

Visual Basic 2005

Programmierhandbuch



Auf CD: Kompletter Jahresinhalt
des dot.net magazin



Günter Born, Benjamin Born

Visual Basic 2005 Programmierhandbuch

Günter Born
Benjamin Born

Visual Basic 2005

Programmierhandbuch

entwickler.press

Günter Born, Benjamin Born : Visual Basic 2005
Programmierhandbuch
ISBN 3-935084-04-2

© 2006 entwickler.press
ein Imprint der Software & Support Verlag GmbH

<http://www.entwickler.press.de>
<http://www.software-support.biz>

Ihr Kontakt zum Verlag und Lektorat: lektorat@entwickler.press.de

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

Korrektur: mediaService, Siegen
Satz: mediaService, Siegen
Umschlaggestaltung: Melanie Hahn
Belichtung, Druck & Bindung: M.P. Media-Print Informationstechnologie GmbH,
Paderborn

Alle Rechte, auch für Übersetzungen, sind vorbehalten. Reproduktion jeglicher Art
(Fotokopie, Nachdruck, Mikrofilm, Erfassung auf elektronischen Datenträgern oder
andere Verfahren) nur mit schriftlicher Genehmigung des Verlags. Jegliche Haftung für
die Richtigkeit des gesamten Werks kann, trotz sorgfältiger Prüfung durch Autor und
Verlag, nicht übernommen werden. Die im Buch genannten Produkte, Warenzeichen
und Firmennamen sind in der Regel durch deren Inhaber geschützt.

Inhaltsverzeichnis

Vorwort	15
Teil 1	
Grundlagen und Entwicklungswerkzeuge	17
1 .NET-Grundlagen	19
1.1 .NET, was ist das?	19
1.2 Das .NET Framework	20
Die Common Language Runtime (CLR)	21
Die .NET-Framework-Klassenbibliothek	22
Was sind Assemblies und Manifeste?	23
1.3 Entwicklungsumgebungen für .NET	23
Visual Studio 2005	24
Visual Basic 2005 Express Edition	25
1.4 Entwicklungsumgebung und .NET installieren	26
Visual Studio 2005 installieren	26
Visual Basic 2005 Express Edition installieren	29
Visual Studio reparieren, deinstallieren oder warten	29
Die .NET Framework Redistributable installieren	30
.NET-Framework-Konfigurierung	31
2 Visual-Studio-2005-Kurzeinführung	33
2.1 Die Entwicklungsumgebungen im Überblick	33
Die Visual Basic 2005 Express Edition im Überblick	33
Visual Studio 2005 im Überblick	35
2.2 Arbeiten mit der Entwicklungsumgebung	37
Techniken zur Verwaltung der Teilfenster	37
Verwaltung der Projektdateien und -mappen	39
Ein Projekt neu anlegen	40
Nutzen des Konfigurations-Manager	44
Die Einstellungen des Projekts anpassen	44
Projektkomponenten hinzufügen	50
Eigene Projektvorlagen erzeugen	52
Die wichtigsten Fenster der Entwicklungsumgebung	54
Arbeiten im Codefenster	55
Der Objektbrowser	58
Verweise im Projekt hinzufügen	59
Das Fenster des Formulardesigners	60
Nutzen des Klassen-Designers	61
Projekt übersetzen und die Anwendung ausführen	63

2.3	Debuggen von .NET-Anwendungen	63
	Debuggen in der Entwicklungsumgebung	64
	Arbeiten mit Haltepunkten	66
	Anzeige der Werte im laufenden Programm	70
	Aufrufliste ansehen	73
	Debug-Aufruf über das Laufzeitsystem	74
	Debuggen mit dem Microsoft CLR-Debugger	74
2.4	Veröffentlichen und Verteilen	76
	Was sollte man über die Bereitstellung wissen?	77
	Bereitstellen eines Projekts mit ClickOnce	78
	Setup-Projekt mit Windows-Installer-Technologie	80
2.5	Visual Studio 2005 anpassen	85
2.6	Die Online-Hilfe nutzen	85

Teil 2

Einführung in Visual Basic 89

3	Visual-Basic-Grundlagen	91
3.1	Die Grundstruktur eines VB-Programms	91
	Anweisungen, Fortsetzungszeichen und Kommentare	91
3.2	Konstante in Visual Basic	97
	Konstanten	97
	Spezielle Hinweise zu Konstanten	101
	Zugriffsberechtigungen auf Konstante	101
3.3	Regeln für Bezeichner	103
3.4	Datentypen in Visual Basic	104
	Hintergrundinformationen zu den Datentypen	105
	Welche primitiven Datentypen gibt es im CTS?	105
	Welche Datentypen kennt Visual Basic 2005?	106
3.5	Arbeiten mit Variablen	112
	Wo dürfen Variablen deklariert werden?	113
	Variablen aus anderen Klassen nutzen	116
	Zugriffsberechtigung auf Variablen beeinflussen	117
4	Arrays und Strukturen	121
4.1	Arbeiten mit Feldern	121
	Deklaration einer Feldvariablen	121
	Felder bei der Deklaration initialisieren	125
	Mehrdimensionale Felder	130
	Anpassen der Feldgröße	132

4.2	Strukturen für benutzerdefinierte Datentypen	135
	Definition einer Datenstruktur mit Structure	135
	Arbeiten mit Nullable-Wertetypen	139
5	Zuweisungen, Operatoren und mehr	143
5.1	Zuweisungen in Visual Basic 2005	143
	Einfache Zuweisungen	143
	Verbundzuweisungen	143
	Typkonvertierung bei Zuweisungen	146
5.2	Operatoren in Visual Basic 2005	148
	Arithmetische Operatoren	149
	Logische Operatoren	149
	Relationale Operatoren	151
	Operatoren für Zeichenfolgen	152
	Priorität der Operatoren	154
5.3	Arbeiten mit Wert- und Verweistypen	155
5.4	Arbeiten mit Zeichenketten	158
	Zeichenkettenvergleich	161
	Wie kann ich in Zeichenketten suchen?	163
	Zeichenketten bearbeiten	168
	Zeichenketten zerlegen und verknüpfen	175
	Die StringBuilder-Klasse	177
5.5	Arbeiten mit Feldern	178
	Felder kopieren und neu initialisieren	178
	Suchen in Feldern	185
	Sortieren von Feldern	189
6	Steueranweisungen, Prozeduren und Fehlerbehandlung	191
6.1	Arbeiten mit der With-Anweisung	191
6.2	Bedingte Anweisungen (If-Anweisungen)	192
	If...Then-Anweisung	192
	If...Then...Else	193
	If...Then...ElseIf	194
6.3	Die Select-Case-Anweisung	194
	Sprünge mit der GoTo-Anweisung	195
6.4	Schleifen	196
	Do...Loop-Schleifen	196
	For...Next-Schleife	198
	Continue, eine Neuerung für Schleifen	199
6.5	Schleifen für Felder und Enumerationen	200
	Was sind Enumerationen?	200
	Zugriff auf Enumerationen mit Schleifen	204
6.6	Schleifen und Prozeduren vorzeitig beenden	209

6.7	Prozeduren und Funktionen	209
	Arbeiten mit Funktionen	209
	Prozeduren	213
	Anmerkungen zur Parameterübergabe (ByRef, ByVal)	216
	Verwendung benannter Argumente	219
	Arbeiten mit optionalen Argumenten	221
	Arbeiten mit ParamArray	222
	Private-, Public- und Static-Deklarationen	224
6.8	Fehlerbehandlung im Code	224
	So reagiert das System auf Exceptions	224
	Fehler mit On Error abfangen	226
	Zugriff auf das Fehlerobjekt	227
	Auslösen einer Exception per Programm	228
	Laufzeitfehler mit Try ... Catch ... Finally behandeln	229
7	Objektorientierte Programmierung	233
7.1	Grundbegriffe	233
	Was sind Klassen, Objekte, Eigenschaften und Methoden?	233
	Vererbung von Klassen	234
	Bemerkungen zu Modulen	235
	Schnittstellen	236
	Klassen über Namespaces identifizieren	236
	Im Code mit Objekten arbeiten	237
7.2	Eigene Klassen implementieren	240
	Eine Klasse mit Klassenvariablen definieren und nutzen	240
	So bekommt die Klasse Eigenschaften	243
	Methoden implementieren	251
	Anpassen des New-Konstruktors	252
	Die Member einer Klasse überladen	257
	Überladung von Operatoren	262
7.3	Vererbung in Klassen	265
	Vererbung einer Klasse von einer Basisklasse	265
	Vererbung mit Konstruktoren in der Basisklasse	268
	Vererbung mit angepassten/erweiterten Members	274
	Klasse vor Vererbung und Methode vor dem Überschreiben schützen	283
	Abstrakte Basisklassen, das steckt dahinter	285
7.4	Ereignisse und Delegates	287
	Arbeiten mit Delegate-Objekten	288
	Delegates und Ereignisse nutzen	290
	Ereignishandler über AddHandler anbinden	292
7.5	Schnittstellen	293
	Ein Szenario zur Verwendung von Schnittstellen	294
	Beispiel zur Verwendung einer Schnittstelle	296
	Nutzen der .NET-Schnittstellen	300

7.6	Collections und allgemeine Auflistungen	302
	Nutzen der ArrayList-Klasse	303
	Verwenden der HashTable-Klasse	305
	Nutzen der Collection-Klasse	307
	Allgemeine Auflistungen mit »Generic«	310
7.7	Neuerungen und weitere Arbeitstechniken	311
	Arbeiten mit dem My-Objekt	312
	Arbeiten mit Partial-Typen	313
	Klassen als Dateien oder Bibliotheken realisieren	313

Teil 3

Benutzeroberflächen 317

8	Einfache Interaktionen mit dem Benutzer	319
8.1	Ein-/Ausgaben auf Konsoleebene	319
	Formatierung der Konsoleausgaben	322
	Benutzereingaben auf Konsoleebene	328
8.2	Windows-Dialoge nutzen	334
	Einfache Ausgaben mit MsgBox()	334
	Dialogfelder mit mehreren Schaltflächen	337
	Benutzerführung über Dialogfelder ganz einfach	338
	So setzen Sie die MessageBox-Klasse ein	342
8.3	Benutzereingaben über InputBox	345
9	Arbeiten mit Formularen	349
9.1	Formularentwurf in der Entwicklungsumgebung	349
	So wird ein Formular angelegt	349
	Steuerelemente in Formulare einfügen	351
	Eigenschaften von Formularen und Steuerelementen ändern	352
	Ein einfaches Formularbeispiel	354
	Formularbeispiel: Anpassen von Eigenschaften	358
	Beispiel zur Demonstration von Formularvarianten	362
	Ein Formular mit Schaltflächen, Beschriftung und Textfeld	366
9.2	Spezielle Techniken für Formulare	368
	Formular programmgesteuert anzeigen	368
	Ein neues zusätzliches Formular im Projekt anlegen	372
	Formulare per Code erzeugen	375
	Formulare als Klassen realisieren	379
	ToolTips und Hilfe für Formularelemente festlegen	387
	Formulare mit dynamischen Inhalten	391
	Ein Formular mit Text- und Bildinhalten realisieren	392
	Willkommensdialog und Verifizieren von Formulareingaben	399
	MaskedTextBox-Elemente	408

10	Weitere Steuerelemente in Formularen	409
10.1	Formular zur Optionsauswahl	409
	Entwurf des Formulars im Designer	410
	Die Ereignisprozeduren für OK und Schließen realisieren	411
	Absenken der mittleren Schaltfläche steuern	412
	Die mittlere Schaltfläche mit einem Symbol versehen	413
	Umschalten der angezeigten Symbole	414
10.2	Arbeiten mit Auswahlfeldern	415
	Das Formular mit den Steuerelementen entwerfen	416
	Auswahlfelder zur Laufzeit mit Werten füllen	417
	Lesen der Benutzerauswahl	417
10.3	Arbeiten mit Einstellfeldern und Fortschrittsanzeige	419
	Einfügen der Steuerelemente in das Formular	419
	Beispiel um Ereignishandler ergänzen	420
10.4	Registerkarten und Textelemente verwenden	421
	Einfügen eines TabControl-Steuerelements	422
	Ein mehrzeiliges Textfeld mit Bildlaufleiste einrichten	424
	Verwenden eines RichTextBox-Elements	426
10.5	Anzeige von Bildern und Kalenderdaten	428
	Entwurf der Registerkarte zur Bildanzeige	428
	Entwurf der Registerkarte zur Kalenderanzeige	433
10.6	Listen- und Strukturansichten nutzen	435
	Vorbereiten des Formulars mit den Steuerelementen	435
	Gestalten des TreeView-Steuerelements	439
	Entwurf des ListView-Steuerelements	444
	Einsatz des Webbrowser-Steuerelements	449
	Verwenden des PropertyGrid-Steuerelements	451
	Container-Steuerelemente zur Layout-Kontrolle	452
11	Menüs und weitere Leisten	455
11.1	Menüs einbinden	455
	Ein Menü in .NET-Formularen erstellen	455
	Menübefehle mit Funktionen belegen	459
	Menüs mit Häkchen und Punkten	464
11.2	Kontextmenüs nutzen	468
	Ein Kontextmenü zum Formular hinzufügen	468
11.3	Symbolleisten verwenden	473
	Symbolleisten zum Formular hinzufügen	474
	Code zur Anbindung der Elemente an Funktionen	481
	Verwenden des alten ToolBar-Steuerelements	487
	Eine Symbolleiste per Programm erzeugen	491
11.4	Eine Statusleiste verwenden	494
	Entwurf einer Anwendung mit einer Statusleiste	494
	Eine StatusBar-Beispielanwendung	498
	Nutzen des alten StatusBar-Controls	499

12	Standarddialoge und MDI-Anwendungen	505
12.1	Standarddialoge nutzen	505
	Eine Beispielanwendung für Standarddialoge	505
	Erstellen des Anwendungsformulars	506
	Öffnen-Dialog zur Anwendung hinzufügen	512
	Realisierung der Speichern-Funktion	515
	Anzeige eines Font-Dialogs	517
	Anzeige des Dialogs zur Farbauswahl	518
	Rückgängig machen einer TextBox-Änderung	519
	Schließen bei ungesicherten Änderungen verhindern	520
12.2	Verwenden des FolderBrowser-Dialogs	521
12.3	MDI-Anwendungen erstellen	523
	Vorbereitungen für das übergeordnete Formular	525
	Das untergeordnete MDI-Formular erstellen	527
	Hinzufügen des Programmcodes für neue Dokumentfenster	528
	Weitere Verwaltungsfunktionen für die Fenster	530
	Weitere Programmier Techniken für die Anwendung	533
Teil 4		
Programmiertechniken		541
13	.NET-Basisfunktionen für Windows	543
13.1	Zugriff auf Programmargumente	543
	Parameterübernahme als Argumente der Main-Prozedur	543
	Parameter über Zusatzmethoden abfragen	545
13.2	Rückgabe eines Statuscodes	547
13.3	Reagieren auf Anwendungsereignisse	548
	Freigabe der Anwendungsereignisse	549
	Einfügen der MyApplication-Ereignisbehandlungsroutinen	549
13.4	Aufrufe externer Anwendungen	550
	Arbeiten mit der Shell-Funktion	550
	Kontrolle externer Prozesse über die Klasse Process	554
13.5	Einbinden externer COM-Komponenten	561
	Spätes Binden (Late Binding) an eine COM-Anwendung	561
	Frühes Binden (Early Binding) an eine COM-Anwendung	561
	Ordnerfenster und Dialogfeld Ausführen öffnen	563
	Shell-Aufrufe unter Verwendung eines COM-Wrappers	563
	Ein weiteres Beispiel für Shell-Aufrufe	564
13.6	Das NotifyIcon-Steuerelement verwenden	566
	Entwurf des Formulars für das NotifyIcon-Beispiel	567
	Code für die Ereignisbehandlungsroutinen	568

13.7	Zugriff auf Systeminformationen	570
	Abrufen der Umgebungseinstellungen	571
	Umgebungsvariable anzeigen	575
	Alle logischen Laufwerke auflisten	576
	Pfade der Systemordner ermitteln	577
13.8	Zugriff auf die Registrierung	578
	Registrierungseintrag schreiben, lesen und löschen	578
	Unterschlüssel und Werte auflisten	583
	Remote-Registrierungszugriffe	586
	Zugriff auf INI-Dateien	588
13.9	Zugriff auf Windows-API-Funktionen	588
	Deklaration einer API-Funktion	589
	Anwenden eines API-Aufrufs	589
14	Dateioperationen in .NET	593
14.1	Auflisten aller Laufwerke samt Eigenschaften	593
	Wie lässt sich der Laufwerkstyp ermitteln?	594
	Weitere Laufwerkseigenschaften abfragen	594
14.2	Zugriff auf Dateien und Ordner	596
	Unterordner und Dateien auflisten	597
	Zugriff auf Ordner- und Dateieigenschaften	599
	Ist der Datenträger/Ordner leer?	600
14.3	Datum und Dateiattribute lesen/setzen	603
	Ein Anwendungsbeispiel zur Attributanzeige	603
	SetFileDate-Anwendung	609
14.4	Verzeichnisse und Laufwerke bearbeiten	611
	Aktuelles Verzeichnis abfragen	611
	Aktuelles Verzeichnis wechseln	612
	Aktuelles Laufwerk wechseln/ermitteln	612
14.5	Kopieren, löschen, verschieben, umbenennen	615
	Verwenden der Visual-Basic-Funktionen	616
	Dateibearbeitung per File- und Directory-Klasse	619
14.6	Zugriffe auf Dateiinhalte	623
	Allgemeine Hinweise zu Dateizugriffen	624
	Eine neue Datei anlegen und mit Text füllen	624
	Den Inhalt einer Textdatei lesen	627
	Textdatei lesen und verändern	630
	Textdatei lesen und beschreiben	632
	Lesen und Schreiben binärer Daten	635
	Anzeige von Dateien als Hexadezimaldump	639
14.7	Nutzen der FileSystemWatcher-Klasse	643
	Beispiel: Überwachen eines Ordners auf Textdateien	644
	Hinweise zur Implementieren des Beispiels	644

14.8	Zugriff auf die EventLog-Dateien	648
	Bemerkungen zur Ereignisprotokollierung	648
	Das EventLog-Element verwenden	649
	Schreiben in das Ereignisprotokoll	651
	Lesen des Ereignisprotokolls	651
	Hinweise zur Implementieren eines Beispiels	652
15	Datenbankfunktionen nutzen	653
15.1	ADO.NET-Grundlagen	653
	Wann werden DataReader und DataSet verwandt?	654
	Wie werden die Klassen in Anwendungen genutzt?	654
15.2	Datenbanken anlegen und verwalten	655
	Verwalten der Datenverbindungen	655
	Eine neue SQL-Server-Datenbankdatei anlegen	656
	Hinzufügen von Tabellen zur Datenbank	659
	Tabellendaten anzeigen, eingeben und bearbeiten	661
	Verbindung mit Datenquellen herstellen	662
	SQL-/Access-Datenbankdatei in das Projekt einbinden	665
15.3	Zugriff auf Datenquellen aus Formularen	666
	Darstellung einer Datenbanktabelle per Formular	666
	Tabelle manuell an ein DataGridView binden	671
	So binden Sie beliebige Steuerelemente beim Entwurf	675
15.4	Datenbankzugriffe mit ADO.NET per Programm	679
	Auslesen einer Tabelle aus einer Access-Datenbank	679
	Daten programmgesteuert in einem DataGridView anzeigen	684
	DataSet an Steuerelemente binden und navigieren	688
	Erstellen von Berichten	692
16	Spezielle Themen und Techniken	699
16.1	Zugriff auf XML-Daten in .NET	699
	XML im Schnellüberblick	699
	Eine Datenbanktabelle in XML exportieren	705
	XML-Daten per DataSet einlesen	706
	Zugriff auf XML-Dateien per DOM	708
	Zugriff auf die XML-Daten per XMLTextReader	711
16.2	Anwendungseinstellungen und Hilfe	716
	Verwalten von Formular- und Anwendungseinstellungen	716
	Hilfdateien einbinden	722
16.3	Drucken in .NET-Anwendungen	724
	Hinweise zum Drucken	724
	Druckfunktionen implementieren, so geht's	725
	Realisierung der Seitenansicht	732
	Mehrseitiges Drucken geht auch	733

Inhaltsverzeichnis

16.4	Grafikprogrammierung in .NET	736
	Einstieg in das Zeichnen mit der Graphics-Klasse	737
	Zeichnen in der Paint-Ereignisbehandlungsroutine	738
	Zeichenbereich per Programm holen und zeichnen	741
	Beispiel: Anwendung weiterer Zeichenfunktionen	747
	Linienstile und Muster beim Zeichnen verwenden	750
	Zeichenfunktionen für Bitmaps und Metagrafiken	752
	Anfertigen von Screenshots einer Anwendung	755
 Teil 5		
Anhang		759
 A Die Begleit-CD		761
Stichwortverzeichnis		763

Vorwort

.NET ist eine Architektur zur Entwicklung von Anwendungen für Microsoft Windows und mobile Geräte, die seit mehreren Jahren von Microsoft als Alternative zu Java propagiert wird. Visual Basic ist dabei eine der von Microsoft unterstützten Sprachen zur Erstellung dieser Anwendungen. Mit der nun vorliegenden Version .NET 2.0 hat Microsoft nicht nur das .NET Framework, also quasi die Infrastruktur für .NET-Anwendungen, sondern auch die verfügbaren Entwicklungsumgebungen sowie die Sprache Visual Basic 2005 überarbeitet und funktional erweitert.

Mit Visual Studio 2005 verfügen Sie über alle benötigten Funktionen zur professionellen Entwicklung von .NET-Anwendungen unter Verwendung von Visual Basic 2005. Auch wer sich die kostenlose Visual Basic 2005 Express Edition und die Microsoft SQL Server Express Edition aus dem Internet herunterlädt (oder ggf. von einer Zeitschriften-CD installiert), kann sofort loslegen. Ideal also, um in die .NET-Programmierung einzusteigen, in Visual Basic 2005 hineinzuschnuppern oder auf kostengünstigem Weg .NET-Anwendungen zu erstellen.

Allerdings gibt es ggf. ein kleines Problem: Obwohl die in den jeweiligen Paketen enthaltene Hilfe umfangreiche Informationen zur jeweiligen Entwicklungsumgebung, zum .NET Framework 2.0 etc. liefert, geht leicht der »rote Faden« verloren. Ein- oder Umsteiger sowie gelegentliche Benutzer verlieren schnell die Übersicht, da oft unklar ist, nach was in der Hilfe zu suchen ist. Die Vielzahl der Neuerungen im .NET 2.0 Framework oder Änderungen in den Entwicklungsumgebungen vereinfachen auch nicht gerade den Umstieg von älteren Visual Studio-, .NET-Framework- oder Visual-Basic-Versionen.

Das vorliegende Buch möchte Sie als Leser beim Einstieg in die (sowie beim Umstieg zur) Visual-Basic-Programmierung unter .NET unterstützen. Ausgehend von einem kurzen Überblick über die Grundzüge von .NET führen Sie die einzelnen Kapitel schrittweise in Visual Basic 2005 sowie in die Erstellung von .NET-Anwendungen ein. Ziel ist dabei nicht die komplette Vorstellung aller Optionen und Möglichkeiten. Vielmehr sollen Sie das erforderliche Wissen erwerben, um Visual Basic 2005 zum Erstellen eigener .NET-Anwendungen für Windows zu verwenden. Für Ein- und Umsteiger ist eine Einführung in die Grundlagen von Visual Basic 2005 im Buch enthalten. Weiterhin wird in den jeweiligen Kapiteln erwähnt, wenn sich Neuerungen (z.B. beim Erstellen von Menüs oder Symbolleisten) gegenüber früheren .NET-Versionen ergeben. Zum Erstellen der Beispiele wurde als Entwicklungsumgebung Visual Studio 2005 unter Windows XP verwendet. Die besprochenen Techniken funktionieren aber weitgehend auch in der kostenlosen Visual Basic 2005 Express Edition. Sie können also auch mit dieser Entwicklungsumgebung die auf der Begleit-CD enthaltenen Beispiele laden, übersetzen und testen.

Angesichts des schieren Funktionsumfangs und dem vom Verlag vorgegebenen Seitenumfang dieses Buches ist es nicht möglich, alle Funktionen der Entwicklungswerkzeuge oder des .NET Framework abzudecken. Die nachfolgenden Kapitel fungieren daher als Leitfaden, um bestimmte Techniken (Gestaltung von Benutzeroberflächen, Nutzen spezieller Funktionen des .NET Framework etc.) zu vermitteln. Details zu bestimmten Funktionen oder ergänzende Informationen zu ausgesparten Themen lassen sich dann bei Bedarf gezielt in der Hilfe zum .NET Framework, zu Visual Studio 2005 oder zum Microsoft SQL Server (bzw. zur Express Edition dieser Produkte) nachlesen.

Vorwort

Zum Abschluss bleibt uns nur der Dank an alle Personen, die uns bei der Arbeit an diesem Buch unterstützten. Erwähnt seien Erik Franz, der uns das Projekt anbot, und Dirk Frischalowski, der das fachliche Lektorat übernahm und in konstruktiv kritischer Art viele nützliche Anregungen lieferte. Ihnen als Leser(in) wünschen wir viel Spaß und Erfolg beim Lesen dieses Buches sowie beim Lernen, Ausprobieren und Arbeiten mit Visual Basic 2005 und .NET.

Benjamin und Günter Born

www.borncity.de

Teil 1

Grundlagen und Entwicklungswerkzeuge

1

.NET-Grundlagen

Wer mit Visual Basic 2005 entwickelt, sollte zumindest einige Grundbegriffe und Grundlagen aus dem .NET-Umfeld kennen. Dieses Kapitel stellt das Konzept von .NET kurz vor und zeigt, welche Entwicklungsumgebungen verfügbar sind. Einsteiger sollten zumindest die ersten Abschnitte dieses Kapitels durchlesen, um eine ungefähre Idee von der .NET-Architektur zu erhalten.

1.1 .NET, was ist das?

Die in Visual Studio 2005 enthaltenen Programmiersprachen setzen auf Microsoft .NET (als »Dot Net« ausgesprochen) auf. Aber was hat es mit .NET und den im Umfeld benutzten Begriffen auf sich? Bei Microsoft .NET handelt es sich um eine Plattform, die die notwendigen Werkzeuge und Technologien bereitstellt, um Anwendungen für Microsoft Windows und andere Plattformen wie mobile Geräte (Handys, PDSs etc.) oder Web-Server zu entwickeln. Diese Anwendungen erlauben die Kommunikation mit vielen (entsprechend ausgerüsteten) Geräten wie Windows-Rechnern, Mobiltelefonen, PDA etc. Letztendlich bildet Microsoft .NET eine Architektur, die sich aus verschiedenen Bausteinen zusammensetzt.

- **Visual Studio 2005** ist dabei die neueste Microsoft-Entwicklerplattform, mit der sich Anwendungen in verschiedenen Sprachen (Visual Basic, C#, C++ etc.) erstellen lassen. Visual Studio 2005 stellt dabei eine recht mächtige Entwicklungsumgebung dar, die von der Erstellung des Quellcodes über das Testen des Codes über die Anbindung an Datenbanken bis hin zur Erstellung von Anwendungspaketen alle Schritte unterstützt. Auf die Details komme ich weiter unten noch zurück.
- **.NET Framework** stellt die Laufzeitumgebung für .NET-Anwendungen bereit und enthält eine sprachenunabhängige Klassenbibliothek. Diese Laufzeitumgebung sowie die Bibliothek wird Sie beim Programmieren in Visual Basic 2005 auf Schritt und Tritt begleiten. Allerdings wurde die Konzeption der .NET-Architektur nicht ausschließlich auf Windows abgestimmt. Sofern die betreffenden Clients verfügbar sind, können .NET-Anwendungen auf weiteren Geräten wie der Xbox, auf Mobiltelefonen, auf PDAs etc. laufen. Microsoft entwickelt für die unterschiedlichen Geräte entsprechende Clients (z.B. .NET Framework für Windows, .NET Compact Framework als Untermenge des .NET Framework für Embedded Windows XP etc.).
- Zusätzlich umfasst die .NET-Architektur noch Funktionen zur Entwicklung von Web Services, die auf entsprechenden Servern als Softwarekomponenten bereitgestellt werden. Bei den Webservices handelt es sich um Software-Module, deren Funktionen sich über Internet, Intranet etc. mittels Standards wie XML, SOAP etc. ansprechen bzw. nutzen lassen.

Die obige Aufstellung zeigt Ihnen, dass .NET eine komplexe Sammlung an Technologien, Anwendungen und Entwicklungsumgebungen ist, die unter einer gemeinsamen Architektur vereint wurden. In diesem Buch wird sich aber darauf beschränkt, Microsoft .NET zu nutzen, um Anwendungen für Windows unter Visual Basic zu entwickeln (also das, was viele Visual-Basic-Programmierer seit Jahren tun).

Hinweis

Microsoft strebt mit .NET an, für die eigenen Plattformen eine Alternative zu Java zu schaffen. Ziel bei der Konzeption von .NET war zudem, die sich abzeichnenden Nachteile der bisherigen ActiveX- und COM-Architektur zu vermeiden. So benötigen auf COM basierende Windows-Anwendungen häufig DLLs und müssen sich bei der Installation im Betriebssystem registrieren. Dies führt u.U. zu Problemen, wenn mehrere Anwendungen unterschiedliche DLL-Versionen benötigen. Zudem müssen auf COM basierende Anwendungen installiert und später ggf. deinstalliert werden. Dabei bleiben häufig DLL-Dateien als Leichen zurück. Der .NET-Ansatz basiert auf der Annahme, dass die Infrastruktur für Programme durch eine Laufzeitumgebung bereitgestellt wird und die Anwendungen ihren Programmcode in eigenen Dateien mitbringen. Die Installation reduziert sich dann auf das Kopieren der Dateien in den Zielordner, zur Deinstallation reicht das Löschen der betreffenden Dateien – es sind keine Eingriffe in die Registrierung mehr erforderlich.

1.2 Das .NET Framework

Das .NET Framework (Framework lässt sich als Rahmenwerk übersetzen) stellt die Infrastruktur für .NET-Anwendungen bereit (Abbildung 1.1).

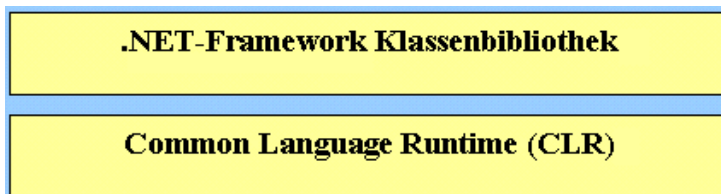


Abbildung 1.1: Komponenten des .NET Framework

- Einmal enthält das .NET Framework die Laufzeitumgebung (CLR), in der .NET-Anwendungen ausgeführt werden.
- Weiterhin enthält das .NET Framework die Klassenbibliothek, die in den .NET-Anwendungen benutzt werden können. Diese Klassenbibliothek bietet Funktionen zum Zugriff auf das Betriebssystem (z.B. Windows), auf das Dateisystem und vieles mehr.

Zudem enthält .NET Framework noch Compiler (C#, Visual Basic etc.) und weitere Tools, um die Programme zu übersetzen, auszuführen und zu testen. .NET Framework ist in Windows 2003 Server (sowie in zukünftigen Microsoft-Betriebssystemen) integra-

ler Bestandteil des Betriebssystems. Unter älteren Windows-Versionen muss das .NET Framework 2.0 aber installiert werden. Andernfalls lassen sich weder .NET-Anwendungen ausführen noch entwickeln. Das .NET Framework wird dabei durch Microsoft in zwei Varianten bereitgestellt.

- **.NET Framework Redistributable:** Dies ist quasi die Minimalversion, die neben der Common Language Runtime (CLR) auch die .NET Framework Klassenbibliothek, ADO.NET sowie die Compiler für die jeweiligen Sprachen enthält.
- **.NET Framework SDK:** Dieses Software Development Kit (SDK) enthält neben den Dateien der Redistributable weitere Tools, Beispiele und Dokumentation (Online-Hilfe), die Entwickler zum Schreiben von .NET-Anwendungen benötigen.

Visual Studio 2005, Visual Basic 2005 Express Edition sowie die mit diesen Umgebungen erzeugten .NET-Anwendungen benötigen zur Ausführung die Version 2.0 des .NET Framework. Diese Version weist gegenüber den älteren 1.x-Versionen umfangreiche Änderungen und Neuerungen in den Klassenbibliotheken auf.

Hinweis

Wenn Sie Visual Studio 2005 bzw. Visual Basic 2005 Express Edition auf einem Rechner installieren, wird automatisch das benötigte .NET Framework auf dem Entwicklungsrechner eingerichtet. Eine detailliertere Übersicht des .NET Framework finden Sie in der Hilfe zu Visual Studio 2005 in der Rubrik »**.NET-Framework-Programmierung**« im Unterthema »Übersicht über .NET Framework« im Abschnitt »Konzeptionelle Übersicht«.

1.2.1 Die Common Language Runtime (CLR)

Die Common Language Runtime (CLR) ist die Laufzeitumgebung, in der .NET-Anwendungen ausgeführt werden. Die zum Erstellen von .NET-Anwendungen benutzten Compiler (C#, Visual Basic, C++ etc.) erzeugen nur einen sogenannten CIL-Zwischencode (Common Intermediate Language Code). Der Zwischencode enthält dabei Anweisungen zum Laden und Initialisieren von Objekten, zum Aufrufen der Methoden von Objekten sowie Anweisungen für arithmetische und logische Operationen, zur Ablaufsteuerung, zur Ausnahmebehandlung und zur Durchführung anderer Operationen.

Der Zwischencode wird zur Laufzeit von einem Just-in-Time-Compiler in den ausführbaren Maschinencode des Zielsystems überführt. Dies erlaubt der CLR, die Ausführung des Programmcodes zu kontrollieren und Aufgaben wie die Speicherverwaltung zu übernehmen. Der in .NET-Anwendungen (.exe- oder .dll-Dateien) oder in Klassenbibliotheken hinterlegte Programmcodes wird daher auch als verwalteter Code (managed Code) bezeichnet. Rufen .NET-Anwendungen allerdings Betriebssystemfunktionen (COM-Objekte oder API-Funktionen) auf, werden diese Programmteile nicht mehr unter der Kontrolle der CLR ausgeführt. Man bezeichnet dies als Ausführen von unmanaged Code. Fehler, die während der Ausführung von unmanaged Code auftreten, können nicht durch die CLR kontrolliert werden. Weitere Details zur CLR finden Sie in der Hilfe zu Visual Studio 2005 in der Rubrik »**.NET-Framework-Programmierung**« im Unterthema »Übersicht über .NET Framework« im Abschnitt »Common Language Runtime«.

1.2.2 Die .NET-Framework-Klassenbibliothek

Das .NET Framework stellt neben der Laufzeitumgebung eine Klassenbibliothek bereit, die von den .NET-Anwendungen genutzt wird. Der Pfiff dabei ist, dass der Zugriff auf die Klassenbibliothek aus jeder der in .NET benutzten Sprachen möglich ist. Ein C#-Programm benutzt also die gleiche Klassenbibliothek wie Visual Basic. Dabei sind die einzelnen Klassen über sogenannte Namensräume streng hierarchisch strukturiert. Der Namensraum (z.B. *System.IO*) gibt dem nutzenden Programm also genau vor, wo es in der betreffenden Bibliothek die benötigte Komponente findet. Existieren innerhalb der Klassenbibliothek zwei Komponenten gleichen Namens, lässt sich durch unterschiedliche Namensräume eindeutig die richtige Komponente identifizieren.

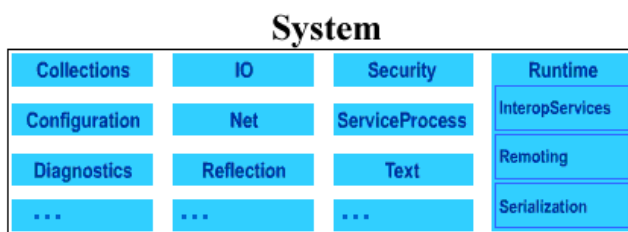


Abbildung 1.2: Hierarchie der Klasse System

Sofern Sie erst in die Thematik einsteigen, können Sie sich die Klassenbibliothek wie eine fertige Werkzeugsammlung vorstellen, aus denen sich die Programmiersprache bzw. der Entwickler bedient. Wenn Sie beispielsweise eine Dateioperation in einem Programm vornehmen, verwendet das Programm die entsprechenden Funktion aus der Klassenbibliothek. Aber es geht noch weiter: Selbst die Datentypen der in .NET-Programmen benutzten Variablen werden über die Klassenbibliothek bereitgestellt.

Das Common Type System (CTS)

Eine Besonderheit beim .NET Framework ist das Common Type System (allgemeines Typensystem). Es handelt sich dabei um ein Modell, welches die Regeln definiert, nach denen die Common Language Runtime Typen verwaltet. Hierzu sind im .NET Framework beispielsweise die Basistypen (Integer, Decimal etc.) für die in Programmiersprachen benutzten Variablen und Konstanten definiert. Zusätzlich sind weitere komplexere Typen (z.B. Objekte) definiert. Deklariert ein Programm Variable oder Konstante, werden diese aus den in der Klassenbibliothek hinterlegten Basistypen abgeleitet.

Was ist die Common Language Specification (CLS)?

.NET ist so angelegt, dass Anwendungen in verschiedenen Programmiersprachen erstellt werden können. Zudem lassen sich in unterschiedlichen Sprachen erstellte Programmteile innerhalb der Anwendung kombinieren. Daher muss sichergestellt sein, dass Typen (z.B. Variablen), die in einer Programmiersprache deklariert und dann im Rahmen eines Funktionsaufrufs an eine in einer anderen Programmiersprache erstellte Komponente übergeben werden, in beiden Sprachen gleich implementiert und behandelt werden. Die Common Language Specification (CLS) ist dabei das Regelwerk, welches den Austausch von Typen aus unterschiedlichen Programmiersprachen steuert.

1.2.3 Was sind Assemblies und Manifeste?

Die übersetzten Programmklassen bzw. ausführbaren Programme werden in .NET in sogenannten Assemblies (kommt von Zusammenstellen) zusammengefasst. Ein Assembly enthält den CIL-Code, der von der Common Language Runtime ausgeführt wird. Statische Assemblies werden in *.exe*- oder *.dll*-Dateien gespeichert und können zudem .NET Framework-Typen (Schnittstellen und Klassen) sowie Ressourcen (Bitmaps, JPEG-Dateien, Ressourcendateien usw.) enthalten. Zudem gibt es noch dynamische Assemblies, die im Arbeitsspeicher erstellt werden und nur CIL-Code enthalten können.

Um den Code einer Assembly auszuführen, benötigt die Common Language Runtime verschiedene Informationen, die beschreiben, wie die Elemente der Assembly miteinander verknüpft sind oder welche Dateien zum Assembly gehören. Dies umfasst auch Angaben zu Versionsanforderungen der Assembly, zum Gültigkeitsbereich oder zur Auflösung von Verweisen auf Ressourcen und Klassen. Alle diese Informationen werden in einem sogenannten Assemblymanifest in Form von Metadaten hinterlegt. Dieses Manifest umfasst dabei mindestens einen Namen, eine Versionsnummer, Informationen über die Sprache (Kultur), in der die Anwendung auszuführen ist, die Liste der benötigten Dateien, Informationen über Assemblies, auf die ggf. verwiesen wird und mehr. Das Assemblymanifest kann dabei in der PE-Datei (d.h. in der *.exe*- oder *.dll*-Datei) hinterlegt sein. Zusätzlich gibt es eigenständige PE-Dateien, die ausschließlich Informationen aus dem Assemblymanifest enthalten.

Hinweis

Der Vorteil der Assemblies in Kombination mit Manifesten liegt darin, dass keine Registrierung der DLL-Bibliotheken mehr erforderlich ist. Sofern das Assemblymanifest vorhanden ist, sucht das CLI die benötigten Assemblies. Werden diese gefunden, wird deren CIL-Code ausgeführt. Existieren mehrere Versionen einer Assembly, werden diese einfach in getrennten Ordnern gespeichert und ausgeführt. Die Versionsangaben im Assemblymanifest stellen sicher, dass nur die gewünschten Komponenten verwendet werden. Eine umfassende Übersicht über die Konzepte von .NET 2.0 finden Sie in der Visual Studio 2005-Hilfe unter der Überschrift »Übersicht über .NET Framework« sowie im Internet auf der Webseite de.wikipedia.org/wiki/Microsoft_.NET_Framework. Zum Einstieg reicht es aber zu wissen, dass für die Ausführung von .NET-Anwendungen eine Laufzeitumgebung in Form des .NET Framework benötigt wird. Die Verwendung der Klassenbibliotheken lernen Sie in den folgenden Kapiteln kennen.

1.3 Entwicklungsumgebungen für .NET

Mit der .NET Framework Redistributable stehen Ihnen bereits alle Werkzeuge zum Kompilieren von .NET-Anwendungen zur Verfügung. Zur effizienten Unterstützung der Programmentwicklung stellt Microsoft aber verschiedene Entwicklungsumgebungen bereit.

1.3.1 Visual Studio 2005

Visual Studio 2005 ist die kostenpflichtige Entwicklungsumgebung zur professionellen Erstellung von .NET-Anwendungen und steht in verschiedenen Varianten zur Verfügung.

- **Visual Studio Standard Edition:** Dies ist die Variante der Entwicklungsumgebung, die sich vor allem an Einzelentwickler richtet und das Erstellen von Client-/Server-Anwendungen für Windows, das Web und mobile Geräte ermöglicht.
- **Visual Studio 2005 Professional Edition:** Diese Version richtet sich an professionelle Entwickler, die ggf. in kleinen Teams arbeiten. In dieser Variante wurde die Entwicklungsumgebung um Funktionen und Komponenten erweitert, um neben Client-/Server-Anwendungen für Windows, das Web und mobile Geräte auch mehrschichtige Anwendungen zu entwickeln.
- **Visual Studio 2005 Team System:** Es handelt sich um eine aus verschiedenen Paketen bestehende Entwickler-Suite, die neben der eigentlichen Visual Studio Entwicklungsumgebung weitere Tools umfasst. Diese bieten Unterstützung während des kompletten Software-Lebenszyklus eines Produkts (z.B. Code-Pflege, Test/Qualitätssicherung, Kommunikation und Zusammenarbeit in Entwicklerteams etc.). Diese Variante richtet sich an Unternehmen, in denen Entwicklergruppen mit Projekten betraut sind und eine bessere Planbarkeit der Anwendungsentwicklung und eine Verbesserung der Qualität des Entwicklungsprozesses zu gewährleisten ist.

Alle diese Pakete beinhalten die Unterstützung für verschiedene Sprachen wie Microsoft Visual Basic, Microsoft Visual C#, Microsoft Visual C++, Microsoft Visual J# und stellen auch Werkzeuge zum Erstellen, Testen und zur Weitergabe von .NET-Anwendungen bereit. Ein Vergleich des Funktionsumfangs der einzelnen Produktvarianten findet sich z.B. unter www.microsoft.com/germany/msdn/vstudio/produktvergleich.mspx im Internet. Die Systemanforderungen für die Standard Edition von Visual Studio 2005 werden von Microsoft folgendermaßen spezifiziert:

- Ein Computer mit mindestens 600 MHz Taktfrequenz oder mehr, mindestens 192 Mbyte Arbeitsspeicher, 2 Gbyte verfügbarer Festplattenspeicher, CD- oder DVD-Laufwerk und 1024 x 768 Bildpunkten und 256 Farben Grafikauflösung.
- Visual Studio 2005 setzt beim Betriebssystem Windows 2000 (ab Service Pack 4), Windows XP (mit Service Pack 2) oder spätere Versionen voraus.

Je nach benutzten Zusatzkomponenten (SQL-Server) werden höhere Systemanforderungen gestellt. Um halbwegs vernünftig arbeiten zu können, empfiehlt sich aber ein Rechner mit mindestens einem 2 GHz-Prozessor und 512 Mbyte Arbeitsspeicher. Aktuelle Windows-Rechner dürften die Anforderungen von Visual Studio 2005 auf jeden Fall erfüllen. Detaillierte Informationen zu den jeweiligen Paketen und Systemanforderungen gibt's unter www.microsoft.com/germany/msdn/vstudio.

1.3.2 Visual Basic 2005 Express Edition

Neben Visual Studio 2005 stellt Microsoft noch die kostenlosen »Visual Studio Express Editionen« für Einsteiger, Schüler, Studenten oder Hobby-Programmierer bereit. Neben Visual Basic 2005 Express Edition und C# Express Edition gibt es noch die Express Editionen des SQL Server 2005 oder des Visual Web Developer 2005. Bei der Visual Basic 2005 Express Edition handelt sich um die Entwicklungsumgebung von Visual Studio, die aber funktional auf das Erstellen von Anwendungen in Visual Basic 2005 begrenzt ist. Trotz dieser Einschränkung lassen sich mit dieser Entwicklungsumgebung auch komplexere .NET-Anwendungen entwickeln – zumindest reicht es, um die Beispiele des Buches zu übersetzen. Es gibt mehrere Möglichkeiten, an die kostenlose Visual Basic 2005 Express Edition zu gelangen. Unter der Adresse:

www.microsoft.com/germany/msdn/vstudio/express/vb/default.msp

können Sie den gut 2,8 Mbyte umfassenden Webinstaller herunterladen. Dieser Installer lädt dann die Dateien des ausgewählten Pakets aus dem Internet nach. Microsoft bietet aber auch das komplette Paket als ISO-Imagedatei (ca. 400 Mbyte) unter folgender Adresse zum Download an:

<http://download.microsoft.com/download/4/c/7/4c758b20-a373-4b00-aa1a-40c90760d0af/vb.iso>

Sofern Sie diese ISO-Datei auf eine CD brennen, lässt sich das Produkt mehrfach von diesem Medium installieren, was im Hinblick auf die Download-Datenmengen häufig günstiger ist. Wer über keine schnelle Internetverbindung verfügt, findet die ISO-Version von Visual Basic 2005 Express Edition auf diversen Zeitschriften-CDs. Zur Installation der Express Edition gelten ähnliche Anforderungen an Hard- und Software wie für Visual Studio 2005.

Hinweis

Beim Schreiben dieses Buches wurde die Visual Studio 2005 Standard Edition (bzw. teilweise die Professional Version) benutzt. Sie können aber den größten Teil der Beispiele und Übungen auch mit der kostenlosen Visual Basic 2005 Express Edition bearbeiten und übersetzen. Zur Verwaltung der Datenbanken müssen Sie zudem den Microsoft SQL Server 2005 Express installieren. Falls Sie bereits eine Visual C# 2005 Express Edition installiert haben, reduziert sich der Download-Umfang für Visual Basic 2005 Express Edition übrigens auf ca. 65 MByte, da Komponenten wie die MSDN-Hilfe, die .NET Framework Redistributable etc. bereits installiert sind.

Sie sind zum Übersetzen von .NET-Anwendungen also nicht auf Visual Studio angewiesen. Zur Not ließe sich sogar das .NET Framework SDK verwenden, da dieses die Compiler für C# und Visual Basic enthält. Der Programmcode lässt sich mit jedem Editor pflegen und die Compiler sind direkt aus dem Fenster der Eingabeaufforderung aufrufbar. Da aber komfortable und zudem kostenlose Entwicklungsumgebungen in Form der Express Editionen bereitstehen, gehe ich in diesem Buch nicht auf die Möglichkeiten zum Übersetzen per Eingabeaufforderung ein.

1.4 Entwicklungsumgebung und .NET installieren

Nachfolgend wird die Installation der verschiedenen Entwicklungsumgebungen sowie des .NET Framework 2.0 besprochen.

1.4.1 Visual Studio 2005 installieren

Visual Studio 2005 wird in verschiedenen Varianten (siehe vorhergehende Seiten) angeboten. Die einzelnen Pakte enthalten einen Satz an Installations-CDs (Visual-Studio-CDs, ggf. die SQL-Server-2005-CDs und die MSDN-CDs mit der Hilfe). Zur Installation gehen Sie folgendermaßen vor:

1. Stellen Sie als Erstes sicher, dass Sie über einen Entwicklungsrechner verfügen, der die Mindestanforderungen an Hard- und Software erfüllt. Weiterhin sollte das Betriebssystem bezüglich Updates und Service Packs auf dem neuesten Stand sein.
2. Melden Sie sich unter einem Administratorenkonto an, damit Sie über die Berechtigungen zur Softwareinstallation verfügen.
3. Legen Sie die Installations-CD (oder DVD) in das Laufwerk ein und warten Sie, bis der Dialog des Setup-Assistenten erscheint (Abbildung 1.3). Falls das Installationsprogramm nicht automatisch vom eingelegten Medium startet, rufen Sie das Setup-Programm *Setup.exe* auf.



Abbildung 1.3: Visual-Studio-2005-Setup-Dialog

4. Klicken Sie auf die im Dialogfeld des Assistenten gezeigte Schaltfläche *Infodatei anzeigen*. Windows öffnet die betreffende Informationsseite im Internet Explorer und Sie können sich über die Hard- und Software-Anforderungen des Pakets, über Installationsprobleme und mehr informieren.
5. Klicken Sie im Dialogfeld auf den Hyperlink *Visual Studio 2005 installieren* und warten Sie, bis die benötigten Installationskomponenten geladen wurden. Anschließend führt Sie ein Assistent durch die einzelnen Schritte der Installation.
6. Sobald der *Willkommen*-Dialog des Installations-Assistenten erscheint, verwenden Sie die Schaltflächen *Weiter* und *Zurück*, um zwischen den einzelnen Dialogschritten zu blättern.
7. Befolgen Sie in den Installationsdialogen die Anweisungen des Assistenten und wählen Sie die gewünschten Optionen. Sobald im Dialogfeld die *Installieren*-Schaltfläche angeklickt wird, beginnt die Installation von Visual Studio 2005.

Je nach Systemzustand werden das .NET Framework 2.0 und weitere erforderliche Systemkomponenten unter Windows installiert. Vor der Installation von Visual Studio 2005 fordert der Assistent Sie in separaten Dialogen zur Anerkennung der Lizenzbedingungen und zur Eingabe des Lizenzschlüssels auf.

Zudem können Sie den Installationsumfang in einem Dialogschritt wählen (Abbildung 1.4). Bei der Option *Standard* werden die häufig benötigten Komponenten von Visual Studio auf dem Zielsystem eingerichtet. Über die Option *Vollständig* können Sie den kompletten Funktionsumfang des Visual Studio 2005-Pakets installieren lassen. Falls Sie ausschließlich mit Visual Basic entwickeln oder über wenig Speicherplatz auf der Festplatte verfügen, lässt sich die Option *Benutzerdefiniert* im Dialogfeld markieren. Im Folgedialog zeigt der Installationsassistent dann die installierbaren Komponenten des Produkts an. Durch Löschen der Markierung der betreffenden Kontrollkästchen können einzelne Module von der Installation ausgeschlossen werden.

Während der Installation informiert Setup Sie über den weiteren Verlauf. Je nach Installationsmedium werden dabei die benötigten CDs (für Visual Studio 2005, SQL Server etc.) angefordert. Sie werden bei einer erfolgreichen Installation durch einen entsprechenden Abschluss-Dialog informiert.

Nach der Installation von Visual Studio 2005 erlaubt das Dialogfeld des Setup-Assistenten (Abbildung 1.3) Ihnen über den Hyperlink *Produktdokumentation installieren* die Hilfe zu Visual Studio 2005 sowie zum .NET Framework einzurichten. Sie benötigen zur Installation die MSDN-CDs. Weiterhin können Sie über den Hyperlink *Service Releases suchen* des Installations-Dialogs prüfen, ob Updates für Visual Studio 2005 oder für andere Komponenten vorhanden sind. Zur Überprüfung muss eine Internetverbindung bestehen und im Internet Explorer ist der Installation einer ActiveX-Komponente zur Überprüfung des Update-Bedarfs zuzustimmen.

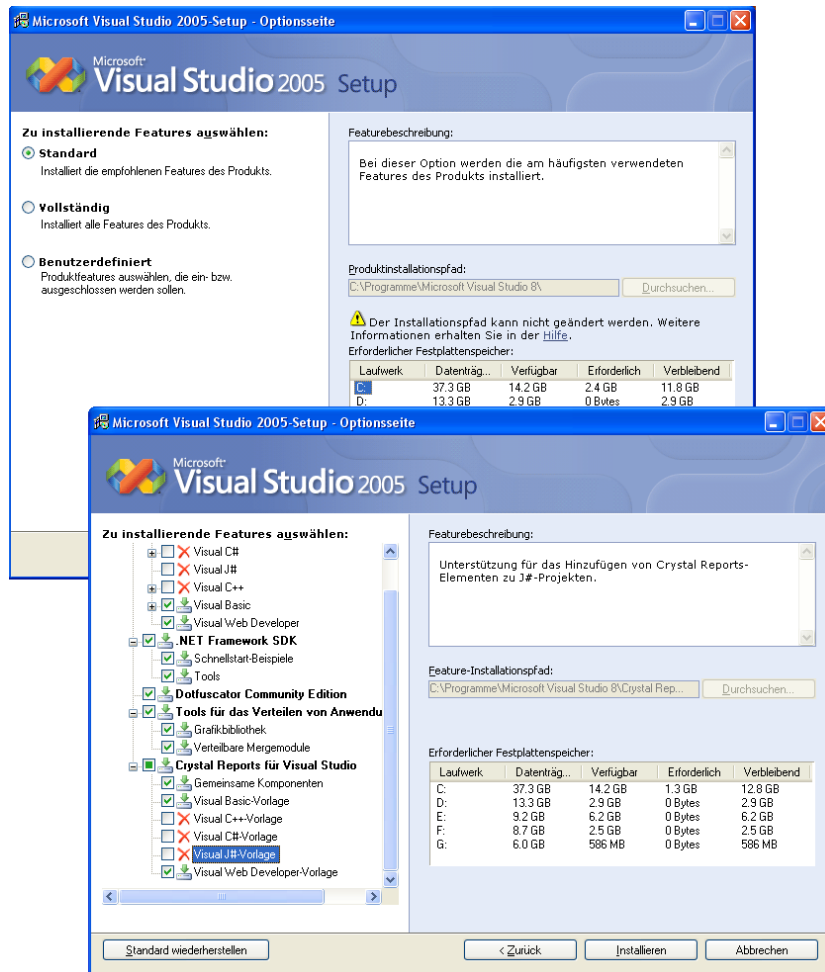


Abbildung 1.4: Auswahl der Installationsoptionen

Sobald Visual Studio 2005 installiert ist, können Sie das Programm über das Windows-Startmenü (Zweig *Programme* bzw. *Alle Programme/Microsoft Visual Studio 2005*) aufrufen.

Hinweis

In den ersten Vertriebsversionen, die durch Microsoft Deutschland herausgegeben wurden, fehlen leider die MSDN-CDs mit der Produkthilfe. Trifft dies bei Ihrem Paket zu, müssen Sie die betreffenden CDs von Microsoft anfordern.

1.4.2 Visual Basic 2005 Express Edition installieren

Verfügen Sie lediglich über die Visual Basic 2005 Express Edition? Die Installation dieses Pakets muss unter einem Administratorenkonto erfolgen. Die genauen Installationschritte hängen davon ab, ob Sie über die Version mit dem Webinstaller oder über die CD-Version verfügen.

Bei der Webinstaller-Version steht nur der Setup-Assistent in Form eines ausführbaren Windows-Programms zur Verfügung. Sobald die etwas über 2,8 Mbyte große Datei aus dem Internet geladen und dann gestartet wurde, führt ein Assistent Sie durch die Schritte zur Installation. Der Installer lädt die benötigten Module (zwischen 64 und gut 400 Mbyte) direkt über die Internetverbindung herunter und installiert dann das .NET Framework 2.0, SQL Server 2005 Express und auch die Visual Basic 2005 Express Edition.

Bei der CD-Version von Visual Basic 2005 Express Edition sind dagegen alle zur Installation benötigten Dateien enthalten. Legen Sie die CD ein und warten Sie, bis der Installationsassistent startet. Falls die Autorun-Funktion abgeschaltet ist, müssen Sie das Programm *Setup.exe* aufrufen. Danach werden Sie durch den Setup-Assistenten durch die Installationsschritte geführt.

In beiden Varianten fordert der Setup-Assistent Sie zur Anerkennung des Endbenutzer-Lizenzvertrags auf. Über die mit *Installieren* beschriftete Schaltfläche lässt sich die Installation anschließend anstoßen. Sie werden über den Installationsverlauf durch Fortschrittsanzeigen und Dialoge informiert. Nach der erfolgreichen Installation können Sie Visual Basic 2005 Express Edition über das Windows-Startmenü aufrufen.

1.4.3 Visual Studio reparieren, deinstallieren oder warten

Haben Sie Visual Studio 2005 im benutzerdefinierten Modus installiert, möchten aber nachträglich den Installationsumfang anpassen? Oder verursacht das Produkt Probleme und soll repariert bzw. deinstalliert werden?

1. Melden Sie sich unter einem Administratorenbenutzerkonto an und rufen Sie über das Startmenü die Systemsteuerung auf.
2. Wählen Sie das Symbol *Software* der Systemsteuerung per Doppelklick an, warten Sie, bis das Dialogfeld *Software* erscheint und stellen Sie sicher, dass die Schaltfläche *Programme ändern oder entfernen* am linken Dialogfeldrand angewählt wurde.
3. Suchen Sie in der Liste der installierten Programme den Eintrag für Visual Studio 2005 oder Visual Basic 2005 Express Edition und markieren Sie diesen durch Anklicken mit der Maus.
4. Anschließend wählen Sie die für den Eintrag angezeigte Schaltfläche *Ändern/Entfernen*.
5. Sobald der Dialog des Wartungs-Assistenten erscheint, klicken Sie auf den Hyperlink zum Anpassen des Funktionsumfangs, zum Reparieren oder zum Deinstallieren des Produkts.
6. Anschließend verwenden Sie die *Weiter*-Schaltfläche, um zwischen den Dialogen des Setup-Assistenten zu wechseln und legen Sie in den Folgedialogen die gewünschten Optionen fest.

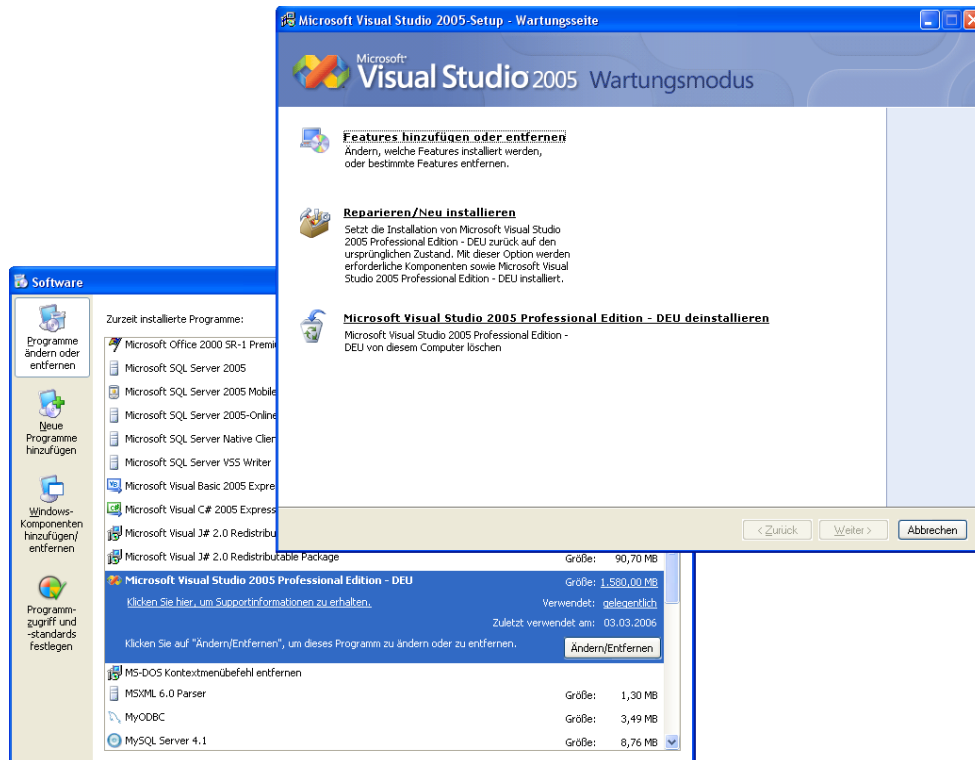


Abbildung 1.5: Aufruf des Wartungsmodus

Das genaue Aussehen der Dialoge des Setup-Assistenten hängt von der zugehörigen Produktvariante ab. Über den Hyperlink *Features hinzufügen oder entfernen* gelangen Sie zum Dialog, in dem Sie in Visual Studio 2005 zusätzliche Komponenten hinzufügen oder installierte Komponenten entfernen können. Beim Reparieren überprüft der Assistent die Einstellungen, kopiert ggf. die Installationsdateien erneut auf den Zielrechner und aktualisiert die Registrierungseinstellungen für das Produkt. Der Hyperlink *xxx deinstallieren* entfernt dagegen alle installierten Komponenten der Entwicklungsumgebung.

1.4.4 Die .NET Framework Redistributable installieren

Um auf einer Zielmaschine .NET-Anwendungen ausführen zu können, muss auf dieser das .NET Framework 2.0 installiert sein. Sie können sich die .NET Framework Redistributable 2.0 von den Microsoft Webseiten herunterladen (unter www.microsoft.com/germany nach dem Begriff suchen lassen). Zudem finden Sie die betreffende Datei auf der ersten CD von den Visual Studio 2005 bzw. auf der Visual Basic 2005 Express Edition-CD im Ordner `wcu\dotNetFramework`.

1. Melden Sie sich ggf. unter einem Benutzerkonto mit Administratorenrechten an, um über die Berechtigungen zum Installieren von Software zu verfügen.

2. Starten Sie die Installationsdatei *dotnetfx.exe* und bestätigen Sie im ersten Dialogschritt die Annahme der Lizenzvereinbarungen. Anschließend befolgen Sie die Anweisungen des Installations-Assistenten, der Sie durch die Schritte zum Entpacken der Daten und zum Einrichten des .NET Framework führt.
3. Sobald die Installation des .NET Framework erfolgreich abgeschlossen wurde, rufen Sie in einem zweiten Schritt die Installationsdatei *langpack.exe* auf, um die deutschen Spracherweiterungen zu installieren.

Sobald Sie die beiden Installationspakete erfolgreich ausgeführt haben, ist .NET Framework 2.0 auf der Zielformaschine vorhanden. Sie können anschließend mit Visual Studio 2005 oder mit Visual Basic 2005 Express Edition entwickelte .NET-Anwendungen auf dieser Maschine ausführen.

Hinweis

Alternativ können Sie statt der .NET Framework Redistributable das .NET Framework SDK auf der Zielformaschine installieren. Das SDK lässt sich von den Microsoft Webseiten herunterladen, liegt teilweise aber auch dem CD-Set von Visual Studio 2005 bei. Das .NET Framework SDK enthält Debugger, Beispiele und Dokumentation. Da der Installationsumfang aber wesentlich größer als bei der .NET Framework Redistributable ist, macht dies für Zielformaschinen, auf denen nur .NET-Anwendungen ausgeführt werden sollen, wenig Sinn.

1.4.5 .NET-Framework-Konfigurierung

Nach der Installation lässt sich das .NET Framework bei Bedarf noch konfigurieren. Sie können beispielsweise die Laufzeitsicherheitsrichtlinien für .NET-Anwendungen anpassen.

1. Zum Ändern der .NET-Framework-Konfigurierung melden Sie sich an einem Administratorenkonto an und öffnen das Fenster der Systemsteuerung (z.B. über das Startmenü).
2. Anschließend wählen Sie das Symbol *Verwaltung* per Doppelklick an. Im Fenster *Verwaltung* ist danach das Symbol *Microsoft .NET Framework 2.0 Konfigurierung* aufzurufen.
3. Sobald das Fenster *Microsoft .NET Framework 2.0 Configuration* erscheint, wählen Sie in der linken Spalte des Fensters die gewünschte Kategorie und klicken danach im rechten Fensterteil auf den Hyperlink der gewünschten Aufgabe.

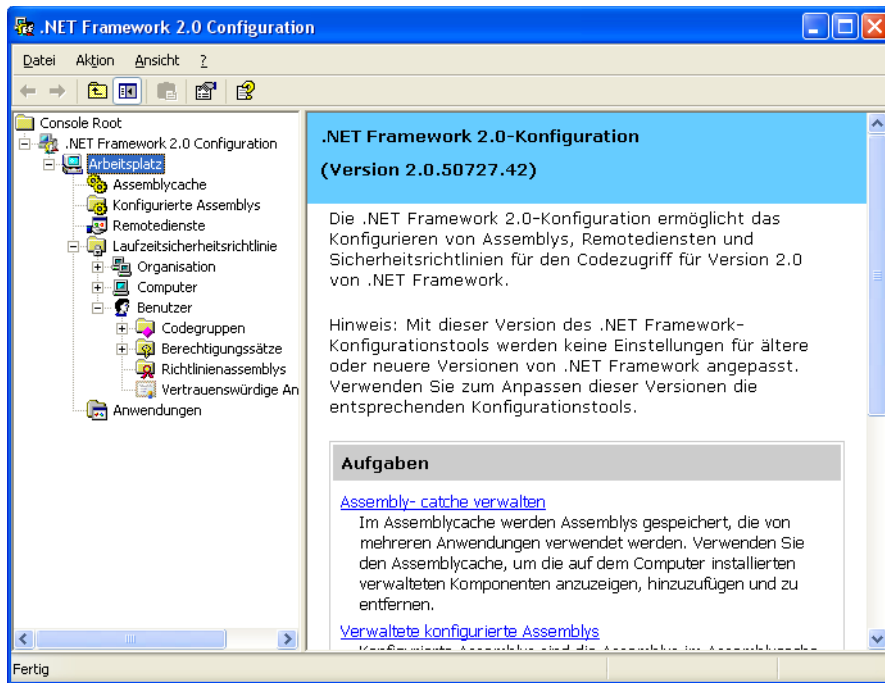


Abbildung 1.6: .NET-Framework-Konfiguration

Die Hyperlinks öffnen Dialoge, in denen Sie die Konfigurierung für die jeweilige Kategorie anpassen können. Details zu den jeweiligen Optionen finden Sie in den im Fenster eingblendeten Hilfetexten. Mit diesem Kapitel verfügen Sie über Informationen zur Installation der benötigten Entwicklungsumgebung/Infrastruktur. Im nächsten Kapitel lernen Sie die Schritte kennen, mit denen Visual-Basic-.NET-Programme übersetzt und getestet werden.

2

Visual-Studio-2005- Kurzeinführung

In diesem Kapitel erhalten Sie eine Kurzeinführung in die Entwicklung von Visual-Basic-Anwendungen unter Visual Studio 2005 bzw. in Visual Basic 2005 Express Edition. Weitere Abschnitte befassen sich mit dem Debuggen und Testen von .NET-Anwendungen. Zudem gibt es am Kapitelende einige Hinweise, wie Sie die Online-Hilfe der betreffenden Produkte nutzen und Informationen über die .NET-Framework-Klassenbibliothek abrufen. Eine umfassende Beschreibung der Visual-Studio-Funktionen finden Sie im Zweig *Entwicklungstools und Sprachen* | *Dokumentation zu Visual Studio*.

2.1 Die Entwicklungsumgebungen im Überblick

Zur Entwicklung von .NET-Anwendungen mit Visual Basic empfiehlt es sich, auf die von Microsoft angebotenen Entwicklungsumgebungen zurückgreifen. Für umfangreichere Projekte im professionellen Bereich wird dies Visual Studio 2005 sein. Alternativ lässt sich die kostenlose Visual Basic 2005 Express Edition zum Erstellen kleinerer Anwendungen verwenden. Diese Entwicklungsumgebung ist daher für Einsteiger, Studenten oder Hobbyprogrammierer besonders interessant. Nachfolgend finden Sie eine kurze Übersicht über diese beiden Entwicklungsumgebungen und deren Funktionen.

2.1.1 Die Visual Basic 2005 Express Edition im Überblick

Die Visual Basic 2005 Express Edition ist eine Entwicklungsumgebung, die alle Funktionen zum Erstellen von Visual-Basic-Anwendungen für .NET enthält. Das Programm lässt sich nach der Installation im Windows-Startmenü über den Zweig *Programme* (bzw. *Alle Programme*) über den Menüeintrag *Microsoft Visual Basic 2005 Express Edition* aufrufen. Das Programm meldet sich nach dem Start mit einem Anwendungsfenster, welches Menü- und Symbolleisten umfasst. Der Dokumentbereich des Fensters ist seinerseits in unterschiedliche Teilfenster unterteilt.

- Im mittleren Bereich des Dokumentfensters wird beim Programmstart eine Webseite mit der Bezeichnung *Startseite* eingeblendet (Abbildung 2.1). Über diese Seite können Sie bereits bearbeitete Projekte über Hyperlinks abrufen, neue Projekte anlegen oder in der Hilfe bzw. im Internet über die eingeblendeten Hyperlinks zum Thema recherchieren.
- Während der Bearbeitung eines Projekts verschwindet die Startseite im Hintergrund und das Fenster des Dokumentbereichs zeigt standardmäßig das Designfenster zum Entwurf von Windows-Formularen (Abbildung 2.2). Alternativ können in diesem Bereich auch die Codefenster von Visual Basic-Modulen, der Code des Formulars etc. abgerufen werden.

Neben der Startseite und den Fenstern zum Formularentwurf und zur Codeansicht lassen sich über die Befehle des Menüs *Ansicht* weitere (Teil-)Fenster am rechten, linken oder unteren Rand des Anwendungsfenster einblenden (Abbildung 2.2). Um den Anzeigebereich möglichst optimal zu nutzen, verwendet die Entwicklungsumgebung teilweise eine Registerkartendarstellung, in der die Teilfenster verwaltet werden. Über die am oberen bzw. unteren Dokumentrand eingeblendeten Registerreiter lassen sich die unterschiedlichen Inhalte jeweils in den Vordergrund holen.

- Das mit **Toolbox** betitelte (und standardmäßig in der linken oberen Spalte eingeblendete) Fenster enthält Schaltflächen zum Abrufen der Werkzeuge, mit denen sich Steuerelemente in Formulare einfügen lassen. Die Schaltflächen werden daher erst sichtbar, wenn ein Formularentwurf im mittleren Teil des Anwendungsfensters ausgewählt ist.

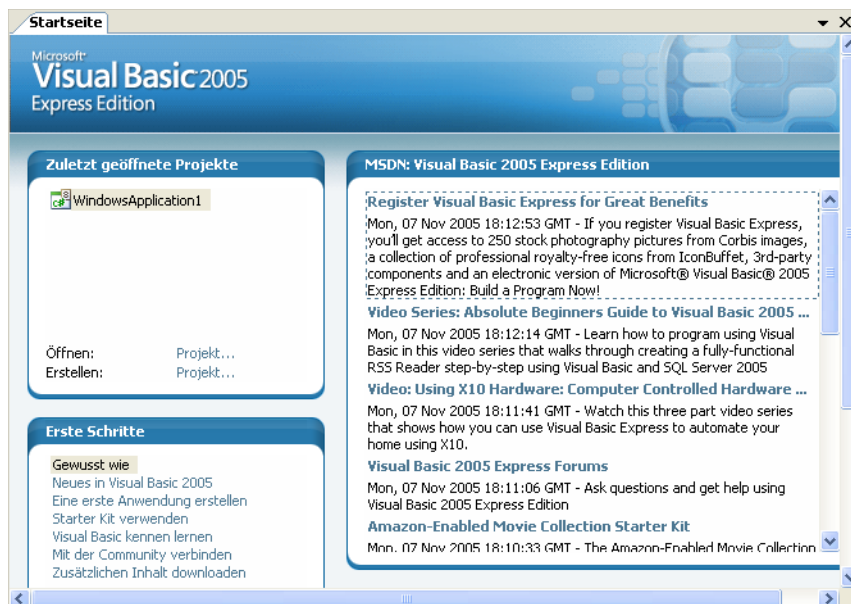


Abbildung 2.1: Startseite der Visual Basic 2005 Express Edition

- Ist SQL Server 2005 Express als Datenbank installiert, kann das mit **Datenbank-Explorer** bezeichnete Fenster (am linken oder rechten Rand) angezeigt werden. Der Datenbank-Explorer erlaubt, die Verbindungen zu SQL-Datenbanken oder anderen Datenquellen (Access, Excel-Tabellen etc.) zu verwalten. Zudem können Sie direkt Tabellen, Abfragen und andere Datenbankobjekte erzeugen, einsehen und handhaben.
- Die rechte Spalte des Anwendungsfensters enthält typischerweise im oberen Teil das Fenster des **Projektmappen-Explorers**. Dieser zeigt alle im aktuell bearbeiteten Projekt enthaltenen Elemente wie Module, Formulare, Klassen etc. In diesem Fenster lassen sich auch neue Elemente wie Formulare zu einem Projekt hinzufügen. Gegenüber Visual Studio gibt es allerdings die Einschränkung, dass standardmäßig keine Projektmappen angezeigt werden. Sie können daher i.d.R. auch keine weiteren Projekte zur Projektmappe hinzufügen! Nur beim Laden von in Visual Studio 2005 erstellten Projektmappen, die mehrere Projekte enthalten, wird die Projektmappe teilweise mit aufgeführt.

- In der rechten Spalte des Anwendungsfensters wird zudem meist das **Eigenschaftensfenster** (als separates Fenster oder als Registerkarte) eingeblendet. Dort werden jeweils die Eigenschaften des aktuell angewählten Elements (z.B. Projekt, Formular, Formularelement) angezeigt und können teilweise auch geändert werden.

Optional lassen sich weitere Teilfenster des Anwendungsfensters über die Befehle des Menüs *Ansicht* ein- oder ausblenden. In Abbildung 2.2 sehen Sie z.B. noch die Aufgabenliste, in der sich vom Benutzer zu erledigende Aufgaben notieren und verwalten lassen. Das in diesem Bereich eingeblendete Ausgabefenster zeigt z.B. die beim Übersetzen von der Entwicklungsumgebung bzw. beim Ausführen von Anwendungen ausgegebenen Meldungen. Treten Fehler beim Übersetzen oder Ausführen auf, werden diese im Fenster *Fehlerliste* aufgeführt.

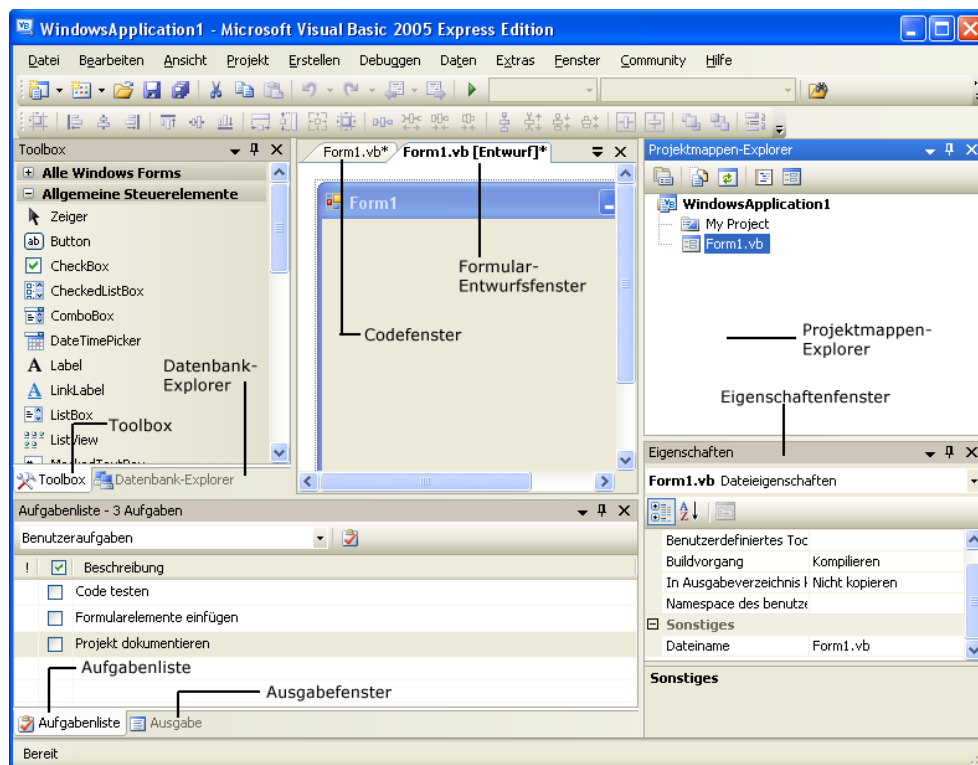


Abbildung 2.2: Anwendungsfenster der Visual Basic 2005 Express Edition

2.1.2 Visual Studio 2005 im Überblick

Ist Visual Studio 2005 auf dem Rechner installiert, steht Ihnen die Entwicklungsumgebung für verschiedene Programmiersprachen (C#, Visual Basic etc.) zur Verfügung. Auch dieses Programm lässt sich im Windows-Startmenü über den Zweig *Programme* (bzw. *Alle Programme*) über die Gruppe *Microsoft Visual Studio 2005* aufrufen. Das nach dem Start gezeigte Anwendungsfenster (Abbildung 2.3) stimmt in den Grundzügen mit

dem oben beschriebenen Fenster der Visual Basic 2005 Express Edition (Abbildung 2.2) überein. Statt des Datenbank-Explorers findet sich bei Visual Studio aber der Server-Explorer. Über dieses Teilfenster lässt sich nicht nur auf Datenverbindungen zu diversen Datenbanken bzw. Datenquellen zugreifen. Sie finden dort auch Einträge zum Zugriff auf Crystal Report-Dienste, Dienste, Ereignisprotokolle etc.

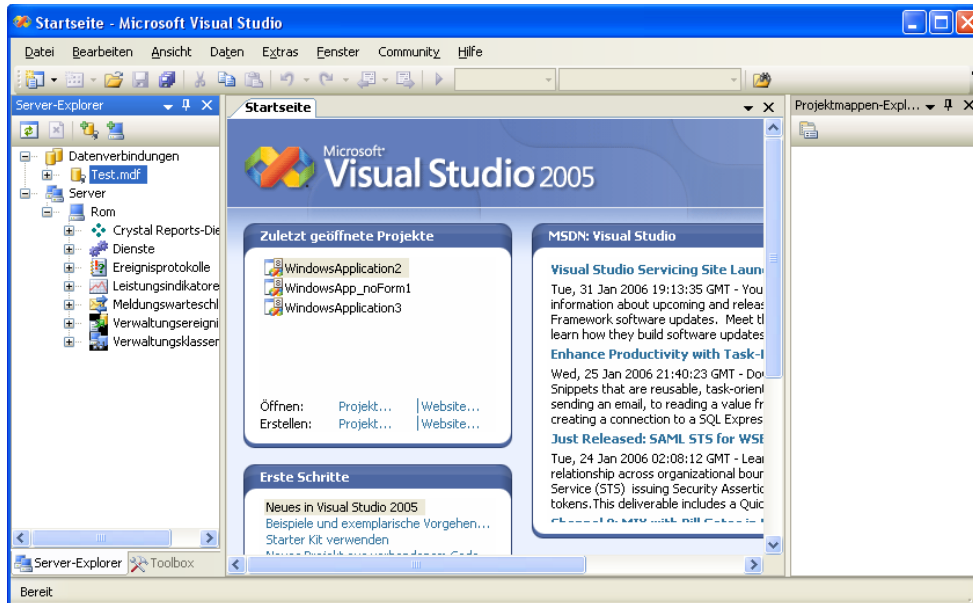


Abbildung 2.3: Visual Studio 2005

Die im mittleren Bereich des Dokumentfensters eingeblendete *Startseite* unterscheidet sich ebenfalls geringfügig vom Pendant der Visual Basic 2005 Express Edition. Über die Hyperlinks der Kategorien *Öffnen* bzw. *Erstellen* können Sie auf bestehende Projekte zurückgreifen oder neue Projekte anlegen.

Hinweis

Visual Studio 2005 unterstützt im Gegensatz zur Visual Basic 2005 Express Edition mehrere Programmiersprachen. Sie können also beim Anlegen von Projekten auf verschiedene Projekttypen zugreifen und im Projektmappen-Explorer wird die komplette Projektmappe angezeigt. Diese kann durchaus mehrere Projekte aufweisen, was in der Visual Basic 2005 Express Edition nicht möglich ist. Weiterhin stehen in Visual Studio 2005 mehr Vorlagen für neue Projektkomponenten als bei unter Visual Basic 2005 Express Edition zur Verfügung. Falls Sie die nachfolgenden Abschnitte lesen, aber mit der Visual Basic 2005 Express Edition arbeiten, kann es daher vorkommen, dass einzelne Features nicht vorhanden sind.

2.2 Arbeiten mit der Entwicklungsumgebung

Nachfolgend werden die grundlegenden Arbeitstechniken zum Erstellen von Projekten sowie zum Arbeiten mit der Entwicklungsumgebung vorgestellt.

2.2.1 Techniken zur Verwaltung der Teilfenster

Beim Arbeiten mit der Entwicklungsumgebung müssen Sie ggf. Teilfenster ein- oder ausblenden und ggf. auch verschieben. Da die Anwendungsfenster von Visual Studio und der Visual Basic Express Edition gleich aufgebaut sind, lassen sich in beiden Anwendungen die gleichen Techniken zur Verwaltung der Teilfenster nutzen.

- Teilfenster werden über die rechts in der Titelleiste des Fensters gezeigte Schaltfläche *Schließen* ausgeblendet (Abbildung 2.4).
- Zum erneuten Einblenden des Teilfensters können Sie anschließend den betreffenden Befehl (z.B. *Eigenschaftenfenster*, *Fehlerliste*, *Ausgabe* etc.) des Menüs *Ansicht* wählen.
- Die Schaltfläche *Automatisch im Hintergrund* (Abbildung 2.4) im Kopfbereich des Fensters erlaubt, das Fenster an der betreffenden Position zu fixieren oder zum automatischen Ausblenden freizugeben (stilisierter Pin ist zur Seite geklappt). In der in Abbildung 2.4 gezeigten Stellung der Nadelspitze wird das Teilfenster automatisch am Fensterrand fixiert – das Teilfenster bleibt ständig sichtbar. Klicken Sie auf die Schaltfläche eines fixierten Fensters, wird der Pin (mit nach links zeigender Spitze) von der Seite dargestellt. Die Entwicklungsumgebung minimiert das Fenster (sobald der Mauszeiger außerhalb des Fensters steht) und zeigt einen Registerreiter für das Fenster am betreffenden Seitenrand an. In Abbildung 2.4 ist die Schaltfläche für das Fenster *Eigenschaften* sichtbar, während das Fenster des *Projektmappen-Explorer* fixiert ist.

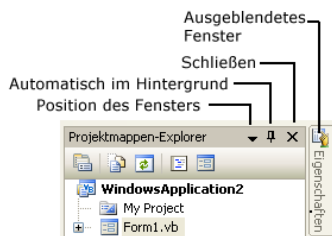


Abbildung 2.4: Schaltflächen im Kopf eines Fensters

- Doppelklicken Sie auf die Titelleiste eines am Rand andockten Teilfensters, wird dieses schwebend angeordnet. Ein zweiter Doppelklick auf das schwebende Fenster verankert dieses erneut an der alten Position.
- Ziehen Sie das Fenster über dessen Titelleiste auf die Leiste mit den Registerreitern am unteren Rand einer Fenstergruppe, ordnet die Entwicklungsumgebung das Fenster als neue Registerkarte in der Gruppe ein.
- Die Schaltfläche *Position des Fensters* im Kopfbereich des Fensters (Abbildung 2.4) öffnet ein Menü, über dessen Befehle Sie die Eigenschaften des Teilfensters vorgeben können. So lässt sich die Docking-Funktion zu- oder abschalten oder der Befehl *Dokument im Registerkartenformat* aktiviert die Darstellung des andockten Fensters als Registerkarte. Bei einem schwebend angeordneten Fenster klicken Sie mit der rechten Maustaste auf die Titelleiste. Dann stehen Ihnen die Befehle im Kontextmenü zur Verfügung.

Gegenüber früheren Visual-Studio-Versionen gibt es eine Neuerung beim Verankern von Fenstern an den Rändern des Dokumentfensters. Ziehen Sie die Titelleiste eines schwebenden Fensters (oder den zugehörigen Registerreiter eines verankerten Fensters) zum Rand des Hauptfensters, zeigt die Entwicklungsumgebung neun stilisierte Verankerungspunkte im Vordergrund an (Abbildung 2.5 zeigt sechs dieser Punkte). Sie können dann den Mauszeiger bei weiterhin gedrückter linker Maustaste zu einem der Verankerungspunkte ziehen. Sobald Sie die Maustaste über einen Verankerungspunkt loslassen, verankert die Entwicklungsumgebung das Fenster an der betreffenden Position. Was dabei genau passiert, hängt vom gewählten Verankerungspunkt ab.

- Das mittlere Symbol in der aus fünf Verankerungspunkten bestehenden Gruppe (Abbildung 2.5) bewirkt, dass das Dokumentfenster als Registerkarte im betreffenden Teilfenster einsortiert wird.
- Ziehen Sie den Mauszeiger auf einen der linken oder rechten äußeren Verankerungspunkte dieser Gruppe, wird das bestehende Teilfenster vertikal geteilt. Das neue Teilfenster wird dann rechts oder links vom aktuellen Teilfenster angeordnet.

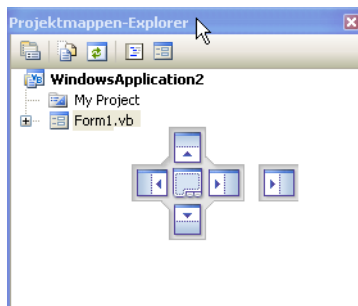


Abbildung 2.5: Fenster und eingeblendete Verankerungspunkte

- Ziehen Sie den Mauszeiger dagegen auf den oberen oder unteren Verankerungspunkt dieser Gruppe, teilt sich das bestehende Teilfenster horizontal. Das neue Teilfenster wird dann in der oberen oder unteren Hälfte des aktuellen Teilfensters angeordnet.
- Ziehen Sie dagegen den Mauszeiger zu einem der einzelnen Verankerungspunkte an den vier Dokumentseiten (in Abbildung 2.5 ist nur der Verankerungspunkt für den rechten Fensterrand zu sehen), verankert die Entwicklungsumgebung das Teilfenster am betreffenden Rand des Anwendungsfensters.

Sie können übrigens erkennen, wo das Teilfenster verankert wird. Solange sich der Mauszeiger beim Ziehen über einem Verankerungspunkt befindet, hebt die Entwicklungsumgebung den Umriss des Teilfensters grau hervor.

Hinweis

Im Menü *Fenster* finden Sie den Befehl *Fensterlayout zurücksetzen*, über den Sie die standardmäßige Fensteranordnung wieder erzwingen können. Das Menü stellt auch Befehle bereit, um Teilfenster horizontal oder vertikal zu splitten oder um Teilfenster aus- bzw. einzublenden.

2.2.2 Verwaltung der Projektdateien und -mappen

Visual Studio 2005 verwaltet die Dateien einer Anwendung als so genannte Projekte in Projektmappen. Die Projektmappe und die zu den Projekten gehörenden »Elemente« werden im Fenster des Projektmappen-Explorers angezeigt (Abbildung 2.6).

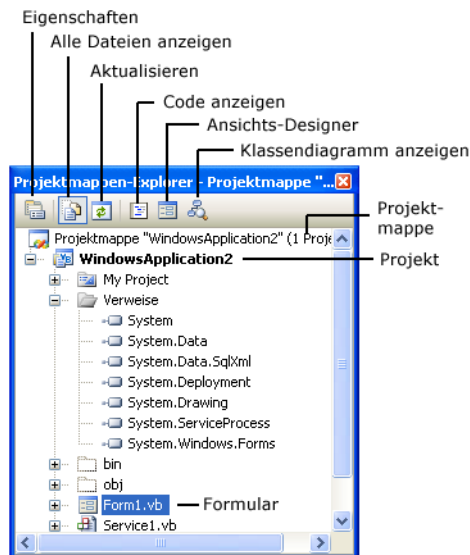


Abbildung 2.6: Fenster des Projektmappen-Explorers

Die Projektmappe enthält ein oder ggf. sogar mehrere Projekte. Ein Projekt wird durch einen separaten Eintrag gekennzeichnet und kann seinerseits wieder Einträge für Verweise, Module, Formulare, Referenzen etc. aufweisen. Über die Schaltfläche *Alle Dateien anzeigen* im Kopf des Projektmappen-Explorers lässt sich dabei wählen, ob alle Projektbestandteile oder nur eine eingeschränkte Auswahl angezeigt wird. Die Entwicklungsumgebung (z.B. Visual Studio 2005) sorgt automatisch dafür, dass alle zum Projekt gehörenden Dateien geladen bzw. gespeichert und übersetzt werden.

Auf Betriebssystemebene werden Projektmappen auf Ordner des Dateisystems abgebildet. Eine Projektmappe kann in Visual Studio ein oder mehrere Projekte aufweisen, deren Dateien ggf. auf Unterordner aufgeteilt werden. Bei der Visual Basic 2005 Express Edition lässt sich dagegen (standardmäßig) nur ein Projekt pro Projektmappe anlegen. Der im Projektmappen-Explorer mit »Projektmappe ...« bezeichnete Eintrag korrespondiert mit der in der Projektmappe hinterlegten *.sln*-Datei. Diese ist im XML-Format aufgebaut und enthält eine Übersicht über die jeweiligen Projekte der Projektmappe. Benennen Sie die Projektmappe im Fenster des Projektmappen-Explorers um, passt die Entwicklungsumgebung automatisch den Namen der *.sln*-Datei an. In den Projektordnern werden auch weitere zum Projekt gehörende Dateien (Module, Ressourcen, Formulare) hinterlegt. Der Visual Basic-Programmcode einzelner Module, Klassen oder Formulare findet sich in Dateien mit der Dateinamenerweiterung *.vb* im Projektordner. Die Erweiterung *.resx* steht für Ressourcendateien, die zur Aufnahme sogenannter Ressourcen (Bilder, Texte, Menüs), die im Programm benutzt werden, dienen. Beim Übersetzen

werden diese Ressourcen mit in die Assembly eingebunden. Die ausführbaren Programmdateien werden beim Compilieren des Projekts in Unterordner (z.B. in `\bin` oder `\bin\Debug`) abgelegt.

Normalerweise brauchen Sie sich aber um die Abbildung der Projektmappe auf Ordner des Dateisystems keine großen Gedanken zu machen. Um die Projektmappe samt den Projekten in der Entwicklungsumgebung zu laden, wählen Sie die `.sln`-Datei auf Windows-Ebene entweder per Doppelklick an. Oder Sie ziehen die betreffende Datei zum Fenster der Entwicklungsumgebung.

Hinweis

Der Pfad zum Ordner, der von der Entwicklungsumgebung zum Speichern der Projektmappen vorgeschlagen wird, lässt sich über den Befehl *Optionen* des Menüs *Extras* vorgeben. Wählen Sie im *Optionen*-Dialog den Zweig *Projekte und Projektmappen/Allgemein*.

Die Darstellung des Projektmappen-Explorers unterscheidet sich etwas zwischen Visual Studio 2005 und der Visual Basic 2005 Express Edition. In Visual Studio 2005 werden immer der Name der Projektmappe und darunter die in der Projektmappe hinterlegten Projekte eingeblendet (Abbildung 2.6). Legen Sie ein neues Projekt in der Visual Basic 2005 Express Edition an, zeigt der Projektmappen-Explorer dagegen nur das Projekt und die zugehörigen Komponenten. Laden Sie allerdings eine in Visual Studio 2005 erstellte Projektmappe, die mehrere Projekte enthält, zeigt der Projektmappen-Explorer auch in dieser Entwicklungsumgebung die Projektmappe und die zugehörigen Projekte. In der Regel wird es beim Laden jedoch zu einer Fehlermeldung mit dem Hinweis kommen, dass die aktuelle Version von Visual Studio die Struktur der Projektmappe nicht unterstützt. Aus diesem Grund enthalten die Projektmappen der Beispiele auf der Begleit-CD jeweils nur ein Projekt.

2.2.3 Ein Projekt neu anlegen

Um in der Entwicklungsumgebung (z.B. Visual Studio 2005) arbeiten zu können, müssen Sie entweder ein bereits vorhandenes Projekt laden oder ein neues Projekt anlegen. Die Entwicklungsumgebung bietet Ihnen hierzu verschiedene Möglichkeiten.

Zuletzt bearbeitete Projekte können Sie in der Startseite in der Kategorie *Zuletzt geöffnete Projekte* durch Anklicken des Eintrags laden. Alternativ lässt sich der Hyperlink *Projekt* in der Zeile *Öffnen* wählen. Ist die Startseite nicht sichtbar, bietet das Menü *Datei* z.B. den Befehl *Öffnen/Projekt öffnen*. In einem Dialogfeld lässt sich dann die Projektdatei auswählen.

Ein neues Projekt erzeugen Sie, indem Sie in der Kategorie *Zuletzt geöffnete Projekte* der Startseite den Hyperlink *Projekt* in der Zeile *Erstellen* wählen. Ist die Startseite nicht sichtbar, wählen Sie die Schaltfläche *Neues Projekt* und klicken im Menü dieser Schaltfläche auf die gleichnamige Schaltfläche. Zudem steht im Menü *Datei* der Befehl *Neu* mit dem Untermenübefehl *Projekt* zur Verfügung. Oder Sie wählen den Befehl *Neues Projekt* der Schaltfläche *Neu* der Symbolleiste.

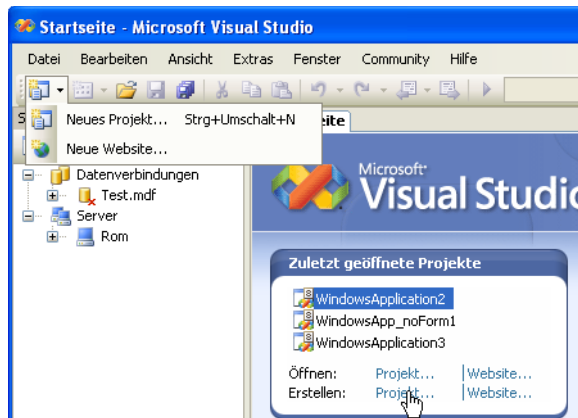


Abbildung 2.7: Anlegen eines neuen Projekts

Die Entwicklungsumgebung öffnet das Dialogfeld *Neues Projekt*, dessen Aussehen etwas von der installierten Programmumgebung abhängt. Die in Abbildung 2.8 gezeigte Rubrik *Projekttypen* ist nur in Visual Studio 2005 vorhanden und erlaubt die Auswahl des gewünschten Projekttyps. Für Visual Basic-Anwendungen ist daher der Zweig *Visual Basic* zu wählen. In der Visual Basic 2005 Express Edition fehlt die Rubrik *Projekttypen* und Sie können nur Visual Basic-Vorlagen wählen.

Sie müssen im Dialogfeld eine Projektvorlage wählen, den Projektnamen, den Projektmappennamen und ggf. den Zielordner zum Speichern anpassen und dann die OK-Schaltfläche anklicken (Abbildung 2.8).

Beim Schließen des Dialogfelds erzeugt die Entwicklungsumgebung in dem unter *Speicherort* gewählten Pfad einen Ordner für die Projektmappe und hinterlegt in diesem die *.sln*-Datei. In Visual Studio 2005 lässt sich im Dialogfeld *Neues Projekt* das Kontrollkästchen *Projektmappenverzeichnis erstellen* markieren. Dann werden die Projektdateien in separaten Projektordnern der Projektmappe hinterlegt. Ohne diese Markierung legt Visual Studio die Projektdateien im Ordner der Projektmappe ab (dies entspricht auch der Struktur, die die Visual Basic 2005 Express Edition beim Speichern neuer Projekte verwendet). Die Benennung des Ordners für die Projektmappe sowie des ggf. separat angelegten Projektordners wird aus dem Anwendungsnamen (Feld *Name*) abgeleitet. In zusätzlichen Unterordnern finden sich die ausführbare Programmdatei (Ordner *bin*) sowie die Debug-Informationen (Ordner *obj*). Die Entwicklungsumgebung generiert aus der Vorlage alle für das Projekt benötigten Dateien und Ordner und nimmt die für das Projekt gültigen Einstellungen vor. Haben Sie als Vorlage eine Windows-Anwendung gewählt, erscheint automatisch ein Formular im Fenster des Designers (Abbildung 2.9). Gleichzeitig wird eine Werkzeugleiste (Toolbox) zum Bearbeiten des Formulars am linken Rand eingeblendet.

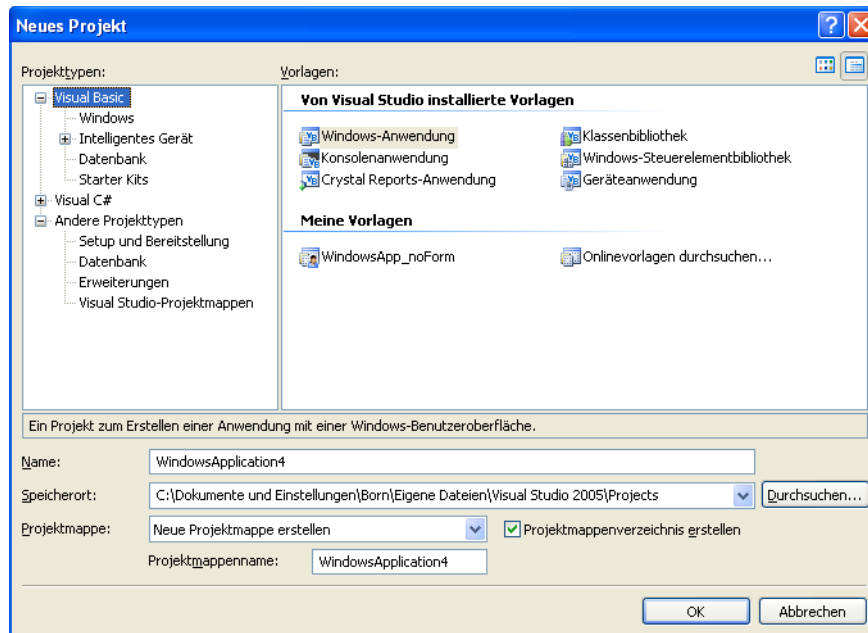


Abbildung 2.8: Dialogfeld zur Auswahl der Projektoptionen

Hinweis

In Visual Studio 2005 können Sie das Symbol der Projektmappe im Fenster des Projektmappen-Explorers mit der rechten Maustaste anwählen. Dann finden Sie Kontextmenübefehle, um z.B. die Projektmappe umzubenennen. Über den Kontextmenübefehl *Hinzufügen* lassen sich zudem Befehle abrufen, um ein neues oder ein bestehendes Projekt zur Projektmappe hinzuzufügen. Dies erleichtert die Verwaltung mehrerer Projekte. Weiterführende Hintergrundinformationen zum Verwalten von Projektmappen mit mehreren Projekten finden Sie in der Rubrik »Projektmappen mit mehreren Projekten« (Zweig *Entwicklungstools und Sprachen* | *Dokumentation zu Visual Studio* | *Integrierte Entwicklungsumgebung für Visual Studio* | *Verwalten von Projektmappen, Projekten und Dateien*) der Visual-Studio-Hilfe.

Die Visual Basic 2005 Express Edition erlaubt normalerweise nur die Verwaltung eines Projekts pro Projektmappe. Sie können aber mehrere Instanzen dieser Entwicklungsumgebung starten und in diesen separate Projekte halten.

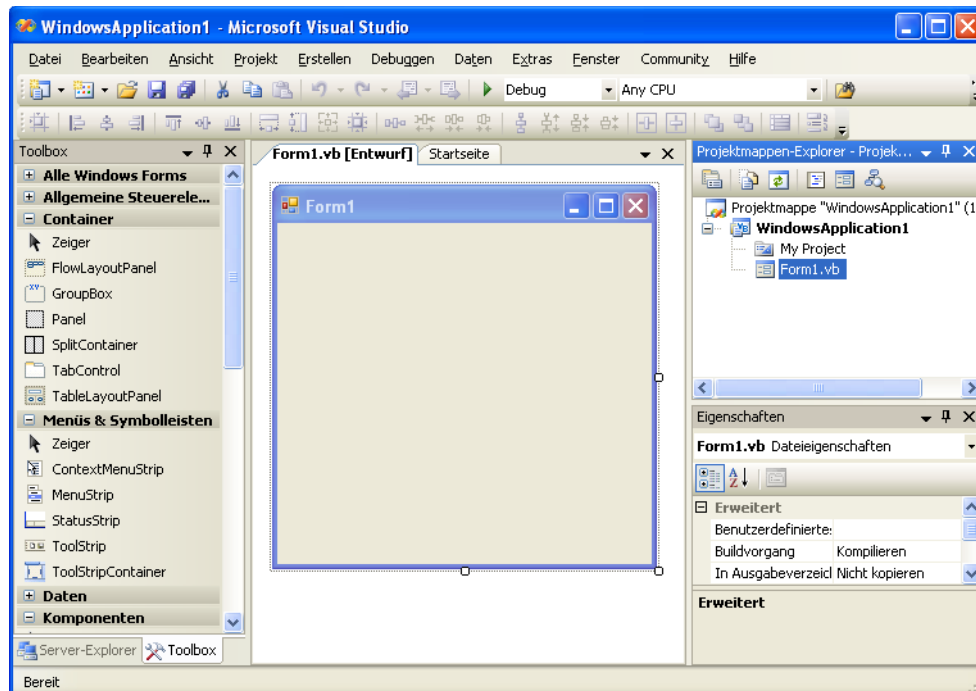


Abbildung 2.9: Neues Projekt einer Windows-Anwendung

Hinweise zu den Projektkategorien

Für die Entwicklung von Visual-Basic-Anwendungen können Sie verschiedene Vorlagen der Entwicklungsumgebung verwenden.

- **Windows-Anwendung:** Diese Vorlage wird benutzt, um Windows-Anwendungen zu erstellen. Die Vorlage fügt automatisch ein Formular für die Windows-Anwendung zum Projekt hinzu.
- **Konsolenanwendung:** Verwenden Sie diese Vorlage, erzeugt die Entwicklungsumgebung eine .NET-Anwendung, die im Fenster der Eingabeaufforderung ausgeführt wird.

Zusätzlich können Sie als Vorlage ein leeres Projekt (ohne weitere Module) oder spezielle Vorlagen für Klassenbibliotheken (diese erlauben wiederverwendbare Softwarebausteine zu erstellen), Windows-Steuerelementbibliotheken (lassen sich in Formulare einbinden oder von Windows-Anwendungen als Objekte nutzen), ASP.NET-Webanwendungen bzw. -Webdienste (für Active Server Pages-Seiten), Windows-Dienste (Windows-Anwendungen, die im Hintergrund ohne Benutzerinteraktion laufen), Webanwendungen (für Webseiten) etc. wählen.

2.2.4 Nutzen des Konfigurations-Managers

Für das in der Projektmappe hinterlegte Projekt (bzw. für die Projekte) lassen sich verschiedene Konfigurationen (Debug, Release) und Zielplattformen (x86- oder x64-CPUs) festlegen. Hierzu rufen Sie den Konfigurations-Manager (z.B. über den gleichnamigen Befehl des Menüs *Erstellen*) der Entwicklungsumgebung auf. Der Konfigurations-Manager meldet sich mit dem in Abbildung 2.10 gezeigten Dialogfeld.

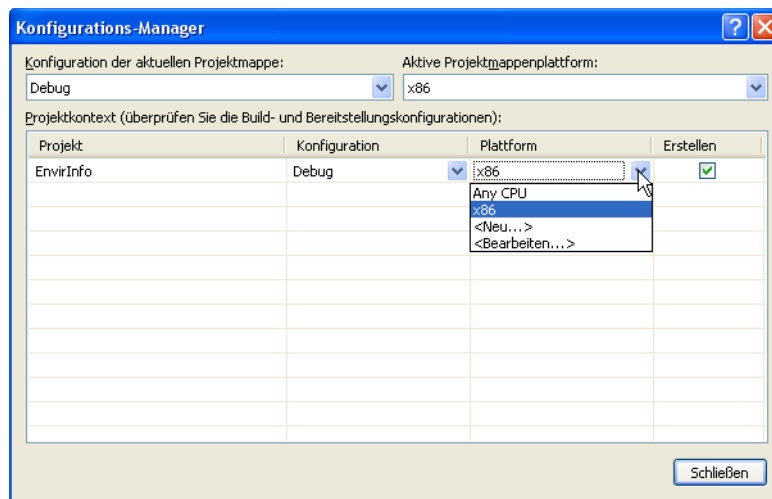


Abbildung 2.10: Konfigurations-Manager

Über das linke Listenfeld *Konfiguration der aktuellen Projektmappe* lässt sich zwischen den Werten *Debug*, *Release*, *Bearbeiten* und *Neu* wählen. Der Wert *Debug* legt die Einstellungen für den Debug-Modus (bei dem Debug-Informationen zum Code hinzugefügt werden) fest. Der Wert *Release* bestimmt die Konfiguration für die Projektvariante, die zur Verteilung im Produktiveinsatz vorgesehen ist. Mit *Bearbeiten* öffnet sich ein Dialogfeld, in dem Sie die Projektmappenkonfigurationen anpassen können (Einträge löschen oder umbenennen). Der Wert *Neu* erlaubt Ihnen eine neue Projektmappenkonfiguration unter einem wählbaren Namen anzulegen. Im Listenfeld *Aktive Projektmappenplattform* wählen Sie, ob das Projekt für alle CPUs (Wert *AnyCPU*) oder nur für x86-CPUs (Wert *x86*) bzw. nur für x64-CPUs (Wert *x64*) übersetzt werden soll. Die beiden letztgenannten Werte stehen im Listenfeld nur zur Verfügung, wenn Sie vorher den Wert *Neu* gewählt und dann die Projektmappenplattformwerte im Zusatzdialog freigegeben haben. Weitere Details zum Konfigurations-Manager lassen sich der Hilfe entnehmen.

2.2.5 Die Einstellungen des Projekts anpassen

Um ein Projekt zu übersetzen, verwendet die Entwicklungsumgebung projektspezifische Einstellungen (Projekteigenschaften), die von der gewählten Vorlage abhängen. In diesen Einstellungen ist beispielsweise hinterlegt, ob eine Konsolenanwendung oder ein Windows-Programm erstellt wird. Auf der gleichen Seite lässt sich der *.exe*-Datei auch ein Symbol zuweisen. Zudem können Sie wählen, ob Debug-Informationen beim Über-

setzen generiert werden sollen. Wenn Sie eine Projektvorlage für das neue Projekt übernehmen, besitzt diese bereits die für den Projekttyp passenden Standard-Einstellungen. Um die Projekteinstellungen nach eigenen Wünschen anzupassen, gehen Sie in folgenden Schritten vor.

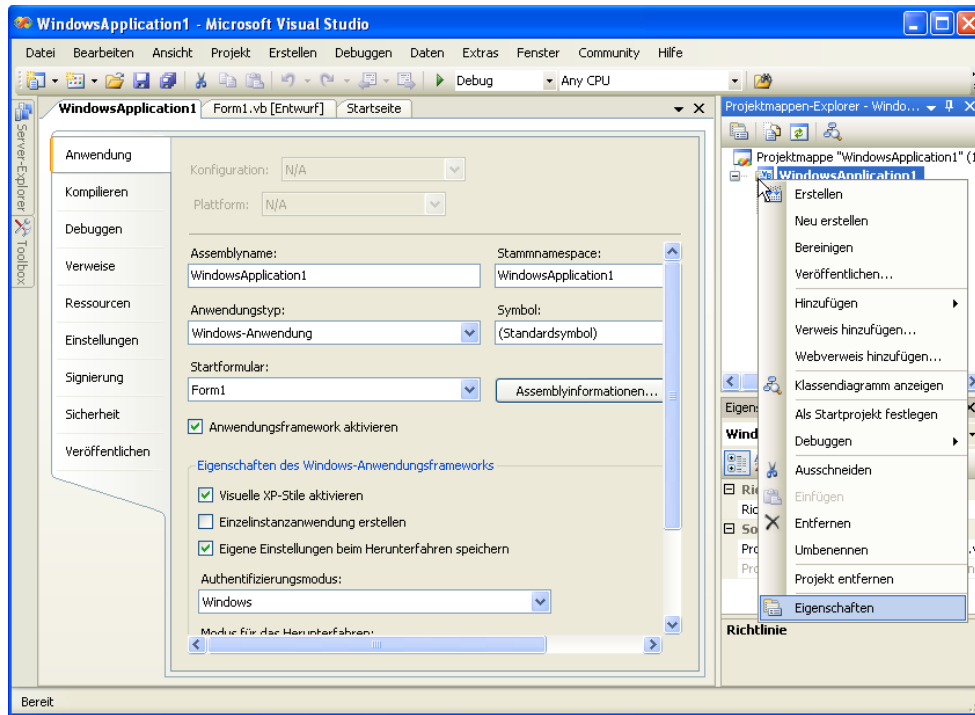


Abbildung 2.11: Anpassen der Projekteinstellungen im Projekt-Designer

1. Wechseln Sie zum Fenster des Projektmappe-Explorers. Das Fenster lässt sich ggf. über den gleichnamigen Befehl des Menüs *Ansicht* im Fenster der Entwicklungsumgebung einblenden. Im Projektmappe-Explorer sind die im Projekt enthaltenen Elemente aufgeführt (Abbildung 2.9).
2. Klicken Sie mit der rechten Maustaste auf das Symbol der VB-Projektdatei. In Abbildung 2.9 ist dies der Eintrag *WindowsApplication1*. Wählen Sie im Kontextmenü den Befehl *Eigenschaften* (Abbildung 2.11, rechts). Oder wählen Sie im Menü *Projekt* den Befehl *<Projektname>-Eigenschaften*.
3. Wählen Sie im eingeblendeten Fenster des Projekt-Designers (Abbildung 2.11, links) den Eintrag für die Eigenschaftenseite (*Anwendung*, *Kompilieren* etc.) und passen Sie die Eigenschaften an.
4. Anschließend können Sie das Fenster des Projekt-Designers über die *Schließen*-Schaltfläche wieder ausblenden.

Das Fenster des Projekt-Designers gruppiert die Eigenschaften in Kategorien und stellt diese auf mehreren Seiten (*Anwendung, Kompilieren* etc.) bereit. Sobald das Fenster des Projekt-Designers angezeigt wird, müssen Sie in der linken Spalte auf die gewünschte Seite (*Anwendung, Kompilieren* etc.) klicken. Dann werden die verfügbaren Eigenschaften im rechten Teil des Fensters angezeigt. Nachfolgenden finden Sie einige Hinweise zum Inhalt der einzelnen Eigenschaftenseiten des Projekt-Designers.

Seite Anwendung

Auf der Eigenschaftenseite *Anwendung* lassen sich die Eigenschaften der Anwendung vorgeben.

- Das Listenfeld *Ausgabetypp* legt den Anwendungstyp (Windows-Anwendung oder Konsolenanwendung) der .NET-Anwendung fest.
- Der im Feld *Assemblyname* hinterlegte Text definiert auch den Namen der Programmdatei beim Übersetzen der Anwendung. Das Feld *Stammnamespace* legt den Namensraum fest, unter dem die betreffende Komponente ggf. angesprochen werden kann (siehe auch *Kapitel 7*, Abschnitt »Bemerkung zur Definition des Namensraums«).
- Über das Listenfeld *Symbol* lässt sich der Anwendung ein Symbol zuweisen (siehe auch folgende Abschnitte).
- Im Feld *Startformular* legen Sie fest, welches Modul oder welche Klasse aus dem Projekt beim Ausführen der Anwendung aufzurufen ist. Bei Windows-Anwendungen lassen sich nur Formulare angeben. Ist das Kontrollkästchen *Anwendungsframework* nicht markiert, können Sie ggf. auch Prozeduren innerhalb des Projekts als Startmodul angeben.

Markieren Sie das Kontrollkästchen *Anwendungsframework*, lassen sich weitere Optionen des Windows-Anwendungsframework (z.B. XP-Stil, Modus zum Herunterfahren, Begrüßungsbildschirm etc.) setzen.

- Das Kontrollkästchen *XP-Stil* bewirkt z.B., dass Schaltflächen Windows XP-konform mit runden Ecken dargestellt werden.
- Standardmäßig lassen sich die erzeugten .NET-Anwendungen in mehreren Instanzen starten. Um dies bei Windows-Anwendungen zu verhindern, markieren Sie das Kontrollkästchen *Einzelinstanzanwendung erstellen*. Der Versuch, das Programm mehrfach zu starten, bewirkt dann lediglich, dass die Fenster der bereits laufenden Instanz in den Vordergrund des Desktops geholt werden.
- Markieren Sie das Kontrollkästchen *Eigene Einstellungen beim Herunterfahren speichern*, sichert die Anwendung Benutzereinstellungen beim Beenden.
- Über das Listenfeld *Begrüßungsbildschirm* lässt sich ein Formular auswählen, welches beim Anwendungsstart kurzzeitig eingeblendet wird.
- Das Listenfeld *Modus beim Herunterfahren* erlaubt festzulegen, wann die .NET-Anwendung terminiert. Dies kann beim Schließen des Startformulars durch den Benutzer erfolgen.

Über die Schaltfläche *Assemblyinformationen* öffnet sich ein Dialogfeld, in dem Zusatzdaten zur Assembly (z.B. Firma, Sprache, Version etc.) hinterlegt werden können.

Symboldatei für Windows-Anwendungen zuweisen

Windows-Anwendungen ist üblicherweise ein Symbol zugeordnet, welches im Ordner-Fenster bei *.exe*-Dateien erscheint. Dieses lässt sich der .NET-Anwendung über das bereits erwähnte Listefeld *Symbol* der Eigenschaftenseite *Anwendung* zuweisen. Stellen Sie den Wert des Listefelds auf den Wert »Durchsuchen« (Abbildung 2.12). Im eingeblendeten Dialogfeld *Vorhandene Datei zum Projekt hinzufügen* können Sie dann eine *.ico*-Datei auswählen und über die *Öffnen*-Schaltfläche des Dialogs als Symbol einbinden.

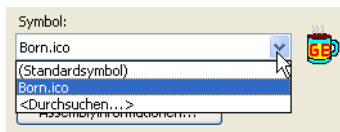


Abbildung 2.12: Einbinden einer Symboldatei

Beachten Sie aber, dass die Entwicklungsumgebung nur Dateien im *.ico*-Format unterstützt. Diese Dateien können Symbole mit unterschiedlichen Abmessungen (32x32 Pixel, 48x48 Pixel etc.) und verschiedenen Farbtiefen enthalten.

