    'letztes Element entfernen ...
    ReDim Preserve strPaths(UBound(strPaths) - 1)
    '... und FindFile mit nun letztem aufrufen
    FindFile strPaths(UBound(strPaths))
End If
Exit Sub

errHandler:
    Err.Raise Err.Number, "shtFindFile:FindFile"
End Sub

```

Die Prozedur cmdFindFile_Click

Nachdem FindFile uns im Prinzip die meiste Arbeit abnimmt, müssen in dieser Ereignisroutine lediglich die Variablen vorbereitet werden und der Aufruf der FindFile mit dem Einstiegsverzeichnis erfolgen. Am Ende werden noch die gefundenen Stellen aus dem Array strFound() in die Tabelle ausgegeben.

```

Private Sub cmdFindFile_Click()
    'Laden der Parameter und Aufruf der Findfile
    Dim iFound As Long          'Zeiger in strFound()
    On Error GoTo errHandler
    'obligatorische 1. Elemente hinzufügen
    ReDim strPaths(0)
    ReDim strFound(0)

```

```

'Ausgabebereich löschen
shtFindFile.Range("B10").CurrentRegion.Offset(0, _
                                     1).ClearContents

'Parameter einlesen
strFile = _
    ThisWorkbook.Names("Filename").RefersToRange.Value
strPaths(0) = _
    ThisWorkbook.Names("InitPath").RefersToRange.Value
'FindFile mit InitPath aufrufen
FindFile strPaths(0)
'Ausgabe
If UBound(strFound) = 0 Then
    shtFindFile.Cells(10, 3).Value = _
        "Die Datei wurde nicht gefunden."
Else
    ReDim Preserve strFound(UBound(strFound) - 1)
    For iFound = 0 To UBound(strFound)

        shtFindFile.Cells(10 + iFound, 3).Value = _
            strFound(iFound)

    Next
End If
'Statusbar zurücksetzen
Application.StatusBar = False
Beep
Exit Sub

errHandler:
If InStr(1, Err.Source, ":") Then
    Err.Source = "shtFindFile:cmdFindFile_Click"
End If
cError.ErrorMessage
End Sub

```

Aufgabe

Die FindFile kann in dieser Form mit Platzhaltern nichts anfangen. Eine Suche nach DAO*.dll würde keine Übereinstimmung finden. Was ist zu tun?

7.10 Programmausführung unterbrechen

Auf der Buch-CD befindet sich eine Datei namens DoEvents.xls, in der sich ein alter Bekannter befindet, wie Abbildung 7.19 zeigt.

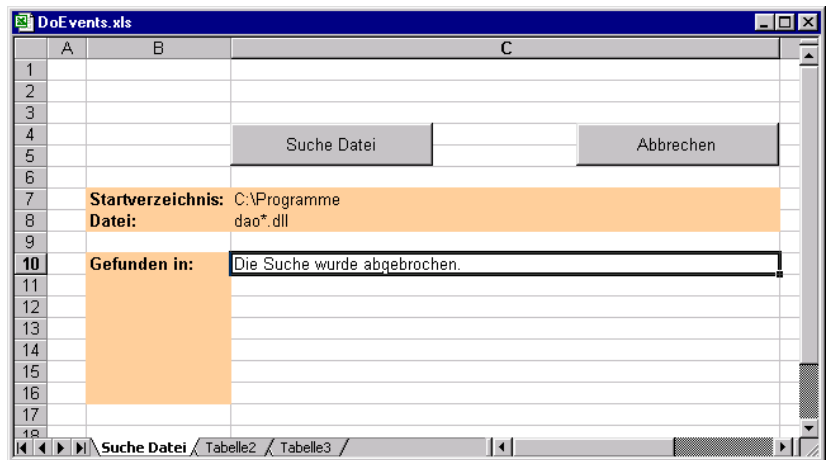


Abbildung 7.19: Abbrechen des Suchvorgangs

Die Tabelle enthält aber eine neue Schaltfläche, mit der unsere Suche nach Dateien abgebrochen werden kann. Solange die `FindFile`-Prozedur aktiv ist, ist es mit regulären Mitteln nicht möglich, den Suchvorgang abzubrechen. Das ist aber ein Stück Anwenderfreundlichkeit, die immer einen guten Eindruck hinterlässt. Wie geht so was?

Zuerst benötigen wir eine Variable, die in allen Programmteilen zugänglich ist. Da sich unser Code auf ein Modul beschränkt, reicht eine private Variable:

```
Private bCancel As Boolean      'Abbruchvariable
```

Das Nächste ist eine Ereignisroutine für unsere neue Schaltfläche, in der die neue Variable `bCancel` auf `True` gesetzt wird:

```
Private Sub cmdCancel_Click()  
    bCancel = True  
End Sub
```

Sodann müssen wir überlegen, an welcher Stelle unsere Anwendung regelmäßig vorbeischaut, und dort per Code den Abbruch des Vorgangs einleiten. Die `Do-Loop`-Schleife, in der die Rückgaben der `Dir`-Funktion abgearbeitet werden, bietet sich hierfür an:

```
Do Until strPath = ""  
    If ...  
  
        strPath = Dir  
        DoEvents
```

```

    If bCancel Then
        ReDim strPaths(0)
        Exit Do
    End If
Loop

```

Die Anweisung `DoEvents` führt (unter anderem) dazu, dass die Ereignisse in unserer Datei abgearbeitet werden. Hat seit dem letzten `DoEvents` jemand auf die Abbrechen-Schaltfläche geklickt, so wird die dortige Ereignisprozedur abgearbeitet und unsere private Variable `bCancel` auf `True` gesetzt. Und genau das fragen wir in der nächsten Zeile ab.

Versuchen Sie nach Möglichkeit der Versuchung zu widerstehen, die Anweisung `End` in den Code zu schreiben. Zwar bricht sie die Ausführung des Codes ab, hinterlässt aber die Anwendung in irgendeinem halbgenen Zustand. Beendet Sie stattdessen die Anwendung in einer geordneten Art und Weise. Wir können unserer Anwendung den Boden entziehen, indem wir das Array `strPaths()` bis auf das letzte Element leeren.

Dadurch können wir die Routine, die unser Programm gestartet hat, zu einem Ende führen. Am Ende der `cmfFindFile_Click` muss der Code wie folgt abgeändert werden (neue Zeilen sind hervorgehoben):

```

If bCancel Then
    shtFindFile.Cells(10, 3).Value = "Die Suche wurde ..."
Else
    If UBound(strFound) = 0 Then
        ...
    Else
        ...
    End If
End If
cmfFindFile.Activate

```

Die letzte Zeile ist notwendig, weil die Start-Schaltfläche andernfalls in eingedrücktem Zustand hinterlassen wird.

Dass `DoEvents` andere Excel-Ereignisse abfragt, ist nur ein Teil der Wahrheit. In Wirklichkeit erhält Windows die Kontrolle über die Anwendung und fragt alle anderen Anwendungen, ob etwas Aufregendes passiert ist. Sind alle soweit zufrieden, erhält unsere aktuelle Excel-Instanz wieder den Fokus und kann dort den Klick auf die Abbrechen-Schaltfläche verarbeiten, sofern dieser erfolgte.



Dialoge

8

Kapitelüberblick

8.1	Funktionsweise von Dialogen	371
8.2	Login-Dialog	374
8.2.1	Erzeugen und Anordnen der Steuerelemente	376
8.2.2	Optische Korrekturen	378
8.2.3	Ergänzung des Formularfußes	380
8.2.4	Aktivierreihenfolge	382
8.2.5	Funktionale Aspekte des Login-Dialogs	383
8.2.6	Codierung des Login-Dialogs	
8.2.7	Aufruf und Auswertung des Logins	390
8.3	Gemeinsamkeiten der Steuerelemente	391
8.3.1	Eigenschaften	391
8.3.2	Font-Objekt	397
8.3.3	Methoden	399
8.3.4	Ereignisse	403
8.4	UserForm	414
8.4.1	Eigenschaften	414
8.4.2	Methoden	414
8.4.3	Ereignisse	415
8.4.4	Eine kleine Spielerei mit einer UserForm	416
8.5	Label	422
8.6	TextBox	423
8.7	ComboBox	424
8.7.1	Eigenschaften	425
8.7.2	Methoden	427
8.7.3	Ereignisse	428

8.7.4	Besonderheit der ComboStyle-ComboBox	429
8.7.5	Wie sortiert man die Einträge einer ComboBox	430
8.8	ListBox	431
8.8.1	SingleSelect ListBoxes	431
8.8.2	MultiSelect ListBoxes	431
8.8.3	Anbindung an Tabellen	433
8.9	CheckBox	433
8.10	OptionButton	434
8.11	CommandButton und ToggleButton	434
8.12	SpinButton und ScrollBar	435
8.13	MultiPage und TabStrip	436
8.14	RefEdit	437
8.15	Steuerelemente in Tabellen	438

Den ersten Kontakt mit Steuerelementen hatten wir bereits in Kapitel 3. Dort wurde eine Schaltfläche aus der Steuerelement-Toolbox in eine Tabelle platziert und in der Ereignisroutine `cmdSquare_Click` die Berechnung der Quadrate codiert. Auch in diesem Kapitel werden wir uns noch einmal mit Steuerelementen beschäftigen, die in Tabellen arbeiten. Doch unser Hauptaugenmerk gilt UserForms und den darin eingesetzten Steuerelementen.

8.1 Funktionsweise von Dialogen

Was sind Dialoge? Dialoge sind graphische Benutzerschnittstellen, die im *Dialog* mit dem Anwender eine fest umrissene Aufgabe lösen. Implementiert sind Dialoge als Formulare, die wiederum als Container für Steuerelemente dienen. Wenn Sie sich den Optionsdialog von Excel selbst vor Augen führen, dann haben Sie ein gutes Beispiel für einen Dialog, den wir in VBA nachbilden könnten, obwohl er für VBA-Verhältnisse etwas umfangreich ist.

Sieht man einmal von der einen oder anderen Feinheit dieses Dialogs ab, so könnten wir ihn ohne weiteres nachbauen. Abbildung 8.1 zeigt die Steuerelemente, die sich standardmäßig in der Entwicklungsumgebung anbieten.

Steuerelemente



Abbildung 8.1: Die Standard-Steuerelemente

Steuerelemente sind ActiveX-Codekomponenten, die in der Regel über eine Oberfläche verfügen. Diese Oberfläche ist in der Lage, auf Aktionen

des Anwenders zu reagieren. Schauen wir uns einmal an, welche Möglichkeiten diese Steuerelemente bieten und wie sie auf Benutzeraktionen reagieren.

Name	Funktion	Reaktionsbeispiele
Label	Anzeige von Texten (und auch Graphiken)	in der Regel keine
TextBox	Editierbarer Textcontainer	Änderung des Inhalts, Fokuserhalt oder -verlust
ComboBox	Auswahl eines Eintrags aus einer Liste, aber auch die Veränderung eines Eintrags	Mausklick, Änderung des Textes
ListBox	Auswahl eines oder mehrerer Einträge, die für den Anwender jedoch nicht veränderbar sind	Mausklick
Checkbox	Aus- oder Abwahl einer unabhängigen Option	Mausklick
OptionButton	Auswahl einer Option unter mehreren	Mausklick
ToggleButton	Ziel für einen Mausklick, wechselt zwischen gedrücktem und nicht gedrücktem Zustand	Mausklick
CommandButton	Ziel für Mausklick	Mausklick
ScrollBar	horizontale und vertikale Navigation in kleinen und großen Schrittweiten	Zustandsänderung
MultiPage	Gruppierung von Steuerelementen	Mausklick
TabStrip	Gruppierung von Steuerelementen	Mausklick
SpinButton	Veränderung eines Wertes in Einer-Schritten	Änderung
RefEdit	Zellbereichsauswahl per Maus bei geöffnetem Dialog	Inhaltsänderung

Diese Liste trägt ein wenig, denn die Steuerelemente sind in der Lage, auf Drag & Drop zu reagieren und Zustandsänderung der Maustasten sowie Tastaturoperationen zu registrieren. Zudem unterscheiden sie zwischen einem einfachen Klick und einem Doppelklick, der übrigens nichts anderes ist als zwei aufeinander folgende Klicks innerhalb eines einstellbaren Zeitfensters von in der Regel 500 Millisekunden.

Ereignisse Diese von den Steuerelementen registrierten Benutzeraktionen nennt man Ereignisse (Events). Unsere Entwicklungsumgebung bietet diesen Steuerelementen eine Schnittstelle an, an der sie diese Ereignisse zu unserem Codemodul weitermelden können. Dort können wir sie entgegen-

nehmen und entsprechend darauf reagieren. Eines der Merkmale der VBA-Entwicklungsumgebung ist es also, Containern und deren untergeordneten Elementen eine bidirektionale Schnittstelle zu bieten:

Steuerelemente können durch Eigenschaften und Methoden von uns beeinflusst werden.



Und ...

Steuerelemente melden Zustandsänderungen über Ereignisse an die gebundenen Codemodule.



Untersuchen wir diese beiden Aussagen anhand einer Schaltfläche. Wir können über die Caption-Eigenschaft der Schaltfläche die Aufschrift *Abbrechen* verpassen, sie über die Cancel-Eigenschaft dazu verdonnern, auf die Escape-Taste genauso wie auf einen Klick zu reagieren.

Im Gegenzug meldet sie uns brav, wenn sie einen Klick verspürt oder die Escape-Taste wahrgenommen hat. Auch wenn die Maus ihre Lufthoheit verletzt, indem sich der Cursor über der Schaltfläche befindet, kann sie uns signalisieren, wobei sie auf jede Veränderung der Mausposition innerhalb ihres Bereiches reagiert. Dabei ist es übrigens unerheblich, ob eine Maustaste gedrückt ist oder nicht. Selbst wenn der Anwender sich per Tabulator-Taste anschleicht, erfahren wir dies von der Schaltfläche.

Das hört sich nach einem wahrhaftigen Feuerwerk von Ereignissen an. Doch hier gibt es eine wesentliche Einschränkung, denn wir können entscheiden, welches Ereignis welches Steuerelements wir überhaupt gemeldet bekommen wollen. Denn stellen wir die Schnittstelle nicht zur Verfügung, wird das Steuerelement diese Schnittstelle auch nicht ansteuern. Das Mittel dazu ist eine Ereignisprozedur, die zum Beispiel so aussehen kann:

```
Private Sub cmdCancel_Click()  
    'hier können wir auf den Klick reagieren  
End Sub
```

Im Falle einer Mausbewegung über der Schaltfläche kann die Ereignisprozedur auch diese Ausmaße annehmen:

```
Private Sub cmd_MouseMove(ByVal Button As Integer, _  
                           ByVal Shift As Integer, _  
                           ByVal X As Single, _  
                           ByVal Y As Single)
```

```
'und hier erfahren wir, an welcher Koordinate die Maus  
'sich befindet, ob und welche Maustasten gedrückt sind  
'und ob die Steuerungs- Alt- oder Umschalt-Taste  
'beteiligt sind.
```

```
End Sub
```

Zum Glück müssen wir solche Methodensignaturen nicht von Hand schreiben, denn der Codeeditor nimmt uns auf Verlangen diese Arbeit ab.

Wie geht's nun weiter?

Damit unsere Arbeit mit Dialogen auch einen über die Übung hinausgehenden Nutzen hat, werden wir das Thema Dialoge an zwei konkreten Beispielen festmachen, die auch für Ihre späteren Projekte von Nutzen sind. Das erste Thema wird ein Login-Dialog sein, der eine authentifizierte Anmeldung der Anwender mit Name und Passwort ermöglicht. Dieses Login werden wir auch protokollieren.

In Kapitel 8.4 werden die Themen behandelt, die in den beiden vorangegangenen Beispielen keine Berücksichtigung fanden.

Doch auch in den weiteren Kapiteln werden wir noch auf Dialoge zurückkommen und die eine oder andere Besonderheit erwähnen, die in diesem Zusammenhang zu Tage tritt.

Doch nun frisch ans Werk, welches wir mit dem Einfügen eines neuen Dialogs beginnen wollen.

Sie fügen eine neue UserForm ein, indem Sie entweder im Kontextmenü des Projekt-Explorers den Eintrag EINFÜGEN | USERFORM oder im Einfügen-Menü den Eintrag UserForm auswählen. Danach strahlt Sie an der Stelle, an der sich üblicherweise die Codefenster befinden, eine leere Form in einem Fenster an.

8.2 Login-Dialog

Bevor Sie beginnen, sollten Sie eine halbwegs klare Vorstellung vom späteren Aussehen des zu erzeugenden Dialogs haben. Bei einem Login-Dialog dürfte das nicht allzu schwer sein, denn er besteht aus zwei Steuerelementen zur Eingabe von Name und Passwort. Letzteres darf keinen Klartext anzeigen, sondern darf nur Platzhalterzeichen wie etwa »*« enthalten. Dann sollte die Titelleiste noch den Programmnamen wiedergeben und einen Hinweis auf Sinn und Zweck des Dialogs, nämlich Login, enthalten. Eine OK-Schaltfläche und eine Abbrechen-Schaltfläche sind in den meisten Dialogen ebenfalls Standard und würden unseren Login-Dialog abrunden.

Die beiden folgenden Abbildungen zeigen Alternativen eines Login-Dialogs:

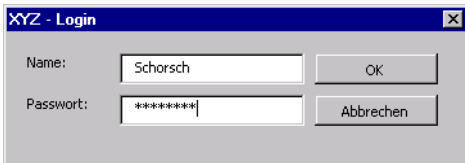


Abbildung 8.2: Login-Dialog, Variante 1

Variante 1 verfügt über alles, was ein Login benötigt, und sieht noch nicht einmal schlecht aus, wenn man ihn mit so manchem vergleicht, was einem da als Anwender in den verschiedensten Programmen zugemutet wird.



Abbildung 8.3: Login-Dialog, Variante 2

Variante 2 sieht schon etwas anders aus.

Hier nun noch eine weitere Variante, auf deren Bedeutung noch im Text eingegangen wird:



Abbildung 8.4: Login-Dialog, Variante 3

Doch wo liegen die Unterschiede zwischen den beiden ersten Varianten?

Nun, es gibt einige offensichtliche Unterschiede:

- ▼ Variante 2 ist etwas größer und auch großzügiger angelegt,
- ▼ die Anordnung der Schaltflächen ist anders,
- ▼ es gibt einen optisch abgetrennten Fußteil mit einem Hinweis und
- ▼ er verfügt über eine Grafik (Icon) oben links.

Aber es gibt auch ein paar Kleinigkeiten, die nicht sofort wahrgenommen werden, oder sagen wir besser, nicht *bewusst* wahrgenommen werden. Und genau darin liegt der Hund begraben, denn wir nehmen unbewusst wohl wahr, ob ein Dialog gut gestaltet ist oder nicht. Ich kann mich an funktional durchaus gelungene Programme erinnern, die mir jedoch von Anfang an dadurch zuwider waren, weil ihre ganze optische Gestaltung einfach eine Zumutung war. Es wird wohl kaum einen Menschen geben, der auf solche Aspekte nicht anspricht, unbewusst und in unterschiedlicher Ausprägung. Daraus folgt:



Eine gut gestaltete Oberfläche öffnet beim Anwender Tür und Tor.

Und nun noch ein Berufsgeheimnis:



Eine gewisse Routine vorausgesetzt, macht das Gestalten einer ansprechenden Oberfläche kaum mehr Arbeit.

Wenden wir uns nun den Besonderheiten von Variante 2 zu, die sich nicht auf den ersten Blick offenbaren:

- ▼ die Ränder des Dialogs sind äquidistant, weisen also denselben Abstand zu den Steuerelementen auf,
- ▼ die Höhe der Textfelder und Schaltflächen ist etwas vergrößert worden, wodurch die Texte jeweils vertikal in der Mitte der Steuerelemente erscheinen und
- ▼ die beiden Bezeichnungsfelder wurden ein wenig nach unten verschoben, so dass ihre Inhalte (zum Beispiel »Name:« und »Schorsch«) auf einer gedachten Linie angeordnet sind.

Wir nehmen uns Variante 2 vor. Einverstanden?

8.2.1 Erzeugen und Anordnen der Steuerelemente

Die Form wurde ja bereits erzeugt. Die Einstellungen der Größe können wir zu diesem Zeitpunkt noch nicht vornehmen. Sie ergibt sich aus Größe und Position der noch hinzuzufügenden Steuerelemente. Beginnen wir nun mit der Einstellung der Eigenschaften, die davon unberührt bleiben.

Als Wert der (Name)-Eigenschaft geben Sie bitte `frmLogin` an. Dieser Name ergibt sich aus dem gebräuchlichen Präfix `frm` für UserForms und dem Zweck des Dialogs, nämlich Login.

Unter **Caption** tragen Sie bitte den Namen Ihrer Anwendung gefolgt von einem » – Login« ein.

Um das Icon zu integrieren, wählen Sie bitte die Eigenschaft **Picture**. In der Wertspalte wird eine Schaltfläche angeboten, die Sie zu einem Dialog führt, in dem Sie die Bilddatei auswählen können. Das zu integrierende Icon heißt `LoginClosed.ico` und befindet sich auf der Buch-CD. Danach wählen Sie bitte in der **PictureAlignment**-Eigenschaft den Wert `fmPictureAlignmentTopLeft` aus.

Im nächsten Schritt werden die benötigten Steuerelemente grob positioniert.

Überprüfen Sie bitte noch Ihre Einstellungen im Register Allgemein des Options-Dialogs. Hier sollten die beiden Optionen *Raster anzeigen* und *Am Raster ausrichten* aktiviert und die Rastereinheiten für Breite und Höhe auf 6 Punkt eingestellt sein.

Sofern der Eintrag *Werkzeugsammlung* im Ansicht-Menü aktiviert ist, hat ein Klick auf unsere Form zur Folge, dass diese Werkzeugsammlung automatisch eingeblendet wird. Tut's das nicht, so müssen Sie diesen Eintrag eben aktivieren. Diese Werkzeugsammlung ist uns in Form der Symbolleiste Steuerelement-Toolbox in Excel schon begegnet. Gegenüber dieser fehlen die Schaltflächen für den *Entwurfsmodus* (wir befinden uns standardmäßig im Entwurfsmodus), das *Eigenschaftsfenster* und *Code anzeigen*, die sich beide unter anderem im Kontextmenü der Form versteckt halten. Im Vergleich zur Steuerelement-Toolbox enthält die Werkzeugsammlung noch drei Steuerelemente, die in Tabellen nicht eingesetzt werden können: *RefEdit*, *Multiseiten* (MultiPage) und *Register* (TabStrip).

Platzieren Sie nun bitte die beiden Bezeichnungsfelder (Labels) für die späteren Texte *Name:* und *Passwort:* links in die Form. Rechts daneben benötigen wir zwei Textfelder (TextBoxes) zur Aufnahme eben dieses Namens und des Passworts. Ganz rechts setzen Sie bitte zwei Schaltflächen (CommandButtons) auf die Form. Als grobe Höhe gilt für TextBoxes und CommandButtons die Rasterweite von drei Einheiten, also 18 Punkt. Für Labels genügen 2 Rasterweiten in der Höhe. Der Dialog müsste nun etwa das Aussehen haben, das in Abbildung 8.3 wiedergegeben ist:

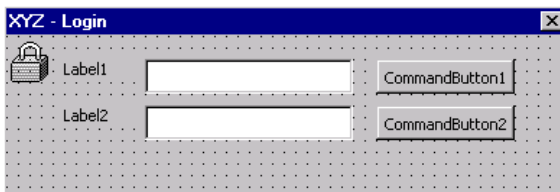


Abbildung 8.5: Login-Dialog, Rohdesign 1. Schritt

Nun können die **(Name)**- und **Caption**-Eigenschaften der einzelnen Steuerelemente eingestellt werden.

Labels

Sofern wir im Programm auf ein Steuerelement nicht zugreifen, so ist eine Namensvergabe nicht zwingend erforderlich. Und auf diese beiden Labels benötigen wir eigentlich keinen Zugriff. Das könnte sich ändern, wenn die spätere Anwendung mehrsprachig werden sollte. Einen universellen Dialog schon einmal prophylaktisch darauf vorzubereiten, kann nicht schaden.

Das obere Label erhält als **(Name)** "lblName" und als **Caption** "Name:" und das untere "lblPassword" und »Passwort:«.

TextBoxes

Als **(Name)** erhalten die beiden "txtName" und "txtPassword".

CommandButtons

Die **(Name)**-Eigenschaft der oberen Schaltfläche wird in »cmdOK« umgetauft, als **Caption** verwenden wir »OK«. Die Eigenschaften der unteren Schaltfläche ändern Sie bitte in »cmdCancel« und »Abbrechen«.

8.2.2 Optische Korrekturen

Abbildung 8.6 zeigt eine 4-fache Vergrößerung der Steuerelemente lblName und txtName, die beide noch an der durch das Raster bestimmten Top-Position erscheinen. Das Bild spricht für sich:

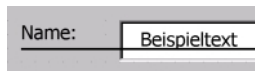


Abbildung 8.6: Label und Textbox ohne Korrektur

Diese Vergrößerung wurde mit der Zoom-Eigenschaft der Userform erzeugt, die offensichtlich für Autoren geschaffen wurde, die dieses Problem demonstrieren wollen ;-).

In Abbildung 8.7 wurde die Top-Eigenschaft des Labels gegenüber der TextBox um 3 Punkte erhöht.

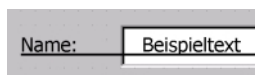


Abbildung 8.7: Label und Textbox mit Korrektur

Nun noch eine Kleinigkeit. Die oberen und unteren Abstände des »Beispieltextes« in der TextBox sind unterschiedlich groß, was durch eine geringfügige Korrektur der Height-Eigenschaft ausgeglichen werden kann.

In Abbildung 8.8, die ebenfalls mit einem Zoom-Wert von 400 erstellt wurde, war ich gezwungen, ein wenig zu mogeln, denn bei einem Zoomwert von 200 reduzierte sich die Fontgröße merklich, wenn die Höhe der TextBox auf 17 Punkt reduziert wurde. Bei 16 Punkt war die Fontgröße wieder normal, um jedoch bei Werten kleiner 16 wieder zu schrumpfen.

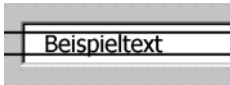


Abbildung 8.8: Höhenkorrektur der TextBox auf 17 Punkt

In der normalen Ansicht mit Zoomwert gleich 100 ist von dieser Absonderlichkeit nichts zu merken.

Im Vergleich zu der Textlage in den Abbildungen 8.4 und 8.5 macht sich der Unterschied bemerkbar. Jetzt, da Sie es wissen, wird Ihnen der Unterschied in den beiden Abbildungen 8.2 und 8.3 bzw. 8.4 auch auffallen.

Es gibt noch einen anderen Ansatz.

In Visual Basic, dem großen Bruder von VBA, präsentieren sich ComboBoxes in einer unveränderlichen Höhe von 315 Twips, was einer Höhe von 15,75 Punkt entspricht. Da die hier verwendete Bibliothek vielen Programmen zu Grunde liegt, haben wir uns an diese Proportionen bereits irgendwie gewöhnt. In gut gestalteten Dialogen werden TextBoxes an diese Einschränkung angepasst und erhalten ebenfalls die Höhe der ihnen äußerlich sehr ähnlichen ComboBoxes, wie Abbildung 8.9 zeigt:

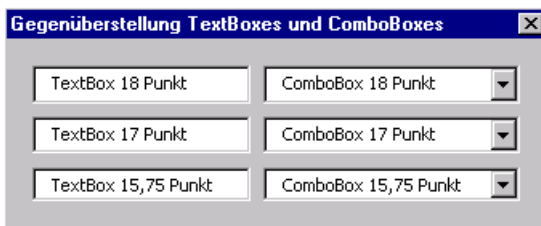


Abbildung 8.9: Gegenüberstellung TextBoxes und ComboBoxes

Die TextBox-Höhe von 15,75 Punkt ist in Variante 3 (Abb. 8.4) zu sehen, wie Sie sich bereits denken konnten.

Gehen wir nun mit dieser Messlatte an die CommandButtons, so ergibt sich dort ein ähnliches Bild:

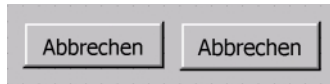


Abbildung 8.10: CommandButtons mit unterschiedlicher Höhe

Die Höhe der rechten Schaltfläche wurde um einen Punkt auf 19 erhöht. Variante 3 in Abb. 8.3 enthält noch die Besonderheit, dass die Top-Eigenschaften der CommandButtons um 1,75 Punkt reduziert wurden, wodurch die Aufschrift dieser CommandButtons auf derselben Höhe stehen wie die des jeweiligen Labels und der TextBox.

Man könnte die hier vorgestellten Korrekturen unter der Kategorie pingelig bis oberpingelig ablegen, wären da nicht zwei Argumente, die dafür sprechen. Die Wirkung ist unbestreitbar, bei einigen Menschen bewusst, bei anderen unbewusst. Und außerdem macht es fast keine Arbeit, denn es gilt, sich in Variante 2 drei Werte und in Variante 3 vier Werte zu merken.

Werte der Variante 2

- ▼ Top-Eigenschaft eines Labels um 3 Punkte gegenüber der Top-Eigenschaft der daneben stehenden TextBox vergrößern,
- ▼ Height-Eigenschaft einer Textbox auf 17 Punkt reduzieren und
- ▼ Height-Eigenschaft eines CommandButtons auf 19 Punkt vergrößern.

Werte der Variante 3

- ▼ Top-Eigenschaft eines Labels um 3 Punkte gegenüber der Top-Eigenschaft der daneben stehenden TextBox vergrößern,
- ▼ Height-Eigenschaft einer Textbox auf 15,75 Punkt reduzieren und
- ▼ Height-Eigenschaft eines CommandButtons auf 19 Punkt vergrößern und Top-Eigenschaft um 1,75 Punkt reduzieren.

Sie haben die Wahl.

8.2.3 Ergänzung des Formularfußes

Im unteren Teil des Login-Dialogs befindet sich noch ein Statustext, der in einem Label erscheint. Eine optische Auflockerung wurde mit einem blauen 3D-Balken erreicht, der den Bedienteil des Formulars vom Statustext trennt.

Statuslabel

Dieses Label erhielt den Namen `lblStatus`. Es ist sinnvoll, sich einen möglichst langen Statustext auszudenken und diesen in die `Caption`-Eigenschaft einzutragen, um später nicht von Texten überrascht zu werden, die nicht mehr vollständig dargestellt werden. Zudem wird die Schriftfarbe verändert, die sich hinter der Eigenschaft `ForeColor` (Vordergrundfarbe) verbirgt. Abbildung 8.11 zeigt einen solchen Farbdialog.

Im Register `System` haben wir Zugriff auf eine ganze Reihe von Systemfarben, die allerdings von der aktuellen Einstellung des Anwenders in der Systemsteuerung abhängen (Einstellungen Anzeige, Register Darstellung). Als Vordergrundfarbe wurde hier die Farbe der aktiven Titelleiste gewählt, wodurch der Hinweischarakter des Statustexts stärker hervortritt. Ein hinreichender Kontrast zur Dialogfeldfarbe ist eigentlich gewährleistet, denn die Einstellungen, bei denen dies schwierig werden könnte, hält ein gesunder Mensch keine zehn Minuten ohne Kopfschmerzen durch.

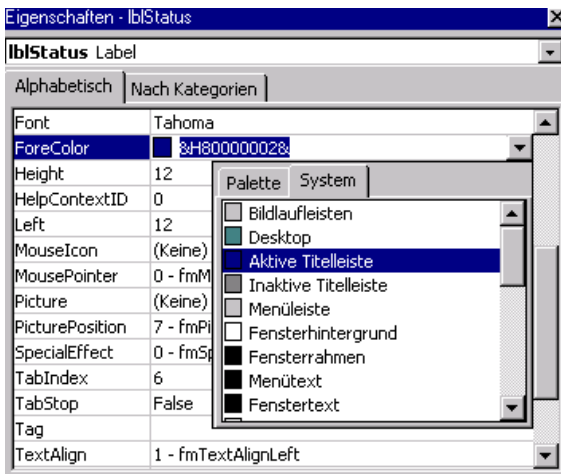


Abbildung 8.11: Farbdialog im Eigenschaftsfenster, Register System

Über das Register `Palette` haben Sie Zugang zu absoluten Farben, wie Abb. 8.12 in zarten Grautönen zeigt. In der Regel reichen die Systemfarben jedoch für unsere Zwecke aus.

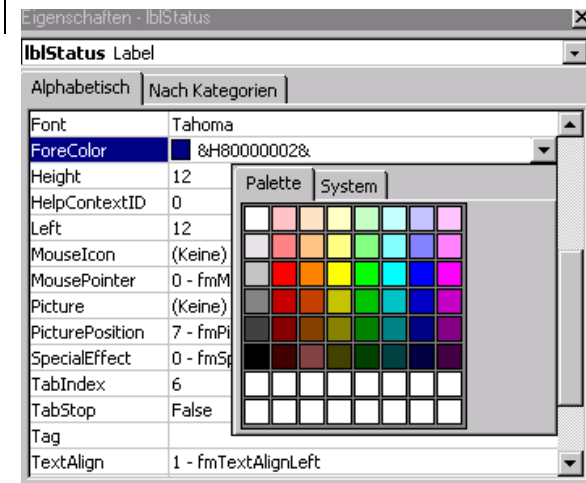


Abbildung 8.12: Farbdialog im Eigenschaftsfenster, Register Palette

Trennlabel

Auch für den blauen Trennbalken wurde ein Label (lblSeparator) verwendet, bei dem ein paar Einstellungen verändert wurden. So wurde als Hintergrundfarbe (**BackColor**) ebenfalls »Aktive Titelleiste« gewählt. Die **Caption** wurde entfernt und als **Style** fmSpecialEffectSunken gewählt.

Nun haben wir alle Designschritte abgeschlossen und können uns der Funktion des Dialogs und der Codierung widmen.

8.2.4 Aktivierreihenfolge

In der Aktivierreihenfolge verbirgt sich die Reihenfolge, in der die Steuerelemente per Tabulatortaste angesteuert werden können. Implizit steckt darin auch noch die Information, welches Steuerelement beim Start des Dialogs diesen so genannten Fokus erhält. In unserem Beispiel bietet sich die Aktivierreihenfolge txtName, txtPassword, cmdOK und schließlich cmdCancel an.

Im Menü Ansicht produziert der Befehl Aktivierreihenfolge den in Abb. 8.13 gezeigten Dialog.

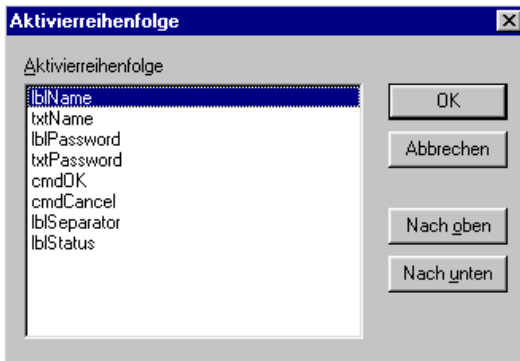


Abbildung 8.13: Dialog Aktivierreihenfolge

Wem das zu einfach ist, der kann auch die beiden Eigenschaften `TabStop` und `TabIndex` verwenden, über die das Tabulatorverhalten bestimmt werden kann. Mit der logischen Eigenschaft `TabStop` legen Sie fest, ob ein Steuerelement überhaupt per Tabulator gewählt werden kann. Außer beim Image-Steuerelement, das über diese Eigenschaft nicht verfügt, und dem Label, das sich davon nicht beeindrucken lässt, funktioniert das überall. Mit dem formularweiten und mit 0 beginnenden `TabIndex` legen Sie die Position des jeweiligen Steuerelements in der Aktivierreihenfolge fest. Nett ist, dass man zur Veränderung eines solchen Wertes den momentan belegten Zielindex nicht freiräumen muss. Haben die Elemente A, B, C und D die Werte 0, 1, 2 und 3, so können Sie bei Element D den Wert 1 eintragen. Dies hat zur Folge, dass sich die Nummern der Elemente B (das ehemalige 1) und C in 2 und 3 ändern.

8.2.5 Funktionale Aspekte des Login-Dialogs

Beginnen wir mit einer scheinbaren Selbstverständlichkeit.

Verstecken der zu schützenden Tabellen

Wird die Datei geöffnet, so muss als Erstes der Login-Dialog auf dem Bildschirm erscheinen. Hat sich der Anwender mit einem registrierten Namen und dem dazugehörigen Passwort authentifiziert, so darf er in die Anwendung. Hat er nach drei Versuchen (oder was auch immer wir ihm eingestehen) dieses Ziel nicht erreicht, so wird die Datei wieder geschlossen.

Was aber, wenn der Anwender die Datei mit deaktiviertem Code öffnet, indem er beispielsweise beim Öffnen die Umschalt-Taste gedrückt hält? Dann kommt per Definition kein Login-Dialog und er ist drin.

Wir müssen nun definieren, was das Schützenswerte an der Arbeitsmappe ist. Vermutlich sind es ein paar Tabellen. Und die darf der Anwender na-

türlich nicht zu sehen bekommen, wenn er sich am Login vorbeigequetscht hat. Das realisieren wir am besten dadurch, dass wir die eigentlich zu schützenden Tabellen verstecken.

Nehmen wir an, die zu schützende Arbeitsmappe bestünde aus zwei Datentabellen und natürlich unserer User-Tabelle. Dass Tabellen über die Menübefehlskette **FORMAT | BLATT | AUSBLENDEN** zum Verschwinden gebracht werden können, wissen Sie. Aber Sie wissen auch, dass der Anwender über **FORMAT | BLATT | EINBLENDEN** an die Liste der solchermaßen ausgeblendeten Tabellen gelangt:

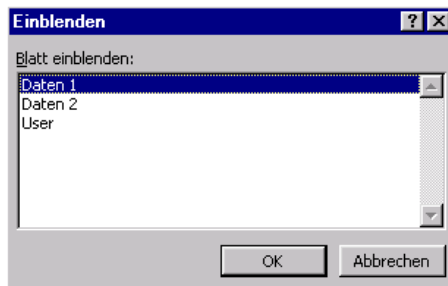


Abbildung 8.14: Dialog Einblenden in Excel

Die Entwicklungsumgebung bietet uns hier einen Weg Tabellen so zu verstecken, dass sie in dieser Liste nicht erscheinen und somit auch keine Begehrlichkeiten wecken. In Abbildung 8.14 ist das Eigenschaftsfenster der Tabelle **User** zu sehen. Die markierte Eigenschaft **Visible** verfügt über die Werte

- ▼ -1 **xlSheetVisible** (eingebledet)
- ▼ 0 **xlSheetHidden** (ausgebledet)
- ▼ 2 **xlSheetVeryHidden** (versteckt)

Die etwas merkwürdige Konstantenbelegung erklärt sich daraus, dass die **Visible**-Eigenschaft in Excel 5 und '95 den Datentype **Boolean** aufwies, also -1 für **True** und 0 für **False**. Da der **Boolean** im Prinzip auch nichts anderes als ein Integer ist, musste lediglich ein dritter Wert ergänzt werden.

Wie auch immer, wir können die beiden angenommenen Datentabellen und die Tabelle mit den Namen und Passwörtern verstecken, indem wir die **Visible**-Eigenschaft auf **xlSheetVeryHidden** setzen und die beiden Zugänglichkeiten in erfolgreicher Anmeldung auf **xlSheetVisible** ändern.

Beim Schließen der Arbeitsmappe müssen natürlich alle auszublendenden Tabellen auf `xlSheetVeryHidden` gesetzt werden.

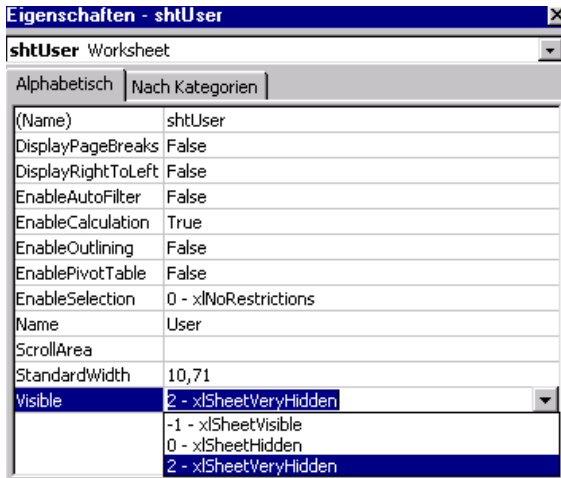


Abbildung 8.15: Visible-Eigenschaft einer Tabelle

Um diese Tabellen verstecken zu können, benötigen wir aber eine Tabelle, die sichtbar sein muss. Diese Tabelle heißt bei mir in der Regel Main. Abb. 8.16 zeigt nun alle bislang verwendeten Objekte im Projekt-Explorer.

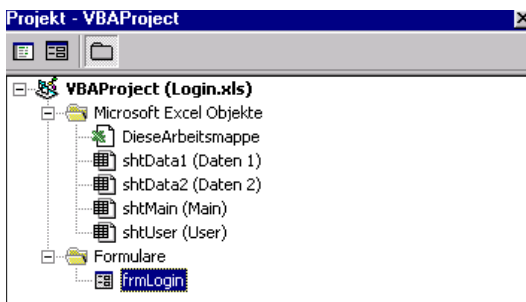


Abbildung 8.16: Projekt-Explorer mit allen Tabellen

Arbeitsweise des Login-Dialogs

Der Anwender gibt in der Regel seinen Namen und anschließend das Passwort an. Das Windows API ist jedoch willens uns den Anmeldenamen des Anwenders mitzuteilen, sofern man mit der `GetUserName`-Funktion (siehe Kapitel 7 – Sprachelemente, die zweite) höflich danach fragt. Zwar ist eine Anmeldung unter Windows 9x nicht zwingend, kann aber durch die in Unternehmen übliche Netzwerkanmeldung vorausgesetzt werden.

Somit können wir den Anmeldenamen über das API ermitteln, in die TextBox `txtName` eintragen und den Cursor in die Passwort-TextBox positionieren.

Der Anwender gibt also noch sein Passwort an, und wir überprüfen beide anhand der in der Tabelle `User` hinterlegten Daten, sobald die OK-Schaltfläche einen Mucks von sich gibt. Existiert der Anmeldenamen nicht in der Usertabelle, so wird er auch nicht in die `txtName` eingetragen. Die TextBox sollte aber dann den Fokus erhalten.

8.2.6 Codierung des Login-Dialogs

Die Diskussion der funktionalen Aspekte unseres doch im Grunde recht einfachen Anmeldedialogs zeigt, dass dieser auf den ersten Blick recht banale Dialog doch noch die eine oder andere Schwierigkeit birgt. So verzichten wir an dieser Stelle auf die Anbindung des Dialogs an die Datenquelle, die zur Verifizierung der Anmeldedaten benötigt wird. Das ist eine typische Aufgabe für eine eigene Klasse. Oder besser gesagt zwei, denn wir sollten für den Zugriff auf in der Arbeitsmappe gespeicherte Anwenderdaten eine Klasse haben, gleichzeitig aber darauf gefasst sein, uns an eine Datenbank anbinden zu müssen. Natürlich werden beide Klassen eine identische Schnittstelle (Methodensignatur) anbieten. Dadurch läuft der Login wahlweise mit beiden Klassen und die Entscheidung der Datenanbindung hängt nur noch damit zusammen, welche Klasse importiert wird. Doch diesen Punkten werden wir uns in den Kapiteln 10 und 12 wieder zuwenden, wo auch die beiden dazugehörigen Klassen gebaut werden.

In der Zwischenzeit werden wir zwei private Variablen verwenden, die den einzig zugelassenen Anmeldenamen mit dem entsprechenden Passwort bereits enthalten. Doch nun zurück zu unserem Thema.

Wir vereinbarten, beim Start des Dialogs den Anmeldenamen zu ermitteln. Woher wissen wir aber, dass der Dialog gestartet wird?

Nun, die `UserForm` hält eine ganze Reihe von Ereignissen für uns bereit, die uns über diverse Vorgänge bei der Form informieren. Von den 22 Ereignissen, die eine Form bedienen kann, interessieren uns in aller Regel die folgenden vier:

Ereignis	Anlass
Activate	Form erhält Fokus
Initialize	Initialisieren der Form
QueryClose	Schließen der Form
Terminate	Entfernen der Form aus dem Speicher

Für unsere Zwecke würde das `Activate`-Ereignis zwar auch genügen, aber das korrekte Ereignis lautet `Initialize`.

Wenn Sie einen Doppelklick auf der Form machen, wechselt die Entwicklungsumgebung in den Codeeditor und legt dort die Prozedur des Standardereignisses `Click(?)` an. Um die `Initialize`-Ereignisprozedur zu erhalten, wählen Sie bitte in der rechten `ComboBox` des Editors den Eintrag `"Initialize"`.

Dieses `Initialize`-Ereignis müssen wir auch dazu nutzen, die beiden (Krücken-)Variablen zuzuweisen:

```
Private Sub UserForm_Initialize()  
    strName = "Schorsch"  
    strPassword = "Ich bins"  
End Sub
```

Bitte vergessen Sie nicht, `strName` und `strPassword` im Modulkopf als `private` Variablen zu erzeugen.

Alle Steuerelemente, die den Fokus entgegennehmen können, verfügen über ein `Enter`- und ein `Exit`-Ereignis. `Enter` meldet sich zu Wort, wenn das Steuerelement den Fokus erhält, und `Enter`, wenn es diesen wieder an ein anderes Steuerelement verliert. Unsere Statusausgabe wiederum soll eben einen jeweils passenden Text ausgeben, wenn die `TextBoxes` `txtName` und `txtPassword` aktiv sind, und diese Zeitspanne liegt bei den `TextBoxes` zwischen den `Enter`- und `Exit`-Ereignissen.

Wählen Sie also in der Objekt-`ComboBox` (die Linke) des Editors `txtName` und in der Prozedur-`ComboBox` das `Enter`-Ereignis und gleich anschließend `Exit`. Die beiden Ereignisprozeduren müssen wir nun um die beiden Statusausgaben erweitern:

```
Private Sub txtName_Enter()  
    lblStatus.Caption = "Bitte geben Sie Ihren Namen an."  
End Sub  
  
Private Sub txtName_Exit(ByVal Cancel As _  
                        MSForms.ReturnBoolean)  
    lblStatus.Caption = ""  
End Sub
```

Da wir nicht wissen, an wen die `TextBox` den Fokus verloren hat, müssen wir die Statusausgabe wieder zurücksetzen. Ein solches Prozedurpaar muss auch für die `txtPassword` angelegt werden:

```

Private Sub txtPassword_Enter()
    lblStatus.Caption = "Bitte geben Sie Ihr Passwort an."
End Sub

Private Sub txtPassword_Exit(ByVal Cancel As _
                                MSForms.ReturnBoolean)
    lblStatus.Caption = ""
End Sub

```

Das Cancel-Argument der Exit-Prozeduren ermöglicht übrigens einen Abbruch des Prozesses, indem die Variable einfach auf True gesetzt wird. Dieses auf den ersten Blick ganz nützliche Ereignis birgt jedoch so seine Probleme. Zwar können wir damit beispielsweise verhindern, dass ein ungültiger Wert in einer TextBox zurückbleibt. Andererseits jedoch müssen wir jederzeit einen Ausstieg per Abbrechen-Schaltfläche ermöglichen. Doch das Exit der TextBox kommt *vor* dem Enter der Schaltfläche. Bleibt zu hoffen, dass Microsoft eine neue UserForm spendiert, denn die ist dafür verantwortlich, dass keine Validierung möglich ist. Validierung heißt, man legt fest, welches Steuerelement eine Validierung bei anderen auslöst, und nur dann kommt ein abbrechbares Validate-Ereignis. Man kann der Abbrechen-Schaltfläche einfach per Eigenschaft auftragen, dieses Validate-Ereignis *nicht* auszulösen, ohne auf die Validierung bei den anderen Steuerelementen verzichten zu müssen.

Was bleibt uns noch? Das Wichtigste. Ein Klick auf die OK-Schaltfläche sollte uns nun veranlassen Name und Passwort zu überprüfen und entsprechend zu reagieren:

```

Private Sub cmdOK_Click()
    If txtName.Text = strName Then
        If txtPassword.Text = strPassword Then
            'alles OK
        Else
            'Name OK, aber Passwort falsch
        End If
    Else
        'Name unbekannt
    End If
End Sub

```

Nun füllen wir die Platzhalter noch mit Leben.

Alles OK heißt, wir schließen den Dialog:

```
Me.Hide
```

Frage: Könnten wir die Datenblätter der Arbeitsmappe hier schon einblenden?

Wir könnten, sollten aber nicht, denn dadurch würden wir den übergreifenden Charakter des Dialogs aufs Spiel setzen und müssten ihn in jedem Projekt anpassen. Dies ist bei der den Dialog aufrufenden Stelle besser aufgehoben (siehe Kapitel 8.1.7).

Ist der Name des Anwenders registriert, aber das Passwort falsch, dann geben wir eine entsprechende MessageBox aus, löschen das Passwort und setzen den Fokus wieder in txtPassword:

```
MsgBox "Das angegebene Passwort ist falsch.", _  
    vbExclamation, Me.Caption  
txtPassword.Text = ""  
txtPassword.SetFocus
```

Hier bietet sich natürlich an, die Caption des Login-Dialogs als Title-Argument der MessageBox auszugeben. Wie Sie wissen, kann das betreffende Objekt eines gebundenen Modulblatts mit Me (Ich) referenziert werden.

Ergab die Überprüfung des Namens, dass dieser in der Benutzerverwaltung unbekannt ist, so müssen wir das kundtun und den Fokus wieder dort hinsetzen. Doch diesmal löschen wir den Inhalt der TextBox nicht, sondern markieren ihn vollständig. Zu diesem Zweck wird durch 0 in der SelStart-Eigenschaft der Anfang der Markierung vor das erste Zeichen gesetzt und als SelLength Länge die des aktuellen Inhalts der TextBox angegeben.

```
MsgBox "Der Name " & txtName.Text & " ist unbekannt.", _  
    vbExclamation, Me.Caption  
txtName.SelStart = 0  
txtName.SelLength = Len(txtName.Text)  
txtName.SetFocus  
Beep
```

Sieht gut aus, nicht wahr? Sagen wir einmal fast gut. Als Fleißaufgabe könnten wir zumindest die txtName noch daraufhin untersuchen, ob sie möglicherweise leer ist, und einen entsprechenden Hinweis ausgeben, dass der Name angegeben werden muss ...

Im Übrigen sind wir mit dem Dialog vorerst fertig. Nun müssen wir uns über den Aufruf Gedanken machen und darüber, wie wir darauf reagieren, ob der Login erfolgreich verlief oder ob er eben nicht erfolgreich war. Danach kommen wir wieder zum Code zurück.

8.2.7 Aufruf und Auswertung des Logins

Der Login muss starten, sobald die Arbeitsmappe geöffnet wird. Dies stellt insofern kein Problem dar, denn die Arbeitsmappe bietet uns ja ein Open-Ereignis an, welches im Prinzip für solche Fälle gedacht ist:

```
Private Sub Workbook_Open()
    frmLogin.Show
    If frmLogin.Successful = False Then ';-)
        ThisWorkbook.Close
    End If
End Sub
```

Schön wär's, wenn die Form eine `Successful`-Eigenschaft hätte, die wir so einfach auswerten könnten. Natürlich könnten wir eine öffentliche Variable in einem Modul schaffen und diese im `frmLogin` setzen und in `Workbook_Open` auswerten. Das ließe vielleicht eine Spur Eleganz vermissen, ginge aber. Aber dagegen spricht, dass unsere `frmLogin` von einer äußeren Variablen abhängig würde.

Es gibt aber zum Glück einen anderen Weg, der unserer angenommenen `Successful`-Eigenschaft eigentlich sehr nahe kommt: der Tag (mit Etikett übersetzbar). Man kann einem Steuerelement oder einer UserForm gewissermaßen ein Etikett ankleben und dieses an anderer Stelle auslesen.

Dieses Tag ist vom Datentyp `String` und kann somit auch Klartext mit auf den Weg bekommen. Also fügen wir vor dem Ausblenden der Form noch eine Zuweisung an den Tag ein:

```
shtData1.Visible = xlSheetVisible
shtData1.Visible = xlSheetVisible
Me.Tag = "OK"
Me.Hide
```

Die Ereignisprozedur der Abbrechen-Schaltfläche sollten wir auch integrieren:

```
Private Sub cmdCancel_Click()
    Me.Tag = "Cancel"
    Me.Hide
End Sub
```

Nun zurück zur `Workbook_Open`, in der wir den Tag der UserForm `frmLogin` auswerten. Ist dieser »OK«, so blenden wir die beiden stellvertretenden Datentabellen ein. Andernfalls wird die Arbeitsmappe wieder geschlossen:

```

Private Sub Workbook_Open()
    frmLogin.Show
    If frmLogin.Tag = "OK" Then
        shtData1.Visible = xlSheetVisible
        shtData2.Visible = xlSheetVisible
    Else
        ThisWorkbook.Close
    End If
End Sub

```

Nun müssen wir beim Schließen der Arbeitsmappe noch dafür Sorge tragen, dass die Datentabellen ausgeblendet werden:

```

Private Sub Workbook_BeforeClose(Cancel As Boolean)
    shtData1.Visible = xlSheetVeryHidden
    shtData2.Visible = xlSheetVeryHidden
End Sub

```

Auf der Buch-CD befindet sich die Datei Login.xls, die den nachfolgend besprochenen Login-Dialog beinhaltet. Das Login können Sie mit dem Namen KlausP und dem Passwort juhu überwinden.

In Kapitel 10 werden wir uns noch näher mit Objekten und Klassen auseinander setzen und noch die eine oder andere Veränderung vornehmen, denn perfekt ist unser Dialog noch nicht und selbst der Aufruf enthält noch einen Mangel, der etwas mit Lebenden und Toten zu tun hat. Doch unser Thema in diesem Kapitel sind Dialoge und als Dialog können wir ihn so lassen.

8.3 Gemeinsamkeiten der Steuerelemente

Die meisten Steuerelemente unterstützen die Caption-Eigenschaft. Zwar erscheint der dort eingegebene Text bei einer UserForm in fester Schrift in der Titelleiste und bei einer CheckBox rechts des Häkchens, aber in beiden Fällen ist es die Beschriftung des Steuerelements. Ein schönes Beispiel für Polymorphismus, nicht wahr.

So gibt es eine Reihe von Eigenschaften und Methoden, die mehreren Steuerelementen anhaften und es somit verdienen, einleitend erwähnt zu werden.

8.3.1 Eigenschaften

BackColor-Eigenschaft

Hiermit legen Sie die Hintergrundfarbe des Steuerelements fest. Sie haben die Möglichkeit per Dialog auf Systemfarben (siehe Abb. 8.11) oder solche

aus der Farbpalette (siehe Abb. 8.12) zurückzugreifen. Aber Sie können auch Ihre eigenen Farben verwenden, indem Sie einen Long-Wert der Eigenschaft zuweisen. Die drei folgenden Anweisungen färben die aktuelle Form in einem knalligen Rot ein:

```
Me.BackColor = 255
Me.BackColor = &HFF
Me.BackColor = RGB(255, 0, 0)
```

BorderColor-Eigenschaft

Wenn die BorderStyle-Eigenschaft auf 1 – fmBorderStyleSingle eingestellt ist, können Sie über die BorderColor die Farbe des Rahmens einstellen. Ist BorderStyle 0 – fmBorderStyleNone, so hat die BorderColor keine Auswirkungen. Die Wertzuweisung erfolgt wie bei der BackColor-Eigenschaft.

BorderStyle-Eigenschaft

Die Steuerelemente TextBox, ComboBox, ListBox, Frame, Image und Label lassen sich einen Rahmen zuweisen. Die Werte sind

- ▼ 0 – fmBorderStyleNone (kein Rahmen)
- ▼ 1 – fmBorderStyleSingle (einfacher Rahmen)

TextBox, ComboBox, ListBox und Frame verlieren jedoch ihr 3D-Aussehen (siehe SpecialEffect-Eigenschaft), Image und Label nicht, da sie über keine darstellbaren SpecialEffects verfügen.

ControlSource-Eigenschaft

TextBox, ComboBox, ListBox, OptionButton, CheckBox, SpinButton und ScrollBar lassen sich über die ControlSource-Eigenschaft an eine Tabellenzeile binden, wo sie dann prompt ihren aktuellen Inhalt hinterlassen:

Steuerelemente	Eigenschaft	Aktualisierung
TextBox	Text	bei Fokusverlust
ComboBox	Text	bei Fokusverlust
ListBox	Text	sofort, sofern kein MultiSelect eingestellt ist
OptionButton	Value	sofort
CheckBox	Value	sofort
SpinButton	Value	bei Fokusverlust
ScrollBar	Value	bei Fokusverlust

Die `ControlSource`-Eigenschaft ist ebenso wie alle anderen Arten von datengebundenen Steuerelementen Geschmackssache. Ich mag sie nicht, weil sich der Anwender unter Umgehung des Programms mit Zellen in Verbindung setzen und dort Blödsinn hinterlassen kann. Zwar spricht das `Change`-Ereignis des `Worksheet`-Objekts an, doch das vermag mich nicht zu trösten.

ControlTipText-Eigenschaft

Die hier eingetragenen Texte erscheinen in Form eines ToolTips (kleines, gelbes Fenster), sobald der Cursor sich über dem Steuerelement befindet:

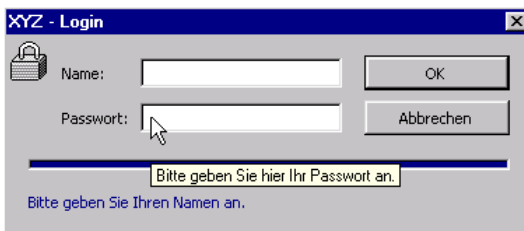


Abbildung 8.17: Login-Dialog mit Tooltip

Dass für einen Sachverhalt in VB und VBA zwei verschiedene Begriffe gewählt wurden, deutet darauf hin, dass die beiden Entwicklergruppen bei Microsoft in verschiedenen Kantinen essen.

Enabled-Eigenschaft

Mit dieser Eigenschaft können Sie festlegen, ob das betreffende Steuerelement auf Anwenderaktionen reagieren kann. Steht diese Eigenschaft auf `False`, so erscheinen Texte der Steuerelemente in einem Grauton. Interessant ist, dass dann das betreffende Steuerelement für Mausklicks transparent wirkt. Somit werden Mauseaktionen an das unmittelbar darunter liegende Steuerelement weiter gemeldet, also in der Regel an die UserForm.

Welchen Sinn hat diese Eigenschaft bei einem Label oder einem Image?

Font-Eigenschaft

Siehe hierzu 8.3.2.

ForeColor-Eigenschaft

Bei den Steuerelementen, die Texte darstellen, beeinflusst die `ForeColor`-Eigenschaft die Textfarbe, bei den anderen die Farbe der Bedienelemente.

Zur Wertzuweisung siehe `BackColor`-Eigenschaft

Height-Eigenschaft

Die `Height`-Eigenschaft legt die Höhe des Steuerelements in der Einheit Punkt fest. Ein Punkt entspricht einem Zweiundsiebzigstel Zoll, also etwa 0,3 mm.

HelpContextID-Eigenschaft

Sofern Ihre Anwendung über eine eigene Hilfedatei verfügt, können Sie über die in der `HelpContextID` einzugebende Zahl eine Verbindung zu einem Verweis in der Hilfedatei herstellen.

HideSelection-Eigenschaft

Diese für `TextBox` und `ComboBox` verfügbare Eigenschaft legt fest, ob die Markierung oder Selektion, die der Anwender vorgenommen hat, auch dann sichtbar bleibt, wenn das betreffende Steuerelement den Fokus verloren hat.

Left-Eigenschaft

Mit der `Left`-Eigenschaft legen Sie die Position des Steuerelements innerhalb des Containers fest. Als Container kommen die `UserForm` und das `Frame`-Steuerelement in Frage.

Locked-Eigenschaft

Hiermit legen Sie fest, ob der Anwender das Steuerelement bedienen kann. In der Regel ist es aber besser, die `Enabled`-Eigenschaft zu verwenden, wenn ein Steuerelement mal nicht bedienbar sein soll, denn der gleichzeitig damit erzielte Effekt des Abblendens signalisiert diesen Zustand auch optisch, was jeder Anwender wohl entsprechend interpretieren müsste.

MouseIcon-Eigenschaft

Sofern die `MousePointer`-Eigenschaft auf `99 – fmMousePointerCustom` eingestellt ist, können Sie hier eine Graphikdatei angeben, die als Mousepointer angezeigt werden soll, sobald die Maus über dem Steuerelement auftaucht. Für diese Zwecke sind `Icon`-Dateien (*.ico) oder `Cursor`-Dateien (*.cur) am besten geeignet. Die Zuweisung zur Laufzeit muss mittels der `LoadPicture`-Funktion erfolgen:

```
cmdOK.MouseIcon = LoadPicture("C:\Test\H_Point.cur")
```

MousePointer-Eigenschaft

Über diese Eigenschaft haben Sie über die anzugebende Konstante Zugang zu 14 voreingestellten Mauszeigern, von denen aber nur der Standardzeiger und die Sanduhr wirklich erforderlich sind. Die Werte und Konstanten

lauten 0 – `fmMousePointerDefault` für den Standardzeiger und 11 – `fmMousePointerHourglass` für die Sanduhr.

Picture-Eigenschaft

Label, CheckBox, OptionButton, ToggleButton, CommandButton, Frame, und Image können in ihrem Innern Bilder darstellen, die über diese Eigenschaft zugewiesen werden können. Zur Laufzeit müssen Sie wie bei der MouseIcon-Eigenschaft die LoadPicture-Funktion zu Rate ziehen.

PictureAlignment-Eigenschaft

PictureAlignment bestimmt, wo das unter der Picture-Eigenschaft spezifizierte Bild dargestellt wird. Es stehen die folgenden Konstanten zur Verfügung:

0 – <code>fmPictureAlignmentTopLeft</code>	obere linke Ecke
1 – <code>fmPictureAlignmentTopRight</code>	obere rechte Ecke
2 – <code>fmPictureAlignmentCenter</code>	Mitte
3 – <code>fmPictureAlignmentBottomLeft</code>	untere linke Ecke
4 – <code>fmPictureAlignmentBottomRight</code>	untere rechte Ecke

SpecialEffect-Eigenschaft

Diese Eigenschaft beeinflusst die Darstellung der Steuerelemente Label, TextBox, ComboBox, ListBox, CheckBox, OptionButton, Frame und Image.

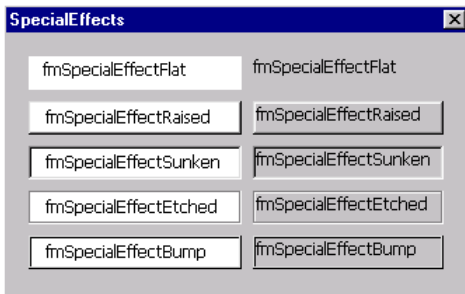


Abbildung 8.18: SpecialEffects-Beispiele anhand eines Labels und einer TextBox

Alle SpecialEffects ungleich Flat setzen die BorderStyle-Eigenschaft zurück.

TabIndex-Eigenschaft

Der formularweite `TabIndex` legt beginnend mit 0 die Reihenfolge aller per Tabulator-Taste erreichbaren Steuerelemente fest, deren `TabStop`-Eigenschaft auf `True` sitzt.

TabStop-Eigenschaft

Die `TabStop`-Eigenschaft legt fest, ob ein Steuerelement per Tabulator-Taste angesteuert werden kann.

Tag-Eigenschaft

Jedem Steuerelement kann ein Text zugewiesen werden, der nach außen hin nicht in Erscheinung tritt. Dieses `Tag` wirkt wie eine innerhalb des Modulkontextes deklarierte private Variable und kann dazu verwendet werden, eine Information von außerhalb des Modulkontexts entgegenzunehmen oder dahin zu transportieren:

Code in `shtMain`:

```
Private Sub cmdDialog_Click()  
    'Instanz der Form erzeugen  
    Dim fDialog As frmDialog  
    Set fDialog = New frmDialog  
    'Hinweis ankleben, dass Dialog aus shtMain aufgerufen  
    fDialog.Tag = "Aufruf aus shtMain"  
    fDialog.Show  
    'Auswerten, ob Dialog per OK-Taste geschlossen wurde  
    If fDialog.Tag = "OK"  
        ...  
    End If  
    UnLoad fDialog  
End Sub
```

Code in `frmDialog`:

```
Private Sub cmdOK_Click()  
    Me.Tag = "OK"  
    Me.Hide  
End Sub  
  
Private Sub UserForm_Initialize()  
    If Me.Tag = "Aufruf aus shtMain" Then  
        ...  
    End If  
End Sub
```

Top-Eigenschaft

Die Top-Eigenschaft bestimmt die obere Position des Steuerelements vom inneren Rand des Containers aus gemessen. Befindet sich das Steuerelement innerhalb eines Frames, so ist dieser Frame der Container. Andernfalls ist es eben die UserForm.

Visible-Eigenschaft

Hiermit legen Sie fest, ob das Steuerelement sichtbar ist.

Width-Eigenschaft

Width definiert die Breite des Steuerelements in Punkt.

8.3.2 Font-Objekt

Hinter dem Eintrag Font im Eigenschaftsfenster verbirgt sich hier keine Eigenschaft im klassischen Sinne, sondern ein eigenes Objekt. Dieses Font-Objekt beinhaltet alle Einstellungen einer Textformatierung.

Die aktuellen Einstellungen des Font-Objekts der UserForm werden automatisch allen Steuerelementen vererbt, die ebenfalls über ein eigenes Font-Objekt verfügen. Ab dem Moment des Einfügens bleiben diese Werte jedoch statisch. Eine Änderung des Font-Objekts der UserForm wirkt also *nicht* auf Steuerelemente, die zum Zeitpunkt der Änderung bereits in der Form Platz gefunden haben, sondern nur auf zukünftig einzufügende.

Die Datei Dialog.xls auf der beiliegenden CD enthält den folgenden Dialog (Abb. 8.19), der die Manipulationsvielfalt des Font-Objekts zeigt.

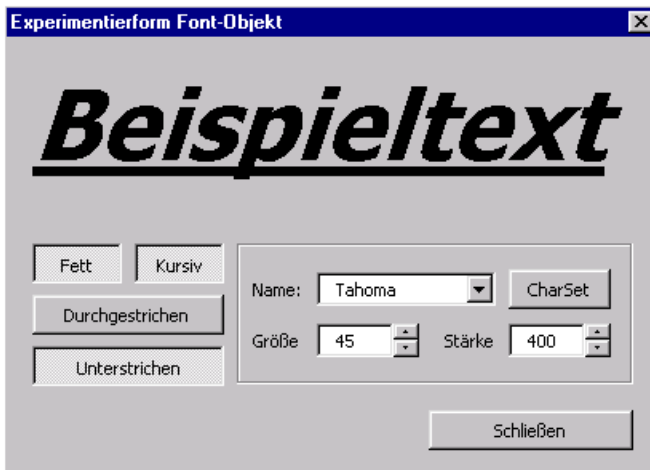


Abbildung 8.19: Experimentierform Font-Objekt

Nun zu den Eigenschaften des Font-Objekts,

Bold-Eigenschaft

Die folgende Anweisung formatiert die Caption fett:

```
lblStatus.Font.Bold = True
```

Diese Zeile wiederum invertiert die aktuelle Einstellung:

```
lblStatus.Font.Bold = Not lblStatus.Font.Bold
```

CharSet-Eigenschaft

Diese Eigenschaft kann nur per Code gesetzt werden.

Hier sind für unsere Zwecke eigentlich nur die beiden Werte 1 für Standard Windows-Zeichensatz und 2 für den Symbol-Zeichensatz möglich laut MSDN, aber es müsste heißen *die symbolischen Zeichensätze*, denn Symbol ist ebenso betroffen wie die Wingdings-Zeichensätze.

```
lblStatus.Font.CharSet = 2
```

zeigt den betreffenden Text im Symbol-Zeichensatz. Aber die folgenden Zeilen führen zur Verwendung des Wingdings-Zeichensatzes:

```
lblStatus.Font.Name = "Wingdings"  
Label1.Font.Charset = 2
```

Hier eine Kuriosität am Rande. Die folgende MessageBox zeigt 2 an:

```
lblStatus.Font.Name = "Symbol"  
Label1.Font.Charset = 2  
MsgBox Label1.Font.Charset
```

Und diese?

```
Label1.Font.Charset = 2  
lblStatus.Font.Name = "Symbol"  
MsgBox Label1.Font.Charset
```



Sie zeigt eine 1, da offensichtlich eine Zuweisung an die Name-Eigenschaft die CharSet-Eigenschaft zurück auf 1 setzt.

Italic-Eigenschaft

Die folgende Anweisung formatiert die Caption kursiv:

```
lblStatus.Font.Italic = True
```

Diese Zeile wiederum invertiert die aktuelle Einstellung:

```
lblStatus.Font.Italic = Not lblStatus.Font.Italic
```

Name-Eigenschaft

Mit dieser Eigenschaft können wir den Font-Namen angeben. Interessanterweise erzeugt die Zuweisung eines nicht existierenden Fonts keinen Fehler. Die Font-Eigenschaft gibt auf Befragen sogar den nicht existierenden Font-Namen zurück, wobei sie allerdings auf den Standard-Font wechselt.

```
lblSample.Font.Name = "Arial"
```

Size

Die Size-Eigenschaft legt die Größe des darzustellenden Textes inklusive Ober- und Unterlänge in Punkt fest. Neben den im Dialog angebotenen Werten ist beginnend mit 0 die Skala nach oben offen. Die Wirkung der Schrittwerte ist fontabhängig.

Strikethrough

Diese logische Eigenschaft legt fest, ob der Text durchgestrichen dargestellt werden soll.

Underline

Mit dieser ebenfalls logischen Eigenschaft bestimmen Sie, ob der Text unterstrichen dargestellt werden soll.

Weight

Die Weight bestimmt in der Regel die Strichstärke einer Text- oder Linien-darstellung. Die meisten Schriftarten zeigen von 0 bis 550 eine kleine Variante und von 560 bis 1000 eine größere Variante des gewählten Schrift-typs an, unabhängig von der Bildschirmauflösung.

8.3.3 Methoden

Es gibt nur wenige übergreifende Methoden:

Move-Methode

Die Move-Methode existiert in zwei Varianten. Die folgende ist die, welche auf ein einzelnes Steuerelement angewendet werden kann:

Move(<i>Left, Top, Width, Height, Layout</i>)	
Left	(optional) neue linke Position innerhalb des Containers
Top	(optional) neue obere Position innerhalb des Containers
Width	(optional) neue Breite

Move(*Left, Top, Width, Height, Layout*)

Height	(optional) neue Höhe
Layout	(optional) wenn True, wird das Layout-Ereignis des Containers ausgelöst

Wie der Name schon sagt, kann man mit Move Steuerelemente in der Gegend herumschieben.

Die Dialoge.xls auf der beiliegenden CD zeigt, wozu man die Move-Methode verwenden kann: zum Hasen jagen (siehe Abb.8.20). Ich dachte schon, mir fällt hierzu überhaupt nichts Sinnvolles ein. So kann man sich täuschen. Dieses Hasenspiel bietet eine Menge Anschauungsmaterial zu einigen nicht alltäglichen Aspekten von Dialogen. Ein paar Worte zum Ablauf.

Im Spielmodus muss man versuchen dem Hasen ein paar Mausklicks zu verpassen. Doch er weiß sich zu wehren und ergreift kurzerhand die Flucht, sobald ihm die Maus zu nahe kommt, wobei er die der Mausannäherung entgegengesetzte Richtung wählt. Die Sprungweite lässt sich über die OptionButtons festlegen. Gerät er hierbei außerhalb des Spielfelds, so kann man mit den beiden ScrollBars das Spielfeld nachschieben. Treibt man ihn allerdings an den Rand, so wird das Spiel beendet. Eine graphische Anzeige im unteren Bereich verändert sich proportional zur verstrichenen Zeit, die rechts daneben angezeigt wird.

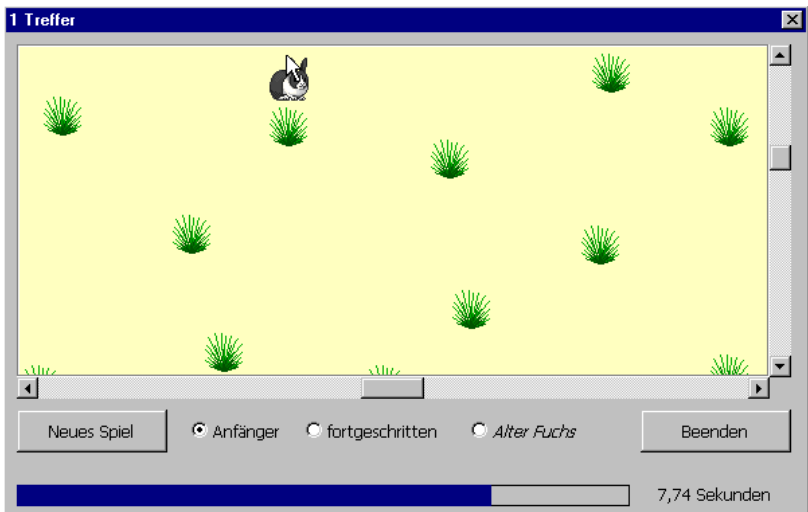


Abbildung 8.20: Das Hasenspiel

Im Animationsmodus bewegt sich der Hase von der linken oberen Ecke nach rechts unten. Gerät er an den Rand des Spielfelds, ändert sich die Richtung und er läuft einer Billardkugel gleich über das Spielfeld. Auch in diesem Modus kann man den Spielfeldausschnitt durch die ScrollBars verändern.

Mit Bordmitteln von Excel-VBA geht so ein Spielchen natürlich nicht. Da helfen die beiden Timer-Funktionen, die uns in Kapitel 7 – Sprachelemente die zweite – schon begegnet sind.

Gönnen Sie sich ruhig ein Spielchen, es dient ja einem guten Zweck ...

Doch nun zurück zur Move-Methode. Hier ein Auszug des den Hasen bewegenden Codes im MouseMove-Ereignis:

```
Private Sub imgRabbit_MouseMove(ByVal Button As Integer, _
                                ByVal Shift As Integer, ByVal X As Single, _
                                ByVal Y As Single)

'=====
'Hasen in der Maus entgegengesetzten Richtung bewegen
'=====

Dim lngNewX As Long      'neuer Left-Eigenschaftswert
Dim lngNewY As Long      'neuer Top-Eigenschaftswert
If Not bGameActive Then Exit Sub
'aktuelle Position ablegen
lngNewY = imgRabbit.Top
lngNewX = imgRabbit.Left
'Y-Delta hinzurechnen
If Y < (imgRabbit.Height / 2) Then
    lngNewY = lngNewY + lngStep
Else
    lngNewY = lngNewY - lngStep
End If
'X-Delta hinzurechnen
If X < (imgRabbit.Width / 2) Then
    lngNewX = lngNewX + lngStep
Else
    lngNewX = lngNewX - lngStep
End If
'Hasen bewegen
imgRabbit.Move lngNewX, lngNewY
...
End Sub
```

Bei der Gelegenheit ist auch noch ein Anwendungsfall für die `Enabled`-Eigenschaft angefallen. Wird bei den Gras-Images die `Enabled`-Eigenschaft auf `True` gesetzt, so kann sich unser Hase hinter den Grasbüscheln verstecken, da sie die Mausereignisse schlucken. Ist diese Eigenschaft jedoch `False`, so reichen die Grasbüschel die Ereignisse an Meister Lampe weiter, der sich darauf hin wieder aus dem Staub macht.

Hier noch die Variante der `Move`-Methode, die auf die `Controls`-Auflistung angewendet werden kann. In dieser `Collection` sind alle Steuerelemente des Containers (`UserForm` oder `Frame`) enthalten:

Move(x, x)

x	Delta in X-Richtung
y	Delta in Y-Richtung



Achten Sie bitte darauf, dass hier relative und nicht, wie bei der ersten Variante der `Move`-Methode, absolute Maße gefragt sind.

Die beiden `ScrollBars` des Hasenspiels verschieben in einer Art Kameranähe alle innerhalb des Frames befindlichen Objekte, also den Hasen und alle Grasbüschel. Hier die Ereignisroutine der horizontalen `ScrollBar`:

```
Private Sub scrollHorizontal_Change()

'=====
'    Ausschnitt horizontal verändern

'=====
    Dim lngMoveX As Long    'X-Wert, um den alle Controls
                           'bewegt werden
    'Veränderung der ScrollBar ermitteln
    lngMoveX = lngOldx - scrollHorizontal.Value
    'alten Wert speichern
    lngOldx = scrollHorizontal.Value
    'Hasen und Grasbüschel bewegen
    fraField.Controls.Move lngMoveX, 0
    'Fokus auf Frame zurücksetzen
    fraField.SetFocus
End Sub
```

SetFocus-Methode

Diese Methode macht das davor stehende Steuerelement zu dem aktiven Steuerelement der Form. Diese Methode kann auf alle Steuerelemente an-

gewendet werden, mit denen der Benutzer in Kontakt treten kann. Image und Label sind ausgeschlossen, obwohl dort die Methode nach dem Punkt angeboten wird.

In der voranstehenden Ereignisroutine wird am Ende der Fokus wieder auf das Frame-Objekt gesetzt, um das (idiotische) Blinken dieses Steuerelements zu unterdrücken.