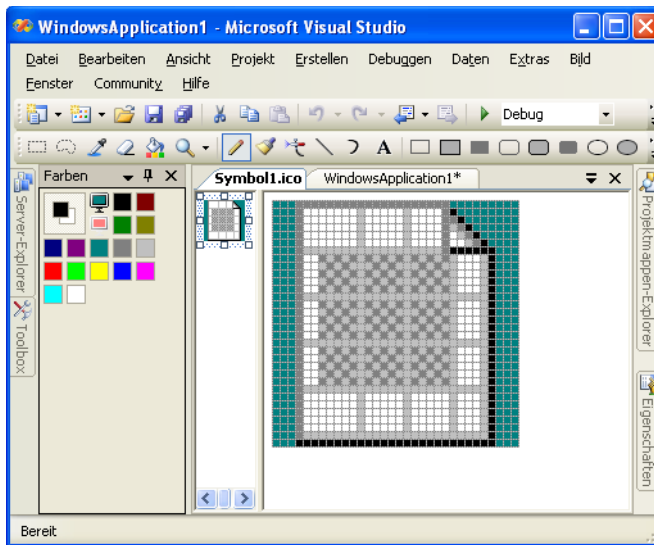


Abbildung 2.13: Erstellen einer Symboldatei

Tipp

Symboldateien im *.ico*-Format lassen sich direkt in Visual Studio 2005 entwerfen. Wählen Sie im Menü *Datei* des Visual Studio-Anwendungsfensters die Befehle *Neu/Datei*. Im Dialogfeld *Neue Datei* ist die Vorlage *Symboldatei* zu wählen und über *Öffnen* zu bestätigen. Nach dem Schließen des Dialogfelds öffnet Visual Studio automatisch das Fenster des Symboleditors (Abbildung 2.13). Sie finden dort eine Farbpalette sowie eine Symbolleiste mit den Werkzeugen des Editors. In einem Designbereich können Sie dann die Symbole entwerfen. Zum Abschluss müssen Sie die Schaltfläche *Speichern* der Symbolleiste oder die Befehle *xxx.ico speichern* bzw. *xxx.ico speichern unter* des Menüs *Datei* wählen. Besitzer von Visual Basic 2005 Express Edition sind zum Entwurf von Symboldateien auf die im Internet verfügbaren Icon-Editoren angewiesen. Falls Sie keine eigenen Symbole entwerfen möchten, können Sie Symboldateien auch direkt aus dem Internet herunterladen (z.B. www.vbarchiv.net).

Hinweise zu den restlichen Eigenschaftenseiten

Neben der Seite *Anwendung* weist der Projekt-Designer einige zusätzliche Eigenschaftenskategorien auf (siehe Abbildung 2.11). Hier noch eine kurze Übersicht über diese Kategorien und die Bedeutung ausgewählter Eigenschaften.

- **Kompilieren:** Über die Seite *Kompilieren* können Sie die Vorgaben zum Übersetzen wie die Prüfungen auf nicht deklarierte Variable oder implizite Typumwandlungen vorgeben (siehe auch Abschnitt »Was bedeuten Option Explicit und Option Strict« in Kapitel 3). Setzen Sie den Wert des Listenfelds *Option Strict* auf »On«. Dies verhindert Programmfehler, die sich durch implizite Typkonvertierungen einschleichen können. Der Wert »On« im Listenfeld *Option Explicit* bewirkt, dass der Visual-Basic-Compiler nicht deklarierte Variable bei der Übersetzung bemängelt. Das Listenfeld *Option Compare* erlaubt Ihnen die Werte für Vergleiche auf »Binär« oder »Text« zu setzen. Der Wert des Felds *Konfiguration* legt fest, ob beim Übersetzen Debug-Informationen erstellt werden oder ob reiner Anwendungscode zu erzeugen ist. Über *Plattform* wird ggf. die Zielplattform (x86- oder x64-Prozessor) für die Codeerzeugung angezeigt. Die Zielplattform lässt sich über den Dialog *Erweiterte Kompilereinstellungen* (Schaltfläche *Erweiterte Kompilereinstellungen*) zwischen x86, x64 und Any-CPU umsetzen. Beide Listenfelder stellen Werte, die im Konfigurations-Manager für die Projektmappe definiert wurden, zur Verfügung. Das Textfeld *Ausgabepfad erstellen* gibt das Unterverzeichnis vor, in den die übersetzte Anwendung zu schreiben ist. Die Entwicklungsumgebung weist in Abhängigkeit von der Einstellung im Listenfeld *Konfiguration* verschiedene Unterordner (z.B. *\Debug*, *\Release*) im Verzeichnis *\bin* für die Speicherung zu. Über die Schaltfläche *Erweiterte Kompilereinstellungen* lässt sich ein Zusatzdialog öffnen, in dem Sie Optimierungsoptionen, den Umfang der generierten Debuginfos, Kompilierungskonstanten etc. anpassen bzw. definieren können. Die Liste *Bedingung* erlaubt Ihnen für verschiedene Übersetzungsvorgaben (z.B. Impliziter Typ "Objekt") bestimmte Warnstufen vorzugeben (z.B. Warnung, wenn ein impliziter Typ im Programm benutzt wird). Über Kontrollkästchen am Seitenende lassen sich Warnungen bei der Übersetzung oder das Generieren von XML-Dokumentation unterdrücken.

Hinweis

Die Compilereinstellungen für *Option Strict*, *Option Explicit* etc. können Sie auch global für alle VB-Projekte der Entwicklungsumgebung setzen, indem Sie im Menü *Extras* den Befehl *Optionen* wählen. Sie finden die Optionen dann im Zweig *Projekte und Projektmappen/VB-Standard*. Zudem wird in den folgenden Kapiteln teilweise die Möglichkeit genutzt, die globalen oder projektspezifischen Compilereinstellungen durch entsprechende Anweisungen im Quellcode eines Moduls oder einer Klasse zu überschreiben.

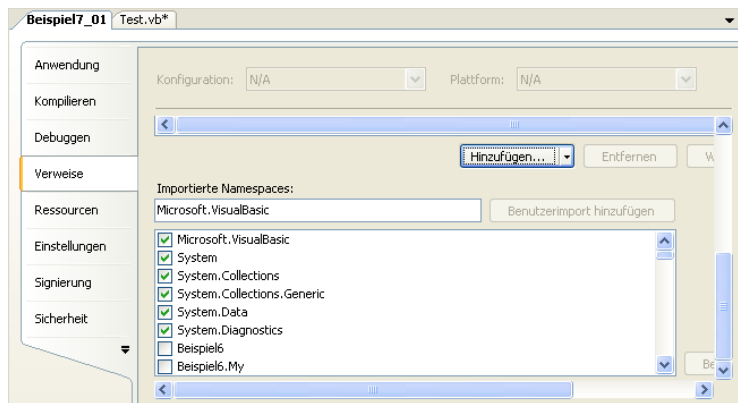


Abbildung 2.14: Namensräume importieren

- *Debuggen*: Auf dieser Seite lässt sich vorgeben, was beim Debuggen einer .NET-Anwendung in der Entwicklungsumgebung passieren soll. Sie können über die Optionsfelder der Gruppe *Startaktion* die Anwendung oder externe Programme bzw. den Browser starten. In der Gruppe *Startoptionen* lassen sich Arbeitsverzeichnisse oder Befehlszeilenargumente verwenden (siehe auch *Kapitel 13* im Abschnitt »Zugriff auf Programmargumente«).
- *Verweise*: Die Seite *Verweise* zeigt Ihnen, welche Verweise (z.B. auf die Assemblies der .NET Framework Klassenbibliotheken) im Projekt definiert sind und erlaubt Verweise auf Klassenbibliotheken oder Webseiten im Projekt einzubinden (siehe auch Abschnitt »Verweise im Projekt einfügen« weiter unten). Zudem finden Sie auf der Seite Optionen, um zusätzlich Namensräume im Projekt zu importieren (Abbildung 2.14).
- *Ressourcen*: Die Seite *Ressourcen* erlaubt Ihnen Ressourcen (Texte, Bilder, Symbole etc.) zu definieren bzw. zum Projekt hinzuzufügen. Auf diese Ressourcen lässt sich im Programmcode zugreifen. Auf der Seite *Einstellungen* lassen sich Anwendungseinstellungen unter bestimmten Namen hinterlegen und mit Werten initialisieren. Dies erlaubt es, Anwendungseinstellungen in Konfigurationsdateien zu hinterlegen. Auf diese Technik wird in *Kapitel 16* in einem Beispiel eingegangen.
- *Signierung*: Über die Seite *Signierung* lassen sich die Assemblies des Projekts mit Signaturen versehen. Voraussetzung ist aber, dass auf der betreffenden Maschine gültige Zertifikate eines Trust Centers installiert sind. Über das Kontrollkästchen *ClickOnce-Manifeste signieren* lassen sich die Zertifikate auswählen, mit denen die Manifeste bei der mittels ClickOnce bereitgestellten Installationspakete signiert werden. Die Schalt-

fläche *Aus Speicher* ermöglicht dann, installierte Zertifikate auszuwählen, während mit der Schaltfläche *Aus Datei* eine lokal in einem Ordner gespeicherte Zertifikatsdatei angegeben werden kann. Über die Schaltfläche *Testzertifikat* lässt sich ein (nicht von einem Trust-Center bestätigtes) Zertifikat zum Testen erzeugen. Markieren Sie das Kontrollkästchen *Assembly signieren*, um das Zertifikat auswählen und die Assembly beim Übersetzen signieren zu lassen.

- *Sicherheit*: Auf der Seite *Sicherheit* lassen sich Vorgaben für die Codesicherheit anpassen und über die Seite *Veröffentlichen* werden die Vorgaben für Installationspakete vergeben.

Details zu den verschiedenen Eigenschaften eines Projekts sowie deren Anpassung im Projekt-Designer liefert die Visual Studio 2005-Hilfe, die Sie über das *Hilfe*-Menü der Entwicklungsumgebung abrufen können.

Hinweis

Die Beschreibung der einzelnen Projektoptionen im Projekt-Designer finden Sie im Zweig *Entwicklungstools und Sprachen* | *Dokumentation zu Visual Studio* | *Integrierte Entwicklungsumgebung für Visual Studio* | *Verwalten von Projektmappen, Projekten und Dateien* | *Projekteigenschaften* der Hilfe.

2.2.6 Projektkomponenten hinzufügen

Projekte bestehen bei Visual Studio 2005 bzw. bei der Visual Basic 2005 Express Edition aus verschiedenen Komponenten. Beim Anlegen eines Projekts aus einer Vorlage generiert die Entwicklungsumgebung automatisch die benötigten Projektkomponenten. Bei Bedarf können Sie aber zusätzliche Komponenten wie Steuerelemente, Module, Klassen, Formulare etc. zum Projekt hinzufügen.

1. Öffnen Sie das Menü *Projekt* und wählen Sie einen der eingeblendeten Befehle, um Formulare, Benutzersteuerelemente, Komponenten, Klassen, Verweise oder geerbte Komponenten einzubinden (Abbildung 2.15, Hintergrund).
2. Wählen Sie dann im angezeigten Dialogfeld *Neues Element hinzufügen* (Abbildung 2.15, Vordergrund) die Vorlage mit dem einzufügenden Element aus und schließen Sie das Dialogfeld über die *Hinzufügen*-Schaltfläche.

Sobald das betreffende Element im Projekt eingefügt wurde, erscheint dieses als Eintrag im Projektmappen-Explorer. Ein Doppelklick auf ein Element öffnet dessen Ansicht im Fenster der Entwicklungsumgebung.

Hinweis

Standardmäßig verwendet Visual Basic Klassen, um die Prozeduren der Anwendung, Formulare und weitere Softwarebausteine abzulegen. Module stellen eine Sonderform von Klassen mit verschiedenen Einschränkungen dar und werden z.B. bei Consoleanwendungen verwendet. In Teil 2 dieses Buchs finden Sie eine Einführung in diese Thematik. Die Zahl der verfügbaren Vorlagen für neue Elemente unterscheidet sich übrigens zwischen Visual Studio 2005 und Visual Basic 2005 Express Edition. Einige Vorlagen aus Visual Studio 2005 stehen in der Express Edition nicht zur Verfügung.

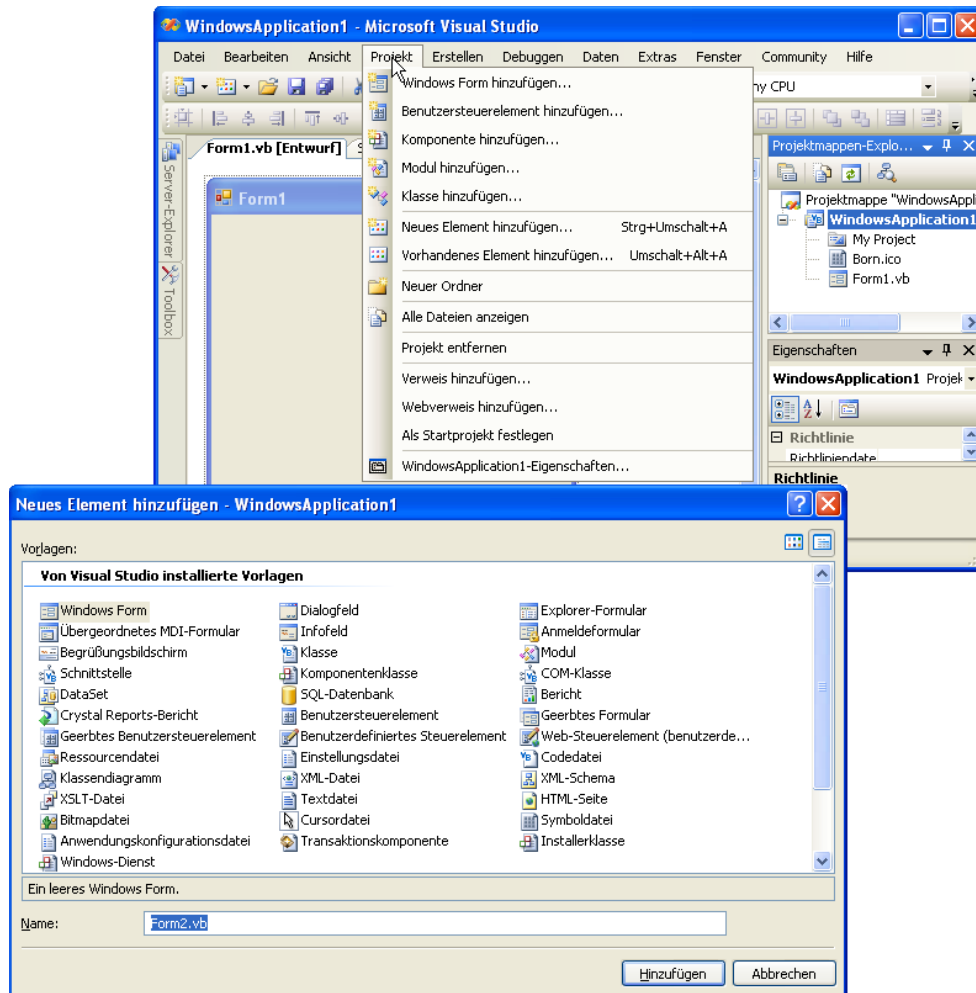


Abbildung 2.15: Neues Projektelement hinzufügen

Elemente umbenennen, entfernen, ausblenden und Code übertragen

Klicken Sie Elemente eines Projekts im Projektmappen-Explorer mit der rechten Maustaste an, öffnet sich ein Kontextmenü mit diversen Befehlen (z.B. zum Umbenennen des Elements). Über den Kontextmenübefehl *Löschen* lässt sich ein Element aus der Projektmappe entfernen. Sie müssen den Vorgang lediglich über die OK-Schaltfläche des angezeigten Zusatzdialogs bestätigen. Bei diesem Vorgang wird auch die Quelldatei des betreffenden Elements im Ordner des Projekts gelöscht.

Möchten Sie den Quellcode eines Elements erhalten, sollten Sie den Kontextmenübefehl *Aus Projekt ausschließen* verwenden. Dann entfernt die Entwicklungsumgebung das Element zwar aus dem Projektmappen-Explorer, belässt die Datei jedoch in der Projektmappe.

Dateien mit existierendem Code (z.B. aus anderen Projekten oder aus über den Befehl *Aus Projekt ausschließen* entfernten Elementen) lassen sich direkt über den Befehl *Vorhandenes Element hinzufügen* im Menü *Projekt* einbinden. Weiterhin können Sie Code aus anderen Projekten markieren und über die Windows-Zwischenablage in das Codefenster des aktuell gewählten Elements einfügen.

Zum Sichern eines Projektelements (Form, Modul, Klasse etc.) klicken Sie dieses im Projektmappen-Explorer an. Dann wählen Sie im Menü *Datei* den Befehl *<Name> Speichern unter*, wobei *<Name>* für den Namen des Elements steht. Dieser Name ist mit dem Dateinamen identisch, der im Projekt zur Speicherung der Komponente verwendet wird. Sie können den Dateinamen über die Eigenschaften des Elements anpassen sowie beim Speichern im Dialogfeld einen neuen Namen wählen. Die Dateinamenerweiterung wird abhängig vom gewählten Element vergeben.

2.2.7 Eigene Projektvorlagen erzeugen

Die Entwicklungsumgebung stellt Ihnen bereits Vorlagen für verschiedene Komponenten und Projektzwecke zur Verfügung. Falls Sie jedoch bestimmte Komponenten oder Einstellungen immer wieder verwenden, empfiehlt es sich, hierzu eine benutzerspezifische Vorlage zu erstellen. Leiten Sie dann ein neues Projekt von dieser Vorlage ab, werden alle Komponenten und Einstellungen automatisch im neuen Projekt wirksam. Dies soll jetzt an einem konkreten Beispiel skizziert werden. Die Entwicklungsumgebung erlaubt Ihnen beim Anlegen eines Projekts für eine .NET-Anwendung nur die Entscheidung zwischen einer Konsolenanwendung und einer Windows-Anwendung mit einem Formular. Gelegentlich ist es aber erwünscht, eine .NET-Anwendung zu haben, die zwar als Windows-Anwendung übersetzt wird, die aber kein Formular aufweist. Die Anwendung könnte bei Bedarf Benutzermeldungen über einfache Dialoge ausgeben. Es soll jetzt eine solche Projektvorlage erzeugt werden.

1. Starten Sie die Entwicklungsumgebung und legen Sie ein neues Projekt vom Typ *Konsolenanwendung* an. Geben Sie als Projektnamen ggf. »Windows-Anwendung-Formularlos« im Dialogfeld *Neues Projekt* an.

Der Projekttyp *Konsolenanwendung* besitzt automatisch ein Modul als Element und verzichtet auf Formulare. Bei Bedarf könnten Sie jetzt, wie auf den vorhergehenden Seiten gezeigt, weitere Komponenten zur Projektmappe hinzufügen. Im aktuellen Beispiel kann jedoch darauf verzichtet werden. Jetzt müssen Sie noch die Projekteigenschaften so anpassen, dass an Stelle der Konsolenanwendung eine Windows-Anwendung erstellt wird.

2. Klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf die Komponente des Visual Basic-Projekts (hier den Eintrag »Windows-Anwendung-Formularlos«) und wählen Sie im Kontextmenü den Befehl *Eigenschaften*.
3. Im eingeblendeten Fenster des Projekt-Designers wählen Sie die Seite *Anwendung*. Im Eigenschaftsfenster des Projekt-Designers darf das Kontrollkästchen *Anwendungsframework aktivieren* nicht markiert sein (da das Projekt kein Formular mehr aufweist).
4. Anschließend setzen Sie den Wert des Listfelds *Anwendungstyp* auf »Windows-Anwendung« (Abbildung 2.16).

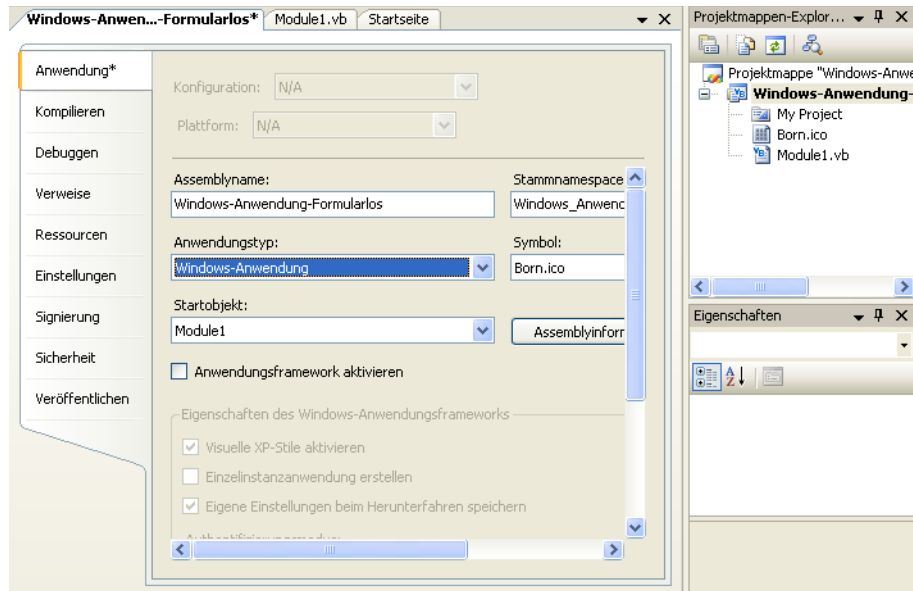


Abbildung 2.16: Projekteinstellungen

5. Stellen Sie sicher, dass im Listenfeld *Startobjekt* der Name des im Projekt eingefügten Moduls (hier *Modul1*) aufgelistet wird. Zudem können Sie über das betreffende Listenfeld ein Symbol zuweisen.

Die im Listenfeld *Startobjekt* angegebene Komponente wird beim Start der .NET-Anwendung aufgerufen und muss den auszuführenden Code aufweisen. Bei den Standard-Windows-Anwendungen wird dagegen die Komponente *Form1* als Startobjekt benutzt, so dass automatisch das Formular beim Programmstart eingeblendet wird.

6. Zum Speichern des Projekts als Vorlage wählen Sie im Menü *Datei* den Befehl *Vorlage exportieren*.
7. Sobald der Export-Assistent startet, markieren Sie im ersten Dialogschritt das Optionsfeld *Projektvorlage* (Abbildung 2.17, Hintergrund) und klicken dann auf die *Weiter*-Schaltfläche.
8. Im Folgedialog (Abbildung 2.17, Vordergrund) sollten Sie ggf. den Vorlagennamen anpassen und eine Vorlagenbeschreibung im Dialogfeld eingeben. Belassen Sie die Markierung der beiden Kontrollkästchen, weisen Sie der Vorlage ggf. ein eigenes Vorlagensymbol zu und schließen Sie den Assistenten über die *Fertig stellen*-Schaltfläche.

Wenn Sie anschließend ein neues Projekt anlegen, wird die Vorlage im Dialogfeld *Neues Projekt* in der Rubrik *Benutzerdefinierte Vorlagen* angezeigt. Bei Visual Studio ist lediglich darauf zu achten, dass in der Rubrik *Projekttypen* nur der Eintrag »Visual Basic« und nicht der Unterzweig »Windows« markiert ist.

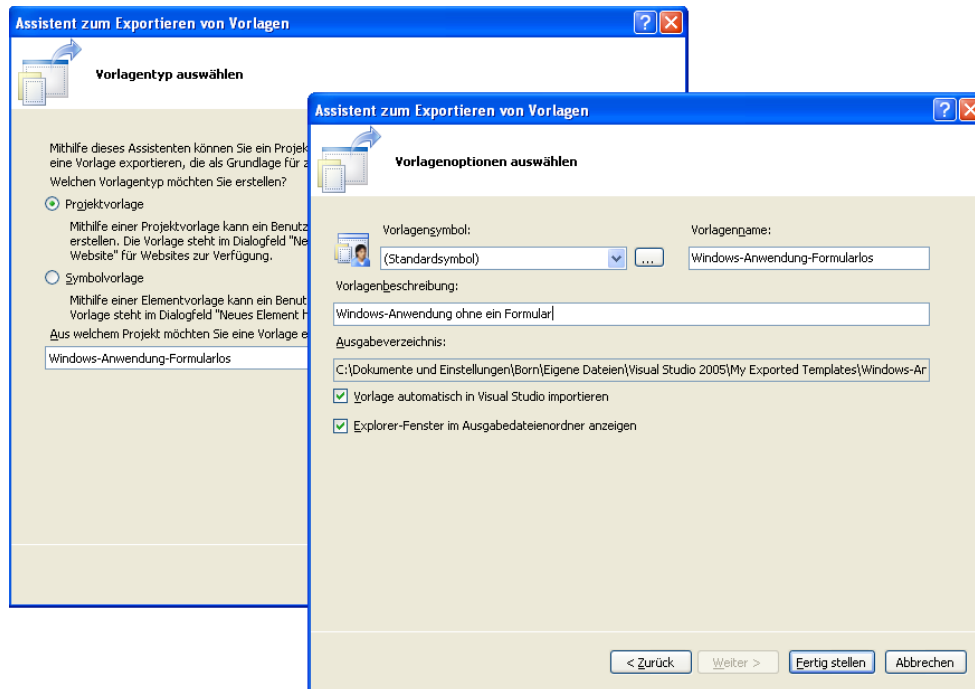


Abbildung 2.17: Exportieren einer Projektvorlage

2.2.8 Die wichtigsten Fenster der Entwicklungsumgebung

Die *Standard*-Symbolleiste der Entwicklungsumgebung besitzt am rechten Rand mehrere Schaltflächen, über die sich verschiedene Fenster ein- und ausblenden lassen (Abbildung 2.18).



Abbildung 2.18: Schaltflächen zum Einblenden von Fenstern

Die Schaltflächen sind mit *Projektmappen-Explorer*, *Eigenschaftenfenster*, *Objektbrowser*, *Toolbox* und *Startseite* benannt. Klicken Sie auf eine Schaltfläche, wird das betreffende Fenster wahlweise ein- oder ausgeblendet. Die Schaltfläche *Toolbox* blendet die üblicherweise am linken Rand sichtbare Toolbox (siehe Abbildung 2.9) mit den Schaltflächen ein, über die sich Steuerelemente für Formulare abrufen oder weitere Komponenten in ein Projekt aufnehmen lassen.

Der Projektmappen-Explorer

Das Fenster des Projektmappen-Explorers (siehe Abbildung 2.19) dient zur Verwaltung der Projektkomponenten. Sie können die Zweige der einzelnen Kategorien ähnlich wie die Ordneranzeige im Windows-Explorer erweitern oder zum Symbol reduzieren. Hierzu müssen Sie lediglich auf das Plus- oder Minuszeichen vor dem jeweiligen Eintrag klicken. Die Schaltflächen unterhalb der Titelleiste des Fensters ermöglichen Ihnen, die Eigenschaften der gewählten Komponente abzurufen oder bei Formularen das Design- bzw. Codefenster zu öffnen. Um den Projektnamen, die Bezeichnung der Projektmappe oder eines Elements zu ändern, klicken Sie das betreffende Element mit der rechten Maustaste an und wählen im Kontextmenü *Umbenennen*. Anschließend können Sie einen neuen Namen vergeben. Alternativ können Sie die Namen im Eigenschaftenfenster der betreffenden Komponente anwählen. Beim Umbenennen von Projektmappen oder Projekt-Elementen passt die Entwicklungsumgebung übrigens auch den zugehörigen Dateinamen an. Ein Doppelklick auf das Symbol eines Elements öffnet diese in einem Fenster in der Entwicklungsumgebung.

Das Eigenschaftenfenster

Visual Studio 2005 (bzw. Visual Basic 2005 Express Edition) verwaltet fast jedes Element als ein Objekt, welches Eigenschaften aufweisen kann. Stellen Sie durch Anwahl der Schaltfläche *Eigenschaftenfenster* in der *Standard*-Symbolleiste sicher, dass das Eigenschaftenfenster in der Entwicklungsumgebung sichtbar ist. Dann reicht es, eine Komponente im Projektmappen-Explorer oder ein Element in einem Formular anzuklicken, um dessen Eigenschaften im Eigenschaftenfenster einzublenden.

Der Inhalt des Eigenschaftenfensters hängt dabei vom jeweiligen Objekt ab. Ein Projektelement wird andere Eigenschaften als ein Formular oder Steuerelement aufweisen. Abbildung 2.19 zeigt im rechten Teil das Eigenschaftenfenster mit den Dateieigenschaften eines im Projektmappen-Explorer angewählten Formularmoduls. Klicken Sie dagegen ein Formular im Formulardesigner an, zeigt das Eigenschaftenfenster die Formulareigenschaften. In den folgenden Kapiteln lernen Sie den Entwurf von Formularen kennen, wobei auch deren Eigenschaften über ein solches Fenster angepasst werden.

2.2.9 Arbeiten im Codefenster

Die Entwicklungsumgebung stellt ein eigenes Codefenster zur Bearbeitung des Quellcodes von Modulen oder Formularen bereit. Sobald Sie eine Komponente (die Programm-anweisungen aufnehmen kann) im Projektmappen-Explorer per Doppelklick anwählen, wird der Quellcode im Codefenster angezeigt. Bei einem Formular müssen Sie im Fenster des Ansicht-Designers allerdings auf den Formularentwurf doppelklicken, um zur Codeansicht zu gelangen. Alternativ können Sie in der Symbolleiste des Projektmappen-Explorers auf die Schaltfläche *Code* klicken.

Bei einer Visual Basic-Komponente erscheinen die Anweisungen im Codefenster in der Syntax dieser Sprache (Abbildung 2.19, links). Sie können auf eine Zeile im Fenster der Codeansicht klicken und die Programmanweisungen ergänzen.

Hinweis

Schlüsselwörter (dies sind die in Visual Basic festgelegten Befehle wie *Sub*, *End Sub* etc.) werden im Codefenster in blauer Farbe hervorgehoben. Normale Anweisungen erscheinen schwarz, während Kommentare grün dargestellt werden. Mit einer blauen geschlängelten Linie unterstrichene Textstellen weisen auf einen Fehler in der Anweisung hin. Bei Variablenamen, die mit einer grünen geschlängelten Linie unterstrichen sind, signalisiert die Entwicklungsumgebung, dass die Variable noch unbenutzt ist. Einige Stellen in Kommentaren sind ggf. mit grauem Text ausgeführt. Es handelt sich meist um XML-Anweisungen, die im Code hinterlegt wurden. Die in früheren Versionen von Visual Studio grau dargestellten Einträge, die auf automatisch vom Formulardesigner generierten Code verwiesen, gibt es in Visual Studio 2005 in neuen Projekten nicht mehr. Die Entwicklungsumgebung lagert den betreffenden Code in separate Dateien aus.

Geben Sie eine Anweisung der Art *Sub Main()* ein und drücken Sie die -Taste, erzeugt der Codeeditor automatisch das benötigte Gegenstück *End Sub*. Dies funktioniert auch bei anderen Schlüsselwörtern wie *For* etc. (siehe folgende Kapitel).

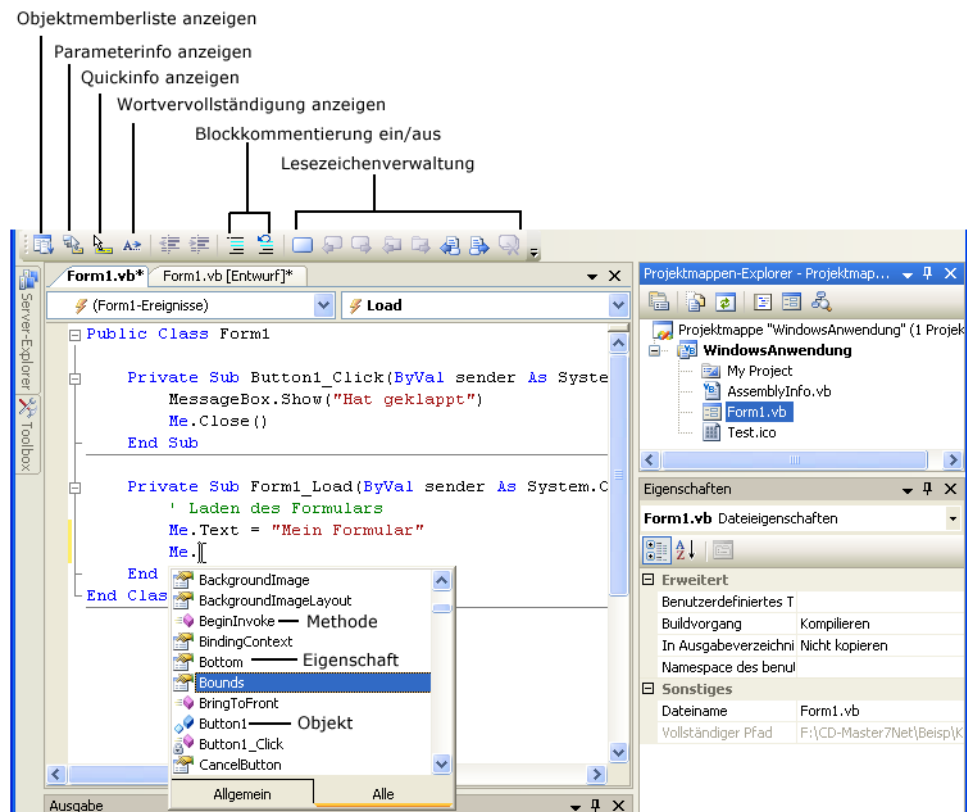


Abbildung 2.19: Code- und Eigenschaftsfenster (rechts)

Fügen Sie Programmcode für Objekte und deren Methoden etc. im Codefenster ein, wird die Intellisense-Unterstützung wirksam. Sobald der Editor einen von Ihnen eingegebenen Ausdruck erkennt (z.B. nach Abschließen eines Objektnamens durch einen Punkt), erscheint ein Kontextmenü mit den für das betreffende Objekt verfügbaren Klassenmitgliedern (Eigenschaften, Konstanten und Methoden). Dies ist in Abbildung 2.19 zu sehen, wo das Schlüsselwort *Me* als Objektname durch einen Punkt abgeschlossen wurde. Im Kontextmenü erscheinen dann die auch als Member bezeichneten Klassenmitglieder. Das vor dem Namen des Members eingeblendete Symbol erlaubt die Unterscheidung zwischen Objekten, Eigenschaften und Methoden. Klicken Sie auf einen Eintrag oder betätigen Sie die - bzw. -Taste, wird der Name automatisch im Code übernommen.

Hinweis

Klicken Sie auf eine entsprechende Codestelle, können Sie die betreffenden Informationen auch direkt über die in Abbildung 2.19 in der Symbolleiste gezeigten Schaltflächen abrufen.

Bei Methoden oder Funktionen wird nach Eingabe des Methodennamens automatisch eine QuickInfo mit der Beschreibung der Schnittstelle dieser Methode eingeblendet (Abbildung 2.20).

```
MessageBox.Show("Hat geklappt")
Public Shared Function Show(text As String) As System.Windows.Forms.DialogResult
```

Abbildung 2.20: QuickInfo zu einer Methode

Die Projektelemente werden in Klassen, Module, Prozeduren etc. unterteilt. Das Codefenster besitzt zwei Listenfelder am oberen Rand, mit denen Sie zwischen den Methoden der jeweiligen Komponente wechseln können. Abbildung 2.21 zeigt die beiden Listenfelder beim geladenen Objekt *Form1*.

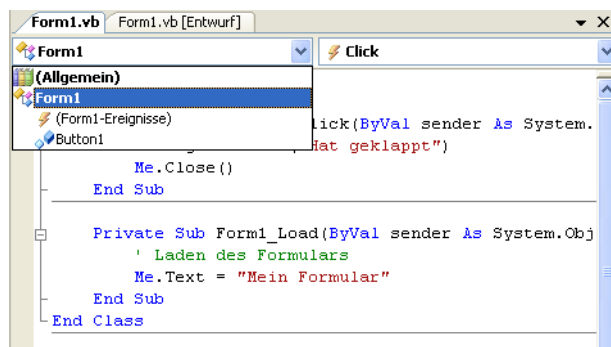


Abbildung 2.21: Listenfelder zur Auswahl der Klassen und Methoden

- Das linke Listenfeld aus Abbildung 2.21 dient zum Abrufen unterschiedlicher Klassennamen. Arbeiten Sie mit einem Formular, kann dieses Steuerelemente enthalten. Diese Steuerelemente (Schaltflächen, Texte, Felder etc.) lassen sich über ihre Klassennamen abrufen. In diesem Beispiel sehen Sie das Objekt *Button1*, welches eine im Formular eingefügte Schaltfläche repräsentiert.
- Eine Klasse kann mehrere Methoden bereitstellen, die sich über das rechte Listenfeld wählen lassen. Bei dem in Abbildung 2.21 gezeigten Formular werden im rechten Listenfeld die Namen der definierten Ereignisprozeduren für die Klasse *Form1* aufgeführt. Dadurch gelangen Sie direkt zum Code der Prozedur.

Ansonsten bietet der Codeeditor die üblichen Funktionen, um Codeteile zu markieren, in die Zwischenablage zu übernehmen oder aus der Zwischenablage einzufügen. In der Symbolleiste des Codefensters gibt es zudem Schaltflächen (Abbildung 2.19), mit denen sich markierte Zeilen auf einen Rutsch als Kommentarzeilen auszeichnen lassen oder die das Einfügen von Lesezeichen erlauben. Über weitere Schaltflächen lässt sich zu gesetzten Lesezeichen springen. Dadurch können Sie noch zu bearbeitende Codeteile im Projekt markieren und sehr schnell im Editor anspringen.

2.2.10 Der Objektbrowser

.NET Framework stellt eine Klassenbibliothek mit spezifischen Funktionen zur Verfügung. Zudem lassen sich weitere Klassen im Projekt einbinden. Jede Bibliothek enthält Informationen über die Objekte, Eigenschaften, Methoden, Ereignisse und Konstanten. Über das Fenster des Objektbrowsers lassen sich Informationen über die Member einer Klasse (sprich über die Objekte, Methoden und Eigenschaften) abrufen (Abbildung 2.22). Den Objektbrowser rufen Sie über den Befehl *Objektbrowser* des Menüs *Ansicht*, durch Drücken der Funktionstaste **[F2]** oder über die Schaltfläche *Objektbrowser* der *Standard-Symbolleiste* auf. Der Objektbrowser wird als Teilfenster im Bereich des Codefensters in der Entwicklungsumgebung eingeblendet.

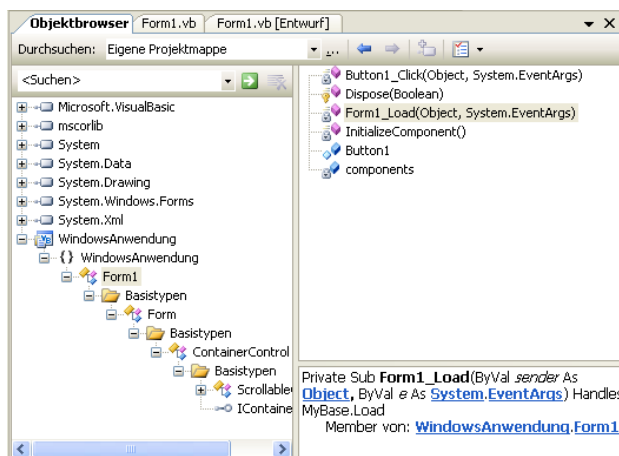


Abbildung 2.22: Objektbrowser

Die dargestellten Einträge lassen sich über das Listenfeld *Durchsuchen* sowie über die Schaltflächen des Fensters nach verschiedenen Kriterien filtern und sortieren. In der linken Liste *Objekte* zeigt der Objektbrowser Klassen an (Abbildung 2.22). Markieren Sie einen solchen Eintrag, werden die in dieser Klasse verfügbaren Elemente (Member) in der rechten Kategorie angezeigt. In Abbildung 2.22 ist beispielsweise die Klasse *Form1* gewählt und Sie sehen im rechten Teilfenster die jeweiligen Elemente dieser Klasse. Dort erkennen Sie auch den Eintrag *Button1*, der für die Schaltfläche des Formulars steht.

Klicken Sie auf einen Eintrag des Objektbrowsers, zeigt dieser dessen Schnittstelle bzw. Beschreibung im Fußteil der rechten Spalte an. Bei Klassen erkennen Sie die Definition und über welchen Namensraum die Klasse erreichbar ist. Bei Einträgen in der rechten Spalte werden die Werte von Konstanten oder die Schnittstellen von Methoden angezeigt. Über den Objektbrowser können Sie sich also sehr schnell über die Definition der Member informieren.

Bei umfangreichen Objektbibliotheken ist es recht aufwändig, ein bestimmtes Element zu finden. Klicken Sie auf das Textfeld *Suchen* im Kopf des Fensters und tippen Sie den Namen des gesuchten Elements ein. Sie können dabei den vollständigen Namen oder nur einen Teilbegriff vorgeben. Anschließend genügt ein Mausklick auf die rechts neben dem Textfeld befindliche *Suchen*-Schaltfläche (oder das Drücken der -Taste). Die Ergebnisse der Suche werden anschließend in der linken Spalte eingeblendet. Handelt es sich um einen Member im Codeteil des Projekts, ruft ein Doppelklick auf dem betreffenden Eintrag die zugehörige Stelle im Codeeditor auf. Bei einem Member aus einer eingebundenen Klassenbibliothek wird dessen Beschreibung im Objektkatalog angezeigt. Weitere Informationen finden Sie in der Hilfe unter dem Stichwort Objektbrowser.

2.2.11 Verweise im Projekt hinzufügen

In den folgenden Kapiteln lernen Sie, dass eine .NET-Anwendung Teile der .NET-Framework-Klassenbibliothek nutzen kann. Bevor Sie jedoch auf die betreffenden Klassen und deren Member zugreifen können, muss Visual Basic wissen, wo diese Klassen im Namensraum zu finden sind. Die Vorlagen für neue Projekte sorgen dafür, dass die wichtigsten Verweise im Projekt vorhanden sind. Sie haben aber die Möglichkeit, zusätzliche Verweise einzubinden.

1. Klicken Sie im Fenster des Projektmappen-Explorers auf die im Kopfbereich eingeblendete Schaltfläche *Alle Dateien anzeigen*. Dann wird ein Zweig mit den bereits bestehenden Verweisen eingeblendet (Abbildung 2.23).
2. Klicken Sie im Fenster des Projektmappen-Explorers den Zweig *Verweise* oder das Symbol der Projektdatei (z.B. der Windows-Anwendung) mit der rechten Maustaste an und wählen Sie im Kontextmenü den Befehl *Verweis hinzufügen*.
3. Wählen Sie im dann eingeblendeten Dialogfeld *Verweis hinzufügen* die Registerkarte der gewünschten Kategorie. Anschließend müssen Sie den gewünschten Eintrag für den Verweis suchen und per Maus anklicken (Abbildung 2.23).

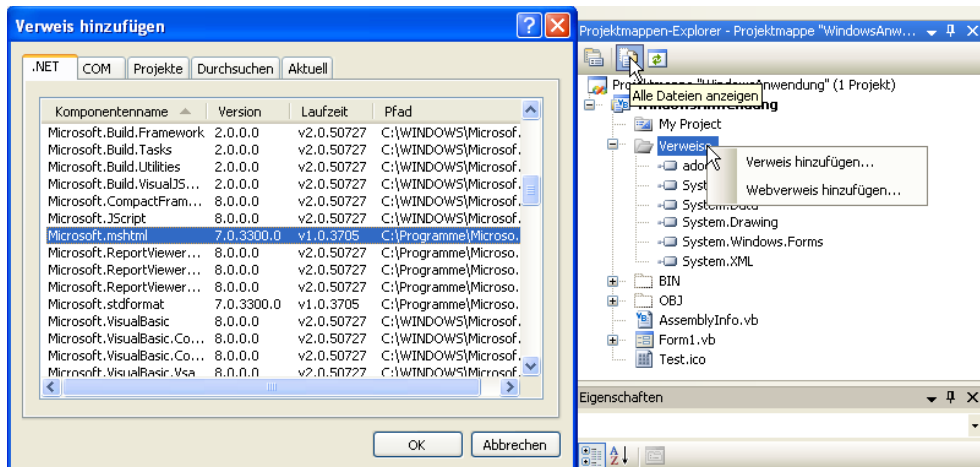


Abbildung 2.23: Verweise hinzufügen

Sobald Sie das Dialogfeld über die OK-Schaltfläche schließen, wird der gewählte Verweis im Projekt übernommen und auch in den Konfigurationsdateien der Projektmappe gesichert. Dies gewährleistet, dass die betreffenden Bibliotheken beim Erstellen des Projekts mit in die ausführbare Anwendung kopiert werden.

Zum Entfernen nicht benötigter Verweise blenden Sie die Anzeige aller Dateien über die betreffende Schaltfläche des Projektmappen-Explorers ein. Danach wählen Sie den betreffenden Verweis im Projektmappen-Explorer mit der rechten Maustaste an und verwenden den Kontextmenübefehl *Entfernen*.

Hinweis

Die Verweise können Sie auch über die Seite *Verweise* der Projekteigenschaften definieren (siehe weiter oben im Abschnitt »Die Einstellungen des Projekts anpassen«).

2.2.12 Das Fenster des Formulardesigners

Sobald Sie ein Formular im Projektmappen-Explorer per Doppelklick anwählen, wird dessen Inhalt im Fenster des Formulardesigners angezeigt (Abbildung 2.24, Mitte). In diesem Fenster lassen sich Formulare interaktiv entwerfen. In der gleichzeitig eingeblendeten Leiste *Toolbox* (Abbildung 2.24, links) wählen Sie die Kategorie *Alle Windows Forms* oder *Allgemeine Steuerelemente*. Diese enthält die Schaltflächen zum interaktiven Einfügen der Steuerelemente innerhalb des Formulars.

Um ein Steuerelement im Formular einzufügen, klicken Sie auf die betreffende Schaltfläche, markieren per Mausklick im Formular die linke obere Position und ziehen dann die Maus zur diagonal entgegengesetzten Ecke. Sobald Sie die Maustaste nach dem Ziehen loslassen, fügt der Editor das Steuerelement im Formular ein. Klicken Sie in der Leiste *Toolbox* auf die Schaltfläche eines Steuerelements und dann auf eine Stelle des Formularlayouts, wird die Schaltfläche in der Standardgröße an der betreffenden Position ange-

fügt. Doppelklicken Sie dagegen auf eine Schaltfläche der Symbolleiste, fügt die Entwicklungsumgebung das Steuerelement in der linken oberen Ecke des Formulars in seiner Standardgröße ein.

Sie können die Elemente des Formulars per Mausklick markieren, um diese anschließend zu verschieben oder deren Eigenschaften im Eigenschaftfenster anzupassen. Die Beschriftung von Schaltflächen, der Formulartitel oder die Texte von Beschriftungsfeldern werden beispielsweise über die Eigenschaft *Text* verwaltet.

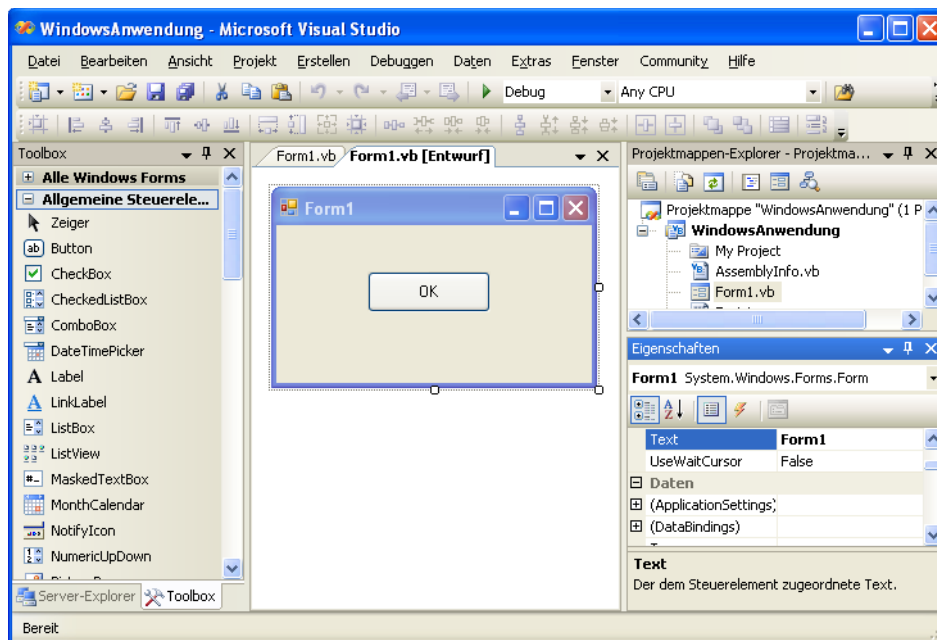


Abbildung 2.24: Formulardesigner mit Toolbox und Eigenschaften

Ereignisprozeduren für die Steuerelemente fügen Sie im Codefenster ein. Die Codeansicht lässt sich durch Doppelklicken auf das Design-Fenster, über den Befehl *Code* im Menü *Ansicht* oder über die Schaltfläche *Code anzeigen* des Projektmappen-Explorers öffnen.

2.2.13 Nutzen des Klassen-Designers

Der Klassen-Designer ist nur in Visual Studio (nicht in Visual Basic 2005 Express Edition) verfügbar und lässt sich über die Schaltfläche *Klassendiagramm anzeigen* des Projektmappen-Explorers einblenden. Der Designer analysiert dann den Quellcode des Projekts und zeigt ein Klassendiagramm mit den im Projekt gefundenen Klassen und deren Abhängigkeiten (Abbildung 2.25).

Klicken Sie auf die in der rechten oberen Ecke einer Klasse eingeblendete Schaltfläche mit den zwei nach unten zeigenden Pfeilsymbolen, wird die Darstellung der Klasse erweitert. Im erweiterten Bereich werden dann Felder, Eigenschaften und Methoden der Klasse aufgelistet. Wählen Sie das Abbild der Klasse mit der rechten Maustaste an, lässt

sich der Kontextmenübefehl *Klassendetails* abrufen. Dann zeigt Visual Studio in einem separaten Fenster alle Member der betreffenden Klasse an. Über die zugehörigen Kontrollkästchen der Spalte *Ausblenden* lassen sich nicht interessierende Details in der Darstellung unterdrücken.

Markieren Sie einen Member (Eigenschaft, Methode) in einer Klasse oder in der Liste mit den Klassendetails, blendet die Entwicklungsumgebung die zugehörigen Eigenschaften im Eigenschaftfenster ein. Sie können dann z.B. den Namen des Members ändern.

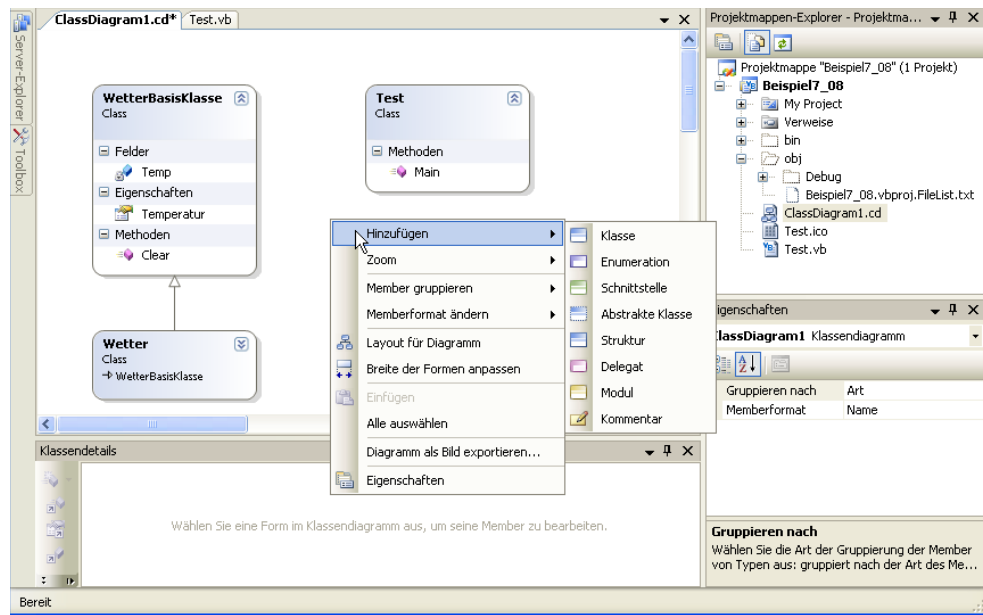


Abbildung 2.25: Klassen-Designer mit Klassendiagramm

Der Klassen-Designer erlaubt Ihnen zudem über die Entwurfs Oberfläche Klassen und andere Typen zum Projekt hinzuzufügen oder zu entfernen. Klicken Sie das Abbild einer Klasse in der Entwurfs Oberfläche mit der rechten Maustaste an, finden Sie im Kontextmenü den Befehl *Hinzufügen*. Dessen Untermenü enthält Befehle, um Methoden, Eigenschaften und weitere Member zur Klasse hinzuzufügen. Klicken Sie dagegen mit der rechten Maustaste auf eine leere Stelle der Entwurfs Oberfläche, enthält der Kontextmenübefehl *Hinzufügen* Einträge, um eine neue Klasse, ein neues Modul, eine neue Enumeration etc. zum Projekt hinzuzufügen.

Hinweis

Details zum Umgang mit dem Klassen-Designer finden Sie in der Visual Studio-Hilfe, wenn Sie nach dem Begriff »Klassen-Designer« oder »Klassendiagramm« suchen. Was sich hinter Klassen verbirgt und wie Sie ggf. Klassen per Code erzeugen und vererben, wird in *Kapitel 7* besprochen.

2.2.14 Projekt übersetzen und die Anwendung ausführen

Sobald Sie den Code für die Anwendung eingegeben und die Formulare entworfen haben, können Sie das Projekt übersetzen und die Anwendung starten. Sie können im Menü *Erstellen* einen der Befehle *<Projektname> erstellen* oder *<Projektname> neu erstellen* anwählen, wobei *<Projektname>* für den Namen des betreffenden Projekts (z.B. »WindowsAnwendung«) steht. Visual Studio 2005 bzw. Visual Basic 2005 Express Edition startet dann den Visual-Basic-.NET-Compiler und sorgt dafür, dass das gesamte Projekt übersetzt wird. Zudem stellt die Entwicklungsumgebung sicher, dass die als Verweise angegebenen Bibliotheken mit in das Anwendungsverzeichnis kopiert (oder im Global Assembly Cache gehalten) werden.

Die Meldungen des Übersetzers werden dabei als einfache Textanweisungen im Fenster *Ausgabe* angezeigt (Abbildung 2.26, oben). Sofern Fehler bei der Übersetzung des Projekts auftreten, generiert die Entwicklungsumgebung eine Fehlerliste (Abbildung 2.26, unten) in einem separaten Fenster. Sie können dabei über die Registerreiter am unteren Fensterrand zwischen den Fenstern *Fehlerliste* und *Ausgabe* umschalten. Doppelklicken Sie in der Fehlerliste auf einen Eintrag, markiert die Entwicklungsumgebung die fehlerhafte Stelle im Codefenster.

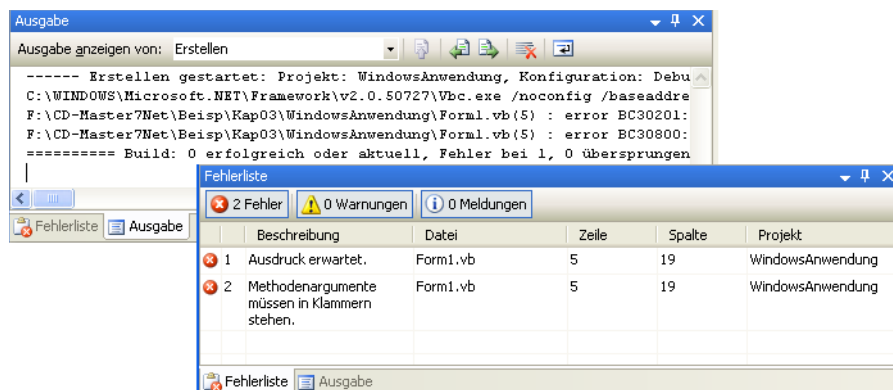


Abbildung 2.26: Meldungen beim Erstellen einer Anwendung

Um eine erfolgreich übersetzte Anwendung auszuführen, wählen Sie im Menü *Debuggen* den Befehl *Starten ohne Debuggen* oder *Debuggen starten*. Alternativ können Sie auch die Tastenkombination **Strg** + **F5** drücken.

2.3 Debuggen von .NET-Anwendungen

Ein wichtiger Aspekt bei der Entwicklung von Anwendungen stellt das Debuggen dar. Hierbei gilt es, die Programmfunktionen auf die Übereinstimmung mit den Anforderungen zu überprüfen. Visual Studio 2005 bzw. Visual Basic 2005 Express Edition erlauben dabei eine sehr komfortable Vorgehensweise, bei der der Quellcode schrittweise im Debugger auf (logische) Fehler untersucht werden kann.

2.3.1 Debuggen in der Entwicklungsumgebung

Bei der Entwicklung von Visual-Basic-Programmen in Visual Studio 2005 oder in Visual Basic 2005 Express Edition wird es selten ohne Fehler abgehen. Sie müssen den Programmcode also testen. Hierzu stellt die Entwicklungsumgebung direkt Debug-Funktionen zur Verfügung. Die Entwicklungsumgebung bietet dabei verschiedene Möglichkeiten, um das Programm unter Kontrolle des Debuggers auszuführen.

- Sie können die Programmausführung im Debugger direkt in der Entwicklungsumgebung über die Schaltfläche *Debuggen starten* der *Standard-Symbolleiste* einleiten (Abbildung 2.27, Hintergrund).
- Die gleiche Wirkung erzielt der Befehl *Debuggen starten* im Menü *Debuggen* sowie die Funktionstaste **[F5]**.
- Das Menü *Debuggen* enthält weitere Befehle, um den Code schrittweise oder prozedurweise durchlaufen zu lassen.

Stoßen Sie die Programmausführung über die obigen Befehle an, wird beim Übersetzen des Projekts eine PDB-Datei mit Debug-Infos generiert. Sobald der Debugger aktiv ist, blenden Visual Studio 2005 bzw. Visual Basic 2005 Express Edition die *Debuggen-Symbolleiste* im Fenster ein (Abbildung 2.27, Vordergrund). Anschließend lässt sich das Programm über die Debug-Funktionen schrittweise ausführen.

Hinweis

Wird eine .NET-Anwendung direkt unter Windows gestartet und tritt ein Laufzeitfehler auf oder wird ein Haltepunkt bzw. ein *Stop*-Befehl erreicht? Sofern Visual Studio 2005 oder ein anderer Debugger auf dem System installiert ist, kann der Debugger gestartet und das Programm unter dessen Kontrolle ausgeführt werden (siehe folgende Seiten).

Programmablaufsteuerung im Debugger

Die Kontrolle des Programmablaufs durch den Debugger kann weitgehend über die Befehle der Menüleiste, über die Schaltflächen der *Debuggen-Symbolleiste* oder über Tastenkürzel erfolgen. Hier eine kurze Übersicht über die Wirkungsweise der einzelnen Funktionen zur Ablaufsteuerung.

- Der Befehl *Debuggen starten* im Menü *Debuggen* oder die gleichnamige Schaltfläche der Symbolleiste *Debuggen* und die Taste **[F5]** veranlassen den Debugger, den Code komplett auszuführen.
- Der Befehl *Weiter* des Menüs *Debuggen* bzw. die gleichnamige Schaltfläche der *Debuggen-Symbolleiste* sowie die Taste **[F5]** bewirken, dass der Debugger das Programm nach einer Unterbrechung fortsetzt.
- Mit der Schaltfläche *Neu starten* der *Debuggen-Symbolleiste* bzw. mit dem gleichnamigen Befehl des Menüs *Debuggen* oder über die Tastenkombination **[Strg] + [↻] + [F5]** startet der Debugger den Programmablauf erneut.

- Über die Schaltfläche *Debuggen beenden* der *Debuggen*-Symbolleiste bzw. über den gleichnamigen Befehl des Menüs *Debuggen* oder über die Tastenkombination $\boxed{\text{⇧}} + \boxed{\text{F5}}$ lässt sich der Debug-Modus beenden und zur Entwicklungsumgebung zurückkehren.

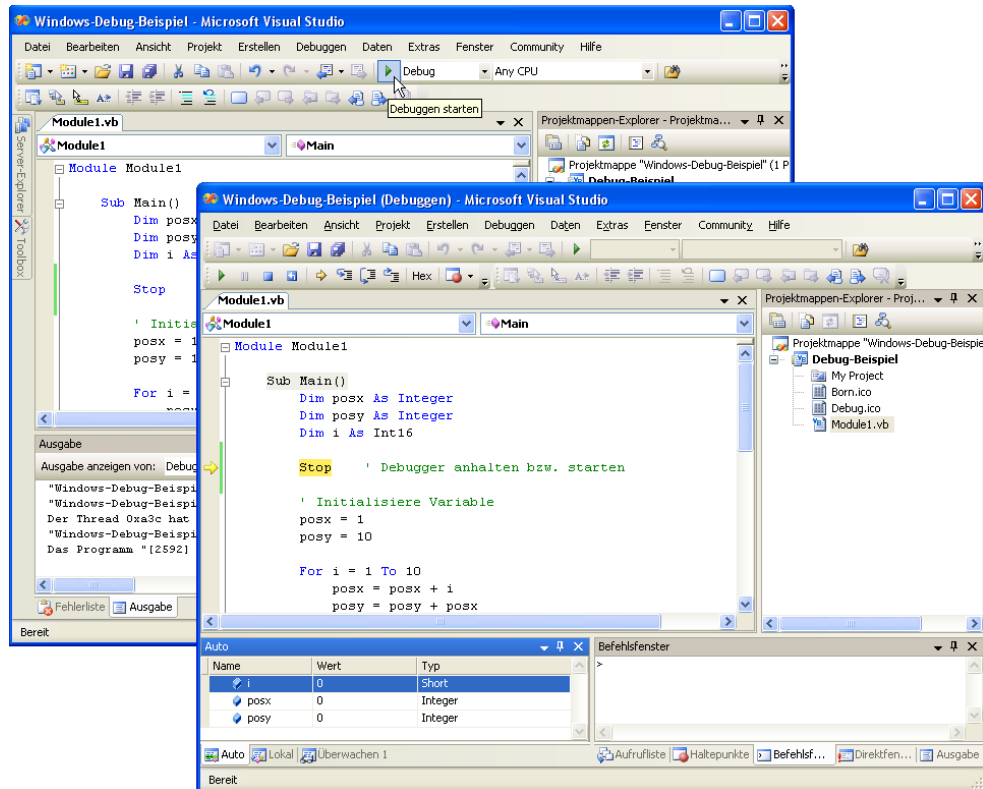


Abbildung 2.27: Fenster der Entwicklungsumgebung und Debug-Modus (Vordergrund)

Die obigen Debug-Funktionen bewirken, dass der Programmcode komplett abgearbeitet wird. Der Debugger hält die Programmausführung nur dann an, falls ein Laufzeitfehler auftritt oder sobald ein im Debugger gesetzter Haltepunkt (bzw. eine *Stop*-Anweisung im Code) erreicht wird. Um den Programmcode vom Programmbeginn an oder nach einer Unterbrechung (z.B. an einem Haltepunkt bzw. einer *Stop*-Anweisung) schrittweise fortzusetzen, können Sie die folgenden Funktionen des Debuggers verwenden.

- Der Befehl *Einzelschritt* des Menüs *Debuggen* oder die gleichnamige Schaltfläche in der Symbolleiste *Debuggen* bzw. die Funktionstaste $\boxed{\text{F11}}$ bewirkt, dass der Debugger nur die nächste Anweisung ausgeführt. Anschließend hält der Debugger das Programm an und Sie können die Inhalte von Variablen oder die Folgeanweisungen im Debugger-Fenster inspizieren.
- Verwenden Sie bereits ausgetestete Prozeduren, ist es im Einzelschrittmodus recht umständlich, wenn jede Anweisung der Prozedur im Debugger angezeigt wird. Mit dem Befehl *Prozedurschritt* des Menüs *Debuggen* bzw. mit der gleichnamigen Schalt-

fläche der Symbolleiste *Debuggen* oder durch Drücken der Funktionstaste **F10** wird die jeweils nächste Anweisung wie beim *Einzelschritt* ausgeführt. Handelt es sich aber um einen Prozeduraufruf, wird die aufgerufene Prozedur komplett durchlaufen. Der Debugger stoppt den Ablauf erst, sobald die Prozedur abgearbeitet ist und die nächste Anweisung im rufenden Programm zur Abarbeitung ansteht.

- Wurde der Ablauf des Programms in einer Prozedur (z.B. durch eine *Stop*-Anweisung oder einen Haltepunkt) unterbrochen, wählen Sie den Befehl *Ausführen bis Rücksprung* im Menü *Debuggen*. Alternativ können Sie die gleichnamige Schaltfläche der Symbolleiste *Debuggen* verwenden oder die Funktionstasten **⇧**+**F11** drücken. Dann wird die bereits teilweise abgearbeitete Prozedur komplett durchlaufen. Der Debugger stoppt, sobald die Prozedur beendet und die nächste Anweisung im rufenden Programm erreicht wird.

Die jeweils nächste im Debugger auszuführende Anweisung wird dabei im Codefenster gelb hinterlegt (Abbildung 2.27, Vordergrund). Gleichzeitig markiert ein gelber Pfeil in der Kennzeichenleiste (der graue linke Rand des Codefensters) die betreffende Zeile. Sie können diesen Pfeil per Maus vertikal nach oben oder unten ziehen, um eine andere Anweisungszeile als nächste ausführbare Anweisung zu markieren. Bei der Fortsetzung des Programms wird diese Anweisung dann ausgeführt. Die gleiche Wirkung erreichen Sie, indem Sie die gewünschte Anweisungszeile mit der rechten Maustaste anklicken und den Kontextmenübefehl *Nächste Anweisung festlegen* wählen. Sie sollten aber beachten, dass durch dieses Überspringen von Befehlen unerwartete Ergebnisse auftreten können (z.B. Variable weisen ggf. falsche Werte auf).

Tipp

Sie können eine Anweisungszeile mit der rechten Maustaste anklicken und dann im Kontextmenü den Befehl *Ausführen bis Cursor* wählen. Dann setzt der Debugger die Programmausführung solange fort, bis die betreffende Zeile erreicht wird. Die Funktion der einzelnen Schaltflächen der *Debuggen*-Symbolleiste lässt sich herausfinden, indem Sie per Maus auf eine der Schaltflächen zeigen. Dann wird die zugehörige Funktion als QuickInfo eingeblendet.

2.3.2 Arbeiten mit Haltepunkten

Beim Debuggen umfangreicher Programme ist es zu aufwändig, das Programm von Anfang an schrittweise durchlaufen zu lassen. Besser ist es, das Programm zu starten und dann den Ablauf gezielt an bestimmten Programmstellen zu unterbrechen. Dies gibt Ihnen die Möglichkeit, die Anweisungen in der Umgebung des Unterbrechungspunkts im Debugger zu inspizieren und schrittweise auszuführen.

Für diesen Zweck müssen Sie die betreffenden Anweisungszeilen mit so genannten Haltepunkten versehen. Hierzu klicken Sie im Codefenster des Debuggers auf die vor der Zeile angezeigte Kennzeichenleiste (die graue Leiste am linken Rand). Zudem finden Sie im Menü *Debuggen* Befehle zum Setzen oder Löschen von Haltepunkten in der aktuellen Zeile.

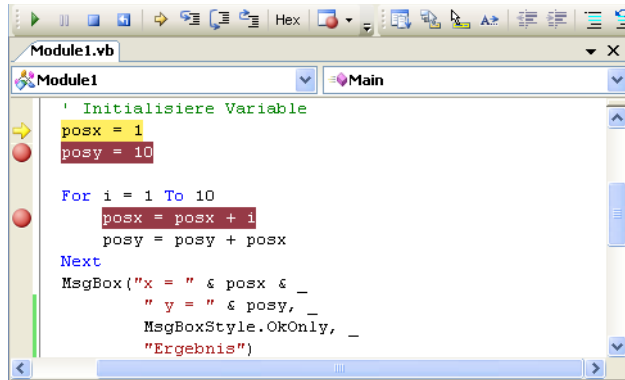


Abbildung 2.28: Haltepunkte im Codefenster

Eine mit einem Haltepunkt versehene Zeile wird braun hinterlegt und am linken Rand erscheint ein roter Punkt (Abbildung 2.28). Sie können mehrere Haltepunkte pro Modul bzw. Klasse setzen. Der Debugger unterbricht den Programmablauf, sobald ein Haltepunkt erreicht wird. Zum Löschen eines Haltepunkts klicken Sie einfach auf dessen roten Punkt in der Kennzeichenleiste. Alternativ können Sie die Befehle im Menü *Debuggen* oder im Kontextmenü verwenden, um einzelne Haltepunkte oder alle Haltepunkte zu löschen. Zudem gibt es einen Befehl, um Haltepunkte lediglich zu deaktivieren.

Hinweis

Beachten Sie beim Testen des Programmcodes, dass Haltepunkte in Ereignisprozeduren nur dann einen Abbruch bewirken, wenn das Ereignis auch auftritt (z. B. Anklicken einer Schaltfläche). Neben Haltepunkten können Sie auch *Stop*-Anweisungen im Code einfügen. Dies ist in dem in Abbildung 2.27 im Vordergrund sichtbaren Debug-Fenster der Fall. Sobald das Laufzeitsystem der .NET-Anwendung auf eine solche *Stop*-Anweisung trifft, übergibt es die Kontrolle an den Debugger, der dann den Programmcode in der Umgebung dieser Anweisung anzeigt und den Programmablauf anhält. Das Einfügen von *Stop*-Anweisungen hat aber den Nachteil, dass ein Eingriff in den Programmcode erforderlich wird. Vor der Auslieferung des Projekts müssen die *Stop*-Anweisungen wieder entfernt werden. Dies ist aufwändig und birgt die Gefahr, dass bei umfangreichen Projekten *Stop*-Anweisungen übersehen werden. Daher ist das Arbeiten mit Haltepunkten vorzuziehen. Diese werden lediglich vom Debugger verwaltet, wirken sich aber nicht auf den Quellcode des Projekts aus.

Bedingungen für Haltepunkte vereinbaren

Der Debugger ist in der Lage, Unterbrechungsbedingungen für Haltepunkte zu vereinbaren. Wird der Haltepunkt erreicht, prüft der Debugger, ob die Bedingung erfüllt ist.

1. Setzen Sie den gewünschten Haltepunkt (z.B. durch Anklicken der Kennzeichenleiste am linken Rand der Anweisungszeile).
2. Klicken Sie mit der rechten Maustaste auf die Zeile mit dem Haltepunkt und wählen Sie im Kontextmenü die Befehle *Haltepunkt/Bedingung* (Abbildung 2.29).

3. Anschließend tragen Sie im Dialogfeld *Bedingung für Haltepunkt* die Bedingung ein (Abbildung 2.29, Vordergrund), setzen ggf. die Optionen und schließen das Dialogfeld über die OK-Schaltfläche.

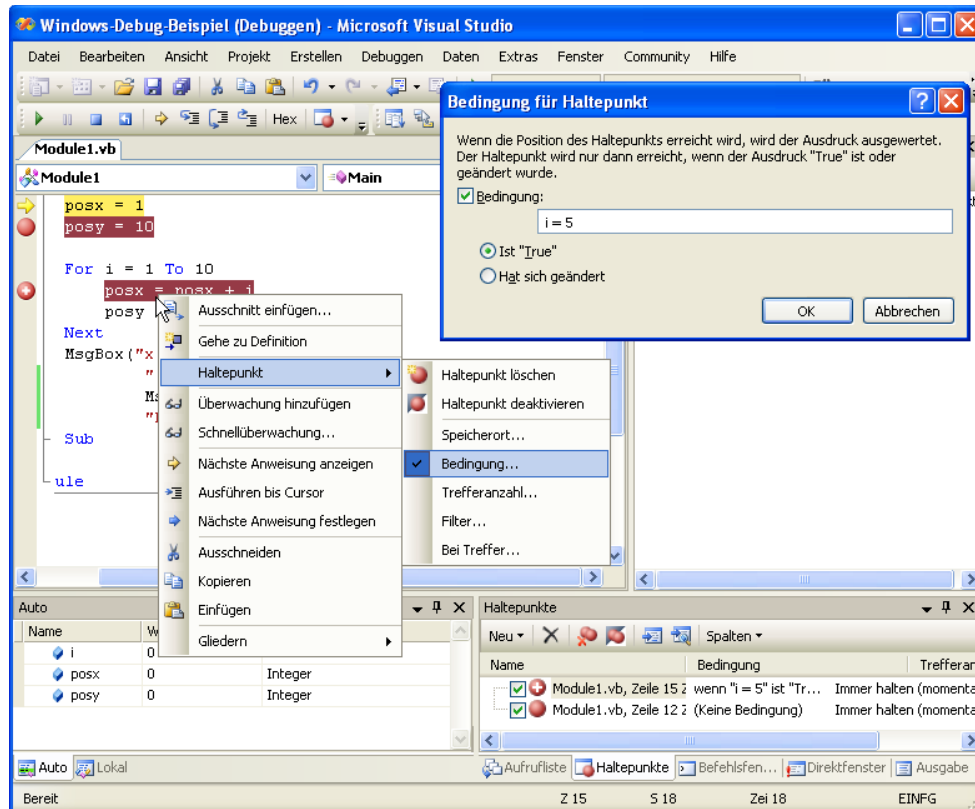


Abbildung 2.29: Bedingungen für einen Haltepunkt setzen

Sie können Bedingungen der Art »posx >= 5«, »i < 4«, »i < 9«, »i = 5« etc. im Textfeld des Dialogfelds eintragen. Im Ausdruck lassen sich Variablennamen, die in der Anweisung vorkommen oder deren Wert beim Ausführen der Anweisung bereits definiert ist, angeben. Ist das Optionsfeld *Ist "True"* markiert, unterbricht der Debugger den Programmablauf, wenn die Bedingung den Wert »true« liefert. Im aktuellen Beispiel unterbricht der Haltepunkt den Programmablauf, wenn die Variable *i* des Schleifenindex den Wert 5 enthält. Haben Sie dagegen das Optionsfeld *Hat sich geändert* markiert, wird der Programmablauf unterbrochen, sobald sich der angegebene Wert ändert. Haltepunkte mit Bedingungen werden übrigens durch ein Pluszeichen im roten Punkt des Haltepunkt-symbols ausgezeichnet.

Standardmäßig unterbricht der Debugger den Programmablauf am Haltepunkt spätestens dann, wenn die ggf. angegebene Bedingung zutrifft. Sie können den Debugger anweisen, den Programmablauf erst dann zu unterbrechen, wenn der Haltepunkt eine bestimmte Trefferzahl erreicht hat.

Tipp

Während der Programmausführung wird im unteren Teil des Anwendungsfensters das Teilfenster *Haltepunkte* eingeblendet (Abbildung 2.29). Falls das Fenster verdeckt ist, müssen Sie notfalls den betreffenden Registerreiter anklicken. Dort werden alle definierten Haltepunkte samt ihren Zeilennummern im Quellcode sowie eventuell definierten Bedingungen angezeigt. Die Statusleiste des Debuggers zeigt übrigens die Zeilennummer der aktuellen Anweisung an. Möchten Sie Zeilennummern im Codefenster sehen? Wählen Sie im Menü *Extras* den Befehl *Optionen* und suchen Sie den Zweig *Text-Editor/Alle Sprachen*. Anschließend ist im Dialogfeld *Optionen* das Kontrollkästchen *Zeilennummern* zu markieren.

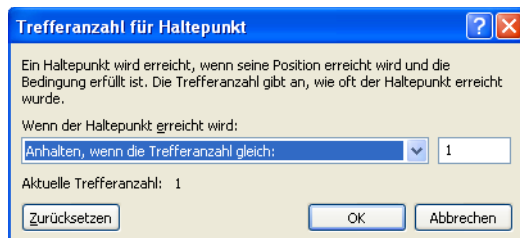


Abbildung 2.30: Trefferanzahl für Haltepunkt setzen

1. Klicken Sie die Zeile mit dem Haltepunkt mit der rechten Maustaste an und wählen Sie im Kontextmenü die Befehle *Haltepunkt/Trefferanzahl* (Abbildung 2.29).
2. Anschließend wählen Sie im Dialogfeld *Trefferanzahl für Haltepunkt* den Wert des Listenfelds *Wenn der Haltepunkt erreicht wird* (Abbildung 2.30), setzen ggf. die Trefferzahl und schließen das Dialogfeld über die *OK*-Schaltfläche.

Die Trefferanzahl für einen Haltepunkt lässt sich übrigens über die *Zurücksetzen*-Schaltfläche des Dialogfelds auf 0 zurückstellen. Anschließend können Sie die Ausführung des Programms im Debugger mit der Funktion *Weiter* starten. Im Fenster *Haltepunkte* zeigt der Debugger Ihnen an, wie oft die Bedingung bereits eingetreten ist. Wird das Abbruchkriterium erreicht, stoppt der Debugger den Programmablauf und Sie können den Inhalt von Variablen ansehen oder die Ausführung in anderen Modi fortsetzen.

Hinweis

Über den Kontextmenübefehl *Haltepunkt* können Sie weitere Unterbefehle wählen (Abbildung 2.29). Der Befehl *Speicherort* öffnet z.B. einen Dialog, in dem der Pfad zur Quelldatei sowie die Spalten- und Zeilennummer der Anweisungszeile aufgeführt wird. Der Befehl *Filter* erlaubt Ihnen die Gültigkeit des Haltepunkts auf bestimmte Prozesse, Maschinen oder Threads zu beschränken, während der Befehl *Bei Treffer* die Möglichkeit zur Ausgabe einer Meldung eröffnet. Details zu diesen Optionen finden Sie in der Programmhilfe.

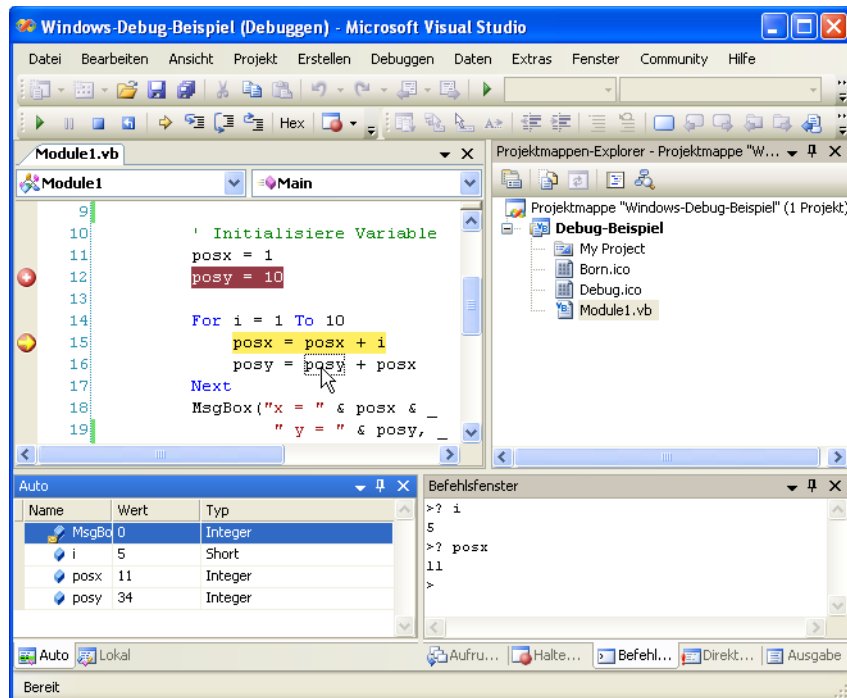


Abbildung 2.31: Anzeige von Variablenwerten im Codefenster

2.3.3 Anzeige der Werte im laufenden Programm

Programme speichern intern Werte in sogenannten Variablen. Diese Werte werden in der Regel beim Ablauf der Anwendung verändert. Bei der Entwicklung ist es daher hilfreich, wenn Sie während des Ablaufs den Wert bestimmter Variablen kennen. Der Debugger bietet verschiedene Möglichkeiten, um die Werte einzelner Variablen (oder einer Eigenschaft) zu ermitteln. Stellen Sie zuerst sicher, dass der Programmablauf im Debugger angehalten wurde.

- Zeigen Sie im Codefenster per Maus auf den Namen einer Variablen oder Eigenschaft, wird deren aktueller Wert als QuickInfo eingeblendet. Dies funktioniert aber nur, falls die Variable bereits mit einem Wert initialisiert wurde.
- Weiterhin listet der Debugger die Namen, die Werte und die Typen der lokalen Variablen und Konstanten im unteren Bereich des Debugger-Fensters auf (Abbildung 2.31). Das Teilfenster *Auto* enthält sowohl Variable als auch Konstanten, während im Fenster *Lokal* nur die lokal im aktuell ausgeführten Modul definierten Variablen aufgeführt werden.

Weiterhin können Sie natürlich auch zum Befehlsfenster des Debuggers wechseln. Wenn Sie dann ein Fragezeichen, gefolgt von einem Leerzeichen und dem Variablennamen eintippen, wird der Wert beim Drücken der -Taste im Befehlsfenster ausgegeben.

Tipp

Fenster wie *Auto* etc. können Sie einblenden, indem Sie im Menü *Debuggen* auf den Befehl *Fenster* klicken und dann im Untermenü den Namen des Fensters wählen. In Visual Basic 2005 Express Edition stehen Ihnen aber nicht alle Fenster beim Debuggen zur Verfügung.

Werte für Variablen setzen und überwachen

Um den Wert einer Variablen gezielt zu setzen, wechseln Sie im Debugger zum Direktfenster. Das Fenster findet sich in der rechten unteren Ecke des Debuggers. Notfalls müssen Sie es durch Anklicken des zugehörigen Registerreiters in den Vordergrund holen. Fehlt der Registerreiter, können Sie das Fenster über die Tastenkombination  +  +  oder über die Befehle des Menüs *Debuggen/Fenster* einblenden. Anschließend können Sie die Anweisung zur Anzeige des Werts bzw. zur Wertzuweisung an die Variable eintippen (Abbildung 2.32).

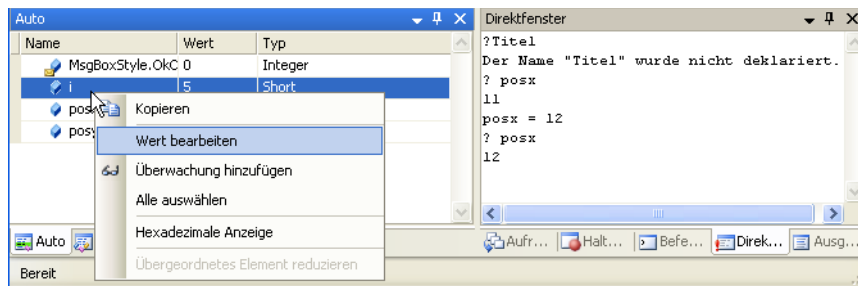


Abbildung 2.32: Direktfenster und Auto-Fenster

Zur Anzeige eines Werts im Befehlsfenster geben Sie ein Fragezeichen, gefolgt vom Namen der Variablen ein. Der Wert wird dann in einer zweiten Zeile angezeigt. Dies entspricht der Anzeige von Werten im Befehlsfenster. Möchten Sie den Wert einer Variablen ändern, tippen Sie den Variablennamen, gefolgt von einem Gleichheitszeichen und dem neuen Wert im Direktfenster ein. Die Eingaben sind jeweils durch Drücken der -Taste abzuschließen.

Alternativ können Sie im *Auto*-Fenster das Feld *Wert* einer Variablen mit der rechten Maustaste anklicken und dann im Kontextmenü den Befehl *Wert bearbeiten* wählen (Abbildung 2.32). Dann lässt sich der angezeigte Wert direkt per Tastatur ändern.

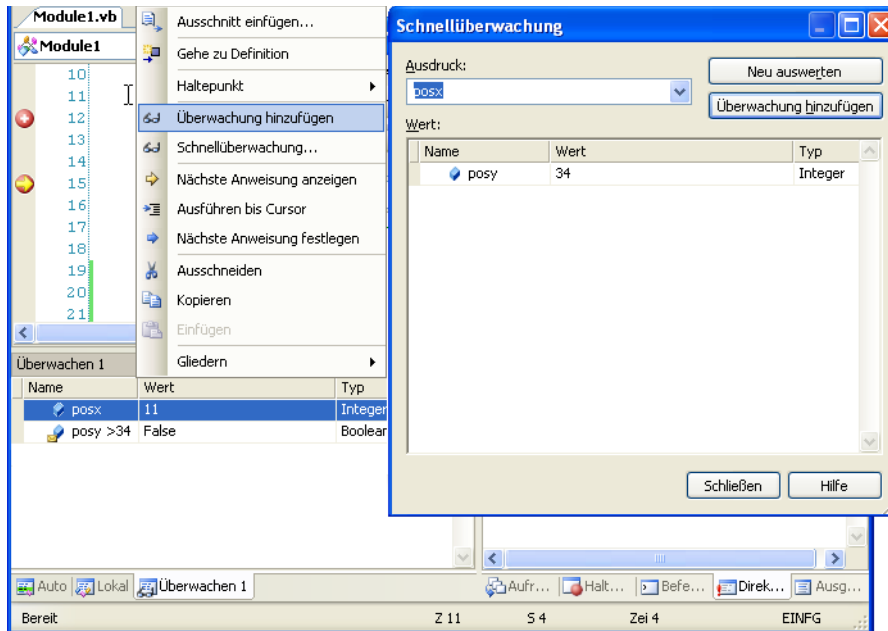


Abbildung 2.33: Schnellüberwachung definieren

Beim umfangreicheren Programmen ist es hilfreich, wenn Sie die Werte bestimmter Ausdrücke oder Variablen übersichtlich in einem Fenster zusammenstellen und dann überwachen können. Auch dies lässt sich mit dem Debugger sehr leicht bewerkstelligen.

- Um eine Variable zu überwachen, klicken Sie diese im Codefenster mit der rechten Maustaste an und wählen im Kontextmenü den Befehl *Überwachung hinzufügen* (Abbildung 2.33). Dann wird der Variablenname sofort in die Überwachungsliste aufgenommen.
- Möchten Sie einen zu überwachenden Ausdruck festlegen? Dann müssen Sie den Kontextmenübefehl *Schnellüberwachung* oder den gleichnamigen Befehl im Menü *Debuggen* wählen oder die Tastenkombination **[Strg] + [Alt] + [Q]** drücken. Dann öffnet die Entwicklungsumgebung das Dialogfeld *Schnellüberwachung* (Abbildung 2.33, Vordergrund). Sie können dann einen Ausdruck (z.B. »i > 9«) oder den Variablennamen im Feld *Ausdruck* eintragen (Abbildung 2.33). Klicken Sie auf die Schaltfläche *Überwachung hinzufügen*, wird der Ausdruck im Überwachungsfenster eingetragen. Die Schaltfläche *Neu berechnen* zeigt den aktuellen Wert des Ausdrucks im Dialogfeld an (Abbildung 2.33). Schließen Sie danach das Dialogfeld über die am unteren Rand eingeblendete Schaltfläche *Schließen*.

Der Debugger blendet ein Überwachungsfenster mit dem Variablennamen bzw. Ausdrücken, dem Wert und dem Typ ein (Abbildung 2.33, unten links). Beim Ablauf des Programms im Debugger zeigt dieser die Werte der überwachten Variablen bzw. Ausdrücke im Überwachungsfenster an. Geänderte Werte werden mit roter Schrift hervorgehoben.

Zum Ändern eines Ausdrucks wählen Sie diesen per Doppelklick im Überwachungsfenster an. Danach lässt sich der Ausdruck direkt korrigieren. Löschen können Sie einen Überwachungsausdruck, indem Sie diesen im Überwachungsfenster mit der rechten Maustaste anklicken und im Kontextmenü den Befehl *Überwachung löschen* wählen. Der Kontextmenübefehl *Alle löschen* entfernt dagegen alle Einträge aus dem Überwachungsfenster.

Tipp

Sie können markierte Ausdrücke vom Codefenster auch direkt in das Überwachungsfenster ziehen, um deren Werte aufzunehmen. Es muss sich aber um vollständige Ausdrücke handeln. Über die Befehle *Fenster/Überwachen* des Menüs *Debuggen* lassen sich bis zu vier Überwachungsfenster definieren. Verwenden Sie die Registerreiter am unteren Fensterrand, um zwischen den definierten Überwachungsfenstern zu wechseln.

2.3.4 Aufrufliste ansehen

Der Debugger führt intern eine Liste der aufgerufenen Methoden und Prozeduren, die Sie sich ansehen können (Abbildung 2.34). In dieser Liste spiegelt sich die Aufrufreihenfolge. Jeder Eintrag besteht aus dem Namen der Programmdatei, gefolgt vom Namen der Klasse und der aufgerufenen Prozedur bzw. Methode. In der Spalte *Sprache* wird zudem die Programmsprache angegeben, in der der Code zum Prozeduraufruf geschrieben wurde.

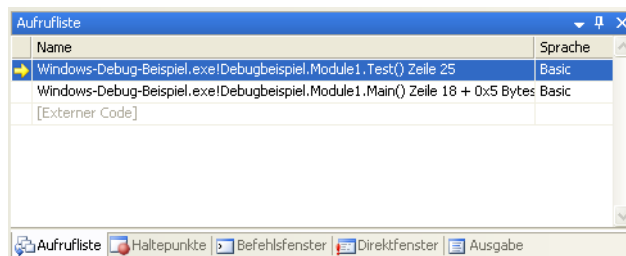


Abbildung 2.34: Fenster mit der Aufrufliste

Hinweis

Der Debugger enthält weitere Funktionen zum Öffnen der Quelldatei (Menü *Ansicht*), zum Verwalten der Prozesse (Menüs *Extras* bzw. *Debuggen*) und zur Verwaltung von Ausnahmen (Menü *Debuggen*). Dies gilt auch für Befehle zur Ausgabe von Meldungen mittels der Klasse *Debug* im Namensraum *System.Diagnostics*. Details zu diesen Fragen finden Sie in der Hilfe.

Sie finden auf der Begleit-CD dieses Buches im Ordner mit den Beispielen den Unterordner *Kap02\Debug*. Dort wurde ein Visual-Basic-Projekt mit etwas Code und einer *Stop*-Anweisung hinterlegt. Sie können die Projektdatei in Visual Studio 2005 oder in der Visual Basic 2005 Express Edition laden und zum Test des Debuggers verwenden.

Ist das Fenster *Aufrufliste* verdeckt, können Sie es über den gleichnamigen Registerreiter in den Vordergrund holen. Fehlt der Registerreiter (z.B. weil das Fenster vom Benutzer ausgeblendet wurde), können Sie es über die Befehle *Fenster/Aufrufliste* des Menüs *Debuggen* erneut einblenden.

2.3.5 Debug-Aufruf über das Laufzeitsystem

Sie können .NET-Anwendungen direkt unter Windows aufrufen. Sobald der Just-In-Time-Compiler auf eine im Visual-Basic-Code hinterlegte *Stop*-Anweisung trifft oder falls ein Laufzeitfehler auftritt, erscheint der in Abbildung 2.35 im Vordergrund gezeigte Dialog.



Abbildung 2.35: Dialogfelder zum Aufrufen und Auswählen des Debuggers

Über die Schaltfläche *Debuggen* lässt sich dann der Debugger aktivieren. Allerdings können mehrere Debugger auf dem betreffenden System installiert sein. Neben dem in Visual Studio 2005 integrierten Debugger gibt es noch den Microsoft CLR-Debugger. Dieser wird mit dem .NET Framework SDK bzw. mit Visual Studio 2005 installiert. Zur Auswahl des Debuggers erscheint nach Auswahl der Schaltfläche *Debuggen* das im Hintergrund von Abbildung 2.35 gezeigte Dialogfeld *Just-In-Time-Debugger von Visual Studio*. Sie können dann in der Liste *Mögliche Debugger* den Microsoft CLR-Debugger oder den Debugger von Visual Studio 2005 auswählen und durch Betätigen der *Ja*-Schaltfläche starten.

2.3.6 Debuggen mit dem Microsoft CLR-Debugger

Wird im Dialogfeld *Just-In-Time-Debugger von Visual Studio* der Eintrag »Microsoft CLR-Debugger« gewählt, erscheint das Anwendungsfenster aus Abbildung 2.36. Das Anwendungsfenster des Debuggers entspricht im Aufbau dem Debugger in der Entwicklungsumgebung. Neben dem Projektmappen-Explorer findet sich ein Fenster mit der Anzeige des

Quellcodes (sofern dieser vorliegt – ist nur die Anwendung vorhanden, wird der Intermediate Language Code angezeigt). Weiterhin sind im unteren Bereich verschiedene Fenster (*Lokal*, *Aufrufliste* etc.) zu sehen. Der Microsoft CLR-Debugger besitzt eine identische Benutzeroberfläche wie der Debugger der Entwicklungsumgebung. Sie können daher auf den vorhergehenden Seiten nachlesen, wie sich die einzelnen Funktionen nutzen lassen.

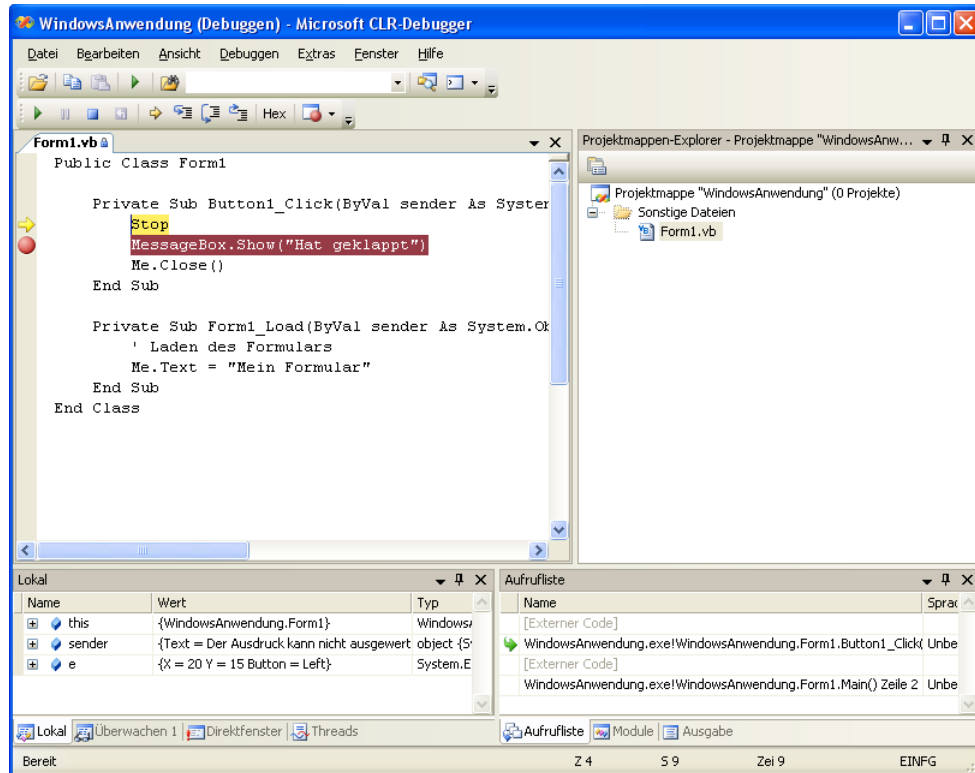


Abbildung 2.36: Fenster des Microsoft CLR-Debuggers mit Codeanzeige und Projekt-Explorer

Ist kein Visual Studio 2005 vorhanden, erscheinen beim Auftreten eines Laufzeitfehlers oder einer *Stop*-Anweisung u.U. die in Abbildung 2.35 gezeigten Dialogfelder nicht. Sie müssen dann den Microsoft CLR-Debugger ggf. manuell starten und die .NET-Anwendung unter der Kontrolle des Debuggers ausführen. Der Debugger findet sich in der Datei `DbgCLR.exe` im Installationspfad des .NET Framework SDK (lassen Sie ggf. nach der betreffenden Datei suchen).

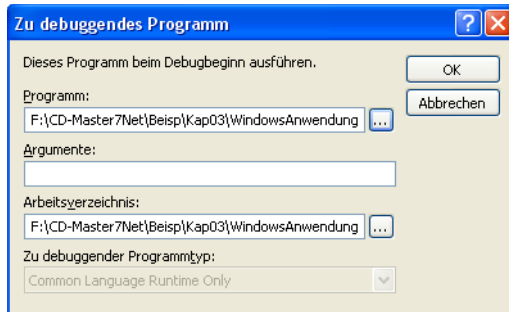


Abbildung 2.37: Auswahl des zu debuggenden Programms im CLR

1. Starten Sie den Debugger und warten Sie, bis das Anwendungsfenster des Debuggers erscheint.
2. Wählen Sie im Anwendungsfenster des Debuggers im Menü *Debuggen* den Befehl *Zu debuggendes Programm*. Der Debugger öffnet das Dialogfeld aus Abbildung 2.37.
3. Wählen Sie im Dialogfeld die zu debuggende *.exe*-Datei im Feld *Programm* und tragen Sie ggf. die beim Programmstart zu übergebenden Argumente im betreffenden Textfeld ein.
4. Schließen Sie das Dialogfeld über die *OK*-Schaltfläche. Mit diesem Schritt wird das Menü *Debuggen* des CLR-Anwendungsfensters um weitere Befehle ergänzt.

Sie können anschließend die *.NET*-Anwendung unter Kontrolle des CLR-Debuggers ablaufen lassen und Haltepunkte setzen, Variablenwerte inspizieren oder überwachen.

Hinweis

Das Debuggen im CLR-Debugger setzt aber voraus, dass die *.exe*-Datei mit der *.NET*-Anwendung auch Debug-Informationen enthält. Wenn Sie eine Anwendung in Visual Studio 2005 mit dem Befehl *Debuggen starten* des Menüs *Debuggen* übersetzen und ausführen lassen, werden diese Debug-Informationen mit abgelegt. Ergänzende Hinweise zum Debuggen und den Debug-Funktionen finden Sie im Zweig *Entwicklungstools und Sprachen | Dokumentation zu Visual Studio | Erstellen, Debuggen und Testen* der Visual Studio-Hilfe.

2.4 Veröffentlichen und Verteilen

Beim Erstellen eines Projekts erzeugt die Entwicklungsumgebung eine ausführbare *.exe*-Datei im eingestellten Zielordner der Anwendung (meist sind dies die Unterordner *\bin* oder *\bin\Debug*). Zum Verteilen der Anwendung an andere Benutzer oder auf andere Rechner lässt sich auf die Funktionen zur Bereitstellung der Entwicklungsumgebung zugreifen. Nachfolgend erhalten Sie eine kurze Übersicht über diese Thematik.

2.4.1 Was sollte man über die Bereitstellung wissen?

Visual Studio 2005 erlaubt die Bereitstellung über die ClickOnce-Technologie oder über den Windows-Installer.

- Die ClickOnce-Technologie erlaubt Ihnen die Bereitstellung von automatisch aktualisierten .NET-Anwendungen (Windows- und Konsolenanwendungen) von einem zentralen Speicherort (z.B. Webseite, Netzwerkfreigabe, CD/DVD etc.). Die Anwendung lässt sich von diesem Speicherort installieren, ggf. aktualisieren und ausführen.
- Alternativ lässt sich ein Setup-Paket unter Verwendung der Windows-Installer-Technologie erstellen. Bei dieser Art der Bereitstellung wird die Anwendung in eine *Setup.exe*-Datei verpackt, die sich (z.B. per CD/DVD) an die Benutzer verteilen lässt. Führt der Benutzer die Datei *Setup.exe* aus, lässt sich die Anwendung auf dem Zielsystem installieren.

Die Bereitstellung über ClickOnce ist die einfachere Variante für Endbenutzer, da der komplette Installationsvorgang nach dem Start der *Setup.exe* automatisch abläuft. Dabei wird die .NET-Anwendung dem Startmenü des Benutzers und der Gruppe *Software* in der Systemsteuerung hinzugefügt. Der Eintrag im Startmenü verweist dabei auf den zentralen Installationsort (in dem auch die ausführbare Anwendung liegt), es wird also nichts im Ordner *Programme* des Systems verändert. Allerdings lassen sich auch keine Verknüpfungen auf dem Desktop bereitstellen und der Uninstall-Eintrag wird in der Registrierung im Zweig *HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Uninstall* hinterlegt. Dies erlaubt ggf. auch die Installation einer Anwendung durch Benutzer, die keine Administratorenberechtigungen besitzen. Nicht mehr benötigte Anwendungen können über den Eintrag *Software* der Systemsteuerung deinstalliert werden. Zudem kann bei ClickOnce die automatische Aktualisierung von Komponenten vorgesehen werden. Dafür ist es aber erforderlich, dass die Anwendung Zugriff auf einen Webserver oder eine Netzwerkfreigabe erhält. Dies kann Probleme bei der Verteilung einer Anwendung über ClickOnce auf CD/DVD bringen.

Die Bereitstellung über ein Setup-Programm, welche auf die Windows-Installer-Technologie aufsetzt, ist aus Entwickler- und Benutzersicht etwas flexibler. Sie können in Visual Studio 2005 ein Setup-Projekt einrichten, in dem die Einstellungen für die zu erstellende Installationsdatei, die Installationsart (Datenträger, Web etc.) und der Installationsumfang vorgegeben wird. Das Setup-Programm startet bei der Ausführung einen Assistenten, der den Benutzer durch die Installationsschritte führt und ggf. das Anpassen einzelner Optionen erlaubt. Der Installer kann dabei die Anwendungsdateien in den Ordner *Programme* kopieren und beliebige Anpassungen am System (Startmenü, Desktop, Registrierung) vornehmen.

Hinweis

Eine Einführung in beide Verteilungstechnologien samt Gegenüberstellung der jeweiligen Features finden Sie in der Visual-Studio-Hilfe, wenn Sie nach dem Begriff »Übersicht über die ClickOnce-Bereitstellung« suchen lassen. In der Visual-Studio-Hilfe finden Sie zudem ausführliche Informationen über die Anforderungen und Features der beiden Installationsvarianten.

2.4.2 Bereitstellen eines Projekts mit ClickOnce

Die Bereitstellung eines Projekts über die ClickOnce-Technologie ist sowohl in Visual Studio 2005 als auch in der Visual Basic 2005 Express Edition verfügbar. Ein Assistent führt Sie durch die einzelnen Schritte und erlaubt die interaktive Auswahl der Bereitstellungsoptionen. Um ein fertig entwickeltes und übersetztes Projekt in der Entwicklungsumgebung mit ClickOnce bereitzustellen, gehen Sie in folgenden Schritten vor:

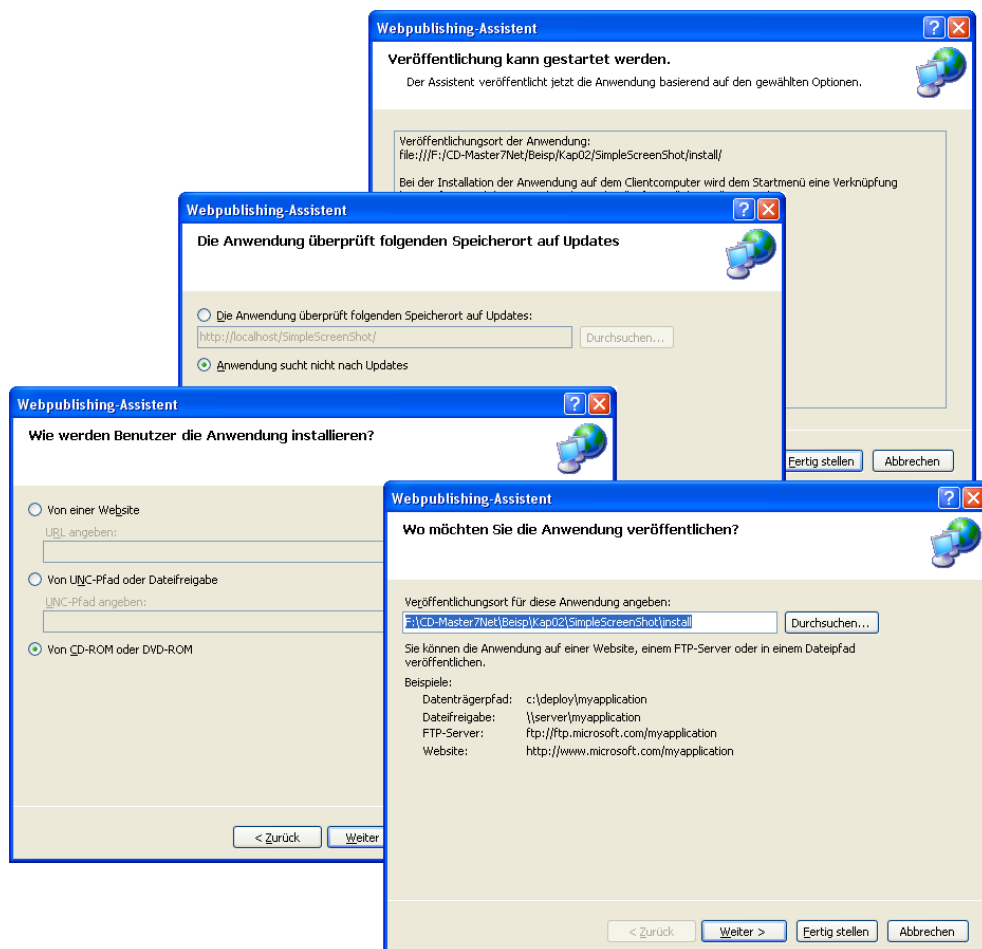


Abbildung 2.38: Webpublishing-Assistent zum Bereitstellen von Anwendungen

1. Laden Sie das Projekt in der Entwicklungsumgebung und wählen Sie im Menü *Erstellen* den Befehl *<Projekt> veröffentlichen*. Der Platzhalter *<Projekt>* steht dabei für den Projektnamen.
2. Sobald der Webpublishing-Assistent zum Bereitstellen von Anwendungen erscheint (Abbildung 2.38), wechseln Sie über die *Weiter*-Schaltfläche zwischen dessen Dialogen und legen die gewünschten Einstellungen fest.

Sobald Sie die *Fertig stellen*-Schaltfläche im Abschlussdialog anklicken, erzeugt die Entwicklungsumgebung die Setup-Dateien im angegebenen Zielverzeichnis.

- *Veröffentlichungsort*: Im ersten Dialogschritt (Abbildung 2.38, rechts unten) legen Sie über das Textfeld den Veröffentlichungsort fest. Dies kann die Adresse eines FTP-Verzeichnisses oder einer Website sein, Sie können aber auch Freigaben in einem Netzwerk über den UNC-Pfad vorgeben. Soll die Veröffentlichung in einem lokalen (bereits bestehenden) Ordner erfolgen, wählen Sie diesen über die Schaltfläche *Durchsuchen*.
- *Installationsart*: Im zweiten Dialogschritt (Abbildung 2.38, links unten im Hintergrund) lässt sich über Optionsfelder die Installationsart wählen. Sie können ein Optionsfeld markieren und im zugehörigen Textfeld die URL- oder UNC-Adresse der Webseite bzw. der Netzwerkfreigabe eingeben. Soll die Anwendung über CDs oder DVDs vertrieben werden, markieren Sie das Optionsfeld *Von CD-ROM oder DVD-ROM*.
- *Updateart*: Die ClickOnce-Technologie erlaubt .NET-Anwendungen eine automatische Aktualisierung über eine Website oder eine Netzwerkfreigabe. Ob Updates zulässig sind, wird im dritten Dialogschritt (Abbildung 2.38, links Mitte im Hintergrund) festgelegt. Soll die Anwendung updatefähig sein, markieren Sie die Option *Die Anwendung überprüft folgenden Speicherort auf Updates* und geben Sie dann die Adresse (URL, UNC-Pfad einer Freigabe etc.) des Speicherorts im zugehörigen Textfeld ein. Über die Schaltfläche *Durchsuchen* lässt sich der Speicherort ggf. interaktiv auswählen. Wird die Anwendung über CDs/DVDs verteilt, markieren Sie das Optionsfeld *Die Anwendung sucht nicht nach Updates*. Diese Option ist übrigens auch ganz hilfreich, wenn Sie keine Updates planen und dem Benutzer den lästigen Update-Dialog bei jedem Start der Anwendung ersparen möchten.

Im letzten Dialogfeld zeigt der Assistent die Einstellungen an. Sie können dann ggf. über die Schaltfläche *Zurück* zu den vorhergehenden Dialogen zurückblättern oder die Bereitstellung mittels der Schaltfläche *Fertig stellen* starten.

Anmerkungen zur Setup-Datei

Der Assistent legt im Zielordner eine Datei mit dem Namen *Setup.exe* sowie weitere Hilfsdateien und ein Unterverzeichnis mit den veröffentlichten Dateien an. Wird die *Setup*-Datei ausgeführt, überprüft diese die Installationsanforderung. Sind diese gegeben (z.B. .NET Framework 2.0 vorhanden), wird die Anwendung installiert. Dabei richtet der Installer den Startmenü- und den Uninstall-Eintrag unter dem aktuellen Benutzerkonto ein. Der Startmenüeintrag verweist dabei auf den Ort, an dem die Bereitstellungsdateien gehalten werden. Der Benutzer kann die Anwendung über deren Eintrag im Dialogfeld *Software* mittels des Moduls *Software* der Systemsteuerung deinstallieren.

Bei dem Startmenüeintrag handelt es sich aber um keine *.lnk*-Datei, wie sie für Windows-Verknüpfungen benutzt wird. Vielmehr finden sich dort Informationen, die vom .NET Framework ausgewertet und zum Start der Anwendung verwendet werden können. Der Uninstall-Eintrag wird im Zweig *HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Uninstall* hinterlegt. Der Befehl benutzt ebenfalls Komponenten des .NET Framework, um den Startmenüeintrag aus dem System zu entfernen.

Hinweis

Sie finden auf der Begleit-CD im Ordner `\Beisp\Kap02\SimpleScreenShot` eine Projektmappe mit einem kleinen Projekt. Der Unterordner *Install* enthält die mit ClickOnce bereitgestellten Dateien. Zudem gibt es den Unterordner `\bin\Publish`, in dem sich die zu veröffentlichen Dateien samt einer Kopie des *Install*-Ordners (Unterordner *SimpleScreenShot.publish*) finden.

2.4.3 Setup-Projekt mit Windows-Installer-Technologie

Visual Studio 2005 unterstützt die Bereitstellung von .NET-Anwendungen über Setup-Pakete, die den Windows-Installer verwenden. Um aus einer bereits übersetzten .NET-Anwendung ein entsprechendes Setup-Paket zu erstellen, sind folgende Schritte erforderlich:

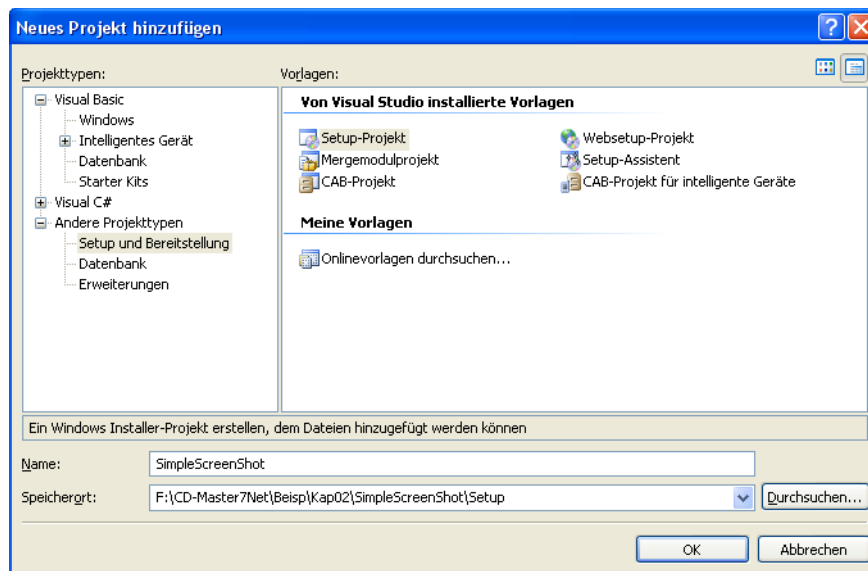


Abbildung 2.39: Auswahl der Setup-Projektvorlage

1. Nachdem Sie das Projekt in Visual Studio 2005 geladen haben, wählen Sie im Menü *Datei* den Befehl *Hinzufügen/Neues Projekt*. Alternativ können Sie im Kontextmenü der im Projektmappen-Explorers angezeigten Projektmappe diese Befehle wählen.
2. Sobald das Dialogfeld *Neues Projekt* erscheint (Abbildung 2.39), wählen Sie in der linken Rubrik *Projekttypen* den Eintrag *Andere Projekttypen/Setup und Bereitstellung*.
3. Anschließend klicken Sie im rechten Teil *Vorlagen* die gewünschte Projektvorlage (hier *Setup-Projekt*) an. Passen Sie ggf. den Namen der Vorlage im Feld *Name* an und wählen Sie bei Bedarf den Speicherort des Projekts.

Sobald Sie den Dialog über die OK-Schaltfläche schließen, fügt Visual Studio 2005 das Setup-Projekt zur Projektmappe hinzu und blendet das Fenster des Dateisystem-Editors ein (Abbildung 2.40, Hintergrund). Dieses Fenster lässt sich bei Bedarf schließen und später über den Befehl *Fenster/Dateisystem* des Menüs *Ansicht* wieder einblenden.