```
fraField.SetFocus
```

ZOrder-Methode

ZOrder verändert die Reihenfolge eines Steuerelements in Z-Richtung, also senkrecht zur Form-Oberfläche:

```
imgRabbit.Zorder msoBringToTop
```

oder einfach nur

```
imgRabbit.Zorder
```

würde unseren Hasen hinter einem Grasbüschel hervorlocken. Mit der folgenden Zeile versteckt er sich wieder hinter dem Grasbüschel:

```
imgRabbit.Zorder msoSendToTop
```

Die beiden Konstanten `fmTop` und `fmBottom`, die in der Hilfe aufgeführt sind, laufen bei mir interessanterweise nicht. Die Konstanten der MSDN stimmen (natürlich).

8.3.4 Ereignisse

AddControl-Ereignis

Um es gleich vorwegzunehmen: Bei unserem Hasen stellt sich kein Nachwuchs ein, obwohl es uns sicherlich alle erfreut hätte. Stattdessen taucht nach Ende eines Spiels ein Eichhörnchen in der Spielfeldmitte auf.

Dieses Ereignis wird dann ausgelöst, wenn per Code ein Steuerelement in den aktuellen Container hinzugefügt wurde. Als Container kommen in Frage UserForm, Frame und das im MultiPage enthaltene Page-Steuerelement.

```
Private Sub fraField_AddControl(ByVal Control As _
                                MSForms.Control)
    lblBack.Caption = lblBack.Caption & _
        " (Eichhörnchen ist da!)"
End Sub
```

Das `AddControl`-Ereignis übergibt das hinzugefügte Steuerelement als Objektverweis.

AfterUpdate-Ereignisse

Dieses Ereignis tritt dann auf, wenn ein vom Anwender veränderbares Steuerelement nach einer Veränderung den Fokus verliert. Es steht somit zum Beispiel nicht für `CommandButtons` zur Verfügung, wohl aber für eine `TextBox`.

```
Private Sub txtName_AfterUpdate()
```

```
End Sub
```

BeforeDragOver-Ereignis

Hier staunt der Laie und der Fachmann wundert sich, denn dahinter verbirgt sich nicht mehr und nicht weniger als OLE-Drag & Drop, also Drag & Drop über Applikationsgrenzen hinweg. Auch hier hat VBA 6 mit VB 6 gleichgezogen.

Sie können in Word einen Ausdruck markieren und über das Word-Fenster hinaus in ein Steuerelement einer Excel-Form fallen lassen. Voraussetzung ist natürlich, dass die Excel-Form neben dem Word-Fenster sichtbar ist.

Da aber das `DataObject`, dem wir noch begegnen werden, über eine `StartDrag`-Methode verfügt, *könnten* wir komplette OLE-Drag & Drop-Operationen durchführen. Aber so, wie es als reines OLE-Drag & Drop implementiert ist, können wir kein Drag & Drop in unsere aktuelle Instanz von Excel durchführen, wohl aber in alle externen ActiveX-Server außerhalb des eigenen Prozesses. Wir werden im `MouseMove`-Ereignis darauf zurückkommen.

Das Ereignis kommt bei jeder Positionsveränderung einer per Drag-Operation geladenen Maus. Ohne sonderlich stark zu zittern kommt es leicht zu einem dutzendfachen Aufruf.

Hier das Ereignis in voller Größe und Schönheit am Beispiel einer `TextBox`:

```
Private Sub txtName_BeforeDragOver( _  
    ByVal Cancel As MSForms.ReturnBoolean, _  
    ByVal Data As MSForms.DataObject, _  
    ByVal X As Single, ByVal Y As Single, _  
    ByVal DragState As MSForms.fmDragState, _  
    ByVal Effect As MSForms.ReturnEffect, _  
    ByVal Shift As Integer)
```

```
End Sub
```

Die Argumente im Einzelnen:

Cancel: `False` bedeutet, dass das Ereignis vom Steuerelement selbst verarbeitet werden soll. Mir ist es nicht gelungen, etwas anderes als ein `False` zu produzieren.

Data: Dieses Argument übergibt ein `DataObject`, das Informationen über das Format und den Inhalt bereitstellt.

X, Y: Da sind die X- und Y-Koordinaten innerhalb des Steuerelements. Diese Angaben bringen uns nicht viel weiter, da wir ohne extensiven Gebrauch diverser API-Funktionen keine Informationen über das erlangen, was sich wirklich an der Stelle befindet, zum Beispiel welcher Eintrag innerhalb einer `ListBox`.

DragState: Dieser Wert gibt uns darüber Aufschluss, in welchem Zustand sich das `DragOver` befindet:

`fmDragStateEnter – 0:`

Der Mauszeiger befindet sich innerhalb des Steuerelements. Dies ist nur beim ersten `BeforeDragOver`-Ereignis der Fall, bei späteren Ereignissen wird `fmDragStateOver – 2` übergeben:

`fmDragStateLeave – 1:`

Der Mauszeiger befindet sich außerhalb des Steuerelements.

`fmDragStateOver – 2:`

Jedes `BeforeDragOver`-Ereignis, das sich innerhalb des Steuerelements bewegt, bewirkt diesen Wert des `DragState`-Arguments.

Effect: Dieses von der Quelle zusammengestellte maskierte (!!!) Argument kann folgende Werte aufweisen:

`fmDropEffectNone – 0:`

Das Ablegen wird nicht möglich sein. Nur dieses Argument darf im direkten Vergleich `Effect = fmDropEffectNone` geprüft werden.

`fmDropEffectCopy – 1:`

Die Daten werden in das Steuerelement kopiert.

`fmDropEffectMove – 2:`

Die Daten werden auf das Steuerelement verschoben.

`fmDropEffectCopyOrMove – 3:`

Da `Effect` maskiert ist, sind die Werte `fmDropEffectCopy - 1` UND `fmDropEffectMove - 2` übergeben worden. Das Drag & Drop weiß es also selber nicht so genau, ob nun verschoben oder kopiert wird.

Nun ist mir bei einem Drag & Drop aus Word heraus schon der Wert 7 begegnet. Nach den Regeln der Kunst muss also noch ein Wert 4 existieren, sodass also `fmDropEffectCopy - 1`, `fmDropEffectMove - 2` und der unbekannte Wert 4 beteiligt gewesen sein müssen. Was wollte mir die Quelle da sagen?

Um zu erfahren, ob nun kopiert wird, dürfen Sie nicht so fragen:

```
If Effect = fmDropEffectCopy Then
```

Da das Argument maskiert ist, müssen Sie so vorgehen:

```
If (Effect And fmDropEffectCopy) = fmDropEffectCopy Then
```

Shift: Dieses ebenfalls maskierte Argument gibt uns den Zustand der Umschalt-, Steuerungs- und Alt-Taste mit den Werten `fmShiftMask = 1`, `fmCtrlMask = 2` und `fmAltMask = 4`. Bei mir funktionierten alle mir bekannten Mask-Konstanten nicht, also mit Präfix `mso`, `fm` und `vb`.

Interessanterweise ist eine nicht modale `UserForm` für diese Drag & Drop-Operationen aus der eigenen Excel-Instanz transparent, das heißt, dass die betreffende Zelle nach dem Loslassen (Drop) an der Stelle der aktuellen Tabelle erscheint, die genau hinter der scheinbaren Drop-Position der `UserForm` liegt. Es verwundert dann auch nicht, wenn das `BeforeDragOver`-Ereignis nicht ausgelöst wird.

Schauen wir uns noch kurz das übergebene `DataObject` an.

Das `DataObject` enthält die einzufügenden Daten sowie Informationen über deren Format. Mit der Methode `Data.GetText` erhalten Sie die Daten und mit `Data.GetFormat(Format)` einen logischen Wert, der aussagt, ob der Inhalt dem durch die Konstante `Format` spezifizierten Typ entspricht. Allerdings funktioniert bislang nur das `Format = 1` (Text), weshalb wir uns die Formatüberprüfung schenken können. Bleibt also die `GetData`-Methode, die uns den Inhalt des `DataObject`s übergibt.

Im folgenden Beispiel wird `DropOrPaste` nur dann zugelassen, wenn der Text »Juhu« ist:

```
Private Sub txtName_BeforeDragOver(...)
    If Data.GetText <> "Juhu" Then
        Cancel = True
    End If
End Sub
```

BeforeDropOrPaste-Ereignis

Seit VB(A) 6 umfasst das so genannte OLE-Drag & Drop neben Drag & Drop auch Kopieren, Ausschneiden und Einfügen über die Zwischenablage. Noch werden, zumindest in VBA, nicht alle denkbaren Formate unterstützt, aber dennoch ist es der richtige Weg. In beiden Fällen gibt es ein Quellobjekt und ein Zielobjekt, und ob der Transport per Cut/Copy and Paste oder Drag & Drop vonstatten geht, ist doch wohl Nebensache.

Schauen wir uns nun die Schnittstelle von Drop or Paste an:

```
Private Sub txtName_BeforeDropOrPaste( _
    ByVal Cancel As MSForms.ReturnBoolean, _
    ByVal Action As MSForms.fmAction, _
    ByVal Data As MSForms.DataObject, _
    ByVal X As Single, ByVal Y As Single, _
    ByVal Effect As MSForms.ReturnEffect, _
    ByVal Shift As Integer)
```

```
End Sub
```

Wir müssen uns bei diesem Ereignis nur durch die Unterschiede zum vorangegangenen BeforeDragOver-Ereignis quälen, und die sind schnell erzählt.

Zum einen kann mit dem Cancel-Argument das Drop or Paste verhindert werden, was die Hilfe verschweigt. Ob es stattdessen den dort beschriebenen Inhalt (siehe BeforeDragOver-Ereignis) hat, ist eigentlich nebensächlich.

Zum anderen gibt es kein DragState-Argument, da wir nicht mehr *Draggen*, sondern *Droppen*; der Anwender hat die Maustaste ja losgelassen. Stattdessen wird uns ein Action-Argument übergeben, das die Werte fmActionPaste (2) oder fmActionDragDrop (3) annehmen kann. In der VBA-Hilfe hat sich da jemand an der Übersetzung verkünstelt, sodass mir der Durchblick verwehrt blieb. Aber die Argumentnamen sagen ja alles aus, denn bei fmActionDragDrop war ein Drag & Drop beteiligt und bei fmActionPaste eben ein Cut/Copy and Paste.

BeforeUpdate-Ereignis

Das abbrechbare BeforeUpdate-Ereignis tritt ein, nachdem ein verändertes Steuerelement verlassen wird.

```
Private Sub txtName_BeforeUpdate(ByVal Cancel As _
    MSForms.ReturnBoolean)
```

```
End Sub
```

Change-Ereignis

Dieses Ereignis tritt auf, sobald ein Steuerelement verändert wurde.

```
Private Sub txtName_Change()
```

```
End Sub
```

Click-Ereignis

Ein Mausklick auf ein bedienbares Steuerelement oder die Auswahl eines von mehreren Werten löst dieses Ereignis aus.

```
Private Sub lstFiles_Click()
```

```
End Sub
```

DbClick-Ereignis

Dieses abbrechbare Ereignis tritt auf, wenn das Steuerelement einen Doppelklick registriert, also zwei aufeinander folgende Klicks innerhalb einer einstellbaren Zeit (Systemsteuerung).

```
Private Sub lstFiles_DbClick(ByVal Cancel As _  
                                MSForms.ReturnBoolean)
```

```
End Sub
```

Enter-Ereignis

Das dem GotFocus-Ereignis in Visual Basic entsprechende Enter-Ereignis tritt auf, wenn das Steuerelement den Fokus erhält.

```
Private Sub txtName_Enter()
```

```
End Sub
```

Exit-Ereignis

Das dem LostFocus-Ereignis in Visual Basic entsprechende Exit-Ereignis tritt auf, wenn das Steuerelement den Fokus wieder abgibt.

Nehmen wir an, Sie wollen erzwingen, dass eine TextBox einen numerischen Inhalt hat.

```
Private Sub txtNumber_Exit(ByVal Cancel As _  
                                MSForms.ReturnBoolean)  
    If txtNumber.Text = "" Then  
        txtNumber.Text = 0  
    ElseIf not IsNumeric(txtNumber.Text) Then  
        MsgBox "Bitte geben Sie eine gültige Zahl ein."  
        Cancel = True
```



```
End If
End Sub
```

Zwei Haken hat diese Vorgehensweise. Der erste betrifft den Cursor, der nicht zu sehen ist, nachdem die MessageBox ihn irgendwie zum Verschwinden gebracht hat. Der zweite Haken liegt darin, dass wir dadurch verhindern, dass der Anwender den Dialog über die Abbrechen-Schaltfläche verlassen kann. Zwar setzt sich die Schließen-Schaltfläche der Titelleiste des Dialogs doch durch, aber die TextBox gibt dennoch ihren Senf noch einmal ab.

Solange kein echtes Validate-Ereignis verfügbar ist, scheint eine zentrale Überprüfung der Werte in der Routine, die das Speichern oder Auswerten der Dialogdaten vornimmt, am sinnvollsten zu sein.

Error-Ereignis

Wenn Sie sich das, was in Kapitel 9 - Fehlerbehandlung steht, zu Herzen nehmen, können Sie auf dieses Ereignis getrost verzichten.

KeyDown-Ereignis

Das KeyDown-Ereignis tritt in dem Moment auf, in dem eine Taste gedrückt wird.

Im folgenden Beispiel wird es dazu benutzt, den Dialog zu schließen, wenn der Anwender in der TextBox txtName die Return-Taste bedient:

```
Private Sub txtName_KeyDown(ByVal KeyCode As _
    MSForms.ReturnInteger, ByVal Shift As Integer)
    If KeyCode = vbKeyReturn Then
        Me.Hide
    End If
End Sub
```

Die KeyCode-Konstanten finden Sie im Anhang und auch im Objektkatalog unter KeyCodeConstants in der VBA-Bibliothek. Umlaute und sonstige nationale Sonderzeichen sind nicht mit Konstanten belegt.

KeyCode-Konstanten

Die Shift-Konstanten fehlen in den Bibliothek. Sie können natürlich mit Zahlen arbeiten, denn Konstanten sind nichts anderes als Integer-Werte. Eine zweifellos sinnvollere Vorgehensweise ist die Deklaration dieser drei Konstanten in einem Modul:

Shift-Konstanten

```
Public Const vbShiftMask As Long = 1
Public Const vbCtrlMask As Long = 2
Public Const vbAltMask As Long = 4
```

Nehmen wir an, Sie erlauben dem Anwender durch die Tastenkombination (Strg)(P) einen Druck zu starten, könnte der Code so aussehen:

```
Private Sub txtName_KeyDown(ByVal KeyCode As _
    MSForms.ReturnInteger, ByVal Shift As Integer)
    If KeyCode = vbKeyP And _
        Shift And vbCtrlMask = vbCtrlMask Then
        'Drucken
    End If
End Sub
```

Das Prinzip der Maskierung wird in Kapitel 7 – Sprachelemente die zweite – behandelt. Hier dennoch eine Kurzfassung.

Der **And**-Operator führt eine Und-Verknüpfung durch und gibt alle Zweierpotenzen zurück, die in beiden Ausdrücken vorhanden sind. Ist die Stelle 2¹ in Shift besetzt, so ist das Ergebnis = 2¹, also gleich vbCtrlMask.

Wie oft wird dieses Ereignis durchlaufen? Zweimal, denn der Anwender wird vermutlich zuerst die Steuerungs-Taste drücken und danach die P-Taste. Beim ersten Mal ist KeyCode = vbKeyControl und Shift = 2 und beim zweiten Mal ist unsere Bedingung erfüllt, denn KeyCode ist vbKeyP und Shift immer noch 2. Das KeyDown-Ereignis wird also mit jeder 0-1-Flanke der Tastatur ausgelöst.

KeyPress-Ereignis

Das KeyPress-Ereignis wird im Gegensatz zu KeyDown oder KeyUp nur dann aufgerufen, wenn ein darstellbares Zeichen eingegeben wurde.

```
Private Sub txtName_KeyPress(ByVal KeyAscii As _
    MSForms.ReturnInteger)

End Sub
```

Der in KeyAscii steckende ANSI-Code (!) des Zeichens differenziert natürlich zwischen kleinen und großen Buchstaben. Mit Chr(KeyAscii) können Sie daraus das aktuelle Zeichen erzeugen.

Die Rück-Taste löst das Ereignis übrigens nicht aus, wie irrtümlich in der Hilfe zu lesen ist.

KeyUp-Ereignis

Das KeyUp-Ereignis wird mit der 1-0-Flanke der Tastatur angestoßen. Im Übrigen gelten die Informationen, die zum KeyDown-Ereignis aufgeführt sind.

Layout-Ereignis

Das Layout-Ereignis wird ausgelöst, wenn die Größe oder Position (fehlt in der Hilfe) des betreffenden Steuerelements oder eines Mitglieds seiner Controls-Auflistung verändert wird. Außerdem wird es nach dem Laden des Dialogs angestoßen. UserForm, Frame und MultiPage unterstützen dieses Ereignis.

```
Private Sub UserForm_Layout()
```

```
End Sub
```

Es kann dazu verwendet werden, nach einer Größenänderung des dazugehörigen Steuerelements die darin enthaltenen Steuerelemente neu zu positionieren. Doch wer sollte denn die Form in der Größe verändern? Der Anwender? Wie?

Neben Größenänderungen auf Grund einer gesetzten AutoSize-Eigenschaft ist die einzige mögliche Größenänderung in der Forms-Bibliothek die Veränderung per Code. Im Moment, also solange eine Form nicht vom Anwender zur Laufzeit vergrößert oder verkleinert werden kann, kann ich mir nur einen Fall vorstellen, bei der das Layout-Ereignis von Nutzen wäre. Und das ist der Fall, wenn Sie Beschriftungen dynamisch erzeugen, bei mehrsprachigen Applikationen zum Beispiel. Und da kann man durch entsprechend großzügige und geschickte Größenzuordnung zur Entwurfszeit dieses Problem zumindest entschärfen.

MouseDown-Ereignis

Das MouseDown-Ereignis tritt in dem Moment auf, in dem eine Maustaste gedrückt wird.

```
Private Sub cmdOK_MouseDown(ByVal Button As Integer, _
                             ByVal Shift As Integer, _
                             ByVal X As Single, _
                             ByVal Y As Single)
```

```
End Sub
```

Im Button-Argument erfahren wir, welche Maustaste gedrückt wurde. Auch hier funktionieren die in der Hilfe genannten Konstanten fmButtonLeft, fmButtonRight und fmButtonMiddle nicht. Warum von den seit Jahren aus VB und VBA (bis zu Version 5) bekannten Konstanten vbLeftButton, vbRightButton und vbMiddleButton auch sprachlich abgewichen wird, ist unverständlich. Also bauen wir uns die Konstanten auch hier selbst:

MouseButton-Konstanten

```
Public Const vbLeftButton As Long = 1
Public Const vbRightButton As Long = 2
Public Const vbMiddleButton As Long = 4
```

Das Shift-Argument wurde bereits im KeyDown-Ereignis vorgestellt. X und Y geben die Position des Mauszeigers innerhalb des Steuerelements an.

MouseMove-Ereignis

MouseMove wird ausgelöst, sobald sich die Maus über dem Steuerelement befindet, und zwar unabhängig davon, ob eine Maustaste gedrückt ist.

Sie erinnern sich noch an unseren Hasen. Der Anlass seiner Flucht ist das MouseMove-Ereignis. Um die Zielkoordinate des Hasen-Images zu bestimmen, werden die übergebenen X- und Y-Argumente untersucht. Ist Y kleiner als die Hälfte der Hasenhöhe, so muss sich die Maus von oben nähern und Meister Lampe bewegt sich nach unten. Andernfalls muss sich die Maus von unten nähern und die Schrittweite wird von der aktuellen Top-Position abgezogen:

```
If Y < (imgRabbit.Height / 2) Then
    lngNewY = lngNewY + lngStep
Else
    lngNewY = lngNewY - lngStep
End If
```

Gleiches passiert natürlich auch beim X-Argument.

MouseUp-Ereignis

Das MouseUp-Ereignis tritt in dem Moment auf, in dem eine Maustaste losgelassen wird. Erst danach wird das Click-Ereignis ausgelöst. Zu Argumenten und Handhabung siehe die beiden anderen Mouse-Ereignisse.

RemoveControl-Ereignis

Wird im Container (UserForm, Frame oder MultiPage) ein Steuerelement entfernt, so wird dieses Ereignis ausgelöst:

```
Private Sub UserForm_RemoveControl(ByVal Control As _
                                         MSForms.Control)

End Sub
```

Man hat also noch einmal Gelegenheit, sich von dem Steuerelement zu verabschieden und ihm für seine Zusammenarbeit zu danken, bevor es aus dieser Welt scheidet.

Scroll-Ereignis

Bei der ScrollBar bedeutet dieses Ereignis, dass die Position des Schiebelements der ScrollBar verändert wurde. Jede Veränderung löst ein Scroll-Ereignis aus. Wird das Schiebelement dann losgelassen, kommt ein einzelnes Change-Ereignis:

```
Private Sub ScrollBar_Scroll()

End Sub
```

Bei UserForm, Frame oder MultiPage wird das Ereignis ausgelöst, wenn das betreffende Steuerelement ScrollBars anzeigt und diese manipuliert wurden.

```
Private Sub UserForm_Scroll( _
    ByVal ActionX As MSForms.fmScrollAction, _
    ByVal ActionY As MSForms.fmScrollAction, _
    ByVal RequestDx As Single, _
    ByVal RequestDy As Single, _
    ByVal ActualDx As MSForms.ReturnSingle, _
    ByVal ActualDy As MSForms.ReturnSingle)

End Sub
```

Hiermit habe ich ein Problem, da mir beim besten Willen nicht klar ist, wozu das gut sein soll. Im Übrigen ist die Handhabung einer eigenständigen ScrollBar selbst so einfach, dass man sich dies nicht antun muss.

Zoom-Ereignis

Das Zoom-Ereignis tritt auf, wenn die Zoom-Eigenschaft einer UserForm, eines Frame oder MultiPage verändert wurde. Auch hier stellt sich die Frage, wer war das und warum hat er das getan? Die Änderung kann nur per Code erfolgen oder zumindest zugelassen worden sein. Wer ist also beantwortet: Sie. Und auch das Warum können nur Sie beantworten. Was ich damit sagen will, ist, dass wir von einem Zoom schlechterdings nicht überrascht werden können. Wozu also das Ereignis?

Dennoch ein paar mehr oder weniger sinnvolle Zeilen dazu:

```
Private Sub Frame_Zoom(Percent As Integer)
    MsgBox "Ich bin gerade auf " & Percent & _
        " eingestellt worden."
End Sub

Private Sub ScrollBar_Change()
    Frame.Zoom = ScrollBar.Value
End Sub
```

Die Min- und Max-Eigenschaften der ScrollBar sind auf die Grenzen von 10 und 400 eingestellt worden, die beiden Grenzwerte der Zoom-Eigenschaft.

8.4 UserForm

Die UserForm verfügt über ein paar Eigenschaften und Methoden, die wir uns noch ansehen sollten:

8.4.1 Eigenschaften

StartPosition-Eigenschaft

Mit dieser Eigenschaft bestimmen Sie die Anfangsposition der UserForm auf dem Bildschirm. Folgende Einstellungen stehen im Eigenschaftsfenster zur Auswahl:

Einstellung	Wert	Bedeutung
Manual	0	abhängig von Left und Top-Eigenschaft
CenterOwner	1	auf dem darunter liegenden Excel-Fenster zentriert
CenterScreen	2	Mitte Bildschirm
Windows Default	3	obere linken Ecke des Bildschirms

Die letzten drei Einstellungen ignorieren die Left- und Top-Eigenschaften.

8.4.2 Methoden

Hide-Methode

Diese Methode entfernt die UserForm aus dem Bildschirmspeicher:

```
Me.Hide
```

Show-Methode

Wir sprachen bereits bei der Behandlung der Tag-Eigenschaft über die verschiedenen Möglichkeiten, eine UserForm anzuzeigen.

Im ersten Beispiel wird die UserForm unter ihrem Namen mit der Show-Methode einfach aufgerufen.

```
frmMain.Show
```

In dieser Zeile sind die beiden Schritte Laden und Anzeigen der Form zusammengefasst. Wenn Sie aber eine Form vor der Anzeige bereits in irgendeiner Art parametrieren wollen, müssen die beiden Schritte getrennt werden:

```

Dim fMain as frmMain
Set fMain = New frmMain 'Laden der Form
' hier haben Sie Gelegenheit zur Parametrierung
fMain.Show               'Anzeigen der Form
' hier können Sie Parameter der Form noch auswerten
Unload fMain              'Form aus Speicher entfernen

```

Wenn Sie nach der Show-Methode noch auf Objekte oder Variablen der Form zugreifen wollen, darf die Form lediglich mit der Hide-Methode aus dem Bildschirmspeicher entfernt werden, ohne sie aus dem Variablenspeicher zu entfernen. Siehe hierzu auch das QueryClose-Ereignis.

Neu in Excel 2000 ist die Möglichkeit einen Dialog nicht-modal aufzurufen.



Modal ist ein Dialog dann, wenn keine Aktionen außerhalb des Dialogs vorgenommen werden können, solange dieser auf dem Bildschirm ist. Typische Vertreter dieser Gattung sind Options-Dialoge. Nicht-modal ist ein Dialog demzufolge dann, wenn der Anwender in der Anwendung weiter arbeiten kann, obwohl der Dialog auf dem Bildschirm ist.

Gesteuert wird dies über ein optionales Argument der Show-Methode mit folgenden Werten:

▼ 0 – vbModeless bedeutet nicht-modal

▼ 1 – vbModal bedeutet modal

Wird das Argument weggelassen, so wird der Dialog modal gestartet.

8.4.3 Ereignisse

Über die meisten Ereignisse der UserForm wurde bereits gesprochen. Lediglich dem Activate- und dem QueryClose-Ereignis müssen wir uns noch zuwenden:

```

frmSample.Show vbModeless
frmSample.Show vbModal

```

Activate-Ereignis

Das Activate-Ereignis, dessen nur die UserForm mächtig ist, wird ausgelöst, wenn die Form wieder den Fokus erhält. Eine UserForm kann nur durch eine andere UserForm den Fokus verlieren, interessanterweise, wie ich finde. Denn sie verliert den Fokus an ein anderes Fenster, und eigentlich hätte ich erwartet, dass eine MessageBox diese Voraussetzungen erfüllt. Oder auch eine Tabelle, die einem nicht modal gestarteten Dialog den Fokus raubt.

```
Private Sub UserForm_Activate()  
  
End Sub
```

QueryClose-Ereignis

Das QueryClose-Ereignis wird ausgelöst, bevor die Form aus dem Variablenspeicher entfernt wird.

```
Private Sub UserForm_QueryClose(Cancel As Integer, _  
                                CloseMode As Integer)  
  
End Sub
```

Mit dem Cancel-Argument lässt sich der Prozess nach gewohnter Art und Weise abbrechen.

Das CloseMode-Argument teilt uns mit, was der Auslöser dieses Entladeprozesses ist:

Konstante	Wert	Bedeutung
vbFormControlMenu	0	Der Anwender will den Dialog über die Schließen-Schaltfläche (Kreuz rechts in Titelleiste) schließen.
vbFormCode	1	Die Unload-Anweisung wird per Code aufgerufen.
bAppWindows	2	Die aktuelle Windows-Betriebsumgebungssitzung wird beendet.
vbAppTaskManager	3	Die Anwendung wird vom Windows-Task-Manager geschlossen.

Gerade, wenn wir nach dem Schließen des Dialogs noch auf diesen per Code zugreifen wollen, muss verhindert werden, dass der Anwender den Dialog *nicht* über unsere bereitgestellte Schließen-Schaltfläche schließt.

8.4.4 Eine kleine Spielerei mit einer UserForm

Es gibt noch eine Reihe von Eigenschaften und Methoden, die bislang nicht erwähnt wurden. Hier folgt ein kleines Beispiel, das man als nette Spielerei bezeichnen könnte.



Abbildung 8.21: Form mit Symbolleiste

In diesem Beispiel ist Cut, Copy and Paste, das ohnehin von der UserForm gemanagt wird, zusätzlich mit einer nachgebauten Symbolleiste unterstützt, die auch noch Undo und Redo anbietet.

Abbildung 8.22 zeigt ein Drag & Drop von der ListBox zur TextBox:

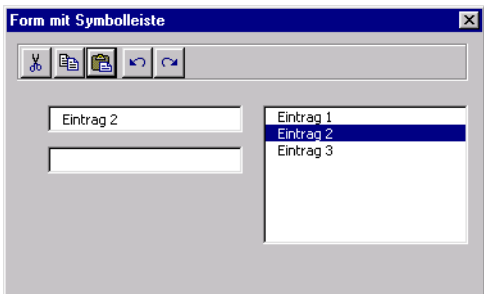


Abbildung 8.22: Form mit Symbolleiste, Drag & Drop

Aber auch die Drag & Drop-Richtung von TextBox in ListBox wurde implementiert, wie Abbildung 8.23 zeigt:

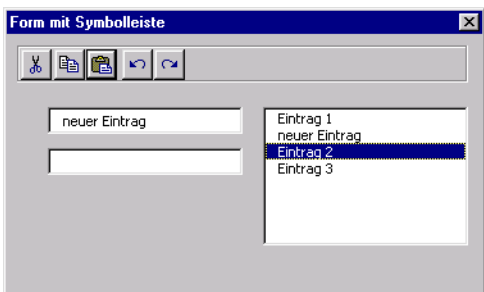


Abbildung 8.23: Form mit Symbolleiste, Einfügen in ListBox

Bei der Symbolleiste handelt es sich um ein Frame, in das fünf `CommandButtons` eingefügt wurden: `cmdCut`, `cmdCopy`, `cmdPaste`, `cmdUndo` und `cmdRedo`. Die beiden `TextBoxes` heißen `txt1` und `txt2` und die `ListBox` `lst`.

Hier nun die privaten Variablen:

```
Private data As DataObject
Private strActualControl As String
```

Hinter `strActualControl` verbirgt sich der Name des aktuellen Steuerelements, den wir noch heftig benötigen. Schauen wir uns zunächst die Prozedur `UserForm_Initialize` an:

```
Private Sub UserForm_Initialize()

'=====
'    Fokus in txt1 setzen und cmdPaste deaktivieren
'=====

    txt1.SetFocus
    Set data = New DataObject
    cmdPaste.Enabled = False
    lst.AddItem "Eintrag 1"
    lst.AddItem "Eintrag 2"
    lst.AddItem "Eintrag 3"
End Sub
```

Es muss dafür gesorgt werden, dass die Variable `strActualControl` immer auf dem Laufenden ist. Hierfür ist bei den `TextBoxes` die Prozedur des `Enter`-Ereignisses zuständig:

```
Private Sub txt1_Enter()

'=====
'    strActualControl zuweisen
'=====

    strActualControl = "txt1"
End Sub
```

Bei der `ListBox` ist es nicht ganz so einfach, weil sichergestellt sein muss, dass auch ein Eintrag ausgewählt wurde. Das wiederum teilt uns der `ListIndex` mit, der ohne ausgewähltes Element auf `-1` steht. Beim ersten `Enter`-Ereignis ist aber noch kein Element ausgewählt, weshalb wir das `Click`-Ereignis zusätzlich bemühen müssen:

```

Private Sub lst_Click()

'=====
'   strActualControl zuweisen
'=====

    strActualControl = "lst"
End Sub

Private Sub lst_Enter()

'=====
'   strActualControl zuweisen
'=====

    If lst.ListIndex = -1 Then
        strActualControl = ""
    Else
        strActualControl = "lst"
    End If
End Sub

```

Um später das Undo und Redo realisieren zu können, müssen wir der UserForm vorgaukeln, dass alle Operationen vom Anwender ausgehen, denn die Methoden Undo und Redo der UserForm greifen hierbei nur auf solche Operationen, die auch vom Anwender durchgeführt wurden. Aus diesem Grund senden die beiden Schaltflächen für Ausschneiden und Kopieren die entsprechende Tastenkombination an das aktive Steuerelement (strActualControl), und alle sind zufrieden:

```

Private Sub cmdCut_Click()

'=====
'   Strg-X an aktuelles Control senden
'=====

    If strActualControl <> "" Then
        Me.Controls(strActualControl).SetFocus
        SendKeys "^X"
    End If
End Sub

Private Sub cmdCopy_Click()

```

```

'=====
'   Strg-C an aktuelles Control senden
'=====

If strActualControl <> "" Then
    Me.Controls(strActualControl).SetFocus
    SendKeys "^C"
End If
End Sub

```

Nun werden für alle Steuerelemente, die Einfügen können dürfen, Ereignisprozeduren für das `KeyDown`-Ereignis benötigt. Für die TextBoxes sehen diese so aus:

```

Private Sub txt1_KeyDown( _
    ByVal KeyCode As MSForms.ReturnInteger, _
    ByVal Shift As Integer)
'=====
'   bei Strg-C oder Strg-X Data-Objekt laden und
'   cmdPaste enablen
'=====

If KeyCode = vbKeyC And (Shift And vbCtrlMask) = _
    vbCtrlMask Or KeyCode = vbKeyX And _
    (Shift And vbCtrlMask) = vbCtrlMask Then
    data.Clear
    data.SetText txt1.Text
    cmdPaste.Enabled = True
End If
End Sub

```

Hier wird also zuerst das `DataObject`-Objekt gefüttert. Da die `Enabled`-Eigenschaft der Einfügen-Schaltfläche beim Start der Form auf `False` sitzt, wird sie nach dem erfolgten Kopieren auf `True` gesetzt. Diese Prozedur wird nicht nur durch die `cmdCut_Click` oder `cmdCopy_Click` aufgerufen, sondern auch durch ein direktes `[Strg]+[X]` oder `[Strg]+[C]` des Anwenders.

Für den Einfügevorgang bei den TextBoxes müssen wir nichts unternehmen, da die TextBox das wohl automatisch zusammen mit dem `DataObject`-Objekt tut.

Anders bei der ListBox, da diese von Hause aus das Einfügen nicht unterstützt:

```

Private Sub lst_KeyDown( _
    ByVal KeyCode As MSForms.ReturnInteger, _
    ByVal Shift As Integer)

'=====
'    bei Strg-C oder Strg-X Data-Objekt laden und
'    cmdPaste enablen, bei Strg-V einfügen
'=====

If KeyCode = vbKeyC And (Shift And vbCtrlMask) = _
    vbCtrlMask Or KeyCode = vbKeyX And _
    (Shift And vbCtrlMask) = vbCtrlMask Then
    data.Clear
    data.SetText lst.Text
    cmdPaste.Enabled = True
ElseIf KeyCode = vbKeyV And (Shift And vbCtrlMask) = _
    vbCtrlMask Then
    lst.AddItem data.GetText, lst.ListIndex
End If
End Sub

```

Nun noch Drag & Drop von ListBox zu TextBox:

```

Private Sub lst_MouseMove(ByVal Button As Integer, _
    ByVal Shift As Integer, _
    ByVal X As Single, _
    ByVal Y As Single)

'=====
'    Drag & Drop starten
'=====

If Button = vbLeftButton Then
    data.Clear
    data.SetText lst.Text
    data.StartDrag
End If
End Sub

```

Lassen Sie sich durch dieses kleine Beispiel inspirieren. Eine gelungene Benutzerschnittstelle ist das A und O einer jeden Applikation, auch einer in VBA geschriebenen.

8.5 Label

Labels sind unproblematische Geschöpfe, die eigentlich nur zu Anzeigezwecken in Formularen platziert werden.

Abbildung 8.24 enthält fünf Labels zu vier verschiedenen Zwecken:

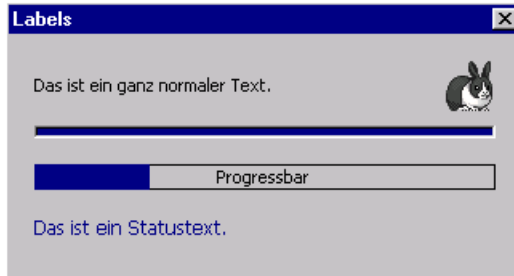


Abbildung 8.24: Labels

Textlabel

Bis auf die `Caption` sind alle Eigenschaften auf ihren Standardwerten.

Bildlabel

Auch Bilder können in Labels problemlos angezeigt werden. Statisch anzuzeigende Bilder können Sie über den Dialog `Bild` laden festlegen, der sich hinter der `Picture`-Eigenschaft verbirgt. Ändern Sie bitte noch die `BackStyle`-Eigenschaft auf `fmBackStyleTransparent` ab.

Ist `AutoSize` auf `False`, wird das Bild in die eingestellte Geometrie gezwängt, wobei auch das original Höhen-Breiten-Verhältnis verändert wird.

Trennlabel

Bei diesem Label, das ich zur optischen Trennung von eigenständigen Dialogabschnitten verwende, wurde die `BackColor` auf die Systemfarbe »Markierung« eingestellt und als `SpecialEffect` `fmSpecialEffect-Sunken` gewählt.

ProgressBar

Diese `ProgressBar` besteht aus den beiden Labels `lblBack` und `lblFore`. `lblBack`, das große Label im Hintergrund, erhielt den `BorderStyle` `fmBorderStyleSingle`, und das blaue `lblFore` als `BackColor` die Systemfarbe »Markierung«. Um von der Reihenfolge der Labels in Z-Richtung unabhängig zu sein, können Sie dem `lblBack` sicherheitshalber den `BackStyle` `fmBackStyleTransparent` zuweisen.

Hier ein Beispiel für die Dynamik einer solchen ProgressBar, das dem Hakenspiel entnommen ist. Der dort verwendete Timer meldet sich alle 20 Millisekunden. Die Spielzeit ist auf 10 Sekunden eingestellt, wodurch die Breitenveränderung ein 500stel der `lblBack` sein muss. Da beim Spielstart die Breite mit `lblFore.Width = 0` zurückgesetzt wird, genügt es, zur alten Breite von `lblFore` dieses 500stel der `lblBack`-Breite hinzuzuzählen:

```
'lblFore vergrößern
lblFore.Width = lblFore.Width + lblBack.Width / 500
lblElapsedTime.Caption = Format(10 * lblFore.Width / _
                               lblBack.Width, "0.00") & "
Sekunden"
```

Wenn Sie Datensätze abarbeiten, so steht Ihnen in Form des Zeilenzeigers ein Maß für die Breitenveränderung zur Verfügung:

```
nRows = rgnDaten.CurrentRegion.Rows.Count
For iRow = 2 To nRows
    lblFore.Width = lblBack.Width / (nRows - 1) * (iRow - 1)
    Me.Repaint
...
Next
```

Die Tabelle besitzt eine Überschriftzeile, die uns nicht interessiert, weshalb die Schleife mit `iRow = 2` beginnt. Wir müssen daher von der Gesamtzeilenzahl `nRows` und dem Zeilenzeiger `iRow` diese Zeile abziehen, um eine korrekte Darstellung zu haben.

Sollte sich die Statusanzeige nicht verändern, so müssen Sie die Aktualisierung über ein `Me.Repaint` erzwingen.

Statuslabel

Hier ist im Vergleich zu dem Textlabel als `BackColor` die Systemfarbe »Markierung« gewählt und die `Size`-Eigenschaft des Font-Objekts um einen Punkt vergrößert worden.

8.6 TextBox

Abbildung 8.25 zeigt drei gängige Beispiele für TextBoxes:

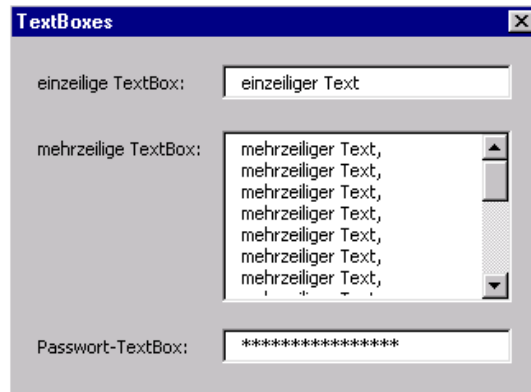


Abbildung 8.25: TextBoxes

Einzeiliger Text

Das ist wohl der Standardfall einer `TextBox`. Abgesehen von eventuellen Änderungen der Höhe (siehe 8.2.2 – Optische Korrekturen).

Mehrzeiliger Text

Um einen mehrzeiligen Text darstellen zu können, sollte die `TextBox` natürlich entsprechend vergrößert werden. Daneben muss die `MultiLine`-Eigenschaft auf `True` gesetzt werden. Bei der `ScrollBar`-Eigenschaft haben Sie die Wahl unter:

- ▼ 0 - `fmScrollBarsNone` (keine ScrollBar)
- ▼ 1 - `fmScrollBarsHorizontal` (horizontale ScrollBar)
- ▼ 2 - `fmScrollBarsVertical` (vertikale ScrollBar)
- ▼ 3 - `fmScrollBarsBoth` (horizontale und vertikale ScrollBar)

Passwort-TextBoxes

Um eine Passwort-`TextBox` darzustellen, tragen Sie bei der `PasswordChar`-Eigenschaft ein geeignetes Zeichen ein, zum Beispiel »*«. Das war's.

8.7 ComboBox

`ComboBoxes` präsentieren sich in zwei verschiedenen Ausprägungen. Die `ListStyle`-Variante ermöglicht dem Anwender nur die Auswahl eines bereits existierenden Eintrags, wohingegen die `ComboStyle`-Variante auch das Verändern eines Eintrags ermöglicht (siehe Abbildung 8.26). Bevor wir uns mit den Unterschieden dieser beiden Typen auseinander setzen, schauen wir uns die Eigenschaften und Methoden sowie das zusätzliche Ereignis an, mit dem die `ComboBox` aufwarten kann.



Abbildung 8.26: ComboBoxes

8.7.1 Eigenschaften

EnterFieldBehavior-Eigenschaft

Die beiden Werte stehen zur Verfügung:

- ▼ 0 - `fmEnterFieldBehaviorSelectAll` bedeutet, dass bei einem erneuten (!) Fokuserhalt der ganze Eintrag markiert wird.
- ▼ 1 - `fmEnterFieldBehaviorRecallSelection` bedeutet, dass bei einem erneuten (!) Fokuserhalt der Eintrag markiert wird, der beim letzten Bearbeiten der ComboBox selektiert war.

Diese nur in der `ComboStyle`-Variante sinnvolle Eigenschaft macht dann Sinn, wenn der Anwender sich an einer anderen Stelle einen Rat einholen kann, welcher Wert denn nun sinnvollerweise einzugeben sei. Wenn Sie eine weitere Form zu diesem Zweck per Schaltfläche anzeigen lassen, sollten Sie den Fokus anschließend wieder auf die ComboBox zurücksetzen.

List-Eigenschaft

Hinter der `List`-Eigenschaft verbirgt sich ein nullbasiertes `String-Array` mit allen Einträgen der ComboBox. Dieses Array eignet sich dazu, die Einträge in einer Schleife zu durchlaufen. Ein Beispiel finden Sie bei der `ListIndex`-Eigenschaft.

ListCount-Eigenschaft

Die `ListCount`-Eigenschaft gibt uns die Anzahl der Einträge zurück, die in der ComboBox enthalten sind. Auch diese Eigenschaft kommt in dem Beispiel der `ListIndex`-Eigenschaft vor.

ListIndex-Eigenschaft

Der Listindex ist ein nullbasierter Zeiger auf den aktuell gewählten Eintrag der ComboBox. Dieser Zeiger kann gelesen und geschrieben werden. Ist (noch) kein Element ausgewählt, so beträgt der Wert -1.

Im folgenden Beispiel werden alle Einträge einer ComboBox mit einer Zeichenkette verglichen und im Falle der Übereinstimmung dieser Wert auch ausgewählt. Ist der Eintrag nicht vorhanden, so wird der ListIndex auf -1 gesetzt:

```
Private Sub SelectComboEntry(cmbAct As ComboBox, _
                             strCompare As String)
'=====
'   strCompare in cmbAct auswählen, wenn vorhanden
'-----
'Argumente:
'   cmbAct:      zu untersuchende ComboBox
'   strCompare:  Vergleichszeichenkette
'=====

Dim iItem As Long          'Zeiger in ComboBox
Dim iListIndex As Long     'zuzuweisender ListIndex
'Abbruch, wenn ComboBox leer ist
If cmbAct.ListCount = 0 Then Exit Sub
'ListIndex auf -1 voreinstellen
iListIndex = -1
'Einträge durchlaufen
For iItem = 0 To cmbAct.ListCount - 1
    'aktuellen Eintrag vergleichen
    If cmbAct.List(iItem) = strCompare Then
        'Eintrag gefunden
        iListIndex = iItem
        Exit For
    End If
Next
'ListIndex zuweisen
cmbAct.ListIndex = iListIndex
End Sub
```

ListRows-Eigenschaft

Mit dieser Eigenschaft legen Sie fest, wie viele Einträge die ComboBox umfasst. Ist dieser Wert größer als die Zahl der tatsächlichen Einträge, so beschränkt sich die Anzahl der dargestellten Einträge auf die vorhandenen.

Diese Eigenschaft ist vor allem dann von Interesse, wenn die Anzahl der gleichzeitig anzuzeigenden Einträge geringer als die Anzahl der vorhandenen Einträge ist. Wenn Sie beispielsweise eine Monatsauswahl ermöglichen wollen, so sollte diese `ListRows`-Eigenschaft auf zwölf erhöht werden.

MatchEntry-Eigenschaft

Diese Eigenschaft steuert das Vergleichsverhalten bei der Eingabe von Zeichen, wenn die `ComboBox` den Fokus hat. Folgende Einstellungen sind möglich:

- ▼ 0 - `fmMatchEntryFirstLetter` bedeutet, dass jede Eingabe eines Zeichens einen neuen Vergleich dieses Zeichens mit den Anfangsbuchstaben der Einträge zur Folge hat.
- ▼ 1 - `fmMatchEntryComplete` bedeutet, dass die eingegebenen Zeichen als Zeichenkette mit den Anfängen der Einträge verglichen werden.
- ▼ 2 - `fmMatchEntryNone` bedeutet, dass kein Vergleich und somit auch kein Vorblenden von Vergleichseinträgen stattfindet.

MatchRequired-Eigenschaft

Steht diese Eigenschaft auf `True`, muss ein passender Eintrag ausgewählt werden, bevor die `ComboStyle-ComboBox` verlassen werden kann.

In der Hilfe steht unter Anmerkungen:

Wenn die **MatchRequired**-Eigenschaft den Wert **True** hat, kann der Benutzer das Kombinationsfeld-Steuerelement erst dann verlassen, wenn der eingegebene Text mit einem Eintrag in der vorhandenen Liste übereinstimmt. **MatchRequired** stellt die Integrität der Liste dadurch sicher, dass nur die Auswahl eines vorhandenen Eintrags möglich ist.

Habe ich da etwas nicht verstanden, oder würde die `ListStyle`-Variante dies nicht automatisch sicherstellen?

Text-Eigenschaft

Die `Text`-Eigenschaft gibt den Text des aktuell gewählten Eintrags zurück.

8.7.2 Methoden

Die folgenden beiden Methoden gelten auch für die `ListBox`.

AddItem-Methode

Mit dieser Methode fügen Sie einen Eintrag in eine `ComboBox` oder eine `ListBox` hinzu.

```
cmbX.AddItem "<bitte auswählen>"
```

Aber die `AddItem`-Methode verfügt auch noch über ein zweites, diesmal optionales Argument, das den künftigen `ListIndex` des hinzuzufügenden Eintrags angibt:

```
cmbX.AddItem "A"
cmbX.AddItem "B"
cmbX.AddItem "C", 1
```

Die Elemente werden in der Reihenfolge »A«, »C« und »B« in der Combo-Box erscheinen, da »C« den künftigen `ListIndex` 1 erhielt. »B« wurde in der letzten Zeile auf Position 2 verschoben.

Clear-Methode

Die `Clear`-Methode entfernt alle Einträge einer Combo- oder `ListBox`. Wird einer dieser beiden Kandidaten potenziell mehrfach in einem Dialog geladen, so sollte die Laderoutine mit einem `X.Clear` beginnen.

RemoveItem-Methode

Mit der `RemoveItem`-Methode entfernen Sie einen Eintrag aus einer Combo- oder `ListBox`. Damit diese auch weiß, welcher Eintrag entfernt werden soll, wird ihr der `ListIndex` des Eintrags übergeben.

```
cmbX.AddItem "A"
cmbX.AddItem "B"
cmbX.AddItem "C", 1
cmbX.RemoveItem 0
```

Danach werden nur noch die Einträge »C« und »B« vorhanden sein.

8.7.3 Ereignisse

Die `ComboBox` kennt noch ein Ereignis, das bislang noch nicht vorgestellt wurde.

DropDownClick-Ereignis

Wird der `DropDown`-Button einer `ComboBox` angeklickt, so kommt dieses Ereignis. Wird ein anderer Eintrag ausgewählt, tritt anschließend das `Change`-Ereignis auf. Wird derselbe Eintrag jedoch wieder ausgewählt, so tritt das `DropDownClick`-Ereignis ein zweites Mal auf.

```
Private Sub cmb_DropButtonClick()

End Sub
```

8.7.4 Besonderheit der ComboStyle-ComboBox

Um es gleich vorwegzunehmen: Ich mag keine ComboBoxes, die über die reine Auswahl eines Eintrags hinausgehen. Kann ich mir denn sicher sein, dass ein neuer Eintrag, der sich durch die *Eingabe* des Anwenders ergeben hat, nun beabsichtigt ist oder ein Versehen darstellt? Und wenn Sie in jedem Fall der Eingabe eine MessageBox hinterherjagen, in der unser geplagter Anwender gefragt wird, ob das sein Ernst ist, wird Ihr Telefon in Kürze klingeln.

Doch das ist lediglich meine Meinung, die uns nicht daran hindern soll, einen Blick auf das Handling zu werfen.

Wenn Sie in Excel die Schriftgröße einer Markierung ändern, wird dieser neue Wert auf die Markierung angewendet, ohne dass der Eintrag der ComboBox hinzugefügt wird.

Ein Beispiel. Eine Form hat ein Label namens lblTest und die ComboBox cmbFontSize.

```
Private Sub cmbFontSize_Click()
    lblTest.Font.Size = cmbFontSize.Text
End Sub

Private Sub cmbFontSize_KeyDown(ByVal KeyCode As _
    MSForms.ReturnInteger, ByVal Shift As Integer)
    If KeyCode = vbKeyReturn Then
        Call cmbFontSize_Click
    End If
End Sub
```

Die KeyDown-Prozedur ermöglicht das Abschließen der Bearbeitung durch die Return-Taste.

Im folgenden Beispiel wird der neue Eintrag dauerhaft in die Liste aufgenommen:

```
Private Sub cmbFontSize_Click()

'=====
'    FontSize zuweisen und evt. ergänzen

'=====
    Dim iItem As Long          'Zeiger in cmbFontSize
    'vorzeitiges Ende beim Laden
    If bStartingUp Then Exit Sub
    'neuer FontSize zuweisen
```

```

lblTest.Font.Size = cmbFontSize.Text
'Sub verlassen, wenn FontSize in ComboBox enthalten
If cmbFontSize.ListIndex > -1 Then Exit Sub
'Einträge untersuchen
For iItem = 0 To cmbFontSize.ListCount - 1
    'prüfen, ob FontSize kleiner als Wert(iItem)
    If CSng(cmbFontSize.Text) < _
        CSng(cmbFontSize.List(iItem)) Then
        'aktueller FontSize hinzufügen und Sub verlassen
        cmbFontSize.AddItem cmbFontSize.Text, iItem
    Exit Sub
End If
Next
'FontSize muss > als letzter Eintrag sein, also anhängen
cmbFontSize.AddItem cmbFontSize.Text
End Sub

```

Zwei Besonderheiten enthält diese Prozedur. Zum einen werden die Einträge und der Vergleichswert mit CSng in Single-Werte umgewandelt, damit 11 nicht zwischen 1 und 2 einsortiert wird. Das zweite betrifft die etwas merkwürdige Variable bStartingUp, die zum vorzeitigen Verlassen der Routine führt. Eine Variable dieses Namens benötigt man im Prinzip in jedem Dialog, der mit einer ComboBox oder ListBox ausgerüstet ist. Sie soll verhindern, dass die Ereignisroutinen abgearbeitet werden, wenn die Steuerelemente beim Initialisieren des Dialogs mit ihren Ausgangswerten gefüllt werden. Deshalb wird sie für die Dauer der Initialisierung auf True gesetzt:

```

Private Sub UserForm_Initialize()
    bStartingUp = True
    cmbFontSize.AddItem 8
    cmbFontSize.AddItem 9
    cmbFontSize.AddItem 10
    cmbFontSize.AddItem 12
    cmbFontSize.AddItem 14
    cmbFontSize.ListIndex = 2
    lblTest.Font.Size = cmbFontSize.Text
    bStartingUp = False
End Sub

```

8.7.5 Wie sortiert man die Einträge einer ComboBox

Bedauerlicherweise fehlt den ComboBoxes und ListBoxes in VBA eine Sort-Methode. Das ist in Word, PowerPoint und Outlook tragisch, aber zum Glück nicht in Excel.

Wenn die zukünftigen Einträge der ComboBox in einer Tabelle stehen, dann sortieren Sie diese Tabelle ganz einfach, bevor Sie die Werte von dort übernehmen. Und stehen Sie nicht in einer Tabelle (oder sonstigen, sortierbaren Datenquelle), dann schreiben Sie die Werte zuerst in eine Tabelle, sortieren diese und füllen die ComboBox aus dieser Liste.

Akrobaten sei empfohlen sich die diversen Sortieralgorithmen von Arrays zu Gemüte zu führen, die in vielen Büchern feilgeboten werden.

An dieser Stelle möchte ich die Verfahren nicht weiter vorstellen, denn Dialoge haben von Amts wegen nichts in irgendwelchen Datenquellen verloren. Datenquellen gehören gekapselt, worüber wir uns in Kapitel 10 unterhalten werden.

8.8 ListBox

Wenn Sie die ComboBox im Griff haben, wird Ihnen auch auf Anhieb eine Listbox zu Diensten sein.

Abbildung 8.27 zeigt die drei wesentlichen Erscheinungsformen von ListBoxes:

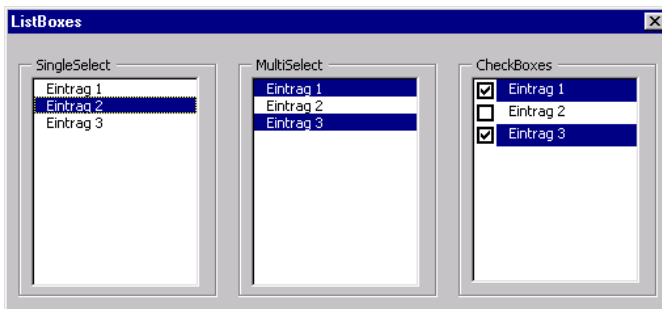


Abbildung 8.27: ListBoxes

Die Variante mit den CheckBoxes wird als MultiSelect-Variante behandelt, denn als SingleSelect ListBox macht sie schlicht und ergreifend keinen Sinn.

8.8.1 SingleSelect ListBoxes

Eigentlich ist diese Variante funktional der ComboBox gleichzusetzen. Das Handling ist in Kapitel 8.7 erklärt.

8.8.2 MultiSelect ListBoxes

Das Verhalten der ListBox wird im Wesentlichen mit der MultiSelect-Eigenschaft beeinflusst.

MultiSelect-Eigenschaft

Folgende Einstellungen sind bei dieser Eigenschaft möglich:

- ▼ 0 - `fmMultiSelectSingle` ermöglicht immer nur die Auswahl eines einzigen Eintrags. Ein Klick auf ein anderes Element hebt die Auswahl des vorangehenden auf. In dieser Einstellung verhält sich die `ListBox` wie eine `ListStyle-ComboBox`.
- ▼ 1 - `fmMultiSelectMulti` bedeutet, dass jeder Klick auf einen Eintrag diesen zusätzlich auswählt. Ein Klick auf ein bereits gewähltes Element hebt dessen Eintrag auf. `[Strg]`- und `[Umschalt]`-Taste werden ignoriert.
- ▼ 2 - `fmMultiSelectExtended` wendet bezüglich Steuerungs- und Umschalt-Taste die Markierungsverfahren an, wie sie aus dem Windows-Explorer bekannt sind. Mit der Umschalt-Taste werden auch alle Elemente hinzugenommen, die zwischen dem zuletzt markierten und dem aktuellem Element liegen. Mit der Steuerungs-Taste lässt sich die Auswahl eines bereits selektierten Elements aufheben.

ListStyle-Eigenschaft

Diese Eigenschaft verfügt über die beiden folgenden Einstellungen:

- ▼ 0 - `fmListStylePlain` stellt die Einträge wie in den beiden linken Beispielen der Abbildung 8.27 dar.
- ▼ 1 - `fmListStyleOption` zeigt am linken Rand eine Leiste mit `CheckBoxes` an, wie rechts in Abbildung 8.27 zu sehen ist.

Selected-Eigenschaft

Das sich hinter dieser Eigenschaft verbergende, nullbasierte `Boolean-Array` gibt den Auswahlzustand jedes einzelnen Eintrags wieder, wie im folgenden Beispiel zu sehen ist:

```
Private Sub cmdReadFiles_Click()

'=====
'    alle selektierten Dateien an ReadFile übergeben
'=====

    Dim iFile As Long          'Zeiger in ListBox
    'Einträge abarbeiten
    For iFile = 0 To lstFiles.ListCount
        'prüfen, ob Eintrag selektiert ist
        If lstFiles.Selected(iFile) Then
            'Aufruf der ReadFile mit selektierter Datei
```



```

        Call ReadFile(lstFiles.List(iFile))
    End If
Next
End Sub

```

8.8.3 Anbindung an Tabellen

Haben Sie die Möglichkeiten zur Anbindung von ComboBoxes und List-Boxes an Tabellen vermisst? Wenn Sie meinem Rat folgen, dann binden Sie bitte keine Steuerelemente an irgendwelche Datenquellen. Kapseln Sie stattdessen alle Zugriffe auf Daten in Klassen, wodurch Ihre Anwendung erst wartbar wird. Wenn Sie diesem Rat nicht folgen wollen, so wissen Sie inzwischen genügend über Steuerelemente, um sich das hierzu Nötige selbst zu erarbeiten.

8.9 CheckBox

Eine CheckBox kann die drei in Abbildung 8.28 gezeigten Zustände annehmen:



Abbildung 8.28: CheckBoxes

Die `Value`-Eigenschaft ist für die drei gezeigten Zustände verantwortlich:

- ▼ `False` bedeutet nicht selektiert.
- ▼ `True` bedeutet selektiert.
- ▼ `Null` bedeutet mit grauem Häkchen.

Dieser dritte Zustand, der durch `Null` repräsentiert wird, ist schon ein wenig ungewöhnlich und auch gelegentlich ein wenig unpraktisch. Versuchen Sie einmal dieses Wert `Null` in der Registry abzulegen:

```

chkTest.Value = Null
SaveSetting "Test", "Test", "Null", chkTest.Value

```

Noch eine Spur konfuser wird die Angelegenheit, wenn man feststellt, dass die CheckBox die Werte der VB-Konstanten `vbChecked (1)` und `vbGrayed (2)` korrekt umsetzt, aber in ihre internen Werte umwandelt.

8.10 OptionButton

Auch hier interessiert uns nur der Wert der `Value`-Eigenschaft, die `True` oder `False` aufweisen kann. `OptionButtons` organisieren sich selbstständig innerhalb ihres Containers, sodass immer nur ein `OptionButton` aktiviert sein kann. Als Container kommen in Frage die `UserForm`, der `Frame`, die einzelnen Seiten (Page-Objekte) eines `MultiPage`-Steuerelements oder aber eine Excel-Tabelle. Benötigen Sie innerhalb eines Containers eine weitere Unterteilung in unabhängige Gruppen von `OptionButtons`, so können Sie einen weiteren `Frame` einfügen und die `OptionButtons` dort hineinplatzieren.

8.11 CommandButton und ToggleButton

Die Entscheidung, welches von beiden Steuerelementen das der Stunde ist, hängt davon ab, ob es nur angeklickt wird oder ob daran auch eine Statusinformation gebunden sein soll.

Die meisten Schaltflächen werden wohl nur angeklickt und das `Click`-Ereignis enthält den entsprechenden Code.

Da der `ToggleButton` seinen Status beibehält, eignet er sich zu solchen Zwecken, wie in Abbildung 8.29 zu sehen. Der Dialog enthält fünf `ToggleButton`s, denen jeweils eine Farbe zugeordnet ist. Die aktivierte Farbe wird dem Label `lblStatus` als `BackColor` zugeordnet. Den Status, also gedrückt oder nicht gedrückt, gibt die `Value`-Eigenschaft mit den Werten `True` und `False` wieder.

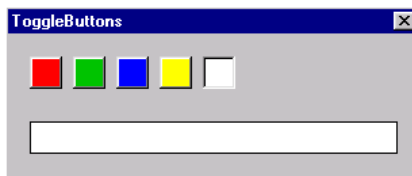


Abbildung 8.29: ToggleButtons

In der folgenden `UserForm_Initialize` wird der rote Button aktiviert.

```
Private Sub UserForm_Initialize()
    btnRed.Value = True
End Sub
```

Stellvertretend für alle Buttons die Click-Routinen des `btnYellow`:

```
Private Sub btnYellow_Click()
    Call SetButtons(btnYellow)
End Sub
```

Die SetButtons-Routine, die in der Click-Ereignisroutine aller Buttons aufgerufen wird, merkt sich den Zustand des übergebenen ToggleButtons und setzt danach die Value-Eigenschaft aller Buttons auf False und zum Schluss den Value des übergebenen Buttons auf den zwischengespeicherten Wert. Abschließend erhält das Label die Hintergrundfarbe des Buttons, falls dessen Value True ist, andernfalls die Hintergrundfarbe der UserForm.

```
Private Sub SetButtons(btnAct As ToggleButton)
    Dim bActValue As Boolean 'aktueller Wert des btnAct
    'aktueller Wert speichern
    bActValue = btnAct.Value
    'alle Buttons
    btnRed.Value = False
    btnBlue.Value = False
    btnGreen.Value = False
    btnWhite.Value = False
    btnYellow.Value = False
    btnAct.Value = bActValue
    If bActValue Then
        lblStatus.BackColor = btnAct.BackColor
    Else
        lblStatus.BackColor = Me.BackColor
    End If
End Sub
```

8.12 SpinButton und ScrollBar

Beide Steuerelemente unterstützen die Veränderung ganzzahliger Werte zwischen zwei definierbaren Grenzen per Maus. Der SpinButton bietet nur eine einstellbare Schrittweite, wohingegen die ScrollBar zwischen SmallChange und LargeChange unterscheidet.

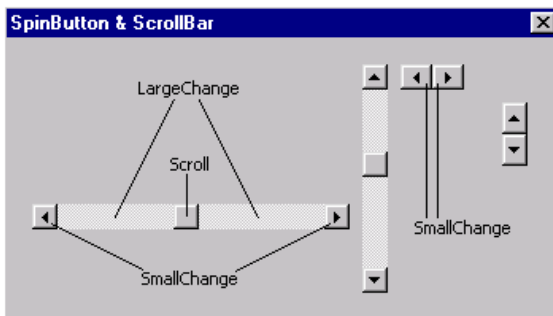


Abbildung 8.30: SpinButton & ScrollBar, Veränderung der Werte

Ein `LargeChange` oder ein `SmallChange` der `ScrollBar` machen sich als `Change`-Ereignis bemerkbar, eine `Scroll`-Änderung meldet sich per `Scroll`-Ereignis. Den aktuellen Wert erfahren wir jeweils von der `Value`-Eigenschaft.

`SmallChanges` des `SpinButtons` resultieren im gemeinsamen `Change`-Ereignis, denen jedoch ein `SpinDown`- bzw. `SpinUp`-Ereignis vorausgeht. Auch hier erfahren wir den aktuellen Wert über die `Value`-Eigenschaft.

SpinDown-Ereignis

Das `SpinDown`-Ereignis des `SpinButtons` meldet sich, sobald in einem horizontalen `SpinButton` der linke oder in einem vertikalen `SpinButton` der untere Button geklickt wird.

SpinUp-Ereignis

Das `SpinUp`-Ereignis des `SpinButtons` meldet sich, sobald in einem horizontalen `SpinButton` der rechte oder in einem vertikalen `SpinButton` der obere Button geklickt wird.

8.13 MultiPage und TabStrip

Der wesentliche Unterschied zwischen den beiden ist, dass die einzelnen Seiten eines `MultiPage` den darin enthaltenen Steuerelementen jeweils einen eigenen Container bieten, während ein `TabStrip` auf allen Registerseiten dieselben Steuerelemente anzeigt.

Nehmen wir das Beispiel des berühmten linksrheinischen Nichtschwimmers, der von Kritikern der Statistik gelegentlich angeführt wird.

Wollen Sie Ihre umfangreiche Sammlung von Steuerelementen übersichtlich gruppieren, so müssen Sie das `MultiPage`-Steuerelement verwenden:

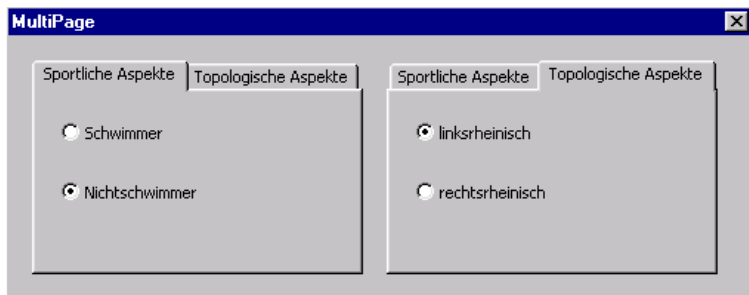


Abbildung 8.31: Der linksrheinische Nichtschwimmer im `MultiPage`

Kommt es jedoch darauf an, einen identischen Satz von Steuerelementen mit kontextbezogenen und somit jeweils anderen Werten zu präsentieren, so ist das TabStrip-Steuerelement Ihr Kandidat:

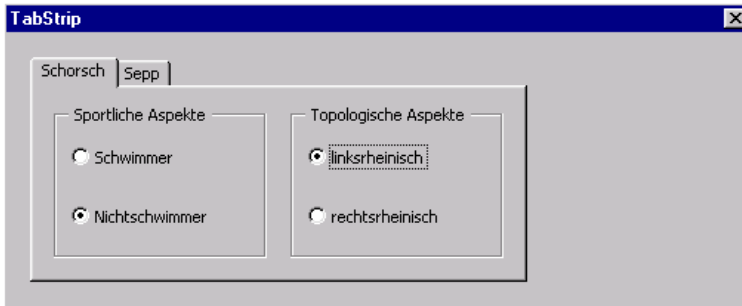


Abbildung 8.32: Der linksrheinische Nichtschwimmer im TabStrip

8.14 RefEdit

Lassen wir Bilder sprechen:

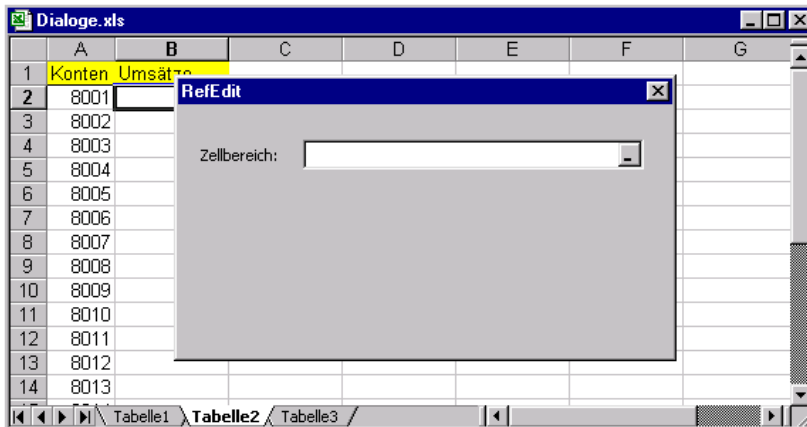


Abbildung 8.33: RefEdit vor Zellauswahl

Aus dem recht normal anmutenden Dialog der Abbildung 8.33 wird durch Anklicken eines Zellbereichs oder des DropDowns in der rechten Ecke des RefEdit ein kleines Fenster, wie wir es aus dem Funktionsassistenten von Excel kennen. Eine Zellmarkierung wird in diesem Zustand im RefEdit als Zellbezug mitprotokolliert.

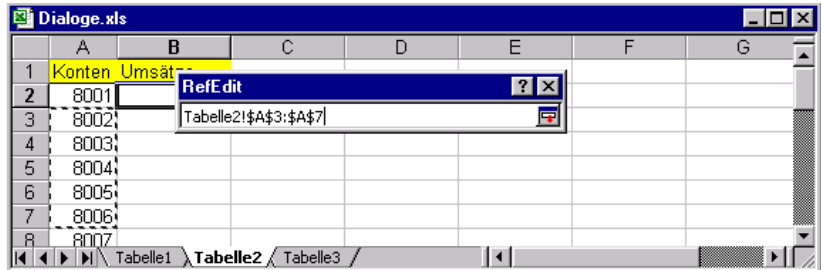


Abbildung 8.34: RefEdit während Zellauswahl

Den Inhalt des RefEdit-Steuerelements erfahren wir aus der Value-Eigenschaft. Im folgenden Beispiel wird der Inhalt des RefEdit an die Range-Methode des Worksheet-Objekts übergeben:

```
Private Sub cmdBalance_Click()
    Dim rngAccounts As Range
    On Error GoTo errHandler
    Set rngAccounts = shtAccounts.Range(refAccounts.Value)
    MsgBox rngAccounts.Cells.Count
    Exit Sub

errHandler:
    If Err.Number = 1004 Then

        End If
    End Sub
```

8.15 Steuerelemente in Tabellen

Im Prinzip lassen sich alle Steuerelemente auch in Tabellen einfügen. Zwar fehlen auf der Steuerelement-Toolbox Frame, MultiPage, TabStrip und RefEdit, aber über die Schaltfläche Weitere Steuerelemente kommt man an alle registrierten Typen ran, die sich dann auch artig in die Tabelle platzieren lassen. Diese liefern auch die gewünschten Ereignisse.

Verwunderung stellt sich aber dann ein, wenn die vier oben genannten Steuerelemente offenbaren, dass sie ihre Fähigkeit, Container für andere Steuerelemente zu sein, eingebüßt haben. Und genau diese Fähigkeit benötigen wir, wenn zwei unabhängig voneinander arbeitende Gruppen von OptionButtons in einer Tabelle ihren Dienst tun sollen.

Was tun? Prüfen Sie bitte, ob ein Dialog angebracht ist. Mir fällt kein Argument ein, was gegen einen Dialog sprechen würde.

Wenn Sie dennoch zwei unabhängige Gruppen von `OptionButtons` benötigen, müssen Sie eben auf die Steuerelemente der Formular-Symbolleiste zurückgreifen. Doch es erweist sich als ausgesprochen schwierig, einen Objektverweis zu einem existierenden `OptionButton` herzustellen. Der einzige Weg, der sich als halbwegs gangbar herausstellte, war Position und Größe des Steuerelements zu merken, dieses dann zu löschen und anschließend mit

```
Set optX = WorksheetObject.Shapes.AddFormControl(...)
```

neu zu erzeugen.

Ein weiterer Weg bietet sich aber noch an. Diese alten Steuerelemente bieten im Register Steuerung des Menüs Steuerelement Formatieren eine Zellverknüpfung. Ändert sich der Wert des Steuerelements, so erscheint in der dort angegebenen Zelle die Ordnungszahl des `FormControlOptionButtons`. Das ist die mit eins beginnende laufende Nummer des `OptionButtons` innerhalb seines Containers. Im Kontextmenü des Steuerelements kann man über Makro zuweisen das Steuerelement einer `Public Sub` zuordnen, in der wiederum die in der Zellverknüpfung angegebene Zelle ausgewertet werden kann.

Mir graust es nicht wenig, wenn ich den letzten Absatz noch einmal durchlese. Wenn es irgendwie vermeidbar ist, dann vermeiden Sie es.

Eine Eigenschaft der Steuerelemente aus der Steuerelement-Toolbox ist noch erwähnenswert:

Placement-Eigenschaft

Die `Placement`-Eigenschaft beeinflusst das Verhalten des Steuerelements beim Ändern von Größe oder Position des unter dem Steuerelement liegenden Zellbereichs mit folgenden Werten:

- ▼ 1 - `xlMoveAndSize` verklebt das Steuerelement mit dem darunterliegenden Zellbereich, sodass ein Verschieben dieser Zellen das Steuerelement ebenfalls verschiebt (`Move`) und eine Größenänderung des Zellbereichs auch die Größe des Steuerelements verändert (`Size`).
- ▼ 2 - `xlMove` hat nur ein Verschieben zur Folge.
- ▼ 3 - `xlFreeFloating` trennt das Steuerelement in Größe und Position von der sie beherbergenden Tabelle.

TakeFocusOnClick-Eigenschaft

Obwohl sie in Excel 2000 entschärft wurde, sei hier noch einmal erwähnt (wir sprachen in Kapitel 3 bereits davon), dass diese Eigenschaft auf `False` gesetzt werden muss, wenn Ihre Arbeitsmappe auch unter Excel 97 laufen soll.





Fehlerbehandlung

9

Kapitelüberblick

9.1	Am Beispiel des Öffnens einer Arbeitsmappe	443
9.1.1	Fehlervermeidung	445
9.1.2	Methodiken der Fehlerbehandlung	448
9.1.3	Wie geht's bei uns weiter?	450
9.2	Noch etwas Theorie	456
9.2.1	Fehlerbehandlungshierarchie	456
9.2.2	Vorgehensweisen im ErrorHandler	458
9.2.3	Das Err-Objekt	459
9.2.4	Aktivieren von Fehlerbehandlungsroutinen	461
9.3	Typen von Fehlerbehandlungsroutinen	462
9.4	Zentrale Fehlerklasse clsError	466
9.4.1	Klasseninitialisierung	467
9.4.2	Fehlerausgabe	469
9.4.3	Fehlerlogbuch	470
9.4.4	Infoausgabe	472
9.4.5	Fazit	474
9.5	Fehlervermeidung	474
9.6	Gängige Fehler und ihre Behandlung	475

Fehlerbehandlung ist in VBA vielfach ein Stiefkind, könnte man meinen. Vielleicht liegt das an dem Haupteinsatzgebiet von (zumindest) Excel-VBA, das unter dem Motto zu stehen scheint: „Wir haben da ein Problem; mach doch bitte mal schnell ...“. Wenn es sich hierbei um isolierte Aufgabenstellungen handelt, die sich vollständig innerhalb einer Arbeitsmappe abspielen, mag das sogar noch angehen. Hier setzen Sie nur Ihren Ruf aufs Spiel. Anders verhält sich die Sache allerdings, wenn Ihre Anwendung Teil einer heterogenen Datenlandschaft ist. Diese häufig anzutreffende Umgebung Ihres Programms setzt voraus, dass Sie sich auf potenziell fehlerträchtige Situationen einstellen müssen. Hierunter fallen lesende oder schreibende Zugriffe auf Datenbanken, andere Excel-Arbeitsmappen und auf verschiedenste Textdateien. Die Fehlersituationen erstrecken sich von »Laufwerk nicht vorhanden«, »Datei bereits exklusiv geöffnet« bis zu Dateifehlern selbst. Es wäre auch nicht das erste Mal zu beobachten, dass aufgrund einer außergewöhnlichen Situation der allseits geliebte Dr. Watson vorbeischaud und dem ganzen Spuk ein Ende bereitet, ohne dass Sie die Chance hatten, Ihre in Bearbeitung befindlichen Daten zu speichern.

Nun hat sich Excel 97 bereits mit einer Sprachmaschine und einer Entwicklungsumgebung präsentiert, die dem professionellen System Visual Basic recht nahe kamen. Dieser Trend hat sich mit Excel 2000 auch fortgesetzt. Da nun die Komponenten der Fehlerbehandlung in VBA denen von VB entsprechen, sollten wir uns auch die Konzepte zu eigen machen, die dort Einzug gehalten haben.

Auf den ersten Blick ist es verblüffend, dass eine Fehlerbehandlung beim Codieren zu Anfang unwesentlich mehr Aufwand verursacht als das Weglassen derselben. Später kann sich dieses Verhältnis sogar so verändern, dass eine geordnete Fehlerbehandlung Zeit einspart. Dies hängt in erster Linie damit zusammen, dass Sie Ihren Fehlerbehandlungscode, der einer kontinuierlichen Optimierung unterliegt (man lernt ja schließlich dazu!), zu großen Teilen in neue Projekte übernehmen können.

Fehlerbehandlung in Komponenten

Es gibt noch eine Reihe von Aspekten der Fehlerbehandlung, die im Zusammenhang mit Komponenten von Interesse sind. Diese Überlegungen sind in diesem Kapitel nicht berücksichtigt, werden aber gründlich in Kapitel 10 behandelt, wo es genau um diese Themen geht.

Bevor Sie also den in diesem Kapitel diskutierten Code in Ihren Projekten einsetzen, sollten Sie sich die Ausführungen in Kapitel 4.10 – Fehlerbehandlung in Klassen ansehen. In einem Punkt wird eine Änderung an der Ausgaberroutine vorzunehmen sein, um eine Maskierung der Fehlernummer vorzunehmen.

9.1 Am Beispiel des Öffnens einer Arbeitsmappe

Nähern wir uns dem Thema anhand eines konkreten Beispiels, das sich quer durch das Kapitel zieht.

Wenn Ihre Applikation auf andere Excel-Arbeitsmappen zugreifen muss, so ist das nicht mit einer Programmzeile abgetan. Ist die Datei bereits geöffnet und wurde nach dem Öffnen verändert, so würde dem Anwender folgende MessageBox ins Antlitz starren:

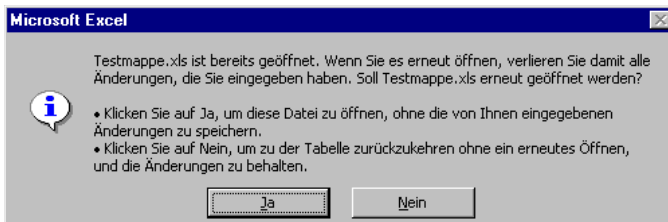


Abbildung 9.1: Konflikt beim Öffnen einer Arbeitsmappe

Klickt der Anwender auf Ja, so sind die bislang in dieser Datei vorgenommenen Änderungen verloren. Schließt er den Dialog jedoch mit Nein, so sieht er sich ohne Fehlerbehandlung folgendem Dialog gegenüber:

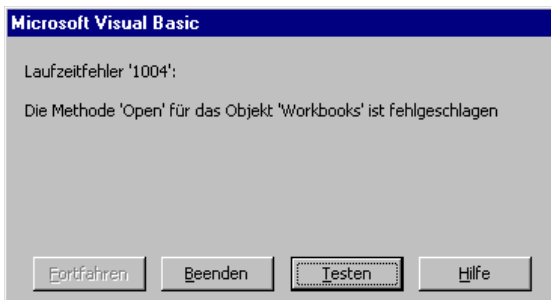


Abbildung 9.2: Laufzeitfehler 1004

Es leuchtet ein, dass der Anwender hiermit in aller Regel überfordert ist. Wählt er nun sogar die Schaltfläche Testen, so bahnt sich eine Katastrophe an, sofern das VBA-Projekt nicht passwortgeschützt ist.

Doch damit nicht genug, denn allein beim Zugriff auf eine Arbeitsmappe muss mit einer Reihe von weiteren Situationen gerechnet werden, die ohne Fehlerbehandlung zu solch unerfreulichen Situationen führen können.

Hier übrigens die Programmzeilen, die zu dem oben stehenden Ergebnis führen:

```

Sub OpenTest()
    Dim wbk As Workbook
    Dim strFile as String
    strFile = ThisWorkbook.Path & "\Testmappe.xls"
    Set wbk = Workbooks.Open(strFile)
End Sub

```

Unter dem Paradigma einer modularen Programmierung empfiehlt es sich, das Öffnen bzw. Zurückgeben der gewünschten Arbeitsmappe zumindest einer separaten Funktion anzuvertrauen. Im folgenden Kapitel über Klassenprogrammierung werden wir sogar noch einen Schritt weiter gehen und das Zurückgeben einer Arbeitsmappe als Methode einer eigenen Klasse implementieren. Fehlerbehandlung und Klassenprogrammierung haben mehr gemein, als man auf den ersten Blick hin anzunehmen geneigt ist. Dennoch gehen beide im Hinblick auf die Realisierung von verlässlichen und somit leicht wiederverwendbaren Code-Komponenten im Grunde genommen Hand in Hand.

Doch zurück zu unserem Thema.

Wir werden eine Funktion erstellen, die uns die gewünschte Arbeitsmappe als *Workbook*-Objekt zurückgibt oder, wenn das nicht möglich sein sollte, eine entsprechende Fehlermeldung erzeugt. Damit sie das auch tun kann, benötigt die Funktion Pfad nebst Laufwerk und den Namen der Arbeitsmappe. Nun ist eine *geöffnete* Arbeitsmappe allerdings nur unter ihrem Dateinamen bekannt. Wollen wir also prüfen, ob die Arbeitsmappe bereits geöffnet ist, so benötigen wir dazu den isolierten Dateinamen. Um in der Funktion den kompletten *String* nicht mühsam zerlegen zu müssen, übergeben wir die beiden Komponenten getrennt. Der Torso unserer Aufgabe sieht demnach so aus:

```

Sub OpenTest()
    Dim wbk As Workbook
    Dim strFile As String
    Dim strPath As String
    strPath = ...
    strFile = ...
    Set wbk = GetWBK(strPath, strFile)
    ...
End Sub

Function GetWBK(ByVal strPath As String, _
                ByVal strFile As String) As Workbook
    ...
End Function

```

Nach dem Skizzieren der Aufgabenstellung nähern wir uns nun den einzelnen Konzepten.

9.1.1 Fehlervermeidung

Fehlervermeidung ist in der Tat der erste Schritt zu einer guten Fehlerbehandlung. Wie wir bereits sahen, führt das Öffnen einer Arbeitsmappe dann zu Konflikten, wenn sie oder – weniger wahrscheinlich – eine Datei gleichen Namens bereits geöffnet ist. Also werden wir alle geöffneten Workbooks innerhalb einer For-Next-Schleife daraufhin untersuchen.