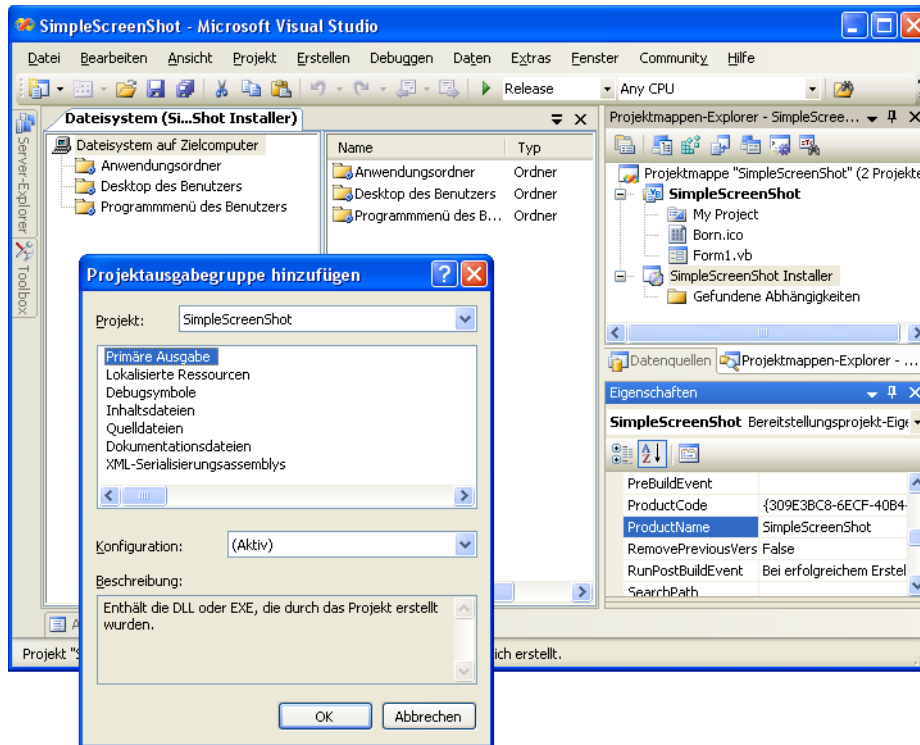


Abbildung 2.40: Setup-Projekt im Projektmappen-Explorer und Dateisystem-Editor

Sie sollten anschließend den Eintrag des Setup-Projekts im Projektmappen-Explorer anklicken und im Eigenschaftensfenster den Eintrag *ProductName* überprüfen bzw. anpassen. Setzen Sie den Wert der Eigenschaften *Author* und *Manufacturer* auf sinnvolle Begriffe, da diese Werte u.a. zur Benennung des Standard-Installationsordners benutzt werden. Die Eigenschaft *InstallAllUser* lässt sich auf die Werte *true* oder *false* setzen und gibt an, ob das Paket standardmäßig für den aktuellen Benutzer oder alle Benutzer eingerichtet werden soll. Diese Standardvorgaben lassen sich vom Benutzer aber im Installationsassistenten überschreiben. Bei Bedarf können Sie auch den Namen des Setup-Projekts im Projektmappen-Explorer wie bei anderen Projekten über den Kontextmenübefehl *Umbenennen* nachträglich ändern (z.B. indem Sie noch den Begriff »Installer« an den Projektnamen anhängen). Anschließend gilt es, das eigentliche Projekt (hier *SimpleScreenShot*) dem Installationspaket hinzuzufügen. Hierzu sind folgende Schritte auszuführen:

1. Markieren Sie im Fenster des Projektmappen-Explorers den Eintrag des Projekts (hier die Projektdatei *SimpleScreenShot*).
2. Wechseln Sie zum Fenster des Dateisystem-Editors und markieren Sie dort das Ordnersymbol des Knotens *Anwendungsordner*. Fehlt das Fenster, lässt es sich bei Bedarf über den Befehl *Editor* des Menüs *Ansicht* einblenden.
3. Wählen Sie im Menü *Aktion* der Entwicklungsumgebung (bzw. im Kontextmenü des Knotens) die Befehle *Hinzufügen/Projektausgabe*.
4. Sobald das Dialogfeld *Projektausgabegruppe hinzufügen* erscheint (Abbildung 2.40, Vordergrund), stellen Sie sicher, dass im Listenfeld *Projekt* der Eintrag des gewünschten Projekts (hier *SimpleScreenShot*) eingestellt ist.
5. Wählen Sie in der angezeigten Liste den Eintrag *Primäre Ausgabe* und stellen Sie sicher, dass im Feld *Konfiguration* die Option *(Aktiv)* eingestellt ist.

Wenn Sie jetzt das Dialogfeld über die *OK*-Schaltfläche schließen, werden die Dateien (.exe und ggf. .dll) des bereitzustellenden Projekts im Installationsprojekt aufgenommen. Sie könnten an dieser Stelle bereits den Befehl *<Projekt> erstellen* im Menü *Erstellen* wählen, um das Installationspaket generieren zu lassen. Allerdings erwarten die Benutzer von Windows-Anwendungen, dass diese nach der Installation sich zumindest über einen Eintrag im Startmenü oder über eine Desktop-Verknüpfung aufrufen lassen. Oft werden auch zusätzliche Dateien (z.B. Hilfe, Readme, Zusatztools etc.) mit eingerichtet. Diese Elemente müssen Sie explizit im Installer-Projekt hinzufügen.

Dateien und Ordner aufnehmen

Benötigen Sie einen Ordner im Installationsverzeichnis? Soll eine Verknüpfung in einer Programmgruppe des Startmenüs hinterlegt werden? Oder möchten Sie zusätzliche Dateien zum Setup-Projekt hinzufügen? Im aktuellen Beispiel soll eine spezielle Symboldatei für die Verknüpfungen auf dem Desktop und im Startmenü verfügbar sein. Zudem soll im Anwendungsordner ein Unterordner *Born* angelegt werden. Um die Symboldatei aufzunehmen, sind folgende Schritte notwendig:

1. Klicken Sie im Fenster des Dateisystem-Editors das Ordnersymbol des Knotens *Anwendungsordner* an.
2. Wählen Sie im Menü *Aktion* (bzw. im Kontextmenü des Knotens) die Befehle *Hinzufügen/Datei*.
3. Im dann eingeblendeten Dialogfeld *Dateien hinzufügen* können Sie in den Ordnern der Festplatte navigieren und die .ico-Datei auswählen.

Sobald Sie das Dialogfeld über die *Öffnen*-Schaltfläche schließen, wird die betreffende Datei im Zweig *Anwendungsordner* des Dateisystem-Editors eingefügt. Auf die gleiche Weise können Sie andere Dateien (z.B. Hilfedateien, Zusatztools, Dokumentdateien etc.) zum Anwendungsordner hinzufügen. Sollen Unterordner im Anwendungsordner oder in einem anderen Knoten angelegt werden? Dann klicken Sie den betreffenden Knoten mit der rechten Maustaste an und wählen die Befehle *Hinzufügen/Ordner*. Sie können die hinzugefügten Elemente (Ordner, Dateien, Verknüpfungen) über Kontextmenübefehle jederzeit umbenennen oder entfernen.

Hinweis

Leere Ordner in den Zweigen des Dateisystem-Editors werden nur dann in das Setup-Projekt mit aufgenommen, wenn deren Eigenschaft *AlwaysCreate* im Eigenschaftsfenster auf *True* gesetzt ist.

Verknüpfungen für Startmenü und Desktop definieren

Jetzt gilt es noch, die Verknüpfungen für das Startmenü sowie für den Desktop im Setup-Projekt einzurichten. Um einen Startmenüeintrag zu konfigurieren, führen Sie folgende Schritte aus.

1. Markieren Sie den Knoten *Programmmenü des Benutzers* des Dateisystem-Editors mit einem Mausklick.
2. Klicken Sie in der rechten Spalte *Name* des Dateisystem-Editors eine freie Stelle mit der rechten Maustaste an und wählen Sie im Kontextmenü den Befehl *Neue Verknüpfung erstellen*.
3. Im dann eingeblendeten Dialogfeld *Element im Projekt auswählen* wählen Sie den Zweig *Anwendungsordner* per Doppelklick an, markieren den Eintrag *Primäre Ausgabe von <Name> (Aktiv)* und schließen den Dialog über die OK-Schaltfläche.

Der Dateisystem-Editor richtet jetzt eine Verknüpfung auf das gewählte Programm im betreffenden Zweig ein. Die Verknüpfung wird bei der Installation der Anwendung im Startmenü des Benutzers eingerichtet. Sie können anschließend die Eigenschaften der Verknüpfung gemäß Ihren Bedürfnissen anpassen.

- Klicken Sie im Dateisystem-Editor zuerst auf den Namen der Verknüpfung, wählen danach den Kontextmenübefehl *Umbenennen* und tragen Sie dann den für den Startmenüeintrag gewünschten Text (z.B. »SimpleScreenShot«) ein.
- Soll eine eigene Programmgruppe für die Verknüpfung(en) im Startmenü angelegt werden? Dann fügen Sie über den Kontextmenübefehl *Hinzufügen/Ordner* des Knotens *Programmmenü des Benutzers* einen Ordner hinzu und benennen diesen gemäß dem Gruppennamen. Danach ziehen Sie die Verknüpfung per Maus in den neuen Unterordner des Knotens *Programmmenü des Benutzers*.
- Wählen Sie die Verknüpfung im Dateisystem-Editor an und wechseln Sie zum Eigenschaftsfenster. Passen Sie dort die Eigenschaften der Verknüpfung nach Ihren Anforderungen an. Um ein eigenes Symbol anzuzeigen, wählen Sie die Schaltfläche der Eigenschaft *Icon* und klicken im eingeblendeten Menü auf den Wert »Durchsuchen«. Wählen Sie im Dialogfeld *Symbol* die Schaltfläche *Durchsuchen*, lässt sich über einen Zusatzdialog zum Knoten *Anwendungsordner* des Zweigs *Dateisystem auf Zielcomputer* navigieren und die in diesem Ordner bereits eingefügte Symboldatei wählen. Über weitere Eigenschaften können Sie den Startmodus, den Installationsordner der Verknüpfung oder die zu übergebenden Argumente anpassen. Sobald Sie eine Eigenschaft anklicken, werden Zusatzinformationen über die Eigenschaft im Fußteil des Fensters angezeigt.

Auf die gleiche Art können Sie eine Desktop-Verknüpfung im Knoten *Desktop des Benutzers* einrichten. Zudem lassen sich weitere Verknüpfungen auf Dateien (z.B. Hilfedatei) einrichten. Sobald alle Einträge des Setup-Projekts in den Knoten des Dateisystem-Editors hinterlegt sind, wählen Sie im Menü *Erstellen* der Entwicklungsumgebung den Befehl *<Projekt> erstellen*. *<Projekt>* entspricht dabei dem Namen des Installationsprojekts (hier *SimpleScreenShot Installer*).

Die Entwicklungsumgebung erstellt bei Anwahl des Befehls *<Projekt> erstellen* das Setup-Paket mit der Datei *Setup.exe* und der *.msi*-Installationsdatei. Je nach Auswahl der Option *Konfiguration* werden die Dateien dabei in den Unterordnern *Debug* bzw. *Release* des Setup-Projekt Projektordners hinterlegt. Sie können anschließend den Inhalt des Ordners an Dritte weitergeben. Die Installation der Anwendung lässt sich über das Programm *Setup.exe* starten. Der Benutzer wird von einem Setup-Assistenten durch die Installationsschritte geführt. Dort lässt sich z.B. der vorgegebene Installationsordner anpassen oder auswählen, ob die Anwendung für alle Benutzer oder nur für den aktuellen Benutzer einzurichten ist.

Tipp

Möchten Sie das Installationspaket direkt auf dem Entwicklungsrechner testen? Dann wählen Sie im Menü *Projekt* den Befehl *Installieren* an. Die Entwicklungsumgebung ruft den Windows-Installer des Projekts auf. Über den Befehl *Deinstallieren* im Menü *Projekt* der Entwicklungsumgebung lässt sich die Anwendung auch wieder vom System entfernen – Sie sparen sich also den Umweg über die Gruppe *Software* der Systemsteuerung.

Hinweis

Werden die Einträge der Knoten *Desktop des Benutzers* bzw. *Startmenü des Benutzers* beim Erstellen des Projekts bemängelt? Dann markieren Sie im Fenster des Dateisystem-Editors das Ordnersymbol des Knotens *Programmmenü des Benutzers*, wechseln zum Eigenschaftenfenster und stellen den Wert der Eigenschaft *AlwaysCreate* auf *True*.

Sie finden übrigens ein Beispieldatensatz mit einem Setup-Projekt im Ordner *\Beisp\Kap02\SimpleScreenShot\Setup* der Begleit-CD. Die *.sln*-Datei befindet sich im übergeordneten Ordner *\Beisp\Kap02\SimpleScreenShot*.

Visual Studio 2005 stellt weitere Funktionen bereit, um optionale Setup-Einstellungen im Installationspaket festzulegen. Sie können beispielsweise Dateitypen registrieren, Registrierungseinstellungen anpassen und vieles mehr. Weiterführende Informationen zu solchen Themen finden Sie in der Hilfe von Visual Studio 2005 unter dem Thema »Windows Installer-Bereitstellung«. Lassen Sie ggf. über die Registerkarte *Suchen* nach dem Begriff suchen und klicken Sie dann auf die Schaltfläche *Mit Inhaltsverzeichnis synchronisieren*, um das betreffende Thema auf der Registerkarte *Inhalt* abzurufen.

2.5 Visual Studio 2005 anpassen

Visual Studio 2005 lässt sich über den Befehl *Optionen* im Menü *Extras* benutzerspezifisch anpassen. Die Entwicklungsumgebung öffnet das Dialogfeld *Optionen* (Abbildung 2.41), in dessen linker Spalte Sie die gewünschte Kategorie und ggf. Unterkategorie anwählen müssen. Existieren Unterkategorien, werden diese bei Anwahl einer Kategorie eingeblendet. Wählen Sie eine Unterkategorie, zeigt das Dialogfeld die verfügbaren Optionen im rechten Teil des Dialogfelds an.

Die Visual Basic 2005 Express Edition besitzt ebenfalls ein Dialogfeld *Optionen*, welches sich über die gleichen Befehle aufrufen lässt. Allerdings stehen nicht alle Optionen aus Visual Studio 2005 zur Verfügung und in den Kategorien sind die Optionen auch etwas unterschiedlich als bei Visual Studio 2005 zusammengefasst.

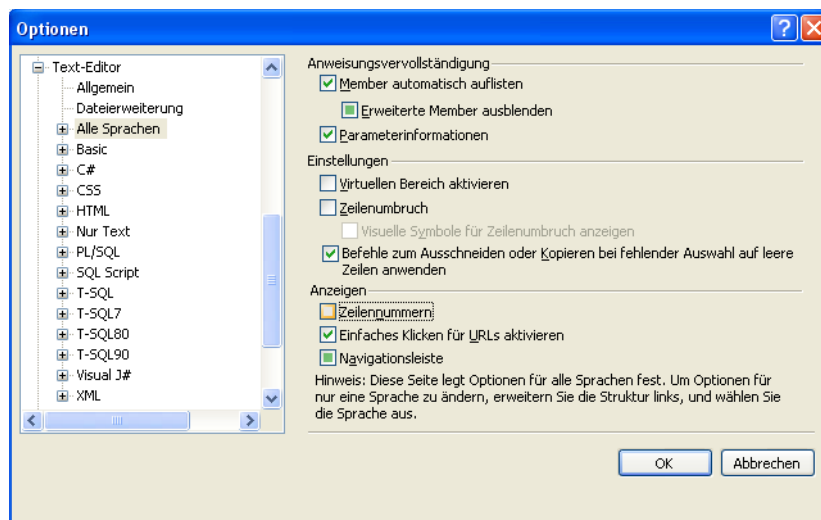


Abbildung 2.41: Optionen in Visual Studio einstellen

2.6 Die Online-Hilfe nutzen

Sowohl Visual Studio 2005 als auch Visual Basic 2005 Express Edition werden mit umfangreicher Dokumentation ausgeliefert. Sie können direkt über das Windows Startmenü auf die betreffenden Hilfedateien zugreifen. Alternativ besteht die Möglichkeit, das Fenster der Hilfe direkt über das *Hilfe*-Menü der Entwicklungsumgebung aufzurufen. Das Hilfenmenü bietet dabei verschiedene Befehle, um direkt über das Inhaltsverzeichnis, das Stichwortverzeichnis, das Suchformular etc. auf die Hilfeinformationen zuzugreifen.

Das Fenster der Hilfe wird in der Regel gemäß der Darstellung aus Abbildung 2.42 angezeigt. Über die Schaltflächen der Kopfleiste (oder die am unteren Rand der linken Spalte angezeigten Registerreiter) lassen sich im linken Teilfenster das Inhaltsverzeichnis, das Stichwortverzeichnis oder die Favoritenliste einblenden. Durch Anklicken der im linken Teilfenster eingeblendeten Themen können Sie den Inhalt der Hilfeseiten im rechten Teil

des Fensters abrufen. In der Indexseite lassen sich Begriffe eintippen, die dann in der Liste der Stichwörter nachgeschlagen werden. Durch Anklicken eines Stichworts wird das Hilfethema ebenfalls im rechten Teilfenster abgerufen. Die Handhabung gleicht im Wesentlichen der Windows-Hilfe. Neu ist lediglich, dass sich der Umfang der in der linken Spalte eingeblendeten Hilfethemen (.NET Framework, SQL Server, etc.) über das Listenfeld *Filter* (im Kopf der linken Spalte) einstellen lässt. Öffnen Sie das Listenfeld, können Sie über die angebotenen Werte den Hilfeumfang reduzieren oder ausweiten.

Das Suchformular wird dagegen im rechten Teilfenster eingeblendet und ist über den Registerreiter *Suchen* erreichbar. Sie können dann Suchbegriffe im Formular eingeben und durch Drücken der -Taste oder über die *Suchen*-Schaltfläche des Formulars die Volltextsuche starten. Bei der Visual Studio-Hilfe lässt sich bei Bedarf in den Hilfeseiten zudem ein Sprachfilter setzen. Klicken Sie auf die mit *Sprachfilter* bezeichnete Schaltfläche, öffnet sich ein Menü mit verschiedenen Optionen (Abbildung 2.42). Löschen Sie die Markierung der Kontrollkästchen aller Themen, die nicht in der Hilfe berücksichtigt werden sollen. Mit der Suche gefundene Hilfethemen werden im rechten Teilfenster eingeblendet. Wählen Sie den Hyperlink des betreffenden Themas an, gelangen Sie zur Ansicht der betreffenden Seite. Möchten Sie wissen, wo sich das betreffende Thema im Inhaltsverzeichnis befindet? Sobald die Trefferseite angezeigt wird, wählen Sie die Schaltfläche *Mit Inhaltsverzeichnis synchronisieren* in der Symbolleiste des Hilfefensters. Wenn Sie dann in der linken Spalte die Registerkarte *Inhalt* anwählen, ist die betreffende Stelle im Inhaltsverzeichnis bereits aufgeschlagen.

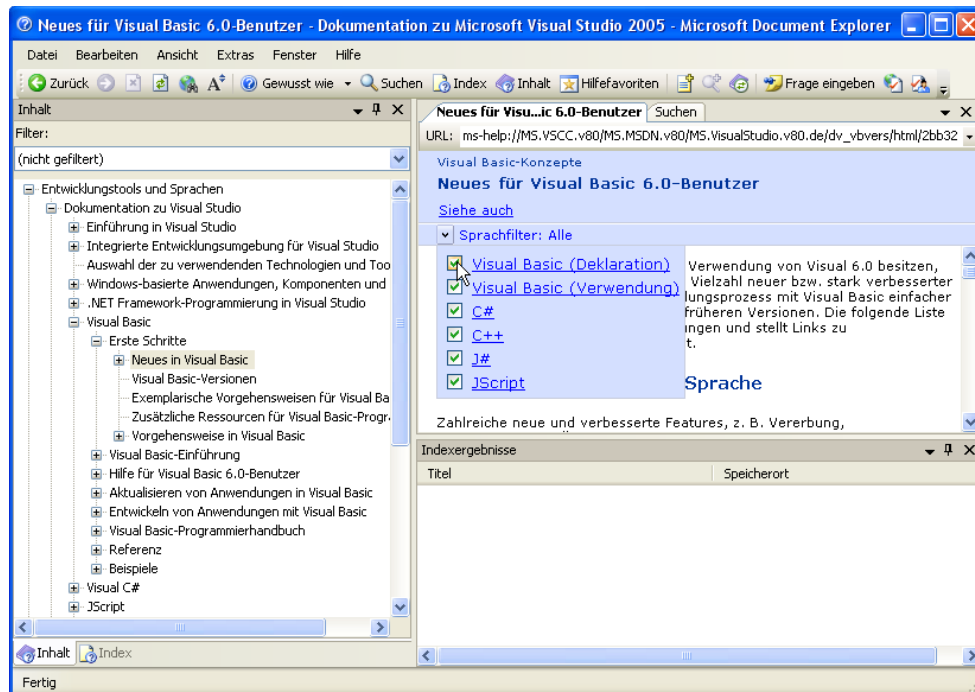


Abbildung 2.42: Fenster der Hilfe

Hinweis

Informationen zu den Klassen der .NET-Klassenbibliothek sowie den zugehörigen Membern (Methoden und Eigenschaften) finden Sie im Inhaltsverzeichnis im Zweig *.NET-Entwicklung/Dokumentation zu .NET Framework SDK/Verweis auf Klassenbibliothek*. Wenn Sie gezielt über die Indexseite auf Member einer Klasse zugreifen möchten, empfiehlt sich die Angabe der betreffenden Klasse im Stichwort. Anstatt z.B. das Stichwort *ShowDialog* als Index einzugeben, wenn Sie etwas über die Anzeige von Formularen wissen möchten, verwenden Sie das Stichwort *Form.ShowDialog*. Dann wird nur ein Eintrag gefunden.

Teil 2

Einführung in Visual Basic

3

Visual-Basic-Grundlagen

Wer bisher noch nicht mit Visual Basic programmiert hat, erfährt in diesem Kapitel etwas über den grundlegenden Aufbau von Visual-Basic-2005-Programmen und kann nachlesen, wie sich Konstanten bzw. Variablen definieren lassen.

3.1 Die Grundstruktur eines VB-Programms

Bevor Sie mit der Erstellung von Visual-Basic-Programmen beginnen, müssen Sie sich über die grundsätzliche Struktur des Quellcodes im klaren sein. Sie sollten auch wissen, wie Anweisungen zu schreiben sind und wie Kommentare oder Fortsetzungszeilen hinterlegt werden.

3.1.1 Anweisungen, Fortsetzungszeichen und Kommentare

Anweisungen sind in Visual Basic alle Programmstellen, die durch den Compiler erkannt und in ausführbaren Code übersetzt werden. Die Anweisungen bestimmen quasi, was das Programm tun soll. So kann eine Anweisung zum Beispiel die Vorschrift zur Addition zweier Zahlen enthalten. Zudem gehören zu den Anweisungen auch Steuerbefehle, die bestimmte Einstellungen bei der Übersetzung vornehmen. Die folgenden Zeilen enthalten gültige Visual-Basic-Anweisungen.

```
Module Test
Sub Main()
x = 13 + 5
```

Anweisungen können dabei spezielle Schlüsselwörter aus Visual Basic enthalten. In obigem Code sind beispielsweise *Module* und *Sub Main* solche Schlüsselwörter. Je nach Schlüsselwort verlangt Visual Basic, dass die Anweisung mit einer zweiten Anweisung in einer der folgenden Programmzeilen abgeschlossen wird. Dies ist beispielsweise bei der Anweisung *Sub* der Fall. Dort muss im Quelltext irgendwann eine abschließende Anweisung *End Sub* auftreten.

Hinweis

Bei der Eingabe von Anweisungen wird in Visual Basic keine Unterscheidung zwischen Groß- und Kleinschreibung getroffen. Erkannte Schlüsselwörter (*Sub*, *End Sub*, *If*, *Then*, *Else*, *End If* etc.) werden im Codefenster der Entwicklungsumgebung (Visual Studio 2005 oder Visual Basic 2005 Express Edition) standardmäßig in blauer Farbe hervorgehoben.

Die Anweisungen sind im Codefenster gemäß den für Visual Basic geltenden Syntaxregeln einzugeben. Die nachfolgenden Zeilen enthalten weitere gültige Visual Basic-Anweisungen:

```
Wert = 10
Wert = Wert + 10
If Wert > 100 Then Wert = 100
Mwst = 0.1 : Ust = Wert * Mwst
```

Hier beginnen alle Anweisungen am linken Seitenrand. Zur besseren Strukturierung des Programms dürfen Sie die Anweisungen aber durch Leerzeichen oder Tabulatoren einrücken. Dies wird in den Beispielen des Buchs häufiger benutzt.

Visual Basic erlaubt Ihnen mehrere Anweisungen in einer Zeile aufzunehmen, wenn Sie diese durch Doppelpunkte : trennen. Aus Gründen der Übersichtlichkeit und Programmentransparenz sollten Sie jedoch auf diese Möglichkeit verzichten.

... und Fortsetzungszeilen

In Visual Basic beginnt eine neue Anweisung immer am Zeilenanfang. Bei sehr umfangreichen Anweisungen tritt dann das Problem auf, dass die Zeilenbreite sehr groß wird. Beim Ansehen und Bearbeiten solcher Anweisungen muss sehr viel horizontal gescrollt werden. Zudem leidet die Übersichtlichkeit des Quellprogramms sehr stark. Die folgende Anweisung verdeutlicht dies:

```
Microsoft.VisualBasic.MsgBox ("Das Ergebnis der ersten Berechnung ergab folgenden  
Wert: " & x)
```

Visual Basic bietet aber eine Möglichkeit, Anweisungen auf mehrere Zeilen aufzuteilen. Sie müssen am Ende einer Zeile jeweils ein Leerzeichen gefolgt von einem Unterstrich _ einfügen.

```
Microsoft.VisualBasic.MsgBox _  
    ("Das Ergebnis der ersten Berechnung ergab folgenden Wert: " _  
    & x)
```

Fortsetzungszeilen dürfen aber hinter dem Fortsetzungszeichen _ keine Kommentare aufweisen! Falls Sie eine über mehrere Zeilen verlaufende Anweisung kommentieren möchten, stellen Sie den Kommentar als eigene Zeile vor den betreffenden Block. Alternativ können Sie den Kommentartext an das Ende der letzten Anweisungszeile, die kein Fortsetzungszeichen mehr enthält, anhängen.

Aufpassen müssen Sie auch bei Zeichenketten. Soll ein längerer Text als Zeichenkette mit einem Fortsetzungszeichen versehen werden, ist mit Teilzeichenketten zu arbeiten, die mit dem &-Zeichen verknüpft werden. Das Fortsetzungszeichen muss weiterhin durch mindestens ein Leerzeichen von dem letzten Zeichen der Anweisung getrennt werden. Die nachfolgenden Anweisungen zeigen dies.

```
Microsoft.VisualBasic.MsgBox ("Das Ergebnis der ersten " & _  
    "Berechnung ergab folgenden Wert: " & x)
```

Die Zeichenkette wurde hier an der Trennstelle jeweils mit einem doppelten Anführungszeichen abgeschlossen bzw. eingeleitet, die Teilstücke sind durch das &-Zeichen wieder zusammenzufügen.

Kommentare im Programmcode

Häufig möchte man als Entwickler zusätzliche Informationen im Quellcode eines Visual-Basic-Programms hinterlegen. Diese Informationen dürfen aber nicht durch Compiler ausgewertet werden. Die Lösung für dieses Problem besteht in der Verwendung von sogenannten Kommentaren. Sie müssen den betreffenden Kommentartext durch ein Hochkomma ' einleiten. Der Visual-Basic-Compiler erkennt das Hochkomma und übersieht den Rest der Zeile. Das folgende Listing verdeutlicht die Verwendung von Kommentaren in einem Visual Basic-Programm.

```
'*****  
' File/Projekt: Module  
' Autor: G. Born www.borncity.de  
' Erzeugt ein einfaches Meldungsfeld unter Windows.  
'*****  
Module Test  
    Sub Main()      ' das ist das Hauptmodul  
        ' Jetzt eine Benutzermeldung ausgeben  
        Microsoft.VisualBasic.MsgBox("Es hat geklappt!")  
    End Sub  
End Module
```

Listing 3.1: Kommentarbeispiel

Die ersten fünf Zeilen bestehen aus einem Kommentar. Ich benutze einen solchen Kommentarkopf, um dort Hinweise zum Dateinamen, zum Autor, grobe Angaben zur Funktion des Programms etc. zu hinterlegen. Stößt man später auf einen Ausdruck dieses Quellcodes, lässt sich sofort nachlesen, wer das Programm erstellt hat und welchen Zweck es erfüllen soll. Sie werden diese Kommentare häufig in meinen Beispielen finden. Für eigene Zwecke können Sie dann zusätzliche Informationen wie das Erstellungsdatum, den Revisionsstand, Änderungen oder Fehlerkorrekturen etc. einfügen.

Sie können auch Anweisungszeilen mit Kommentaren versehen. Hierzu fügen Sie den durch ein Hochkomma eingeleiteten Kommentar einfach an das Zeilenende ein. Die folgende Anweisung aus obigem Listing nutzt dies:

```
Sub Main()      ' das ist das Hauptmodul
```

Die Anweisung *Sub Main()* wird durch den Visual-Basic-Compiler erkannt und entsprechend umgesetzt. Sobald der Compiler aber das Hochkomma erkennt, ignoriert er den

kompletten Rest der betreffenden Zeile und beginnt mit der Bearbeitung der Folgezeile. In obigem Fall teilt der Kommentar dem Leser mit, dass das Hauptmodul beginnt. Die beiden folgenden Zeilen verdeutlichen diesen Ansatz noch einmal:

```
' Eine komplette Kommentarzeile
Brutto = Netto * (100.00 + Mwst)/100.00 ' Brutto ermitteln
```

In der zweiten Zeile beginnt der Kommentar erst hinter der Anweisung, d.h., die am Zeilenanfang befindliche Anweisung wird noch ausgeführt.

Hinweis

Aus älteren Visual-Basic-Dialekten ist Ihnen vielleicht noch die *REM*-Anweisung bekannt, mit der sich Kommentare einleiten lassen. Das Schlüsselwort *REM* wird in Visual Basic 2005 ebenfalls erkannt. Aus Vereinfachungsgründen verwenden die meisten Entwickler aber das Hochkomma *'*, um die Kommentare einzuleiten. Sie finden die Projektdateien im Ordner *\Beisp\Kap03\Module* der Begleit-CD.

Neu: XML-Kommentare

Eine Neuerung in Visual Basic 2005 ist die Möglichkeit zur Verwendung von XML-Kommentaren. Dieser erlauben fortgeschrittenen Benutzern das Einfügen strukturierter Dokumentation im Visual-Basic-Code.

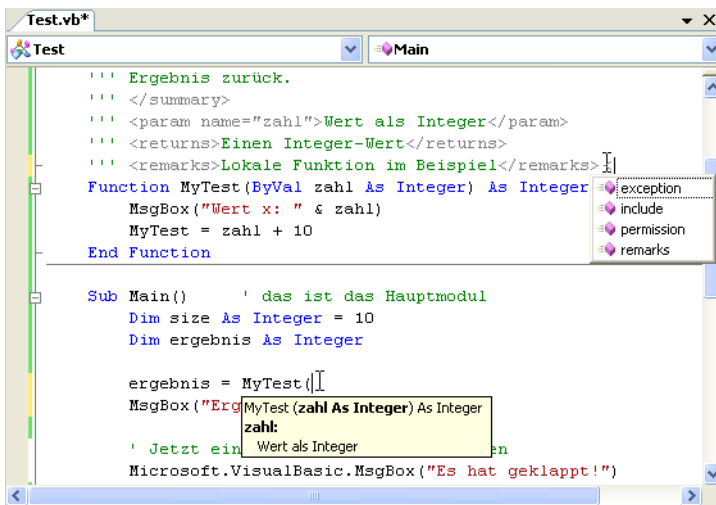


Abbildung 3.1: Eingabe und Wirkung von XML-Kommentaren

In Abbildung 3.1 sehen Sie den Inhalt des Codefensters mit dem Hauptprogramm *Main* und einer benutzerdefinierten Funktion *MyTest*. Die Schnittstelle dieser Funktion lässt sich mit XML-Kommentaren im Quellcode beschreiben.

1. Hierzu positionieren Sie die Einfügemarke des Editors auf die Zeile oberhalb der Funktionsdefinition und fügen drei Hochkommata ' im Quellcode ein.
2. Anschließend wird automatisch ein Rahmen für den Kommentar erstellt. Fügen Sie das gewünschte XML-Kommentarelement hinter den Hochkommata im Quellcode ein.

Das erste Hochkomma bewirkt, dass der Compiler die Zeile als Kommentar erkennt, während die beiden restlichen Anführungszeichen die Entwicklungsumgebung darauf hinweisen, dass ein XML-Kommentar folgen kann. Wenn Sie danach die öffnende spitze Klammer < für ein XML-Element eintippen, öffnet die Entwicklungsumgebung ein Kontextmenü mit den verfügbaren Elementen. Sobald der erste Buchstabe des XML-Elements eingetippt wurde, markiert die Entwicklungsumgebung den zutreffenden XML-Elementnamen im Menü. Klicken Sie auf einen Kontextmenübefehl oder drücken Sie die -Taste, wird das XML-Element in der Zeile eingefügt. Sie können dann ggf. noch Attribute des XML-Elements angeben und mit Werten füllen. Danach ist das Element mit der schließenden spitzen Klammer > abzuschließen. Der Editor ergänzt dann automatisch den abschließenden XML-Tag des Elements.

Das <summary>-Element dient zur Aufnahme einer Beschreibung des kommentierten Code-Features. Zu übergebende Parameter werden in <param>-Elementen spezifiziert. Das name-Attribut gibt dabei den Parameternamen an, während der Wert des <param>-Elements die Parameterbeschreibung aufnimmt. Über das <returns>-Element hinterlegen Sie die Information über zurückgelieferte Ergebnisse, während mit dem <remarks>-Element zusätzliche Bemerkungen im Code hinterlegt werden können.

Wurden Prozeduren oder Funktionen auf diese Weise kommentiert und tippen Sie an anderer Stelle im Quellcode den Prozedur- oder Funktionsnamen ein, kann die Entwicklungsumgebung die Schnittstellenbeschreibung als QuickInfo einblenden (Abbildung 3.1, unterer Teil des Codefensters). Die Info erscheint, sobald Sie die öffnende Klammer für die Parameter der Prozedur oder Funktion eintippen. Alternativ können Sie die Information auch über die Schaltflächen *Parameter anzeigen* in der Symbolleiste der Entwicklungsumgebung abrufen. Klicken Sie dagegen auf den Namen der Funktion/Prozedur oder eines Parameters, blendet die Schaltfläche *QuickInfo anzeigen* der Symbolleiste die Funktions-/Prozedurdefinition oder die Definition des Parameters ein.

Nutzen der Gliederungsfunktion

Wenn Sie im Codefenster Funktionen, Prozeduren, Datenstrukturen oder XML-Kommentare eingeben, handelt es sich um Codeblöcke, die eine in sich geschlossene Struktur besitzen. Um die Übersichtlichkeit bei umfangreichen Quellprogrammen zu erhöhen, können Sie die Gliederungsfunktion des Editors verwenden, um den Inhalt solcher Codeblöcke ein- oder auszublenden.

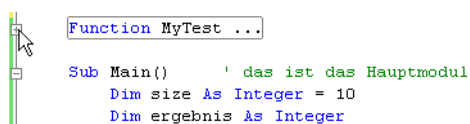


Abbildung 3.2: Nutzen der Gliederung

Hierzu klicken Sie einfach auf die am linken Rand des Codefensters angezeigten stilisierten Plus- oder Minuszeichen. Ein Pluszeichen expandiert den Block, während ein Mausklick auf ein Minuszeichen den expandierten Block zu einer Zeile reduziert.

Hinweis

Beim Erstellen des Projekts generiert die Entwicklungsumgebung automatisch eine XML-Datei unter dem Projektnamen und speichert diese mit im Ausgabeverzeichnis. Diese XML-Datei enthält die im Quellcode hinterlegten XML-Kommentare und lässt sich ggf. durch andere Werkzeuge (z.B. Ndoc) in eine Projektdokumentation umsetzen.

Der Rahmen für den Quellcode: Modul oder Klasse?

Der Programmcode muss für Visual Basic 2005 ganz bestimmten Regeln entsprechen bzw. ist in bestimmten Strukturen zu hinterlegen.

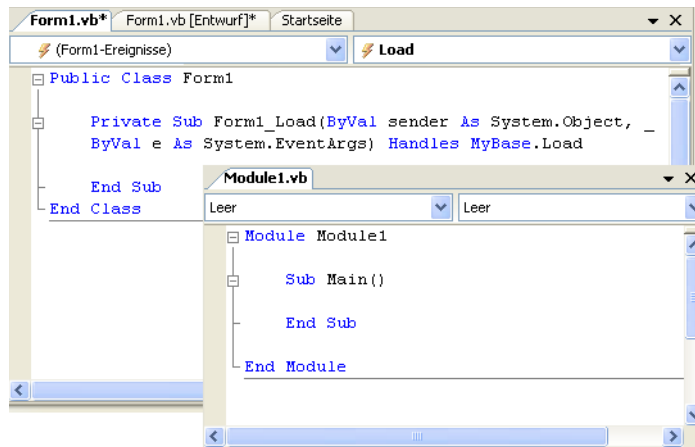


Abbildung 3.3: Programmrahmen im Code-Editor

Abbildung 3.3 zeigt die Ausschnitte des Code-Editors zweier Projekte. Der Code im Hintergrund gehört zu einer Windows-Anwendung, während der im Vordergrund gezeigte Codeausschnitt zu einer Konsolenanwendung gehört. Beide Varianten verwenden eine Anweisungsfolge der Art

```
Sub xxx ()
    ....
End Sub
```

Das Visual-Basic-Schlüsselwort `Sub` leitet eine Prozedur ein. Die Zeichen `xxx` stehen hier für den Namen der Prozedur (z.B. `Main()`, `Form1_Load()` etc.). Zwischen diese beiden Anweisungen wird der eigentliche Programmcode für das Hauptprogramm oder die Prozedur gepackt. Da ein Visual Basic-Programm mehrere Prozeduren oder Funktionen enthalten kann, muss das Ganze noch als Modul oder als Klasse gekapselt werden. Hierzu werden Schlüsselwörter wie `Module` oder `Class` verwendet.

Allerdings können Sie die Entscheidung, welche Schlüsselwörter benutzt werden, Visual Studio 2005 bzw. der Visual Basic 2005 Express Edition überlassen. Die Entwicklungsumgebung generiert beim Anlegen eines neuen Projekts oder eines Projektelements den betreffenden Rahmen zur Aufnahme des Visual-Basic-Codes.

- Legen Sie ein Projekt für eine Windows-Anwendung an, wird über die Vorlage automatisch ein Formular hinzugefügt. Das Formular wird dabei als Klasse angelegt, d.h., die im Formular benutzten Prozeduren werden durch die Anweisungen *Class xxx* und *End Class* eingeschlossen.
- Wählen Sie beim Anlegen eines neuen Projekts die Vorlage für Konsolenanwendungen, wird automatisch ein Modul mit den Anweisungen *Module xxx* und *End Module* eingefügt.

Der Name der Klasse oder des Moduls (hier mit dem Platzhalter *xxx* dargestellt), lässt sich im Dialogfeld zum Anlegen neuer Projekte wählen. Der Name des Moduls oder der Klasse wird relevant, wenn Sie von außen auf den Inhalt zugreifen möchten (z.B. auf eine Prozedur oder auf das Startformular der Anwendung).

Hinweis

Über optionale Schlüsselwörter wie *Public* oder *Friend* legen Sie die Zugriffsebene für das Modul fest. Die Zugriffsebene regelt, von wo aus auf das betreffende Modul zugegriffen werden darf. Verwenden Sie das Schlüsselwort *Public*, ist das Modul öffentlich, d.h. aus dem Projekt, aus anderen Projekten, die auf dieses Projekt verweisen und aus den daraus erstellten Assemblies zugreifbar. Mit dem Schlüsselwort *Friend* beschränken Sie den Zugriff dagegen auf die gleiche Assembly. Zudem erlaubt die *Module*-Deklaration noch optionale Parameter, die hier aber weggelassen werden. Details zu diesen Themen finden sich in der Visual-Basic-Hilfe.

3.2 Konstante in Visual Basic

In Visual-Basic-Programmen können Sie Konstante zum Speichern fester Werte verwenden. Der nachfolgende Abschnitt geht auf Fragen rund um das Arbeiten mit Konstanten ein.

3.2.1 Konstanten

In den Visual-Basic-Anweisungen können feste Werte (Literele) wie in folgender Anweisung auftauchen:

```
Gewicht = Netto + 100.0
```

Der (numerische) Wert *100.0* ist ein Literal und geht hier in die Berechnung ein. Sie können auch eine Zeichenkette als Literal im Programmcode angeben.

```
Text = "Es hat geklappt"
```

Die Anweisung weist der Variablen *Text* die Zeichenkette "Es hat geklappt" zu. Die Verwendung von Literalen im Code ist aber nicht immer günstig. Nehmen wir an, ein Programm weist eine Reihe von Anweisungen auf, in denen der Mehrwertsteuersatz von 16.0 hinterlegt ist. Bei umfangreichen Programmen mit vielen Anweisungen ist häufig nicht klar, was eine bestimmte Zahl wie 16.0 im Programmcode bedeutet. Das zweite Problem tritt auf, wenn sich ein Wert wie der Mehrwertsteuersatz ändert. Dann müssen Sie alle Stellen im Code ändern, an denen das betreffende Literal vorkommt.

Achtung

Für Einsteiger noch zwei Anmerkungen. Werte mit Nachkommastellen werden mit einem Dezimalpunkt im Quelltext angegeben. Die deutsche Schreibweise 100,00 wird bei Visual Basic einen Fehler auslösen. Die richtige Schreibweise ist dagegen 100.00. Zeichenketten werden in Visual Basic derart angegeben, dass der Text durch Anführungszeichen eingerahmt wird (z.B. "Dies ist ein Text").

Um den Code besser les- und pflegbar zu gestalten, empfiehlt sich die Verwendung von Konstanten in Visual Basic. Solche Konstanten besitzen einen Namen (Bezeichner) und einen festen Wert (der sich – im Gegensatz zu Variablen – während der Ablaufzeit des Programms nicht ändern kann).

Für die folgenden Ausführungen soll eine Konstante *Tara* benutzt werden, die z.B. das Nettogewicht eines Behälters angibt. Die Konstante muss im Deklarationsteil des Moduls oder der Klasse explizit definiert werden. Dies kann zum Beispiel so aussehen:

```
Const Tara = 100.00
```

Die Konstantendeklaration wird mit dem Schlüsselwort *Const* eingeleitet. Daran schließt sich der Name der Konstanten an. Weiterhin wird in der Anweisungszeile der Wert der Konstanten gesetzt. Diese Konstante lässt sich anschließend unter ihrem Namen im Visual Basic-Programm benutzen.

```
Bruttogewicht = Nettogewicht + Tara
```

Die benannte Konstante ist in obiger Anweisung sicherlich aussagekräftiger als der Wert 100.00. Zudem erleichtert sich die Programmpflege, wenn die wichtigsten Konstanten direkt im Programmkopf aufgeführt sind.

Wo darf ich die Const-Anweisung einsetzen?

Kommen wir auf die Frage zurück, wo innerhalb einer Programmstruktur eigentlich *Const*-Anweisungen eingefügt werden dürfen. Die Antwort darauf geht mit der Frage der Gültigkeit einer Konstanten einher. Das folgende Beispiel demonstriert einen typischen Fall zur Verwendung einer Konstanten.

```
'*****
' File/Projekt: Beispiel03_01
' Autor: G. Born www.borncity.de
' Erzeugt ein einfaches Meldungsfeld unter Windows.
'*****
Class Test
    Shared Sub Main()          ' das ist das Hauptmodul
        Const Tara = 100.0    ' Konstante auf Sub-Ebene
        ' Jetzt eine Benutzermeldung ausgeben
        MsgBox("Der Wert für Tara ist: " & Tara)
    End Sub
End Class
```

Listing 3.2: Programm mit lokaler Konstante

Die Konstante *Tara* wird hier im Kopf der Hauptprozedur *Main*, also nach der *Sub*-Anweisung eingefügt. Damit ist diese Konstante innerhalb des *Main*-Blocks verfügbar. In den folgenden Kapiteln werden Sie Beispiele kennen lernen, wo mehrere *Sub*-Blöcke innerhalb einer Klasse oder eines Moduls vorkommen.

Benötigen Sie innerhalb mehrerer *Sub*-Blöcke eine Konstante, müsste diese für das komplette Modul oder die Klasse gültig sein. Sie können dies lösen, indem die Konstantendeklaration auf die Ebene des Moduls oder der Klasse verlegt wird. Dies ist in folgendem Listing der Fall.

```
'*****
' File/Projekt: Beispiel03_02
' Autor: G. Born www.borncity.de
' Erzeugt ein einfaches Meldungsfeld unter Windows.
'*****

Class Test

    Const Tara = 100.00        ' Konstante auf Class-Ebene
    Shared Sub Main()          ' das ist das Hauptmodul
        ' Jetzt eine Benutzermeldung ausgeben
        MsgBox ("Der Wert für Tara ist: " & Tara)
    End Sub
End Class
```

Listing 3.3: Programm mit globaler Konstante

In diesem Beispiel steht die Konstantendeklaration nach der *Class*-Anweisung aber vor der *Sub*-Anweisung. Folglich ist die Konstante innerhalb der gesamten Klasse definiert. Gleiches gilt übrigens auch bei Verwendung einer *Module*-Anweisung.

Hinweis

Diese Ausführungen gelten übrigens auch für die Deklaration von Variablen, die auf den folgenden Seiten behandelt wird.

Wichtig ist aber, dass Sie keine Konstantendeklaration außerhalb der *Module*- oder *Class*-Blöcke vornehmen können. Schauen Sie sich einmal das folgende Codefragment an.

```
Const Fehler = "Falsche Position" ' Hier ist Const unzulässig

Class Test

    Const Tara = 100.00          ' Konstante auf Class-Ebene
    Shared Sub Main()           ' das ist das Hauptmodul
        Const Tara = 200.00     ' Konstante auf Prozedur-Ebene
        ' Jetzt eine Benutzermeldung ausgeben
        MsgBox ("Der Wert für Tara ist: " & Tara)
        Const Test = "hallo"    ' eine weitere Konstante
    End Sub
End Class
```

Die Deklaration der Konstante *Fehler* ist unzulässig, die Entwicklungsumgebung wird die Anweisung als fehlerhaft bemängeln. Visual Basic erlaubt übrigens die Konstantendeklaration überall innerhalb eines Moduls, einer Klasse oder einer Prozedur bzw. Funktion vorzunehmen. Dies lässt sich an der Konstanten *Test* erkennen, die in diesem Beispiel nach der Anweisung mit der *MsgBox*-Funktion vereinbart wird. Obwohl dies zulässig ist, empfehle ich aber, alle Konstantendeklarationen in den Kopf eines Moduls, einer Klasse oder einer Prozedur bzw. Funktion zu ziehen. Dies führt zu ordentlich strukturierten Quellcodes, in dem man auf einen Blick alle Konstantendeklarationen im Kopfteil des jeweiligen Abschnitts findet (also dort, wo man sie auch vermutet).

An dieser Stelle noch ein kurzer Hinweis auf die mehrfach deklarierte Konstante *Tara*. Einmal findet sich eine Deklaration auf Klassenebene und einmal innerhalb der *Main*-Prozedur. Dies ist ebenfalls zulässig. Kommt es zu einem solchen Fall, gilt immer die Deklaration im aktuellen Block. In obigem Beispiel bedeutet dies, dass die Konstante *Tara* innerhalb der Prozedur *Main* den Wert 200.00 besitzt, der global definierte Wert 100.00 wird einfach überschrieben.

Hinweis

Führen Sie eine weitere Prozedur in der Klasse ein und verwenden Sie in dieser Prozedur die global in der Klasse vereinbarte Konstante *Tara*, besitzt diese den Wert 100.00. Auf Fragen zur Gültigkeit von Variablen komme ich weiter unten bei der Variablendeklaration nochmals zurück. Sie finden die VB-Beispiele *Beispiel3_01* bis *Beispiel3_03* im Verzeichnis *\Beisp\Kap03* der Begleit-CD.

3.2.2 Spezielle Hinweise zu Konstanten

Wer nun glaubt, alles über Konstanten zu wissen, sollte sich doch einmal die folgenden Ausführungen zu Gemüte führen. Einmal dürfen Sie gleich mehrere Konstanten in einer Zeile definieren:

```
Const MwSt = 0.15, Tara = 100.0
```

Die einzelnen Konstantendeklarationen sind durch Kommata zu trennen. Weiterhin ist es so, dass Konstanten auch in Visual Basic einen Typ besitzen. Eine numerische Konstante wie 100.00 ist sicherlich etwas anderes als eine Zeichenkettenkonstante wie "Hallo, ein schöner Tag". Sobald eine Konstantendeklaration in einer Anweisungszeile erkannt wird, weist Visual Basic dieser Konstanten neben dem Wert automatisch den betreffenden Datentyp zu. Wird kein Datentyp angegeben, verwendet Visual Basic 2005 automatisch den Datentyp *Object*. Sie haben aber die Möglichkeit, den Datentyp innerhalb der Konstantendefinition zu vereinbaren.

```
Const Zuschlag As Long = 100.0
Const MwSt As Long = 0.15
Const Limit As Integer = 100.0
```

Das Schlüsselwort *As* mit der folgenden Typangabe wird Ihnen bei der Deklaration von Variablen weiter unten noch begegnen. Dort finden Sie auch Hinweise zu den für Variablen und Konstanten zulässigen Datentypen.

3.2.3 Zugriffsberechtigungen auf Konstante

An dieser Stelle möchte ich auch noch auf die Zugriffsberechtigungen zu sprechen kommen. Zugriffsberechtigungen gelten sowohl für Konstanten als auch für Variablen. Über diese Berechtigungen wird festgelegt, wo eine Konstante oder eine Variable sichtbar ist, d.h. ob aus einer Klasse oder einer Prozedur darauf zugegriffen werden kann. Auf den vorhergehenden Seiten wurde bereits in einem Beispiel gezeigt, dass man eine Konstante gleichen Namens auf Modul- bzw. Klassenebene und dann nochmals auf Prozedurebene definieren kann. Eine Konstante gilt standardmäßig immer nur in dem Block, in dem sie definiert wurde (bzw. in den zugehörigen Unterblöcken). Eine auf Klassenebene definierte Konstante oder Variable kann in allen Blöcken dieser Klasse benutzt werden. Außerhalb der Klasse ist sie nicht sichtbar. Eine in einer Prozedur deklarierte Konstante ist nur in der Prozedur sichtbar. Diese Zugriffsberechtigung lässt sich aber über spezielle Schlüsselwörter wie *Public*, *Private*, *Shared*, *Friend* etc. ändern. Sehen Sie sich einmal das folgende Beispiel an.

```
'*****
' File/Projekt: Beispiel03_04
' Autor: G. Born www.borncity.de
' Erzeugt einige einfache Meldungsfelder unter Windows.
'*****
```

Listing 3.4: Klasse für öffentliche Konstanten

```

Class Test

Class Konstanten
    Public Const Sprache = "Englisch"
    Public Const Tara = 100.00
End Class

Shared Sub Main()          ' das ist das Hauptmodul
    Const Tara = 200.00    ' Konstante auf Class-Ebene
    ' Jetzt eine Benutzermeldung ausgeben
    MsgBox ("Der Wert für Tara ist: " & Tara)

    MsgBox ("Sprache: " & Konstanten.Sprache & vbCrLf & _
           "Tara in Klasse: " & Konstanten.Tara)
End Sub
End Class

```

Listing 3.4: Klasse für öffentliche Konstanten (Forts.)

Die Klasse *Test* dient lediglich als Klammer für das Beispiel und enthält eine weitere Unterklasse *Konstanten* sowie eine Prozedur *Main*. Beides sind getrennte Blöcke. Möchte man nun aus der Prozedur *Main* auf die in der Klasse *Konstanten* definierten Konstanten *Sprache* und *Tara* zugreifen, muss der betreffende Typname angegeben werden. Daher enthalten die *MsgBox*-Aufrufe auch Angaben der Art *Konstanten.Sprache* bzw. *Konstanten.Tara*. Damit die Konstanten auch in anderen Prozeduren und Klassen sichtbar sind, muss die betreffende Zugriffsberechtigung erteilt werden. In obigem Beispiel wird dies durch das der *Const*-Anweisung vorangestellte Zugriffsmodifizierer *Public* erreicht. Insgesamt kennt Visual Basic für Konstanten, Variablen, Funktionen, Prozeduren etc. folgende Zugriffsmodifizierer:

- *Private*: Private Konstanten, die nur im Kontext der entsprechenden Blöcke (z.B. in einer Prozedur, in einer Klasse etc.) sichtbar und zugreifbar sind. *Private* ist der Standard-Zugriffsmodus, der vom Compiler zugewiesen wird, falls Angaben wie *Public* etc. fehlen.
- *Public*: Mit dem Schlüsselwort wird eine öffentliche Konstante (oder Variable etc.) definiert, die auch in anderen Blöcken (z.B. Klassen oder Prozeduren sowie deren Assemblies) sichtbar und zugreifbar ist.
- *Protected*: Diese Konstanten sind geschützt, d.h., es ist nur ein Zugriff in der eigenen Klasse oder aus einer daraus abgeleiteten Klasse möglich. Sie können das Schlüsselwort daher nur für Mitglieder einer Klasse (Klassenmember) verwenden.
- *Friend*: Mit diesem Schlüsselwort ausgezeichnete Konstanten können sowohl in dem Programm, welches die Deklaration enthält, als auch aus allen anderen Modulen und Klassen derselben Assembly gelesen werden.
- *Protected Friend*: Stellt eine Kombination der einzelnen Schlüsselwörter dar. Die Konstante kann im Code der Assembly sowie im Code abgeleiteter Klassen verwendet werden. Eine Verwendung dieses Schlüsselworts ist nur für Klassenmember zulässig.

Für lokale Konstanten (in einer Prozedur) gilt standardmäßig der öffentliche Zugriff. Für diese Konstanten können keine Zugriffsmodifizierer verwendet werden. Bei Konstanten, die auf Klassen-, Modul- oder Strukturebene deklariert werden, gelten wiederum andere Zugriffsmodi. Für Konstanten auf Klassen- und Modulebene (außerhalb einer Prozedur) gilt standardmäßig ein privater Zugriff, während Konstanten (Member) von Strukturen öffentlich sind. Sie können aber die Zugriffsebenen dieser Konstanten mit einem Zugriffsmodifizierer anpassen.

Hinweis

Wer bereits mit früheren Versionen von Visual Basic oder mit VBScript programmiert hat, kennt eine Reihe vordefinierter Konstanten (z. B. *vbOkOnly* etc.). Diese sind in Visual Basic 2005 nicht mehr Bestandteil der Sprache, sondern liegen in der Kompatibilitätsklasse *Microsoft.VisualBasic* vor. Die Entwicklungsumgebung (Visual Studio 2005 bzw. Visual Basic 2005 Express Edition) bindet einen Verweis auf die Assembly dieser Kompatibilitätsklasse im Projekt ein. Ist der Import auf den Namensraum (Namespace) nicht automatisch gesetzt, müssten Sie dies ggf. im Programmcode nachholen.

Für die frühere Visual-Basic-Konstante *vbCrLf*, die einen Zeilenumbruch in Meldungsfeldern durchführt, lässt sich z.B. auch die neue *newline*-Eigenschaft des Objekts, welches die betreffende Ausgabemethode bereitstellt, verwenden. Die Eigenschaft *newline* ist dabei intern als Zeichenkettenkonstante `"\r\n"` hinterlegt.

3.3 Regeln für Bezeichner

Bezeichner sind die im Programm verwendeten Namen für Konstanten, Variablen, Funktionen, Prozeduren etc. Diese Bezeichner müssen in Visual Basic 2005 gewissen Regeln genügen. So darf ein Bezeichner mit einem Unterstrich `_` beginnen und kann aus den Zeichen des Unicode-Zeichensatzes bestehen. Die folgende Regeln lassen sich zur Konstruktion von Bezeichnern verwenden:

- Bezeichner dürfen maximal 1.024 Zeichen lang sein. Aus Aufwandsgründen sollten Sie aber nie Bezeichner festlegen, die länger als 15 bis 20 Zeichen sind. In Visual Basic wird keine Unterscheidung zwischen Groß-/Kleinschreibung dieser Bezeichner gemacht.
- Ein Bezeichner darf mit einem Unterstrich beginnen, muss dann aber mindestens ein weiteres gültiges Bezeichnerzeichen (Ziffer oder Buchstabe) aufweisen – andernfalls wäre keine Unterscheidung zu einem Fortsetzungszeichen möglich.
- Ein Bezeichner kann aus Zahlen, Buchstaben und Zeichen des Unicode-Zeichensatzes bestehen. Nicht zulässig ist ein Bezeichner, der nur aus Ziffern besteht (dann ist keine Unterscheidung möglich, ob es sich um eine Zahl oder einen Bezeichner handelt).
- Fehlt der Unterstrich, muss das erste Zeichen eines Bezeichners ein Buchstabe sein (keine Ziffer).

- In einem Bezeichner dürfen keine Leerzeichen vorkommen. Die Verwendung reservierter Visual-Basic-Schlüsselwörter wie *Sub*, *Public* etc. ist nicht zulässig. Auch die Verwendung von Zeichen, die als Operanden verwendet werden (z.B. +, -, *, /), ist unzulässig.

Eine Auflistung aller Visual-Basic-Schlüsselwörter, die nicht als Bezeichner nutzbar sind, finden Sie in der Hilfe, wenn Sie nach den Stichwörtern »Schlüsselwörter von Visual Basic« suchen lassen. Ein Bezeichner *If* wäre also unzulässig, da dies einem der reservierten Schlüsselwörter entspricht. Auch die Angabe *123* ist gemäß den obigen Regeln als Bezeichner unzulässig, da es sich um eine Zahl handeln kann. Sie könnten aber einen Bezeichner der Art *_1* definieren – wobei sich aber die Frage nach dem Sinn dieses Namens stellt. Die Angabe *1Messung* wird von Visual Basic 2005 als Bezeichner ebenfalls abgelehnt, da das erste Zeichen kein Buchstabe ist. Als Sonderzeichen können Sie beispielsweise Umlaute einfügen. Die Angabe *Müllmenge* wäre als Bezeichner zulässig. Möchten Sie den Bezeichner optisch strukturieren, dürfen Sie keine Leerzeichen verwenden. Greifen Sie einfach auf den Unterstrich zu. Die Angabe *Einkommen_2006* ist ein zulässiger Bezeichner. Punkte sind dagegen im Bezeichnernamen unzulässig, da ein Punkt zur Trennung von Namen in Objektangaben (z.B. Hierarchie mehrerer Objekte, Objekt.Eigenschaft, Objekt.Methode oder in Namensräumen genutzt wird).

Achtung

Wählen Sie die Bezeichner so, dass diese Namen eine gewisse Aussagekraft besitzen und selbst dokumentierend sind. Konstantennamen der Art *x1*, *x2* etc. sind kaum aussagekräftig (was hat der Programmierer bloß damit gemeint?). Namen wie *MwSt*, *Tara*, *Offset* etc. dürften dagegen im Kontext des jeweiligen Programms einen Sinn ergeben. Ihre Bezeichner sollten möglichst nicht mit Sonderzeichen wie &, !, \$, %, &, # und @ enden. Diese Typzeichen (Typbezeichner) wurden in früheren Basic-Dialekten zur Kennzeichnung des Datentyps verwendet. Die Angabe *Name\$* stand beispielsweise für einen Bezeichner vom Datentyp *String*. Visual Basic 2005 unterstützt diese Typbezeichner aus Kompatibilitätsgründen. Wegen der schlechten Lesbarkeit ist diese implizite Typangabe aber sehr fehleranfällig.

3.4 Datentypen in Visual Basic

Im vorhergehenden Abschnitt haben Sie bereits erfahren, dass Konstanten einen Datentyp besitzen. Der Datentyp legt fest, wie die betreffende Information intern gespeichert wird und welche Verarbeitungsanweisungen auf den Werten zulässig sind. Eine Textkonstante wird sicherlich etwas anders als eine Zahl zu behandeln sein. Neben der impliziten Zuordnung eines Datentyps kann einer Konstanten auch explizit ein Datentyp zugewiesen werden. Das Gleiche gilt für die im folgenden Abschnitt behandelten Variablen, die in Visual Basic 2005 ebenfalls einen Datentyp besitzen.

3.4.1 Hintergrundinformationen zu den Datentypen

Visual Basic 2005 kennt eine ganze Sammlung an Datentypen, die Konstanten oder Variablen zugeordnet werden können. Wer aber bereits mit früheren Versionen von Visual Basic gearbeitet hat, muss an dieser Stelle etwas umlernen. Was früher Bestandteil der Sprache Visual Basic war, findet sich nun im .NET Framework.

Wie bereits in *Kapitel 1* erwähnt, stellt das .NET Framework eine Klassenbibliothek bereit. Für das .NET Framework hat Microsoft das Common Type System (CTS) definiert, welches für alle .NET-Programmiersprachen einen gemeinsamen Satz an primitiven Datentypen bereitstellt. Dies stellt sicher, dass ein Visual-Basic-Programm eine Variable oder Konstante mit einem bestimmten Datentyp definieren und ggf. einem in C# geschriebenen Modul als Argument übergeben kann. C# benutzt die gleichen primitiven Datentypen.

Hinweis

Allerdings ist es so, dass in den Programmiersprachen eigene Datentypen von den primitiven Datentypen abgeleitet werden können. Zudem ist es so, dass die Namen für die Datentypen in der Syntax der Programmiersprachen voneinander abweichen können. Sofern Sie nur mit Visual Basic 2005 programmieren, muss Sie dies aber nicht sonderlich interessieren.

Die primitiven Datentypen und die darauf zulässigen Operationen werden letztendlich über die .NET-Framework-Klassenbibliothek bereitgestellt. Sie finden die betreffenden Objekte im Namensraum (Namespace) *System*. Dies hat zur Folge, dass ein Visual Basic-Programmierer Datentypen sowohl über die internen Typen des *System*-Namensraums wie *System.Object* als auch über die Visual Basic-Schlüsselwörter für Datentypen wie *Object* vereinbaren kann. Verwenden Sie die Bezeichner des *System*-Namensraums, bleiben Sie in der Benennung über alle .NET-Sprachen einheitlich.

Für Programmierer, die bereits mit älteren Visual-Basic-Versionen gearbeitet haben, ergibt sich noch eine zusätzliche Änderung. Wurde in diesen Visual-Basic-Versionen der Variablen oder Konstanten kein Datentyp zugeordnet, führte dies zu einem Datentyp *Variant*. Dieser Datentyp kann unterschiedliche Inhalte (Zahlen, Datumswerte, Texte etc.) aufnehmen. Um es gleich vorweg zu nehmen, den Datentyp *Variant* gibt es in Visual Basic 2005 nicht mehr. Da die Datentypen über eine Klassenbibliothek bereitgestellt werden, sind Konstanten und Variablen letztendlich nur Objekte. Wird einer Konstanten oder einer Variablen kein Datentyp zugewiesen, erhält diese in Visual Basic 2005 daher automatisch den Typ *Object*.

3.4.2 Welche primitiven Datentypen gibt es im CTS?

Das Common Type System stellt neben dem Typ *Object* nur die wichtigsten primitiven Datentypen bereit, mit denen sich Zahlen, Zeichen und logische Werte abbilden lassen. Die nachfolgende Tabelle enthält eine Übersicht über die betreffenden Datentypen des Common Type System. In der Spalte *Beschreibung* habe ich die betreffenden von Visual Basic 2005 benutzten Aliasnamen für die Datentypen mit aufgenommen.

CTS-Datentyp/ Klassenname	Beschreibung
System.Byte	Eine 8-Bit-Ganzzahl ohne Vorzeichen. Visual Basic 2005 verwendet den Aliasnamen <i>Byte</i> für diesen Datentyp.
System.Int16	Eine 16-Bit-Ganzzahl mit Vorzeichen. Dies entspricht dem Visual-Basic-2005-Datentyp <i>Short</i> .
System.Int32	Eine 32-Bit-Ganzzahl mit Vorzeichen. Dies entspricht dem Visual-Basic-2005-Datentyp <i>Integer</i> .
System.Int64	Eine 64-Bit-Ganzzahl mit Vorzeichen. Dies entspricht dem Visual-Basic-2005-Datentyp <i>Long</i> .
System.Single	Eine Gleitkommazahl einfacher Genauigkeit (32 Bit). Dies entspricht dem Visual-Basic-2005-Datentyp <i>Single</i> .
System.Double	Eine Gleitkommazahl doppelter Genauigkeit (64 Bit). Dies entspricht dem Visual-Basic-2005-Datentyp <i>Double</i> .
System.Boolean	Ein boolescher Wert (true oder false), der 16 Bit belegt. Entspricht dem Visual-Basic-2005-Datentyp <i>Boolean</i> .
System.Char	Ein Unicode-Zeichen (16 Bit). Dies entspricht dem Visual-Basic-2005-Datentyp <i>Char</i> .
System.Decimal	Ein 96-Bit-Dezimalwert, der für finanzmathematische Berechnungen, die eine signifikante Zahl an Vor- und Nachkommastellen benötigen, geeignet ist. Der Datentyp entspricht dem Visual-Basic-2005-Datentyp <i>Decimal</i> . Intern werden die Daten auf 128 Bit abgebildet, wobei die binäre Darstellung einer Instanz von <i>Decimal</i> ein Bit für das Vorzeichen, 96 Bit für eine Ganzzahl und einen Skalierungsfaktor in den restlichen Bits umfasst. Der Skalierungsfaktor ist implizit 10 mit einem Exponenten zwischen 0 und 28.
System.Object	Dieser universelle Datentyp stellt den Stamm der Objekthierarchie dar und wird für alle Variablen und Konstanten ohne Typbezeichner benutzt. In Visual Basic 2005 wird der Aliasname <i>Object</i> für den Datentyp verwendet.
System.String	Eine unveränderliche Zeichenfolge fester Länge mit Unicode-Zeichen. Dies entspricht dem Visual-Basic-2005-Datentyp <i>String</i> .

Tabelle 3.1: Datentypen des Common Type Systems (CTS) in Visual Basic

Hinweis

Beachten Sie beim Datentyp *Object*, dass Visual Basic der Konstanten oder Variablen bei der Zuweisung eines Werts den internen Datentyp am Wert anpasst. Eine Variable vom Typ *Object* erhält z.B. bei der Zuweisung einer Zeichenkette den Typ *String*.

3.4.3 Welche Datentypen kennt Visual Basic 2005?

Die über die .NET-Klassenbibliothek im Namensraum *System* bereitgestellten primitiven Datentypen des Common Type Systems werden von allen .NET-Programmiersprachen unterstützt. Visual Basic 2005 kennt aber weitere Datentypen. Die folgende Tabelle listet alle in Visual Basic 2005 unterstützten Datentypen auf. In der Spalte *CTS-Datentyp* finden

Datentypen in Visual Basic

Sie die entsprechenden Pendanten des CTS-Datentyps. Enthält die betreffende Zeile keinen Eintrag, ist der Visual-Basic-2005-Datentyp nicht von CTS abgeleitet und daher auch nicht mit CTS kompatibel.

CTS-Datentyp	VB.NET-Datentyp	Kategorie	Wertebereich	Beschreibung
Byte	Byte	Ganze Zahl	0 bis 255	Eine 8-Bit-Ganzzahl ohne Vorzeichen
	SByte		-128 bis 127	Eine 8-Bit-Ganzzahl mit Vorzeichen. Dieser Visual-Basic-2005-Datentyp ist nicht CTS-kompatibel.
Int16	Short		-32.768 bis 32.767	Eine 16-Bit-Ganzzahl mit Vorzeichen
Int32	Integer		-2.147.483.648 bis 2.147.483.647	Eine 32-Bit-Ganzzahl mit Vorzeichen
Int64	Long		-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	Eine 64-Bit-Ganzzahl mit Vorzeichen
	UInt16		0 bis 65.535	Eine 16-Bit-Ganzzahl ohne Vorzeichen. Dieser Visual-Basic-2005-Datentyp ist nicht CTS-kompatibel.
	UInt32		0 bis 4.294.967.295	Eine 32-Bit-Ganzzahl ohne Vorzeichen. Dieser Visual-Basic-2005-Datentyp ist nicht CTS-kompatibel.
	UInt64		0 bis 18.446.744.073.709.551.615	Eine 64-Bit-Ganzzahl ohne Vorzeichen. Dieser Visual-Basic-2005-Datentyp ist nicht CTS-kompatibel.
Single	Single	Gleitkomma	-3,402823E+38 bis -1,401298E-45 für negative und 1,401298E-45 bis 3,402823E+38 für positive Werte	Eine Gleitkommazahl einfacher Genauigkeit (32 Bit) mit bis zu 7 Nachkommastellen
Double	Double		-1,79769313486231570E+308 bis -4,94065645841246544E-324 für negative und 4,94065645841246544E-324 bis 1,79769313486231570E+308 für positive Werte	Eine Gleitkommazahl doppelter Genauigkeit (64 Bit) mit bis zu 16 Nachkommastellen
Boolean	Boolean	Logisch	<i>True</i> oder <i>False</i>	Ein boolescher Wert (True oder False)
Char	Char	Sonstige	1 Unicode-Zeichen	Ein Unicode-Zeichen (16 Bit)

CTS-Datentyp	VB.NET-Datentyp	Kategorie	Wertebereich	Beschreibung
Decimal	Decimal		+/- 1E-28 bis +/-7,92E+28	Ein 96-Bit-Dezimalwert mit bis zu 28 Nachkommastellen. Der genaue darstellbare Zahlenbereich ist der Visual-Basic-Hilfe zu entnehmen.
Object	Object	Klassenobjekte	-	Der Stamm der Objekthierarchie
String	String		0 bis ca. 2 Milliarden Unicode-Zeichen	Eine unveränderliche Zeichenfolge fester Länge mit Unicode-Zeichen

Tabelle 3.2: Visual-Basic-2005-Datentypen

Hinweis

Auch die nicht zu CTS kompatiblen Visual-Basic-2005-Datentypen werden durch .NET Framework bereitgestellt und lassen sich über den Namensraum *System* ansprechen. Zudem stehen in Visual Basic 2005 die vorzeichenlosen, ganzzahligen Datentypen UShort, UInteger und ULong zur Verfügung.

Zwei Beispiele zum Umgang mit Datentypen

An dieser Stelle möchte ich Ihnen an zwei Beispielen demonstrieren, wie sich Datentypen zuweisen lassen und wie Sie den Datentyp einer Konstanten bzw. Variablen abfragen können. Nebenbei lernen Sie noch den Umgang mit Ausgabemeldungen auf Konsolebene kennen und erfahren, wie sich Datentypen ggf. konvertieren lassen.

Im ersten Beispiel werden einige Konstanten definiert. Hierbei weise ich diesen teilweise Datentypen aus dem Common Type System zu. Anschließend sollen die Werte der Konstanten im Fenster der Eingabeaufforderung angezeigt werden. Zusätzlich gibt das Programm noch den Datentyp der jeweiligen Konstanten aus (Abbildung 3.4). Die Visual Basic 2005-Anweisungen sind folgendem Listing zu entnehmen.

```

file:///F:/CD-Master 7Net/Beisp/Kap03/Beispiel3_05/bin/Debug/Beispiel3_05.EXE
Der Wert für Test0 ist: Ein Text - und der Typ ist: System.String
Der Wert für Test1 ist: 100 - und der Typ ist: System.Int32
Der Wert für Test2 ist: 200 - und der Typ ist: System.Int32
Der Wert für Test3 ist: 750 - und der Typ ist: System.Int32
Der Wert für Test4 ist: 50.01 - und der Typ ist: System.Double
Bitte die Eingabetaste drücken

```

Abbildung 3.4: Ausgaben im Fenster der Eingabeaufforderung

```

'*****
' File/Projekt: Beispiel03_05
' Autor: G. Born www.borncity.de
' Definiert verschiedene Konstanten und weist diesen
' CTS-Datentypen zu. Listet die Werte und deren Typen
' in der Console auf.
'*****
Imports System.Console      ' für WriteLine
Class Test
    Shared Sub Main()        ' das ist das Hauptmodul
        Const Test0 = "Ein Text"
        Const Test1 As System.Int32 = 100
        Const Test2 As Int32 = 200
        Const Test3 As Integer = 750
        Const Test4 As Double = 50.01

        ' Jetzt eine Benutzermeldung ausgeben
        Write("Der Wert für Test0 ist: " & Test0 )
        WriteLine(" - und der Typ ist: " & Test0.GetType.ToString)
        Write("Der Wert für Test1 ist: " & Test1 )
        WriteLine(" - und der Typ ist: " & Test1.GetType.ToString)
        Write("Der Wert für Test2 ist: " & Test2 )
        WriteLine(" - und der Typ ist: " & Test2.GetType.ToString)
        Write("Der Wert für Test3 ist: " & Test3 )
        WriteLine(" - und der Typ ist: " & Test3.GetType.ToString)
        Write("Der Wert für Test4 ist: " & Test4 )
        WriteLine(" - und der Typ ist: " & Test4.GetType.ToString)

        Write ("Bitte die Eingabetaste drücken")
        ReadLine ' verhindert das Schließen des Konsolenfensters
    End Sub
End Class

```

Listing 3.5: Beispiel zur Anzeige von Konstantenwerten

Schauen wir uns doch einmal die einzelnen Anweisungen des Beispiels etwas detaillierter an. Mit

```
Const Test0 = "Ein Text"
```

wird eine Konstante *Test0* vereinbart, der ein Text zugewiesen wird. Da wir keinen Datentyp spezifizieren, verwendet der Visual-Basic-Compiler automatisch einen passenden Datentyp (hier wird wegen des Texts der Typ *String* benutzt). Die folgende Anwei-

sung zeigt beispielhaft, wie ein Datentyp aus der Common Language Specification der Konstanten *Test1* zugewiesen werden kann:

```
Const Test1 As System.Int32 = 100
```

Diese Konstruktion ist bereits aus den Beispielen der vorherigen Seiten bekannt, es wurde lediglich an Stelle des Visual Basic 2005-Alias *Integer* der Verweis auf den Namensraum *System.Int32* benutzt. Um sich den Schreibaufwand zu sparen, besteht auch die Möglichkeit, die folgende Kurzform

```
Const Test2 As Int32 = 200
```

zu verwenden. Der Compiler bindet dann das betreffende Objekt mit der Definition des Datentyps ein.

Hinweis

Der Namensraum (Namespace) gibt an, wo in der .NET-Klassenbibliothek (z.B. in *System*) die betreffenden Einträge (z.B. die Struktur *Int32*) zu finden sind. Sie können den Namensraum in einer Anweisung explizit angeben (z.B. *System.Int32*). Um sich Schreibarbeit zu sparen, bietet Visual Basic 2005 Möglichkeiten, den Namensraum global zu vereinbaren. Hierzu lassen sich entsprechende Verweise im Projekt einbinden (siehe *Kapitel 2*, Abschnitt »Die Einstellungen des Projekts anpassen«). Die für Visual Basic 2005 verfügbaren Projektvorlagen der Entwicklungsumgebung enthalten bereits standardmäßig einen Verweis bzw. den Import des Namensraums *System*. Sie können also statt *System.Int32* problemlos die Kurzform *Int32* im Programmcode verwenden. Anders sieht es aber ggf. beim Zugriff auf andere Klassen der Klassenbibliothek aus. Im Beispiel wird die Klasse *Console* aus dem Namensraum *System* benötigt, da dort Methoden zur Ausgabe von Text in der Konsole bereitgestellt werden. Falls Sie nicht bei jedem Zugriff auf die Methoden den Namensraum angeben möchten, müssen Sie diesen als Import vereinbaren. Sie können dies, wie in *Kapitel 2* im Abschnitt »Die Einstellungen des Projekts anpassen« erwähnt, global über die Projekteigenschaften definieren. Persönlich bevorzuge ich es aber, die *Imports*-Anweisung im betreffenden Projektelement (Modul, Klasse) anzugeben (wird in den Buchbeispielen genutzt, damit erkennbar ist, welche zusätzlichen Imports erforderlich sind). Das Schlüsselwort *Imports* muss vor der *Module*- oder *Class*-Definition auftauchen. Im obigen Beispiel wurde die Anweisung *Imports System.Console* im Code hinterlegt. Anschließend kann im Visual-Basic-Code direkt auf die Namen der betreffenden Methoden (z.B. *WriteLine*) zurückgegriffen werden.

Bleibt noch die Frage, wie das Programm dem Anwender Informationen über die Konstanten und deren Datentypen liefern kann. Denkbar wäre die Verwendung der *MsgBox()*-Methode aus Visual Basic (die in den vorhergehenden Beispielen zum Einsatz kam). An dieser Stelle möchte ich aber eine Neuerung benutzen. .NET Framework unterstützt die Ausgabe auf Konsolenebene über die Methoden *Write()* und *WriteLine()* der Klasse *System.Console*. Daher wird im Programmkopf die Anweisung *Imports System.Console* angegeben. Anschließend lassen sich die Methoden in der Form

```
Write("Der Wert für Test0 ist: " & Test0)
WriteLine(" - und der Typ ist: " & Test0.GetType.ToString)
```

aufrufen. *Write()* erzeugt eine einfache Textausgabe, während *WriteLine()* nach der Textausgabe einen Zeilenvorschub auslöst. Der Ausgabetext ist der Methode als in Klammern gesetztes Textargument zu übergeben. Der Wert der betreffenden Konstante wird daher mit dem *&*-Operator mit der ebenfalls in Klammern stehenden Zeichenkettenkonstante verkettet (der Fachausdruck ist Konkatination). Durch den *&*-Operator führt der Visual-Basic-Compiler bei der Textausgabe automatisch eine Typumwandlung von Zahlenwerten durch – wie Sie dies durch die *Option Strict*-Direktive verhindern können, wird im nächsten Abschnitt gezeigt. Die Ausgabe mittels der *WriteLine()*-Methode soll dem Benutzer den Typ des Ausgabewerts anzeigen. Dadurch erhalten wir eine Rückmeldung, was Visual Basic 2005 aus den einzelnen Anweisungen gemacht hat. Den Datentyp einer Konstanten oder Variablen können Sie mit der *GetType()*-Methode ermitteln. Diese ist im Format *Object.GetType()* anzugeben, wobei *Object* für den Namen der Konstanten (oder Variablen) steht. Verknüpfen Sie den Rückgabewert der Methode über *&* mit einer Zeichenkettenkonstante, meldet der Compiler, dass dies für den String-Datentyp nicht unterstützt wird. Sie müssen daher das Ergebnis des Methodenaufrufs explizit in eine Zeichenkette umwandeln. Daher kommt die *ToString*-Methode zum Einsatz. Die Anweisung

```
Test0.GetType.ToString
```

ermittelt durch Anwendung der *GetType()*-Methode auf die Konstante *Test0* (die ein Objekt darstellt) den aktuellen Datentyp. Dieses Ergebnis wird dann mittels der *ToString()*-Methode in eine Zeichenkette umgewandelt. Wenn Sie sich die Ausgaben des Beispiels ansehen, stellen Sie fest, dass Visual Basic trotz Verwendung der Aliase wie *Double* intern mit den Datentypen aus dem Namensraum *System* arbeitet.

Hinweis

Die obige Anweisung zeigt sehr schön die Punktssyntax, die auf Objekte und Namensräume angewandt wird. *Test0* ist zwar der Name einer Konstanten. Da Datentypen aber über die .NET Framework Klassenbibliothek bereitgestellt werden, ist *Test0* gleichzeitig ein Objekt. Und das Objekt stellt die benutzten Methoden bereit. Die Methodennamen werden jeweils durch einen Punkt vom Objektnamen getrennt.

Die letzte *WriteLine()*-Anweisung in Kombination mit *ReadLine()* hat übrigens die Aufgabe, das Schließen des Konsolenfensters zu verhindern (falls das Programm aus Windows per Doppelklick aufgerufen wird). Erst wenn der Benutzer die -Taste drückt, schließt sich das Konsolenfenster. Diese Technik kommt bei allen Beispielen zum Einsatz, die im Fenster der Eingabeaufforderung ablaufen. Sie finden das Beispiel als Projekt im Ordner *Beisp\Kap03\Beispiel03_05* der Begleit-CD.

3.5 Arbeiten mit Variablen

Variablen sind benannte Platzhalter, die im Programm in einzelnen Anweisungen auftreten können. Dabei kann der Wert der Variablen (im Gegensatz zur Konstante) im Programm verändert werden. Bezüglich der Bildung der Variablennamen gelten die bereits weiter oben im Abschnitt »Regeln für Bezeichner« besprochenen Regeln. Variable lassen sich in Visual Basic 2005 durch das Schlüsselwort *Dim* deklarieren. Zudem kann der Variablen bei der Deklaration ein Datentyp zugewiesen werden. Die folgenden Anweisungen vereinbaren einige Variable:

```
Dim Alter
Dim Monat, Tag, Jahr
Dim Limit As Long
Dim Grenze As System.Int32
Dim Std, Min, Sek As Integer
```

Die ersten beiden Zeilen vereinbaren die Variable ohne die Angabe eines Datentyps. Visual Basic 2005 weist dann implizit den Datentyp *Object* zu. Dieser Datentyp ist in der Lage, Werte unterschiedlicher Typen (Datum, Zahlen, Texte, Objektreferenzen) aufzunehmen. In der dritten Zeile wird ein Visual-Basic-2005-Alias *Long* als Datentyp benutzt, die vierte Zeile greift dagegen auf die im Common Type System vereinbarten Datentypen zu und vereinbart einen 4-Byte-Integerwert. Sie können für jede Variable eine eigene *Dim*-Anweisung verwenden. Alternativ lassen sich, wie in der letzten Anweisungszeile gezeigt, mehrere Variablen in einer Anweisungszeile vereinbaren. Dann sind die Variablennamen durch Kommata zu trennen. Möchten Sie der Variablen einen Datentyp zuweisen, hängen Sie den Datentyp über eine *As*-Anweisung an. Die Vereinbarung eines Datentyps bewirkt, dass die Variable nur Daten dieses Typs aufnehmen kann. Eine Variable vom Datentyp *Integer* kann also keine Dezimalzahlen speichern. Bei Zuweisungen werden nicht passende Datentypen nach Möglichkeit konvertiert. Ist eine Konvertierung nicht möglich, löst dies einen Fehler aus.

Achtung

Leser, die bereits mit früheren Versionen von Visual Basic programmiert haben, seien auf eine Neuerung hingewiesen. Die Deklaration in der letzten Zeile der obigen Codesequenz vereinbart drei Variablen mit den Namen *Std*, *Min* und *Sek*. In früheren VB-Versionen hätte die Variable *Sek* den Datentyp *Integer* erhalten, während den Variablen *Std* und *Min* implizit der Datentyp *Variant* zugeordnet wurde. In Visual Basic 2005 erhalten alle Variablen den Datentyp zugewiesen, der am Zeilenende hinter *As* angegeben wird.

Zur Verbesserung der Lesbarkeit Ihres Quellcodes empfehle ich Ihnen, jeweils nur einen Variablennamen pro Zeile zu deklarieren und der Variablen über *As* einen Datentyp zuzuweisen. Verzichten Sie, wegen der Fehleranfälligkeit, auf die (aus früheren Basic-Dialekten bekannte) implizite Typdefinition durch an den Variablennamen angehängte Sonderzeichen (z.B. *Text\$*, *Zahl&*). Seien Sie auch zurückhaltend bei der Verwendung des impliziten Datentyps *Object*.

Spätestens bei der Weitergabe solcher Variable über die Programmgrenzen hinweg (z.B. beim Speichern in Datenbanken oder bei Aufrufen externer Funktionen) kann es durch unverträgliche Werte zu Laufzeitfehlern kommen. Zudem kann der Compiler optimaleren Code erzeugen, wenn der Typ einer Variable bereits zur Übersetzungszeit bekannt ist. Bei einer Variable vom Typ *Object* muss der Compiler hingegen Code erzeugen, der ggf. eine Typumwandlung zur Laufzeit (wenn ein Wert zugewiesen wird) ermöglicht.

Bei der Deklaration werden die Variablen mit einem sogenannten Initialisierungswert belegt. Alle numerischen Datentypen (einschließlich Byte) erhalten den Wert 0, in Variablen vom Typ *Char* wird eine binäre Null hinterlegt (entspricht einem leeren Zeichen). Eine Boolesche Variable wird auf den Wert *False* gesetzt, ein Datumswert erhält den Vorgabewert 00:00 Uhr am 1. Januar des Jahres 1, mit dem die Datumsberechnung startet. Verweistypen wie *Object* oder *String* werden auf den Wert *Nothing* gesetzt.

Nachdem eine Variable deklariert wurde, kann dieser im Programm ein Wert zugewiesen werden. Die Anweisung

```
Alter = 10
```

weist der Variablen *Alter* einen Wert zu. Alternativ lassen sich Variablen bereits bei der Deklaration mit einem Initialisierungswert versehen. Die folgenden Anweisungen demonstrieren dies:

```
Dim Alter As Integer = 18
Dim Limit As Long = 300
Dim Grenze As System.Int32 = -13
Dim Text = "Hallo, dies ist ein Text"
```

Der Initialisierungswert wird einfach durch ein Gleichheitszeichen getrennt an die Deklaration angehängt. Beachten Sie, dass in der letzten Anweisung kein Datentyp vereinbart wurde. Der Visual-Basic-Compiler verwendet den Datentyp *Object* und weist diesem dann den Text zu. Intern wird der Text als Datentyp *String* in der Variablen gespeichert.

3.5.1 Wo dürfen Variablen deklariert werden?

Die Variablendeklaration ist im Programm an verschiedenen Stellen möglich. Der Ort der Deklaration bestimmt auch die Gültigkeit der Variablen bzw. welche Schlüsselwörter zur Deklaration verwendet werden können.

- Sie können Variablen innerhalb einer Prozedur (Sub) oder innerhalb einer Funktion (Function) mit *Dim* deklarieren. Dann ist die Variable innerhalb der Prozedur oder Funktion verwendbar.
- Zusätzlich lassen sich Variablen auch in Blöcken (z.B. innerhalb eines *If..End If*-Blocks) vereinbaren. Dann sind die Variablen nur innerhalb dieses Blocks verwendbar.

- Sie dürfen Variablen auch außerhalb einer Prozedur oder Funktion auf Modulebene oder auf Klassenebene vereinbaren. Dann ist die Variable innerhalb des Moduls oder innerhalb der Klasse verwendbar.

Das hört sich etwas kompliziert an, gleicht aber den weiter oben bei Konstanten erläuterten Gültigkeitskriterien. Schauen wir uns einfach einige Fälle an Programmbeispielen an. Das folgende Listing vereinbart eine lokale Variable *Gewicht* innerhalb einer Prozedur.

```
*****
' File/Projekt: Beispiel03_06
' Autor: G. Born www.borncity.de
' Zeigt, wie sich Variablen deklarieren lassen.
*****
Imports System.Console

Class Test
    Shared Sub Main()                ' das ist das Hauptmodul
        Dim Gewicht As Single = 200.00 ' Variable auf Sub-Ebene
        ' Gebe jetzt die Variablenwerte aus
        WriteLine ("Gewicht: {0}", Gewicht)

        Write ("Bitte die Eingabetaste drücken")
        ReadLine ' verhindert das Schließen des Konsolenfensters
    End Sub
End Class
```

Listing 3.6: Verwendung einer lokalen Variablen

Die Variable *Gewicht* ist nur innerhalb der Prozedur *Sub Main* sichtbar bzw. lässt sich nur dort benutzen.

Tipp

An dieser Stelle möchte ich Ihnen noch einen kleinen Hinweis zur Verwendung der *WriteLine()*-Methode geben. Diese und die *Write()*-Methode erlaubt das Arbeiten mit Platzhalterzeichen. Sie können der Methode mehrere Parameter übergeben. Die Zeichenkette im ersten Parameter darf dabei Platzhalter der Art {0}, {1} etc. enthalten. Dann ersetzt die Methode diese Platzhalter zur Laufzeit des Programms durch die Werte der folgenden Parameter. Der Platzhalter {0} wird durch den ersten Wert (d.h. durch das zweite Argument) ersetzt und so weiter.

Betrachten wir jetzt die Situation, bei der auch Variablen auf Modulebene deklariert werden. Gemäß den Ausführungen am Kapitelanfang dürfen Sie ja aus Kompatibilitätsgründen noch die *Module*-Deklaration verwenden. Das folgende Listing vereinbart zusätzlich zur lokalen Variablen *Gewicht* noch eine Variable *Alter* auf Modulebene. Beide Variablen werden mit *Dim* vereinbart und der Einfachheit halber mit einem festen Wert

initialisiert. Die *WriteLine()*-Methode kommt hier mit zwei Platzhaltern zum Einsatz und sorgt für die Anzeige der beiden Werte auf der Konsole.

```
'*****
' File/Projekt: Beispiel03_07
' Autor: G. Born www.borncity.de
' Zeigt, wie sich Variablen deklarieren lassen.
'*****

Imports System.Console

Module Test
    Dim Alter As Integer = 18

    Sub Main()
        ' das ist das Hauptmodul
        Dim Gewicht As Single = 200.00 ' Variable auf Sub-Ebene
        ' Gebe jetzt die Variablenwerte aus
        WriteLine ("Alter: {0} Gewicht: {1}", Alter, Gewicht)
        Write ("Bitte die Eingabetaste drücken")
        ReadLine ' verhindert das Schließen des Konsolenfensters
    End Sub
End Module
```

Listing 3.7: Beispiel mit lokalen und globalen Variablen

Sie sehen im Listing, dass die global im Modulkopf vereinbarte Variable auch innerhalb der Prozedur genutzt werden kann. Dies liegt daran, dass die Prozedur *Main* innerhalb des Moduls liegt.

Bei Klassen heißt es aufpassen

Die obigen Beispiele sind eigentlich recht einfach und leicht nachzuvollziehen. Falls Sie statt mit Modulen mit Klassen arbeiten, müssen Sie aber das Schlüsselwort *Shared* vor die *Sub Main()*-Anweisung stellen, damit die Prozedur *Main* ohne explizite Instantiierung genutzt werden kann. Der Compiler wird aber beim Übersetzen eine Fehlermeldung bringen, sobald Sie in der Prozedur die auf Klassenebene vereinbarte Variable verwenden. Hier wurde zusätzlich das Schlüsselwort *Shared* innerhalb der Variablendeklaration mit *Dim* benutzt, damit die Variable auch ohne Instantiierung in der Prozedur verwendet werden kann. Das folgende Listing zeigt das betreffende für eine Klasse angepasste Beispiel:

```
'*****
' File/Projekt: Beispiel03_08
' Autor: G. Born www.borncity.de
' Zeigt, wie sich Variablen deklarieren lassen.
'*****
```

Listing 3.8: Beispiel mit globaler Variable in einer Klasse


```
Imports System.Console

Class Test
    Dim Shared Alter As Integer = 18

    Shared Sub Main()                ' das ist das Hauptmodul

        Dim Gewicht As Single = 200.00 ' Variable auf Sub-Ebene
        ' Gebe jetzt die Variablenwerte aus
        WriteLine ("Alter: {0} Gewicht: {1}", Alter, Gewicht)

        Write ("Bitte die Eingabetaste drücken")
        ReadLine ' verhindert das Schließen des Konsolenfensters
    End Sub
End Class
```

Listing 3.8: Beispiel mit globaler Variable in einer Klasse (Forts.)

3.5.2 Variablen aus anderen Klassen nutzen

In einer Klasse ist es durchaus zulässig, Unterklassen neben Prozeduren oder Funktionen einzufügen. Dabei lässt sich vorstellen, dass eine solche Unterklasse ebenfalls Variablendeklarationen besitzt, die in Prozeduren verwendet werden müssen. Ich habe dies einmal an einem einfachen Beispiel realisiert.

```
'*****
' File/Projekt: Beispiel03_08
' Autor: G. Born www.borncity.de
' Zeigt, wie sich Variablen in einer Klasse deklarieren
' und in einer Prozedur verwenden lässt.
'*****
Imports System.Console

Class Test
    Dim Shared Alter As Integer = 18

    Class MyData
        Friend Shared Geschlecht As String = "Männlich"
        Private Geburtsdatum As Date = "15.3.2002"
    End Class

    Shared Sub Main()                ' das ist das Hauptmodul
```

Listing 3.9: Variablen einer anderen Klasse nutzen

```

Dim Gewicht As Single = 200.00 ' Variable auf Sub-Ebene
' Gebe jetzt die Variablenwerte aus
WriteLine ("Alter: {0} Gewicht: {1}", Alter, Gewicht)
WriteLine ("Geschlecht: {0}", MyData. Geschlecht)

Write ("Bitte die Eingabetaste drücken")
ReadLine ' verhindert das Schließen des Konsolenfensters
End Sub
End Class

```

Listing 3.9: Variablen einer anderen Klasse nutzen (Forts.)

Die Variable *Alter* auf der obersten Ebene der Klasse *Test* wird mit dem Schlüsselwort *Shared* vereinbart, damit sie innerhalb der Klasse direkt (ohne Instantiierung) verwendbar ist. Neu hinzugekommen ist eine Unterklasse *MyData*, in der zwei Variablen *Geschlecht* und *Geburtsdatum* vereinbart wurden. Diese Variablen sind standardmäßig nur innerhalb dieser Unterklasse bekannt. Um die Variable *Geschlecht* auch außerhalb der Klasse *MyData* auf der Ebene der Prozedur oder der Klasse *Test* verwenden zu können, habe ich diese mit dem Zugriffsmodifizierer *Friend* versehen. Zudem muss das Schlüsselwort *Shared* eingefügt werden, damit die Variable direkt in der Prozedur *Main* verwendbar ist. Die Variable *Geburtsdatum* wird dagegen als *Private* deklariert und ist damit außerhalb der Unterklasse unbekannt.

Um die Variable *Geschlecht* in der Prozedur *Main* ansprechen zu können, muss der komplette Namensraum mit angegeben werden. Die Variable lässt sich also über *MyData.Geschlecht* ansprechen, wobei *MyData* der Klassenname ist, in dem sich die Variable befindet.

Hinweis

Sie finden die Beispiele im Pfad *Beisp\Kap03* der Begleit-CD. Der betreffende Unterordner des Projekts ist im Kommentarkopf des jeweiligen Listings vermerkt.

3.5.3 Zugriffsberechtigung auf Variablen beeinflussen

Bei der Definition einer Variablen wird die Zugriffsberechtigung (d.h. von wo auf die Variable zugegriffen werden darf) implizit durch die Position im Programm beeinflusst. Dies wurde in den vorhergehenden Abschnitten skizziert. Auf eine mit *Dim* deklarierte Variable kann immer innerhalb des betreffenden Blocks zugegriffen werden. Sie können die Zugriffsberechtigungen auf Variablen aber durch verschiedene Schlüsselwörter beeinflussen. Die folgenden Anweisungen deklarieren ebenfalls Variable, legen gleichzeitig aber eine Gültigkeit fest:

```

Public Alter As Integer = 20
Private Sex As String = "weiblich"
Friend Gewicht As Single = 2.5

```

Eine Erläuterung hinsichtlich der Zugriffsmodifizierer wie *Public*, *Protected* etc. finden Sie nachfolgend. Es gilt praktisch das Gleiche wie bereits weiter oben im Abschnitt zu den Konstanten erläutert.

- *Public*: Dieses Schlüsselwort ist nur innerhalb von Modulen, Klassen oder Strukturen verwendbar. In Prozeduren darf *Public* nicht angegeben werden. Wird das Schlüsselwort *Public* angegeben, kann das Schlüsselwort *Dim* entfallen. Als *Public* deklarierte Variablen haben keine Zugriffsbeschränkungen.
- *Protected*: Erlaubt eine Variable zu schützen, d.h., der Zugriff ist nur in der eigenen Klasse oder in einer von dieser Klasse abgeleiteten Klasse zulässig. *Protected* lässt sich nur auf Klassenebene, nicht jedoch in Prozeduren oder Modulen, verwenden.
- *Friend*: Eine mit diesem Schlüsselwort deklarierte Variable lässt sich im gesamten Programm, welches die Deklaration enthält, sowie innerhalb der Assembly verwenden. Sie dürfen das Schlüsselwort nur innerhalb von Modulen oder Klassen oder Strukturen verwenden. Zudem ist noch eine Kombination aus *Protected* und *Friend* zulässig.
- *Private*: Wird eine Variable mit diesem Schlüsselwort deklariert, ist Sie nur im jeweiligen Deklarationskontext (z.B. innerhalb einer Klasse oder Prozedur) verwendbar. Sie können das Schlüsselwort innerhalb eines Moduls, einer Klasse oder einer Struktur (nicht jedoch in einer Prozedur) einsetzen.

Neben diesen Zugriffsmodifizierern für die Zugriffsmodi gibt es bei der Variablendeklaration noch Anweisungen, die die Gültigkeit und Verfügbarkeit regeln. Eine in einer Prozedur (*Sub xx End Sub*) deklarierte Variable verliert ihre Gültigkeit, sobald die Prozedur beendet wird. Mit dem Schlüsselwort *Static* lässt sich erreichen, dass der Wert einer Variable auch nach Beenden der Prozedur erhalten bleibt und den Wert behält. Das Schlüsselwort kann in Prozeduren und auf Blockebene verwendet werden. Beachten Sie aber, dass *Static* nicht zusammen mit *Shared* oder *Shadows* in einer Variablendeklaration auftreten darf.

Wenn Sie eine Variable innerhalb einer Klasse deklarieren, können Sie diese zur Verwendung freigeben, indem Sie das Schlüsselwort *Shared* in der Deklaration (auf Modul-, Klassen- oder Strukturebene) angeben. Zum Zugriff auf die Variable ist dann der Klassen- oder Strukturname anzugeben (z.B. *MyData.Sex*). Bei *Shared*-Variablen ist die Option *Static* unzulässig.

Hinweis

Auf der Ebene eines Moduls können Sie auch das Schlüsselwort *Shadows* verwenden, um eine Variable zu deklarieren, die den gleichen Bezeichner wie eine Variable einer Basisklasse (siehe Kapitel 7) besitzt. Dann wird der Bezeichner der Basisklasse durch die neu deklarierte Variable »überschattet«. Ein überschattetes Element ist in der abgeleiteten Klasse, von der es überschattet wird, nicht verfügbar. Zudem darf das *Static*-Schlüsselwort nicht zusammen mit *Shadows* benutzt werden.

Eine mit dem Schlüsselwort *ReadOnly* deklarierte Variable lässt sich nicht beschreiben. Dies ist in Modulen, Klassen oder Strukturen hilfreich, wenn eine Konstante als Member einer Klasse vereinbart werden soll. Weitere Details sind der .NET Framework SDK-Hilfe zu entnehmen.

Was bedeuten Option Explicit und Option Strict?

Visual Basic 2005 verlangt standardmäßig, dass alle im Programm benutzten Variablen explizit definiert werden. Vertippt man sich beim Eingeben des Programmcodes bei einem Variablennamen, wird dies beim Übersetzen des Programms bemängelt.

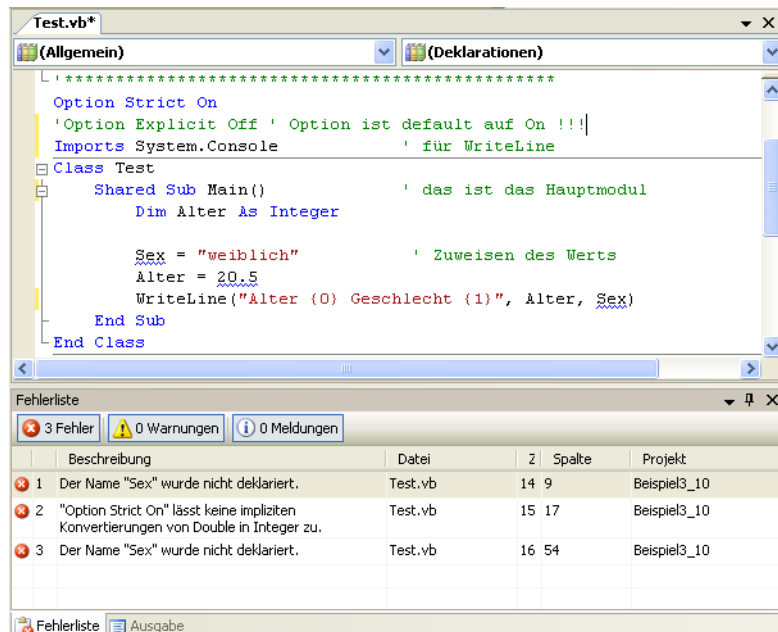


Abbildung 3.5: Fehlermeldungen beim Übersetzen

Dieses Verhalten lässt sich aber über die Option *Option Explicit* beeinflussen. In *Kapitel 2* wurde bereits erwähnt, dass sich diese Option in den Projekteigenschaften auf der Seite *Kompilieren* sowie global in der Entwicklungsumgebung auf *On* oder *Off* setzen lässt. Sie können diese Voreinstellung aber durch die im Programmkopf (oberhalb des Moduls oder der Klasse) hinterlegte Anweisung *Option Explicit Off* bzw. *Option Explicit On* überschreiben. In Abbildung 3.5 ist die Zeile mit *Option Explicit* auskommentiert, d.h., die Standard-Projekteinstellung *Option Explicit On* wird wirksam. Daher wird die Variable *Sex* als nicht deklariert bemängelt und im Codefenster mit einer blau geschlängelten Linie markiert.

In den auf den vorhergehenden Seiten gezeigten Beispielen kam häufiger eine automatische Typkonvertierung durch Visual Basic zum Einsatz. Zum Beispiel wurden Integerwerte mit der *MsgBox()*-Methode in einem Dialogfeld angezeigt. Die Verknüpfung mit einer Zeichenkette mittels des *&*-Operators (z.B. *"Text " & Alter*) führte dazu, dass Visual Basic automatisch die Zahlen in eine Zeichenkette umwandelte. Es ist aber nicht immer erwünscht, dass diese automatische Umwandlung erfolgt. Werden Variablen mit Datentypen versehen, soll der Compiler ggf. erkennen, wenn ein nicht kompatibler Wert zugewiesen wird. In *Kapitel 2* (Abschnitt »Die Einstellungen des Projekts anpassen«) wurde bereits erwähnt, dass sich die standardmäßig zugelassene implizite Typkonvertierung

mit der Einstellung *Option Strict On* unterdrücken lässt. Diese Einstellung der Entwicklungsumgebung (bzw. die Projekteinstellung) lässt sich mit der im Programmkopf hinterlegten Anweisung *Option Strict On* überschreiben. In Abbildung 3.5 ist diese Anweisung im Programmkopf vorhanden, wodurch die Typkonflikte (z.B. die Variable *Alter*) in der Fehlerliste gemeldet werden.

4 Arrays und Strukturen

Nachdem Sie im vorhergehenden Kapitel das Grundwissen zum Deklarieren und Nutzen von Variablen und Konstanten in Visual Basic 2005 erworben haben, geht es auf den nachfolgenden Seiten um Felder und selbstdefinierte Datentypen. Sie erfahren, wie sich Felder und benutzerdefinierte Datentypen unter Visual Basic 2005 definieren und nutzen lassen.

4.1 Arbeiten mit Feldern

Felder stellen spezielle Varianten von Variablen dar. In einer mit *Dim name As String* deklarierten Variable lässt sich jeweils nur ein Wert hinterlegen. Gelegentlich kommt es aber vor, dass mehrere Werte zu speichern sind. Nehmen wir als Beispiel die Namen der Wochentage. Schön wäre es, eine Variable *Wochentag* zu haben, über die sich die Namen der sieben Wochentage abrufen lassen. Realisiert wird dies mit so genannten Feldern, im englischen als Arrays bezeichnet.

4.1.1 Deklaration einer Feldvariablen

Eine Feldvariable wird im einfachsten Fall mit einer *Dim*-Anweisung angelegt. Ein einfaches so genanntes eindimensionales Feld kann mit folgender Deklaration

```
Dim Wochentag (6) As String
```

oder mit der neu in Visual Basic 2005 aufgenommenen Deklaration

```
Dim Wochentag (1 to 7) As String
```

angelegt werden. Diese Deklaration gleicht der Vereinbarung einer Variablen mit dem Datentyp *String*. Im Unterschied zu einer normalen Variablendeklaration findet sich jedoch hinter dem Variablen- bzw. Feldnamen eine Klammer. In dieser Klammer kann eine Zahl stehen, die den sogenannten oberen Feldindex angibt. Die erste Deklaration erzeugt eine Feldvariable *Wochentag* mit sieben Feldelementen. Da keine Arraygrenzen angegeben wurden, beginnt der Index bei 0, d.h., die Feldvariable lässt sich über *Wochentag(0)*, *Wochentag(1)* bis *Wochentag(6)* im Programm ansprechen. Die zweite Deklaration gibt die Feldgrenzen explizit zwischen 1 und 7 vor, d.h., der Zugriff muss über *Wochentag(1)* bis *Wochentag(7)* erfolgen.

Generell gleicht die Deklaration eines Feldes der Deklaration einer Variablen. Sie können also die gleichen Datentypen und auch die Schlüsselwörter wie *Public*, *Shared* etc. in der Deklaration verwenden. Allerdings kann die Zahl der Dimensionen für ein Feld nach der Deklaration nicht mehr verändert werden.

Um die Werte des, in der ersten Variante deklarierten, Feldes *Wochentag* zu setzen, lässt sich beispielsweise folgender Code verwenden:

```
Wochentag(0) = "Sonntag"
Wochentag(1) = "Montag"
Wochentag(2) = "Dienstag"
Wochentag(3) = "Mittwoch"
Wochentag(4) = "Donnerstag"
Wochentag(5) = "Freitag"
Wochentag(6) = "Samstag"
```

Im Programm können Sie also auf die einzelnen Elemente des Feldes zugreifen, indem Sie in der Klammer hinter dem Feldnamen den betreffenden Indexwert angeben. Natürlich darf sich der Indexwert nur in dem Bereich bewegen, der in der Deklaration der Variablen vorgesehen ist. Angaben der Art

```
Wochentag(-1) = "---"
Wochentag(7) = ""
```

sind dagegen unzulässig und lösen einen Fehler zur Laufzeit des Programms aus. Benötigen Sie entsprechende Indexwerte, müssen Sie die oben gezeigte zweite Variante mit vorgegebenen Indexgrenzen zur Deklaration verwenden.

Den Umgang mit einem Feld lernen Sie jetzt an einem einfachen Beispiel kennen. Das Programm soll die sieben Wochentage in einem Feld definieren, mit den Namen versehen und das Ergebnis auf der Konsole ausgeben (Abbildung 4.1). Der gesamte Quellcode des Programms ist folgendem Listing zu entnehmen:

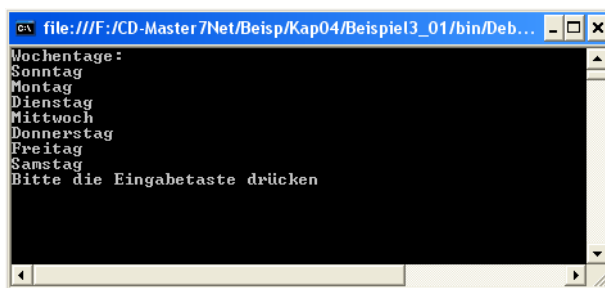


Abbildung 4.1: Anzeige der Wochentage im Fenster der Eingabeaufforderung

```
'*****
' File/Projekt:  Beispiel04_01
' Autor: G. Born www.borncity.de
' Definiert eine Feldvariable und weist dieser
' Werte zu. Listet die Werte in der Console auf.
```

Listing 4.1: Beispiel mit Feld zur Anzeige der Wochentage

```

'*****
Option Strict
Imports System.Environment ' für newline-Eigenschaft
Imports System.Console    ' für WriteLine
Class Test
    Shared Sub Main()      ' das ist das Hauptmodul
        Dim Wochentag (6) As String ' unser Feld

        Wochentag(0) = "Sonntag"
        Wochentag(1) = "Montag"
        Wochentag(2) = "Dienstag"
        Wochentag(3) = "Mittwoch"
        Wochentag(4) = "Donnerstag"
        Wochentag(5) = "Freitag"
        Wochentag(6) = "Samstag"

        WriteLine("Wochentage:" & newline & _
            Wochentag(0) & newline & _
            Wochentag(1) & newline & _
            Wochentag(2) & newline & _
            Wochentag(3) & newline & _
            Wochentag(4) & newline & _
            Wochentag(5) & newline & _
            Wochentag(6))

        Write ("Bitte die Eingabetaste drücken")
        ReadLine
    End Sub
End Class

```

Listing 4.1: Beispiel mit Feld zur Anzeige der Wochentage (Forts.)

Die Anweisungen zur Deklaration der Feldvariablen kennen Sie ja bereits aus den obigen Ausführungen. Die Namen der Wochentage werden anschließend in einfachen Zuweisungsoperationen in den Feldelementen *Wochentag(0)* etc. gespeichert.

Zur Ausgabe wurde hier die *WriteLine()*-Methode der Klasse *System.Console* des Namensraums *System* benutzt. Diese gibt den als Argument hinterlegten Text aus und erzeugt einen Zeilenvorschub im Fenster der Konsole. Auch dies ist nichts wesentlich Neues. In den vorherigen Abschnitten hatte ich erwähnt, dass sich die Werte als Argumente an die Methode übergeben und über Platzhalter der Art {0}, {1} etc. im Ausgabe-text einfügen lassen. Auf diesen Ansatz habe ich an dieser Stelle verzichtet. Vielmehr soll die Ausgabe manuell formatiert werden. Hierzu werden die jeweiligen Werte der Feldvariablen *Wochentag* mit dem &-Operator verknüpft. Da die Feldelemente vom Typ *String* sind, ist auch keine Typkonvertierung erforderlich. Ein Anweisung der Art


```
WriteLine("Wochentage:" & Wochentag(0) & Wochentag(1) ....)
```

hätte aber zur Folge, dass die Ausgabe in einer Zeile erfolgt. Abhilfe schafft die Konstante *newline*, die einen Zeilenumbruch bei der Ausgabe erzwingt. Diese Konstante ist als Eigenschaft der *Environment*-Klasse definiert. Daher müssen wir im Programmkopf neben dem Befehl

```
Imports System.Console
```

der zur Nutzung der *WriteLine()*-Methode erforderlich ist, auch die Klasse für *newline* importieren:

```
Imports System.Environment
```

Erst wenn diese Anweisung erkannt wird, kann der Compiler die Konstante aus der betreffenden Klasse übernehmen. Die Konstruktion *Wochentag(0) & newline* hängt einen Zeilenumbruch an den betreffenden Text für den Namen des Wochentags an. Da die Anweisung recht lang wird, verwende ich hier das Fortsetzungszeichen `_` und teile die Anweisung auf mehrere Zeilen auf.

Hinweis

In obigem Code werden die Elemente des Felds *Wochentag* explizit abgerufen. Mit den in *Kapitel 6* besprochenen Schleifen sind aber intelligentere Ansätze möglich. Mit einer *For Each i in Wochentag ... Next*-Sequenz werden die Feldelemente in der Variablen *i* bereitgestellt und lassen sich anschließend mit *WriteLine()* anzeigen.

Da es sich bei dem Beispiel um eine Konsolenanwendung handelt, muss dieses im Fenster der Eingabeaufforderung ausgeführt werden. Rufen Sie das Programm aus dem Fenster der Eingabeaufforderung auf, zeigt dieses die Ausgaben an und kehrt dann zur Eingabeaufforderung zurück. Startet der Benutzer die Anwendung aber per Doppelklick aus einem Ordnerfenster, öffnet sich das Fenster der Eingabeaufforderung und die Ausgaben erscheinen. Damit das Fenster sich nicht sofort schließt, habe ich (erneut) am Programmende die beiden Anweisungen

```
Write ("Bitte die Eingabetaste drücken")  
ReadLine
```

eingefügt. Die *Write()*-Anweisung gibt die Aufforderung an den Benutzer aus, eine Taste zu drücken. Die *ReadLine()*-Methode liest von der Tastatur, wartet aber solange, bis der Benutzer die -Taste drückt. Erst wenn *ReadLine()* abgeschlossen ist, terminiert das Programm und das Fenster der Eingabeaufforderung wird geschlossen.

Hinweis

Sie finden das Beispiel im Projektordner `\Beisp\Kap04\Beispiel04_01` der Begleit-CD. Die .NET-Anwendung wird im Fenster der Eingabeaufforderung ausgeführt.

4.1.2 Felder bei der Deklaration initialisieren

Sofern Sie den im vorhergehenden Beispiel gezeigten Ansatz zur Deklaration einer Feldvariablen verwenden, initialisiert Visual Basic die Feldvariablen mit Vorgabewerten, die vom jeweiligen Datentyp abhängen (siehe *Kapitel 3* im Abschnitt »Arbeiten mit Variablen«). Anschließend müssen die Werte den Feldelementen im Programm zugewiesen werden. Gerade bei den Wochentagen hätte es sich angeboten, die betreffenden Namen bei der Deklaration anzugeben. Bei Arrays muss die Deklaration mit anschließender Wertzuweisung in folgender Form erfolgen:

```
Dim Feldname () As Datentyp = {Wert, Wert, ...}
```

Wichtig ist dabei, dass keine Obergrenze für den Feldindex angegeben wird, d.h., die Klammer bleibt leer! Die eigentlichen Werte für die Feldelemente werden dann getrennt durch Kommata in geschweiften Klammern hinter dem Zuweisungszeichen aufgeführt. Der Visual-Basic-Compiler ermittelt bei der Übersetzung die Zahl der Initialisierungswerte und legt ein Feld in der betreffenden Größe an. Das folgende Listing zeigt das Beispiel mit den Wochentagen in der entsprechenden Darstellung

```
'*****
' File/Projekt: Beispiel04_02
' Autor: G. Born www.borncity.de
' Definiert eine Feldvariable und initialisiert diese.
' Listet die Werte anschließend in der Console auf.
'*****
Option Strict
Imports System.Console      ' für WriteLine
Class Test
    Shared Sub Main()        ' das ist das Hauptmodul
        Dim Wochentag () As String = _
            {"Sonntag", "Montag", "Dienstag", _
             "Mittwoch", "Donnerstag", _
             "Freitag", "Samstag"}

        WriteLine("Wochentage: {0} {1} {2} {3} {4} {5} {6}", _
            Wochentag)
        Write ("Bitte die Eingabetaste drücken")
        ReadLine
    End Sub
End Class
```

Listing 4.2: Beispiel mit Initialisierung der Feldwerte

Nach der Initialisierung enthält die Feldvariable *Wochentag* sieben Feldelemente, die über die Indexwerte 0 bis 6 angesprochen werden können. Die Ausgabe über *WriteLine()* hätte normalerweise folgendermaßen ausgesehen:

```
WriteLine("Wochentage: {0} {1} {2} {3} {4} {5} {6}", _  
        Wochentag(0), Wochentag(1), Wochentag(2), _  
        Wochentag(3), Wochentag(4), Wochentag(5), _  
        Wochentag(6))
```

Ich habe hier auf einen Zeilenumbruch nach jedem Namen verzichtet und Platzhalter {0}, {1} etc. für die sieben Wochentage im ersten Argument benutzt. Da es sich bei *Wochentag* um ein Feld handelt, lässt sich die ganze Anweisung aber auch kürzer schreiben:

```
WriteLine("Wochentage: {0} {1} {2} {3} {4} {5} {6}", Wochentag)
```

Als zweites Argument übergeben wir einfach den Namen der Feldvariable ohne die Klammern und den Indexwert. Dann übergibt Visual Basic alle Feldelemente an die betreffende Methode, die dann für die Ausgabe sorgt. Der Programmcode wird also wesentlich kompakter.

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap04\Beispiel04_02` auf der Begleit-CD. Das übersetzte Projekt wird als Konsolenanwendung ausgeführt.

Beispiel: Willkommensmeldung beim Windows-Start

In einem weiteren Beispiel soll jetzt das Wissen zu Feldern genutzt werden. Es ist eine Windows-Anwendung zu erstellen, die in Abhängigkeit vom Wochentag eine Begrüßungsmeldung in einem Dialogfeld anzeigt (Abbildung 4.2).



Abbildung 4.2: Persönliche Begrüßung des Benutzers

Mit den bisherigen Kenntnissen über Felder sowie etwas Wissen, wie der aktuelle Benutzername, das Datum und eventuell der Wochentag ermittelt werden kann, lässt sich so etwas realisieren. Das Feld mit den auszugebenden Sprüchen lässt sich mit folgender Anweisung definieren und gleich initialisieren:

```
Dim cSpruch () As String = { _  
    "Eh Kumpel, Sonntags wird geruht!", _  
    "Die Woche fängt aber bescheiden an, mach mal Pause", _  
    "Hallo Alter, ein Arbeitstag ist schon rum", _  
    "Klasse, wir haben ja bereits Mittwoch", _  
    "Prima, wie die Zeit vergeht, 's ist Donnerstag", _
```

```
"Thank god, it's Friday", _  
"Samstag, Du wolltest doch ausspannen?"}
```

Die Feldvariable *cSpruch* enthält im ersten Element den Text, der Sonntags erscheinen soll, im zweiten Element den Text für Montag und so weiter. Bei Bedarf lassen sich die Texte für die einzelnen Wochentage sehr einfach anpassen.

Nun müssen wir noch wissen, wie sich der aktuelle Datumswert ermitteln lässt. Aus diesem Wert können alle benötigten Informationen wie der Wochentag abgeleitet werden. Hierzu gibt es zwei Ansätze. Einmal stellt die Kompatibilitätsklasse *Microsoft.VisualBasic* die Funktion *Now()* bereit, die das aktuelle Datum samt Zeit liefert. Zudem kann das aktuelle Datum über den Namensraum *System* in der Klasse *System.DateTime* ermittelt werden. Sobald die Definition

```
Imports System.DateTime
```

im Programmkopf vereinbart wurde, lässt sich auf die betreffende Klasse zugreifen. Sie können beispielsweise eine Objektvariable des Typs *DateTime* anlegen:

```
Dim Datum As System.DateTime = Now
```

Die Variable *Datum* ist ein *DateTime*-Objekt vom Datentyp *DateTime*. Über diese Variable kann auf alle Eigenschaften und Methoden des *DateTime*-Objekts der Klasse *System.DateTime* zugegriffen werden. Die Eigenschaft *Now* der *DateTime*-Klasse liefert beispielsweise das aktuelle Datum samt Uhrzeit. In der obigen Anweisung wird nicht nur eine Datumsvariable deklariert, sondern gleich über die Eigenschaft *Now* mit dem aktuellen Datum initialisiert. Anschließend kann der Datumswert über die Methoden und Eigenschaften des *DateTime*-Objekts in seine Bestandteile zerlegt werden.

- *Day*: Diese Eigenschaft liefert den Tag des betreffenden Datumsobjekts zurück.
- *Month*: Liefert den Monat des angegebenen Datums zurück.
- *Year*: Liefert das Jahr zurück, wenn ein Datumswert übergeben wird.
- *DayOfWeek*: Liefert den Wochentag aus dem Datum als Zahl zwischen 0 und 6 für die Wochentage (0 = Sonntag, 1 = Montag etc.).

Um den Tag aus dem Datumsobjekt zu extrahieren ist die Anweisung *Datum.Day* zu verwenden. Die Namen der Wochentage lassen sich als Feld definieren:

```
Dim WeekDayName () As String = _  
    {"Sonntag", "Montag", "Dienstag",  
     "Mittwoch", "Donnerstag", _  
     "Freitag", "Samstag"}
```

Mit diesen Informationen lässt sich der gewünschte Text für die Ausgabe gemäß folgender Anweisung zusammenbauen:

```
Text = "Es ist heute " & WeekDayName(Datum.DayOfWeek) & _
      ", der " & Datum.Day & "." & _
      MonthName(Datum.Month) & " " & _
      Datum.Year
```

Bei `WeekDayName()` greifen wir über einen Index, der durch `Datum.DayOfWeek` geliefert wird, auf das Feldelement mit dem Namen des Wochentags zurück. Mit `Datum.Day` wird der Tag ermittelt, während `Datum.Month` den Monat zurückgibt. Dieser Wert wird als Index in `MonthName()` benutzt, um den Monatsnamen aus dem Feld zu holen. Der Ausdruck `Datum.Year` liefert das Jahr über die betreffende Eigenschaft `Year`.

Das Ergebnis wird dann mit dem Text *"Es ist heute "* über den `&`-Operator verknüpft und in die Textvariable `Text` gespeichert. Die Anzeige des Dialogfelds erfolgt in unserem Programm mittels der bereits mehrfach benutzen `MsgBox()`-Anweisung. Diese wird in Visual Basic 2005 standardmäßig über die Microsoft-Visual-Basic-Kompatibilitätsklasse bereitgestellt. Die `MsgBox()`-Methode kann dabei mit mehreren optionalen Parametern aufgerufen werden, wobei der erste Parameter den Dialogfeldtext, der zweite Parameter die Schaltflächen im Dialogfeld und der dritte Parameter den Titeltext spezifiziert. Mit der Anweisung

```
MsgBox (Text, vbOkOnly, Titel)
```

wird ein Meldungsfeld mit einer OK-Schaltfläche (definiert durch die Konstante `vbOkOnly`) erzeugt, welches den Inhalt von `Text` sowie die Titelzeile mit dem Wert von `Titel` anzeigt.

Bleibt noch die Frage, wie sich der Name des aktuell angemeldeten Benutzers in der Stringvariablen `Titel` eintragen lässt. Im Namensraum `System.Environment` der .NET-Framework-Klassenbibliothek findet sich die Eigenschaft `UserName`. Wird der Zugriff auf die Klasse im Programmkopf mit

```
Imports System.Environment
```

vereinbart, lässt sich der Benutzername folgendermaßen ermitteln:

```
Titel = "Willkommen " & UserName
```

Das folgende Listing zeigt den kompletten Quellcode des Beispiels:

```
'*****
' File/Projekt:  Welcome1
' Autor: G. Born www.borncity.de
' Erstellt eine Begrüßungsnachricht abhängig vom
' Wochentag und zeigt diese in einem Dialogfeld an.
'
' Verwendet die .NET-Framework-Klasse System.DateTime
```

Listing 4.3: Beispiel mit Zugriff auf die `System.DateTime`-Klasse

```
'*****
Option Strict

Imports System.Environment      ' für Benutzername
Imports System.DateTime        ' für Datum

Class Test
    Shared Sub Main()           ' das ist das Hauptmodul
        Dim Text, Titel As String
        Dim Datum As System.DateTime = Now ' Datumsobjekt mit
                                           ' aktuellem Datum

        ' Wochentage definieren
        Dim WeekDayName () As String = _
            {"Sonntag", "Montag", "Dienstag", _
             "Mittwoch", "Donnerstag", _
             "Freitag", "Samstag"}

        ' Hier ein Feld mit Sprüchen definieren
        Dim cSpruch () As String = { _
            "Eh Kumpel, Sonntags wird geruht!", _
            "Die Woche fängt aber bescheiden an, mach mal Pause", _
            "Hallo Alter, ein Arbeitstag ist schon rum", _
            "Klasse, wir haben ja bereits Mittwoch", _
            "Prima, wie die Zeit vergeht, 's ist Donnerstag", _
            "Thank god, it's Friday", _
            "Samstag, Du wolltest doch ausspannen?"}

        ' Ermittle den aktuellen Benutzernamen über die UserName-Eigenschaft
        ' der Klasse System.Environment und fülle Titel damit
        Titel = "Willkommen " & UserName

        ' Stelle jetzt den Text der Begrüßungsmeldung zusammen
        Text = "Es ist heute " & WeekDayName(Datum.DayOfWeek) & _
            ", der " & Datum.Day & "." & _
            MonthName(Datum.Month) & " " & _
            Datum.Year

        ' Hänge den "Motivator" an den Text an
        Text = Text & vbCrLf & vbCrLf & cSpruch(Datum.DayOfWeek)
```

Listing 4.3: Beispiel mit Zugriff auf die System.DateTime-Klasse (Forts.)

```
' Jetzt die Meldung als Dialogfeld ausgeben
  MsgBox (Text, vbOkOnly, Titel)
End Sub
End Class
```

Listing 4.3: Beispiel mit Zugriff auf die System.DateTime-Klasse (Forts.)**Hinweis**

Sie finden die Projektdateien im Ordner `\Beisp\Kap04\Welcome` der Begleit-CD. Das Projekt ist als Windows-Anwendung implementiert, bei der die Klasse *Test* (statt eines Formulars) als Startobjekt dient. Das hier gezeigte Beispiel befindet sich im Element *Test.vb*. Das im Projekt zusätzlich eingebundene Element *Welcome.vb* benutzt die Funktionen *Now()*, *Date()* etc. aus der bereits oben erwähnten Visual Basic-Kompatibilitätsklasse *Microsoft.VisualBasic*. Falls Sie von Visual Basic 6 umsteigen, können Sie ggf. den Quellcode inspizieren, um nachzusehen, wie sich die betreffenden Funktionen der Kompatibilitätsklasse nutzen lassen.

4.1.3 Mehrdimensionale Felder

Die Beispiele auf den vorhergehenden Seiten benutzen nur Felder mit einer Dimension. Visual Basic erlaubt aber Felder mit einer beliebigen Anzahl Dimensionen anzugeben (limitierend wirkt nur der verfügbare Speicher). Die Deklaration eines mehrdimensionalen Feldes erfolgt mit den gleichen Mechanismen wie bei eindimensionalen Feldern. Es muss lediglich für jede Dimension eine Obergrenze für den Feldindex oder ein Platzhalter angegeben werden. Die folgende Deklaration vereinbart ein zweidimensionales Feld, welches jeweils 2 x 2 Elemente aufweist:

```
Dim i(1,1) As Integer
```

Die Feldindizes lassen sich zu (0,0), (0,1), (1,0) und (1,1) angeben. Die Elemente dieses Feldes können individuell mit Werten belegt werden. Bei der Zuweisung sind nur die jeweiligen Feldindizes erforderlich. Die folgenden Anweisungen weisen den Feldelementen Werte zu:

```
i(0,0) = 0
i(0,1) = 1
i(1,0) = 2
i(1,1) = 3
```

Alternativ kann ein Feld folgendermaßen deklariert werden:

```
Dim j( , ) As Integer = {{0, 1, 2}, {3, 4, 5}}
```

Hier wird ebenfalls ein zweidimensionales Feld angelegt. Anstelle fester Indexobergrenzen werden nur Leerzeichen, getrennt durch Kommas in der Klammer hinterlegt. Die Festlegung der Feldgrenzen erfolgt bei dieser Deklaration durch die Initialisierungswerte, die in der gleichen Zeile zugewiesen werden. Die Werte einer Zeile sind dann durch zusätzliche geschweifte Klammern (z.B. {0, 1, 2}) einzufassen.

Die Verwendung von Feldern wird an einem kurzen Beispielprogramm demonstriert. Das Programm deklariert jeweils ein Feld *i* und ein Feld *j*, wobei beide Felder vom Typ *Integer* sind, also ganze Zahlen aufnehmen sollen. Beim Feld *i* werden die Indexobergrenzen fest vorgegeben. Die Zuweisung der Feldwerte erfolgt durch individuelle Anweisungen. Das Feld *j* wird als Feld ohne Indexwerte definiert und gleichzeitig mit Startwerten initialisiert. Die Startwerte sorgen dafür, dass der Visual-Basic-2005-Compiler ein Feld *j*(1,2) definiert. Das Beispielprogramm zeigt die Werte der definierten Felder im Fenster der Eingabeaufforderung an (Abbildung 4.3).



Abbildung 4.3: Ausgaben des Beispielprogramms mit den Feldwerten

Der Quellcode dieses sehr einfachen Beispiels ist dem folgenden Listing zu entnehmen:

```
'*****
' File/Projekt: Beispiel4_04
' Autor: G. Born www.borncity.de
' Definiert eine mehrdimensionale Feldvariable.
'*****
Option Strict
Imports System.Environment ' für newline-Eigenschaft
Imports System.Console    ' für WriteLine

Class Test
    Shared Sub Main() ' das ist das Hauptmodul
        Dim i(1,1) As Integer ' Felddefinition
        Dim j( , ) As Integer = {{0, 1, 2}, {3, 4, 5}}

        i(0,0) = 0
        i(0,1) = 1
        i(1,0) = 2
```

Listing 4.4: Beispiel mit zweidimensionalen Feldern


```

i(1,1) = 3

' Werte auf Konsoleebene anzeigen
WriteLine("Feld i(0,0): {0} i(0,1): {1}", i(0,0), i(0,1))
WriteLine("Feld i(1,0): {0} i(1,1): {1}", i(1,0), i(1,1))
WriteLine

WriteLine("Feld j(0,0): {0} j(0,1): {1} j(0,2): {2}" & _
        "{3}Feld j(1,0): {4} j(1,1): {5} j(1,2): {6}", _
        j(0,0), j(0,1), j(0,2), newline, j(1,0), j(1,1), j(1,2))

Write ("Bitte die Eingabetaste drücken")
ReadLine
End Sub
End Class

```

Listing 4.4: Beispiel mit zweidimensionalen Feldern (Forts.)

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap04\Beispiel04_04` auf der Begleit-CD. Das Beispiel läuft als Konsolenanwendung im Fenster der Eingabeaufforderung.

4.1.4 Anpassen der Feldgröße

Wenn Sie ein Feld mit der *Dim*-Anweisung vereinbaren und die Obergrenzen für die Feldindizes festlegen, reserviert der Compiler den entsprechenden Speicherplatz für die Feldelemente. Dies bedeutet aber auch, dass dynamische Felder nicht möglich sind. Müssen Sie während des Programmablaufs das Feld vergrößern oder verkleinern, ist hierzu die *ReDim*-Anweisung erforderlich. Diese Anweisung erzeugt ein neues Feld, wobei aber der gleiche Name für die Feldvariable benutzt werden kann. Die folgende Sequenz erzeugt ein Feld und dimensioniert es anschließend neu:

```

Dim i(4) As Integer
...
ReDim i(6)

```

Beim Redimensionieren von Feldern sind noch einige Besonderheiten zu beachten. Im vorherigen Abschnitt haben Sie mehrdimensionale Felder kennen gelernt. Das *ReDim* funktioniert nur bei eindimensionalen Feldern, bzw. bei mehrdimensionalen Feldern lässt sich nur der letzte Indexwert neu dimensionieren. Beim Ausführen der *ReDim*-Anweisungen gehen zudem die Inhalte aller Feldvariablen verloren. Möchten Sie den Inhalt eines Feldes erhalten, müssen Sie das *Preserve*-Schlüsselwort beim *ReDim* verwenden:

```
Dim i(4) As Integer
...
Redim Preserve i(6)
```

Dieses Verhalten soll jetzt an einem einfachen Beispiel demonstriert werden. Im Programm wird ein eindimensionales Feld *i* vereinbart und mit vier Integerwerten initialisiert. Dann gibt das Programm die Werte der Feldelemente sowie die Zahl der Elemente im Fenster der Eingabeaufforderung aus. Die Zahl der Feldelemente aller Dimensionen lässt sich direkt über die *Length*-Eigenschaft des Feldes (z.B. *i.Length*) abfragen. Bei einem eindimensionalen Feld lässt sich mit *i.Length-1* der größte Indexwert berechnen.

Um bei mehrdimensionalen Feldern die Zahl der Feldelemente für einen Index zu ermitteln, muss dagegen auf die *GetLength()*-Methode zurückgegriffen werden. Die Methode erwartet als Argument den nullbasierenden Wert der Dimension (bei einem zweidimensionalen Feld steht der Wert 0 z.B. für die erste Dimension). Um den oberen Indexwert einer Dimension zu ermitteln, lässt sich die *GetUpperBound()*-Methode anwenden. Auch hier muss der nullbasierte Wert der Dimension angegeben werden.

Hinweis

Für Umsteiger von früheren Visual-Basic-Versionen an dieser Stelle ein Hinweis. Felder werden, wie auch die Datentypen, direkt von der .NET-Framework-Klassenbibliothek als Objekte bereitgestellt. Da es sich bei Feldern um Objekte handelt, stellen diese Eigenschaften und Methoden bereit. Sie können also bei Feldern auf *Length* und *GetLength()* zugreifen.

Das Neudimensionieren eines Feldes mit und ohne Übernahme der alten Werte sowie die Anwendung der *Length*-Eigenschaft sowie der *GetLength()*-Methode wird an folgendem Listing demonstriert. In Abbildung 4.4 sehen Sie die Ausgaben des Programms.

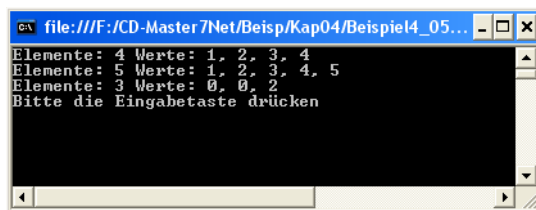


Abbildung 4.4: Ausgaben des Beispielprogramms beim Redimensionieren

```
'*****
' File/Projekt: Beispiel04_05
' Autor: G. Born www.borncity.de
' Definiert eine Feldvariable und weist dieser
' Werte zu. Listet die Werte in der Console auf.
' Demonstriert dabei, wie sich das Feld vergrößern
```

Listing 4.5: Redimensionieren von Feldern

```
' oder verkleinern lässt.
'*****
'Option Strict
Imports System.Console      ' für WriteLine

Class Test
    Shared Sub Main()        ' das ist das Hauptmodul
        Dim i() As Integer = {1, 2, 3, 4} ' Felddefinition

        ' Werte auf Konsoleebene anzeigen, verwende Length-Eigenschaft
        WriteLine("Elemente: {0} Werte: {1}, {2}, {3}, {4} ", _
            i.Length, i(0), i(1), i(2), i(3))

        ' jetzt Feld um ein Element ergänzen
        Redim Preserve i(4)
        i(4) = 5                ' letzter Wert neu belegen

        ' Werte auf Konsoleebene anzeigen, verwende GetLength()-Methode
        WriteLine("Elemente: {0} Werte: {1}, {2}, {3}, {4}, {5}", _
            i.GetLength(0), i(0), i(1), i(2), i(3), i(4))

        ' jetzt Feld auf 3 Element kürzen
        Redim i(2)
        i(2) = 2                ' letzter Wert neu belegen

        ' Werte auf Konsoleebene anzeigen
        WriteLine("Elemente: {0} Werte: {1}, {2}, {3}", _
            i.GetUpperBound(0)+1, i(0), i(1), i(2))

        Write ("Bitte die Eingabetaste drücken")
        ReadLine
    End Sub
End Class
```

Listing 4.5: Redimensionieren von Feldern (Forts.)

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap04\Beispiel4_05` auf der Begleit-CD. Nach dem Übersetzen können Sie das Beispiel als Konsolenanwendung im Fenster der Eingabeaufforderung ausführen.

4.2 Strukturen für benutzerdefinierte Datentypen

Auf den vorhergehenden Seiten wurden Beispiele zum Deklarieren von Variablen und Feldern mit bestimmten Datentypen vorgestellt. Allen diesen Beispielen gemeinsam ist die Tatsache, dass eine Variable oder ein Feld immer nur Werte eines Datentyps aufnehmen kann. Gelegentlich ist es aber erwünscht, komplexe Daten in einer Datenstruktur zu hinterlegen. Nehmen wir beispielsweise die Datenstruktur zur Speicherung von Personendaten, die einen Namen, Vornamen, ein Geburtsdatum etc. aufweist. Schön wäre es, einen Datentyp für die Aufnahme der Personendaten definieren und später Variablen zuweisen zu können. Dies ist in Visual Basic mit der *Structure*-Anweisung möglich.

Achtung

Umsteiger von VBA, VBScript oder älteren Visual-Basic-Versionen kennen vielleicht die *Type*-Anweisung, mit der sich benutzerdefinierte Datentypen vereinbaren ließen. Diese *Type*-Anweisung gibt es jetzt nicht mehr. Vielmehr wird das Schlüsselwort *Structure* zur Vereinbarung der Datenstruktur benutzt.

4.2.1 Definition einer Datenstruktur mit Structure

Die *Structure*-Anweisung lässt sich auf Modul- oder Klassenebene verwenden, um eine Struktur zu deklarieren. Die Struktur besteht aus einzelnen (als Member bezeichneten) Mitgliedern, die bestimmte Eigenschaften aufweisen. Nehmen wir einmal das Beispiel einer Struktur zur Aufnahme von Personaldaten. Diese ließe sich folgendermaßen definieren:

```
Public Structure Personaldaten
    Dim Name As String
    Dim Vorname As String
    Dim Plz As String
    Dim Ort As String
    Dim Strasse As String
    Dim Hausnr As String
    Dim Gehalt As Single
End Structure
```

Die Struktur vereinbart einen eigenen Datentyp *Personaldaten*, der mehrere Einträge für Name, Vorname etc. enthält. Diese Einträge (Member) können wiederum mit Datentypen wie *String*, *Integer* etc. deklariert werden.

Um eine auf diesem benutzerdefinierten Datentyp basierende Variable anzulegen, ist dann noch eine zweite Anweisung der Form:

```
Dim a As Personaldaten
```

erforderlich. Anschließend existiert die Variable *a* mit den in der Struktur vereinbarten Mitgliedern. Um den Namen des Angestellten zu setzen, ist folgende Anweisung zulässig:

```
a.Name = "Born"
```

Dies bedeutet, Sie müssen für den Zugriff auf die Member der Struktur jeweils den Variablennamen sowie den Namen des Members, getrennt durch einen Punkt, mit angeben. Natürlich können Sie auch eine Feldvariable auf Basis eines benutzerdefinierten Datentyps deklarieren.

```
Dim a (3) As Personaldaten
```

Die obige Anweisung erzeugt ein Feld mit drei Feldelementen *a* (0) bis *a* (2), in denen die Daten der Struktur abgelegt werden. Ein Zugriff auf den dritten Namen ist dann mit

```
a(2).Name = "Born"
```

möglich. Strukturen bilden also ein sehr mächtiges Hilfsmittel, um komplexere Daten in Variablen abzubilden. Dies soll jetzt an einem sehr einfachen Visual-Basic-Beispiel demonstriert werden. Dieses Beispiel verwendet die obige Struktur, um zwei Variablen *a* und *b* mit diesem Typ zu belegen. Zusätzlich wird eine Feldvariable *c* mit drei Elementen vereinbart und auf diesen Typ gesetzt. Anschließend sind die Inhalte der einzelnen Variablen mit Werten zu belegen. Dies geschieht teilweise durch einfache Zuweisungen der Art

```
a.Name = "Born"  
a.Vorname = "Klaus"  
a.Plz = "10230"
```

Der Inhalt einer Variablen lässt sich zudem mit einer einfachen Zuweisung einer zweiten Variablen gleichen Typs zuweisen. Die Anweisung

```
b = a
```

bewirkt, dass der Inhalt der Variablen *a* in die Variable *b* kopiert wird. Anschließend lassen sich die Inhalte der einzelnen Member von *b* folgendermaßen überschreiben:

```
b.Vorname = "Ulrike"  
b.Gehalt = 2100.00
```

Alle Member, die nicht überschrieben werden, behalten die Werte, die beim Kopieren von *a* zugewiesen wurden. Das Programm zeigt anschließend die Werte der verschiedenen Variablen im Fenster der Eingabeaufforderung an (Abbildung 4.5).

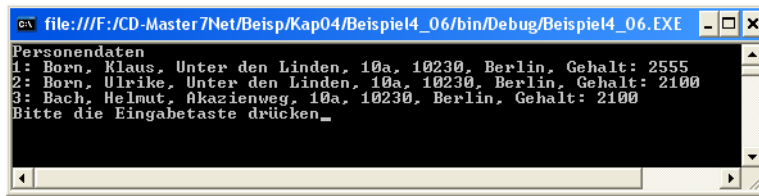


Abbildung 4.5: Anzeige des Personendaten

Das Beispielprogramm definiert ein Feld *c* mit drei Feldelementen, wobei aber nur ein Feldelement *c(2)* benutzt wird. Die einzelnen Anweisungen zum Deklarieren der Struktur und der Variablen sowie die Befehle zum Zuweisen der Werte an die Member der Struktur und die anschließende Ausgabe können Sie dem folgenden Listing entnehmen:

```
'*****
' File/Projekt:  Beispiel04_06
' Autor: G. Born www.borncity.de
' Definiert eine Struktur und initialisiert
' Variable mit dieser Struktur. Weist dann Werte
' zu und listet die in der Console auf.
'*****

Option Strict
Imports System.Console      ' für WriteLine

Class Test

' Vereinbare die Struktur für die Personaldaten
Public Structure Personaldaten
    Dim Name As String
    Dim Vorname As String
    Dim Plz As String
    Dim Ort As String
    Dim Strasse As String
    Dim Hausnr As String
    Dim Gehalt As Single
End Structure

Shared Sub Main()           ' das ist das Hauptmodul
    Dim a, b As Personaldaten
    Dim c(3) As Personaldaten

    a.Name = "Born"          ' erster Datensatz
    a.Vorname = "Klaus"
```

Listing 4.6: Beispiel zur Verwendung von Strukturen

```

a.Plz = "10230"
a.Ort = "Berlin"
a.Strasse = "Unter den Linden"
a.Hausnr = "10a"
a.Gehalt = 2555.00

b = a                                ' erster Datensatz in 2. kopieren
b.Vorname = "Ulrike"                 ' einzelne Werte anpassen
b.Gehalt = 2100.00

c(2) = b                             ' kopiere in Feld
c(2).Name = "Bach"
c(2).Vorname = "Helmut"
c(2).Strasse = "Akazienweg"

' Werte 1. Datensatz auf Konsoleebene anzeigen
WriteLine("Personendaten")
WriteLine("1: {0}, {1}, {2}, {3}, {4}, {5}, Gehalt: {6}", _
    a.Name, a.Vorname, a.Strasse, a.Hausnr, _
    a.Plz, a.Ort, a.Gehalt)

' Werte 2. Datensatz auf Konsoleebene anzeigen
WriteLine("2: {0}, {1}, {2}, {3}, {4}, {5}, Gehalt: {6}", _
    b.Name, b.Vorname, b.Strasse, b.Hausnr, _
    b.Plz, b.Ort, b.Gehalt)

' Werte 3. Datensatz auf Konsoleebene anzeigen
WriteLine("3: {0}, {1}, {2}, {3}, {4}, {5}, Gehalt: {6}", _
    c(2).Name, c(2).Vorname, c(2).Strasse, c(2).Hausnr, _
    c(2).Plz, c(2).Ort, c(2).Gehalt)
Write ("Bitte die Eingabetaste drücken")
ReadLine
End Sub
End Class

```

Listing 4.6: Beispiel zur Verwendung von Strukturen (Forts.)

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap04\Beispiel4_06` auf der Begleit-CD. Nach dem Übersetzen können Sie das Beispiel als Konsoleanwendung im Fenster der Eingabeaufforderung ausführen.

4.2.2 Arbeiten mit Nullable-Wertetypen

Bei der Definition von Variablen kommt es insbesondere bei Datenbank Anwendungen vor, dass diese keinen Wert (Null) aufweisen. Um solche Fälle abbilden zu können, wurde in .NET Framework 2.0 eine neue Klasse im Namensraum *System* eingeführt, die Nullwerte (Nullable) unterstützt. Nehmen wir das obige Beispiel mit der Struktur zur Abbildung von Personaldatensätzen. Der Eintrag *Gehalt* macht nur dann einen Sinn, wenn die betreffende Person auch ein Gehalt bezieht. Anstatt jetzt den Wert 0.0 für ein fehlendes Gehalt einzugeben, könnten Sie den Wert auch auf »Null« setzen. Die modifizierte Definition der Struktur zur Aufnahme der Personaldaten sieht folgendermaßen aus:

```
Public Structure Personaldaten
    Dim Name As String
    Dim Vorname As String
    Dim Plz As String
    Dim Ort As String
    Dim Strasse As String
    Dim Hausnr As String
    Dim Gehalt As Nullable(Of Single)
End Structure
```

Der Member *Gehalt* ist weiterhin vom Typ *Single*, wurde jetzt aber als *Nullable* definiert. Nehmen wir an, es wurde eine entsprechende Variable *b* vom Typ *Personaldaten* definiert. Dann lässt sich dem Member *Gehalt* mittels der Anweisung

```
b.Gehalt = Nothing
```

ein Nullwert zuweisen. Die Prüfung, ob der betreffende Member der Variablen einen Wert aufweist, lässt sich mit der *IsNothing()*-Funktion (bzw. Methode) erreichen. Die Angabe *IsNothing(b.Gehalt)* liefert den Wert *False*, falls der Member einen Wert aufweist. Ist der Wert des Members auf *Nothing* gesetzt, liefert *IsNothing()* den Wert *True* zurück.

Dies wurde in einem kleinen Beispielprogramm genutzt, welches zwei Personaldatensätze in den Variablen *a* und *b* vereinbart. Alle Member der Variablen *a* werden mit Werten initialisiert. Danach wird die Variable *a* der Variablen *b* zugewiesen. Der Member *Vorname* der Variablen *b* ist anschließend zu ändern und der Member *Gehalt* soll keinen Wert aufweisen. Abbildung 4.6 zeigt die modifizierte Ausgabe auf der Konsole.

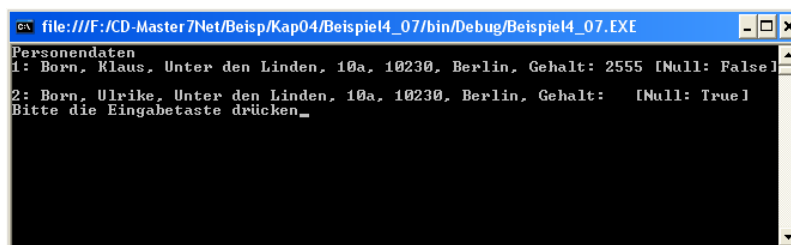


Abbildung 4.6: Anzeige des Personendaten und Auswertung des Gehalts mit *IsNothing*

Bei jedem Datensatz wird am Zeilenende noch angegeben, ob der Member *Gehalt* einen Wert aufweist oder nicht. Der Wert *Gehalt* ist nur beim ersten Datensatz mit einem Wert belegt (und *IsNothing()* liefert den Wert *false*), während der betreffende Eintrag beim zweiten Datensatz leer bleibt. Das Programm wurde aus dem Code des vorherigen Beispiels abgeleitet. Dem Member *b.Gehalt* wird nach dem Kopieren mit der Anweisung

```
b.Gehalt = Nothing
```

ein Nullwert zugewiesen. Zudem wurde die Konsolenausgabe so modifiziert, dass über *IsNothing* ein Hinweis erfolgt, ob ein Gehalt als Wert vorliegt. Details zum Programmcode können Sie dem folgenden Listing entnehmen:

```
'*****
' File/Projekt:  Beispiel04_07
' Autor: G. Born www.borncity.de
' Definiert eine Struktur und initialisiert
' Variable mit dieser Struktur. Weist dann Werte
' zu und listet die in der Console auf.
' Das Beispiel demonstriert die Verwendung von
' Null-Werten.
'*****
Option Strict On
Imports System.Console      ' für WriteLine

Class Test
    Public Structure Personaldaten
        Dim Name As String
        Dim Vorname As String
        Dim Plz As String
        Dim Ort As String
        Dim Strasse As String
        Dim Hausnr As String
        Dim Gehalt As Nullable(Of Single)
    End Structure

    Shared Sub Main()          ' das ist das Hauptmodul
        Dim a As Personaldaten
        Dim b As Personaldaten

        a.Name = "Born"        ' erster Datensatz
        a.Vorname = "Klaus"
        a.Plz = "10230"
```

Listing 4.7: Beispiel zur Verwendung von Strukturen

```
a.Ort = "Berlin"
a.Strasse = "Unter den Linden"
a.Hausnr = "10a"
a.Gehalt = 2555.0

b = a                                ' erster Datensatz in 2. kopieren
b.Vorname = "Ulrike"                 ' einzelne Werte anpassen
b.Gehalt = Nothing                   ' erhält einen Nullwert

' Werte 1. Datensatz auf Konsoleebene anzeigen
WriteLine("Personendaten")
WriteLine("1: {0}, {1}, {2}, {3}, {4}, {5}, " & _
          Gehalt: {6} [Null: {7}]", _
          a.Name, a.Vorname, a.Strasse, a.Hausnr, _
          a.Plz, a.Ort, a.Gehalt, IsNothing(a.Gehalt))
' Werte 2. Datensatz auf Konsoleebene anzeigen
WriteLine("2: {0}, {1}, {2}, {3}, {4}, {5}, " & _
          Gehalt: {6} [Null: {7}]", _
          b.Name, b.Vorname, b.Strasse, b.Hausnr, _
          b.Plz, b.Ort, b.Gehalt, IsNothing(b.Gehalt))

Write("Bitte die Eingabetaste drücken")
ReadLine()

End Sub
End Class
```

Listing 4.7: Beispiel zur Verwendung von Strukturen (Forts.)

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap04\Beispiel4_07` auf der Begleit-CD. Nach dem Übersetzen können Sie das Beispiel als Konsolenanwendung im Fenster der Eingabeaufforderung ausführen.

Damit möchte ich das Kapitel über den Umgang mit Feldern und Strukturen schließen. Das .NET Framework bietet eine Reihe zusätzlicher Möglichkeiten, um mit Feldern zu arbeiten oder Daten über Klassen zu strukturieren. Diese Fragen werden im Laufe der folgenden Kapitel behandelt.

5

Zuweisungen, Operatoren und mehr

Das vorliegende Kapitel zeigt, wie sich Zuweisungen und Operatoren in Visual Basic 2005 nutzen lassen. Zudem erfahren Sie, wie Typkonvertierungen oder Operationen auf Zeichenketten und Feldern in Visual Basic 2005 auszuführen sind.

5.1 Zuweisungen in Visual Basic 2005

Um in Visual Basic 2005 einer Variablen einen Wert zuzuweisen, wird der Zuweisungsoperator verwendet. Microsoft unterscheidet in Visual Basic 2005 dabei einfache Zuweisungsanweisungen und sogenannte Verbundzuweisungsanweisungen. Schauen wir uns einmal an, was sich dahinter verbirgt und wie sich so etwas nutzen lässt.

5.1.1 Einfache Zuweisungen

Einfache Zuweisungen haben Sie bereits mehr oder weniger kommentarlos in den bisherigen Beispielen genutzt. Hierbei wird das Ergebnis eines Ausdrucks, der auf der rechten Seite des Gleichheitszeichens steht, der Variablen auf der linken Seite des Gleichheitszeichens zugewiesen. Das Gleichheitszeichen fungiert dabei als Zuweisungsoperator. Die Anweisungen

```
Dim a As Integer  
a = 100
```

deklarieren eine Variable *a* vom Typ *Integer*. Anschließend wird der Variablen *a* in der zweiten Zeile der Wert 100 zugewiesen. Nichts Aufregendes, die zweite Zeile stellt die allgemeine Form einer Zuweisung dar. Auch innerhalb einer Deklaration sind Zuweisungen möglich:

```
Dim Wochentag () As String = {"Montag", "Dienstag", "Mittwoch"}
```

Die obige Anweisung benutzt den Zuweisungsoperator, um das Feld mit drei Werten zu initialisieren.

5.1.2 Verbundzuweisungen

Verbundzuweisungen erlauben die Zuweisung mit den Operatoren $\wedge$, $*$, $/$, $\backslash$, $+$, $-$ und $\&$ zu kombinieren. Schauen wir uns einmal die einfachste Variante, die Kombination des $+$ -Operators mit der Zuweisung an. Nehmen wir einmal an, eine Variable *a* sei folgendermaßen deklariert:

```
Dim a As Integer = 1
```

Die Variable *a* ist vom Typ *Integer* und weist den Initialisierungswert 1 auf. Um den Wert der Variablen *a* um 1 zu erhöhen, lässt sich die Anweisung

```
a = a + 1
```

verwenden. Dies ist eine einfache Zuweisung, die das Ergebnis des Ausdrucks auf der rechten Seite der Zuweisung der Variablen zuordnet. Es gibt aber auch eine verkürzte Schreibweise der Art

```
a += 1
```

Diese Anweisung besagt, dass der aktuelle Inhalt der Variablen *a* mit dem Wert 1 zu erhöhen ist. Das Ergebnis soll dann wieder der Variablen *a* zugewiesen werden. Es wird als der *+*-Operator mit dem Zuweisungsoperator kombiniert.

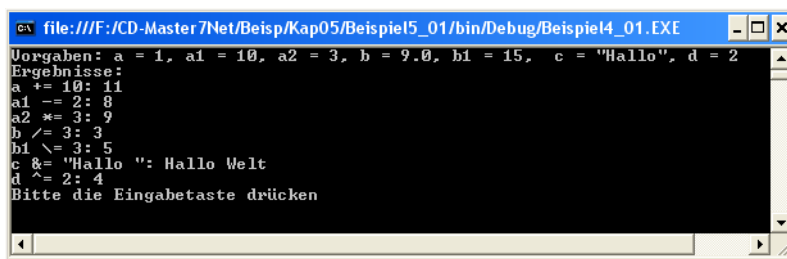


Abbildung 5.1: Ergebnisse des Beispielprogramms

Die folgenden Anweisungen zeigen die Verwendung dieser Verbundzuweisungen:

```
Dim a As Integer = 1
Dim b As Single = 9
Dim c As String = "Hallo "
a += 10           ' a = a + 10
a -= 2           ' a = a - 2
a *= 3           ' a = a * 3
b /= 3           ' b = b / 3 (Gleitkomma-division)
a \= 3           ' a = a \ 3 (Ganzzahl-division)
c &= "Welt"      ' c = c & "Welt" (Zeichenkettenverknüpfung)
a ^= 10          ' a = a ^10 (Potenzierung)
```

Visual Basic wendet bei allen Operationen den Operanden, der vor dem Gleichheitszeichen steht, auf den aktuellen Wert der Variablen sowie auf den Ausdruck rechts vom Gleichheitszeichen an. Die erste Anweisung *a += 10* sollte das Ergebnis 11 liefern. Sie können die Wirkung der einzelnen Verbundzuweisungen an folgendem Beispiel studieren, welches die in Abbildung 5.1 gezeigten Ausgaben erzeugt:

```

'*****
' File/Projekt: Beisp05_01.vb
' Autor: G. Born www.borncity.de
' Definiert einige Variablen und zeigt die Anwendung
' der Verbundzuweisungen.
'*****

Option Strict On
Imports System.Console      ' für WriteLine
Class Test
    Shared Sub Main()        ' das ist das Hauptmodul
        Dim a As Integer = 1
        Dim a1 As Integer = 10
        Dim a2 As Integer = 3
        Dim b As Single = 9.0
        Dim b1 As Integer = 15
        Dim c As String = "Hallo "
        Dim d As Double = 2.0

        a += 10              ' a = a + 10
        a1 -= 2              ' a1 = a1 - 2
        a2 *= 3              ' a2 = a2 * 3
        b /= 3               ' b = b / 3 (Gleitkomma-division)
        b1 \= 3              ' b1 = b1 \ 3 (Ganzzahl-division)
        c &= "Welt"          ' c = c & "Welt" (Zeichenkettenverknüpfung)
        d ^= 2               ' d = d ^2 (Potenzierung)

        WriteLine("Vorgaben: a = 1, a1 = 10, a2 = 3, " & _
            "b = 9.0, b1 = 15, c = ""Hallo"", d = 2")
        WriteLine("Ergebnisse:")
        WriteLine("a += 10: {0}", a)
        WriteLine("a1 -= 2: {0}", a1)
        WriteLine("a2 *= 3: {0}", a2)
        WriteLine("b /= 3: {0}", b)
        WriteLine("b1 \= 3: {0}", b1)
        WriteLine("c &= ""Hallo """: {0}", c)
        WriteLine("d ^= 2: {0}", d)

        Write ("Bitte die Eingabtaaste drücken")
        ReadLine
    End Sub
End Class

```

Listing 5.1: Beispiel mit Verbundzuweisungen

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap05\Beispiel5_01` auf der Begleit-CD. Das Programm wird als Konsolenanwendung übersetzt und anschließend im Fenster der Eingabeaufforderung ausgeführt.

5.1.3 Typkonvertierung bei Zuweisungen

Bei Zuweisungen muss das Ergebnis des Ausdrucks auf der rechten Seite des Gleichheitszeichens immer dem Datentyp der auf der linken Seite stehenden Variablen entsprechen. Lediglich Variablen vom Typ *Object* kann jeder beliebige Wert zugewiesen werden, Visual Basic 2005 speichert dann den Wert so, dass er dem Datentyp entspricht.

Standardmäßig führt Visual Basic bei nicht passendem Datentyp eine automatische Typumwandlung durch. Wenn Sie also einer Integervariablen einen Dezimalwert zuweisen, werden die Nachkommastellen des Dezimalwerts abgeschnitten, bevor die Zuweisung erfolgt. Die implizite Typkonvertierung ist zwar recht komfortabel, bietet aber das Risiko von Fehlern, die sich nur schlecht aufdecken lassen. Daher sollten Sie gemäß meiner Empfehlung die Anweisung *Option Strict On* im Programmkopf (oder in den Projekteinstellungen bzw. global in den Optionen der Entwicklungsumgebung) hinterlegen. Dann führt Visual Basic intern keine Typkonvertierung durch, sondern diese ist explizit im Programm durchzuführen. Hierzu stellt Visual Basic 2005 verschiedene Funktionen zur Typkonvertierung gemäß folgender Tabelle bereit.

Schlüsselwort	Bemerkung
CBool	Ausdruck in einen logischen Wert (<i>Boolean</i>) wandeln
CByte	Ausdruck in einen Bytewert (<i>Byte</i>) wandeln
CChar	Ausdruck in einen Character-Wert (<i>Char</i>) wandeln
CDate	Ausdruck in einen Datumswert (<i>Date</i>) wandeln
CDec	Ausdruck in eine Dezimalzahl (<i>Decimal</i>) wandeln
Cdbl	Ausdruck in eine Kommazahl doppelter Genauigkeit (<i>Double</i>) wandeln
CInt	Ausdruck in eine Integerzahl (<i>Integer</i>) wandeln
CLng	Ausdruck in einen Wert vom Typ <i>Long</i> wandeln
CObj	Ausdruck in einen Wert vom Typ <i>Object</i> wandeln
CShort	Ausdruck in einen Wert vom Datentyp <i>Short</i> wandeln
CStr	Ausdruck in einen Wert vom Typ <i>String</i> wandeln
CSng	Ausdruck in einen Wert vom Typ <i>Single</i> wandeln

Tabelle 5.1: Schlüsselwörter zur Typkonvertierung

Zusätzlich erlaubt Visual Basic noch die beiden Funktionen *CType* und *DirectCast*, die als ersten Parameter den Ausdruck und als zweiten Parameter den Zieldatentyp aufnehmen.

Hinweis

Konvertierungen mit den Schlüsselwörtern schlagen fehl, wenn keine Umwandlung des Ausdrucks in den Zieldatentyp möglich ist. Der Unterschied zwischen *CType* und *DirectCast* besteht darin, dass sich *CType* verwenden lässt, sobald eine gültige Konvertierung zwischen Ausdruck und Datentyp definiert ist. Bei *DirectCast* muss der Laufzeittyp einer Objektvariablen mit dem angegebenen Typ übereinstimmen. In diesem Fall bietet *DirectCast* zur Laufzeit etwas bessere Leistungen als *CType*. Einen Ausweg bietet die neu in Visual Basic 2005 eingeführte *TryCast*()-Methode. Diese lässt sich wie *DirectCast* anwenden (z.B. *h = TryCast("23.1", Integer)*). Geht die Typumwandlung bei der Objektvariablen schief, lautet das zurückgegebene Ergebnis auf »Nothing«.

Das folgende Beispielprogramm zeigt, wie sich die Konvertierung zwischen verschiedenen Datentypen bewältigen lässt:

```
'*****
' File/Projekt: Beispiel05_02
' Autor: G. Born www.borncity.de
' Definiert einige Variable und zeigt die Anwendung
' von Typkonvertierungen.
'*****

Option Strict On
Imports System.Console      ' für WriteLine

Class Test
    Shared Sub Main()        ' das ist das Hauptmodul
        Dim a As Integer
        Dim h As Integer
        Dim o As Object = 23
        Dim b As Byte
        Dim c As Boolean
        Dim d As String
        Dim e As Date
        Dim f As Double
        Dim g As Integer
        Dim a, h As Integer
        Dim o As Object = 23

        b = CByte(100.3)
        c = CBool(-1)
        d = CStr(True)
        e = CDate("1.1.2002")
```

Listing 5.2: Beispiele zur Typkonvertierung


```
f = CDb1(100)
g = CInt(23.56)

a = CType(o, Integer)
h = DirectCast(o, Integer)

' schlägt fehl, da Ausdruck Double, Ziel aber Integer ist
'Dim o1 As Object = "23.1"
'h = DirectCast(o1, Integer)
'h = DirectCast("23.1", Integer)

WriteLine("{0}, {1}, {2}, {3}, {4}, {5}, {6}", _
          a, h, b, c, d, e, f, g)
Write("Bitte die Eingabetaste drücken")
ReadLine()
End Sub
End Class
```

Listing 5.2: Beispiele zur Typkonvertierung (Forts.)**Hinweis**

Sie finden die Projektdateien für diese Konsolenanwendung im Ordner `\Beisp\Kap05\Beispiel5_02` auf der Begleit-CD.

5.2 Operatoren in Visual Basic 2005

Operatoren werden in Visual Basic 2005 eingesetzt, um bestimmte Auswertungen auf Ausdrücke durchzuführen. Die einfache Addition zweier Werte, die wir in den obigen Beispielen benutzt haben, stellt bereits einen Operator dar. In Visual Basic 2005 lassen sich die Operatoren in verschiedene Kategorien klassifizieren:

- Arithmetische Operatoren (mathematische Berechnungen)
- Logische Operatoren (logische Vergleiche)
- Relationale Operatoren (Vergleichsoperatoren)
- Verkettungs- und Zeichenkettenoperatoren (Verketten von Zeichenfolgen)

Die nachfolgenden Abschnitte enthalten eine grobe Übersicht über die jeweiligen Operatoren und erläutern deren Einsatz an Beispielen.

5.2.1 Arithmetische Operatoren

Arithmetische Operatoren erlauben mathematische Operationen wie Addition, Subtraktion etc. auf Werten. Die folgende Tabelle enthält eine Auflistung dieser Operatoren.

Operator	Erklärung
$\wedge$	Potenzieren ($x = y^{\text{Exponent}}$)
+	Addition ($x = a + b$)
-	Subtraktion oder negatives Vorzeichen ($x = -10$ oder $x = a - 100$)
*	Multiplikation ($x = b * 30$)
/	Gleitkomma-Division ($x = a / b$)
\	Integer-Division ($x = a \backslash b$)
Mod	Modulo, Rest aus einer Division ($x = a \text{ Mod } b$)

Tabelle 5.2: Arithmetische Operatoren

Die arithmetischen Operatoren sind auf den numerischen Datentypen wie *Byte*, *Short*, *Integer*, *Long*, *Single*, *Double* und *Decimal* definiert. Das Ergebnis muss aber im für den Datentyp darstellbaren Bereich liegen.

Hinweis

Der +-Operator lässt sich auch zur Verknüpfung von Zeichenketten einsetzen (z. B. *Text* = "Hallo " + "Welt"). Diesen Ansatz sollten Sie aber vermeiden, da die Operation nicht eindeutig ist. Die Anweisung *Zahl* = "12" + "13" könnte als Ergebnis die Zeichenkette "1213" oder den Wert 25 ergeben. Dies hängt davon ab, ob eine implizite Typkonvertierung zulässig ist und welcher Datentyp für *Zahl* vorgesehen ist. Zur Verkettung von Strings sollten Sie daher immer den &-Operator anwenden, da dann die Absicht eindeutig ist.

Sie finden die Projektdateien des Beispiels, welches die Anwendung einiger dieser Operatoren demonstriert, im Ordner \Beisp\Kap05\Beispiel5_03 auf der Begleit-CD.

5.2.2 Logische Operatoren

Logische Operatoren lassen sich auf dem Datentyp *Boolean* oder auf Typen wie *Byte*, *Short*, *Integer* und *Long* als bitweise Operation durchführen. Die folgende Tabelle enthält eine Aufstellung aller in Visual Basic 2005 definierten logischen Operatoren.

Operator	Bedeutung
Not	<i>Negation</i> ($x = \text{Not } y$). Der Operator dreht den Wert oder die einzelnen Bits einfach um. Aus einem Wert <i>true</i> wird der Wert <i>false</i> und aus <i>false</i> wird <i>true</i> . Bei Binärwerten werden die Bits mit dem Wert 0 zu 1 und umgekehrt.

Operator	Bedeutung
And	<i>Und</i> ($x = a \text{ And } b$). Der Operator führt eine Und-Verknüpfung zwischen den Werten durch. Die Ergebnisse dieser Verknüpfung lassen sich folgendermaßen angeben: False And False = False False And True = False True And False = False True And True = True
Or	<i>Oder</i> ($x = a \text{ Or } b$). Der Operator führt eine Oder-Verknüpfung zwischen den Werten durch. Die Ergebnisse dieser Verknüpfung lassen sich folgendermaßen angeben: False Or False = False False Or True = True True Or False = True True Or True = True
XOr	<i>Exklusiv-Oder</i> ($x = a \text{ XOr } b$). Der Operator führt eine Exclusive-Oder-Verknüpfung zwischen den Werten durch. Die Ergebnisse dieser Verknüpfung lassen sich folgendermaßen angeben: False XOr False = False False XOr True = True True XOr False = True True XOr True = False
AndAlso	Wertet einen zweiten Ausdruck nur aus, wenn der erste Ausdruck das Ergebnis <i>True</i> liefert. Liefert <i>True</i> zurück, wenn beide Teilausdrücke <i>True</i> sind. True AndAlso True = True True AndAlso False = False False AndAlso --- = False In der letzten Zeile wird der 2. Ausdruck nicht mehr ausgewertet, da der 1. Ausdruck <i>False</i> ist, d.h. der Operator liefert <i>False</i> .
OrElse	Liefert automatisch das Ergebnis <i>True</i> , sobald der erste Teilausdruck <i>True</i> ist. Liefert <i>True</i> zurück, wenn beide Teilausdrücke <i>True</i> sind. False OrElse False = False False OrElse True = True True OrElse --- = True In der letzten Zeile wird der 2. Ausdruck nicht mehr ausgewertet, da der 1. Ausdruck <i>True</i> ist, d.h. der Operator liefert <i>True</i> .

Tabelle 5.3: Logische Operatoren

Logische Operatoren lassen sich für Binäroperationen (Bitvergleiche) einsetzen. Oft werden diese Operatoren aber in *If*-Anweisungen benutzt, um mehrere Bedingungen zu vergleichen. Die Anweisung

```
If ((a > 100) And (a < 1000)) Then
```

führt zwei Vergleiche auf ($a > 100$) und ($a < 1000$) durch. Beide Vergleiche können einen Wert *true* oder *false* liefern. Die *Und*-Verknüpfung führt die beiden Ergebnisse zusammen. Nur wenn beide Ergebnisse zutreffen (a liegt im Bereich zwischen 101 und 999), wird der auf die *If*-Anweisung folgende Aktionsteil ausgeführt.

5.2.3 Relationale Operatoren

Hierbei handelt es sich um Vergleichsoperatoren, die den Vergleich von Ausdrücken (die auch Zahlen und Zeichenketten beinhalten können) erlauben. Die nachfolgende Tabelle enthält die in Visual Basic 2005 verfügbaren relationalen Operatoren.

Operator	Erklärung
<	Kleiner als ($a < b$)
>	Größer als ($a > b$)
=	Gleich ($a = b$)
<=	Kleiner oder gleich ($a \leq b$)
>=	Größer oder gleich ($a \geq b$)
<>	Ungleich ($a \neq b$)

Tabelle 5.4: Relationale Operatoren

Die relationalen Operatoren wie <, >, =, <>, <= und >= sind für die Datentypen *Byte*, *Short*, *Integer* und *Long* definiert und vergleichen die numerischen Werte der zwei Ganzzahloperanden. Bei den Datentypen *Single* und *Double* werden die relationalen Operatoren gemäß den Regeln nach Standard IEEE 754 verglichen. Beim Datentyp *Decimal* vergleichen die Operatoren die numerischen Werte der beiden Dezimaloperanden.

Die Operatoren = und <> sind zudem für den Datentyp *Boolean* definiert. Der Operator = liefert *True* zurück, wenn beide Operanden *True* oder wenn beide *False* sind. Der Operator <> gibt *False* zurück, wenn beide Operanden *True* oder wenn beide *False* sind. Bei Werten vom Typ *Date* liefern die Operatoren = bzw. <> das Ergebnis des Vergleichs (*True* oder *False*) zurück. Bei Werten vom Typ *Char* geben die Operatoren das Ergebnis aus dem Vergleich der beiden Unicode-Werte zurück. Beim Datentyp *String* wird das Ergebnis des Zeichenkettenvergleichs als *True* oder *False* zurückgegeben.

Hinweis

Beim Datentyp *String* wird das Ergebnis des Vergleichs von den Compileereinstellungen beeinflusst. Mit *Option Compare Text* erfolgen die Vergleiche auf Unicode-Basis (*Text*) über den Text. Bei *Option Compare Binary* werden die Vergleiche dagegen direkt auf den Binärwerten der Zeichenfolgen ausgeführt.

Vergleichsoperatoren werden häufig in Schleifen und Verzweigungen eingesetzt:

```
While a < 10
..
End While
If a > 100 Then
...
End If
```

Achten Sie bei solchen Vergleichen darauf, dass die Ausdrücke auch vom gleichen Datentyp sind. Bei impliziter Typkonvertierung kann es vorkommen, dass zwei ungleiche Werte durch die Konvertierung als gleich interpretiert werden. Ist *Option Strict On* gesetzt, löst der Compiler beim Vergleich zweier unterschiedlicher Typen einen Fehler aus. Bei Gleitkommazahlen sollte kein Vergleich auf Gleichheit durchgeführt werden, da dieser zu Problemen führt. Ermitteln Sie die Differenz der beiden Gleitkommazahlen und prüfen Sie, ob diese unterhalb einer bestimmten Schwelle liegt (z.B. *if (a-b) < 0.0001 Then*).

5.2.4 Operatoren für Zeichenfolgen

Bezüglich Zeichenfolgen gibt es die beiden bereits erwähnten Operatoren `+` und `&`, die eine Verkettung von Teilzeichenketten durchführen. Zudem erlaubt Visual Basic 2005 noch die beiden Operatoren *Like* und *Is*. Der Operator *Like* wird folgendermaßen eingesetzt:

```
result = string Like pattern
```

Eine Zeichenkette (*string*) wird dabei auf ein Muster (*pattern*) überprüft. Stimmen beide Zeichenketten überein, liefert *Like* das Ergebnis *True*, andernfalls *False*. Interessant ist dabei der integrierte Mustervergleich, mit dem sich Zeichenfolgen mit Hilfe von Platzhalterzeichen, Zeichenlisten oder Zeichenbereichen in beliebiger Kombination vergleichen können. Die folgende Tabelle enthält die in *pattern* zulässigen Zeichen sowie deren Übereinstimmung:

Zeichen in <i>pattern</i>	Übereinstimmungen in <i>string</i>
<code>?</code>	Ein beliebiges einzelnes Zeichen
<code>*</code>	Null oder mehrere Zeichen
<code>#</code>	Beliebige einstellige Ziffer [0 - 9]
<code>[charlist]</code>	Beliebiges Zeichen in <i>charlist</i>
<code>[!charlist]</code>	Beliebiges Zeichen nicht in <i>charlist</i>

Tabelle 5.5: Platzhalterzeichen für *pattern*

Setzen Sie ein Fragezeichen `?` in *pattern* ein, steht dieses für ein beliebiges Zeichen. Das Sternchen `*` ist der Platzhalter für mehrere Zeichen. Ziffern lassen sich mit `#` darstellen. Zusätzlich können Sie eine Gruppe von einem oder mehreren Zeichen (*charlist*) in eckige Klammern einschließen. Dann werden diese Zeichen mit einem einzelnen Zeichen in *string* abgeglichen. Ein Bindestrich (z.B. `[a-c]`) definiert einen Bereich an Zeichen für den Vergleich. Die Variante mit einem vorangestellten Ausrufezeichen `!` schließt die Zeichen in den eckigen Klammern aus.

Tipp

Sollen beim Vergleich Sonderzeichen wie die eckigen Klammern (`[` und `]`), `?`, `#` oder `*` mit einbezogen werden, müssen diese in eckige Klammern gestellt werden.

Der zweite Operator *Is* erlaubt den Vergleich zweier Objektvariablen. Der Operator liefert den Wert *True* zurück, sobald beide Objektvariablen auf das gleiche Objekt verweisen. Mit:

```
Dim a, b, c As Object
a = "Test"
b = 19
c = a
```

wird der Vergleich *a Is b* den Wert *False* liefern, da es sich um verschiedene Objekte handelt. Die Objektvariable *c* erhält dagegen den Wert der Objektvariablen *a* zugewiesen. Damit zeigen beide Variable auf den gleichen Wert und *Is* liefert das Ergebnis *True*. Das folgende Beispielprogramm zeigt den Einsatz von *Like* und *Is*:

```
'*****
' File/Projekt: Beispiel05_04
' Autor: G. Born www.borncity.de
' Zeigt die Verwendung des LIKE-Operators.
'*****
Option Strict On
Imports System.Console      ' für WriteLine
Class Test
    Shared Sub Main()        ' das ist das Hauptmodul
        Dim a, b As Object

        ' Nutzen des IS-Operators
        a = "Test"
        b = 19
        WriteLine ("a=Test b = 19, a Is b: {0}", a Is b)
        b = a
        WriteLine ("a = b, a Is b: {0}", a Is b)

        ' Nutzen des LIKE-Operators
        WriteLine("Meier LIKE M*er = {0}", "Meier" LIKE "M*er")
        WriteLine("Maier LIKE M[a,e]ier = {0}", "Meier" LIKE "M[a,e]ier")
        Write ("Bitte die Eingabtaste drücken")
        ReadLine
    End Sub
End Class
```

Listing 5.3: Anwendung von Like und Is

Hinweis

Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap05\Beispiel5_04` auf der Begleit-CD. In Visual Basic 2005 wurde der *IsNot*-Operator neu eingeführt. Um z.B. zu prüfen, ob eine Objektvariable einen Wert aufweist, war eine Konstruktion der Art *Not object1 Is Nothing* erforderlich. Dieser Ausdruck ist *true*, wenn das Objekt *object1* einen Wert aufweist. Nun lässt sich die Anweisung als *object1 IsNot Nothing* schreiben, was wesentlich transparenter ist. Der Vergleich zweier Objektvariable auf Übereinstimmung lässt sich mit *obj1 IsNot obj2* testen. Bei ungleichen Inhalten liefert der Ausdruck den Wert *true*.

5.2.5 Priorität der Operatoren

Kommen mehrere Operatoren in einem Ausdruck vor, steuert die Priorität der Operatoren die Auswertung. Beim Ausdruck $x + y * z$ hat die Multiplikation eine höhere Priorität als die Addition, es wird der Ausdruck $x + (y * z)$ ausgewertet. Die folgende Tabelle enthält alle Operatoren in absteigender Priorität. Operatoren innerhalb einer Zeile haben gleiche Priorität und werden von links nach rechts in der Reihenfolge ausgewertet, in der sie im Ausdruck auftreten:

Kategorie der Operatoren
Primär: Alle Nichtoperator-Ausdrücke
Potenzierung: ^
Unäre Negation: +, -
Multiplikativ: *, /
Division ganzer Zahlen: \
Modulo-Operator: Mod
Additiv: +, -
Verkettung: &
Relational: =, <>, <, >, <=, >=, Like, Is, TypeOf...Is
Bedingtes NOT: Not
Bedingtes AND: And, AndAlso
Bedingtes OR: Or, OrElse
Bedingtes XOR: Xor

Tabelle 5.6: Prioritäten der Operatoren (oberste Einträge haben die höchste Priorität)

Wenn sich ein Operand zwischen zwei Operatoren gleichen Rangs befindet, wird anhand der Orientierung des Operators festgelegt, in welcher Reihenfolge die Operationen ausgeführt werden. Alle binären Operatoren sind links-assoziativ, d.h., sie werden von links nach rechts ausgeführt. Operatoren lassen sich klammern, um bestimmte Prioritäten bei der Abarbeitung zu erzwingen. Diese Klammerung sollte allein aus Gründen der besseren Lesbarkeit immer verwendet werden.

5.3 Arbeiten mit Wert- und Verweistypen

In *Kapitel 3* haben Sie die Möglichkeit zum Deklarieren von Konstanten und Variablen mit primitiven Datentypen kennen gelernt. Dies sind alle Datentypen, die über Schlüsselwörter wie *Byte*, *Integer* etc. vereinbart werden können, wobei die Schlüsselwörter Aliasnamen für die in *System* vordefinierten Datentypen (z.B. *System.Int32*) sind. Wenn Sie eine Variable mit der Anweisung

```
Dim a As Integer
```

deklarieren, wird diese vom Programm im Speicher angelegt. Sie können dieser Variablen einen Wert zuweisen und die Inhalte von Variablen lassen sich untereinander zuweisen. Dies wird beispielsweise in folgender Sequenz demonstriert:

```
Dim a As Integer
Dim b As Integer
a = 100
b = a
b = b + 10
```

Bei dieser Sequenz wird die Variable *a* den Wert 100 erhalten, der dann der Variablen *b* zugewiesen wird. Ändert sich der Wert von *b*, bleibt der Wert der Variablen *a* unverändert.

In Visual Basic bezeichnet man solche Variablen als Werttypen (Value Types). Sobald die Variable deklariert wurde, erhält sie einen Wert. Weisen Sie eine solche Variable einer anderen Variablen zu, wird letztendlich deren Wert in die Zielvariable kopiert. Dies ist das erwartete Verhalten.

Variablen können aber noch auf eine andere Art verwaltet werden, indem der Wert im Laufzeit-Heap hinterlegt wird. Die Variable enthält dann lediglich einen Verweis auf den Wert im Laufzeit-Heap. Existiert noch kein Wert auf dem Laufzeit-Heap, enthält die Variable einen Nullverweis (auch als *Nothing* bezeichnet). In diesem Fall spricht man von Verweistypen (Reference Types). Interessant ist dabei, dass beim Kopieren von Verweistypen nicht deren Wert, sondern die Referenz auf den Speicher im Laufzeit-Heap kopiert wird.

Hinweis

Variablen, die auf primitiven Typen, auf den noch nicht behandelten Auflistungen (Enumerationen) sowie auf Strukturen basieren, werden als Werttypen verwaltet. Klassen, Zeichenfolgen, Standardmodule, Schnittstellen, Arrays und Delegaten werden dagegen über sogenannte Verweistypen realisiert.

Objektvariable werden als Verweistypen angelegt, d.h., die Variable enthält eine Referenz auf das Objekt im Laufzeit-Heap (oder einen Nullverweis, falls keine Objektinstanz existiert). Der folgende Code definiert im Vorgriff auf das kommende *Kapitel 7* eine einfache Klasse, die ihrerseits nur eine Variable enthält.


```
Class Class1
    Friend a As Integer = 0
End Class
```

Die Variable *a* dieser Klasse ist als *Friend* deklariert, lässt sich also in anderen Modulen oder Klassen der gleichen Assembly referenzieren. Dies wäre beispielsweise mit folgenden Anweisungen möglich:

```
Dim r1 As New Class1()
r1.a = 123
```

Bei der Deklaration der Variablen *r1* wird das Schlüsselwort *New* verwendet. Dieses Schlüsselwort erlaubt, Objekte auf dem Heap anzulegen (und ist ein Hinweis auf einen Verweistyp). Hinter dem Schlüsselwort folgt dann der Name des betreffenden Objekts – hier geben wir die Klasse *Class1* an. Mit dieser Konstruktion wird einmal eine Variable *r1* auf dem Heap angelegt. In der zweiten Anweisungszeile sehen Sie, wie sich das betreffende Objekt über die Variable *r1* ansprechen lässt. Wir weisen einfach der Variablen *a*, die in der Klasse *Class1* deklariert wurde, einen Wert zu. Bei der Zuweisung wird der Wert 123 auf dem Heap untergebracht. Gleichzeitig trägt das Laufzeitsystem in der Variablen *r1* einen Zeiger ein, der auf den Speicherplatz des Objekts im Heap verweist. Solche Verweistypen bieten aber die Möglichkeit, dass mehrere Variable auf den gleichen Speicherplatz verweisen. Nehmen wir einmal die zusätzliche Deklaration

```
Dim r2 As Class1 = r1
```



Abbildung 5.2: Ausgaben des Beispielprogramms

Hier wird nur eine Variable *r2* vom Typ *Class1* deklariert. Da das Schlüsselwort *New* fehlt, legt das Laufzeitsystem keine neue Instanz von *Class1* auf dem Heap an, es wird lediglich Speicherplatz für die Referenz erzeugt. In der Deklaration wird der Wert der Variablen *r2* jedoch auf den Wert von *r1* initialisiert. Da es sich um einen Verweistyp handelt, kopiert das Laufzeitsystem aber nicht den Wert der Variablen *r1*, sondern lediglich den Verweis. Als Folge zeigt die Variable *r2* auf den Speicherplatz, der auch durch *r1* adressiert wird. Weisen Sie nun einer der Variablen *r1* bzw. *r2* einen anderen Wert zu, wird sich dies auch in der anderen Variablen auswirken. Das folgende Listing zeigt, wie sich Wertetypen und Verweistypen vereinbaren lassen:

```

'*****
' File/Projekt: Beispiel05_05
' Autor: G. Born www.borncity.de
' Zeigt die Verwendung von Werttypen und Verweistypen.
'*****
Option Strict On
Imports System.Console      ' für WriteLine

Class Class1                ' Klasse, definiert Variable a
    Friend a As Integer = 0
End Class

Class Test                  ' Klasse mit Hauptmodul
    Shared Sub Main()
        Dim x1 As Integer = 0    ' Werttyp
        Dim x2 As Integer = x1    ' Werttyp

        x2 = 12                ' neuer Wert in x2

        ' jetzt Verweistypen über Class1 erzeugen
        Dim r1 As New Class1()    ' 1. Verweistyp auf Variable a
        Dim r2 As Class1 = r1      ' zeigt auf r1

        r2.a = 123                ' Wert zuweisen, ist auch in r1

        ' Zeige die Werte
        WriteLine("Werte: x1 = {0}, x2 = {1}, r1 = {2}, r2 = {3}", x1, x2, r1.a, r2.a)
        Write ("Bitte die Eingabtaete drücken")
        ReadLine
    End Sub
End Class

```

Listing 5.4: Vereinbarung von Verweistypen

Im Programmkopf wird eine Klasse *Class1* vereinbart, die intern nur eine Integervariable *a* aufweist. Im Hauptmodul der Klasse *Test* werden dann lokale Variable *x1*, *x2*, *r1* und *r2* deklariert. Bei den Variablen *x1* und *x2* handelt es sich um Werttypen, die lokal auf dem Stapel gespeichert werden, während *r1* und *r2* Verweistypen darstellen. Die Variablen *r1* und *r2* sind dabei so deklariert, dass sie auf den gleichen Speicherbereich zeigen.

Nach der Deklaration werden den Variablen Werte zugewiesen. Hierbei wird der Variablen *r2.a* der Wert 123 zugewiesen. Anschließend sorgt eine *WriteLine()*-Anweisung für die Anzeige der Ergebnisse im Fenster der Konsole. Während die beiden Variablen *x1* und *x2* unterschiedliche Werte aufweisen, zeigen *r1.a* und *r2.a* den gleichen Wert 123.

Ursache ist die Verwendung eines Verweistyps, wodurch beide Variable auf den gleichen Speicherplatz im Heap zeigen.

Hinweis

Sie finden die Projektdateien dieser Konsolenanwendung im Ordner `\Beisp\Kap05\Beispiel5_05` auf der Begleit-CD.

Für einfache Programme werden Sie in der Regel mit Werttypen arbeiten, da diese einfacher zu handhaben sind und etwas effizienter vom Laufzeitsystem verwaltet werden. Verweistypen werden dann erforderlich, wenn Sie mit Objekten arbeiten. Dann sollten Sie sich merken, dass bei Zuweisungen zwischen Objektvariablen nicht deren Wert, sondern nur die Referenz kopiert wird. Ein Verweistyp lässt sich mit

```
Variablenname = Nothing
```

auf einen Nullzeiger zurücksetzen. Der Verweis ist dann undefiniert und von dieser Variablen kann nicht mehr auf den Wert zugegriffen werden.

5.4 Arbeiten mit Zeichenketten

Zeichenketten werden wie andere Variablen intern als Objekte verwaltet, da die primitiven Datentypen von der .NET-Framework-Klassenbibliothek bereitgestellt werden. Diese Klassenbibliothek stellt eine Reihe von Methoden und Eigenschaften für Variablen vom Typ *String* bereit. Die nachfolgende Tabelle enthält eine Übersicht über die Eigenschaften und Methoden (d.h. der Mitglieder) der *String*-Klasse.

Mitglied	Beschreibung
Chars	Eigenschaft, die das Zeichen an der angegebenen Zeichenposition innerhalb der Instanz der Zeichenkette zurückliefert
Length	Eigenschaft, die die Zahl der Zeichen des Strings (als der Instanz) zurückliefert
Clone()	Methode, die einen Verweis auf die Instanz des Strings zurückgibt
Compare()	Methode, die zwei angegebene <i>String</i> -Objekte miteinander vergleicht und <i>True</i> oder <i>False</i> zurückgibt
CompareOrdinal()	Methode zum Vergleichen zweier <i>String</i> -Objekte, ohne die lokalen Spracheinstellungen zu berücksichtigen
CompareTo()	Methode zum Vergleichen einer Instanz mit einem angegebenen Objekt
Concat()	Methode, die eine oder mehrere Instanzen von <i>String</i> -Objekten verkettet
Copy()	Erstellt eine neue Instanz von <i>String</i> mit demselben Wert wie eine angegebene Instanz
CopyTo()	Kopiert eine angegebene Anzahl von Zeichen von einer angegebenen Position in dieser Instanz an eine angegebene Position in einem Array von Unicode-Zeichen
EndsWith	Bestimmt, ob das Ende dieser Instanz mit dem angegebenen <i>String</i> übereinstimmt

Mitglied	Beschreibung
Equals()	Methode die ermittelt, ob zwei String-Objekte denselben Wert haben
Format()	Die Methode ersetzt jede Formatangabe in einem angegebenen String durch Text, der dem Wert eines entsprechenden Objekts entspricht
GetEnumerator()	Die Methode ruft ein Auflistungsobjekt ab, das die einzelnen Zeichen der String-Instanz durchlaufen kann
IndexOf()	Gibt den Index des ersten Vorkommens eines Strings oder einer Folge von Zeichen innerhalb der Instanz an
IndexOfAny()	Die Methode liefert den Index des ersten Vorkommens eines beliebigen Zeichens aus einem angegebenen Array von Unicode-Zeichen in dieser Instanz zurück
Insert()	Fügt eine angegebene Instanz von String an einer angegebenen Indexposition in diese Instanz ein
Join()	Die Methode fügt zwischen je zwei Elemente eines String-Arrays einen angegebenen trennenden String ein und liefert eine einzige verkettete Zeichenfolge
LastIndexOf()	Indexposition des letzten Vorkommens eines angegebenen Unicode-Zeichens oder String in dieser Instanz
LastIndexOfAny()	Indexposition des letzten Vorkommens eines oder mehrerer angegebener Zeichen eines Unicode-Arrays in dieser Instanz
PadLeft()	Die Methode richtet die Zeichen dieser Instanz rechtsbündig aus und füllt links Leerzeichen oder das angegebene Unicode-Zeichen auf, um die geforderte Länge zu erreichen
PadRight()	Die Methode richtet die Zeichen dieser Instanz linksbündig aus und füllt rechts Leerzeichen oder das angegebene Unicode-Zeichen auf, um die geforderte Länge zu erreichen
Remove()	Die Methode löscht die angegebene Anzahl von Zeichen ab der angegebenen Position aus dieser Instanz
Replace()	Die Methode ersetzt in dieser Instanz jedes Vorkommen eines angegebenen Unicode-Zeichens oder eines String durch ein anderes angegebenes Unicode-Zeichen oder durch einen String
Split()	Ermittelt die Teilzeichenfolgen in dieser Instanz, die durch ein oder mehrere in einem Array angegebene Zeichen getrennt sind, und legt die Teilzeichenfolgen anschließend in einem String-Array ab
StartsWith()	Die Methode ermittelt, ob der Anfang dieser Instanz mit dem angegebenen String übereinstimmt
Substring()	Die Methode ermittelt eine Teilzeichenfolge dieser Instanz
ToCharArray()	Mit der Methode lassen sich die Zeichen der String-Instanz in ein Feld von Unicode-Zeichen kopieren
ToLower()	Die Methode gibt eine Kopie dieses String in Kleinbuchstaben zurück
ToString()	Die Methode konvertiert den Wert dieser Instanz in einen String
ToUpper()	Gibt eine Kopie dieses Strings in Großbuchstaben zurück
Trim()	Entfernt sämtliche Zeichen aus einer angegebenen Menge von Zeichen am Anfang und am Ende dieser Instanz

Mitglied	Beschreibung
TrimEnd()	Entfernt sämtliche Zeichen aus einer in einem Array von Unicode-Zeichen angegebenen Menge von Zeichen vom Ende dieser Instanz
TrimStart()	Entfernt sämtliche Zeichen aus einer in einem Array von Unicode-Zeichen angegebenen Menge von Zeichen am Anfang dieser Instanz

Tabelle 5.7: Eigenschaften und Methoden der *String*-Klasse

Die Klasse *String* bietet also verschiedene Methoden und Eigenschaften, über die Sie Zeichenketten bearbeiten und deren Eigenschaften herausbekommen können. Wenn Sie bereits Programmiererfahrung besitzen, an dieser Stelle eine Entwarnung. Sie brauchen nicht alles zu vergessen, was Sie über die Bearbeitung von Zeichenketten wissen. Visual Basic 2005 ist so implementiert, dass einfache Zuweisungen und Zeichenkettenverknüpfungen intern automatisch auf die betreffenden Methoden umgesetzt werden. Nehmen wir als einfaches Beispiel die Zuweisung von Zeichenketten.

```
Dim T1 As String = "Hallo"
Dim T3 As String
T3 = String.Copy(T1)
```

In dieser Sequenz wird die Variable *T1* bei der Deklaration mit einem Wert initialisiert. Anschließend kommt die *Copy()*-Methode zum Einsatz, um die Zeichenkette der Variablen *T1* in *T3* zu kopieren. Wer bereits in Visual Basic programmiert hat, wird die letzte Anweisung aber als

```
T3 = T1
```

schreiben. Man sieht sofort, dass der Inhalt der Variablen *T1* in *T3* kopiert wird. Sie können den letzten Ansatz ruhig weiter verwenden. Intern sorgt das Laufzeitsystem dafür, dass bei der Zuweisung des Werts der *String*-Variablen *T3* die *Copy()*-Methode aufgerufen wird.

Hinweis

In Visual Basic werden Zeichenketten als Objekte mit fester Länge verwaltet. Ändern Sie eine Zeichenkette in der Länge, erzeugt das Laufzeitsystem einen neuen Speicherplatz für die Zeichenkette und kopiert den alten Inhalt auf diese Stelle. Danach wird der Speicherplatz des alten Objekts freigegeben und der Zeiger in der *String*-Variablen auf die neue Speicherposition umgesetzt. Daher bietet Visual Basic verschiedene Methoden zur Verwaltung der Zeichenketten.

Ähnliches gilt auch für die *Concat()*-Methode, die zwei Zeichenketten verknüpft. Um zwei Texte zu verknüpfen, werden Sie wohl den *&*-Operator benutzen.

```
Dim txt As String = "Hallo"
txt = txt & " Welt"
```

Die Variable *txt* enthält anschließend die Zeichenkette *Hallo Welt*. Dies kennen Sie vielleicht schon aus den Beispielen der vorhergehenden Kapitel. Intern benutzt Visual Basic dabei die *Concat()*-Methode.

5.4.1 Zeichenkettenvergleich

Der Vergleich von Zeichenketten lässt sich über Vergleichsoperatoren oder über die Methoden des *String*-Objekts ausführen. Sie können beispielsweise zwei Texte definieren und auf Gleichheit prüfen.

```
Dim T1 As String = "Welt"
Dim T2 As String = "WELT"
WriteLine("Welt=WELT: {0}", "Welt"= "WELT")
```

Hier wurden zwei Variable *T1* und *T2* vereinbart, die zwar das Wort *Welt*, aber in unterschiedlicher Schreibweise enthalten. In der *WriteLine()*-Anweisung wird das Ergebnis des Vergleichs auf Konsolenebene angezeigt.

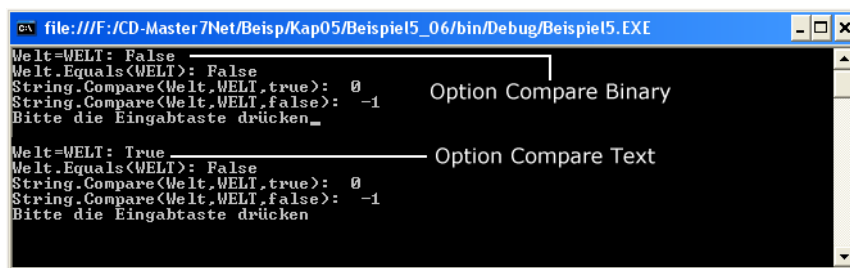


Abbildung 5.3: Ausgaben des Vergleichsprogramms

Dieser Vergleich wird aber, je nach Einstellung von *Option Compare Text* bzw. *Option Compare Binary*, unterschiedliche Ergebnisse ausgeben (in Abbildung 5.3 wurden die Ergebnisse zweier Programmdurchläufe zusammengefasst). Bei einem Textvergleich werden nur die Buchstaben verglichen, Groß-/Kleinschreibung besitzt keine Bedeutung. Bei der *Option Binary* vergleicht man dagegen die Unicodedarstellung der Zeichenketten, und da Groß-/Kleinbuchstaben unterschiedliche Codes benutzen, wird ein ungleiches Ergebnis angezeigt. Alternativ können Sie aber die *Equals()*-Methode verwenden, um die Gleichheit zweier Zeichenketten unabhängig von der *Option Compare*-Einstellung zu prüfen:

```
WriteLine("Welt.Equals(WELT): {0}", T1.Equals(T2))
```

Die obige Anweisung gibt das Ergebnis des Vergleichs der Variablen *T1* und *T2* zurück. Mit den Vorgabewerten ist das Ergebnis immer *False*. Die Alternative besteht darin, die *Compare()*-Methode zum Vergleichen von Strings zu verwenden. Diese Methode lässt sich folgendermaßen aufrufen:

```
Ergebnis = String.Compare(T1, T2, False)
```

Die beiden ersten Parameter der Methode enthalten die zu vergleichenden Strings. Im dritten Parameter legen Sie fest, ob Groß-/Kleinschreibung eine Rolle spielt (Parameter *True* deaktiviert diesen Test). Die *Compare()*-Methode liefert den Wert 0 zurück, falls die Strings gleich sind. Ein Wert kleiner Null bedeutet, dass der Wert des ersten Parameters kleiner als der Wert des zweiten Parameters ist. Ein Wert größer Null signalisiert, dass der Wert des zweiten Parameters größer als der Wert des ersten Parameters ist. Das folgende Listing demonstriert den Umgang mit den Vergleichsoperatoren:

```

'*****
' File/Projekt: Beispiel05_06
' Autor: G. Born www.borncity.de
' Zeichenketten vergleichen.
'*****
Option Strict On
Option Compare Text ' bzw. Binary
Imports System.Console      ' für WriteLine

Class Test                  ' Klasse mit Hauptmodul
    Shared Sub Main()
        Dim T1 As String = "Welt"
        Dim T2 As String = "WELT"

        ' Zeige die Werte
        WriteLine("Welt=WELT: {0}", "Welt"="WELT")
        WriteLine("Welt.Equals(WELT): {0}", T1.Equals(T2))
        ' Ohne Groß-/Kleinschreibung
        WriteLine("String.Compare(Welt,WELT,true): {0}", _
            String.Compare(T1,T2,True))
        ' Mit Groß-/Kleinschreibung
        WriteLine("String.Compare(Welt,WELT,false): {0}", _
            String.Compare(T1,T2,False))
        Write ("Bitte die Eingabtaste drücken")
        ReadLine
    End Sub
End Class

```

Listing 5.5: Beispiel mit Vergleichsoperatoren

Hinweis

Die Projektdateien des Beispiels finden sich im Ordner `\Beisp\Kap05\Beispiel5_06` auf der Begleit-CD. Übersetzen Sie das Beispiel als Konsolenanwendung und starten Sie es von der Eingabeaufforderung. In einem zweiten Durchgang müssen Sie die Anweisung *Option Compare Binary* im Programmkopf setzen.

5.4.2 Wie kann ich in Zeichenketten suchen?

Häufig kommt es vor, dass ein Muster, bestehend aus einem oder mehreren Zeichen, in einer Zeichenkette gefunden werden soll. Im vorhergehenden Abschnitt »Operatoren für Zeichenfolgen« wurde gezeigt, wie sich zwei Zeichenketten über den *Like*-Operator auf Übereinstimmung prüfen lassen. Dies reicht aber nicht aus, falls Sie wissen möchten, ob der Begriff »Born« in einem bestimmten Text wie »This star was born in Memphis« vorkommt. Glücklicherweise bietet die .NET-Framework-Klassenbibliothek die entsprechende Methode *IndexOf()* für den Datentyp *String*. Die folgenden Anweisungen demonstrieren, wie sich eine Suche im Text realisieren lässt:

```
Const txt As String = "This star was born in Memphis"
Dim muster As String = "Born"
WriteLine (txt.IndexOf(muster))
```

Die Methode *IndexOf()* wird auf die zu durchsuchende Zeichenkette (hier die Konstante *txt*) angewandt. Als Parameter erwartet die Methode das Suchmuster (hier die Variable *muster*). Die Methode liefert dann einen Wert zurück, der die Position des Musters innerhalb der Zeichenkette angibt. Ein Wert -1 bedeutet, dass das Muster im String nicht gefunden wurde. Ein Wert größer gleich 0 gibt die Position des Musters innerhalb der Zeichenkette an.

```
file:///F:/CD-Master7Net/Beisp/Kap05/Beispiel5_07/bin/Debug/Beispiel5.EXE
Muster: Born
Suchergebnis (-1 nicht gefunden): -1
Text: This star was born in Memphis
Muster: Born
Suchergebnis (-1 nicht gefunden): 14
Text: This star was born in Memphis
Muster: Born
Suchergebnis (-1 nicht gefunden): 14
Text: This star was born in Memphis
Muster: Born
Suchergebnis (-1 nicht gefunden): -1
Text: This star was born in Memphis
Muster: Born
Suchergebnis (-1 nicht gefunden): 14
Text: This star was born in Memphis
Buchstaben S, B
Suchergebnis (-1 nicht gefunden): 3
-----
Text: This star was born in Memphis
Muster: this
beginnt mit 'This': True
Text: This star was born in Memphis
Muster: memphis
endet mit 'memphis': True
Bitte die Eingabetaste drücken
```

Abbildung 5.4: Ausgaben bei der Textsuche

Verwenden Sie den obigen Code, wird die *IndexOf()*-Methode den Wert -1 zurückliefern (dies ist in Abbildung 5.4 in der ersten Ergebniszeile zu sehen). Bei der Suche innerhalb einer Zeichenkette müssen Sie zwei Probleme mit berücksichtigen. Die Methode *IndexOf()* des *String*-Objekts unterscheidet zwischen Groß-/Kleinschreibung. Zudem erfolgt die Suche ohne Berücksichtigung der landespezifischen Einstellungen. In obigem Beispielcode enthält der Text den Teilausdruck »born« in Kleinbuchstaben, während das Suchmuster als »Born« geschrieben wurde. Gerade bei variablen Texten ist die Schreibweise selten vorbestimmt, Sie müssen also Groß-/Kleinschreibung abfangen.

Möchten Sie bei der Suche die Groß-/Kleinschreibung unberücksichtigt lassen, greifen Sie zu einem Trick und konvertieren einfach das Muster sowie den zu durchsuchenden String in Groß- oder Kleinbuchstaben. Dies wird in folgender Codesequenz demonstriert:

```
Dim txt As String = "This star was born in Memphis"  
Dim muster As String = "Born"  
WriteLine (txt.IndexOf.ToLower(muster.ToLower))
```

Die Methoden *ToLower()* bzw. *ToUpper()* des *String*-Objekts erlauben eine Zeichenkette in Groß- oder Kleinbuchstaben zu wandeln. Um das Muster in Kleinbuchstaben zu konvertieren, verwenden Sie *muster.ToLower()*, d.h., dem Namen des Objekts *muster* hängen Sie den Methodennamen getrennt durch einen Punkt an. Bei der Zeichenkette sollen zwei Methoden angewandt werden, folglich wird an den Objektnamen *txt* die Methode *IndexOf()* und dann die Methode *ToLower()* angehängt. Als Argument wird der Suchstring übergeben. Dies führt dazu, dass zur Laufzeit erst der Inhalt von *txt* in Kleinbuchstaben konvertiert und dann die *IndexOf()*-Methode auf das Ergebnis angewandt wird.

Hinweis

Falls häufige Vergleiche mit *IndexOf()* durchzuführen sind, empfiehlt es sich aber aus Effizienzgründen, vor dem Vergleich das Muster und den Vergleichstext mit der *ToLower()*-Methode in Kleinbuchstaben zu wandeln.

Die *IndexOf()*-Methode unterstützt optionale Argumente, die die Suche in der Zeichenkette ab einer bestimmten Position zulassen. Zusätzlich bietet die *String*-Klasse noch die Methode *IndexOfAny()* an, die das Vorkommen eines Zeichens aus einem Feld vom Datentyp *Char* in der Zeichenkette signalisiert. Sie können mit dieser Methode gleichzeitig nach verschiedenen Buchstaben in einer Zeichenkette suchen lassen. Wird einer der Buchstaben gefunden, liefert die Methode die Position in der Zeichenkette. Zusätzlich stellt die Klasse noch die Methoden *StartsWith()* und *EndsWith()* bereit, die prüfen, ob eine Zeichenkette mit einem bestimmten Muster beginnt oder endet und den Wert *True* oder *False* zurückgeben.

Tipp

Die Anwendung einiger dieser Methoden finden Sie im Projekt *Beispiel5_07* der Begleit-CD. Details zu den einzelnen Methoden liefert die Hilfe zu .NET Framework. Suchen Sie auf der Registerkarte *Index* der nach dem Stichwort *String.IndexOf*.

Länderspezifische Variante

Um verschiedene länder- bzw. kulturspezifische Eigenheiten bei der Suche bzw. beim Vergleich zu berücksichtigen, müssen Sie auf die *CompareInfo*-Klasse ausweichen. Diese Klasse findet sich im Namensraum *System.Globalization* und umfasst einen Satz an Methoden, die Sie zum Vergleichen und Suchen von kulturabhängigen Zeichenfolgen verwenden können. (Die *String*-Klasse findet sich dagegen im Namensraum *System* und braucht nicht extra über *Imports* referenziert zu werden.) Die Klasse *CompareInfo* bietet

auch eine *IndexOf()*-Methode, die bei der Suche den gewünschten Kulturkreis berücksichtigt. Auch wenn dies in für den deutschen Kulturkreis erstellten Visual-Basic-2005-Anwendungen selten vorkommt, möchte ich den Ansatz prinzipiell vorstellen.

Schauen wir uns doch einmal an, wie sich die betreffende Klasse samt ihren Methoden und Eigenschaften verwenden lässt. Bevor Sie auf die Klasse und deren Member (Methoden bzw. Eigenschaften) zugreifen können, muss eine Instanz erzeugt werden. *CompareInfo* ist aber eine Eigenschaft einer Instanz der *CultureInfo*-Klasse. Folglich muss als Erstes eine Instanz der *CultureInfo*-Klasse erzeugt werden. Beim Instantiieren dieser Klasse legen Sie auch den Kulturkreis fest. Letztendlich erfordert die Instantiierung von *CultureInfo* nur eine einfache Variablendeklaration in Visual Basic 2005:

```
Dim ci As New CultureInfo("da-DK")
```

Hier wird eine Objektvariable *ci* vereinbart und über *New* mit einer Instanz der Klasse *CultureInfo* referenziert. Beim Instantiieren ist dabei das Kürzel für den Kulturkreis anzugeben. Hier habe ich "da-DK" für Dänemark angegeben.

Hinweis

Die Kürzel für die Kulturkreise finden Sie in der Hilfe zum .NET Framework. Hier die Kürzel für verschiedene Kulturkreise, die unterschiedliche Sortierungen unterstützen: en-US Englisch (USA), es-ES Spanisch, zh-TW Chinesisch (Taiwan), zh-CN Chinesisch (Festland-China), zh-HK Chinesisch (Hongkong), zh-SG Chinesisch (Singapur), zh-MO Chinesisch (Macao), ja-JP Japanisch, ko-KR Koreanisch, de-DE Deutsch, hu-HU Ungarisch, ka-GE Georgisch (Georgien).

Die Objektvariable *ci* lässt sich nun einsetzen, um Methoden und Eigenschaften der Klasse abzurufen. Da wir die Methoden der *CompareInfo*-Klasse benötigen, rufen wir von der Instanz der *CultureInfo*-Klasse einfach die *CompareInfo*-Eigenschaft (die ihrerseits ein Objekt ist) ab. Um einen Vergleich über die *IndexOf()*-Methode der *CompareInfo*-Klasse durchzuführen, lässt sich folgende Anweisung verwenden:

```
ci.CompareInfo.IndexOf(txt, muster)
```

Die Objektvariable *ci* verweist auf die Instanz der *CultureInfo*-Klasse. Über diese Objektvariable rufen wir die *CompareInfo*-Eigenschaft ab. Mit *ci.CompareInfo* steht nun ein Objekt bereit, dessen *IndexOf()*-Methode nutzbar ist. Die Methode dieses Objekts weicht etwas von der gleichnamigen Methode des *String*-Objekts ab. Bei *CompareInfo.IndexOf()* müssen Sie sowohl den Text als auch das Suchmuster als Parameter übergeben. Als Ergebnis wird der gleiche Integerwert wie bei der *String.IndexOf()*-Methode zurückgegeben. Ein Wert -1 bedeutet, dass das Muster nicht gefunden wurde. Die Methode erlaubt noch einen optionalen dritten Parameter, der steuert, ob eine Unterscheidung hinsichtlich Groß-/Kleinschreibung zulässig ist. Wird in diesem dritten Parameter die Konstante *CompareOptions.IgnoreCase* angegeben, unterbleibt bei der Suche die Unterscheidung von Groß-/Kleinschreibung. Die folgende Anweisung verdeutlicht diesen Ansatz:

```
ci.CompareInfo.IndexOf(txt, muster, CompareOptions.IgnoreCase)
```

Weitere Details zum Umgang mit den Klassen *CompareInfo* und *CultureInfo* sowie deren Methoden lassen sich der Hilfe des .NET Framework entnehmen.

Ein Beispiel zum Suchen

Zur Demonstration der Vorgehensweise beim Suchen mittels unterschiedlicher Methoden lässt sich das folgende Beispielprogramm verwenden. An dieser Stelle möchte ich noch kurz auf die Deklaration eines Feldes vom Typ *Char* eingehen. Dieses Feld wird bei der *IndexOfAny()*-Methode zur Definition der zu suchenden Zeichen benötigt. Standardmäßig wird man die folgende Anweisung verwenden:

```
Dim zchn() As Char = {"S", "B"}
```

Die Anweisung vereinbart ein Feld *zchn* vom Typ *Char* und initialisiert dieses mit zwei Werten. Sobald im Programm aber die Anweisung *Option Strict On* auftritt, wird der Compiler die obige Anweisung bemängeln. Die Konstanten "S" und "B" werden als Zeichenketten interpretiert. Eine Variable vom Datentyp *Char* kann aber immer nur ein Zeichen aufnehmen. Um das Problem zu lösen, müssen Sie die Konstanten mit dem Datentyp *Char* definieren. Dies kann durch ein angehängtes Zeichen *c* geschehen:

```
Dim zchn() As Char = {"S"c, "B"c}
```

Die Möglichkeit, einen Datentyp bei einem Literal anzugeben (z.B. 123i) hatte ich bereits weiter oben angesprochen. Durch den Buchstaben *c* erkennt der Compiler, dass es sich um eine Konstante vom Datentyp *Char* handelt, eine Typumwandlung kann also unterbleiben. Alternativ können Sie auch die *CType*-Funktion zur Zuweisung einer *Char*-Konstanten verwenden:

```
Dim zchn() As Char = {CType("S", Char), CType("B", Char)}
```

Bei mehreren Feldelementen ist dies aber mit sehr viel Schreibaufwand verbunden. Weiter unten lernen Sie noch die *ToCharArray()*-Methode kennen, die die Aufteilung einer Zeichenkette auf ein *Char*-Array ermöglicht. Weitere Details sind dem folgenden Listing zu entnehmen:

```
'*****  
' File/Projekt: Beispiel5_07  
' Autor: G. Born www.borncity.de  
' Suche in Zeichenketten unter Verwendung der  
' Methoden der String-Klasse und der  
' CultureInfo.CompareInfo-Klasse.  
'*****  
Option Strict On  
Option Compare Text
```

Listing 5.6: Beispiel zum Suchen in Zeichenketten

```
' Vereinbare die benutzten Namensräume
Imports System.Environment      ' für newline-Eigenschaft
Imports System.Globalization    ' für CompareInfo
Imports System.Console          ' für WriteLine

Class Test                      ' Klasse mit Hauptmodul
    Shared Sub Main()
        Dim txt As String = "This star was born in Memphis"
        Dim muster As String = "Born"      ' Suchmuster

        ' gebe Literal 'c' an, um die Konstanten der Initialisierung
        ' als Character zu markieren, da wegen Option Strict On keine
        ' Konvertierung möglich ist - verwende Großbuchstaben für Suchmuster
        Dim zchn() As Char = {"S"c, "B"c} ' Suchbuchstaben

        Dim ci As New CultureInfo("da-DK")

        ' Zeige die Werte ohne Groß-/Kleinschreibung
        WriteLine ("Text: {0}" & newline & "Muster: {1}" & _
            newline & "Suchergebnis (-1 nicht gefunden): {2}", _
            txt, muster, txt.IndexOf(muster))

        ' Ohne Groß-/Kleinschreibung
        WriteLine ("Text: {0}" & newline & "Muster: {1}" & _
            newline & "Suchergebnis (-1 nicht gefunden): {2}", _
            txt, muster, txt.ToLower.IndexOf(muster.ToLower))

        ' Ohne Groß-/Kleinschreibung, mit der toUpper()-Methode
        WriteLine ("Text: {0}" & newline & "Muster: {1}" & _
            newline & "Suchergebnis (-1 nicht gefunden): {2}", _
            txt, muster, txt.ToUpper.IndexOf(muster.ToUpper))

        ' Mit CompareInfo.IndexOf - mit Groß-/Kleinschreibung
        WriteLine ("Text: {0}" & newline & "Muster: {1}" & _
            newline & "Suchergebnis (-1 nicht gefunden): {2}", _
            txt, muster, ci.CompareInfo.IndexOf(txt, muster))

        ' Mit CompareInfo.IndexOf - ohne Groß-/Kleinschreibung
        WriteLine ("Text: {0}" & newline & "Muster: {1}" & _
            newline & "Suchergebnis (-1 nicht gefunden): {2}", _
```

Listing 5.6: Beispiel zum Suchen in Zeichenketten (Forts.)

```

        txt, muster, ci.CompareInfo.IndexOf(txt, muster, _
        CompareOptions.IgnoreCase))

' Mit CompareInfo.IndexOf - ohne Groß-/Kleinschreibung
WriteLine ("Text: {0}" & newline & "Buchstaben {1}, {2}" & _
        newline & "Suchergebnis (-1 nicht gefunden): {3}", _
        txt, zchn(0), zchn(1), txt.ToUpper.IndexOfAny(zchn))

WriteLine ("----")
' Text beginnt mit ".." (Groß-/Kleinschreibung berücksichtigen)
WriteLine ("Text: {0}" & newline & "Muster: {1}" & _
        newline & "beginnt mit 'This': {2}", _
        txt, "this", txt.ToUpper.StartsWith("this".ToUpper))

' Text endet mit ".." (Groß-/Kleinschreibung berücksichtigen)
WriteLine ("Text: {0}" & newline & "Muster: {1}" & _
        newline & "endet mit 'memphis': {2}", _
        txt, "memphis", txt.ToUpper.EndsWith("memphis".ToUpper))

Write ("Bitte die Eingabtaete drücken")
ReadLine
End Sub
End Class

```

Listing 5.6: Beispiel zum Suchen in Zeichenketten (Forts.)

Hinweis

Die Projektdateien des Beispiels finden sich im Ordner `\Beisp\Kap05\Beispiel5_07` auf der Begleit-CD. Übersetzen Sie das Beispiel als Konsolenanwendung und starten Sie es von der Eingabeaufforderung. Das Programm gibt die Ergebnisse der Textsuche zurück. Wird das Muster »Born« gefunden, sollte der Indexwert 14 angezeigt werden.

5.4.3 Zeichenketten bearbeiten

Das Bearbeiten von Zeichenketten dürfte recht häufig auftreten. Die Verkettung zweier Teilstrings stellt dabei die trivialste Variante dar, die Sie bereits kennen gelernt haben. Um an einer Zeichenkette eine bestimmte Zahl von Zeichen links oder rechts anzufügen, lassen sich die Methoden `PadLeft()` und `PadRight()` der `String`-Klasse ganz gut verwenden. Beide Methoden besitzen zwei Parameter:

```

txt.PadLeft(Länge, Zeichen)
txt.PadRight(Länge, Zeichen)

```

wobei *Länge* die Zahl der Zeichen in der Zeichenkette angibt. Der Parameter *Zeichen* kann ein beliebiges Unicode-Zeichen (z.B. ein Leerzeichen, ein Punkt etc.) sein. *PadLeft()* richtet dann den Text rechtsbündig aus und füllt den String links mit dem angegebenen Zeichen auf, um die Länge zu erreichen. Bei *PadRight()* wird das Zeichen rechts an den Text angehängt, bis die angegebene Länge erreicht wird.

Sie können dies nutzen, um beispielsweise eine Zeichenkette mit n Leerzeichen zu erzeugen. Die folgenden Anweisungen erstellen eine Zeichenkette mit zehn Leerzeichen:

```
Dim txt As String = ""  
txt = txt.PadLeft(10, " ")
```

Wenn Sie also die Anweisung *txt.Length* verwenden, wird die *Length*-Eigenschaft des *String*-Objekts den Wert 10 zurückliefern.

Achtung

An dieser Stelle noch ein kurzer Hinweis zur Initialisierung von *String*-Variablen. Wenn Sie eine Variable mit *Dim text As String* deklarieren, wird diese als Verweistyp angelegt (siehe auch vorhergehende Abschnitte). Ein Verweistyp besitzt einen Wert auf dem Heap und die Variable enthält einen Zeiger auf diesen Wert. Wird die Variable nur deklariert, besitzt sie den Wert *Nothing*. Das Laufzeitsystem erkennt an diesem Wert, dass die Variable nicht initialisiert ist. Die Anweisung *txt = txt.PadLeft(10, " ")* führt bei einer nicht initialisierten Variablen *txt* zu einem Laufzeitfehler! Dies lässt sich vermeiden, wenn Sie Stringvariablen bei der Deklaration mit einer leeren Zeichenkette "" initialisieren.

Um an eine bestehende Zeichenkette ein Zeichen n-Mal anzufügen, ist folgende Anweisung erforderlich:

```
newtxt = txt.PadLeft(txt.Length + 5, blank)
```

Hier wird die Länge des Texts mittels der *Length*-Eigenschaft ermittelt und die Zahl der hinzuzufügenden Zeichen addiert. Bei *PadRight()* gilt das Gleiche.

Möchten Sie in einer Zeichenkette führende oder anhängende Zeichen (z.B. Leerzeichen) links oder rechts abschneiden? Dann stehen Sie vor der Qual der Wahl, denn Visual Basic 2005 unterstützt sowohl die alten Visual-Basic-Funktionen *Trim*, *LTrim* und *RTrim* über die *Microsoft.VisualBasic*-Kompatibilitätsklasse als auch Methoden wie *Trim*, *TrimStart* und *TrimEnd* der *String*-Klasse. Dummerweise erfordern die Funktionen und Methoden unterschiedliche Aufrufe. Für die alten Funktionen aus Visual Basic sind folgende Anweisungen zu verwenden:

```
Imports Microsoft.VisualBasic  
...  
Dim txt As String = "  This star was born in Memphis  "  
txt = LTrim(txt) ' entfernt führende Leerzeichen
```

```
txt = RTrim (txt)      ' entfernt anhängende Leerzeichen
txt = Trim (txt)       ' entfernt führende und anhängende Leerzeichen
```

Wichtig ist die Vereinbarung des Namensraums *Microsoft.VisualBasic*, da sonst die Funktionen unbekannt sind. Bei neuen Programmen sollten Sie aber konsequent auf die Methoden *TrimStart()*, *TrimEnd()* des *String*-Objekts setzen. Fehlt das Argument, entfernt die betreffende Methode Leerzeichen am Anfang (*TrimStart*), am Ende (*TrimEnd*) oder an beiden Seiten (*Trim*) des *String*-Objekts. Als Argument lässt sich auch ein Feld mit Werten vom Typ *Char* übergeben. Dann entfernen die Methoden die Zeichen, die im Feld angegeben wurden.

```
Dim txt As String = "heini"
Dim zchn1() As Char = {"h"c, "i"c }
' jetzt Zeichen h und i rechts und links entfernen ...
txt = txt.Trim (zchn1)
```

In obigem Code werden die Zeichen *h* und *i* am Anfang und Ende der Zeichenkette entfernt. Das *String*-Objekt bietet darüber hinaus recht mächtige Methoden, um Teile einer Zeichenkette abzuschneiden, Teilausdrücke herauszulösen oder Muster in einer Zeichenkette zu ersetzen. Mit der *Substring()*-Methode lassen sich Teilzeichenketten aus einer *String*-Instanz herauslösen. Die folgenden Anweisungen zeigen die Anwendung dieser Methode:

```
Dim txt As String = "This star was born in Memphis"
WriteLine ("ab 10. Zeichen: {0}", txt.SubString(10))
WriteLine ("1. bis 10. Zeichen: {0}", txt.SubString(0, 9))
WriteLine ("die letzten 10 Zeichen: {0}", _
    txt.SubString(txt.Length-10))
```

Möchten Sie einen bestimmten Ausdruck in einer Zeichenkette durch einen anderen Ausdruck ersetzen? Dann ist die *Replace()*-Methode des *String*-Objekts die richtige Wahl. Die *Replace()*-Methode erwartet im ersten Parameter das Suchmuster, während im zweiten Parameter der Ersetzungstext anzugeben ist. Anschließend werden alle Stellen, an denen das Suchmuster vorkommt, durch den neuen Text ersetzt. Der folgende Code demonstriert die Anwendung der Methode:

```
Dim txt As String = "This star was born in Memphis"
txt = txt.Replace ("Memphis", "Orlando")
WriteLine (txt)
```

Die Codesequenz wird die Zeichenfolge »This star was born in Orlando« auf der Konsole ausgeben. Der Teilausdruck »Memphis« wurde durch »Orlando« ersetzt.

Um Teile einer Zeichenkette herauszuschneiden, greifen Sie auf die *Remove()*-Methode des *String*-Objekts zurück. Die Methode erwartet zwei Parameter, die den Anfang und das Ende des zu entfernenden Bereichs angeben. Nehmen wir einmal an, Sie möchten

den Begriff »born« aus dem Text entfernen. Dies lässt sich mit folgender Codefolge bewerkstelligen:

```
Dim txt As String = "This star was born in Memphis"
Dim idx As Integer
idx = txt.ToLower.IndexOf("born") ' Suche den Beginn von "Born"
txt = txt.Remove(idx, "born".Length)
WriteLine(txt)
```

Hier wird die *IndexOf()*-Methode benutzt, um das erste Auftreten des Musters zu ermitteln. Der Rückgabewert entspricht dem ersten Parameter der *Remove()*-Methode. Die Länge der auszuschneidenden Zeichenkette wird für den zweiten Parameter der Methode benutzt.

Tipp

Der hier gezeigte Ansatz stellt sicher, dass Groß/Kleinschreibung berücksichtigt und lediglich das erste auftretende Muster ausgeschnitten wird. Kennen Sie die genaue Schreibweise eines herauszuschneidenden Musters und dürfen mehrere Stellen im String bearbeitet werden, lässt sich die *Replace()*-Methode verwenden. Setzen Sie einfach den zweiten Parameter mit dem Ersetzungstext auf einen leeren String (z.B. *txt = txt.Replace(txt_muster, "")*).

Ein Beispiel mit Zeichenkettenoperationen

Der Einsatz der verschiedenen Methoden soll an einem einfachen Beispiel demonstriert werden. Das Programm definiert eine Zeichenkette und ergänzt diese um Leerzeichen sowie Punkte. Dann wird die Zeichenkette schrittweise am linken/rechten Rand beschnitten. Abschließend versucht das Programm Wörter in der Zeichenkette zu ersetzen sowie Ausdrücke im Text zu entfernen. Dabei kommen die im vorhergehenden Abschnitt beschriebenen Methoden zum Einsatz. Die Ausgaben des Programms finden Sie in Abbildung 5.5. Weitere Details sind dem Programmlisting zu entnehmen.

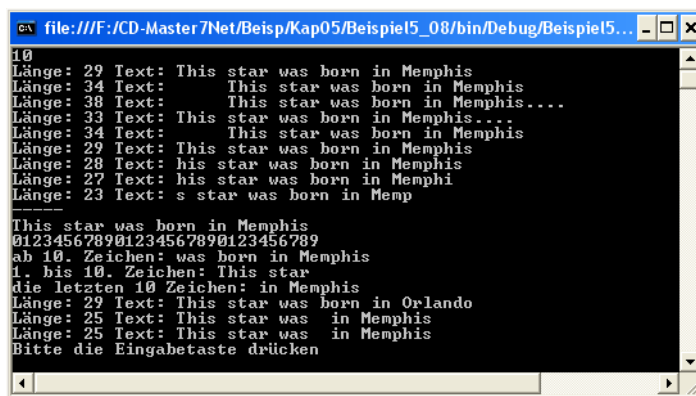


Abbildung 5.5: Ausgaben des Programms zur Bearbeitung von Zeichenketten


```

'*****
' File/Projekt: Beispiel05_08
' Autor: G. Born www.borncity.de
' Demonstriert die Bearbeitung von Zeichenketten
' mit den Methoden der String-Klasse.
'*****
Option Strict On
Option Compare Text

' Vereinbare die benutzten Namensräume
Imports Microsoft.VisualBasic
Imports System.Environment      ' für newline-Eigenschaft
Imports System.Console          ' für WriteLine

Class Test                      ' Klasse mit Hauptmodul
Shared Sub Main()
    Dim txt As String = "This star was born in Memphis"
    Dim newtxt As String
    Dim tmp As String

    Dim dot As Char = "."c      ' Zeichen
    Dim blank As Char = " "c

    Dim text As String = ""

    ' erstelle eine Zeichenkette mit 10 Leerzeichen
    text = text.PadLeft(10, CType(" ",Char))
    WriteLine("{0}", text.Length)

    ' Zeige die Textlänge von txt an
    WriteLine ("Länge: {0} Text: {1}", txt.Length, txt)

    ' jetzt 5 Blanks am Anfang ...
    newtxt = txt.PadLeft(txt.Length + 5, blank)
    WriteLine ("Länge: {0} Text: {1}", _
        newtxt.Length, newtxt)

    ' ... und 4 Punkte am Ende anhängen
    newtxt = newtxt.PadRight(newtxt.Length + 4, dot)
    WriteLine ("Länge: {0} Text: {1}", _

```

Listing 5.7: Bearbeitung von Strings

```
newtxt.Length, newtxt)

' ### Zugriff auf die VB-Funktionen LTRIM und RTRIM
tmp = newtxt           ' merke Text
' jetzt Leerzeichen links abschneiden ...
newtxt = LTrim(newtxt)
WriteLine ("Länge: {0} Text: {1}", _
    newtxt.Length, newtxt)

' ### Anwenden der Methoden der Klasse String
newtxt = tmp           ' alten Text holen

' jetzt Punkte am Ende abschneiden ...
newtxt = newtxt.TrimEnd(dot)
WriteLine ("Länge: {0} Text: {1}", _
    newtxt.Length, newtxt)

' jetzt Leerzeichen links und rechts abschneiden ...
newtxt = newtxt.Trim ()
WriteLine ("Länge: {0} Text: {1}", _
    newtxt.Length, newtxt)

' jetzt Zeichen links entfernen ...
newtxt = newtxt.TrimStart ("T"c)
WriteLine ("Länge: {0} Text: {1}", _
    newtxt.Length, newtxt)

' jetzt Zeichen rechts entfernen ...
newtxt = newtxt.TrimEnd ("s"c)
WriteLine ("Länge: {0} Text: {1}", _
    newtxt.Length, newtxt)

Dim zchn1() As Char = {"h"c,"i"c }
' jetzt Zeichen h und i rechts und links entfernen ...
newtxt = newtxt.Trim (zchn1)
WriteLine ("Länge: {0} Text: {1}", _
    newtxt.Length, newtxt)

' Herauslösen von Teilzeichenketten
WriteLine ("-----" & newline & txt & _
```

Listing 5.7: Bearbeitung von Strings (Forts.)

```

        newline & "012345678901234567890123456789")
WriteLine ("ab 10. Zeichen: {0}", txt.SubString(10))
WriteLine ("1. bis 10. Zeichen: {0}", _
    txt.SubString(0, 9))
WriteLine ("die letzten 10 Zeichen: {0}", _
    txt.SubString(txt.Length-10))

' jetzt den Ort in der Zeichenkette austauschen ...
newtxt = txt.Replace ("Memphis", "Orlando")
WriteLine ("Länge: {0} Text: {1}", _
    newtxt.Length, newtxt)

' jetzt den Namen "born" aus der Zeichenkette entfernen ...
Dim idx As Integer
idx = txt.ToLower.IndexOf("born") ' Suche den Beginn von "Born"
newtxt = txt.Remove (idx, "born".Length)
WriteLine ("Länge: {0} Text: {1}", _
    newtxt.Length, newtxt)

' oder den Namen "born" trickreicher aus der
' Zeichenkette entfernen ...
newtxt = txt.Replace ("born", "")
WriteLine ("Länge: {0} Text: {1}", _
    newtxt.Length, newtxt)

Write ("Bitte die Eingabtaste drücken")

ReadLine
End Sub
End Class

```

Listing 5.7: Bearbeitung von Strings (Forts.)

Hinweis

Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap05\Beispiel5_08` auf der Begleit-CD.

5.4.4 Zeichenketten zerlegen und verknüpfen

Mit den obigen Methoden wie *Substring* lassen sich Zeichenketten in Teile zerlegen. Das *String*-Objekt stellt aber noch Methoden wie *Split()*, *Join()* oder *ToCharArray()* bereit, die mitunter ganz hilfreich sein können. Schauen wir uns zuerst einmal die *ToCharArray()*-Methode an. Auf den vorhergehenden Seiten hatte ich auf die Schwierigkeit hingewiesen, ein Feld vom Datentyp *Char* mit Vorgabewerten zu füllen, wenn *Option Strict On* gesetzt ist. Sie müssen jede Zeichenkonstante mit dem Buchstaben *c* abschließen, um den Datentyp *Char* implizit anzugeben. Dies lässt sich aber elegant mit folgendem Ansatz umgehen:

```
Dim zchn() As Char = "isha".ToCharArray
```

Das Feld *zchn()* besitzt anschließend vier Elemente, die mit den Buchstaben des Musters, also i, s, h und a, gefüllt sind.

Mit der *Split()*-Methode lässt sich ein Text in Teiltex te zerlegen. Die Methode liefert die Ergebnisse in ein *String*-Array zurück. Die Auftrennung des Texts erfolgt durch ein Trennzeichen, welches als Parameter an die Methode übergeben wird:

```
Dim txt As String = "This star was born in Memphis"
Dim sWort() As String = txt.Split (" ")
```

Nach Ausführung der obigen Befehle wird *sWort* insgesamt sechs Elemente mit den Wörtern des Strings aufweisen, da hier das Leerzeichen als Separator angegeben wurde.

Möchten Sie die in einem Array vorliegenden Teilzeichenketten zu einer einzelnen Kette verknüpfen, lässt sich die *Join()*-Methode verwenden. Die Methode erlaubt dabei im ersten Parameter ein Zeichen vorzugeben, welches bei der Verknüpfung der Feldelemente eingefügt wird:

```
newtxt = String.Join (newline, sWort)
```

Die obige Anweisung fügt einen Zeilenumbruch zwischen jeden in *sWort* enthaltenen Teilstring ein. Das folgende kleine Beispielpogramm nutzt die beiden Methoden *Split()* und *Join()*, um einen einfachen Text in Wörter zu zerlegen und diese Wörter in eigenen Zeilen auf der Konsole anzuzeigen (Abbildung 5.6).

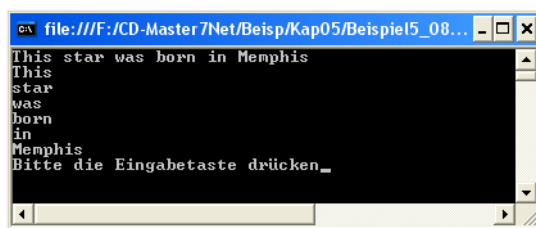


Abbildung 5.6: Zerlegen eines Texts in Wörter mit zeilenweiser Ausgabe

```
'*****
' File/Projekt: Beispiel5_09
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung der Methoden
' Join und Split der String-Klasse.
'*****

Option Strict On
Option Compare Text

' Vereinbare die benutzten Namensräume
Imports System.Environment      ' für newline-Eigenschaft
Imports System.Console         ' für WriteLine

Class Test                      ' Klasse mit Hauptmodul
    Shared Sub Main()
        Dim txt As String = "This star was born in Memphis"
        Dim newtxt As String

        WriteLine (txt)         ' Text anzeigen

        ' Text in einzelne Wörter zerlegen
        Dim sWort() As String = txt.Split (" ")

        ' Wörter + Zeilenumbrüche mit Join kombinieren
        newtxt = String.Join (newline, sWort)

        WriteLine (newtxt)      ' Text anzeigen

        Write ("Bitte die Eingabtaste drücken")

        ReadLine()
    End Sub
End Class
```

Listing 5.8: Beispiel zur Zerlegung eines Texts

Hinweis

Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap05\Beispiel5_09` auf der Begleit-CD.