```
For i = 1 To Workbooks.Count
    If Workbooks(i).Name = strFile Then
        If Workbooks(i).Path = strPath Then
            Exit For
        Else
            'Nun haben wir ein Problem.
        End If
    End If
Next
```

Wenn eine Arbeitsmappe des übergebenen Namens `strFile` bereits geöffnet ist, überprüft der folgende If-Ausdruck, ob deren Pfad mit dem gesuchten `strPath` übereinstimmt. Wenn ja, verlassen wir die Schleife, wenn nein, haben wir ein Problem.

Um mit diesem Fall umzugehen, könnten wir eine Variable erzeugen und in ihr eine Beschreibung des Problems ähnlich einer Fehlermeldung hinterlegen, etwa „Eine Arbeitsmappe gleichen Namens ist bereits geöffnet, wobei es sich allerdings nicht um die angeforderte Datei handelt.“ Danach verlassen wir die Funktion mit `Exit Function` und reiben unsere Hände in Unschuld ...

Vermutlich ist die betreffende Arbeitsmappe in diesem Kontext aber nicht mit Intention geöffnet, sondern eher zufällig. Da wir jedoch keine sichere Aussage treffen können, wäre das die Gelegenheit, den Anwender entscheiden zu lassen. Drei Ausgänge birgt diese Situation allerdings in sich:

- ▼ Die Arbeitsmappe wird gespeichert und geschlossen.
- ▼ Die Arbeitsmappe wird ohne zu speichern geschlossen.

In beiden Fällen würde das Programm fortgeführt werden. Oder aber:

- ▼ Das Programm wird an dieser Stelle abgebrochen.

Die nun anstehende Entscheidung lässt sich mit einer `MessageBox` realisieren, die mit den Schaltflächen Ja, Nein und Abbrechen zu bestücken

wäre, also `vbYesNoCancel`. Die Erzeugung des Arguments `Prompt` der `MessageBox` schenken wir uns hier, aber so etwa könnte sie aussehen:

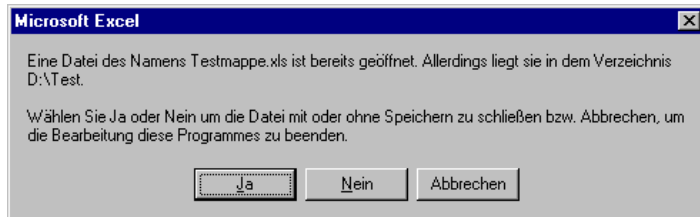


Abbildung 9.3: `MessageBox` im Falle eines Dateinamenkonflikts

So könnte der `Else`-Zweig (hervorgehoben) aussehen, der diesen Dialog auf den Bildschirm bringt und den Wünschen des Anwenders gemäß verfährt:

```
For i = 1 To Workbooks.Count
    If Workbooks(i).Name = strFile Then
        If Workbooks(i).Path = strPath Then
            Exit For
        Else
            strMessage = "Eine Datei des Namens ..."
            intReturn = MsgBox(strMessage, vbYesNoCancel)
            Select Case intReturn
            Case vbYes
                Workbooks(i).Close savechanges:=True
            Case vbNo
                Workbooks(i).Close savechanges:=False
            Case vbCancel
                Exit Function
            End Select
        End If
    End If
Next
```

Nun wartet aber gleich das nächste Problem auf uns. Wie Sie sicherlich noch wissen, kann das vorzeitige Verlassen einer `For-Next`-Schleife recht einfach dadurch ermittelt werden, dass geprüft wird, ob die Zählvariable größer ist als die obere Grenze der `For`-Zeile, also `Workbooks.Count` in unserem Beispiel. Es wird Ihnen auch noch geläufig sein, dass die obere Grenze einer `For-Next`-Schleife nur beim ersten Mal ermittelt wird. Wenn wir also großzügig ein `Workbooks(i).Close` absetzen, reduziert sich `Workbooks.Count` um 1. Zwangsläufig wird im letzten Durchlauf `i` einen Wert annehmen, der um 1 größer ist als `Workbooks.Count`, und somit einen Fehler produzieren, der sich dann so äußert:

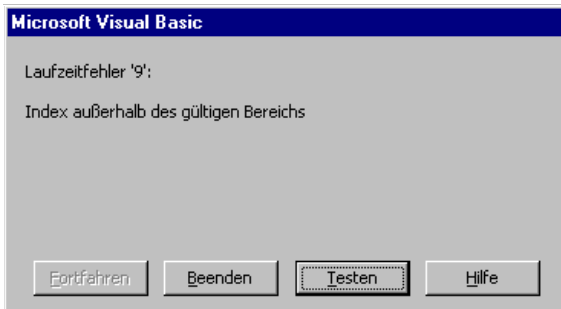


Abbildung 9.4: Der Laufzeitfehler nach `Workbooks(i).Close`

Es stünden noch eine Reihe von Vorgehensweisen zur Verfügung, um dieses Problem zu umgehen. Die einfachste in solchen Situationen ist allerdings, den Zähler rückwärts laufen zu lassen. Auch beim Durchlaufen einer Excel-Tabelle, aus der fallweise Zeilen gelöscht werden müssen, ist diese Vorgehensweise geboten, da Ihnen andernfalls die jeweils unmittelbar folgende Zeile durch die Lappen geht. Die `For`-Zeile müsste also dahingehend geändert werden:

```
For i = Workbooks.Count To 1 Step -1
```

Die Zählvariable einer `For-Next`-Schleife ist – sofern die Schleife nicht vorzeitig verlassen wurde – betragsmäßig um den bei `Step` angegebenen Wert größer als die obere Grenze bei positivem `Step` bzw. kleiner als die untere Grenze bei negativem `Step`. In unserem Beispiel müsste `i` also den Wert 0 vorweisen, wenn die Schleife vollends durchlaufen worden wäre. Und dieser Fall würde bedeuten, dass die Arbeitsmappe noch zu öffnen wäre. Im Umkehrschluss heißt das allerdings, ist $i > 0$, so ist die Datei bereits geöffnet und muss nun lediglich dem Funktionsnamen `GetWBK` zugewiesen werden:

```
If i > 0 Then
    Set GetWBK = Workbooks(i)
    Exit Function
End If
```

Diese vier Zeilen werden nach der `Next`-Anweisung ergänzt, und damit sind wir in unserem Beispiel mit den Bordmitteln der Fehlervermeidung vorläufig am Ende angelangt; der Rest bedarf der Sprachelemente der Fehlerbehandlung.

Dennoch haben wir mit unseren immerhin 25 Programmzeilen in der Funktion `GetWBK` schon einiges geleistet. Ist die gewünschte Datei bereits geöffnet, wird sie zurückgegeben. Ist eine Datei identischen Namens mit

anderem Pfad geöffnet, so wird diese nach Rücksprache mit dem Anwender geschlossen oder aber der Abbruch des Programms eingeleitet.

9.1.2 Methodiken der Fehlerbehandlung

Im nun folgenden Teil der Funktion `GetWBK` müssen wir uns auf folgende Fehlersituationen einrichten:

- ▼ Das Laufwerk existiert nicht.
- ▼ Das Laufwerk ist nicht betriebsbereit (Wechseldatenträger ohne Medium, z. B. Zip, CD oder Diskette).
- ▼ Das Verzeichnis existiert nicht.
- ▼ Das Verzeichnis existiert zwar, aber die Datei nicht (zumindest dort nicht).
- ▼ Die Datei ist beschädigt und kann daher nicht geöffnet werden.
- ▼ Die Datei ist bereits von einem anderen Prozess mit Schreibrechten geöffnet.

Gerade der letzte Fall könnte auch bedeuten, dass wir die Datei eben schreibgeschützt öffnen, da wir möglicherweise nur lesend darauf zugreifen wollen. Doch auf diesen Punkt werden wir noch eingehen. Zunächst schauen wir uns die Methodiken der Fehlerbehandlung einmal im Überblick an.

On Error Resume Next

Diese Anweisung sagt nur aus, dass im Falle einer Fehler produzierenden, aber nicht notwendigerweise fehlerhaften Programmzeile die nachfolgende Zeile abgearbeitet werden soll. Zumeist erfordert diese Vorgehensweise einen Codeblock, der so aussehen könnte:

```
On Error Resume Next
... potentiell fehlerhafte Zeile
If Err.Number <> 0 Then
    MsgBox Err.Description, vbExclamation
End If
On Error GoTo 0
... restlicher Code
End Sub ' bzw. Function
```

Wir haben es hier mit einem eigenen Fehlerobjekt des Namens `Err` zu tun, dessen Eigenschaft `Number` die Fehlernummer des Objektmodells beinhaltet. Daneben kommt die Eigenschaft `Description` zum Einsatz, die in einer `MessageBox` präsentiert wird.

Je nach Natur und Schwere des Fehlers muss die Prozedur oder Funktion auch mit `Exit Sub` bzw. `Exit Function` verlassen werden.

In einer Applikation sollte diese Anweisung sparsam eingesetzt werden.

On Error GoTo ErrorHandler

Diese Vorgehensweise, die auf den ersten Blick keine gravierenden Vorzüge in sich birgt, folgt zumeist folgendem Muster:

```
On Error GoTo ErrHandler
... Code
Exit Sub ' bzw. Function
ErrorHandler:
'Fehlerbehandlung
End Sub ' bzw. Function
```

Der Begriff `ErrHandler` ist ein frei gewählter Ausdruck, der dem Zweck allerdings recht nahe kommt. Wie funktioniert dieses Konstrukt nun?

Innerhalb der Prozedur oder Funktion muss nach dem rein funktionalen Codeblock ein durch Doppelpunkt abgeschlossener Begriff stehen, in unserem Falle `ErrorHandler:`. Dieses so genannte Label leitet die eigentliche Fehlerbehandlung ein, die ein wenig an das BASIC der 80er-Jahre erinnert. Die Anweisung

```
On Error GoTo ErrHandler
```

definiert diesen Teil nun als benannte Ansprungsadresse. Kommt es also zu einem Fehler, so wird die Programmausführung in der dem Label folgenden Zeile ausgeführt. Damit nun die reguläre Programmausführung nicht generell in den Fehlercode hineinläuft, muss vor dem Label ein `Exit Sub` bzw. `Exit Function` stehen.

Jeder innerhalb der Prozedur oder Funktion auftretende Fehler bricht an der den Fehler produzierenden Zeile ab und springt in das Label. Kommt es dort zur Ausführung eines `Resume Next`, so wird der Code nach der fehlerhaften Zeile fortgeführt; andernfalls wird die Bearbeitung der Prozedur oder Funktion abgebrochen. Das leuchtet ein, denn am Ende des Codes steht `End Sub`, `End Function` oder aber auch `End Property`.

Wo liegt nun aber der Vorzug dieser Vorgehensweise? Zum einen könnte man anführen, dass der Code etwas kompakter und auch übersichtlicher wird. Doch das ist längst nicht alles. Richtig ertragreich wird die Sache, wenn eine ohnehin anzustrebende Modularisierung vorliegt, wobei es unerheblich ist, ob die betreffenden Prozeduren oder Funktionen im selben Kontext liegen, in Module ausgelagert sind oder sich in Klassenmodulen oder gar eigenen DLLs befinden.

Ein Beispiel. Die folgende Prozedur ruft eine Methode einer externen DLL namens `ProduceError.dll` auf, die als einzige Programmzeile die Generierung eines Fehlers zum Inhalt hat:

```
Public Sub xyz()
    Dim cError As New ProduceError.clsError
    On Error GoTo ErrHandler
    cError.ProduceError
    Exit Sub
ErrHandler:
    MsgBox Err.Description
End Sub
```

Das Ergebnis ist, dass das Label `ErrorHandler` angesprungen wird und die Eigenschaft `Description` des `Error` Objekts zur Anzeige kommt. Hier wird nun das Geheimnis gelüftet, denn die einzige Prozedur der einzigen Klasse der in VB geschriebenen DLL sieht so aus (stören Sie sich im Moment bitte nicht an dem etwas merkwürdigen Ausdruck `vbObjectError + 9999`):

```
Public Sub ProduceError()
    Err.Raise vbObjectError + 9999, , "Ein Fehler ..."
End Sub
```



Das ist des Pudels Kern bei der Verwendung von Labels in der Fehlerbehandlung: Alle der jeweiligen Prozedur oder Funktion untergeordneten Programmkonstrukte können in einer Programmzeile einen aussagekräftigen, prozessweiten Fehler erzeugen, der zentral im `ErrorHandler` verarbeitet werden kann.

Das muss man sich in aller Ruhe auf der Zunge zergehen lassen.

9.1.3 Wie geht's bei uns weiter?

Zu den beiden kurz skizzierten Methodiken ist noch einiges zu sagen, was wir allerdings für einen Moment aufschieben wollen. Kehren wir stattdessen zu unserem Beispiel zurück.

Bisher haben wir folgenden Stand:

```
Function GetWBK(strPath, strFile) As Workbook
```

```
    For i = Workbooks.Count To 1 Step - 1
        'Wenn Datei bereits geöffnet, dann Exit For
        'Wenn falsche Datei gleichen Namens geöffnet, dann
        'Datei schließen oder Funktion vorzeitig verlassen
    Next
```

```

If i > 0 Then 'Gesuchte Datei unter den geöffneten
    'Datei als GetWBK zurückgeben und Funktion verlassen
End If

```

End Function

Bevor wir nun weitermachen, müssen wir uns bezüglich des per `strPath` übergebenen Pfades der zu öffnenden Arbeitsmappe noch ein paar Gedanken machen. Wenn wir die `Path`-Eigenschaft eines `Workbook`-Objekts auslesen, so erhalten wir den Pfad ohne den abschließenden „\“. Wir könnten nun festlegen, dass der Pfad von dem aufrufenden Programmteil generell ohne „\“ übergeben werden muss. Das ist jedoch brandgefährlich, da wir oder auch die, die unsere Funktion ebenfalls nutzen wollen, diese Vereinbarung früher oder später vergessen. Dann aber geht bereits die Überprüfung des Dateipfades in der `For-Next`-Schleife möglicherweise schief.

Im bisherigen Programmteil setzen wir also voraus, dass `strPath` nicht durch einen „\“ abgeschlossen ist, was wir durch diesen Ausdruck, der vor der `For-Next`-Schleife eingefügt werden muss, sicherstellen können:

```

If Right(strPath, 1) = "\" Then _
    strPath = Left(strPath, Len(strPath) - 1)

```

Am Ende unseres bisherigen Codes können wir `strPath` nun mit dem abschließenden „\“ versehen:

```
strPath = strPath & "\"
```

Laufwerk und Verzeichnis prüfen

Der einfachste Fall wäre ein Testzugriff auf den Pfad, wozu sich die `Dir`-Funktion anbieten würde. Ist das Laufwerk nicht vorhanden, so meldet sich der Fehler 68 mit dem Text „Gerät nicht verfügbar“. Ich liebe diesen Text. Er könnte einer Ballade von Wieland oder Goethe entnommen sein. Ist zwar das Laufwerk, aber offensichtlich kein Medium vorhanden oder beispielsweise ein CD-Laufwerk geöffnet, so meldet sich Fehler 71 mit „Datenträger nicht bereit“. Der geht gerade noch, aber den Text „Gerät nicht verfügbar“ kann man nun wirklich niemandem zumuten. Bevor wir uns nun den stilistischen Elementen zuwenden, widmen wir uns noch kurz der Überprüfung des Verzeichnisses. Die `Dir`-Funktion kann mit dem Attribut `vbDirectory` veranlasst werden zu prüfen, ob eine Zeichenkette ein gültiges Verzeichnis darstellt. Gibt sie eine Nullzeichenfolge zurück, so ist das Verzeichnis nicht vorhanden. Mit

```
If Dir(strPath, vbDirectory) = "" Then
    ...
End If
```

haben wir zum einen diesen Fall abgehandelt, aber auch gleichzeitig den beiden erstgenannten Fällen die Chance eingeräumt sich bemerkbar zu machen. Damit unser noch unbehandelter Code nicht mit einem Laufzeitfehler stehen bleibt, schreiben wir `On Error GoTo ErrorHandler` davor und ergänzen am Ende der Funktion diesen `ErrorHandler`:

```
Function GetWBK(strPath, strFile) As Workbook

    For i = Workbooks.Count To 1 Step - 1
        'Wenn Datei bereits geöffnet, dann Exit For
        'Wenn falsche Datei gleichen Namens geöffnet, dann
        'Datei schließen oder Funktion vorzeitig verlassen
    Next

    If i > 0 Then 'Gesuchte Datei unter den geöffneten
        'Datei als GetWBK zurückgeben und Funktion verlassen
    End If

    strPath = strPath & "\"

    On Error GoTo ErrHandler
    If Dir(strPath, vbDirectory) = "" Then
        ...
    End If
    ...
    Exit Function

ErrHandler:
    ...
End Function
```

Wir haben es also jetzt mit zwei Arten von Fehlern zu tun. Die erste besteht aus realen Fehlern, die beim Zugriff auf Laufwerke entstehen (Fehler 68 oder 71). Die zweite beinhaltet potenzielle Fehlersituationen, die bereits im Vorfeld erkannt werden (in unserem Beispiel das Nichtvorhandensein eines Verzeichnisses). Dieser potenzielle Fehler tritt gar nicht auf. Somit können wir uns nicht darauf beschränken, in der Fehlerbehandlung das den aktuellen Fehler beinhaltende `Err`-Objekt abzufragen, denn er tritt ja nicht auf.

Wenn er also nicht freiwillig erscheint, so können wir nachhelfen. Mit der `Raise`-Methode des `Err`-Objekts können wir bereits definierte Fehler er-

zeugen, aber auch welche erfinden. Fehler 76 „Pfad nicht gefunden“ entspricht jedoch dem, was wir benötigen:

```
If Dir(strPath, vbDirectory) = "" Then
    Err.Raise 76
End If
```

Ziel unserer Anstrengungen ist es, in der Fehlerbehandlung einfach und übersichtlich alle Fehler zu verarbeiten. Würden wir zwischen aufgetretenen und vorher aufgedeckten Fehlern unterscheiden, müssten wir vermutlich mit Hilfsvariablen arbeiten und in der Fehlerbehandlung mit Ausnahmen operieren. So aber können wir uns auf die Auswertung des Fehlerobjekts `Err` beschränken.

Nach gleichem Muster verfahren wir jetzt bei dem Fall der nicht vorhandenen Datei. Wir müssen nun also die folgende `If`-Anweisung ergänzen:

```
If Dir(strPath & strFile) = "" Then
    Err.Raise 73 'Datei nicht gefunden
End If
```

Nun steht uns noch eine kleine Hürde bevor, denn wir müssen prüfen, ob unsere Datei bereits von einem anderen Prozess geöffnet ist. Prozess meint hier irgendein eigenständiges Programm auf dem aktuellen Rechner oder irgendeinem Rechner im Netz mit entsprechenden Zugriffsrechten auf die Datei. Die `Open`-Methode des `Workbook`-Objekts öffnet eine bereits von einem anderen Prozess benutzte Datei, ohne einen Fehler zu produzieren. Selbst mit dem Argument `readonly:=False` können wir die `Open`-Methode nicht davon abhalten, die Datei zu öffnen.

Aus dieser Patsche hilft uns das testweise Öffnen der Datei mittels der `Open`-Anweisung im Modus `Binary` und mit der Dateisperre `Lock Write`. Vorher lassen wir uns noch den obligatorischen `Datei-Handle` zuweisen:

```
hFile = FreeFile
Open strPath & strFile For Binary Lock Write As #hFile
Close hFile
```

Da wir die `Open`-Anweisung nur benutzen, um einen Fehler zu erzeugen, wird die Datei gleich wieder geschlossen. `Open` erzeugt uns in dieser Parametrierung den Fehler 70 „Zugriff verweigert“, wenn die Datei bereits in Benutzung ist.

Nun verbleibt lediglich der Fall einer defekten Datei. Da wir ihn schlecht im Vorfeld abfangen können, lassen wir uns von der `Open`-Methode einfach überraschen:

```
Set GetWBK = Workbooks.Open(strPath & strFile, , False)
```

Danach folgt nur noch `Exit Function` und der `ErrHandler`. Dieser wiederum wird in allen bislang behandelten Fällen mit einem geladenen Fehlerobjekt aufgerufen und sieht so aus:

`ErrHandler`:

```
intErrNr = Err.Number
Select Case intErrNr
Case 68 'Gerät nicht verfügbar
    strMessage = "Der angegebene Datenträger ..."
Case 70 'Zugriff verweigert
    strMessage = "Die angegebene Arbeitsmappe ..."
Case 71 'Datenträger nicht bereit
    Do
        strMessage = "Der Datenträger ist nicht ..." & _
            "Bitte überprüfen Sie ..."
        intReturn = MsgBox(strMessage, vbOKCancel)
        If intReturn = vbOK Then
            Resume
        Else
            strMessage = "Der Datenträger ist nicht ..."
            Exit Do
        End If
    Loop
Case 73 'Datei nicht gefunden
    strMessage = "Die Datei " & strFile & " ist ..."
Case 76 'Pfad nicht gefunden
    strMessage = "Das Verzeichnis " & strPath & " ..."
Case Else 'Fehler beim Öffnen
    strMessage = "Die Datei konnte nicht ..."
End Select
Err.Raise intErrNr, "clsWBK:GetWBK", strMessage
End Function
```

In der Variablen `intErrNr` legen wir die Fehlernummer ab, die nachfolgend in der `Select Case` Struktur untersucht wird. Dort weisen wir der Variablen `strMessage` sinnvolle Ausgabetexte zu. Am Ende wird der betreffende Fehler erneut erzeugt und mit dem Namen der Funktion als Eigenschaft `Source` und unserer `strMessage` als `Description` an die aufrufende Prozedur weitergeleitet. Der Fehler 71 „Datenträger nicht bereit“ stellt insoweit eine Ausnahme dar, da er möglicherweise vom Anwender durch Schließen des Laufwerks behoben werden kann. In einer Schleife wird die folgende `MessageBox` so lange wiederholt (OK-Schaltfläche), bis der Anwender sie mit Abbrechen verlässt:

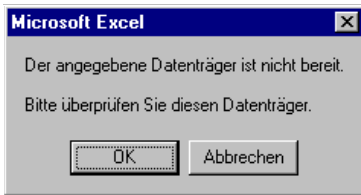


Abbildung 9.5: MessageBox in ErrorHandler

Unsere Funktion ist inzwischen auf 70 Zeilen angewachsen und hat eine Menge Arbeit gemacht. Aber, und das wurde ja eingangs dieses Kapitels gesagt, sie wird uns zukünftig auch eine Menge Arbeit ersparen. Sie ist getestet und funktioniert und kann zukünftig bedenkenlos eingesetzt werden, wenn es mal wieder darum geht, eine Arbeitsmappe zu öffnen. Im Kapitel Klassenprogrammierung wird diese Funktion in eine Klasse integriert, die Sie dann nur noch Ihren zukünftigen Projekten hinzufügen.

Obwohl die Funktion fertig ist, sind wir noch nicht fertig, denn wir müssen dafür sorgen, dass die Fehler, die wir im ErrorHandler erzeugen, an der aufrufenden Stelle auch verarbeitet werden:

```
Sub OpenTest()
    Dim wbk As Workbook
    On Error GoTo ErrHandler
    Set wbk = GetWBK(ThisWorkbook.Path, "Testmappe.xls")
    MsgBox "Die Datei " & wbk.Name & _
        " wurde erfolgreich geöffnet."
Exit Sub

ErrHandler:
    MsgBox "Fehler " & Err.Number & " in " & Err.Source & _
        ":" & vbCrLf & vbCrLf & Err.Description
End Sub
```

Auch hier legen wir einen ErrorHandler an und geben dort den Fehler mit all seinen Parametern in einer MessageBox aus. Da generell nur nicht verarbeitbare Fehler *hoch*gemeldet werden (sollten), erübrigt sich die weitere Bearbeitung in der fehlererzeugenden Prozedur und selbige wird einfach beendet. Unser per Err.Raise erzeugter Fehler wird also in der Aufrufhierarchie eine Ebene nach oben weitergeleitet.

Diesen Mechanismus schauen wir uns unter anderem im nächsten Abschnitt etwas näher an.

9.2 Noch etwas Theorie

Aufrufhierarchie

Unser Beispiel bietet uns einen praktischen Einblick in das Thema Fehlerbehandlung. Richtig zur Geltung kommen diese Mechanismen allerdings nur, wenn eine Modularisierung der Applikation vorliegt. Denn wir können recht einfach Prozeduren und Funktionen *geordnet* vorzeitig beenden und die Kontrolle an die nächst höhere Instanz der Aufrufhierarchie zurückgeben.

9.2.1 Fehlerbehandlungshierarchie

Wenn die Sub A die Sub A1 aufruft, handelt es sich bereits um eine Aufrufhierarchie, allerdings die kleinstmögliche. Die folgende Abbildung zeigt eine vierstufige Hierarchie: A ruft A1 auf, A1 ruft A2 auf, A2 wiederum A3 und diese schließlich A4:

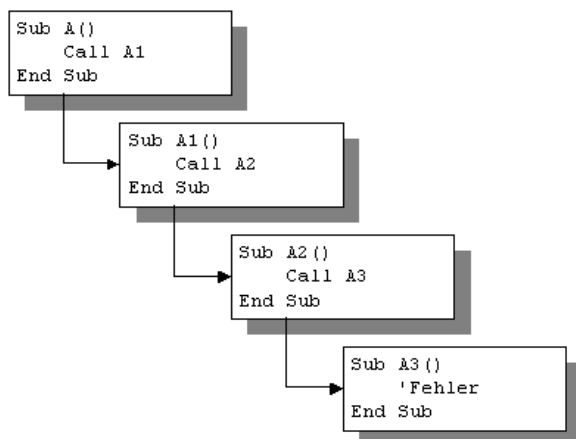


Abbildung 9.6: Vierstufige Aufrufhierarchie

Die Elemente dieser Aufrufhierarchie nebst ihrer vertikalen Ordnung lassen sich übrigens im Haltemodus des Programmes in der im Menü Ansicht befindlichen Aufrufeliste bewundern:

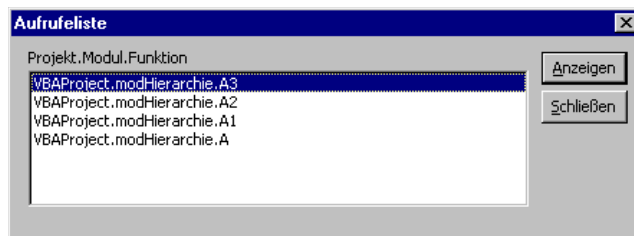


Abbildung 9.7: Aufrufeliste

Wie wir in unserem Beispiel bereits sahen, erhielt die aufrufende Prozedur `OpenTest` auch eine Fehlerbehandlung. Dies ist auch erforderlich, sonst würde das Erzeugen des Fehlers in `GetWBK` dazu führen, dass eben dort ein Laufzeitfehler aufträte. Und das ist es schließlich, was wir vermeiden wollen. Mit anderen Worten: Der in der Funktion `GetWBK` erzeugte Fehler muss in einer der in der Aufrufhierarchie vertretenen Prozeduren oder Funktionen ebenfalls eine Fehlerbehandlungsroutine finden. Oder besser gesagt gefunden haben, denn bereits beim Aufruf einer Prozedur oder Funktion wird eine eventuell vorhandene Fehlerbehandlungsroutine *aktiviert* und in einer internen Liste vermerkt.

Eine aktivierte Fehlerbehandlungsroutine ist also ein `ErrorHandler`, der in einer der an der aktuellen Aufrufhierarchie beteiligten Prozeduren oder Funktionen vereinbart worden ist.

**aktivierte Fehler-
behandlungsroutine**

Unser auftretender oder erzeugter Fehler arbeitet gewissermaßen die Aufrufeliste von oben nach unten ab meldet sich in der ersten aktivierten Fehlerbehandlungsroutine zu Wort, welche dann zur aktiven Fehlerbehandlungsroutine wird.

Das wollen wir nun an unserem Beispiel aus Abb. 1.7 verdeutlichen und zu diesem Behuf in Prozedur `A4` einen Fehler provozieren, indem einem nicht vorhandenen Blatt der Name `Hallo` zugewiesen wird. Da die Arbeitsmappe nur über drei Tabellenblätter verfügt, muss dieser Zugriff fehlschlagen. In `A3` existiert kein `ErrorHandler`. Rüsten wir statt dessen die Prozedur `A1` mit einer solchen aus. Die vier Prozeduren sehen nun wie folgt aus:

```
Public Sub A() (1)
    Call A1 (2)
End Sub (12)

Private Sub A1() (3)
    On Error GoTo ErrHandler (4)
    Call A2 (5)
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ": " & Err.Description (10)
End Sub (11)

Private Sub A2() (6)
    Call A3 (7)
End Sub

Private Sub A3() (8)
    ThisWorkbook.Worksheets(17).Name = "Hallo" (9)
End Sub
```

Die fett dargestellten Zeilen werden in der in den Klammern vermerkten Reihenfolge durchlaufen.

Lassen Sie uns jedoch zuerst einen kleinen Umweg machen und danach wieder kurz auf dieses schematische Beispiel zurückkommen.

9.2.2 Vorgehensweisen im ErrorHandler

Wie wir bereits sahen, kommt das Schlüsselwort `Resume` in der Anweisung `On Error Resume Next` vor. Hierbei bedeutet `Resume Next`, dass im Fehlerfall eben die nächste Anweisung ausgeführt werden soll.

Nun wird das Schlüsselwort `Resume` allerdings auch innerhalb von Fehlerbehandlungsroutinen eingesetzt. In solchen Routinen sind im Prinzip folgende Fälle zu unterscheiden:

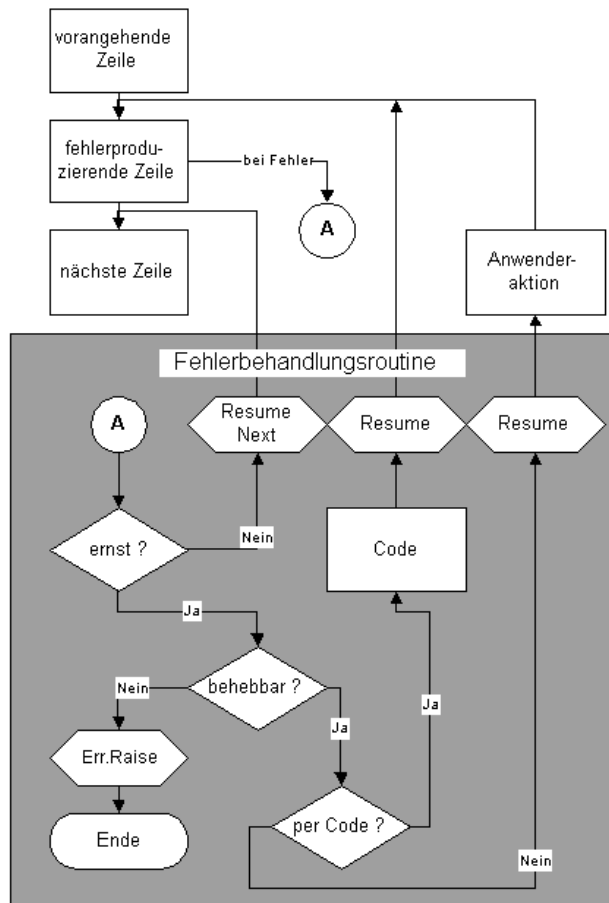


Abbildung 9.8: Flussdiagramm einer Fehlerbehandlung

Fehler nicht ernst

Hierunter fallen sämtliche Situationen, die die weitere Bearbeitung des Programms nicht unmittelbar beeinflussen und eventuell weggelassen oder später nachgeholt werden können. In diesem Fall wird das ursprüngliche Programm an der nächsten Zeile fortgeführt. Dieses Verhalten wird durch die Anweisung `Resume Next` erreicht.

Fehler ernst und per Code behebbar

Sie stoßen zum Beispiel beim Verarbeiten einer Kostenstellenabrechnung auf eine unbekannte Kostenstelle. Dies könnte durchaus ein Grund zum Abbrechen der weiteren Bearbeitung sein. Ein sehr zuvorkommendes Programm allerdings könnte den Anwender fragen, ob die fragliche Kostenstelle angelegt werden soll. Möchte er dies, legt das Programm die Kostenstelle in der angenommenen Kostenstellenreferenz an. Das Programm kann danach an der den Fehler auslösenden Stelle fortgeführt werden, was durch `Resume` eingeleitet wird. Das heißt, die ehemals Fehler produzierende Zeile wird erneut ausgeführt.

Fehler ernst und per Anwenderaktion behebbar

Hierunter fallen etwa nicht bereite Datenträger. Der Anwender kann durch Einlegen eines Datenträgers bzw. Schließen eines Laufwerks den Fehler beseitigen. Auch hier wird die ehemals Fehler Produzierende Zeile erneut ausgeführt: also `Resume`.

Fehler ernst und nicht behebbar

Diese Fehlerkategorie wird beispielsweise durch Dateifehler oder Inkonsistenzen innerhalb von Dateien verursacht. In diesem Fall macht eine Fortführung des Programmes zumeist keinen Sinn und die nächste aktivierte Fehlerbehandlungsroutine wird durch `Err.Raise` hierüber in Kenntnis gesetzt.

9.2.3 Das Err-Objekt

Hinter `Err` verbirgt sich das Fehlerobjekt mit folgenden Eigenschaften und Methoden:

E/M	Name	Beschreibung
M	Clear	Löscht alle Eigenschaften des Err-Objekts
E	Description	Enthält die beschreibbare Fehlerbezeichnung im Klartext
E	HelpContext	Enthält die beschreibbare HelpContext-ID des Fehlers
E	HelpFile	Enthält den vollständigen, beschreibbaren Namen der Hilfedatei

E/M	Name	Beschreibung
E	LastDllError	Gibt einen eventuellen Fehlercode aus einer DLL-Operation zurück
E	Number	Enthält die beschreibbare Fehlernummer
M	Raise	Erzeugt einen näher zu spezifizierenden Fehler
E	Source	Enthält den beschreibbaren Namen des auslösenden Objekts

In der Fehlerbehandlungsroutine der Function `GetWBK`, die wir in Kapitel 9.1 konstruiert haben, erzeugten wir in der letzten Zeile mittels der `Raise`-Methode einen Fehler, der in der aufrufenden Routine weiterverarbeitet werden sollte:

```
Err.Raise intErrNr, "GetWBK", strMessage
```

Die hier verwendete `Raise`-Methode vermag folgende Argumente zu verarbeiten, wobei die optionalen Argumente – wie immer – in Klammern dargestellt sind:

```
Err.Raise number, [source], [description], [helpfile], _
               [helpcontext]
```

Wir sehen also hier fünf Eigenschaften des `Err`-Objekts als Argumente der `Raise`-Methode, von den ersten Dreien werden wir ergiebigen Gebrauch machen.

Number-Argument

Wenn Sie einen vom System erzeugten Fehler weiter leiten, empfiehlt es sich zumeist, diese erzeugte Fehlernummer weiter zu verwenden. Wird Ihnen Fehler „73 – Datei nicht gefunden“ gemeldet, so spricht nichts dagegen, den Fehler an die Fehlerbehandlungsroutine der aufrufenden Prozedur oder Funktion unter derselben Nummer weiter zu melden. Allerdings könnte eine Bestimmung der in diesem Falle fehlenden Datei nicht schaden. Ihre Anwendung arbeitet möglicherweise mit einer Reihe von Dateien, und dann teilt Ihr Programm dem Anwender schlicht und ergreifend mit: Datei nicht gefunden. Damit kann der Arme vermutlich wenig anfangen.

Source-Argument

Die `Source`-Eigenschaft enthält zumeist einen wenig aussagekräftigen Ausdruck wie „VBAProjekt“. Da wir dieses Argument allerdings beschreiben können, spricht nichts gegen einen Ausdruck, der den Modulnamen und den Prozedur- oder Funktionsnamen wiedergibt, etwa

▼ `frmMain.lstFiles_Change`,

▼ `clsUtil.ClearNull` oder

▼ clsWBK:GetWBK

Sofern Sie einen Laufzeitfehler einkalkuliert haben, empfiehlt es sich, einen etwas genaueren Fehlerhinweis zu bieten. „Datei nicht gefunden“ hilft in der Regel nicht weiter. In einer Routine zum Öffnen einer Datenbank wissen wir, in welchem Pfad sie liegt und wie sie heißen soll. Die Beschreibbarkeit der `Description`-Eigenschaft bietet uns die Möglichkeit, eine detaillierte und verarbeitbare Fehlermeldung zu produzieren.

Description-Argument

9.2.4 Aktivieren von Fehlerbehandlungsroutinen

Nach diesem kleinen Umweg, der einige Informationen zum Aufbau von Fehlerbehandlungsroutinen liefert, müssen wir noch einmal zu unserem Beispiel der Abbildung 9.1 zurückkommen.

Dort gingen wir davon aus, dass eine Prozedur A in eine lineare Prozedurkette verzweigt. In den Fällen, in denen Sie über ein einziges Ereignis in eine solche Kette abtauchen, funktioniert das Modell auch. Wenn jedoch Dialoge ins Spiel kommen, gilt dies nicht mehr. Möglicherweise haben wir im Zuge der Initialisierung des Dialogs eine solche Kette abzuarbeiten. Aber irgendwann ist die Ereignisprozedur des Dialoges abgearbeitet und unser Programm läuft über ein End Sub oder Exit Sub der letzten aktivierten Fehlerbehandlungsroutine, was zwangsläufig zum Ableben dieser Routine führt. Der Programmcode ist vollständig abgearbeitet und der Dialog harrt auf dem Bildschirm der weiteren Dinge und steht praktisch ohne Reserven in der Liste der aktivierten Routinen da. Der nächste Fehler resultiert zwangsläufig in einem Laufzeitfehler, den wir ja gerade vermeiden möchten.

Jede Ereignisprozedur – also jeder Einstieg in die Anwendung – muss notwendigerweise mit einer eigenen Fehlerbehandlungsroutine ausgestattet werden. Ruft diese Ereignisprozedur wiederum andere Prozeduren oder Funktionen auf, so können Sie wie gehabt verfahren und mit dieser Routine alle Fehler der Aufrufkette behandeln, quasi zentral abfangen.



Man könnte vermuten, dass man an der Stelle, an der eine UserForm aufgerufen wird, auch die Fehler zentral abfangen kann. Aber dem ist nicht so. Ein an ein Objekt im Projektextplorer gebundenes Modul meldet ohne zusätzliche Maßnahmen keine Fehler weiter, auch wenn eine der in der Aufrufeliste enthaltenen Prozeduren über eine aktivierte Fehlerbehandlungsroutine verfügt!



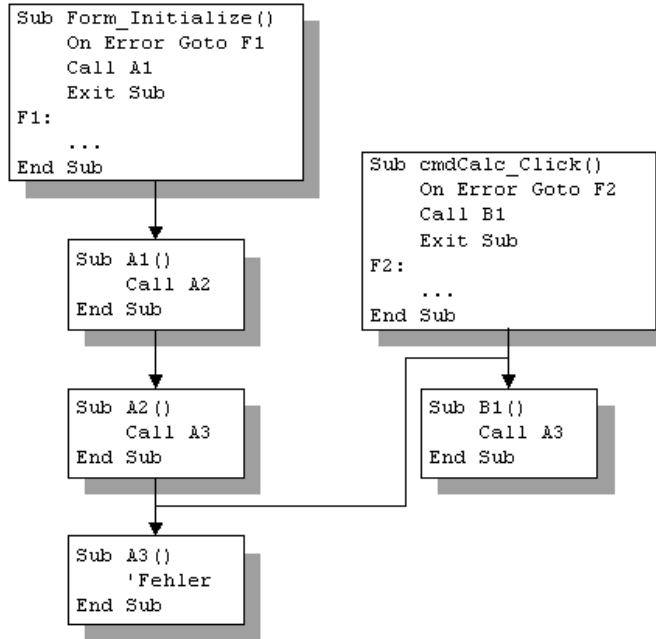


Abbildung 9.9: Aktivieren von Fehlerbehandlungsroutinen

Allerdings ist das nicht tragisch, denn eine Fehlerbehandlung in einer Anwendung, die sich darauf reduziert, an einer möglicherweise sogar künstlich geschaffenen „zentralen“ Stelle eine Routine mit einer MessageBox aufzuweisen, ist fast so gut (oder schlecht) wie keine Fehlerbehandlung.

Zugegebenermaßen ist der bislang geschilderte Imperativ mit Arbeit verbunden, zumal sich herausstellte, dass möglicherweise an vielen Stellen viele Programmzeilen geschrieben oder kopiert werden müssen. Wir werden gleich sehen, dass es so viele Zeilen gar nicht sein müssen, denn eine zentrale Fehlerausgabe kann uns zumindest einige Zeilen einsparen helfen.

9.3 Typen von Fehlerbehandlungsroutinen

Modulkontext des
aufgetretenen Fehlers

Eine Fehlerausgabe muss, wie wir bereits sahen, nicht nur den aufgetretenen Fehler beschreiben. Sie muss uns auch mitteilen, *wo*, also in welchem Modul-kontext, der Fehler aufgetreten ist. Zum Transport dieser Information nutzen wir die `Source`-Eigenschaft des `Err`-Objekts. Die korrekte Zuweisung dieser Eigenschaft ist eine der Aufgaben aller Fehlerbehandlungsroutinen und hier liegt ein wesentlicher Unterschied der vorgestellten Typen von Fehlerbehandlungsroutinen.

Eine weitere Überlegung müssen wir in unsere Betrachtungen noch mit aufnehmen. Würde eine aufgerufene Routine einen dort aufgetretenen Fehler der Fehlerausgabe zuleiten und die Angelegenheit im Übrigen für sich behalten, so erführe die aufrufende Routine hiervon nichts und würde mit der Abarbeitung der weiteren Aufgaben fortfahren. Es leuchtet sofort ein, dass dies nicht sein darf.

Beispiel

Ein Programm enthält unter anderem Routinen, die Umsätze verschiedener Konten verdichten. Hierzu werden in der Listbox `lstAccounts` alle verfügbaren Konten aufgelistet. Ein Klick auf die Schaltfläche `cmdBalanceAccounts` führt dazu, dass alle ausgewählten Konten saldiert werden. Der Pseudocode der Ereignisroutine, die diesen Prozess startet, könnte etwa so aussehen:

```
Private Sub cmdBalanceAccounts()  
    Listbox-Einträge abarbeiten  
    ausgewähltes Konto an BalanceAccount übergeben  
End Sub
```

Hier wird also nur in einer Schleife ermittelt, welche der angezeigten Konten nun verdichtet werden sollen, die eigentliche Verdichtung übernimmt jeweils die Prozedur `BalanceAccount`, die wiederum folgendermaßen aufgebaut sein könnte:

```
Private Sub BalanceAccount(ByRef lngAccount As Long)  
    jeweiligen Kontenumsatz ermitteln und ...  
    ... irgendwo hin schreiben  
End Sub
```

Betrachten wir diese Bearbeitung unter der Hypothese, dass es bei der angestrebten Kontenverdichtung darum geht, sich einen Überblick über den derzeitigen Stand der Dinge zu verschaffen, so ist es nicht von Bedeutung, ob die zu ermittelnden Konten überhaupt bebucht sind. Auch ein eventuell auftretender Fehler würde – je nach Schwere – nicht weiter interessieren.

Frage: Woher soll die Routine `BalanceAccount` denn wissen, ob sie nicht im Rahmen einer Bilanz oder Gewinn- und Verlustrechnung eingesetzt wird, wo natürlich jeder Fehler zum Abbruch des gesamten Prozesses führen muss?

Antwort: Sie kann es nicht wissen und muss die Entscheidung über den Abbruch der Berechnung der einzigen Stelle überlassen, die diese Entscheidung treffen kann. Und das ist die jeweils aufrufende Routine.



Alle aufgerufenen Routinen müssen die Entscheidung über die Reaktion auf einen Fehler bis zu der Stelle weiter leiten, die an oberster Stelle der Aufrufkette steht.

Somit hätten wir neben der Angabe des Kontexts die zweite Aufgabe der Fehlerbehandlungsroutinen gefunden und auch ein Kriterium definiert, worin sie sich grundlegend unterscheiden.

Um diesen beiden Teilaufgaben gerecht zu werden, benötigen wir drei Typen von Fehlerbehandlungsroutinen, die davon abhängen, in welcher Stufe der Aufrufhierarchie die betreffende Routine liegt:

1. Einstiegsroutinen, die von keiner anderen Routine aufgerufen werden,
2. Zwischenroutinen, die aufgerufen werden, aber auch aufrufen,
3. Basisroutinen, die keine anderen Routinen aufrufen.

Hier nun die Gegenüberstellung der Teilaufgaben dieser Typen von Fehlerbehandlungsroutinen:

	Einstiegsroutine	Zwischenroutine	Basisroutine
Kontext	prüfen und evtl. zuweisen		zuwiesen
Fehler	ausgeben	weiter melden	

Einstiegsroutinen

Die Besonderheit liegt darin, dass diese Routinen in der Aufrufhierarchie an oberster Stelle angesiedelt sind. Hierunter fallen alle Ereignisroutinen, beispielsweise die von Steuerelementen, Tabellen oder Arbeitsmappen. Liegen sie an erster Stelle der Aufrufhierarchie, verfügen Sie auch nicht über fremde aktivierte Fehlerbehandlungsroutinen.



Einstiegsroutinen müssen die Fehlerausgabe einleiten und dürfen somit keinerlei `Raise`-Methoden aufweisen.

Im `ErrorHandler` einer Einstiegsroutine laufen Fehler zusammen, die aus der Routine selbst oder – falls vorhanden – einer aufgerufenen stammen können.

Lassen Sie uns noch einen Sonderfall betrachten. Wenn Sie eine Ereignisroutine geschrieben haben, die irgendwelche Aktionen durchführt, so ist es probat, diese Routine von anderer Stelle aus aufzurufen. In diesem Fall wird sie zu einer Zwischenroutine, die einen Fehler per `Raise` weiter melden könnte. Dennoch *muss* sie als Einstiegsroutine behandelt werden, die einen nicht mehr auffangbaren Fehler produzieren würde, sobald sie in ihrer Rolle als Ereignisroutine angestoßen würde.


```

ErrorHandler:
    'Prüfen, ob ein Fehlerkontext in Source enthalten ist:
    If Instr(1, Err.Source, ":") = 0 Then
        Err.Source = "frmX:cmdY_Click"
    Endif
    'Fehlerausgaberoutine aufrufen:
    Call ErrorMessage
End Sub

```

Zwischenroutinen

In der Fehlerbehandlung einer Zwischenroutine können sowohl Fehler aus der Routine selbst auflaufen, als auch welche, deren Ursprung in einer aufgerufenen Routine liegt. Somit muss auch hier geprüft werden, ob die **Source-Eigenschaft** bereits gemäß unseren Konventionen belegt ist.

Zwischenroutinen müssen darüber hinaus den Fehler an die Einstiegsroutine weiter melden:

```

ErrorHandler:
    'Prüfen, ob ein Fehlerkontext in Source enthalten ist:
    If Instr(1, Err.Source, ":") = 0 Then
        Err.Source = "frmX:cmdY_Click"
    Endif
    'Fehler weiter melden:
    Err.Raise Err.Number
End Sub

```

Basisroutinen

Basisroutinen müssen nicht prüfen, ob eine gültige **Source-Eigenschaft** existiert (wo soll die denn herkommen?), sondern ihr nur den eigenen Stempel aufdrücken und den Fehler zurückgeben:

```

ErrorHandler:
    'Fehler mit aktueller Source weiter melden:
    Err.Raise Err.Number, "modABC:MakeSomething"
End Sub

```

Nachdem wir nun die Codekomponenten unserer Routinen hinreichend definiert haben, wenden wir uns der Aufgabe zu, die Fehlerausgabe in wieder verwendbarer Form zu verfassen.

9.4 Zentrale Fehlerklasse clsError

Die zentrale Fehlerklasse clsError besteht aus folgenden Routinen:

Klassenroutinen:

Class_Initialize:	Anmeldennamen ermitteln, Titel und Fehlereinleitung vorbelegen

Methoden:

ClearErrorLog:	Löschen der Logbuchdaten
ErrorMessage:	Ausgabe des aktuellen Fehlers, LogEntry aufrufen
InfoMessage:	Zentrale Ausgabe einer Message
LoadTerms	Laden der Texte bei mehrsprachigen Programmen
LogEntry:	Logbucheintrag erzeugen (Private)

Eigenschaften:

ErrorIntroduction:	einleitender Fehlertext (Schreibzugriff)
HasError:	True, wenn nach letztem ClearErrorLog ein Fehler auftrat
LogSheet:	Logbuch-Tabelle (Schreibzugriff)
Title:	Titel für MessageBoxes (Schreibzugriff)
UserName:	Anwendername (Schreibzugriff)

Folgende Private Variablen finden Verwendung:

Private shtLog As Worksheet	'Logbuch-Tabelle
Private strErrIntro As String	'einleitender Fehlertext
Private strErrIntroSAs String	'einleitender Standard-Fehlertext
Private strTitle As String	'Titel für MessageBoxes
Private strContext As String	'"Kontext:" als Text
Private strNumber As String	'"Nummer:" als Text
Private strText As String	'"Text:" als Text
Private strUserName As String	'Anwendername
Private bHasError As Boolean	'Fehler-Flag

Zur Ermittlung des Anmeldennamens wird folgende API-Funktion verwendet:

```
Private Declare Function GetUserName Lib "advapi32.dll" _
    Alias "GetUserNameA" (ByVal lpBuffer As _
    String, nSize As Long) As Long
```

9.4.1 Klasseninitialisierung

Die Objektvariable cErr wird im Modul modMain angelegt:

```
Public cErr As clsError          'Instanz auf Fehlerklasse
```

Für die Klasseninitialisierung selbst bietet sich das Workbook_Open-Ereignis an:

```
Private Sub Workbook_Open()
    'Fehlerklasse instantiieren und parametrieren
    Set cErr = New clsError
    cErr.LogSheet = shtErrorLog
    cErr.ClearErrorLog
End Sub
```

In der Ereignisroutine Class_Initialize der Klasse clsError wird der Anmeldename für den eventuellen Logbucheintrag ermittelt.

```
Private Sub Class_Initialize()

'=====
    '1999-08-08, Klaus Prinz
    '  Anmeldename ermitteln, Titel und Fehlereinführung
    '  vorlegen

'=====

    Dim nSize As Long          'Länge des zurückgegebenen
                                'Namens aus API
    Dim lngDummy As Variant    'Dummy für API-Aufruf
    On Error GoTo errHandler
    'Anmeldename ermitteln
    strUserName = Space(255)
    nSize = 255
    lngDummy = GetUserName(strUserName, nSize)
    strUserName = Left(strUserName, nSize - 1)
    'Titel und Fehlereinführung vorlegen
    strTitle = "Projekt " & Left(ThisWorkbook.Name, _
                                Len(ThisWorkbook.Name) - 4)
    strErrIntro = "Folgender Fehler ist aufgetreten:"
    strErrIntroS = strErrIntro
    'Fehlertexte vorlegen
    strContext = "Kontext:"
    strNumber = "Nummer:"
    strText = "Text:"
Exit Sub
```

```
errHandler:
    Err.Source = "clsError:Class_Initialize"
    ErrorMessage
End Sub
```

Ein Wort noch zu den Variablen `strErrIntro`, `strContext`, `strNumber` und `strText`. In `Class_Initialize` werden sie mit den Standardtexten vorgelegt. In einer fremdsprachlichen Anwendung beispielsweise können Sie die Zuweisung dieser Texte in der `Class_Initialize` durch Aufrufe an eine Übersetzungsklasse abändern (siehe auch Kapitel 11 – Applikationsdesign). Dies darf jedoch nicht in der `Class_Initialize` erfolgen, da die Fehlerklasse immer als erste instantiiert werden sollte und die `Terms`-Klasse dann noch nicht geladen sein kann! Aus diesem Grund muss eine eigene Methode dafür herangezogen werden, die `LoadTerms`-Prozedur:

```
Public Sub LoadTerms()

'=====
' 1999-08-08, Klaus Prinz
'   Laden der Texte bei mehrsprachigen Programmen
'=====

On Error GoTo errHandler
strTitle = cTerms.GetTerm("Project") & " " & _
           Left(ThisWorkbook.Name, _
           Len(ThisWorkbook.Name) - 4)
strErrIntro = cTerms.GetTerm("ErrorIntroduction")
strErrIntroS = strErrIntro
strContext = cTerms.GetTerm("Context") & ":"
strNumber = cTerms.GetTerm("Number") & ":"
strText = cTerms.GetTerm("Text") & ":"
Exit Sub

errHandler:
    Err.Source = "clsError:LoadTerms"
    ErrorMessage
End Sub
```

In diesem Falle benötigen Sie natürlich noch etwas Code, der sich um die Rückgabe dieser Begriffe kümmert. Mit der `Terms`-Klasse `clsTerms` alleine ist es nicht getan. Doch dieses Thema ist Gegenstand des Kapitels 11 – Applikationsdesign. Der Aufbau der `Workbook_Open`-Ereignisroutine würde dann natürlich auch anders ausfallen.

Erfolgt nun während des Programmlaufs eine Änderung der eingestellten Sprache, so können Sie einfach die Methode `LoadTerms` aufrufen, um die Fehlerklasse zu veranlassen sich die aktuellen Begriffe zu besorgen:

```
cErr.LoadTerms
```

Doch nun zurück zu unserem Thema.

Der den Fehler einleitende Text ist zusätzlich als Eigenschaft herausgeführt, um ihn bei Bedarf ändern zu können. Wenn zum Beispiel die Parameter einer Datenbank-Verbindung nach Änderung getestet wird, so wird der einleitende Standardtext der **nächsten (!)** Fehlerausgabe verändert.

```
cErr.ErrorIntroduction = "Der Verbindungsaufbau schlug" & _  
                        " aus folgendem Grund fehl:"  
cErr.ErrorMessage
```

Danach wird wieder der in `Class_Initialize` erzeugte Einleitungstext zugewiesen. Dieses Kunststück wird durch die Verwendung zweier Variablen für diesen Text ermöglicht. Die Eigenschaftsroutine `ErrorIntroduction` weist den übergebenen Text der Variablen `strErrIntro` zu, die in der `MessageBox` der Routine `ErrorMessage` ausgegeben wird. Danach aber wird dieser Variablen wieder der in `strErrIntroS` hinterlegte Standardtext zugewiesen.

9.4.2 Fehlerausgabe

Die eigentliche Fehlerausgabe ist reichlich unspektakulär:

```
Public Sub ErrorMessage()  
  
    '=====   
    '1999-08-08, Klaus Prinz  
    '   Ausgabe des aktuellen Fehlers, Standard-Text wieder  
    '   zuweisen, LogEntry aufrufen  
  
    '=====   
    MsgBox strErrIntro & vbCrLf & vbCrLf & _  
           strContext & vbTab & Err.Source & vbCrLf & _  
           strNumber & vbTab & Err.Number & vbCrLf & _  
           strText & vbTab & Err.Description, _  
           vbExclamation, strTitle  
    'Standard-Fehlereinleitung wieder zuweisen  
    strErrIntro = strErrIntroS  
    'Logbucheintrag vornehmen  
    Call LogEntry
```

```
'Fehler-Flag auf True setzen
bHasError = True
End Sub
```

Durch die Konstante vbTab wird erreicht, dass die Anfänge der einzelnen Informationen untereinander stehen:



Abbildung 9.10: Beispiel einer Fehlerausgabe

Allerdings sollten Sie für den Fehlertext keinen zu langen Begriff wählen, weil sonst der Tabulator der Kontext- und Nummernzeile nicht mehr reicht. Aus diesem Grund hab ich »Text« gegenüber dem zweifellos treffenderen »Beschreibung« den Vorzug gegeben.

Diese Routine hat selbst keinen ErrorHandler, da ein `On Error ...` innerhalb dieser Routine zur Folge hätte, dass der aktuelle Fehler gelöscht würde. Somit müssten wir uns die Werte in Variablen merken und diese dann ausgeben. Und genau das schenke ich mir.

9.4.3 Fehlerlogbuch

Unter Angabe eines Zeitstempels und des Anwendernamens werden Kontext, Nummer und Beschreibung des Fehlers in der Logbuchtafel abgelegt, was dann so aussehen kann:

	A	B	C	D	E
1	TimeStamp	User	Context	Number	Description
2	2000-01-22 15:18:06	Schorsch	shtMain.cmdProduceError_Click	6	Überlauf
3	2000-01-22 15:18:11	Schorsch	shtMain.cmdProduceError_Click	6	Überlauf
4					
5					
6					
7					
8					
9					
10					

Abbildung 9.11: Fehlerlogbuch

Der Eintrag in das Fehlerlogbuch wird von der Routine LogEntry vorgenommen, die nun vorgestellt wird:

```
Private Sub LogEntry()

'=====
'1999-08-08, Klaus Prinz
'    Logbucheintrag erzeugen
'=====

Dim iRow As Long           'Zeilenzeiger in
Logbuch
Dim strContext As String   'Fehlerkontext
Dim lngNumber As String    'Fehlernummer
Dim strText As String      'Fehlerbeschreibung
'Fehlerwerte zuweisen
strContext = Err.Source
lngNumber = Err.Number
strText = Err.Description
On Error GoTo ErrHandler
'Sub verlassen, wenn Worksheet-Variable Nothing ist
If shtLog Is Nothing Then Exit Sub
'Zeilenzeiger in nächste Zeile positionieren
iRow = shtLog.Range("A1").CurrentRegion.Rows.Count + 1
'Informationen eintragen
shtLog.Cells(iRow, 1).Value = Now()
shtLog.Cells(iRow, 2).Value = strUserName
shtLog.Cells(iRow, 3).Value = strContext
shtLog.Cells(iRow, 4).Value = lngNumber
shtLog.Cells(iRow, 5).Value = strText
Exit Sub

errHandler:
Err.Source = "clsError:LogEntry"
ErrorMessage
End Sub
```

Wenn das Worksheet-Objekt shtLog keinen gültigen Verweis enthält, wird die Prozedur logischerweise verlassen. Die Zuweisung dieses Worksheet-Objekts erfolgt in der Eigenschaft LogSheet:

```
Public Property Let LogSheet(ByRef shtLogSheet As _
                               Worksheet)
```

```
Set shtLog = shtLogSheet
End Property
```

Man ist versucht einfach die Existenz einer Tabelle namens `shtLog` anzunehmen. Wenn ich aber keine Protokollierung wünsche, muss ich entweder die Tabelle in der Arbeitsmappe lassen oder den Klassencode anpassen. Ich habe mich dazu entschlossen, die Klasse auf beide Fälle vorzubereiten, und muss meinen geplagten Kopf nicht damit belasten, bei jedem Projekt nun zu überlegen, ob ich die Klasse anpassen muss. Deshalb eine eigene Variable und eine Eigenschaftsprozedur.

Üblicherweise wird beim Programmstart die Logbuchtafel gelöscht:

```
cErr.ClearErrorLog
```

Die Routine selbst löscht alle Daten ab Zeile 2, wozu die `CurrentRegion` durch die `Offset`-Methode um eine Zeile nach unten versetzt wird:

```
Public Sub ClearErrorLog()

'=====
' 1999-08-08, Klaus Prinz
'   Löschen der Logbuchdaten
'=====

If shtLog Is Nothing Then Exit Sub
On Error Goto Errhandler
shtLog.Range("A1").CurrentRegion.Offset(1, _
                                           0).ClearContents

bHasError = False
Exit Sub

errHandler:
Err.Source = "clsError:ClearErrorLog"
ErrorMessage
End Sub
```

9.4.4 Infoausgabe

Die Ausgabe einer `MessageBox` mit einer Information für den Anwender ist nicht notwendigerweise eine Kernaufgabe für eine Fehlerklasse. Sollte Ihnen die Methode in dieser Klasse nicht passen, so können Sie diese ja an anderer Stelle unterbringen.

```
Public Sub InfoMessage(ByVal strMessage As String, _
    Optional ByVal strMessage2 As Variant)
```



```
'=====
'1999-08-08, Klaus Prinz
'   Zentrale Ausgabe einer Message
```

```
'=====
On Error GoTo errHandler
If Not IsMissing(strMessage2) Then
    strMessage = strMessage & vbCrLf & vbCrLf & _
                strMessage2
End If
MsgBox strMessage, vbInformation, strTitle
Exit Sub
```

```
errHandler:
    Err.Source = "clsError:InfoMessage"
    ErrorMessage
End Sub
```

`strMessage2` ist als Variant angelegt, weil nur dieser Datentyp den Wert Missing beinhalten kann.

Hier nun die erste Parametrierung der Methode:

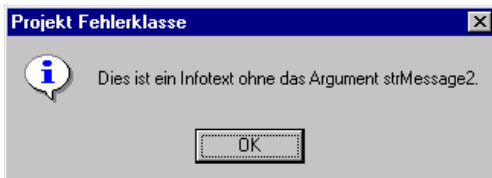


Abbildung 9.12: InfoMessage ohne `strMessage2`

Und die zweite:

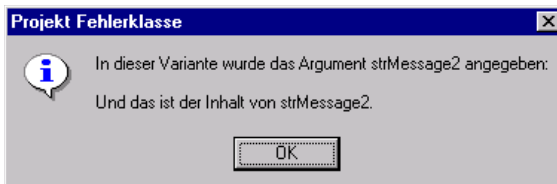


Abbildung 9.13: InfoMessage mit `strMessage2`

9.4.5 Fazit

Fehlerbehandlung macht Arbeit, ist aber wie fast alles im Leben reine Gewöhnungssache. Fehlerbehandlung macht aber mit der Zeit immer weniger Arbeit, denn zum einen lernt man die Feinheiten der Fehlerbehandlung von Anwendung zu Anwendung besser zu verstehen und zu gestalten. Zum anderen aber bringt allein das Vorlesen einer aussagekräftigen MessageBox durch den geplagten Anwender eine wesentliche Erleichterung bei der Lokalisierung des (Programmier-)Fehlers, denn darum handelt es sich ja letztendlich ...



Übrigens: Vergessen Sie bitte nicht, vor Auslieferung Ihrer Anwendung im Register Allgemein der Optionen »Bei nicht verarbeiteten Fehlern unterbrechen« zu aktivieren.

9.5 Fehlervermeidung

Wenn unser Finanzminister für jeden Fehler der Nummern 9, 13, 91 oder 1004 eine Mark erhielte, wären unsere Staatsschulden vermutlich beglichen. Genau diese Fehler sind Kandidaten, die bei gebotener Umsicht in der Codierung vermeidbar sind.

Ein `If IsNumeric()` oder `If IsDate()` vor weiterer Berechnung wirkt bei dem Fehler 13 Wunder. Dieser Fehler darf eigentlich nur während der Entwicklung auftreten.

Die 1004, die oft mit der `Find`-Methode des `Range`-Objekts einhergeht, lässt sich durch ein

```
If Not RangeObjekt.Find(Parameter) Is Nothing Then
    iRow = RangeObjekt.Find(Parameter).Row
Else
    'Alternativen berücksichtigen
Endif
```

recht einfach vermeiden. Denn wenn der in dem Argument `what` angegebene Begriff nicht gefunden wird, ist die Rückgabe der Methode eben `Nothing`, und `Nothing` hat nun mal keine `Row`-Eigenschaft. Das heißt:



Den regulären Möglichkeiten zum Vermeiden eines Fehlers gilt das Hauptaugenmerk.

Wer eine `Find`-Methode verwendet, muss entweder sicher sein, dass sie einen `Range` zurückgibt, oder aber sie mit einer entsprechenden Prüfung auf `Nothing` absichern. Alles andere ist Leichtsinn.

Doch nicht nur aus Find-Methoden zurückzugebende Range-Objekte sind die alleinigen Problemfälle, auch Zugriffe auf Tabellenblätter bergen ein Risiko, denn wir haben praktisch keine Chance, das Umbenennen, Verschieben oder gar Löschen von Tabellen in externen Dateien zu verhindern. Auch hier gehört eine entsprechende Prüfung vor dem Zugriff auf die betreffende Tabelle zum guten Ton.

9.6 Gängige Fehler und ihre Behandlung

Hier eine Liste der gängigsten Fehler nebst Hinweisen zu deren Behandlung. Fehler wie etwa `35 - Sub oder Function nicht definiert` sind allerdings nicht in die Betrachtung mit aufgenommen worden, da sie genügend Hinweise zur Behebung des Fehlers beinhalten.

5 - Unzulässiger Prozeduraufruf oder ungültiges Argument

Zumeist liegt die Übergabe eines irgendeine Grenze verletzenden Arguments vor. Die Hilfe zeigt dies am Beispiel der Sin-Funktion recht anschaulich.

Weitaus interessanter ist jedoch folgender Fall, der unter NT ebenfalls den Fehler 5 auslöst.