5.4.5 Die StringBuilder-Klasse

Die Klasse *String* kann nur unveränderliche Zeichenketten verwalten. Ändern Sie etwas an einer Zeichenkette, wird ein neues Objekt erzeugt und die geänderte Zeichenkette in die Objektinstanz kopiert. Alternativ bietet .NET Framework noch die *StringBuilder*-Klasse, die veränderbare Zeichenketten verwaltet. Die Klasse wird im Namensraum *System.Text* bereitgestellt. Möchten Sie die Klasse nutzen, um veränderliche Zeichenkettenobjekte anzulegen, geht dies auf folgende Art:

```
Dim txt As New StringBuilder
```

Hier kommt der *New*-Konstruktor zum Anlegen der Objektinstanz zum Einsatz. Um dann dieser Variablen *txt* einen Text zuzuweisen, können Sie auf die Methode *Append()* der Klasse zurückgreifen:

```
txt.Append ("Hallo Welt")
```

Die *StringBuilder*-Klasse besitzt ähnliche Methoden wie die Klasse *String*. Details liefert die Hilfe des .NET Framework. Das folgende Listing demonstriert die Anwendung einiger Methoden dieser Klasse:

```
'*****
' File/Projekt: Beispiel5_10
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung der StringBuilder-Klasse.
'*****
Option Strict On
Option Compare Text

' Vereinbare die benutzten Namensräume
Imports System.Console          ' für WriteLine
Imports System.Text             ' für StringBuilder

Class Test                      ' Klasse mit Hauptmodul
    Shared Sub Main()
        Dim txt As New StringBuilder
        Dim newtxt As String

        txt.Append ("Hallo Welt")
        WriteLine (txt)          ' Text anzeigen

        txt.Replace ("Welt", "Günter")
        WriteLine (txt)
```

Listing 5.9: Verwenden der *StringBuilder*-Klasse

```

newtxt = txt.ToString          ' in String-Objekt
WriteLine (newtxt)

Write ("Bitte die Eingabtaete drücken")
ReadLine
End Sub

```

Listing 5.9: Verwenden der *StringBuilder*-Klasse (Forts.)**Hinweis**

Der Vorteil der Klasse besteht darin, dass dynamische Zeichenketten wesentlich schneller als mit der Klasse *String* vergrößert werden. Zudem lässt sich beim Anlegen eines Objekts die Zeichenzahl angeben. Dies erlaubt eine effizientere Verwaltung der Zeichenketten. Einsteiger können die *StringBuilder*-Klasse ignorieren und mit der Klasse *String* arbeiten. Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap05\Beispiel5_10` auf der Begleit-CD.

5.5 Arbeiten mit Feldern

Nachfolgend möchte ich noch kurz auf die Methoden der Klasse *Array* aus dem Namensraum *System* eingehen. Diese ergänzt die Möglichkeit zum Anlegen und Nutzen von Feldern (Kapitel 4).

5.5.1 Felder kopieren und neu initialisieren

Als Erstes möchte ich auf die Frage eingehen, wie sich Felder kopieren lassen und was es zu beachten gilt. Zur Demonstration soll an dieser Stelle ein einfaches Beispielprogramm dienen, welches drei eindimensionale Felder *a(2)*, *b(2)* und *c(2)* aufweist. Da bisher keine Schleifen besprochen wurden, habe ich die Felder auf drei Elemente begrenzt. Die Deklaration der Felder sieht jetzt folgendermaßen aus:

```

Dim a() As Integer = {1, 2, 3}
Dim b(2) As Integer
Dim c(2) As Integer

```

Nichts Großartiges, Sie kennen dies bereits aus den vorhergehenden Kapiteln. Jetzt sollen ein paar Operationen auf diesen Feldern ausgeführt werden. So ließe sich der Inhalt von Feld *a* nach Feld *b* kopieren, dann könnten die Werte der Elemente von Feld *b* überschrieben werden etc. Um das Verhalten von Visual Basic 2005 zu untersuchen, habe ich ein kleines Beispielprogramm *Beisp06_11.vb* erstellt, welches nach jedem Schritt die Werte der betreffenden Felder auf der Konsolenebene ausgibt. In Abbildung 5.7 sind die einzelnen Schritte mit Nummern von 1 bis 5 versehen. Die Ergebnisse für einen dieser Schritte lassen sich beispielsweise mit folgenden Anweisungen ausgeben:

```

WriteLine ("### Originalwerte ###")
WriteLine ("Ursprung Feld a {0}, {1}, {2}", a(0), a(1), a(2))
WriteLine ("Ursprung Feld b {0}, {1}, {2}", b(0), b(1), b(2))

```

Aus Abbildung 5.7 lässt sich erkennen, dass im ersten Schritt die Initialisierungswerte der Felder angezeigt werden. Feld *a* weist Werte auf, während die Elemente von Feld *b* noch mit Nullen belegt sind (das haben wir erwartet).

Nun soll das Feld *a* der Feldvariablen *b* zugewiesen werden. Diese Zuweisung lässt sich mit dem Befehl

```
b = a
```

bewerkstelligen. Wird die Klammer für den Feldindex weggelassen, bearbeitet Visual Basic 2005 das komplette Feld. Da beide Feldvariablen eindimensionale Felder gleicher Größe darstellen, klappt diese Zuweisung. Das Feld *b* weist nun alle Werte des Feldes *a* auf (siehe Abbildung 5.7). Ein weiterer Schritt dient jetzt dazu, einzelne Feldelemente in der Feldvariablen *b* mit neuen Werten zu belegen. Hierzu dienen die folgenden Anweisungen:

```

b(1) = 3
b(2) = 10

```

Ich habe hier nur zwei Feldelemente geändert. Das in Abbildung 5.7 in Schritt 3 gezeigte Ergebnis überrascht dann doch vielleicht etwas. Obwohl wir Elemente der Feldvariablen *b* geändert haben, wirkt sich dies auch auf den Inhalt der Feldvariablen *a* aus!

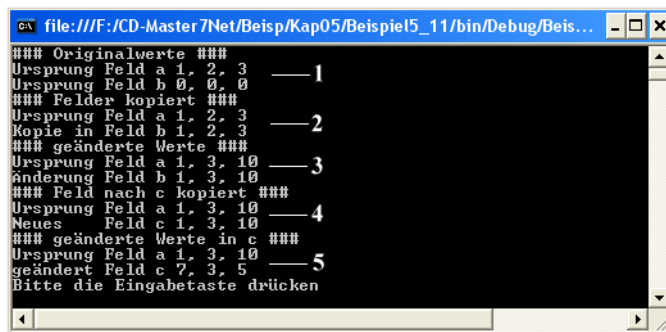


Abbildung 5.7: Kopieren und Anpassen von Feldern

Achtung

Verwenden Sie in Visual Basic eine Zuweisung der Art $b = a$ zwischen Feldern, überträgt das Laufzeitsystem nur den Verweis auf die Variable *b* in die neue Variable *a*. Es gibt dann zwei Variablen, die auf die gleichen Werte zeigen! Eine Änderung an einzelnen Werten wirkt sich daher auf beide Variable aus.

Möchten Sie verhindern, dass die beiden Feldvariablen auf die gleichen Daten zeigen, müssen Sie die Feldelemente einzeln kopieren. Dies könnte für das behandelte Beispiel mit folgenden Anweisungen erfolgen:

```
c(0) = a(0)
c(1) = a(1)
c(2) = a(2)
```

Der zweite Vorteil dieses Ansatzes besteht darin, dass auch nur Teile des Quellfeldes in das Zielfeld übernommen werden können (z.B. weil das Zielfeld weniger Elemente aufweist). Werden die Feldelemente von *c* geändert, wirkt sich dies nicht auf die Elemente von Feld *a* aus. Sie können dies in Schritt 5 in Abbildung 5.7 erkennen. Der komplette Beispielcode ist dem folgenden Listing zu entnehmen:

```
'*****
' File/Projekt: Beispiel5_11
' Autor: G. Born www.borncity.de
' Demonstriert das Kopieren von Feldern.
'*****
Option Strict On
Option Compare Text

' Vereinbare die benutzten Namensräume
Imports System.Console          ' für WriteLine

Class Test                      ' Klasse mit Hauptmodul
    Shared Sub Main()
        Dim a() As Integer = {1, 2, 3} ' ein einfaches Feld
        Dim b(2) As Integer
        Dim c(2) As Integer

        ' Zeige die Werte der beiden Felder
        WriteLine ("### Originalwerte ###")
        WriteLine ("Ursprung Feld a {0}, {1}, {2}", a(0), a(1), a(2))
        WriteLine ("Ursprung Feld b {0}, {1}, {2}", b(0), b(1), b(2))

        b = a                    ' das Feld zuweisen (Verweis kopieren)
        WriteLine ("### Felder kopiert ###")
        WriteLine ("Ursprung Feld a {0}, {1}, {2}", a(0), a(1), a(2))
        WriteLine ("Kopie in Feld b {0}, {1}, {2}", b(0), b(1), b(2))

        b(1) = 3                 ' Ändere Werte in b
```

Listing 5.10: Beispiel zur Feldbearbeitung

```

b(2) = 10
WriteLine ("### geänderte Werte ###")
WriteLine ("Ursprung Feld a {0}, {1}, {2}", a(0), a(1), a(2))
WriteLine ("Änderung Feld b {0}, {1}, {2}", b(0), b(1), b(2))

c(0) = a(0)           ' Feldinhalte nach c kopieren
c(1) = a(1)
c(2) = a(2)

WriteLine ("### Feld nach c kopiert ###")
WriteLine ("Ursprung Feld a {0}, {1}, {2}", a(0), a(1), a(2))
WriteLine ("Neues Feld c {0}, {1}, {2}", c(0), c(1), c(2))

c(0) = 7               ' Ändere Werte in c
c(2) = 5

WriteLine ("### geänderte Werte in c ###")
WriteLine ("Ursprung Feld a {0}, {1}, {2}", a(0), a(1), a(2))
WriteLine ("geändert Feld c {0}, {1}, {2}", c(0), c(1), c(2))
Write ("Bitte die Eingabtaste drücken")
ReadLine
End Sub
End Class

```

Listing 5.10: Beispiel zur Feldbearbeitung (Forts.)

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap05\Beispiel5_11` auf der Begleit-CD. Führen Sie die Anwendung im Fenster der Eingabeaufforderung aus.

Bei umfangreicheren Feldern wird das Kopieren aller Feldelemente mit diesem Ansatz, selbst bei Verwendung der im nächsten Kapitel vorgestellten Schleifen, etwas aufwendig. Um große Felder effizienter zu kopieren, bietet die .NET-Framework-Klassenbibliothek eine Lösung. Die Klasse *Array* im Namensraum *System* stellt Methoden bereit, um den Inhalt eines Felds in ein anderes Feld zu übertragen.

- *Copy()*: Kopiert einen Abschnitt eines Array in ein anderes Feld und führt erforderliche Typumwandlungen durch.
- *CopyTo()*: Die Methode kopiert alle Elemente des aktuellen eindimensionalen Array in das angegebene eindimensionale Array. Im Zielfeld lässt sich ein Index vorgeben, ab dem die Elemente einzuordnen sind.
- *Clone()*: Diese Methode legt eine Kopie des Feldes unter neuem Namen an. Dabei werden aber nur die Referenzen auf die Daten kopiert.

Mit der *Clear()*-Methode lässt sich dagegen ein Feld oder ein Teil davon initialisieren, d.h. auf Nullwerte zurücksetzen. In der Hilfe finden sich auch Beispiele in Visual Basic, die das Anlegen von Objektvariablen vom Datentyp *Array* demonstrieren.

Viel interessanter ist aber die Frage, ob und wie sich die Methoden der *Array*-Klasse auf Felder anwenden lassen, die mit einer *Dim*-Anweisung der folgenden Art vereinbart wurden und ob dies unser Problem beim Kopieren löst:

```
Dim a() As Integer = {1, 2, 3}
Dim b(2) As Integer
```

Die Methoden der *Array*-Klasse finden sich in der Klasse *Array* des Namensraums *System*. Sie brauchen diesen Namensraum aber nicht mit *Imports* anzugeben. Sobald Sie eine Feldvariable deklarieren, können die Methoden und Eigenschaften benutzt werden. Zum Kopieren eines eindimensionalen Feldes in eine zweite Feldvariable lässt sich die *Copy()*-Methode verwenden. Die Methode ist folgendermaßen aufzurufen:

```
Copy(Quellfeld, Zielfeld, Länge)
```

Die ersten beiden Parameter nehmen die Felder auf, der dritte Parameter gibt die Zahl der zu kopierenden Elemente an. Möchten Sie mehrdimensionale Felder kopieren, müssen diese die gleichen Dimension aufweisen. Die Methode interpretiert das mehrdimensionale Feld einfach wie ein eindimensionales Feld und kopiert alle Elemente auf einen Rutsch. Der Aufruf sieht folgendermaßen aus:

```
Copy(Quellfeld, Index, Zielfeld, Index, Länge)
```

Die Parameter *Index* geben einen Index im Quell- und Zielfeld an, ab dem bzw. zu dem die Elemente zu kopieren sind. *Länge* definiert die Zahl der zu kopierenden Elemente. Um das komplette Feld zu kopieren, muss *Länge* entsprechend gesetzt werden. Andernfalls kopiert die Methode zuerst die Elemente der ersten Zeile, dann die Elemente der nächsten Zeile, bis die angegebene Zahl erreicht ist.

Der zweite Aspekt betrifft das Löschen der Inhalte der Feldvariablen. Ein Feld wird bei der Definition automatisch initialisiert. Je nach Datentyp werden die Feldelemente auf den Wert 0, *False* oder auf *Nothing* (bei Objekten) gesetzt. Alternativ haben Sie die Möglichkeit, ein Feld bei der Deklaration mit Vorgabewerten zu belegen. Möchten Sie später ein komplettes Feld auf die Initialisierungswerte zurücksetzen, müssten Sie dies für jedes Feldelement einzeln tun.

```
Dim a() As Integer = {0, 1}
...
a(0) = 0
a(1) = 0
```

Oder Sie verwenden Schleifen, um die Feldelemente zurückzusetzen. Allerdings stellt die .NET-Framework-Klassenbibliothek in der Klasse *Array* im Namensraum *System* eine Methode *Clear* für diesen Zweck bereit. Die Methode wird folgendermaßen aufgerufen:

```
Clear (Feld, Index, Länge)
```

Feld ist der Name der Feldvariablen, *Index* gibt den Indexwert an, ab dem zu initialisieren ist und *Länge* definiert die Zahl der zu initialisierenden Feldelemente. Dies soll jetzt an einem einfachen Beispiel ausprobiert werden. Es wird eine Feldvariable *a* angelegt und initialisiert. Dann ist deren Inhalt mittels der *Copy()*-Methode in eine zweite Feldvariable *b* zu kopieren. Nach Anpassung verschiedener Werte in der Feldvariablen *b* wird dieses Feld teilweise mittels der *Clear()*-Methode initialisiert. Bei jedem Schritt sind die Werte auf der Konsolenebene anzuzeigen, um die Auswirkungen jeder Änderung zu dokumentieren. In Abbildung 5.8 sehen Sie die Ausgaben des Beispielprogramms.

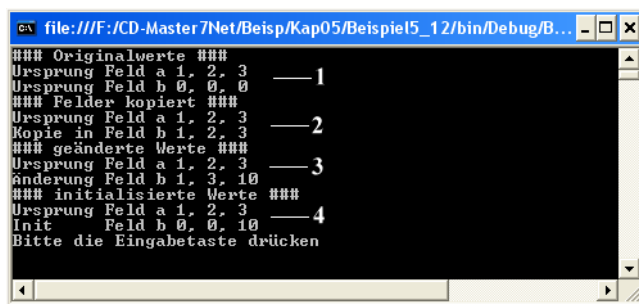


Abbildung 5.8: Kopieren und Initialisieren von Feldern

Die Schritte 1 und 2 dokumentieren die Werte der beiden Felder *a* und *b*, die folgendermaßen deklariert wurden:

```
Dim a() As Integer = {1, 2, 3}
Dim b(2) As Integer
```

In Schritt 2 wurde lediglich das Feld *a* mittels der *Copy()*-Methode in Feld *b* kopiert. In Schritt 3 setzt das Programm die Werte einzelner Feldelemente in *b* um. Die Ausgabe zeigt, dass sich dies nicht auf Feld *a* auswirkt. Schritt 4 zeigt die Auswirkung der *Clear()*-Methode auf Feld *b*, bei deren Anwendung die beiden ersten Feldelemente neu initialisiert werden. Auch hier gibt es keine Auswirkungen auf Feld *a*. Mittels der *Copy()*-Methode erhalten Sie also ein sehr mächtiges Werkzeug, um komplette Felder oder deren Elemente zu kopieren. Die *Copy()*-Methode erlaubt eine sehr elegante Initialisierung der Felder mit Nullwerten. Weitere Details zum Umgang mit den Methoden sind dem nachfolgenden Listing mit dem Beispielcode zu entnehmen:

```
'*****
' File/Projekt: Beispiel5_12
' Autor: G. Born www.borncity.de
' Demonstriert das Kopieren und das Initialisieren von
' Feldern über die Array-Klasse.
'*****
```

Listing 5.11: Beispiel zur Feldbearbeitung

```
Option Strict On
Option Compare Text

' Vereinbare die benutzten Namensräume
Imports System.Console          ' für WriteLine
Imports System.Array            ' für Array()-Methoden

Class Test                      ' Klasse mit Hauptmodul
    Shared Sub Main()
        Dim a() As Integer = {1, 2, 3} ' ein einfaches Feld
        Dim b(2) As Integer

        ' Zeige die Werte der beiden Felder
        WriteLine ("### Originalwerte ###")
        WriteLine ("Ursprung Feld a {0}, {1}, {2}", a(0), a(1), a(2))
        WriteLine ("Ursprung Feld b {0}, {1}, {2}", b(0), b(1), b(2))

        Copy(a, b, a.Length)          ' Das Feld kopieren
        WriteLine ("### Felder kopiert ###")
        WriteLine ("Ursprung Feld a {0}, {1}, {2}", a(0), a(1), a(2))
        WriteLine ("Kopie in Feld b {0}, {1}, {2}", b(0), b(1), b(2))

        b(1) = 3                      ' Ändere Werte in b
        b(2) = 10
        WriteLine ("### geänderte Werte ###")
        WriteLine ("Ursprung Feld a {0}, {1}, {2}", a(0), a(1), a(2))
        WriteLine ("Änderung Feld b {0}, {1}, {2}", b(0), b(1), b(2))

        System.Array.Clear (b,0,2)    ' init erste zwei Elemente neu

        WriteLine ("### initialisierte Werte ###")
        WriteLine ("Ursprung Feld a {0}, {1}, {2}", a(0), a(1), a(2))
        WriteLine ("Init    Feld b {0}, {1}, {2}", b(0), b(1), b(2))

        Write ("Bitte die Eingabtaste drücken")
        ReadLine()
    End Sub
End Class
```

Listing 5.11: Beispiel zur Feldbearbeitung (Forts.)

Hinweis

Im Listing ist noch eine Besonderheit zu finden. Die *Clear()*-Methode kommt sowohl in der Klasse *System.Array* als auch in *System.Console* (allerdings mit unterschiedlicher Bedeutung) vor. Der Visual-Basic-2005-Compiler erkennt dies und weist Sie beim Erstellen der Anwendung auf dieses Problem hin. Aus diesem Grund wurde der Namensraum *System.Array* beim Aufruf der *Clear()*-Methode mit angegeben. Sie finden die Projektdateien im Ordner `\Beisp\Kap05\Beispiel5_12` auf der Begleit-CD. Die Projektdateien im Ordner `\Beisp\Kap05\Beispiel5_12a` demonstrieren zur Vertiefung, wie Sie Felder auf Basis der *Array*-Klasse aufbauen und Methoden wie *Copy()* oder *Clear()* anwenden.

5.5.2 Suchen in Feldern

Felder eignen sich hervorragend, um Daten zu speichern. Denken Sie beispielsweise an eine Namens- oder Telefonliste. Naheliegender ist es daher, wenn nach Funktionen zum Suchen nach bestimmten Werten innerhalb eines Feldes gefragt wird. Standardmäßig sind alle Elemente eines Feldes manuell mit dem Suchmuster zu vergleichen. In Visual Basic 2005 gibt es elegantere Möglichkeiten, indem Sie auf die Methoden der *Array*-Klasse zurückgreifen. Dies soll jetzt an einem kleinen Beispielprogramm demonstriert werden.

Als Erstes möchte ich die *BinarySearch()*-Methode in einem Beispiel verwenden. Diese Methode führt eine Suche in einem eindimensionalen Feld durch. Die Feldelemente können dabei verschiedene Datentypen (Integer, String etc.) aufweisen, sollten aber von den Werten beim Suchen Sinn machen (eine Suche nach Dezimalzahlen ist wohl selten sinnvoll).

Achtung

Die *BinarySearch()*-Methode verwendet einen binären Algorithmus zur Suche. Dieser Algorithmus teilt die Werteliste in Suchintervalle ein, die bei jedem Schritt halbiert werden. Dann vergleicht der Algorithmus, ob der Suchbegriff größer oder kleiner als der Wert in der Mitte des Intervalls ist. Anschließend wird die Intervallhälfte bearbeitet, in der der Wert liegen kann. Das Ganze lässt sich mit der Suche eines Namens in einem Telefonbuch vergleichen, bei dem man im ersten Schritt genau die Mitte aufschlägt. Dann wird entschieden, ob der Name in der vorderen oder hinteren Hälfte vorkommen kann. Im nächsten Schritt wird die gewählte Hälfte wieder geteilt und der Vorgang wiederholt sich. Dieses Verfahren ist sehr einfach zu implementieren und führt zu schnellen Ergebnissen. Es setzt aber voraus, dass die Elemente des Feldes in sortierter Reihenfolge vorliegen!

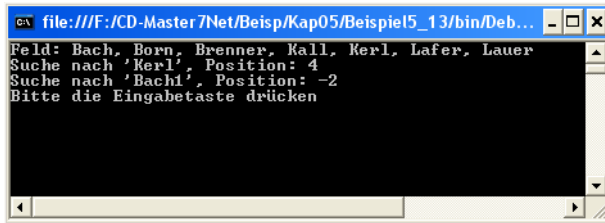


Abbildung 5.9: Suchen in Feldern

Die Abbildung 5.9 zeigt die Ausgaben des Beispielsprogramms. Die erste Zeile enthält die im Feld gespeicherten Einträge in sortierter Form. Die beiden Folgezeilen weisen den jeweiligen Suchbegriff sowie den Index, an dem der Begriff gefunden wurde, auf. Bei der Suche nach »Kerl« wird der Index 4 als Position zurückgemeldet. Da die Indizierung bei Feldern mit 0 beginnt, ist »Kerl« der fünfte Eintrag im Feld. In der nächsten Zeile meldet das Programm, dass der Begriff »Bach1« gesucht wurde. Als Position meldet das Programm den Wert -2, d.h., der Begriff wurde nicht gefunden. Schauen wir uns jetzt einmal den Beispielcode an. Das Feld mit den sortierten Einträgen lässt sich folgendermaßen definieren:

```
Dim name() As String = _
    {"Bach", "Born", "Brenner", _
     "Kall", "Kerl", _
     "Lafer", "Lauer"}
```

Zum Suchen nach dem Begriff »Kerl« im Feld *Name* lässt sich dann folgende Anweisung verwenden:

```
i = BinarySearch(name, muster)
```

Voraussetzung zum Aufruf der Methode *BinarySearch()* ist aber, dass die Klasse *System.Array* im Programm verfügbar gemacht wird.

```
'*****
' File/Projekt: Beispiel5_13
' Autor: G. Born www.borncity.de
' Demonstriert die Suche in einem Feld über die
' Array-Klasse.
'*****
Option Strict On
Option Compare Text

' Vereinbare die benutzten Namensräume
Imports System.Console      ' für WriteLine
Imports System.Array        ' für Array()-Methoden
```

Listing 5.12: Beispiel zum Suchen in Feldern

```

Class Test                                ' Klasse mit Hauptmodul
Shared Sub Main()
' Deklariere ein Testfeld, welches sortierte Einträge hat
Dim name() As String = _
    {"Bach", "Born", "Brenner", _
    "Kall", "Kerl", _
    "Lafer", "Lauer"}

Dim i As Integer                          ' Index des Ergebnisses
Dim muster As String = "Kerl"             ' Suchbegriffe
Dim muster1 As String = "Bach1"

' Zeige den Inhalt des Feldes
WriteLine ("Feld: {0}, {1}, {2}, {3}, {4}, {5}, {6}", name)

' rufe jetzt die BinarySearch()-Methode auf
' Ergebnis: Wert > 0 ist der Index des Eintrags
'           Wert < 0 Wert nicht gefunden
i = BinarySearch(name, muster)
' Zeige das Ergebnis an
WriteLine ("Suche nach '{0}', Position: {1}", muster, i)

i = BinarySearch(name, muster1)
' Zeige das Ergebnis an
WriteLine ("Suche nach '{0}', Position: {1}", muster1, i)

Write ("Bitte die Eingabtaste drücken")
ReadLine()
End Sub
End Class

```

Listing 5.12: Beispiel zum Suchen in Feldern (Forts.)

Hinweis

Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap05\Beispiel5_13` auf der Begleit-CD.

Suche in unsortierten Feldern

Nicht immer ist sichergestellt, dass der Inhalt eines Feldes sortiert vorliegt. Im nächsten Abschnitt erfahren Sie zwar, wie sich eindimensionale Felder sortieren lassen. Aber gelegentlich ist die Suche nach einem Begriff in einem unsortierten Feld erforderlich. Bevor Sie jetzt eine eigene Lösung stricken, die alle Feldelemente mit dem Suchbegriff ver-

gleich, möchte ich Ihnen eine Alternative zeigen. Die Klasse *Array* im Namensraum *System* bietet neben der Methode *BinarySearch Copy()* noch die Methode *IndexOf()*. Eine gleichnamige Methode kennen Sie bereits aus den vorherigen Abschnitt von der Klasse *String*. Dort ließ sich mit der Methode nach einem Muster in einer Zeichenkette suchen. Bei Feldern durchläuft die Methode *IndexOf()* alle Feldelemente, bis das letzte Element erreicht oder der Suchbegriff gefunden wurde. Das nachfolgende Listing demonstriert, wie sich die Suche über *IndexOf()* gestalten lässt. Ich habe die Initialisierungswerte des Feldes in unsortierter Reihenfolge angegeben.

```

'*****
' File/Projekt: Beispiel5_14
' Autor: G. Born www.borncity.de
' Demonstriert die Suche in einem Feld über die
' IndexOf()-Methode der Array-Klasse.
'*****
Option Strict On
Option Compare Text

' Vereinbare die benutzten Namensräume
Imports System.Console          ' für WriteLine
Imports System.Array            ' für Array()-Methoden

Class Test                      ' Klasse mit Hauptmodul
    Shared Sub Main()
        ' Deklariere ein Testfeld mit unsortierten Einträgen
        Dim name() As String = _
            {"Born", "Kerl", "Brenner", "Kall", "Lafer", "Bach", "Lauer"}

        Dim i As Integer        ' Index des Ergebnisses
        Dim muster As String = "Kerl" ' Suchbegriffe
        Dim muster1 As String = "Bach1"

        ' Zeige den Inhalt des Feldes
        WriteLine ("Feld: {0}, {1}, {2}, {3}, {4}, {5}, {6}", name)

        ' rufe jetzt die BinarySearch()-Methode auf
        ' Ergebnis: Wert > 0 ist der Index des Eintrags
        '           Wert < 0 Wert nicht gefunden
        i = IndexOf(name, muster)
        ' Zeige das Ergebnis an
        WriteLine ("Suche nach '{0}', Position: {1}", muster, i)
    End Sub
End Class

```

Listing 5.13: Beispiel zur linearen Suche in Feldern

```

i = BinarySearch(name, muster1)
' Zeige das Ergebnis an
WriteLine ("Suche nach '{0}', Position: {1}", muster1, i)

Write ("Bitte die Eingabetaste drücken")
ReadLine()
End Sub
End Class

```

Listing 5.13: Beispiel zur linearen Suche in Feldern (Forts.)

Hinweis

Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap05\Beispiel5_14` auf der Begleit-CD. Beachten Sie aber, dass beide Beispiele zum Suchen in Feldern einen kleinen Nachteil aufweisen: Erfolgt die Suche in Texten, wird nach der Groß-/Kleinschreibung unterschieden. Benötigen Sie eine Suchfunktion für Felder mit Textinhalten, die keine Unterscheidung der Groß-/Kleinschreibweise besitzt, müssen Sie diese entweder selbst schreiben, oder Sie kopieren die einzelnen Feldelemente und wandeln die Texte vor dem Vergleich in eine einheitliche Schreibweise um. Im Ordner `\Beisp\Kap05\Beispiel5_14a` der Begleit-CD wird dieser erweiterte Ansatz demonstriert.

5.5.3 Sortieren von Feldern

Zur binären Suche innerhalb eines Feldes mittels der `BinarySearch()`-Methode muss das Feld sortiert sein. Sicherlich stoßen Sie mit der Zeit auf weitere Fälle, wo eine Funktion zum Sortieren von Daten ganz angenehm wäre. Die `Array`-Klasse kann mit einer entsprechenden `Sort()`-Methode aufwarten, die den Inhalt eines eindimensionalen Feldes mithilfe der `IComparable`-Implementierung jedes Elements des Array anordnet. Das folgende Beispielprogramm demonstriert, wie Sie diese Methode für eigene Zwecke nutzen können.

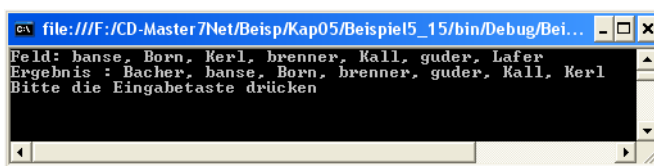


Abbildung 5.10: Sortieren eines Feldes

Die Abbildung 5.10 zeigt die Anweisungen zum Übersetzen und Aufrufen des Beispielsprogramms sowie dessen Ausgaben. In der ersten Ergebniszeile werden noch die unsortierten Textwerte des Feldes ausgegeben. Dabei wurden Wörter mit gemischter Groß-/Kleinschreibung benutzt. In der Folgezeile finden Sie die Anzeige der sortierten Werte. Der `Sort()`-Methode wird nur das Feld als Parameter übergeben. Weitere Details sind dem folgenden Listing zu entnehmen:

```
'*****
' File/Projekt: Beispiel5_15
' Autor: G. Born www.borncity.de
' Demonstriert das Sortieren eines Felds mit
' Texteinträgen über die Sort()-Methode der
' Array-Klasse.
'*****

Option Strict On
Option Compare Text

Imports System.Environment      ' für newline-Eigenschaft
Imports System.Console         ' für WriteLine
Imports System.Array           ' für Array()-Methoden

Class Test                      ' Klasse mit Hauptmodul
    Shared Sub Main()
        ' Deklariere ein Testfeld mit unsortierten Einträgen
        Dim name() As String = _
            {"banse", "Born", "Kerl", "brenner", "Kall", "guder", "Lafer", "Bacher", "lauer"}

        ' Zeige den Inhalt des Feldes
        WriteLine ("Feld: {0}, {1}, {2}, {3}, {4}, {5}, {6}", name)

        Sort(name)      ' Sortiere den Feldinhalt
        ' Zeige das Ergebnis
        WriteLine ("Ergebnis : {0}, {1}, {2}, {3}, {4}, {5}, {6}", name)
        Write ("Bitte die Eingabtaste drücken")
        ReadLine()
    End Sub
End Class
```

Listing 5.14: Beispiel zum Sortieren von Feldern

Hinweis

Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap05\Beispiel5_15` auf der Begleit-CD.

Damit möchte ich das Kapitel schließen. Sie kennen nun Zuweisungen sowie die Operatoren in Visual Basic 2005. Weiterhin haben Sie einige Spezialitäten im Umgang mit Strings und Feldern kennen gelernt. Im nächsten Kapitel beschäftigen wir uns mit Kontrollanweisungen und Schleifen, die Visual Basic 2005 bietet. Damit können Sie bedingte Programmabläufe und Wiederholungen programmieren.

6 Steueranweisungen, Prozeduren und Fehlerbehandlung

Zur Steuerung der Programmabläufe bietet Visual Basic 2005 Befehle, mit denen eine bedingte Ausführung verschiedener Codeabschnitte möglich ist. Zudem gibt es Anweisungen, mit denen sich Schleifen realisieren lassen, die eine bedingte Wiederholung einer Codesequenz ermöglichen. In diesem Kapitel möchte ich auf diese Anweisungen eingehen und die Verwendung an Beispielen demonstrieren. Zusätzlich lernen Sie weitere Anweisungen wie *With* oder den Aufruf von Prozeduren und Funktionen kennen. Ein weiterer Abschnitt geht auf das Thema »Fehlerbehandlung« ein, mit denen sich Laufzeitfehler (z.B. fehlende Dateien) abfangen lassen. Dieses Wissen erlaubt Ihnen, beliebig komplexe Programme zu erstellen.

6.1 Arbeiten mit der With-Anweisung

Als Erstes möchte ich auf eine Art Komfortfunktion zu sprechen kommen, die das Arbeiten mit Objekten erleichtert. Nehmen wir einmal eines der in den vorhergehenden Kapiteln beschriebenen Beispiele, in dem ein benutzerdefinierter Datentyp vereinbart wurde.

```
Public Structure Personaldaten
    Dim Name As String
    Dim Vorname As String
End Structure
```

Dann lässt sich innerhalb einer Prozedur eine Variable auf diesem Datentyp definieren und später verwenden.

```
Dim PersonA, PersonB As Personaldaten
...
PersonA.Name = "Born"
PersonA.Vorname = "Klaus"
```

Beim Zugriff auf die einzelnen Mitglieder des Datentyps *Personaldaten* muss jedes Mal der Name der Variablen, gefolgt von einem Punkt und dem Namen des Members, angegeben werden. Bei der Variablen *PersonA* hält sich der Schreibaufwand noch in Grenzen. Je häufiger solche Ausdrücke aber im Programmcode vorkommen, um so aufwändiger wird die Sache. Die *With*-Anweisung erlaubt es, eine Reihe von Anweisungen zum Zugriff auf ein Objekt zu einem Block zusammenzufassen, wobei zum Zugriff der Name dieses Objekts nicht mehr angegeben werden muss. Die Anweisung hat die Form:

```
With Objektname  
... Anweisungen  
End With
```

Im hier gezeigten Code ist die Variable *PersonA* der Name des Objekts. Zum Zugriff auf die Member dieses Objekts muss folglich so etwas wie *PersonA.Membername* benutzt werden. Innerhalb des *With*-Blocks kann der Objektname entfallen, der Zugriff auf ein Mitglied reduziert sich also auf die Angabe *.Membername*. Für den hier gezeigten Beispielcode lässt sich der Zugriff auf die Member des Objekts *PersonA* folgendermaßen schreiben:

```
With PersonA  
  .Name = "Bach"  
  .Vorname = "Anne"  
End With
```

Im Block kommen also nur noch die Namen der Member mit einem vorangestellten Punkt vor. Beim Zugriff auf Objekte erleichtert dies das Schreiben des Programmcodes ungemein. Zudem erlaubt die *With*-Anweisung dem Compiler die Optimierung des Codes.

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap06\Beispiel6_01`. Das Beispiel demonstriert den Zugriff auf Objekte und deren Mitglieder mittels *With*.

6.2 Bedingte Anweisungen (If-Anweisungen)

Bedingte Anweisungen für Verzweigungen steuern den Ablauf innerhalb der Programme und erlauben abhängig von einer bestimmten Bedingung einen Programmzweig auszuführen. Der nachfolgende Abschnitt enthält eine kurze Übersicht über die für diese Zwecke in Visual Basic 2005 verfügbaren Anweisungen.

6.2.1 If...Then-Anweisung

Die einfachste Anweisung für Verzweigungen wird durch das Schlüsselwort *If* eingeleitet und mit dem Schlüsselwort *Then* in der gleichen Zeile abgeschlossen. Man spricht auch von einer *If...Then*-Anweisung. Diese *If*-Anweisung erlaubt es, Programmteile oder einzelne Anweisungen in Abhängigkeit von einer vorgebbaren Bedingung auszuführen. Nur wenn die Bedingung erfüllt ist, wird der betreffende Code des Blocks ausgeführt. Andernfalls setzt das Programm die Ausführung des Codes am Ende des *If*-Blocks fort. Die einfachste Variante der *If*-Anweisung sieht so aus:

```
If a > 100 Then a = 100
```

Die komplette Anweisung besteht aus einer Zeile, die mit *If* eingeleitet wird und das Schlüsselwort *Then* enthält. Der Code mit der Zuweisung $a = 100$ wird nur dann ausgeführt, wenn die Vergleichsoperation $a > 100$ den Wert *True* zurückliefert. Die obige Anweisung setzt die Variable a auf den Wert 100 zurück, falls der Wert von a beim Vergleich größer als 100 ist. In dieser Form kann die *If*-Anweisung nur einen Befehl umfassen.

Bei Bedarf lässt sich die *If*-Anweisung auch so gestalten, dass sie mehrere Anweisungen umfasst. Dann muss der Block mit der *If*-Anweisung eingeleitet und mit den Schlüsselworten *End If* abgeschlossen werden. Die folgende Sequenz verdeutlicht diesen Ansatz:

```
If a > 100 Then
  a = 100
  b = 20
End If
```

Innerhalb der Verzweigung wird die Vergleichsoperation $a > 100$ ausgeführt. Liefert der Vergleich das Ergebnis *True*, d.h., ist a größer 100, werden die Anweisungen zwischen *If...Then* und *End If* ausgeführt. Trifft die Bedingung nicht zu, wird das Programm hinter der *End If*-Zeile fortgesetzt.

6.2.2 If...Then...Else

Die obige *If*-Anweisung weist nur einen bedingt auszuführenden Zweig auf. Gelegentlich kommt es vor, dass man alternative Codeblöcke ausführen möchte. Trifft eine Bedingung zu, wird Block 1 bearbeitet, andernfalls kommt Block 2 zur Ausführung. Die folgende Sequenz nutzt diesen Ansatz:

```
If a > 100 Then
  ' Block 1
  a = 100
  b = 20
Else
  ' Block 2
  a = a + 10
  b = a \ 10
End If
```

Im Eingang der Verzweigung überprüft die *If*-Anweisung den Wert der Variablen a . Ist der Wert größer als 100, werden die Anweisungen zwischen *If...Then* und *Else* ausgeführt. Trifft die Bedingung nicht zu, werden die Anweisungen zwischen *Else* und *End If* bearbeitet.

6.2.3 If...Then...ElseIf

Diese Variante der *If*-Anweisung erlaubt eine Schachtelung mehrerer *If*-Blöcke. Die folgende Sequenz wendet diese Konstruktion an:

```
If a = 1 Then
  b = 100
  c = 20
ElseIf a = 2 Then
  b = 200
  c = 40
ElseIf a = 3 Then
  b = 300
  c = 60
End If
```

Hier wird die Variable *a* auf verschiedene Werte überprüft. Liefert ein Vergleich den Wert *True*, werden die Bedingungen zwischen *ElseIf* und der nächsten *ElseIf*- oder *End If*-Anweisung ausgeführt. Sie sollten die *Select Case*-Anweisung vorziehen, da diese etwas besser lesbar ist.

6.3 Die Select-Case-Anweisung

Die *Select Case*-Anweisung ermöglicht die Angabe mehrerer Blöcke, die in Abhängigkeit von verschiedenen Bedingungen auszuführen sind. Mit dieser Anweisung können Sie also eine Variable auf mehrere Werte überprüfen und in Abhängigkeit vom Wert bestimmte Anweisungsblöcke formulieren. Die folgende Sequenz wendet diese Konstruktion an:

```
Select Case a
  Case 1
    b = 100
    c = 20
  Case 2
    b = 200
    c = 40
  Case 3
    b = 300
    c = 60
  Case Else
    b = 0
    c = 0
    a = 1
End Select
```

Hier wird die Variable *a* ebenfalls ausgewertet. Die nachfolgenden *Case*-Anweisungen geben den zu überprüfenden Wert vor. Trifft die Bedingung zu, werden die Anweisungen hinter der *Case*-Anweisung bis zur nächsten *Case*-Anweisung ausgeführt. Trifft keine der Abfragen zu, wird der (optionale) *Case Else*-Zweig ausgeführt.

Hinweis

Sie finden die Projektdateien, die die Anwendung einer *Select Case*-Anweisung demonstrieren, im Ordner `\Beisp\Kap06\Beispiel6_02` der Begleit-CD.

6.3.1 Sprünge mit der GoTo-Anweisung

Visual Basic 2005 unterstützt zwar einige der aus Visual Basic 6 bekannten Sprunganweisungen wie *On GoTo* nicht mehr. Der absolute Sprung zu einer Marke gehört aber noch zum Befehlsumfang. Mit der Anweisung *GoTo L1* verzweigt das Programm sofort zur Marke *L1*. Eine Marke muss im Programm in der Form *Name:* vereinbart sein, wobei *Name* als Platzhalter für eine Zahl oder einen beliebigen Namen (gebildet aus den Regeln für Bezeichner) steht. Das folgende Programmfragment demonstriert die Verwendung dieser Anweisung:

```
Public Shared Sub Main()  
    Dim i As Integer  
  
L1:  
    Console.WriteLine ("Eine Zahl eingeben (1 = Abbruch)")  
    i = CInt (Console.ReadLine())  
    If i = 1 Then GoTo L2 ' Beenden  
    Console.WriteLine("Wert ist '{0}'", i)  
    GoTo L1 ' An Anfang zurück  
  
L2: Console.WriteLine("Ende der Eingabe")  
End Sub
```

Im Programm sind zwei Marken *L1* und *L2* vorhanden, die durch *GoTo*-Anweisungen adressiert werden. Das Programm liest eine Zahl vom Benutzer ein. Ist die Zahl = 1, bewirkt die *If*-Anweisung einen Sprung zur Marke *L2*, das Programm wird beendet. Andernfalls wird die Zeile mit der Anweisung

```
GoTo L1
```

erreicht. Dann verzweigt das Programm zu dieser Marke und fordert eine erneute Benutzereingabe an.

Hinweis

GoTo kann nur zu Zeilen innerhalb der umgebenden Prozedur verzweigen. Zudem dürfen Sie das Sprungziel einer *GoTo*-Anweisung nicht in einen Block wie *For...Next*, *For Each...Next*, *SyncLock...End SyncLock*, *Try...Catch...Finally* oder *With...End With* legen. Sie dürfen auch nicht in einen *Try...Catch...Finally*-Block zur Fehlerbearbeitung springen. Auch der Sprung aus einem *Catch*-Block heraus ist unzulässig. Da die Programmentransparenz bei Verwendung der *GoTo*-Anweisung leidet, sollten Sie in normalem Code möglichst auf diesen Befehl verzichten. Lediglich zur Fehlerbehandlung können Sprunganweisungen nützlich sein (siehe Kapitelende).

6.4 Schleifen

Schleifen erlauben die wiederholte Ausführung bestimmter Anweisungen innerhalb der Visual Basic-Programme. Wie bei den Verzweigungen kennt Visual Basic 2005 verschiedene Varianten an Schleifen, die nachfolgend kurz vorgestellt werden.

6.4.1 Do...Loop-Schleifen

Die beiden Schlüsselwörter *Do...Loop* klammern einen Block mit den Anweisungen der Schleifen:

```
Do
...
Loop
```

Um die Schleife definiert auszuführen, muss diese mit einer Abbruchbedingung versehen werden. Diese darf dabei am Schleifenkopf hinter dem Schlüsselwort *Do* stehen, kann aber auch am Schleifenende hinter dem Schlüsselwort *Loop* folgen. Zudem lässt sich das Abbruchkriterium mit dem Schlüsselwort *While* oder mit dem Schlüsselwort *Until* kombinieren. Dies ergibt unterschiedliche Varianten der Schleife, die nachfolgend an einfachen Aufgaben skizziert werden.

Do While...Loop-Schleife

Die *Do While*-Anweisung erzeugt eine Schleife, die in Abhängigkeit von einer am Schleifenanfang ausgewerteten Bedingung die Anweisungen des Blocks ausführt oder überspringt. Die Anweisungen der Schleife werden so lange ausgeführt, wie die Bedingung den Wert *True* liefert. Die folgende Sequenz zeigt den Einsatz dieser Anweisung:

```
a = 1
Do While a < 10
    a = a + 1
...
Loop
```

In dieser Schleife wird beim Eintritt die Bedingung $a < 10$ geprüft. Solange dies erfüllt ist, führt das System die Anweisungen bis zum Schlüsselwort *Loop* aus. Danach wird die Bedingung am Schleifenanfang erneut geprüft. Trifft die Bedingung nicht mehr zu, endet die Schleife, indem der auf die *Loop*-Anweisung folgende Befehl ausgeführt wird.

Achtung

Achten Sie bei Verwendung dieser Schleifenkonstruktion darauf, dass das Prüfkriterium beim Eintritt bereits definiert ist und die Werte *False* oder *True* liefern kann. Zudem ist sicherzustellen, dass das Abbruchkriterium irgendwann eintritt, damit die Schleife verlassen wird.

Do Until...Loop-Schleife

Die *Do Until*-Anweisung bewirkt ebenfalls eine Überprüfung beim Eintritt in die Schleife. Liefert die Bedingung den Wert *False*, wird der Code innerhalb der Schleife bis zur *Loop*-Anweisung ausgeführt. Anschließend wird wieder die Bedingung am Schleifenkopf geprüft. Die Schleife wird verlassen, sobald die Überprüfung der Bedingung den Wert *True* liefert. Die folgende Sequenz zeigt den Einsatz dieser Anweisung:

```
a = 1
Do Until a > 10
  a = a + 1
...
Loop
```

In dieser Schleife wird beim Eintritt die Bedingung $a > 10$ geprüft. Solange dies nicht erfüllt ist, führt das System die Anweisungen bis zur *Loop*-Anweisung aus.

Do...Loop While-Schleife

Schleifen, die die Anweisungsfolge *Do...Loop While* verwenden, werden erst am Schleifenende überprüft. Ist die Bedingung dieser Prüfung *True*, wird die Schleife weiter ausgeführt. Die Ausführung der Schleife endet, sobald die Bedingung den Wert *False* liefert. Die folgende Sequenz zeigt den Einsatz dieser Anweisung:

```
a = 1
Do
  a = a + 1
...
Loop While a < 10
```

In dieser Schleife wird die Bedingung $a < 10$ am Ende geprüft. Solange dies erfüllt ist, werden die Anweisungen der Schleife zwischen *Do* und *Loop* abgearbeitet. Trifft die Bedingung nicht mehr zu, endet die Schleife und der auf die *Loop*-Anweisung folgende Befehl wird ausgeführt.

Do...Loop Until-Schleife

Eine mit den Schlüsselwörter *Do...Loop Until* konstruierte Schleife wird ebenfalls erst am Schleifenende überprüft. Ergibt das Ergebnis dieser Prüfung den Wert *False*, werden die Anweisungen innerhalb der Schleife weiter ausgeführt. Die Schleife wird beendet, sobald die Bedingung den Wert *True* zurückliefert. Die folgende Sequenz zeigt den Einsatz dieser Anweisung:

```
a = 1
Do
  a = a + 1
...
Loop Until a > 10
```

In dieser Schleife wird am Ende die Bedingung $a > 10$ geprüft. Solange dies nicht erfüllt ist, beginnt die Abarbeitung wieder bei der *Do*-Anweisung. Ist der *Loop*-Befehl erreicht, wird die Bedingung am Schleifenende erneut geprüft. Liefert die Auswertung der Bedingung den Wert *True* zurück, endet die Schleife und der auf die *Loop*-Anweisung folgende Befehl wird ausgeführt.

6.4.2 For...Next-Schleife

Mit den Schlüsselwörtern *For* und *Next* lässt sich eine Schleife erzeugen, die eine vordefinierte Anzahl an Durchläufen erlaubt. Die bei jedem Durchlauf auszuführenden Anweisungen sind innerhalb des *For...Next*-Blocks einzufügen. Die folgende Sequenz zeigt den Einsatz dieser Anweisung:

```
For i = 1 To 10
  a = a + 1
Next i
```

Der Wert i wird als Schleifenindex bezeichnet. Es muss sich hierbei um eine Variable handeln, die üblicherweise vom Typ *Integer* ist, aber auch andere numerische Datentypen aufweisen kann. Der Schleifenindex muss im Schleifenkopf hinter *For* angegeben werden und kann auch am Schleifenende hinter *Next* stehen. Ein hinter *Next* aufgeführter Schleifenindex erhöht die Lesbarkeit des Quellcodes. Vor dem Schlüsselwort *To* steht der Startwert für die Schleifenindex. Der Endwert folgt hinter diesem Schlüsselwort. Dieser Schleifenindex beginnt in der obigen Sequenz mit dem Startwert 1 und wird bei jedem Durchlauf um 1 erhöht. Dies geht so lange, bis der Indexwert 10 erreicht. Beim Wechsel auf den Wert 11 endet die Schleife.

Hinweis

Innerhalb einer Schleife können Sie wiederum Anweisungen für zusätzliche Schleifen einbauen. Man spricht dann von geschachtelten Schleifen. Die innere Schleife wird dann bei jedem Durchlauf der äußeren Schleife komplett abgearbeitet.

Die Schrittweite des Schleifenindex wird standardmäßig auf 1 gesetzt. Visual Basic erlaubt aber die Schrittweite über das Schlüsselwort *Step* anzugeben. Die folgende Sequenz verwendet eine Schrittweite von 10:

```
For i = 1 To 100 Step 10
    a = a + 1
Next i
```

Handelt es sich beim Schleifenindex um eine Gleitkommazahl, können Sie auch die Schrittweite entsprechend angeben. Zusätzlich ist ein absteigender Schleifenindex möglich. In diesem Fall muss der Startwert größer als der Endwert sein, mit *Step* lässt sich die Schrittweite (z.B. *-1*) vorgeben.

6.4.3 Continue, eine Neuerung für Schleifen

Gelegentlich möchte man in einer Schleife bestimmte Anweisungen nicht mehr ausführen. Während in älteren Visual-Basic-Versionen eine *GoTo*-Anweisung oder andere Hilfskonstruktionen wie *If .. Then ... End If* erforderlich waren, haben die Entwickler in Visual Basic 2005 die *Continue*-Anweisung eingeführt. Die *Continue*-Anweisung bewirkt, dass die vor dem Ende der jeweiligen Schleife befindlichen Befehle übersprungen werden. Der folgende Codeausschnitt demonstriert die Anwendung von *Continue* in einer *For*-Schleife:

```
Dim i As Integer
Dim b As Integer = 1
For i = 1 To 100 Step 10
    b = b + i
    If b > 30 Then _
        Continue For
    MsgBox(i & ":" & b)
Next i
```

Die Schleife wird in Zehnerschritten durchlaufen, wobei der Wert des Schleifenindex sowie der Inhalt der Variablen *b* in einem Dialogfeld angezeigt werden. Sobald jedoch der Wert der Variablen *b* größer als 30 ist, soll die Anzeige der Ergebnisse in einem Dialogfeld unterbleiben. Hierzu wurde die *Continue*-Anweisung im *If ... Then*-Block eingebracht. Sobald der Ausdruck in der *If*-Anweisung »wahr« wird, bewirkt die *Continue*-Anweisung, dass alle im Block bis zur *Next*-Anweisungen folgenden Befehle übersprungen werden.

Hinweis

Sie finden ein Beispielprojekt, welches die *Continue*-Anweisung demonstriert, im Ordner `\Beisp\Kap06\Beispiel6_02` der Begleit-CD.

6.5 Schleifen für Felder und Enumerationen

In den vorhergehenden Kapiteln haben Sie bereits Felder kennen gelernt. Zur Bearbeitung von Feldern werden meist Schleifen herangezogen. Ein weiteres Gestaltungselement sind Enumerationen, die eine Sammlung von Elementen aufweisen. Der folgende Abschnitt zeigt Ihnen, was Enumerationen sind und wie sich Schleifen zur Bearbeitung von Feldern und Enumerationen einsetzen lassen.

6.5.1 Was sind Enumerationen?

Eine Enumeration ist eine Gruppe benannter Konstanten, die als ganzzahlige Werte (vom Typ *Byte*, *Short*, *Integer*, *Long*) zu vereinbaren sind. Diese Gruppe benannter Konstanten (Enumerationsliste) stellen die Member (Mitglieder) der Enumeration dar. Enumerationen bieten eine elegante Möglichkeit, Konstanten zu vereinbaren und diese anschließend über ihre Namen anzusprechen. Sie können die Member einer Enumeration auch als Parameter an Prozeduren oder Funktionen (bzw. Methoden) übergeben. Zudem kann in *For ... Each*-Schleifen auf die Enumerationskonstanten zugegriffen werden.

Achtung

Wichtig ist, dass Enumerationen nur auf Modul-, Klassen-, Struktur- oder Namespace-Ebene definierbar sind. Sie können die Enumeration also nicht auf der Ebene einer Prozedur anlegen. Der Begriff der Enumeration ist allerdings zweideutig, denn neben den hier benutzten benannten Konstanten gibt es Auflistungen auch bei Objekten (Enumerationen).

Die Anweisung zur Definition einer Enumeration besteht aus den Schlüsselwörtern *Enum* und *End Enum*. Dazwischen werden die Member der Enumeration als Konstanten definiert. Vor das Schlüsselwort *Enum* lassen sich, wie bei der Deklaration von Variablen, optional die Zugriffsmodifizierer wie *Public*, *Protected*, *Friend*, *Protected Friend*, *Private*, *Friend* und *Shadows* setzen. Diese Zugriffsmodifizierer definieren die Gültigkeit sowie die Zugriffe auf die Enumeration. Zusätzlich lässt sich der Enumeration ein Datentyp zuweisen – fehlt diese Angabe, verwendet Visual Basic den Typ *Integer*.

Schauen wir uns das Ganze an einem einfachen Beispiel an. Es soll eine Enumeration *Color* mit den Membern *Rot*, *Gruen* und *Blau* definiert werden. Im einfachsten Fall sieht dies dann so aus:

```
Enum Color
    Rot
    Gruen
    Blau
End Enum
```

Hier wurde kein Typ für die Enumeration angegeben, daher verwendet Visual Basic den Datentyp *System.Int32* (also *Integer*). Da auch keine Wertzuweisungen in der Enumera-

tion auftauchen, weist Visual Basic intern den Membern der Enumeration fortlaufend die Wert 0, 1, 2 etc. (also Member *Rot* = 0, *Grün* = 1 und *Blau* = 2) zu. Sie können die Member der Enumeration aber auch mit eigenen Werten und einem eigenen Typ versehen, indem Sie die Definition folgendermaßen aufbauen:

```
Enum Color As Integer
    Rot = &HFF0000
    Gruen = &H00FF00
    Blau = &H0000FF
End Enum
```

Mit den Schlüsselwörtern *As Integer* erhält die Enumeration explizit einen Basisdatentyp zugewiesen. Im aktuellen Beispiel wurde für die als Literale angegebenen Werte der einzelnen Enumerationsmember die Hexadezimalschreibweise verwendet. Die Zeichenfolge *&H* vor dem Literal vereinbart in Visual Basic eine Hexadezimalzahl. Blau erhält dann den Dezimalwert 255 bzw. Hexadezimal *&H00FF* zugewiesen. Bei Farbangaben hat die Hexadezimalschreibweise den Vorteil, dass die RGB-Farbanteile sehr leicht festlegbar sind (je zwei Hexadezimalziffern für die Farben Rot, Grün und Blau). Nach der obigen Definition können Sie über den Namen der Enumeration auf die Member zugreifen.

```
Dim Farbel As Integer
Dim Farbe2 As Integer
Dim Farbe3 As Integer
Farbel = Color.Rot
Farbe2 = Color.Grün
Farbe3 = Color.Blau
```

Hier wurden drei *Integer*-Variable definiert, denen dann die Werte der Enumeration zugewiesen werden. Sie kennen jetzt verschiedene Möglichkeiten zur Definition von Enumerationen. Bei der Definition einer Enumeration im ersten Beispiel erhielten die Variablen implizit die Werte 0, 1 und 2 zugewiesen, während die zweite Definition der Enumeration die hexadezimalen Werte *&HFF0000*, *&H00FF00* und *&H0000FF* liefert. Das folgende Beispiel demonstriert die Verwendung einer Enumeration an zwei Konstanten *Color* und *Namen*, die außerhalb der Klasse, quasi auf Namespace- bzw. Dateiebene (*Color*), sowie innerhalb der Klasse (*Namen*) definiert wurden. Das Programm soll die diesen Membern zugewiesenen Werte auf der Konsoleebene ausgeben (Abbildung 6.1).

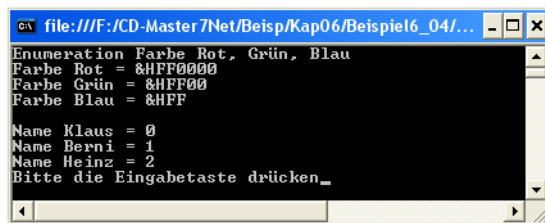


Abbildung 6.1: Anzeige der Namen und Werte der Enumerationen

Möchten Sie Variable vom Typ der Enumeration definieren und dann mit Werten belegen, lässt sich dies mit folgenden Anweisungen erledigen:

```
Dim Farbe As Color
Dim i As Global.Beispiel6.Color
Farbe = Color.Blau
i = Color.Rot
```

Die Variable *Farbe* wird einfach vom Typ *Color* vereinbart, kann also nur Werte der Auflistung aufnehmen.

Hinweis

Die Definition der Variable *i* enthält noch eine Neuerung aus Visual Basic 2005. Die Enumeration *Color* wurde auf Namespace-Ebene definiert und liegt somit im Namensraum des Projekts (hier *Beispiel6*). Mit dem Schlüsselwort *Global* ermöglicht Visual Basic 2005 den Zugriff auf den Stamm-Namespace an erster Stelle der Namensraum-Hierarchie. Die Anweisung *Global.Beispiel6.Color* stellt also sicher, dass sich die Referenzen wirklich auf den betreffenden Namensraum beziehen. Dies ist hilfreich, falls es zu Konflikten mit gleichnamigen Elementen in anderen Namenräumen kommen könnte.

Die Werte der Enumeration *Color* lassen sich mit folgender Anweisung ausgeben:

```
WriteLine("Enumeration Farbe {0}, {1}, {2}", _
        Color.Rot, Color.Grün, Color.Blau)
```

Mit der folgenden Anweisung wird der Name des Members sowie dessen Wert als Hexadezimalzahl angezeigt:

```
WriteLine("Farbe {0} = &H{1}", Color.Rot, Hex(Color.Rot))
```

Die Angabe *Color.Rot* bewirkt innerhalb der *WriteLine()*-Anweisung, dass Visual Basic den Member im Platzhalter {0} an die Methode übergibt. Die Ausgabe der *WriteLine()*-Anweisung ist in Abbildung 6.1 als erste Ergebniszeile zu sehen. Offenbar zeigt die Methode den Namen des Members an. Um den Wert des betreffenden Members als Hexadezimalzahl zu erhalten, wird die Visual-Basic-Funktion *Hex()* benutzt. Die Funktion *CInt* übernimmt ggf. die Wandlung des Werts in eine Integerzahl. Der Namensraum *Microsoft.VisualBasic* ist im Gegensatz zu früheren Visual-Basic-Funktionen bereits im Projekt eingebunden, muss also nicht mehr als *Imports*-Anweisung im Programmkopf aufgeführt werden. Weitere Details zum Beispiel können Sie nachfolgendem Listing entnehmen.

```
'*****
' File/Projekt: Beispiel06_04
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung der Enum-Anweisung
' zur Definition benannter Konstante.
'*****
Option Strict On
Imports System.Console      ' für WriteLine

Public Enum Color As Integer ' Enumeration für Farben
    Rot = &HFF0000
    Grün = &HFF00
    Blau = &HFF
End Enum

Class Test
    Public Enum Namen      ' Enumeration für Namen
        Klaus
        Berni
        Heinz
    End Enum

    Shared Sub Main()
        Dim Farbe As Color
        Dim i As Global.Beiispiel6.Color

        ' jetzt die Konstanten an Variable zuweisen und ausgeben
        Farbe = Color.Blau
        i = Color.Rot

        ' hier listen wir nur die Namen auf
        WriteLine("Enumeration Farbe {0}, {1}, {2}", _
            Color.Rot, Color.Grün, Color.Blau)
        ' jetzt die Werte der Konstanten als Dezimal & Hexadezimal
        WriteLine("Farbe {0} = &H{1}", Color.Rot, Hex(i))
        WriteLine("Farbe {0} = &H{1}", Color.Grün, _
            Hex(CInt(Color.Grün)))
        WriteLine("Farbe {0} = &H{1}", Color.Blau, _
            Hex(CInt(Color.Blau)))
    End Sub
End Class
```

Listing 6.1: Datei Beispiel6_02.vb mit einer Enumeration


```

WriteLine()

' Jetzt die Name-Enumeration anzeigen
WriteLine("Name {0} = {1}", Namen.Klaus, CInt(Namen.Klaus))
WriteLine("Name {0} = {1}", Namen.Berni, CInt(Namen.Berni))
WriteLine("Name {0} = {1}", Namen.Heinz, CInt(Namen.Heinz))

Write("Bitte die Eingabetaste drücken")
ReadLine()
End Sub
End Class

```

Listing 6.1: Datei Beispiel6_02.vb mit einer Enumeration (Forts.)

Hinweis

Sie finden das als Konsoleanwendung realisierte Beispiel im Projektordner `\Beisp\Kap06\Beispiel6_04` der Begleit-CD.

6.5.2 Zugriff auf Enumerationen mit Schleifen

Da Enumerationen aus Membern bestehen, bieten sich ggf. Schleifen zur Bearbeitung aller Elemente an. Wenn Sie sich noch einmal in Erinnerung rufen, dass eine Enumeration nichts anders ist als eine Sammlung benannter ganzzahliger Konstante, lassen sich die Start- und Endwerte auch innerhalb einer einfachen *For...Next*-Schleife einsetzen. Nehmen wir das bereits benutzte Beispiel, welches aus dem aktuellen Datum den Namen des Wochentags bestimmt. Wie wäre es, die Wochentage als Enumeration in Visual Basic zu vereinbaren?

```

Public Enum WeekDay As Byte
    Sonntag = 1
    Montag
    Dienstag
    Mittwoch
    Donnerstag
    Freitag
    Samstag
End Enum

```

Als Datentyp wurde *Byte* verwendet. Ich habe die Konstante *Sonntag* auf den Wert 1 gesetzt, damit die Zählung der Wochentage bei 1 und nicht bei 0 beginnt (damit die Enumeration mit der Visual-Basic-Implementierung von *FirstDayOfWeek.System* kompatibel ist). Müssen Sie in einem Programm die einzelnen Wochentage in einer Schleife bearbeiten, ließe sich dies folgendermaßen realisieren:

```
For i = 1 To 7 ' Alle Wochentage
...
Next i
```

Dies setzt aber voraus, dass Sie wissen, dass Sonntag mit dem Wert 1 und Samstag mit dem Wert 7 belegt ist. Um diese Details zu verstecken, lassen sich die betreffenden Konstanten in einer Enumeration ablegen. Sie können die Schleife daher auch folgendermaßen definieren:

```
For i = WeekDay.Sonntag To WeekDay.Samstag
...
Next i
```

Da der Member *Sonntag* mit dem Wert 1 und *Samstag* mit dem Wert 7 belegt ist, erfolgen sieben Schleifendurchläufe – der Schleifenindex *i* nimmt die Werte zwischen 1 und 7 an. Benötigen Sie den Namen des Members (der hier zufällig dem Wochentagsnamen entspricht), lässt sich die *GetName()*-Methode der *Enum*-Klasse aus dem Namensraum *System.Enum* verwenden. Die Methode *GetName()* erwartet zwei Parameter und liefert dann den Namen der angegebenen Enumerationskonstanten zurück. Im ersten Parameter ist die Enumeration zu übergeben, während der zweite Parameter die gewünschte Konstante übernimmt. Die Enumeration lässt sich mittels der *GetType()*-Methode holen. Die folgende Anweisung gibt den Namen für den Wochentag mit dem Indexwert 2 zurück:

```
Wtag = GetName(GetType(Weekday), 2)
```

Aus dem obigen Beispiel lässt sich erkennen, dass Enumerationen eine elegante Möglichkeit darstellen, mit benannten Konstanten zu arbeiten. Bei Verwendung der Enumerationskonstanten in einer *For...Next*-Schleife gibt es aber noch ein Problem, das Sie kennen sollten. Schauen wir uns die folgende Enumeration an:

```
Public Enum Color As Integer
    Rot = 1
    Grün = 2
    Blau = 4
End Enum
```

Hier werden drei Konstante vereinbart, die die Werte 1, 2 und 4 aufweisen. Möchten Sie die Konstanten der Enumeration innerhalb einer Schleife auswerten, dürfen Sie keinen *For...Next*-Anweisungsblock verwenden. Der folgende Ansatz

```
Dim i As Integer
For i = Color.Rot To Color.Blau
    Writeline ("Farbe {0} = {1}", GetName(GetType(Color), i), i)
Next i
```

bewirkt insgesamt vier Schleifendurchläufe, obwohl die Auflistung nur drei Member besitzt. Für den Indexwert 3 gibt es keinen Member, d.h., die *GetName*-Methode liefert eine leere Zeichenkette zurück (siehe erster Durchlauf in Abbildung 6.2). Daher wurde die Schleife um eine Überprüfung, ob der Member für einen Wert auch definiert ist, erweitert.

```
For i = Color.Rot To Color.Blau
    If Color.IsDefined(GetType(Color), i) Then ' Wert definiert
        WriteLine("Farbe {0} = {1}", GetName(GetType(Color), i), i)
    Else
        WriteLine("Undefinierte Farbe für Wert {0} ", i)
    End If
Next i
```

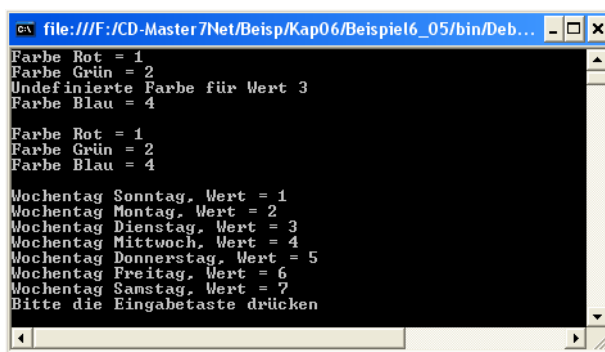


Abbildung 6.2: Anzeige der Namen und Werte der Enumerationen

Hinweis

Bei Enumerationen, Auflistungen (Collection) oder Feldern (Array) wissen Sie nicht immer, wie viele Elemente diese beinhalten. Auflistungen sind beispielsweise Objekte, die eine Sammlung gleichwertiger Objekte (z.B. alle Bedienelemente in einem Formular oder alle Wörter in einem Text etc.) aufweisen. Um sicherzustellen, dass jedes Element einer Enumeration, Auflistung oder eines Feldes in einer Schleife adressiert wird, lässt sich die *For Each*-Schleife verwenden. Diese besitzt die Struktur *For Each index In Objekt ... Next index*. Dabei ist *Objekt* der Platzhalter für eine Enumeration, eine Auflistung oder ein Feld. Die Schleife wertet das betreffende Objekt aus und überträgt den ersten Member in den Schleifenindex. Bei jedem Durchlauf wird der jeweils nächste Member des Objekts im Schleifenindex hinterlegt. Die Schleife terminiert, sobald alle Member des Objekts bearbeitet wurden.

Um sicherzustellen, dass nur die Member der Enumeration bearbeitet werden, können Sie auf die im nächsten Abschnitt detaillierter vorgestellte *For Each*-Schleife ausweichen. Hier der entsprechende Code zum Auswerten aller Member einer Enumeration:

```
Dim i As Integer
For Each i In GetValues(GetType(Color))
    Writeline ("Farbe {0} = {1}", GetName(GetType(Color),i), i)
Next i
```

Die in der Enumeration verfügbaren Werte werden über die *GetValues()*-Methode bereitgestellt. Dieser Methode wird als Parameter die über *GetType()* ermittelte Enumeration übergeben. Das folgende Listing demonstriert den Sachverhalt nochmals, indem zwei Enumerationen definiert und dann in Schleifen bearbeitet werden. Dabei gibt das Programm die Schleifenindizes sowie die Namen der Enumerations-Member aus (Abbildung 6.2). Im ersten Ergebnisblock sehen Sie, dass die Schleife vier Mal durchlaufen wurde und der Indexwert 3 keinem Member entspricht. Der nächste Ergebnisblock umfasst nur die Member der Enumeration *Color*. Weitere Details sind dem nachfolgenden Listing zu entnehmen.

```
'*****
' File/Projekt: Beispiel06_05
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung der Enum-Anweisung
' in Verbindung mit Schleifen.
'*****

Option Strict On
Imports System.Console      ' für WriteLine
Imports System.Enum         ' für Enumeration

Public Enum Color As Integer ' vereinbare die Enumeration
    Rot = 1
    Grün = 2
    Blau = 4
End Enum

Public Enum WeekDay As Byte  ' vereinbare die Enumeration
    Sonntag = 1
    Montag
    Dienstag
    Mittwoch
    Donnerstag
    Freitag
    Samstag
End Enum

Class Test
```

Listing 6.2: Zugriff auf Enumerationen mit Schleifen

```

Shared Sub Main()
    Dim i As Integer          ' Schleifenindex

    ' Schleife über die beiden Grenzen der Enumeration Farben
    For i = Color.Rot To Color.Blau
        If Color.IsDefined(GetType(Color), i) Then ' Wert definiert
            WriteLine("Farbe {0} = {1}", GetName(GetType(Color), i), i)
        Else
            WriteLine("Undefinierte Farbe für Wert {0} ", i)
        End If
    Next i

    WriteLine()
    ' Schleife über die Member der Enumeration
    For Each i In GetValues(GetType(Color))
        WriteLine("Farbe {0} = {1}", GetName(GetType(Color), i), i)
    Next i

    WriteLine()
    ' Schleife über Start-/Endwert der Enumeration Tag
    For i = WeekDay.Sonntag To WeekDay.Samstag
        WriteLine("Wochentag {0}, Wert = {1}", _
            GetName(GetType(WeekDay), i), i)
    Next i

    Write("Bitte die Eingabetaste drücken")
    ReadLine()
End Sub
End Class

```

Listing 6.2: Zugriff auf Enumerationen mit Schleifen (Forts.)

Hinweis

Sie finden die Projektdateien mit dem Beispiel im Ordner `\Beisp\Kap06\Beispiel6_05`. Im Ordner `\Beisp\Kap06\Beispiel6_05a` finden Sie ein Beispiel, welches den Zugriff auf eindimensionale Felder mittels Schleifen demonstriert.

6.6 Schleifen und Prozeduren vorzeitig beenden

Bei Schleifen und bei den nachfolgend vorgestellten Prozeduren gibt es Fälle, in denen die Bearbeitung unterbrochen werden muss, es ist eine Verzweigung auf die Anweisung hinter der Schleife oder hinter dem Prozeduraufruf erforderlich. Visual Basic 2005 bietet die *Exit*-Anweisung für diesen Zweck. Für eine *Do*-Schleife ist die Anweisung *Exit Do* innerhalb der Schleife vorgesehen. Bei *For*-Schleifen verwenden Sie die Anweisung *Exit For* und bei Prozeduren gilt die Anweisung *Exit Sub*. Die folgenden Anweisungen zeigen die Verwendung dieser Anweisung:

```
Do
  If wert < 0 Then
    Exit Do
  Else
    summe = summe + wert
  End If
  i = i + 2
Loop While i <= 10
```

Die obige *Do*-Schleife wird beendet, sobald die Variable *i* den Wert 12 annimmt. Die Schleife wird aber sofort über *Exit Do* beendet, falls die Variable *wert* einen negativen Wert aufweist.

6.7 Prozeduren und Funktionen

Die bisherigen Beispiele enthalten lediglich linearen Programmcode, bei denen ggf. Schleifen und bedingte Anweisungen bzw. Verzweigungen eingefügt werden können. Möchte man Codeteile häufiger verwenden, bietet sich die Verlagerung in Funktionen oder Prozeduren an. Dann kann die Funktion oder Prozedur aus dem Hauptprogramm beliebig aufgerufen und der Code ausgeführt werden. Der Unterschied zwischen Prozedur und Funktion besteht darin, dass eine Funktion einen Wert an das rufende Programm als Funktionsergebnis zurückliefert. Daher lassen sich Funktionen auf der rechten Seite einer Zuweisung einsetzen, Prozeduren sind dagegen in einer getrennten Zeile aufzurufen.

6.7.1 Arbeiten mit Funktionen

Verwenden Sie Funktionen, falls ein Wert als Ergebnis an das rufende Programm zurückzugeben ist. Eine Funktion wird mit den Anweisungen

```
Function name (Parameter) As Typ
...
End Function
```

definiert. Das Schlüsselwort *Function* leitet die Definition ein, die andererseits mit *End Function* abgeschlossen werden muss. Die Anweisungen innerhalb dieses Blocks werden beim Aufruf der Funktion abgearbeitet. Nach dem einleitenden Schlüsselwort *Function* folgt der Name der Funktion. Dieser Name ist nach den Regeln für Bezeichner aufzubauen und dient später im rufenden Programm zur Identifikation der gewünschten Funktion (z.B. *Multiply*). Die Parameter des Funktionsaufrufs werden in Klammern hinter dem Funktionsnamen aufgeführt. Der (optionale) Typ in der Deklaration spezifiziert den Datentyp für den Rückgabewert der Funktion. Fehlt eine Typangabe, so wird standardmäßig *Object* für den zurückgelieferten Funktionswert angenommen.

Der Rückgabewert kann in Visual Basic 2005 entweder dem Funktionsnamen innerhalb der Funktion zugewiesen werden. Oder Sie verwenden das *Return*-Schlüsselwort.

Hinweis

Ähnlich wie bei Variablen kann vor dem Schlüsselwort *Function* noch ein Schlüsselwort wie *Public*, *Private* etc. auftreten, mit dem die Zugriffe auf die Funktion oder deren Eigenschaften geregelt werden.

Schauen wir uns in einem einfachen Beispiel den Aufbau einer Funktion sowie deren Verwendung in einem Programm an. Im kaufmännischen Bereich ist häufiger der Bruttobetrag aus dem Nettobetrag und der Mehrwertsteuer zu ermitteln. Diese Rechenvorschrift ließe sich als Funktion mit dem Namen *Brutto* realisieren. Dann braucht im Programm nur noch die Funktion *Brutto* mit dem Nettowert und der Mehrwertsteuer als Parameter aufgerufen zu werden. Die Funktion gibt den berechneten Bruttowert zurück. Bei häufigen Mehrwertsteuerberechnungen eine komfortable Sache. Der Quellcode zur Definition der Funktion könnte zum Beispiel so aussehen:

```
Shared Function Brutto(Netto As Decimal, MwSt As Decimal) _
    As Decimal
    Return Netto * (1.0 + MwSt/100.0)
End Function
```

Als Funktionsname wird hier *Brutto* verwendet. Das Schlüsselwort *Shared* ist erforderlich, damit die Funktion in den Prozeduren des Moduls oder der Klasse ohne Instantiierung abrufbar wird. Der Parameter *Netto* ist vom Datentyp *Decimal* und enthält den Nettowert. Der Mehrwertsteuersatz in Prozent wird ebenfalls als Datentyp *Decimal* übergeben. Die Funktion gibt selbst ebenfalls einen Wert vom Typ *Decimal* zurück. Innerhalb der Funktion findet sich nur eine Anweisung, die den Bruttobetrag ermittelt und das Ergebnis über das Schlüsselwort *Return* zurückgibt.

Achtung

Der Code einer Funktion (oder Prozedur) muss auf Modul- oder Klassenebene (oder in Strukturen) definiert werden.

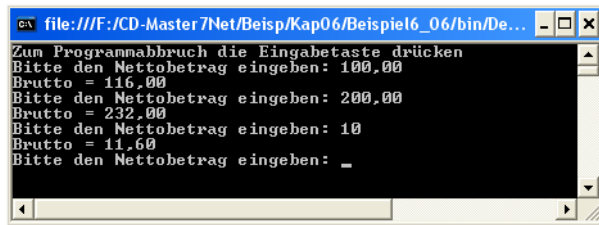


Abbildung 6.3: Ein-/Ausgaben des Programms zur Berechnung der Bruttowerte

Kommen wir jetzt zu dem Programmteil, welches die neue Funktion nutzt. Ich habe mich für ein einfaches Beispiel entschieden, welches vom Benutzer eine Zahl einliest, diese auf Plausibilität überprüft, den Bruttobetrag errechnet und das Ergebnis wieder anzeigt. Das Programm liest solange Benutzereingaben ein, bis dieser die -Taste drückt. Die Ein-/Ausgaben des Programms sind in Abbildung 6.3 zu sehen.

Bevor Sie einen Blick in das gesamte Listing werfen, möchte ich noch einige Erläuterungen zur Realisierung geben. Um mehrere Benutzereingaben zu verarbeiten, habe ich eine Endlosschleife realisiert.

```
Do                ' Wiederholen bis Leereingabe erkannt
  Write ("Bitte den Nettobetrag eingeben: ")
  tmp = ReadLine()
  If tmp = "" Then Exit Do  ' Eingabeschleife beenden
  ...
Loop Until False
```

Die Schleife endet nie, da am Schleifenende die *Until*-Bedingung fest auf den Wert *False* gesetzt ist. Über einen Aufruf der *Write()*-Methode wird der Benutzer zur Eingabe eines Werts aufgefordert. Die *ReadLine()*-Methode liest dann die Benutzereingabe ein, sobald diese mit abgeschlossen wird. Diese Eingabe wird der Hilfsvariablen *tmp* zugewiesen, da die Methode nur Zeichenketten liefert. Zudem lässt sich über die Stringvariable *tmp* prüfen, ob der Benutzer die -Taste ohne weitere Eingabe gedrückt hat. In diesem Fall wird ein leerer String "" zurückgegeben. Die *If*-Anweisung nutzt dies aus und prüft, ob die Variable *tmp* eine leere Zeichenkette enthält. Trifft dies zu, bewirkt die *Exit Do*-Anweisung ein Verlassen der Schleife (dies ist das einzige Abbruchkriterium).

Die eingelesenen Zeichen müssen anschließend in einen Wert vom Typ *Decimal* konvertiert werden. Dies kann über die Visual Basic-Funktion *CDec()* erfolgen. Allerdings sollte der Fall abgefangen werden, dass der Benutzer keine Zahl, sondern eine beliebige Zeichenkette (z.B. 123Hallo) eintippt. Die Visual Basic-Kompatibilitätsklasse bietet die Funktion *IsNumeric()*, die prüft, ob die eingegebenen Zeichen eine gültige Zahl ergeben. Die folgenden Zeilen verifizieren daher die Eingabe und wandeln die Eingabezeichen in eine Zahl um.

```
If IsNumeric(tmp) Then  ' eine Zahl?
  Netto = CDec(tmp)     ' String in Dezimalzahl
  ...
```



```
Else
...
End If
```

Der Aufruf der Funktion reduziert sich auf eine Zeile. Die Anweisung enthält auf der linken Seite die Variable *Preis* zur Aufnahme des Ergebnisses. Auf der rechten Seite steht der Funktionsname *Brutto*. Die zu übergebenden Parameter werden in Klammern hinter dem Namen angegeben.

```
Preis = Brutto (Netto, MwSt)
```

Nach Ausführung dieser Zeile liegt der Bruttowert in der Variablen *Preis* vor und kann ausgegeben werden. Hierzu verwende ich die bereits bestens bekannte *WriteLine()*-Methode.

```
WriteLine ("Brutto = {0:###.00}", Preis)
```

Um jedoch zu verhindern, dass die Methode bei der Ausgabe nicht signifikante Dezimalziffern abschneidet, habe ich diesmal eine Formatschablone im Platzhalter {0} verwendet. Formate lassen sich getrennt durch einen Doppelpunkt angeben und bestehen aus verschiedenen Zeichen. In der Formatschablone {0:###.00} steht der Punkt für den Dezimalpunkt im Ausgabewert. Eine 0 signalisiert eine Ziffer, die immer ausgegeben werden muss. Das Zeichen # steht für eine Ziffer, die optional ausgegeben werden kann. Ohne diese Formatschablone würde der Wert 116,00 als 116 angezeigt werden. Weitere Details können Sie dem folgenden Listing entnehmen:

```
'*****
' File/Projekt: Beispiel06_06
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung einer Funktion.
'*****
Option Strict On
Imports System.Console      ' für WriteLine

Class Test
' Berechnung der Mehrwertsteuer

    Shared Function Brutto(ByVal Netto As Decimal, _
        ByVal MwSt As Decimal) As Decimal
        Return CDec(Netto * (1.0 + MwSt / 100.0))
    End Function

    Shared Sub Main()        ' das ist das Hauptmodul
        Const MwSt As Decimal = 16 ' 16 Prozent Mehrwertsteuer
```

Listing 6.3: Beispiel zur Verwendung einer Funktion

```

Dim tmp As String          ' Hilfsvariable für Eingaben
Dim Netto As Decimal
Dim Preis As Decimal

WriteLine("Zum Programmabbruch die Eingabetaste drücken")
Do                          ' Wiederholen bis Leereingabe erkannt
    Write("Bitte den Nettobetrag eingeben: ")
    tmp = ReadLine()
    If tmp = "" Then Exit Do ' Eingabeschleife beenden
    If IsNumeric(tmp) Then  ' eine Zahl?
        Netto = CDec(tmp)   ' String in Dezimalzahl
        Preis = Brutto(Netto, MwSt) ' Berechne Bruttopreis
        WriteLine("Brutto = {0:###0.00}", Preis) ' Preis ausgeben
    Else
        WriteLine(" *** Fehleingabe ***") ' Fehlermeldung
    End If
Loop Until False           ' Endlosschleife
End Sub
End Class

```

Listing 6.3: Beispiel zur Verwendung einer Funktion (Forts.)

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap06\Beispiel6_06` auf der Begleit-CD. Führen Sie das Beispiel im Fenster der Eingabeaufforderung aus und geben Sie dann Dezimalzahlen für die Nettowerte ein. Die -Taste beendet den Programmablauf.

6.7.2 Prozeduren

Neben Funktionen werden Prozeduren häufig in Visual Basic-Anwendung benutzt. Prozeduren werden in der Form

```

Sub Name (Parameter)
...
End Sub

```

innerhalb eines Moduls oder einer Klasse definiert. Sie können also keine Prozedur innerhalb einer Prozedur oder außerhalb eines Moduls/einer Klasse auf Namespace-Ebene vereinbaren! Die Definition der Prozedur wird durch das Schlüsselwort *Sub* eingeleitet und mit *End Sub* beendet – der Code der Prozedur wird also durch die Schlüsselwörter eingeschlossen. *Name* steht für den Prozedurnamen, der sich nach den Regeln für

Bezeichner frei wählen lässt. Die Parameter der Prozedur werden in Klammern hinter dem Prozedurnamen übergeben. Da eine Prozedur keinen Rückgabewert zurück liefert, wird auch kein Prozedurtyp vereinbart.

Um innerhalb der Prozedur auf Wert zuzugreifen, könnten Sie die Variablen auf Modul- oder Klassenebene als *Public* deklarieren. *Public*-Variable sind automatisch auch innerhalb der Prozedur gültig und lassen sich dort lesen oder sogar ändern. Da dies jedoch eine tückische Fehlerquelle ist, ist dieser Ansatz verpönt. Es gibt eine bessere Alternative: Sie übergeben die Variablen als Parameter (Argumente) an die Prozedur. Hierbei lässt sich sogar noch festlegen, ob innerhalb der Prozedur durchgeführte Änderungen an den Werten im rufenden Programm wirksam werden oder nicht. Aber darauf kommen wir später zurück.

Ähnlich wie bei Variablen können Schlüsselwörter wie *Public*, *Shared*, *Friend* etc. im Kopf der Prozedurdefinition angegeben werden, um die Zugriffsmöglichkeiten auf die Prozedur festzulegen.

Es wurde bereits auf der vorhergehenden Seite erwähnt: Eine Prozedur lässt sich auf Modul- oder Klassenebene vereinbaren. Bei Klassen muss vor dem Schlüsselwort *Sub* aber das Schlüsselwort *Shared* angegeben werden (gibt die Prozedur als Member der Klasse frei). Gleichzeitig wird der Prozedur der Zugriffsmodus *Public* zugewiesen.

Hinweis

Sie haben bereits ausgiebig mit Prozeduren gearbeitet, ohne dies vielleicht zu wissen. Die Hauptprogramme aller bisherigen Beispiele stellen eine Prozedur dar, die mit *Shared Sub Main()* deklariert und mit *End Sub* abgeschlossen wurde. *Main* ist dabei der von mir gewählte Prozedurname und die Klammer () signalisiert, dass diese Prozedur keine Parameter verwendet.

An dieser Stelle möchte ich den Einsatz einer zweiten Prozedur demonstrieren. Das vorherige Beispiel wird so abgewandelt, dass die Berechnung der Bruttowerte und die Ausgabe des Ergebnisses in eine Prozedur verlagert werden. Zudem kommt die Funktion *Brutto* weiter zum Einsatz. Der Code der Prozedur *Anzeige* enthält folgende Anweisungen:

```
Shared Sub Anzeige (txt As String, MwSt As Decimal)
    Dim Netto, Preis As Decimal

    If IsNumeric(txt) Then          ' eine Zahl?
        Netto = CDec(txt)          ' String in Dezimalzahl
        Preis = Brutto (Netto, MwSt) ' Berechne Bruttopreis
        WriteLine ("Brutto = {0:###0.00}", Preis) ' Preis ausgeben
    Else
        WriteLine (" *** Fehleingabe ***") ' Fehlermeldung
    End If
End Sub
```

Als Parameter erwartet die Prozedur einen Textwert mit der Eingabe sowie die Mehrwertsteuer als *Decimal*-Typ. Ansonsten habe ich gegenüber dem vorhergehenden Beispiel die Anweisungen zur Berechnung und Ausgabe in die Prozedur *Anzeige* gezogen. Die als Parameter übergebenen Werte gelten innerhalb der Prozedur als lokale Variable. Daher muss der Code so angepasst werden, dass anstelle der Variablen *tmp* das übergebene Argument *txt* eingesetzt wird. Auch die Variablen *Netto* und *Preis* habe ich als lokale Variable innerhalb der Prozedur definiert. Im Hauptprogramm lässt sich die Prozedur mit folgender Anweisung aufrufen:

```
Anzeige (tmp, MwSt)
```

Sie müssen lediglich den Prozedurnamen, gefolgt von der in Klammern stehenden Parameterliste angeben.

Achtung

Bei früheren Basic-Varianten war es bei Prozeduraufrufen möglich, die Parameter mit oder ohne Klammern anzugeben. Trat nur der Prozedurname auf, mussten die Parameter ohne Klammern angegeben werden. Erfolgt der Aufruf mit *Call Prozedurname* (), waren die Parameter in Klammern zu setzen. In Visual Basic 2005 können Prozeduren mit oder ohne *Call* aufgerufen werden, die Parameter sind aber immer in Klammern zu setzen.

Weitere Details sind dem folgenden Listing zu entnehmen:

```
'*****
' File/Projekt:  Beispiel06_07
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung einer Prozedur.
'*****
Option Strict On
Imports System.Console          ' für WriteLine

Class Test
    ' Berechnung der Mehrwertsteuer
    Shared Function Brutto(ByVal Netto As Decimal, _
                          ByVal MwSt As Decimal) _
        As Decimal
        Return CDec(Netto * (1.0 + MwSt / 100.0))
    End Function

    ' Sorge für die Ausgabe
    Shared Sub Anzeige(ByVal txt As String, ByVal MwSt As Decimal)
```

Listing 6.4: Beispiel mit Prozeduraufrufen

```

Dim Netto, Preis As Decimal

If IsNumeric(txt) Then      ' eine Zahl?
    Netto = CDec(txt)       ' String in Dezimalzahl
    Preis = Brutto(Netto, MwSt) ' Berechne Bruttopreis
    WriteLine("Brutto = {0:###0.00}", Preis) ' Preis ausgeben
Else
    WriteLine(" *** Fehleingabe ***") ' Fehlermeldung
End If
End Sub

Shared Sub Main()          ' das ist das Hauptmodul
    Const MwSt As Decimal = 16 ' 16 Prozent Mehrwertsteuer

    Dim tmp As String      ' Hilfsvariable für Eingaben

    WriteLine("Zum Programmabbruch die Eingabetaste drücken")
    Do                    ' Wiederholen bis Leereingabe erkannt
        Write("Bitte den Nettobetrag eingeben: ")
        tmp = ReadLine()
        If tmp = "" Then Exit Do ' Eingabeschleife beenden
        Anzeige(tmp, MwSt)      ' Prozeduraufruf
    Loop Until False           ' Endlosschleife
End Sub
End Class

```

Listing 6.4: Beispiel mit Prozeduraufrufen (Forts.)

Hinweis

Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap06\Beispiel6_07` auf der Begleit-CD.

6.7.3 Anmerkungen zur Parameterübergabe (ByRef, ByVal)

Beim Aufruf von Funktionen und Prozeduren können Sie Parameter, getrennt durch Kommas, in Klammern übergeben. Jeder übergebene Parameter stellt für die Prozedur eine Variable dar. Für die Programmierung ist es dabei aber relevant, wie sich Änderungen an den Werten der Parameter, die innerhalb der Prozedur oder Funktion erfolgen, auswirken.

- Eine Variante besteht darin, dass in der Prozedur oder Funktion vorgenommene Änderungen an den Werten der übergebenen Parameter nur lokal in der Prozedur oder Funktion wirksam werden – der rufende Programmblock erfährt nichts von diesen Änderungen. Dies hat bei der Programmierung den Vorteil, dass im rufenden

Block sichergestellt ist, dass übergebene Werte nicht unbeabsichtigt durch die Prozedur geändert werden können (bei einer Funktion ist ja eine explizite Zuweisung des Funktionsergebnisses erforderlich).

- Die andere Variante besteht darin, dass sich jede durch die Prozedur oder Funktion vorgenommene Änderung an den als Parameter übergebenen Werten auch im rufenden Block auswirkt. Diese Variante erlaubt dem rufenden Block einen Zugriff auf die in der Prozedur (oder Funktion) berechneten Ergebnisse.

Der Fachausdruck für dieses Verhalten ist *Call by Reference* bzw. *Call by Value*. Beim *Call-by-Value*-Verfahren wird einfach der Wert einer Variablen als Kopie mit der Parameterliste übergeben. Ändert die Prozedur bzw. Funktion den Wert des Parameters, wirkt sich dies nur auf die lokale Kopie aus, die Originalvariable im rufenden Block bleibt unverändert. Endet die Prozedur, geht die Kopie (und damit auch die Änderung) verloren. Beim *Call-by-Reference*-Aufruf wird als Argument dagegen kein Wert, sondern ein Verweis auf die betreffende Variable als Argument übergeben. Die Prozedur oder Funktion greift über diese Referenz also direkt auf die Variable des rufenden Blocks zu, jede Änderung innerhalb der gerufenen Prozedur oder Funktion wirkt sich daher sofort auf den Wert im rufenden Block aus.

Standardmäßig verwendet Visual Basic 2005 die Einstellung *Call by Value*, wodurch nur lokal gültige Kopien von Parametern in der Prozedur bzw. Funktion verfügbar sind. Dies weicht übrigens von früheren Visual-Basic-Versionen ab, bei denen *Call by Reference* als Standardübergabemechanismus vereinbart war.

Sie können bei der Parameterdefinition im Deklarationsteil der Funktion oder Prozedur vereinbaren, welcher Übergabemechanismus benutzt werden soll. Hierzu ist jeweils das Schlüsselwort *ByRef* oder *ByVal* vor den jeweiligen Parameter zu stellen (Objekte sind als *ByRef*-Parameter zu übergeben). Fehlt das Schlüsselwort, wird *ByVal* angenommen.

Das vorherige Beispiel soll jetzt so umgestellt werden, dass die Prozedur *Ausgabe* sowohl den Bruttowert als auch den Nettowert und den Mehrwertsteuersatz an das rufende Modul zurückgibt. Im ersten Schritt wird die Konstante für die Mehrwertsteuer global auf Klassenebene vereinbart. Damit kann die Prozedur *Ausgabe* auf diese Konstante zugreifen, ohne dass diese als Parameter übergeben wird. Zusätzlich werden die Variablen *Preis*, *Netto* und *USt* als lokale Variable in der Prozedur *Main()* deklariert. Die drei Variablen werden beim Aufruf der Prozedur *Ausgabe* als Parameter übergeben. Damit sich Änderungen in *Main* auswirken, muss dies über *ByRef* erfolgen. Die geänderte Zeile zur Prozedurdeklaration sieht dann folgendermaßen aus:

```
Shared Sub Anzeige (txt As String, ByRef Preis As Decimal, _  
                    ByRef Netto As Decimal, ByRef USt As Decimal)
```

Bis auf den Parameter *txt* wird alles als Referenz übergeben. Die Parameter können also in der Prozedur zum Speichern der Änderungen benutzt werden und wirken sich auf die Variablen der rufenden Prozedur *Main* aus. Zur Kontrolle habe ich die Hauptprozedur um eine Ausgabeanweisung ergänzt, die die berechneten Werte ausgibt. Weitere Details sind dem folgenden Listing zu entnehmen:

```

'*****
' File/Projekt: Beispiel06_08
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung einer Prozedur
' unter Verwendung von ByRef.
'*****

Option Strict On
Imports System.Console      ' für WriteLine

Class Test
    Const MwSt As Decimal = 16 ' 16 Prozent Mehrwertsteuer

    ' Berechnung der Mehrwertsteuer
    Shared Function Brutto(Netto As Decimal, MwSt As Decimal) _
        As Decimal
        Return CDec(Netto * (1.0 + MwSt/100.0))
    End Function

    ' Sorge für die Berechnung und Ausgabe
    Shared Sub Anzeige (txt As String, ByRef Preis As Decimal, _
        ByRef Netto As Decimal, ByRef USt As Decimal)
        If IsNumeric(txt) Then ' eine Zahl?
            Netto = CDec(txt) ' String in Dezimalzahl
            Preis = Brutto (Netto, MwSt) ' Berechne Bruttopreis
            USt = Preis - Netto
            WriteLine ("Brutto = {0:###0.00} " & _
                "Netto = {1:###0.00} MwSt {2:###0.00} ", _
                Preis, Netto, USt) ' Preis ausgeben
        Else
            WriteLine (" *** Fehleingabe ***") ' Fehlermeldung
        End If
    End Sub

    Shared Sub Main() ' das ist das Hauptmodul
        Dim Preis, Netto, USt As Decimal ' Hilfsvariable
        Dim tmp As String = "" ' Hilfsvariable für Eingaben

        WriteLine ("Zum Programmabbruch die Eingabetaste drücken")
        Do ' Wiederholen bis Leereingabe erkannt
            Write ("Bitte den Nettobetrag eingeben: ")

```

Listing 6.5: Beispiel zum Prozeduraufruf mit ByRef-Parameterübergabe

```
tmp = ReadLine()
If tmp = "" Then Exit Do ' Eingabeschleife beenden
Anzeige (tmp, Preis, Netto, Ust) ' Prozeduraufruf
WriteLine("Kontrolle {0}, {1}, {2}", Preis, Netto, Ust)
Loop Until False ' Endlosschleife
End Sub
End Class
```

Listing 6.5: Beispiel zum Prozeduraufruf mit ByRef-Parameterübergabe (Forts.)

Hinweis

Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap06\Beispiel6_08` auf der Begleit-CD.

6.7.4 Verwendung benannter Argumente

Mit den obigen Informationen sind Sie in der Lage, Funktionen und Prozeduren zu erstellen und zu nutzen. Gelegentlich gibt es aber den Fall, dass Sie die Reihenfolge der an eine Prozedur oder Funktion übergebenen Parameter nicht gemäß der Funktions- oder Prozedurdefinition festlegen können oder möchten. Dann müssen Sie die übergebenen Argumente beim Aufruf benennen. Hierzu wird einfach der Name des betreffenden Parameters aus der Definition der Prozedur bzw. der Funktion beim Aufruf in der Form *name:=wert* als Argument übertragen. Ist die Prozedur *Ausgabe* folgendermaßen definiert:

```
Shared Sub Anzeige (txt As String, ByRef Preis As Decimal, _
    ByRef Netto As Decimal, ByRef USt As Decimal)
```

kann die Übergabe der Argumente beim Prozeduraufruf folgendermaßen erfolgen:

```
Anzeige (Preis:=Preis1, txt:=tmp, Ust:=Ust1, Netto:=Netto1)
```

Jedes Argument weist den Namen des betreffenden Parameters, gefolgt von einem Doppelpunkt mit Gleichheitszeichen und dem Wert auf. Visual Basic 2005 sorgt dann dafür, dass die betreffenden Parameter in der richtigen Reihenfolge an die Prozedur übergeben werden. Weitere Details können Sie folgendem Listing entnehmen:

```
'*****
' File/Projekt: Beispiel06_09
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung einer Prozedur
' unter Verwendung benannter Parameter.
'*****
```

Listing 6.6: Prozeduraufruf mit benannten Argumenten


```

Option Strict On
Imports System.Console      ' für WriteLine

Class Test
    Const MwSt As Decimal = 16 ' 16 Prozent Mehrwertsteuer

    ' Berechnung der Mehrwertsteuer
    Shared Function Brutto(ByVal Netto As Decimal, _
        ByVal MwSt As Decimal) _
        As Decimal
        Return CDec(Netto * (1.0 + MwSt / 100.0))
    End Function

    ' Sorge für die Berechnung und Ausgabe
    Shared Sub Anzeige(ByVal txt As String, ByRef Preis As Decimal, _
        ByRef Netto As Decimal, ByRef USt As Decimal)
        If IsNumeric(txt) Then ' eine Zahl?
            Netto = CDec(txt) ' String in Dezimalzahl
            Preis = Brutto(Netto, MwSt) ' Berechne Bruttopreis
            USt = Preis - Netto
            WriteLine("Brutto = {0:###0.00} Netto = {1:###0.00} " _
                & "MwSt {2:###0.00} ", _
                Preis, Netto, USt) ' Preis ausgeben
        Else
            WriteLine(" *** Fehleingabe ***") ' Fehlermeldung
        End If
    End Sub

    Shared Sub Main() ' das ist das Hauptmodul
        Dim Preis1, Netto1, USt1 As Decimal ' Hilfsvariable
        Dim tmp As String = "" ' Hilfsvariable für Eingaben

        WriteLine("Zum Programmabbruch die Eingabetaste drücken")
        Do ' Wiederholen bis Leereingabe erkannt
            Write("Bitte den Nettobetrag eingeben: ")
            tmp = ReadLine()
            If tmp = "" Then Exit Do ' Eingabeschleife beenden
        ' Prozeduraufruf
            Anzeige(Preis:=Preis1, txt:=tmp, _
                Ust:=USt1, Netto:=Netto1)
    End Sub

```

Listing 6.6: Prozeduraufruf mit benannten Argumenten (Forts.)

```

        WriteLine("Kontrolle {0}, {1}, {2}", Preis1, Netto1, USt1)
    Loop Until False          ' Endlosschleife
End Sub
End Class

```

Listing 6.6: Prozeduraufruf mit benannten Argumenten (Forts.)

Hinweis

Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap06\Beispiel6_09` auf der Begleit-CD.

6.7.5 Arbeiten mit optionalen Argumenten

Bei der Deklaration von Funktionen und Prozeduren legen Sie auch die Zahl der Argumente fest. Ist die Zahl der Argumente variabel, d.h. sind einige Argumente optional, müssen Sie dies bei der Deklaration berücksichtigen. Hierzu ist das Schlüsselwort *Optional* im Prozedur- oder Funktionskopf vor die jeweiligen Parameter zu setzen. Allerdings ist die aus früheren Visual-Basic-Versionen bekannte Funktion *IsMissing* entfallen. Stattdessen weisen Sie dem Parameter einfach innerhalb der Deklarationszeile einen Initialisierungswert zu. Die Funktion *Brutto* aus den vorherigen Beispielen könnte mit dem optionalen Parameter *MwSt* dann so deklariert werden:

```

Shared Function Brutto(Netto As Decimal, _
    Optional MwSt As Decimal = 16) _
    As Decimal
    Return CDec(Netto * (1.0 + MwSt/100.0))
End Function

```

Die Angabe `Optional MwSt As Decimal = 16` signalisiert dem Visual-Basic-Compiler, dass der Parameter optional ist. Gleichzeitig wird diesem Parameter ein Initialisierungswert (hier 16) zugewiesen. Fehlt der Parameter beim Aufruf, setzt der Compiler einfach den Initialisierungswert ein. Der Aufruf dieser modifizierten Funktion ist in nachfolgendem Listing zu sehen. Beim zweiten Aufruf habe ich den Mehrwertsteuersatz als Parameter weggelassen.

```

'*****
' File/Projekt: Beispiel6_10
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung einer Prozedur/Funktion
' unter Verwendung optionaler Parameter.
'*****
Option Strict On
Imports System.Console          ' für WriteLine

```

Listing 6.7: Funktionsaufruf mit optionalen Funktionsparametern

```

Class Test
    Const MwSt As Decimal = 16.0D ' 16 Prozent Mehrwertsteuer

    ' Berechnung der Mehrwertsteuer, MwSt optional
    Shared Function Brutto(Netto As Decimal, _
        Optional MwSt As Decimal = 16) _
        As Decimal
        Return CDec(Netto * (1.0 + MwSt/100.0))
    End Function

    Shared Sub Main()                ' das ist das Hauptmodul
        Dim Netto As Decimal = 100.0D ' Hilfsvariable
        Dim tmp As String = ""        ' Hilfsvariable für Eingaben

        ' Sorge für die Berechnung und Ausgabe mit MwSt
        WriteLine ("Brutto = {0:###0.00} Netto = {1:###0.00}", _
            Brutto (Netto, MwSt), Netto) ' Preis ausgeben

        ' Aufruf von Brutto ohne MwSt
        WriteLine ("Brutto = {0:###0.00} Netto = {1:###0.00}", _
            Brutto (Netto), Netto) ' Preis ausgeben

        ReadLine()
    End Sub
End Class

```

Listing 6.7: Funktionsaufruf mit optionalen Funktionsparametern (Forts.)

Hinweis

Sie finden die Projektdateien der Konsolenanwendung im Ordner `\Beisp\Kap06\Beispiel6_10` auf der Begleit-CD.

6.7.6 Arbeiten mit ParamArray

In Visual Basic 2005 können Sie wie in älteren Visual-Basic-Versionen oder in VBA auch eine variable Anzahl an Parametern an Prozeduren oder Funktionen übergeben. Dann müssen Sie aber das *ParamArray*-Schlüsselwort in der Deklaration verwenden. Die Anzahl der Parameter wird erst zur Laufzeit definiert. Solche Argumente werden immer als *ByVal* übergeben. Zudem darf ein *ParamArray*-Schlüsselwort immer nur im letzten Argument auftreten. Die Werte des Arrays müssen dabei den gleichen Typ wie das Argument haben. Die Definition einer Funktion mit einer variablen Parameteranzahl sähe dann so aus:

```
Function xSum (ParamArray zahl()) As Double
```

Das nachfolgende Listing zeigt, wie sich dies zur Berechnung einer Summe nutzen lässt:

```
'*****
' File/Projekt: Beispiel6_11
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung von ParamArray.
'*****
Option Strict On
Imports System.Console      ' für WriteLine

Class Test
' Die Summe ermitteln
Shared Function MySum(ParamArray x() As Double) As Double
    Dim zSum As Double = 0.0    ' Hilfsvariable
    Dim i As Integer
' Summieren über alle Parameter
    For i = 0 To UBound(x)      ' Alle Parameter
        zSum = zSum + x(i)      ' Addieren
    Next i
    MySum = zSum                ' Ergebnis zurückgeben
End Function

Shared Sub Main()              ' das ist das Hauptmodul
    Dim Result As Double

    Result = MySum (100, 200, 300)      ' Berechne Summe
    WriteLine ("Summe = {0}", Result)
    Result = MySum (100, 200, 300, 400, 500) ' Berechne Summe
    WriteLine ("Summe = {0}", Result)
    ReadLine()
End Sub
End Class
```

Listing 6.8: Funktionsaufrufe mit ParamArray-Argumenten

Hinweis

Sie finden die Projektdateien der Konsoleanwendung im Ordner `\Beisp\Kap06\Beispiel6_11` auf der Begleit-CD.

6.7.7 Private-, Public- und Static-Deklarationen

Bei der Deklaration von Prozeduren und Funktionen lassen sich die Schlüsselwörter *Private* und *Public* verwenden, um die Zugriffsmöglichkeit auf die Prozedur- und Funktionsnamen zu ändern.

- Das Schlüsselwort *Public* legt fest, dass der Prozedur-/Funktionsname auch außerhalb des Moduls bekannt ist. Sie können die Prozedur/Funktion aus anderen Modulen aufrufen.
- Das Schlüsselwort *Private* vereinbart einen Prozedur- oder Funktionsnamen als lokal für das Modul. Der Versuch, die Prozedur aus einem anderen Modul aufzurufen, führt zu einem Übersetzungsfehler.

Geben Sie keines der Schlüsselwörter an, wird der Name als *Public* interpretiert. Das Gleiche gilt für eine als *Shared* vereinbarte Prozedur auf Klassenebene.

Static-Deklaration für lokale Variable

Deklarieren Sie Variable innerhalb einer Prozedur (bzw. Funktion), sind diese immer lokal. Sie können daher auf Prozedurebene die Zugriffsmodifizierer *Public* oder *Private* nicht in einer Variablendeklaration einsetzen. Allerdings ist in einer Variablendeklaration innerhalb einer Prozedur das Schlüsselwort *Static* zulässig. Das Schlüsselwort *Static* legt fest, dass die deklarierte lokale Variable nach Beendigung der Prozedur (in der sie deklariert wurde), mit ihrem letzten Wert erhalten bleibt.

Weitere Informationen zum Umgang mit Prozeduren und Funktionen finden Sie in der Hilfe des .NET Framework SDK.

6.8 Fehlerbehandlung im Code

Bei der Programmausführung kann es zu Fehlern kommen. Syntaktische Fehler im Code werden bereits zur Übersetzungszeit erkannt und vom Compiler gemeldet. Logische Fehler im Programmcode müssen Sie dagegen durch Debuggen herausfinden (siehe Kapitel 2). Als dritte Gruppe gibt es noch die sogenannten Laufzeitfehler. Diese treten auf, wenn das Programm versucht, eine unzulässige Operation auszuführen. Dies könnte beispielsweise der Zugriffsversuch auf eine Datei sein, die aber zur Laufzeit nicht vorhanden ist.

6.8.1 So reagiert das System auf Exceptions

Tritt ein Laufzeitfehler während der Programmausführung auf, löst das Laufzeitsystem eine Ausnahme (Exception) aus. Wurde keine Laufzeitfehlerbehandlung im Programmcode vorgesehen, hängt die genaue Reaktion von der Umgebung ab, in dem das Programm ausgeführt wird. Wird die Anwendung im Debugger der Entwicklungsumgebung ausgeführt, wird die Ausnahme vom Debugger abgefangen und löst die Anzeige des Ausnahmen-Assistenten aus (Abbildung 6.4, Hintergrund). Dieser zeigt eine Fehlerbeschreibung in einem eigenen Dialogfeld im Debugger an.

- Über das Feld *Hinweise zur Fehlerbehebung* erhalten Sie verschiedene Informationen zur Fehlerursache. Klicken Sie auf einen dieser Hyperlinks, wird die Hilfe mit dem betreffenden Thema aufgerufen. Sie erhalten dann nicht nur Hinweise zur Fehlerursache, sondern vielfach auch eine Beschreibung zur Behebung der Ursache.
- Klicken Sie auf den am unteren Dialogrand angezeigten Hyperlink *Details anzeigen*, öffnet sich ein Zusatzdialog (Abbildung 6.4, rechts unten), in dem Sie über das Fehlerobjekt zusätzliche Informationen zur betreffenden Ausnahme erhalten.

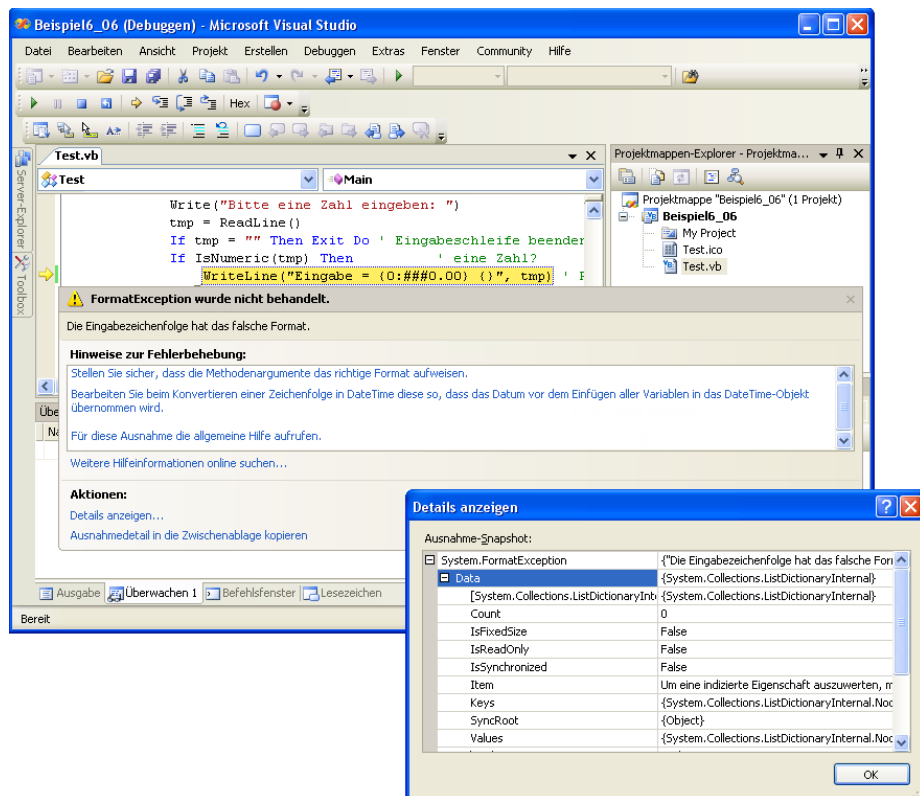


Abbildung 6.4: Anzeige einer Exception in der Entwicklungsumgebung

Bei Bedarf können Sie die Ausnahmedetails auch über den Hyperlink *Ausnahmedetails in die Zwischenablage kopieren* des *Exception*-Dialogs per Zwischenablage in ein Textdokument übertragen und speichern.

Wird die Anwendung dagegen direkt im Betriebssystem ausgeführt und ist keine Behandlung von Ausnahmen im Programm enthalten, löst die Exception im Laufzeitsystem die Anzeige eines Fehlerbericht-Dialogs aus (Abbildung 6.5). Es wird zwar der Name der Anwendung, aber keine Fehlerbeschreibung angegeben. Der Benutzer hat dann die Möglichkeit, einen Fehlerbericht an Microsoft zu senden oder mittels der Schaltfläche *Debuggen* einen Debugger aufzurufen. Wird diese Schaltfläche betätigt und ist ein Debugger installiert, erscheint dann der Dialog zur Auswahl des Debuggers.

Dann lässt sich ein Debugger (z.B. der Entwicklungsumgebung) wählen und der Programmcode inspizieren. Sind der Quellcode der Anwendung sowie die PDB-Datei mit den Debug-Informationen im Zielsystem verfügbar, kann die fehlerhafte Anweisung im Debugger angezeigt werden (siehe auch *Kapitel 2*). Andernfalls zeigt der Debugger die betreffende Stelle im Inline-Code an.

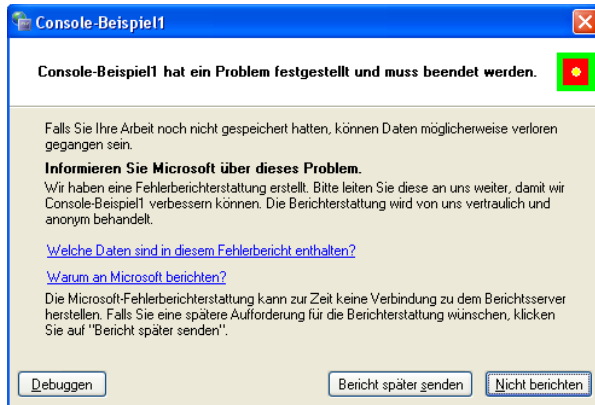


Abbildung 6.5: Exception bei der Ausführung im Betriebssystem

Aus Entwicklersicht ist das Auftreten von Laufzeitfehlern beim Kunden problematisch, da dies zum Abbruch der Anwendung führt, ohne dass die genaue Fehlerursache bekannt ist. Um dies zu vermeiden, kann eine Behandlung von Laufzeitfehlern im Programmcode erfolgen. Dann kann die Anwendung den vom Laufzeitsystem ausgelösten Fehler abfangen und in geeigneter Weise auf das Problem reagieren.

Hinweis

Sie finden das Beispiel, welches den oben gezeigten Laufzeitfehler mit der `FormatException` auslöst, im Ordner `\Beisp\Kap06\Exception` der Begleit-CD. Ursache für den Laufzeitfehler ist die Angabe zweier Platzhalter `{}` in der Anweisung `WriteLine("Eingabe = {0:###0.00} {}", tmp)`, wobei in der zweiten geschweiften Klammer die Nummer für das Argument fehlt. Sie können das Beispiel in der Entwicklungsumgebung und direkt aus dem Betriebssystem heraus aufrufen, um das Verhalten zu studieren.

6.8.2 Fehler mit `On Error` abfangen

Um Laufzeitfehler abzufangen, unterstützt Visual Basic die aus älteren Sprachdialekten bekannte `On Error`-Konstruktion. Mit der Anweisungsfolge

```
On Error Goto Fehler
... Anweisungen

Fehler:
... hier kommt die Fehlerbehandlung
```

lässt sich eine zentrale Fehlerbehandlung in einem Programm implementieren. Sobald eine Ausnahme durch einen Laufzeitfehler auftritt, verzweigt der Programmablauf zur Sprungmarke (hier *Fehler:*). Dort lässt sich dann Code zur Behandlung der Ausnahme hinterlegen.

Soll der Programmablauf beim Auftreten eines Fehlers fortgesetzt werden, fügen Sie die Anweisung *On Error Resume Next* im Programmcode ein.

6.8.3 Zugriff auf das Fehlerobjekt

Sobald eine Ausnahme auftritt, hinterlegt das Laufzeitsystem Informationen über den Fehler im systeminternen *Err*-Objekt. Da dieses Objekt einen globalen Gültigkeitsbereich besitzt, kann in Visual Basic direkt auf die Member des Objekts zugegriffen werden.

- Über den Member *Err.Description* lässt sich die Fehlerbeschreibung im Klartext abrufen.
- Der Member *Err.Number* enthält den internen Fehlercode des Laufzeitsystems.
- Das Modul, welches den Fehler ausgelöst hat, lässt sich über *Err.Source* ermitteln.
- Die Zeilennummer der zuletzt ausgeführten Anweisung wird in *Err.Erl* hinterlegt. Dieser Wert wird aber in der Regel 0 sein, da Visual Basic keine Zeilennummer ermitteln kann. Nur wenn Sie vor der betreffenden Anweisung, die den Laufzeitfehler auslöst, eine Marke der Art *10:* im Quellcode hinterlegen, wird *Erl* diesen Wert als Zeilennummer anzeigen.