```
strFile = "C:\pagefile.sys"
If (GetAttr(strFile) And vbDirectory) = vbDirectory Then
...
```

Hinweise zu Ursache und Vorgehensweise in diesem Fall finden Sie in Kapitel »7 – Sprachelemente, die Zweite« unter der GetAttr-Funktion.

6 - Überlauf

Sie haben die Grenzen des Datentyps verletzt. Überprüfen Sie den Wert und ändern Sie ggf. den Datentyp zum Beispiel von Integer zu Long.

7 - Nicht genügend Speicher

Dieser Fehler taucht in zwei Erscheinungsformen hin und wieder auf. Die Ursache der ersten, der harmlosen, Variante ist, dass Sie den Variablenspeicher überfordert haben, indem Sie zum Beispiel ein Array mit zu vielen Elementen erzeugt haben:

```
Dim strTest(100, 100, 100)
```

Diese Zeile erzeugt ein Array mit 1.030.301 Elementen. Sollten Sie es wirklich mit so vielen Daten zu tun haben, dann legen Sie diese in Tabellen ab. Soweit die gute Nachricht.

Liegt eine solche Ursache nicht vor, so ist man praktisch hilflos, oder sagen wir besser fast hilflos. Scheinbar ist der Codeumfang nicht bestimmend, wohl aber Anzahl und Typ der im Speicher gehaltenen Variablen. Außerdem scheint die Größe der aktivierten Module eine Rolle zu spielen. Sie sehen, Genaues weiß ich auch nicht, obwohl mir der Fehler schon einige Male begegnete und ich ursprünglich eigentlich vermutete, dass er ab Windows NT der Vergangenheit angehören müsste.

Keinen Einfluss haben gemäß Microsoft:

- ▼ Länge der Namen von Bezeichnern
- ▼ Kommentare
- ▼ Leerzeilen

Einfluss dagegen haben gemäß derselben Quelle:

- ▼ Anzahl der geladenen Objekte (siehe Dim as New ...)
- ▼ Typ der Variablen (no Variants!, wenn's so einfach wäre)

Wenn nichts mehr hilft und gleichzeitig Klassenmodule im Projekt vorhanden sind, bringt eine Kompilierung in einer eigenen DLL Erleichterung. Hierdurch reduziert sich zwangsläufig der Speicherbedarf.

Daneben findet man in der Literatur auch den Hinweis, eine neue Datei anzulegen und die Tabelleninhalte (nicht die Tabellen selbst!!) portionsweise in diese neue Datei zu übertragen und gleichermaßen mit dem Code zu verfahren. Das ist jedoch Flickwerk.

Im Frühjahr 1999 hatte ich einen solchen Fall bei einem Kunden, für den unser Haus eine Reihe von umfangreichen statistischen Auswertungen in Excel VBA entwickelte. Diese Programme wuchsen mit den Anforderungen über fast zwei Jahre und produzierten gegen Ende fast nur noch Fehler wie »Nicht genügend Speicher« oder »Fehler beim Lesen von/Schreiben auf Gerät« (Fehler 57). Jede der drei betroffenen Arbeitsmappen lag zwischen 1000 und 2000 Zeilen Code. Ich entschied mich für eine Neuentwicklung in Visual Basic unter weit gehender Verwendung der Algorithmen aus VBA, aber mit neuer Oberfläche. Etwa eine Woche dauerte die Implementierung der ehemaligen Funktionen unter VB, noch drei Tage für eine komfortable Oberfläche, und das Programm war fertig. Die Ausgabe der Statistiken erfolgte nach wie vor in Excel-Dateien, die allerdings nur noch aus zwei Tabellen bestanden und keinen Code mehr aufwiesen. Das Programm war anschließend um den Faktor 5 schneller und produzierte keinerlei Fehler mehr.

9 - Index außerhalb des gültigen Bereichs

Sie haben wahrscheinlich versucht, per Index (`Worksheets(i)`) oder aber per Name (`Worksheets(„Hallo“)`) auf ein Element einer Auflistung zuzugreifen, das nicht existiert. Wenn Sie nicht sicher sein können, dass das Element existiert, versuchen Sie einen Testzugriff:

```
For iSheet = 1 to ThisWorkbook.Worksheets.Count
    If ThisWorkbook.Worksheets(i).Name = strSheet Then
        Exit For
    Endif
Next
If iSheet > ThisWorkbook.Worksheets.Count Then
    'Tabelle nicht vorhanden
End If
```

oder

```
On Error GoTo ErrorHandler
ThisWorkbook.Worksheets(strSheet).Name = _
    ThisWorkbook.Worksheets(strSheet).Name
'weiterer Code
Exit Sub
```

Hierfür gibt es keine Schönheitspunkte, aber es funktioniert.

11 - Division durch Null

Wie die Beschreibung schon treffend formuliert, haben Sie eine Division versucht, deren Divisor den Wert 0 aufweist. Testen Sie den Wert vor Ausführung der Division:

```
If dblDivisor = 0 Then
    'evt. Maßnahmen
Else
    dblErgebnis = dblDividend / dblDivisor
End If
```

13 - Typen unverträglich

Vermutlich liegt eine numerische Operation, eine logische Operation oder eine Datumsberechnung mit einem Wert vor, der nicht numerisch, logisch oder kein Datumswert ist. Prüfen Sie bitte mit `IsNumeric()` oder `IsDate()`, ob es sich um eine numerische oder um eine Datumsgröße handelt.



Doch Vorsicht: die `IsNumeric()`-Funktion liefert auch ein `True`, wenn es sich um einen gültigen Datumswert handelt.

Bei logischen Werten ist es einfacher, da diese von außen im Gegensatz zu den anderen genannten Typen praktisch nicht in Programme einfließen können. Numerische Werte oder Datumswerte gelangen ohne weiteres als Zellinhalte oder Inhalte von Dialogelementen ins Innere von Programmen, wohingegen logische Werte entweder abgeleitete Größen sind oder über CheckBoxes oder OptionButtons Einzug halten. Und diese Wege dürften keine Schwierigkeiten darstellen.

28 - Nicht genügend Stapelspeicher

Verweise auf offene, das heißt noch nicht beendete Prozeduren werden mitsamt den Instanzwerten ihrer Variablen in einem Stapelspeicher abgelegt, der dann LIFO (Last In First Out) abgearbeitet wird. Und genau diesen Speicher haben Sie gesprengt, wenn Ihnen dieser Fehler begegnet. Sie können sich jedoch darauf verlassen, dass Ihr Programm durch eine fehlerhafte Prozedur das Malheur verursacht. Überprüfen Sie die Prozedur, die den Fehler hervorrief, und Sie werden mit an Sicherheit grenzender Wahrscheinlichkeit feststellen, dass die aktuelle Prozedur sich selbst immer wieder aufruft. Überprüfen Sie die Abbruchbedingung dieser Prozedur.

48 - Fehler beim Laden einer DLL

Dies weist auf entweder auf eine nicht registrierte, eine versionsinkompatible oder eine nicht vorhandene DLL hin. Stellen Sie sicher, dass sie existiert, und starten Sie im Ausführen-Dialog (Start-Button) REGSVR32 DLL-Name, wenn selbige im Windows-, im System- bzw. System32-Verzeichnis liegt. Andernfalls müssen Sie bei DLLName den vollständigen Pfad nebst Laufwerk angeben.

Fehler beim Lesen von/Schreiben auf Gerät

Dieser Fehler kann durchaus auf Hardwarefehler oder verbogene Treiber zurückzuführen sein, muss aber nicht. In Grenzfällen kann ein reines Speicherproblem die Ursache sein. Weitere Informationen hierzu finden Sie bei der Beschreibung des Fehlers 7, »Nicht genügend Speicher«.

68 - Gerät nicht verfügbar

Hierunter kann sich alles verbergen, was sich außerhalb des Motherboards befindet, in der Regel jedoch ein Laufwerk. Den Laufwerksfehler im Vorfeld abzu prüfen ist hier das beste Gegenmittel (`Dir()`-Funktion).

Im Zusammenhang mit (oder besser nach) dem Fehler „Nicht genügend Speicher“ sollte man die Meldung nicht ernst nehmen und zuerst das Speicherproblem lösen.

70 - Zugriff verweigert

Dieser Fehler erscheint hoffentlich nur, wenn Sie ihn durch folgenden Code hervorgelockt haben:

```
hFile = FreeFile
Open strPath & strFile For Binary Lock Write As #hFile
Close hFile
```

Ist die Datei für die weitere Fortführung des Programms erforderlich, so sollten Sie dem Anwender signalisieren, dass die Datei geschlossen werden muss. In der Regel enthält die `Description`-Eigenschaft des Fehlerobjekts auch einen Hinweis darauf, wer die Datei gerade exklusiv geöffnet hält.

Innerhalb der Fehlerbehandlung können Sie in einer Schleife mit der Rückgabe einer `MessageBox` den Zugriff immer wieder nach folgendem Muster versuchen:

```
Select Case Err.Number
Case 70
Do
    strMessage = Err.Description & _
                "Wollen Sie es erneut versuchen?"
    intReturn = MsgBox(strMessage, vbOKCancel)
    If intReturn = vbOK Then
        Resume
    Else
        Exit Do
    End If
Loop
```

71 - Datenträger nicht bereit

Dies bedeutet, der Datenträger ist zwar vorhanden, enthält aber kein Speichermedium (Diskette, CD, Wechsellplatte etc.). Auch hier sollten Sie dem Anwender Gelegenheit geben, die Ursache zu beheben und den Zugriff dann erneut zu versuchen. Der Code im `ErrorHandler` ist dann ähnlich dem Fragment des vorherigen Fehlers 70.

73 - Datei nicht gefunden

Wem diese Fehlermeldung begegnet, der weiß in der Regel sofort, was die Ursache ist und vor allem wer verantwortlich ist. Dateien festen Namens,

die das Programm ständig benötigt, verschwinden nicht so einfach. Handelt es sich jedoch um Dateien nicht determinierbaren Namens, so sollte der Zugriff generell über ein Formular mit einer ListBox erfolgen, in der die potenziellen Kandidaten zur Auswahl stehen:

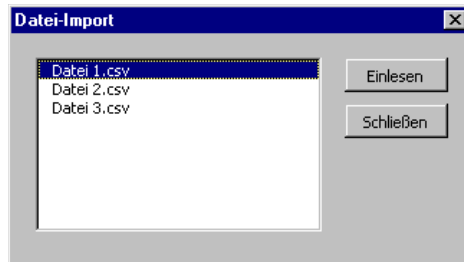


Abbildung 9.14: Dialog zum Öffnen nicht bekannter Dateien

Die in dieser ListBox aufgeführten Dateien haben praktisch keine Chance zu verschwinden, denn zum Zeitpunkt des Füllens der ListBox müssen sie schließlich vorhanden gewesen sein.

Ein entsprechender Fehler kann allerdings beim ersten Aufruf Ihres Programms beim Anwender deshalb in Erscheinung treten, weil eben die Pfade und Dateinamen noch nicht eingerichtet sind und immer noch Ihre alten Einstellungen von Ihrem Entwicklungsrechner enthalten. Wenn Sie den Rat beherzigt haben, in einer Tabelle namens Steuerung zentral solche Einstellungen zu hinterlegen, so aktivieren Sie die Tabelle Steuerung, setzen den Cursor in die erste relevante Zelle und fordern den Anwender in einer MessageBox auf, diese Bezüge anzupassen:

	A	B	C
	Einstellungen in Registry ablegen		
1	Titel	Pfad bzw. Name	interner Name
2	Pfad der Importdateien:	D:\Projekte aktuell\Excel\EnBW\KK-Tool\Import	ImportPath
3	Pfad der Exportdateien:	D:\Projekte aktuell\Excel\EnBW\KK-Tool\Export	ExportPath
4	Pfad der Versorgerübersicht:	D:\Projekte aktuell\Excel\EnBW\KK-Tool	EVUPath
5	Name der Versorgerübersicht:	Mitglieder_VDEW.xls	EVUFile
6	Namenskomponente der Exportdateien:	IMP	ExportNameKomp

Abbildung 9.15: Beispiel von Pfad- und Dateiangaben in der Tabelle Steuerung

Weitere Informationen hierzu finden Sie im Kapitel Applikationsdesign.

76 - Pfad nicht gefunden

Hier gelten die voranstehenden Erläuterungen zum Fehler 73 – Datei nicht gefunden.

91 - Objektvariable oder With-Blockvariable nicht festgelegt

Das ist *der* Excel-VBA-Fehler schlechthin. Er hat zwei potenzielle Ursachen:

- ▼ Sie haben in der Tat vergessen, eine Variable vor ihrer ersten Verwendung zuzuweisen.
- ▼ Eine dynamische Zuweisung war nicht von Erfolg gekrönt.
- ▼ Ein anderes Objekt hat dem Container den Fokus geraubt.

Für Fall 1 ist zumeist eine Änderung des Codes angesagt. Ein Sonderfall kann aber auch dadurch vorliegen, dass sich zwei Codekomponenten blockieren, indem beide in der `Class_Initialize`-Routine die jeweils andere referenzieren, wie folgendes Codefragment beispielhaft zeigt:

```
'Klasse cA (clsA)
Private Sub Class_Initialize()
    strB = cB.MethodeX()
End sub

'Klasse cB (clsB)
Private Sub Class_Initialize()
    strA = cA.MethodeY()
End sub
```

Hier hilft nur Nachdenken und Umbauen einer der beiden Klassen.

Unser Fall 2, also eine fehlerhafte dynamische Zuweisung, kommt bei Methoden der Excel-Objekte vor, die selbst wieder ein Objekt des (zumeist) gleichen Typs zurückgeben. So verfügt etwa unser geliebtes `Range`-Objekt über eine ganze Reihe dieser Methoden. Denken Sie zum Beispiel an `Find`, `Offset`, `Resize` oder `SpecialCells`.

Diese Fehler sind im Prinzip alle vermeidbar. Auf die Rückgabe der `Find`-Methode darf nicht zugegriffen werden, ohne vorher geprüft zu haben, ob die Rückgabe nicht doch `Nothing` ist:

```
If rngX.Find(...) Is Nothing Then
    'gesuchter Begriff wurde nicht gefunden
Else
    'Rückgabe ist ein gültiger Range
    iRowX = rngX.Row
End If
```

oder

```

Set rngFound = rngX.Find(...)
If rngFound Is Nothing Then
    'gesuchter Begriff wurde nicht gefunden
Else
    iRowX = rngFound.Row
End If

```

Bei den anderen Methoden müssen Sie eben die entsprechenden Argumente überprüfen, bevor sie auf die Methode losgelassen werden.

Fall 3 hat schon etwas Hinterhältiges an sich. Wenn Sie beispielsweise einen `CommandButton` aus der `Steuerelement-Toolbox` auf eine Tabelle platzieren und dabei vergessen, die `TakeFocusOnClick`-Eigenschaft auf `False` zu setzen, so funktionieren selbst einfachste, im Übrigen syntaktisch völlig korrekte Zugriffe auf das `Worksheet`- oder das `Range`-Objekt nicht mehr.

94 - Unzulässige Verwendung von Null

Hiermit ist der Wert `Null` (nicht die Zahl `null`) gemeint. Er tritt üblicherweise dann auf, wenn Sie den Inhalt eines mit `Null` initialisierten Datenbankfeldes an einen Datentyp übergeben wollen, der diesen Wert `Null` nicht trägt. Und das sind schlicht und ergreifend alle Datentypen mit Ausnahme des `Variants`.

Die folgende Methode `ClearNull` der Klasse `clsUtil` nimmt ein `ADODB-Datenbankfeld` entgegen und erzeugt eine dem Feldtyp angemessene Rückgabe. So wird bei logischen Feldern `False`, bei numerischen `0` und bei Zeichen- sowie Datumsfeldern ein `NullString ""` zurückgegeben:

```

Public Function ClearNull(ByVal fldAct As ADODB.Field) As _
    Variant

    '=====
    '1999-07-25, Klaus Prinz
    '    Rückgabe eines definierten Wertes, auch wenn Feld
    '    Null ist
    '-----
    'Argumente:
    '    fldAct:    das zu bearbeitende Datenbankfeld
    '=====

    On Error GoTo errHandler
    If IsNull(fldAct.Value) Then
        Select Case fldAct.Type

```

```

Case adBoolean
    ClearNull = False
Case adSmallInt, adInteger, adSingle, adDouble, _
    adCurrency
    ClearNull = 0
Case adDate, adChar, adVarChar
    ClearNull = ""
End Select
Else
    If fldAct.Type = adChar Or _
        fldAct.Type = adVarChar Then
        ClearNull = SQLString(fldAct.Value)
    Else
        ClearNull = fldAct.Value
    End If
End If
Exit Function

```

```

errHandler:
    Err.Raise Err.Number, "clsUtil:ClearNull"
End Function

```

Verwendet wird diese Methode beispielsweise so:

```

Public Property Get Name() As String
    Name = cUtil.ClearNull(rsDaten.Fields("Name"))
End Property

```

1004 - Anwendungs- oder objektdefinierter Fehler

Immer dann, wenn VB (oder VBA) den genauen Grund für den Fehler bei einem verwendeten Objekt nicht kennt, erscheint diese hilfreiche Botschaft auf dem Bildschirm. Das Durchlesen des dazugehörigen Hilfetextes können Sie sich schenken, denn dort werden Sie im Prinzip dazu aufgefordert, sich gründlicher mit dem verwendeten Objekt auseinander zu setzen.

Im Prinzip können Sie darauf vertrauen, dass Sie das Objekt mit dem übergebenen Argument schlichtweg überfordert haben. So produziert beispielsweise die folgende Zeile diesen ominösen 1004:

```
rngX.Cells(1, 300).Value = "Hallo"
```

Da die Tabelle nur über 256 Spalten verfügt, können wir die Spalte 300 nicht referenzieren. Im folgenden Fall schlägt der Aufruf fehl, weil dieser Bereich eben kein SubTotal hat:

```
rngX.CurrentRegion.RemoveSubtotal
```

Die folgende Methode schlägt dann fehl, wenn beim Verschieben der rechts des Ranges befindlichen Zellen eine dieser Zellen über die horizontale Blattgrenze hinausgelangen würde:

```
rngX.Insert xlShiftToRight
```

In so einem Fall wäre es schon gut zu wissen, dass eben diese Grenze verletzt wird. Wir würden möglicherweise auf die Fehlerausgabe verzichten und statt dessen die Ursache dafür per Programm beseitigen. Aber pauschal beim Fehler 1004 so vorzugehen finde ich nicht überzeugend. Der Fehler, den diese Insert-Methode hervorruft, würde selbst den abgebrühtesten Anwender schlichtweg überfordern:

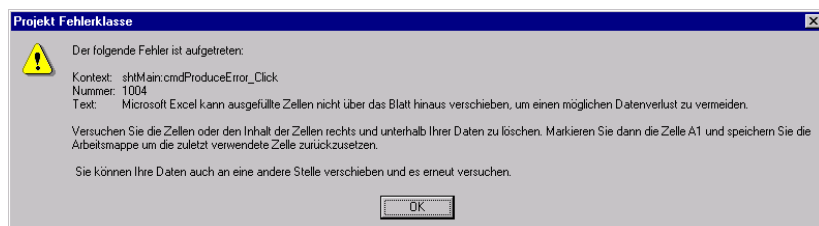


Abbildung 9.16: Err.Description des obigen Insert-Fehlers

Bleibt zu hoffen, dass die Entwickler dieser Objektbibliotheken sich zukünftig etwas mehr Mühe beim Erzeugen von Fehlern geben, denn solche fehlerhaften Parameter werden selbst dem besten Kenner einer solchen Bibliothek hin und wieder unterlaufen.



Klassen- programmierung

10

Kapitelüberblick

10.1	Word.Application	486
10.2	Klassen	487
10.2.1	Klassenmodul erstellen	488
10.2.2	Elemente einer Klasse	490
10.3	Eine Datenklasse für den Login-Dialog	500
10.3.1	Die Tabelle User	501
10.3.2	Funktionale Überlegungen	501
10.3.3	Implementierung	502
10.3.4	Verwendung der Klasse	505
10.4	Fehlerbehandlung in Klassen	506
10.4.1	Ein Fehler im Initialize-Ereignis einer Klasse	507
10.4.2	Fehler im Terminate-Ereignis einer Klasse	509
10.4.3	Die Konstante vbObjectError	510
10.5	Das Rechnungsbeispiel	513
10.5.1	Die Aufgabenstellung	513
10.5.2	Die Tabellen	513
10.5.3	Klassenarchitektur	516
10.5.4	Klassencodierung	517
10.5.5	Codierung des Moduls shtInvoice	525
10.5.6	Zusammenfassung	528
10.6	DLLs – wann und wozu	529
10.6.1	Zur Abschreckung ein Beispiel	529

Ich möchte mit einem Zitat, das aus Dan Appleman's Buch *COM/ActiveX-Komponenten mit Visual Basic 6 entwickeln* entnommen ist, beginnen:

Mythos: ActiveX und COM sind nur etwas für fortgeschrittene Visual-Basic-Programmierer.

Tatsache: ActiveX und COM sollten zu den ersten Dingen gehören, die ein Visual-Basic-Einsteiger lernt.

Nachdem er dann ein wenig Dampf abgelassen hat, schreibt er weiter:

»Objektorientiertes Programmieren« ist nicht bloß Marketing-Geschwätz, das Programmiersprachen-Anbieter für ihre Anzeigen ausgeheckt haben, um den Umsatz zu vervielfachen. Sicher, darum geht es auch, aber es steckt viel mehr dahinter. Nämlich eine Entwurfstechnik und eine Implementationsmethodologie, die Ihnen hilft, besseren Code zu schreiben. Jeder Visual-Basic-Programmierer sollte das lernen, insbesondere Neueinsteiger.

Dieses sehr empfehlenswerte Buch ist bei Markt & Technik erschienen.

Klassenprogrammierung ist nicht IN, sondern der erste Schritt zu änderungsfreundlichen und skalierbaren Anwendungen. Das Gegenteil davon ist übrigens Wegwerfcode, also Programme, die nur in einem Projekt Verwendung finden. Der Anpassungsaufwand in neuen Projekten ist größer als eine Neuentwicklung.

Halten wir uns an die Worte des Direktors: Der Worte sind genug gewechselt, lasst mich auch endlich Taten sehn!

10.1 Word.Application

Eine Klasse ist ein Codemodul, welches durch öffentliche Methoden, Eigenschaften und Ereignisse ein Objekt repräsentiert. Mit Objekten hatten wir ja schon reichlich zu tun.

Die folgenden Programmzeilen erzeugen ein neues Objekt und verwenden anschließend eine Methode und zwei Eigenschaften:

```
Dim wdApp As Word.Application
Set wdApp = New Word.Application
MsgBox wdApp.CheckSpelling("Schreibbfehler")
```

Sofern Sie einen Verweis zur Microsoft Word 8.0 Object Library eingerichtet haben, wird eine neue Instanz der Klasse `Word.Application` erzeugt. Diese Instanz ist nun zu einem Objekt geworden, welches im Code mit der Objektvariablen `appWord` angesprochen wird. Dieses Objekt `wdApp`, gewissermaßen eine Inkarnation der Klasse `Word.Application`, verfügt über eine Methode namens `CheckSpelling`, die das Literal »Schreibbfeh-

ler« entgegennimmt. Sie wird uns vermutlich ein `False` zurückgeben, da das übergebene Wort einen Schreibfehler enthält.

Dieses Beispiel ist natürlich ein schlechtes Beispiel, denn wir werden in Excel-VBA nichts schreiben können, was auch nur entfernte Ähnlichkeit mit Word hat. Andererseits ist es allerdings ein gutes Beispiel, weil die Instanz der Klasse `Word.Application` genauso erzeugt wird wie eine Instanz der Klassen, die in diesem Kapitel geschrieben werden.

Verweise auf selbst verfasste Klassen werden immer über das Schlüsselwort `New` erzeugt. Dies kann man auch mit dem Erstellen eines neuen Word-Dokuments vergleichen. Denn wenn Sie ein neues Word-Dokument erzeugen, wird eine Kopie der entsprechenden Dokumentvorlage erzeugt. Betrachten wir die Dokumentvorlage als Klasse, so wird das danach erzeugte Dokument ein Objekt.

Schlüsselwort `New`

Ein Objekt ist ein Stück zum Leben erweckte Software, das in einem eigenen Adressraum seine Methoden, Eigenschaften und Ereignisse implementiert. Ein Objekt *kann* für die Dauer seiner Lebenszeit Zustände annehmen.



Eine kleine Ergänzung: Ein Objekt *kann* während seiner Lebensdauer Zustände speichern, *muss* aber nicht und – in verteilten Anwendungen – *darf* zum Teil nicht. Doch diese verteilten Anwendungen interessieren uns im Moment noch nicht, wir werden aber darauf zu sprechen kommen.

10.2 Klassen

In Kapitel 5 wurde das Themengebiet Objekte bereits besprochen. In der Regel greifen wir in VBA auf bereits bestehende Objekte zu. Einen *zwingenden* Grund zum Erzeugen eigener Objekten gibt es eigentlich nicht. Die dahinterstehende Funktionalität ließe sich auch über ein ungebundenes Modul und öffentliche Prozeduren erreichen. Aber es gibt Argumente, die für den Einsatz von Klassen sprechen.

- ▼ Klassen belegen nur Speicherplatz, wenn eine Instanz davon erzeugt wurde.
- ▼ Klassen erhalten je Instanz einen eigenen Adressraum für Variablen.
- ▼ Klassen bieten perfekte Möglichkeiten zur Kapselung von Daten und Code.
- ▼ Klassen ermöglichen eine freie Skalierbarkeit von Anwendung und Daten.

- ▼ Klassen ersparen wiederholtes Codieren identischer Funktionalität.
- ▼ Klassen lassen sich direkt in DLLs integrieren.

Daneben zeugt die Verwendung von Klassen einfach von einem besseren Stil.

10.2.1 Klassenmodul erstellen

Sie fügen Ihrem Projekt ein Klassenmodul hinzu, indem Sie entweder im Menü Einfügen den Eintrag Klassenmodul oder im Kontextmenü des Projekt-Explorers Einfügen Klassenmodul wählen. Im Eigenschaftsfenster stehen uns nun die bereits bekannte (Name)-Eigenschaft, sowie die in VBA 6 neu hinzu-gekommene Instancing-Eigenschaft zur Verfügung. Als Präfix für Klassenmodule benutze ich `cls`, doch Sie finden in der Literatur Argumente für und wider die Verwendung eines Präfix bei Namen von Klassen und Objekten. Ändern Sie den Namen der Klasse bitte in `cls-Info`.

Der Standardwert der Instancing-Eigenschaft ist `Private`, was bedeutet, dass eine Instanz der Klasse nur innerhalb des Projekts erzeugt werden kann. Der zweite Wert `PublicNotCreatable` meint, dass die Klasse in anderen Projekten zwar sichtbar ist und somit auch angesprochen werden kann, aber das andere Projekt kann keine Instanz davon bilden. Bleiben wir bei `Private`.

In unserer neuen Klasse schreiben wir jetzt eine Prozedur, die einen übergebenen Text in einer `MessageBox` mit dem Info-Piktogramm ausgeben soll:

```
Public Sub InfoMessage(ByVal strMessage As String)
    MsgBox strMessage, vbInformation
End Sub
```

In einem normalen Modul schreiben Sie danach bitte folgende Prozedur:

```
Private Sub ClassTest()
    Dim cInfo As clsInfo
    Set cInfo = New clsInfo
    cInfo.InfoMessage "Hallöchen"
End Sub
```

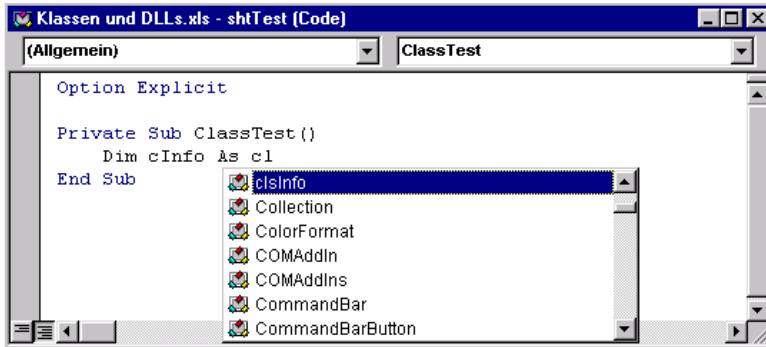



Abbildung 10.1: Code-Fenster mit Klasse clsInfo als Datentyp

Sobald Sie das Leerzeichen hinter `As` geschrieben haben, wird in der Liste der Datentypen unsere Klasse `clsInfo` bereits angeboten, wie Abbildung 10.1 zeigt. Auch in der nächsten Zeile wird sie als Element angeboten, nachdem das Leerzeichen hinter `New` angegeben wurde.

Wir haben jetzt ein Objekt des Namens `cInfo` erzeugt, welches uns seine Eigenschaften und Methoden dann anbietet, wenn nach dem Namen ein Punkt eingegeben wird, wie Abbildung 10.2 eindringlich zeigt:

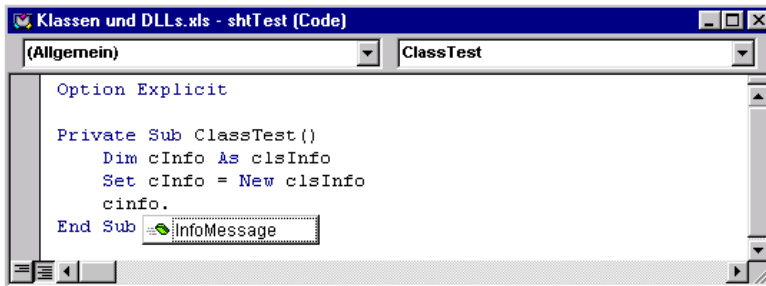


Abbildung 10.2: Code-Fenster mit Elementen des Objekts cInfo

Wenn Sie die Methode `InfoMessage` übernommen haben und ein Leerzeichen ergänzen, wird Ihnen die Methodensignatur als Parametrierungshilfe angeboten:

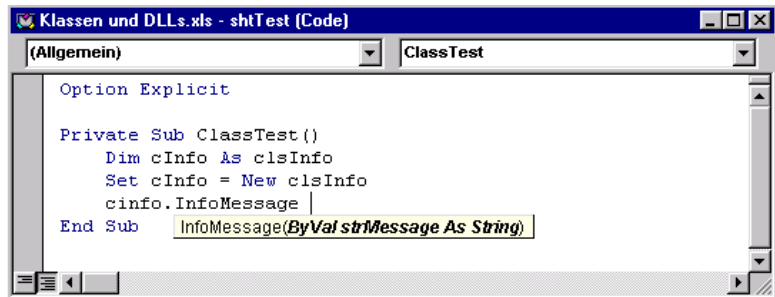


Abbildung 10.3: Code-Fenster mit Methodensignatur des Objekts cInfo

Lassen Sie unser kleines Programmchen einmal laufen und Sie werden, wie erwartet, eine MessageBox auf dem Bildschirm sehen. Spätestens jetzt haben Sie Ihre erste ActiveX-Komponente geschrieben.

10.2.2 Elemente einer Klasse

Objekte können über Methoden, Eigenschaften oder Ereignisse verfügen.

Methoden

Die Sub-Prozedur InfoMessage hat sich mit dem Symbol einer Methode in Abbildung 10.2 gezeigt. Somit können wir festhalten, dass Sub-Prozeduren Methoden sind. Ein Unterscheidungskriterium für Methoden ist deren Fähigkeit Werte zurückzugeben. Die Sort-Methode des Range-Objekts beispielsweise sortiert zwar die Tabelle, gibt aber keinen Wert zurück. Anders hingegen die SpecialCells-Methode, welche uns ein Range-Objekt übergibt, das den Regeln des Arguments Type entspricht. Methoden, die Werte zurückgeben sollen, werden als Function-Prozeduren realisiert.

Die folgende Prozedur fragt den Anwender, ob die Änderungen gespeichert werden sollen, und gibt die Antwort als True oder False zurück:

```
Public Function SaveChanges() As Boolean
    Dim strMessage As String
    strMessage = "Sollen die Änderungen gespeichert werden?"
    SaveChanges = MsgBox(strMessage, vbYesNo + vbQuestion)
End Function
```

An der aufrufenden Stelle könnte diese neue Methode so eingesetzt werden:

```
If cInfo.SaveChanges Then
    'Speichern
End If
```

Eigenschaften

Die einfachste Form einer Eigenschaft ist eine öffentliche Variable eines Klassenmoduls:

```
'in clsTest:
Public TestString As String

'in einem Modul:
cTest.TestString = "Hallo"
MsgBox cTest.TestString
```

Die MessageBox in diesem Beispiel wird »Hallo« ausgegeben.

VBA bietet uns aber ein Konstrukt, mit dem wir Eigenschaften etwas flexibler gestalten können. Wir können per Code festlegen, ob die Leserichtung, die Schreibrichtung oder beide unterstützt werden sollen:

```
Public Property Get TestString() As String
    TestString = ...
End Property

Public Property Let TestString(ByVal strNewValue As String)
    ... = strNewValue
End Property
```

Wenn Sie die Benennung der beiden Richtungsprozeduren `Property Get` und `Property Let` möglicherweise ein wenig verwirrt, so liegt das an der Betrachtungsrichtung. Stellen Sie sich die Klasse einfach als Dienstleister vor, übergeben Sie (`Let`) einen Wert an diesen Dienstleister und nehmen Sie einen Wert von diesem Dienstleister entgegen (`Get`). Abbildung 10.4 verdeutlicht diesen Zusammenhang.

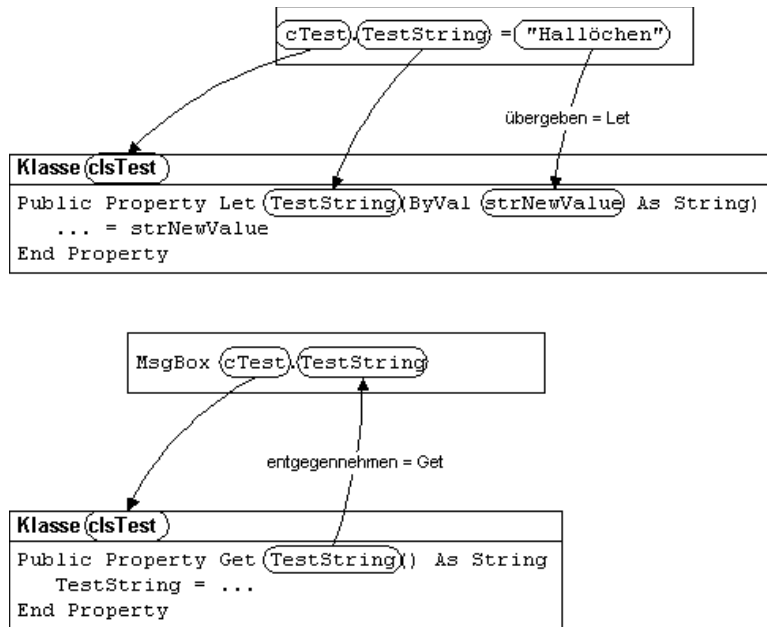


Abbildung 10.4: Wirkungsweise von Property Let und Get

Weshalb sollte man sich die Arbeit machen und anstelle einer Variablen mindestens sechs Zeilen Code schreiben, nur um einen Wert zu übergeben?

COM setzt eine Public Variable ohnehin in zwei Property-Prozeduren um und erfindet eine interne Variable, um den Wert zu speichern:

```
Private strTestString As String

Public Property Get TestString() As String
    TestString = strTestString
End Property

Public Property Let TestString(ByVal strNewValue As String)
    strTestString = strNewValue
End Property
```

Schön, wenn es COM hilft, soll COM es tun. Aber warum soll ich mir die Arbeit machen? Hier folgen einige Argumente für Property-Prozeduren:

- ▼ COM legt die Prozeduren immer paarweise an, ob Sie nun die Schreibrichtung erlauben wollen oder nicht.
- ▼ COM führt keine Gültigkeitsüberprüfung für Sie durch.

- ▼ Sie haben keine Kontrolle, wann die Werte geschrieben oder gelesen werden, kennen als Entwickler also den Zustand Ihres Objekts nicht.

Stellen Sie sich bei Klassen vor, dass nicht Sie, der die Interna der Klasse beherrscht, diese im Code verwendet, sondern ein anderer. Und dieser andere verlässt sich auf die korrekte Implementierung und die Datenintegrität der Klasse. Und mit Recht tut er dies, denn das ist ein Bestandteil des Klassenkonzepts.

Eigenschaft oder Methode?

In einer Reihe von Fällen ist die Entscheidung einfach.

Ist die Prozedur nicht mit dem Transport eines Wertes verbunden, dann scheidet eine Eigenschaft aus, wie zum Beispiel:

```
Application.Calculate
rngTest.Copy
```

Dies sind zwei typische Vertreter für eine per Sub implementierte Methode.

In weiterer Fall für die Verwendung einer Methode liegt dann vor, wenn Sie eine Reihe von Argumenten übergeben, aber keine Rückgabe erwarten:

```
rngTest.Sort Key1:=rngTest, Header:=xlYes, ...
```

Wenn Sie eine Rückgabe erwarten, haben Sie im Prinzip die Wahl, wie folgende Beispiele zeigen:

```
Public Property Get Cells(Optional ByVal RowIndex As Long, _
    Optional ByVal ColIndex As Long) As Range

Public Function Cells(Optional ByVal RowIndex As Long, _
    Optional ByVal ColIndex As Long) As Range
```

Microsoft hat sich dafür entschieden, eine Eigenschaft daraus zu machen, und hat sie auch nicht mit einer, sondern vermutlich mit mehreren Prozeduren polymorph implementiert. Doch auf diese Technik müssen wir in VBA noch bis zur Version 7 warten.

Wollen Sie hingegen schreiben und lesen, so ist die Entscheidung für eine Eigenschaft gefallen.

Initialize- und Terminate-Ereignis

Eine Klasse bietet uns intern noch zwei besondere Ereignisse. Das erste tritt auf, wenn die Klasse mit instantiiert wird.

Wenn Sie im Klassenmodul in der linken ComboBox `Class` auswählen, wird die Ereignisprozedur `Class_Initialize()` als Private Sub angelegt. In der rechten ComboBox stehen Ihnen nach bewährtem Muster beide Ereignisse zur Verfügung.

Wir können das `Class_Initialize`-Ereignis dazu nutzen, vorbereitende Arbeiten durchzuführen. Hierunter fällt all das, was Zeit raubend ist und ohnehin erledigt werden muss, zum Beispiel externe Werte ermitteln. Wir werden in Kapitel 10.4 noch sehen, dass dies ein zweischneidiges Schwert ist.

Das `Class_Terminate`-Ereignis wiederum tritt auf, bevor die Klasseninstanz aus dem Speicher entfernt wird.

Kehren wir kurz zu unserer Klasse `clsInfo` zurück. Wir stehen öfters vor dem Problem, den Anwender an verschiedenen Stellen unseres Programms dieselben Fragen zu stellen. Also könnten wir eine Methode schreiben, die dem Anwender eine der Standardfragen stellt und uns seine Entscheidung zurückgibt. Um nicht in jedem Aufruf denselben Fragetext an die Methode zu übergeben, greifen wir auf eine Enumeration zurück, die unsere vordefinierten Fragen als Konstanten bereithält. Ein `String-Array` enthält die eigentlichen Texte:

```
Private strMessage(2) As String
Public Enum enumMessage
    enumSaveChanges = 0
    enumDeleteFile = 1
    enumCalcuatateNow = 2
End Enum

Private Sub Class_Initialize()
    strMessage(0) = "Sollen die Änderungen gespeichert w.?"
    strMessage(1) = "Sollen die Dateien gelöscht werden?"
    strMessage(2) = "Sollen die Daten nun berechnet werden?"
End Sub

Public Function ShallWe(ByVal lngMessage As enumMessage) _
    As Boolean
    ShallWe = MsgBox(strMessage(lngMessage), _
        vbYesNo + vbQuestion)
End Function
```

Hier wird das `Class_Initialize`-Ereignis dazu genutzt, die vordefinierten Texte in das Array zu laden. Die Variable `lngMessage` der Methode `ShallWe`, die unseren Zeiger entgegennimmt, deklarieren wir als `enumMessage`. Dadurch bietet die Methode beim Schreiben des Aufrufs die in

der Enumeration deklarierten Konstanten an, wie Abbildung 10.5 zeigt. Eine solche Programmzeile liest sich auch im Code absolut interpretationsfrei, sofern Ihre Konstantennamen gut gewählt sind:

```
If cInfo.ShallWe(enumSaveChanges) Then
    ...
End If
```

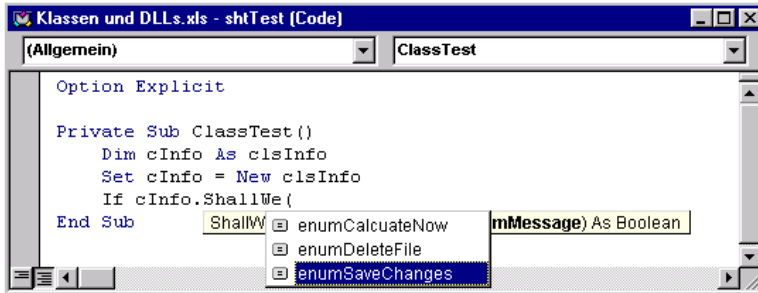


Abbildung 10.5: Die Enumeration beim Schreiben der Methode

Die Ereignisprozedur `Class_Terminate` kann beispielsweise dazu verwendet werden, Dateien zu schließen oder Ähnliches. Aber auch sie hat ihre Tücken, da sie sich nicht unserer aktivierten Fehlerbehandlungsroutinen bedient oder besser gesagt bedienen kann. Potenziell fehlerträchtige Anweisungen sind in einer separaten Aufräummethode besser aufgehoben.

As New versus = New

Es stehen folgende Wege zur Disposition:

```
Dim cTest As New clsTest
```

und

```
Dim cTest As clsTest
Set cTest = New clsTest
```

Dies ist ein Punkt, über den man trefflich streiten kann. Betrachten wir zunächst das unterschiedliche Verhalten beider Wege bezüglich des Auslösens des Ereignisses `Class_Initialize`. In den beiden folgenden Beispielen ist die Zeile jeweils hervorgehoben, die dieses klasseninterne Ereignis auslöst:

```
Dim cInfo As clsInfo
Set cInfo = New clsInfo
```

```
Dim cInfo As New clsInfo
' ...
cInfo.InfoMessage "Hallöchen"
```

Im ersten Fall wissen wir konkret, wann die Klasse mit Leben gefüllt wird, denn es ist die Stelle, an der wir `Set cInfo = New clsInfo` geschrieben haben. Steht hingegen bereits in der Deklaration `As New`, so wird die Instanz der Klasse automatisch dann gebildet, wenn das Objekt zum ersten Mal verwendet wird. Man könnte den Diskurs dadurch entscheiden, dass man es davon abhängig macht, ob man im `Class_Initialize`-Ereignis umfangreiche Operationen durchführen muss oder nicht. Doch auch das ist problematisch, wie Kapitel 10.4 noch zeigen wird. Wenn Sie umfangreiche Initialisierungen durchführen müssen, empfiehlt es sich, eine eigene Methode (z. B. `Intitalize`) zu schreiben, und die können Sie auch sicher dadurch starten, dass sie einfach aufgerufen wird:

```
Dim cInfo As New clsInfo
' ...
cInfo.Initialize
```

Mit dem Kriterium der Initialisierung lässt sich die Frage demnach nicht klar beantworten. Werfen wir also einen Blick auf das Zerschlagen der Objektinstanz, was uns durch das Ereignis `Class_Terminate` signalisiert wird. In den folgenden Beispielen ist die Zeile jeweils hervorgehoben, die dieses Ereignis auslöst:

```
Dim cInfo As clsInfo
Set cInfo = New clsInfo
cInfo.InfoMessage "Juhu"
Set cInfo = Nothing
```

```
Dim cInfo As clsInfo
Set cInfo = New clsInfo
cInfo.InfoMessage "Juhu"
End Sub
```

```
Dim cInfo As New clsInfo
' ...
End Sub
```

Eine Objektinstanz wird also dann zerschlagen, wenn sie über die Anweisung `Set ... = Nothing` läuft. Aber auch dann ist ihr Ende besiegelt, wenn der Code den Gültigkeitsbereich verlässt. Ist die Klasse lokal deklariert, läutet ihre Glocke bei der Anweisung `End Sub` oder `End Function` und das `Class_Terminate`-Ereignis wird ausgelöst.

Bei privaten Variablen wird das Objekt zerschlagen, wenn der Kontext des Moduls, in dem sie privat deklariert ist, aufgelöst wird. Wenn Sie beispielsweise eine Klasse in einem Form-Modul privat deklariert haben und die Instanz der Form zerschlagen, wird auch die Klasseninstanz aufgelöst. Das folgende Beispiel zeigt den Code in einem Formmodul:

```
Private cTest As clsTest

Private Sub UserForm_Initialize()
    Set cTest = New clsTest
End Sub
```

In einem anderen Modul wird die Form per Objektvariable verwendet:

```
Dim fTest As frmTest
Set fTest = New frmTest
fTest.Show
Set fTest = Nothing
```

Wird die Form über die Methode `Me.Hide` geschlossen, wird das `Terminate`-Ereignis der Klasse `clsTest` mit der Zeile `Set fTest = Nothing` ausgelöst. Schließen Sie die Form aber über `Unload Me` oder per Button in der Titelleiste, so ist die Klasseninstanz schon zerstört, bevor die aufrufende Stelle (dort, wo `fTest.Show` steht) den Fokus wieder erhält.

Ich habe auch reichlich darüber nachgedacht, ob ich Ihnen das Folgende noch erzählen soll, denn es ist dazu geeignet, Sie vollends zu verwirren. Aber es dient dem Verständnis von Klassen und Objekten.

Nehmen wir einmal an, wir hätten eine Form namens `frmTest` in einem Projekt. Wenn wir die Form als Objektinstanz laden, wird sie in der ersten markierten Zeile als Instanz in den Speicher geladen und in der zweiten markierten Zeile wieder aus dem Speicher entfernt:

```
Dim fTest As frmTest
Set fTest = New frmTest 'hier wird die Instanz geladen
fTest.Show
Set fTest = Nothing     'hier wird die Instanz aufgelöst
```

Wann wird aber die Form in folgender Variante wieder aus dem Speicher entfernt:

```
frmTest.Show
```

Wenn Sie eine Form ohne explizite Instanz laden, wird sie beim Schließen des Projekts aus dem Speicher geladen, denn sie verhält sich wie eine per `New` deklarierte Instanz:

```
Public frmTest As New frmTest
```

Wenn Sie diese Instanz aus dem Speicher entfernen wollen, müssen Sie eine der beiden folgenden Anweisungen nach dem Aufruf `frmTest.Show` durchlaufen:

```
Set frmTest = Nothing  
Unload frmTest
```

Und nun kommt ein Effekt, der unter dem Namen Stehaufmännchen bekannt geworden ist. Wenn Sie eine per `As New` deklarierte Instanz zerschlagen haben und anschließend prüfen wollen, ob sie auch wirklich verschwunden ist, wird sie erneut geladen. Probieren Sie einmal folgenden Code aus:

```
Dim cTest As New clsTest  
Set cTest = Nothing  
If Not cTest Is Nothing Then  
    MsgBox "Da bin ich wieder."  
End If
```

Na, ist die `MessageBox` gekommen?

Nach `Set cTest = Nothing` ist sie aus dem Speicher verschwunden, was Sie ja leicht nachprüfen können, indem Sie ein `Initialize`- und ein `Terminate`-Ereignis anlegen und eine entsprechende `MessageBox` ausgeben lassen. Wenn Sie aber fragen, ob `cTest` denn `Nothing` sei, passiert Folgendes: VBA weiß durch die Deklaration `As New`, dass es sich eine Instanz von `clsTest` dann erzeugen soll, wenn darauf zugegriffen wird. Die frei übersetzte Frage: »Schau mal nach, ob `cTest` `Nothing` ist.« wird interpretiert mit: »Einen Moment bitte, ich muss mir noch schnell eine Instanz davon bilden, ehe ich die Frage beantworten kann.« Und schon ist sie wieder da, ohne dass sie wieder entfernt wird.

Das sollten Sie wissen, wenn Sie Instanzen durch `As New` deklarieren oder Formulare ohne explizite Instanzenbildung laden. Der folgende Codeausschnitt funktioniert zwar, lädt die Form in der letzten Zeile aber erneut, sofern sie über `Unload Me` oder den Button in der Titelleiste der Form geschlossen wurde:

```
frmTest.Show  
Unload frmTest
```

Sie ist zwar anschließend wirklich aus dem Speicher entfernt, wurde aber zum Entfernen möglicherweise zuerst geladen. Wenn Ihnen das klar ist, dann haben Sie Objektinstanzen verstanden.

Ereignisse

Seit Excel 2000 können Klasseninstanzen Ereignisse erzeugen. Um ein Ereignis zu deklarieren, müssen Sie im Klassenkopf etwa Folgendes schreiben:

```
Public Event Change()
```

In einer beliebigen Prozedur können Sie dieses Ereignis dann mit dieser Anweisung erzeugen:

```
RaiseEvent Change
```

In dem Modul, das die Klasse verwendet, müssen Sie VBA davon in Kenntnis setzen, dass Sie von der Objektinstanz Ereignisse erwarten:

```
Private WithEvents cTest As clsTest
```

Danach steht cTest in der linken ComboBox des Codefensters und legt Ihnen bei Auswahl automatisch eine Ereignisprozedur an, die so aussieht:

```
Private Sub cTest_Change()  
End Sub
```

Ist das eine nette Spielerei oder macht das tatsächlich Sinn?

Das hängt ganz von Ihrem Projekt ab. Im folgenden Beispiel gehen wir davon aus, dass unser Code an verschiedensten Stellen Eigenschaften der Klasse verändern kann, wir aber wissen möchten, ob das der Fall war. Um dies zu bewerkstelligen, könnten wir an jeder Stelle, die schreibend auf die Klasse zugreift, eine logische Variable auf True setzen, ein so genanntes Dirty-Flag. Dadurch handeln wir uns aber möglicherweise viele identische Zeilen ein. Einfacher ginge das über ein Ereignis, welches von der Klasse dann ausgelöst wird, wenn eine Property Let durchlaufen wird. Hier der Klassencode:

```
Private strName As String  
Public Event Change()  
  
Public Property Let Name(ByVal strNewName As String)  
    strName = strNewName  
    RaiseEvent Change()  
End Property
```

Im aufrufenden Modul sieht das Ganze so aus:

```
Private WithEvents cTest As clsTest  
Private bTestDirty As Boolean
```

```
Sub AlterName()
    Set cTest = New clsTest
    cTest.Name = "Ich bin's."
End Sub
```

```
Private Sub cTest_Change()
    bTestDirty = True
End Sub
```

Die Zeile, die das Dirty-Flag verändert, taucht somit nur einmal im Code auf.

Sie können bei diesen eigenen Ereignissen auch Argumente angeben, die mit dem Ereignis übergeben werden müssen oder auch können, wenn sie optional deklariert sind:

```
Public Event Change(ByVal VariableName As String, _
                    ByVal NewValue As Variant)

Public Property Let Name(ByVal strNewName As String)
    ...
    RaiseEvent Change("Name", strNewName)
```

Die Ereignisprozedur nimmt dann diese Form an:

```
Private Sub cTest_Change(ByVal VariableName As String, _
                        ByVal NewValue As Variant)
```

Ich möchte Ihnen noch ein Beispiel skizzieren, das einem aktuellen Projekt entnommen ist. Ein AuswahlDialog ermöglicht eine flexible Angabe von freien Parametern, die eine riesige Menge an Produktvarianten transparent machen soll. Jedes Dialogelement war an eine Klassenprozedur gekoppelt, in der diese über die aktuelle Einstellung informiert wurde. Wir implementierten ein Ereignis in der Klasse, das die Formularinstanz davon in Kenntnis setzte, wenn die aktuelle Auswahl zu einer Produktvariante führte, die als Sonderfertigung einen Aufpreis zur Folge hatte. Der Klasse lag kein Variantenkatalog zugrunde, sondern ein Variantengenerator mit einer Reihe recht interessanter Regeln, unter denen sich auch einige Ausschlussregeln befanden, die ihrerseits zu einem Geht-nicht-Ereignis führen können. Eine vorherige Prüfung aller Varianten ist aus Performancegründen nicht möglich.

10.3 Eine Datenklasse für den Login-Dialog

In Kapitel 5 bauten wir einen Login-Dialog, den wir allerdings mit einer Reihe von Provisorien verließen. Darunter der Zugriff auf die in der Tabelle User zu hinterlegenden Namen nebst dazugehörigen Passwörtern. Es

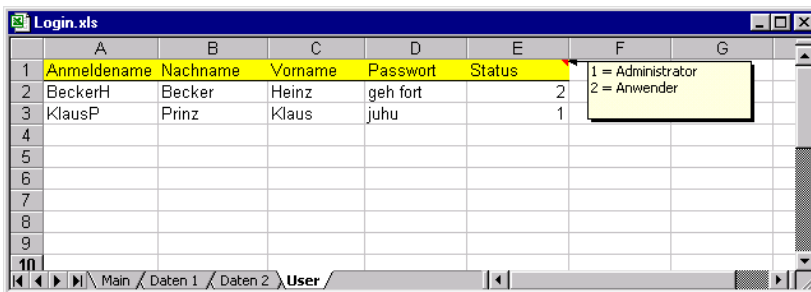
war auch die Rede davon, diesen Dialog so zu gestalten, dass er sowohl mit dem Zugriff auf diese in derselben Arbeitsmappe hinterlegten Anwenderdaten arbeiten, aber auch zusammen mit einer Codekomponente auf eine (zentrale) Benutzerdatenbank zugreifen soll. Die Entscheidung darüber sollte im Prinzip damit fallen, welche Klasse in das Projekt integriert wird. Danach soll der Dialog sofort loslaufen.

10.3.1 Die Tabelle User

Es hat sich als sinnvoll erwiesen, nicht nur den Zugang zu Arbeitsmappen über einen Login zu steuern, sondern auch innerhalb der weiteren Anwendung zumindest zwischen Anwender und Administrator zu unterscheiden. Wer sollte denn sonst die Anwenderdaten pflegen? Eine weitere Gliederung in drei oder gar mehr Stufen sollte für Sie am Ende dieses Kapitels kein Problem mehr darstellen.

Da es in vielen Unternehmen üblich ist, mehr oder weniger verunstaltete Anmeldenamen aus Vor- und Nachnamen zu bilden, macht es Sinn, neben dem Anmeldenamen auch noch Name und Vorname mitzuführen, um ihn zum Beispiel für ein Logbuch zu verwenden oder als Klartextname in Dokumente einzufügen, die von dieser Applikation möglicherweise generiert werden. Den Anmeldenamen selbst wollten wir ja mittels des API ermitteln.

Abb. 10.6 zeigt unsere Tabelle User, die also nun die Spalten Anmeldenamen, Nachname, Vorname, Passwort und Status enthält.



	A	B	C	D	E	F	G
1	AnmeldeName	Nachname	Vorname	Passwort	Status	1 = Administrator 2 = Anwender	
2	BeckerH	Becker	Heinz	geh fort	2		
3	KlausP	Prinz	Klaus	juhu	1		
4							
5							
6							
7							
8							
9							
10							

Abbildung 10.6: Login, Tabelle User

10.3.2 Funktionale Überlegungen

Da wir keine Validierung mit der momentanen Forms-2-Bibliothek durchführen können, erübrigt sich eine Überprüfung des Anwendernamens beim Verlassen der `txtName`, in der sich der Name des anzumeldenden Anwenders befindet. Wie in Kapitel 8 bereits besprochen, könnten wir ihn über Variablen im Code nachbilden.

Somit könnte die Überprüfung von Anmeldenamen und Passwort zentral mit einer Methode statt finden, die als Boolean ein `True` oder ein `False` zurückgibt. Nachteilig daran wäre, dass sie nicht unterscheiden könnte, ob der Name falsch ist, das Passwort oder gar beide. Es wäre wohl besser, die Existenz des Anmeldenamens separat zu prüfen. Theoretisch würde es ausreichen, danach den Zeilenzeiger des ersten Zugriffs zu verwenden und nur noch das Passwort zu prüfen. Aber irgendwie erzeugt es ein mulmiges Gefühl, den Zugang zu dem Programm und somit zu sensiblen Daten darauf aufzubauen, dass die User-Klasse immer ordnungsgemäß verwendet wird. Worüber reden wir eigentlich? Doch wohl nur darüber, den Anmeldenamen bei der Prüfung des Passworts erneut zu übergeben. Sicherheit geht vor. Also übergeben wir ihn erneut.

Daneben benötigen wir drei Eigenschaftsprozeduren für Nachname (Name), Vorname (FirstName) und Status (UserLevel), die nur gelesen werden müssen; auf den Anmeldenamen und das Passwort können wir im weiteren Programmverlauf verzichten. Da im späteren Code zumindest bekannt sein muss, ob der Anwender Administrator ist oder eben nicht, stehen uns mehrere Möglichkeiten zur Verfügung:

- ▼ eine Eigenschaft, die den Status im Klartext zurückgibt,
- ▼ eine Administrator-Eigenschaft des Typs Boolean oder
- ▼ eine Enumeration mit Konstanten für die einzelnen Stati.

Sie erinnern sich sicherlich noch an den Tag der UserForm, dem wir die Literale »OK« bzw. »Cancel« mit auf den Weg gaben, um sie in der Ereignisroutine `Workbook_Open` auszuwerten. Es wäre aber auch denkbar, eine Eigenschaft zu schreiben, die das Ergebnis `True` oder `False` zurückgibt.

10.3.3 Implementierung

Die letzte Variante stellt sich wohl als die flexibelste Lösung dar. Schauen wir uns die Schnittstelle an:

```
Public Function CheckName(ByVal strName As String) _
    As Boolean
End Function

Public Function CheckLogin(ByVal strName As String, _
    ByVal strPassword As String) As Boolean
End Function

Public Property Get Name() As String
End Property
```

```
Public Property Get FirstName() As String
End Property
```

```
Public Property Get UserLevel() As enumUserLevel
End Property
```

Somit ergibt sich die Notwendigkeit eine Enumeration zu bauen:

```
Public Enum enumUserLevel
    enumUserLevelAdmin = 1
    enumUserLevelUser = 2
End Enum
```

Einer guten Sitte folgend sollten wir Zeiger auf die verwendeten Spalten bilden:

```
Private Const iColLoginName As Long = 1
Private Const iColName As Long = 2
Private Const iColFirstName As Long = 3
Private Const iColPassword As Long = 4
Private Const iColUserLevel As Long = 5
```

Wir könnten zwar die Werte für die Eigenschaftsroutinen in Variablen ablegen, ein privater Zeilenzeiger ist aber wohl wirtschaftlicher:

```
Private iRow As Long                      'Zeilenzeiger
```

Last but not least sollten wir eine private Variable deklarieren, die einen Verweis auf das Worksheet enthält:

```
Private shtUser As Worksheet            'Verweis auf Tabelle User
```

Für die Zuweisung dieser Variablen bietet sich die Class_Initialize-Prozedur an, mit deren Codierung wir nun beginnen:

```
Private Sub Class_Initialize()
    ' Zuweisen der Worksheet-Variablen
    On Error GoTo errHandler
    Set shtUser = ThisWorkbook.Worksheets("User")
    Exit Sub

```

```
errHandler:
    Err.Raise Err.Number
End Sub
```

Der einzige Fehler, der hier auftauchen kann, ist der fehlgeschlagene Zugriff auf die Tabelle »User«. Eine besondere Behandlung ist hier nicht er-

forderlich, da alle Fehler in der `Workbook_Open`-Prozedur, wo dieser Fehler auch landen wird, zum Schließen der Arbeitsmappe führen (müssen).

Unsere Methode `CheckName` muss den übergebenen Namen in der Tabelle suchen, auf die unsere `shtUser` zeigt. Somit scheint eine Suche in der durch den Spaltenzeiger `iColLoginName` verzeigten Spalte mittels der `Find`-Methode probat. Ist das Ergebnis `Nothing`, so existiert der Anmelde-name nicht:

```
Public Function CheckName(ByVal strName As String) _
    As Boolean
    ' Prüfen, ob der übergeben Name existiert
    On Error GoTo errHandler
    CheckName = Not shtUser.Columns(iColLoginName).Find( _
        what:=Name, LookIn:=xlValues, _
        lookat:=xlWhole) Is Nothing
    Exit Function

errHandler:
    Err.Raise Err.Number, "clsUser:CheckName"
End Function
```

Der eigentliche Code ist ein Einzeiler, indem das Ergebnis des Vergleichs der Rückgabe der `Find`-Methode mit dem Wert `Nothing` invertiert dem Funktionsnamen `CheckName` zugewiesen wird. Diese Zeile noch einmal in etwas stilisierter Form:

```
CheckName = Not (Spalte.Find(...) Is Nothing)
```

Dass ein Anmelde-name mehrfach vorhanden ist, können wir ausschließen, denn so viel Sorgfalt müssen wir von dem Administrator erwarten können. Somit können wir das erste Auftauchen des Anmelde-namens als das einzige betrachten und nur noch das Passwort in der so gefundenen Spalte überprüfen. Diesmal reicht uns eine Zeile nicht:

```
Public Function CheckLogin(ByVal strName As String, _
    ByVal strPassword As String) As Boolean
    ' Prüfen, ob Name zusammen mit Passwort existiert
    Dim rngFound As Range 'Rückgabe der Find-Methode
    On Error GoTo errHandler
    Set rngFound = shtUser.Columns(iColLoginName).Find( _
        what:=strName, LookIn:=xlValues, lookat:=xlWhole)
    If Not rngFound Is Nothing Then
        If shtUser.Cells(rngFound.Row, iColPassword).Value _
            = strPassword Then
            iRow = rngFound.Row
```



```

        CheckLogin = True
    End If
End If
Exit Function

```

```

errHandler:
    Err.Raise Err.Number, "clsUser:CheckLogin"
End Function

```

Bleiben noch die drei Eigenschaftsprozeduren, deren Modulköpfe und ErrorHandler weggelassen wurden:

```

Public Property Get Name() As String
    Name = shtUser.Cells(iRow, iColName).Value
End Property

Public Property Get FirstName() As String
    FirstName = shtUser.Cells(iRow, iColFirstName).Value
End Property

Public Property Get UserLevel() As enumUserLevel
    UserLevel = shtUser.Cells(iRow, iColUserLevel).Value
End Property

```

10.3.4 Verwendung der Klasse

Wir benötigen ein Modulblatt, um die öffentlichen Variablen der Fehler- und User-Klassen zu deklarieren. Die Instantiierung erfolgt in `Workbook_Open` an der frühest möglichen Stelle, wobei aus alter Gewohnheit zuerst die Fehlerklasse erzeugt wird. Im Übrigen bleibt die `Workbook_Open` unverändert.

Im Code der `frmLogin` wird die Klasse natürlich verwendet, aber nur in der Ereignisroutine der OK-Schaltfläche, wo ihre Aufrufe an die Stelle der ehemaligen Textvergleiche treten, wodurch wir aus Gründen der Vereinfachung in Kapitel 8 den Klassenzugriff vorwegnahmen. Hier nun ein Auszug dieser Routine, in der die beiden veränderten Zeilen hervorgehoben sind:

```

Private Sub cmdOK_Click()

    If cUser.CheckName(txtName.Text) Then
        If cUser.CheckLogin(txtName.Text, _
                               txtPassword.Text) Then

            'alles OK
        Else

```

```

        'Passwort falsch
    End If
Else
    'Name unbekannt
End If
End Sub

```

An beliebiger Stelle im weiteren Code der Anwendung könnten Zugriffe nach folgendem Muster enthalten sein:

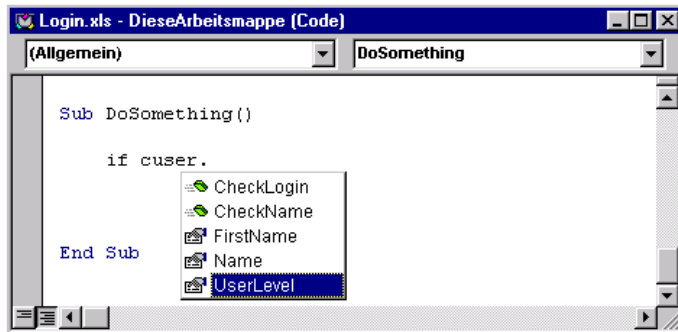


Abbildung 10.7: Login, Zugriff auf Elemente der Klasse clsUser

Und nun noch die hinter der UserLevel steckende Enumeration:

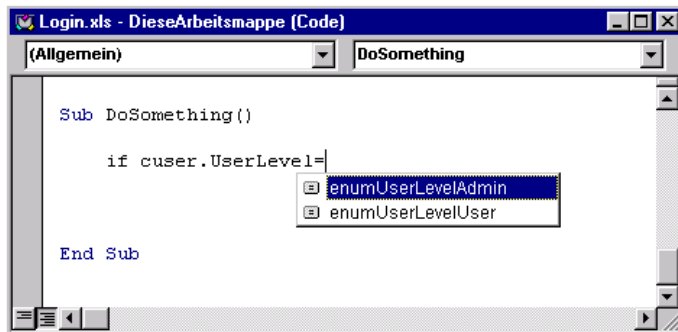


Abbildung 10.8: Login, die Enumeration hinter UserLevel

10.4 Fehlerbehandlung in Klassen

Im Kapitel 9 haben wir uns gründlich mit der Fehlerbehandlung auseinander gesetzt. Einige Aspekte blieben dort unberücksichtigt, die im Zusammenhang mit Klassen zu Tage treten. Und diese möchte ich Ihnen hier noch vorstellen. Außerdem werden wir uns noch mit Fehlernum-

mern befassen, die im Zusammenhang mit Komponenten von Interesse sind.

Auf der Buch-CD finden Sie eine Datei namens Klassenfehler, in der Sie diese nachfolgend untersuchten Probleme nachvollziehen können.

10.4.1 Ein Fehler im Initialize-Ereignis einer Klasse

Wenn ein Fehler in der Ereignisprozedur `Class_Initialize` einer Klasse auftritt, so passiert Folgendes:

- ▼ Die Klasse wird nicht initialisiert, wodurch kein gültiger Verweis auf die Klasse vorliegt.
- ▼ `Err.Number` wird mit 440 überschrieben und der Ursprungsfehler geht verloren, als `Err.Source` wird der Projektname zurückgegeben und als `Err.Description` erscheint Automatisierungsfehler.

Hierbei ist es einerlei, ob der Fehler in der `Class_Initialize`-Routine erzeugt wird oder der dortige `ErrorHandler` nur als aktivierte Fehlerbehandlungsroutine ins Spiel gekommen ist.

Als Erstes sehen wir uns den Code im `Class_Initialize`-Ereignis an:

```
Private Sub Class_Initialize()
    Dim lngTest As Long
    On Error GoTo errHandler
    lngTest = 10 / 0
    Exit Sub

errHandler:
    Err.Raise Err.Number, "clsNOK:Class_Initialize"
End Sub
```

Hier wird ein einfacher Überlauffehler erzeugt. Im `ErrorHandler` erfolgt nun das Weiterreichen des Fehlers an die aufrufende Stelle, die folgendermaßen codiert ist:

```
Private Sub cmdCreateInstanceNOK_Click()
    On Error GoTo errHandler
    Set cNOK = New clsNOK
    Exit Sub

errHandler:
    MsgBox Err.Number & vbCrLf & Err.Source & vbCrLf & _
        Err.Description

End Sub
```

Was passiert da? Ist es ein Bug oder nicht?

Der einzige Fehler, der hier passiert, besteht darin, dass VBA uns ernst nimmt. Das `Initialize`-Ereignis dient dazu, die Klasse auf ihre erste Verwendung vorzubereiten, denn wir erwarten ein voll funktionstüchtiges Objekt, nachdem wir die Klasse initialisiert haben. Anderenfalls hätten wir den Code schließlich nicht an dieser Stelle untergebracht. Ein Fehler, die nun im `Initialize`-Ereignis oder einer von dort aus aufgerufenen Prozedur auftritt, veranlasst VBA zu der Annahme, dass die Klasse nun nicht funktionstüchtig sei. Und genau das meldet die Klasse über den Fehler 440 und nennt dies einen Automatisierungsfehler, wogegen im Prinzip nichts einzuwenden ist.

Was ist aus unserem Fehlerkontext `clsNOK:Class_Initialize` geworden? Es hat den Anschein, dass VBA sein abschließendes Urteil über Erfolg oder Misserfolg der Initialisierung unserer Klasse erst im allerletzten Moment fällt, also nach der Anweisung

```
Err.Raise Err.Number, "clsNOK:Class_Initialize".
```

Und dadurch ist der Fehler 440 ein neuer Fehler, der zwangsläufig auch nicht den Kontext seines Vorgängers übernimmt.

Diese These, dass die Klasse sich streng an den Sinn unseres Codes hält, lässt sich dadurch belegen, dass ein `Resume Next` als letzte Anweisung des `ErrorHandlers` bewirkt, dass die Initialisierung problemlos zu Ende geführt wird. Hätten wir der Klasse also mit einem `Resume Next` zu verstehen gegeben, dass wir den Fehler für nicht gravierend genug halten, dass er die Initialisierung im Ergebnis in Frage stellen kann, so wäre alles gut gegangen. Wir aber entschieden uns, den Fehler nach oben weiter zu melden. Logischerweise führt ein `On Error Resume Next` auch dazu, dass die Objektinstanz problemlos erzeugt wird, ohne dass wir den Fehler 440 zu Gesicht bekommen.

Also kein Bug.

Die Moral von der Geschichte ist, dass im `Class_Initialize` kein Code auftauchen darf, der zu irgendeinem Fehler führen könnte. Das Öffnen von Arbeitsmappen, einer Datenbank oder einer Verbindung zu einer Datenbank hat im `Initialize`-Ereignis nichts verloren. Wir haben nämlich keine Chance, dem Anwender so banale Sachverhalte wie das Fehlen eines Verzeichnisses, das er in der Zwischenzeit umbenannt oder gelöscht haben könnte, per `MessageBox` mitzuteilen, da Fehlertext und Fehlerkontext überschrieben wurden. Für die genannten Dateioperationen sollten wir also eine eigene Initialisierungsroutine schreiben, deren eventuelle

Fehler im Original auch an der gewünschten Stelle unverfremdet ankommen:

```
Set cTest = New clsTest
cTest.Initialize
```

Auf `Class_Initialize` sollten wir fortan verzichten.

10.4.2 Fehler im Terminate-Ereignis einer Klasse

Tritt ein Fehler in der `Class_Terminate`-Prozedur auf, den wir nach den Regeln der Kunst im `ErrorHandler` mit `Err.Raise` an die aufrufende Stelle weitermelden, so hat es den Anschein, als ob keine aktivierte Fehlerbehandlungsroutine verfügbar ist.

Schauen wir uns auch hier den Code der Klasse an:

```
Private Sub Class_Terminate()
    Dim lngTest As Long
    On Error GoTo errHandler
    lngTest = 10 / 0
    Exit Sub

errHandler:
    Err.Raise Err.Number, "clsNOK:Class_Terminate"
End Sub
```

Nun verfügt unser Code aber wohl über eine aktivierte Fehlerbehandlungsroutine, wie die betreffende Prozedur beweist:

```
Private Sub cmdReleaseInstanceNOK_Click()
    On Error GoTo errHandler
    Set cNOK = Nothing
    Exit Sub

errHandler:
    MsgBox Err.Number & vbCrLf & Err.Source & vbCrLf & _
        Err.Description

End Sub
```

Die Frage stellt sich nun, ob ein Bug in VBA die Liste der aktivierten Fehlerbehandlungsroutinen nun nicht findet oder gar löscht?

Aber auch hier ist kein Bug zugange, wie ein Blick in die MSDN offenbart, denn unter Visual Basic Concepts steht in *Coding Robust Initialize and Terminate Events* zu lesen:

Errors in the Terminate event require careful handling. Because the Terminate event is not called by the client application, there is no procedure above it on the call stack.

Das Terminate-Ereignis wird also gar nicht von der Applikation selbst aufgerufen und hat somit auch keinen Zugriff auf die aktivierten ErrorHandler der Applikation!

Die Konsequenz hieraus ist die gleiche, die sich auch aus dem Problem im Initialize-Ereignis ergab. Dateizugriffe und sonstige Operationen, die zu einem Fehler führen könnten, sollten somit nicht im Terminate-Ereignis erscheinen. Schreiben Sie eine eigene Methode, die solche Operationen durchführt, und rufen Sie diese auf, *bevor* Sie die Klasseninstanz auflösen:

```
cTest.CleanUp 'oder auch cTest.Terminate
Set cTest = Nothing
```

10.4.3 Die Konstante vbObjectError

Solange wir keine Klassenmodule schreiben, müssen wir uns nicht weiter um diese geheimnisvolle Konstante kümmern. Bei Klassen stehen wir aber mitten in diesem Thema und sollten uns auch ein wenig an den Regeln der Komponentenentwicklung orientieren.

ActiveX-Komponenten liefern im Fehlerfalle einen Long-Wert als Fehlernummer, der folgendermaßen aufgebaut ist:

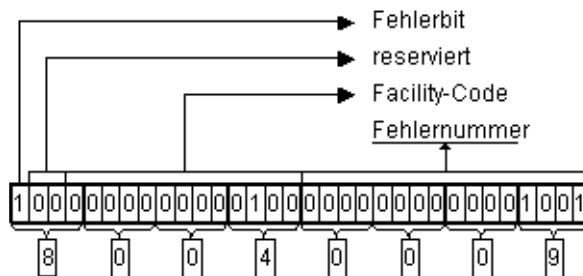


Abbildung 10.9: Struktur einer ActiveX-Fehlernummer am Beispiel des Fehlers 9

Diese Konstante ist im Ursprung ein OLE-Rückgabewert, dessen Bit 31 über Erfolg und Misserfolg eines OLE-Aufrufs Aufschluss gibt. Die beiden reservierten Bits sind *for further use*, wie es so schön heißt. Und dann gibt es noch einen 13-stelligen Facility-Code, der bis auf wenige Ausnahmen den Wert 4 innehat.


```
MsgBox strErrIntro & vbCrLf & vbCrLf & _
    strContext & vbTab & Err.Source & vbCrLf & _
    strNumber & vbTab & _
    (Err.Number And lngErrorMask) & vbCrLf & _
    strText & vbTab & Err.Description, vbExclamation, _
    strTitle
```

Auslösen eines Komponentenfehlers

Microsoft empfiehlt, einem Fehler in Komponenten, also Klassen, die Konstante `vbObjectError` zu unserem Fehler hinzuzuaddieren. Des weiteren beansprucht Microsoft die ersten 512 Fehler für sich und räumt uns den ganzen Rest von 513 bis 65535 ein, also:

```
Err.Raise vbObjectError + 512 + x
```

Hier stehen Ihnen gleich mehrere Möglichkeiten zur Auswahl:

- ▼ Wir verwenden mehr oder weniger pauschale Fehlernummern wie 9999 und 9998 oder Ähnliches.
- ▼ Die gewohnten Fehlernummern werden anstelle von `x` den beiden Konstanten `vbObjectError` und 512 hinzuaddiert. Aus dem berühmten Indexfehler würde dann der demaskierte Wert 521 ($512 + 9$).
- ▼ Wir bilden eine runde dezimale Basis von 1000 oder 10000, zählen den gewünschten Fehlerwert hinzu und erhalten demaskiert 10009 als Indexfehler.

Eine Empfehlung ist hier schwierig, außer vielleicht: Wählen Sie Ihren Weg und bleiben Sie dabei.

Fehlerbehandlung in Komponentenketten

Wenn Sie jedoch eine Kette solcher Komponenten verwenden, wie im folgenden Beispiel zu sehen ist, dann müssen Sie damit rechnen, dass Sie einen Fehler in einer Klasse erzeugen und im ErrorHandler einer zweiten Klasse weiterleiten. In diesem Falle dürfen Sie die Konstante `vbObjectError` natürlich nur einmal zu dem Fehler hinzuaddieren, weil Sie sonst einen Überlauf in der `Number`-Eigenschaft produzieren.

In Fehlerbehandlungsroutinen von Klassenprozeduren müssen Sie den Fehler vor jedem Weiterleiten mit `lngErrorMask` maskieren, bevor Sie die Konstante `vbObjectError` hinzuzählen:

```
Err.Raise (Err.Number And lngErrorMask) + vbObjectError
```


10.5 Das Rechnungsbeispiel

In Kapitel 10.4 haben wir eine Klasse geschrieben, die im Prinzip über alles verfügt, was eine Klasse ausmacht. Dennoch möchte ich Ihnen noch ein Beispiel bieten. In diesem von der Aufgabenstellung her betrachtet recht einfach gehaltenen Beispiel bekommen wir es mit Klassenhierarchien zu tun und werden etwas tiefer in die Aspekte der Klassenarchitektur einsteigen.

10.5.1 Die Aufgabenstellung

Die Anwendung soll das Erstellen von Rechnungen ermöglichen, hierbei auf eine Artikelreferenz zugreifen und eine kundenabhängige Rabattberechnung durchführen, in der auch das Kaufverhalten des jeweiligen Kunden innerhalb einer definierbaren Periode untersucht wird.

10.5.2 Die Tabellen

Die folgenden fünf Abbildungen zeigen die verwendeten Tabellen, die Sie in der Datei Rechnung.xls auf der Buch-CD finden.

Tabelle Rechnung

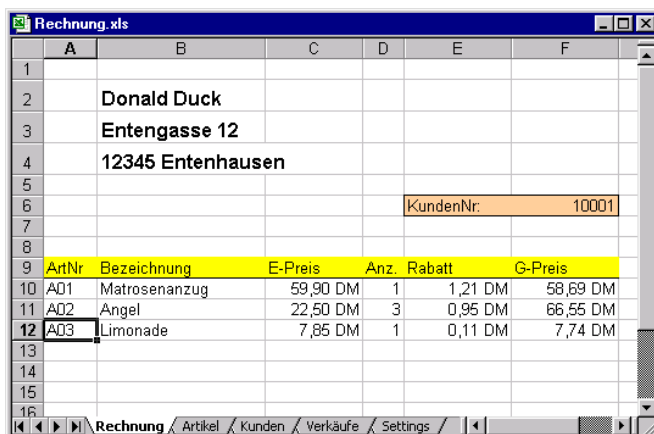
In der Zelle F6 stellen Sie über eine eindeutige Kundennummer eine Beziehung zu den in der Tabelle Kunden angelegten Kunden her. Die Anschrift in den Zellen B2 bis B4 wird nach Eingabe der Kundennummer per Programm eingetragen.

In den Zellen unterhalb von A9 haben Sie nun die Möglichkeit, Artikelnummern anzugeben. Nach der Eingabe (Change-Ereignis) werden Bezeichnung und Einzelpreis aus den Artikeldaten übernommen und als Anzahl standardmäßig eine 1 eingetragen. In Spalte E wird der Rabatt eingetragen und in der Nachbarzelle aus Einzelpreis, Anzahl und Rabatt der Gesamtpreis errechnet.

Als Vorgriff auf den Code sehen Sie hier die fünf Programmzeilen, die in die Zellen der Spalten B bis F diese Werte eintragen:

```
Target.Cells(1, 2).Value = cInv.ArticleName  
Target.Cells(1, 3).Value = cInv.Price  
Target.Cells(1, 4).Value = cInv.Number  
Target.Cells(1, 5).Value = cInv.Discount  
Target.Cells(1, 6).Value = cInv.TotalPrice
```

Konventionell codiert, hätten Sie hier vielleicht 100, vielleicht 200 Zeilen vor Augen. Wir aber bringen es auf fünf Zeilen, in denen jeweils eine Eigenschaft dieser Klasse `cInv` erscheint.

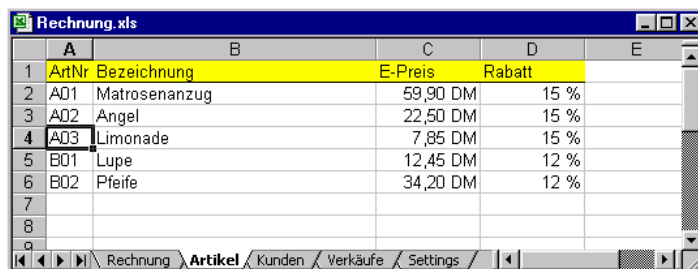


	A	B	C	D	E	F
1						
2		Donald Duck				
3		Entengasse 12				
4		12345 Entenhausen				
5						
6				KundenNr:	10001	
7						
8						
9	ArtNr	Bezeichnung	E-Preis	Anz.	Rabatt	G-Preis
10	A01	Matrosenanzug	59,90 DM	1	1,21 DM	58,69 DM
11	A02	Angel	22,50 DM	3	0,95 DM	66,55 DM
12	A03	Limonade	7,85 DM	1	0,11 DM	7,74 DM
13						
14						
15						
16						

Abbildung 10.12: Rechnung.xls, Tabelle Rechnung

Tabelle Artikel

Diese Tabelle enthält alle definierten Artikel mit Artikelnummer, Bezeichnung, Einzelpreis und dem maximalen Rabatt des jeweiligen Artikels.



	A	B	C	D	E
1	ArtNr	Bezeichnung	E-Preis	Rabatt	
2	A01	Matrosenanzug	59,90 DM	15 %	
3	A02	Angel	22,50 DM	15 %	
4	A03	Limonade	7,85 DM	15 %	
5	B01	Lupe	12,45 DM	12 %	
6	B02	Pfeife	34,20 DM	12 %	
7					
8					
9					

Abbildung 10.13: Rechnung.xls, Tabelle Artikel

Tabelle Kunden

Die Tabelle Kunden beinhaltet die Kundennummer sowie die Anschrift der Kunden.



	A	B	C	D	E
1	KundenNr	Name	Straße	PLZ	Ort
2	10001	Donald Duck	Entengasse 12	12345	Entenhausen
3	10002	Mickey Mouse	Mäusebergweg 6	12346	Entenhausen
4					
5					
6					

Abbildung 10.14: Rechnung.xls, Tabelle Kunden

Tabelle Verkäufe

In der Tabelle Verkäufe befindet sich eine Verkaufshistorie, die zur Berechnung des aktuellen Rabatts herangezogen wird.

	A	B	C	D	E	F
1	KundenNr	Datum	Nettoumsatz			
2	10001	2000-01-02	100,00 DM			
3	10001	2000-01-07	80,00 DM			
4	10001	2000-01-12	50,00 DM			
5	10001	2000-01-17	70,00 DM			
6	10001	2000-01-22	60,00 DM			
7	10002	2000-01-23	80,00 DM			
8	10001	2000-01-27	40,00 DM			
9	10001	2000-02-01	70,00 DM			
10	10001	2000-02-06	90,00 DM			
11	10001	2000-02-11	40,00 DM			
12	10001	2000-02-16	60,00 DM			
13						
14						

Abbildung 10.15: Rechnung.xls, Tabelle Verkäufe

Tabelle Settings

In Settings stehen neben den Umsatzsteuersätzen die Parameter zur Ermittlung des Rabattkoeffizienten, einem Wert zwischen null und eins, mit dem der maximale Rabatt des jeweiligen Artikels multipliziert wird. Die Mechanik werden wir uns noch detailliert ansehen, aber bei fünf Käufen (Zelle C6) in 14 Tagen (C4) zu mindestens 100 DM (C5) hat der Kandidat 100 Punkte, also eine Koeffizienten von 1.

	A	B	C	D	E
1	Name	Bedeutung	Wert		
2	FullVATRate	voller USt-Satz	7		
3	ReducedVATRate	halber USt-Satz	16		
4	SalesInterval	Kaufintervall	14 Tage		
5	SalesTreshold	Umsatzschwelle	100,00 DM		
6	Intervals	Anzahl der Prüfintervalle	5		
7					
8					
9					
10					
11					
12					
13					
14					

Abbildung 10.16: Rechnung.xls, Tabelle Settings

10.5.3 Klassenarchitektur

Das Entscheidende bei der Realisierung einer solchen Applikation ist der Entwurf der Klassen. Ich kann mir nicht vorstellen, dass es ein allgemein gültiges Rezept dafür gibt. Aber ein paar Hinweise werde ich in Kapitel 10.4.6 zum Besten geben. Der Rest ist Versuch und Irrtum. Sie müssen Ihre Fehler selbst machen und daraus lernen. Zu Anfang werden Sie sich reichlich blaue Flecken einhandeln, doch jede Klasse wird besser als die vorangegangene.

Werfen wir nun einen Blick auf die Klassenstruktur, die in Abbildung 10.17 wiedergegeben ist.

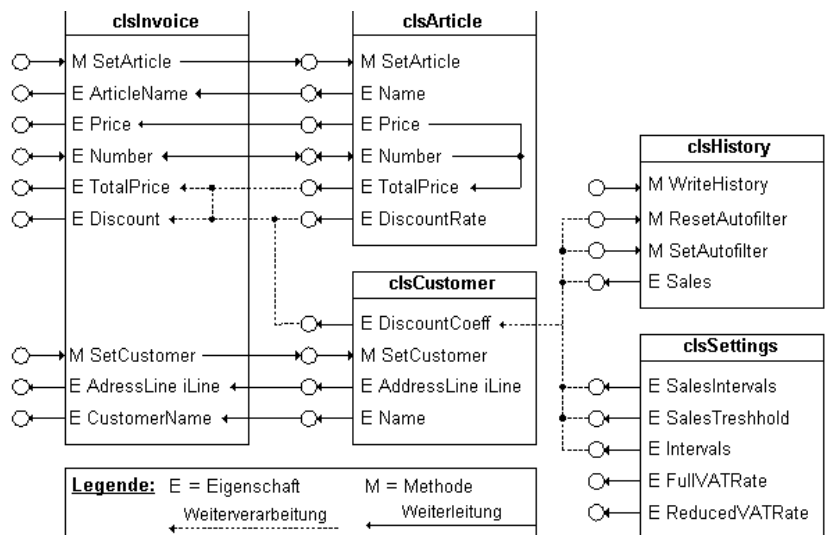


Abbildung 10.17: Rechnung.xls, Klassenstruktur

Die Klasse `clsInvoice` ist die einzige, die nach außen in Erscheinung tritt. Wie wir an den paar vorweggenommenen Codezeilen in Kapitel 10.5.2 sahen, stellt sie die zentrale Schnittstelle zum Code der Tabelle Rechnung dar. In den Ereignisroutinen dieser Tabelle werden die Eigenschaften dieser Klasse direkt in die Zellen eingetragen, ohne dass dort zu erkennen ist, wie diese Werte ermittelt werden und woher die dazu nötigen Daten stammen. `clsInvoice` kapselt also die Programmlogik und den Datenzugriff.

`clsArticle` bedient die Tabelle Artikel und die darin enthaltenen Daten. Um den Gesamtpreis `TotalPrice` berechnen zu können, nimmt sie die aktuelle Anzahl des gewünschten Artikels in der Eigenschaft `Number` entgegen. Dieser `TotalPrice` beinhaltet noch keine Rabatte.

Die Klasse `clsCustomer` hält über die `Name`-Eigenschaft Namen und Vornamen des Kunden bereit. Die `AddressLine`-Eigenschaft gibt die beiden Anschriftzeilen zurück, die in der Tabelle Rechnung eingetragen werden. Man hätte auch Straße, PLZ und Ort als separate Eigenschaften herausführen können, doch wozu soll sich das Ausgabemodul mit der Zusammensetzung von PLZ, einem Leerzeichen und dem Ort auseinander setzen? Des weiteren verfügt die Klasse `clsCustomer` über die Eigenschaft `DiscountCoeff`, die den kundenspezifischen Rabattkoeffizienten bereithält. Hierzu bedient sie sich über die Klasse `clsHistory` der Verkaufshistorie in Tabelle Verkäufe sowie dreier Eigenschaften der Klasse `clsSettings`.

Durch `clsHistory` werden die Daten der Verkaufshistorie nach außen gekapselt. Sie verfügt über eine Methode namens `SetFilter`, mit der die Daten gefiltert werden können, sowie über `ResetFilter` zum Aufheben des Filters. In der Eigenschaft `Sales` stecken die Umsätze, die in dem Filterzeitraum angefallen sind. Die Methode `WriteHistory`, die hier nur angedeutet ist, dient dazu, einen Kauf in die Historie zu übertragen.

Neben den drei Eigenschaften zur Berechnung des Koeffizienten stellt die Klasse `clsSettings` noch den vollen und den reduzierten Umsatzsteuersatz zur Verfügung.

10.5.4 Klassencodierung

Beginnen wir mit den Basisklassen, die an den Daten kleben.

Die Klasse `clsHistory`

Schauen wir uns zuerst den Deklarationsteil an. Neben einem `Range`-Objekt wird hier eine Enumeration gebildet, um einen sprechenden Bezug zwischen Spaltenzeiger und Spalteninhalt herzustellen:

```
Private rngHistory As Range 'Range-Objekt in Tabelle
'Enumeration für Spaltenzeiger
Public Enum enumHistoryCol
    enumHistoryColCustNr = 1
    enumHistoryColDate = 2
    enumHistoryColSales = 3
End Enum
```

In `Class_Initialize` wird das `Range`-Objekt zugewiesen:

```
Private Sub Class_Initialize()
    Set rngHistory = shtHistory.Range("A1")
End Sub
```

Die Methode `ResetFilter` hebt einen eventuell vorhandenen Filter auf:

```
Public Sub ResetFilter()  
    ' Filter aufheben  
    rngHistory.AutoFilter  
End Sub
```

`SetFilter` ist universell gehalten und bildet die Parameter der `AutoFilter`-Methode nach außen ab. Im Spaltenzeiger kommt die obige Enumeration zum Einsatz. Außerdem werden das erste Filterkriterium und optional der Operator und das zweite Kriterium entgegen genommen:

```
Public Sub SetFilter(ByVal iHistoryCol As enumHistoryCol, _  
    ByVal varCriterion1 As Variant, _  
    Optional ByVal lngOperator As xlAutoFilterOperator, _  
    Optional ByVal varCriterion2 As Variant)  
    ' angegebene Spalte filtern  
    'Argumente:  
    ' iHistoryCol:    Spaltenzeiger  
    ' varCriterion1: 1. Kriterium  
    ' lngOperator:   Operator zwischen beiden Kriterien  
    ' varCriterion1: 2. Kriterium  
    On Error GoTo errHandler  
    If IsMissing(varCriterion2) Then  
        rngHistory.AutoFilter iHistoryCol, varCriterion1  
    Else  
        rngHistory.AutoFilter iHistoryCol, varCriterion1, _  
                               lngOperator, varCriterion2  
    End If  
    Exit Sub  
  
errHandler:  
    Err.Raise Err.Number + vbObjectError, _  
        "clsHistory:SetFilter"  
End Sub
```

Zur `IsMissing`-Überprüfung wurde `varCriterion2` verwendet, da diese Variable als `Variant` im Gegensatz zu `lngOperator` den Wert `Missing` abbilden kann. `lngOperator` kann als `Long`-Datentyp nur Zahlen enthalten.

Bleibt noch die Eigenschaft `Sales`, in der unser gefilterter Zellenbereich in ein `Range`-Objekt eingetütet wird, damit die `Sum`-Methode des `Application`-Objekts daraus die Summe ermitteln kann:

```

Public Property Get Sales() As Double
    ' gefilterter Umsatz
    Dim rngFilter As Range 'Range-Objekt für Filterrückgabe
    On Error GoTo errHandler
    Set rngFilter = shtHistory.Columns(3). _
        SpecialCells(xlCellTypeVisible)
    Sales = Application.Sum(rngFilter)
    Exit Sub

errHandler:
    Err.Raise Err.Number + vbObjectError,
        "clsHistory:SetFilter"
End Property

```

Die Klasse clsSettings

Diese Klasse verwaltet die in der Tabelle Settings hinterlegten Parameter, die auch als private Variablen hinterlegt sind:

```

Private sngFullVATRate As Single 'voller MwSt-Satz
Private sngReducedVATRate As Single 'halber MwSt-Satz
Private sngSalesTreshold As Single 'Umsatzschwellwert
Private lngSalesInterval As Long 'Verkaufsintervall
Private lngIntervals As Long 'Intervalle
Private Const iColValue As Long = 3 'Zeiger auf Wertspalte

```

Der Initialize-Methode kommt die Aufgabe zu, die Eigenschaften zu ermitteln, da diese regelmäßig angefordert werden und die Wertzuweisung zu viel Zeit in Anspruch nimmt, um bei jeder Anfrage neu ermittelt zu werden. Es wurde absichtlich eine separate Methode geschaffen, da das Class_Initialize Ereignis unseren Fehler vollständig verstümmeln würde (siehe Kapitel 10.4.1). Den Anwender mit einem Automatisierungsfehler zu konfrontieren, weil ein Wert in der Tabelle Settings nicht gefunden werden konnte, wäre keine Ruhmestat.

```

Public Sub Initialize()
    ' Werte des Variablen ermitteln
    On Error GoTo errHandler
    sngFullVATRate = GetValue("FullVATRate")
    sngReducedVATRate = GetValue("ReducedVATRate")
    sngSalesTreshold = GetValue("SalesTreshold")
    lngSalesInterval = GetValue("SalesInterval")
    lngIntervals = GetValue("Intervals")
    Exit Sub

```

```
errHandler:
    If InStr(1, Err.Source, ":") = 0 Then _
        Err.Source = "clsSettings:Initialize"
    Err.Raise (Err.Number And lngErrorMask) + vbObjectError
End Sub
```

Um dem Rechnung zu tragen, wird die Klasse auch nicht mit dem Schlüsselwort **New** deklariert, sondern in der Ereignisprozedur **Workbook_Open** zugewiesen. Dort wird auch die Methode **Initialize** aufgerufen:

```
Private Sub Workbook_Open()
    ' Settings-Klasse instatiieren
    On Error GoTo errHandler
    Set cSettings = New clsSettings
    cSettings.Initialize
    Exit Sub

errHandler:
    If InStr(1, Err.Source, ":") = 0 Then Err.Source = _
                                                "Workbook_Open"
    cError.ErrorMessage
End Sub
```

Wenn der Anwender nun Werte der Tabelle **Settings** ändert, so müssen wir dafür sorgen, dass diese **Initialize**-Methode erneut aufgerufen wird. Hierfür bietet sich die **Worksheet_Deactivate**-Methode der Tabelle **Settings** an:

```
Private Sub Worksheet_Deactivate()
    On Error GoTo errHandler
    cSettings.Initialize
    Exit Sub

errHandler:
    If InStr(1, Err.Source, ":") = 0 Then _
        Err.Source = "shtSettings:Worksheet_Deactivate"
    cError.ErrorMessage
End Sub
```

Zurück zu unserer Klasse **clsSettings**. Die eigentliche Arbeit wird dort von der privaten Methode **GetValue** erledigt, die den Namen des gewünschten Wertes erhält. Dieser Wert wird in Spalte A der Tabelle **Settings** gesucht:


```

Private Function GetValue(strName As String) As Variant
    ' Wert des übergebenen Namens ermitteln
    Dim rngFind As Range      'Range-Objekt für Find-Methode
    On Error GoTo errHandler
    Set rngFind = shtSettings.Columns(1).Find( _
        what:=strName, LookIn:=xlValues, lookat:=xlWhole)
    If rngFind Is Nothing Then
        Err.Raise 10009, , "Der Wert " & strName & _
            " existiert nicht in der Settings-Tabelle."
    End If
    GetValue = rngFind.Cells(1, iColValue).Value
    Exit Sub

errHandler:
    Err.Raise Err.Number + vbObjectError, _
        "clsSettings:GetValue"
End Function

```

Nun fehlen noch die fünf Ereignisroutinen, von denen beispielhaft Sales-Interval dargestellt wird:

```

Public Property Get SalesInterval() As Single
    SalesInterval = lngSalesInterval
End Property

```

Die Klasse clsCustomer

Nachdem Sie sich mit den beiden von der clsCustomer verwendeten Datenklassen clsHistory und clsSettings vertraut gemacht haben, können wir uns diese clsCustomer aus der Nähe ansehen.

Nachdem ein Kunde ausgewählt wurde, werden die Eigenschaften dieser Klasse öfters verwendet, um Werte dieses Kunden zu erfragen. Deswegen macht es Sinn, sich zumindest den Zeilenzeiger in einer privaten Variablen zu merken. Da eine solche Tabelle in Bezug auf die Anzahl und Anordnung der Spalten mitunter recht dynamische Züge annehmen kann, sind Spaltenzeiger eigentlich auch sinnvoll:

```

Private cHistory As New clsHistory 'Instanz auf clsHistory
Private iRow As Long               'Zeilenzeiger
Private lngCustNr As Long          'Kundennummer
'Spaltenzeiger in Tabelle Kunden
Private Const iColCustNr As Long = 1
Private Const iColName As Long = 2
Private Const iColStreet As Long = 3

```

```
Private Const iColPLZ As Long = 4
Private Const iColTown As Long = 5
```

Klassen dieses Zuschnitts statte ich grundsätzlich mit einer Methode aus, um den Zeilenzeiger zu ermitteln. Dies erledigt die Find-Methode für uns. Wird der Kunde nicht gefunden, erzeugt die Methode einen Fehler mit der nicht auffindbaren Kundennummer im Fehlertext:

```
Public Sub SetCustomer(ByVal lngNewCustNr As Variant)
    ' Zeilenzeiger ermitteln und Fehler erzeugen,
    ' wenn Kundennummer nicht existiert
    Dim rngFind As Range      'Range-Objekt für Find-Methode
    On Error GoTo errHandler
    Set rngFind = shtCust.Columns(iColCustNr).Find( _
        what:=CStr(lngNewCustNr), _
        LookIn:=xlValues, lookat:=xlWhole)
    If rngFind Is Nothing Then
        iRow = 0
        Err.Raise 10009, , "Die Kundennummer " & _
            lngNewCustNr & " existiert nicht."
    End If
    lngCustNr = lngNewCustNr
    iRow = rngFind.Row
    Exit Sub

errHandler:
    Err.Raise Err.Number + vbObjectError, _
        "clsCustomer:SetCustomer"

End Sub
```

Die Eigenschaft AddressLine nimmt mit iLine einen Zeiger entgegen, der die zusammenzustellende Adresszeile angibt:

```
Public Property Get AddressLine(ByVal iLine As Long) _
    As String
    ' Adresszeile zurückgeben
    On Error GoTo errHandler
    If iRow = 0 Then Err.Raise 10091, _
        "Es wurde kein Kunde ausgewählt."
    Select Case iLine
    Case 1
        AddressLine = shtCust.Cells(iRow, iColStreet).Value
    Case 2
        AddressLine = shtCust.Cells(iRow, iColPLZ).Value & _
            " " & shtCust.Cells(iRow, iColTown).Value
```

```

Case Else
    Err.Raise 10005, , "Dies ist keine gültige Zeile."
End Select
Exit Property

```

```

errHandler:
    Err.Raise Err.Number + vbObjectError, _
        "clsCustomer:Property Get AddressLine"
End Property

```

Die Eigenschaft `DiscountCoeff` multipliziert nun die Eigenschaften `Intervals` und `SalesInterval` der `Settings`-Klasse und ermittelt daraus den Betrachtungszeitraum für den Vergleich von Soll- und Ist-Umsatz. Durch die Multiplikation von `Intervals` und `SalesTreshold` ergibt sich der Sollumsatz des Betrachtungszeitraums, zu dem der Ist-Umsatz ins Verhältnis gesetzt wird. Da der Ist-Umsatz den Soll-Umsatz übersteigen und der Quotient somit größer als 1 werden kann, wird über `Application.Min` der kleinere der beiden Werte Quotient und 1 ermittelt. Das grenzt den Koeffizienten nach oben bei 1 ab:

```

Public Property Get DiscountCoeff() As Single
    ' Kundenrabatt zurückgeben
    Dim rngFilter As Range           'gefilterter Range
    Dim lngDays As Long              'Anzahl der Tage
    Dim sngTotalSales As Single      'Sollumsatz
    Dim sngCustSales As Single       'Istumsatz im
    On Error GoTo errHandler
    lngDays = cSettings.Intervals * cSettings.SalesInterval
    sngTotalSales = cSettings.Intervals * _
        cSettings.SalesTreshold
    cHistory.ResetFilter
    cHistory.SetFilter enumHistoryColCustNr, lngCustNr
    cHistory.SetFilter enumHistoryColDate, _
        ">=" & CDb1(Date - lngDays), xlAnd, _
        "<=" & CDb1(Date)
    sngCustSales = cHistory.Sales
    DiscountCoeff = Application.Min( _
        sngCustSales / sngTotalSales, 1)
    cHistory.ResetFilter
    Exit Property
errHandler:
    If InStr(1, Err.Source, ":") = 0 Then Err.Source = _
        "clsCustomer:Property Get DiscountCoeff"

```

```
Err.Raise (Err.Number And lngErrorMask) + vbObjectError  
End Property
```

Bleibt noch die Eigenschaft `Name`, die Name und Vorname des ausgewählten Kunden zurückgibt:

```
Public Property Get Name() As String  
    ' Kundenname zurückgeben  
    On Error GoTo errHandler  
    If iRow = 0 Then Err.Raise 10091, _  
        "Es wurde kein Kunde ausgewählt."  
    Name = shtCust.Cells(iRow, iColName).Value  
    Exit Property  
  
errHandler:  
    Err.Raise Err.Number + vbObjectError, _  
        "clsCustomer:Property Get Name"  
End Property
```

Die Klasse `clsInvoice`

Hauptaufgabe dieser Klasse ist das Durchreichen der Methodenaufrufe an die betreffende Klasse und die Rückgabe der Ergebnisse, was jeweils in einer Zeile passiert. Die beiden Klassen `clsArticle` und `clsCustomer` werden in `clsInvoice` als `cArt` und `cCust` privat deklariert und im `Initialize`-Ereignis instantiiert.

```
Private cArt As clsArticle  
Private cCust As clsCustomer  
  
Private Sub Class_Initialize()  
    Set cArt = New clsArticle  
    Set cCust = New clsCustomer  
End Sub
```

Die Klasse verfügt über zwei Methoden, mit denen `clsArticle` und `clsCustomer` positioniert werden:

```
Public Sub SetArticle(ByVal strArtNr As String)  
    ' Zeilenzeiger ermitteln und Fehler erzeugen, wenn  
    ' Artikelnummer nicht existiert  
    On Error GoTo errHandler  
    cArt.SetArticle strArtNr  
    Exit Sub  
  
errHandler:  
    If InStr(1, Err.Source, ":") = 0 Then Err.Source = _
```

```

        "clsInvoice:SetArticle"
    Err.Raise (Err.Number And lngErrorMask) + vbObjectError
End Sub

```

Von der zweiten Routine wird hier die Signatur und der eigentliche Klassenaufwurf gezeigt:

```

Public Sub SetCustomer(ByVal lngCustNr As Long)
    ...
    cCust.SetCustomer lngCustNr

```

Da alle Prozeduren im Prinzip gleich aufgebaut sind, reicht es aus, einen Aufruf der beiden verwendeten Klassen `clsArticle` und `clsCustomer` beispielhaft zu zeigen. Da diese Prozeduren Fehler möglicherweise durchreichen, müssen wir die Fehlernummer mit `lngErrorMask` maskieren, bevor `vbObjectError` hinzugezählt werden kann, um einen Überlauf der Number-Eigenschaft des `Err`-Objekts zu vermeiden:

```

Public Property Get Discount() As Single
    '    maximalen Artikelrabatt zurückgeben
    On Error GoTo errHandler
    Discount = cArt.TotalPrice * cArt.DiscountRate / 100 * _
        cCust.DiscountCoeff
    Exit Property
errHandler:
    If InStr(1, Err.Source, ":") = 0 Then Err.Source = _
        "clsInvoice:Property Get Discount"
    Err.Raise (Err.Number And lngErrorMask) + vbObjectError
End Property

```

10.5.5 Codierung des Moduls `shtInvoice`

In diesem Modul können wir die Früchte unserer Bemühungen nun endlich ernten und mit einfachen Aufrufen eine Menge Arbeit erledigen. Auf Modulebene wird die Klasseninstanz der `clsInvoice` deklariert. Außerdem benötigen wir eine Variable, mit der ein wiederholter Aufruf von Ereignisprozeduren unterdrückt werden kann. Einer alten Gewohnheit folgend nenne ich sie schlicht `bForgetIt`.

```

Private cInv As New clsInvoice
Private bForgetIt As Boolean

```

Folgende Ereignisprozeduren werden verwendet:

- ▼ `Worksheet_Activate`, in der die aktuelle Kundennummer an die Klasse weitergereicht wird,

- ▼ **Worksheet_Change**, die auf Eingaben in der Tabelle reagiert und
- ▼ **Worksheet_SelectionChange**, die beim Wechsel auf eine Artikelzeile die Positionierung der Artikelklasse einleitet.

Beginnen wir mit **Worksheet_Activate**, die eigentlich recht einfach ist:

```
Private Sub Worksheet_Activate()  
    ' Kunden einstellen  
    On Error GoTo errHandler  
    If ThisWorkbook.Names("Customer").RefersToRange.Value _  
        <> "" Then  
        cInv.SetCustomer _
```

```
        ThisWorkbook.Names("Customer").RefersToRange.Value  
    End If  
    Exit Sub
```

```
errHandler:  
    If InStr(1, Err.Source, ":") = 0 Then Err.Source = _  
        "shtInvoice:Worksheet_Change"  
    cError.ErrorMessage  
End Sub
```

Worksheet_Change ist die zentrale Routine der ganzen Anwendung.

```
Private Sub Worksheet_Change(ByVal Target As Excel.Range)  
    ' Trägt die Kunden- oder Artikeldaten ein  
    On Error GoTo errHandler  
    'Prozedur verlassen, wenn mehr als 1 Zelle markiert  
    If Target.Cells.Count > 1 Or bForgetIt Then Exit Sub  
    'bForgetIt auf True setzen, damit alle nachfolgenden  
    'Zelländerungen die Ereignisroutine abbrechen  
    bForgetIt = True  
    'Prüfen, ob veränderte Zelle die KundenNr-Zelle ist  
    If Target.Address = ThisWorkbook.Names("Customer"). _  
        RefersToRange.Address Then  
        cInv.SetCustomer Target.Value  
        shtInvoice.Cells(2, 2).Value = cInv.CustomerName  
        shtInvoice.Cells(3, 2).Value = cInv.AddressLine(1)  
        shtInvoice.Cells(4, 2).Value = cInv.AddressLine(2)  
    End If  
    'Prüfen, ob veränderte Zelle eine ArtikelNr-Zelle ist  
    'und gerade geleert wurde und, wenn ja, Zeile leeren  
    If Target.Column = 1 And Target.Row > 9 And _
```

```

                Target.Value = "" Then
    Target.EntireRow.ClearContents
ElseIf Target.Row > 9 Then
    If Target.Column = 1 Then 'Artikelnummernspalte
        cInv.SetArticle Target.Value
        cInv.Number = 1
        Target.Cells(1, 2).Value = cInv.ArticleName
        Target.Cells(1, 3).Value = cInv.Price
        Target.Cells(1, 4).Value = cInv.Number
        Target.Cells(1, 5).Value = cInv.Discount
        Target.Cells(1, 6).Value = cInv.TotalPrice
    ElseIf Target.Column = 4 Then 'Anzahl-Spalte
        cInv.Number = Target.Value
        Target.Cells(1, 2).Value = cInv.Discount
        Target.Cells(1, 3).Value = cInv.TotalPrice
    End If
End If
bForgetIt = False
Exit Sub

errHandler:
    bForgetIt = False
    If InStr(1, Err.Source, ":") = 0 Then Err.Source = _
        "shtInvoice:Worksheet_Change"
    cError.ErrorMessage
End Sub

```

Zum Schluss werfen wir noch einen Blick auf die Ereignisroutine, die uns eine Änderung der Zellauswahl signalisiert:

```

Private Sub Worksheet_SelectionChange( _
    ByVal Target As Range)
    ' Artikelklasse positionieren
    If Target.Cells.Count > 1 Then Exit Sub
    If Target.Column <= 6 And Target.Row > 9 And _
        Target.Value <> "" Then
        cInv.SetArticle shtInvoice.Cells(Target.Row, 1).Value
    End If
Exit Sub

errHandler:
    If InStr(1, Err.Source, ":") = 0 Then Err.Source = _
        "shtInvoice:Worksheet_Change"
    cError.ErrorMessage
End Sub

```

10.5.6 Zusammenfassung

Das war ein kleines Beispiel, das aber auf einen Umfang von recht ansehnlichen 600 Codezeilen angewachsen ist. Zieht man Kommentare und ErrorHandler ab, so bleibt noch etwa die Hälfte übrig, was aber ein durchaus repräsentatives Verhältnis darstellt.

Wenn Sie sich fragen, ob ich das genau so realisiert hätte, muss ich mit einem klaren Ja antworten. Gerade VBA-Anwendungen stehen in dem Ruf, schnell entwickelt zu sein und vor allem aber ebenso schnell wieder geändert werden zu können. VBA-Anwendungen neigen aufgrund dessen zu unkontrollierbarem Wachstum und den daraus entstehenden Schwierigkeiten, die nicht selten zu einem Re-Design führen, können Sie nur dadurch begegnen, dass Sie modularisieren und Funktionalität und Datenzugriff streng kapseln.

Welche Designregeln lassen sich hieraus ableiten?



So, wie man verschiedenartige Daten in unterschiedlichen Tabellen anordnet, sollte man auch diese Daten durch separate Klassenmodule kapseln.

Auch wenn Sie der Überzeugung sind, dass der Zugriff nur an einer Stelle erfolgt, sollten Sie dieser Regel folgen.

Die `clsHistory` wäre durchaus verzichtbar gewesen, denn die Filterung der Daten hätte man in wenigen Zeilen in die Eigenschaft `DiscountCoeff` der Klasse `clsCustomer` integrieren können. Doch wie werden die Verkäufe darin historisiert? Werden diese Daten zum Beispiel auch anderen Anwendungen zur Verfügung gestellt, wie etwa der Forderungsverwaltung? Wann werden ältere Daten gelöscht oder archiviert? Alles das spricht für eine separate Klasse als Schnittstelle.



Die so genannte Programmlogik, also all das, was zwischen den eigentlichen Daten und der Ausgabe liegt, sollte in separaten Strukturen abgebildet werden, also in Klassenmodulen.

Stellen Sie sich spaßeshalber einmal vor, Sie hätten dieses Programm ohne Modularisierung komplett in dem Modul der Tabelle Rechnung codiert. Nun kommt der Kunde mit der sensationellen Idee daher, man könne doch auf die Artikel- und Kundendaten der hausinternen Datenbank zugreifen, die Verkaufshistorie aber in der Excel-Datei belassen. Sie werden fast keine Zeile unverändert lassen und im Prinzip das Programm von innen heraus neu schreiben, falls Sie sich nicht ohnehin gleich zum Neuschreiben entschließen. Haben Sie die Applikation hingegen nach dem vorgestellten Muster geschrieben, ändern Sie lediglich die betreffenden Datenklassen. Sie können sich darauf verlassen, dass der Rest nach diesem Umbau problemlos funktioniert, da sich die Änderung nur innerhalb einer oder mehrerer streng isolierter Klassen abspielt.

Eine ganze Reihe von Visual-Basic-Programmen, die in unserem Hause geschrieben worden sind, nahmen ihren Anfang als *einfache* Excel-VBA-Anwendungen. Dann kamen Datenbanken als schnelles und effektives Datenmedium hinzu, weil die Excel-Dateien anwuchsen und ein kaum mehr vertretbares Zeitverhalten zu Tage legten. Wenn dann der Code geschickt organisiert ist, können Sie fast alles in Visual Basic importieren. Und was spricht in diesem Falle dagegen, die alten Excel-Tabellen, allerdings völlig entrümpelt und in viele einzelne Dateien aufgeteilt, weiterhin als Ausgabemedium zu nutzen? In diesem Falle können Sie sogar Ihren Ausgabecode weiter verwenden, der dann allerdings nicht mehr von Ereignisroutinen des `Worksheet`- oder `Workbook`-Objekts angestoßen wird, sondern durch Ereignisse aus dem VB-Container seinen Impuls erhält.

10.6 DLLs – wann und wozu

Sie sind auf dem direkten Weg in des Teufels Küche, wenn Sie eine umfangreiche Funktionalität in einer Arbeitsmappe zusammenfassen und von anderen Arbeitsmappen darauf zugreifen wollen. Solche Codekomponenten gehören in eine COM-DLL!

10.6.1 Zur Abschreckung ein Beispiel

In einem großen deutschen Unternehmen wurde von einer großen internationalen Unternehmensberatung ein Konzerncontrolling eingerichtet. Der Hersteller der im Unternehmen verwendeten EDV-Lösung stellte ein Tool in Form einer Excel-Arbeitsmappe zur Verfügung, das in der Lage war, sich an der EDV-Lösung anzumelden und dort Daten abzu ziehen und in Excel-Arbeitsmappen abzulegen. Nennen wir diese Datei `X.xls`. Um jedoch von einer anderen Excel-Arbeitsmappe auf diese `X.xls` zugreifen zu können, musste in dieser Arbeitsmappe ein Verweis auf die `X.xls` eingerichtet werden. Danach stehen die `Public Subs` und `Public Functions` wie eigene Prozeduren zur Verfügung. Tolle Sache, nicht wahr.

Dann bat mich diese Unternehmensberatung, die einzelnen Bedienschr itte zu automatisieren, weil es immer wieder zu Fehl eingaben kam. Und da beging ich einen Fehler. Zwar versuchte ich mehrfach während der Entwicklung aus der Sache herauszukommen, hatte aber keine Chance. Auch mehrere Vorstöße bei dem Hersteller der EDV-Lösung und der `X.xls`, aus diesem unsäglichen Ding eine DLL zu machen, blieben erfolglos. Wo lagen die Probleme?

`X.xls` hatte eine Fehlerbehandlung, die aus `On Error Resume Next` und einer Reihe aussageloser `Message Boxes` bestand. Der Code aller verwendeten Master-Dateien duplizierte sich bei jedem Erzeugen einer neuen Auswertung virenartig. `X.xls` fummelte geradezu gnadenlos jeweils im

ActiveWorkbook herum, das man ihm bereitstellen musste. X.xls suchte dort nach benannten Zellen und forderte zu einer Markierung der Zielspalte auf, falls es diesen Namen nicht finden konnte. X.xls griff dabei auf Code in anderen Arbeitsmappen zurück, die wiederum Routinen in X.xls verwendeten, wozu Verweise in den jeweiligen Arbeitsmappen eingerichtet waren. So oft bekam ich den Dr. Watson noch nie zu Gesicht.

Meine Arbeitsmappe benötigte jedenfalls einen Verweis zu wechselnden Master-Dateien, die so eingerichtet wurden:

```
ThisWorkbook.VBProject.References.AddFromFile  
strMasterFile
```

Dieser MasterFile existiert als wbkMaster. Das Auflösen des Verweises funktioniert dann so:

```
For Each Ref In ThisWorkbook.VBProject.References  
    If Ref.fullpath = strMasterFile Then  
        ThisWorkbook.VBProject.References.Remove Ref  
    Exit For  
End If  
Next  
...  
wbkMaster.SaveAs strDETFile  
wbkMaster.Close savechanges:=False
```

Wie in Abbildung 10.18 zu sehen, blieb gelegentlich ein Verweis mit dem Hinweis NICHT GESPEICHERT im Verweis-Dialog stehen, obwohl der Verweis auf die dazugehörigen wbkMaster längst entfernt und diese geschlossen war. Die Folge waren ständige Abstürze von Excel 97 und gelegentliche bei Excel 2000.

Ein Hinweis in der MSDN offenbarte, dass das Code enthaltende Workbook auf Null (da war wohl Nothing gemeint) gesetzt werden muss, was aber nur von außen funktioniert, nicht aber in ThisWorkbook.

```
Set wbCode = Workbooks.Open "First.XLA"  
wbCode.Close  
set wbCode = Nothing  
Workbooks("Second.XLA").Close
```

Die Lösung war dann, die VBA-Anwendung durch Schließen aller Dialoge herunter zu fahren, wodurch die Objektreferenz des eigenen Workbooks zumindest im Code offensichtlich zu Nothing wurde.

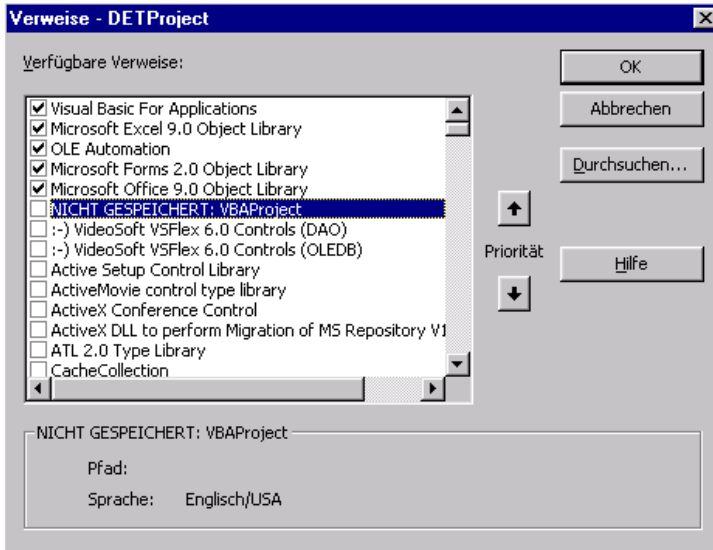


Abbildung 10.18: Nicht gespeicherter Verweis

Wenn Sie anderen Excel-Arbeitsmappen Code zur Verfügung stellen wollen, dann tun Sie das bitte mit einer COM-DLL und widerstehen Sie der Versuchung Verweise zu einer Arbeitsmappe einzurichten.



Kapitelüberblick

11.1 Allgemeine Aspekte des Applikationsdesigns	535
11.2 Modularisierung und Auslagerung	535
11.3 Variablen	536
11.3.1 Datentypen und Präfixbildung	536
11.3.2 Objektvariablen	537
11.3.3 Namen von Standardvariablen	538
11.4 Kommentierung des Codes	539
11.4.1 Kommentarblock im Prozedurkopf	539
11.4.2 Kommentierung der Variablen und Konstanten	541
11.4.3 Kommentierung im Code	542
11.5 Anordnung von Prozeduren im Codemodul	542
11.6 Konstanten und Enumerationen	544
11.6.1 Spaltenzeiger	544
11.6.2 Enumerationen	545
11.7 Die Benutzerschnittstelle	546
11.7.1 Statusanzeige	547
11.7.2 ComboBoxes	548
11.7.3 ControlTipText	549
11.8 Einfache Parameterablage	549
11.9 Settings.xls	550
11.9.1 Die Klasse clsSettingsSheet	551
11.9.2 Die Klasse clsRegistry	554
11.9.3 Die Klasse clsNLS	558
11.9.4 Schlussbemerkungen zu Settings.xls	558

11.10 Mehrsprachige Applikationen	558
11.10.1 Die Beispielanwendung	560
11.10.2 Die Tabellen	561
11.10.3 Der Code	563
11.11 Tabellendesign	575
11.11.1 Worum geht es ? – Ein Beispiel	575
11.11.2 Normalisierung in der Theorie	576
11.11.3 Normalisierung in der (Excel-) Praxis	580
11.11.4 Skalierbarkeit	581

Viele Aspekte des Applikationsdesigns wurden bereits in den vorangegangenen Kapiteln am Rande behandelt. Dem Thema Fehlerbehandlung wurde hierbei sogar ein eigenes Kapitel gewidmet. Die meisten Aspekte sind jedoch neu.

11.1 Allgemeine Aspekte des Applikationsdesigns

Softwareentwicklung ist zweifellos ein kreativer Vorgang, der aber dennoch ein paar Regeln erfordert, die in folgenden Ansätzen gründen:

- ▼ personenneutrale Projektbearbeitung
- ▼ Wiederverwendbarkeit von Code-Komponenten
- ▼ Qualitätssicherung

Obwohl einige Entwickler dazu neigen, ihre Programme als ihre Babies zu betrachten und Dritten so wenig Ansatzpunkte wie möglich einzuräumen, muss professionelle Softwareentwicklung anderen Ansätzen folgen – nicht nur aus ökonomischen Gründen.

Mit jeder Wiederverwendung von Code-Komponenten steigt auch deren Ausfallsicherheit, wodurch dauerhaft eine solide Qualitätsbasis für neue Projekte geschaffen wird. Und Qualität ist noch vor Termintreue und Kostenaspekten anzusiedeln.

11.2 Modularisierung und Auslagerung

Die Erfahrung bringt es immer wieder zutage, dass die Modularisierung im Code, also die Auslagerung von Funktionalitäten in separate Subs oder Functions, zumeist einmal zu wenig erfolgte. Und eine nachträgliche Modularisierung hat den unangenehmen Nebeneffekt, dass bereits abgeschlossene und somit getestete Programmteile angefasst werden müssen. Da diese Hürde nun mal sehr hoch liegt, ist man geneigt zu kopieren, womit dem Chaos Tür und Tor geöffnet wird.

Wenn Sie bislang noch keinen Gebrauch von wieder verwendbaren Codekomponenten gemacht haben, so wird Ihnen der damit verbundene Aufwand möglicherweise als *entschieden* zu hoch erscheinen. Doch spätestens dann, wenn Sie zum ersten Mal in einem neuen Projekt diese Module importiert haben, und diese funktionieren auf Anhieb – und weshalb sollten sie das nicht tun –, werden Sie den bis dahin investierten Aufwand zu schätzen wissen. Im Übrigen müssen Sie ja nicht bei null anfangen, sondern können die Module der in diesem Kapitel verwendeten Beispiele in Ihre zukünftigen Projekte integrieren.

Das Öffnen von Dateien, Excel-Arbeitsmappen oder Datenbanken mit den damit verbundenen vielfältigen Fehlerquellen sollte in Klassenmodu-

len ausgelagert werden. Ebenso die Fehler- und Ausnahmebehandlung (`clsError`) sowie die Schnittstelle zu Programmparametern `clsSettings`. Zentrale Utilities, die immer wieder verwendet werden, liegen in dem separaten Modul `modUtil`.

Diese Module und Klassenmodule mit universalem Charakter haben ihren Platz in einem separaten Verzeichnis, von wo aus sie in neue Projekte importiert werden können. Im Übrigen gibt es eine Reihe von Produkten, mit der die Verwaltung und die Versionskontrolle solcher Programmkomponenten durchgeführt werden kann. Der umfangreichste Vertreter dieser Gattung dürfte wohl Microsofts Visual SourceSafe sein, ein tolles Werkzeug, welches aber seinen Tribut an Einarbeitungszeit kostet. Visual SourceSafe ist u. a. Bestandteil der Office Developer Edition.

11.3 Variablen

11.3.1 Datentypen und Präfixbildung

Im Grunde können Sie Ihre Variablen nennen, wie Sie wollen. Solange Sie alleine an den Projekten arbeiten und die Garantie übernehmen, für die nächsten zehn Jahre auch die Wartungsarbeiten durchzuführen, wird ja außer Ihnen niemand den Code je zu Gesicht bekommen. Sollte sich das jedoch nicht mit Ihren Zukunftsplänen decken, so müssen Sie einen Weg finden, kryptische Namen zu vermeiden und sich an eine wie auch immer gestaltete Konvention zu halten.

Ungarische Notation

In den meisten Unternehmen, die sich mit Softwareentwicklung beschäftigen, gibt es so etwas wie ein gemeinsames Regelwerk, in dem Präfixe und Standardnamen für immer wieder benötigte Standardaufgaben vereinbart sind. Noch zu DOS-Zeiten entwarf Charles Simonyi, damals Chefarchitekt bei Microsoft, eine Benennungskonvention, die als Ungarische Notation in die Technik Einzug hielt. Der Name dieses heutzutage nur noch *The Hungarian* genannten Werkes ergab sich aus Herkunft von Charles Simonyi und dem fremdländischen Aussehen dieser Konstrukte, die wohl irgendjemand für ungarische Wörter hielt. Aus diesem Basiswerk, mit dem wir VB- und VBA-Entwickler kaum etwas anfangen können, entwickelte sich mit der Zeit ein Sammelsurium an Adaptionen für VB und VBA, die zwar noch einige Gemeinsamkeiten aufweisen, aber keineswegs als Standard bezeichnet werden können. Unter dem Namen *Microsoft Consulting Services Naming Conventions for Visual Basic* (ID Q110264) befindet sich in der MSDN ein Vorschlag zur Namensbildung in Visual Basic (for Applications), an den sich allerdings auch nicht alle Microsoft Autoren zu halten scheinen, wenn man sich so umschaut. Vor einiger Zeit schaute ich dort auch mal nach und stellte fest, dass es zwei Abweichungen gegenüber meiner Konvention gab: Variants werden dort mit Präfix `vnt` aufgeführt

und ComboBoxes mit `cbo`. Doch alternativ sind `var` und `cmb` recht verbreitet.

Wenn in Ihrem Hause Konventionen existieren, dann sind diese natürlich vorzuziehen. Wenn nicht, so wäre dies eine gute Gelegenheit damit zu beginnen.

Die folgende Tabelle enthält die Präfixe der einzelnen Datentypen sowie Bemerkungen zum Einsatz der Datentypen:

Datentyp	Präfix	Bemerkungen
Integer	int	durch Long zu ersetzen
Long	lng	für tabellenzeilenorientierte Operationen zwingend
Single	sng	nur für Steuersätze, Provisionen und ähnliche Werte, die eine Genauigkeit von 8 Stellen definitiv nie erreichen können
Double	dbl	Standarddatentyp für Gleitkommawerte
Currency	cur	nur für Datentypen, bei denen die 16-stellige Genauigkeit des Double nicht ausreicht
String	str	
Date	date	
Boolean	b, bo(o)l	
Variant	vnt, (var)	Einsatz nur dort, wo er unverzichtbar ist, z. B. bei optionaler Argumentübergabe
Decimal	dec	Einsatz nur dort, wo er unverzichtbar ist
Object		verboten, stattdessen müssen die entsprechenden Typen bereits bei der Erzeugung verwendet werden (Early Binding)

Von diesen Regeln zur Präfixbildung gibt es einige Abweichungen, die generell eingehalten werden sollten:

- ▼ Zeiger (Long) werden mit dem Präfix `i` versehen: `iRow`
- ▼ Anzahlen erhalten den Präfix `n`: `nRows`
- ▼ Handles sind in der Regel mit `h` eingeleitet: `hFile` oder `hKey`

11.3.2 Objektvariablen

Auch die Interpretation und somit Lesbarkeit von Objektvariablen wird durch die einheitliche Verwendung von Präfixen erheblich vereinfacht:

Klasse	Präfix	Bemerkungen
CheckBox	chk	
Class Module	cls	
ComboBox	cbo, cmb	
CommandButton	cmd	
Connection	cn	
Database	db	
Image	img	
Label	lbl	Name nur, wenn Zugriff per Code
Listbox	lst	
Module	mod	
OptionButton	opt	
Range	rng	generell auf A1, nur in Ausnahmefällen an anderer Position
Recordset	rs	
Textbox	txt	
UserForm	frm	
Workbook	wbk	
Worksheet	sht, wsh	Einstellung über Eigenschaftsfenster

11.3.3 Namen von Standardvariablen

Es gibt eine Reihe von Variablen, die immer wieder in Programmen vorkommen. Hält man sich bei diesen Variablen an einen Standard, so wird hierdurch das Lesen eines Programmes erleichtert. Zum anderen aber erspart es das Nachdenken über Variablennamen, was mitunter eine große Erleichterung darstellt:

Name	Bemerkungen
bStartingUp	zum vorzeitigen Abbruch von Ereignisprozeduren während der Initialisierung
bForgetIt	Abbruch von Ereignisprozeduren außerhalb der Initialisierung
rsTemp	temporäres Recordset zur Ermittlung lokaler Zwischenwerte
strSQL	enthält alle SQL-Anweisungen
xxxNewValue	Argument in Property-Prozeduren

Name	Bemerkungen
varValues	Variant-Array zur Rückgabe in Methoden
strFile	Dateiname
strPath	Verzeichnisname
strFullName	vollständiger Dateiname

11.4 Kommentierung des Codes

Bei der Kommentierung von Code stellt sich an nahezu jeder Stelle die Frage, muss das nun kommentiert werden oder ist das nicht vielmehr selbst erklärend? Lassen Sie uns die Frage anders formulieren: Ist die zu kommentierende Stelle auch noch in einem Jahr jedem beliebigen Entwickler auf Anhieb verständlich? Denn Sie sollten nicht davon ausgehen, dass nur Sie in Ihren Programmen Änderungen vornehmen werden.

Sie werden sicherlich auch schon beobachtet haben, dass sich Ihr eigener Stil im Zuge der Zeit verändert, in der Regel zur besseren Seite hin. Hierin sind auch grundlegende Veränderungen enthalten, so dass man sich gelegentlich über seine eigenen, wenige Monate alte Algorithmen wundert, höflich formuliert. Allein schon um sich ein »Was habe ich mir denn dabei gedacht?« zu vermeiden, sollte man die eine oder andere Erläuterung in den Code einbauen. Gedanken sind bekanntermaßen flüchtig.

Der Gedanke »Da baue ich mir mal schnell eine Methode.« ist der Anfang manches Designfehlers. In dem Moment, in dem Sie im Kommentarblock Ihres Prozedurkopfes beschreiben, was die Prozedur denn eigentlich tun soll, überlegen Sie in der Regel etwas gründlicher, was sie denn nun tun soll, Ihre Methode.

Noch einen Aspekt möchte ich nicht unerwähnt lassen. Wenn Sie sich bei jeder zur Kommentierung anstehenden Variablen fragen, ob sie kommentiert werden soll oder nicht, so verschwenden Sie Zeit und geistige Kapazitäten. Denn bis Sie sich das überlegt haben, könnte sie bereits kommentiert sein. Hier stellt sich mit der Zeit ein beachtlicher Automatismus ein. Versuchen Sie es.

11.4.1 Kommentarblock im Prozedurkopf

Die folgende Prozedur bedarf keines Kommentars:

```
Private Sub cmdClose_Click()
    Unload Me
End Sub
```

Anders sieht es hingegen bei vielen Prozeduren aus, die in den vorangehenden Kapiteln besprochen wurden.

Beginnen wir mit der Beschreibung der Prozedur selbst. Als Regel ließe sich definieren, dass ein Kommentarblock im Prozedurkopf dann kein Luxus mehr ist, wenn eine der folgenden Voraussetzungen erfüllt ist:

- ▼ Es ist nicht auf den ersten Blick völlig zweifelsfrei, was die Prozedur tut.
- ▼ Sie haben die Prozedur verändert.
- ▼ Argumente werden entgegengenommen.
- ▼ Die Prozedur erzeugt eine Rückgabe.

Die folgenden Zeilen sind ein Beispiel für einen einfachen Kommentarblock:

```
'=====
'1999-05-11, Klaus Prinz
'   Erzeugen der Datenbanktabelle
'=====
```

Wenn Sie die Prozedur später verändern, so sollten Sie einen entsprechenden Kommentar anfügen und die Veränderung dokumentieren. Auch wenn die Prozedur Argumente entgegennimmt, so sollte jede einzelne in einer eigenen Zeile beschrieben werden. Erzeugt die Prozedur einen Rückgabewert, so ist die Bedeutung der Werte eine hilfreiche Information.

Im folgenden Beispiel ist eine Function zu sehen, in der alle Komponenten vertreten sind.

```
Private Function CreateTable(shtAct As Worksheet)As
Boolean
```

```
'=====
'   '1999-05-11, Klaus Prinz
'   '   Erzeugen der Datenbanktabelle
'   '1999-05-18, Klaus Prinz
'   '   Rückgabe ergänzt
'Argumente:
'   shtAct:      aktuelles Tabellenblatt
'   Rückgabe:    True, wenn Tabelle fehlerfrei
'=====
```

Wird in einem größeren Projekt eine Prozedur von verschiedenen Stellen aus aufgerufen, so füge ich zumeist einen Abschnitt in den Prozedurkopf ein, der all diese Stellen auflistet:

```
'Aufgerufen von:
'   frmMain:UserForm_Activate
'   modFile:ReadFile
```

Wird zu einem späteren Zeitpunkt diese Prozedur verändert, so sind alle Programmteile bekannt, die überprüft werden müssen. Damit wird verhindert, dass später unliebsame Überraschungen in betroffenen, aber nicht getesteten Programmteilen auftauchen.

Nur um Missverständnisse zu vermeiden, sei hier noch einmal betont, dass dies keine theoretische Betrachtung ist. In all meinen Projekten gehe ich so vor und habe mir hierbei schon eine Menge Arbeit und auch Ärger erspart, denn erfahrungsgemäß begegnet einem früher oder später fast jedes Programm wieder.

11.4.2 Kommentierung der Variablen und Konstanten

Der folgende Variablenblock ist der bereits oben beschriebenen Prozedur CreateTable entnommen:

```
Dim rngChk As Range           'Range-Objekt in shtChk
Dim rngLog As Range           'Range-Objekt in Logbuch
Dim tDef As TableDef           'neue Tabelle
Dim fld As Field               'aktuell anzuhängendes Feld
Dim rs As Recordset            'Recordset in aktueller Tabelle
Dim iRowChk As Long            'Zeilenzeiger in rngChk
Dim iColChk As Long            'Spaltenzeiger in rngChk
Dim iRowTmp As Long            'Zeilenzeiger in rngTmp
Dim strFName As String         'Titel der Spalte in rngChk
Dim lngFType As String         'Feldtypkonstante
Dim strTable As String         'Name der Datenbanktabelle
Dim nHeaderRows As Long       'Anzahl der Kopfzeilen
Const iColFName As Long = 2    'Spalte "Feldname"
Const iColFType As Long = 3    'Spalte "Datentyp"
Const iColFLen As Long = 4     'Spalte "Feldlänge"
```

Ist die Bedeutung einer Variablen nicht in dieser Kürze zweifelsfrei möglich, so scheuen Sie sich bitte nicht, mehrere Zeilen hinzuschreiben:

```
Dim iColChk As Long           'Spaltenzeiger in rngChk
                                'gleichzeitig Zeilenzeiger in
rngTmp
```

In diesem Fall besteht ein direkter Zusammenhang zwischen dem Spaltenzeiger in rngChk und dem Zeilenzeiger in rngTmp, denn der Zeilenzeiger ist gleich Spaltenzeiger plus 1. Zwei separate Variablen sind hier nicht er-

forderlich, wohl aber ein Hinweis auf die doppelte Verwendung der Variablen `iColChk`.

11.4.3 Kommentierung im Code

Die in diesem Abschnitt schon herangezogene Prozedur `CreateTable` enthält folgende Kommentierungen im Code:

```
'Datenbanktabelle erzeugen
...
'Felder erzeugen
...
'Daten übertragen
...
'Fehlereintrag in Logbuch
```

Der Sinn und Zweck dieser Zeilen leuchtet ohne weitere Anmerkungen ein.

11.5 Anordnung von Prozeduren im Codemodul

Alphabetische Anordnung von Prozeduren

Wenn Sie sich Ereignisprozeduren von VBA durch Auswahl des Objektes und des gewünschten Ereignisses erzeugen lassen, so orientiert sich VBA an einer alphabetischen Struktur. Da der Prozedurname aus dem Objekt-namen und dem Ereignisnamen besteht, ergibt sich somit automatisch auch eine Sortierung der Ereignisprozeduren innerhalb des Objektes.

```
Private Sub cmdCheck_Click()
...
Private Sub cmdClose_Click()
...
Private Sub cmdCreateTable_Click()
...
Private Sub UserForm_Activate()
...
Private Sub UserForm_Terminate()
```

Was nun, wenn die `cmdCreateTable_Click()` aber eine Funktion `CheckSheet()` aufrufen soll, so stellt sich die Frage, wo sie denn nun eingeordnet werden soll. Rein alphabetisch gesehen müsste sie je vor den `cmd`-Ereignisroutinen erscheinen. Das wiederum hat zwangsläufig ein häufiges Blättern im Code zur Folge, wenn zu einer Aufgabe mehrere Prozeduren gehören, die aufgrund ihres Namens an den unterschiedlichsten Stellen des Codemoduls zu finden sind.

Einen Ausweg stellt die thematische Anordnung dar, der ich in größeren Codemodulen gewöhnlich den Vorzug gebe. Sie erinnern sich noch an die XlsToDB.xls, aus der auch obiges Codefragment entlehnt ist. Neben den allgemeinen Prozeduren rund um die Initialisierung und das Schließen der UserForm gibt es im Codemodul der frmTrans zwei wesentliche Aufgaben:

- ▼ Überprüfen der zu übertragenden Excel-Tabelle *sowie*
- ▼ Anlegen der Datenbanktabelle und Übertragen der Daten.

Diese drei Aufgaben sind unter den Bezeichnungen

- ▼ Überprüfen der Tabelle(n),
- ▼ Erzeugen der Datenbanktabelle(n) *sowie*
- ▼ Allgemeine Prozeduren

gruppiert worden. Die allgemeinen Prozeduren sind übrigens ans Ende gewandert, da sie üblicherweise als erste erstellt und danach selten verändert werden.

```
'*****
'Überprüfen der Tabelle(n)
'*****

Private Sub cmdCheck_Click()
...

Private Sub CheckSheet(ByVal shtAct As Worksheet)
...

'*****
'Erzeugen der Datenbanktabelle(n)
'*****

Private Sub cmdCreateTable_Click()
...
Private Sub CreateTable(ByVal shtAct As Worksheet)
...

'*****
'Allgemeine Prozeduren
'*****

Private Sub cmdClose_Click()
```

```
...  
Private Sub cmdSetup_Click()  
...  
Private Sub UserForm_Activate()  
...  
Private Sub UserForm_Terminate()  
...
```

Um die Kommentarzeilen für die Gruppierung von den Prozedurköpfen abzuheben, wurde zur Abgrenzung eines solchen Abschnittsbezeichners das Multiplikationszeichen * verwendet.

Bei besonders komplexen Aufgaben gibt es noch die Möglichkeit, diese komplett in eigene Modulblätter auszulagern. Das macht es allerdings erforderlich, dass die Zentrale Prozedur Public deklariert wird. Im obigen Beispiel wären dies die Funktionen `CheckSheet()` und `CreateTable()`.

11.6 Konstanten und Enumerationen

11.6.1 Spaltenzeiger

Auf den ersten Blick leuchtet es kaum ein, weshalb Spaltenzeiger in Form von Konstanten eine Erleichterung bringen sollen. Diese Einschätzung ändert sich jedoch spätestens dann, wenn Sie gezwungen sind in einer Tabelle eine Spalte einzufügen und anschließend in Dutzenden von Programmzeilen statische Spaltenzeiger von Hand anzupassen.

Hier noch einmal ein Beispiel aus der bereits bekannten `XLS-ToDB.xls`. Diese Applikation ermöglicht es, Excel-Tabellen in Access-Tabellen zu konvertieren. In der Prozedur `CheckSheet` werden die Spalten der Excel-Tabelle analysiert und Konvertierungsparameter in einer Template-Tabelle hinterlegt. Die Zugriffe auf die Spalten dieser Template-Tabelle erfolgen über Konstanten:

```
Const iColSName As Long = 1 'Spalte "Spaltenname"  
Const iColFName As Long = 2 'Spalte "Feldname"  
Const iColFTyp As Long = 3 'Spalte "Datentyp"  
Const iColFLen As Long = 4 'Spalte "Feldlänge"
```

Im weiteren Code dieser Prozedur sehen Zugriffe auf diese Tabelle beispielhaft so aus:

```
rngTmp.Cells(iColChk + 1, iColSName).Value = strTitle
```

Wird nun vor der Spalte C eine Spalte eingefügt, so ändern Sie die beiden letzten Konstanten zu 4 und 5 und Ihr Programm muss laufen!

Haben Sie sprechende Konstantennamen verwendet (und weshalb sollten Sie das nicht), wird auch das Schreiben des Codes vereinfacht, denn Sie können statt Zahlen die einfacher zu merkenden Namen einsetzen.

11.6.2 Enumerationen

In Kapitel 10 haben wir bereits über Enumerationen gesprochen, die gerade in Klassenmodulen sehr angenehm sein können. Wie schön wäre es, wenn wir statt des gewohnten Bildes bei der Parametrierung der Find-Methode, wie in Abbildung 11.1 zu sehen, eine aussagekräftige Enumeration zur Auswahl vorfinden.

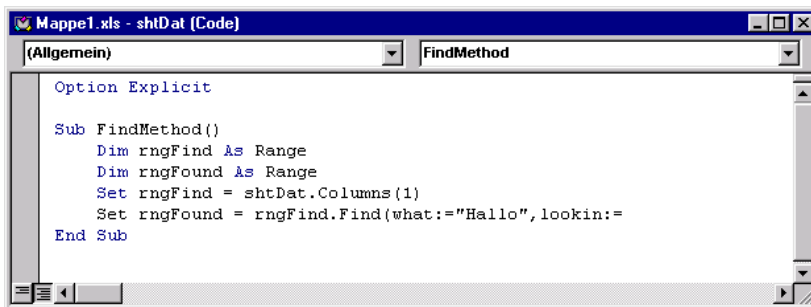


Abbildung 11.1: Find-Methode ohne Enumeration

Abbildung 11.2 zeigt, wie es aussehen *könnte*.

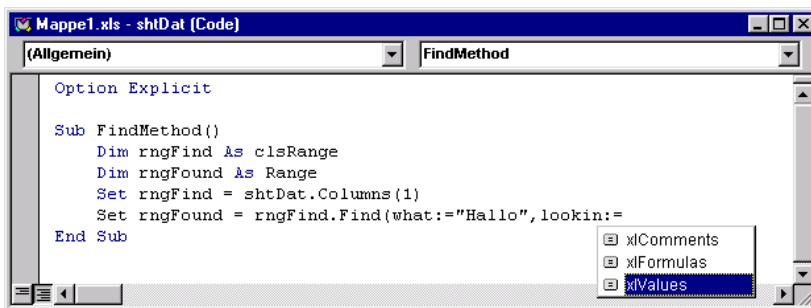


Abbildung 11.2: Find-Methode mit Enumeration

Für diese enorme Erleichterung sorgt folgender Code der Klasse `clsRange`:

```

Private rng As Range

Public Enum enumLookIn
    xlFormulas = -4123
  
```

```

        xlValues = -4163
        xlComments = -4144
    End Enum

    Public Enum enumLookAt
        xlPart = 2
        xlWhole = 1
    End Enum

    Public Function Find(What As String, _
        Optional ByVal LookIn As enumLookIn = xlValues, _
        Optional LookAt As enumLookAt = xlWhole) As Range
        Set Find = rng.Find(What:=What, LookIn:=LookIn, _
            LookAt:=LookAt)
    End Function

    Public Property Set cRange(ByVal rngNewRange As Range)
        Set rng = rngNewRange
    End Property

```

Verwendet wird die Klasse so:

```

Sub FindMethod()
    Dim rngFind As clsRange
    Dim rngFound As Range
    Set rngFind = New clsRange
    Set rngFind.cRange = shtDat.Columns(1)
    Set rngFound = rngFind.Find(What:="Hallo", _
        LookIn:=xlValues, LookAt:=xlWhole)
    If Not rngFound Is Nothing Then
        MsgBox rngFound.Address
    End If
End Sub

```



Machen Sie reichlich Gebrauch von Enumerationen und Sie werden von größerem Komfort und einer größeren Datensicherheit profitieren. Denken Sie bitte daran, dass Excel 97 diese Enumerationen leider nicht unterstützt.

11.7 Die Benutzerschnittstelle

Auch wenn's schwer fällt, aber:

Das Programm muss dem Anwender gefallen, nicht dem Entwickler.

Und wenn Sie ein guter Entwickler werden wollen, so sollten Sie im Idealfall bereits reichlich Erfahrung mit schlechten Programmen gesammelt haben und mit dem festen Vorsatz ans Werk gehen, es einfach besser zu machen. Hierzu einige Tipps.

11.7.1 Statusanzeige

Wenn der Anwender sich anhand der LED der Festplatte darüber informieren muss, ob sein Programm noch läuft, so tun Sie weder dem Anwender noch sich selbst einen Gefallen. Zeigen Sie dem Anwender, dass sein Programm arbeitet. Und das können Sie am besten dadurch tun, dass Sie ihm erzählen, *was* das Programm gerade tut.

In Kapitel 8 sahen Sie, wie man mit zwei Labels eine ProgressBar nachbilden kann.

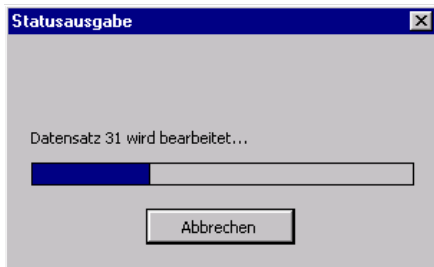


Abbildung 11.3: Formular mit ProgressBar

Sie benötigen hierfür folgende Steuerelemente:

lblStatus zur Ausgabe des Textes

Geben Sie in der Caption einen Hinweis darauf, was das Programm gerade tut.

lblBack als hinterer Rahmen

Der BorderStyle ist fmBorderStyleSingle, die Caption wird entfernt.

lblFore als vorderer Fortschrittsbalken

Der BorderStyle ist auch fmBorderStyleSingle, die BackColor erhält die Systemfarbe Markierung und die Caption wird ebenfalls entfernt. Die Width-Eigenschaft errechnet sich durch:

```
lblFore.Width = lblBack.Width / nRows * iRow
```

`nRows` steht in diesem Zusammenhang für die Anzahl der durchzuführenden Operationen und `iRow` beinhaltet die laufende Nummer der aktuellen Aktion.

cmdCancel zum Abbrechen

Schauen Sie einmal je Schleifendurchlauf über `DoEvents` nach, ob etwas anliegt. In der Ereignisroutine der Schaltfläche setzen Sie eine öffentliche Variable (`bCancel`) auf `True`, die wiederum in der Schleife nach `DoEvents` abgefragt wird.

Nach Änderung von `lblStatus.Caption` oder `lblFore.Width` bitte ein `Me.Repaint` folgen lassen, um die Form zu aktualisieren.

Wenn Sie kein Formular auf dem Bildschirm haben und auch keine dedizierte `StatusForm` einsetzen wollen, dann steht Ihnen ja immer noch die `StatusBar`-Eigenschaft des `Application`-Objekts zur Verfügung:

```
Application.StatusBar = "Datensatz " & iRow & _
                        " wird bearbeitet..."
```

Denken Sie bitte daran, am Ende die Kontrolle über die `StatusBar` wieder an Excel zurückzugeben:

```
Application.StatusBar = False
```

Ein verträumtes Beep nach Ende der Bearbeitung signalisiert dem Anwender akustisch, dass Ihr Programm fertig ist.

11.7.2 ComboBoxes

Wenn eine `ComboBox` Einträge enthält, aber noch nichts ausgewählt wurde, so kann der `Listindex` nicht einfach auf `-1` stehen bleiben. Entweder Sie treffen eine Vorauswahl oder teilen dem Anwender mit, dass er etwas auswählen soll, wie Abbildung 11.4 zeigt.

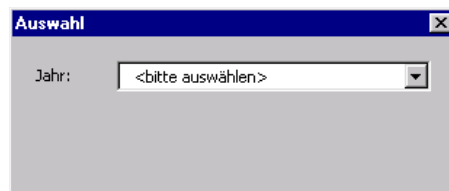


Abbildung 11.4: `ComboBox` mit Auswahlhinweis

Der dafür erforderliche Code könnte so aussehen:

```
Private Sub cmbYear_Load()
    Dim iYear As Long      'Schleifenvariable
```

```

cmbYear.AddItem "<bitte auswählen>"
For iYear = -2 To 0
    cmbYear.AddItem Year(Date) + iYear
Next
cmbYear.ListIndex = 0
End Sub

```

11.7.3 ControlTipText

Nutzen Sie die `ControlTipText`-Eigenschaft für zusätzliche Hinweise:

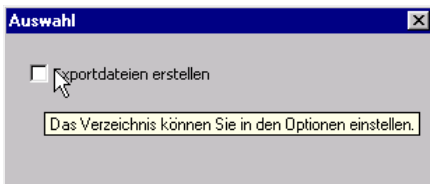


Abbildung 11.5: CheckBox mit `ControlTipText`

11.8 Einfache Parameterablage

Wenn eine Anwendung auf andere Dateien zugreifen muss, so ist in der Regel nicht davon auszugehen, dass Pfade und Namen dieser Dateien bekannt und unveränderlich sind. In vorausschauend entworfenen Programmen sind diese Parameter jederzeit durch den Anwender einstellbar. Auch wir sind daran gewöhnt, dass sich die Programme, mit denen wir arbeiten, an die von uns vorgenommenen Einstellungen beim nächsten Programmstart erinnern. Wir benötigen also eine Möglichkeit diese Parameter abzulegen und beim nächsten Programmstart wieder auszulesen.

Wenn es darum geht, *einen* oder *zwei* Werte abzulegen und wieder zu finden, so reicht folgende Lösung aus:

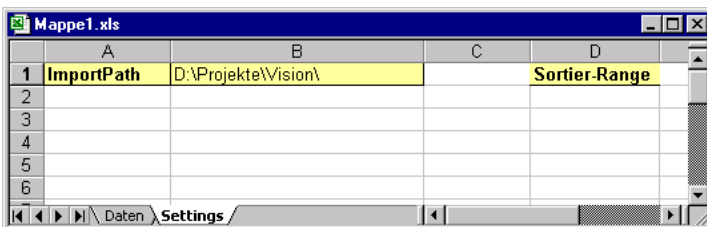


Abbildung 11.6: Settings-Tabelle, einfache Variante

Der Zugriff kann über einen mehr oder weniger direkten Weg erfolgen:

```
strPath = shtSettings.Range("B1").Value
```

Solange niemand auf die Idee kommt, diese Zelle B1 zu verschieben, funktioniert diese Lösung. Hiergegen hilft natürlich auch die Verwendung benannter Zellen, auf die Sie dann so zugreifen können:

```
strImportPath = Thisworkbook.Names("ImportPath"). _  
                RefersToRange.Value
```

oder

```
strImportPath = shtSettings.[ImportPath].Value
```

Offen gesagt halte ich nicht viel von benannten Zellen, aber das ist eine private Meinung. Vermutlich liegt sie in den schlechten Erfahrungen begründet, die ich mit Dateien sammeln durfte, in denen Hunderte solcher Namen herumgeisterten. Natürlich wusste dann niemand mehr, was wozu diente ...

Doch zurück zur Sache. Auch wenn es nur ein Wert ist, sollten Sie dennoch den Wert in Spalte B schreiben und die betreffende Zelle der Spalte A mit einem Hinweis darauf versehen, was das denn nun für ein Wert ist, der da in der benachbarten Zelle residiert.

Sortier-Range Die Abbildung zeigt auch eine Zelle, die mit Sortier-Range betitelt ist. Das ist eine sehr nützliche Angelegenheit, denn es kommt häufig vor, dass irgendwelche Werte sortiert werden müssen. Akrobaten unter den VBA-Entwicklern greifen hier zu raffinierten Sortieralgorithmen wie Quick Sort oder Bubble Sort. In PowerPoint mag man zu solchen Klimmzügen genötigt sein, aber einen Excel-Range zu beschreiben, sortieren und wieder auszulesen ist eine sehr einfache Angelegenheit.

11.9 Settings.xls

Auf der Buch-CD finden Sie eine Datei des Namens Settings.xls, in der einige interessante Techniken vorgestellt werden:

- ▼ Zugriff auf eine Settings-Tabelle
- ▼ Zugriff auf Registry-Settings
- ▼ Zugriff auf spezielle Systemverzeichnisse über API
- ▼ Aufruf eines Verzeichnisdialogs über API
- ▼ Zugriff auf Ländereinstellungen über API

Die Beispieldatei enthält einen Dialog, in dem alle diese Punkte eingesetzt sind. Abbildung 11.7 zeigt die Dialog dieser Datei in Aktion.

Da Zugriffe auf die Registry mit `SaveSetting` und `GetSetting` in Kapitel 7 schon anhand eines Beispiels demonstriert wurde, können wir hier auf eine intensive Behandlung verzichten. Der dazugehörige Code befindet sich in der Klasse `clsSettingsRegistry`, deren Schnittstellen kompatibel sind zu der Klasse `clsSettingsSheet`. Sie können beide Klassen wechselseitig einbinden, indem Sie die Deklaration in `modMain` ändern und eine der beiden folgenden Varianten wählen:

```
Public cSettings As New clsSettingsRegistry
```

oder

```
Public cSettings As New clsSettingsSheet
```

Wollen wir uns nun also der Klasse `clsSettingsSheet` zuwenden.

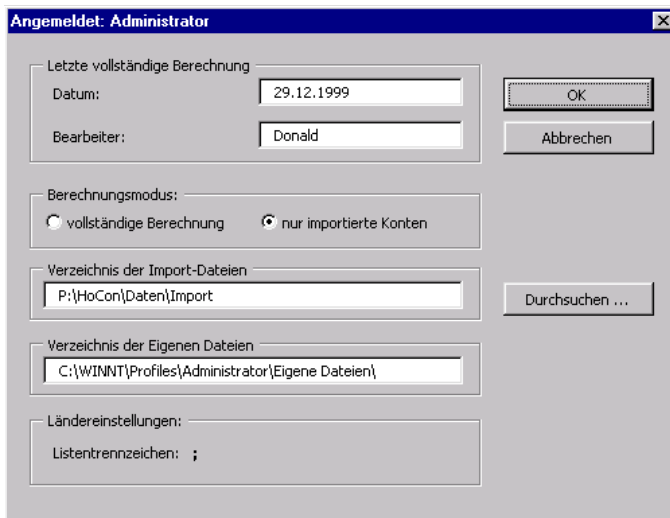
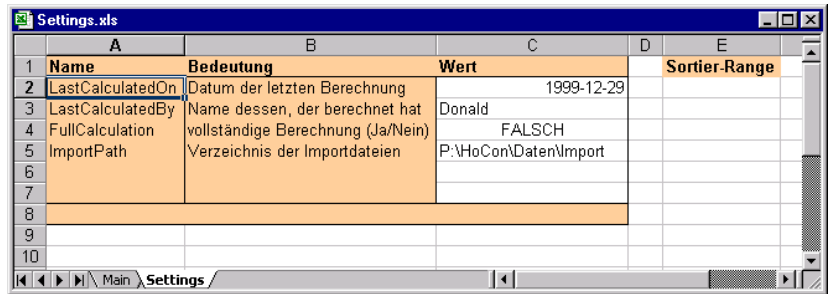


Abbildung 11.7: Settings.xls, Dialog

11.9.1 Die Klasse `clsSettingsSheet`

Diese Klasse stellt die Daten der Tabelle Settings als Eigenschaften zur Verfügung. Abbildung 11.8 zeigt die vier dort hinterlegten Werte:



	A	B	C	D	E
1	Name	Bedeutung	Wert		Sortier-Range
2	LastCalculatedOn	Datum der letzten Berechnung	1999-12-29		
3	LastCalculatedBy	Name dessen, der berechnet hat	Donald		
4	FullCalculation	vollständige Berechnung (Ja/Nein)	FALSCH		
5	ImportPath	Verzeichnis der Importdateien	P:\HoCon\Daten\Import		
6					
7					
8					
9					
10					

Abbildung 11.8: Settings.xls, Tabelle Settings

Die Eigenschaften dieser Klasse, die jeweils in Schreib- und Leserichtung implementiert sind, tragen sinnvollerweise den Namen, der auch in Spalte A der Tabelle Settings zu sehen ist.

Allen gemeinsam ist, dass sie die Zeile ermitteln müssen, in der sich der gesuchte Wert befindet. Diese Arbeit wird von einer privaten Funktion erledigt, die den Begriff übergeben bekommt, den sie in Spalte A suchen soll:

```
Private Function GetRow(strName As String) As Long
    ' Rückgabe der Zeile, in der strName gefunden wurde
    Dim rngFound As Range 'Range-Objekt für Find-Methode
    Dim iRow As Long      'Zeilerzeiger in shtSettings
    On Error GoTo errHandler
    Set rngFound = shtSettings.Columns(1).Find( _
        what:=strName, LookIn:=xlValues, lookat:=xlWhole)
    If rngFound Is Nothing Then
        Err.Raise vbObjectError + 9, , "Es existiert " & _
            "kein Wert des Namens " & strName & "."
    Else
        GetRow = rngFound.Row
    End If
    Exit Function
errHandler:
    Err.Raise Err.Number, "clsSettings:GetRow"
End Function
```

Eine Find-Methode macht sich mit dem Suchbegriff über die Spalte A her und gibt die Row-Eigenschaft der gefundenen Zelle zurück; war die Suche erfolglos, erzeugt sie den Fehler 9. Da es sich um eine private Methode handelt, wird die Konstante vbObjectError nicht hinzuaddiert.

Da sich die Eigenschaftsroutinen sehr ähnlich sind, begnügen wir uns mit den beiden Routinen, die sich um den ImportPath kümmern:

```
Public Property Get ImportPath() As String
    Dim iRowSetting As Long    'Zeilenzeiger in shtSettings
    On Error GoTo errHandler
    iRowSetting = GetRow("ImportPath")
    ImportPath = shtSettings.Cells(iRowSetting, 3).Value
    Exit Property
```