Das nachfolgend gezeigte Programmbeispiel definiert ein Feld mit drei String-Elementen. Anschließend sollen die Feldelemente in einer Schleife auf der Konsole angezeigt werden. Die Schleife enthält eine Endebedingung, die vier Durchläufe zulässt. Dies sollte dann einen Laufzeitfehler beim Zugriffsversuch auf das Feld auslösen. Die *On Error Goto Fehler*-Anweisung im Programm bewirkt, dass dann der Programmablauf ab der betreffenden Marke *Fehler:* fortgesetzt wird. Dort wird im aktuellen Beispiel eine Sequenz an *WriteLine()*-Anweisungen verwendet, um aus dem Fehlerobjekt *Err* ermittelte Informationen anzuzeigen. Wichtig ist, dass Sie vor der betreffenden Marke eine Anweisung hinterlegen, die den Fehlerzweig überspringt, falls das Programm fehlerfrei abläuft. Im aktuellen Beispiel wird dies durch die eingefügte *Exit Sub*-Anweisung erreicht.

```
'*****
' File/Projekt: OnError
' Autor: G. Born www.borncity.de
' Demonstriert das Abfangen eines Laufzeitfehlers
' über On Error Goto xxx.
'*****
Option Strict On
Imports System.Console      ' für WriteLine
```

Listing 6.9: Exceptions mit *On Error Goto xxx* abfangen


```

Class Test
    Shared Sub Main()                ' das ist das Hauptmodul
        Dim i As Integer
        Dim txt() As String = {"Bach", "Bayer", "Born"}
        Dim tmp As String = "123"    ' Hilfsvariable für Eingaben

        On Error GoTo Fehler
        For i = 0 To 3
10:   WriteLine("Name: {0}", txt(i)) ' Feldwert ausgeben
        Next
        tmp = ReadLine()

        Exit Sub                    ' Fertig - > Main-Prozedur beenden
Fehler:
        WriteLine("Es ist ein Fehler aufgetreten")
        WriteLine("Fehler: {0}", Err.Description)
        WriteLine("Zeile: {0}", Err.Err)
        WriteLine("Code: {0}", Err.Number)
        WriteLine("Source: {0}", Err.Source)
        WriteLine("Hilfekontext: {0}", Err.HelpContext)
        WriteLine("Hilfedatei: {0}", Err.HelpFile)
        WriteLine("Zum Beenden die Eingabetaste drücken")
        tmp = ReadLine()
    End Sub
End Class

```

Listing 6.9: Exceptions mit On Error Goto xxx abfangen (Forts.)

Hinweis

Sie finden das Beispiel im Ordner `\Beisp\Kap06\OnError` auf der Begleit-CD. Beim Ausführen tritt sofort der Laufzeitfehler auf und einige Eigenschaften des `Err`-Objekts werden angezeigt. Im Ordner `\Beisp\Kap06\OnError1` der Begleit-CD finden Sie noch ein modifiziertes Beispiel, welches einen `FormatException`-Error beim Aufruf der `WriteLine()`-Methode abfängt.

6.8.4 Auslösen einer Exception per Programm

Sie können in Visual Basic auch programmgesteuert einen Laufzeitfehler auslösen, indem Sie die `Raise()`-Methode des `Err`-Objekts aufrufen. Die Methode erwartet eine Fehlernummer als Argument. Die folgende kleine Codesequenz demonstriert die Anwendung dieser Methode. Dabei kommt auch die `Resume Next`-Anweisung zum Einsatz, die beim Auftreten eines Laufzeitfehlers die Fortsetzung des Programms mit der nächsten Anweisung bewirkt.

```

'*****
' File/Projekt: RaiseError
' Autor: G. Born www.borncity.de
' Löst einen Laufzeitfehler aus und setzt den
' Code fort.
'*****
Option Strict On
Imports System.Console      ' für WriteLine

Class Test
    Shared Sub Main()        ' das ist das Hauptmodul
12: On Error Resume Next    ' bei Fehlern nicht abbrechen

14: Err.Clear()              ' Fehlerobjekt zurücksetzen
15: Err.Raise(6, , "Borns Laufzeitfehler") ' Ausnahme
    ' Infos ausgeben
    WriteLine("Fehler: {0}", Err.Description)
    WriteLine("Code: {0}", Err.Number)
    WriteLine("Source: {0}", Err.Source)
    WriteLine("Zeile: {0}", Err.Err)
    WriteLine("Zum Beenden die Eingabetaste drücken")
    ReadLine()
    End Sub
End Class

```

Listing 6.10: Auslösen eines Laufzeitfehlers

Hinweis

Sie finden das Beispiel im Ordner `\Beisp\Kap06\RaiseError` auf der Begleit-CD. Beim Ausführen tritt sofort der Laufzeitfehler auf und einige Eigenschaften des *Err*-Objekts werden angezeigt.

6.8.5 Laufzeitfehler mit Try ... Catch ... Finally behandeln

Der Einsatz der *On Error Goto*- bzw. *On Error Resume Next*-Konstrukte erlaubt zwar eine generelle Fehlerbehandlung in einer Anwendung, führt aber mitunter zu recht untransparentem Programmcode. Erwarten Sie an bestimmten Stellen im Code, dass Laufzeitfehler auftreten (z.B. beim Versuch eines Dateizugriffs), ist es ggf. sinnvoller, dort eine Laufzeitfehlerbehandlung einzubauen. Visual Basic unterstützt für diesen Zweck die *Try ... Catch ... Finally*-Konstruktion.

```

Try
  Try-Befehle
  [ Exit Try ]
[ Catch [ exception [ As type ] ] [ When expression ]
  Catch-Befehle
  [ Exit Try ] ]
[ Finally]
  [ finally Befehle]
End Try

```

Die hier in eckigen Klammern angegebenen Schlüsselwörter sind optional und können entfallen. Das Schlüsselwort *Try* leitet den Codeblock mit den auszuführenden Anweisungen ein. Am Ende dieser Sequenz kann noch die optionale *Try Exit*-Anweisung auftreten, um den Zweig vorzeitig zu verlassen. Tritt in den Anweisungen des *Try*-Zweigs ein Laufzeitfehler auf, kann dieser im *Catch*-Zweig abgefangen werden. Hinter dem Schlüsselwort *Catch* wird eine Bedingung zur Ausführung des Codeblocks angegeben. So könnten in verschiedenen *Catch*-Zweigen bestimmte Fehler abgefangen werden. Die Anweisungen im optionalen *Finally*-Zweig werden immer ausgeführt. Abgeschlossen wird der *Try*-Block mit einer *End Try*-Anweisungen. Um auf einen bestimmten Laufzeitfehler zu reagieren, ließe sich folgender Codeausschnitt verwenden:

```

Try
  sr As New IO.StreamReader ("Text.txt")
  ...
Catch ex As IO.FileNotFoundException
  MsgBox ("Datei Text.txt nicht gefunden!")
Catch ex As Exception
  MsgBox ("Fehler: " & err.Description)
End Try

```

Das nachfolgend gezeigte Listing demonstriert die Anwendung des *Try ... Catch*-Blocks zum Abfangen eines Laufzeitfehlers. Beim Start des Beispiels wird vom Benutzer eine Zahl zwischen 1 und 3 abgefragt. Anschließend versucht das Programm in einer Schleife im *Try*-Zweig die angegebene Anzahl an Namen aus einem Feld anzuzeigen. Erreicht der Schleifenindex den Wert 3, löst dies einen Laufzeitfehler aus, da die Feldgrenzen überschritten werden. Der Code zur Fehlerbehandlung findet sich im *Catch*-Block, dem hier eine *System.Exception* als Bedingung zugeordnet wurde. Konkret geben die Anweisungen die Fehlerinformationen auf der Konsole aus. Abschließend wird noch der *Finally*-Zweig durchlaufen. Drückt der Benutzer zum Abschluss die -Taste, terminiert das Programm.

```

'*****
' File/Projekt: TryCatch
' Autor: G. Born www.borncity.de
' Demonstriert das Abfangen eines Laufzeitfehlers
' über Try ... Catch.
'*****
Option Strict On
Imports System.Console      ' für WriteLine

Class Test
    Shared Sub Main()        ' das ist das Hauptmodul
        Dim i As Integer
        Dim x As Integer = 3
        Dim txt() As String = {"Bach", "Bayer", "Born"}
        Dim tmp As String = "123"    ' Hilfsvariable für Eingaben

        WriteLine("Bitte eine Zahl zwischen 1 und 3 eingeben")
        tmp = ReadLine()
        If IsNumeric(tmp) Then x = CInt(tmp)

        Try
            ' hier folgenden die zu überwachenden Anweisungen
            For i = 0 To x
                WriteLine("Name: {0}", txt(i)) ' Feldwert ausgeben
            Next
        Catch ex As Exception
            ' dies ist der Fehlerteil
            WriteLine("Es ist ein Fehler aufgetreten")
            WriteLine("Fehler: {0}", Err.Description)
            WriteLine("Code: {0}", Err.Number)
            WriteLine("Source: {0}", Err.Source)
        Finally
            ' dieser Teil wird immer ausgeführt
            WriteLine("Finally Zweig wurde erreicht")
        End Try

        WriteLine("Zum Beenden die Eingabetaste drücken")
        tmp = ReadLine()
    End Sub
End Class

```

Listing 6.11: Fehler mit Try ... Catch abfangen

Hinweis

Sie finden das Beispiel im Ordner `\Beisp\Kap06\TryCatch` auf der Begleit-CD. Durch Eingeben verschiedener Werte können Sie das Verhalten des Programms mit und ohne Laufzeitfehler analysieren.

Tritt innerhalb eines *Catch*-Zweigs ein zweiter Laufzeitfehler auf, findet keine Fehlerbehandlung statt. Das Laufzeitsystem löst dann eine Ausnahme aus, die zum Aufruf des Debuggers führt (siehe Ausführungen am Anfang des Abschnitts). Um diesen Fall ggf. abzufangen, können Sie einen zweiten *Try ... Catch ... End Try*-Block im *Catch*-Zweig unterbringen.

7

Objektorientierte Programmierung

In Visual Basic 2005 können Sie nicht nur die Klassen des .NET Framework nutzen. Visual Basic 2005 bietet Ihnen zusätzlich die Möglichkeit zur objektorientierten Programmierung. Sie können eigene Klassen mit Methoden und Eigenschaften erstellen oder von existierenden Klassen ableiten. In diesem Kapitel finden Sie eine Einführung in diese Themen.

7.1 Grundbegriffe

In Visual Basic 2005 begegnen Ihnen eine Reihe an Begriffen aus der objektorientierten Programmierung. Sie nutzen bei der objektorientierten Programmierung das Prinzip der Kapselung (von Daten) und die Techniken der Vererbung. Weiterhin greifen Sie auf Klassen zu, erzeugen Instanzen und verwenden deren Methoden und Eigenschaften. Nachfolgend möchte ich kurz auf die Thematik eingehen und die benutzte Nomenklatur vorstellen.

7.1.1 Was sind Klassen, Objekte, Eigenschaften und Methoden?

Klassen stellen so etwas wie Softwarebausteine (oder eine Blaupause) dar, die Funktionen und Prozeduren (auch Methoden genannt) bereitstellen und zudem Einstellmöglichkeiten (als Eigenschaften bezeichnet) besitzen. Zudem kann eine Klasse auch interne Variable enthalten, die dann als Felder (Fields) bezeichnet werden.

Die Methoden, Eigenschaften, Ereignisse, Konstruktoren und Felder einer Klasse stellen quasi deren Mitglieder dar und werden als Member bezeichnet. Um eine Klasse nutzen zu können, muss eine Instanz (quasi eine Kopie der Klasse im Arbeitsspeicher) erzeugt werden. Die Instanz einer Klasse wird dann ein Objekt bezeichnet, welches die Methoden, Eigenschaften, Ereignisse etc. der Klasse bereitstellt. Sie können im Programm viele Instanzen einer Klasse erzeugen, wobei jede dieser Instanzen ein Objekt darstellt.

Hinweis

Die Begriffe Klasse und Objekt werden manchmal synonym verwendet. Sie sollten sich aber merken, dass die Klasse ausschließlich die Struktur der Objekte beschreibt, während die Objekte im Programm verwendbare Instanzen von Klassen darstellen.

Klassen bieten also eine gute Möglichkeit, verwandte Elemente einer Gruppe zusammenzufassen und die Zugriffe auf diese Elemente zu steuern. Man spricht deshalb auch von Kapselung, d.h., die Klasseninstanz kapselt die enthaltenen Elemente und stellt Zugriffsmöglichkeiten auf diese Elemente bereit. Nehmen Sie beispielsweise die vom

.NET Framework bereitgestellte *Forms*-Klasse, die sich zur Erstellung von Formularobjekten verwenden lässt. Sie können nun mehrere Instanzen dieser Klasse anlegen, die alle separate Formulare (Objekte) darstellen. Jede dieser Instanzen ist eine exakte Kopie der Klasse, deren Daten von den Daten anderer Objekte dieser Klasse getrennt (gekapselt) sind. Die Formulare können dadurch verschiedene Inhalte, unterschiedliche Abmessungen, abweichende Titeltexpte etc. aufweisen. Die Kapselung durch die Klasseninstanz besitzt aber noch einen zweiten, bereits nebenbei erwähnten Vorteil: Es können standardisierte Zugriffe auf die Daten der Instanzen über Eigenschaften und Methoden bereitgestellt werden. Sprechen Sie die Eigenschaft eines Objekts an, bezieht sich dies auf eine Objektinstanz. Der Aufruf einer Methode führt zwar den in der Klasse hinterlegten Code aus. Die Methode bezieht sich aber auf die Instanz der Klasse (d.h. auf das Objekt).

Hinweis

Mit dem *Shared*-Modifizierer deklarierte, freigegebene Member stellen allerdings eine Ausnahme dar, da sie unabhängig von bestimmten Instanzen einer Klasse vorhanden sind (siehe folgende Abschnitte).

Diese Sichtweise erlaubt es, Softwarefunktionen auf globalere und abstraktere Weise zu definieren. Eine der grundlegenden Regeln der Kapselung ist, dass die Daten einer Klasse nur über Eigenschaften (implementiert über die *Property*-Prozeduren) oder Methoden geändert bzw. abgerufen werden sollten. Dadurch werden die Implementierungsdetails der Klassen komplett ausgeblendet, sprich: Als Softwareentwickler muss man die Interna einer Klasse nicht kennen. Es interessiert nur, welche Member (Methoden, Eigenschaften, Ereignisse) die Klasse bereitstellt und wie diese genutzt werden können. Dies verhindert die unerwünschte Verwendung der Klassendaten durch Anwendungscode. Zudem können die Daten später geändert werden, ohne dass dies zu Kompatibilitätskonflikten führt. So ließe sich in einer Klasse der Typ einer internen Variablen ändern, ohne dass dies Auswirkungen auf Programme hat. Die betreffende Klasse muss nur sicherstellen, dass Datenzugriff auf diese Variable über Eigenschaften oder Methoden weiterhin in der gewohnten Weise funktioniert. Bei einem direkten Zugriff auf die Werte einer Klasse von »Außen« wäre dies nicht sichergestellt. Wenn das »direkt zugreifende« Programm einen bestimmten Datentyp erwartet, dieser aber in der Klasse geändert wurde, führt dies zu einem Laufzeitfehler.

7.1.2 Vererbung von Klassen

In *Kapitel 6* wird die Definition von Datentypen als Strukturen gezeigt. Auf ähnliche Weise lassen sich auch Klassen zur Definition von Datentypen nutzen, da Klassen die betreffenden Elemente kapseln. Eine Besonderheit stellt aber die Möglichkeit der Vererbung von Klassen dar. Sie könnten einen Datentyp als Klasse vereinbaren, der einen Satz an verwandten Einzeldaten enthält. Im Unterschied zu Strukturen lässt sich dann eine neue Klasse aus dieser Grundklasse ableiten, die automatisch alle Merkmale der Grundklasse erbt. In der neuen Klasse ließen sich dann die Merkmale, z.B. durch Einfügen neuer Daten, erweitern. Klassen, die als Basis neuer Klassen fungieren, werden als Basis-klassen bezeichnet, während die neuen Klassen »abgeleitete Klassen« genannt werden. Abgeleitete Klassen erben alle Felder, Eigenschaften, Methoden und Ereignisse der

Basisklasse. Die so abgeleiteten Klassen lassen sich natürlich erneut vererben. Man spricht in diesem Zusammenhang von Mehrfachvererbung.

Was sind freigegebene Member?

Gemäß den obigen Erläuterungen sind alle Member einer Klasse klasseninstanzspezifisch, d.h., eine Eigenschaft oder eine Methode bezieht sich immer auf die Klasseninstanz (also auf das betreffende Objekt). Gelegentlich ist es jedoch erwünscht, dass ein einzelnes Element der Klasse gemeinsam von allen Objekten genutzt werden soll. In diesem Fall lässt sich der betreffende Member über den Modifizierer *Shared* freigeben. Dies gewährleistet, dass z.B. eine Variable in allen Instanzen einer Klasse den gleichen Wert hat. Geben Sie auf diese Weise Methoden frei, lassen sich diese mittels des Klassennamens, also ohne eine Instantiierung der Klasse, direkt aufrufen.

Diesen Ansatz haben Sie übrigens bereits in Kapitel 3 im Abschnitt »Bei Klassen heißt es aufpassen« kennen gelernt. Dort wurde erwähnt, dass die in einer Klasse hinterlegte Startprozedur *Main()* eines Programms mit dem Schlüsselwort *Shared* versehen werden muss. Dadurch lässt sich die Startprozedur aufrufen, ohne dass die Klasse mit *New* als Objekt instantiiert werden muss.

Zudem ist vorstellbar, Klassenvariablen über *Public* generell für einen Zugriff von außen freizugeben. Dies sollte aber im Hinblick auf die Kapselung der Interna einer Klasse verpönt sein. Denkbar ist es aber, dass bestimmte Klassenvariablen in allen Klasseninstanzen den gleichen Wert aufweisen sollen (quasi eine gemeinsame Variable für alle Objekte). Dies lässt sich erreichen, indem diese Variablen als *Public Shared* in der Klasse deklariert werden. Die so deklarierte Klassenvariable verhält sich wie eine *Public*-Klassenvariable eines Moduls (siehe unten). Da diese öffentlichen Variablen aber unbeabsichtigte Nebenwirkungen haben können (der Wert ist ja für alle Objekte gleich), sollten Sie solche *Public Shared*-Variablen nicht oder nur sehr vorsichtig einsetzen.

7.1.3 Bemerkungen zu Modulen

In Kapitel 3 wurde bereits gezeigt, wie die Grundstruktur einer Klasse aussieht und wie Sie auf die Member einer Klasse (z.B. die als Felder bezeichneten internen Variablen) zugreifen. In diesem Kapitel haben Sie auch Module kennen gelernt. Module stellen einen Sonderfall von Klassen dar, die eine Deklarationen von Variablen, Prozeduren etc. enthalten können. Allerdings weisen die Module einige Einschränkungen auf.

- So lassen sich Module nicht instantiieren, d.h. es gibt auch keinen *New*-Konstruktor (siehe auch folgende Abschnitte). Module können nicht vererbt werden noch können Sie Klassen erben.
- Ein Modul muss direkt auf der untersten Ebene der Codedatei hinterlegt werden und kann selbst keine weiteren Module enthalten. Es lassen sich aber Klassen innerhalb eines Moduls implementieren.
- Es kann direkt auf die innerhalb des Moduls enthaltenen Prozeduren und öffentlichen Variablen zugegriffen werden. Die betreffenden Prozeduren entsprechen den als *Shared* freigegebenen Methoden einer Klasse.

Zudem lassen sich für Module keine Schnittstellen implementieren. Module kommen zum Einsatz, wenn global verfügbare Konstanten, Variablen oder Prozeduren deklariert werden sollen. Module werden beispielsweise in den Vorlagen von Konsoleanwendungen benutzt, da sich dann die *Main()*-Prozedur als Startpunkt für das Programm angeben lässt. Bei Klassen müssen Sie dagegen die betreffende Prozedur über *Shared* als Member freigeben.

7.1.4 Schnittstellen

Schnittstellen (Interfaces) ermöglichen eine Beschreibung gemeinsamer Merkmale unterschiedlicher Klassen (und Strukturen). Eine Schnittstelle enthält keinen Code und besteht aus einer Aufzählung der Deklarationen von Methoden, Eigenschaften und Ereignissen. Klassen, die eine solche Schnittstelle implementieren, müssen dann für die in der Schnittstelle aufgeführten Deklarationen den entsprechenden Code enthalten. Schnittstellen erlauben also bei Verwendung von Klassen eine Prüfung, ob die Klasse die geforderten Methoden, Eigenschaften und Ereignisse auch in der deklarierten Weise enthält. Ein Beispiel lernen Sie in einem getrennten Abschnitt kennen.

7.1.5 Klassen über Namespaces identifizieren

Da die .NET-Klassenbibliothek eine Vielzahl von Klassen bereitstellt, die zudem noch in verschiedenen Assemblies (DLL-Bibliotheksdateien) hinterlegt sind, benötigt man eine strenge Hierarchie. Zudem können Sie ebenfalls eigene Klassen erstellen, die ihrerseits Klassen enthalten. Ein Problem gibt es speziell dann, wenn einzelne Klassen innerhalb der Klassenhierarchie sogar gleiche Namen aufweisen. Dann ist unklar, welche Klasse anzusprechen ist.

Um einen gezielten Zugriff auf bestimmte Klassen (auch bei mehrfacher Verwendung eines Namens für verschiedene Klassen) sicherzustellen, wird das Konzept der bereits erwähnten Namespaces (Namensräume) benutzt. Ein Namespace ist nichts anderes als ein in .NET Framework verwendeter logischer Container für die Organisation der Typen (z.B. Klassen), die in Anwendungen verwendet werden. Wie Sie Namensräume für eigene Klassen definieren, wird im Laufe dieses Kapitels an einem Beispiel gezeigt. Um im Visual-Basic-Quellcode eine spezielle Klasse innerhalb der Klassenbibliothek zu nutzen, muss genau angegeben werden, wo diese zu finden ist.

Wenn Sie in der Entwicklungsumgebung einen Verweis zum Projekt hinzufügen, bedeutet dies nichts anderes, als dass ein externes »Objekt« eingebunden wird. Der Verweis kann zum Beispiel auf eine Assembly zeigen, die u.a. (ähnlich wie eine Typ-Bibliothek) die Beschreibung der verfügbaren Klassen samt deren Member enthält. Eine Assembly kann nun einen oder mehrere Namenräume enthalten.

Um auf bestimmte Elemente wie z.B. Klassen zuzugreifen, müssten Sie den Namespace mit angeben, in dem die Klasse hinterlegt ist (z.B. *System.Console* adressiert die Klasse *Console* im Namespace *System*). Eine Angabe der Art *System.Console.WriteLine()* ruft beispielsweise die Methode *WriteLine()* der Klasse *Console* auf. Bei Angaben dieser Art spricht man auch von vollgekennzeichneten (vollqualifizierten) Namen (es wird die Langform benutzt). Vollqualifizierte Namen stellen sicher, dass die gewünschte Klasse bzw. deren Objektinstanzen eindeutig im Programmcode angesprochen wird. Um die Schreibarbeit zu reduzieren, unterstützt Visual Basic die *Imports*-Anweisung, die außerhalb eines Modul- oder Klassennamens angegeben werden muss, in folgenden Varianten:

```
Imports Namespace  
Imports Alias = Namespace
```

In der ersten Variante wird nur der Namensraum spezifiziert, während in der zweiten Zeile ein Alias (Kurzname) für Zugriffe auf den Namespace vereinbart wird. Verwenden Sie eine *Imports*-Anweisung ohne Alias, können Sie auf alle Namen in diesem Namespace ohne Qualifizierung zugreifen (z.B. *WriteLine()*). Allerdings ist Voraussetzung, dass die Namen im Projekt eindeutig sind. Enthält das Projekt jedoch *Imports*-Anweisungen für Namespaces, die Elemente mit gleichem Namen umfassen, bliebe Ihnen nur die Verwendung vollqualifizierter Namen. Um sich Schreibarbeit zu sparen, können Sie für die verschiedenen Namespaces jedoch Aliasnamen in den *Imports*-Anweisungen definieren. Dann lässt sich beim Zugriff auf die Klassen und deren Member der Alias statt des vollqualifizierten Namens verwenden und dem Klassennamen voranstellen (z.B. ist *con* als *Imports con = System.Console* vereinbart, können Sie die verkürzte Schreibweise *con.WriteLine()* im Programm nutzen).

Hinweis

Wenn Sie Klassen des .NET Framework nutzen möchten, müssen Sie zuerst herausfinden, welche Klasse Sie eigentlich benötigen. Dann ist der Namensraum, in dem die Klasse hinterlegt ist, zu ermitteln. Der Namespace wird in der Hilfe zum .NET Framework bei der betreffenden Methode aufgeführt. Zudem sehen Sie den Namensraum auch, wenn Sie eine Methode oder eine Eigenschaft im Objektkatalog abrufen.

7.1.6 Im Code mit Objekten arbeiten

Im Visual-Basic-2005-Code können Sie Objekte (mit *New*) instantiieren und dann auf deren Eigenschaften und Methoden zugreifen. Letztendlich haben Sie das im Rahmen der bisherigen Beispiele bereits mehrfach genutzt. Zur Vertiefung möchte ich auf diesen Aspekt noch etwas eingehen. Der Zugriff auf eine Eigenschaft erfolgt über den Objektnamen, gefolgt von einem Punkt und dem Namen der jeweiligen Eigenschaft. Dies soll jetzt nochmals exemplarisch am Beispiel eines Formulars demonstriert werden.

Hinweis

Formulare für Windows lassen sich über Instanzen der Klasse *Form* im Namensraum *System.Windows.Forms* bereitstellen. Im Projekt ist daher ein Verweis auf *System.Windows.Forms* erforderlich und im Programmcode des *Class*-Elements ist ein Import des Namensraums *System.Windows.Forms* hinzuzufügen, um nicht mit vollqualifizierten Namen arbeiten zu müssen. Wenn Sie in der Entwicklungsumgebung eine Windows-Anwendung anlegen, wird automatisch der Code zur Anzeige des Hauptformulars über die betreffende Vorlage im Projekt hinterlegt. Sie müssen sich dann also um nichts mehr kümmern.

Eine Instanz des Formularobjekts lässt sich aus der Klasse *Form* über die folgende Anweisung erzeugen:

```
Dim Form1 As New Form
```

Das Schlüsselwort *New* erzeugt dabei die neue Instanz und weist das Objekt der Objektvariablen *Form1* zu. Sie können nun über die Objektvariable *Form1* auf die von der Klasse *Form* angebotenen Eigenschaften und Methoden zugreifen. Der Titeltext im Formularfenster wird über die *Text*-Eigenschaft definiert. Mit *Height* lässt sich die Formularhöhe festlegen, während die Eigenschaft *Width* die Formularbreite angibt. Die folgende Sequenz definiert einen Titel, die Formularabmessungen sowie die Formularposition:

```
Form1.Text = "Testformular" ' Titeltext
Form1.Width = 300          ' Breite des Formulars
Form1.Height = 200         ' Höhe des Formulars
Form1.Left = 100           ' X/Y-Position des linken/oberen
Form1.Top = 200            ' Formlarrands zum Desktop "
```

Es wird also der Objektname und der Name der Eigenschaft, getrennt durch einen Punkt, in der Anweisung kombiniert.

Werfen wir nun noch einen kurzen Blick auf den Einsatz von Methoden. Die meisten Klassen bzw. Objekte bieten Methoden an, die sich auf die Objekte anwenden lassen. Um auf eine Methode zuzugreifen, ist deren Name, getrennt durch einen Punkt, hinter dem Objektnamen anzugeben. Die folgende Anweisung

```
Form1.ShowDialog()
```

ruft die *ShowDialog()*-Methode auf. Diese Methode erwartet keine Parameter und bewirkt die Anzeige des Formulars (siehe auch folgende Abschnitte).

Hinweis

Erinnern Sie sich noch an die Verwendung von Funktionen und Prozeduren im vorhergehenden Kapitel? Eine Funktion oder Prozedur wird über ihren Namen aufgerufen, wobei Parameter in Klammern zu übergeben sind. Eine Funktion liefert dabei noch einen Wert an das rufende Programm zurück. Diese Funktionen oder Prozeduren müssen aber im Quellcode des Programms mit eingegeben werden und können nur innerhalb der Anwendung aufgerufen werden. Ein Methodenaufruf ist letztendlich der Aufruf einer Prozedur oder einer Funktion, die in der dem Objekt zugrunde liegenden Klasse hinterlegt ist. Daher muss beim Methodenaufruf der Name des Objekts mit angegeben werden. Ob eine Methode als Prozedur oder Funktion arbeitet, hängt nur davon ab, ob ein Rückgabewert geliefert wird. Dies ist natürlich ein mächtiger Ansatz. Statt die Funktionen oder Prozeduren mühsam in Ihrem Quellcode einzubauen, benötigen Sie nur ein entsprechendes Objekt. Schon können Sie die von diesem Objekt angebotenen Methoden aus beliebigen Anwendungen nutzen (Sie müssen nicht einmal den Quellcode kennen, es genügt die Beschreibung der Schnittstellen der Methode mit den Parametern, Rückgabewerten und der Funktionalität). Objekte können ihrerseits wieder als Container für Unterobjekte dienen. Es ergibt sich dann ggf. eine Objekthierarchie, wobei über die Objektnamen von den Elternobjekten auf die Kindobjekte zugegriffen werden kann.

Das nachfolgend gezeigte Listing enthält den Code, um ein Formular zweimal anzuzeigen. Dabei wird auch die Verwendung der *With*-Anweisung demonstriert. Innerhalb des *With*-Blocks braucht der Objektname beim Zugriff auf Eigenschaften oder Methoden dann nicht mehr angegeben werden.

```
'*****
' File/Projekt: Beispiel07_01
' Autor: G. Born www.borncity.de
' Demonstration des Zugriffs auf Eigenschaften
' und Methoden eines Form-Objekts.
'*****
Option Strict On

Imports System.Windows.Forms

Class Test
    Shared Sub Main()
        Dim Form1 As New Form          ' Instanz anlegen

        Form1.Text = "Testformular"    ' Titeltext
        Form1.Width = 300              ' Breite des Formulars
        Form1.Height = 200             ' Höhe des Formulars
        Form1.Left = 100               ' X-Abstand
        Form1.Top = 200                ' Y-Abstand
        Form1.ShowDialog()             ' Formular anzeigen

        With Form1
            .Text = "Testformular - neu" ' Titeltext
            .Width = 300                 ' Breite des Formulars
            .Height = 200               ' Höhe des Formulars
            .Left = 300                 ' Abstand linker Rand Desktop
            .Top = 500
            .ShowDialog()               ' Formular anzeigen
        End With
    End Sub
End Class
```

Listing 7.1: Zugriffe auf die Eigenschaften eines Arrays

Hinweis

Sie finden die Projektdateien dieses als Windows-Anwendung realisierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_01` auf der Begleit-CD. Sobald Sie die .NET-Anwendung ausführen, wird ein leeres Formular auf dem Desktop angezeigt. Schließen Sie das Formular über seine in der rechten oberen Fensterecke angezeigte *Schließen*-Schaltfläche, wird das Formular mit neuem Titel an einer etwas anderen Stelle des Desktop eingeblendet. Größe, Titel und Position der Formulare werden über die betreffenden Eigenschaften im Programm gesetzt.

7.2 Eigene Klassen implementieren

In diesem Abschnitt lernen Sie, wie sich Klassen implementieren und mit Klassenvariablen, Methoden und Eigenschaften ausstatten können. Instantiieren Sie eine Klasse, lässt sich über die betreffende Objektinstanz auf die Methoden und Eigenschaften zugreifen.

7.2.1 Eine Klasse mit Klassenvariablen definieren und nutzen

Für die ersten Schritte soll nun eine sehr einfache Klasse mit zwei Variablen definiert und zur Untersuchung des Verhaltens erzeugt werden. Eine Klasse lässt sich mit wenigen Anweisungszeilen definieren. Sehen wir uns einmal den folgenden Programmcode an:

```
Class MyData
    Friend Name As String = "Born"
    Friend Alter As Integer = 20
End Class
```

Die Anweisungen definieren die Klasse *MyData* mit den beiden Variablen *Name* und *Alter*. Diese stellen Klassenvariablen dar, da sie interne Variablen der Klasse sind. Im Hinblick auf eine saubere Kapselung der Klasse sind solche Klassenvariablen standardmäßig *Private*, d.h. von außerhalb der Klasse nicht zugreifbar. Um in diesem Beispiel überhaupt etwas mit der Klasse anfangen zu können, habe ich hier die Deklarationen mit dem Schlüsselwort *Friend* versehen. Damit lässt sich zumindest in der Assembly auf die Werte zugreifen.

Hinweis

Sofern Sie die Klassenvariablen mit dem Zugriffsmodifizierer *Public* deklarieren, sind die Variablen sogar öffentlich (zugreifbar aus allen Assemblies). Fügen Sie noch das Schlüsselwort *Shared* hinzu, ist der Member nicht nur öffentlich, sondern auch freigegeben, d.h., in allen Instanzen der Klasse besitzt die Variable den gleichen Wert – das Prinzip der Kapselung wäre dann aufgehoben.

Möchten Sie nun die Klasse innerhalb der Hauptprozedur verwenden, müssen als Erstes entsprechende Objektvariable deklariert werden.

```
Dim pers1 As MyData
Dim pers2 As MyData
```

Den beiden Variablen wird als Datentyp die betreffende Klasse zugewiesen. Um jetzt auf die betreffenden Objekte zugreifen zu können, muss die betreffende Klasse mit dem Schlüsselwort *New* instantiiert werden.

```
pers1 = New MyData()
pers2 = New MyData()
```

Falls Sie eine kürze Schreibweise bevorzugen, lässt sich die Deklaration der Variablen und die Instantiierung über *New* in eine Zeile packen (*Dim pers1 As New MyData()*). Nach diesen Schritten existieren Instanzen der Klasse, die sich über die Objektvariable *pers1* und *pers2* ansprechen lassen. Die Objekte stellen zwei Member *Name* und *Alter* bereit, die nach der Instantiierung mit den gleichen Werten belegt sind. Da es sich bei *pers1* und *pers2* aber um zwei unterschiedliche Objekte mit getrennten Speicherbereichen handelt, lassen sich deren Membern neue Werte zuweisen.

```
pers1.name = "Meier"
pers1.alter = 18

pers2.name = "Bach"
pers2.alter = 13
```

Abbildung 7.1 zeigt die Ausgaben eines entsprechenden Beispielprogramms. In der ersten und dritten Ergebniszeile sehen Sie die Werte der Member, wie sie direkt nach der Instantiierung der Objekte vorliegen. Obwohl es sich um unterschiedliche Objektvariable handelt, weisen deren Member *Name* und *Alter* die gleichen Werte auf. Dies liegt daran, dass die Werte bei der Instantiierung durch den Code der Klasse vorbelegt wurden. Die zweite und vierte Ausgabezeile zeigt dagegen den Wert der Member *Name* und *Alter*, nachdem diesen im Hauptprogramm neue Werte zugewiesen wurden. Sie erkennen, dass nun unterschiedliche Werte für den Namen und das Alter einer Person angegeben werden. Das folgende Listing zeigt die Einzelheiten des Beispielprogramms:

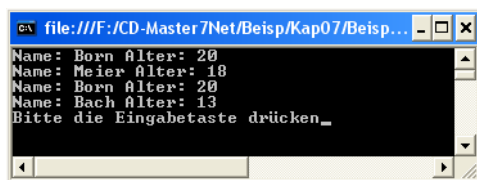


Abbildung 7.1: Ausgabe des Beispielprogramms

```
'*****
' File/Projekt: Beispiel07_02
' Autor: G. Born www.borncity.de
' Erstelle eine Klasse mit Konstanten und Variablen.
'*****

Option Strict On
Imports System.Console      ' für WriteLine

Class MyData
    Friend Name As String = "Born"
    Friend Alter As Integer = 20
End Class

Class Test
    Shared Sub Main()
        Dim pers1 As MyData      ' Typ auf Klasse setzen
        Dim pers1 As MyData      ' "

        pers1 = New MyData()      ' erzeuge neue Objektinstanzen
        pers2 = New MyData()

        With pers1
            WriteLine("Name: {0} Alter: {1}", .Name, .Alter)
            .Name = "Meier"        ' neue Werte zuweisen
            .Alter = 18
            WriteLine("Name: {0} Alter: {1}", .Name, .Alter)
        End With

        With pers2
            WriteLine("Name: {0} Alter: {1}", .Name, .Alter)
            .Name = "Bach"        ' neue Werte zuweisen
            .Alter = 13
            WriteLine("Name: {0} Alter: {1}", .Name, .Alter)
        End With
        Write("Bitte die Eingabetaste drücken")
        ReadLine()
    End Sub
End Class
```

Listing 7.2: Implementieren einer Klasse und Nutzung als Objekt

Hinweis

Auf den ersten Blick sieht es so aus, als ob die öffentlichen Klassenvariablen sich wie Eigenschaften verhalten, d.h., sie lassen sich direkt über den Objekt- und Eigenschaftennamen (*pers1.Name*) lesen und schreiben. Allerdings handelt es sich um keine Eigenschaften, da diese auf spezielle Weise in der Klasse implementiert werden müssen (siehe folgende Seiten). Wenn Sie sich die Klasse im Objektbrowser oder im Klassendesigner ansehen, zeigt dieser daher die betreffenden Variablen als Klassenvariablen mit dem betreffenden Zugriffsmodifizierer (*Private*, *Friend* etc.) und nicht mit dem Symbol für Eigenschaften an.

Sie finden die Projektdateien des als Konsoleanwendung ausgeführten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_02` auf der Begleit-CD. Die Wirkung von *Friend*, *Public*, *Private* oder *Shared* zu demonstrieren, finden Sie ein leicht modifiziertes Beispiel im Ordner `\Beisp\Kap07\Beispiel7_02a`. Die Klasse *MyData* wurde um zwei weitere Klassenvariablen *Geschlecht* und *Version* ergänzt. Die Variable *Geschlecht* ist dabei als *Private* deklariert, d.h., Sie können von außerhalb der Klasse nicht auf dieses Feld zugreifen. Die Variable *Version* habe ich als öffentlichen Member mit *Public Shared* freigegeben. Wenn Sie das Beispiel ausführen und den zugehörigen Code studieren, werden Sie feststellen, dass der Wert dieses Felds in allen Instanzen der Klasse gleich ist. Im Beispiel wird der Wert in der Klasse initialisiert und dann in der Klasseninstanz *Pers1* geändert. Dieser Wert ist dann auch im Feld *Pers2.Version* vorhanden. Sie sollten sich also merken, dass Klassenvariablen separat in den Instanzen gehalten werden. Nur mit *Shared* freigegebene Member besitzen den gleichen Wert in allen Klasseninstanzen.

7.2.2 So bekommt die Klasse Eigenschaften

Das obige Beispiel benutzt lediglich einige Variable (Felder) innerhalb einer Klasse, die über entsprechende Zugriffsmodifizierer als öffentliche Member freigegeben wurden. Um das Prinzip der Kapselung zu nutzen, sollten Zugriffe auf Daten des Objekts über Eigenschaften erfolgen. Eingangs dieses Kapitels wurde ja bereits erwähnt, dass Klassen bzw. deren Objekte über Eigenschaften verfügen können, die sich lesen und ggf. setzen lassen. Schauen wir uns doch einmal an, wie sich eine solche Klasse mit Eigenschaften implementieren lässt und wo der Unterschied zu den im vorherigen Abschnitt beschriebenen Klassenvariablen liegt. Als Beispiel möge eine Klasse dienen, die Eigenschaften einer Wetterstation implementiert. Die Wetterstation misst die Temperatur, die Luftfeuchtigkeit, den barometrischen Druck und den Niederschlag. Da es mehrere Wetterstationen geben kann, reicht es, zur Speicherung der Daten, eine weitere Instanz der Klasse anzulegen. Mit dem obigen Ansatz einer Klasse, die Klassenvariablen aufweist, ließe sich dies zwar bereits realisieren. Aber der direkte Zugriff auf die Klassenvariablen kann bei Änderungen innerhalb der Klasse zu Problemen führen. Zudem gilt es zu bedenken, dass die Messgeräte in den Wetterstationen feste Messbereiche besitzen. Ein negativer Niederschlag macht keinen Sinn, eine Temperatur von 100 Grad ebenfalls nicht, da der Messbereich von -40 bis +80 Grad Celsius reicht. Die Luftfeuchtigkeit reicht von 0 bis 100 Prozent und der Druck von 700 hPa bis 1.500 hPa (Hectopascal (hPa)). Unsinnige Werte sind also abzulehnen. Zur Vereinfachung des Beispiels soll das Objekt die betreffenden Eigenschaften dann auf den jeweils nächsten Grenzwert legen.

Eigenschaften werden innerhalb der Klasse im Gegensatz zu Variablen über zwei Prozedur- bzw. Funktionsaufrufe mit den Namen *Get* und *Set* implementiert. Liest ein Anwendungsprogramm die Eigenschaft eines Objekts, wird intern die *Get*-Funktion aufgerufen, die dem rufenden Programm den aktuellen Eigenschaftenwert überträgt. Weist die Anwendung der Eigenschaft einen Wert zu, löst dies intern die *Set*-Prozedur aus. In beiden Prozeduren lässt sich nun aber Code hinterlegen, der eine Überprüfung der angegebenen Eigenschaftenwerte vornimmt bzw. die Rückgabe des Eigenschaftenwerts übernimmt. Das folgende Codefragment zeigt, wie sich eine Eigenschaft *Temperatur* innerhalb der Klasse *Wetter* implementieren lässt:

```
Class Wetter
  Private Temp As Integer = 0

  Property Temperatur () As Integer ' Temperaturwert
    Get
      Return Temp                  ' Temperatur lesen
    End Get

    Set (byVal Wert As Integer)     ' Temperatur schreiben
      Temp = Wert                  ' Speichern
      If Wert < -40 then Temp = -40 ' auf Grenzen fixieren
      If Wert > 80 then Temp = 80
    End Set
  End Property

End Class
```

Die Klasse wird mit dem Schlüsselwort *Class* vereinbart. Die auf Klassenebene deklarierte Variable *Temp* dient zur Speicherung der Temperaturwerte. Durch die Vereinbarung als *Private* lässt sich auf die Variable außerhalb der Klasse nicht zugreifen. Eine Eigenschaft der Klasse wird dann über die Schlüsselwörter

```
Property Name () As Typ
...
End Property
```

vereinbart. Innerhalb des *Property*-Blocks findet sich der Programmcode zur Implementierung der Eigenschaft. Der Name der Eigenschaft wird über den Platzhalter *Name* in der *Property*-Deklaration festgelegt. Da Eigenschaften so etwas wie Variable darstellen, besitzen diese auch einen Typ, der in der Deklaration aufgeführt wird. In obigem Programmfragment wurde die Eigenschaft *Temperatur* mit dem Datentyp *Integer* versehen.

Tipp

Achten Sie darauf, dass der Typ der Eigenschaft möglichst mit dem Typ der lokalen Klassenvariable übereinstimmt. Andernfalls sind Typkonvertierungen innerhalb der Klasse unumgänglich.

Sobald die Eigenschaft deklariert wurde, muss nur noch der Programmcode zum Lesen und Schreiben der Werte hinterlegt werden. Die Funktion *Get*, die beim Lesezugriff auf die Eigenschaft ausgeführt wird, lässt sich innerhalb der Klasse folgendermaßen implementieren:

```
Get
    Return Temp                ' Temperatur lesen
End Get
```

Die Funktion liefert einfach den Wert der lokalen Variable, der in der Klasse zur Speicherung der Eigenschaft vorgesehen wurde, an das rufende Programm zurück. Handelt es sich dagegen um eine nicht änderbare Eigenschaft, können Sie auch einen Konstantenwert zurückgeben (z.B. in der Art *Return 100*).

Achtung

Obwohl *Get* eine Funktion ist, dürfen Sie keine Klammern (z.B. *Get ()*) mit angeben. Andernfalls löst der Compiler eine Fehlermeldung aus.

Versucht ein Programm den Wert einer Eigenschaft zu ändern, wird intern im Objekt die *Set*-Prozedur aufgerufen.

```
Set (byVal Wert As Typ)
... Code zum Setzen der Eigenschaft
End Set
```

Set erwartet den Wert für die Eigenschaft als Parameter. Dabei muss der Typ des Arguments mit dem Datentyp der Eigenschaft übereinstimmen. Innerhalb der Prozedur können Sie beliebigen Code unterbringen. Im obigen Beispiel wurden folgende Anweisungen verwendet:

```
Set (byVal Wert As Integer)    ' Temperatur schreiben
    Temp = Wert                ' Speichern
    If Wert < -40 then Temp = -40 ' auf Grenzen fixieren
    If Wert > 80 then Temp = 80
End Set
```

Innerhalb der *Set*-Prozedur wird einfach der Wert der lokalen Variablen *Temp* zugewiesen. Dann prüft die Prozedur, ob der Wert innerhalb des Messbereichs liegt und setzt ggf. einen fehlerhaften Wert auf die jeweilige Messbereichsgrenze. Natürlich können Sie

zusätzlichen Code verwenden, der zum Beispiel einen Fehlerstatus setzt oder irgend etwas anderes veranlasst.

Im Anwendungsprogramm lässt sich das Objekt mit folgenden Anweisungen instantiieren und mit Werten belegen:

```
Dim station1 As Wetter
station1 = New Wetter()      ' Objekt instantiieren
station1.Temperatur = 35
```

Im ersten Schritt wird die Objektvariable *station1* vereinbart. Dieser wird der Typ *Wetter* zugewiesen, was dem Klassennamen entspricht. Die zweite Anweisungszeile instantiiert über das Schlüsselwort *New* das Objekt. Dann lässt sich über die Objektvariable auf die Eigenschaft *Temperatur* zugreifen. Das folgende Listing zeigt den kompletten Quellcode des Beispiels. In der Klasse *Wetter* wurden insgesamt vier Eigenschaften als Member implementiert. Das Hauptprogramm deklariert zwei Objektvariable *station1* und *station2* für zwei Wetterstationen und erzeugt zwei Instanzen der Klasse *Wetter*. Anschließend werden den Eigenschaften der beiden Objekte Werte zugewiesen. Die den Eigenschaften des zweiten Objekts zugewiesenen Werte liegen teilweise bewusst außerhalb der Grenzwerte. Nachdem die Eigenschaften belegt wurden, zeigt das Programm die aktuellen Werte auf Konsolebene an. Die Abbildung 7.2 zeigt die Daten beider Stationen. Bei Station 2 ist erkennbar, dass die fehlerhaften Werte nicht übernommen, sondern durch die jeweiligen Messbereichsgrenzen ersetzt wurden. Weitere Details sind dem folgenden Listing zu entnehmen:

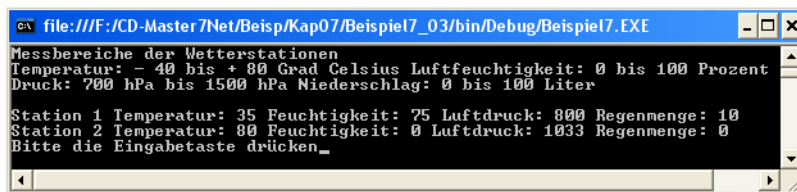


Abbildung 7.2: Ausgaben der Wetterdaten

```
'*****
' File/Projekt:  Beispiel07_03
' Autor: G. Born www.borncity.de
' Erstelle eine Klasse mit Eigenschaften für eine
' Wetterstation.
'*****
Option Strict On
Imports System.Console      ' für WriteLine

Class Wetter
    ' Member der Klasse - Eigenschaften
```

Listing 7.3: Beispiel zur Implementierung von Eigenschaften

```
'
' Temperatur: - 40 bis + 80 Grad Celsius
' Luftfeuchtigkeit: 0 bis 100 Prozent
' Druck: 700 hPa bis 1500 hPa
' Niederschlag: 0 bis 100 Liter
Private Temp As Integer = 0      ' private Klassenvariablen
Private Humid As Integer = 0     ' initialisieren
Private Press As Integer = 0
Private Rain As Integer = 0

Property Temperatur() As Integer  ' Temperaturwert
    Get
        Return Temp              ' Temperatur lesen
    End Get

    Set(ByVal Wert As Integer)    ' Temperatur schreiben
        Temp = Wert              ' Speichern
        If Wert < -40 Then Temp = -40 ' auf Grenzen fixieren
        If Wert > 80 Then Temp = 80
    End Set
End Property

Property Feuchtigkeit() As Integer ' Luftfeuchtigkeit
    Get
        Return Humid             ' Wert lesen
    End Get

    Set(ByVal Wert As Integer)    ' Wert schreiben
        Humid = Wert             ' Speichern
        If Wert < 0 Then Humid = 0 ' auf Grenzen fixieren
        If Wert > 100 Then Humid = 100
    End Set
End Property

Property Luftdruck() As Integer   ' Luftdruck
    Get
        Return Press             ' Wert lesen
    End Get

    Set(ByVal Wert As Integer)    ' Wert schreiben
```

Listing 7.3: Beispiel zur Implementierung von Eigenschaften (Forts.)

```

        Press = Wert                ' Speichern
        If Wert < 700 Then Press = 700 ' auf Grenzen fixieren
        If Wert > 1500 Then Press = 1500
    End Set
End Property

Property Regenmenge() As Integer    ' Niederschlag
    Get
        Return Rain                ' Wert lesen
    End Get

    Set(ByVal Wert As Integer)      ' Wert schreiben
        Rain = Wert                ' Speichern
        If Wert < 0 Then Rain = 0   ' auf Grenzen fixieren
        If Wert > 100 Then Rain = 100
    End Set
End Property
End Class

Class Test
    Shared Sub Main()                ' das ist das Hauptmodul
        ' Zeige jetzt, wie sich die Daten der Klasse verwenden lassen
        Dim station1 As Wetter      ' Auf Klasse beziehen
        Dim station2 As Wetter      ' "

        station1 = New Wetter()      ' Objekt instantiieren
        station2 = New Wetter()

        With station1                ' Eigenschaften Station 1 setzen
            .Temperatur = 35
            .Feuchtigkeit = 75
            .Luftdruck = 800
            .Regenmenge = 10
        End With

        With station2                ' Eigenschaften Station 2 setzen
            .Temperatur = 135        ' Temperatur zu hoch
            .Feuchtigkeit = -5        ' Feuchtigkeit falsch
            .Luftdruck = 1033
            .Regenmenge = -1         ' Regenmenge falsch
        End With
    End Sub
End Class

```

Listing 7.3: Beispiel zur Implementierung von Eigenschaften (Forts.)

```

End With

WriteLine("Messbereiche der Wetterstationen")
WriteLine("Temperatur: - 40 bis + 80 Grad Celsius" & _
    " Luftfeuchtigkeit: 0 bis 100 Prozent")
WriteLine("Druck: 700 hPa bis 1500 hPa " & _
    "Niederschlag: 0 bis 100 Liter")

WriteLine()

WriteLine("Station 1 Temperatur: {0} Feuchtigkeit: {1} " & _
    "Luftdruck: {2} Regenmenge: {3}", _
    Station1.Temperatur, Station1.Feuchtigkeit, _
    Station1.Luftdruck, Station1.Regenmenge)

WriteLine("Station 2 Temperatur: {0} Feuchtigkeit: {1} " & _
    "Luftdruck: {2} Regenmenge: {3}", _
    Station2.Temperatur, Station2.Feuchtigkeit, _
    Station2.Luftdruck, Station2.Regenmenge)

Write("Bitte die Eingabetaste drücken")
ReadLine()
End Sub
End Class

```

Listing 7.3: Beispiel zur Implementierung von Eigenschaften (Forts.)

Hinweis

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_03` auf der Begleit-CD.

Nur-Lesen- und Nur-Schreiben-Eigenschaften

Bei der Deklaration von Eigenschaften können Sie die Schlüsselwörter *ReadOnly* und *WriteOnly* verwenden. Eine als *ReadOnly* vereinbarte Eigenschaft besitzt keine *Set*-Prozedur.

```

ReadOnly Property Feuchtigkeit() As Integer ' Luftfeuchtigkeit
Get
    Return Humid ' Wert lesen
End Get
End Property

```

Sie können dieser Eigenschaft dann keinen Wert zuweisen. Der Versuch einer Wertzuweisung löst einen Übersetzungsfehler aus.

Eine als *WriteOnly* deklarierte Methode erlaubt keinen Lesezugriff auf die betreffende Eigenschaft. Bei dieser Variante entfällt die *Get*-Funktion innerhalb der Klasse. Nachfolgendes Codefragment zeigt die Implementierung einer solchen Eigenschaft:

```
WriteOnly Property Luftdruck() As Integer ' Luftdruck
Set (byVal Wert As Integer) ' Wert schreiben
    Press = Wert ' Speichern
    If Wert < 700 then Press = 700 ' auf Grenzen fixieren
    If Wert > 1500 then Press = 1500
End Set
End Property
```

Diese Variante kommt seltener vor und kann ggf. benutzt werden, um Voreinstellungen für ein Objekt zu definieren (z.B. einen Indexwert).

Hinweis

Im Ordner `\Beisp\Kap07\Beispiel7_03a` der Begleit-CD finden Sie Projektdateien, die eine Klasse *Wetter* mit *ReadOnly*- und *WriteOnly*-Eigenschaften aufweisen. Beim Versuch, das Projekt zu übersetzen, erhalten Sie mehrere Fehlermeldungen, da im zugehörigen Hauptprogramm versucht wird, schreibgeschützte Eigenschaften zu verändern bzw. lesegeschützte Eigenschaften abzufragen.

Verschiedene Zugriffsberechtigungen für Eigenschaften-Accessoren

In früherer Visual-Basic-Versionen mussten die Eigenschaften-Accessoren *Get* und *Set* immer die gleichen Zugriffsberechtigungen (*Public*, *Friend*, *Private*) aufweisen. In Visual Basic 2005 können Sie unterschiedliche Zugriffsberechtigungen verwenden, sofern diese für den *Set*-Accessor restriktiver als für den *Get*-Accessor sind. Dies ist bei folgendem Codefragment zu sehen:

```
Public Property Temperatur() As Integer ' Temperaturwert
    Get
        Return Temp ' Temperatur lesen
    End Get

    Friend Set(ByVal Wert As Integer) ' Temperatur schreiben
        Temp = Wert ' Speichern
        If Wert < -40 Then Temp = -40 ' auf Grenzen fixieren
        If Wert > 80 Then Temp = 80
    End Set
End Property
```

Die Eigenschaft *Temperatur* wurde als *Public* deklariert, d.h., standardmäßig weisen die Accessoren *Get* und *Set* diese Zugriffsberechtigungen auf. Beim *Set*-Accessor wurde aber die Zugriffsberechtigung über *Friend* so eingeschränkt, dass Schreibzugriffe nur noch aus der gleichen Assembly möglich sind.

Hinweis

Sie finden die Projektdateien eines entsprechenden Beispiels im Ordner `\Beisp\Kap07\Beispiel7_03b` der Begleit-CD.

7.2.3 Methoden implementieren

Methoden stellen nichts anderes als Prozeduren oder Funktionen dar, die in einer Klasse implementiert sind. Das obige Beispiel mit der Klasse *Wetter* soll jetzt um zwei Methoden *Clear()* und *SetAll()* erweitert werden. *Clear()* löscht die Eigenschaften des Objekts und *SetAll()* soll die Eigenschaften eines Objekts mit Werten beschreiben. Die Implementierung einer Methode ist sehr einfach, Sie müssen lediglich eine Prozedur oder Funktion innerhalb der Klasse einbauen. Die folgende Prozedur implementiert die *Clear()*-Methode.

```
Sub Clear ()                                ' lösche Eigenschaften
    Temp = 0
    Humid = 0
    Press = 0
    Rain = 0
End Sub
```

Innerhalb der Prozedur werden die internen Variablen zur Zwischenspeicherung der Eigenschaften auf 0 zurückgesetzt. Die Methode lässt sich dann mit folgenden Anweisungen aufrufen:

```
Dim station1 As Wetter = New Wetter()
...
station1.Clear()
```

Die Methode *SetAll()* erwartet die Werte für alle vier Eigenschaften als Parameter und ist folgendermaßen implementiert:

```
Sub SetAll (Temp As Integer, Humid As Integer, _
            Press As Integer, Rain As Integer)
    Temperatur = Temp      ' benutze intern die Eigenschaften
    Feuchtigkeit = Humid
    Luftdruck = Press
    Regenmenge = Rain
End Sub
```


Die der Methode zugrunde liegende Prozedur weist in der aktuellen Implementierung einfach die als Parameter übergebenen Werte den in der Klasse vereinbarten Eigenschaften zu. Dies stellt sicher, dass nur Werte im für die jeweilige Eigenschaft gültigen Bereich gespeichert werden. Mit diesem Ansatz können Sie beliebige Funktionalitäten als Methoden implementieren. Sobald die Methode *SetAll()* implementiert wurde, lässt sie sich mit folgenden Anweisungen nutzen:

```
Dim station1 As Wetter = New Wetter() ' Objekt instantiieren
station1.SetAll(35, -2, 800, 10)      ' Eigenschaften setzen
```

Die zweite Anweisungszeile setzt die Eigenschaften der Objektvariablen *station1* über die *SetAll()*-Methode, wobei die neuen Werte als Parameter übergeben werden.

Hinweis

Alle Details der Implementierung der Klasse *Wetter* sowie deren Verwendung in einer Anwendung finden Sie in einem Projektbeispiel, dessen Dateien im Ordner *\Beisp\Kap07\Beispiel7_04* der Begleit-CD hinterlegt sind.

7.2.4 Anpassen des New-Konstruktors

Die obigen Beispiele verwenden die Klasse *Wetter* und legen für jede Wetterstation eine Instanz an. Zur Unterscheidung der einzelnen Instanzen wird in den vorherigen Beispielen der Objektname (*station1*, *station2* etc.) benutzt. Bei vielen Stationen geht aber schnell der Überblick verloren. Zudem ist es besser, mit Feldern zu arbeiten. Die folgende Anweisung vereinbart ein Feld mit sechs Elementen, von denen jedes eine Objektinstanz der Klasse *Wetter* abbilden kann:

```
Dim station (5) As Wetter
```

Wird dann im Programm *station(0)* als Referenz angegeben, greift die Laufzeitumgebung auf die Member der ersten Objektinstanz zurück. Zur besseren Unterscheidung der einzelnen Instanzen könnte dieser beim Anlegen gleich der Name der Station zugeordnet werden. Das Anlegen einer neuen Instanz erfolgt in Visual Basic 2005 immer über den *New*-Konstruktor, der bisher ohne weitere Parameter aufgerufen wurde:

```
station (0) = New Wetter()
```

Visual Basic 2005 erlaubt es, dem Klassennamen beliebig viele Parameter zu übergeben. Dies könnte dann folgendermaßen aussehen:

```
station (0) = New Wetter("Station Rom")
```

Beim Instantiieren des Objekts wird automatisch ein sogenannter Konstruktor aufgerufen, der die Instanz anlegt. Bei diesem Konstruktor handelt es sich letztendlich um eine Prozedur, die in Visual Basic 2005 den Namen *New* aufweist. Sie können in der Klasse also eine Prozedur *New* mit den gewünschten Parametern vereinbaren. Innerhalb der

Konstruktorprozedur lassen sich dann Anweisungen unterbringen, die einmalig beim Anlegen der Instanz ausgeführt werden.

Hinweis

Konstrukturen sind spezielle Methoden einer Klasse, die eine Initialisierung der Instanz ermöglichen. Die Prozeduren werden beim Instantiieren eines Objekts ausgeführt.

Für unser Beispiel könnte folgender Code innerhalb der Klasse hinterlegt sein:

```
Class Wetter
  Private sName As String = "" ' private Klassevariable
  ...
  ' hier kommt die Konstruktorprozedur
  Sub New (byVal Station As String) ' Setze Stationsname
    sName = Station
  End Sub
  ...

  Function GetName () As String
    Return sName ' Hole Stationsname des aktuellen Objekts
  End Function
End Class
```

Die Prozedur *New* wird automatisch aufgerufen, sobald das Programm den *New*-Konstruktor zum Erzeugen einer Objektinstanz aufruft. Als Parameter wird eine Zeichenkette erwartet, die dann in der internen Variablen *sName* hinterlegt wird. Die Anwendung könnte den Stationsnamen beispielsweise über eine (*ReadOnly*-)Eigenschaft auslesen. Ich habe aber auf die Implementierung einer Eigenschaft *Name* verzichtet. Vielmehr weist die Klasse eine Methode *GetName()* auf, die den Stationsnamen der Objektinstanz zurückliefert. Sie erhalten also gleich ein Beispiel zur Implementierung einer Methode als Funktion. Das nachfolgende Listing zeigt den kompletten Quellcode des Beispiels. Es werden zwei Stationen als Objektinstanzen angelegt, mit Werten versehen und dann auf der Konsoleebene ausgegeben.

```
'*****
' File/Projekt: Beispiel07_05
' Autor: G. Born www.borncity.de
' Erstelle eine Klasse mit Eigenschaften und
' Methoden für eine Wetterstation. Dabei wird
' der New-Konstruktor modifiziert, so dass der
' Stationsname mit angegeben werden kann.
```

Listing 7.4: Beispiel zur Neuimplementierung des New-Konstruktors

```

'*****
Option Strict On
Imports System.Console      ' für WriteLine

Class Wetter
    ' Member der Klasse - Eigenschaften
    ,
    ' Temperatur: - 40 bis + 80 Grad Celsius
    ' Luftfeuchtigkeit: 0 bis 100 Prozent
    ' Druck: 700 hPa bis 1500 hPa
    ' Niederschlag: 0 bis 100 Liter
    Private sName As String = ""
    Private Temp As Integer = 0
    Private Humid As Integer = 0
    Private Press As Integer = 0
    Private Rain As Integer = 0

    Sub New(ByVal Station As String)      ' Setze Stationsname
        sName = Station
    End Sub

    Property Temperatur() As Integer      ' Temperaturwert
        Get
            Return Temp                    ' Temperatur lesen
        End Get

        Set(ByVal Wert As Integer)      ' Temperatur schreiben
            Temp = Wert                    ' Speichern
            If Wert < -40 Then Temp = -40 ' auf Grenzen fixieren
            If Wert > 80 Then Temp = 80
        End Set
    End Property

    Property Feuchtigkeit() As Integer ' Luftfeuchtigkeit
        Get
            Return Humid                    ' Wert lesen
        End Get

        Set(ByVal Wert As Integer)      ' Wert schreiben
            Humid = Wert                    ' Speichern
    End Set
End Class

```

Listing 7.4: Beispiel zur Neuimplementierung des New-Konstruktors (Forts.)

```
        If Wert < 0 Then Humid = 0      ' auf Grenzen fixieren
        If Wert > 100 Then Humid = 100
    End Set
End Property

Property Luftdruck() As Integer      ' Luftdruck
    Get
        Return Press                  ' Wert lesen
    End Get

    Set(ByVal Wert As Integer)      ' Wert schreiben
        Press = Wert                  ' Speichern
        If Wert < 700 Then Press = 700 ' auf Grenzen fixieren
        If Wert > 1500 Then Press = 1500
    End Set
End Property

Property Regenmenge() As Integer    ' Niederschlag
    Get
        Return Rain                   ' Wert lesen
    End Get

    Set(ByVal Wert As Integer)      ' Wert schreiben
        Rain = Wert                  ' Speichern
        If Wert < 0 Then Rain = 0     ' auf Grenzen fixieren
        If Wert > 100 Then Rain = 100
    End Set
End Property

' Hier kommen die Methoden
Sub Clear()                          ' lösche Eigenschaften
    Temp = 0
    Humid = 0
    Press = 0
    Rain = 0
End Sub

' setze alle Eigenschaften
Sub SetAll(ByVal Temp As Integer, ByVal Humid As Integer, _
    ByVal Press As Integer, ByVal Rain As Integer)
```

Listing 7.4: Beispiel zur Neuimplementierung des New-Konstruktors (Forts.)

```

    Temperatur = Temp      ' benutze intern die Eigenschaften
    Feuchtigkeit = Humid
    Luftdruck = Press
    Regenmenge = Rain
End Sub

' Hole Stationsname des aktuellen Objekts
Function GetName() As String
    Return sName
End Function
End Class

Class Test
    Shared Sub Main()      ' das ist das Hauptmodul
        ' Zeige jetzt, wie sich die Daten der Klasse verwenden lassen
        Dim station(5) As Wetter      ' Feld zur Aufnahme der Instanzen
        Dim i As Integer

        station(0) = New Wetter("Station Rom") ' Objekt instantiieren
        station(1) = New Wetter("Station Wien")

        station(0).SetAll(35, -2, 800, 10)      ' Eigenschaften Station1
        station(1).SetAll(15, 2, 930, 0)        ' Eigenschaften Station2

        WriteLine("Messbereiche der Wetterstationen")
        WriteLine("Temperatur: - 40 bis + 80 Grad Celsius" & _
            " Luftfeuchtigkeit: 0 bis 100 Prozent")
        WriteLine("Druck: 700 hPa bis 1500 hPa " & _
            "Niederschlag: 0 bis 100 Liter")

        WriteLine()

        For i = 0 To 1
            With station(i)
                WriteLine("{0} Temperatur: {1} Feuchtigkeit: {2} " & _
                    "Luftdruck: {3} Regenmenge: {4}", _
                    .GetName(), .Temperatur, .Feuchtigkeit, _
                    .Luftdruck, .Regenmenge)
            End With
        Next i
    End Sub
End Class

```

Listing 7.4: Beispiel zur Neuimplementierung des New-Konstruktors (Forts.)

```
Write("Bitte die Eingabetaste drücken")
ReadLine()
End Sub
End Class
```

Listing 7.4: Beispiel zur Neuimplementierung des New-Konstruktors (Forts.)**Hinweis**

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_05` der Begleit-CD.

7.2.5 Die Member einer Klasse überladen

Nachdem das Zuweisen des Stationsnamens beim Instantiieren mittels *New* sehr gut geklappt hat, wäre es doch schön, bei diesem Aufruf auch gleich die Werte mit übergeben zu können. Allerdings soll der *New*-Konstruktor auch die alte Form, bei der nur der Stationsname übergeben wird, unterstützen. Leider gibt es ein Problem, denn die Member einer Klasse (Eigenschaften, Methoden) müssen eindeutig innerhalb der Klasse sein. Dies gilt auch für die Namen der Prozeduren. Die Lösung, um eine dieser Prozeduren mit gleichem Namen, aber unterschiedlichen Parametern aufrufen zu können, liegt in der Verwendung des Mechanismus des Überladens. In der Basisklasse werden die betreffenden Prozeduren oder Eigenschaften einfach mehrfach hintereinander geschrieben. Das nachfolgende Programmfragment demonstriert dies an der *New*-Prozedur, die einmal mit einem Parameter und ein zweites Mal mit fünf Parametern auftritt.

```
Sub New (byVal Station As String) ' Setze Stationsname
    sName = Station
End Sub

' Überladen des New-Konstruktors
Sub New (Station As String, _
        Temp As Integer, Humid As Integer, _
        Press As Integer, Rain As Integer)
    sName = Station          ' Setze Stationsname
    Temperatur = Temp        ' und Werte
    Feuchtigkeit = Humid
    Luftdruck = Press
    Regenmenge = Rain
End Sub
```

Hinweis

Beim Überladen können Sie das Schlüsselwort *Overloads* mit in der Prozedurdeklaration angeben (z.B. *Overloads Sub New (...)*). Dann müssen Sie dies aber bei allen überladenen Membern der Klasse tun.

Der Visual-Basic-Compiler erkennt dies und generiert im rufenden Programm die betreffenden Aufrufe. Dies könnte in der Anwendung dann folgendermaßen aussehen:

```
station(0) = New Wetter("Station Rom") ' Objekt instantiieren
station(1) = New Wetter("Station Wien", 15, 2, 930, 0)
station(0).SetAll (35, -2, 800, 10) ' Eigenschaften Station 1
```

Die erste Zeile erzeugt eine Instanz für die Station »Rom« und weist dieser nur den Stationsnamen zu. Die Eigenschaften des Objekts können in diesem Beispiel später mit der *SetAll()*-Methode explizit gesetzt werden. Bei der zweiten Zeile wird eine weitere Instanz für die Station »Wien« angelegt. Dort werden dem *New*-Konstruktor aber gleich die Initialisierungswerte für die Messwerte übergeben. Das folgende Listing zeigt den kompletten Code des Beispiels, der den *New*-Konstruktor überlädt:

```
'*****
' File/Projekt: Beispiel7_06
' Autor: G. Born www.borncity.de
' Erstellt eine Klasse mit Eigenschaften und
' Methoden für eine Wetterstation. Verwendet eine
' Überladung für den New-Konstruktor.
'*****
Option Strict On
Imports System.Console ' für WriteLine

Class Wetter
' Member der Klasse - Eigenschaften
' Temperatur: - 40 bis + 80 Grad Celsius
' Luftfeuchtigkeit: 0 bis 100 Prozent
' Druck: 700 hPa bis 1500 hPa
' Niederschlag: 0 bis 100 Liter
Private sName As String = ""
Private Temp As Integer = 0
Private Humid As Integer = 0
Private Press As Integer = 0
Private Rain As Integer = 0
```

Listing 7.5: Implementierung mit Überladung des New-Konstruktors

```
Sub New (byVal Station As String) ' Setze Stationsname
    sName = Station
End Sub

' Überladen des New-Konstruktors
Sub New (Station As String, _
        Temp As Integer, Humid As Integer, _
        Press As Integer, Rain As Integer)
    sName = Station          ' Setze Stationsname
    Temperatur = Temp        ' und Werte
    Feuchtigkeit = Humid
    Luftdruck = Press
    Regenmenge = Rain
End Sub

Property Temperatur () As Integer ' Temperaturwert
    Get
        Return Temp          ' Temperatur lesen
    End Get

    Set (byVal Wert As Integer)    ' Temperatur schreiben
        Temp = Wert            ' Speichern
        If Wert < -40 then Temp = -40 ' auf Grenzen fixieren
        If Wert > 80 then Temp = 80
    End Set
End Property

Property Feuchtigkeit() As Integer ' Luftfeuchtigkeit
    Get
        Return Humid          ' Wert lesen
    End Get

    Set (byVal Wert As Integer)    ' Wert schreiben
        Humid = Wert          ' Speichern
        If Wert < 0 then Humid = 0 ' auf Grenzen fixieren
        If Wert > 100 then Humid = 100
    End Set
End Property

Property Luftdruck() As Integer    ' Luftdruck
```

Listing 7.5: Implementierung mit Überladung des New-Konstruktors (Forts.)


```

Get
    Return Press                ' Wert lesen
End Get

Set (byVal Wert As Integer)    ' Wert schreiben
    Press = Wert               ' Speichern
    If Wert < 700 then Press = 700 ' auf Grenzen fixieren
    If Wert > 1500 then Press = 1500
End Set
End Property

Property Regenmenge() As Integer ' Niederschlag
Get
    Return Rain                ' Wert lesen
End Get

Set (byVal Wert As Integer)    ' Wert schreiben
    Rain = Wert               ' Speichern
    If Wert < 0 then Rain = 0   ' auf Grenzen fixieren
    If Wert > 100 then Rain = 100
End Set
End Property

' Hier kommen die Methoden
Sub Clear ()                   ' lösche Eigenschaften
    Temp = 0
    Humid = 0
    Press = 0
    Rain = 0
End Sub

' setze alle Eigenschaften
Sub SetAll (Temp As Integer, Humid As Integer, _
            Press As Integer, Rain As Integer)
    Temperatur = Temp          ' benutze intern die Eigenschaften
    Feuchtigkeit = Humid
    Luftdruck = Press
    Regenmenge = Rain
End Sub

```

Listing 7.5: Implementierung mit Überladung des New-Konstruktors (Forts.)

```

' Hole Stationsname des aktuellen Objekts
Function GetName () As String
    Return sName
End Function
End Class

Class Test
    Shared Sub Main()
        ' das ist das Hauptmodul
    ' Zeige jetzt, wie sich die Daten der Klasse verwenden lassen
    Dim station (5) As Wetter
        ' Feld zur Aufnahme der Instanzen
    Dim i As Integer

    station (0) = New Wetter("Station Rom") ' Objekt instantiieren
    station (1) = New Wetter("Station Wien", 15, 2, 930, 0)

    station(0).SetAll (35, -2, 800, 10) ' Eigenschaften Station 1 setzen

    WriteLine ("Messbereiche der Wetterstationen")
    WriteLine ("Temperatur: - 40 bis + 80 Grad Celsius" & _
        " Luftfeuchtigkeit: 0 bis 100 Prozent")
    WriteLine ("Druck: 700 hPa bis 1500 hPa " & _
        "Niederschlag: 0 bis 100 Liter")
    WriteLine ()

    For i = 0 To 1
        With station(i)
            WriteLine ("{"0} Temperatur: {1} Feuchtigkeit: {2} " & _
                "Luftdruck: {3} Regenmenge: {4}", _
                .GetName(), .Temperatur, .Feuchtigkeit, _
                .Luftdruck, .Regenmenge)
        End With
    Next i
    Write ("Bitte die Eingabetaste drücken")
    ReadLine
End Sub
End Class

```

Listing 7.5: Implementierung mit Überladung des New-Konstruktors (Forts.)

Hinweis

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_06` der Begleit-CD.

7.2.6 Überladung von Operatoren

Im vorhergehenden Beispiel wurde der *New*-Konstruktor überladen. Visual Basic 2005 bietet als Neuerung die Möglichkeit, auch Operatoren zu überladen.

```
Shared Operator <operator>(argumente) As typ
    ' hier kommen die Anweisungen zum Überladen
End Operator
```

Hinter dem Schlüsselwort *Operator* folgt der Operator (z.B. +) und daran schließt sich die Liste mit Argumenten an. Nehmen wir als Beispiel die Operatoren zur Addition, Subtraktion oder Multiplikation von Zahlen. Wenn Sie gezwungen sind, einen Datentyp (z.B. für komplexe Zahlen) einzufügen, lassen sich die Operatoren nicht mehr verwenden. Ähnliches gilt für Konvertierungsoperatoren wie *CType()*, die nur die Standard-Datentypen unterstützen. Die Lösung besteht darin, die Operatoren so zu überladen, dass sie sich auch auf den neuen Datentyp für komplexe Zahlen anwenden lassen. Das folgende Codefragment zeigt den Code einer Klasse, die sowohl den komplexen Datentyp *Complex* definiert als auch Operatoren für Addition, Subtraktion und Multiplikation sowie die Typkonvertierung mit *CType* überlädt:

```
Public Class Complex
    ' Member der Klasse Real und Imag
    Public Real As Double
    Public Imag As Double

    Public Sub New(ByVal RealZweig As Double, _
                  ByVal ImagZweig As Double)
        Real = RealZweig ' weise Real- und Imaginärteil zu
        Imag = ImagZweig
    End Sub

    ' Überladen des +-Operators
    Shared Operator +(ByVal zahl1 As Complex, _
                     ByVal zahl2 As Complex) As Complex
        ' Addiere Real- und Imaginärteile der komplexen Zahlen
        Return New Complex(zahl1.Real + zahl2.Real, _
                           zahl1.Imag + zahl2.Imag)
    End Operator

    ' Überladen des --Operators
    Shared Operator -(ByVal zahl1 As Complex, _
                     ByVal zahl2 As Complex) As Complex
        ' Subtrahiere Real- und Imaginärteile der komplexen Zahlen
        Return New Complex(zahl1.Real - zahl2.Real, _
```

```

        zahl1.Imag - zahl2.Imag)

End Operator

' Überladen des *-Operators
Shared Operator *(ByVal zahl1 As Complex, _
                  ByVal zahl2 As Complex) As Complex
    ' Multipliziere Real- und Imaginärteile der komplexen Zahlen
    '(r1 + i1·i) · (r2 + i2·i) = r1·r2 - i1·i2 + (r1·i2 + i2·r2)·i
    Return New Complex(zahl1.Real * zahl2.Real - _
                       zahl1.Imag * zahl2.Imag, _
                       zahl1.Real * zahl2.Imag + _
                       zahl1.Imag * zahl1.Real)

End Operator

' Zur Ausgabe als String
Public Shared Narrowing Operator CType( _
    ByVal wert As Complex) As String
    Return wert.Real.ToString & "+" & wert.Imag.ToString & "i"

End Operator
End Class

```

Eine komplexe Zahl lässt sich über den *New*-Konstruktor mit einem Real- und Imaginäranteil einer neuen Variablen vom Typ *Complex* zuweisen.

```

Dim Leistung1 As Complex = New Complex(10.0, 20.0)
Dim Leistung2 As Complex = New Complex(15.0, 10.0)
Dim Ergebnis As Complex

```

Die obigen drei Anweisungen definieren drei Variablen mit dem gewünschten Datentyp, wobei die beiden Variablen *Leistung1* und *Leistung2* bereits mit Werten initialisiert werden. Anschließend können Sie Operationen wie +, - oder * auf Variablen dieses Datentyps ausführen.

```

Ergebnis = Leistung1 + Leistung2
Ergebnis = Leistung1 - Leistung2
Ergebnis = Leistung1 * Leistung2

```

Die betreffenden Operatoren wurden so überladen, dass sie die erforderliche mathematische Operation korrekt auf den Real- und Imaginärteil der betreffenden Zahlen anwenden. Für den Additionsoperator sieht der Code zum Überladen folgendermaßen aus:

```

Shared Operator +(ByVal zahl1 As Complex, ByVal zahl2 As Complex) As Complex
    ' Addiere Real- und Imaginärteile der komplexen Zahlen

```

```
Return New Complex(zahl1.Real + zahl2.Real, zahl1.Imag + zahl2.Imag)
End Operator
```

Da die *Double*-Variablen *Real* und *Image* als *Public* innerhalb der Klasse vereinbart sind, können Sie auch im Anwendungsprogramm auf den Real- und Imaginärteil einer Variable vom Typ *Complex* zugreifen. Mit Anweisungen der Art

```
WriteLine("Leistung 1: {0}+{1}i", _
    Leistung1.Real.ToString, Leistung1.Imag.ToString)
```

lassen sich die Ausgangswerte und Ergebnisse im Konsolenfenster ausgeben. Mit *Leistung1.Real.ToString* wird der Realteil der komplexen Zahl gelesen und in eine Zeichenkette konvertiert. Um sich dieses Auftrennen einer komplexen Zahl in Real- und Imaginärteil zu sparen, wurde in der Klasse noch der Konvertierungsoperator *CType* so überladen, dass ein Wert vom Typ *Complex* automatisch korrekt in den Real- und Imaginärteil aufgeteilt als Zeichenkette zurückgegeben wird.

```
Public Shared Narrowing Operator CType(ByVal wert As Complex) _
    As String
    Return wert.Real.ToString & "+" & wert.Imag.ToString & "i"
End Operator
```

Mit diesem überladenen *CType*-Operator lässt sich eine komplexe Zahl direkt in einer *WriteLine*-Anweisung auf der Konsole ausgeben:

```
WriteLine("Ergebnis +: {0}", CType(Ergebnis, String))
```

Der erste Parameter des Konvertierungsoperators *CType* enthält den Wert vom Typ *Complex*, während der zweite Parameter den Datentyp des Ergebnisses angibt. Abbildung 7.3 zeigt das Konsolenfenster mit den Ergebnissen des Programms.

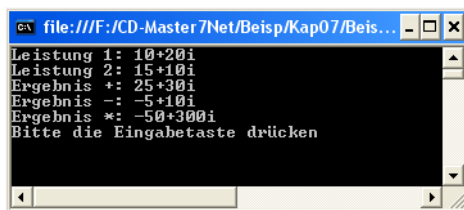


Abbildung 7.3: Anzeige der Ergebnisse

Hinweis

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_07` der Begleit-CD.

7.3 Vererbung in Klassen

Auf den vorhergehenden Seiten dieses Kapitels haben Sie gelernt, wie sich Klassen erstellen und als Objekte instantiieren lassen. Im Grundlagenabschnitt wurde aber erwähnt, dass sich Klassen von Basisklassen ableiten (vererben) lassen. Der folgende Abschnitt geht auf verschiedene Fragen zur Vererbung einer Klasse ein.

7.3.1 Vererbung einer Klasse von einer Basisklasse

Wenn Sie eine Klasse mit verschiedenen Mitgliedern (Eigenschaften und Methoden) besitzen, können Sie diese mit dem *New*-Konstruktor als Objekt instantiieren. Was ist aber, falls die Klasse nicht genau Ihren Anforderungen entspricht. Vielleicht fehlt eine Methode oder Eigenschaft. Oder Sie wollen eine Methode erweitern bzw. ändern? Sofern Sie über den Quellcode der Klasse verfügen, könnten Sie diesen entsprechend anpassen. Fehlt der Quellcode, oder möchten Sie diesen nicht verändern, weil er in anderen Applikationen unverändert benötigt wird? Dann müssen Sie sich für einen anderen Weg entscheiden. Die Lösung liegt in der Möglichkeit zur Vererbung der Member einer bestehenden Klasse an eine neue Klasse. Dies möchte ich jetzt einmal an einem sehr einfachen Beispiel demonstrieren. Hierzu habe ich die Klasse *Wetter* aus den vorherigen Beispielen etwas abgespeckt (es sind nur noch die Eigenschaft *Temperatur* und die Methode *Clear()* implementiert). Gleichzeitig wurde die Klasse in *WetterBasisklasse* umbenannt. Diese Klasse soll als Basis für eine neue Klasse mit dem Namen *Wetter* dienen. Um die neue Klasse von der Basisklasse abzuleiten, sind nur wenige Anweisungen erforderlich.

```
Class Wetter ' abgeleitete Klasse
  Inherits WetterBasisklasse
End Class
```

In der abgeleiteten Klasse ist nur das Schlüsselwort *Inherits* samt dem Namen der Basisklasse anzugeben. Anschließend könnten in der Klasse weitere Member deklariert werden (was hier aber nicht genutzt wird).

In der *Main()*-Prozedur der Anwendung lässt sich nun eine Objektvariable mit den folgenden Anweisungen deklarieren und mit *New* instantiieren.

```
Dim station As Wetter
station = New Wetter()
```

Ich habe an dieser Stelle darauf verzichtet, Stationsnamen in der Klasse zu speichern. Daher werden beim Klassennamen *Wetter* keine Parameter übergeben. Nach diesen Anweisungen verweist die Objektvariable *station* auf die Objektinstanz der Klasse *Wetter*, die ihrerseits alle Methoden und Eigenschaften der Basisklasse geerbt hat. Wir könnten im Hauptprogramm daher die Eigenschaft *Temperatur* mit der Anweisung

```
station.Temperatur = 35
```

setzen. Obwohl in der Klasse *Wetter* keine Eigenschaft *Temperatur* deklariert ist, kennt das Objekt den Eigenschaftennamen, da dieser Name aus der Basisklasse geerbt wurde.

Das folgende Listing zeigt den gesamten Quellcode des Beispiels. Ich habe die Objektvariable zur Aufnahme der Stationsdaten als Feld belassen. Das Programm initialisiert zwei Stationsinstanzen, weist diesen Temperaturwerte zu, zeigt diese Werte auf Konsolebene an, ruft die Methode *Clear()* zum Löschen der Temperaturwerte auf und zeigt anschließend das Ergebnis erneut an.

```

'*****
' File/Projekt: Beispiel7_08
' Autor: G. Born www.borncity.de
' Leitet eine Klasse von einer Basisklasse ab.
'*****
Option Strict On
Imports System.Console      ' für WriteLine

Class WetterBasisKlasse
' Member der Klasse
' Eigenschaft: Temperatur, Methode: Clear
'
' Temperatur: - 40 bis + 80 Grad Celsius
Private Temp As Integer = 0

Property Temperatur () As Integer ' Temperaturwert
Get
    Return Temp                  ' Temperatur lesen
End Get

Set (byVal Wert As Integer)      ' Temperatur schreiben
    Temp = Wert                  ' Speichern
    If Wert < -40 then Temp = -40 ' auf Grenzen fixieren
    If Wert > 80 then Temp = 80
End Set
End Property

' Hier kommen die Methoden
Sub Clear ()                      ' lösche Eigenschaften
    Temperatur = 0                ' benötigt Temperaturwert
End Sub
End Class

Class Wetter                      ' abgeleitete Klasse
Inherits WetterBasisKlasse

```

Listing 7.6: Beispiel mit abgeleiteter Klasse

```

End Class

Class Test
    Shared Sub Main()                ' das ist das Hauptmodul
    ' Zeige jetzt, wie sich die Daten der Klasse verwenden lassen
    Dim station (5) As Wetter      ' Feld zur Aufnahme der Instanzen
    Dim i As Integer

    station (0) = New Wetter()      ' Objekt instantiieren
    station (1) = New Wetter()

    station(0).Temperatur = 35        ' Eigenschaft Station 1 setzen
    station(1).Temperatur = 25        ' Eigenschaft Station 2 setzen

    For i = 0 To 1
        With station(i)
            WriteLine ("{0} Temperatur: {1} ", _
                "Station " & i.ToString, .Temperatur)
            .Clear()                  ' lösche Wert
        End With
    Next i

    WriteLine()

    For i = 0 To 1                    ' zweiter Durchlauf
        With station(i)
            WriteLine ("{0} Temperatur: {1} ", _
                "Station " & i.ToString, .Temperatur)
        End With
    Next i

    Write ("Bitte die Eingabetaste drücken")
    ReadLine
    End Sub
End Class

```

Listing 7.6: Beispiel mit abgeleiteter Klasse (Forts.)

Hinweis

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_08` der Begleit-CD.

7.3.2 Vererbung mit Konstruktoren in der Basisklasse

Das obige Beispiel ist ziemlich trivial. Es wird nichts an der Funktionalität der Basisklasse verändert und die abgeleitete Klasse enthält auch keine eigenen Konstruktoren. Bei der Auswahl des Beispiels war es mein Ziel, Ihnen das Prinzip der Vererbung, die dafür sorgt, dass eine abgeleitete Klasse die Funktionalität der Basisklasse erhält, zu zeigen.

Sie könnten jetzt auf die Idee kommen, das Beispiel aus dem Projektordner *Beispiel7_06* auf die gleiche Weise zu modifizieren, indem Sie die Klasse *Wetter* in *WetterBasisKlasse* umbenennen. Dann ließe sich eine neue Klasse *Wetter* definieren, die die Eigenschaften der Basisklasse erbt. Leider gibt es ein Problem, die abgeleitete Klasse *Wetter* weist keinen zugreifbaren *New*-Konstruktor auf, da sich die Konstruktoren einer Klasse nicht vererben lassen! Sobald die Basisklasse einen *New*-Konstruktor aufweist, muss dies in der abgeleiteten Klasse berücksichtigt werden.

Für den nächsten Schritt wird der Beispielcode der Basisklasse um zwei *New*-Konstruktoren erweitert. Diese erlauben wieder den Stationsnamen sowie die Temperatur beim Instanzieren anzugeben. Die Basisklasse *WetterBasisKlasse* sieht dann folgendermaßen aus:

```
Class WetterBasisKlasse
' Member der Klasse
' Eigenschaft: Temperatur, Methode: Clear()
'
' Temperatur: - 40 bis + 80 Grad Celsius
Private sName As String = ""
Private Temp As Integer = 0

Sub New (byVal Station As String) ' Setze Stationsname
    sName = Station
End Sub

' Überladen des New-Konstruktors
Sub New (Station As String, Temp As Integer)
    sName = Station          ' Setze Stationsname
    Temperatur = Temp        ' und Werte
End Sub

... ' hier folgen die Eigenschaften und Methoden
End Class
```

Im Grunde hat sich nichts zu den ersten Beispielen dieses Kapitels, in denen der *New*-Konstruktor genutzt wurde, geändert. Wie sieht es jetzt bei der abgeleiteten Klasse aus? Jetzt müssen wir aber noch dafür sorgen, dass die abgeleitete Klasse *Wetter* die von der Prozedur *Main* benutzten *New*-Konstruktoren unterstützt. Dies bedeutet, in der Klassendefinition muss ein *New*-Konstruktor auftreten.

```
Sub New () ' Setze Stationsname
...
End Sub
```

Sie könnten natürlich den obigen Code in die Basisklasse und in der abgeleiteten Klasse einfügen, um zumindest einen leeren Konstruktor zu haben.

```
Class WetterBasisKlasse
    Private sName As String = ""
    Private Temp As Integer = 0

    Sub New () ' leerer Konstruktor

    End Sub

    Sub New (byVal Station As String) ' Setze Stationsname
        sName = Station
    End Sub
    ...
End Class

Class Wetter ' abgeleitete Klasse
    Inherits WetterBasisKlasse
    Sub New () ' leerer Konstruktor

    End Sub
End Class
```

Im Hauptprogramm, welches die abgeleitete Klasse benutzt, wäre dann zumindest der folgende Aufruf zulässig:

```
Dim station As Wetter = New Wetter()
```

Die Anweisung legt eine neue Instanz der Klasse *Wetter* an und weist die Referenz auf das Objekt der Variablen *station* zu. Leider wäre es in diesem Fall nicht mehr möglich, den *New*-Konstruktor mit dem Stationsnamen als Parameter (z.B. *New Wetter("Station 1")*) aufzurufen. Der entsprechende Konstruktor der Basisklasse wird ja nicht an die Klasse *Wetter* vererbt, lässt sich also auch nicht von der Anwendung nutzen.

Das Problem lässt sich aber recht einfach lösen, indem die Konstruktoren der Basisklasse in geeigneter Form in die abgeleitete Klasse übernommen werden. Das folgende Codefragment zeigt die Erweiterung der abgeleiteten Klasse *Wetter* um die betreffenden Konstruktoren:

```

Class Wetter                                ' abgeleitete Klasse
  Inherits WetterBasisKlasse

  ' rufe die Konstruktoren der Basisklasse
  Sub New (byVal Station As String) ' Setze Stationsname
    MyBase.New(Station)
  End Sub

  Sub New (Station As String, Temp As Integer)
    MyBase.New(Station)
    Me.Temperatur = Temp           ' 2. Parameter verarbeiten
  End Sub
End Class

```

Im ersten Schritt wurden einfach die Deklarationen des Konstruktors aus der Basisklasse in die abgeleitete Klasse kopiert. Damit stimmen die Definitionen der Parameter für beide Klassen automatisch überein (wobei dies aber nicht zwingend sein muss). Der springende Punkt steckt im Innenleben der betreffenden Konstruktoren. Im Konstruktor der Klasse *Wetter* muss als Erstes der Konstruktor der Basisklasse aufgerufen werden. Hier kommt eine Neuerung ins Spiel: Beim Aufruf muss angegeben werden, dass der Konstruktor der Basisklasse gemeint ist. Dies erfolgt über das Schlüsselwort *MyBase*. Die Anweisung

```
MyBase.New(Station)
```

ruft den Konstruktor der Basisklasse auf, wobei die Variante benutzt wird, die einen Parameter unterstützt. Beim zweiten Konstruktor, der die beiden Parameter mit dem Stationsnamen und dem Temperaturwert unterstützt, ließe sich folgende Anweisung verwenden:

```
MyBase.New(Station, Temp)
```

Diese Anweisung ruft den betreffenden Konstruktor der Basisklasse auf. Ich habe mich in dem oben gezeigten Codeabschnitt aber für einen anderen Ansatz entschieden. Der Konstruktor der Klasse *Wetter* instantiiert die Basisklasse über *MyBase.New(Station)*. Anschließend wird der als zweiter Parameter übergebene Temperaturwert explizit der Eigenschaft *Temperatur* der Instanz übergeben. Dies gibt mir die Möglichkeit, auf eine zweite Neuerung in Form des Schlüsselworts *Me* einzugehen. Beim Zugriff auf die Eigenschaft *Temperatur* gibt es noch ein Problem. Die Angabe der Basisklasse über *MyBase* oder *WetterBasisKlasse* hilft uns hier nicht weiter. Zur Laufzeit beziehen wir uns ja nicht auf eine Klasse, sondern auf die Instanz, also auf ein Objekt. Daher muss ich in der abgeleiteten Klasse, die ja quasi den Bauplan für das Objekt darstellt, einen Mechanismus haben, der einen Bezug auf das aktuelle Objekt erlaubt. So etwas wie *station* scheidet aus, da der Name der Objektvariablen nur im Programm, welches die Instanz erzeugt, bekannt ist. Die Lösung steckt im Schlüsselwort *Me*, welches sich vereinbarungsgemäß immer auf die aktuelle Objektinstanz bezieht. Die Anweisung

```
Me.Temperatur = Temp
```

innerhalb des Konstruktors weist also den Wert des Parameters *Temp* an die Eigenschaft *Temperatur* der aktuellen Instanz des Objekts zu. Genau das, was wir benötigt haben.

Das nachfolgende Listing zeigt den kompletten Quellcode des Beispiels. Sie sehen die Anweisungen zur Implementierung der Basisklasse sowie der abgeleiteten Klasse *Wetter*. Die Prozedur *Main* nutzt anschließend die Klasse *Wetter*, um über deren Instanzen die Daten zweier Wetterstationen zu speichern, abzurufen und auf der Konsoleebene anzuzeigen.

```
'*****  
' File/Projekt: Beispiel7_08a  
' Autor: G. Born www.borncity.de  
' Leitet eine Klasse von einer Basisklasse ab.  
' Berücksichtigt die Konstruktoren der Basisklasse  
'*****  
Option Strict On  
Imports System.Console      ' für WriteLine  
  
Class WetterBasisKlasse  
' Member der Klasse  
' Eigenschaft: Temperatur, Methode: Clear()  
'  
' Temperatur: - 40 bis + 80 Grad Celsius  
Private sName As String = ""  
Private Temp As Integer = 0  
  
Sub New (byVal Station As String) ' Setze Stationsname  
    sName = Station  
End Sub  
  
' Überladen des New-Konstruktors  
Sub New (Station As String, Temp As Integer)  
    sName = Station          ' Setze Stationsname  
    Temperatur = Temp        ' und Werte  
End Sub  
  
Property Temperatur () As Integer ' Temperaturwert  
Get  
    Return Temp              ' Temperatur lesen  
End Get
```

Listing 7.7: Beispiel mit abgeleiteter Klasse und Konstruktoren

```
Set (byVal Wert As Integer)      ' Temperatur schreiben
    Temp = Wert                  ' Speichern
    If Wert < -40 then Temp = -40 ' auf Grenzen fixieren
    If Wert > 80 then Temp = 80
End Set
End Property

' Hier kommen die Methoden
Sub Clear ()                     ' lösche Eigenschaften
    Temperatur = 0               ' benötigt Temperaturwert
End Sub

' Hole Stationsname des aktuellen Objekts
Function GetName () As String
    Return sName
End Function
End Class

Class Wetter                     ' abgeleitete Klasse
    Inherits WetterBasisKlasse

' rufe die Konstruktoren der Basisklasse
Sub New (byVal Station As String) ' Setze Stationsname
    MyBase.New(Station)
End Sub

Sub New (Station As String, Temp As Integer)
    MyBase.New(Station)
    Me.Temperatur = Temp          ' 2. Parameter verarbeiten
End Sub
End Class

Class Test
    Shared Sub Main()             ' das ist das Hauptmodul
        ' Zeige jetzt, wie sich die Daten der Klasse verwenden lassen
        Dim station (5) As Wetter ' Feld zur Aufnahme der Instanzen
        Dim i As Integer

        station (0) = New Wetter("Station Rom") ' Objekt instantiieren
```

Listing 7.7: Beispiel mit abgeleiteter Klasse und Konstruktoren (Forts.)

```
station (1) = New Wetter("Station Wien", 28)

station(0).Temperatur = 32           ' Eigenschaft Station 1 setzen

For i = 0 To 1
    With station(i)
        WriteLine ("{0} Temperatur: {1} ", _
                    .GetName(), .Temperatur)
        .Clear()                     ' lösche Wert
    End With
Next i

WriteLine()

For i = 0 To 1                       ' zweiter Durchlauf
    With station(i)
        WriteLine ("{0} Temperatur: {1} ", _
                    .GetName(), .Temperatur)
    End With
Next i

Write ("Bitte die Eingabetaste drücken")
ReadLine
End Sub
End Class
```

Listing 7.7: Beispiel mit abgeleiteter Klasse und Konstruktoren (Forts.)

Hinweis

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_08a` der Begleit-CD. Auf diese Weise können Sie die Konstruktoren der Basisklasse in der abgeleiteten Klasse übernehmen. Lassen Sie einen *New*-Konstruktor in der Basisklasse weg, steht dieser für die Objektinstanziierung in der Anwendung nicht zur Verfügung. Auf der anderen Seite können Sie in der abgeleiteten Klasse neue Konstruktoren hinzufügen, die sich dann nutzen lassen. Im Projektordner finden Sie die nicht im Projekt eingebundenen Dateien *Beisp07_08a.vb* und *Beisp07_08c.vb*. Die erste Datei benutzt leere *New()*-Konstruktoren bei der Ableitung der Klasse, während in der zweiten Datei eine dritte Variante des *New*-Konstruktors hinterlegt wurde, die drei Parameter beim Aufruf unterstützt.

7.3.3 Vererbung mit angepassten/erweiterten Mitgliedern

Das Schöne bei der objektorientierten Programmierung ist, dass Sie die vererbte Klasse quasi nach Belieben anpassen können. Neben den ererbten Mitgliedern können Sie weitere Eigenschaften oder Methoden (Member) hinzufügen. Es besteht sogar die Möglichkeit, einen ererbten Member durch Überschreiben anzupassen. Dies möchte ich jetzt an einem modifizierten Beispiel schrittweise diskutieren. Es soll wieder die Basisklasse *WetterPrimitiv* und eine abgeleitete Klasse *Wetter* verwendet werden. Gegenüber den vorherigen Beispielen habe ich die Basisklassen aber etwas in der Funktionalität reduziert. Die Klasse besitzt nur noch die Member *Temperatur* und *Feuchtigkeit*, die als Eigenschaften implementiert sind. Zudem lässt sich der Instanz über den *New*-Konstruktor der Stationsname zuweisen und mittels der *GetName()*-Methode abfragen. Der nachfolgende Quellcode zeigt die Implementierung der betreffenden Basisklasse:

```
Class WetterPrimitiv ' Die Basisklasse
' Member der Klasse
' Klassenname als String
' Temperatur: - 40 bis + 80 Grad Celsius
' Luftfeuchtigkeit: 0 bis 100 Prozent
Private sName As String = ""
Private Temp As Integer = 0
Private Humid As Integer = 0

' Implementiere einen neuen Konstruktor
Sub New (byVal Station As String) ' Setze Stationsname
    sName = Station
End Sub

' Hier kommen die Eigenschaften der Basisklasse
' Member Temperatur darf überschrieben werden
Overridable Property Temperatur () As Integer ' Temperaturwert
Get
    Return Temp ' Temperatur lesen
End Get

Set (byVal Wert As Integer) ' Temperatur schreiben
    Temp = Wert ' Speichern
    If Wert < -40 then Temp = -40 ' auf Grenzen fixieren
    If Wert > 80 then Temp = 80
End Set
End Property

' Member Feuchtigkeit nicht für das Überschreiben vorsehen
Property Feuchtigkeit() As Integer ' Luftfeuchtigkeit
```

```

Get
    Return Humid                ' Wert lesen
End Get

Set (byVal Wert As Integer)    ' Wert schreiben
    Humid = Wert                ' Speichern
    If Wert < 0 then Humid = 0   ' auf Grenzen fixieren
    If Wert > 100 then Humid = 100
End Set
End Property

' Hier kommen die Methoden der Basisklasse
' Hole Stationsname des aktuellen Objekts
Function GetName () As String
    Return sName
End Function
End Class

```

Mit dieser Klasse ließen sich also nur Wetterstationen abbilden, die Temperatur und Feuchtigkeit messen. Aus den vorhergehenden Beispielen wissen Sie aber, dass die meisten Wetterstationen auch die Niederschlagsmenge und den Luftdruck abbilden können. Zudem soll in dem neuen Projekt eine Wetterstation in der Antarktis hinzukommen. Der ursprünglich vorgesehene Messbereich von –40 bis +80 Grad erscheint daher zu gering und soll auf –80 bis +100 Grad ausgeweitet werden. Natürlich könnte die Basisklasse um die betreffenden Member erweitert werden bzw. die Eigenschaft *Temperatur* ließe sich in den Grenzwerten anpassen.

Die objektorientierte Programmierung erlaubt aber einen eleganteren Ansatz. Es wird einfach eine neue Klasse *Wetter* von der Basisklasse abgeleitet. Diese kann dann bereits den Stationsnamen, die Temperatur und die Feuchtigkeit abbilden. Bezüglich der Temperatur ist nur sicherzustellen, dass diese die neuen Grenzwerte zulässt. In der abgeleiteten Klasse sind dann die fehlenden Member für Niederschlagsmenge und Luftdruck hinzuzufügen. Zudem könnten weitere Methoden zum Setzen und Löschen der Werte hinzukommen.

Viele Teile des Codes der abgeleiteten Klasse kennen Sie bereits aus den vorhergehenden Beispielen. Um eine neue Eigenschaft oder Methode in der Klasse *Wetter* aufzunehmen, fügen Sie den betreffenden Code ein. Das folgende Codefragment implementiert die neue Eigenschaft *Luftdruck*.

```

Private Press As Integer = 0    ' neu anlegen
' Member Luftdruck neu zur Klasse hinzufügen
Property Luftdruck() As Integer ' Luftdruck
Get
    Return Press                ' Wert lesen

```



```

End Get

Set (byVal Wert As Integer)      ' Wert schreiben
    Press = Wert                  ' Speichern
    If Wert < 700 then Press = 700 ' auf Grenzen fixieren
    If Wert > 1500 then Press = 1500
End Set
End Property

```

Der Code lässt sich aus den vorherigen Beispielen direkt übernehmen. Einziger Unterschied, die Anweisungen befinden sich nicht mehr in der Basisklasse, sondern in der abgeleiteten Klasse *Wetter*. Vielleicht möchten Sie einen Konstruktor *New* innerhalb der neuen Klasse nutzen, der in Anlehnung an die vorherigen Beispiele sowohl den Stationsnamen als auch die vier Messwerte als Parameter übernimmt. Der Code könnte dann so aussehen:

```

' Einführen eines neuen New-Konstruktors mit 5 Parametern
Sub New (Station As String, _
        Temp As Integer, Humid As Integer, _
        Press As Integer, Rain As Integer)
' Instantiiere Basisklasse mit dem Stationsnamen
MyClass.New(Station)      ' eigenen Konstruktor aufrufen
' Jetzt müssen wir die Parameter in Eigenschaften übertragen
Me.Temperatur = Temp      ' Member dieser Klasse
MyBase.Feuchtigkeit = Humid ' Member der Basisklasse
Luftdruck = Press         ' lokale Eigenschaft
Regenmenge = Rain        ' lokale Eigenschaft
End Sub

```

Da die Basisklasse nur einen *New*-Konstruktor mit dem Stationsnamen als Parameter aufweist, müssen wir diesen als Erstes innerhalb des zu implementierenden Konstruktors aufrufen. Dies kann über die Anweisung

```
MyBase.New(Station)
```

erfolgen. Ich habe an dieser Stelle zur Demonstration aber ein neues Schlüsselwort *MyClass* benutzt:

```
MyClass.New(Station)
```

Dieses Schlüsselwort signalisiert dem Visual-Basic-2005-Compiler, dass sich die Referenz auf die aktuelle Methode bezieht. Hier veranlasst die Anweisung den Aufruf des *New*-Konstruktors der Klasse *Wetter*. Diese leitet den Aufruf dann an den entsprechenden Konstruktor der Basisklasse weiter.

Hinweis

In der Praxis sollten Sie aus Effizienzgründen direkt den Konstruktor der Basisklasse aufrufen (der Compiler kann Verweise auf die Basisklasse generieren).

Die restlichen Anweisungen innerhalb des Konstruktors speichern die übergebenen Parameter als Eigenschaften der aktuellen Instanz. Über das Schlüsselwort *Me* legen Sie bei der Temperatur fest, dass es sich um die lokale Eigenschaft der Klasse handelt, da dort der Temperaturbereich erweitert wurde. Das Schlüsselwort *MyBase* bewirkt, dass sich die Eigenschaft *Feuchtigkeit* auf die Basisklasse bezieht. Geben Sie nichts vor dem Eigenschaftennamen an, wird automatisch die lokale Eigenschaft der Klasse benutzt.

Bleibt noch die Frage, wie sich die Methode *Temperatur* der Basisklasse so erweitern lässt, dass diese in der Klasse *Wetter* einen erweiterten Temperaturbereich von -80 Grad bis +100 Grad zulässt? Geben Sie nichts in der Klasse an, wird automatisch die gleichnamige Methode der Basisklasse übernommen. Sie könnten die Anweisungen zur Implementierung der Eigenschaft aus der Basisklasse in die abgeleitete Klasse kopieren und dann die Grenzwerte anpassen. Beim Übersetzen wird der Compiler aber eine Fehlermeldung liefern, da der gleiche Member in beiden Klassen auftritt. Sie müssen dem Visual-Basic-2005-Compiler mitteilen, dass der Member überschrieben wird. Dies erfordert einen Eingriff an zwei Stellen. In der Basisklasse müssen Sie den Member als überschreibbar markieren. Hierzu genügt es, das Schlüsselwort *Overridable* in der Deklaration der Eigenschaft aufzunehmen.

```
Overridable Property Temperatur () As Integer ' Temperaturwert
```

Wenn Sie dann in der abgeleiteten Klasse den Member mit dem Schlüsselwort *Overrides* versehen, akzeptiert der Compiler dies. Die Methode oder die Eigenschaft wird dann in der abgeleiteten Klasse überschrieben. Hier sehen Sie den Code zur Implementierung der abgeleiteten Eigenschaft *Temperatur* als Member der Klasse *Wetter*:

```
' Überschreibe Member Temperatur (neue Grenzen -80 bis 100 Grad)
Overrides Property Temperatur () As Integer ' Temperaturwert
Get
    Return Temp ' Temperatur lesen
End Get

Set (byVal Wert As Integer) ' Temperatur schreiben
    Temp = Wert ' Speichern
    If Wert < -80 then Temp = -80 ' auf Grenzen fixieren
    If Wert > 100 then Temp = 100
End Set
End Property
```

Da innerhalb der Klasse *Wetter* eine private Variable *Temp* vereinbart wurde, lässt sich direkt auf den internen Wert zur Speicherung zugreifen. Auf diese Weise können Sie die Member einer Klasse aus einer anderen Klasse ableiten, anpassen und ergänzen.

Hinweis

Member einer Klasse lassen sich über das Schlüsselwort *Protected* schützen. Dann sind die Mitglieder nur in abgeleiteten Klassen zugreifbar. Werden zwei Variable gleichen Namens innerhalb des Kontexts benutzt, sorgt das Schlüsselwort *Shadows* vor der zweiten Variablendeklaration dafür, dass diese den übergeordneten Wert abschattet (d.h., alle Referenzen beziehen sich auf die mit *Shadows* deklarierte Variable). Dies wird beispielsweise bei der Variablen *Temp* in der Klasse *Wetter* genutzt.

Das folgende Listing zeigt die komplette Implementierung des Beispiels mit der Basis-Klasse *WetterPrimitiv* und der abgeleiteten, aber erweiterten Klasse *Wetter*. Die Testwerte in der Prozedur *Test* zeigen, dass die neue Klasse auch Temperaturen unterhalb -40 Grad und oberhalb von +80 Grad in einer Instanz speichern kann.

```

'*****
' File/Projekt: Beispiel07_09
' Autor: G. Born www.borncity.de
' Demonstriert das Vererben einer Klasse. Dabei sind
' Member zu übernehmen, anzupassen und neu zu erstellen.
' Temperatur Grenzwerte neu setzen -80 bis 100
' Feuchtigkeit aus Basisklasse übernehmen
' Luftdruck neu implementieren
' Regenmenge neu implementieren
' New-Konstruktoren implementieren
'*****
Option Strict On
Imports System.Console ' für WriteLine

Class WetterPrimitiv ' Die Basisklasse
' Member der Klasse
' Klassenname als String
' Temperatur: - 40 bis + 80 Grad Celsius
' Luftfeuchtigkeit: 0 bis 100 Prozent
Private sName As String = ""
Private Temp As Integer = 0
Private Humid As Integer = 0

' Implementiere einen neuen Konstruktor
Sub New (byVal Station As String) ' Setze Stationsname

```

Listing 7.8: Beispiel mit Erweiterung der abgeleiteten Klasse

```

    sName = Station
End Sub

' Hier kommen die Eigenschaften der Basisklasse
' Member Temperatur darf überschrieben werden
Overridable Property Temperatur () As Integer ' Temperaturwert
Get
    Return Temp ' Temperatur lesen
End Get

Set (byVal Wert As Integer) ' Temperatur schreiben
    Temp = Wert ' Speichern
    If Wert < -40 then Temp = -40 ' auf Grenzen fixieren
    If Wert > 80 then Temp = 80
End Set
End Property

' Member Feuchtigkeit nicht für das Überschreiben vorsehen
Property Feuchtigkeit() As Integer ' Luftfeuchtigkeit
Get
    Return Humid ' Wert lesen
End Get

Set (byVal Wert As Integer) ' Wert schreiben
    Humid = Wert ' Speichern
    If Wert < 0 then Humid = 0 ' auf Grenzen fixieren
    If Wert > 100 then Humid = 100
End Set
End Property

' Hier kommen die Methoden der Basisklasse
' Hole Stationsname des aktuellen Objekts
Function GetName () As String
    Return sName
End Function
End Class

' #####
' Leite die neue Klasse aus der
' Basisklasse WetterPrimitiv ab.

```

Listing 7.8: Beispiel mit Erweiterung der abgeleiteten Klasse (Forts.)

```
' #####
Class Wetter
' erweitere die neue Klasse um Eigenschaften und Methoden
'
' Stationsname: aus Basisklasse mit Shadows überschreiben
' Temperatur: - 80 bis + 100 Grad Celsius
' (modifiziert aus Basisklasse)
' Luftfeuchtigkeit: 1:1 aus Basisklasse übernehmen
' Druck: 700 hPa bis 1500 hPa neu hinzufügen
' Niederschlag: 0 bis 100 Liter neu hinzufügen
' Konstruktoren: New (Station) und New (Station, P1, P2, P3, P4)
' Methoden aus Basisklasse direkt übernehmen.

Inherits WetterPrimitiv          ' aus Basisklasse ableiten

' Lokale Variablen festlegen
Shadows Temp As Integer = 0      ' neu, aus Basisklasse schattieren
Private Press As Integer = 0      ' neu anlegen
Private Rain As Integer = 0      ' neu anlegen

' Überladen des New-Konstruktors der Basisklasse
Sub New (byVal Station As String) ' Setze Stationsname
    MyBase.New (Station)
End Sub

' Einführen eines neuen New-Konstruktors mit 5 Parametern
Sub New (Station As String, _
        Temp As Integer, Humid As Integer, _
        Press As Integer, Rain As Integer)
' Instanziiere Basisklasse mit dem Stationsnamen
    MyClass.New(Station)          ' eigenen Konstruktor aufrufen
' Jetzt müssen wir die Parameter in Eigenschaften übertragen
    Me.Temperatur = Temp          ' Member dieser Klasse
    MyBase.Feuchtigkeit = Humid   ' Member der Basisklasse
    Luftdruck = Press             ' lokale Eigenschaft
    Regenmenge = Rain            ' lokale Eigenschaft
End Sub

' Überschreibe Member Temperatur (neue Grenzen -80 bis 100 Grad)
Overrides Property Temperatur () As Integer ' Temperaturwert
```

Listing 7.8: Beispiel mit Erweiterung der abgeleiteten Klasse (Forts.)

```

Get
    Return Temp                ' Temperatur lesen
End Get

Set (byVal Wert As Integer)    ' Temperatur schreiben
    Temp = Wert                ' Speichern
    If Wert < -80 then Temp = -80 ' auf Grenzen fixieren
    If Wert > 100 then Temp = 100
End Set
End Property

' Member Luftdruck neu zur Klasse hinzufügen
Property Luftdruck() As Integer ' Luftdruck
Get
    Return Press                ' Wert lesen
End Get

Set (byVal Wert As Integer)    ' Wert schreiben
    Press = Wert                ' Speichern
    If Wert < 700 then Press = 700 ' auf Grenzen fixieren
    If Wert > 1500 then Press = 1500
End Set
End Property

' Member Regenmenge neu zur Klasse hinzufügen
Property Regenmenge() As Integer ' Niederschlag
Get
    Return Rain                ' Wert lesen
End Get

Set (byVal Wert As Integer)    ' Wert schreiben
    Rain = Wert                ' Speichern
    If Wert < 0 then Rain = 0    ' auf Grenzen fixieren
    If Wert > 100 then Rain = 100
End Set
End Property

' Hier kommen die neuen Methoden
Sub Clear ()                    ' lösche Eigenschaften
    Me.Temperatur = 0           ' Member dieser Klasse

```

Listing 7.8: Beispiel mit Erweiterung der abgeleiteten Klasse (Forts.)

```

MyClass.Feuchtigkeit = 0          ' Member der Basisklasse
Press = 0                        ' Member dieser Klasse
Rain = 0                         ' "
End Sub

' neue Methode, setzt alle Eigenschaften der aktuellen Klasse
Sub SetAll (Temp As Integer, Humid As Integer, _
            Press As Integer, Rain As Integer)
    Temperatur = Temp            ' Member dieser Klasse
    MyBase.Feuchtigkeit = Humid  ' Member der Basisklasse
    Luftdruck = Press            ' Member dieser Klasse
    Regenmenge = Rain           ' "
End Sub
End Class

Class Test
Shared Sub Main()                ' das ist das Hauptmodul
' Zeige jetzt, wie sich die Daten der Klasse verwenden lassen
    Dim station (5) As Wetter    ' Feld zur Aufnahme der Instanzen
    Dim i As Integer

    station (0) = New Wetter("Station Scott") ' Objekt instantiieren
    station (1) = New Wetter("Station Tunis", 115, 30, 1030, 0)
    station (2) = New Wetter("Station Wien", 15, 70, 930, 3)

' Eigenschaften Station 1 setzen
    station(0).SetAll (-55, -2, 800, 0)

    WriteLine ("Messbereiche der Wetterstationen")
    WriteLine ("Temperatur: - 80 bis + 100 Grad Celsius" & _
                " Luftfeuchtigkeit: 0 bis 100 Prozent")
    WriteLine ("Druck: 700 hPa bis 1500 hPa " & _
                "Niederschlag: 0 bis 100 Liter")

    WriteLine ()

    For i = 0 To 2
        With station(i)
            WriteLine ("{0} Temperatur: {1} Feuchtigkeit: {2} " & _
                        "Luftdruck: {3} Regenmenge: {4}", _

```

Listing 7.8: Beispiel mit Erweiterung der abgeleiteten Klasse (Forts.)

```

        .GetName(), .Temperatur, .Feuchtigkeit, _
        .Luftdruck, .Regenmenge)
    End With
Next i

Write ("Bitte die Eingabetaste drücken")
ReadLine
End Sub
End Class

```

Listing 7.8: Beispiel mit Erweiterung der abgeleiteten Klasse (Forts.)

Hinweis

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_09` der Begleit-CD.

7.3.4 Klasse vor Vererbung und Methode vor dem Überschreiben schützen

Auf den vorhergehenden Seiten haben Sie erfahren, wie sich Klassen vererben und Methoden ggf. überschreiben lassen. Möchten Sie verhindern, dass Dritte die von Ihnen erstellten Klassen als Basisklassen verwenden, lassen sich diese mit dem Schlüsselwort *NotInheritable* zur weiteren Vererbung sperren. Die folgende Klasse *Grenzen* fixiert die Grenzwerte für die Temperatur und ist als nicht vererbbar gekennzeichnet:

```

NotInheritable Class Grenzen
' Temperaturgrenzen: - 40 bis + 80 Grad Celsius
Friend Shared TMin As Integer = -40
Friend Shared TMax As Integer = 80
End Class

```

Der Ansatz in der folgenden Codesequenz, die Klasse in einer neuen Klasse zu erben und ggf. zu erweitern, wird daher fehlschlagen:

```

Class GrenzenNeu ' Erweiterung der Klasse Grenzen
Inherits Grenzen ' ### Löst ggf. Fehler aus!!!!
Shared HumidMin As Integer = 0
Shared HumidMax As Integer = 100
End Class

```

Um die Klasse abzuleiten, muss das Schlüsselwort *NotInheritable* in der Basisklasse *Grenzen* entfernt werden.

Beim Vererben von Klassenmitgliedern lassen sich diese in der abgeleiteten Klasse überschreiben. Falls Sie verhindern möchten, dass andere Programmierer bestimmte Methoden der Basisklasse vererben und dann in der neuen Klasse überschreiben, können Sie dies in der Basisklasse vermerken. Hierzu ist das Schlüsselwort *NotOverridable* vorgesehen. Dieses Schlüsselwort lässt sich jedoch nur einsetzen, wenn in der Deklaration des Members der Basisklasse ein Member einer geerbten Klasse überschrieben wird. Andererseits hat jeder Member einer Basisklasse automatisch den Status *NotOverridable*. Sie können daher zu einem Trick greifen und alle erweiterbaren Member der Basisklasse mit dem Schlüsselwort *Overridable* versehen. Die Member, bei deren Deklaration dieses Schlüsselwort fehlt, sind dann in abgeleiteten Klassen nicht überschreibbar. Dies ist in folgender Codesequenz der Fall. Die Methode *Clear()* der Basisklasse *WetterKlasse* enthält kein Schlüsselwort *Overridable*, während dies bei *ClearTemp()* der Fall ist. In der abgeleiteten Klasse *Wetter* wird daher das Überschreiben der Methode *Clear()* einen Übersetzungsfehler auslösen, während *ClearTemp* überschreibbar ist.

```

Class WetterKlasse
' Member der Klasse
' Eigenschaft: Temperatur, Methode: Clear

Public sName As String = ""      ' Stationsname
Public Temperatur As Integer     ' Temperaturwert

' Hier kommen die Methoden
Sub Clear () ' geschützt: löscht Temperatur
    Temperatur = 0                ' Temperaturwert auf Null
End Sub

Overridable Sub ClearTemp () ' Setze Temperatur auf Min
    Temperatur = GrenzenNeu.TMin
End Sub
End Class

Class Wetter                        ' abgeleitete Klasse
Inherits WetterKlasse

Overrides Sub Clear ()            ' Überschreiben unzulässig !!!!
    Temperatur = 5                ' Temperaturwert auf Min
End Sub

Overrides Sub ClearTemp ()        ' Hier klappt das Override
    Temperatur = GrenzenNeu.TMin
End Sub
End Class

```

Hinweis

Sie finden die Projektdateien des als Konsolleanwendung realisierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_09a` der Begleit-CD. Die Anweisung mit dem Schlüsselwort `NotInheritable` wurde auskommentiert, so dass die Klasse vererbt werden kann. Weiterhin wurde in der Methode `Clear()` der abgeleiteten Klasse in der Deklaration das Schlüsselwort `Overridable` eingefügt, um das Beispiel übersetzen zu können.

7.3.5 Abstrakte Basisklassen, das steckt dahinter

Hinter dem Begriff »abstrakte Basisklasse« steckt letztendlich eine Klasse, die sich nicht direkt (z.B. mit `New`) instantiieren, sondern nur mit `Inherits` ableiten lässt. Diese Klasse wird mit dem Schlüsselwort `MustInherit` eingeleitet.

```
MustInherit Class Name
...
End Class
```

Weil sich die Klasse wegen dieses Schlüsselworts nicht mehr instantiieren lässt, darf innerhalb der Klasse auch kein Konstruktor (z.B. `New`) implementiert werden. Bei Methoden, die beim Vererben überschrieben werden müssen, wird in der Basisklasse das Schlüsselwort `MustOverride` angegeben. Bei einer abstrakten Basisklasse gilt dann, dass die mit `MustOverride` versehene Methode keine Anweisungen aufweisen darf. Die Methodendeklaration innerhalb der abstrakten Basisklasse besteht in diesem Fall nur aus einer Zeile mit dem Methodenkopf

```
MustOverride Function GetName () As String
```

Die Implementierung der betreffenden Methode erfolgt anschließend in der abgeleiteten Klasse. Dort lassen sich auch Konstruktoren implementieren.

Hinweis

Sinn macht dieser Ansatz dann, wenn nur eine Art »Blaupause« einer Klasse geschaffen werden soll, die die Nutzer dazu zwingt, die Vorgaben der Blaupause selbst durch Implementierung der Methoden und Konstruktoren nach eigenem Gusto auszuführen. Durch die Blaupause wird quasi der Rahmen, in dem sich die abgeleiteten Klassen bewegen, vorgegeben.

Das Ganze möchte ich an einem stark vereinfachten Beispiel verdeutlichen. Aus den vorhergehenden Beispielen habe ich die Basisklasse `WetterPrimitiv` so weit entkernt, dass nur noch zwei Methoden zum Setzen bzw. Lesen des Stationsnamens übrig geblieben sind. `WetterPrimitiv` wird als abstrakte Basisklasse vereinbart, wobei die Methode `GetName` mit `MustOverride` versehen ist. Die Basisklasse wird der Klasse `Wetter` vererbt. In dieser Klasse werden der `New`-Konstruktor sowie die Methode `GetName()` implementiert. Die Methode `SetName` wird dagegen aus der abstrakten Basisklasse geerbt. Ein Haupt-

programm instantiiert dann die Klasse *Wetter* und weist dieser über die Methoden zwei unterschiedliche Stationsnamen zu. Die Ergebnisse werden im Fenster der Konsole angezeigt. Das folgende Listing zeigt die Details der Implementierung:

```
'*****
' File/Projekt: Beispiel7_12
' Autor: G. Born www.borncity.de
' Beispiel für eine abstrakte Basisklasse.
'*****
Option Strict On
Imports System.Console      ' für WriteLine

' ### die abstrakte Basisklasse
MustInherit Class WetterPrimitiv
    Public sName As String = ""

    ' Hier kommen die Methoden der Basisklasse
    Sub SetName (name As String) ' Setze Name
        sName = name
    End Sub

    ' GetName darf keine Befehle enthalten
    MustOverride Function GetName () As String
End Class

' #####
' Leite die neue Klasse aus der abstrakten
' Basisklasse WetterPrimitiv ab.
' #####
Class Wetter
    Inherits WetterPrimitiv      ' aus Basisklasse ableiten

    ' Implementieren des New-Konstruktors für die Klasse
    Sub New (byVal Station As String) ' Erzeuge Instanz
        MyBase.New ()
        MyBase.SetName (Station)      ' Setze Stationsname
    End Sub

    ' Implementiere GetName
    Overrides Function GetName () As String
        GetName = MyClass.sName      ' lese Namen
    End Function
End Class
```

Listing 7.9: Beispiel zur Implementierung einer abstrakten Basisklasse

```
End Function
End Class

Class Test
    Shared Sub Main()                ' das ist das Hauptmodul
        ' Objekt aus Klasse Wetter instantiieren
        Dim station As Wetter = New Wetter("Bern")

        With station
            WriteLine ("Station: {0}", .GetName)' zeige Namen
            .SetName ("Berlin")                ' ändere den Namen
            WriteLine ("Station: {0}", .GetName)' zeige Namen
        End With

        Write ("Bitte die Eingabetaste drücken")
        ReadLine
    End Sub
End Class
```

Listing 7.9: Beispiel zur Implementierung einer abstrakten Basisklasse (Forts.)**Hinweis**

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_12` der Begleit-CD.

7.4 Ereignisse und Delegates

Zu den Members einer Klasse können auch Ereignisse gehören. Ereignisse bieten eine Möglichkeit, auf einen geänderten Zustand (Anklicken einer Schaltfläche per Maus, in einem Verzeichnis wird eine Datei angelegt etc.) zu reagieren und über ein Objekt Informationen zu dieser Änderung zu übertragen. Die Reaktion auf Ereignisse in Ereignisbehandlungsroutinen wird (zum Beispiel bei der Formularbearbeitung) in den folgenden Kapiteln noch ausgiebig genutzt).

Ein Delegate ist ein Objekt, welches einen Mechanismus bereit stellt, um Prozeduren, Funktionen oder Methoden aufzurufen, von denen nur die Adresse (nicht aber der Name) bekannt ist. Letztendlich handelt es sich bei einem Delegate um eine Datenstruktur, die auf eine statische Methode oder auf eine Klasseninstanz sowie auf eine Methode dieser Instanz verweist. Beim Aufruf einer Ereignisbehandlungsprozedur kommen intern Delegates zum Einsatz. Sie können Delegates aber auch in eigenen Programmen zum Aufruf von Methoden einer Klasse verwenden. Nachfolgend finden Sie eine kurze Einführung in das Arbeiten mit Delegates und Ereignissen.

7.4.1 Arbeiten mit Delegate-Objekten

Die Klasse *Delegate* wird im Namensraum *System.Delegate* bereitgestellt. Die Nutzung eines Delegate-Objekts möchte ich an einem einfachen Beispiel verdeutlichen. Eine Funktion zur Addition zweier Integerwerte soll über ein Delegate aufgerufen werden. Nehmen wir an, die Funktion sei folgendermaßen in einer Klasse definiert:

```
Shared Function AddInt (ByVal I1 As Integer, _
                      ByVal I2 As Integer) As Integer
    Return I1 + I2
End Function
```

Aus den vorhergehenden Kapiteln wissen Sie, dass sich die Funktion ganz einfach mit der Anweisung

```
i = AddInt(100,200)
```

aufrufen lässt. In diesem Fall werden die beiden Konstanten 100 und 200 addiert, die Funktion gibt das Ergebnis 300 zurück. Jetzt soll diese Funktion indirekt über ein *Delegate*-Objekt aufgerufen werden.

Als Erstes ist ein Delegate über das Schlüsselwort *Delegate* zu vereinbaren. Die Delegates lassen sich übrigens auf Namespaces-, Modul- oder Klassen-Ebene, nicht jedoch auf der Ebene der *Main()*-Prozedur deklarieren. Dann kann das Delegate wie ein Typ oder eine neue Klasse verwendet werden.

```
Public Delegate Function AddVal (ByVal Wert1 As Integer, _
                                ByVal Wert2 As Integer) As Integer
```

Die obige Anweisung vereinbart, dass sich das Delegate *AddVal* auf eine Funktion bezieht, die zwei Parameter *Wert1* und *Wert2*, beide vom Typ *Integer*, erwartet. Die Funktion liefert dann ein Ergebnis vom Typ *Integer* zurück. Innerhalb einer Klasse lässt sich nun eine Instanz auf dieses Delegate vereinbaren:

```
Dim Addiere As New AddVal (AddressOf AddInt)
```

Die Anweisung deklariert eine Objektvariable mit dem Namen *Addiere*. In dieser Objektvariablen wird eine Instanz des Delegate *AddVal* hinterlegt. Beim Aufruf des *New*-Konstruktors wird dem Delegate dabei als Parameter die Adresse der gewünschten Prozedur, Methode oder Funktion (hier die Funktion *AddInt*) übergeben. Die Adresse lässt sich mit dem Schlüsselwort *AddressOf xx* ermitteln, wobei *xx* hier für den Funktionsnamen steht. Mit der Anweisung erreichen wir also, dass die Objektvariable *Addiere* auf die Funktion verweist. Zur Addition zweier Integerwerte lässt sich dann die Anweisung

```
i = Addiere (100, 200)
```

verwenden.

Das folgende Listing zeigt den kompletten Code des Beispiels mit dem Delegate und der Funktion. Die Funktion wird dabei einmal konventionell über den Funktionsnamen und dann über das Delegate aufgerufen. Das Ergebnis der Addition erscheint in einem Meldungsfeld.

```
'*****
' File/Projekt: Beispiel7_14
' Autor: G. Born www.borncity.de
' Nutzen eines Delegate.
'*****
Option Strict On

Imports System.Delegate          ' für Delegate

' Delegate deklarieren
Public Delegate Function AddVal(ByVal Wert1 As Integer, _
                                ByVal Wert2 As Integer) As Integer

Class Test
    Shared Sub Main()
        Dim Addiere As New AddVal (AddressOf AddInt) ' Delegate-Objekt
        Dim i, j As Integer

        ' Funktion über den normalen Funktionsnamen aufrufen
        i = AddInt(100, 200)

        ' Jetzt kommt der Delegate-Aufruf
        j = Addiere(100, 200)

        ' Ergebnisse anzeigen
        MsgBox ("Addiere 100 + 200" & vbCrLf & _
                "Ergebnis per AddInt   : " & i.ToString & vbCrLf & _
                "Ergebnis per Delegate : " & j.ToString)
    End Sub

    Shared Function AddInt (ByVal I1 As Integer, _
                            ByVal I2 As Integer) As Integer
        Return I1 + I2
    End Function
End Class
```

Listing 7.10: Verwendung eines Delegate

Hinweis

Das obige Additions-Ergebnis hätte man auch einfacher durch den direkten Aufruf der betreffenden Funktion haben können. Es handelt sich lediglich um ein Beispiel, welches den Einsatz von Delegates demonstrieren soll. Delegates erlauben Ihnen z.B. unterschiedliche Methoden unter einem Namen aufzurufen. Eine weitere Variante zum indirekten Aufruf von Prozeduren oder Funktionen wird zur Ereignisbehandlung genutzt. Jedes Mal, wenn ein Benutzer auf eine Schaltfläche in einem Formular klickt, wird ein Ereignis ausgelöst. Auf dieses Ereignis kann in einer Ereignisbehandlungsroutine reagiert werden. Sie können die Anweisung *AddHandler* als Funktionsdelegate verwenden, um eine Ereignisbehandlungsprozedur anzumelden. Die Prozedur wird dann aufgerufen, sobald das Ereignis auftritt. Im Kapitel zur Gestaltung von Formularen wird dies noch behandelt.

Sie finden die Projektdateien des als Windows-Anwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_14` der Begleit-CD.

7.4.2 Delegates und Ereignisse nutzen

In einem weitere Beispiel soll die Kombination von Delegates und Ereignissen genutzt werden. Ereignisse lassen sich dabei durch *RaiseEvent* auslösen. Anschließend muss der Empfang der betreffenden Ereignisse implementiert werden. Dies kann über eine mit *WithEvents* deklarierte Objektvariable geschehen. Die Ereignisbehandlungsprozeduren müssen dann eine *Handles*-Anweisung beinhalten, die festlegt, welches Ereignis abgefangen wird. Ist dieser Mechanismus nicht korrekt eingerichtet, verfallen die mit *RaiseEvent* ausgelösten Ereignisse. Der Rahmen für die Ereignisbehandlungsprozedur wird übrigens durch die Delegates vereinbart.

Das Ganze lässt sich durch Abwandeln des vorherigen Beispiels demonstrieren. Die im vorhergehenden Beispiel implementierte Methode *AddInt()* soll während der Addition prüfen, ob die Ergebnisse oberhalb oder unterhalb bestimmter Grenzwerte liegen. Werden Grenzwerte verletzt, soll die Methode automatisch ein betreffendes Ereignis auslösen.

In der Klasse *Add* werden die beiden Delegates *MinLim_delegate* und *MaxLim_delegate* als Prozeduren mit den Parametern deklariert. Im ersten Parameter wird ein Objekt übergeben, während der zweite Parameter einen Integer-Wert enthält. Diese Deklaration kann dann als Muster für die Parameterliste der beiden Ereignisse dienen.

```
Public Delegate Sub MinLim_delegate( _
    ByVal oObj As Object, _
    ByVal Wert As Integer)

Public Delegate Sub MaxLim_delegate( _
    ByVal oObj As Object, _
    ByVal Wert As Integer)
```

Jetzt gilt es, die Ereignisse mit dem Schlüsselwort *Event* zu definieren. Diese Definition kann dabei in Klassen, Modulen, Strukturen und Schnittstellen erfolgen. Hinter dem Schlüsselwort *Event* folgt der Name des Ereignisses sowie der Typ des zugehörigen Delegate.

```
Public Event MaxLim1 As MaxLim_delegate  
Public Event MinLim1 As MinLim_delegate
```

Hier wurden zwei Ereignisse *MaxLim1* und *MinLim1* definiert, wobei diese vom Typ *MaxLim_delegate* bzw. *MinLim_delegate* sind. Sie können aber in der Typangabe auch Windows-Ereignisse wie *Windows.Forms.MouseEventHandler* etc. angeben.

Nun gilt es, den Empfang der Ereignisse vorzubereiten. Hierzu wird eine Objektvariable *obj* auf Klassenebene (hier in der Klasse *Add*) mit *WithEvents* vereinbart.

```
Dim WithEvents oObj As Add
```

WithEvents gibt an, dass mindestens eine deklarierte Membervariable auf die Instanz der Klasse verweist, die Ereignisse auslösen kann.

Die deklarierten Ereignisse lassen sich mit *RaiseEvent* auslösen. Hierzu wurden im aktuellen Beispiel die betreffenden Anweisungen in der Methode *AddInt()* untergebracht.

```
Function AddInt(ByVal I1 As Integer, _  
    ByVal I2 As Integer) As Integer  
    Dim i As Integer  
    i = I1 + I2      ' addiere  
    If (i < 10) Then    ' minimale Grenze unterschritten?  
        RaiseEvent MinLim1(Me, i) ' Ereignis auslösen  
        Return 10  
    ElseIf (i > 100) Then  
        RaiseEvent MaxLim1(Me, i) ' Ereignis auslösen  
        Return 100  
    Else  
        Return I1 + I2  
    End If  
End Function
```

Nach diesen Vorbereitungen lassen sich Ereignisbehandlungsprozeduren innerhalb der Klasse anlegen. Wenn Sie im Codefenster der Entwicklungsumgebung die Klasse und dann die mit *WithEvents* deklarierte Objektvariable *obj* im linken Listefeld *Klassennamen* anwählen, lassen sich im rechten Feld *Membersnamen* die verfügbaren Ereignisse abrufen. Deren Prozedurrumpf wird dann von der mit der *Delegate*-Anweisung vereinbarten »Schablone« abgeleitet. Die nachfolgenden Anweisungen zeigen zwei Ereignisbehandlungsroutinen:


```

Private Sub Add_MaxLim(ByVal oObj As Object, _
    ByVal Wert As Integer) Handles Me.MaxLim1
' wird aus der Add()-Methode aufgerufen,
' wenn Ergebnis Max überschreitet
    MsgBox("Wert: " & Wert.ToString, MsgBoxStyle.OkOnly, _
        "MaxLimit überschritten")
End Sub

Private Sub Add_MinLim(ByVal oObj As Object, _
    ByVal Wert As Integer) Handles Me.MinLim1
' wird aus der Add()-Methode aufgerufen,
' wenn Ergebnis Min unterschreitet
    MsgBox("Wert: " & Wert.ToString, MsgBoxStyle.OkOnly, _
        "MinLimit unterschritten ")
End Sub

```

Die *Handles*-Angabe legt dabei fest, welches der Ereignisse des *Event*-Objekts behandelt wird. Im Beispiel wird in der Startprozedur der Anwendung ein neues Objekt der Klasse *Add* instantiiert. Dann ruft das Programm die *AddInt()*-Methode der Klasse auf, wobei verschiedene Werte übergeben werden. Bei jeder Grenzwertverletzung tritt ein Ereignis auf, welches in der Ereignisbehandlungsroutine durch einen Dialog gemeldet wird.

Hinweis

Sie finden die Dateien des Beispiels samt dem gesamten Code der Implementierung im Ordner `\Beisp\Kap07\Beispiel7_14a` der Begleit-CD.

7.4.3 Ereignishandler über *AddHandler* anbinden

In obigem Beispiel wurden die Ereignisbehandlungsprozeduren in der Klasse *Add* hinterlegt. Sie können aber auf das Schlüsselwort *Handles* im Kopf der Ereignisbehandlungsprozedur verzichten. Voraussetzung ist lediglich, dass Sie in der *Main()*-Prozedur die Ereignisbehandlung mit *AddHandler* vereinbaren. Ausgehend von dem obigen Beispiel wurden entsprechende Anweisungen in der *Main()*-Prozedur der Anwendung hinterlegt.

```

Shared Sub Main()
    Dim oAdd As New Add() ' hole neue Instanz der Klasse

    AddHandler oAdd.MaxLim1, AddressOf Add_MaxLim
    AddHandler oAdd.MinLim1, AddressOf Add_MinLim

    ...
End Sub

```

Die Anweisungen binden die Ereignisse *MaxLim1* und *MinLim1* der über die *oAdd*-Objektvariablen referenzierten Instanz an die beiden Ereignishandler *Add_MaxLim* und *Add_MinLim*. Der Ereignishandler für das *MaxLim1*-Ereignis wurde innerhalb der Klasse *Text* mit folgendem Code implementiert:

```
Shared Sub Add_MaxLim(ByVal oObj As Object, ByVal Wert As Integer)
' wird aus der Add()-Methode aufgerufen,
' wenn Ergebnis Max überschreitet
  MsgBox("Wert: " & Wert.ToString, MsgBoxStyle.OkOnly, _
        "MaxLimit überschritten")
End Sub
```

Gegenüber dem vorherigen Beispiel fehlt lediglich die *Handles Me.MaxLim1*-Angaben im Prozedurkopf. Zudem findet sich die Prozedur in der Klasse *Test* und nicht in der Klasse *Add*.

Hinweis

Sie finden die Dateien des Beispiels samt dem gesamten Code der Implementierung im Ordner *\Beisp\Kap07\Beispiel7_14b* der Begleit-CD. Die *AddHandler*-Anweisung stellt eine elegante Möglichkeit dar, sogenannte MultiCast-Delegates zu vereinbaren. Hierbei besteht die Möglichkeit, mehrere Ereignisbehandlungsprozeduren mit einem Ereignis zu verbinden. Hierzu wird einfach die *AddHandler*-Anweisung für jede zusätzliche Ereignisbehandlungsprozedur erneut angegeben.

```
AddHandler oAdd.MaxLim1, AddressOf Add_InfoMaxLim1
AddHandler oAdd.MaxLim1, AddressOf Add_InfoGrenze
```

Die beiden obigen Anweisungen weisen das *MaxLim1*-Ereignis den Prozeduren *Add_InfoMaxLim1* und *Add_InfoGrenze* zu. Eine mit *AddHandler* eingerichtete Verbindung zum Ereignishandler lässt sich mit *RemoveHandler* wieder aufheben.

7.5 Schnittstellen

Schnittstellen erlauben gemeinsame Merkmale unterschiedlicher Klassen zu definieren. Die Schnittstelle legt dabei die von der Klasse bereitgestellten Member fest. Schnittstellen können (wie Klassen) Methoden, Eigenschaften und Ereignisse als Member enthalten. Der Code zur Implementierung dieser Member ist aber niemals Bestandteil dieser Schnittstelle. Vielmehr muss dieser Code in Klassen oder Strukturen implementiert und für die Schnittstelle bereitgestellt werden. Es lassen sich auch keine Objekte vom Typ einer Schnittstelle erzeugen. Die Schnittstelle erlaubt es, die Definition der Fähigkeiten von der Implementierung zu trennen.

7.5.1 Ein Szenario zur Verwendung von Schnittstellen

An einem fiktiven Szenario wird der Nutzen von Schnittstellen deutlich. Sie greifen in Ihren Anwendungen auf eine Klassenbibliothek (z.B. das .NET Framework oder die bereits in Beispielen vorgestellte Klasse zur Abbildung einer Wetterstation) zurück. Diese Klassenbibliothek unterliegt aber noch der Entwicklung, d.h., es könnten zukünftig neue Versionen auftauchen, die bestehende Klassen anpassen, erweitern oder neue Klassen definieren. Für unsere Klasse *Wetter* könnte es sein, dass diese erweitert wird oder durch eine andere Klasse *Wetter1* ersetzt wird. Nehmen wir mal an, die neue Klasse *Wetter1* verspricht eine dringend benötigte neue Methode, soll aber die Funktion der alten Klasse *Wetter* beinhalten. Durch den objektorientierten Ansatz ist es ein Leichtes, ein bestehendes Programm mit einer einzigen Anweisung auf die neue Klasse umzusetzen. Das folgende Codefragment vereinbart in der ersten Zeile eine Objektvariable und weist dieser eine Instanz der Klasse *Wetter* zu. Die Folgezeilen benutzen dann Member dieser Instanz, um Eigenschaften zu setzen. Aus Platzgründen habe ich hier nur zwei Zeilen des betreffenden alten Programmcodes gezeigt.

```
Dim oStation As Wetter = New Wetter()
    oStation.Name = "Bern"
    oStation.Temperatur = 15
...
```

Jetzt soll dieser Code nicht mehr mit der Klasse *Wetter*, sondern mit der neuen Klasse *Wetter1* benutzt werden. Die obige Sequenz muss nur in einer Zeile angepasst werden.

```
Dim oStation As Wetter1 = New Wetter1()
```

Statt der Klasse *Wetter* wird bei der Deklaration der Variablen *oStation* die neue Klasse *Wetter1* angegeben. Die restlichen Anweisungen

```
oStation.Name = "Bern"
oStation.Temperatur = 15
```

bleiben unverändert, da sich über das Objekt ein Bezug auf die Klasse ergibt. Tolle Sache, oder nicht? Der Ansatz ermöglicht Ihnen quasi auf »Knopfdruck«, komplette Projekte auf eine andere Klasse umzustellen. Bietet die Klasse neue Member, können Sie diese zusätzlich im Code einbauen.

Leider hat die ganze Sache einen Haken. Sie müssten darauf vertrauen, dass die neue Klasse mindestens alle Member der alten Klasse bereitstellt. Hat der Entwickler der neuen Klasse einen Member vergessen oder umbenannt, wird die Umstellung nicht funktionieren. Bei einem Trivialprogramm werden Sie den fehlenden Member, sofern dieser benutzt wurde, recht schnell bemerken. Was ist aber bei umfangreichen Projekten mit vielen tausend Codezeilen?

An dieser Stelle kommen Schnittstellen ins Spiel. Die Schnittstelle vereinbart einen Vertrag, der von der implementierenden Klasse eingehalten werden muss. Fordert die Schnittstelle beispielsweise eine Methode *SetName()*, ist diese aber in der Implementie-

rung der Klasse nicht vorgesehen, wird dieser Vertrag gebrochen. Der Visual-Basic-2005-Compiler kann beim Übersetzen anhand der Schnittstellenbeschreibung feststellen, ob die Vereinbarungen von den benutzten Klassen eingehalten werden. Zudem lässt sich im Programm feststellen, ob ein Member einer Klasse eine bestimmte Schnittstelle unterstützt. Die Definition einer Schnittstelle erfolgt mit folgenden Anweisungen:

```
Interface IName
...
End Interface
```

Der Begriff *IName* steht für den Namen der Schnittstelle. Es hat sich dabei eingebürgert, diesen Namen mit dem Großbuchstaben I einzuleiten. Microsoft schlägt vor, Substantive oder Adjektive als Schnittstellennamen zu verwenden. Zwischen diesen beiden Zeilen stehen dann die Deklarationen für die Member (Ereignisse, Eigenschaften, Methoden). Dabei werden jedoch nur die Deklarationsköpfe, aber kein Code angegeben. Die folgenden Anweisungen definieren eine Schnittstelle *IWetter*:

```
Interface IWetter
Property Temperatur As Integer
Sub SetName (name As String)
Function GetName () As String
End Interface
```

Die Schnittstelle *IWetter* stellt dann nichts weiter als einen Typ dar (der sich in einem Programm auswerten lässt). Die Implementierung der Schnittstelle kann in Klassen oder Strukturen erfolgen. Nachfolgend sehen Sie das Skelett der Klasse, die die obige Schnittstelle implementiert:

```
Class Wetter: Implements IWetter
Property Temperatur () As Integer Implements IWetter.Temperatur
' Anweisungen für die Eigenschaft
End Property

Sub SetName (name As String) Implements IWetter.SetName
' Anweisungen für die Methode
End Sub

Function GetName () As String Implements IWetter.GetName
' Anweisungen für die Methode
End Function
End Class
```

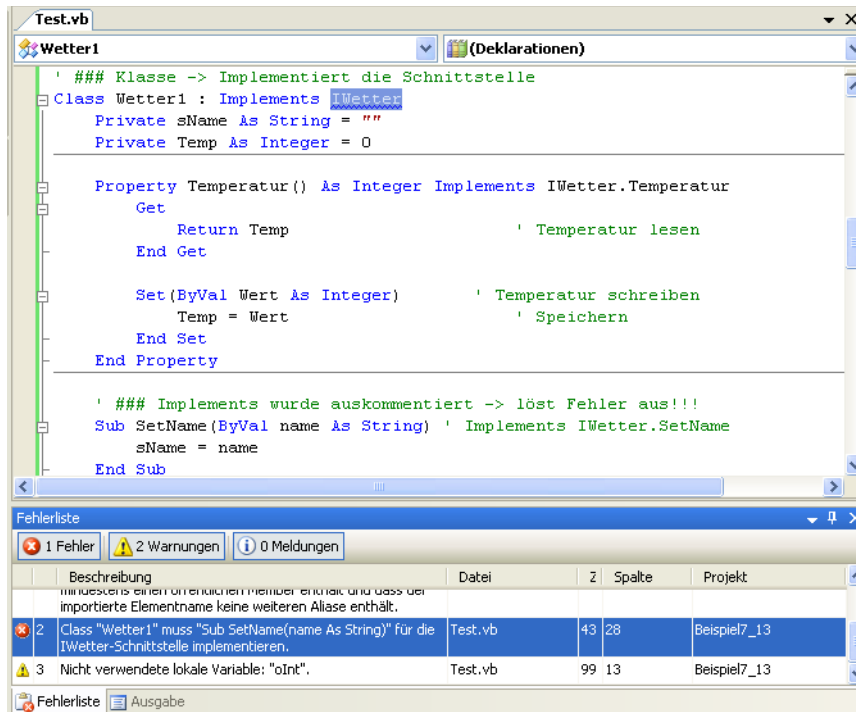


Abbildung 7.4: Fehlermeldung bei nicht erfüllter Schnittstelle

Ich habe die Details zur Implementierung der Eigenschaften und Methoden an dieser Stelle einmal weggelassen, um den Blick für das Wesentliche zu schärfen. Jede Eigenschaft und jede Methode, die in der Schnittstellenbeschreibung aufgeführt wurde, findet sich ebenfalls in der Klasse wieder. Hinter der betreffenden Deklarationszeile steht aber noch ein Hinweis, dass etwas implementiert wird. Bei der Klasse findet sich *Implements IWetter*, d.h., die Angabe signalisiert, dass diese Klasse behauptet, die Schnittstelle *IWetter* zu erfüllen. Die Methoden, die mit der Schnittstelle konform sind, enthalten Angaben der Art *Implements IWetter.SetName*, d.h., es wird genau aufgeführt, welchen Teil des Schnittstellenmembers der betreffende Klassenmember erfüllt. Der Compiler kann nun sehr schnell nach formalen Kriterien prüfen, ob die angegebene Klasse alle in der Schnittstelle angegebenen Member enthält. Die Klasse darf weitere Member enthalten. Sobald aber ein in der Schnittstelle geforderter Member in der angegebenen Klasse fehlt, schlägt der Compiler Alarm (Abbildung 7.4).

7.5.2 Beispiel zur Verwendung einer Schnittstelle

Das Ganze lässt sich an einem kompletten Beispiel demonstrieren. Dieses Beispiel benutzt zwei unterschiedliche Klassen, die eine Wetterstation abbilden sollen (aus Vereinfachungsgründen habe ich die Klassen um einige Member bereinigt). Die Schnittstellenbeschreibung legt fest, dass die Implementierung die Eigenschaft *Temperatur* und die beiden Methoden *SetName* und *GetName* bereitstellen muss. Dabei erfolgt die Implementierung

der Eigenschaft *Temperatur* in den beiden Klassen unterschiedlich. In der Klasse *Wetter1* wurde eine neue Methode *Clear()* implementiert, die alle Eigenschaften der Klasse auf Standardwerte zurücksetzt. Zum Test der Schnittstelle habe ich dabei noch die Angabe *Implements IWetter.SetName* in der Methode *SetName* der Klasse *Wetter1* auskommentiert. Wenn Sie das Beispiel übersetzen, wird der Compiler die in Abbildung 7.4 gezeigte Fehlermeldung bringen, denn die Schnittstelle verlangt, dass die Methode *SetName* von der Klasse bereitgestellt wird. Dass die Klasse eine zusätzliche Methode enthält oder dass die Eigenschaft *Temperatur* in den Klassen unterschiedlich implementiert wurde, interessiert dagegen nicht (die Schnittstelle sagt nichts über die Implementierung aus).

In einem Programm lässt sich mit der Anweisung *TypeOf Objekt Is IWetter* feststellen, ob eine Objektinstanz eine bestimmte Schnittstelle unterstützt:

```
Dim station As Wetter = New Wetter()  
If TypeOf station Is IWetter Then ...
```

Die *If*-Bedingung ist nur wahr, wenn die der Objektvariablen *station* zugrunde liegende Klasse auch die Schnittstelle *IWetter* unterstützt. Das folgende Listing demonstriert die Verwendung einer Schnittstelle *IWetter*, die in der Klasse *Wetter* korrekt und in der Klasse *Wetter1* teilweise implementiert ist. Weitere Details entnehmen Sie den nachfolgenden Anweisungen:

```
'*****  
' File/Projekt: Beispiel7_13  
' Autor: G. Born www.borncity.de  
' Beispiel für die Implementierung einer Schnittstelle.  
'*****  
Option Strict On  
Imports System.Console      ' für WriteLine  
  
' ### Definition der Schnittstelle  
Interface IWetter  
    Property Temperatur As Integer  
    Sub SetName (name As String)  
    Function GetName () As String  
End Interface  
  
' ### Klasse -> Implementiert die Schnittstelle IWetter  
Class Wetter: Implements IWetter  
    Private sName As String = ""  
    Private Temp As Integer = 0  
  
    Property Temperatur () As Integer Implements IWetter.Temperatur  
        Get
```

Listing 7.11: Implementierung einer Schnittstelle

```

    Return Temp                ' Temperatur lesen
End Get

Set (byVal Wert As Integer)    ' Temperatur schreiben
    Temp = Wert                ' Speichern
    If Wert < -40 then Temp = -40 ' auf Grenzen fixieren
    If Wert > 80 then Temp = 80
End Set
End Property

Sub SetName (name As String) Implements IWetter.SetName
    sName = name
End Sub

Function GetName () As String Implements IWetter.GetName
    GetName = sName            ' lese Namen
End Function
End Class

' ### Klasse -> Implementiert die Schnittstelle
Class Wetter1: Implements IWetter
    Private sName As String = ""
    Private Temp As Integer = 0

    Property Temperatur () As Integer Implements IWetter.Temperatur
        Get
            Return Temp          ' Temperatur lesen
        End Get

        Set (byVal Wert As Integer)    ' Temperatur schreiben
            Temp = Wert              ' Speichern
        End Set
    End Property

    ' ### Implements wurde auskommentiert -> löst Fehler aus!!!
    Sub SetName (name As String) ' Implements IWetter.SetName
        sName = name
    End Sub

    Sub Clear ()

```

Listing 7.11: Implementierung einer Schnittstelle (Forts.)

```
sName = ""
Temp = 0
End Sub

Function GetName () As String Implements IWetter.GetName
    GetName = sName          ' lese Namen
End Function
End Class

Class Test
    Shared Sub Main()          ' das ist das Hauptmodul
        ' Objekte aus Klassen Wetter und Wetter1 unter Verwendung
        ' des Schnittstellentyps IWetter instantiieren
        Dim station As IWetter = New Wetter()
        Dim station1 As IWetter = New Wetter1()

        With station
            .SetName("Bern")
            .Temperatur = 25
            WriteLine ("Station: {0}, Temperatur: {1}", _
                .GetName, .Temperatur)          ' zeige Werte
        End With

        With station1
            .SetName("Berlin")
            .Temperatur = 27
            WriteLine ("Station: {0}, Temperatur: {1}", _
                .GetName, .Temperatur)          ' zeige Werte
        End With

        WriteLine()
        WriteLine("Unterstützung der Schnittstelle " & _
            "IWetter durch die Klassen")
        WriteLine ("In Klasse Wetter: {0}", _
            TypeOf Station Is IWetter)
        WriteLine ("In Klasse Wetter1: {0}", _
            TypeOf Station1 Is IWetter)

        Dim oInt As Object      ' zum Test Typ Object verwenden
        WriteLine ("IWetter-Unterstützung in Object: {0}", _
```

Listing 7.11: Implementierung einer Schnittstelle (Forts.)


```

        TypeOf oInt Is IWetter)

    Write ("Bitte die Eingabetaste drücken")
    ReadLine
End Sub
End Class

```

Listing 7.11: Implementierung einer Schnittstelle (Forts.)

Hinweis

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_13` der Begleit-CD. Wenn Sie das Projekt laden und erstellen lassen, wird die in Abbildung 7.4 gezeigte Fehlermeldung erscheinen. Sie müssen dann das Kommentarzeichen in der im obigen Listing fett ausgezeichneten Zeile

```
Sub SetName (name As String) ' Implements IWetter.SetName
```

entfernen. Dann sollte sich das Beispiel übersetzen und ausführen lassen.

7.5.3 Nutzen der .NET-Schnittstellen

Die .NET-Basisklassen stellen ebenfalls viele Schnittstellen zur Verfügung. Die *IComparable*-Schnittstelle ermöglicht zum Beispiel den Vergleich zweier Objekte. Sie können damit eigene Klassen entwerfen, die eine Sortierung von Objekten erlauben. Hierzu müssen Sie in der Klasse nur die *IComparable*-Schnittstelle sowie die Methode *CompareTo()* implementieren. Dies soll an einem einfachen Beispiel demonstriert werden. Es liegt eine Reihe von Messungen der Mittagstemperatur von verschiedenen Orten vor. Die Werte werden nun mit dem jeweiligen Ort in eine Liste eingefügt, wobei diese nach aufsteigenden Temperaturwerten anzulegen ist (Abbildung 7.5).



Abbildung 7.5: Anzeige der sortierten Liste

Die Liste lässt sich über die *System.Collections.SortedList*-Klasse erstellen. Diese erlaubt eine Auflistung von Schlüssel-Wert-Paaren, die nach Schlüsseln sortiert sind. Dabei lässt sich über Schlüssel oder Indizes auf die Auflistung zugreifen. Die aus einer Temperatur und einem Stationsnamen bestehenden Werte-Paare sollen so in der Auflistung hinterlegt werden, dass die Temperaturwerte als Schlüssel fungieren. Die Temperaturen liegen als Dezimalzahlen vor und sollen mit dem Typ *Double* abgelegt werden. Weiterhin sollen

die *ICompare*-Schnittstelle der Klasse und deren *CompareTo()*-Methode in einer abgeleiteten Klasse implementiert werden.

Nachfolgendes Listing zeigt den Code der betreffenden Klasse *Temperatur*. Die Definition der Schnittstelle steckt ja in der *.NET-SortedList*-Klasse. Folglich muss nur die Methode *CompareTo()* der Schnittstelle implementiert werden – konkret wird hier dafür gesorgt, dass die *CompareTo()*-Methode so überladen wird, dass *Double*-Werte bei der Sortierung berücksichtigt werden. Hierzu wird die geschützte Klassenvariable *m_val* vom Typ *Double* zur Aufnahme der einzelnen Temperaturwerte deklariert. Dann folgt die Überladung der *CompareTo()*-Methode mit *Implements IComparable*-Angabe. Bei der Überladung wird einfach die *CompareTo()*-Methode mit dem übergebenen Wert aufgerufen und das Ergebnis in *m_val* zurückgegeben.

Innerhalb der Klasse werden zusätzlich noch die Eigenschaft *Celsius* zum Schreiben und Lesen der Temperaturwerte sowie der *New*-Konstruktor für die Klasse implementiert. Der *New*-Konstruktor schreibt einfach den beim Instantiieren übergebenen Wert in die Eigenschaft *Celsius*.

```
Public Class Temperatur
    Implements IComparable(Of Temperatur)

    Protected m_val As Double = 0.0 ' Temperaturwert

    ' Implementiert die generische CompareTo-Methode.
    Public Overloads Function CompareTo( _
        ByVal other As Temperatur) As Integer _
        Implements IComparable(Of Temperatur).CompareTo
        ' CompareTo-Methode überladen (gebe Wert zurück)
        Return m_val.CompareTo(other.m_val)
    End Function

    Public Property Celsius() As Double ' Eigenschaft Celsius
    Get
        Return m_val ' Wert zurückgeben
    End Get

    Set(ByVal Value As Double)
        m_val = Value
    End Set
End Property

    Public Sub New(ByVal Wert As Double)
        Me.Celsius = Wert
    End Sub
End Class
```

Mit dieser Klasse lässt sich nun die *SortedList*-Aufzählung sehr einfach realisieren. Die entsprechende Objektvariable zur Aufnahme der Liste wird in der *Main()*-Prozedur mit folgender Anweisung deklariert:

```
Dim Temp As New SortedList(Of Temperatur, String)
```

Die Variable *Temp* verweist auf eine Instanz der *SortedList*-Klasse. Der *New*-Konstruktor erwartet im ersten Argument einen Schlüssel und im zweiten Argument einen Wert. Dem Schlüssel wurde hier der Typ der Klasse *Temperatur* zugewiesen. Damit wird erreicht, dass die *SortedList*-Klasse die in der Klasse *Temperatur* implementierte *CompareTo()*-Methode benutzt. Der Typ *String* im zweiten Parameter erlaubt die Stationsnamen als Werte abzulegen. Um einen neuen Wert in die *SortedList*-Auflistung aufzunehmen, ist dann folgende Anweisung erforderlich:

```
Temp.Add(New Temperatur(15.0), "Paris")
```

Die *Add()*-Methode fügt einen neuen Eintrag zur Auflistung hinzu. Der Schlüssel wird dabei über den *New*-Konstruktor der Klasse *Temperatur* mit dem angegebenen Initialisierungswert gefüllt. Als zweites Argument erwartet die *Add()*-Methode den Wert (hier den Stationsnamen als *String*). Über mehrfache Aufrufe der *Add()*-Methode lässt sich die Liste füllen, wobei die überschriebene *CompareTo()*-Methode der Klasse *Temperatur* dafür sorgt, dass die Schlüsselwerte nach den Kriterien für *Double*-Werte sortiert werden. Um die Listeneinträge anschließend in sortierter Reihenfolge abzufragen, lässt sich eine *For Each*-Schleife einsetzen:

```
For Each i As KeyValuePair(Of Temperatur, String) In Temp
    txt = txt & vbCrLf & i.Value & ": " & vbTab & i.Key.Celsius
Next
```

KeyValuePair() liefert eine Struktur mit dem jeweils in *Temp* enthaltenen Schlüssel-Wert-Paar an die Schleifenvariable *i*, wobei der Schlüssel vom Typ *Temperatur* ist. Der Wert des Eintrags (hier der Stationsname) steht in der *Value*-Eigenschaft der Struktur, während sich der Schlüssel über die *Key*-Eigenschaft ermitteln lässt. In obiger Schleife wird der Temperaturwert über die Eigenschaft *Celsius* der *Temp*-Instanz gelesen.

Hinweis

Sie finden die Projektdateien mit dem kompletten Code des als Windows-Anwendung implementierten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_13a` der Begleit-CD.

7.6 Collections und allgemeine Auflistungen

In *Kapitel 4* haben Sie die Möglichkeit der Verwendung von Feldern zum Speichern mehrerer Werte gleichen Typs kennen gelernt. Die Speicherung in Feldern ist zwar sehr einfach, hat aber den Nachteil, dass Sie vorab die Zahl der Elemente kennen und in den Felddefinitionen berücksichtigen müssen. .NET Framework stellt daher noch einige

Klassen (Namensraum *System.Collections* und *System.Collections.Specialized*) bereit, über die sich Auflistungen (Collections) erstellen und verwalten lassen. Solche Auflistungen erlauben die Speicherung gleichartiger Werte (Strings, Objekte etc.), wobei die Zahl der speicherbaren Werte einer Auflistung vom verfügbaren Speicher abhängt. In den folgenden Abschnitten möchte ich noch einige Techniken zum Umgang mit Auflistungen besprechen.

7.6.1 Nutzen der ArrayList-Klasse

Die Klasse *ArrayList* implementiert die *ICollection*-Schnittstelle unter Verwendung eines Arrays, das nach Bedarf dynamisch vergrößert wird. Die *ICollection*-Schnittstelle stellt eine nicht generische Auflistung von Objekten dar, auf die einzeln über einen Index zugegriffen werden kann. Die *ArrayList*-Klasse kommt also sehr nahe an das Arbeiten mit Feldern heran. Eine Objektvariable vom Typ *ArrayList* mit einer neuen Instanz der *ArrayList* lässt sich mit der Anweisung

```
Dim feld As New ArrayList()
```

deklarieren. Da die *ArrayList* die Werte als Objekte verwaltet, ist sie bezüglich der Werte allerdings nicht typsicher. Sie können also mittels der *Add()*-Methode der Klasse beliebige Werte zur Auflistung hinzufügen.

```
feld.Add("Müller")  
feld.Add("Bach")  
feld.Add(11)
```

Möchten Sie die Elemente der Auflistung sortieren, müssen diese vom gleichen Typ sein.

```
feld.Sort()
```

Der Aufruf der *Sort()*-Methode löst einen Laufzeitfehler aus, wenn unterschiedliche Typen in den Elementen gefunden werden. Der Zugriff auf die Elemente der Auflistung ist über einen Index der Art *feld(1)* möglich. Alternativ können Sie die Elemente der Auflistung über eine *For Each*-Schleife abrufen. Um Elemente aus der Auflistung zu entfernen, bietet die *ArrayList*-Klasse die *Remove()*-Methode.

```
feld.Remove(entries(0))  
feld.Remove(entries(1))  
feld.Remove(11)
```

Die Methode erwartet dabei das aus der Auflistung zu entfernende Objekt. Wird das Objekt nicht gefunden, bleibt der Methodenaufruf folgenlos (es tritt kein Laufzeitfehler auf). Nachfolgendes Listing demonstriert den Einsatz der *ArrayList*-Klasse und der hier beschriebenen Methoden:

```

'*****
' File/Projekt: Beispiel07_10
' Autor: G. Born www.borncity.de
' Beispiel zur Verwendung einer ArrayList-Collection.
'*****

Option Strict On

Imports System.Windows.Forms

Public Class Test
    Shared Sub Main()
        Const Title As String = "ArrayList-Beispiel"
        Dim entries() As String = {
            "Müller", "Bach", "Huber", "Braun", "Bauer", "Bayer"}
        Dim feld As New ArrayList() ' erzeuge eine neue ArrayList
        Dim txt As String = ""      ' temporäre Variable
        Dim i As Integer           ' Schleifenindex

        ' Jetzt Einträge in die ArrayList einfügen
        For i = 0 To entries.Length - 1
            feld.Add(entries(i))    ' Wert an Instanz der Klasse
        Next

        If MessageBox.Show("Integer einfügen und auf Sort verzichten?", _
            Title, MessageBoxButtons.YesNo) = DialogResult.Yes Then
            ' jetzt noch ein paar Integer einfügen
            ' -> geht, da feld ein Objekt ist!
            feld.Add(11)
            feld.Add(10)
        Else ' bei Zahlen kein Sort(), da sonst Laufzeitfehler
            feld.Sort()             ' Sortiere die ArrayList
        End If

        ' jetzt Auflistung auswerten
        txt = ""
        For i = 0 To feld.Count - 1
            txt &= feld(i).ToString & vbCrLf ' lese über Index
        Next

        MsgBox(txt, MsgBoxStyle.Information, Title)
    End Sub
End Class

```

Listing 7.12: Anwendung einer ArrayList-Auflistung

```

' jetzt über For Each ausgeben
txt = ""
For Each j As Object In feld
    txt &= j.ToString & vbCrLf
Next
MsgBox(txt, MsgBoxStyle.Information, Title)

' jetzt Elemente aus der ArrayList entfernen
' löst keinen Laufzeitfehler aus, wenn Element nicht gefunden
feld.Remove(entries(0))
feld.Remove(entries(1))
feld.Remove(11)

' jetzt über For Each ausgeben
txt = ""
For Each j As Object In feld
    txt &= j.ToString & vbCrLf
Next
MsgBox(txt, MsgBoxStyle.Information, Title)
End Sub
End Class

```

Listing 7.12: Anwendung einer ArrayList-Auflistung (Forts.)

Hinweis

Die Projektdateien des als Windows-Anwendung realisierten Beispiels finden Sie im Ordner `\Beisp\Kap07\Beispiel7_10` auf der Begleit-CD.

7.6.2 Verwenden der HashTable-Klasse

Die Klasse *HashTable* implementiert die *IDictionary*-Schnittstelle, die eine nicht generische Auflistung von Wert-Schlüssel-Paaren darstellt. Die *HashTable* erlaubt sehr schnelle Zugriffe auf die Schlüssel, wenn der Hashcode vorliegt. Allerdings müssen Sie beim Einfügen der Elemente dafür sorgen, dass diese einen eindeutigen Hashcode aufweisen. Eine neue Instanz der *HashTable* lässt sich mit der Anweisung

```
Dim tableAs New ArrayList()
```

deklarieren und der Variablen *table* zuweisen. Zum Hinzufügen von Werten müssen Sie diesen immer einen Schlüssel mit einem eindeutigen Hashcode beifügen (existiert bereits ein Schlüssel, wird dessen Wert überschrieben). Nachfolgend werden numerische Indizes aus dem Schleifenindex generiert und mittels der *Add()*-Methode der Auflistung hinzugefügt.

```
For i = 0 To entries.Length - 1
    table.Add(i, entries(i))    ' i = key, Wert aus entries
Next
```

Die Werte, eine Folge von als Strings hinterlegten Namen, finden sich hier im Feld *entries(i)*. Um anschließend auf die Werte der *HashTable*-Auflistung zuzugreifen, lässt sich eine *DictionaryEntry*-Struktur nutzen.

```
Dim dict As DictionaryEntry
txt = ""
For Each dict In table
    txt &= "Nr: " & dict.Key.ToString & _
        " -> " & dict.Value.ToString & vbCrLf
Next
```

Die *For Each*-Schleife liefert die Einträge als *DictionaryEntry*-Struktur an die Variable *dict*. Anschließend lässt sich über den Member *Key* auf den Schlüssel und über *Value* auf den Wert zugreifen. Zum Entfernen eines Eintrags lässt sich die *Remove()*-Methode einsetzen.

```
table.Remove(3)
```

Als Argument ist der Methode der gewünschte Schlüssel zu übergeben. Mittels der *Clear()*-Methode lässt sich die gesamte *HashTable*-Auflistung leeren. Nachfolgendes Listing demonstriert den Einsatz der *HashTable*-Klasse:

```
'*****
' File/Projekt: Beispiel07_10a
' Autor: G. Born www.borncity.de
' Beispiel zur Verwendung einer HashTable-Collection.
'*****
Option Strict On

Imports System.Windows.Forms

Public Class Test
    Shared Sub Main()
        Const Title As String = "HashTable-Beispiel"
        Dim entries() As String = { _
            "Müller", "Bach", "Huber", "Braun", "Bauer", "Bayer"}
        Dim table As New HashTable() ' erzeuge eine neue HashTable
        Dim txt As String = ""        ' temporäre Variable
        Dim i As Integer              ' Schleifenindex
```

Listing 7.13: Anwendung einer *HashTable*-Auflistung

```
Dim dict As DictionaryEntry ' für DictionaryEinträge

' Jetzt Einträge in die HashTable einfügen
For i = 0 To entries.Length - 1
    table.Add(i, entries(i)) ' i = key, Wert aus entries
Next

' jetzt mit Dictionary-Klasse über Schlüssel ausgeben
txt = ""
For Each dict In table
    txt &= "Nr: " & dict.Key.ToString & _
        " -> " & dict.Value.ToString & vbCrLf
Next
MsgBox(txt, MsgBoxStyle.Information, Title)

' jetzt das Element mit dem Schlüssel 3 löschen
table.Remove(3)

' jetzt mit Dictionary-Klasse über Schlüssel ausgeben
txt = ""
For Each dict In table
    txt &= "Nr: " & dict.Key.ToString & _
        " -> " & dict.Value.ToString & vbCrLf
Next
MsgBox(txt, MsgBoxStyle.Information, Title)
End Sub
End Class
```

Listing 7.13: Anwendung einer HashTable-Auflistung (Forts.)

Hinweis

Die Projektdateien des als Windows-Anwendung realisierten Beispiels finden Sie im Ordner `\Beisp\Kap07\Beispiel7_10a` auf der Begleit-CD.

7.6.3 Nutzen der Collection-Klasse

Ein *Collection*-Objekt lässt sich in Visual Basic über die Klasse *Collection* instantiieren. Das *Collection*-Objekt stellt eine geordnete Gruppe von Elementen (Schlüssel-Wert-Paare) bereit, auf die über Schlüssel gezielt zugegriffen werden kann. Eine Instanz lässt sich durch folgende Anweisung anlegen und einer Objektvariablen zuweisen:

```
Dim Namen As New Collection()
```


In dieser *Collection* lassen sich jetzt zum Beispiel Klasseninstanzen speichern. Nehmen wir an, eine Klasse ist folgendermaßen definiert:

```
Public Class NamenListe
    Public instanceName As String
End Class
```

Dann wird mit der Anweisung *Dim inst As New NamenListe()* eine neue Instanz der Klasse erzeugt und der Objektvariablen *inst* zugewiesen. Der hier ausnahmsweise als öffentlich deklarierten Klassenvariable *instanceName* lässt sich dann über die Anweisung

```
inst.instanceName = "Bauer"
```

ein Wert zuweisen. Über die *Add()*-Methode kann nun diese Klasseninstanz mit dem in der Klassenvariablen gespeicherten Wert in die *Collection* übernommen werden.

```
Namen.Add(inst, CStr(key))
```

Das erste Argument der Methode enthält den Wert, während im zweiten Argument der Schlüssel zur Einordnung in die *Collection* erwartet wird. Um die Werte der *Collection* auszulesen, lässt sich eine *For ... Each*-Schleife verwenden.

```
For Each j As NamenListe In Namen
    txt &= j.instanceName & vbCrLf
Next
```

Dem Schleifenindex wird ein Eintrag der *Namen*-Collection vom Typ *NamenListe* der Klasseninstanz zugewiesen. Daher kann direkt über den Schleifenindex auf die *instanceName*-Klassenvariable zugegriffen werden. Möchten Sie gezielt über Schlüssel auf einzelne Elemente der *Collection* zugreifen, ist dies mit folgender Anweisungsfolge möglich:

```
k = CType(Namen.Item(3), NamenListe)
txt &= k.instanceName & vbCrLf
```

Zum Entfernen eines Elements lässt sich die *Remove()*-Methode der Klasse aufrufen. Als Parameter wird der Index des gewünschten Elements angegeben. Die *Collection* wird anschließend neu indiziert. Nachfolgendes Listing zeigt, wie sich Elemente in einer *Collection*-Auflistung speichern, abfragen und entfernen lassen:

```
'*****
' File/Projekt: Beispiel07_10b
' Autor: G. Born www.borncity.de
' Beispiel zur Verwendung einer Collection.
'*****
```

Listing 7.14: Arbeiten mit einer *Collection*-Auflistung

```
Option Strict On
Imports System.Console          ' für WriteLine
Imports Microsoft.VisualBasic

Public Class NamenListe        ' wird für Instanzen gebraucht
    Public instanceName As String
End Class

Public Class Test
    Shared Sub Main()
        Dim entries() As String = {"Bach", "Bauer", "Bayer", "Braun"}
        Dim Namen As New Collection() ' erzeuge eine neue Collection
        Dim txt As String = ""         ' temporäre Variable
        Dim i As Integer               ' Schleifenindex

        ' Jetzt Einträge in die Collection einfügen
        For i = 0 To entries.Length - 1
            Dim inst As New NamenListe() ' neue Instanz der Liste
            inst.instanceName = entries(i) ' Wert an Instanz der Klasse
            Namen.Add(inst, CStr(i + 1)) ' zur Collection addieren
        Next

        Dim k As NamenListe           ' lese jetzt gezielt
        For i = 1 To Namen.Count
            k = CType(Namen.Item(i), NamenListe)
            txt &= k.instanceName & vbCrLf
        Next
        MsgBox(txt, , "Instanzen-Namen in Collection 'Namen'")

        txt = ""
        ' jetzt liegt eine Auflistung von inst-Instanzen vor
        For Each j As NamenListe In Namen
            txt &= j.instanceName & vbCrLf
        Next
        MsgBox(txt, , "Instanzen-Namen in Collection 'Namen'")

        ' jetzt Elemente aus der Auflistung entfernen
        For i = 1 To Namen.Count
            Namen.Remove(1) ' immer 1. Objekt, da Re-Indizierung!
        Next
    End Sub
End Class
```

Listing 7.14: Arbeiten mit einer Collection-Auflistung (Forts.)

```

' prüfe, ob noch was vorliegt
txt = ""
For Each j As NamenListe In Namen
    txt &= j.instanceName & vbCrLf
Next
MsgBox(txt, , "Instanzen-Namen in Collection 'Namen'")
End Sub
End Class

```

Listing 7.14: Arbeiten mit einer Collection-Auflistung (Forts.)

Hinweis

Die Projektdateien des als Windows-Anwendung realisierten Beispiels finden Sie im Ordner `\Beisp\Kap07\Beispiel7_10b` auf der Begleit-CD.

7.6.4 Allgemeine Auflistungen mit »Generic«

In den bisherigen Visual-Basic-Versionen ließen sich zwar Auflistungen erstellen. Das .NET Framework 2.0 enthält nun im Namensraum `System.Collections.Generic` Schnittstellen und Klassen, die generische Auflistungen definieren. Der Vorteil der *Generic*-Elemente besteht darin, dass Sie beim Erstellen einer Instanz der Auflistung einen Typ für das betreffende Objekt angeben können. Sie müssen also nicht mehr einzelne benutzerdefinierte Auflistungen für unterschiedliche Objekttypen erstellen. Nehmen wir einmal folgende Klassendefinition, die einen benutzerdefinierten Datentyp *Kunde* vereinbart:

```

Public Class Kunde ' Klasse für Kundendatenstruktur
    Public Name As String
    Public Vorname As String
    Public Tel As String

    Public Sub New(ByVal TheName As String, _
        ByVal TheVorname As String, _
        ByVal TheTel As String)
        Name = TheName ' Eigenschaften-Member mit Werten belegen
        Vorname = TheVorname
        Tel = TheTel
    End Sub
End Class

```

Sie können in einer Instanz dieser Klasse den Namen, Vornamen und die Telefonnummer einer Person hinterlegen. Möchten Sie nun eine benutzerdefinierte Auflistung mit folgender Anweisung deklarieren:

```
Dim Kunden As New System.Collections.Generic.List(Of Kunde)
```

Bei der Instantiierung der *List*-Klasse wird für das Objekt *Kunden* eine strikte Typbindung an den Datentyp *Kunde* vorgenommen. Sie können also in dieser Auflistung nur Elemente vom Typ *Kunde* speichern. Um nun entsprechende Objekte vom Typ *Kunde* anzulegen, verwenden Sie folgende Anweisung:

```
Dim Kunde1 As New Kunde("Bauer", "Andreas", "069-301-222")
```

Anschließend benutzen Sie die *Add*-Methode des *Kunden*-Objekts, um das neue *Kunde*-Objekt zur Auflistung hinzuzufügen.

```
Kunden.Add(Kunde1)
```

Eine auf diese Weise erstellte benutzerspezifische Auflistung ist nicht nur an den bei der Deklaration angegebenen Typ gebunden. Die interne Behandlung bei Zugriffen auf die Auflistung ist auch effizienter, da nicht mehr mit dem allgemeinen Typ *Object* gearbeitet werden muss.

Hinweis

Der Namensraum *System.Collections.Generic.List* stellt neben der *List*-Klasse weitere Klassen (Dictionary, Queue etc.) bereit. Details finden Sie in der Hilfe, wenn Sie nach dem Namensraum *System.Collections.Generic.List* suchen lassen. Die Projektdateien eines als Windows-Anwendung realisierten Beispiels finden Sie im Ordner *\Beisp\Kap07\Beispiel7_15* auf der Begleit-CD. Das Element *Test.vb* enthält die Visual-Basic-Code, um eine *Generic-List* anzulegen und mit drei Kundenobjekten zu füllen. Anschließend wird der Inhalt der Auflistung in Dialogfeldern ausgegeben. Das Element *Class1.vb* enthält Visual-Basic-Code, der eine typlose Auflistung *Kunden* über die *ArrayList*-Klasse erzeugt.

7.7 Neuerungen und weitere Arbeitstechniken

Auf den vorhergehenden Seiten wurden an verschiedenen Stellen einige in Visual Basic 2005 eingeführte Neuerungen besprochen. In den folgenden Abschnitten möchte ich noch auf einige spezielle Neuerungen eingehen, die Microsoft in der aktuellen Visual-Basic-Version spendiert hat. Zudem wird noch gezeigt, wie sich Klassen in separaten Dateien oder als Bibliotheken anlegen lassen.

7.7.1 Arbeiten mit dem My-Objekt

Für Entwickler, die von Visual Basic 6.0 zu .NET wechseln, ist die durch das .NET Framework bereitgestellte Klassenbibliothek mit den angebotenen Klassen recht verwirrend. Microsoft hat daher in Visual Basic 2005 das *My*-Feature implementiert, welches den Zugriff auf einige Klassen des .NET Framework erleichtert. Das *My*-Feature macht zur Laufzeit häufig benötigte Objekte verfügbar, so dass Zugriffe auf deren Methoden vereinfacht werden. Die folgende Codesequenz verdeutlicht die Anwendung des *My*-Features:

```
Dim txt As String = ""

' Ermittle den NetBios-Namen des Rechners
txt = txt & "Computer: "

' alte Variante
txt = txt & SystemInformation.ComputerName & vbCrLf

' neue Variante
txt = txt & "Computer: " & My.Computer.Name & vbCrLf

txt = txt & "Speicher: " & My.Computer.Info.TotalPhysicalMemory _
    & vbCrLf
txt = txt & "OS: " & My.Computer.Info.OSFullName & vbCrLf
txt = txt & "Anwendung: " & My.Application.GetType.ToString

MsgBox(txt, MsgBoxStyle.OkOnly, "Infos")
```

Um beispielsweise den Computernamen abzufragen, müssten Sie den Namensraum *System.Windows.Forms* über die *Imports*-Anweisung im Projekt importieren und anschließend über *SystemInformation.ComputerName* auf die betreffende Eigenschaft zugreifen. Durch das *My*-Objekt entfällt der Import des Namensraums und Sie können die Eigenschaft direkt über

```
Name = My.Computer.Name
```

abfragen. Diese Schreibweise kommt Entwicklern mit Erfahrungen in Visual Basic 6.0 entgegen. Wer sich jedoch mit der .NET-Framework-Klassenbibliothek gut auskennt, wird kaum auf das *My*-Feature zurückgreifen. Daher wird das Feature hier im Buch auch kaum genutzt.

Hinweis

Detaillierte Hinweise auf die von *My* bereitgestellten Objekte *Application*, *User*, *Computer* und die Objekthierarchie finden Sie in der Hilfe, wenn Sie nach dem Begriff »My-Objekt« suchen lassen. Sie finden im Ordner *\Beisp\Kap07\Beispiel7_16* ein kleines Projektbeispiel, welches die Verwendung des *My*-Feature demonstriert.

7.7.2 Arbeiten mit Partial-Typen

Das neue Schlüsselwort *Partial* ermöglicht die Definition von Typen über mehrere Quelldateien hinweg. Dies wird beispielsweise von der Entwicklungsumgebung bei der Formularerstellung genutzt, wo der vom Assistenten erzeugte Code zum Aufbau des Formulars getrennt in einer eigenen versteckten Datei gehalten wird und Sie nur die Ereignisbehandlungsroutinen des Formulars zu sehen bekommen. Sie können die Partial-Typen aber auch selbst einsetzen, um eine Definition über mehrere Quelldateien zu verteilen. Nehmen wir die folgende Codesequenz, die drei Variable in einer Klasse definiert:

```
Public Class Test
    Public t1 As Integer
    Public t2 As Integer
    Public t3 As Integer
End Class
```

Sie können nun in einer separaten Datei den *New*-Konstruktor für die Klasse *Test* implementieren, indem Sie das Schlüsselwort *Partial* vor die Klassendefinition stellen.

```
Partial Public Class Test
    Sub New(ByVal w1 As Integer, ByVal w2 As Integer, _
        ByVal w3 As Integer)
        t1 = w1
        t2 = w2
        t3 = w3
    End Sub
End Class
```

Der Übersetzer erkennt dann die verteilten Definitionen und fasst diese unter einem Klassennamen zusammen.

Hinweis

Sie finden die Projektdateien eines als Windows-Anwendung erstellten Beispiels im Ordner `\Beisp\Kap07\Beispiel7_17` der Begleit-CD, welches die Verwendung des *Partial*-Schlüsselworts demonstriert.

7.7.3 Klassen als Dateien oder Bibliotheken realisieren

Die bisherigen Beispiele enthielten die Anweisungen der Klasse in der gleichen Datei wie das eigentliche Beispielprogramm. Es kam also immer nur eine Quelldatei innerhalb des Projekts zum Einsatz. Dies ist bei umfangreicheren Projekten aber sehr unübersichtlich. Falls Sie häufiger mit Klassen arbeiten, empfiehlt es sich, den Quellcode auf mehrere Dateien oder gar auf separate Bibliotheken aufzuteilen. Nachfolgend wird skizziert, wie sich so etwas in der Entwicklungsumgebung realisieren lässt.

Den Klassencode in separate Elemente auslagern

Der einfachste Ansatz besteht darin, den Code der Klassen in separate Elemente des Projekts auszulagern. Abbildung 7.6 zeigt den Ausschnitt aus dem Codefenster der Entwicklungsumgebung sowie den Projektmappen-Explorer. Das Projekt weist einmal das Element *Wetter.vb* auf, welches den Code zur Definition der Klasse beinhaltet. Der eigentliche Code des Anwendungsbeispiels wurde dagegen in einer separaten Datei *Test.vb* hinterlegt. Um diese Konstellation zu realisieren, legen Sie ein neues Projekt (z.B. als Konsolenanwendung oder als Windows-Anwendung) an. Anschließend fügen Sie im Projektmappen-Explorer ein weiteres Klassen-Element hinzu (siehe *Kapitel 2*). Im Codefenster dieser Klasse lassen sich dann die Anweisungen zur Definition der betreffenden Klasse hinterlegen. Der Code für die eigentliche Anwendung wird dann in einem separaten Modul- oder Klassenelement hinterlegt.

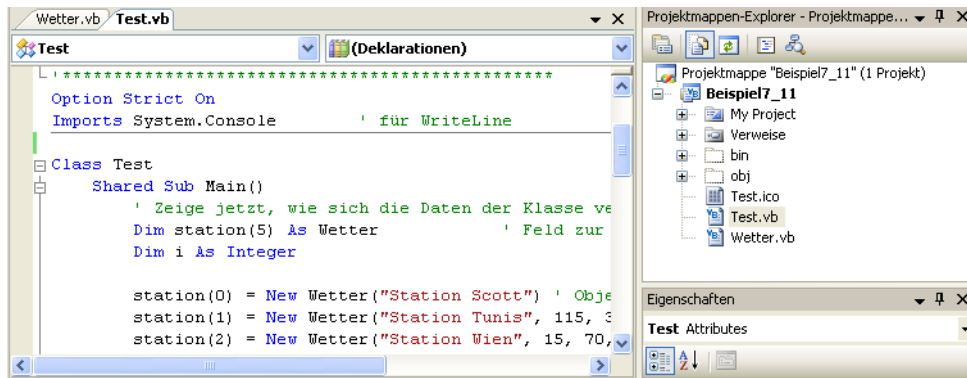


Abbildung 7.6: Projekt mit mehreren Dateien für den Programmcode

Hinweis

Sie finden ein entsprechendes Beispielprojekt im Ordner `\Beisp\Kap07\Beispiel7_11` der Begleit-CD.

Klassen als Bibliotheken realisieren

Noch einen Schritt weiter geht der Ansatz, die Klassen als eigene Bibliotheken zu realisieren. Wird die Klasse in eine eigene *.dll*-Bibliotheksdatei kompiliert, lässt sie sich wie die .NET-Framework-Klassenbibliotheken in die Anwendung einbinden. Die Vorgehensweise soll an einem einfachen Beispiel demonstriert werden. Der Code zur Implementierung der Basisklasse *WetterPrimitiv* und der abgeleiteten Klasse *Wetter* soll in der Datei *Wetter.vb* hinterlegt und in eine Bibliotheksdatei übersetzt werden.

1. Hierzu müssen Sie in der Entwicklungsumgebung ein neues Visual-Basic-Projekt anlegen (siehe *Kapitel 2*). Als Vorlage wählen Sie im Dialogfeld *Neues Projekt* den Eintrag »Klassenbibliothek« und passen ggf. den Namen dieser Klassenbibliothek an (z.B. anstatt »ClassLibrary1« ließe sich der Name »Wetter« verwenden). In Visual Studio 2005 können Sie im Dialogfeld *Neues Projekt* zudem den Pfad angeben, in dem die Projektdateien hinterlegt werden.

2. Kontrollieren Sie in den Projekteigenschaften, dass der Anwendungstyp auf »Klassenbibliothek« gestellt wurde. Zudem können Sie in der Seite *Anwendung* noch den Namen der Assembly und den Namensraum angeben.
3. Tragen Sie den Code für die Klasse im Codefenster des Klassenelements ein. Anschließend starten Sie die Übersetzung des Codes über den Befehl *xxx neu erstellen* des Menüs *Erstellen*.

Mit dem Befehl *xxx neu erstellen* wird das Klassenmodul in eine *DLL*-Bibliotheksdatei übersetzt. Im aktuellen Beispiel wird das Ganze in die Bibliotheksdatei *Wetter.dll* kompiliert. Diese Bibliotheksdatei können Sie direkt weitergeben. Sie lässt sich von Dritten verwenden. Der Zugriff aus einer Anwendung auf die Klasse kann dann über den innerhalb der Assembly spezifizierten Namespace erfolgen.

Bemerkung zur Definition des Namensraums

Der der Klasse im Assembly zugeordnete Namensraum lässt sich auf zwei verschiedene Arten definieren. Einmal lässt sich der Namensraum über das Feld *Stammnamespace* der Seite *Anwendung* der Projekteigenschaften definieren. Alternativ können Sie den Code der Klasse durch *Namespace x ... End Namespace* einfassen. Das folgende Codefragment zeigt die Struktur der betreffenden Quelldatei (aus Platzgründen habe ich die Befehle zur Implementierung der Klassen weggelassen):

```
Namespace MyWetter
    Public Class WetterPrimitiv ' Die Basisklasse
    ... hier steht der Code der Basisklasse
    End Class

    Public Class Wetter
    ... hier steht der Code der abgeleiteten Klasse
    End Class
End Namespace
```

Wird im Quellcode der *Namespace*-Bezeichner gefunden, überschreibt diese Angabe die Projekteigenschaft *Stammnamespace*.

Hinweis

Sie finden die Datei *Wetter.vb* im Ordner *\Beisp\Kap07\Wetter* auf der Begleit-CD. Im Quellcode sind die Anweisungen zur Definition des Namespace auskommentiert, da die globale Definition für den Stammnamespace des Projekts gelten soll. Der Unterordner *bin\Debug* des Projektordners enthält die übersetzte *.dll*-Datei.

Realisieren der Anwendung und Einbinden der Bibliotheksdatei

Liegt die Bibliothek in Form einer *.dll*-Datei vor, können Sie das Projekt schließen und in einem weiteren Schritt die Anwendung implementieren.

1. Legen Sie in der Entwicklungsumgebung ein neues Projekt (als Konsolenanwendung oder als Windows-Anwendung) an. Dieser Schritt entspricht der üblichen Vorgehensweise.
2. Um die Assembly mit der Klassenbibliothek im Projekt einzubinden, müssen Sie anschließend per Kontextmenü (Befehl *Verweis hinzufügen* des Zweigs *Verweise*) einen Verweis auf die *.dll*-Bibliotheksdatei im Projekt anlegen (siehe auch *Kapitel 2*). Die Auswahl der *.dll*-Bibliotheksdatei kann auf der Registerkarte *Durchsuchen* des Dialogfelds *Verweis hinzufügen* erfolgen.
3. Zum Import des Namensraums können Sie entweder die Seite *Verweise* der Projekteigenschaften öffnen und dann das Kontrollkästchen mit dem benötigten Namensraum markieren. Oder Sie fügen im Codefenster der Anwendung eine Anweisung der Art *Import MyWetter* hinzu, wobei *MyWetter* hier der für die Klasse vereinbarte Stammnamespace ist.

Anschließend können Sie im Codefenster des Elements, welches das Startobjekt der Anwendung enthält, den Code der Anwendung eingeben. Das Anwendungsprogramm zur Speicherung und Anzeige der Wetterdaten reduziert sich dann (nach dem Import des Namensraums *MyWetter*) auf den Code zur Deklaration und Instantiierung der Objektvariablen sowie auf die Anweisungen zur Implementierung der Anwendungsfunktion.

Hinweis

Sie finden die Datei *Test.vb* eines als Konsoleanwendung implementierten Beispielprojekts, welches die Bibliothek *Wetter.dll* benutzt, im Ordner *\Beisp\Kap07\Beispiel7_1a* auf der Begleit-CD. Im Projekt sind bereits die erforderlichen Verweise gesetzt. Der Namensraum *MyWetter* wird mit einer *Imports*-Anwendung im Quellcode des Elements importiert. Weitere Details sind dem Quellcode zu entnehmen. Um eine lauffähige *.exe*-Datei zu bekommen, müssen Sie nur das Projekt übersetzen bzw. erstellen lassen. Dabei wird auch die benötigte *.dll*-Datei in den Ordner mit der *.exe*-Anwendungsdatei kopiert. Möchten Sie das Programm weitergeben, ist neben der *.exe*-Datei auch die Bibliotheksdatei *Wetter.dll* mitzuliefern.

Sie kennen nun die wichtigsten Techniken zum Umgang mit Klassen und zum Vererben. Sie können diese Techniken z.B. auf die Basisklassen der .NET-Framework-Klassenbibliothek anwenden, um die betreffenden Klassen in Anwendungen zu nutzen.

Teil 3

Benutzeroberflächen

8

Einfache Interaktionen mit dem Benutzer

Dieses Kapitel widmet sich der Frage, wie in Visual Basic eine Interaktion mit dem Benutzer auf einfachem Wege zu realisieren ist. Der Bogen reicht von der Behandlung der Methoden *WriteLine()* und *ReadLine()* der Klasse *System.Console* in Konsolenanwendungen bis hin zu Ausgaben mit *MsgBox()* (bzw. *MessageBox.Show()*) oder zu Benutzerabfragen mit *InputBox()* in Windows-Anwendungen, die einfache Dialoge aufweisen sollen. Einige der benötigten Methoden wurden bereits in den vorherigen Kapiteln benutzt. In diesem Kapitel möchte ich auf die Details zur Verwendung dieser Funktionen eingehen. Mit diesem Wissen können Sie sehr schnell kleinere .NET-Anwendungen erstellen, die ohne Formulare als Benutzeroberfläche auskommen sollen. Aber auch in .NET-Anwendungen, die über eine Formularoberfläche verfügen, sind gelegentlich Dialogfelder oder Eingabedialoge hilfreich, um Ergebnisse anzuzeigen oder einen Wert vom Benutzer abzufragen.

8.1 Ein-/Ausgaben auf Konsoleebene

Das .NET Framework ist vom Ansatz her nicht unbedingt auf Windows als Betriebssystem begrenzt. Die Kommunikation mit dem Benutzer muss daher nicht zwingend über Formulare erfolgen. Viele Systeme verfügen neben der grafischen Benutzeroberfläche über eine einfache Konsole, auf der Ein-/Ausgaben erfolgen können. Eine .NET-Anwendung kann dann auf vielen Plattformen laufen, ohne dass die unterschiedlichen Benutzeroberflächen zu berücksichtigen wären (es muss lediglich eine Unterstützung für die Ein-/Ausgabeklassen des .NET Framework vorhanden sein). Die .NET-Framework-Klassenbibliothek bietet daher im Namensraum *System* die Klasse *Console*. Um eine Ausgabe auf Konsolenebene vorzunehmen, importieren Sie die Klasse *System.Console* in einer *Imports*-Anweisung und rufen Sie dann die gewünschte Methode auf.

```
Imports System.Console
...
Class Test
    Shared Sub Main()
        Write ("Ein Test")
        WriteLine (" mit Zeilenwechsel")
    End Sub
End Class
```

Die betreffende Methode erwartet einen oder mehrere Parameter mit den auszugebenden Informationen (auf diesen Aspekt komme ich gleich zurück). Bei der *Write()*-Methode wird nur die als Parameter übergebene Information (ohne Zeilenwechsel) auf

der Konsolebene ausgegeben. *WriteLine()* gibt den gewünschten Inhalt als Text aus und erzeugt anschließend einen zusätzlichen Zeilenumbruch. Bei dem obigen Codefragment geben die Methodenaufrufe jeweils Zeichenketten aus.

Schauen wir uns einmal genauer an, wie sich Werte über diese beiden Methoden ausgeben lassen. Als Ausgangslage seien in einem Programm die folgenden Variablen deklariert:

```
Dim Name() As String = {"Bach", "Born", "Gruber"}
Dim i As Integer = 100
Dim Zahl As Double = 23.45
Dim Preis As Decimal = 12.45d
```

Bei der Variablen *Name* handelt es sich um ein Feld vom Typ *String*, welches mit drei Namen initialisiert wurde. Um die Werte der Feldelemente auf der Konsole auszugeben, ließe sich folgende Anweisungsfolge verwenden:

```
Write("Namen: ")
Write(Name(0))
Write(", ")
Write(Name(1))
Write(", ")
WriteLine(Name(2))
```

Jede dieser Anweisungen übergibt der *Write()*-Methode eine Zeichenkette als Parameter. In der letzten Anweisung habe ich die *WriteLine()*-Methode benutzt, um einen Zeilenumbruch auszulösen. Der obige Ansatz ist aber recht aufwändig. Einfacher ist es, die betreffenden Zeichenketten bei der Ausgabe miteinander zu verknüpfen und den Ausdruck als einen Parameter an die gewünschte Methode zu übergeben. Dies könnte dann so aussehen:

```
Write("Namen: " & Name(0) & " " & Name(1) & " " & Name(2))
WriteLine()
```

Die Zeichenkettenverknüpfung kann mit dem *&*-Operator erfolgen. Da das Feld *Name* vom Typ *String* ist, klappt das ohne Probleme, selbst wenn *Option Strict On* im Programmkopf vereinbart wurde. Der Aufruf der *WriteLine()*-Methode ohne Parameter erzeugt übrigens nur einen Zeilenwechsel – folgt der Aufruf auf die Ausgabe einer *WriteLine()*-Methode, ergibt sich eine Leerzeile.

Die Methoden *Write()* und *WriteLine()* des .NET Framework sind zwar mehrfach überladen und akzeptieren jeden beliebigen Datentyp als Argument. In diesem Fall darf aber nur ein Parameter übergeben werden. Um die weiter oben vereinbarten Ganz- und Gleichkommazahlen auszugeben, ließe sich folgende Sequenz an *Write()*-Aufrufen verwenden:

```
Write ("i: ")
WriteLine (i)
Write ("Zahl: ")
```

```
WriteLine (Zahl)
Write ("Preis: ")
WriteLine (Preis)
```

Die Sequenz gibt die drei Werte in getrennten Zeilen aus. Der Textvorspann wird durch einen eigenen Methodenaufruf über *Write()* generiert. Um den Programmcode kompakter zu gestalten, sollte die Ausgabe aber in einer Zeile erfolgen. In diesem Fall kann alles in eine Zeichenkette konvertiert werden. Dies könnte dann so aussehen:

```
WriteLine ("i: " & i.ToString & " Zahl: " & _
        Zahl.ToString & " Preis: " & Preis.ToString)
```

Die Teilausdrücke werden durch den *&*-Operator zu einer Zeichenkette verknüpft. Ist *Option Strict On* vereinbart, verweigert der Compiler eine implizite Typ-Konvertierung. Daher muss bei jeder numerischen Variable die *.ToString()*-Methode angewandt werden. Die Methode bezieht sich auf den aktuellen Variablenwert und wandelt diesen in eine Zeichenkette um. Die *ReadLine*-Anweisung sorgt im nachfolgenden Beispiel lediglich dafür, dass das Programm bis zum Drücken der -Taste wartet (Abbildung 8.1). Dies verhindert beim Aufruf der *.exe*-Datei per Doppelklick aus einem Ordnerfenster, dass sich das Fenster der Eingabeaufforderung nach der Ausgabe sofort schließt. Das folgende Listing zeigt den gesamten Quellcode des Beispiels:

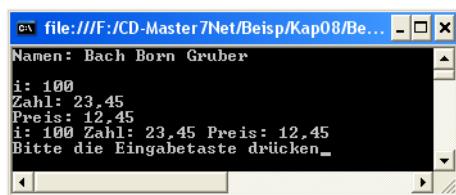


Abbildung 8.1: Text- und Zahlenausgaben auf Konsoleebene

```
'*****
' File/Projekt: Beispiel8_01
' Autor: G. Born www.borncity.de
' Demonstriert die Möglichkeiten zur Textausgabe
' auf der Konsole mittels Write() und WriteLine().
'*****

Option Strict On
Imports System.Console      ' für WriteLine()

Module Test
    Sub Main()
        Dim Name() As String = {"Bach", "Born", "Gruber"}
        Dim i As Integer = 100
```

Listing 8.1: Ausgaben auf Konsoleebene

```
Dim Zahl As Double = 23.45
Dim Preis As Decimal = 12.45d

Write("Namen: " & Name(0) & " " & Name(1) & " " & Name(2))
WriteLine()           ' Zeilenumbruch
WriteLine()           ' Zeilenumbruch -> Leerzeile

' Jetzt Zahlen ausgeben
Write ("i: ")
WriteLine (i)
Write ("Zahl: ")
WriteLine (Zahl)
Write ("Preis: ")
WriteLine (Preis)

' jetzt alles in einer Zeile
WriteLine ("i: " & i.ToString & " Zahl: " & _
          Zahl.ToString & " Preis: " & Preis.ToString)

Write ("Bitte die Eingabetaste drücken")
ReadLine
End Sub
End Module
```

Listing 8.1: Ausgaben auf Konsoleebene (Forts.)

Hinweis

Sie finden die Projektdateien des als Konsolenanwendung realisierten Beispiels im Ordner `\Beisp\Kap08\Beispiel8_01` der Begleit-CD.

8.1.1 Formatierung der Konsoleausgaben

Das Beispiel im vorherigen Abschnitt befasste sich lediglich mit der Ausgabe einfacher Werte. In der täglichen Praxis stehen Sie aber häufig vor der Aufgabe, die Ausgabe in irgendeiner Art zu formatieren. Dies kann vom einfachen Zeilenumbruch über die Verwendung von Tabulatoren bis hin zur Festlegung der Anzahl der Vor- und Nachkommastellen bei Zahlenausgaben reichen. Die Techniken sollten bei der `Write()`- und `WriteLine()`-Methode einsetzbar sein (eine Verwendung ist eingeschränkt auch bei der weiter unten behandelten `MsgBox()`-Methode und der `MessageBox`-Klasse möglich). Die Ansätze haben Sie bereits in den Beispielen der vorhergehenden Kapitel kennen gelernt. An dieser Stelle möchte ich noch etwas Systematik in die Angelegenheit bringen.

Ein schönes Feature der Methoden *Write()* und *WriteLine()* besteht darin, dass eine Variante der überladenen Methoden als ersten Parameter eine Format-Zeichenkette erlaubt. Die Format-Zeichenkette darf dabei Platzhalter der Art {0}, {1} etc. aufweisen. Die Methode wertet dann den Platzhalter aus und setzt die Werte der Folgeparameter an Stelle der Platzhalter im Ausgabetext ein. Kommen die beiden Platzhalter {0} und {1} im Format-String vor, lassen sich zwei Parameter mit Ausgabewerten beim Methodenaufruf übergeben. Das Elegante an dieser Lösung ist dabei die Tatsache, dass die Parameter Werte mit beliebigen Typen aufweisen können. Die Methode übernimmt selbstständig die Konvertierung in eine Zeichenkette und fügt die Ergebnisse an Stelle der Platzhalter im Ausgabetext ein. Die folgende Anweisung weist drei Platzhalter im Format-String auf. Als zweiter Parameter wird ein Feld *Name* übergeben, welches drei Feldelemente enthält.

```
WriteLine ("Namen {0}, {1}, {2}", Name)
```

Enthält das Feld mehr Elemente, als Platzhalter im Format-String angegeben sind, gibt die Methode die restlichen Werte einfach nicht aus. Um gezielt die Werte der Feldelemente auszugeben, können Sie diese natürlich beim Methodenaufruf angeben. Die Parameter sind dabei durch Kommas zu trennen. Der Aufruf zur Ausgabe dreier Werte sieht dann beim Feld *Name* so aus:

```
WriteLine("Namen {0}, {1}, {2}", Name(0), Name(1), Name(2))
```

Ein Zeilenumbruch wird automatisch bei Verwendung der *WriteLine()*-Methode am Ende der Ausgabezeile eingefügt. Möchten Sie Zeilenumbrüche innerhalb der Ausgabezeile an beliebiger Stelle einer Zeichenkette erzwingen, müssen Sie die Steuerzeichen für den Zeilenumbruch einfügen. Visual Basic 2005 bietet Ihnen mehrere Varianten an. Wer mit älteren Versionen von Visual Basic, mit VBA oder mit VBScript gearbeitet hat, kennt die benannte Konstante *vbCrLf*, die für die Zeichenfolge »Carriage-Return/Line-Feed« (also Wagenrücklauf und Zeilenvorschub) steht. Die Visual-Basic-Konstanten werden in Visual Basic 2005 in der Kompatibilitätsklasse *Microsoft.VisualBasic* bereitgestellt. Sie können in Visual Basic 2005 sogar auf den Import des betreffenden Namensraums verzichten, da dieser automatisch im Projekt importiert wird. Konstante wie *vbCrLf* oder auch *vbTab* (für ein Tabulatorzeichen) lassen sich daher ebenfalls im Programm verwenden. Die folgende Anweisung trennt die Werte der Ausgabe durch Tabulatorzeichen und löst einen zusätzlichen Zeilenumbruch aus:

```
Write ("Namen{0}{1}{2}{3}{4}{5}{6}", _  
      vbTab, Name(0), vbTab, Name(1), _  
      vbTab, Name(2), vbCrLf)
```

Beachten Sie, dass ich hier die benannten Konstanten einfach als Parameter verwende, die dann von der Methode in die korrespondierenden Platzhalter eingesetzt werden.

Ein plattformunabhängiges Zeichen für den Zeilenumbruch ist in der Klasse *System.Environment* über die Eigenschaft *newline* vereinbart. Die folgenden Anweisungen bewirken einen Zeilenumbruch hinter jedem ausgegebenen Parameter:


```
Imports System.Environment
...
WriteLine("Namen"& newline & Name(0) & newline & _
        Name(1) & newline & Name(2))
```

Ob Sie also mit benannten Konstanten wie *vbCrLf* der Visual-Basic-Kompatibilitätsklasse arbeiten oder im Hinblick auf die Kompatibilität mit anderen Umgebungen auf *newline* zurückgreifen, bleibt Ihnen überlassen.

Tipp

Sind Sie etwas schreibfaul und bevorzugen Sie anstelle von *newline* die kürzere benannte Konstante *vbCrLf*? Dann greifen Sie zu einem Trick und definieren sich eigene Variablen wie *cr* oder *cr2x* (für zwei Zeilenwechsel), indem Sie einer *String*-Variablen den Wert der Eigenschaft *newline* zuweisen. Dies wird im nachfolgend gezeigten Listing demonstriert.

Eine offene Frage betrifft die Ausgabe numerischer Werte. Bei Ganzzahlen ist es klar, dass diese mit allen Ziffern erscheinen. Dezimalzahlen sind aber nicht so ganz eindeutig darstellbar. Die Angabe 23,00 entspricht vom Wert her auch der Angabe 23. Gerade bei Währungsangaben ist es aber vorteilhaft, wenn die Zahl der Nachkommastellen vorgegeben werden kann. In Abbildung 8.2 sehen Sie die Ausgaben des Beispielsprogramms. In der mit »5:« beginnenden Zeile sind die numerischen Werte in der Standarddarstellung zu sehen, d.h., die benutzte Methode bestimmt die Zahl der Vor- und Nachkommastellen. Dies führt dazu, dass der mit 23,00 vereinbarte Wert nur als 23 angezeigt wird. In der mit »6:« beginnenden Zeile werden dagegen immer zwei Nachkommastellen angezeigt. Der Wert *Preis* weist sogar eine führende Null auf.

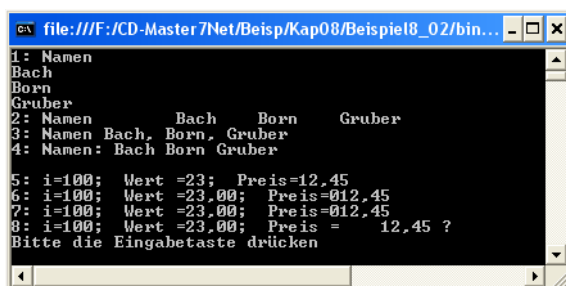


Abbildung 8.2: Formatierte Ausgaben auf Konsoleebene

.NET Framework erlaubt Zeichenketten mit Formatzeichenfolgen (Formatbezeichner). Mit Hilfe dieser Formatbezeichner lässt sich jeder Basisdatentyp (*String*, *Integer*, *Byte*, *Double* etc.) als eine Zeichenfolge darstellen. Das .NET Framework unterstützt dabei Standardformatbezeichner und benutzerdefinierte Formatbezeichner. Die nachfolgende Tabelle enthält eine Übersicht über die Standardformatbezeichner. Diese bestehen aus einem Buchstaben, wobei nicht nach Groß-/Kleinschreibung unterschieden wird.

Formatbezeichner	Bedeutung
C	Währungsformat für die Konvertierung eines Werts in eine Zeichenfolge. Die Zahl der Nachkommastellen, das Währungszeichen etc. werden über das <i>NumberFormatInfo</i> -Objekt gesteuert. Bei fehlender Genauigkeitsangabe wird die durch die <i>NumberFormatInfo</i> -Eigenschaft angegebene Standardgenauigkeit für Währungen verwendet.
D	Dezimalformat für ganzzahlige Typen, die aus den Ziffern 0 bis 9 bestehen und bei negativen Werten ein vorangestelltes Minuszeichen aufweisen
E	Exponentialformat für Zahlen, die in Zeichenketten der Form »d.dddE+ddd« konvertiert werden. Das Zeichen <i>d</i> steht hier für Ziffern, und die Zahl kann ein vorangestelltes Minuszeichen aufweisen. Das Gleiche gilt für den Exponenten, der hinter dem Zeichen E folgt und das Vorzeichen sowie mindestens drei Ziffern umfasst.
F	Festkommaformat für in Zeichenketten zu konvertierende Zahlen in der Form »d.ddd«. Ein vorangestelltes Minuszeichen ist zulässig. Die Genauigkeit wird über das <i>NumberFormatInfo</i> -Objekt gesteuert.
G	Kompakte Dezimalzahl unter Verwendung der Festkomma- oder der wissenschaftlichen Notation. Die Zahl der signifikanten Ziffern wird durch die Genauigkeitsangabe oder durch den Typ der zu konvertierenden Zahl bestimmt. Nullen nach dem Dezimaltrennzeichen werden entfernt.
N	Zahlen mit numerischem Format, die ein Vorzeichen und Nachkommastellen aufweisen (»d.ddd.ddd«)
P	Prozentformat, sorgt für die Konvertierung der Zahl in einen Prozentwert

Tabelle 8.1: Standardformatbezeichner

Hinweis

Zusätzlich gibt es noch den Standardformatbezeichner *R*, der dafür sorgt, dass die Zeichenkette wieder in den gleichen numerischen Wert zurückkonvertiert werden kann. Mit dem Standardformatbezeichner *X* erreichen Sie eine Anzeige einer Zahl in Hexadezimalziffern. Weitere Details zu den Standardformatbezeichnern finden Sie in der Hilfe zum .NET Framework.

Sofern Ihnen diese Standardformatbezeichner nicht ausreichen, bietet .NET auch benutzerdefinierte Formatbezeichner.

- 0: Platzhalter für eine Ziffer, die auf jeden Fall anzugeben ist. Existieren keine signifikanten Stellen, werden Nullen eingesetzt (z.B. 00.00 erzeugt bei 1,10 den Wert 01,10).
- #: Platzhalter für eine Ziffer. Enthält die auszugebende Zahl keine signifikante Ziffer für die betreffende Stelle, wird nichts angegeben (z.B. #0.00 erzeugt bei 1,10 den Wert 1,10).

Der Punkt wird dabei als Platzhalter für den Dezimalpunkt benutzt, während als Tausendertrennzeichen ein Komma einzusetzen ist. Ein Prozentzeichen signalisiert die Formatierung als Prozentzahl. Details finden Sie in der Hilfe zum .NET Framework.

Formatzeichenfolgen lassen sich dabei in dem im ersten Parameter übergebenen String in der Form {x:yyy} hinterlegen oder als Argument der *ToString()*-Methode übergeben. Weiterhin erlaubt das *Format*-Objekt die Verwendung solcher Formatzeichenfolgen. Die folgende Anweisung benutzt Formatzeichenfolgen, um die Zahlen mit definierter Stellenzahl anzuzeigen.

```
WriteLine ("6: i={0}; Wert ={1}; Preis={2:#000.00}", _
    i, Wert, Preis)
```

Die Formatangabe im Platzhalter {1} gibt eine Zahl mit mindestens einer Vorkommate und zwei Nachkommastellen an. Der Platzhalter {2} verwendet eine Formatangabe, die drei signifikante Stellen vor dem Komma und zwei Nachkommastellen vorschreibt (Abbildung 8.2). Die obige Zeile hätte sich auch unter Verwendung der *ToString()*-Methode folgendermaßen schreiben lassen:

```
WriteLine ("6: i={0}; Wert ={1:##0.00}; Preis={2}", _
    i, Wert.ToString("##0.00"), Preis.ToString("#000.00"))
```

Die dritte Variante benutzt die *Format()*-Methode der Klasse *String*. Die folgende Anweisung wandelt eine Zahl in eine formatierte Zeichenkette um und weist diese einer Variablen zu:

```
Dim Preis As Decimal = 12.45d
Dim sPreis As String = _
    String.Format("Preis = {0,10:C}", Preis)
```

In der Formatangabe kommt einmal der Standardformatbezeichner »C« vor, der eine Währungsformatierung vornimmt. Dann ergänzt .NET Framework die betreffende Zeichenkette mit dem Währungszeichen.

Achtung

In Abbildung 8.2 sehen Sie in Zeile 8 ein Fragezeichen hinter der letzten Zahl. Eigentlich sollte dort das Eurozeichen stehen. Leider unterstützt der Zeichensatz der Eingabeaufforderung in Windows 2000/XP dieses Zeichen nicht. Bei der Ausgabe des formatierten Werts unter Windows (z.B. mittels der nachfolgend besprochenen *MsgBox()*-Methode) erscheint dagegen das Eurozeichen.

Die Angabe »{0,10:C}« verbirgt noch eine zweite Besonderheit. Die erste Ziffer 0 steht hier für den Platzhalter des ersten Parameters (also so was wie die bekannte Angabe {0}). Dann folgt, getrennt durch ein Komma, die Zahl 10. Erst daran schließt sich, getrennt durch einen Doppelpunkt, der Formatierungsbezeichner an. Die 10 legt die Stellenzahl, die in der Ausgabe für den Wert vorgesehen ist, auf zehn Zeichen fest. Sie können also auf recht einfache Weise Listen erzeugen, indem Sie für die Werte feste Stellenzahlen vorgeben. Die *Format()*-Funktion ergänzt die fehlenden Stellen notfalls durch Leerzeichen. Das folgende Listing zeigt den gesamten Quellcode des Beispiels:

```
'*****
' File/Projekt: Beispiel8_02
' Autor: G. Born www.borncity.de
' Demonstriert die Möglichkeiten zur formatierten Textausgabe
' auf der Console-Ebene mittels Write() und WriteLine().
'*****

Option Strict On
Imports System.Environment      ' für newline-Eigenschaft
Imports System.Console         ' für WriteLine()

Class Test
  Shared Sub Main()
    Dim cr2x As String = newline & newline

    Dim Name() As String = _
      {"Bach", "Born", "Gruber", "Pille"}

    Dim i As Integer = 100
    Dim Wert As Double = 23.00
    Dim Preis As Decimal = 12.45d

    ' konvertiere mit Mindeststellenzahl in sPreis
    Dim sPreis As String = _
      String.Format("Preis = {0,10:C}", Preis)

    WriteLine("1: Namen"& newline & _
      Name(0) & newline & _
      Name(1) & newline & _
      Name(2))

    Write ("2: Namen{0}{1}{2}{3}{4}{5}{6}", _
      vbTab, Name(0), vbTab, Name(1), _
      vbTab, Name(2), vbCrLf)

    WriteLine ("3: Namen {0}, {1}, {2}", Name)

    Write("4: Namen: " & Name(0) & " " & Name(1) & " " & Name(2) & cr2x)

    WriteLine ("5: i={0}; Wert ={1}; Preis={2}", i, Wert, Preis)
```

Listing 8.2: Beispiel mit formatierten Ausgaben auf Konsoleebene

```

WriteLine ("6: i={0}; Wert ={1:##0.00}; " & _
          " Preis={2:##000.00}", i, Wert, Preis)

' oder alternative Schreibweise mit ToString()
WriteLine ("7: i={0}; Wert ={1:##0.00}; Preis={2}", _
          i, Wert.ToString("##0.00"), Preis.ToString("##000.00"))

' verwende Mindeststellenzahl in sPreis
WriteLine ("8: i={0}; Wert ={1:##0.00}; {2}", i, Wert, sPreis)

Write ("Bitte die Eingabetaste drücken")
ReadLine
End Sub
End Class

```

Listing 8.2: Beispiel mit formatierten Ausgaben auf Konsoleebene (Forts.)

Hinweis

Sie finden die Projektdateien des als Konsoleanwendung implementierten Beispiels im Ordner `\Beisp\Kap08\Beispiel8_02` der Begleit-CD.

Farben bei Konsoleausgaben

In .NET Framework 2.0 wurde als Neuerung die Möglichkeit eingeführt, bestimmte Eigenschaften des Konsolfensters zu beeinflussen. Sie können Textausgaben auf der Konsole nicht nur formatieren, sondern auch farbig auszeichnen. Die folgende Code-Sequenz setzt die Vorder- und Hintergrundfarben sowie die Höhe (Zeilenzahl) und Breite (Zeichenzahl) des Fensters der Eingabeaufforderung.

```

Console.BackgroundColor = ConsoleColor.White
Console.ForegroundColor = ConsoleColor.Blue
Console.WindowHeight = 10 ' 10 Zeilen
Console.WindowWidth = 80 ' 80 Zeichen

```

Diese Eigenschaften werden im folgenden Beispiel (Abbildung 8.3) verwendet, um die Ausgaben farbig hervorzuheben.

8.1.2 Benutzereingaben auf Konsoleebene

Benutzereingaben lassen sich auf Konsoleebene über die beiden Methoden `Read()` und `ReadLine()` abfragen. `Read()` liest ein Zeichen und gibt dieses zurück, während `ReadLine()` alle eingegebenen Zeichen, allerdings ohne den Zeilenvorschub, zurückgibt.

Das nachfolgende Beispiel zeigt, wie sich eine Benutzereingabe von der Konsole einlesen und in einer Variablen speichern lässt. Das Programm fordert den Benutzer auf, eine

Zahl einzugeben (Abbildung 8.3). Diese Eingabe muss mittels der -Taste bestätigt werden. Dann gibt das Programm die eingegebene Zahl zur Kontrolle wieder im Fenster der Konsole aus. Die Textausgaben werden dabei mit einer Textfarbe und einer Hintergrundfarbe versehen.

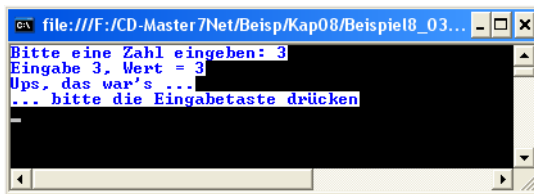


Abbildung 8.3: Benutzereingabe auf Konsoleebene

In diesem Beispiel kommt die *ReadLine()*-Methode der Klasse *Console* zum Einsatz, die eine komplette Eingabe von der Konsole liest. Die Methode wartet nach dem Aufruf, bis der Benutzer die Eingabe mittels der -Taste bestätigt hat. Anschließend wird diese Eingabe als Zeichenkette zurückgegeben und lässt sich einer *String*-Variablen zuweisen. Um die Methode nutzen zu können, muss daher, wie bei der *WriteLine()*-Methode, der Zugriff auf die Klasse über die *Imports*-Anweisung vereinbart werden:

```
Imports System.Console
```

Anschließend reicht es, die Methode aufzurufen und das Ergebnis einer *String*-Variablen zuzuweisen. Die folgenden Anweisungen zeigen, wie sich die Benutzereingabe einlesen lässt:

```
Dim tmp As String
...
tmp = ReadLine()
```

Sobald die Anweisung mit dem *ReadLine()*-Aufruf ausgeführt wurde, enthält die Variable *tmp* den Rückgabewert.

Nun kann es aber sein, dass der Benutzer nichts eingibt, sondern einfach die -Taste drückt. Die Methode gibt dann eine leere Zeichenkette zurück. Falls Sie eine Eingabe erwarten, sollte dieser Fall im Programm abgefangen werden. Dies kann beispielsweise mit folgender Anweisung geschehen:

```
If tmp = "" Then ...
```

Das aktuelle Beispiel fordert vom Benutzer eine Zahl an, die dieser eingeben soll. Daher muss die Eingabe beim Zuweisen an eine numerische Variable konvertiert werden. Ist *Option Strict On* gesetzt, muss diese Konvertierung explizit erfolgen:

```
Wert = CDec(tmp)
```

Die Konvertierfunktion *Cdec()* wandelt das Argument in den Datentyp *Decimal* um. Je nach Datentyp müssen Sie eine der verfügbaren Konvertierfunktionen verwenden. Allerdings ist noch etwas zu beachten. Was passiert denn, wenn der Benutzer statt einer Zahl so etwas wie »123 – ein Text« eingibt? Versuchen Sie diesen Ausdruck in eine Zahl zu konvertieren, löst dies einen Laufzeitfehler aus. Wie Sie auf solche Laufzeitfehler reagieren, erfahren Sie weiter unten. Besser ist es, wenn Sie den Eingabewert vor dem Konvertieren darauf überprüfen, ob dieser als gültige Zahl erkannt wird. Dies lässt sich mit der Anweisung

```
If IsNumeric(tmp) Then
```

erledigen. Die Visual-Basic-Anweisung *IsNumeric* prüft lediglich, ob der übergebene Parameter gültige Zeichen für Zahlen aufweist.

Das nachfolgende Listing enthält alle Anweisungen des Beispiels. An dieser Stelle möchte ich noch auf zwei Besonderheiten hinweisen, die Ihnen vielleicht auffallen könnten. Die *ReadLine()*-Methode liefert einen Rückgabewert zurück. Normalerweise steht daher der Methodenaufruf auf der rechten Seite einer Zuweisung oder zumindest in einem Ausdruck. In vielen bisherigen Beispielen finden Sie einfach die Anweisung *ReadLine* im Programmcode, ohne dass deren Rückgabewert ausgewertet wird. Diese Anweisung hat einfach die Aufgabe, das Programm solange anzuhalten, bis der Benutzer die -Taste drückt. Wurde das Programm per Doppelklick aus einem Ordnerfenster aufgerufen, verhindert die *ReadLine*-Anweisung also das automatische Schließen des Fensters, der Benutzer kann die Konsoleausgaben noch in Ruhe studieren.

Hinweis

Interessant ist, dass der Visual-Basic-2005-Compiler es toleriert, dass die leere Klammer für die Argumente der Methode weggelassen werden kann und keine Zuweisung des Rückgabewertes erfolgen muss.

Weiterhin habe ich zu Demonstrationszwecken im Programm die *Exit Sub*-Anweisung verwendet, um das Programm vorzeitig zu beenden. Nach einer Leereingabe endet das Programm ohne weitere Benutzerabfrage. Geben Sie dagegen etwas ein, wartet das Programm am Ende auf das Betätigen der -Taste. Weitere Details können Sie dem folgenden Listing entnehmen:

```
*****
' File/Projekt: Beispiel08_03
' Autor: G. Born www.borncity.de
' Demonstriert das Einlesen einer Benutzereingabe
' auf der Konsole mittels ReadLine. Demonstriert
' auch die neuen Eigenschaften für die Konsole.
*****
Option Strict On
```

Listing 8.3: Eingaben auf Konsoleebene

```
Imports System.Console          ' für Readline/Writeline

Class Test
    Shared Sub Main()
        Dim tmp As String        ' für Eingaben
        Dim Wert As Decimal      ' Ergebnis

        ' Farbe im Konsolefenster setzen
        Console.BackgroundColor = ConsoleColor.White
        Console.ForegroundColor = ConsoleColor.Blue
        Console.WindowHeight = 10 ' 10 Zeilen
        Console.WindowWidth = 80  ' 80 Zeichen

        ' Benutzermeldung mit Einlesen der Eingabe

        Write("Bitte eine Zahl eingeben: ")
        tmp = ReadLine()
        If tmp <> "" Then          ' nur Eingabetaste?
            ' Benutzer hat etwas eingegeben
            If IsNumeric(tmp) Then ' eine gültige Zahl?
                Wert = CDec(tmp)   ' String in Dezimalzahl
                WriteLine("Eingabe {0}, Wert = {0:###0.00}", _
                    tmp, Wert)     ' Daten ausgeben
            Else ' Fehlermeldung
                WriteLine(" *** Fehleingabe {0} ***", tmp)
            End If
        Else
            WriteLine("Leereingabe, Programm endet!")
            Exit Sub              ' Programm beenden
        End If
        WriteLine("Ups, das war's ...")
        WriteLine("... bitte die Eingabetaste drücken")
        ReadLine()
    End Sub
End Class
```

Listing 8.3: Eingaben auf Konsoleebene (Forts.)

Hinweis

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap08\Beispiel8_03` der Begleit-CD.

Spezialfall: Eingaben mit Read() übernehmen

Die Klasse *Console* stellt zusätzlich die Methode *Read()* zur Verfügung, die laut der Hilfe jeweils ein Zeichen von der Eingabeebene lesen und zurückgeben soll. Im Gegensatz zur *ReadLine()*-Methode, die eine Zeichenkette zurückgibt, liefert die Methode *Read()* den Code des Zeichens als *Integer*-Wert zurück. Ist kein Zeichen mehr verfügbar, gibt die Methode den Wert *-1* zurück. Auf den ersten Blick scheint dies der Stein der Weisen zu sein, um die Tastatur abzufragen. Allerdings gibt es einige Hürden, die ich an folgendem kleinen Beispiel skizzieren möchte. Die Abbildung 8.4 zeigt die Ein-/Ausgaben des Programms auf Konsoleebene.

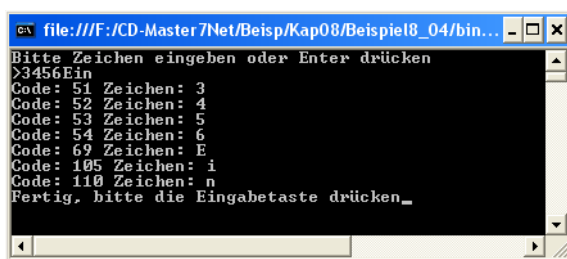


Abbildung 8.4: Benutzereingabe auf Konsoleebene

Das Programm fordert den Benutzer auf, einige Zeichen per Tastatur einzugeben. Der Programmablauf wird dabei so lange gestoppt, bis der Benutzer die Eingabe mittels der -Taste abschließt. Die Methode wartet also auf einen Zeilenabschluss, auch wenn bereits Zeichen im Eingabepuffer vorliegen. Ein Test, ob eine Taste gedrückt wurde, ist also nicht möglich!

Erst nachdem die -Taste gedrückt wurde, liest die Methode das erste Zeichen im Eingabestrom und liefert dessen Code zurück. Dieser Code lässt sich dann über die Visual-Basic-Funktionen *Chr()* bzw. *ChrW()* in ein Zeichen konvertieren. Die Funktion *ChrW()* berücksichtigt dabei die Unicode-Codierung, deren Zeichensätze auch zwei Byte umfassen können (z.B. bei asiatischen Sprachen).

Achtung

Sobald alle Zeichen eingelesen wurden, soll laut Hilfe der Rückgabewert *-1* beim *Read()*-Methodenaufruf auftreten. Beachten Sie aber, dass das hierzu benötigte »Dateiendezeichen« nur im Eingabepuffer auftritt, wenn der Benutzer die Tastenkombination + (oder die Funktionstaste) drückt.

Um das Ende der Eingabezeile zu erkennen, lässt sich der Rückgabewert auf die Codes 10 (LF für das Linefeed-Zeichen) und 13 (CR für das Wagenrücklauf-Zeichen) überprüfen. Das folgende Listing demonstriert dies, indem es eine Benutzereingabe von der Konsoleebene einliest, die einzelnen Eingabecodes samt dem konvertierten Zeichen ausgibt und beim Endezeichen CR abbricht. Weitere Details können Sie dem folgenden Listing entnehmen:

```
'*****
' File/Projekt: Beispiel8_04
' Autor: G. Born www.borncity.de
' Demonstriert das Einlesen einzelner Zeichen
' auf der Console-Ebene mittels Read.
'*****

Option Strict On
Imports System.Console      ' für Read/Writeline

Class Test
  Shared Sub Main()
    Dim i As Integer         ' Rückgabewert Read
    Dim c As Char            ' gewandeltes Zeichen

    WriteLine("Bitte Zeichen eingeben oder Enter drücken")
    Write(">")
    While True               ' Endlosschleife zur Eingabe
      i = Read()             ' lese ein Zeichen
      If i = 13 Then         ' CR erkannt, Einlesen beenden!
        i = Read()          ' lese LF-Zeichen
        Exit While          ' Schleife verlassen
      End If
    ' Konvertiere Code in ein Zeichen und zeige Ergebnisse an
      c = Chr(i) ' oder c = ChrW(i) für Unicode
      WriteLine("Code: {0} Zeichen: {1}", i, c)
    End While

    Write ("Fertig, bitte die Eingabetaste drücken")
    Read                    ' warten auf Benutzereingabe
  End Sub
End Class
```

Listing 8.4: Eingaben auf Konsoleebene zeichenweise lesen

Hinweis

Sie finden die Projektdateien des als Konsolenanwendung implementierten Beispiels im Ordner `\Beisp\Kap08\Beispiel8_04` der Begleit-CD.

8.2 Windows-Dialoge nutzen

Neben den Konsolenanwendungen unterstützt das .NET Framework auch die Ausgaben von Windows-Dialogen. Auch wenn .NET-Anwendungen für Windows standardmäßig mit Formularen zur Benutzerkommunikation versehen werden, sind einfache Standard-dialoge manchmal ganz hilfreich. .NET Framework stellt Visual Basic 2005 für diesen Zweck einen Satz an Methoden zur Verfügung. Wenn Sie dann noch auf die in *Kapitel 2* vorgestellte Projektvorlage für Windows-Anwendungen ohne Formular zurückgreifen, lassen sich kleinere Programme mit wenig Aufwand realisieren. Nachfolgend möchte ich auf diese Thematik eingehen.

Achtung

Visual Basic 2005 bietet für diesen Zweck die Methode *MsgBox()* der Visual Basic-Kompatibilitätsklasse sowie die Klasse *MessageBox* des Namensraums *System.Windows.Forms*. Die Hilfe des .NET Framework empfiehlt, auf die *MsgBox()*-Methode zugunsten von *MessageBox* zu verzichten. Gerade bei der Umstellung von älteren Anwendungen dürften Sie meines Erachtens nicht um *MsgBox()* herumkommen. Aus diesem Grunde stelle ich nachfolgend beide Ansätze vor und überlasse Ihnen die Entscheidung, welche Methode Sie bevorzugen.

8.2.1 Einfache Ausgaben mit MsgBox()

In den vorhergehenden Kapiteln habe ich in einigen Beispielen bei Ausgaben auf die Visual-Basic-Methode *MsgBox()* zurückgegriffen. Diese erlaubt den Aufruf einfacher Dialogfelder zur Anzeige von Benutzermeldungen (Abbildung 8.5). An dieser Stelle möchte ich einige zusätzliche Erläuterungen geben, um Ihnen den sicheren Umgang mit *MsgBox()* zu erleichtern.

Hinweis

Umsteiger von VBScript, VBA oder älteren Visual-Basic-Versionen kennen *MsgBox()* als Funktion bzw. Prozedur der Sprache. In Visual Basic 2005 hat Microsoft die *MsgBox()*-Methode im Rahmen der Visual Basic-Kompatibilitätsklasse bereitgestellt.

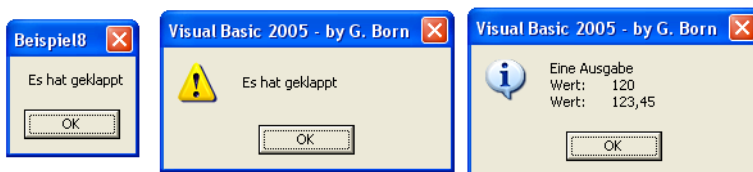


Abbildung 8.5: Einfache Meldungsfelder

Um ein einfaches Dialogfeld mit einer Benutzermeldung anzuzeigen, brauchen Sie nur die *MsgBox()*-Methode aufzurufen (der Namensraum *Microsoft.VisualBasic* wird in Visual Basic 2005 standardmäßig importiert).

```
MsgBox ("Es hat geklappt")
```

Beim Aufruf von *MsgBox()* wird der auszugebende Text in Klammern als Parameter übergeben. Der in Klammern stehende Parameter muss dabei eine Zeichenkette sein. Die obige *MsgBox()*-Anweisung erzeugt das Dialogfeld aus Abbildung 8.5, links. Der als Parameter übergebene Text wird im Meldungsfeld angezeigt. Als Titelleiste benutzt Visual Basic 2005 standardmäßig den Namen der Anwendung (konkret wird der Name der Assembly benutzt). Außerdem besitzt das Dialogfeld eine OK-Schaltfläche zum Schließen.