```
errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "clsSettings:Property Get ImportPath"
    End If
    Err.Raise Err.Number + vbObjectErrorEnd Property
End Property
```

```
Public Property Let ImportPath(ByVal strNewValue As String)
    Dim iRowSetting As Long    'Zeilenzeiger in shtSettings
    On Error GoTo errHandler
    iRowSetting = GetRow("ImportPath")
    shtSettings.Cells(iRowSetting, 3).Value = strNewValue
    Exit Property
```

```
errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "clsSettings:Property Get ImportPath"
    End If
    Err.Raise Err.Number + vbObjectError
End Property
```

Beide Prozeduren ermitteln zuerst den Spaltenzeiger mit Hilfe der privaten GetRow-Methode und greifen damit auf das Worksheet-Objekt zu.

Der Zugriff auf die Eigenschaften dieser Klasse erfolgt in den Ereignisroutinen UserForm_Initialize und cmdOK_Click:

```
Private Sub UserForm_Initialize()
    ...
    txtLastCalculatedOn.Text = cSettings.LastCalculatedOn
    txtLastCalculatedBy.Text = cSettings.LastCalculatedBy
    optNewAccounts.Value = True
    optFullCalculation.Value = cSettings.FullCalculation
    txtImportPath.Text = cSettings.ImportPath
    ...
End Sub
```

```

Private Sub cmdOK_Click()
    ...
    cSettings.LastCalculatedOn = txtLastCalculatedOn.Text
    cSettings.LastCalculatedBy = txtLastCalculatedBy.Text
    cSettings.FullCalculation = optFullCalculation.Value
    cSettings.ImportPath = txtImportPath.Text
    ...
End Sub

```

Erwähnenswert bei der `UserFrom_Initialize` ist vielleicht noch, dass durch Zuweisen von `True` zu dem ersten `OptionButton` und anschließendes Zuweisen der Eigenschaftswerts `cSettings.FullCalculation` zu dem zweiten `OptionButton` einer von beiden sicher aktiviert ist.

11.9.2 Die Klasse `clsRegistry`

Diese Klasse ist wesentlich spannender als die vorangegangene, denn hier kommen gleich fünf API-Funktionen zum Einsatz, die uns höchst nützliche Dienste erweisen.

Das Verzeichnis Eigene Dateien des angemeldeten Users

In Abbildung 11.7 ist das Verzeichnis Eigene Dateien des aktuell angemeldeten Anwenders zu sehen, wofür die folgende Zeile in `UserFrom_Initialize` verantwortlich ist:

```
txtPersonalFolder.Text = cReg.PersonalFolder
```

Im Beispiel ist nur eine Eigenschaft dargestellt, aber die daran beteiligten Funktionen können eine ganze Reihe von verschiedensten Systemverzeichnissen ermitteln. Aus diesem Grund ist die eigentliche Arbeit in einer privaten Funktion untergebracht, die auf den Namen `GetSpecialFolder` hört:

```

Private Function GetSpecialFolder(ByVal lngCSIDL As Long) _
    As String
    ' Systemverzeichnis ermitteln
    'Argumente:
    ' lngCSIDL: Konstante für Systempfad
    Dim lngResult As Long 'Rückgabe der Funktionen
    Dim strPath As String 'gesuchter Pfad
    Dim IDL As ITEMIDLIST 'Item-ID-List
    On Error GoTo errHandler
    lngResult = SHGetSpecialFolderLocation(0, lngCSIDL, IDL)
    If lngResult = 0 Then
        strPath = Space(260)
        lngResult = SHGetPathFromIDList(ByVal IDL.mkid.cb, _

```

```

ByVal strPath)

If lngResult <> 0 Then
    GetSpecialFolder = Left$(strPath, InStr(strPath, _
        Chr(0)) - 1) & "\"
End If
End If
Exit Function

```

errHandler:

```

Err.Raise Err.Number, "clsRegistry:GetSpecialFolder"
End Function

```

Die `SHGetSpecialFolderLocation` erhält eine Konstante übergeben, die den Typ des Verzeichnisses festlegt. Diese Konstante wird in der aufrufenden Eigenschaft erzeugt, wie wir noch sehen werden. Außerdem wird sie mit einer geschachtelten benutzerdefinierten Variable des Namens `IDL` versorgt, in deren Teilkomponente `IDL.mkid.cb` sie einen Zeiger auf das gewünschte Verzeichnis zurückgibt. Ist die Rückgabe der Funktion = 0 (`lngResult = 0`), so ist alles in Ordnung und die zweite Funktion `SHGetPathFromIDList` ermittelt anhand der ID den Klartextnamen. Wie es so üblich ist, ist die Rückgabe Null-terminiert und wird über die `Left`-Funktion zurechtgeschnitten, indem die Stelle gesucht wird, an der sich der erste `Chr(0)` befindet. Backslash dahinter und fertig.

Den benutzerdefinierten Datentyp und die Deklarationen der Funktionen selbst ersparen wir uns hier; Sie können sie im Klassenkopf besichtigen.

Die Eigenschaftsroutine `PersonalFolder`, die von dieser privaten Funktion Gebrauch macht, sieht so aus:

```

Public Property Get PersonalFolder() As String
    ' Rückgabe des PersonalFolders
    Const CSIDL_PERSONAL = &H5 'Eigene Dateien in
    Profilen
    PersonalFolder = GetSpecialFolder(CSIDL_PERSONAL)
End Property

```

Die Konstante `CSIDL_PERSONAL` ist eigentlich überflüssig, denn man könnte auch den Wert direkt übergeben:

```

PersonalFolder = GetSpecialFolder(5)

```

Sie ist nur deshalb aufgeführt, damit man im Zweifel auch weiß, unter welchem Begriff man in der MSDN suchen muss, wenn man Details darüber in Erfahrung bringen will.

Die GetPath-Funktion

Diese Funktion produziert auf Kommando den folgenden Dialog:

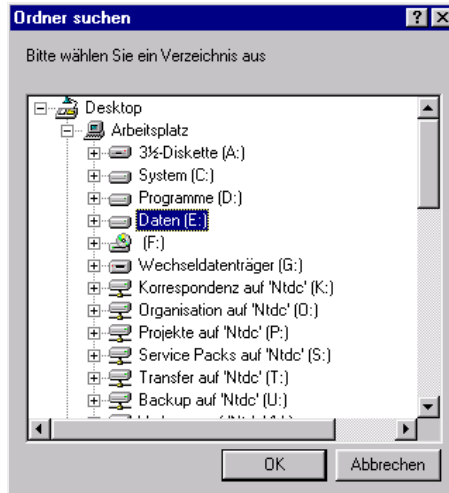


Abbildung 11.9: Settings.xls, SHBrowseForFolder

Als ich diesen Dialog zum ersten Mal in einem VBA-Programm einsetzte, war ich schlichtweg begeistert. Ich hoffe, dass er auch bei Ihnen eine Lücke schließt. Allerdings war ich auch ein wenig sauer, dass man zuerst die MSDN umgraben muss, um solche Spezialitäten nutzen zu können, denn eigentlich gehört das in den Funktionsumfang von VBA.

Werfen wir auch hier einen Blick auf die Funktion:

```
Public Function GetPath() As String
    ' Verzeichnis ermitteln und zurückgeben
    Dim lngIDList As Long 'Zeiger auf den ausgewählten Pfad
    Dim strTitle As String 'Titel des Dialogs
    Dim tBrowseInfo As BrowseInfo 'UDF mit Informationen
    On Error GoTo errHandler
    strTitle = "Bitte wählen Sie ein Verzeichnis aus"
    tBrowseInfo.lpszTitle = lstrcat(strTitle, "")
    tBrowseInfo.ulFlags = BIF_RETURNONLYFSDIRS + _
                        BIF_DONTGOBELOWDOMAIN
    lngIDList = SHBrowseForFolder(tBrowseInfo)
    If lngIDList <> 0 Then
        GetPath = Space(260)
        SHGetPathFromIDList lngIDList, GetPath
        GetPath = Left(GetPath, InStr(GetPath, Chr(0)) - 1)
    End If
    errHandler:
    GetPath = ""
End Function
```

```
End If
Exit Function
```

```
errHandler:
    Err.Raise Err.Number + vbObjectError, "clsReg:GetPath"
End Function
```

Zuerst wird die Funktion `lstrcat` zweckentfremdet, die zwei Zeichenketten zusammensetzt und einen Zeiger darauf zurückgibt. Hier erhält sie unseren gewünschten Titel, ergänzt ihn um nichts (`»«`) und gibt den Zeiger darauf zurück. Dieser Zeiger verschwindet dann in dem benutzerdefinierten Datentyp `tBrowseInfo`, der schließlich der Funktion `SHBrowseForFolder` überantwortet wird. Im Element `ulFlags` des `tBrowseInfo` wird mit den beiden Konstanten festgelegt, dass nur Verzeichnisse ausgewählt werden dürfen und dass kein Zugriff auf Domainenmitglieder erlaubt ist. Die Rückgabe der Funktion ist ein Zeiger auf das ausgewählte Verzeichnis bzw. 0, wenn der Dialog über Abbrechen geschlossen wurde.

Dieser Zeiger wird zusammen mit einer entsprechend vorbelegten Zeichenkette (`GetPath`) an die zweite Funktion `SHGetPathFromIDList` verfüttert, der in dieser ID-Liste nachschaut, wie denn das Verzeichnis mit der ID `lngListID` denn nun heißt. Danach wird auch hier die Rückgabe bei dem obligatorischen Null abgeschnitten.

Der Aufruf befindet sich in der Ereignisroutine der Schaltfläche Durchsuchen:

```
Private Sub cmdBrowse_Click()
    ' Aufruf des Pfad-Dialogs
    Dim strNewPath As String           'Rückgabe aus Klasse
    On Error GoTo errHandler
    strNewPath = cReg.GetPath
    If strNewPath <> "" Then
        txtImportPath.Text = strNewPath
    End If
Exit Sub
```

```
errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "frmSettings:cmdBrowse_Click"
    End If
    Err.Raise Err.Number
End Sub
```

Wenn Sie diese Klasse in ein zukünftiges Projekt einbinden, gestaltet sich dieser Dialog nebst Rückgabe zu einem Einzeiler:

```
strNewPath = cReg.GetPath
```

11.9.3 Die Klasse clsNLS

NLS steht für National Language Support und beinhaltet alle Einstellungen, die von Sprache oder nationalen Gepflogenheiten geprägt werden. Hierbei wird zwischen Systemeinstellungen und solchen unterschieden, die vom aktuellen Benutzer beeinflussbar sind. Uns interessieren hier allerdings nur die benutzerabhängigen Parameter.

Hierunter fallen Informationen über die eingestellte Sprache, Datums- und Zeitformate, Währungssymbole, aber auch die Monatsnamen und die Namen der Wochentage. Um diese, auch Locale Settings genannten Werte, geht es in dieser Klasse, die bereits in Kapitel 7.7.7 vorgestellt wurde.

Der Dialog der Settings.xls enthält exemplarisch in einem Label das aktuell eingestellte Listentrennzeichen, das im UserForm_Initialize-Ereignis geladen wird:

```
lblListDelimiter.Caption = cNLS.ListDelimiter
```

11.9.4 Schlussbemerkungen zu Settings.xls

In dieser Datei wurde alles zusammengestellt, was sich bei mir im Laufe der Zeit an Zugriffen auf Einstellungen so angesammelt hat. Und immer wenn ich dachte, nun wirklich alles beisammen zu haben, kam dann wieder eine Anforderung, die bislang nicht realisiert wurde. Der jüngste Fall betrifft die Auflistung aller vorhandenen Data Sources der ODBC zur bequemeren Auswahl.

Mit Settings.xls haben Sie einen soliden Grundstock in der Hand, der Ihre Anwendungen etwas eleganter und flexibler machen wird.

11.10 Mehrsprachige Applikationen

Es ist wohl nicht die Ausnahme, dass Unternehmen Daten mit ausländischen Unternehmen austauschen, seien es nun Töchter, Kunden oder Lieferanten. Also etwa die Controlling-Anwendung, die Planzahlen an Niederlassungen verschickt und diese mit Ist-Zahlen angereichert zurück-erhält. Ein Weg wäre es, die Benutzerschnittstelle gleich in Englisch zu entwerfen, in der Annahme, dass Controller dies ohnehin beherrschen. Doch es geht – wie bei allen Programmen – auch bei Excel-VBA-Anwendungen um Akzeptanz, Akzeptanz beim Anwender. Somit spricht einiges

für eine mehrsprachige Anwendung, zumal dies gar nicht so kompliziert ist, wie es vielleicht jetzt noch aussehen mag.

Nachdem wir uns darauf geeinigt haben, unsere Anwendung mehrsprachig zu machen, sollten wir uns über das Design dieser Code-Komponente unterhalten. Klammern wir einmal alle möglichen Quick-and-Dirty-Lösungen aus, so kommt nur eine wieder verwendbare Schnittstelle in Frage; auf Deutsch eine Klasse. Aufgabe aller Klassen ist es, zum einen eine standardisierte Schnittstelle zu bieten, die ohne großen Aufwand in andere Projekte integriert werden kann, und zwar auch im fortgeschrittenen Stadium. Zum anderen soll sie die Komplexität vor der Client-Anwendung verbergen, sodass wir sie ohne Bedenken einsetzen können.

Überlegen wir nun, welche Aufgaben unsere Klasse erledigen soll. So soll beispielsweise die Schaltfläche `cmdCancel` wahlweise »Abbrechen« oder »Cancel« als Aufschrift tragen. Aber auch Hinweise in Statuszeilen oder `MessageBoxes` sollten dem sprachlichen Kontext folgend etwa »Die neuen Planwerte sind berechnet worden.« bzw. »The new forecast figures have been processed.«. Somit haben wir bereits zwei Teilaufgaben:

▼ Bereitstellen von Begriffen

▼ Bereitstellen von Meldungen

Um Fehlermeldungen müssen wir uns zumeist nicht kümmern, da sie nationalisierten Windows-Bibliotheken entnommen werden. Das gilt natürlich nicht für eigene Fehlertexte, womit wir den nächsten Punkt hätten:

▼ Bereitstellen von Fehlertexten

Das könnte man wohl mit dem Bereitstellen von Meldungen zusammenführen, eine Trennung macht aber Sinn, weil man sie in getrennten Tabellen (oder allgemeiner gehalten Datenquellen) bereitstellen kann.

Stellt sich noch die Frage, woher diese Klasse überhaupt weiß, welche Sprache sie verwenden soll. Nun, wir könnten es ihr mitteilen. Im vorangehenden Abschnitt über Anwendungsparameter haben wir uns eine komfortable Schnittstelle geschaffen, die solche Informationen verwaltet und auf Anforderung zurückgibt. Damit diese Einstellung verändert werden kann, müssen wir eine Benutzerschnittstelle in Form eines Dialogs bereitstellen. Was machen wir beim ersten Kontakt mit dem Benutzer? Legen wir eine Annahme zugrunde? Wir müssten also eine Standardsprache vorgeben, die der Anwender eben ändern muss, wenn ihm diese nicht zusagt. Oder lassen wir die Information offen und fragen den Anwender beim ersten Start? Das kommt der Sache schon näher. Optimal ist unsere Lösung aber erst dann, wenn wir Windows fragen, welche Sprache aktuell eingestellt ist.

11.10.1 Die Beispielanwendung

Auf der CD finden Sie die kleine Beispielanwendung Mehrsprachig.xls mit einem nur angedeuteten Dialog `frmMain` und dem zur Spracheinstellung verwendeten `frmLanguage`. Schauen wir uns zuerst den Hauptdialog an:

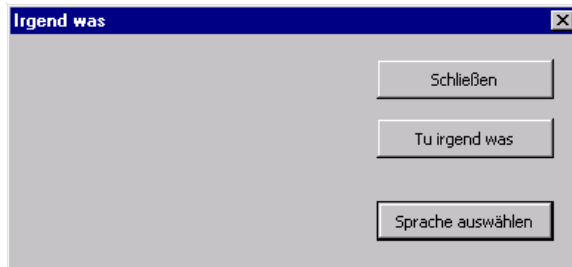


Abbildung 11.10: Mehrsprachig.xls, der Hauptdialog

Der Button »Sprache auswählen« führt zu diesem Dialog:

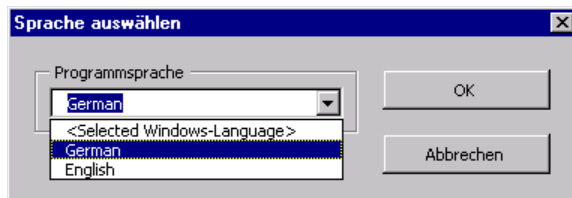


Abbildung 11.11: Mehrsprachig.xls, Sprachdialog

Hier stehen uns als Sprachen die Windows-Sprache des aktuellen Users, German und English zur Auswahl. Wurde English ausgewählt, ändert der Dialog selbst bereits die Sprache.

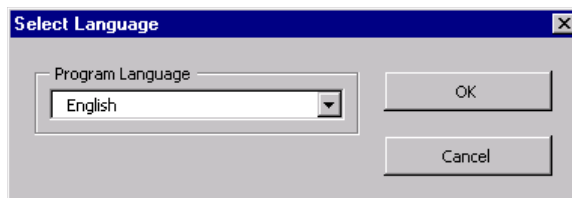


Abbildung 11.12: Mehrsprachig.xls, Sprachdialog nach Umstellung

Wird dieser Dialog nun geschlossen, so erscheint wieder unser Hauptdialog, nun allerdings in geänderter Sprache:

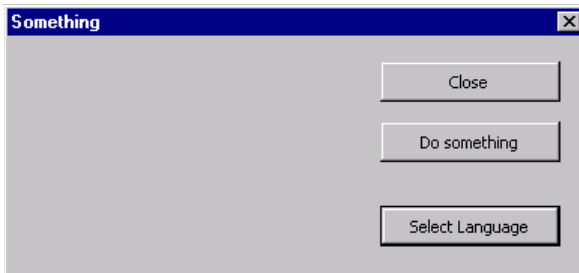


Abbildung 11.13: Mehrsprachig.xls, Hauptdialog nach Sprachwechsel

Die Schaltfläche Do something erzeugt einen Fehler, anhand dessen wir die korrekte Übersetzung der Fehlerausgabe überprüfen können. Wie Abbildung 11.14 zeigt, wurde der Fehlertext selbst nicht übersetzt. Für diesen Text ist eine VBA-Bibliothek verantwortlich.



Abbildung 11.14: Mehrsprachig.xls, Fehlerausgabe

Ein recht kompaktes Beispiel, das aber alle Komponenten enthält, die eine mehrsprachige Applikation ausmachen.

Bevor wir zu dem innigst erwarteten Code kommen, werfen wir noch einen Blick oder zwei auf die beteiligten Tabellen.

11.10.2 Die Tabellen

Mehrsprachig.xls. verfügt über folgende Tabellen:

Tabelle Main

Diese Tabelle hat einfach nur einen angenehmen Hintergrund und eine Schaltfläche zum Start des in unserem Beispiel nur angedeuteten Hauptdialogs.

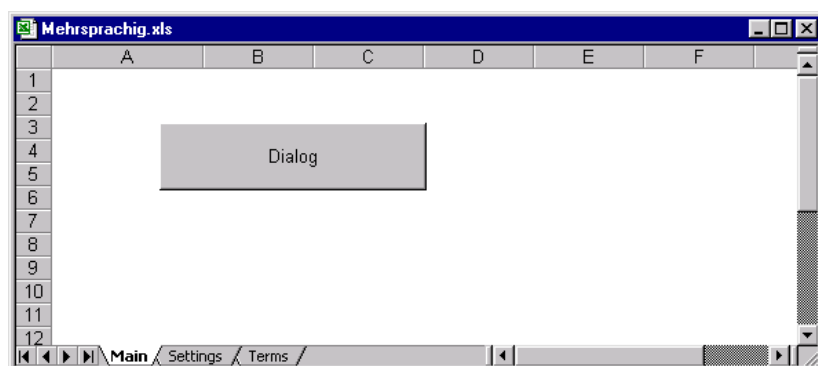


Abbildung 11.15: Mehrsprachig.xls, Tabelle Main

Tabelle Settings

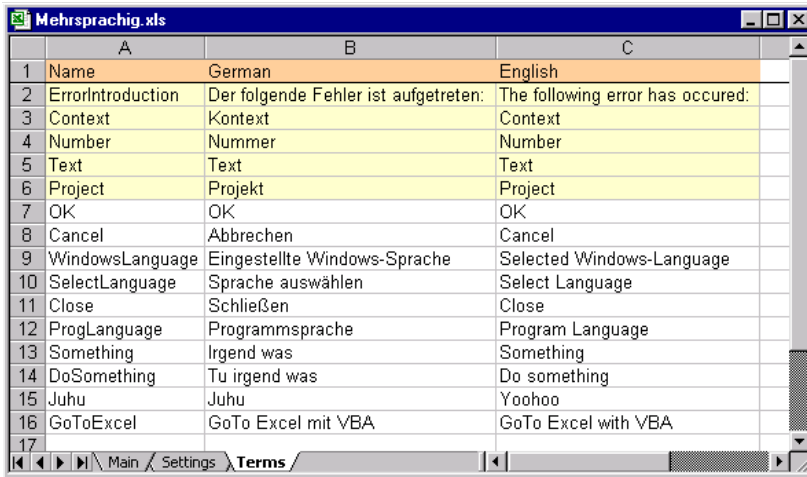
Hier liegen die Anwendungsparameter, in diesem Beispiel nur die eingestellte Sprache.

	A	B	C	D	E
1	Name	Bedeutung	Wert		
2	ProgLanguage	eingestellte Sprache	English		
3					
4					
5					
6					

Abbildung 11.16: Mehrsprachig.xls, Tabelle Settings

Tabelle Terms

Die Tabelle Terms enthält im oberen Teil die von der Fehler-MessageBox verwendeten Begriff so wie darunter alle programmspezifischen Termini in allen verwendeten Sprachen. Aus den horizontal angeordneten Sprachen werden die verfügbaren Sprachen abgelesen und zur Auswahl angeboten.



	A	B	C
1	Name	German	English
2	ErrorIntroduction	Der folgende Fehler ist aufgetreten:	The following error has occurred:
3	Context	Kontext	Context
4	Number	Nummer	Number
5	Text	Text	Text
6	Project	Projekt	Project
7	OK	OK	OK
8	Cancel	Abbrechen	Cancel
9	WindowsLanguage	Eingestellte Windows-Sprache	Selected Windows-Language
10	SelectLanguage	Sprache auswählen	Select Language
11	Close	Schließen	Close
12	ProgLanguage	Programmsprache	Program Language
13	Something	Irgend was	Something
14	DoSomething	Tu irgend was	Do something
15	Juhu	Juhu	Yooohoo
16	GoToExcel	GoTo Excel mit VBA	GoTo Excel with VBA
17			

Abbildung 11.17: Mehrsprachig.xls, Tabelle Terms

11.10.3 Der Code

Die folgende Abbildung zeigt die Struktur der Codekomponenten:

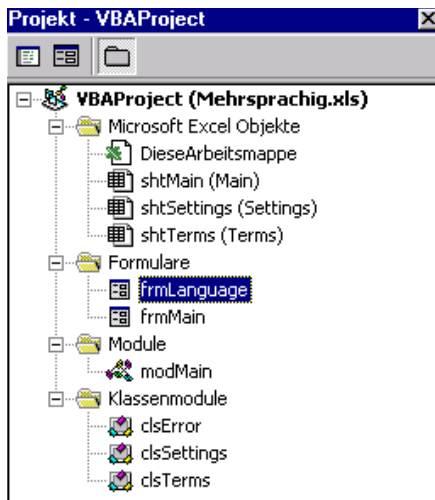


Abbildung 11.18: Mehrsprachig.xls, Codekomponenten

Bei allen nachfolgenden Prozeduren wurde der obligatorische Modulkopf auf eine kurze Kommentarzeile verkürzt, die Fehlerbehandlungsroutine wird ebenfalls nicht dargestellt. Alles dies finden Sie jedoch selbstverständlich im Originalcode vor.

Modul shtMain

Dieses Modul enthält lediglich eine öffentliche Prozedur, die den Dialog frmMain auf den Bildschirm bringt.

```
Public Sub cmdDialog_Click()  
    frmMain.Show vbModal  
End Sub
```

Modul modMain

Auch hier passiert relativ wenig, lediglich die öffentlichen Variablen für die Klassen werden gebildet:

```
Option Explicit  
Public cError As New clsError  
Public cSettings As New clsSettings  
Public cTerms As clsTerms
```

Modul frmMain

Der nachfolgende Code der frmMain ist durch den konsequenten Klasseinsatz auch noch sehr einfach gehalten:

```
Private Sub UserForm_Initialize()  
    'Klassenvariablen erzeugen und SetTerms aufrufen  
    On Error GoTo errHandler  
    Set cTerms = New clsTerms  
    cError.LoadTerms  
    Call SetTerms  
End Sub
```

```
Private Sub cmdClose_Click()  
    'Dialog ausblenden  
    Me.Hide  
End Sub
```

```
Private Sub cmdDoSomething_Click()  
    'MessageBox ausgeben und Fehler erzeugen  
    MsgBox cTerms.GetTerm("Juhu"), vbInformation, _  
        cTerms.GetTerm("GoToExcel")  
    Err.Raise 9  
End Sub
```

```
Private Sub cmdLanguage_Click()  
    'Aufruf der frmLanguage  
    frmLanguage.Show
```

```

    'auf Sprachwechsel prüfen
    If cTerms.LanguageHasChanged Then
        cError.LoadTerms
        Call SetTerms
        cTerms.LanguageHasChanged = False
    End If    Unload frmLanguage
End Sub

```

```

Private Sub SetTerms()
    'Captions aus Terms-Klasse laden
    Me.Caption = cTerms.GetTerm("Something")
    cmdClose.Caption = cTerms.GetTerm("Close")
    cmdDoSomething.Caption = cTerms.GetTerm("DoSomething")
    cmdLanguage.Caption = cTerms.GetTerm("SelectLanguage")
End Sub

```

Die letzte Routine `SetTerms` ist für die sprachabhängige Beschriftung des Dialogs verantwortlich, die jeweils durch Aufruf der Methode `GetTerm` der Klasseninstanz `cTerms` produziert wird. Diese Routine wird natürlich im `Initialize`-Ereignis der Form aufgerufen. Auch noch nicht sonderlich aufregend.

Die in `cmdDoSomething_Click` erzeugte `MessageBox` enthält ebenfalls sprachabhängige Ausgaben, die ebenfalls über den Methodenaufruf `GetTerm` ermittelt werden.

Die einzige Prozedur, die eventuell eines zweiten Blickes bedarf, ist die Ereignisroutine `cmdLanguage_Click`, die den Sprachwechseldialog aufruft. Nach dessen Schließung wird die Eigenschaft `LanguageHasChanged` der Termsklasse abgefragt. Ist diese `True`, so werden `cError.LoadTerms` und `SetTerms` erneut aufgerufen, um die Texte der Fehlerklasse und die sprachabhängigen Teile des Dialogs neu zu erzeugen. Danach wird diese Eigenschaft wieder auf `False` gesetzt.

Klasse `clsTerms`

Um den Code im Sprachwechseldialog `frmLanguage` interpretieren zu können, müssen wir uns zuerst das Klassenmodul `clsTerms` etwas näher anschauen. Hier sieht der Code zwangsläufig etwas komplizierter aus. Doch das liegt in der Absicht des Erfinders, denn wir wollen ja Komplexität aus den vielen Clients heraushalten, die diese Klasse verwenden; aber irgendwo muss er ja schließlich bleiben.

Widmen wir uns zuerst der zentralen Methode `GetTerm`, um danach den Code zu vervollständigen, den diese Methode benötigt, um korrekt arbeiten zu können.

```

Public Function GetTerm(ByVal strName As String) As String
    'Rückgabe des angeforderten Begriffs
    'Range-Objekt zum Auffinden der Sprachspalte erzeugen
    Dim rngFound As Range
    'Abbruch, wenn Klasse nicht bereit
    If Not bTermsIsReady Then
        GetTerm = strName
        Exit Function
    End If
    'Zeilenzeiger ermitteln und Begriff zurückgeben
    Set rngFound = shtTerms.Columns(1).Find(what:=strName, _
        LookIn:=xlValues, lookat:=xlWhole)
    If rngFound Is Nothing Then
        'Fehler erzeugen
        Err.Raise vbObjectError + 9, , _
            "Sheet >>Terms<< is obviously corrupted."
    Else
        GetTerm = shtTerms.Cells(rngFound.Row, _
            iColTerms).Value
    End If
End Function

```

Die entscheidende Zeile ist

```
GetTerm = shtTerms.Cells(rngFound.Row, iColTerms).Value
```

Im Worksheet-Objekt `shtTerm` der Tabelle `Terms` wird der Inhalt einer Zelle dem Methodennamen `GetTerm` zugewiesen. Die Lage der Zelle wird durch den Zeilenzeiger `rngFound.Row` bestimmt. Wie der Name bereits sagt, ist dies die `Row`-Eigenschaft eines Range-Objekts. Gebildet wird `rngFound` dadurch, dass in Spalte A des Tabellenobjekts `shtTerms` das dieser Funktion übergebene Argument `strName` gesucht wird:

```
set rngFound = shtTerms.Columns(1).Find(strName...)
```

Der Spaltenzeiger der Zielzelle ist `iColTerms`, der nicht in dieser Methode ermittelt werden muss, da er nur bis zum Sprachwechsel konstant bleibt.

Der Rest dieser Prozedur besteht aus nicht-funktionalem Code. Da wäre zu nennen Abbruch und Rückgabe des übergebenen Begriffs, wenn die Klasse nicht bereit ist:

```

If Not bTermsIsReady Then
    ...

```

Und Erzeugen eines Fehlers, sollte die Find-Methode den zu suchenden Begriff strName nicht finden:

```
If rngFound Is Nothing Then
    ...
```

Soweit zur GetTerm-Methode.

Wenden wir uns nun der Prozedur Class_Initialize zu, in der die Vorbereitungen zum Einsatz der eben beschriebenen GetTerm-Methode getroffen werden müssen:

```
Private Sub Class_Initialize()
    'Sprache ermitteln und SetTermsColumn aufrufen
    strLanguage = cSettings.ProgLanguage
    If strLanguage = "" Then
        'Keine Sprache in Settings hinterlegt
        strLanguage = cSettings.UserLanguage
    End If
    Call SetTermsColumn
End Sub
```

Der Private Variablen strLanguage wird zuerst der Inhalt der Eigenschaft ProgLanguage zugewiesen. Ist noch keine Sprache in den Settings hinterlegt worden, so gibt die Eigenschaft einen leeren String zurück. In diesem Falle wird die Eigenschaft UserLanguage der Klasse clsSettings abgefragt, um die eingestellte Windows-Sprache als Klartext zu ermitteln. Danach wird die Routine SetTermsColumn aufgerufen, die den Spaltenzeiger iColTerms einstellt. Doch zuerst zur GetUserLanguage, der es obliegt, die Windows-Sprache zu bestimmen.

Somit verbleibt noch, den Spaltenzeiger iColTerms festzulegen, was in der folgenden Prozedur SetTermsColumn passiert:

```
Private Sub SetTermsColumn()
    'Spaltenzeiger ermitteln und
    'TermsIsReady-Eigenschaft vorbereiten
    'Range-Objekt zum Auffinden der Sprachspalte:
    Dim rngFound As Range
    Set rngFound = shtTerms.Rows(1).Find( _
        what:=strLanguage, LookIn:=xlValues, _
        lookat:=xlWhole)
    If rngFound Is Nothing Then
        bTermsIsReady = False
    Else
        bTermsIsReady = True
```

```

        iColTerms = rngFound.Column
    End If
End Sub

```

Die Tabelle Terms ist, wie Sie sich erinnern, so aufgebaut, dass in Zeile 1, beginnend mit Spalte 2, die verfügbaren Sprachen enthalten sind. Diese Sprachen sind natürlich eine Teilmenge der in Spalte 2 der Tabelle Languages definierten Sprachen.

Eine Find-Methode ermittelt in dieser Prozedur die Spalte, in der die Begriffe in der gewünschten Sprache stehen. Sollte die Find-Methode fehl schlagen, so wird die auf Modulebene definierte Variable bTermsIsReady auf False gesetzt. Hat's hingegen geklappt, so wird der ebenfalls auf Modulebene deklarierten Variablen iColTerms die Column-Eigenschaft des Range-Objekts zugewiesen und bTermsIsReady auf True eingestellt.

Bleibt uns noch eine Methode, für die wir im Moment noch keine Verwendung haben. Es bietet sich jedoch an, sie auch in diesem Kontext vorzustellen.

Um im Sprachdialog eine Auswahl der Sprachen zu ermöglichen, benötigen wir eine Methode, die diese Sprachen ermittelt:

```

Public Function GetLanguages() As String
    'Rückgabe der im Kontext definierten Sprachen
    'lokaler Spaltenzeiger in Tabelle Terms:
    Dim iColTermsLocal As Long
    Dim strLanguages() As String    'temporäres Array
    ReDim strLanguages(shtTerms.UsedRange.Columns.Count - 2)
    For iColTermsLocal = 2 To _
        shtTerms.UsedRange.Columns.Count
        strLanguages(iColTermsLocal - 2) = _
            shtTerms.Cells(1, iColTermsLocal).Value
    Next
    GetLanguages = strLanguages
End Function

```

In dem temporären Array strLanguages() werden die Sprachen durch Abarbeiten aller Spaltentitel in Tabelle shtTerms, beginnend mit der zweiten Spalte bis zu UsedRange.Columns.Count, hinterlegt und an die aufrufende Prozedur in frmLanguage zurückgegeben, die wir noch kennen lernen werden.

Was bleibt uns noch?

Wir benötigen eine Schnittstelle für die eingestellte Sprache, die wir in der Variablen `strLanguage` abgelegt haben:

```
Public Property Get Language() As String
    Language = strLanguage
End Property
```

```
Public Property Let Language(ByVal strProgLanguage _
                               As String)
    strLanguage = strProgLanguage
    Call SetTermsColumn
End Property
```

Wird eine neue Sprache übergeben (Property `Let Language`), so muss nun natürlich der Spaltenzeiger `iColTerms` neu eingestellt werden, was durch den erneuten Aufruf der `SetTermsColumn` erfolgt.

Außerdem müssen wir eine Schnittstelle für die beiden internen Variablen bilden, wodurch die Bereitschaft der Klasse `bTermsIsReady` und eine eventuelle Sprachänderung `bLanguageHasChanged` signalisiert wird, wobei erstere nur gelesen werden kann:

```
Public Property Get LanguageHasChanged() As Boolean
    LanguageHasChanged = bLanguageHasChanged
End Property
```

```
Public Property Let LanguageHasChanged(ByVal bNewValue _
                                         As Boolean)
    bLanguageHasChanged = bNewValue
End Property
```

```
Public Property Get TermsIsReady() As Boolean
    TermsIsReady = bTermsIsReady
End Property
```

Klasse `clsSettings`

Diese Klasse stellt uns die Eigenschaften `ProgLanguage` und `UserLanguage` bereit. Die erste wird über die bereits bekannten Wege (siehe 11.9.1) aus der Tabelle `Settings` ermittelt beziehungsweise dorthin zurückgeschrieben. Sie bedient sich einer `private GetRow`-Methode, die den Zeilenzeiger des angeforderten Begriffs zurückgibt.

Die `UserLanguage` wird über die ebenfalls bereits bekannte `GetLocale-Info` aus dem NLS-API erfragt (siehe 11.9.3).

Wozu, so werden Sie sich fragen, müssen wir das API bemühen, wo es doch in Office 2000 ein Objekt namens `LanguageSettings` gibt, welches uns die Sprache übermittelt?

Um durch diese Vorgehensweise Deutsch als eingestellte Sprache sicher ermitteln zu können, benötigten wir etwa folgenden Code:

```
If Application.LanguageSettings. _  
    LanguagePreferredForEditing( _  
        msoLanguageIDGerman) = True Then  
    strLanguage = "German"  
ElseIf Application.LanguageSettings. _  
    LanguagePreferredForEditing( _  
        msoLanguageIDGermanAustria) = True Then  
    strLanguage = "German"  
ElseIf Application.LanguageSettings. _  
    LanguagePreferredForEditing( _  
        msoLanguageIDGermanLiechtenstein) = True Then  
    strLanguage = "German"  
ElseIf Application.LanguageSettings. _  
    LanguagePreferredForEditing( _  
        msoLanguageIDGermanLuxembourg) = True Then  
    strLanguage = "German"  
ElseIf Application.LanguageSettings. _  
    LanguagePreferredForEditing( _  
        msoLanguageIDSwissGerman) = True Then  
    strLanguage = "German"  
...  

```

Um also die deutsche Sprache in allen regionalen Varianten erfassen zu können, müssen wir diese fünf bandwurmartigen `If`s und `ElseIf`s zum Besten geben, für die englische Sprache benötigten wir gleich 13 dieser `If`s.

Modul `frmLanguage`

Aufgabe der Sprachform `frmLanguage` ist es, die aktuell eingestellte Sprache wiederzugeben und die Auswahl einer anderen verfügbaren Sprache zu ermöglichen. Außerdem soll es auch möglich sein, die Windows-Einstellung zu aktivieren.

Schauen wir uns zuerst die Ereignisprozedur des `Initialize`-Ereignisses der `UserForm` an:

```
Private Sub UserForm_Initialize()  
    Call SetTerms
```

```

    strPrevLanguage = cTerms.Language
    Call cmbLanguage_Load
End Sub

```

Nach dem Aufruf der `SetTerms`, die auch hier die Captions einstellt, wird zuerst in der auf Modulebene deklarierten Variablen `strPrevLanguage` die momentan eingestellte Sprache abgelegt. Danach wird die Prozedur `cmbLanguage_Load` aufgerufen, welche die Combobox lädt:

```

Private Sub cmbLanguage_Load()
    'Sprache aus Settings (kann auch leer sein):
    Dim strLanguage As String
    'Verfügbare Sprachen aus cTerms:
    Dim strLanguages() As String
    'Zeiger in strLanguages():
    Dim iLanguage As Long
    'spätere ListIndex der ListBox:
    Dim iListIndex As Long
    'Sprache aus Settings-Klasse ermitteln
    strLanguage = cSettings.ProgLanguage
    'verfügbare Sprachen aus cTerms laden
    strLanguages = cTerms.GetLanguages
    'Combobox leeren und "Windows-Sprache" einfügen
    cmbLanguage.Clear
    cmbLanguage.AddItem "<" & _
        cTerms.GetTerm("Windows-Language") & ">"
    'restliche Sprachen ergänzen
    For iLanguage = 0 To UBound(strLanguages)
        cmbLanguage.AddItem strLanguages(iLanguage)
    Next
    'ListIndex ermitteln
    For iLanguage = 0 To cmbLanguage.ListCount - 1
        'bei Übereinstimmung mit Registry-Sprache
        'ListIndex positionieren
        If cmbLanguage.List(iLanguage) = strLanguage Then
            iListIndex = iLanguage
            Exit For
        End If
    Next
    'ListIndex zuweisen
    bForgetIt = True
    cmbLanguage.ListIndex = iListIndex
    bForgetIt = False
End Sub

```

Der Variablen `strLanguage` wird zu Anfang dieser Prozedur die in den Settings hinterlegte Sprache zugewiesen:

```
strLanguage = cSettings.ProgLanguage
```

wobei dieser Wert auch einen leeren String enthalten kann. Nun werden dem String-Array `strLanguages()` alle definierten Sprachen der Terms-Klasse übergeben:

```
strLanguages = cTerms.GetLanguages
```

Da der Begriff Windows-Language als oberster Eintrag der Combobox erscheinen soll, wird er, nachdem die ComboBox geleert wurde, danach mit der Methode `AddItem` hinzugefügt:

```
cmbLanguage.Clear  
cmbLanguage.AddItem "<" & cTerms.GetTerm(...) & ">"
```

Anschließend werden die Array-Einträge in der Combobox ergänzt:

```
For iLanguage = 0 To UBound(strLanguages)  
    cmbLanguage.AddItem strLanguages(iLanguage)  
Next
```

Wir müssen noch den `ListIndex` ermitteln, auf dass die Combobox die Sprache anzeigt, der aktuell eingestellt ist:

```
For iLanguage = 0 To cmbLanguage.ListCount - 1  
    If cmbLanguage.List(iLanguage) = strLanguage Then  
        iListIndex = iLanguage  
    Exit For  
End If  
Next
```

Was passiert, wenn die eingestellte Sprache als `NullString` aus den Settings zurückkam? Da kein Eintrag der Combobox einen solchen Nullstring beinhalten kann, wird keine Übereinstimmung gefunden und `iListIndex` wird ihren Initialisierungswert 0 aufweisen, der wiederum auf den ersten Eintrag Windows-Language zeigen wird. Da eine Veränderung des `ListIndex` einer Combo- oder ListBox das `Click`-Ereignis provozieren würde, nutzen wir die bereits bekannte Variable `bForgetIt`, um die Ereignisprozedur abubrechen. Diese muss zuvor allerdings auf `True` gesetzt werden:

```
bForgetIt = True  
cmbLanguage.ListIndex = iListIndex  
bForgetIt = False
```

Unser Dialog harrt nun der Dinge, die da noch kommen werden.

Hierunter fällt beispielsweise die Auswahl einer anderen Sprache, was uns in der Ereignisprozedur des Click-Ereignisse der Combobox mitgeteilt wird:

```
Private Sub cmbLanguage_Click()
    If cmbLanguage.ListIndex = 0 Then 'Windows-Sprache
        If cTerms.Language <> cSettings.UserLanguage Then
            cSettings.ProgLanguage = ""
            cTerms.Language = cSettings.UserLanguage
            Call SetTerms
        End If
    Else
        cSettings.ProgLanguage = cmbLanguage.Text
        cTerms.Language = cmbLanguage.Text
        Call SetTerms
    End If
End Sub
```

Hier müssen wir unterscheiden, ob der neue ListIndex nun 0 ist, was der Einstellung Windows-Language entspricht. In diesem Fall fragen wir die Terms-Klasse nach der derzeit eingestellten Sprache und vergleichen diese mit der eingestellten Benutzersprache, die uns wiederum die GetUserLanguage-Methode dieser Klasse mitteilen kann. Stimmen die beiden nicht überein, so stellen wir die Benutzersprache als aktuelle Sprache der Terms-Klasse ein und rufen die lokale SetTerms zum Laden der neuen Captions auf. Steht der ListIndex nicht auf der Windows-Sprache, so übergeben wir diese Sprache an die Terms-Klasse, wobei wir auch hier die neuen Captions laden müssen. Da wir die zum Ladezeitpunkt des Dialogs gültige Sprache in strPrevLanguage abgelegt haben, können wir einen Sprachwechsel bedenkenlos an die Klasse übergeben, denn wir kennen die Ursprungssprache ja noch und können diese im Falle des Abbruchs des Dialogs wieder an die Klasse übergeben:

```
Private Sub cmdCancel_Click()
    cTerms.Language = strPrevLanguage
    Me.Hide
End Sub
```

Wird der Dialog allerdings mit der OK-Schaltfläche geschlossen, müssen wir Ordnung in unsere Sprachen bringen, was durch den Aufruf der Prozedur cmbLanguage_Save geschieht:

```
Private Sub cmdOK_Click()
    Call cmbLanguage_Save
```

```

        Me.Hide
    End Sub

```

Genau diese `cmbLanguage_Save` ist nun die letzte Routine, die wir noch betrachten müssen:

```

Private Sub cmbLanguage_Save()
    If strPrevLanguage <> cmbLanguage.Text Then
        If cmbLanguage.ListIndex = 0 Then 'Windows-Sprache
            cSettings.ProgLanguage = ""
        Else
            'neue Sprache an Settings
            cSettings.ProgLanguage = cmbLanguage.Text
        End If
    End If 'Terms-Klasse über Sprachänderung informieren
    If strPrevLanguage <> cTerms.Language Then
        cTerms.LanguageHasChanged = True
    End If
End Sub

```

Diese Prozedur hat Ähnlichkeiten mit der `cmbLanguage_Click`, allerdings wird zuerst untersucht, ob `strPrevLanguage` ungleich dem in der Combobox ausgewählten Eintrag ist. Ist darüber hinaus die Windows-Sprache eingestellt, so wird kurzerhand `ProgLanguage` zu »«. Andernfalls wird die neue Sprache, die ja dann eine gültige Sprache sein muss, in den Settings dauerhaft hinterlegt. Sind nun die aktuelle Sprache der Terms-Klasse und die in `strPrevLanguage` hinterlegte Sprache unterschiedlich, so setzen wir die `LanguageHasChanged`-Eigenschaft der Klasse auf `True`. Und diese Eigenschaft wird ja in der diesen Sprachdialog aufrufenden Prozedur abgefragt, um über die `SetTerms` die Captions dieser UserForm zu aktualisieren.

Schlussbemerkungen zu Mehrsprachig.xls

Dieses Beispiel hat uns nun über rund 15 Seiten begleitet. Wenn Sie den Ausführungen einigermaßen folgen konnten, so bereiten Ihnen die Grundzüge der Komponentenentwicklung wohl keine Schwierigkeiten mehr.

Die Klasse `clsError` haben wir bereits behandelt und, wie es sich für richtige Komponenten gehört, konnten wir sie einsetzen, ohne auch nur in den Code hineinzuschauen. Dieser Grundsatz gilt natürlich auch für die hier entwickelte Terms-Klasse, die bereits bekannte Settings-Klasse und den Sprachendialog. Da der verwendete Hauptdialog `frmMain` auch recht einfach gehalten ist, können Sie ihn als Vorlage für zukünftige, mehrsprachige Anwendungen verwenden. Lediglich die Schaltfläche `cmdDoSome-`

thing nebst Ihrer Ereignisprozedur würde ich an Ihrer Stelle noch entfernen.

Viel Erfolg damit.

11.11 Tabellendesign

In diesem Abschnitt werden wir ein wenig Datenbanktheorie betreiben. Natürlich vertrete ich nicht die Ansicht, dass alle Theoreme der Datenbankwelt in Excel zur Anwendung kommen sollen. Bei dem einen oder anderen Buch zum Thema Datenbanken erschleicht einen auch der Verdacht, im falschen Film zu sein. Aber immerhin hinterlassen diese Bücher einen guten Eindruck bei Besuchern, wenn sie schön nach Farbe geordnet in einem der Regale stehen.

Problematisch ist diese Anleihe bei Datenbanken schon aus dem Grund, weil Excel-Tabellen im Grunde mit Datenbanktabellen nicht verglichen werden können. Aber der tiefere Sinn der hier vorgestellten Regeln liegt zum einen in Redundanzvermeidung, woran uns ja allen gelegen sein sollte. Zum anderen – und das ist vielleicht noch wichtiger – geht es darum, bei der Änderung eines Wertes alle Stellen im Unternehmen zu finden, an denen diese Information aktualisiert werden muss. Excel verleitet dazu, Daten zu kopieren und in modifizierter Form weiterzuverarbeiten. Redundanzen und Excel-Tabellen sind allzu oft Synonyme. Nach einem kurzen Ausflug zur Normalisierung kehren wir dann zur Normalität zurück und prüfen, was wir davon übernehmen können.

11.11.1 Worum geht es ? – Ein Beispiel

Es geht uns bei den folgenden Betrachtungen nicht um Analyse- oder Ausgabetabellen, in denen Informationen verschiedenster Basistabellen oder gar externer Datenquellen zusammen dargestellt werden. Das ist ja vielfach der Grund für den Einsatz von Excel, weil sich Daten eben mit einfachen Mitteln in vielerlei Form ansprechend gestalten lassen. Dass hier auch das eine oder andere Mal mit oder ohne VBA kopiert wird, ist nicht der Stein des Anstoßes. Es geht vielmehr genau um diese Basisdaten, also originäre Informationen, die sonst nirgendwo zu finden sind.

Es geht um Datengräber wie in Abbildung 11.19 ersichtlich. Diese Tabelle ist die erste Verdichtung einzelner Produktkalkulationen, die auch genauso aussehen. Der betreffende Controller pflegte jeden Monat Dutzende dieser Tabellen, die per Formelbezügen zu zwei weiteren Ebenen verdichtet wurden. Da sich in den Basistabellen originäre Daten befanden, wurden diese Tabellen enorm wichtig für das Unternehmen. Aber sie erforderten einen sehr hohen Pflegeaufwand. Als dann die Zeilenstruktur

dynamisch werden sollte, war der Controller mit den Bordmitteln von Excel schlichtweg am Ende.

The screenshot shows an Excel spreadsheet with a complex data table. The table has multiple columns with headers like 'Aktiver ST', 'Reaktiv ST', and various sub-headers. The data is organized into rows, with some rows highlighted in yellow. The spreadsheet is titled 'csp_aaa.xls'.

Abbildung 11.19: Zur Abschreckung

Das ist dann die Situation, in der sich die Anwender daran erinnern, dass es ein VBA in Excel gibt. Und wenn Sie sich nicht rechtzeitig aus dem Staub gemacht haben, so stehen Sie vor einem mächtigen Problem, denn das sind die unerfreulichsten Projekte, die einem so unterkommen. Sie können in diesen Dimensionen nicht mehr mit benannten Zellen operieren. Ihnen bleibt nur der Weg, über die Suche von (hoffentlich) konstanten Zellwerten zu versuchen, in der Tabelle zu navigieren. Als Zeilen- und Spaltenzeiger der Cells-Methode des Range-Objekts ergeben sich dann solche Konstruktionen:

$$iCol = nColsBasis + (iQuarter * 4)$$

$$iRow = nRowsHeader + (iKoSt * 2) + nRowsOffset$$

Begrüßenswert ist die Praxis der FHs und Unis, in nahezu allen Fachrichtungen eine informationstechnische Grundausbildung anzubieten. Ob die aber an Grundsätze des Tabellendesigns in Excel denken?

11.11.2 Normalisierung in der Theorie

Professionelle Datenbanken kennzeichnet die Erfüllung der so genannten Normalisierung. Dahinter verbergen sich im Kern fünf Regeln, die als erste bis dritte Normalform, Boyce-Codd-Normalform und vierte Normalform bezeichnet werden. Wir machen jedoch nach der dritten Normalform den Sack zu und ersparen uns damit eine Menge Akrobatik.

Sinn und Zweck dieser Normalisierung ist die Vermeidung von redundanten Datenbeständen. Hierbei dreht es sich nicht in erster Linie um die Einsparung von Speicherplatz, sondern um die Datensicherheit bei Änderungen.

Hier wird der Begriff des Schlüsselfelds (Primary Key) eingeführt. Dieser besagt, dass das Feld einen eindeutigen Begriff beinhaltet, der als Schlüssel zu weiteren Informationen herangezogen werden kann. Der Inhalt eines Nicht-Schlüssel-Felds hingegen dient im gesamten Datenkontext nicht als Schlüssel zu weiteren Informationen. In Abbildung 11.20 sehen wir in der ersten Spalte eine Niederlassungsnummer, die sicherlich im gesamten Unternehmen eindeutig ist, denn D101 ist die Nummer der Niederlassung Dresden. Herr Sachs aber, derzeitiger Leiter der Niederlassung, ist kein Schlüsselkriterium. Der Ort der Niederlassung auch nicht, denn diese Niederlassung könnte beispielsweise umziehen und unter derselben Niederlassungsnummer in Freiberg weiter existieren.



	A	B	C	D	E	F
2	Schlüsselfeld					
3	Nicht-Schlüssel-Felder					
4	Nr.	Niederlassung	Leiter			
5	D101	Dresden	Hr. Sachs			
6	D103	Radeberg	Hr. Berg			
7	D102	Frankfurt	Hr. Hess			
8	D104	Kassel	Hr. Hassel			

Abbildung 11.20: Schlüssel- und Nicht-Schlüssel-Felder

Schauen wir uns die drei Normalformen einmal an. Generell gilt übrigens, dass die Normalform n die Erfüllung der Normalform $n - 1$ voraussetzt.

Erste Normalform (1NF)

Die erste Normalform ist erreicht, wenn Attribute (Feldinhalte, Zellinhalte) atomar sind.

Hier ein Beispiel für nicht atomare Felder, die somit die erste Normalform nicht erreichen, weil eben die Städte nicht atomar vorliegen, sondern in einer Zelle zusammen auftreten:

	A	B	C	D
1	Bundesland	Niederlassungen		
2	Sachsen	Dresden, Radeberg		
3	Hessen	Frankfurt, Kassel		

Abbildung 11.21: Erste Normalform nicht erreicht

Wie so oft können wir die Verkettung der beiden Städte in je einem Feld auf zwei Arten auflösen. Abbildung 11.22 zeigt die Variante, die man nicht wählen sollte:

	A	B	C	D	E
1	Bundesland	Niederlassungen			
2	Sachsen	Dresden	Radeberg		
3	Hessen	Frankfurt	Kassel		

Abbildung 11.22: Erste Normalform nicht erreicht II

Die Anforderung der ersten Normalform wäre erfüllt, wenn wir die Tabelle in die Länge ziehen. Nun können wir auch zum Beispiel den Niederlassungsleiter eindeutig zuordnen. Abbildung 11.23 zeigt diesen Zustand.

	A	B	C	D
1	Bundesland	Niederlassungen	Leiter	
2	Sachsen	Dresden	Hr. Sachs	
3	Sachsen	Radeberg	Hr. Berg	
4	Hessen	Frankfurt	Hr. Hess	
5	Hessen	Kassel	Hr. Hassel	

Abbildung 11.23: Erste Normalform erreicht

Werfen wir nun einen Blick auf die zweite Normalform.

Zweite Normalform (2NF)

In der zweiten Normalform befindet sich eine Tabelle, wenn jedes Nicht-schlüssel-Feld von dem Schlüsselfeld (oder den Schlüsselfeldern als Ganzes) abhängig ist.

In der Umkehrung bedeutet dieser Satz, dass die 2NF nicht erreicht ist, wenn eine Abhängigkeit zwischen einem Nicht-Schlüssel-Feld und einem Schlüsselfeld vorhanden ist.

	A	B	C	D	E	F
1	Nr.	Niederlassung	Jahr	Umsatz		
2	D101	Dresden	1997	180		
3	D101	Dresden	1998	200		
4	D101	Dresden	1999	225		

Abbildung 11.24: Zweite Normalform nicht erreicht

In diesem Beispiel bilden die beiden Teil-Schlüssel Niederlassungsnummer (Nr.) und das Jahr den zusammengesetzten Schlüssel, denn nur durch die Nr. und das Jahr lässt sich der Umsatz eindeutig zuordnen. Aber der Name der Niederlassung ist das Hindernis, denn es gibt eine Abhängigkeit zwischen der Niederlassungsnummer und dem Namen. Um der zweiten Normalform gerecht zu werden, müssen wir also nur den Niederlassungsnamen entfernen:

	A	B	C	D	E	F
1	Nr.	Jahr	Umsatz			
2	D101	1997	180			
3	D101	1998	200			
4	D101	1999	225			

Abbildung 11.25: Zweite Normalform erreicht

Dritte Normalform (3NF)

Die dritte Normalform fordert, dass keine funktionalen Abhängigkeiten zwischen Nicht-Schlüsselfeldern existieren.

In der folgenden Abbildung 11.26 ist die Rohertragsquote zu sehen, die jedoch aus Umsatz und Rohertrag errechnet werden kann. Sie selbst stellt ein Nicht-Schlüssel-Feld dar und ist von den beiden Nicht-Schlüssel-Feldern Umsatz und Rohertrag abhängig.

	A	B	C	D	E	F
1	Nr.	Jahr	Umsatz	Rohertrag	Quote	
2	D101	1997	180	34	18,9 %	
3	D101	1998	200	38	19,0 %	
4	D101	1999	225	43	19,1 %	
5						

Abbildung 11.26: Dritte Normalform nicht erreicht

Auch hier erreichen wir durch entfernen der Spalte Quote unser Ziel:

	A	B	C	D	E	F
1	Nr.	Jahr	Umsatz	Rohertrag		
2	D101	1997	180	34		
3	D101	1998	200	38		
4	D101	1999	225	43		
5						

Abbildung 11.27: Dritte Normalform erreicht

Nun existieren keine Abhängigkeiten mehr zwischen Nicht-Schlüssel-Feldern.

Schlüssel werden in Datenbanken übrigens in der Regel als Long-Datentyp eingesetzt. Genau genommen wäre D101 kein Schlüssel, sondern bereits ein Datum. Doch das Arbeiten mit Long-Schlüsseln in Excel-Tabellen macht ja nun wirklich keinen Sinn.

11.11.3 Normalisierung in der (Excel-) Praxis

Der wichtigste Grundsatz betrifft das grundlegende Tabellendesign:



Trennen Sie Datentabellen von Analyse- und Ausgabetabellen.

Nach dieser Trennung reicht meistens ein gründlicher Blick auf die Datentabellen, um die Felder zu identifizieren, die den oben vorgestellten Normalisierungsregeln widersprechen. Ich bin übrigens sicher, dass eine ganze Menge Excel-Anwender und Entwickler ihre Tabellen sauber nach eigenen Kriterien anlegen, ohne zu wissen, dass sie der dritten Normalform genügen. Denn so revolutionär sind die Regeln eigentlich auch nicht.



Obwohl ein Datum in vielen anderen Datentabellen als Schlüssel-Kriterium verwendet werden kann, darf es nur einmal als Datum in Erscheinung treten.

Nennen wir diesen Satz den Kategorischen Imperativ des Tabellendesigns. Und nun kommen wir zu dem wichtigsten Grundsatz:

Machen Sie in Ihren Analyse- und Ausgabetabellen was sie wollen, sofern sie Ihre Daten schön säuberlich getrennt in Datentabellen archiviert haben.



In Datenbanken fällt es durch interne Mechanismen leicht, die Daten einzelner Tabellen über Ansichten oder Views zusammenzuführen und, was noch wichtiger ist, zu bearbeiten, als *wären* sie in einer Tabelle. Das gelingt uns in Excel nicht. Zwar sind wir in der Lage, über SVERWEIS die Daten einer anderen Tabelle über ein Schlüsselfeld in unserer Tabelle darzustellen. Aber sie sind dort nicht editierbar. Wollen wir das Schlüsselfeld selbst oder ein Nicht-Schlüsselfeld aus der Domain des Schlüsselfeldes verändern, müssen wir in diese Tabelle wechseln, in der sich die Nicht-Schlüsselattribute des besagten Schlüsselfeldes befinden und die Änderungen dort durchführen.

In Excel-Datenbeständen ohne VBA-Unterstützung sind solche Betrachtungen reines Wunschdenken. Dort wird kopiert, was die Zwischenablage erlaubt. Wenn jedoch VBA ins Spiel kommt, haben wir es in der Hand.

11.11.4 Skalierbarkeit

Bislang behandelten wir diese Normalisierungsregeln unter dem Aspekt der reinen Excel-Anwendung. Aber viele Anwendungen, die als reine Excel-Lösungen begannen, erhalten früher oder später eine Datenbank als Speichermedium. Wenn Sie dann die oben besprochenen Grundsätze des Tabellendesigns eingehalten haben und außerdem jede Datentabelle mit einer eigenen Datenklasse entkoppelt haben, wird die Portierung der Daten in eine Datenbank kaum Probleme aufwerfen. Da Ihre Datentabellen nach professionellen Grundsätzen gestaltet sind, werden die Tabellen- und Klassenstrukturen identisch sein. Sie müssen also lediglich die Datenklassen abändern, und ihr Programm müsste im Übrigen auf Anhieb funktionieren.

Wir werden in den Kapiteln 12 und 13 noch darauf zurückkommen.



Excel und Datenbanken

12

Kapitelüberblick

12.1	Data Access Objects	586
12.1.1	Wege des Datenzugriffs	588
12.1.2	Das DAO-Objektmodell	589
12.1.3	Zugriff auf eine JET-Datenbank	590
12.1.4	Zugriff auf eine ODBC-Datenbank	592
12.1.5	Zugriff auf Recordsets	594
12.1.6	Zugriff auf Felder	595
12.2	ActiveX Data Objects	596
12.2.1	Das ADO-Objektmodell	597
12.2.2	Navigation in einem Recordset	599
12.3	ANSI-SQL	601
12.3.1	SELECT	603
12.3.2	UPDATE	609
12.3.3	INSERT	610
12.3.4	DELETE	611
12.4	Die Klasse clsUtil	611
12.5	DBRechnung.xls	614
12.5.1	Die neuen Datenklassen	614
12.5.2	Der Kundendialog	617
12.5.3	Zusammenfassung	624

Excel wird vielfach dazu eingesetzt, ein Ergebnis aufzubereiten und in Form einiger weniger Werte zu präsentieren. Die hierzu erforderlichen Eingangsgrößen bestehen aus einer Reihe von Zahlen, die entweder vom Volumen her als gering einzuschätzen sind oder keiner hohen Dynamik unterliegen. Zu dieser Kategorie gehören auch Zwischenergebnisse oder Ausgangsgrößen, für die sich in der Folge niemand mehr interessiert. In diesen Fällen stellt die Arbeitsmappe eine in sich geschlossene Einheit dar.

Excel ist aber auch ein exzellentes Analysewerkzeug, dessen Einsatz dann sehr sinnvoll sein kann, wenn diese Voraussetzungen der mehr oder weniger starken Abgeschlossenheit der Anwendung nicht vorliegen. In der Praxis bedeutet dies das Vorliegen einer oder mehrerer der folgenden Bedingungen:

- ▼ Das zur Aufgabenstellung gehörende Datenvolumen ist groß.
- ▼ Die Daten unterliegen einer gewissen Dynamik.
- ▼ Die Ergebnisse stellen wiederum Eingangsgrößen anderer EDV-gestützten Unternehmensprozesse dar.
- ▼ Mehrere Personen arbeiten potenziell gleichzeitig mit den Daten.

Jede dieser genannten Kriterien ist dem Grunde nach ein hinreichendes Kriterium für den Einsatz einer Datenbank. Es gibt eine Reihe von Definitionen des Phänomens Datenbank, die dann sofort einleuchten, wenn man sich jahrelang mit Datenbanken beschäftigt hat. Deshalb wählen wir einen anderen Weg und untersuchen, was eine Excel-Arbeitsmappe alles können müsste, damit sie als Datenbank bezeichnet werden könnte. Der Konjunktiv des vorangehenden Satzes weist darauf hin, dass Excel-Dateien keine Datenbanken sind.

Datenbanktabellen

Eine Datenbank enthält Tabellen. Diese wiederum sind in Datensätze und Felder organisiert. Datensätze in Tabellen konventioneller Prägung sind alle gleich lang, weil eben jedes Datenfeld eine genau festzulegende Länge aufweist. Die Länge eines Datensatzes entspricht also der Gesamtlänge aller Felddefinitionen. Datenfelder wiederum sind stringent, das heißt, sie gehorchen jeweils einer festen Datentypdefinition, die dem Feld zugeordnet ist.

Solche Datentypen sind Ihnen längst bekannt, denn in Form von beispielsweise Integer, Long, Single, Double, String, Date oder Boolean sind sie uns ja schon begegnet. Der Vorteil dieser Art der Dateiorganisation liegt schlicht und ergreifend darin, dass die Adresse einer Zelle in einer Datenbank sehr schnell errechnet werden kann, da jedes Feld eine definierte Länge aufweist, denn

Startadresse + Kopflänge + (Datensatznummer - 1) * Datensatzlänge

ergibt die Speicheradresse des ersten Zeichens des ersten Feldes des gewünschten Datensatzes. Und diesen Algorithmus kann Excel nicht bieten, da die Zellinhalte variabler Länge sind. Auch die Datentypstringenz ist in Excel nicht gewährleistet.

Excel erfüllt diese ersten Anforderungen nicht. Eine Zeile in einer Excel-Tabelle kann zwar mit einem Zahlenformat versehen werden, hat allerdings einer Zeichenkette als Zellinhalt nichts entgegenzusetzen. Das Zahlenformat der Zelle sagt ja bekanntlich nichts anderes als: Wenn der betreffende Zellinhalt numerisch sein soll, wird das vereinbarte Zahlenformat zur Anwendung gebracht. Nicht mehr und nicht weniger, denn darin liegt zu einem großen Teil auch die Flexibilität einer Excel-Tabelle begründet.

Datenbanken verfügen über Automatismen, die es erlauben, Relationen zwischen einzelnen Datenbanktabellen zu bilden.

Relationen

Unsere erste Tabelle enthält zum Beispiel Firmendaten, etwa korrekte Firmierung, Anschrift, Telefonnummer, die letzte Bilanzsumme, Mitarbeiterzahl und ähnliches. Auf der anderen Seite pflegen wir personenbezogene Daten wie Name, Position, Durchwahl, Titel, Mail-Adresse sowie Kontaktnotizen. Eine Datenbank wäre in zwei Tabellen organisiert, die Firmen- und Personendaten getrennt aufbewahrt. Ein beiden Tabellen gemeinsames Feld würde dafür sorgen, dass in einem Datensatz sowohl alle personenbezogenen als auch firmenbezogenen Daten verfügbar wären. Das wäre eine so genannte Relation.

Das kann Excel so nicht. Entweder wir führen die Daten in einer Tabelle und erkaufen uns damit redundante Firmendaten, oder wir führen Personen- und Firmendaten in zwei Tabellen. Somit hätten wir zwar Redundanzfreiheit, wären allerdings auf Formeln (SVERWEIS) oder VBA-Code angewiesen.

Datenbanken verstehen SQL.

SQL (Structured Query Language)

SQL (Structured Query Language) ist ein Standard im wahrsten Sinne des Wortes, der in allen Datenbanken funktioniert. Es handelt sich hierbei um eine Datenbanksprache, die aus vergleichsweise wenigen Schlüsselworten besteht, aber dennoch (man mag mir den Hyperlativ verzeihen) *komplexe* Analysen in einer Zeile erlaubt. SQL ist zusammen mit Datenbankzugriffen einfach unverzichtbar.

Das Excel zugrunde liegende Objektmodell versteht natürlich kein SQL. Es gäbe natürlich die Möglichkeit, über die Datenbankobjektmodelle DAO (Data Access Objects) oder ADO (ActiveX Data Objects) über ISAM-Treiber (Indexed Sequential Access Method) sogar auf Excel-Arbeitsmap-pen zuzugreifen, was aber zwangsläufig zu Konflikten bezüglich der Schreibrechte führen würde. Außerdem gehören diese Überlegungen in die Kategorie Mumpitz. Excel-Tabellen sind keine Datenbanktabellen.

SQL gehört zum Handwerkszeug eines Excel-VBA-Entwicklers. Solche Tools wie Query sind zwar für einmalige Datenbankzugriffe ganz nützlich, sie decken allerdings die Erfordernisse der professionellen Anwendungsentwicklung mit Datenbankanbindung in Excel-VBA nicht ab.

Datenbanken verarbeiten Stored Procedures und Views.

Stimmt diese Behauptung? Das hängt davon ab, ob man Access-Datenbanken als Datenbanken bezeichnet oder nicht.

Stored Procedure

Ein DBMS (Database Management System) wie beispielsweise der SQL-Server oder ORACLE sind in der Lage, so genannte Stored Procedures, letztendlich kleine Programme, auf dem Server zu speichern und aufzurufen. Diese Programme bestehen aus SQL-Anweisungen, die auch über Kontrollstrukturen verfügen können. Der Witz bei diesen Stored Procedures liegt einfach darin, dass sie auf dem Server ablaufen und zur Ermittlung der tatsächlich zurückzugebenden Daten somit keinerlei Netzbelastung zur Folge haben. Erst der Transport der Ergebnisdaten der Analyse beansprucht das Netz.

12.1 Data Access Objects

Die DAO ist ein Veteran, gleichwohl aber in der Visual Basic Welt die Standardschnittstelle zu Datenbanken schlechthin. Den Verlautbarungen Microsofts zufolge wird die DAO nebst den anderen Komponenten nicht mehr weiterentwickelt. ADO heißt das neue Zauberwort. Aufgrund ihrer weiten Verbreitung sollten wir dennoch einen Blick auf die DAO werfen.

Zunächst benötigt Ihr Projekt einen Verweis auf die DAO-Bibliothek. Doch zu welcher? Um auf Access 2000 Datenbanken zugreifen zu können, muss es die DAO 3.6 sein. Bei Access 97 Datenbanken muss es sich mindestens um Version 3.5 handeln.

DAO heißt auf deutsch Datenzugriffsobjekte. Damit ist gemeint, dass Datenbanken über ein Objektmodell erschließbar sind. Ebenso wie das Excel-Objektmodell auf der einen Seite sehr komplex ist, verfügt auch das DAO-Objektmodell über eine Reihe von Objekten, die natürlich alle ihre durchaus verständlichen Eigenschaften und Methoden rechtfertigen. Doch hier setzt die gute Nachricht ein, denn wir werden ebenso wie beim

Datenzugriff in Excel nur die wichtigen Objekte beleuchten und den Rest über SQL abdecken.

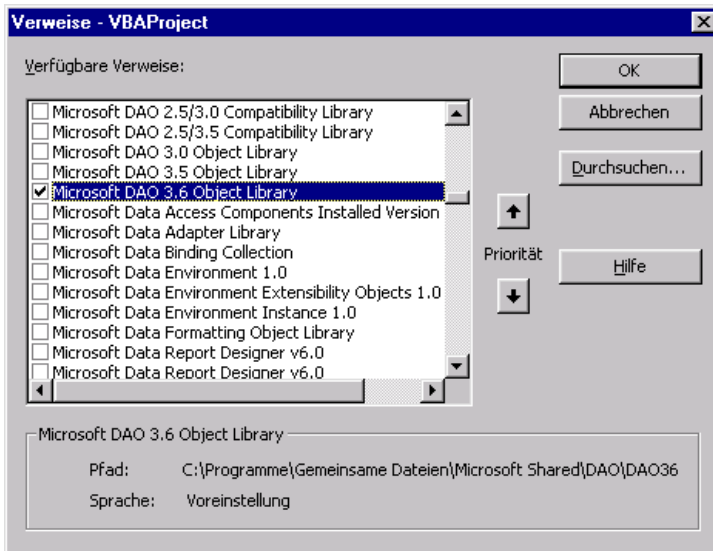


Abbildung 12.1: Einrichten eines Verweises zur DAO

Schauen wir uns zuerst generell die Architektur der Datenbankzugriffe unter Windows an, die in 12.2 abgebildet ist.

Wir sehen, dass zwischen unserer VBA-Bibliothek und den Datenbanken noch eine Reihe anderer Schichten erscheinen.

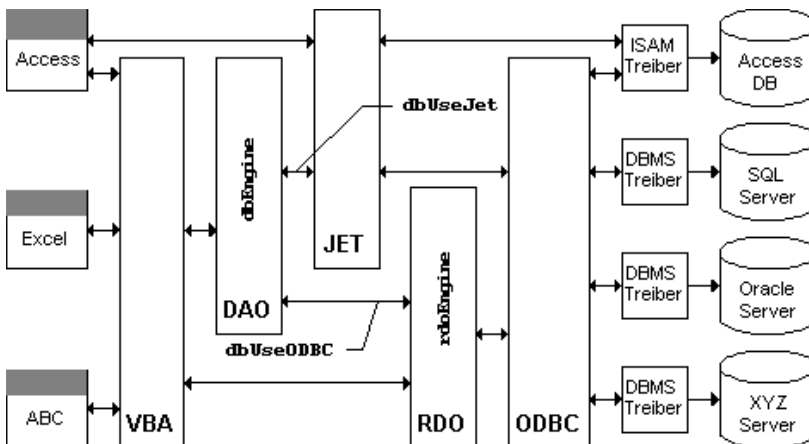


Abbildung 12.2: Datenbankzugriffe über DAO

DAO - Data Access Objects

Die DAO ist eine Objektbibliothek, die den Zugriff auf Datenbanken und deren Komponenten erlaubt.

JET - Joint Engine Technologies

Die JET ist eine für uns Entwickler unsichtbare Bibliothek, die speziell für Access-Datenbanken und sonstige über ISAM-Treiber (Indexed Sequential Access Mode) erschließbare Datenbanken entwickelt wurde. Selbst das Programm Access bedient sich der JET, um auf die Access-Datenbanken zuzugreifen.

RDO - Remote Data Objects

Der Zugriff auf Datenbanken, die auf einem anderen Rechner liegen, erfolgt unter Zuhilfenahme dieser für uns ebenfalls unsichtbaren Bibliothek. Interessanterweise heißt dieser Weg aus Sicht der DAO ODBC-Direct, obwohl er ja keineswegs direkt ist.

ODBC - Open Database Connectivity

Die ODBC ist eine Schnittstelle, die Anwendungsprogrammen und Datenbanken eine universelle Schnittstelle bietet.

12.1.1 Wege des Datenzugriffs

Sehen wir einmal von dem direkten Weg zwischen Access und der JET ab, so verlaufen die Wege des Datenzugriffs über die DAO.

Der Weg DAO – JET – Datenbank

Dieser Weg ist der Standard beim Einsatz von Access-Datenbanken. Nicht ganz den Kern treffend heißen alle hierüber erschließbaren Datenbanken JET- oder ISAM-Datenbanken, obwohl sie eigentlich recht unterschiedlicher Natur sind.

So sind typische ISAM-Datenbanken in Bezug auf ihre Tabellen üblicherweise dateiorientiert. Eine Datenbank, die aus drei einzelnen Tabellen aufgebaut ist, besteht auch aus drei einzelnen Dateien. Indizes werden ebenfalls in eigenen Dateien abgelegt, Relationen und Abfragen hingegen existieren nur zur Laufzeit. Typische Vertreter dieser Kategorie sind dBASE, Paradox, FoxPro oder Btrieve.

So sah es in den 80er Jahren auf PCs aus. Eine Datenbank, obwohl so genannt, war eigentlich nur eine Tabelle. Wenn man von einer Datenbank im heutigen Sinne sprechen konnte, dann nur auf einer virtuellen Ebene. Erst in den 90ern begannen sich Datenbanken heutiger Prägung auf PCs niederzulassen. Eine solche, moderne Datenbank besteht aus einer oder

mehreren (Basis-) Tabellen, Indizes und Abfragen, aber auch so genannten Stored Procedures, gewissermaßen dezentrale Programmteile, die Aufgaben bereits in der Datenbank ausführen können. Letztere sind typische Client/Server Komponenten, die vor allem der Netzentlastung dienen. Eine Access-Datenbank liegt irgendwo in der Mitte zwischen typischen Vertretern der ISAM-Datenbanken und richtigen Client/Server Datenbanken.

Der Weg DAO – JET – ODBC – Datenbank

Dieser heutzutage eher unbedeutende Weg erlaubte den Datenzugriff auf ODBC-Datenbanken wie SQL-Server, ORACLE oder SYBASE. Da die JET nicht nur eine passive Schnittstelle darstellt, sondern einiges an Intelligenz im Umgang mit den dahinter liegenden aufzuweisen hat, stellte sie sich als Flaschenhals im Umgang mit diesen Datenbanken heraus. Wenn wir den Ausdruck Intelligenz durch »Sie wollte mitreden« übersetzen, tritt der Charakter des Flaschenhalses stärker zutage. Sie betrachtete die Datenbanken durch die ISAM-Brille, denn dafür war sie schließlich geschaffen worden. Sie bremste diese Verbindungen mächtig aus und zeigte sich auch nicht offen für Fähigkeiten, über die zum Beispiel die Datenbanken, aber nicht sie, verfügten.

Der Weg DAO – RDO – ODBC – Datenbank

Es gab zwei Probleme mit Datenbanksystemen wie SQL-Server oder ORACLE und unserer DAO. Das erste Problem lag in der eingeschränkten Eignung der JET zum Zugriff auf diese Datenbanken. Das zweite war darin begründet, dass solche Datenbanken üblicherweise auf eigenen Servern liegen, und der Zugriff somit über mindestens zwei ODBC-Schnittstellen erfolgen musste, nämlich auf Client- und auf Serverseite. Diesen Anforderungen begegnete man mit einer neuen Schnittstelle, in der die Verteilung von Client und Server auf verschiedenen Rechnern mit dem Attribut Remote (fern) Rechnung getragen wurde.

Die neue Bibliothek hieß Remote Data Objects und schaltete sich zwischen die DAO und die ODBC, wodurch der Flaschenhals JET der Vergangenheit angehörte.

12.1.2 Das DAO-Objektmodell

Die folgende Abbildung 12.3 zeigt die Objekte und Auflistungen des DAO-Objektmodells, die man benötigt, um erfolgreich auf Datenbanken zuzugreifen. Im Gegensatz zum ADO-Objektmodell, dem wir in Kapitel 12.2 begegnen werden, sind alle Objekte in einem Baum mit klaren Abhängigkeiten zu finden.

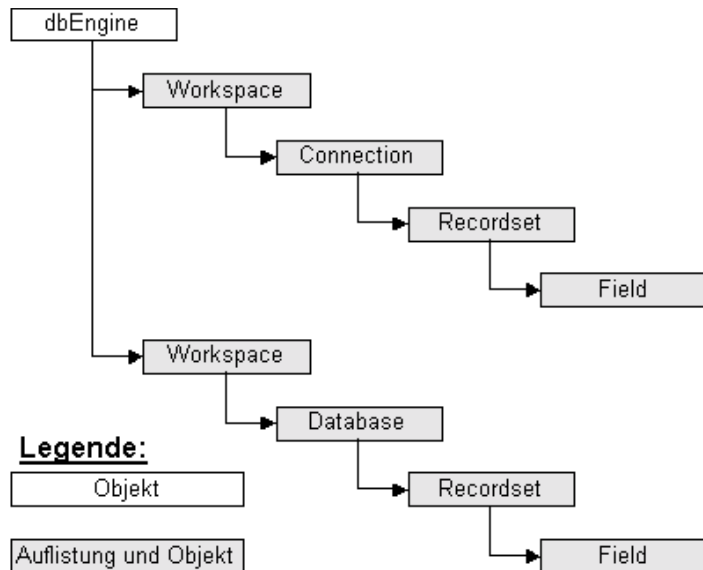


Abbildung 12.3: Auszug aus dem DAO-Objektmodell

Hierin spiegeln sich auch unsere beiden Wege des Datenzugriffs wieder, denn *in der Regel* steht der Weg über das Connection-Objekt für die Beteiligung der ODBC und der Weg über das Database-Objekt für JET-Zugriffe.

12.1.3 Zugriff auf eine JET-Datenbank

Wie Abbildung 12.3 zeigt, benötigen wir ein Workspace-Objekt, um auf eine Datenbank zugreifen zu können. Eine Besonderheit der DAO ist es, dass Sie für JET-Zugriffe bereits einen Workspace unter dem Index 0 bereithält. Um eine Datenbank zu öffnen, können wir schreiben:

```
Dim db As Database
Set db = dbEngine.Workspaces(0).OpenDatabase(...)
```

Wenn sich in der Deklarationszeile hinter As die ListBox mit den verfügbaren Objekten öffnet, merken Sie schnell, ob Sie den Verweis vergessen haben, denn der Eintrag Database bleibt dann aus.

Ich möchte Ihnen hier eine Methode namens GetDB präsentieren, die eine geöffnete Datenbank zurückgibt oder einen aussagekräftigen Fehler produziert, falls das aus irgendeinem Grund nicht möglich sein sollte:

```

Public Function GetDB(ByVal strPath As String, _
                    ByVal strFile As String, _
                    Optional ByVal strPassword As String) As Database
    ' Gibt die Datenbank geöffnet zurück
    'Argumente:
    '   strPath:      Pfad der Datenbank
    '   strFile:      Name der Datenbank
    '   strPassword:  Passwort der Datenbank
    Dim strError As String      'Fehlertext
    On Error GoTo ErrorHandler
    If Right(strPath, 1) <> "\" Then strPath = strPath & "\"
    'Prüfen, ob Verzeichnis und Datei vorhanden sind
    If Dir(strPath, vbDirectory) = "" Then
        Err.Raise 76
    End If
    If Dir(strPath & strFile) = "" Then
        Err.Raise 53
    End If
    'Datenbank öffnen
    Set GetDB = DBEngine.Workspaces(0).OpenDatabase( _
        strPath & strFile, False, False, _
        ";PWD=" & strPassword)

    Exit Function

ErrorHandler:
    ...
    Err.Raise Err.Number + vbObjectError ...
End Function

```

Unter Angabe der Arguments `strPassword` können Sie eine passwort-geschützte Datenbank öffnen. Wird kein Passwort benötigt, so lassen Sie das Argument einfach weg. Der Aufruf zum Öffnen der beiliegenden `Rechnung.mdb` sieht so aus:

```

Dim db As Database
Dim cDB as New clsDB
Set db = cDB.GetDB(ThisWorkbook.Path, "Rechnung.mdb")

```

Da ausgehend von einem Database-Objekt oder einem Connection-Objekt der Zugriff auf Recordsets im Prinzip gleich ist, werden wir uns zuerst den Zugriff auf eine ODBC-Datenbank anschauen. Danach kümmern wir uns um Recordsets.

12.1.4 Zugriff auf eine ODBC-Datenbank

Abbildung 12.2, die generell den Datenbankzugriff über die DAO skizziert, vereinfacht insofern, da es so aussieht, als sei der SQL-Server eine Datenbank ähnlich einer Access-Datenbank und womöglich noch auf demselben Rechner installiert. Um die generellen Wege über die verwendeten Bibliotheken zu skizzieren, ist das in Ordnung. Aber Abbildung 12.4 verdeutlicht die Wege anhand eines Database Management Systems (DBMS) wie dem SQL-Server etwas detaillierter.

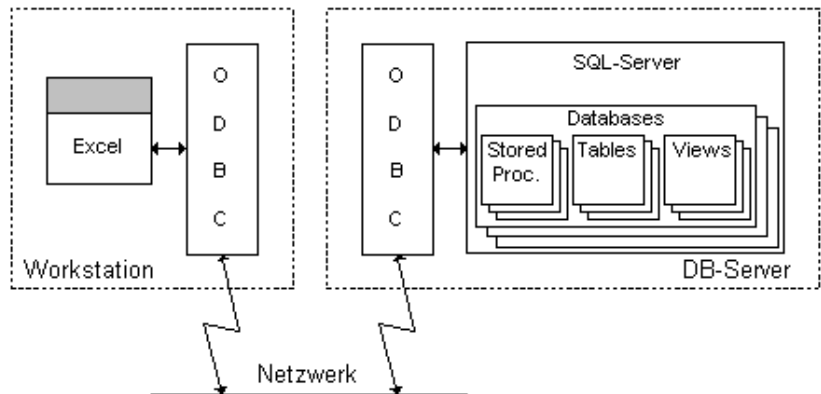


Abbildung 12.4: Zugriff auf SQL-Server

An dieser Abbildung können wir ersehen, dass unsere ODBC gar nicht mit der Database Management System SQL-Server direkt verhandelt, sondern über das Netz und mindestens eine zwischengeschaltete ODBC mit dem SQL-Server verbunden wird. Außerdem haben wir nie einen unmittelbaren Zugriff auf die Daten, denn alles läuft über die Mechanismen des DBMS.

Unser Ansprechpartner ist in jedem Fall die lokale ODBC, die wir mit allen Informationen versorgen müssen, sodass sie zuerst einmal den passenden DB-Server findet und uns dort als authentifizierten Anwender anmelden kann. All diese Informationen können in einem Connect-String abgelegt werden, der zur ODBC transportiert wird. Man kann sich allerdings auch mit dem ODBC Data Source Manager eine Datenbank-Verbindung per Assistenten zusammensetzen lassen und diesen im Connect-String als DSN-Attribut angeben.

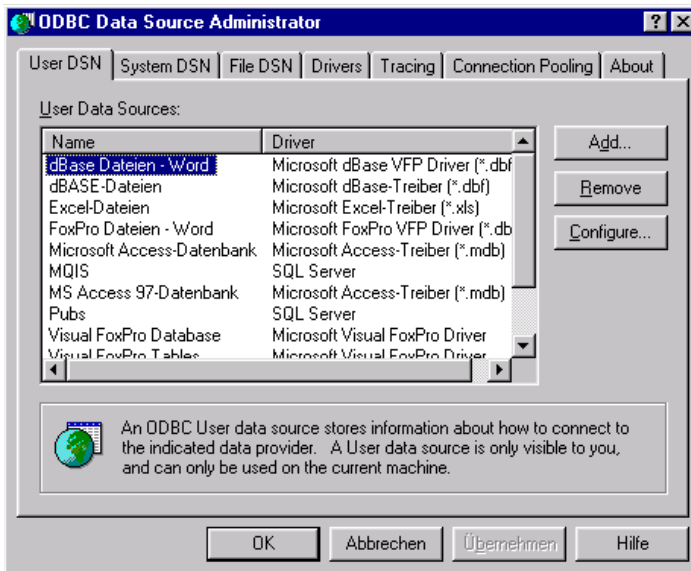


Abbildung 12.5: ODBC Data Source Manager

Man kann in diesen DSNs alle Informationen inklusive UID (User ID) und PWD (Password) unterbringen, sodass der Verbindungsaufbau so codiert ist:

```
Dim wsODBC As Workspace
Dim cn As Connection
On Error GoTo errHandler
Set wsODBC = DBEngine.CreateWorkspace("", "Klaus", "", _
                                         dbUseODBC)
Set cn = wsODBC.OpenConnection("", dbDriverNoPrompt, _
                                False, "ODBC;DSN=MyDSN")
```

Hier wird also zuerst mit der `CreateWorkspace`-Methode des `DBEngine`-Objekts ein ODBC-Workspace erzeugt. Die Richtung ODBC wird durch das letzte Argument `dbUseODBC` festgelegt. Die Connection selbst wird dann ausgehend von diesem Workspace-Objekt über `OpenConnection` eingeleitet. Im ersten Argument kann ein Name für diese Connection vergeben werden. Da wir aber einen Objektverweis auf `cn` erzeugen, können wir uns den Namen sparen. Das zweite Argument legt fest, wie sich die ODBC bei Unklarheiten verhalten soll. `dbDriverNoPrompt` meint, dass sie das Ganze vergessen soll, wenn es nicht funktioniert. Mit `dbDriverPrompt` schaltet sich der ODBC Data Source Manager ein und bietet Unterstützung bei der Auswahl der DSN an und fragt nach UID und PWD. Diese Option sollten Sie zu Testzwecken wählen, aber keinesfalls

dem Anwender den ODBC Data Source Manager vor die Nase setzen. Mit dem dritten Argument steuern Sie, ob Sie nur Lesezugriff (`True`) oder auch Schreibrechte (`False`) anfordern. Das vierte und letzte Argument enthält den Connect-String, der mit »ODBC;« beginnen muss. Die einzelnen Parameter werden mit »;« voneinander getrennt.

Praxis in vielen Unternehmen ist es jedoch, UID und PWD nicht in der DSN unterzubringen, sondern separat in diesem Connect-String anzugeben:

```
Set cn = wsODBC.OpenConnection("", dbDriverNoPrompt, _  
    False, "ODBC;DSN=MyDSN;UID=Klaus;PWD=IchBins")
```

Danach müssten Sie über eine Verbindung zur Datenbank verfügen.

12.1.5 Zugriff auf Recordsets

Wenn Sie ein gültiges Database- oder Connection-Objekt in Händen halten, so ist der Rest im Wesentlichen identisch. Zwar bestehen einige Datenbanken und Datenbank-Managementsysteme darauf, Ihren eigenen SQL-Dialekt zu pflegen, doch diese Einfärbungen betreffen in der Regel nicht den Teil von SQL, den wir in Programmen verwenden.

Nun sollten wir uns einmal über den Begriff des Recordsets einigen. Ein Recordset ist eine Gruppe von Records, also Datensätzen. Ein Datensatz *kann* zusammengesetzt sein aus:

- ▼ einem Feld oder mehreren existierenden Feldern einer Tabelle,
- ▼ eventuell verknüpft über eine Relation zu Daten in anderen Tabellen, sodass beide Teile aussehen wie ein Datensatz
- ▼ und bzw. oder künstlichen Feldern, die aus Aggregationen realer oder künstlicher Felder bestehen können

Und in Datensatzrichtung können von null bis was weiß ich wie vielen Datensätzen daran beteiligt sein. Daneben sind Kreuztabellenauswertungen in Form der Pivot-Tabellen möglich und so weiter. Alles, was sich in einer oder gar mehreren Datenbanken befindet, lässt sich nach irgendeinem Kriterium miteinander verbinden oder verdichten. Ihrer Phantasie sind nahezu keine Grenzen gesetzt. Das Zauberwort heißt SQL. In diesem Abschnitt erzeugen wir ein paar einfache Recordsets, ohne uns um die Details der SQL-Anweisungen zu kümmern. In Kapitel 12.3 werden wir uns SQL etwas näher anschauen.

Das Recordset-Objekt ist unmittelbar hinter dem Connection- oder dem Database-Objekt angesiedelt und wird wie folgt mit einer `OpenRecordset`-Methode erzeugt:

```
Dim rs As Recordset
```

```
Dim strSQL as String
```

```
strSQL = "SELECT * FROM Customers WHERE PLZ = 12345"
```

```
Set rs = db.OpenRecordset(strSQL, dbOpenSnapshot)
```

oder

```
Set rs = cn.OpenRecordset(strSQL, dbOpenSnapshot)
```

Der Variablen `strSQL` wird ein SQL-Statement zugewiesen, welches danach der `OpenRecordset`-Methode übergeben wird. Die Variable ist im Prinzip verzichtbar, es ist jedoch zweifellos übersichtlicher, das SQL-Statement in einer Zeile (oder mehreren) zu erzeugen. Mit dem zweiten Argument legen Sie fest, welchen Cursortyp sie möchten:

Konstante	Bedeutung
<code>dbOpenTable</code>	Eine ganze Tabelle (nur JET-Datenbanken)
<code>dbOpenDynamic</code>	Dynamisches, in beiden Richtungen bewegbares Recordset (nur ODBC-Datenbanken)
<code>dbOpenDynaset</code>	Dynamisches, in beiden Richtungen bewegbares Recordset (nur JET-Datenbanken)
<code>dbOpenSnapshot</code>	Statisches, in beiden Richtungen bewegbares Recordset
<code>dbOpenForwardOnly</code>	Statisches, nur vorwärts bewegbares Recordset

Ich verwende grundsätzlich nur statische Recordsets, da ich Datensätze nur über separate Anweisungen verändere, hinzufüge oder lösche. Dies erfordert zwar mehr Phantasie und Disziplin im Code, ermöglicht andererseits durch die Entkopplung der Anwendung von den Daten eine wesentlich höhere Performance und blockiert andere Anwendungen nicht. Das liegt auch voll im Trend der zustandslosen Datenschicht, von denen auch im Zusammenhang mit Distributed interNet Applications (DNA) ständig die Rede ist. In Abschnitt 12.5 werden wir kurz auf diese Technik (Technologie?) eingehen.

12.1.6 Zugriff auf Felder

Ein Recordset-Object verfügt über eine Fields-Collection, auf deren Elemente mit dem Feldnamen oder einem Null-basierten Index zugegriffen werden kann:

```
strName = rs.Fields("Name").Value
```

```
strName = rs.Fields(1).Value
```

Ebenso wie beim Range-Objekt kann man die Value-Eigenschaft weglassen, sollte man aber nicht. Microsoft hat angekündigt, in Visual Basic 7

die Option `Strict` einzuführen, mit der eine typgerechte Datenübergabe Einzug halten wird. Dadurch werden vermutlich Konvertierungs- und Prüfungsroutinen ausgeschaltet und der Code wird schneller und natürlich stabiler. Die folgenden Zeilen würden dann erbarmungslos den Fehler 13 provozieren:

```
Dim strName As String
strName = rngKunden.Cells(iRow, iColKunde)
strName = rsKunden.Fields("Name")
```

Die `Fields`-Collection verfügt natürlich auch über eine `Count`-Eigenschaft, die uns die Anzahl der verfügbaren Felder mitteilt, um sie beispielsweise in einer `For-Next`-Schleife abzuarbeiten. Im folgenden Beispiel werden die Feldnamen des Recordsets in die erste Tabellenzeile eingetragen:

```
For iField = 0 To rs.Fields.Count - 1
    rng.Cells(1, iField + 1).Value = rs.Fields(iField).Name
Next
```

12.2 ActiveX Data Objects

Die Akronyme DAO und ADO wecken den Gedanken, dass es sich bei dieser Technik um alten Wein in neuen Schläuchen handelt, ähnlich dem Begriffspaar OLE und ActiveX. Doch hinter ADO steht nicht die Marketing-Abteilung von Microsoft, sondern die Entwicklung.

Gegenüber Abbildung 12.2 zeigt die Abbildung 12.6 auf den ersten Blick gewaltige Veränderungen:

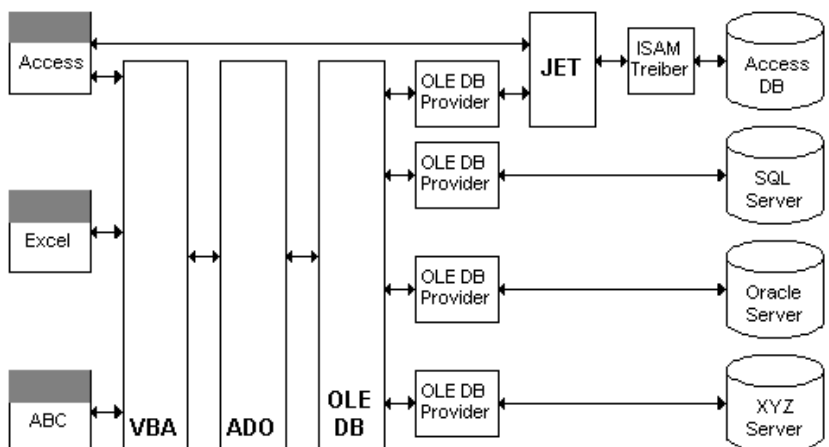


Abbildung 12.6: Datenbankzugriffe über ADO

Die ADO präsentiert sich uns als alleiniger Ansprechpartner und hängt ihrerseits nur an der OLE DB. Die OLE DB wiederum, der Nachfolger der ODBC, verhandelt nur über OLE DB-Provider mit den Daten. Das ist das Ergebnis eines Re-Designs, bei dem man sich offensichtlich von der Last der Historie befreit hat. Dem gegenüber sieht man der DAO in Verbindung mit der ODBC schon an, dass es sich um gewachsene Strukturen handelt: Hier eine Erweiterung, da ein neuer Weg. Lediglich die gute alte JET hat diesen Prozess überlebt.

12.2.1 Das ADO-Objektmodell

Ein Objektmodell mit einer Hierarchie, wie wir sie von der DAO her kennen, gibt es nicht. Abbildung 12.7 zeigt das vollständige ADO-Objektmodell.

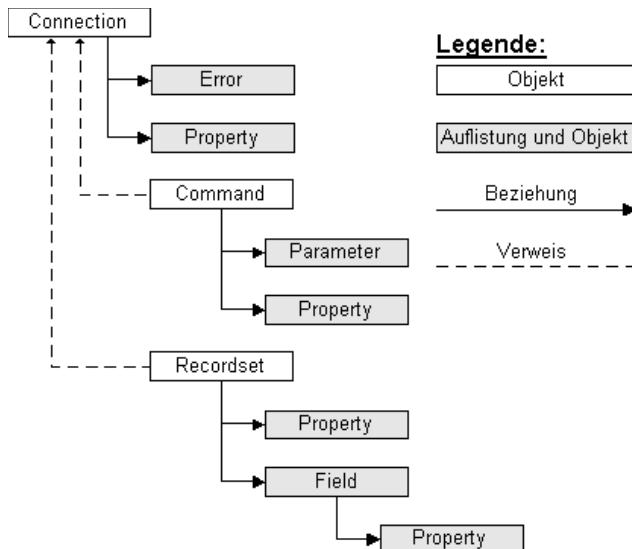


Abbildung 12.7: Das ADO-Objektmodell

Zwischen dem Recordset-Objekt und der Fields-Auflistung beispielsweise existiert eine Beziehung, wie sie auch von anderen Objektketten bekannt ist:

```
Recordset.Fields(...)
```

Connection-, Command- und Recordset-Objekt werden instantiiert, wie wir es von unseren selbst entworfenen Klassen kennen:

```
Dim cn As ADODB.Connection  
Set cn = New ADODB.Connection
```

oder

```
Dim cn As New ADODB.Connection
```

Sie haben nun mehrere Möglichkeiten, die Verbindung zur Datenbank aufzubauen. Die übersichtlichste Vorgehensweise ist wohl die, der **Provider-Eigenschaft** des **Connection-Objekt** zuerst den benötigten **OLE DB Provider** anzugeben und anschließend die **ConnectionString-Eigenschaft** mit allen weiteren Informationen zu versehen:

```
cn.Provider = "Microsoft.Jet.OLEDB.4.0"  
cn.ConnectionString = "Data Source=" & _  
    ThisWorkbook.Path & "\Rechnung.mdb"
```

Danach kann die Verbindung schlicht und ergreifend mit der **Open-Methode** geöffnet werden:

```
cn.Open
```

Statt den **OLE DB Provider** separat zu übergeben, können Liebhaber langer Zeilen diese Information auch in den **ConnectionString** hineinpacken oder als **ConnectionString-Argument** der **Open-Methode** folgen lassen:

```
cn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _  
    "Data Source=" & ThisWorkbook.Path & "\Rech..."
```

Unter der **DAO** wäre es mit **cn.OpenRecordset** oder etwas Ähnlichem weitergegangen. Die **ADO** hingegen verfährt mit dem **Recordset-Objekt** in der Art, wie ein **Connection-Objekt** erzeugt und belebt wird:

```
Dim rs As ADODB.Recordset  
Set rs = New ADODB.Recordset
```

oder

```
Dim rs As New ADODB.Recordset
```

Danach können wir – wie auch bei der **Connection** – die erforderlichen Parameter des **Recordset-Objekts** in Form von **Eigenschaften** übergeben und das **Recordset** anschließend mit der **Open-Methode** öffnen:

```
rs.ActiveConnection = cn  
rs.CursorType = adOpenStatic  
rs.Source = "SELECT * FROM Customers"  
rs.Open
```

Da SQL-Statements in der Regel schon ein bisschen größer ausfallen als dieses Beispiel suggeriert, halte ich hierfür eine `Private Variable` namens `strSQL` bereit, die das SQL-Statement aufnimmt. Gemeinsam mit der `Connection`-Objektvariablen und dem `Cursortyp` ergibt sich somit eine überschaubare Anzahl von Zeichen, die der `Open`-Methode auch nachgestellt werden können:

```
strSQL = "SELECT * FROM Customers"  
rs.Open strSQL, cn, adOpenStatic
```

Der Zugriff auf die Felder des Recordsets wiederum erfolgt nach bewährtem Muster, wie es auch bei der DAO üblich ist:

```
strName = rsKunden.Fields("Name").Value
```

Ein geöffnetes Recordset wird über eine erneute `Open`-Methode nicht erfreut sein. Die DAO lässt das zu, die ADO nicht. Sofern es sich also um ein Recordset handelt, das möglicherweise bereits geöffnet ist, müssen Sie es vorher schließen. Die `State`-Eigenschaft teilt uns mit, in welchem Zustand sich das Recordset gerade befindet.

Die `State`-Eigenschaft des Recordset-Objekts kennt folgende Zustände:

Wert und Konstante	Bedeutung
0 – <code>adStateClosed</code>	Recordset ist geschlossen
1 – <code>adStateOpen</code>	Recordset ist geöffnet
2 – <code>adStateConnecting</code>	Recordset stellt gerade eine Verbindung zur Datenquelle her
4 – <code>adStateExecuting</code>	Recordset führt gerade eine Operation aus
8 – <code>adStateFetching</code>	Recordset lädt gerade Daten

Die Werte dieser Konstanten deuten es schon an: Wir müssen maskieren, um eine Aussage über den Status der Recordsets zu erhalten, denn ein Recordset kann gleichzeitig geöffnet sein *und* Daten nachladen.

```
If (rs.State And adStateOpen) = adStateOpen Then rs.Close
```

12.2.2 Navigation in einem Recordset

Wenn Sie es gewohnt sind, in Excel-Tabellen mit Zeigern in Spalten- und Zeilenrichtung festzulegen, wo sie was auslesen oder schreiben wollen, so müssen Sie hier ein wenig umdenken. In modernen Datenbanken interessiert man sich nicht für einen absoluten Datensatzzeiger, man bewegt sich stattdessen durch die Datensätze, indem man zum ersten, vorangehen-

den, nächsten oder letzten Datensatz springt. Die Methoden hierzu lauten `MoveFirst`, `MovePrevious`, `MoveNext` und `MoveLast`.

In dem folgenden Beispiel werden Namen und Vornamen aus einem Recordset in eine Excel-Tabelle geschrieben:

```
For iRow = 2 to rs.RecordCount + 1
    sht.Cells(iRow, 1).Value = rs.Fields("Name").Value
    sht.Cells(iRow, 2).Value = rs.Fields("Firstname").Value
    rs.MoveNext
Next
```

Das sieht auf den ersten Blick ganz gut aus, ist aber grottenfalsch. JET und OLE DB laden ein Recordset nicht notwendigerweise vollständig, sondern in Blöcken (die JET à 4kByte, den OLE DB-Wert kenne ich nicht). Passen alle Daten in diesen gewöhnlich Page genannten Puffer, so ist alles in Ordnung. Wenn aber nicht, dann haben wir ein Problem. Die Eigenschaft `RecordCount` gibt immer nur die Anzahl der bislang durch den Puffer gewanderten Datensätze zurück. Da die `For-Next`-Schleife jedoch eine Veränderung ihrer oberen Grenze nicht registriert, werden Sie später nur einen Teil der angeforderten Daten in der Excel-Tabelle vorfinden. Was tun?

Die zweitbeste Lösung ist, das Recordset vollständig zu populieren, wie das so schön heißt:

```
rs.Open
rs.MoveLast
rs.MoveFirst
For iRow = 2 to rs.RecordCount + 1
```

Gelegentlich führt kein Weg daran vorbei, die Anzahl der Datensätze zu ermitteln. In diesem Fall sollte man sich allerdings darauf beschränken, nur das kleinste Feld des gewünschten Datenbereichs zu laden, um die `RecordCount`-Eigenschaft zu ermitteln.

In der Regel ist es jedoch nicht erforderlich, das Recordset vollständig zu populieren. Wir müssen nur statt der deterministischen `For-Next`-Schleife eine an einen Wert gekoppelte `Do-Loop`-Schleife einsetzen. Hierzu bietet sich die `EOF`-Eigenschaft des Recordsets an, die ein `True` zurückgibt, sobald das End of File (EOF) erreicht wird:

```
iRow = 1
Do Until rs.EOF
    iRow = iRow + 1
    sht.Cells(iRow, 1).Value = rs.Fields("Name").Value
```



```
sht.Cells(iRow, 2).Value = rs.Fields("Firstname").Value  
rs.MoveNext
```

Loop

Dies verhindert auch, dass versucht wird, auf ein leeres Recordset zuzugreifen.

12.3 ANSI-SQL

SQL (Standard oder auch Structured Query Language) ist eine Datenbanksprache, die Anfang der 70er Jahre entstand. Die Standardisierung begann 1982 und führte zu einer Reihe von Normen: SQL86 im Jahre 1986, SQL89 1989, SQL92 oder auch SQL2 anno 1992. Der SQL3-Standard wurde im November 1999 in Heidelberg von einem Fachgremium faktisch verabschiedet. Mit der *Formsache*, der offiziellen Verabschiedung rechnet das DIN Mitte 2000 (www.din.de). Es wird ferner bereits angedroht, dass SQL alle vier Jahre angepasst werden soll. Man achtet also allerorten darauf, dass wir nicht einrosten. Wir können jedoch davon ausgehen, dass die Standardsprachelemente, die wir hier verwenden, nicht angetastet werden.

SQL besteht aus den drei Komponenten

- ▼ DCL – Data Control Language (Rechte etc.),
- ▼ DDL – Data Definition Language (Strukturen) und
- ▼ DML – Data Manipulation Language (Abfragen und Veränderungen).

Mit DCL haben wir als Entwickler eigentlich nichts zu tun.

Sprachelemente aus DDL kommen gelegentlich vor, wenn wir per Code Strukturen an bereits im Einsatz befindlichen Datenbanken anpassen müssen, also bei Updates. In großen Database Management Systemen kann es gelegentlich vorkommen, dass Sie zwar Eigentümer eines TableSpace oder NameSpace werden, aber keinerlei Administrationswerkzeuge erhalten. Dann bleibt ihnen nur DDL, um Tabellen anzulegen. Wenn man Datenbank-Tools wie Access oder den Enterprise Manager des SQL-Servers gewöhnt ist, so kann man das damit vergleichen, in einer geschlossenen Dose Kekse zu sortieren; höchst unerfreulich.

In Programmen arbeiten wir, von Software-Updates einmal abgesehen, nur mit der Data Manipulation Language. Das ist die gute Nachricht. Auch, dass die Anzahl der darin verwendeten Schlüsselwörter eine sehr überschaubare Größenordnung hat, ist noch Teil dieser guten Nachricht. Die unglaubliche Komplexität aber, die solche SQL-Statements annehmen können, ist aber dazu geeignet, uns Entwickler gelegentlich in die Ver-

zweiflung zu treiben. Aber es liegt an uns selbst, ob wir uns darauf einlassen oder nicht.

Bis etwa Ende 1998 war ich ständig auf Rekordjagd und überraschte meine Gattin allabendlich mit Meldungen wie: »Heute schrieb ich einen SQL, der war viereinhalb Meter lang und hatte drei Sub-Selects.« Dann kam der Moment, als ein Kunde nach langer Vorrede (Wie geht's der Familie? Hund gesund? Dabei haben wir keinen Hund!) damit herausrückte, dass ein Paar Tabellen im PPS, die in der Beauftragung als völlig statisch vereinbart waren, eine kleine, *unwesentliche* Änderung erfahren haben. Den Rest können Sie sich denken.

Diese Erfahrung brachte mich dazu, wesentlich mehr Energie in das intelligente Tabellen- und vor allem Klassendesign zu stecken. Und wenn mir heute ein SQL-Statement begegnet, das nicht mehr vollständig und in einer Zeile darstellbar ist, so suche ich sofort nach dem Fehler im Datenbank- oder Klassendesign.

Nach dieser kleinen Beichte sollten wir aber endlich anfangen.

Sie werden in diesem Abschnitt einen kleinen Einblick in SQL vorfinden. Damit sind Sie in der Lage, kleinere Probleme zu lösen, wie wir anhand des Beispiels Rechnung.xls aus Kapitel 10 sehen werden. Wenn Sie diese Sprachelemente sicher im Griff haben, können Sie diese sukzessive erweitern. Die MSDN bietet viel Anschauungsmaterial.

Um ein wenig üben zu können, öffnen Sie am besten die beiliegende Datenbank Rechnung.mdb mit Access und erstellen eine neue Abfrage in der Entwurfsansicht. Den Dialog Tabellen einfügen schließen Sie bitte, ohne eine Tabelle hinzuzufügen. Wählen sie im Menü Ansicht bitte die SQL-Ansicht. Wenn Sie dann ihre Fenster geschickt anordnen, können Sie gleichzeitig die SQL-Ansicht und ein paar Tabellen auf dem Bildschirm sehen, wie Abbildung 12.8 zeigt.

Bei SELECT-Anweisungen können Sie über die Symbolleiste zwischen Datenblatt- und Entwurfsansicht wechseln. Zum Ausführen der anderen Anweisungen dient das rot-braune Ausrufungszeichen, wie Abbildung 12.9 zeigt.

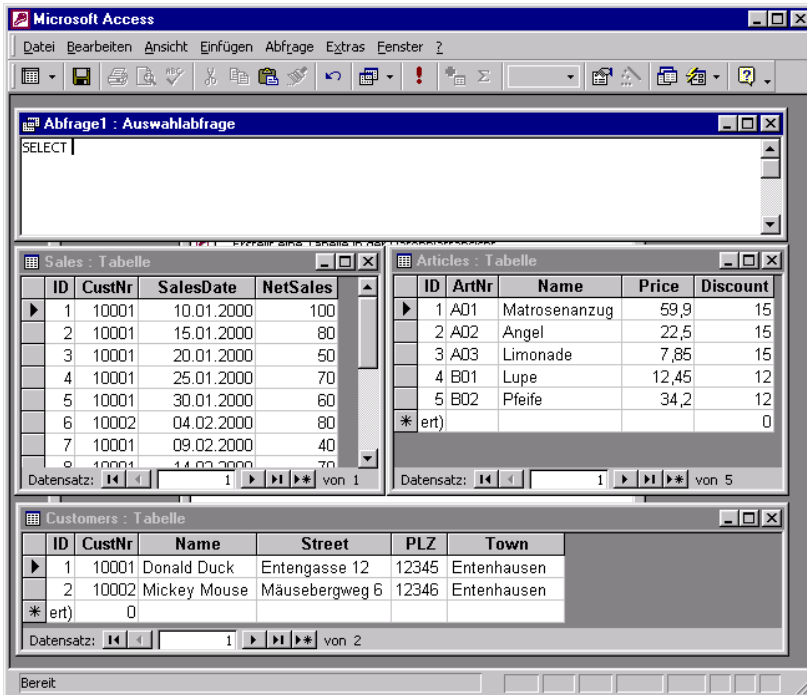


Abbildung 12.8: Access mit SQL-Ansicht und Tabellen

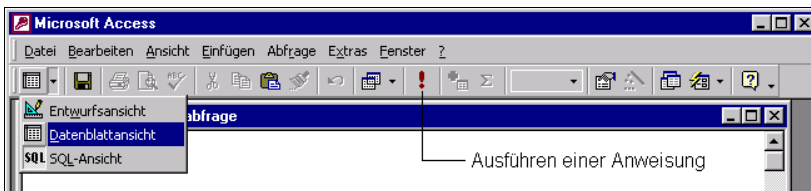


Abbildung 12.9: Access, wechseln zwischen Ansichten

12.3.1 SELECT

Mit SELECT werden Abfragen eingeleitet, die im Ergebnis dazu geeignet sind, Daten zurückzugeben. Die Kernelemente der SELECT-Anweisung lauten:

```
SELECT [Feldliste] FROM [Tabelle(n)]
```

Im folgenden Beispiel werden alle Datensätze der Tabelle Customers angezeigt:

```
SELECT * FROM Customers
```

Das * steht hierbei für alle Felder. Sie können aber auch eine Feldliste angeben, in der die einzelnen Felder durch Komma voneinander getrennt sind:

```
SELECT CustNr, Name FROM Customers
```

WHERE-Klausel

Mit der WHERE-Klausel können wir den Datenbestand eingrenzen. Das folgende Statement gibt nur den Datensatz des Kunden 10001 zurück:

```
SELECT * FROM Customers WHERE CustNr = 10001
```

Bei CustNr handelt es sich um ein numerisches Feld. Beinhaltet das Feld jedoch eine Zeichenkette, so muss der Begriff in ein einfaches Hochkomma eingeschlossen werden:

```
SELECT * FROM Customers WHERE PLZ = '12346'
```

Bei Gleitkommawerten müssen wir das Komma durch einen Punkt ersetzen. Aber auch ein Datumswert erfordert etwas Nachbearbeitung. Doch das müssen Sie übrigens nicht selbst machen, denn wir werden nachher noch eine Klasse namens clsUtil vorstellen, die hierfür Methoden bereitstellt.

Neben dem Gleichheitszeichen können wir aber auch die bereits von VBA bekannten Operatoren für Vergleiche verwenden:

```
SELECT * FROM Sales WHERE NetSales > 80 'größer 80  
SELECT * FROM Sales WHERE NetSales >= 80 'größer gleich 80  
SELECT * FROM Sales WHERE NetSales <> 80 'ungleich 80
```

Sogar der Vergleich einer Teilzeichenkette über LIKE ist möglich, und das sogar bei numerischen Feldinhalten, wie die folgenden Beispiele zeigen:

```
SELECT * FROM Articles WHERE Name LIKE 'L*'  
SELECT * FROM Sales WHERE NetSales LIKE '8?'
```

Gerade das letzte Beispiel ist dazu geeignet, aus strukturierten Nummern Teile zu selektieren, sofern Sie das Glück haben, auf wirklich strukturierte, klassifizierende Systeme zu treffen.

Sie können mehr als eine Teilbedingung in einer WHERE-Klausel unterbringen, indem Sie die Operatoren AND, OR und NOT verwenden:

```
SELECT * FROM Sales WHERE CustNr = 10001 AND NetSales >=50
```

Wie im richtigen Leben gelten natürlich auch hier die Regeln der Booleschen Algebra nebst derer zur Klammersetzung.

Aggregationen

Neben der Rückgabe einzelner Datensätze können wir einzelne Felder auch aggregieren, das heißt, Summen bilden, Mittelwert oder Datensätze auszählen. In den folgenden Beispielen wird je ein Datensatz mit einem Feld zurückgegeben, das unseren angeforderten Wert enthält:

```
SELECT SUM(NetSales) FROM Sales
```

Die folgenden Aggregatfunktionen existieren in SQL

Name	Bedeutung
AVG	Mittelwert (Average) über die Auswahl
COUNT	zählt die betroffenen Datensätze
MIN	kleinster der beteiligten Werte
MAX	größter der beteiligten Werte
STDEV	Standardabweichung der beteiligten Werte bezogen auf eine Stichprobe
STDEVP	Standardabweichung der beteiligten Werte bezogen auf die Grundgesamtheit
SUM	Summe der beteiligten Werte
VAR	Varianz der beteiligten Werte bezogen auf eine Stichprobe
VARP	Varianz der beteiligten Werte bezogen auf die Grundgesamtheit

Bei allen vorangegangenen Beispielen präsentierten sich die Felder unter ihren Feldnamen. Hier aber verkauft uns Access (oder besser die JET) das Ergebnis als Expr1000, wie Abbildung 12.10 zeigt.



Abbildung 12.10: Rückgabe einer Aggregation

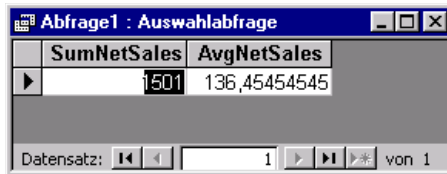
Wenn Sie eine einfache Aggregation nach diesem Muster durchführen, stört uns das nicht weiter, denn wir können bequem über den Index der Fields-Collection drauf zugreifen:

```
strSQL = " SELECT SUM(NetSales) FROM Sales"
rsSum.Open strSQL, cn, adOpenStatic
dblSum = rsSum.Fields(0).Value
```

Handelt es sich aber um mehrere Felder, so ist der Code zweifellos besser lesbar, wenn wir sprechende Feldnamen verwenden. Dies ist über einen so genannten Alias möglich, der mit dem Schlüsselwort AS eingeleitet wird:

```
SELECT SUM(NetSales) AS SumNetSales, AVG(NetSales) AS
AvgNetSales FROM Sales
```

Und nun präsentieren sich die beiden Felder mit eindeutigen Namen:



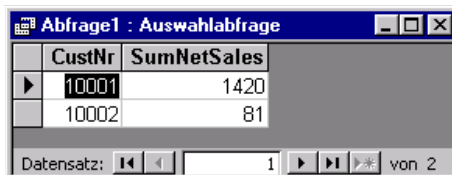
	SumNetSales	AvgNetSales
▶	1501	136,45454545

Datensatz: 1 von 1

Abbildung 12.11: Aggregationen mit Alias-Namen

Gruppierungen

In der Tabelle Sales sind alle Verkäufe mit den Feldern CustNr, SalesDate und NetSales enthalten. Schön wäre doch zum Beispiel folgende Rückgabe:



	CustNr	SumNetSales
▶	10001	1420
	10002	81

Datensatz: 1 von 2

Abbildung 12.12: Gruppierung der NetSales

Das dazugehörige SQL-Statement enthält neben der bereits verwendeten Aggregat-Funktion SUM(NetSales) nebst Alias AS SumNetSales auch noch die GROUP BY-Klausel, die für die Gruppierung der verschiedenen Werte verantwortlich ist. Um eine klare Zuordnung zu treffen, wird die CustNr mit in das Recordset integriert.

```
SELECT CustNr, SUM( NetSales) AS SumNetSales FROM Sales
GROUP BY CustNr
```

Wenn wir nur an den Kunden interessiert sind, deren Gesamtumsatz über 1000 liegt, so können wir die HAVING-Klausel ergänzen.

```
SELECT CustNr, SUM( NetSales) AS SumNetSales FROM Sales
GROUP BY CustNr HAVING SUM(NetSales) > 1000
```

Mit `HAVING` können wir die Ergebnisse von Aggregat-Funktionen zu Vergleichen und somit Einschränkungen heranziehen, denn in der `WHERE`-Klausel können wir keine Aggregat-Funktionen unterbringen.

Verknüpfte und berechnete Felder

Neben der Rückgabe vorhandener Felder haben Sie durch die Aggregation bereits Felder erzeugt. Auf diese Art können wir aber auch Felder zu einem neuen Feld verbinden oder bei numerischen Feldern auch Berechnungen ausführen lassen:

```
SELECT (PLZ & " " & Town) AS PLZTown FROM Customers
```

Im Ergebnis haben wir ein Feld namens `PLZTown`, in dem wir mit einem Leerzeichen getrennt Postleitzahl und Ort vorfinden, wie in Abbildung 12.13 zu sehen.



Abbildung 12.13: Verknüpftes Feld erzeugen

Arithmetische Operationen, wie beispielsweise das Berechnen eines Brutowertes, sind aber auch möglich:

```
SELECT (Price * 1.16) AS FullPrice FROM Articles
```

Auch hier wird über `AS` ein Aliasname definiert, um das Feld besser identifizieren zu können.

Sortierung

Bislang kamen die Daten in der Reihenfolge an, in der sie in den Tabellen angeordnet waren. Zumeist wollen wir das Recordset aber sortiert vorfinden. Hierzu dient die `ORDER BY`-Klausel, die generell am Ende der Anweisung steht:

```
SELECT * FROM Customers ORDER BY Name
SELECT * FROM Customers ORDER BY PLZ, Town, Street
```

Wir können also durch Komma getrennt eine ganze Reihe von Feldern angeben, wobei die Reihenfolge der Sortierung durch die Anordnung der Feldnamen bestimmt wird.

Relationen

Es gelang uns zwar, per GROUP-Klausel die Umsätze je Kunde zu kumulieren, aber der Name des Kunden stand uns nicht zu Gebote, da er in der Tabelle Sales schlicht und ergreifend fehlt. Es würde auch der zweiten Normalform widersprechen, diesen Namen dort zusätzlich aufzunehmen. Über das Instrument der Relation können wir ein Recordset erzeugen, das so aussieht, als wäre der Name Bestandteil der Sales-Tabelle. In Access können Sie sich eine solche Relation zusammenklicken. Wir schauen uns hier einmal an, wie eine solche Relation per SQL erzeugt werden kann.

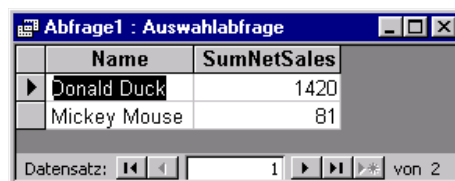
Wenn beispielsweise eine Kundennummer in verschiedenen Tabellen existiert, sollte sie logischerweise überall unter demselben Feldnamen erscheinen. Wir müssen aber in der SQL-Anweisung die beiden identisch lautenden Feldnamen miteinander in Beziehung setzen können. Damit der SQL-Interpreter sie sauber auseinander halten kann, benötigt er ein Unterscheidungskriterium. Das können wir ihm dadurch liefern, dass wir in diesen Fällen den Tabellennamen voranstellen:

```
SELECT Name, NetSales From Customers, Sales  
WHERE Customers.CustNr = Sales.CustNr
```

Sind aber mehr als eine Tabelle an einer SQL-Anweisung beteiligt, so können wir den bereits bekannten Alias-Namen dazu verwenden, allen beteiligten Tabellen einen ein- oder zweistelligen Alias-Namen zuzuordnen, wie folgendes Beispiel zeigt. Der eindeutigen Zuordenbarkeit der Felder macht es aber Sinn, den Alias jedem Feldnamen voranzustellen:

```
SELECT C.Name, S.NetSales  
FROM Customers AS C, Sales AS S  
WHERE C.CustNr = S.CustNr
```

Nun verbinden wir die bisherigen Erkenntnisse über Relationen mit der Gruppierung und erzeugen eine gruppierte Übersicht der Kundenumsätze, wobei wir statt der Kundennummer den Namen einblenden:



Name	SumNetSales
Donald Duck	1420
Mickey Mouse	81

Abbildung 12.14: Gruppierung der NetSales mit Anzeige des Kundennamen

Und so sieht die SQL-Anweisung aus:

```
SELECT C.Name, SUM(S.NetSales) AS SumNetSales
FROM Customers AS C, Sales AS S
WHERE C.CustNr = S.CustNr
GROUP BY C.Name
```

Damit können Sie die meisten Probleme lösen, die mit verknüpften Tabellen zu tun haben.

Damit ist SELECT aber noch lange nicht am Ende, eher am Anfang. Was wir hier auslassen, füllt locker ein kräftiges Kapitel:

▼ INNER, OUTER, LEFT und RIGHT JOINS, (CROSS JOINS)

▼ SELECT INTO-Anweisungen

▼ Sub-Selects

Doch den Grundstock für Abfragen haben wir zusammen.

12.3.2 UPDATE

Die UPDATE-Anweisung dient dazu, vorhandene Daten zu verändern. Hier die Grundstruktur zur Veränderung eines Feldes aller Datensätze einer Tabelle:

```
UPDATE [Tabelle] SET [Feld] = [Wert]
```

Wollen Sie mehrere Felder aktualisieren, so werden die Felder angehängt:

```
UPDATE [Tabelle] SET [Feld1] = [Wert1], [Feld2] = [Wert2]
```

Sind nur Teile der Tabelle betroffen, wird eine WHERE-Klausel ergänzt:

```
UPDATE [Tabelle] SET [Feld] = [Wert] WHERE ...
```

Es folgt ein kleines Codefragment, das einem aktuellen Projekt entnommen ist. Ein Handicap ist dort eine Gutschrift, mit der zum Beispiel bauliche Besonderheiten bei der Umsatzvorgabe berücksichtigt werden. Dieser Methode wird die Gutschrift nebst der Handicap-ID übergeben, lngStoreID ist als private Variable im Kontext definiert:

```
Public Sub LetHandicapValue(ByVal lngHandID As Long, _
                           ByVal lngValue As Long)
    ' Schreiben des übergebenen Handicap-Werts
    strSQL = "UPDATE Handicaps SET HandValue = " & _
             lngValue & " WHERE StoreID = " & lngStoreID & _
             " AND HandID = " & lngHandID
```

```
cn.Execute strSQL
End Sub
```

Hier sehen Sie auch, dass man einen UPDATE über die `Execute`-Methode des `Connection`-Objekts an die Datenbank schickt. Hier ist also kein Recordset im Spiel.

12.3.3 INSERT

Mit der INSERT-Anweisungen können Sie neue Datensätze in einer Tabelle anfügen. Die Anweisung hat folgende Grundstruktur:

```
INSERT INTO [Tabelle] ([Feldliste]) VALUES ([Werteliste])
```

Die INSERT-Anweisung muss nicht alle Felder übergeben, die in einer Tabelle vorhanden sind. In der folgenden Anweisung wird ein neuer Datensatz erzeugt und nur die Kundennummer eingetragen:

```
INSERT INTO Customers (CustNr) VALUES (10003)
```

Hier müssen Sie darauf achten, ob Ihre Datenbank nicht eigene Vorstellungen davon entwickelt hat, welche Felder erforderlich sind und welche nicht.

Wollen Sie mehrere Felder übergeben, so müssen die Feldnamen und die Werte durch Kommata voneinander getrennt sein:

```
INSERT INTO Customers (CustNr, Name) VALUES
(10003, 'Goofy')
```

Bei umfangreichen Feldlisten verwende ich in der Regel zwei separate Variablen für Felder und Werte:

```
strFields = "(CustNr, Name, Street, PLZ, Town)"
strValues = "(" & lngCustNr & ", '" & strName & "', '" &...
strSQL = "INSERT INTO Customers " & strFields & _
        " VALUES " & strValues
cn.Execute strSQL
```

Bei der INSERT-Anweisung sollten Sie bei fremden DBMS fragen, ob das Schlüsselwort `VALUES` unterstützt wird. Die JET beispielsweise verlangt es, der Oracle-Server hingegen mag es nicht.

Das Beispiel zur UPDATE-Anweisung war unvollständig, weil der Handicap-Wert dort geändert wurde, ohne zu prüfen, ob der Datensatz überhaupt existiert. Hier ergänzen wir den Makel:

```

Dim rsTemp As New ADODB.Recordset
strSQL = "SELECT * FROM Handicaps WHERE StoreID = " & _
        lngStoreID & " AND HandID = " & lngHandID
rsTemp.Open strSQL, cn, adOpenStatic
If rsTemp.EOF Then
    strSQL = "INSERT INTO Handicaps ..."
Else
    strSQL = "UPDATE Handicaps SET ..."
End If
rsTemp.Close
cn.Execute strSQL

```

Das Recordset nach Gebrauch zu schließen, ist eine Angewohnheit, die daher rührt, dass ich nicht weiß, ob es in einer der beteiligten Bibliotheken Ressourcen in Anspruch nimmt und ob die andernfalls auch aufgelöst würden. Nach einem schließen der Connection werden alle eventuell noch offenen Recordsets geschlossen, sagen alle, wie sich das aber mit einem Recordset verhält, weiß ich eben nicht.

12.3.4 DELETE

Mit der DELETE-Anweisung können Sie alle Datensätze einer Tabelle löschen, aber auch in Form der WHERE-Klausel Bedingungen definieren, die von den zu löschenden Datensätzen erfüllt sein müssen:

```

DELETE FROM Sales
DELETE FROM Sales WHERE CustNr = 10003

```

12.4 Die Klasse clsUtil

Auf Dauer ist es unbefriedigend, immer wieder dieselben elementaren Probleme mit frischen Codezeilen zu lösen. So entstand mit der Zeit eine Klasse, die sich dieser Standardaufgaben zuverlässig annimmt.

ClearNull

Wir sprachen bereits darüber, dass Datenbankfelder den Wert Null beinhalten können. Die folgende Funktion namens ClearNull erhält ein Datenbankfeld und gibt einen typgerechten Wert zurück, sofern der Inhalt Null ist. Sie wird generell dann eingeschaltet, wenn eine Datenbankrückgabe aus einer Methode zurückgegeben wird:

```

Public Function ClearNull(ByVal fldAct As ADODB.Field) _
        As Variant
    On Error GoTo errHandler
    'Prüfen, ob Wert Null ist:
    If IsNull(fldAct.Value) Then

```

```

        'abhängig von Type Rückgabe zuweisen
        Select Case fldAct.Type
        Case adBoolean
            ClearNull = False
        Case adSmallInt, adInteger, adSingle, adDouble, _
            adCurrency
            ClearNull = 0
        Case adDate, adChar, adVarChar
            ClearNull = ""
        End Select
    Else
        ClearNull = fldAct.Value
    End If
    Exit Function

errHandler:
    Err.Raise Err.Number, "clsUtil:ClearNull"
End Function

```

Hier ist nun ein Beispiel für die Verwendung dieser Methode:

```

Public Property Get Name() As String
    Name = cUtil.ClearNull(rsPerson.Fields("Name"))
End Property

```

SQLDate

SQL erwartet ein Datumsfeld generell in dieser Form:

#M/T/YYYY#

Die Methode `SQLDate` transformiert das übergebene Datum in diese Form:

```

Public Function SQLDate(ByVal actDate As Date) As String
    On Error GoTo errHandler
    SQLDate = "#" & Month(actDate) & "/" & Day(actDate) & _
        "/" & Year(actDate) & "#"
    Exit Function

errHandler:
    Err.Raise Err.Number, "clsUtil:SQLDate"
End Function

```

Auch hier ein Beispiel für die Verwendung:

```

Public Property Let SentOn(ByVal dateSentOn As Date)
    strSQL = "UPDATE Reports SET SentOn = " & _

```

```

        cUtil.SQLDate(dateSentOn) & _
        " WHERE ID = " & rsReport.Fields("ID").Value
cn.Execute strSQL
rsReport.Requery
End Property

```

Diese Eigenschaft aktualisiert ein Datum in einer Tabelle und aktualisiert das der Klasse zugrunde liegende, im Übrigen natürlich statische Recordset durch den Aufruf der ReQuery-Methode.

SQLDecimal

In SQL ist das Dezimaltrennzeichen generell der Punkt, was die folgende Methode erledigt:

```

Public Function SQLDecimal(ByVal dblValue As Double) _
                        As String
    On Error GoTo errHandler
    SQLDecimal = Replace(CStr(dblValue), ",", ".")
    Exit Function

errHandler:
    Err.Raise Err.Number, "clsUtil:SQLDecimal"
End Function

```

Alle Werte des Typs Double, Single oder Currency werden beim Einbau in eine SQL-Anweisung durch diese Methode korrekt umgewandelt:

```

strSQL = "UPDATE Handicaps SET HandValue = " & _
        cUtil.SQLDecimal(sngValue) & _
        " WHERE StoreID = " & lngStoreID & _
        " AND HandID = " & lngHandID

```

Kommt Ihnen das bekannt vor? Das Beispiel hatten wir schon, doch dort erschien das Handicap kurzerhand als Long-Datentyp. In Wirklichkeit ist es natürlich ein Single, wie in diesem Beispiel korrekt dargestellt.

SQLString

Wenn eine Zeichenkette ein Apostroph beinhaltet, bekommen Sie mit dem SQL-Interpreter ein Problem, wie dieses Beispiel zeigt:

```

INSERT INTO Customers (Name) VALUES ('L'Oreal')

```

Das Apostroph wird natürlich von SQL als Ende der Zeichenkette interpretiert. Dadurch wird das nachfolgende Oreal' als SQL-Sprachelement aufgefasst. Die Methode SQLString stellt einem eventuellen Apostroph ein zweites voran, und SQL ist wieder zufrieden:

```

Public Function SQLString(ByVal strValue As String) _
    As String
    On Error GoTo errHandler
    SQLString = Replace(strValue, "'", "'")
    Exit Function

errHandler:
    Err.Raise Err.Number, "clsUtil:SQLString"
End Function

```

Diese Methode sollte man besser einmal zuviel verwenden:

```

strSQL = "UPDATE Persons SET Name = '" & _
    cUtil.SQLString(strName) & _
    "' WHERE ID = " & lPersID

```

12.5 DBRechnung.xls

Auf der Buch-CD finden Sie die aus Kapitel 10 bereits bekannte Rechnung.xls in neuem Gewand. Die alten Datenklassen mussten natürlich ersetzt werden. Darum werden wir uns gleich kümmern. Außerdem erhielt sie einen Kundendialog, in dem auch ein paar grundlegende Techniken des Datenzugriffs zur Sprache kommen.

12.5.1 Die neuen Datenklassen

Als DBRechnung.xls hat sie drei neue Datenklassen. Die in 12.4 vorgestellte clsUtil ist natürlich mit von der Partie.

Die Variable cn wurde im modMain ergänzt

```
Public cn As ADODB.Connection
```

und in der Ereignisprozedur Workbook_Open wird die Verbindung geöffnet:

```

Private Sub Workbook_Open()
    ...
    Set cn = New ADODB.Connection
    cn.Provider = "Microsoft.Jet.OLEDB.4.0"
    cn.ConnectionString = "Data Source=" & _
        ThisWorkbook.Path & "\Rechnung.mdb"
    cn.Open
    ...
End Sub

```

Der Rest spielt sich vollständig innerhalb der vier Datenklassen `clsArticle`, `clsCustomer`, und `clsHistory` ab. Die Klasse `clsInvoice` blieb unverändert, da sich dort kein Datenzugriff befand.

Klasse `clsArticle`

Die `SetArticle`-Methode ermittelte in der alten Version den Zeilenzeiger, der in den Eigenschaftsroutinen verwendet wurde. In der Datenbank-Version wird das private Recordset `rs` erzeugt. Existiert die Kundennummer nicht, so ist die `EOF`-Eigenschaft `True`, und wir erzeugen einen Fehler:

```
Public Sub SetArticle(ByVal strArtNr As String)
    On Error GoTo errHandler
    strSQL = "SELECT * FROM Articles WHERE ArtNr = '" & _
        strArtNr & "'"
    If (rs.State And adStateOpen) = adStateOpen Then
        rs.Close
    End If
    rs.Open strSQL, cn, adOpenStatic
    If rs.EOF Then
        Err.Raise vbObjectError + 10009, , "Der Artikel ..."
    End If
    Exit Sub

errHandler:
    Err.Raise ...
End Sub
```

Betrachten wir uns noch exemplarisch die Eigenschaft `DiscountRate`. Das Feld `Discount` wird durch die `ClearNull`-Methode entschärft und die Rückgabe der Methode an den Eigenschaftsnamen übergeben:

```
Public Property Get DiscountRate() As Single
    On Error GoTo errHandler
    DiscountRate = cUtil.ClearNull(rs.Fields("Discount"))
    Exit Property

errHandler:
    Err.Raise ...
End Property
```

Die anderen Eigenschaften sind im Prinzip genauso aufgebaut.

Klasse `clsCustomer`

Auch hier wird in einer Methode das Recordset positioniert:

```

Public Sub SetCustomer(ByVal lngNewCustNr As Variant)
    On Error GoTo errHandler
    strSQL = "SELECT * FROM Customers WHERE CustNr = " & _
        lngNewCustNr
    If (rs.State And adStateOpen) = adStateOpen Then
        rs.Close
    End If
    rs.Open strSQL, cn, adOpenStatic
    If rs.EOF Then
        Err.Raise 10009, , "Die Kundennummer ..."
    End If
    lngCustNr = lngNewCustNr
    Exit Sub

errHandler:
    Err.Raise ...
End Sub

```

Die Name-Eigenschaft wird ähnlich gebildet, wie wir das bei der Eigenschaft DiscountRate der clsCustomer bereits sahen:

```

Public Property Get Name() As String
    ...
    Name = cUtil.ClearNull(rs.Fields("Name"))
    ...
End Property

```

Die Eigenschaft DiscountCoeff greift ja auf die History-Klasse zu, in der bei der Excel-Variante durch zweimaligen Aufruf der Methode SetFilter der Datenbereich eingegrenzt wurde. Das können wir hier etwas einfacher gestalten und die Filterinformationen in den Eigenschaftsaufruf mit hineinpacken. Somit entfällt auch die ResetFilter-Methode. Hier sehen wir den alten und den neuen Code untereinander:

```

lngDays = cSettings.Intervals * cSettings.SalesInterval
sngTotalSales=cSettings.Intervals * cSettings.SalesTreshold
cHistory.ResetFilter
cHistory.SetFilter enumHistoryColCustNr, lngCustNr
cHistory.SetFilter enumHistoryColDate, ">=" & _
    Cdbl(Date - lngDays), xlAnd, "<=" & Cdbl(Date)
sngCustSales = cHistory.Sales
DiscountCoeff = Application.Min(sngCustSales / _
    sngTotalSales, 1)
cHistory.ResetFilter

```


Und nun der neue Code:

```
lngDays = cSettings.Intervals * cSettings.SalesInterval
sngTotalSales=cSettings.Intervals * cSettings.SalesTreshold
sngCustSales = cHistory.CustomersPeriodSales(lngCustNr, _
    Date - lngDays, Date)
DiscountCoeff = Application.Min(sngCustSales / _
    sngTotalSales, 1)
```

Klasse clsHistory

Diese Klasse weist nur noch eine Eigenschaftsroutine auf. Die Kundennummer sowie die beiden Datumswerte werden übergeben und in das SQL-Statement integriert. cUtil.SQLDate sorgt für die SQL-konforme Umsetzung der Datumswerte:

```
Public Property Get CustomersPeriodSales( _
    ByVal lngCustNr As Long, _
    ByVal dateFrom As Date, _
    ByVal dateTo As Date) As Double
    On Error GoTo errHandler
    strSQL = "SELECT SUM(NetSales) FROM Sales " & _
        "WHERE CustNr = " & lngCustNr & _
        " AND SalesDate >= " & _
        cUtil.SQLDate(dateFrom) & _
        " AND SalesDate <= " & cUtil.SQLDate(dateTo)
    If (rs.State And adStateOpen) = adStateOpen Then
        rs.Close
    End If
    rs.Open strSQL, cn, adOpenStatic
    CustomersPeriodSales = cUtil.ClearNull(rs.Fields(0))
    Exit Property

errHandler:
    Err.Raise Err.Number ...
End Property
```

Das Zurückgeben der gefilterten Werte ist gegenüber der Excel-Variante wesentlich kompakter geworden.

12.5.2 Der Kundendialog

Da die Kundentabelle nicht mehr zur Verfügung steht, wurde der Anwendung ein Kundendialog spendiert, der in Abbildung 12.18 dargestellt ist. Der Aufruf erfolgt über eine Schaltfläche in der Tabelle Rechnung (siehe Abbildung 12.15).

The screenshot shows an Excel spreadsheet titled 'DBRechnung.xls'. The data is organized as follows:

	A	B	C	D	E	F
1						
2		Donald Duck				
3		Entengasse 12				
4		12345 Entenhausen				
5						
6					KundenNr:	10001
7						
8						
9	ArtNr	Bezeichnung	E-Preis	Anz	Rabatt	G-Preis
10	A01	Matrosenanzug	59,90 DM	3	10,24 DM	169,46 DM
11	A02	Angel	22,50 DM	3	0,95 DM	66,55 DM
12	A03	Limonade	7,85 DM	2	0,89 DM	14,81 DM

The status bar at the bottom indicates 'Rechnung / Settings /'.

Abbildung 12.15: Tabelle Rechnung

Befinden sich keine Datensätze in der Kundentabelle, wird das in der ComboBox signalisiert:

Kunde: <keine Kundendaten vorhanden>

Abbildung 12.16: Kundendialog, keine Kundendaten vorhanden

Ist die Zelle F6 leer, wird der Eintrag Bitte auswählen als erster Eintrag ergänzt:

Kunde: <bitte auswählen>

Abbildung 12.17: Kundendialog, Bitte auswählen

Steht in F6 hingegen eine Kundennummer, so werden die Daten dieses Kunden angezeigt.

The 'Kundendialog' window displays the following information:

- Kunde: 10001 - Donald Duck
- Name: Donald Duck
- Straße: Entengasse 12
- Ort: 12345 Entenhausen

Buttons on the right include: Kunde hinzufügen, Kunde löschen, Schließen, and Auswählen.

Abbildung 12.18: Kundendialog

GetCustomers-Methode in clsCustomer

Die Klasse clsCustomer wurde um die Methode GetCustomers erweitert, die Kundennummer und -name aller Kunden in einem Variant-Array zurückgibt. Sind keine Kunden in der Tabelle, gibt sie Empty zurück. Mit dieser Technik habe ich beste Erfahrungen gesammelt. Der Dialog wurde auch nur aus dem Grund ergänzt, um ihnen diese Technik an einem Beispiel demonstrieren zu können.

```
Public Function GetCustomers() As Variant
    ' Rückgabe aller Kundennummern und -namen
    Dim rsTemp As ADODB.Recordset 'temporäres Recordset
    Dim varValues As Variant       'temporäres Array
    Dim iValue As Long             'Zeiger auf 2.
    Dimension
    On Error GoTo errHandler
    'Recordset erzeugen
    strSQL = "SELECT CustNr, Name FROM Customers " & _
        "ORDER BY Name"
    Set rsTemp = New ADODB.Recordset
    rsTemp.Open strSQL, cn, adOpenStatic
    'Empty zurück geben, wenn keine rs leer
    If rsTemp.EOF Then
        GetCustomers = Empty
        Exit Function
    End If
    'Array füllen
    ReDim varValues(1, 0)
    Do Until rsTemp.EOF
        'neue obere Grenze ermitteln
        iValue = UBound(varValues, 2)
        varValues(0, iValue) = _
            cUtil.ClearNull(rsTemp.Fields("CustNr"))
        varValues(1, iValue) = _
            cUtil.ClearNull(rsTemp.Fields("Name"))
        'obere Grenze erhöhen
        ReDim Preserve varValues(1, iValue + 1)
        'Recordset weiterbewegen
        rsTemp.MoveNext
    Loop
    'Recordset schließen und Rückgabe zuweisen
    rsTemp.Close
    ReDim Preserve varValues(1, iValue)
```

```
GetCustomers = varValues
Exit Function
```

```
errHandler:
    Err.Raise Err.Number + vbObjectError, "clsCustomer:..."
End Function
```

Aufruf des Dialogs und Füllen der ComboBox

Ist die Zelle der Kundennummer leer, wird die Routine `cmbCustomer_Load` ohne Argument aufgerufen. Steht hingegen eine Nummer in dieser Zelle, wird sie als Argument an die Routine übergeben.

```
Private Sub cmdCustomer_Click()
    ' Kundendialog aufrufen
    Dim varCustNr As Variant    'Inhalt der KNr-Zelle
    Dim fCust As frmCustomer    'Instanz auf Kundendialog
    On Error GoTo errHandler
    Set fCust = New frmCustomer
    varCustNr = _
        ThisWorkbook.Names("Customer").RefersToRange.Value
    If IsEmpty(varCustNr) Then
        fCust.cmbCustomer_Load
    Else
        fCust.cmbCustomer_Load varCustNr
    End If
    fCust.Show vbModal
    Unload fCust
    Exit Sub

errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "shtInvoice:cmdCustomer_Click"
    End If
    cError.ErrorMessage
End Sub
```

In der Routine `cmbCustomer_Load` wird die ComboBox nun endlich geladen:

```
Public Sub cmbCustomer_Load(Optional ByVal lngCustNr As _
                            Variant)
    ' Kunden laden
    Dim varValues As Variant    'Array
    Dim iValue As Long          'Zeiger in varValues
    Dim iListIndex As Long      'späterer ListIndex
```

```

On Error GoTo errHandler
cmbCustomer.Clear
varValues = cCust.GetCustomers
If IsEmpty(varValues) Then
    cmbCustomer.AddItem "<keine Kundendaten vorhanden>"
    cmbCustomer.ListIndex = 0
    Exit Sub
End If
If IsMissing(lngCustNr) Then
    cmbCustomer.AddItem "<bitte auswählen>"
End If
For iValue = 0 To UBound(varValues, 2)
    cmbCustomer.AddItem varValues(0, iValue) & " - " & _
        varValues(1, iValue)
    If Not IsMissing(lngCustNr) Then
        If varValues(0, iValue) = lngCustNr Then
            iListIndex = iValue
        End If
    End If
Next
cmbCustomer.ListIndex = iListIndex
Exit Sub

errHandler:
If InStr(1, Err.Source, ":") = 0 Then
    Err.Source = "frmCustomer:cmbCustomer_Load"
End If
Err.Raise Err.Number
End Sub

```

Das Exit-Ereignis der einzelnen TextBoxes bietet uns die Möglichkeit, der betreffende Eigenschaft der Klasse clsCustomer den neuen Wert zu übergeben. Die Klasse verfügt über ein Eigenschaftspaar für Name, Straße, PLZ und Ort. Die Property Get wurde bereits vorgestellt, sodass wir uns hier exemplarisch nur die Property Let Name ansehen müssen:

```

Public Property Let Name(ByVal strNewValue As String)
    On Error GoTo errHandler
    strSQL = "UPDATE Customers SET Name = '" & _
        cUtil.SQLString(strNewValue) & _
        "' WHERE CustNr = " & lngCustNr
    cn.Execute strSQL
    rs.Requery
Exit Property

```

```

errHandler:
    Err.Raise Err.Number + vbObjectError, ...
End Property

```

Wichtig ist, dass Sie nach einer solchen Änderung des statischen Recordsets, das die Eigenschaftsroutinen versorgt, mit der ReQuery-Methode aktualisieren.

Die Ereignisroutine txtName_Exit versorgt die Name-Eigenschaft:

```

Private Sub txtName_Exit(ByVal Cancel As _
                        MSForms.ReturnBoolean)
    cCust.Name = txtName.Text
    cmbCustomer.List(cmbCustomer.ListIndex) = _
        Left(cmbCustomer.Text, InStr(1, cmbCustomer.Text, _
        & " - ") + 2) & txtName.Text
End Sub

```

Nur bei der txtName wird zusätzlich der neu eingegebene Name in der ComboBox aktualisiert, was der hervorgehobene Teil erledigt.

Kunden hinzufügen

Abbildung 12.19: Kundendialog, Neuen Kunden anlegen

Die Klasse clsCustomer beinhaltet eine Add-Methode, die einen neuen Datensatz anlegt und die Kundennummer zurückgibt. Diese Kundennummer wird in die ComboBox aufgenommen und die txtName erhält den Fokus, damit der Name eingegeben werden kann. Den Rest kennen wir bereits: Im Exit-Ereignis wird der Name in der ComboBox angehängt.

Schauen wir uns zuerst die neue Methode an:

Ein temporäres Recordset aggregiert die größte vorhandene Kundennummer. Sind keine Kunden in der Datenbank, wird die erste Nummer mit 10001 festgelegt. Sind jedoch Kunden in der Datenbank, wird die größte Nummer inkrementiert. Die Execute-Methode der Connection übergibt einen INSERT-SQL an die Datenbank, in der nur das Feld CustNr belegt wird:

```

Public Function Add() As Long
    '    Neuen Datensatz anlegen und Kundennummer zurückgeben
    Dim rsTemp As ADODB.Recordset      'temporäres Recordset
    Dim lngCustNr As Long               'neue Kundennummer
    On Error GoTo errHandler
    'größte bisherige Kundennummer ermitteln
    strSQL = "SELECT MAX(CustNr) FROM Customers"
    Set rsTemp = New ADODB.Recordset
    rsTemp.Open strSQL, cn, adOpenStatic
    '1. Nummer auf 10001, wenn keine rs leer
    If rsTemp.EOF Then
        lngCustNr = 10001
    Else
        lngCustNr = cUtil.ClearNull(rsTemp.Fields(0)) + 1
    End If
    rsTemp.Close
    'Datensatz erzeugen
    strSQL = "INSERT INTO Customers (CustNr) VALUES (" & _
        lngCustNr & ")"
    cn.Execute strSQL
    Add = lngCustNr
    Exit Function

errHandler:
    Err.Raise Err.Number + vbObjectError, "clsCustomer:Add"
End Function

```

Die Ereignisroutine der Schaltfläche im Dialog hat folgenden Inhalt:

```

Private Sub cmdAddNew_Click()
    Dim lngCustNr As Long
    On Error GoTo errHandler
    lngCustNr = cCust.Add
    cmbCustomer.AddItem lngCustNr & " - "
    cmbCustomer.ListIndex = cmbCustomer.ListCount - 1
    txtName.SetFocus
    Exit Sub

errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "frmCustomer:cmdAddNew_Click"
    End If
    cError.ErrorMessage
End Sub

```

Kunden löschen

Auch hier steht eine Methode im Klassenmodul `clsCustomer` bereit, die den Datensatz löscht. Das Kriterium ist die Kundennummer:

```
Public Sub Delete(ByVal lngCustNr As Long)
    ' Kunden löschen
    On Error GoTo errHandler
    strSQL = "DELETE FROM Customers WHERE CustNr = " & _
        lngCustNr
    cn.Execute strSQL
    Exit Sub

errHandler:
    Err.Raise Err.Number + vbObjectError, "clsCustomer:D..."
End Sub
```

Die Ereignisroutine `cmdDelete_Click` bedient diese Methode und löscht anschließend den Eintrag aus der ComboBox:

```
Private Sub cmdDelete_Click()
    On Error GoTo errHandler
    cCust.Delete CLng(Left(cmbCustomer.Text, _
        InStr(1, cmbCustomer.Text, " - ")))
    cmbCustomer.RemoveItem cmbCustomer.ListIndex
    Exit Sub

errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "frmCustomer:cmdDelete_Click"
    End If
    cError.ErrorMessage
End Sub
```

Die Ereignisroutine der ComboBox sorgt nun selbständig für die Aktualisierung der TextBoxes.

12.5.3 Zusammenfassung

Arbeitet Excel als Frontend für Datenbanken, so verlieren wir die einfache Editiermöglichkeit der Daten, da sie nicht mehr so schön offen in Tabellenblättern zugänglich sind, wie wir das von geschlossenen Excel-Anwendungen her gewöhnt sind. Ist die weitere Bearbeitung der Daten dennoch erforderlich, können wir einen Dialog in der hier vorgestellten Art und Weise erstellen.

Eine andere Alternative ist das Einlesen der Daten in eine Tabelle. Hierbei können Sie entweder alle eintragen oder aber mittels Filter nur einen Teil darstellen, falls der Gesamtdatenbestand nicht erforderlich oder zu umfangreich wäre. Dann müssen Sie eine Schaltfläche in die Tabelle platzieren, die das Rückübertragen der Daten einleitet. Doch hier handeln Sie sich eventuell Konflikte ein, wenn die Daten – jetzt übertreibe ich ein wenig – wochenlang in der Tabelle herumgammelten und nun zurückübertragen werden sollen. Womöglich hat ein anderer in der Zwischenzeit auch Daten verändert, die Sie nun überschreiben. Was tun? Das Originalabbild irgendwo speichern, mit den neuen Datenbankwerten abgleichen und bei Konflikten den Anwender um Rat fragen? Das geht, ist aber sicherlich mühsam. Es gäbe natürlich die Möglichkeit, in Flag in der Datenbank zu hinterlegen, das anderen signalisiert, dass diese Daten von Anwender X gerade bearbeitet werden.

Die Lösung liegt vermutlich in einem gesunden Verhältnis von Orgware und Software. Vielleicht müssen Sie irgendwo hinterlegen, wer überhaupt welche Daten ändern. Und wenn Ihnen das überschneidungsfrei gelingt, sind Ihre Probleme gelöst.

Das Wichtigste aber ist und bleibt die Kapselung ihrer Datenzugriffe in Klassen. Als ich das hier zugrunde liegende Beispiel aus Kapitel 10 umbaute, begann ich mit der Klasse `clsCustomer`. Als sie fertig war und die alte Kundentabelle aus der Arbeitsmappe den Weg alles irdischen gegangen war, ließ ich das Programm laufen. Die Anwendung griff also teilweise auf die alten Excel-Datentabellen zu, besorgte sich aber alle Kundendaten aus der Datenbank. Alles funktionierte auf Anhieb so, wie es sollte.

Versuchen Sie das einmal mit einer Anwendung, in der die Datenzugriffsmechanismen nicht säuberlich geordnet in Klassenmodulen vorliegen. Dann haben sie genau das, was man so schön als Wegwerfcode bezeichnet: Eine Änderung in den Datenstrukturen, und sie schreiben alles neu, weil es schneller geht, als die überall verstreuten Datenzugriffe anzupassen.



Add-Ins

13

Kapitelüberblick

13.1	Komponenten von Add-Ins	628
13.2	Erstellen von Add-Ins	629
13.2.1	CommandBar erzeugen	629
13.2.2	Der Dummy-Dialog	632
13.2.3	Add-In erzeugen	633

Add-Ins sind ausgelagerte Komponenten des Funktionsumfangs eines Programms. Der Grund für die Auslagerung wird klar, wenn man sich die Größe der Excel.exe oder anderer Komponenten der Office-Familie vor Augen hält. Wären alle mit Excel ausgelieferten Add-Ins in die Excel.exe integriert, so wäre das Programm um einiges größer mit den daraus resultierenden Geschwindigkeitseinbußen. Bei den heutigen Hardwarekonfigurationen mit zumeist mindesten 64 MByte RAM bei neueren Workstations würde dies vermutlich kein großes Problem darstellen. Erinnern Sie sich aber noch an die Zeiten mit Excel 5 und 486er PCs mit 8 MByte RAM? Ist eigentlich gar nicht so lange her ...

Der Vorteil liegt also ganz klar darin, daß man mittels des Add-In Managers festlegen kann, welche Programmkomponenten aktuell oder dauerhaft zur Verfügung stehen sollen. Autosave beispielsweise ist ein Kandidat für dauerhafte Aktivierung, die Analysefunktionen werden zumeist nur so lange aktiviert, wie sie benötigt werden.

Man kann allerdings trefflich über die Auslagerung verschiedener Add-Ins streiten und tut es auch – bestes Beispiel ist das genannte Autosave Add-In von Excel, das nach Ansicht vieler Zeitgenossen Bestandteil der Excel.exe sein müßte.

Seit Excel 5 können wir Add-Ins auch selbst erstellen und verbreiten. Im Grunde genommen ist ein Add-In eine ganz normale Excel-Arbeitsmappe, die durch das Speichern als Add-In eine Art Kompilation erfährt. Add-Ins sind im Übrigen statisch und auch nicht im Fenster Menü vertreten, somit für den Anwender unsichtbar. Wird das VBA-Projekt durch ein Paßwort geschützt, ist auch der Code für den Anwender nicht sichtbar.

13.1 Komponenten von Add-Ins

Add-Ins können im Prinzip aus all den Komponenten bestehen, die auch in gewöhnlichen Excel-Dateien enthalten sein können, also

- ▼ Tabellenblätter,
- ▼ Module, Klassenmodule und Formulare.

Allerdings sind lediglich Formulare für den Anwender sichtbar. Sie können weder Tabellenblätter noch Diagrammblätter zur Anzeige bringen. Interessanterweise bietet Excel den Dateityp Microsoft Excel-Add-In erst gar nicht an, wenn das aktive Blatt ein Diagrammblatt ist. Das soll uns nicht weiter stören, denn es wäre ohnehin unsinnig.

Es stellt sich nun die Frage, welche Funktionalität wir mit einem Add-In zur Verfügung stellen können. Zum einen sind da die `Public Functions` zu nennen, die Bestandteil der Tabellenfunktionen von Excel werden. Somit

können Sie sich die lang ersehnte Summenfunktion bauen, die zu den Summanden immer noch das aktuelle Datum hinzuzählt :-).

Zum anderen können Sie beliebige Programme integrieren, die alles das machen können, was gewöhnliche Excel-Anwendungen auch durchführen. Hierunter fällt beispielsweise das Bearbeiten der aktiven Arbeitsmappe oder aber das Erzeugen und Bearbeiten einer neuen Arbeitsmappe.

Zwar stehen Ihnen auch die Tabellen des Add-Ins selbst zur Verfügung, Änderungen lassen sich jedoch nicht in dem Add-In speichern, oder sagen wir besser nicht ohne Tricks. Das stellt insofern kein Problem dar, weil wir zum Ablegen von Settings die Registry zur Verfügung haben.

Sie können aber die Tabellen des Add-Ins dazu verwenden, um Formatierungen zu hinterlegen, die in die Zieldatei übertragen werden.

13.2 Erstellen von Add-Ins

Hier fassen wir die Schritte zusammen, die zur Erstellung eines Add-Ins erforderlich sind. Wie versprochen, wird Sie nichts Neues erwarten.

Dennoch finden Sie auf der Buch-CD die Datei AddIn.xls, in der ein Add-In-Rahmen enthalten ist, den Sie ausbauen können. Darin befinden sich die Routinen zum Erzeugen und Löschen einer Symbolleiste, ein Beispieldialog und die obligatorischen Klassen `clsError` und `clsSettings`.

13.2.1 CommandBar erzeugen

In Excel existiert im Gegensatz zu den anderen Office-Produkten *noch* die Möglichkeit, Symbolleisten von Hand zu erzeugen und an eine Arbeitsmappe anzubinden. Diesen Dialog erreichen Sie über den Anpassen-Dialog (siehe Abbildung 13.1), den Sie per rechter Maustaste aus einer beliebigen Symbolleiste hervorlocken können.

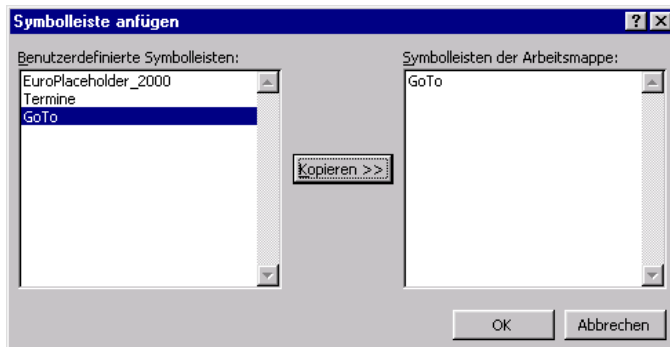


Abbildung 13.1: Dialog Symbolleiste anfügen

Da sich die Office-Produkte immer mehr anpassen, ist jedoch damit zu rechnen, dass diese Möglichkeit irgendwann verschwunden sein wird. Uns steht jedoch über die `CommandBars-Collection` des `Application`-Objekts ein recht flexibles Instrumentarium zur Verfügung, deshalb müssen wir der Möglichkeit des Anbindens von Symbolleisten keine Träne nachweinen.

In `clsSettings` befindet sich die Eigenschaft `ProjectName`, die aus dem Dateinamen den eigentlichen Namen extrahiert, der zum Identifizieren der gleichnamigen Symbolleiste dient, aber auch den Bezug zu der Routine herstellt, die dem `CommandBarButton`-Objekt zugeordnet ist:

```
Public Property Get ProjectName() As String
    ProjectName = Left(ThisWorkbook.Name, _
        Len(ThisWorkbook.Name) - 4)
End Property
```

Die Ereignisroutine `Workbook_Open` instantiiert die `Error`-Klasse und ruft die Routine zum Erzeugen der `CommandBar` auf:

```
Private Sub Workbook_Open()
    On Error GoTo errHandler
    Set cError = New clsError
    Call CreateCommandBar
    Exit Sub

errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "Workbook_Open"
    End If
    cError.ErrorMessage
End Sub
```

Die Prozedur `CreateCommandBar` ermittelt den Projektnamen, also den Dateinamen ohne Dateierweiterung, und den Dateinamen. Um zwischen der `xls`-Datei und der späteren `AddIn`-Datei unterscheiden zu können, wird die Eigenschaft `IsAddin` zu `Rate` gezogen. Diesen Namen benötigen wir zur Zuordnung der auszuführenden Prozedur, die wir der `OnAction`-Eigenschaft des `CommandBarButton`-Objekts zuweisen. Sie sollten die `Caption` und die `FaceID` anpassen, da Sie die Schaltfläche vermutlich nicht mit HalliHallo und dem Smiley geschmückt sehen wollen.

```
Private Sub CreateCommandBar()
    On Error GoTo errHandler
    Dim cb As CommandBar           'CommandBar-Objekt
    Dim ccbDummy As CommandBarButton 'Dummy-Button
    Dim strProjectName As String    'Projektname
```

```

Dim strFileName As String           'Dateiname
On Error GoTo errHandler
'Projektname und Dateiname ermitteln
strProjectName = cSettings.ProjectName
If ThisWorkbook.IsAddin Then
    strFileName = strProjectName & ".xla"
Else
    strFileName = strProjectName & ".xls"
End If
'Prüfen, ob CommandBar bereits existiert
For Each cb In Application.CommandBars
    If cb.Name = strProjectName Then Exit For
Next
If cb Is Nothing Then
    'CommandBar erzeugen
    Set cb = Application.CommandBars.Add( _
        Name:=strProjectName)
    'Button 1 erzeugen
    Set ccbDummy = cb.Controls.Add(msoControlButton)
    ccbDummy.OnAction = strFileName & "!frmDummy_Show"
    ccbDummy.Caption = "Hallihallo"
    ccbDummy.FaceId = 59
    ccbDummy.Style = msoButtonIconAndCaption
End If
cb.Enabled = True
cb.Visible = True
Exit Sub

errHandler:
If InStr(1, Err.Source, ":") = 0 Then
    Err.Source = "CreateCommandBar"
End If
Err.Raise Err.Number
End Sub

```

Abbildung 13.2 zeigt unsere Symbolleiste:



Abbildung 13.2: Unsere Symbolleiste

Der Titel AddIn resultiert aus dem Dateinamen Addin.xls respektive Addin.xla.

Die Routine `frmDummy_Show` bringt den Dialog auf den Bildschirm, der in Abbildung 13.3 dargestellt ist.

13.2.2 Der Dummy-Dialog

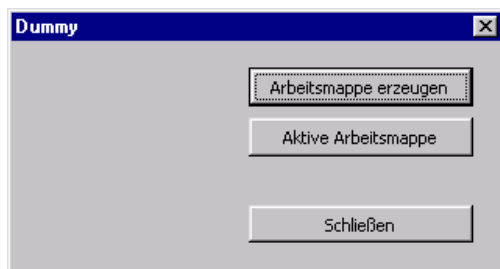


Abbildung 13.3: Der Dummy-Dialog

Die beiden oberen Schaltflächen hinterlassen jeweils den folgenden Ausdruck in einer Tabelle:

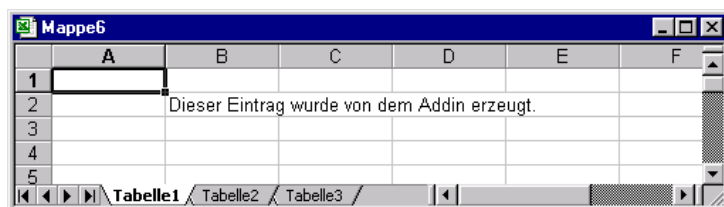


Abbildung 13.4: Spuren des Add-In

Im gebundenen Modul des Dialogs ist eine private Variable des Namens `sht`, die eine Referenz auf das zu bearbeitende Tabellenblatt darstellt. Im einfachsten Fall wird die aktive Tabelle der aktiven Arbeitsmappe referenziert:

```
Private Sub cmdActiveWBK_Click()
    On Error GoTo errHandler
    Set sht = ActiveWorkbook.Worksheets(1)
    sht.Range("B2").Value = "Dieser Eintrag wurde von ..."
    Exit Sub

errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "frmDummy:cmdActiveWBK_Click"
    End If
    cError.ErrorMessage
End Sub
```


Die Worksheet-Variable werden Sie vermutlich benötigen, da Sie öfters auf diese (oder eine andere) Tabelle zugreifen werden.

Die folgende Prozedur, die der oberen Schaltfläche zugeordnet ist, erzeugt zudem eine neue Arbeitsmappe und hinterlässt ihre Spuren in der ersten Tabelle dieser neuen Arbeitsmappe:

```
Private Sub cmdCreateFile_Click()
    On Error GoTo errHandler
    Dim wbk As Workbook          'neues Workbook
    Set wbk = Workbooks.Add
    Set sht = wbk.Worksheets(1)
    sht.Range("B2").Value = "Dieser Eintrag wurde von ... "
    Exit Sub
errHandler:
    If InStr(1, Err.Source, ":") = 0 Then
        Err.Source = "frmDummy:cmdCreateFile_Click"
    End If
    cError.ErrorMessage
End Sub
```

13.2.3 Add-In erzeugen

Bevor Sie die Datei nun als Add-In speichern, sollten Sie im Datei-Menü von Excel den Eigenschafts-Dialog (siehe Abbildung 13.6) aufrufen und dort einen sprechenden Namen für Ihr Add-In angeben. Dieser Name wird später im Add-Ins-Manager angezeigt, wie Abbildung 13.5 zeigt.

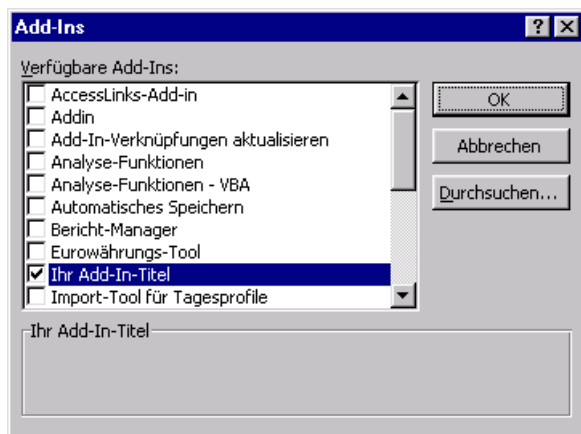


Abbildung 13.5: Add-Ins-Manager

AddIn.xls Eigenschaften ? x

Allgemein Zusammenfassung Statistik Inhalt Anpassen

Titel: Ihr Add-In-Titel

Thema:

Autor: Ihr Name

Manager:

Firma: Ihre Firma

Kategorie:

Stichwörter:

Kommentar:

Hyperlink-basis:

Vorlage: ☐ Vorschaugrafik speichern

OK Abbrechen

Abbildung 13.6: Add-In-Eigenschaften

Kapitelüberblick

14.1	Von VBA nach VB	637
14.2	Voraussetzung für eine Portierung	640
14.2.1	Komponentenprogrammierung	640
14.2.2	Verweise auf Excel-Objekte im Excel-Code	640
14.2.3	Verweis auf die Excel-Bibliothek	641
14.2.4	Verweis auf Excel in Visual Basic	641
14.3	Excel-Objekte im Visual Basic Code	642
14.3.1	Zugriff auf Excel-Objekte im Code	642
14.3.2	Excel-Verweise auflösen	643
14.3.3	Tabellenblätter im OLE-Steuerelement	646

Ist die VBA-IDE eine seitenverkehrte VB-DIE? Schauen Sie selbst:

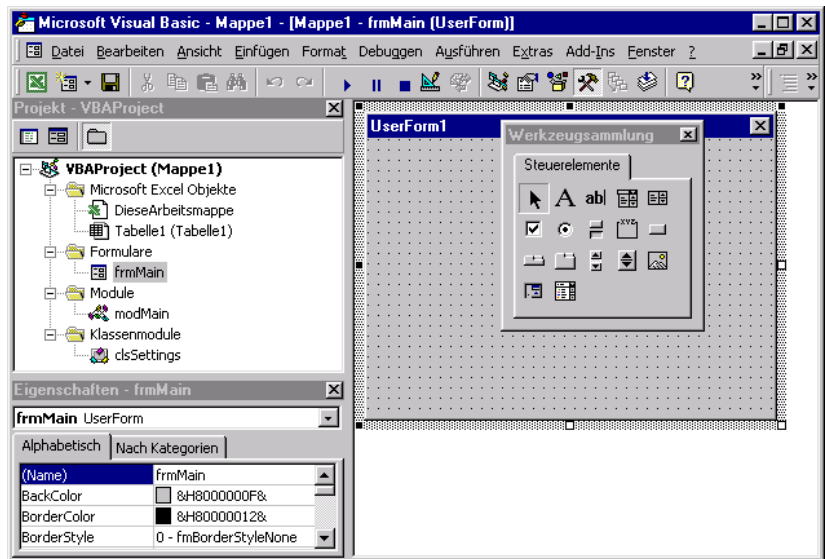


Abbildung 14.1: Die VBA-Entwicklungsumgebung

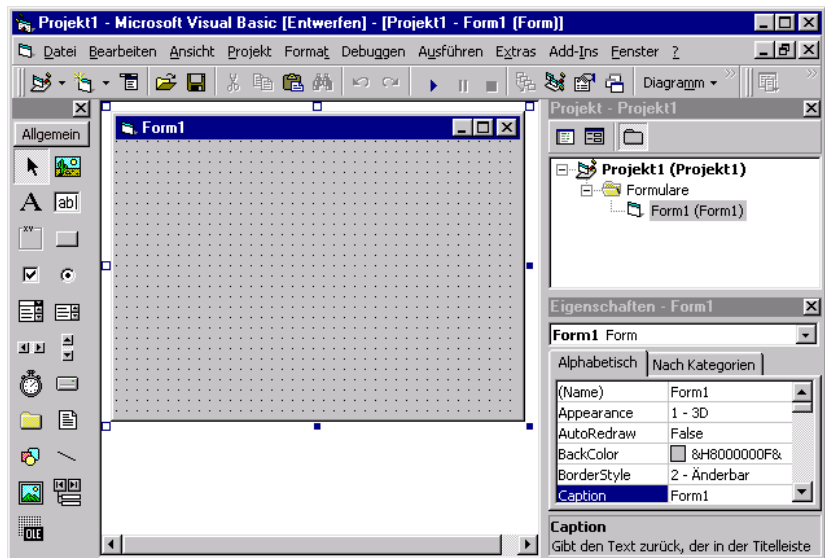


Abbildung 14.2: Die VB-Entwicklungsumgebung

»Ist denen bei Microsoft da was verrutscht?«, ist man geneigt zu fragen, angesichts dieser beiden Screenshots. Was auch immer die Gründe für die spiegelbildliche Anordnung des Projekt-Explorers und des Eigenschaftsfensters auch sein mögen, die Abbildungen zeigen aber eine große Verwandtschaft der beiden Entwicklungsumgebungen. Trotz einiger völlig überflüssiger Unterschiede in der Anordnung einzelner in beiden IDEs vertretenen Menüpunkte sind die Gemeinsamkeiten so groß, dass sich ein Kenner der einen mühelos in der anderen Entwicklungsumgebung zurechtfindet. Sie werden anfangs den Verweise-Dialog beispielsweise an der falschen Stelle suchen, doch das legt sich nach ein paar Jahren ;-)

Doch nicht nur die Entwicklungsumgebung beider Produkte sind sehr ähnlich, auch die Sprache ist es. Seit VBA 6 nun Enumerationen beherrscht, die Deklaration von Events in Klassenmodulen erlaubt oder Arrays als Rückgabe von Funktionen unterstützt, kann man die Sprachen als gleich bezeichnen.

Unterschiede gibt es nach wie vor bezüglich der Formulare, zwar beherrscht VB die Steuerelemente der in VBA verwendeten Forms.2-Collection, allerdings ist uns der umgekehrte Weg verwehrt. Das liegt eben daran, dass die verwendeten Forms schon in einigen Punkten abweichen. Ich nehme einmal an, dass es dafür gute Gründe gibt, hoffe aber gleichzeitig, dass auch hier eine Annäherung in Zukunft stattfinden wird.

Ich möchte mit diesen Gemeinsamkeiten nicht behaupten, dass ein versierter VBA-Entwickler auf Anhieb exzellente VB-Programme schreiben kann. Vielmehr ist es meine Absicht, den VBA-Entwicklern nahe zu legen, sich mit VB zu beschäftigen. Das sprachliche Rüstzeug müsste vorhanden sein, die Entwicklungsumgebungen ähneln sich auch, und der Rest lässt sich lernen. Denn wir werden nun über einige Ansätze reden, die eine Portierung einer VBA-Anwendung nach Visual Basic angeraten sein lassen.

14.1 Von VBA nach VB

Unter den Anlässen, die eine Portierung einer bestehenden VBA-Anwendung nach Visual Basic nahe legen, ist der unerfreulichste wohl in Speicherproblemen größerer Anwendungen anzutreffen. Meldet Ihre Fehlerbehandlungsroutine den Fehler 7 – nicht genügend Speicher, so haben Sie ein Problem, dessen sauberste Lösung die Portierung der Anwendung nach Visual Basic darstellt.

Speicherprobleme

Stellt die Funktionalität von Excel nur eine (ActiveX-)Komponente der Anwendung dar, so kann es gelegentlich auch angeraten sein, Excel und beispielsweise PowerPoint oder Word in den Container Visual Basic zu pflanzen.

Mehrere ActiveX-Komponenten

**Fehlende
Steuerelemente**

Ein weiterer Grund kann darin zu suchen sein, dass man mit schöner Regelmäßigkeit mit den engen Grenzen der Dialoge in VBA konfrontiert wird. Hiermit sind nicht die mitgelieferten Steuerelemente selbst gemeint, die ja so ziemlich alles bieten, was man sich wünschen kann. Es sind die fehlenden Steuerelemente, die in einem von Visual Basic verwöhnten Entwickler die Sehnsüchte wecken.

Anwenderniveau

Excel-VBA-Programme sind für verantwortungsvolle Anwender geeignet, die über ein gerütteltes Maß Grundkenntnisse in Excel verfügen. Hier spreche ich aus Erfahrung: Wenn nur der geringste Zweifel an der Verlässlichkeit der zukünftigen Anwender besteht, sollten Sie von einer Realisierung in Excel-VBA absehen, denn Sie bekommen das Programm mit vertretbaren Aufwänden nicht dicht. Nutzen Sie Excel-Tabellen sogar zur Datenhaltung, potenzieren sich die Schwachpunkte.

Ein Beispiel

Lassen Sie uns ein Beispiel aus der Praxis wählen. Im Laufe der Jahre 1997 bis 1999 schrieb ich eine Reihe von Statistiken für einen großen Filialisten, dessen Konzern in Länder, Regionen, Bezirke und Märkte unterteilt ist. Diese Statistiken folgten genau diesem Mix aus geographischen und organisatorischen Komponenten. So konnte man eine Krankenstatistik konzernweit aufbauen, oder aber auf deutsche Märkte beschränken. Oder man wählte die Region Nord aus oder innerhalb dieser Region Nord den Bezirk des Herrn Heine. Oder aber nur einen Markt des Bezirks Heine.

In VBA sah der Dialog etwa so aus:

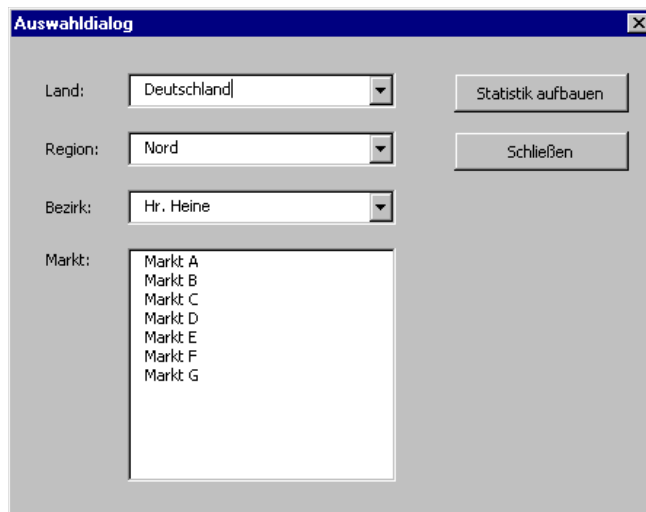


Abbildung 14.3: Beispiel einer hierarchischen Auswahl in VBA

Die Auswahl eines Landes füllt die ComboBoxes Region und Bezirk, die einer Region nur die Bezirke, und die Auswahl eines Bezirks populierte die ListBox. Als klassische Kaskade erlaubt sie keinen Überblick über die Strukturen, sondern nur die eindimensionale Navigation.

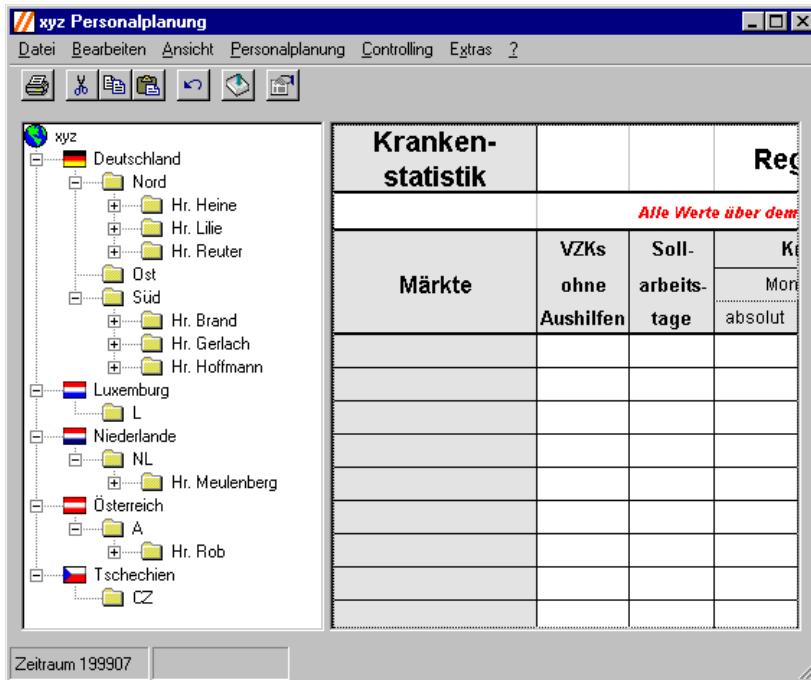


Abbildung 14.4: Beispiel einer hierarchischen Auswahl in Visual Basic

Abgesehen von dem ansprechenderen Design erlaubt diese hierarchische Ansicht einen Überblick über die Strukturen und eine Auswahl per Mausklick, wobei es unerheblich ist, ob der Konzern, ein Land, eine Region, ein Bezirk oder ein einzelner Markt das Ziel der Auswahl ist. In dem rechten Steuerelement, einem sogenannten OLE-Control, können beliebige Excel-Dateien – genau genommen das Blatt, welches beim Speichern aktiviert war – angezeigt werden. Ein Doppelklick öffnet die betreffende Datei.

Es ist weder sinnvoll noch möglich, die Programmstrukturen hier zu schildern, aber Sie können sich die Möglichkeiten vorstellen, die eine solches Interface bietet. Wir werden nun die Aspekte des Designs einer VBA-Applikation beleuchten, die eine reibungslose Portierung in einen VB-Container ermöglicht.

Die Integration der sechs verschiedenen Excel-VBA-Anwendungen in den VB-Container war alles andere als reibungslos, denn die Anfänge lagen in Excel 5, teils in bestehenden Applikationen, teils in neuen. Es ist im Prinzip überflüssig zu erwähnen, aber auch ich wusste damals nicht alles das, was ich heute weiß, und trug meinen Teil durch Designfehler in den VBA-Programmen zu den Problemen bei dieser Portierung bei.

14.2 Voraussetzung für eine Portierung

14.2.1 Komponentenprogrammierung

Dass Datenbankzugriffe in VBA-Projekten und VB-Projekten in Klassen mit neutralen Methoden und Eigenschaften gekapselt sein sollten(!), dürfte jedem einsichtig sein, der einmal in einem Projekt eine Datenquelle gegen eine völlig anders geartete austauschen musste oder auch aufmerksam die bisherigen Ausführungen gelesen hat. Diese Erkenntnis gilt sicherlich auch für den Einsatz von Excel-Dateien als Datenkomponente in VB-Projekten, denn auch hier muss man damit rechnen, dass irgendwann doch die Arbeitsmappen gegen etwas anderes ausgetauscht werden. Und dann ist das Erstellen einer neuen Klasse eine Leichtigkeit, da sie, saubere Codierung vorausgesetzt, keine Auswirkung auf das eigentliche Programm (Programmlogik und Präsentationsschicht) haben kann.

Dieses Thema wurde aber hinreichend diskutiert, weshalb wir uns einem anderen Aspekt zuwenden sollten.

14.2.2 Verweise auf Excel-Objekte im Excel-Code

Eine Reihe von kleineren und größeren Problemen erwartet uns, wenn ein VBA-Projekt in eine VB-Anwendung integriert werden soll. Der Fall, dass Sie bestehende VBA-Modulblätter einfach in ein VB-Projekt einfügen und das Projekt läuft, hat einen rein theoretischen Charakter.

Es beginnt schon damit, dass eine `Workbook`-Referenz wie `ThisWorkbook` mit dem Fehler 1004 – »Anwendungs- oder objektorientierter Fehler« belohnt wird. Verweise auf `ActiveWorkbook` hingegen funktionieren, sofern die laufende Instanz von Excel über ein geöffnetes `Workbook` verfügt. Aber die Verwendung von Excel in einem VB-Projekt erzeugt in der Regel eine neue Instanz von Excel, wodurch `ActiveWorkbook` auch nicht verwendbar sei dürfte.

Vermutlich haben Sie jedoch in Ihrer nun zur Portierung anstehenden Anwendung mit `Worksheet`-Objekten gearbeitet. Wenn Sie das `Worksheet`-Objekt über den Eigenschaftsdialog eingestellt haben, war es einfach da. Auch weitergehende Objekte sind von diesem `Worksheet`-Objekt abgeleitet worden. Somit ist unser Ansatzpunkt das `Worksheet`-

Objekt selbst. Wenn wir die Verweise in dem Visual Basic-Projekt auf diese Worksheet-Variablen neu erzeugen, sind alle nachgeordneten Verweise ebenfalls referenziert.

Bei Verweisen auf ehemals fremde Arbeitsmappen dürfte auch kein Problem auftauchen, da Sie auch hier vermutlich Objektvariablen eingesetzt haben.

14.2.3 Verweis auf die Excel-Bibliothek

Schauen wir uns nun einmal an, wie ein Verweis auf Excel in einem Visual Basic Programm hergestellt werden kann.

14.2.4 Verweis auf Excel in Visual Basic

Wie auch in Excel müssen wir dem Programm mitteilen, dass wir eine fremde Bibliothek benötigen. Abbildung 14.5 zeigt einen Verweis auf die Excel-Bibliothek:

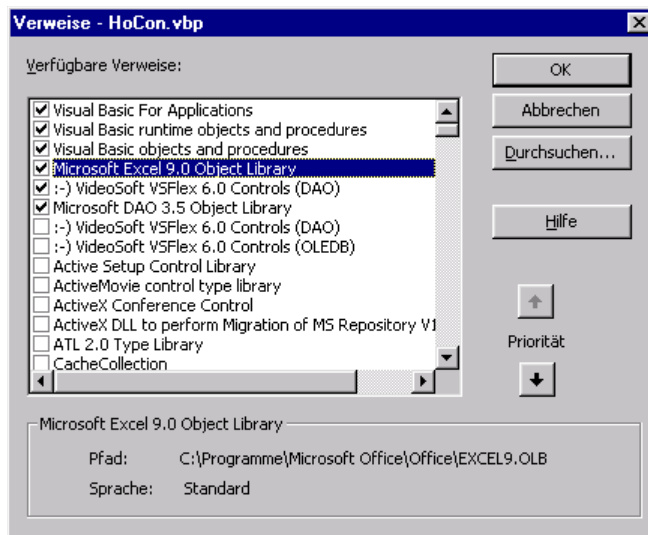


Abbildung 14.5: Verweis auf Excel

Spätestens hier benötigen Sie einen eigenen Rechner zum Erzeugen der Installationsdateien, denn dieses Projekt würde nicht mehr auf einem Rechner laufen, auf dem *nur* Excel 97 installiert ist. Die Entwicklungsumgebung von Visual Basic arbeitet im Gegensatz zu der kompilierten Version unseres Programmes nicht mit der Class-ID der Bibliothek sondern mit dem Namen. Aus diesem Grund können Sie dieses Projekt auf einem Rechner laufen lassen, auf dem nur Excel 97 existiert. Wenn Sie aber eine



Exe daraus bauen, wird dieser Namensverweis gegen die Class-ID ausgetauscht und ein Versionsvergleich durchgeführt. Zwar läuft das Programm später mit einer binärkompatiblen Bibliothek einer höheren Versionsnummer, aber nicht mehr mit einer kleineren Version.

14.3 Excel-Objekte im Visual Basic Code

Wenn man mit Objekt-Referenzen arbeitet stellt der Code in Visual Basic keine wirkliche Herausforderung dar. Es ist nur die logische Fortsetzung dessen, was wir in Excel ohnehin tun. Aber dennoch *kann* das eine oder andere Problem auftauchen. Und darüber möchte ich hier auch reden.

14.3.1 Zugriff auf Excel-Objekte im Code

In der Regel werden Sie mehrere Module oder Klassenmodule mit Excel-Code in Ihrem Projekt haben, weswegen sich eine öffentliche Variable anbietet:

```
Public xlApp As Excel.Application 'Instanz auf Excel
```

An geeigneter Stelle füllen Sie dieses Objekt mit Leben:

```
Set xlApp = New Excel.Application
```

Nachdem Sie so eine Instanz auf Excel erzeugt haben, benötigen Sie vermutlich noch einige Excel-Objekte. Die folgenden Codezeilen deklarieren je ein Workbook-, ein Worksheet und ein Range-Objekt:

```
Private wbkStat As Excel.Workbook 'Workbook Statistikdatei
Private shtStat As Excel.Worksheet 'Tabelle in wbkStat
Private rngStat As Excel.Range 'Range-Objekt in shtStat
```

Nun weisen wir der Workbook-Variablen eine gültige Arbeitsmappe zu. Hier wird die Eigenschaft `xlTemplatePath` unterstellt, die das Verzeichnis der Excel-Dateien zurückgibt. Der Dateiname wird ebenfalls einer Eigenschaft entnommen.

```
strPath = cSettings.xlTemplatePath
strFile = cSettings.xlStatFile
Set wbkStat = cWBK.getWBK(strPath, strFile)
```

Das Öffnen der Datei wird über eine Klasse erledigt, die wir bereits kennen. Allerdings bedarf sie einer kleinen Korrektur, damit sie laufen kann. Die Zeile, in der die Datei letztlich geöffnet wird, erfordert vor der Workbooks-Auflistung natürlich noch den Verweis auf unsere Excel-Instanz, was wir bei der Verwendung der Klasse innerhalb von Excel nicht benötigen:

```
Set getWBK = xlApp.Workbooks.Open(strPath & strFile)
```

Der Verweis der Worksheet- und Range-Objekte wiederum bezieht sich auf das bereits existierende Workbook-Objekt:

```
Set shtStat = wbkStat.Worksheets("Statistik")
Set rngStat = shtStat.Range("A1")
```

Und das war's im Prinzip, der Rest müsste laufen.

14.3.2 Excel-Verweise auflösen

Das ist eigentlich kein Thema, oder sagen wir besser, das sollte kein Thema sein. Denn dafür gibt es einen COM-Mechanismus, der alle Referenzen auf eine Bibliothek sauber mitschreibt und beim Auflösen des letzten Verweises auf die Bibliothek diese automatisch aus dem Speicher entfernt. Ich lade Sie nun zu einem kleinen Abstecher in die COM-Realität ein.

Im ersten Beispiel wird die Instanz auf Excel sauber zerstört:

```
Private Sub DoSomething
    Dim xlApp As Excel.Application
    Set xlApp = New Excel.Application
    ...
    xlApp.Quit
    Set xlApp = Nothing
End Sub
```

Ist diese Routine abgearbeitet, ist die Instanz auf Excel aus der Prozessliste des Task-Managers verschwunden.

Im nächsten Beispiel ebenfalls:

```
Private Sub DoSomething
    Dim xlApp As Excel.Application
    Dim wbkStat As Excel.Workbook
    Set xlApp = New Excel.Application
    Set wbkStat = xlApp.Workbooks.Open("D:\Test.xls")
    wbkStat.Close
    xlApp.Quit
    Set xlApp = Nothing
End Sub
```

Sobald wir allerdings noch ein Worksheet-Objekt hinzufügen, warten die ersten Probleme auf uns:

```

Private Sub DoSomething
    Dim xlApp As Excel.Application
    Dim wbkStat As Excel.Workbook
    Dim shtStat As Excel.Worksheet
    Set xlApp = New Excel.Application
    Set wbkStat = xlApp.Workbooks.Open("D:\Test.xls")
    Set shtStat = wbkStat.Worksheets(1)
    wbkStat.Close
    xlApp.Quit
    Set xlApp = Nothing
End Sub

```

Eigentlich keine aufregenden Veränderungen, nicht wahr? Aber der Task-Manager beinhaltet nach Ende der Routine noch eine Instanz auf Excel, wie man leicht nachprüfen kann. Schuld daran ist natürlich das Worksheet-Objekt `shtStat`, welches beim Erzeugen den internen Referenzzähler inkrementiert hat, ihn aber nicht mehr dekrementierte. Dass ein Worksheet-Objekt zur Klasse `Workbook` gehört und automatisch mit dem Parent `Workbook` zusammen zerstört wird, stellt sich als begründete Hoffnung heraus, der die Excel-Instanz aber nicht nachkommt.

Es ist übrigens nicht entscheidend, ob Sie das `Worksheet`-Objekt per `early binding` an die Excel-Bibliothek binden, auch `shtStat` per `late binding` führt dazu, dass die Excel-Instanz danach durch `Set xlApp = Nothing` nicht zerstört wird.

Fügen wir einfach ein `OLE-Control` zu unserer Form hinzu und lassen dieses Control aktualisieren:

```

Private Sub DoSomething
    Dim xlApp As Excel.Application
    Dim wbkStat As Excel.Workbook
    Dim shtStat As Excel.Worksheet
    Set xlApp = New Excel.Application
    Set wbkStat = xlApp.Workbooks.Open("D:\Test.xls")
    Set shtStat = wbkStat.Worksheets(1)
    wbkStat.Close
    xlApp.Quit
    Set xlApp = Nothing
    OLE.Update
End Sub

```

Das OLE-Control schaut sich nach einer laufenden Instanz von Excel um, findet sie auch und verzichtet somit auf das Erzeugen einer neuen Instanz von Excel. Die existierende ist allerdings mehr tot als lebendig, was sich durch das Erscheinen des Dr. Watson nach `OLE.Update` bemerkbar macht. Wenn der Dr. Watson nicht erscheint, dann lassen Sie den Code spaßeshalber im Schrittbetrieb laufen, und er wird anklopfen. Das Ganze hat natürlich nichts mit dem Schrittbetrieb zu tun, denn zumindest in einem Projekt tauchte das Verhalten in einer ganz normalen Exe auf. Aber um das Problem zu isolieren, reduzierte ich den Code auf die elementaren Teile, die sie hier gesehen haben und stellte überrascht fest, dass Dr. Watson ausblieb. Im Schrittbetrieb wiederum stellte er sich ein, was auf eine wie auch immer geartete Zeitkomponente schließen lässt. Offen gesagt ist mir das Ganze einigermaßen rätselhaft. Noch bin ich mit meiner Lösung auch keineswegs zufrieden, aber sie hilft mit mein Problem zu lösen.

Die Ursache ist offensichtlich ein Referenzzähler des `Worksheet`-Objekts in der Excel-Instanz, der weder durch das Schließen des übergeordneten `Workbook`-Objekts, noch durch Beenden der Excel-Instanz selbst aufgelöst wird.

Es empfiehlt sich somit, den Server im `modMain Public` zu deklarieren, in der Sub `Main` zu erzeugen und im weiteren Programm schön in Ruhe zu lassen. Da Sie ja stabile Programme schreiben wollen, empfiehlt sich vor jeder Verwendung von `x1App` im Code einen Testzugriff auf den Server durchzuführen, den ich `GuaranteeExcelServer` getauft habe:

```
Public Sub GuaranteeExcelServer()
    Dim strDummy As String      'Dummy zum Testen des Servers
    On Error GoTo errHandler
    strDummy = x1App.Name
    Exit Sub

errHandler:
    If (Err.Number = And lngObjectError) = 91 Or _
        (Err.Number = And lngObjectError) = 462 Then
        Set x1App = New Excel.Application
        Exit Sub
    End If
    Err.Raise Err.Number, "modUtil:GuaranteeExcelServer"
End Sub
```

Ist `x1App` nicht mehr unter den Lebenden, so wird der Fehler 91 – »Objektvariable nicht mit Set festgelegt« oder 462 – »Der Remote-Server-Computer existiert nicht oder ist nicht verfügbar« – erzeugt. Hier fragt

man sich jedoch, was das Wort Computer in dieser Fehlermeldung zu bedeuten hat.

14.3.3 Tabellenblätter im OLE-Steuerelement

Wenn Sie nur innerhalb eines Modulkontexts auf Excel-Komponenten zugreifen, so können Sie Excel innerhalb dieses Kontexts `Private` deklarieren. Haben Sie in Ihrem Projekt jedoch beispielsweise das OLE-Control verwendet, so ist auch in diesem Kontext ein Zugriff auf Excel erforderlich, auch wenn Sie ihn nicht per Code bereitstellen.

Wenn das OLE-Control eine Excel-Tabelle anzeigen will, so prüft es, ob Excel als sichtbare Anwendung oder als Prozess läuft. Wenn ja, nutzt es diesen Server zur Erzeugung des Abbilds der Tabelle, die im Metafile-Format angezeigt wird.

Eine Excel-Arbeitsmappe lässt sich mit den folgenden Zeilen zur Laufzeit in einem OLE-Steuerelement darstellen:

```
strFile = App.Path & "\Krankenstatistik.xls"  
OLE.Class = "Excel.Sheet.8"  
OLE.CreateLink strFile
```

Welche der Tabellen der `Krankenstatistik.xls` wird aber nun angezeigt? Es ist die Tabelle, die beim letzten Speichern der Datei selektiert war. Wenn wir also eine bestimmte Tabelle angezeigt haben wollen, so müssen wir die Datei öffnen, die gewünschte Tabelle auswählen, die Datei speichern und schließen. Versuchen wir es damit:

```
'Tabelle >Jahresverlauf< aktivieren  
Set cWBK = New clsWBK  
Set xlApp = CreateObject("Excel.Application")  
Set wbk = cWBK.getWBK(App.Path, "Krankenstatistik.xls")  
Set cWBK = Nothing  
wbk.Worksheets("Jahresverlauf").Select  
wbk.Close savechanges:=True  
xlApp.Quit  
'OLE Laden  
strFile = App.Path & "\Krankenstatistik.xls"  
OLE.Class = "Excel.Sheet.8"  
OLE.CreateLink strFile
```

Obwohl dieser Code durchaus so aussieht, als würde er funktionieren, tut er's aber nicht. Die Ursache liegt in der fett markierten Zeile. Offensichtlich erachtet Excel eine geänderte Tabellenauswahl nicht als Grund, die Datei tatsächlich zu speichern. Der Ausdruck `wbk.Close savechanges:=True` bedeutet offensichtlich, dass Excel nur dann speichert, wenn es dies auch

für nötig erachtet. Die Zeile muss also durch die beiden folgenden Zeilen ersetzt werden.

```
wbk.Save
```

```
wbk.Close
```

Danach wird Ihr OLE-Steuerelement die gewünschte Tabelle anzeigen.



Nur wenige Antworten auf die im Zuge des Buches gestellte Fragen sind wirklich eindeutig zu beantworten. VBA als Sprache ist sehr mächtig, und Objekte haben ebenfalls ein enormes Vokabular.

Vielleicht werden Sie Ihre Antworten nicht in den Lösungen finden. Das heißt aber noch lange nicht, dass sie falsch sind. Vielleicht sind sie besser als die hier präsentierten Lösungen, möglicherweise absolut gesehen, vielfach sicherlich, weil sie ihnen *liegen*, ihnen einleuchtender vorkommen als die hier gezeigten. Wichtig ist in erster Linie, dass sie funktionieren.

Sie werden Ihren eigenen Weg finden. Wenn die Lösung nicht optimal sein sollte, so werden Sie der Erste sein, der es erfährt. Man spürt so etwas.

Excel und Visual Basic for Applications

Hier war gefragt, ob man die Lokomotive auch mit Eigenschaften anstelle der folgenden Methode zu Fahren bewegen könnte:

Lokomotive.Fahren vorwärts, 120 (km/h)

Betrachtet man den Prozess des Beschleunigens als den Abstand zwischen zwei Zuständen, könnte man den ersten Teil, den Stillstand, folgendermaßen beschreiben:

Lokomotive.Geschwindigkeit = 0

Um den Fahrprozess einzuleiten, geben wir zuerst die beabsichtigte Fahrtrichtung an:

Lokomotive.Fahrtrichtung = vorwärts

Durch Definition der (Ziel-)Geschwindigkeit weiß unser Objekt Lokomotive alles, was es wissen muss:

Lokomotive.Geschwindigkeit = 120

Nicht der reine Aufruf der Methode bringt die Lokomotive zum fahren, sondern die detaillierten Befehle, die in dieser Methode stecken. Und ob nun der Code, der hinter der Eigenschaft steckt, prüft ob eine Fahrtrichtung definiert ist, und dann die Einzelbefehle abarbeitet, oder ob eine Methode mit beiden Werten dies tut, ist im Prinzip fast Geschmackssache.

Wie wäre es mit einer Synthese beider Wege:

Lokomotive.Fahrtrichtung = vorwärts

Lokomotive.ZielGeschwindigkeit = 120

Lokomotive.BewegDich

Kapitel 5 – Objekte

Zuerst wird die MessageBox auf dem Bildschirm erscheinen, die in der Ereignisroutine steht, denn das Ende des Berechnens wird unmittelbar das Ereignis auslösen. Und erst, wenn diese Ereignisroutine abgearbeitet ist, wird die Prozedur X weiterbearbeitet.

Kapitel 7 – Sprachelemente, die Zweite

Arrays

Hier war die Frage, wie die Beispielprozedur `FilterTest` abgeändert werden müsste, damit sie das Array `strAcc` über die `Array`-Funktion aufbaut. Die geänderten Zeilen sind hervorgehoben:

```
Sub FilterTest()  
    Dim strAcc As Variant  
    Dim strCostAcc() As String  
    Dim iAcc As Long  
    Dim strMessage As String  
    strAcc = Array("Fahrzeugkosten", "Bürokosten", _  
               "Projekterlöse", "Verkaufserlöse", "Lohnkosten")  
    strCostAcc = Filter(strAcc, "kosten", True)  
    For iAcc = 0 To UBound(strCostAcc)  
        strMessage = strMessage & strCostAcc(iAcc) & vbCrLf  
    Next  
    MsgBox strMessage  
End Sub
```

Der wichtigste Unterschied betrifft wohl den Umstand, dass die `Array`-Funktion einen `Variant` links des Gleichheitszeichens erwartet.

Registry-Zugriffe

In dieser Aufgabe ging es darum, die Routine zu codieren, in der alle in der Tabelle aufgelisteten Einträge in der Registry gelöscht werden.

Die Prozedur unterscheidet sich in drei Punkten von der Routine `cmdDeleteCurrentSetting_Click`, die den Eintrag der aktuellen Zeile aus der Registry entfernt:

- ▼ `iRow` wird nicht über die Funktion `GetValidatedRow` ermittelt, da alle Zeilen gelöscht werden sollen. Die Zeile entfällt also.
- ▼ Eine For-Next-Schleife über den belegten Zellbereich bildet statt dessen den Zeilenzeiger `iRow`.
- ▼ Im `ErrorHandler` wird nach der Fehlerausgabe mit `cError.InfoMessage` durch `Resume Next` die weitere Bearbeitung wieder aufgenommen. Aber das konnten Sie nicht wissen, sofern Sie Kapitel 9 noch nicht durchgearbeitet haben.

```
Private Sub cmdDeleteAllListedSettings_Click()
    ' Alle aufgelisteten Einträge aus Registry löschen
    On Error GoTo errHandler
    For iRow = 2 To rngReg.CurrentRegion.Rows.Count
        strSection = rngReg.Cells(iRow, iColSection).Value
        strKey = rngReg.Cells(iRow, iColKey).Value
        DeleteSetting strAppName, strSection, strKey
    Next
Exit Sub

errHandler:
    If Err.Number = 5 Then
        cError.InfoMessage "Der Key " & strKey & _
            " in Section " & strSection & " konnte " & _
            "nicht gelöscht werden.", "Vermutlich " & _
            "existiert er nicht mehr."
        Resume Next
    End If
    Err.Source = "shtReg:cmdDeleteAllListedSettings_Click"
    cError.ErrorMessage
End Sub
```

Rekursionen

Die `FindFile`-Funktion ermöglicht keine Wildcard-Suche, da der Vergleich mit dem Gleichheitsoperatoren durchgeführt wird:

```
ElseIf UCase(strPath) = UCase(strFile) Then
```

Setzen Sie stattdessen den `Like`-Operator ein, und die Wildcard-Suche funktioniert anstandslos:

```
ElseIf UCase(strPath) Like UCase(strFile) Then
```

Ganz ehrlich: Haben Sie daran gedacht, eine eigene Funktion dafür zu schreiben?



Anhang

A

Kapitelüberblick

A.1	Abkürzungen und Begriffe	654
A.2	KeyCode-Konstanten	656



A.1 Abkürzungen und Begriffe

ActiveX	COM-basierte Technologie, die das Zusammenwirken von Objekten, also Codekomponenten, regelt. Für den Begriff ActiveX stand die Marketingabteilung Pate, nicht die Entwicklung. Technologisch ist ActiveX identisch mit OLE 2.
ActiveX-Client	Eine COM-basierte Codekomponente, die einen ActiveX-Server verwendet.
ActiveX-Server	Eine COM-basierte Codekomponente, die über öffentliche Methodenschnittstellen verfügt, die von anderen COM-basierten Codekomponenten genutzt werden können. Die gebräuchlichsten ActiveX-Server in VBA-Projekten stellen wohl die Klassenmodule dar. Aber auch ActiveX-DLL-Server (z.B. ADO oder DAO) oder ActiveX-EXE-Server (z.B. Word oder PowerPoint) können in VBA benutzt werden. Genau genommen sind auch Steuerelemente oder Formulare ActiveX-Server.
ADO	ActiveX Data Objects
	Auf OLE DB basierendes Objektmodell, das einen bequemen Zugang zu Datenbanken ermöglicht. ADO löst die in die Jahre gekommene DAO ab.
ANSI	American National Standards Institute
	Amerikanische Normungsbehörde. ANSI wird in der Regel zusammen mit dem Wort-Zeichensatz gebraucht und meint damit den von ANSI genormten aus 256 Zeichen bestehenden Zeichensatz. Seit Windows ist ANSI zum Standardzeichensatz auf PCs geworden und löst den ASCII (American Standard Code for Information Interchange) ab.
API	Application Programming Interface
	Öffentliche Methoden in C-Bibliotheken, die wir verwenden können.
COM	Component Object Model
	Technologie, in der das Zusammenspiel von (Windows-) Bibliotheken ermöglicht wird.
DAO	Data Access Objects
	In DAO350.DLL für Office 97 implementiertes Objektmodell, welches die JET und die ODBC kapselt.
DCL	Data Control Language
	Der Teil der SQL-Sprache, der sich mit der Steuerung von Berechtigungen von Datenbankobjekten auseinander setzt.
DDL	Data Definition Language
	Der Teil der SQL-Sprache, der Erzeugen, Verändern und Löschen von Datenbankstrukturen beinhaltet.

DLL	Dynamic Link Library
	Bibliothek, die dynamisch, also zur Laufzeit des Programms, in den eigenen Prozess eingebunden wird.
DML	Data Manipulation Language Der Teil der SQL-Sprache, der das Anfügen, Verändern und Löschen von Datenbankinhalten ermöglicht.
IDE	Integrated Development Environment Die Entwicklungsumgebung in (z. B.) VBA
NLS	National Language Support Dieses API stellt einen Zugang zu allen Parametern der Ländereinstellungen der Systemsteuerung dar.
ODBC	Open Database Connectivity Etwas betagte Datenbankschnittstelle, die durch OLE DB abgelöst wird.
OLE	Object Linking and Embedding Mit Windows 3 wurde das Ende des programmzentrierten Paradigmas in Windows durch einen dokumentenzentrierten Ansatz ersetzt. Das damalige OLE 1 ermöglichte beispielsweise einem Word-Dokument eine Excel-Tabelle aufzunehmen, darzustellen und auch quasi im Dokument zu bearbeiten. Mit OLE 2 wurden die auch schon bei OLE 1 im Wesentlichen verwendeten Mechanismen offengelegt und um OLE-Automation erweitert. OLE-Automation erlaubt Programmen über Schnittstellen mit Fremdoobjekten zu arbeiten, ohne deren Implementierung zu kennen. Sie können über diese Mechanismen zum Beispiel Excel per Code eine Datenreihe übergeben und die von Excel berechnete Standardabweichung entgegennehmen, ohne zu wissen, wie Excel im Detail an diese Weisheiten gelangt.
OLE DB	OLE-basierte Datenbankschnittstelle (Richtiger wäre COM-basierte Datenbankschnittstelle, da COM die Basis dieser Technologie darstellt, und OLE begrifflich zunehmend durch COM bzw. ActiveX ersetzt wird.) OLE DB ermöglicht den Zugriff auf alle möglichen Datenquellen, für die ein so genannter OLE DB-Provider zur Verfügung steht.
OLTP	Online Transaction Processing OLTP-Datenbanken bilden den Stand eines Unternehmens zu einem gewissen Zeitpunkt ab. Üblicherweise arbeiten viele Benutzer gleichzeitig an den Datenbeständen.
RDO	Remote Data Objects Bibliothek zum Zugriff auf entfernte Daten

SQL	Standard (Structured) Query Language
	SQL ist eine von IBM ursprünglich für Großrechner entworfene Datenbanksprache, die seit ihrer Entwicklung Anfang der 80er Jahre eine sehr große Verbreitung auf verschiedenen Plattformen erlangt hat. Obwohl der Kern von SQL inzwischen in einer ANSI-Norm festgelegt ist, gibt es jedoch noch einige herstellerabhängige Varianten.

A.2 KeyCode-Konstanten

Ich gestehe, diese Informationen aus der MSDN abgeschrieben zu haben. Die Tasten, deren Bedeutung mir unbekannt ist, habe ich in Originalschreibweise gelassen, z. B. CANCEL oder CLEAR.

Hier nun zuerst die Sondertasten:

Konstante	Hexadezimalwert	Beschreibung
vbKeyLButton	0x1	linke Maustaste
vbKeyRButton	0x2	rechte Maustaste
vbKeyCancel	0x3	CANCEL
vbKeyMButton	0x4	mittlere Maustaste
vbKeyBack	0x8	Rück-Taste
vbKeyTab	0x9	Tabulator-Taste
vbKeyClear	0xC	CLEAR
vbKeyReturn	0xD	Return-Taste
vbKeyShift	0x10	Umschalt-Taste
vbKeyControl	0x11	Steuerungs-Taste
vbKeyMenu	0x12	Menü-Taste
vbKeyPause	0x13	Pause-Taste
vbKeyCapital	0x14	CapsLock-Taste
vbKeyEscape	0x1B	Escape-Taste
vbKeySpace	0x20	Leerzeichen-Taste
vbKeyPageUp	0x21	Bild-Auf-Taste
vbKeyPageDown	0x22	Bild-Ab-Taste
vbKeyEnd	0x23	Ende-Taste
vbKeyHome	0x24	Pos.-1-Taste
vbKeyLeft	0x25	Pfeiltaste nach links
vbKeyUp	0x26	Pfeiltaste nach oben
vbKeyRight	0x27	Pfeiltaste nach rechts

Konstante	Hexadezimalwert	Beschreibung
vbKeyDown	0x28	Pfeiltaste nach unten
vbKeySelect	0x29	SELECT
vbKeyPrint	0x2A	Druck-Taste
vbKeyExecute	0x2B	EXECUTE
vbKeySnapshot	0x2C	SNAPSHOT
vbKeyInsert	0x2D	Einfügen-Taste
vbKeyDelete	0x2E	Entfernen-Taste
vbKeyHelp	0x2F	HELP
vbKeyNumlock	0x90	NumLock-Taste

Der nächste Block enthält die Buchstabentasten, wobei hier nicht zwischen Groß- und Kleinschreibung unterscheiden wird. Diese Information müssen Sie dem Shift(-Mask)-Argument entlocken. Nationale Sonderzeichen sind nicht mit Konstanten versorgt.

Konstante	Dezimalwert	Beschreibung
vbKeyA	65	A-Taste
vbKeyB	66	B-Taste
vbKeyC	67	C-Taste
vbKeyD	68	D-Taste
vbKeyE	69	E-Taste
vbKeyF	70	F-Taste
vbKeyG	71	G-Taste
vbKeyH	72	H-Taste
vbKeyI	73	I-Taste
vbKeyJ	74	J-Taste
vbKeyK	75	K-Taste
vbKeyL	76	L-Taste
vbKeyM	77	M-Taste
vbKeyN	78	N-Taste
vbKeyO	79	O-Taste
vbKeyP	80	P-Taste
vbKeyQ	81	Q-Taste
vbKeyR	82	R-Taste
vbKeyS	83	S-Taste

Konstante	Dezimalwert	Beschreibung
vbKeyT	84	T-Taste
vbKeyU	85	U-Taste
vbKeyV	86	V-Taste
vbKeyW	87	W-Taste
vbKeyX	88	X-Taste
vbKeyY	89	Y-Taste
vbKeyZ	90	Z-Taste

Die nächste Gruppe umfasst die numerischen Tasten:

Konstante	Dezimalwert	Beschreibung
vbKey0	48	0-Taste
vbKey1	49	1-Taste
vbKey2	50	2-Taste
vbKey3	51	3-Taste
vbKey4	52	4-Taste
vbKey5	53	5-Taste
vbKey6	54	6-Taste
vbKey7	55	7-Taste
vbKey8	56	8-Taste
vbKey9	57	9-Taste

Die folgenden Konstanten betreffen die Tasten auf dem numerischen Tastaturblock:

Konstante	Hexadezimalwert	Beschreibung
vbKeyNumpad0	0x60	0-Taste
vbKeyNumpad1	0x61	1-Taste
vbKeyNumpad2	0x62	2-Taste
vbKeyNumpad3	0x63	3-Taste
vbKeyNumpad4	0x64	4-Taste
vbKeyNumpad5	0x65	5-Taste
vbKeyNumpad6	0x66	6-Taste
vbKeyNumpad7	0x67	7-Taste
vbKeyNumpad8	0x68	8-Taste

Konstante	Hexadezimalwert	Beschreibung
vbKeyNumpad9	0x69	9-Taste
vbKeyMultiply	0x6A	Multiplikationszeichen (*)
vbKeyAdd	0x6B	Plus-Taste (+)
vbKeySeparator	0x6C	Return-Taste
vbKeySubtract	0x6D	Minus-Taste (-)
vbKeyDecimal	0x6E	Dezimalzeichen (.)
vbKeyDivide	0x6F	Divisionszeichen (/)

Nun die Funktionstasten:

Konstante	Hex-Wert	Beschreibung
vbKeyF1	0x70	F1-Taste
vbKeyF2	0x71	F2-Taste
vbKeyF3	0x72	F3-Taste
vbKeyF4	0x73	F4-Taste
vbKeyF5	0x74	F5-Taste
vbKeyF6	0x75	F6-Taste
vbKeyF7	0x76	F7-Taste
vbKeyF8	0x77	F8-Taste
vbKeyF9	0x78	F9-Taste
vbKeyF10	0x79	F10-Taste
vbKeyF11	0x7A	F11-Taste
vbKeyF12	0x7B	F12-Taste
vbKeyF13	0x7C	F13-Taste
vbKeyF14	0x7D	F14-Taste
vbKeyF15	0x7E	F15-Taste
vbKeyF16	0x7F	F16-Taste



Stichwortverzeichnis

A

Activate-Ereignis 236, 246, 254, 277, 387, 415
Activate-Methode 201–202, 241
ActiveSheet 178, 184, 192, 224, 279
ActiveWorkbook 177–178, 184, 192, 530, 632, 640
ActiveX 16, 64–65, 89, 101, 168, 196, 238, 248, 337, 371, 404, 486, 490, 510–511, 586, 596, 637, 654–655
ActiveX Data Objects 60, 64–66, 482, 586, 589, 596–599, 611, 614, 619, 623, 654
AddComment-Methode 199, 202, 207, 209
AddControl-Ereignis 403–404
AddCustomList-Methode 282
AddIns-Auflistung 265, 281, 283–284
AddItem-Methode 427–428
Add-Methode 190, 253, 262, 264, 284, 314, 622
Adressierung 218, 257
Aggregation 210, 605, 607
AltStartupPath-Eigenschaft 281, 284
API (Application Programming Interface) 79, 117, 153, 318, 324, 337, 339, 343–344, 347, 385, 405, 466–467, 501, 550, 554, 569–570, 654–655
API-Aufruf 467
Apostroph in SQL-Statement 613

Application-Objekt 18–19, 49–50, 95, 125, 170, 176–177, 190–192, 196, 202, 204, 207, 211, 233, 236, 244, 253–257, 260, 263, 275, 281, 283–296, 298, 330–331, 337, 363, 365, 486–487, 493, 518–519, 523, 548, 570, 616–617, 630–631, 642–646, 654
Applications 15–16, 60, 536, 595, 649
Array-Element 216, 304–305, 310, 312
Auflistung (Collection) 184, 199–201, 206, 243, 275, 314–317, 349, 402, 595–596, 605, 630, 637
Auflistungen 176, 184–187, 189–190, 197–198, 245, 275, 302, 314, 589
Aufrufhierarchie, Fehlerbehandlung 455–457, 464
AutoFilter-Methode 199, 203–206, 209, 217, 236, 518

B

BackColor-Eigenschaft 391–393
BeforeDragOver-Ereignis 404–407
Binomialkoeffizient 361
BorderAround-Methode 206
BorderColor-Eigenschaft 392
Border-Objekt 199, 205–206
Borders-Eigenschaft 205–206
BorderStyle-Eigenschaft 392, 395

C

Calculate-Methode 19, 173, 175, 251, 261, 281, 284, 493

Calculation-Eigenschaft 281, 284–285

Cells 44, 103–105, 107, 118, 146, 171, 183, 185, 193–194, 198–199, 201, 204, 206–210, 213, 215–217, 219, 221–223, 227, 233–234, 238, 241–243, 247–248, 250, 255, 258–259, 273, 295, 299–300, 310–311, 325–326, 328–329, 336, 355, 358, 365, 367, 438, 471, 483, 493, 504–505, 513, 521–522, 524, 526–527, 544, 553, 566, 568, 576, 596, 600, 651

Cells-Auflistung 185, 198, 201, 206–207, 209, 217, 233–234, 242, 273

Cells-Collection 206

Cells-Methode 104, 208, 576

Change-Ereignis 203, 257, 335, 393, 408, 413, 428, 436, 513

Characters 199, 208, 217

Characters-Eigenschaft 208

Characters-Objekt 217

ClearComments-Methode 199, 202, 207–208

Click-Ereignis 101, 408, 412, 418, 434–435, 572–573

Close-Methode 263, 270

Codeüberprüfung 61

Color-Eigenschaft 206, 332–333, 335–336

ColorIndex-Eigenschaft 206, 223–225, 227, 259, 332, 335–336

Column-Eigenschaft 194, 199, 204, 208–209, 218, 248, 256–257, 260, 526–527, 568

Columns-Eigenschaft 209, 242

ColumnWidth-Eigenschaft 209

COM 643

ComboBox 30, 32, 45, 47, 49, 58, 68, 101–102, 246, 342, 372, 379, 387, 392, 394–395, 424–433, 494, 499, 537–538, 548, 572, 618, 620, 622, 624, 639

CommandBars-Auflistung 47, 253–256, 281, 285, 294–298, 630–631

Comment-Eigenschaft 209

Comment-Objekt 199–200, 202, 207, 209–210, 238, 243

Connection 247

Connection-Objekt 247, 538, 590–591, 593–594, 597–599, 610–611, 614, 622

Controls 89, 186, 248, 253, 255–256, 295–296, 299, 314, 402, 411, 419–420, 631

ControlSource-Eigenschaft 392–393

ControlTipText-Eigenschaft 393, 549

Copy-Methode 206, 210–211, 217, 244

Count-Methode 209, 234

CreateWorkspace-Methode 593

CSV-Dateien 248

CurrentRegion 107, 170, 200, 204–206, 209, 212, 215–216, 219, 222, 227, 229–230, 233–234, 236, 246, 249–250, 255, 259, 310–311, 324–327, 329, 355, 358, 365, 423, 471–472, 484, 651

CurrentRegion-Methode 204, 212

CustomListCount-Eigenschaft 282, 285, 287

CutCopyMode-Eigenschaft 211, 255, 282, 285

D

Database-Objekt 590–591, 594

DbClick-Ereignis 408

Delete 19, 187, 200, 208, 212–213, 222, 236–238, 244, 249, 254–255, 299, 326, 624

DELETE-Anweisung (SQL) 611

DeleteCustomList-Methode 282, 285

Delete-Methode 208, 212–213, 244

DisplayCommentIndicator-Eigenschaft 282, 286, 290

DisplayFormulas-Eigenschaft 277

DisplayHeadings-Eigenschaft 277

DisplayHorizontalScrollbar-Eigenschaft 277

DisplayOutline-Eigenschaft 277

DisplayPageBreaks-Eigenschaft 277

DisplayVerticalScrollbar-Eigenschaft 277

DisplayWorkbookTabs-Eigenschaft 277

- DisplayZeros-Eigenschaft 277
- DLL 53, 60, 65, 89, 168, 339, 450, 460, 476, 478, 529, 531, 654–655
- DoEvents 365–367, 548
- Do-Loop-Schleife 135–137, 300, 350, 366, 600
- E**
- Eigenschaftsprozeduren 171, 502, 505
- Enabled-Eigenschaft 393–394, 402, 420
- Enter-Ereignis 387, 408, 418
- EntireColumn 198–200, 204, 213, 215–217
- EntireColumn-Eigenschaft 213
- EntireRow 200, 204, 208, 213, 223, 227, 527
- EntireRow-Eigenschaft 213
- Entwicklungsumgebung 24
- Enumeration 48, 225–226, 333–334, 341–342, 494–495, 502–503, 506, 517–518, 545–546
- Excel-Objektmodell 11, 16, 49, 113, 161, 187, 189, 192, 196, 198, 586
- Execute-Methode 610–611, 613, 621–624
- Exit-Ereignis 387, 408, 621–622
- F**
- Fehler 1004 640
- Fehler 462 645
- Fehlerbehandlung 11, 13, 27, 70, 75, 324, 409, 441–445, 447–450, 452–453, 456–458, 462, 465, 474, 479, 506, 512, 529, 535
- Fehlerbehandlung, Aufrufhierarchie 455–457, 464
- Fehlerbehandlungsroutine, aktivierte 457, 459, 461, 464, 507, 509
- Fehlerbehandlungsroutinen 70, 314, 324, 329, 449–450, 452, 455, 457–458, 461–462, 464, 470, 477, 479, 495, 505, 507, 509–510, 512, 528, 591, 651
- Fields-Collection 595–596, 605
- FindFile-Methode 282, 286, 363–366, 651
- Find-Methode 173, 204, 209, 213–216, 259, 269, 474–475, 481, 504, 521–522, 545, 552, 567–568
- FindNext 216–217
- Font-Eigenschaft 217, 393, 399
- Font-Objekt 200, 208, 217, 397–398, 423
- For-Each-Schleife 187, 207, 272, 316
- Formula-Eigenschaft 217–218
- FormulaLocal-Eigenschaft 217–218
- For-Next-Schleife 104, 132–134, 136, 140, 204, 207, 212, 215, 220–221, 234, 340, 445–447, 451, 596, 600, 651
- Function 86, 121–126, 129–130, 136, 140–141, 236, 266, 272–273, 286, 307–308, 310–312, 325, 331, 333–334, 339, 341–344, 347–348, 354, 360–361, 444–452, 454, 460, 466, 475, 482–483, 490, 493–494, 496, 502, 504, 521, 540, 546, 552, 554–557, 566, 568, 591, 611–614, 619–620, 623
- Functions 529, 535, 628
- G**
- GetCustomListNum-Methode 282–283, 285, 287
- GetCustomListContents-Methode 282
- GetLocaleInfo (NLS-API) 347–348, 569
- GetOpenFilename-Methode 282, 287–288
- GetSaveAsFilename-Methode 282, 288–289
- GridlineColor-Eigenschaft 277
- GridlineColorIndex-Eigenschaft 277
- H**
- HasFormula-Eigenschaft 218
- Height-Eigenschaft 226, 379–380, 394
- Steuerelemente 394
- I**
- IDE (Integrated Development Environment) 24
- Initialize-Ereignis 291, 387, 494, 496, 507–508, 510, 558, 565, 570

Initialize-Ereignis (Klassenmodul) 291,
466–469, 481, 494–496, 503, 507–
509, 517, 519, 524, 567

InputBox-Funktion 161, 165

Instantiierung (von Klassen) 19, 505

ISAM (Index Sequential Access Method)
93, 586, 588–589

Item-Methode 186–187, 191, 206, 237,
242–243, 262–263, 271–272, 283,
315–316, 554

J

JET-Datenbanken 595

K

Kapselung 187–188, 314, 344, 487, 625

KeyCode-Konstanten 409, 656

KeyDown-Ereignis 409–410, 412, 420

KeyPress-Ereignis 410

KeyUp-Ereignis 289, 410

Klassen 11, 13, 28, 47–49, 59, 70, 94,
117, 121, 126, 168–169, 185–186,
188–189, 193, 220, 225, 243, 253,
265, 291–292, 317, 324, 327, 332,
337, 344, 346–347, 386, 391, 433,
442, 444, 450, 455, 467–468, 472,
481–482, 486–491, 493–494, 496–
497, 499–502, 505–513, 516–517,
519–525, 528, 538, 545–546, 551–
554, 557–559, 564–567, 569, 571–
574, 597, 604, 611, 613, 615–617,
619, 621–622, 624–625, 629–630,
640, 642, 644

Knowledge Base 21

L

Ländereinstellungen (Systemsteue-
rung) 151, 154–155, 217, 228, 269,
349, 550, 655

Layout-Ereignis 400, 411

Left-Funktion 62, 138–140, 142, 145,
200, 220, 222, 227, 269, 299–300,
340, 348, 394, 399, 401, 414, 451,
467–468, 555–556, 622, 624, 630

Let 471, 491–492, 499–500, 553, 569,
612, 621

LibraryPath-Eigenschaft 282, 284, 289

LineFeed 145

LineStyle-Eigenschaft 205–206

ListBox 90, 338, 372, 392, 395, 405,
417–418, 420–421, 427–428, 430–
432, 480, 571–572, 590, 639

ListBoxes 11, 49, 430–431, 433

M

Max-Eigenschaft 414

Member 206

Microsoft Developer Network 20

MouseButton-Konstanten 411

MouseDown-Ereignis 411

MouseMove-Ereignis 401, 404, 412

MouseUp-Ereignis 412

MSDN 20–21

MultiSelect 288, 392, 431–432

N

Name-Eigenschaft 98, 171–172, 177,
185, 197, 239–240, 271, 398–399,
517, 616, 622

Name-Objekt 271–273

Names-Auflistung 270–272

National Language Support (NLS) 153,
347, 558, 569, 655

New 487

Nothing 120, 182, 207, 209–210, 214–
217, 259, 272, 296, 299, 471–472,
474, 481–482, 496–498, 504, 509–
510, 521–522, 530, 546, 552, 566–
567, 631, 643–644, 646

O

ODBC (Open Database Connectivity)
247, 558, 588–595, 597, 654–655

ODBC Data Source Manager 592–593

ODBC-Datenbanken 589, 595

OLE DB 597–598, 600, 654–655

OnKey-Methode 282, 286, 289–290

OpenConnection-Methode 593–594

Open-Methode 11, 65, 190, 264, 267,
453, 598–599

OpenRecordset-Methode 210, 220, 247,
594–595, 598

OpenText-Methode 267, 349

ORACLE-Datenbanken 65, 586, 589

Outlook 430

P

Palette-Schaltfläche 206, 223
 Parent 192, 200, 230, 238, 299, 644
 Parent-Eigenschaft 230
 PasteSpecial-Methode 34–35, 200, 211, 231–232, 255, 285
 Persistenz 214
 Polymorphismus 391
 Präfixbildung 125, 192–193, 536–537
 Property Get 348, 483, 491–493, 502–503, 505, 519, 521–525, 553, 555, 569, 612, 615–617, 621, 630
 Property Let 471, 491–492, 499–500, 553, 569, 612, 621
 Property Set 546
 Punkt 394

Q

Quit-Methode 282, 290, 643–644, 646

R

Range-Methode 232–233, 438
 Range-Objekt 12–13, 18–19, 49–51, 62, 94–99, 101, 103, 107, 118, 145, 169–170, 173, 178–180, 183, 189–194, 196–203, 205–217, 219, 222–236, 238–239, 241–243, 247–249, 254–257, 259–260, 269, 271–273, 282–283, 287, 293–294, 300, 310–311, 324–326, 333, 335–336, 365, 438, 471–472, 474–475, 481–482, 490, 493, 504, 517–519, 521–523, 526–527, 538, 541, 545–546, 549–550, 552, 566–568, 576, 595, 632–633, 642–643
 Range-Variable 207, 215
 RecordCount 600
 RecordCount-Eigenschaft 600
 Records 594
 Recordset 65, 135, 210, 219, 247
 Recordset-Objekt 65, 135, 210, 219, 247, 538, 541, 591, 594–601, 606–608, 610–611, 613, 615, 619, 622–623
 Redundanzvermeidung 575
 RefersToRange-Eigenschaft 273, 355, 358, 365, 526, 550, 620

Rekursion 125–126, 360–361
 RemoveControl-Ereignis 412
 RGB-Array 206
 RGB-Farben 206, 332–336, 392
 Row 44, 103–105, 146, 194, 200, 207, 215–218, 223, 233, 248, 250, 257, 259, 295, 299–300, 325–326, 328–329, 336, 355, 358, 471, 474, 481–482, 504–505, 522, 524, 526–527, 552, 566, 596, 600, 651
 Row-Eigenschaft 194, 215, 233, 474, 552, 566
 Rows-Methode 213

S

SaveAs-Methode 248
 SaveCopyAs-Methode 274
 Save-Methode 274
 ScreenUpdating-Eigenschaft 18, 282, 290–291
 Scroll-Ereignis 413, 436
 SELECT-Anweisung (SQL) 602–603
 Shift-Konstanten 409
 Shortcuts 32, 212
 Sort-Methode 234, 287, 430, 490
 Spaltenzeiger 115, 194, 204, 207, 209, 218–219, 221, 298, 300, 324, 336, 353, 504, 517–518, 521, 541, 544, 553, 566–569, 576
 SpecialCells-Methode 217, 235, 490
 SpinDown-Ereignis 436
 SpinUp-Ereignis 436
 SQL (Structured Query Language) 65, 123, 141, 203, 209–210, 219, 538, 585–587, 589, 592, 594–595, 599, 601–603, 605–606, 608–609, 612–613, 617, 622, 654–656
 SQL-Server 65, 586, 589, 592, 601
 StartupPath-Eigenschaft 282, 291
 StatusBar-Eigenschaft 260, 282, 290–291, 363, 365, 548
 Systemsteuerung 154–155, 178, 228, 332, 381, 408, 655

T

Tabellenfunktionen in VBA 292–294
 TemplatesPath-Eigenschaft 282, 291
 Terminate-Ereignis (Klassenmodul)
 344, 493–498, 509–510
 TextBox 226, 338, 372, 378–380, 386–
 389, 392, 394–395, 404, 408–409,
 417, 420–421, 423–424
 Text-Methode 202
 Typelibrary 199

U

Ungarische Notation 536
 UPDATE-Anweisung (SQL) 609–610
 UsedRange 134, 210–211, 239, 249–
 250, 299, 568

V

Value 18, 44, 96, 98–99, 103–105, 107,
 118, 169–171, 179, 183, 201–202,
 204, 207, 210, 215–218, 220–222,
 224, 227, 236, 241–243, 256–259,
 273, 295, 299, 310–311, 320, 322,
 325–326, 328–329, 335–336, 354–
 355, 358, 365, 367, 392, 402, 413,
 433–436, 438, 471, 482–483, 504–
 505, 513, 521–522, 524, 526–527,
 544, 549–550, 553–554, 566, 568,
 595–596, 599–600, 605, 611, 613,
 620, 632–633, 651
 Variants 118, 476, 482, 536
 VBA-Hilfe 19–21, 407
 VBAListe.xls 292–293
 Verankerbar 29

W

Weight 205–206, 227, 399
 WHERE-Klausel (SQL) 203, 604, 607,
 609, 611
 WinAPI-Viewer 79, 337–338
 With-Objektbezug 182–183, 223
 Workbook-Objekt 18–19, 50, 73–74, 94,
 96–97, 170, 176, 181–182, 190–193,
 196, 238–239, 244, 253–254, 256,
 261–263, 266, 270–271, 274, 277–
 281, 286, 290, 292, 296, 321–322,
 390–391, 444, 450–453, 455, 467–
 468, 502, 504–505, 520, 529–530,
 538, 614, 630, 633, 640, 642–645
 Workbooks-Collection 314, 349
 WorksheetFunction-Objekt 292–294
 Worksheet-Objekt 18–19, 50–51, 94,
 98–99, 103, 134, 170–172, 175–176,
 178–179, 181, 184–185, 188, 190,
 192–194, 196–198, 200, 203, 210–
 211, 213, 223, 230–231, 236–237,
 239–241, 243–244, 246, 248, 251,
 253, 255–261, 270–271, 277, 280–
 281, 300, 315, 317, 324, 335, 393,
 438, 466, 471, 482, 503, 520, 525–
 527, 529, 538, 540, 543, 553, 566,
 633, 640, 642–645
 Worksheets-Collection 314
 Workspace-Objekt 590, 593

Z

Zellhintergrund 206, 212, 223
 Zellinhalt 40, 214, 217, 257, 585
 Zellkommentar 185, 202, 209, 290
 Zoom-Ereignis 413



Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt.

Dieses eBook stellen wir lediglich als **Einzelplatz-Lizenz** zur Verfügung!

Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschliesslich der Reproduktion, der Weitergabe, des Weitervertriebs, der Platzierung im Internet, in Intranets, in Extranets anderen Websites, der Veränderung, des Weiterverkaufs und der Veröffentlichung bedarf der schriftlichen Genehmigung des Verlags.

Bei Fragen zu diesem Thema wenden Sie sich bitte an:

<mailto:info@pearson.de>

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website



(<http://www.informit.de>)

herunterladen