Über zusätzliche Parameter lässt sich das Aussehen des Dialogfelds in gewissen Grenzen gestalten. Einmal können Sie den Titeltext als dritten Parameter beim *MsgBox()*-Aufruf vorgeben. Im zweiten Parameter lässt sich zudem ein Stil definieren, der ggf. die Art der Schaltflächen angibt oder ein Symbol einblendet. Die Anweisungsfolge

```
Const Titel As String = " Visual Basic 2005 - by G. Born"
...
MsgBox ("Es hat geklappt", vbExclamation, Titel)
```

erzeugt das Dialogfeld aus Abbildung 8.5, Mitte. Der dritte Parameter *Titel* ist als Konstante vom Typ *String* vereinbart und enthält den auszugebenden Titeltext. Im ersten Parameter steht ein Stringwert, der im Dialogfeld angezeigt wird. Der zweite Parameter enthält die benannte Konstante *vbExclamation*. Es handelt sich um eine der vordefinierten Visual-Basic-Konstanten (wie z.B. *vbCrLf*). Die Konstante *vbExclamation* bewirkt, dass das Ausrufezeichen mit im Dialogfeld eingeblendet wird. Die folgende Tabelle enthält die symbolischen Konstanten, die Sie zum Abrufen der Symbole verwenden können:

Konstante	Symbol	Bemerkung
<i>vbCritical</i>		Meldung für kritischen Fehler
<i>vbQuestion</i>		Warnung mit Abfrage
<i>vbExclamation</i>		Warnmeldung
<i>vbInformation</i>		Informationsmeldung

Tabelle 8.2: Konstanten für *MsgBox()*-Symbole

Hinweis

Geben Sie die Parameter für den *MsgBox()*-Aufruf in der Entwicklungsumgebung ein, blendet diese in der QuickInfo standardmäßig die *MsgBoxStyle*-Konstanten (z.B. *MsgBoxStyle.Exclamation*) ein. Diese besitzen aber die gleichen Werte wie die hier in der Tabelle aufgeführten *vb*-Konstanten. Wenn Sie in den hier aufgeführten Tabellen den Vorspann *vb* im Konstantennamen durch *MsgBoxStyle.* ersetzen, ergibt sich der Name der jeweiligen *MsgBoxStyle*-Konstanten. Welche Variante Sie benutzen, bleibt Ihrer Vorliebe überlassen.

In Abbildung 8.5, rechts, wird beispielsweise das *vbInformation*-Symbol innerhalb des Dialogfelds benutzt. Dieses Dialogfeld enthält auch einige formatierte Werte, die in mehreren Zeilen ausgegeben werden.

Da die *MsgBox()*-Methode nur einen Parameter für den auszugebenden *String*-Wert vorsieht, müssen Sie die Formatierung selbst vornehmen. Um einen Zeilenumbruch im Ausgabestring einzuleiten, lässt sich beispielsweise die Visual-Basic-Konstante *vbCrLf* einfügen. Zahlen müssen vor der Ausgabe mit *ToString()* in eine Zeichenkette umgewandelt werden. Dabei lässt sich der Methode ein Formatstring als Parameter übergeben, um die Anzeige der Zahl zu beeinflussen. Die folgende Anweisung zeigt den Befehl zur Ausgabe des Dialogfelds aus Abbildung 8.5, rechts:

```
MsgBox (txt & vbCrLf & _  
        "Wert: " & vbTab & Wert.ToString & vbCrLf & _  
        "Wert: " & vbTab & Wert1.ToString("#0.00"), _  
        vbInformation, Titel)
```

Der Aufbau des Formatstrings wurde ja bereits auf den vorhergehenden Seiten beschrieben. Sie sehen also, mit etwas Know-how ist die formatierte Ausgabe von Daten in Dialogfeldern kein Problem mehr. Weitere Einzelheiten können Sie dem folgenden Listing entnehmen:

```
'*****  
' File/Projekt: Beisp08_05  
' Autor: G. Born www.borncity.de  
' Demonstriert die Anwendung der MsgBox-Methode  
' zur Anzeige von Meldungsfeldern.  
'*****  
Option Strict On  
  
Class Test  
    Shared Sub Main()  
        Const Titel As String = " Visual Basic 2005 - by G. Born"  
        Dim txt As String = "Eine Ausgabe"  
        Dim tmp As Integer      ' Auswahl  
        Dim Wert As Integer = 120  
        Dim Wert1 As Double = 123.45  
  
        ' Primitivste Benutzermeldung  
        MsgBox ("Es hat geklappt")  
  
        ' jetzt mit Titel und Symbol  
        MsgBox ("Es hat geklappt", vbExclamation, Titel)
```

Listing 8.5: Ausgabe über *MsgBox()*

```

' jetzt Werte ausgeben
MsgBox (txt & vbCrLf & _
        "Wert: " & vbTab & Wert.ToString & vbCrLf & _
        "Wert: " & vbTab & Wert1.ToString("#0.00"), _
        vbInformation, Titel)
End Sub
End Class

```

Listing 8.5: Ausgabe über MsgBox() (Forts.)

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung implementierten Beispiels im Ordner `\Beisp\Kap08\Beispiel8_05` der Begleit-CD.

8.2.2 Dialogfelder mit mehreren Schaltflächen

Die im vorhergehenden Beispiel gezeigten Dialogfelder besaßen nur eine *OK*-Schaltfläche. Der zweite Parameter der *MsgBox()*-Methode erlaubt aber die Angabe eines Stils, bei dem nicht nur die Symbole, sondern auch verschiedene Kombinationen an Schaltflächen (z.B. *OK*, *Abbrechen*) auswählbar sind. Hierzu sind zusätzliche Konstante aus der folgenden Tabelle im Parameter *Style* mit anzugeben.

Konstante	Bemerkung
vbOKOnly	Nur <i>OK</i> -Schaltfläche (Standardwert). Lassen Sie die Konstante weg, wird die <i>OK</i> -Schaltfläche automatisch eingeblendet, da die Konstante dem Wert 0 entspricht.
vbOKCancel	Schaltflächen <i>OK</i> und <i>Abbrechen</i> anzeigen
vbAbortRetryIgnore	Schaltflächen <i>Abbrechen</i> , <i>Wiederholen</i> und <i>Ignorieren</i> einblenden
vbYesNoCancel	Schaltflächen <i>Ja</i> , <i>Nein</i> und <i>Abbrechen</i> anzeigen
vbYesNo	Schaltflächen <i>Ja</i> und <i>Nein</i> verwenden
vbRetryCancel	Verwende die Schaltflächen <i>Wiederholen</i> und <i>Abbrechen</i>

Tabelle 8.3: MsgBox-Konstanten zur Auswahl der Schaltflächen

Dabei besteht die Möglichkeit, die verschiedenen Konstanten zu einem *Style*-Wert zu kombinieren. Sie können also beispielsweise ein Dialogfeld mit einer *Ja/Nein*-Schaltflächenkombination sowie einem Ausrufezeichen erzeugen (Abbildung 8.6).

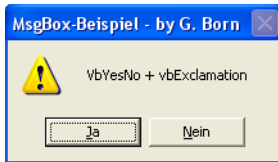


Abbildung 8.6: Dialogfeld mit mehreren Schaltflächen und Symbol

Hinweis

Das Projektbeispiel im Ordner `\Beisp\Kap08\Beispiel8_05a` zeigt nicht nur die verschiedenen Symbole der `MsgBox()`-Anweisung, sondern enthält auch die betreffenden Schaltflächenkonstanten, um verschiedene Schaltflächensätze anzuzeigen. Sofern Sie bereits mit VBScript, VBA oder älteren Visual-Basic-Versionen gearbeitet haben, kennen Sie den Ansatz, die symbolischen Konstanten für die Schaltflächenauswahl und für das Symbol einfach zu addieren:

```
MsgBox ("VbYesNo + vbExclamation", vbYesNo + vbExclamation, Titel)
```

Dies klappt in Visual Basic 2005 aber nur, solange die Option `Strict Off` gesetzt ist. Wurde diese Option dagegen vereinbart, meldet der Compiler einen Fehler, dass keine implizite Typkonvertierung möglich ist. Sie können dies umgehen, indem Sie folgende Hilfskonstruktion verwenden:

```
Dim Style As MsgBoxStyle
Style = vbYesNo Or vbExclamation
```

Setzen Sie die Variable `Style` anschließend im zweiten Parameter des `MsgBox()`-Aufrufs ein, wird der Compiler die Konstanten akzeptieren. Sie finden die Projektdateien eines als Windows-Anwendung implementierten Beispiels im Ordner `\Beisp\Kap08\Beispiel8_05a` der Begleit-CD.

8.2.3 Benutzerführung über Dialogfelder ganz einfach

Gelegentlich benötigt man in Programmen eine Art Benutzerführung, bei der ein Dialogfeld erscheint und den Benutzer auffordert, eine Option auszuwählen. Windows-Anwender kennen diese Dialoge mit Schaltflächen wie *OK*, *Abbrechen*, *Ignorieren*, bei denen abhängig von der angeklickten Schaltfläche eine Aktion ausgeführt wird. Hierzu kann der Rückgabewert der `MsgBox()`-Methode ausgewertet werden, da diese die vom Benutzer angeklickte Schaltfläche identifiziert. Die folgende Anweisungsfolge demonstriert diesen Ansatz:

```
Dim tmp As MsgBoxResult
Dim Style As MsgBoxStyle
...
Style = vbAbortRetryIgnore Or vbQuestion Or vbDefaultButton1
tmp = MsgBox ("Bitte eine Schaltfläche wählen", Style, Titel)
```

Die ersten beiden Zeilen vereinbaren zwei Variable, die den Rückgabewert von `MsgBox()` sowie den Stil des Dialogfelds aufnehmen sollen. Dann wird die Stilkonstante bestimmt und in der letzten Anweisung wird `MsgBox()` aufgerufen. An dieser Stelle möchte ich aber noch auf verschiedene Besonderheiten hinweisen.

In der ersten Anweisung wird die Variable `tmp` mit dem Datentyp `MsgBoxResult` vereinbart. Bei diesem Datentyp handelt es sich um eine Enumeration, die Konstanten für die `MsgBox()`-Rückgabewerte enthält. Die nachfolgende Tabelle listet die Rückgabewerte der jeweiligen Schaltflächen auf.

Wert	Name	VB-Konstante	Bedeutung
1	OK	vbOK	Benutzer hat die <i>OK</i> -Schaltfläche angeklickt
2	Cancel	vbCancel	Benutzer hat die <i>Abbrechen</i> -Schaltfläche (Cancel) angeklickt
3	Abort	vbAbort	Benutzer hat die <i>Abbrechen</i> -Schaltfläche (Abort) angeklickt
4	Retry	vbRetry	Benutzer hat die <i>Wiederholen</i> -Schaltfläche angeklickt
5	Ignore	vbIgnore	Benutzer hat die <i>Ignorieren</i> -Schaltfläche angeklickt
6	Yes	vbYes	Benutzer hat die <i>Ja</i> -Schaltfläche angeklickt
7	No	vbNo	Benutzer hat die <i>Nein</i> -Schaltfläche angeklickt

Tabelle 8.4: Konstanten der `MsgBoxResult`-Enumeration

Verwenden Sie den Datentyp `MsgBoxResult` für die Variable, enthält diese Werte, die der Spalte *Name* entsprechen. Vereinbaren Sie den Typ der Variablen zur Aufnahme des Rückgabewerts als `Integer`, werden Zahlen der Spalte *Wert* hinterlegt. In beiden Fällen lässt sich der Rückgabewert mit den Werten der Spalte *VB-Konstante* vergleichen. Diese Konstanten lassen sich im Programm verwenden, sobald Sie den Namensraum *Microsoft.VisualBasic* importieren.

Achtung

Haben Sie *Option Strict On* vereinbart, müssen Sie peinlichst genau im Code darauf achten, die richtigen Datentypen zu verwenden. Daher wurden in obiger Code-Sequenz in der Variablendeklaration die Datentypen `MsgBoxResult` (und im folgenden Beispiel `Integer`) für die Rückgabewerte und `MsgBoxStyle` für den Stil benutzt. Dies stellt sicher, dass keine Typkonvertierung beim Methodenaufruf und bei der Zuweisung des Ergebnisses erfolgen muss. Benutzer älterer Versionen von Visual Basic, VBA oder VBScript erscheint dies vielleicht kompliziert. Die strenge Beachtung von Typen vermeidet aber die Generierung überflüssigen Codes zur Konvertierung der Datentypen durch den Compiler. Zudem vermeidet dies Fehler, die aufgrund impliziter Typkonvertierungen auftreten können.

Die Vereinbarung der Stilkonstante, die die Zahl der Schaltflächen und das anzuzeigende Symbol festlegen, ist Ihnen bereit aus den vorhergehenden Beispielen bekannt. Die Werte für die Schaltflächenkombination sowie das anzuzeigende Symbol finden Sie in den Tabellen der vorhergehenden Seiten. Neu ist hier die Festlegung einer Standard-

Schaltfläche über die Konstante `vbDefaultButton1`. Diese veranlasst `MsgBox()`, der ersten Schaltfläche (von links gesehen) den Fokus zuzuweisen. Drückt der Benutzer sofort nach dem Erscheinen des Dialogfelds die `[↵]`-Taste, wird das Dialogfeld geschlossen. `MsgBox()` gibt dann den Code der Schaltfläche, die den Fokus besitzt, zurück. Klickt der Benutzer dagegen eine andere Schaltfläche an, wird deren Code zurückgegeben. Sie haben die Möglichkeit, jeder der drei im Dialogfeld angezeigten Schaltflächen den Fokus zuzuweisen, indem Sie eine der in der folgenden Tabelle aufgeführten Konstanten im Parameter `Style` an die Methode übergeben. Fehlt der Wert in der Stildefinition, verwendet die `MsgBox()`-Methode die linke Schaltfläche als Standardschaltfläche.

Konstante	Bedeutung
<code>vbDefaultButton1</code>	Erste Schaltfläche ist Standardschaltfläche (Standardwert)
<code>vbDefaultButton2</code>	Zweite Schaltfläche ist Standardschaltfläche
<code>vbDefaultButton3</code>	Dritte Schaltfläche ist Standardschaltfläche

Tabelle 8.5: Konstanten zum Festlegen der `MsgBox`-Standardschaltfläche

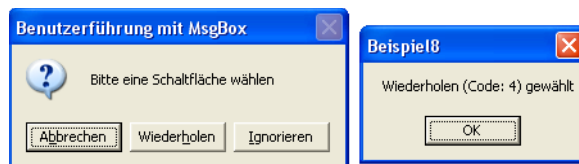


Abbildung 8.7: Dialoge des Beispiels zur Auswertung der angeklickten Schaltfläche

Sobald Sie den Stil für die `MsgBox()`-Methode vereinbart haben und den Methodenaufruf ausführen, erscheint das entsprechende Dialogfeld. Nach dem Schließen lässt sich dann der Rückgabewert auswerten und das Programm kann in Abhängigkeit von der gewählten Schaltfläche reagieren. Das folgende Beispielprogramm erzeugt das in Abbildung 8.7 (links) gezeigte Dialogfeld und listet nach dem Schließen der Meldung den Wert der angeklickten Schaltfläche auf (Abbildung 8.7, rechts). In diesem Beispiel habe ich übrigens eine `Select Case`-Anweisung benutzt, um auf die drei Fälle zu reagieren. Weitere Details sind dem folgenden Listing zu entnehmen:

```

' *****
' File/Projekt:  Beispiel08_06
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung der MsgBox()-Methode
' zur Benutzerführung. Die Rückgabecodes der vom
' Benutzer angeklickten Schaltflächen werden ausgewertet.
' *****
Option Strict On

Class Test

```

Listing 8.6: Beispiel zur Auswertung der angeklickten Schaltfläche

```

Shared Sub Main()
    Const Titel As String = "Benutzerführung mit MsgBox"
    ' Rückgabewert als Integer oder MsgBoxResult vereinbaren
    Dim tmp As Integer      ' Schaltflächencode
    Dim Style As MsgBoxStyle ' Anzeigestil

    ' Abbrechen, Wiederholen, Ignorieren, Abbrechen als Standard,
    Style = vbAbortRetryIgnore Or vbQuestion Or vbDefaultButton1

    ' Gebe Dialogfeld mit Titel und Symbol aus, lese Rückgabewert
    tmp = MsgBox ("Bitte eine Schaltfläche wählen", Style, Titel)

    Select tmp                ' Auswertung des Rückgabecodes
        Case Is = vbAbort
            MsgBox ("Abbrechen (Code: " & tmp.ToString & ") gewählt")
        End

        Case Is = vbRetry
            MsgBox ("Wiederholen (Code: " & tmp.ToString & ") gewählt")
        End

        Case Is = vbIgnore
            MsgBox ("Ignorieren (Code: " & tmp.ToString & ") gewählt")
        End

        Case Else             ' dürfte nicht vorkommen
            MsgBox ("Unbekannter Code: " & tmp.ToString)
        End
    End Select
End Sub
End Class

```

Listing 8.6: Beispiel zur Auswertung der angeklickten Schaltfläche (Forts.)

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung implementierten Beispiels im Ordner `\Beisp\Kap08\Beispiel8_06` der Begleit-CD.

Spezielle Konstante für den MsgBox-Stil

Die `MsgBox()`-Methode unterstützt einige zusätzliche Konstante zur Steuerung der Sichtbarkeit der Dialoge. Ein von einer Anwendung angezeigtes Dialogfeld verschwindet im Hintergrund, sobald der Benutzer das Fenster einer anderen Anwendung in den Vorder-

grund holt. Diese als anwendungsmodal bezeichnete Eigenschaft ist die Standardvorgabe unter Windows. Arbeitet eine .NET-Anwendung im Hintergrund, z.B. weil Sie ein anderes Programmfenster angeklickt haben, sehen Sie bei Dialogen lediglich eine Schaltfläche in der Windows-Taskleiste. Sie können aber einen systemmodalen Anzeigestil verwenden, indem Sie beim *MsgBox()*-Aufruf die Konstante *vbSystemModal* angeben. Dann bleibt die Schaltfläche immer im Vordergrund, egal welche Anwendung den Fokus besitzt. Die folgende Tabelle listet weitere von *MsgBox()* unterstützte Stilkonstanten auf.

Konstante	Bedeutung
<i>vbApplicationModal</i>	An die Anwendung gebundenes Meldungsfeld (Standardwert) - der Dialog kann durch andere Fenster verdeckt werden und kommt nur in den Vordergrund, wenn das Anwendungsfenster sichtbar ist
<i>vbSystemModal</i>	An das System gebundenes Meldungsfeld
<i>vbMsgBoxSetForeground</i>	Meldungsfeld in den Vordergrund setzen
<i>vbMsgBoxRight</i>	Text im Meldungsfeld rechts ausrichten
<i>vbMsgBoxRtlReading</i>	Text, der von rechts nach links gelesen wird (hebräische und arabische Systeme)

Tabelle 8.6: Weitere Konstanten für den *MsgBox*-Stil

Hinweis

Im Ordner `\Beisp\Kap08\Beispiel8_06a` der Begleit-CD finden Sie ein Projektbeispiel, welches die Anwendung eines systemmodalen Dialogfelds demonstriert. Nach dem Start sehen Sie mehrere Dialogfelder, die sich entweder in den Hintergrund schalten lassen oder zwangsweise im Vordergrund bleiben.

8.2.4 So setzen Sie die *MessageBox*-Klasse ein

Kommen wir nun zur Frage, wie sich mit der von Microsoft empfohlenen *MessageBox*-Klasse Dialoge anzeigen lassen. Im Grunde funktioniert dies ähnlich wie bei *MsgBox()*. Sie müssen lediglich die Parameter in anderer Reihenfolge übergeben und die Konstanten werden aus der *MessageBox*-Klasse abgeleitet. Um die *MessageBox*-Klasse überhaupt nutzen zu können, müssen Sie im Programm den folgenden Namensraum vereinbaren:

```
Imports System.Windows.Forms
```

Dieser Namensraum ist normalerweise zur Anzeige von Formularen und Standarddialogen unter Windows vorgesehen. Microsoft hat in diesem Namensraum quasi als Anhängsel auch eine Klasse *MessageBox* zur Anzeige von Dialogen spendiert. Um ein simples Dialogfeld mit einem Text auszugeben, müssen Sie die *Show*-Methode dieser Klasse verwenden. Die einfachste Variante sieht dann so aus:

```
MessageBox.Show (Text)
```

Der Aufruf ist nicht wesentlich komplizierter als *MsgBox(Text)*, der Hauptunterschied liegt in der etwas abweichenden Anordnung weiterer (optionaler) Parameter. Das folgende kurze Listing zeigt ein minimales Programm, welches mit *MessageBox.Show* ein einfaches Dialogfeld ausgibt (Abbildung 8.8). Dem Dialogfeld fehlt der Titeltext, folglich wird die Schaltfläche in der Taskleiste auch keinerlei Bezeichnung tragen.



Abbildung 8.8: Dialogfeld mit *MessageBox.Show()* ausgegeben

Um ein halbwegs sinnvolles Dialogfeld anzuzeigen, müssen Sie bei der *MessageBox*-Klasse immer einen Titeltext für das Dialogfeld angeben. Dieses wird aber, im Gegensatz zur *MsgBox()*-Methode, nicht im dritten, sondern im zweiten Parameter der *Show*-Methode übergeben. Die *Show*-Methode erwartet folgendes Aufrufformat:

```
Tmp = MessageBox.Show (Text, Titel, Buttons, Icons, DefaultButton)
```

Die Methode möchte also für die Optionen, die bei *MsgBox()* in einer Konstante zusammengefasst wurden, drei getrennte Argumente. Die Argumente der Methode sind folgendermaßen festgelegt:

- *Text*: Dieses Argument ist erforderlich und gibt den im Dialogfeld anzuzeigenden Text an.
- *Titel*: Ein optionales Argument, welches den Titeltext des Dialogfelds aufnimmt.
- *Buttons*: Optionales Argument mit den Konstanten für die anzuzeigende Schaltflächenkombination. Erlaubt sind die folgenden Werte der *MessageBoxButtons*-Enumeration: *AbortRetryIgnore*, *OK*, *OKCancel*, *RetryCancel*, *YesNo*, *YesNoCancel*. Die Wirkung der betreffenden Konstanten dürfte anhand des Namens selbsterklärend sein.



Abbildung 8.9: Mit *MessageBox.Show()* darstellbare Dialogfeldsymbole

- *Icons*: Optionales Argument mit einer Konstante aus der *MessageBoxIcon*-Enumeration, die das anzuzeigende Symbol festlegt (Abbildung 8.9). Die Enumeration enthält folgende benannte Konstanten: *Asterisk* (Symbol: kleines i in einer Blase), *Error* (Symbol: weißes X in einem Kreis mit rotem Hintergrund), *Exclamation* (Symbol: Ausrufezeichen in einem Dreieck mit gelbem Hintergrund), *Hand* (Symbol: weißes X in einem Kreis mit rotem Hintergrund), *Information* (Symbol: kleines i in einer Blase), *None* (kein Symbol im Meldungsfeld), *Question* (Symbol: Fragezeichen in einer Blase), *Stop* (Symbol: weißes X in einem Kreis mit rotem Hintergrund), *Warning* (Symbol: Ausrufezeichen in einem Dreieck mit gelbem Hintergrund).

- *DefaultButton*: Optionales Argument, welches die Standardschaltfläche angibt. Die *MessageBoxDefaultButton*-Enumeration enthält die Konstanten *Button1*, *Button2* und *Button3*, die die jeweilige Schaltfläche auswählen.

Sie erkennen, dass einige der Konstanten im Parameter *Icons* das gleiche Symbol erzeugen. Zudem sind die Namen der Enumerationen recht lang, was viel Schreibarbeit beim Eingeben des Programmcodes bedeutet.

Das folgende Listing zeigt, wie sich ein Dialogfeld mit einer *Ja/Nein*-Schaltflächenkombination und einem *Exclamation*-Symbol erzeugen lässt. Anschließend wird in Abhängigkeit von der angeklickten Schaltfläche eine Meldung ausgegeben.

```

'*****
' File/Projekt: Beispiel08_07
' Autor: G. Born www.borncity.de
' Demonstriert die Anwendung der MessageBox-Methode
' zur Benutzerführung. Die Rückgabecodes der vom
' Benutzer angeklickten Schaltflächen werden
' teilweise ausgewertet.
'*****
Option Strict On
Imports System.Windows.Forms ' für MessageBox

Class Test
    Shared Sub Main()
        Const Titel As String = "Benutzerführung mit MessageBox"
        Const Text As String = "Bitte eine Schaltfläche wählen"

        Dim tmp As Integer ' Schaltflächencode

        MessageBox.Show("Hallo, es hat geklappt")

        ' Ja/Nein, Nein als Standard und Ausrufezeichen als Symbol
        tmp = MessageBox.Show(Text, Titel, MessageBoxButtons.YesNo, _
                               MessageBoxIcon.Exclamation, _
                               MessageBoxDefaultButton.Button2)
        If tmp = DialogResult.Yes Then
            MessageBox.Show("Schaltfläche Ja wurde angeklickt")
        Else
            MessageBox.Show("Schaltfläche Nein wurde angeklickt")
        End If
    End Sub
End Class

```

Listing 8.7: Anzeige eines Dialogfelds mit der *MessageBox*-Klasse

Hinweis

Im Ordner `\Beisp\Kap08\Beispiel8_07` der Begleit-CD finden Sie ein Projektbeispiel, welches die Anwendung der *MessageBox*-Klasse demonstriert. Auf diese Weise können Sie die *Show*-Methode der *MessageBox*-Klasse einsetzen. Ob Sie für einfache Visual-Basic-2005-Anwendungen auf diese Klasse oder auf *MsgBox()* zurückgreifen, bleibt Ihnen überlassen. Die Vorteile der Klasse kommen erst zum Tragen, wenn Meldungsfelder an ein Formular angebunden werden sollen (dann lässt sich eine Referenz auf das Formularobjekt als erster Parameter übergeben). Befindet sich das Formular im Vordergrund, ist dann auch das Dialogfeld zu sehen.

8.3 Benutzereingaben über InputBox

Bleibt noch die Frage, wie sich einfache Benutzerabfragen unter Windows realisieren lassen. Formulare bieten eine komfortable Möglichkeit, um Benutzereingaben abzufragen. Auf diese Möglichkeit komme ich später zurück. Um in einer kleinen .NET-Anwendung einen Wert abzufragen, lohnt sich der Aufwand zur Gestaltung eines Formulars nicht. Umsteiger von älteren Visual-Basic-Versionen kennen dagegen die *InputBox()*-Funktion, die Bestandteil der Sprache ist. Die Funktion ist auch in VBA oder VBScript vorhanden. Microsoft hat diese Funktion als Methode in der Visual-Basic-Kompatibilitätsklasse implementiert. Die Anwendung dieser Methode ist trivial:

```
Tmp = InputBox(Text, Titel, Default, X, Y)
```

Die Parameter dieser Methode sind folgendermaßen definiert:

- *Text*: Ein *String*-Wert, der den im Eingabedialog anzuzeigenden Text enthält.
- *Titel*: Ein optionaler *String*-Wert, der die Titelleiste des Eingabedialogs definiert.
- *Default*: Ein optionaler *String*-Ausdruck, der den Vorgabewert für den Eingabewert angibt.
- *X*: Ein optionaler *Integer*-Wert, der die X-Koordinate für das Dialogfeld bezüglich des Desktops festlegt.
- *Y*: Ein optionaler *Integer*-Wert, der die Y-Koordinate für das Dialogfeld bezüglich des Desktops festlegt.

Lassen Sie den *Default*-Parameter weg, bleibt das Eingabefeld leer. Fehlende Positionsangaben für *X* und *Y* bewirken, dass das Dialogfeld zentriert auf dem Desktop angeordnet wird.

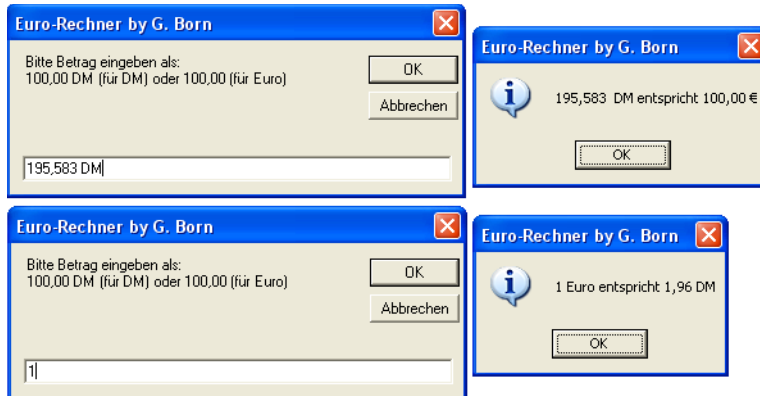


Abbildung 8.10: Ein-/Ausgabedialoge zur Währungsumrechnung

Das nachfolgende Beispiel demonstriert die Anwendung der *InputBox*-Methode bei der Umrechnung von Währungsbeträgen. Ich habe die Umrechnung Euro zu DM gewählt, da dort der Kurs im Programm über die Konstante *EuroFaktor* fest eingetragen werden kann. Sie können das Beispiel auch für andere Währungsumrechnungen verwenden, sofern Sie den Kurs in der Konstante *EuroFaktor* und ggf. die Dialogfeldtexte entsprechend anpassen. In einem Dialogfeld (Abbildung 8.10) werden Währungsbeträge abgefragt. Eine Zahl ohne Währungskennzeichen wird als Euro-Betrag interpretiert und dann in den äquivalenten DM-Betrag zurückgerechnet. Gibt der Benutzer hinter dem Betrag die Währungseinheit »DM« an, erkennt die Anwendung dies und rechnet den Betrag in Euro um.

Diese Aufgabe lässt sich mit dem bisherigen Wissen über Visual Basic 2005 lösen. Zur Eingabe der Währungsbeträge verwendet das Beispiel die *InputBox*-Methode:

```
Const Titel As String = "Euro-Rechner by G. Born"
Text = "Bitte Betrag eingeben als:" & vbCrLf & _
      "100,00 DM (für DM) oder 100,00 (für Euro)"
tmp = InputBox (Text, Titel, "")
```

Die Methode liefert eine Zeichenkette zurück, die leer ist, falls der Benutzer im Dialogfeld auf die Schaltfläche *Abbrechen* klickt. Dann soll das Programm enden. Andernfalls beginnt die Umrechnung der Währungsbeträge.

Das Programm fordert so lange Währungsbeträge an, bis der Benutzer die Schaltfläche *Abbrechen* betätigt. Die Abprüfung auf das Endekriterium ist mit folgender Anweisung möglich:

```
Do Until false ' Endlosschleife
    tmp = InputBox (Text, Titel, "")
    If tmp = "" Then Exit Do
    ...
End Do
```

Wurde etwas eingegeben, prüft das Programm zuerst mittels der *EndsWith*-Methode des *String*-Objekts, ob die Angabe »DM« im String vorkommt. Um die Groß-/Kleinschreibung zu umgehen, konvertiert die *ToUpper*()-Methode die Eingabe in Großbuchstaben:

```
If tmp.ToUpper.EndsWith("DM") Then ' DM-Eingabe
    tmp = tmp.Remove(tmp.Length-2, 2)
```

Die *Remove*-Methode entfernt anschließend die beiden letzten Zeichen der Zeichenkette. Damit sind Fälle wie »123,45DM« und »123,45 DM« abgefangen. Trotzdem können noch fehlerhafte Eingaben auftreten wie »DM 123,45« oder »123,45 x DM«. Daher prüft das Programm anschließend, ob der Reststring eine gültige Zahl enthält:

```
If IsNumeric (tmp) Then
```

Trifft dies zu, berechnet das Programm den jeweils anderen Währungsbetrag und gibt dann das Ergebnis in einem Dialogfeld mittels *MsgBox*() aus. Weitere Details sind nachfolgendem Listing zu entnehmen:

```
'*****
' File/Projekt: Beispiel08_08
' Autor: G. Born www.borncity.de
' Übersetzen in eine Windows-Anwendung mit:
'*****
Option Strict On
Imports Microsoft.VisualBasic ' für InputBox und MessageBox

Class Test
    Shared Sub Main()
        Const EuroFaktor As Double = 1.95583
        Const Titel As String = "Euro-Rechner by G. Born"
        Dim Wert As Double
        Dim tmp As String
        Dim Text As String

        Text = "Bitte Betrag eingeben als:" & vbCrLf & _
            "100,00 DM (für DM) oder 100,00 (für Euro)"

        Do Until False ' Endlosschleife
            tmp = InputBox(Text, Titel, "")
            If tmp = "" Then Exit Do ' Euro-Rechner beenden

            If tmp.ToUpper.EndsWith("DM") Then ' DM-Eingabe
                tmp = tmp.Remove(tmp.Length - 2, 2)
```

Listing 8.8: Datei EuroRechner.vb zum Umrechnung von Währungsbeträgen


```

        If IsNumeric(tmp) Then
            Wert = CDb1(tmp) / EuroFaktor ' DM in
            MsgBox(tmp & " DM entspricht " & _
                Wert.ToString("C"), _
                vbOkOnly Or vbInformation, Titel)
        Else
            MsgBox(tmp & " ist eine fehlerhafte Eingabe", _
                vbOkOnly Or vbCritical, Titel)
        End If
    Else
        If IsNumeric(tmp) Then
            Wert = CDb1(tmp) * EuroFaktor ' Euro in DM
            MsgBox(tmp & " Euro entspricht " & _
                Wert.ToString("#,##0.00") & " DM", _
                vbOkOnly Or vbInformation, Titel)
        Else
            MsgBox(tmp & " ist eine fehlerhafte Eingabe", _
                vbOkOnly Or vbCritical, Titel)
        End If
    End If
Loop
End Sub
End Class

```

Listing 8.8: Datei EuroRechner.vb zum Umrechnung von Währungsbeträgen (Forts.)

Hinweis

Sie finden das als Windows-Anwendung implementierte Projektbeispiel im Ordner `\Beisp\Kap08\Beispiel8_08` der Begleit.

Damit möchte ich dieses Kapitel schließen. Mit den bisherigen Ausführungen verfügen Sie über genügend Kenntnisse, um .NET-Anwendungen, die eine einfache Benutzerführung erfordern, zu realisieren.

9

Arbeiten mit Formularen

Formulare stellen bei der Entwicklung von .NET-Anwendungen vielfältige Möglichkeiten zur Interaktion mit dem Benutzer bereit. In diesem Kapitel finden Sie eine Einführung in das Erstellen von Formularen.

9.1 Formularentwurf in der Entwicklungsumgebung

Der Entwurf von Formularen wird in einer Entwicklungsumgebung wie Visual Studio 2005 oder Visual Basic 2005 Express Edition durch einen Formdesigner (auch als Windows Forms-Designer bezeichnet) unterstützt. In den nachfolgenden Abschnitten wird gezeigt, wie der Formularentwurf in der Entwicklungsumgebung funktioniert und was zu beachten ist.

9.1.1 So wird ein Formular angelegt

Formulare werden automatisch erzeugt, sobald Sie ein neues Windows-Projekt in der Entwicklungsumgebung mit folgenden Schritten anlegen:

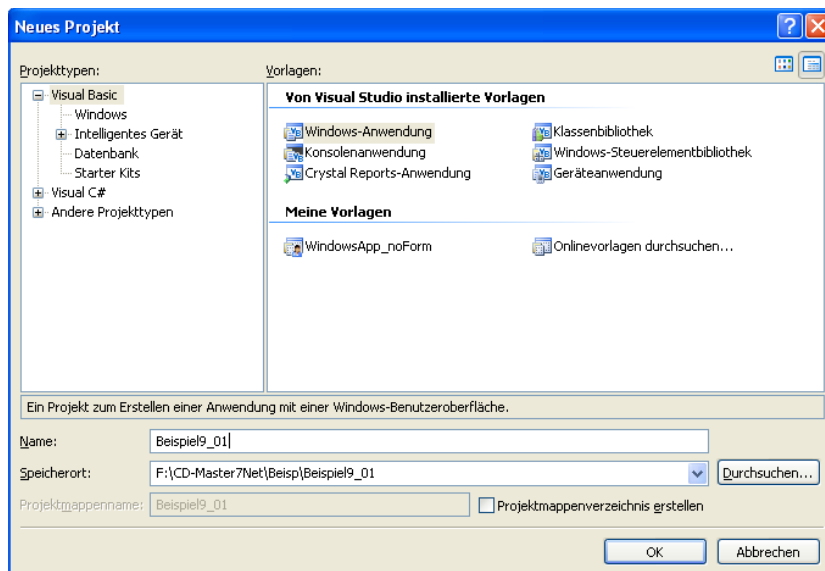


Abbildung 9.1: Projektauswahl in Visual Studio 2005

1. Starten Sie die Entwicklungsumgebung und wählen Sie anschließend im Programmfenster im Menü *Datei* den Befehl *Neu/Projekt*.
2. In Visual Studio 2005 müssen Sie im Dialogfeld *Neues Projekt* (Abbildung 9.1) den Eintrag *Visual Basic* in der Kategorie *Projekttypen* wählen.
3. Klicken Sie in der Kategorie *Vorlagen* des Dialogfelds *Neues Projekt* auf die Vorlage *Windows-Anwendung*. Passen Sie bei Bedarf im Dialogfeld *Neues Projekt* den Text im Feld *Name* an und setzen Sie ggf. im Feld *Speicherort* das Zielverzeichnis zur Ablage der Projektdateien.

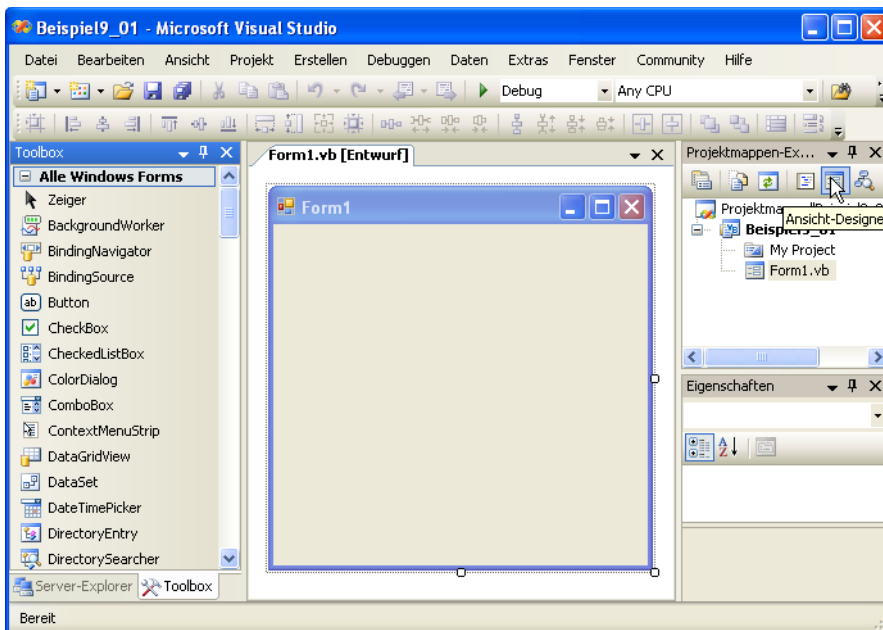


Abbildung 9.2: Visual-Studio-2005-Fenster mit dem Formulardesigner (Ansicht-Designer)

Sobald Sie das Dialogfeld über die *OK*-Schaltfläche schließen, erzeugt die Entwicklungsumgebung die Projektdateien aus der von Ihnen gewählten Vorlage und zeigt diese im Projektmappe-Explorer an. Die Vorlage *Windows-Anwendung* beinhaltet auch ein Startformular für die zu erstellende Anwendung. In diesem Fall sehen Sie einen Eintrag »Form1.vb« im Projektmappe-Explorer. Gleichzeitig wird das Formular im Fenster des Formulardesigners eingeblendet (Abbildung 9.2, mittleres Fenster). Der Designer wird hier im Buch, in Anlehnung an die Schaltfläche des Projektmappe-Explorers zum Einblenden des Formulardesigns, auch als *Ansicht-Designer* bezeichnet. Sie können anschließend die Formulareigenschaften anpassen oder das Formular über die am linken Fensterrand eingeblendete Toolbox um weitere Steuerelemente erweitern.

Tipp

Über das Fenster des Projektmappen-Explorers können Sie die gewünschten Formulare jederzeit per Doppelklick anwählen und im Formulardesigner zur Anzeige bringen. Über die Schaltflächen *Code anzeigen* und *Design-Ansicht* in der Kopfzeile des Projektmappen-Explorers lässt sich dabei zwischen der hier gezeigten Designansicht und der zum Bearbeiten der Ereignisroutinen benötigten Codeansicht wechseln. Die Funktionstaste **[F7]** schaltet ebenfalls zur Codeansicht mit den Ereignisprozeduren der Steuerelemente um.

9.1.2 Steuerelemente in Formulare einfügen

Das Einfügen von Steuerelementen in Formulare ist mit dem Formulardesigner recht einfach. Die verfügbaren Steuerelemente werden in der Werkzeugleiste *Toolbox* angezeigt. Diese Werkzeugleiste lässt sich bei Bedarf über den gleichnamigen Befehl des Menüs *Ansicht* bzw. über die gleichnamige Schaltfläche in der *Standard*-Symbolleiste im Anwendungsfenster einblenden. Um ein Steuerelement im Formular einzufügen, sind folgende Schritte auszuführen:

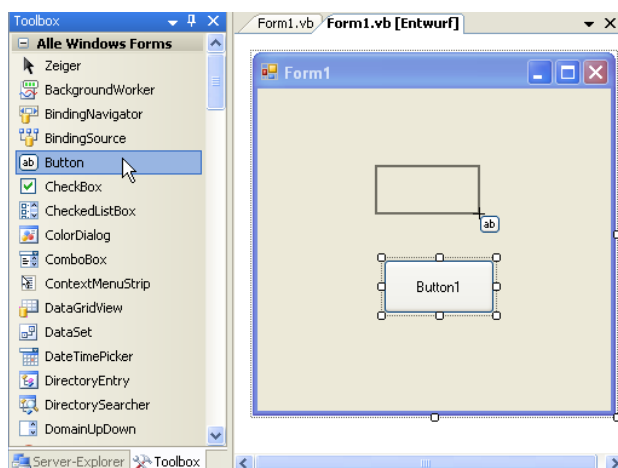


Abbildung 9.3: Einfügen eines Steuerelements in ein Formular

1. Klicken Sie in der Symbolleiste *Toolbox* auf die Schaltfläche des gewünschten Steuerelements (Abbildung 9.3).
2. Zeigen Sie im Formularlayout auf die linke obere Ecke der Steuerelementposition und ziehen Sie den Mauszeiger zur diagonal entgegengesetzten Ecke des Steuerelements (Abbildung 9.3).

Im Designfenster signalisiert eine Linie die Größe des Steuerelements. Sobald Sie die linke Maustaste loslassen, fügt der Designer das Steuerelement in der vorgegebenen Größe in das Formularlayout ein. In Abbildung 9.3 sehen Sie im unteren Teil des Formulars eine mit diesen Schritten eingefügte Schaltfläche.

Tipp

Sie können in der Symbolleiste *Toolbox* die Schaltfläche des gewünschten Steuerelements per Mausklick anwählen und anschließend im Formular klicken. Dann fügt der Designer das Steuerelement automatisch mit einer Standardgröße an dieser Position ein. Ein Doppelklick auf eine Schaltfläche der *Toolbox* fügt das Steuerelement ebenfalls im Formularentwurf ein.

Im Formularentwurf eingefügte Steuerelemente lassen sich anschließend markieren und recht komfortabel per Maus bearbeiten:

- Zum Markieren klicken Sie einfach das betreffende Steuerelement per Maus an. Mehrere Steuerelemente lassen sich markieren, indem Sie die **[Strg]**-Taste gedrückt halten und die Steuerelemente im Formular anklicken.
- Sie können die Größe des markierten Steuerelements ändern, indem Sie die Ziehmarken am Markierungsrand per Maus verschieben. Verschieben lassen sich markierte Steuerelemente, indem Sie deren Positionsrahmen per Maus an die gewünschte Stelle ziehen.
- Sind bereits Steuerelemente im Formular vorhanden? Verschieben Sie ein Steuerelement, blendet der Designer Hilfslinien zur Ausrichtung ein. Lassen Sie dann die Maustaste los, wird das Steuerelement automatisch in der horizontalen oder vertikalen Flucht des betreffenden, durch die Hilfslinien gekennzeichneten, Steuerelements ausgerichtet.
- Zum Ausrichten mehrerer Steuerelemente innerhalb des Formulars können Sie diese auch markieren. Anschließend verwenden Sie die Befehle im Menü *Format* bzw. die Schaltflächen der *Layout*-Symbolleiste. Dort finden Sie zusätzliche Optionen, um mehrere markierte Steuerelemente zu einer Gruppe zusammenzufassen oder um einzelne Steuerelemente zueinander oder gegenüber dem Formular auszurichten.

Über das Kontextmenü lassen sich zudem Befehle auf markierte Steuerelemente anwenden. Gelöscht wird ein Element, indem Sie dieses mit der rechten Maustaste anklicken und im Kontextmenü den Befehl *Löschen* wählen. Über die Tastenkombination **[Strg] + [C]** lässt sich ein markiertes Steuerelement in die Zwischenablage kopieren und mit **[Strg] + [V]** im Formularentwurf einfügen.

9.1.3 Eigenschaften von Formularen und Steuerelementen ändern

Der Formulardesigner vergibt Standardnamen (z.B. *Form1*) für die visuellen Elemente wie Formulare und Steuerelemente. Auch die anderen Eigenschaften werden mit Standardwerten belegt. In der Entwicklungsumgebung ist es aber sehr einfach, die Eigenschaften eines Elements (Formular, Steuerelement) zu ändern.

1. Markieren Sie das betreffende Element (Formular, Steuerelement) durch einen Mausklick im Entwurfsfenster.

2. Wechseln Sie zum Eigenschaftfenster (dieses lässt sich ggf. über die Schaltfläche *Eigenschaftfenster* der *Standard-Symbolleiste* bzw. über die Taste **[F4]** einblenden).
3. Passen Sie die gewünschte Eigenschaft (Abbildung 9.4) in der Liste der Eigenschaften an, indem Sie auf die betreffende Zeile klicken und den Wert überschreiben oder aus den angebotenen Optionen wählen.

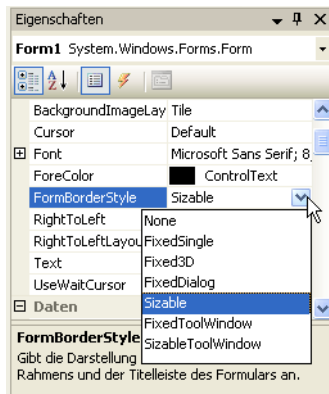


Abbildung 9.4: Eigenschaften eines Objekts

Um beispielsweise den Titeltext eines Formulars anzupassen, markieren Sie das Formular in der Design-Ansicht und ändern im Eigenschaftfenster die Eigenschaft *Text*. Möchten Sie die Beschriftung einer Schaltfläche anpassen, markieren Sie diese im Fenster des Formularsdesigners und setzen den Beschriftungstext in der Eigenschaft *Text*. Auf diese Weise können Sie die Elemente individuell gestalten. Tabelle 9.1 enthält eine Auflistung einiger Eigenschaften und deren Bedeutung.

Eigenschaft	Bedeutung
<i>AllowDrop</i>	Der Wert <i>True</i> signalisiert, dass das Steuerelement Daten akzeptiert, die der Benutzer per Drag&Drop zum Element zieht
<i>Anchor</i>	Diese Eigenschaft definiert die Verankerung an den Rändern (Top, Left, Right, Bottom) des umgebenden Containers. Standardmäßig sind Steuerelemente an den Rändern Top und Left verankert. Eine Verankerung an allen vier Rändern bewirkt, dass die Größe des Steuerelements an der Größe des umgebenden Containers angepasst wird.
<i>AutoSize</i>	Der Wert <i>True</i> erlaubt dem Steuerelement seine Größe automatisch am Inhalt anzupassen
<i>BackColor</i>	Legt die Hintergrundfarbe des Steuerelements fest
<i>Dock</i>	Definiert, welche Rahmen des Steuerelements an den umgebenden Container angebunden sind. Mit <i>Dock = Top</i> wird das Steuerelement am oberen Rand verankert. Mit <i>Dock = Fill</i> belegt das Steuerelement den kompletten Container und <i>None</i> löst den Dockmodus.
<i>Enabled</i>	Ermöglicht ein Steuerelement über die Werte <i>true</i> oder <i>false</i> freizugeben bzw. zu sperren
<i>ForeColor</i>	Legt die Vordergrundfarbe des Steuerelements für die Textanzeige fest
<i>MaximumSize</i>	Spezifiziert die maximale Größe des Steuerelements (als <i>Punkt</i> -Typ mit Breite und Höhe in Pixel)

Eigenschaft	Bedeutung
<i>MinimumSize</i>	Spezifiziert die minimale Größe des Steuerelements (als <i>Punkt</i> -Wert)
<i>Text</i>	Legt den im Element angezeigten (Titel-)Text fest
<i>Visible</i>	Erlaubt das Steuerelement über die Werte <i>true</i> und <i>false</i> ein- oder auszublenden

Tabelle 9.1: Eigenschaften von Steuerelementen und Formularen

Hinweis

Über im Kopfbereich des Eigenschaftensfensters eingeblendete Schaltflächen lassen sich die Eigenschaften nach Kategorien oder alphabetisch sortiert anzeigen. Benötigt eine Eigenschaft spezielle Vorgabewerte, wählen Sie diese über die Werte der betreffenden Listenelemente. Sobald Sie im Eigenschaftensfenster auf eine Eigenschaft klicken, wird zudem eine kurze Beschreibung im Fußbereich des Fensters eingeblendet.

9.1.4 Ein einfaches Formularbeispiel

Die auf den vorhergehenden Seiten erläuterten Techniken sollen jetzt zur Gestaltung eines einfachen Formulars verwendet werden. Dieses soll einen vorgegebenen Titeltext sowie eine OK-Schaltfläche aufweisen. Klickt der Benutzer auf diese Schaltfläche, erscheint ein Meldungsfeld und nach dem Schließen des Meldungsfelds durch den Benutzer soll auch das Formular ausgeblendet werden. Abbildung 9.5 zeigt die Visual-Studio-2005-Entwicklungsumgebung mit der Toolbox, den Fenstern des Ansicht-Designers, des Projektmappen-Explorers und der Eigenschaften im Hintergrund. Im Vordergrund ist das zur Laufzeit angezeigte Formular sowie das eingeblendete Dialogfeld zu sehen.

1. Um das Formular zu erstellen, legen Sie in der Entwicklungsumgebung ein neues Projekt über die Vorlage *Windows-Anwendung* an und öffnen dann das Element *Form1.vb* im Projektmappen-Explorer per Doppelklick.
2. Setzen Sie anschließend im Fenster des Formulardesigners die Formularabmessungen (durch Ziehen der Ränder per Maus) auf die gewünschte Größe. Um den Titeltext des Formulars zu ändern, klicken Sie dieses im Ansicht-Designer an und korrigieren die Eigenschaft *Text* im Fenster *Eigenschaften*. In Abbildung 9.5, Hintergrund, wurde der Standardtext »Form1« durch eine Zeichenkette ergänzt. Eine geänderte Eigenschaft wird direkt im Formulardesign angezeigt.
3. Wählen Sie in der Toolbox die Schaltfläche *Button* und fügen Sie eine Schaltfläche mit den gewünschten Abmessungen an der gewünschten Position im Formular ein. Anschließend klicken Sie die Schaltfläche im Fenster *Ansicht-Design* an und setzen im Fenster *Eigenschaften* die Eigenschaft *Text* auf den Wert »OK«.

Mit diesen Schritten haben Sie das gewünschte Formular entworfen. Wenn Sie das Projekt übersetzen und ausführen, wird bereits ein Formular mit der vorgegebenen Titellezeile und einer OK-Schaltfläche angezeigt. Über die *Schließen*-Schaltfläche in der rechten oberen Ecke des Fensters lässt sich das Formular auch schließen. Je nach Projektvorgaben (siehe *Kapitel 2*, Eigenschaft *Modus für das Herunterfahren*) wird zudem die Anwendung beim Schließen des Formulars u.U. automatisch beendet.

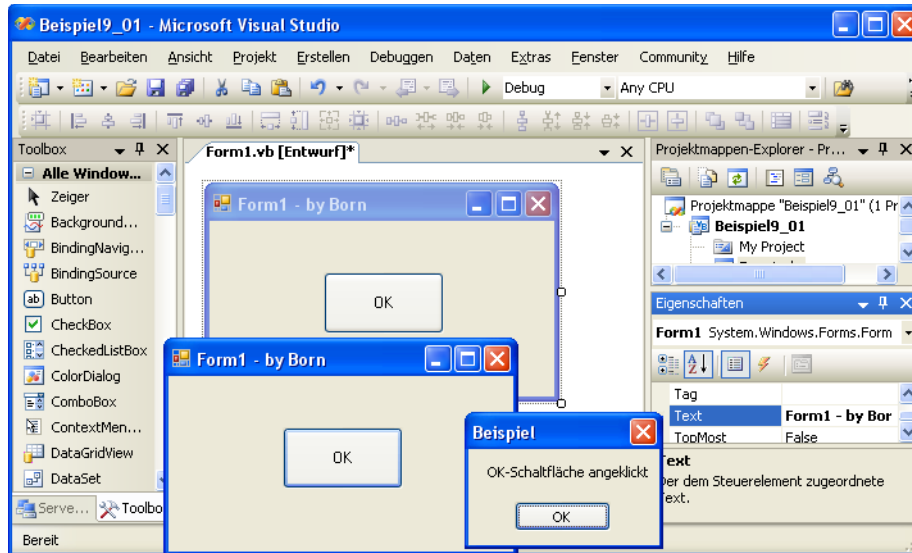


Abbildung 9.5: Projekt mit dem Beispielformular

Der Designer der Entwicklungsumgebung hat also den zur Formularanzeige benötigten Code selbständig generiert, d.h., Sie müssen sich um solche Fragen nicht weiter kümmern. Allerdings bleiben Mausklicks auf die OK-Schaltfläche folgenlos. Sie müssen also in einem weiteren Schritt die für das Formular geforderten Funktionen implementieren.

Hinweis

In Kapitel 7 wurde das neu eingeführte Konzept der *Partial*-Typen erwähnt, über das sich Klassendefinitionen über mehrere Dateien verteilen lassen. Dabei wird in der Datei mit den »ausgelagerten« Definitionen die Klasse mit dem Schlüsselwort *Partial* versehen. Visual Studio 2005 bzw. die Visual Basic 2005 Express Edition nutzt diesen Ansatz, um bei neuen Formularelementen den vom Designer generierten Visual-Basic-Code in eine separate Datei mit dem Namen `<Elementname>.Designer.vb` anzulegen. Das zugehörige Projektelement wird aber nur dann im Projektmappen-Explorer angezeigt, wenn Sie die Schaltfläche *Alle Dateien anzeigen* des Fensters aktiviert haben. Diese Aufteilung des Codes der Formalklasse auf zwei Dateien soll sicherstellen, dass der Entwickler möglichst nichts am vom Designer generierten Code ändert. Der Entwickler braucht ja nur Code für die Ereignisbehandlungsrouinen in der Klassendatei des Formulars (`Form1.vb` etc.) zu pflegen. In den betreffenden Klassendateien wird aber das optionale Schlüsselwort *Partial* in der Klassendefinition weggelassen. Konvertieren Sie ein mit einer älteren Visual Studio erstelltes Projekt mit Formularen in das Visual-Studio-2005-Format, verbleiben die vom Formulardesigner generierten Codeteile übrigens in der Klassendatei des Formulars – werden aber über die Gliederungsfunktion automatisch auf eine graue Kommentarzeile reduziert. Achten Sie beim Bearbeiten solcher Formularelemente darauf, dass Sie den vom Designer generierten Code nicht ändern – sonst kann es sein, dass der Formulardesigner das Formular nicht mehr anzeigen kann.

Ereignisprozeduren im Formular hinterlegen

Der Code zur Implementierung der im Formular geforderten Funktionen wird in Visual Basic 2005 in der Regel in sogenannten Ereignisbehandlungsprozeduren hinterlegt. Dies sind Prozeduren, die immer dann aufgerufen werden, wenn ein bestimmtes Ereignis (z.B. Laden eines Formulars, Mausklick auf eine Schaltfläche, Eingabe eines Werts etc.) auftritt.

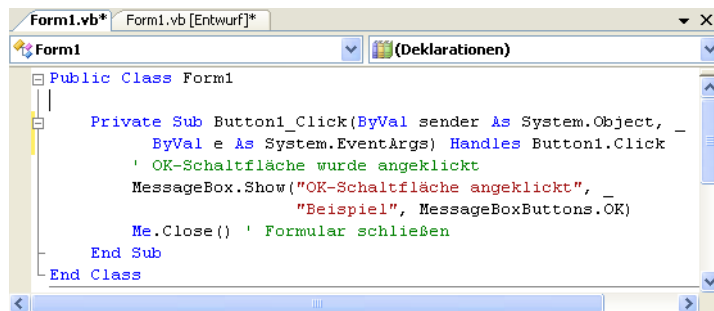


Abbildung 9.6: Codefenster mit der Ereignisbehandlungsprozedur des Beispielformulars

Im konkreten Formularbeispiel soll etwas beim Anklicken der OK-Schaltfläche des Formulars passieren. Um den Code für die betreffende Ereignisbehandlungsprozedur einzutippen, müssen Sie lediglich die betreffende Schaltfläche im Fenster *Ansicht-Designer* per Doppelklick anwählen. Die Entwicklungsumgebung öffnet dann das Codefenster und generiert automatisch den Prozedurrumpf für die betreffende Ereignisbehandlungsroutine. In Abbildung 9.6 sehen Sie den Inhalt des Codefensters mit dem für das Beispiel verwendeten Code. Tabelle 9.2 gibt die Bedeutung verschiedener Ereignisse an.

Ereignis	Bedeutung
<i>Click</i>	Wird beim Anklicken oder Anwählen des Steuerelements (per Maus/Tastatur) ausgelöst
<i>DoubleClick</i>	Wird beim Doppelklicken auf das Steuerelement ausgelöst
<i>FormClosing</i>	Wird beim Schließen eines Formulars ausgelöst
<i>GotFokus</i>	Das Steuerelement hat den Fokus erhalten
<i>KeyDown,KeyUp,KeyPress</i>	Wird beim Drücken oder Loslassen einer Taste ausgelöst
<i>Load</i>	Wird beim Laden eines Formulars ausgelöst
<i>LostFokus</i>	Wird ausgelöst, wenn das Steuerelement den Fokus verliert
<i>MouseMove</i>	Wird bei Mausbewegungen über dem Element ausgelöst.
<i>MouseClick</i>	Wird beim Anklicken des Steuerelements per Maus ausgelöst
<i>Paint</i>	Wird beim Neuzeichnen des Elements ausgelöst

Tabelle 9.2: Ereignisse und deren Bedeutung

Die Ereignisbehandlungsroutinen werden dabei in einer als Public definierten Klasse *Form1* hinterlegt. *Form1* ist der (über die *Name*-Eigenschaft des Formulars) definierte Objektname des Formulars. Im aktuellen Beispiel wurde eine Ereignisbehandlungsroutine für das *Click*-Ereignis durch den Designer im Codefenster eingefügt. Die betreffende Prozedur ist als *Private* deklariert, da ein Aufruf von außerhalb der Klasse nicht allzu sinnvoll ist. Der Name der Ereignisbehandlungsprozedur wird aus dem Objektnamen, einem Unterstrich und dem Ereignisnamen zusammengesetzt. Hier wurde der Objektname *Button1*, der sich ggf. über die *Name*-Eigenschaft der Schaltfläche anpassen lässt, benutzt. Das von der Prozedur behandelte Ereignis wird hinter dem Schlüsselwort *Handles* als Kombination aus dem Objektnamen und dem Ereignisnamen angegeben. Sie brauchen sich um die Feinheiten des Prozedurrumpfs nicht zu kümmern, da der Formulardesigner den Code automatisch generiert.

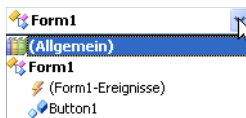


Abbildung 9.7: Listenfeld Klassenname des Codefensters

Hinweis

Sind in der Formulkasse sehr viele Ereignisbehandlungsroutinen (für verschiedene Steuerelemente) hinterlegt, können Sie im Codefenster das in der linken oberen Ecke gezeigte Listenfeld *Klassenname* öffnen (Abbildung 9.7). Dann werden alle in der Klasse hinterlegten Elemente (Objekte, Ereignisse etc.) eingeblendet. Durch Anklicken des gewünschten Eintrags lässt sich direkt im Codefenster zum zugehörigen Codeblock springen.

Falls Sie weitere Ereignisbehandlungsprozeduren für andere Ereignisse eines Objekts im Codefenster hinterlegen möchten, wählen Sie im Listenfeld *Klassenname* das gewünschte Objekt (z.B. *Button1*, falls sich das Ereignis auf ein bestimmtes Steuerelement des Formulars bezieht) oder ein Eintrag *Form1-Ereignisse* (falls sich das Ereignis auf das Formular selbst bezieht). Anschließend öffnen Sie das im Codefenster in der rechten oberen Ecke angezeigte Listenfeld *Methodenname* und wählen die zum gewählten Eintrag passende Methode (z.B. *Load*-Ereignis, welches beim Laden eines Formulars ausgelöst wird). Der Designer generiert dann im Codefenster den Prozedurrumpf der betreffenden Ereignisbehandlungsprozedur oder zeigt die bereits vorhandene Prozedur an. Sie brauchen dann nur noch den Code zur Implementierung der Funktionalität im Prozedurrumpf zu hinterlegen bzw. vorhandene Anweisungen anzupassen. Hinweise zu den jeweiligen Ereignissen finden Sie in der Hilfe sowie in den in den folgenden Kapiteln behandelten Beispielen.

An dieser Stelle noch einige Hinweise auf den Code innerhalb der *Click*-Ereignisbehandlungsroutine. Diese wird immer vom Laufzeitsystem aufgerufen, sobald der Benutzer die OK-Schaltfläche des Formulars anwählt. Im aktuellen Beispiel soll dann ein Meldungsfeld erscheinen. Dieses Meldungsfeld wird mit folgender Anweisung angezeigt:

```
MessageBox.Show("OK-Schaltfläche angeklickt", _
    "Beispiel", MessageBoxButtons.OK)
```

Details zur *MessageBox*-Klasse und der *Show()*-Methode finden Sie im Abschnitt »So setzen Sie die *MessageBox*-Klasse ein« in *Kapitel 8* sowie in der Hilfe. Falls Sie bisher mit der Visual-Basic-Kompatibilitätsklasse *MsgBox* gearbeitet haben, denken Sie daran, dass *MessageBox.Show()* eine etwas andere Reihenfolge für die Parameter benutzt – der Titeltext des Dialogfelds wird als zweiter Parameter erwartet und nicht als dritter Parameter wie bei *MsgBox()*. Der Aufruf der *MessageBox.Show()*-Methode bewirkt die Anzeige des Dialogfelds, wobei der Programmablauf bis zum Schließen des Dialogs unterbrochen wird.

Im aktuellen Beispiel soll dann aber auch das im Hintergrund des Dialogfelds sichtbare Formular verschwinden und die Anwendung soll terminieren. Dies lässt sich im Programmcode durch Verwendung der *Close()*-Methode des Formulars realisieren. Die betreffende Anwendung sieht folgendermaßen aus:

```
Me.Close() ' Formular schließen
```

Das Schlüsselwort *Me* besitzt dabei eine besondere Bedeutung. Es verhält sich wie eine Objektvariable, die auf die Instanz der Klasse (also hier das Formular) mit dem ausgeführten Code zeigt. Und damit ist der komplette Code des Formularbeispiels bereits fertig.

Hinweis

Sie finden die Projektdateien des Formularbeispiels im Ordner *Beisp\Kap09\Beispiel9_01* auf der Begleit-CD. Sie können das Projekt übersetzen und ausführen lassen. Dann wird das Formular angezeigt und beim Anklicken der OK-Schaltfläche erscheint das gewünschte Dialogfeld.

9.1.5 Formularbeispiel: Anpassen von Eigenschaften

Im vorherigen Beispiel haben Sie gesehen, wie sich ein einfaches Formular im Designer entwerfen lässt. Über das Eigenschaftenfenster lassen sich zudem die Formulareigenschaften wie Titeltext, Abmessungen, Farben etc. setzen. Die Eigenschaften eines Formulars lassen sich aber auch zur Laufzeit im Programmcode manipulieren (siehe auch folgende Abschnitte). In einem modifizierten Beispiel soll jetzt ein Formular (Abbildung 9.8) entworfen werden, welches einmal ein eigenes Symbol im Systemmenü aufweist und welches beim Laden die Formularposition per Code festlegt. Klickt der Benutzer auf die OK-Schaltfläche des Formulars, soll sich dessen Position und Größe ändern. Gleichzeitig wird die Position und die Bezeichnung der Schaltfläche modifiziert und eine andere Hintergrundfarbe für das Formular zugewiesen. Während der einzelnen Schritte sorgen Meldungsfelder dafür, dass der Programmablauf unterbrochen wird, damit Sie die Änderungen ansehen können. Nach dem Schließen des letzten Dialogs soll auch das Formular verschwinden.

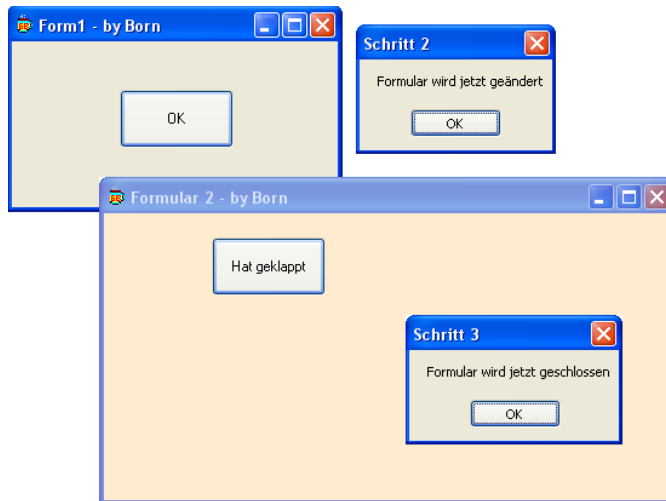


Abbildung 9.8: Formulardarstellung mit veränderten Eigenschaften

Formularentwurf mit Symbol im Systemmenü

Zum Entwerfen des Formulars gehen Sie wie im vorhergehenden Beispiel beschrieben vor. Sobald das Formular mitsamt der OK-Schaltfläche im Fenster des Ansicht-Designers entworfen wurde, können Sie dem Systemmenü das gewünschte Symbol zuweisen. Hierzu benötigen Sie eine .ico-Datei, wie sie auch für das Anwendungssymbol in der .exe-Datei benutzt wird. Bei Bedarf können Sie die Symboldatei in Visual Studio 2005 selbst entwerfen (siehe Kapitel 2, Abschnitt »Symboldatei für Windows-Anwendungen zuweisen«).

1. Stellen Sie im Fenster *Ansicht-Designer* sicher, dass das Formular angewählt ist und suchen Sie anschließend im Eigenschaftenfenster die Eigenschaft *Icon*.
2. Klicken Sie auf die in der Eigenschaftenzeile sichtbare Schaltfläche und wählen Sie im eingblendeten Dialogfeld *Öffnen* die Symboldatei (.ico).

Sobald das Dialogfeld *Öffnen* geschlossen wurde, wird das Symbol in der Eigenschaft (Abbildung 9.9) und im Systemmenü eingeblendet. Die Entwicklungsumgebung bindet dabei die Symboldatei als Ressource im Projekt ein und fügt diese beim Übersetzen zur .exe-Programmdatei hinzu.

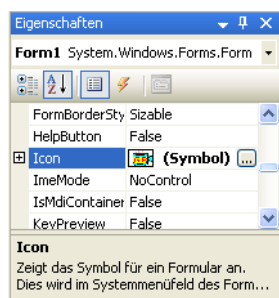


Abbildung 9.9: Zuweisen des Symbols zum Systemmenü

Hinweis

Sie können für das Systemmenü durchaus eine von der Anwendung abweichende Symboldatei im Dialogfeld *Öffnen* wählen. Beachten Sie aber, dass die meisten Windows-Anwendungen das gleiche Anwendungssymbol für die Programmdatei und für das Systemmenü verwenden, um dem Anwender die Wiedererkennung zu erleichtern.

Die Formulareigenschaften beim Laden ändern

Bereits beim Laden des Formulars können Sie dessen Eigenschaften wie Titeltext, Größe oder Position ändern. Hierzu müssen Sie den betreffenden Programmcode in der *Load*-Ereignisbehandlungsprozedur des Formulars hinterlegen. Die folgende Codesequenz zeigt den kompletten Code der Ereignisbehandlungsroutine:

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    ' wird beim Laden des Formulars aufgerufen
    MessageBox.Show("Formular wird jetzt geladen", _
        "Schritt 1", MessageBoxButtons.OK)

    With Me ' Formular auf Bildschirm zentrieren
        .StartPosition = FormStartPosition.CenterScreen
    End With
End Sub
```

Die Formularinstanz wird über das Schlüsselwort *Me* angesprochen. Durch die *With Me*-Konstruktion erreichen Sie, dass innerhalb des Konstrukts nur noch die Eigenschaften mit einem vorangestellten Punkt angegeben werden müssen (z.B. *.Text = "Hallo"* ändert den Formulartitel). In obiger Codesequenz wird lediglich die Eigenschaft *StartPosition* des Formulars mit einem Wert belegt. Das .NET Framework bietet für diesen Zweck eine *FormStartPosition*-Auflistung von Konstantenwerten (*CenterScreen*, *CenterParent*, *Manual* etc.) an. *CenterParent* zentriert das Formular innerhalb des dem übergeordneten Formulars (Parent). Mit *CenterScreen* soll das Formular eigentlich in der aktuellen Anzeige (Desktop) zentriert angezeigt werden. Allerdings wird das Formular unter Windows etwas versetzt angeordnet. Über den Wert *WindowsDefaultBounds* wird das Formular an der Windows-Standardposition positioniert und mit den im Windows-Standard festgelegten Begrenzungen angezeigt. Verwenden Sie den Wert *WindowsDefaultLocation*, um das Formular an der Windows-Standardposition mit den im Formular festgelegten Begrenzungen anzuzeigen. Mit dem Wert *Manual* erreichen Sie, dass die *Location*-Eigenschaft die Formularposition festlegt. Die folgenden zwei Codezeilen demonstrieren, wie sich der Konstantenwert nutzen lässt:

```
Me.StartPosition = FormStartPosition.Manual
Me.Location = New Point(10, 20)
```

Beachten Sie, dass die Eigenschaft *Location* ein Koordinatenpaar als Wert aufweist. Die Eigenschaft können Sie durch Verwendung einer Instanz des *Point*-Datentyps mit *New Point()* setzen.

Die Formular- und Schaltflächeneigenschaften zur Laufzeit ändern

Im hier benutzten Beispiel soll während der Laufzeit das angezeigte Formular etwas modifiziert werden. So ist der Titeltext, die Hintergrundfarbe des Formulars, die Lage und Größe des Formulars sowie die Lage der Schaltfläche zu verändern. Dies erfolgt innerhalb der *Click*-Ereignisprozedur der *OK*-Schaltfläche über folgende Codesequenz:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    ' wird beim Anklicken der OK-Schaltfläche aufgerufen
    ' Meldungsfeld anzeigen
    MessageBox.Show("Formular wird jetzt geändert", _
        "Schritt 2", MessageBoxButtons.OK)
    ' Jetzt die Formulareigenschaften anpassen
    With Me
        .Text = "Formular 2 - by Born" ' Titel anpassen
        .Width = .Width + 200         ' Breite anpassen
        .Height = .Height + 100       ' Höhe anpassen
        .BackColor = Color.BlanchedAlmond ' Hintergrundfarbe
        .Location = New Point(100, 100) ' Position ändern
    ' Jetzt die OK-Schaltfläche umbenennen
    With .Button1
        .Text = "Hat geklappt"
        .Location = New Point(.Left, .Top - 20) ' verschieben
    End With
    End With
    MessageBox.Show("Formular wird jetzt geschlossen", _
        "Schritt 3", MessageBoxButtons.OK)

    Me.Close() ' Formular schließen
End Sub
```

Der Titeltext des Formulars lässt sich über *Me.Text* direkt manipulieren. Ähnliches gilt für die Formularabmessungen, die über die Eigenschaften *Width* und *Height* zu erreichen sind. Die *Location*-Eigenschaft des Formularobjekts beeinflusst dessen Anzeigeposition und muss als *Point*-Wert angegeben werden.

Um auf die Eigenschaften der *OK*-Schaltfläche zuzugreifen, muss die Objekthierarchie des betreffenden Objekts berücksichtigt werden. Die Schaltfläche befindet sich im Formularcontainer und besitzt den Objektnamen *Button1*. Die Objekthierarchie zum Zugriff auf die *Text*-Eigenschaft lässt sich also zu *Me.Button1.Text* angeben. In der obigen Code-

sequenz wurde die *With*-Konstruktion verwendet, um sich die Angabe der jeweiligen Objekthierarchie zu sparen. Die beiden Anweisungen in der *With .Button1*-Konstruktion passen die Beschriftung der Schaltfläche an und positionieren diese innerhalb des Formularcontainers neu.

Hinweis

Sie finden die Projektdateien des Formularbeispiels im Ordner *Beisp\Kap09\Beispiel9_02* auf der Begleit-CD. Sie können das Projekt übersetzen und ausführen lassen. Dann wird das Formular angezeigt und beim Anklicken der OK-Schaltfläche angepasst. Während der Formularanzeige unterbrechen verschiedene Dialoge den Programmablauf. Schließen Sie diese Dialogfelder jeweils über die OK-Schaltfläche.

9.1.6 Beispiel zur Demonstration von Formularvarianten

Über die Eigenschaften des Formularobjekts lässt sich dessen Aussehen stark verändern (Abbildung 9.10). Sie können zum Beispiel erreichen, dass die Schaltflächen *Minimieren* und *Maximieren* in der Titelzeile wegfallen (Abbildung 9.10, mittleres Formular in der oberen Reihe). Oder das Formular wird mit einem Bild als Hintergrund versehen und transparent dargestellt (Abbildung 9.10, rechts oben). Mit einem kleinen Trick lassen sich sogar benutzerdefinierte Formulare ohne Fenster anlegen (Abbildung 9.10, rechts unten).

Die Nutzung dieser Formularvarianten soll jetzt an einem kleinen Beispiel demonstriert werden. Es wird dabei das in Abbildung 9.10, links oben, gezeigte Grundformular mit einem Hintergrundbild benutzt und im Programmcode entsprechend variiert. Um die folgenden Schritte nachzuvollziehen, legen Sie ein neues Projekt mit einem Formular an.

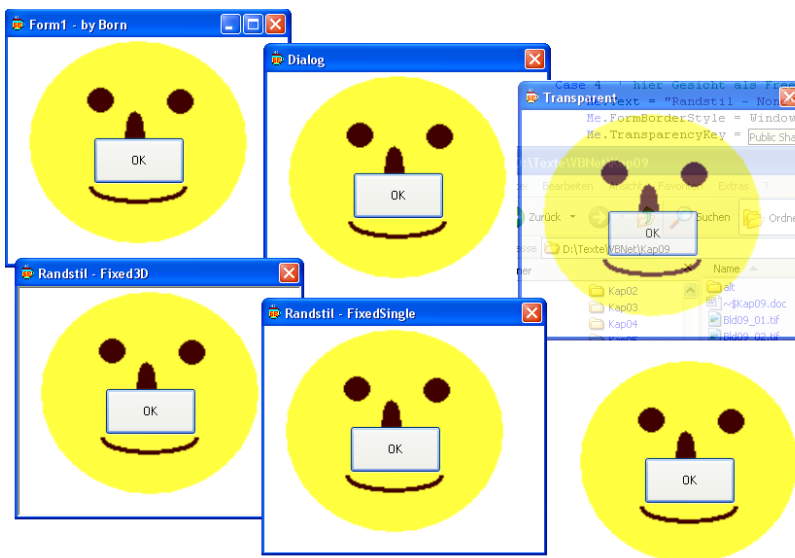


Abbildung 9.10: Formularvarianten

Formulare mit Hintergrundbild

Möchten Sie ein Formular mit einem (Hintergrund-)Bild versehen? Sie können einem Formular über die Eigenschaft *BackgroundImage* ein Hintergrundbild zuweisen.

1. Wählen Sie im Formulardesigner das Formular mit einem Mausklick an und wechseln Sie zum Eigenschaftensfenster.
2. Wählen Sie die Eigenschaft *BackgroundImage* an und klicken Sie auf die in der Eigenschaftenzeile eingblendete Schaltfläche, um das Dialogfeld *Ressource auswählen* zu öffnen (Abbildung 9.11, Hintergrund).
3. Anschließend wählen Sie im Dialogfeld *Ressource auswählen* eines der Optionsfelder *Lokale Ressource* oder *Projektressourcendatei*. Danach können Sie über die *Importieren*-Schaltfläche eine Bitmapdatei (.bmp) über ein Dialogfeld wählen und als Bild in einer Ressourcendatei oder im Projekt einbetten.
4. Sobald Sie das Dialogfeld über die *OK*-Schaltfläche schließen, wird das Bitmap-Bild im Formular eingeblendet. Sie sollten dann noch über die Eigenschaft *BackGroundImageStyle* vorgeben, ob das Bild zentriert, gekachelt oder gestreckt im Formular darzustellen ist.

Nach diesen Vorbereitungen können Sie das Formular, wie auf den vorhergehenden Seiten gezeigt, mit Steuerelementen versehen.

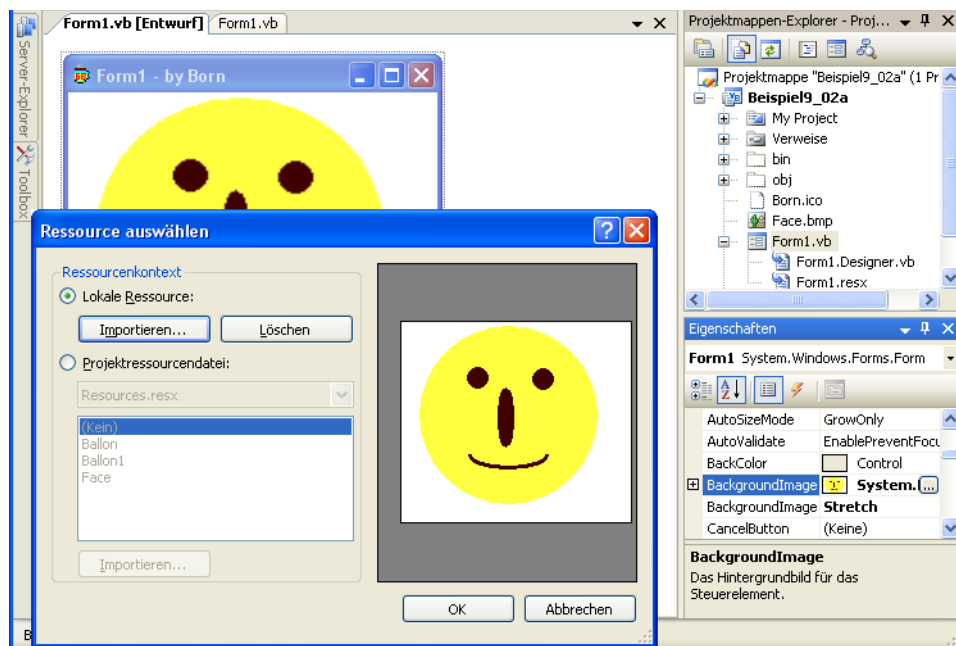


Abbildung 9.11: Hintergrundbild in Formular einbinden

Schaltflächen Minimieren/Maximieren im Formular ausblenden

Standardmäßig versieht die Entwicklungsumgebung ein Formular mit allen Elementen eines Windows-Fensters. Neben dem Systemmenü weist die Titelleiste noch die drei Schaltflächen *Minimieren*, *Maximieren* und *Schließen* auf. Zudem wird das Formular als Schaltfläche in der Windows-Taskleiste angezeigt. Windows-Dialogfelder werden aber weder in der Taskleiste eingeblendet noch besitzen Sie die Schaltflächen *Minimieren/Maximieren*. Diese Elemente lassen sich aber zur Entwurfszeit oder zur Laufzeit über bestimmte Eigenschaften abschalten.

Eigenschaft	Bedeutung
MinimizeBox	Der Wert <i>false</i> blendet die <i>Minimieren</i> -Schaltfläche im Fenster aus
MaximizeBox	Der Wert <i>false</i> blendet die <i>Maximieren</i> -Schaltfläche im Fenster aus
ShowIcon	Der Wert <i>false</i> blendet das Symbol des Systemmenüs im Fenster aus
ShowInTaskbar	Der Wert <i>false</i> bewirkt, dass bei einem minimierten Fenster kein Symbol in der Windows-Taskleiste erscheint
WindowState	Über die Werte <i>Normal</i> , <i>Minimized</i> und <i>Maximized</i> lässt sich festlegen, ob das Formular bereits als minimiertes oder maximiertes Fenster erscheint

Tabelle 9.3: Eigenschaften für Formularobjekte

Sie können die Eigenschaften im Eigenschaftenfenster auf die betreffenden Werte einstellen. Alternativ besteht die Möglichkeit, zur Laufzeit die Eigenschaften über Programmcode anzupassen.

```
Me.Text = "Dialog"
Me.MinimizeBox = False
Me.MaximizeBox = False
Me.ShowInTaskbar = False ' kein Symbol in Taskleiste
```

Die obigen Anweisungen verändern den Titeltext des Formulars, blenden die *Minimieren-/Maximieren*-Schaltflächen aus und verhindern, dass das Fenster eine Schaltfläche in der Taskleiste erhält.

Tipp

Für Windows-Dialoge empfiehlt es sich ein neues Element im Projektmappen-Explorer (Kontextmenübefehl *Hinzufügen/Neues Element* der Projektdatei) hinzuzufügen. Wenn Sie im Dialogfeld *Neues Element hinzufügen* (Abbildung 9.15) die Vorlage *Dialogfeld* wählen, wird dem Projekt eine Formularklasse zugewiesen, die bereits alle Windows-Einstellungen für Dialoge aufweist.

Transparente Formulare

Für spezielle Effekte gibt es die Möglichkeit, mit transparenten Formularfenstern zu arbeiten. Je nach gesetztem Transparenzwert wird der durch das Formular verdeckte Hintergrund mehr oder weniger sichtbar. Dies kann über die Eigenschaft *Opacity* des Formulars gesteuert werden. Sie finden diese Eigenschaft bei Auswahl des Formulars im Eigenschaftsfenster. Dort lassen sich Opazitätswerte (Transparenzwerte) von 0 bis 100% vorgeben. Ein Wert von 100% bedeutet, dass das Formular 100 Prozent deckend gezeichnet wird. Ab 50 Prozent kommt der Hintergrund bereits deutlich heraus, so dass das Formular nur noch schemenhaft zu sehen ist. Möchten Sie die Transparenz zur Laufzeit ändern, verwenden Sie folgende Anweisung:

```
Me.Opacity = 0.7
```

Beachten Sie, dass der Eigenschaftswert als *Double* vereinbart ist und im Bereich zwischen 0.0 und 1.0 liegen darf. Weisen Sie irrtümlich einen Wert von 70 (für 70 %) der Eigenschaft zu, wird die Transparenz nie angezeigt.

Formularstile nutzen

Formulare lassen sich mit verschiedenen Randstilen darstellen. Hier kommt die Eigenschaft *FormBorderStyle* zum Einsatz, die sich im Eigenschaftsfenster oder zur Laufzeit mit vorgegebenen Werten aus der Enumeration *Windows.Forms.FormBorderStyle* belegen lässt. Die folgende Anweisung

```
Me.FormBorderStyle = Windows.Forms.FormBorderStyle.Fixed3D
```

setzt den Formularstil auf *Fixed3D*. Möchten Sie verhindern, dass der Benutzer die Größe des Formulars ändern kann? Dann wählen Sie einen der »Fixed«-Randstile (*Fixed3D*, *FixedDialog*, *FixedSingle*). Nur wenn die Eigenschaft *FormBorderStyle* auf einen der *Sizeable*-Werte eingestellt ist, lässt sich die Formulargröße vom Benutzer per Maus ändern.

Benutzerdefiniertes Formular ohne Fenster

Möchten Sie Formulare mit selbst gestalteten Formen verwenden, die ohne die üblichen Windows-Fenster auskommen? Auch dies ist kein größeres Problem, die meisten Techniken haben Sie mit den obigen Schritten bereits kennen gelernt.

1. Legen Sie ein Formular mit einem Hintergrundbild im Fenster *Ansicht-Design* an (siehe Ausführungen auf den vorherigen Seiten).
2. Setzen Sie anschließend den Fensterstil des Formulars über die Eigenschaft *FormBorderStyle* auf den Wert *None*.
3. Im dritten Schritt müssen Sie der Eigenschaft *TransparencyKey* die Hintergrundfarbe außerhalb des Bereichs der selbstdefinierten Formularform zuweisen.

Sie können die Eigenschaften direkt im Eigenschaftsfenster der Entwicklungsumgebung anpassen. Oder Sie setzen die Werte zur Laufzeit. Die folgende Codesequenz demonstriert, wie sich der Stil und die Transparenzfarbe vorgeben lässt:

```
Me.FormBorderStyle = Windows.Forms.FormBorderStyle.None  
Me.TransparencyKey = Color.White ' Weiß = Transparent
```

Hier wird die Farbe Weiß als Hintergrundfarbe benutzt und auf transparent gestellt. Bei dem eingebundenen Smiley ist dieses dann nur noch als Formular sichtbar.

Tipp

Bei der Gestaltung des Bitmaps sollten Sie darauf achten, dass die Farben Weiß oder Schwarz möglichst nicht als Hintergrundfarbe benutzt werden. Andernfalls tritt meist der Effekt auf, dass in weißer oder schwarzer Farbe gezeichnete Schriften im Formular ebenfalls transparent gestaltet werden.

Hinweis

Sie finden die Dateien eines Projektbeispiels im Ordner `\Beisp\Kap09\Beispiel9_02a` der Begleit-CD. Das Formular besitzt eine Schaltfläche, die bei jedem Mausklick eine andere Formularvariante zur Anzeige bringt. Der Code innerhalb der *Click*-Ereignisbehandlungsprozedur verwendet eine *Select Case*-Anweisungsfolge mit der global in der Klasse definierten und in der *Load*-Ereignisbehandlungsprozedur auf den Wert 1 gesetzten Variablen *Schritt*, um die verschiedenen Zustände zu durchlaufen.

9.1.7 Ein Formular mit Schaltflächen, Beschriftung und Textfeld

Nach diesen vorbereitenden Beispielen ist es an der Zeit, ein erweitertes Formular zu erstellen. Dieses soll gemäß Abbildung 9.12 eine Beschriftung, ein Textfeld sowie zwei Schaltflächen aufweisen. Wird die *OK*-Schaltfläche angeklickt, ist die Benutzereingabe im Textfeld in einem Dialogfeld einzublenden, während die *Schließen*-Schaltfläche das Formular sofort ausblendet.

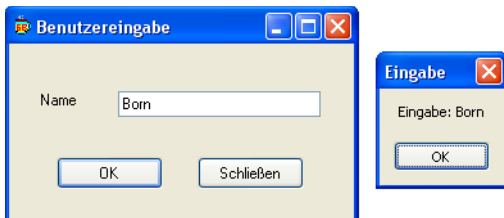


Abbildung 9.12: Formular mit Textfeld und Schaltflächen

Das Beispiel lässt sich mit wenigen Schritten gemäß den Anleitungen der vorherigen Seiten in der Entwicklungsumgebung erstellen.

1. Legen Sie ein neues Projekt als Windows-Anwendung im Fenster der Entwicklungsumgebung an.
2. Anschließend fügen Sie über die Toolbox die beiden Schaltflächen, ein Bezeichnungsfeld und ein Textfeld im Formular ein.
3. Passen Sie danach die Eigenschaften der Steuerelemente nach den Anforderungen an und ergänzen Sie danach die Ereignisprozeduren um den benötigten Code.

Bei den beiden Schaltflächen müssen Sie die *Text*-Eigenschaft im *Eigenschaften*-Fenster so anpassen, dass die Beschriftungen *OK* und *Schließen* angezeigt werden. Das Bezeichnungsfeld *Name* (auch als Label bezeichnet), lässt sich über das Werkzeug *Label* der Toolbox im Formular einfügen. Den Text für das Bezeichnungsfeld legen Sie über die *Text*-Eigenschaft fest. Bei Bedarf können Sie im Eigenschaftenfenster die Vorgaben für die Eigenschaften der Kategorie *Font* (*Name*, *Size*, *Bold* etc.) anpassen. Über die Eigenschaft *ForeColor* lässt sich die Schriftfarbe aus einer der für Windows-GUI-Elemente vordefinierten Farben anpassen.

Hinweis

Für jedes der im Formular eingefügten Steuerelemente lässt sich der Darstellungsstil über die Eigenschaft *FlatStyle* beeinflussen. In den Buchbeispielen wird der Wert »Standard« belassen, der ähnlich wie »System« die unter Windows gewohnte Darstellung zur Anzeige benutzt. Sie können aber auch Werte wie »Flat« oder »Popup« zuweisen, um eine abweichende Darstellung der Steuerelemente zu erreichen. Dies kann sinnvoll sein, wenn Sie .NET-Anwendungen für andere Geräteklassen entwickeln. Das Steuerelement wird über die *Anchor*-Eigenschaft an den Formularrändern verankert. Klicken Sie auf die Schaltfläche dieser Eigenschaft, wird standardmäßig eine Verankerung am oberen und linken Rand des übergeordneten Containers (hier des Formulars) angezeigt. Verankern Sie das Steuerelement zusätzlich am rechten und/oder unteren Rand, wird die Größe des Steuerelements bei Größenänderungen des übergeordneten Containers (z.B. des Formulars) angepasst. Schaltflächen sollten Sie daher nur am oberen und linken bzw. rechten Rand verankern. Um zu erreichen, dass das Textfeld des obigen Beispielformulars bei Größenänderungen des Formulars in der Breite mit angepasst wird, verankern Sie dieses am linken und rechten Rand. Die Verankerung am oberen Rand bewirkt, dass die Position des Textfelds zum Formulartitel bei Größenänderungen des Formulars erhalten bleibt.

Das Textfeld zur Aufnahme der Benutzereingaben wird über das Steuerelement *TextBox* der Toolbox im Formular eingefügt. Der beim Aufruf des Formulars eingeblendete Wert lässt sich über die Eigenschaft *Text* vorgeben. Das Textfeld wird über dessen *Anchor*-Eigenschaft am oberen, linken und rechten Formularrand verankert, um dessen Größe automatisch an die Formularbreite anzupassen.

Kommen wir noch kurz auf die Ereignisbehandlung für die Mausklicks auf die beiden Schaltflächen zu sprechen. Hier sehen Sie den Code für die Behandlung des *Click*-Ereignisses der *OK*-Schaltfläche. Die zugehörige Objektvariable ist mit *Button1* benannt:

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
    ' wird beim Anklicken der OK-Schaltfläche aufgerufen  
    ' Meldungsfeld anzeigen  
    MessageBox.Show("Eingabe: " & Me.TextBox1.Text, _  
        "Eingabe", MessageBoxButtons.OK)  
End Sub
```

Innerhalb der Prozedur wird der Wert des Textfelds *TextBox1* über die *Text*-Eigenschaft gelesen. Der Bezug auf das Formular erfolgt über das Schlüsselwort *Me*. Der Text wird dann über die *Show()*-Methode der *MessageBox*-Klasse in einem Dialogfeld angezeigt. Die Prozedur für die Behandlung des *Click*-Ereignisses der Schaltfläche *Schließen* besitzt die gleiche Struktur, enthält aber lediglich den Befehl

```
Me.Close()
```

um das Formular über die *Close()*-Methode zu schließen. Die Methode blendet das Formular aus, lässt Ihnen aber noch die Möglichkeit, über die *FormClosing*-Ereignisbehandlungsprozedur auf das Schließen zu reagieren (siehe Abschnitt »Auf das Schließen des Formulars reagieren« am Kapitelende).

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung realisierten Beispiels im Ordner `\Beisp\Kap09\Beispiel9_03` der Begleit-CD.

9.2 Spezielle Techniken für Formulare

In diesem Abschnitt möchte ich Ihnen einige Techniken vorstellen, die zur Verwaltung von Formularen oder zu deren Gestaltung recht hilfreich sind.

9.2.1 Formular programmgesteuert anzeigen

Die mit Visual Basic 2005 Express Edition bzw. Visual Studio 2005 mitgelieferten Projektvorlagen für Windows-Anwendungen haben einen gravierenden Nachteil: Die Voreinstellungen gehen davon aus, dass die Anwendung direkt mit der Anzeige des Formulars startet. Der gesamte Programmcode muss dann über die Ereignisbehandlungsroutinen des Formulars verwaltet werden. In der Praxis gibt es aber häufiger den Fall, dass eine Anwendung verschiedene Formulare, abhängig von Bedingungen, einblenden soll. Oder Formulare sind gezielt während des Programmablaufs einzublenden. In einem neuen Beispiel soll jetzt eine Windows-Anwendung erstellt werden, die nach dem Start dem Benutzer über einen Zusatzdialog die Wahl lässt, ob ein Formular anzuzeigen ist oder nicht. Zur Realisierung des Projekts gehen Sie in folgenden Schritten vor:

1. Legen Sie in der Entwicklungsumgebung ein neues Projekt mit der Vorlage *Windows-Anwendung* an.
2. Wählen Sie das Formularelement im Projektmappen-Explorer an und gestalten Sie das Formular im Fenster des Formulardesigners. Ergänzen Sie ggf. im Codefenster die Anweisungen für die Ereignisbehandlungsroutinen der Formularelemente.
3. Fügen Sie im Projektmappen-Explorer ein neues Element (entweder als Class oder als Module) ein. Da dieses neue Element den Startcode der Anwendung aufnehmen soll, fügen Sie eine Prozedur *Sub Main()* hinzu. Falls Sie eine Klasse verwendet haben, muss die Prozedur mit dem Schlüsselwort *Shared* versehen werden (damit der öffentliche Member freigegeben wird, siehe Kapitel 7). Anschließend ergänzen Sie diese Prozedur um die Anweisungen zur Realisierung des Startcodes (Details siehe unten).

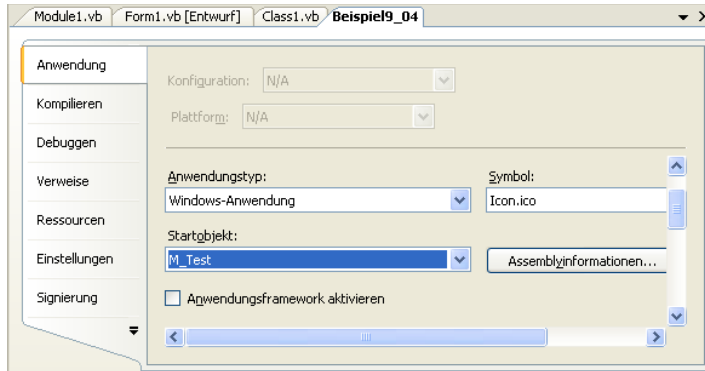


Abbildung 9.13: Projekteigenschaften für das Beispiel

4. Rufen Sie das Eigenschaftenfenster des Projekts auf und wechseln Sie zur Seite *Anwendung* (Abbildung 9.13). Dort müssen Sie zuerst die Markierung des Kontrollkästchens *Anwendungsframework aktivieren* löschen. Anschließend stellen Sie den Wert des Listenfelds *Startobjekt* von »Form1« auf den Namen des Moduls bzw. der Klasse um.

Mit dem letzten Schritt erreichen Sie, dass die Entwicklungsumgebung beim Übersetzen des Projekts keinen Startcode zum automatischen Aufruf des Formulars generiert, sondern die Prozedur in dem von Ihnen hinzugefügten Modul bzw. der Klasse aufruft.

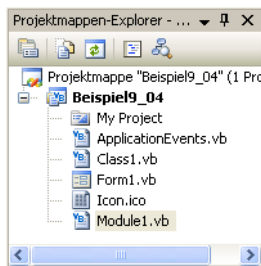


Abbildung 9.14: Projektmappen-Explorer mit Elementen des Beispiels

In Abbildung 9.14 sehen Sie die Ansicht des Projektmappen-Explorers mit den Elementen des Beispiels. Zur Demonstration habe ich sowohl ein *Class*-Element als auch ein *Modul*-Element zum Aufruf des Formulars eingefügt. Je nach eingestellten Projekteigenschaften wird dann die Prozedur *Main* des Elements *Class1.vb* oder *Module1.vb* beim Anwendungsstart aufgerufen.

Jetzt bleibt nur noch die Implementierung des Startcodes zum Aufruf des Formulars im Modul oder in der Klasse. Die Methoden zum Erzeugen von Formularen werden vom .NET Framework über eine eigene Klasse *Forms* im Namensraum *System.Windows.Forms* bereitgestellt. Sie brauchen sich aber um den Import des Namensraums nicht zu kümmern, da dies bereits über die Projekteinstellungen einer Windows-Anwendung erfolgt. Zudem ist es ganz praktisch, dass das über die Projektvorlage eingefügte Formular *Form1.vb* bereits als Standardinstanz vorliegt. Dadurch können Sie direkt über den

Objektnamen *Form1* auf das Formular zugreifen. Zur Anzeige einer Formularinstanz bietet das .NET Framework verschiedene Methoden.

- *ShowDialog()*: Diese Methode wird von der Klasse *Forms* bereitgestellt und sorgt für die Anzeige eines modalen Dialogfelds. Der Programmablauf des rufenden Moduls wird während der Anzeige des Formulars unterbrochen.
- *Show()*: Diese Methode wird ebenfalls von der Klasse *Form* bereitgestellt und zeigt das Formular an. Dabei wird das Dialogfeld aber nichtmodal dargestellt, d.h., der Programmablauf wird im rufenden Modul nicht unterbrochen!

Hinweis

Die ebenfalls mögliche Variante *Application.Run (Formularobject)* sollte dagegen nicht zur Anzeige von Formularen benutzt werden.

Ich habe an dieser Stelle die zwei Varianten in der Startprozedur vereint. Beim Start des Programms fragt dieses den Benutzer über ein Dialogfeld mit *Ja/Nein*-Schaltfläche, welche Variante zur Formularanzeige genutzt werden soll. Die zur Anzeige des Dialogs verwendete *MessageBox.Show()*-Methode liefert den Rückgabecode der gewählten Schaltfläche. Abhängig vom Rückgabecode wird dann der Titeltext des Formulars über die *Text*-Eigenschaft modifiziert und anschließend die *Show()*- oder die *ShowDialog()*-Methode zur Anzeige des Formulars aufgerufen. Sie erkennen also an der Titelzeile des Formulars, welche Methode zum Formularaufruf benutzt wurde. Gleichzeitig wird der Unterschied zwischen den Methoden zur Formularanzeige deutlich. Während bei der *Show()*-Methode der Programmablauf fortgesetzt wird (der Abschlussdialog erscheint sofort), hält *ShowDialog()* den Programmablauf für die Dauer der Formularanzeige an (der Abschlussdialog erscheint erst nach dem Schließen des Formulars). Das folgende Codefragment zeigt die Anweisungen der Startprozedur:

```
Sub Main()          ' Startprozedur
  If MessageBox.Show( _
    "Formular mit Run anzeigen -> Ja anklicken", _
    "Formularanzeigemodus wählen", MessageBoxButtons.YesNo, _
    MessageBoxIcon.Question, MessageBoxDefaultButton.Button1) _
    = DialogResult.Yes Then
    Form1.Text = "Mit Run gestartet"
    Form1.Show()          ' über Show anzeigen
    MessageBox.Show("Fertig")
  Else
    Form1.Text = "Mit ShowDialog gestartet"
    Form1.ShowDialog()    ' als modalen Dialog anzeigen
    MessageBox.Show("Fertig")
  End If
End Sub
```

Listing 9.1: Startprozedur zum Aufruf des Formulars

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung realisierten Beispiels im Ordner `\Beisp\Kap09\Beispiel9_04` der Begleit-CD. Standardmäßig ist das Modul *Module1.vb* als Startobjekt im Projekt eingerichtet. Sie können die Projekteinstellungen auf der Seite *Anwendung* aber so ändern, dass die Klasse *Class1.vb* als Startobjekt benutzt wird.

Ein Formular nach dem Schließen wieder einblenden

Um ein Formular mehrfach einzublenden, verwenden Sie die *ShowDialog()*-Methode (der mit dem *New*-Konstruktor erzeugten Formularinstanz). Die Formularinstanz wird beim Klicken auf die *Schließen*-Schaltfläche bzw. beim Ausführen der *Close()*-Methode zwar verworfen. Beim nächsten Aufruf der *ShowDialog()*-Methode wird es aber erneut aus der Klasse erzeugt. Die im nachfolgenden Listing gezeigte Codesequenz demonstriert diesen Sachverhalt. Das Formular wird nach dem Schließen ein weiteres Mal als modales Dialogfeld eingeblendet.

```
'*****
' File/Projekt:  Beispiel9_04a
' Autor: G. Born www.borncity.de
' Demonstriert das erneute Einblenden eines
' Formulars
'*****
Option Strict On
Class C_Test
    Shared Sub Main() ' Startprozedur
        Dim oForm As New Form1() ' Formularobjekt anlegen

        oForm.Text = "Mit ShowDialog gestartet"
        oForm.ShowDialog() ' als modalen Dialog anzeigen
        MessageBox.Show("Text: " & oForm.TextBox1.Text)
        oForm.Text = "Mit ShowDialog erneut angezeigt"
        oForm.ShowDialog() ' als modalen Dialog anzeigen
        MessageBox.Show("Text: " & oForm.TextBox1.Text)
    End Class
```

Listing 9.2: Beispielcode zum erneuten Einblenden eines Formulars

Hinweis

Im betreffenden Formular wurde übrigens ein Textfeld eingefügt. Dessen Inhalt wird nach dem Schließen des Formular in dem durch *MessageBox.Show()* angezeigten Dialog ausgegeben. Sie finden die Projektdateien des als Windows-Anwendung realisierten Beispiels im Ordner `\Beisp\Kap09\Beispiel9_04a` der Begleit-CD.

9.2.2 Ein neues zusätzliches Formular im Projekt anlegen

Möchten Sie ein weiteres Formular zur Anwendung hinzufügen, ist dies kein großer Aufwand. Sie können dabei sowohl mit einem neuen leeren Formular starten als auch ein bereits existierendes Formular als Basis für ein neues abgeleitetes Formular verwenden (hier kommt die in *Kapitel 7* erwähnte Vererbung ins Spiel). Ein neues Formular legen Sie mit folgenden Schritten an:

1. Wählen Sie im Menü *Projekt* der Entwicklungsumgebung den Befehl *Windows Form hinzufügen*. Alternativ können Sie im Projektmappen-Explorer die Projektdatei mit der rechten Maustaste anklicken und die Befehle *Hinzufügen/Windows Form* wählen. Visual Studio öffnet ein Dialogfeld zur Auswahl der Formularvorlage.
2. In diesem Dialogfeld wählen Sie die gewünschte Vorlage (z.B. *Windows Form*) aus, passen ggf. den Formularnamen an (Abbildung 9.15) und klicken dann auf die Schaltfläche *Hinzufügen*.

Die Entwicklungsumgebung (z.B. Visual Studio 2005) erzeugt dann automatisch ein neues Formular aus der gewählten Vorlage, legt die betreffenden Dateien im Projektordner an und blendet das Formular im Fenster des Designers ein.

Tipp

Neben einfachen *Windows Form*-Vorlagen bietet die Entwicklungsumgebung spezielle Vorlagen für Dialogfelder, Begrüßungsbildschirme und Infofelder.

Formular von Vorlagen ableiten

Haben Sie dagegen bereits ein Formular entworfen und soll dieses eventuell in einer modifizierten Variante neu genutzt werden? Dann leiten Sie das neue Formular einfach von diesem bestehenden Formular ab.

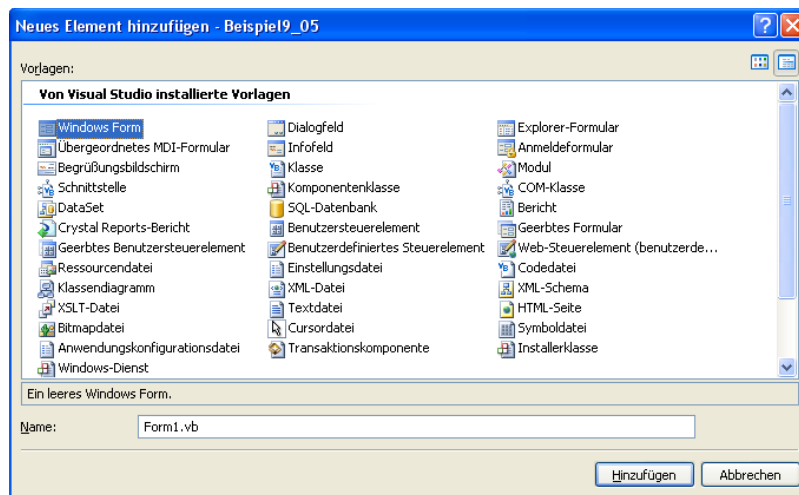


Abbildung 9.15: Auswahl der Vorlage im Visual-Studio-.NET-Formulardesigner

1. Gehen Sie wie oben beim Einfügen eines neuen Formulars gezeigt vor, um das Dialogfeld zur Auswahl der Formularvorlage zu öffnen.
2. Im dann angezeigten Dialogfeld (Abbildung 9.15) ist die Vorlage *Geerbtes Formular* auszuwählen, ggf. der Formularname anzupassen und dann auf die Schaltfläche *Hinzufügen* zu klicken.
3. Die Entwicklungsumgebung zeigt anschließend ein Dialogfeld mit den für die Vererbung verfügbaren Formularen (Abbildung 9.16). Wählen Sie ggf. ein Formular aus, und klicken Sie dann auf die OK-Schaltfläche.

In diesem Fall wird das neue Formular von der Formularklasse abgeleitet und besitzt deren Eigenschaften. Das neue abgeleitete Formularelement erscheint im Projektmappen-Explorer und im Fenster des Formulardesigners unter dem neu gewählten Namen.

Achtung

Die Elemente in geerbten Formularen lassen sich nicht im Eigenschaftfenster anpassen. Sie müssen vielmehr das Elternformular bearbeiten. In ein abgeleitetes Kindformular können Sie aber zusätzliche Steuerelemente aufnehmen. Falls eine im Elternformular vorgenommene Änderung im Kindformular nicht dargestellt wird, müssen Sie das Projekt neu erstellen lassen.

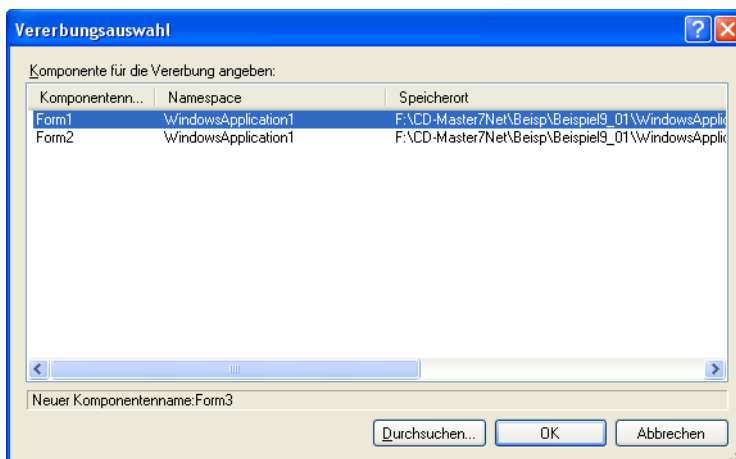


Abbildung 9.16: Auswahl der Komponente für die Vererbung

Formulare aus dem Projekt löschen

Die dem Projekt hinzugefügten Formulare werden im Fenster des Projektmappen-Explorers mit ihren Dateinamen aufgelistet. Sie können also jederzeit die betreffenden Komponenten anwählen. Haben Sie ein Formular im Projekt eingefügt, stellen aber fest, dass dieses nicht benötigt wird, können Sie das komplette Modul aus dem Projekt entfernen:

1. Klicken Sie den Eintrag für das betreffende Symbol im Fenster des Projektmappen-Explorers mit der rechten Maustaste an.
2. Wählen Sie im Kontextmenü den Befehl *Löschen* und bestätigen Sie ggf. die Warnung, dass der komplette Code aus dem Projekt gelöscht wird, über die *OK*-Schaltfläche.

Dieser Schritt bereinigt den Projektordner und sorgt auch dafür, dass kein unbenutzter Formularcode in der zu erstellenden Anwendung verbleibt.

Unterformulare aufrufen

Haben Sie ein weiteres Formular zum Projekt hinzugefügt, müssen Sie einen Weg schaffen, der dem Benutzer erlaubt, dieses Formular aufzurufen. Sie können den auf den vorhergehenden Seiten gezeigten Weg wählen und den Formularaufruf über eine Startprozedur mittels der *ShowDialog()*-Methode vornehmen. Meist wird man aber dem Benutzer die Möglichkeit bieten wollen, um ein Unterformular durch Anklicken von Schaltflächen aufzurufen. Dann lässt sich der Code zum Aufruf des Unterformulars in der Ereignisbehandlungsroutine des *Click*-Ereignisses der betreffenden Schaltfläche hinterlegen. Der nachfolgende Codeausschnitt zeigt die Anweisungen, um zwei Unterformulare aus einem Hauptformular durch Anklicken einer Schaltfläche anzuzeigen:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    Dim oForm As New Form2
    oForm.Show() ' erstes Unterformular
    ' die folgenden Anweisungen werden auch bei
    ' geöffnetem Unterformular sofort ausgeführt!
    Dim oForm1 As New Form3
    oForm1.ShowDialog()
    MessageBox.Show("Formular3 geschlossen", "Formularbeispiel", _
        MessageBoxButtons.OK)
End Sub
```

Listing 9.3: Aufruf von Unterformularen in einer Click-Ereignisbehandlung

Im Code wird jeweils eine Instanz des betreffenden Unterformulars über den *New*-Konstruktor erzeugt und einer Objektvariablen zugewiesen. Zum Aufruf des ersten Unterformulars kommt die *Show()*-Methode zum Einsatz. Sobald das Formular angezeigt wird, gibt die Methode die Kontrolle an das rufende Programm zurück. Dort werden dann die Anweisungen zum Aufruf des zweiten Unterformulars ausgeführt. Die benutzte *ShowDialog()*-Methode unterbricht den Programmablauf bis zum Schließen des Formulars. Beim Ausführen des Beispiels sehen Sie daher die zwei Unterformulare gleichzeitig. Nach dem Schließen des zweiten Unterformulars wird der Programmablauf in der *Click*-Ereignisbehandlungsroutine fortgesetzt. Dann erscheint das Meldungsfeld mit dem Hinweis, dass das Formular geschlossen wurde.

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung realisierten Beispiels im Ordner `\Beisp\Kap09\Beispiel9_05` der Begleit-CD. Das Element *Form3.vb* ist von *Form2.vb* abgeleitet. Sie können also die Eigenschaften der geerbten Steuerelemente nicht anpassen (sondern müssen diese im Elternformular *Form2.vb* ändern).

Die Projektdateien im Ordner `\Beisp\Kap09\Beispiel9_05a` der Begleit-CD gehören zu einem Beispiel, welches in einem Hauptformular mehrere Schaltflächen zum Aufruf von Unterformularen zeigt. Die Schaltfläche *Fensterstil* schaltet bei jedem Mausklick den Fensterstil des Hauptformulars um (es wird die betreffende Eigenschaft geändert – siehe auch weiter oben im Abschnitt »Formularstile nutzen«).

Die Formulareigenschaften (Position, Größe, Titeltext etc.) lassen sich dynamisch zur Laufzeit durch Änderung der betreffenden Eigenschaften anpassen. Das Gleiche gilt für die Steuerelemente, die im Formular positioniert sind. Lesen Sie notfalls am Kapitelanfang nach, wie auf die Eigenschaften zugegriffen werden kann.

Aktivierungsreihenfolge der Formularelemente festlegen

Wird ein Formular angewählt, erhält eines der Steuerelemente den Fokus. Durch Drücken der -Taste kann der Benutzer den Fokus auf weitere Steuerelemente setzen. Standardmäßig bekommt das erste im Formulardesigner eingefügte Steuerelement den Fokus, das nächste eingefügte Element erhält als Nächstes den Fokus und so weiter. Die Reihenfolge der beim Fokuswechsel angesprungenen Steuerelemente lässt sich aber über die Eigenschaft *TabIndex* festlegen. Der Wert 0 bezeichnet das erste Steuerelement, welches den Fokus erhält. Das nächste Steuerelement erhält den Wert 1 in der Eigenschaft *TabIndex*. Drückt der Benutzer die -Taste, erhalten die Steuerelemente in der vorgegebenen Reihenfolge den Fokus.

9.2.3 Formulare per Code erzeugen

Der Formulardesigner der Entwicklungsumgebung erlaubt Ihnen, Formulare per Drag&Drop zu gestalten. Der Designer generiert dann den zur Anzeige und zur Verwaltung des Formulars benötigten Code. Sie haben aber auch die Möglichkeit, Formulare zur Laufzeit dynamisch über Visual-Basic-Programmcode zu erzeugen. Weiter oben hatte ich bereits erläutert, dass die .NET-Framework-Klassenbibliothek eine eigene *Form*-Klasse im Namensraum *System.Windows.Forms* bereitstellt, mit der sich Formulare unter Windows anzeigen lassen. Sobald der Namensraum (z.B. über die *Imports*-Anweisung) bekannt ist, können Sie mit der Anweisung

```
Dim oForm As New Form()
```

ein Formular aus der Klasse *Form* als Objekt instantiieren. Über die Objektvariable *oForm* lässt sich dann auf die Eigenschaften des Formulars zugreifen. Sie können beispielsweise den Titeltext des Formulars (Eigenschaft *Text*), die Abmessungen (Eigenschaft *Size*), die Formularposition (Eigenschaft *Location*) etc. festlegen. Benutzen Sie dann z.B. die *Show*-

Dialog()-Methode, um das Formular zur Laufzeit anzuzeigen. Die folgende Sequenz an Anweisungen erstellt ein einfaches Formular:

```
Dim oForm As New Form()      ' Formular anlagen
With oForm                  ' Formulareigenschaften
    .Text = "Formular mit OK-Schaltfläche" ' Titeltext
    .Width = 250              ' Abmessungen
    .Height = 100
    .StartPosition = FormStartPosition.CenterScreen
    .CancelButton = b1        ' mit Esc belegen
    ' .AcceptButton = b1      ' mit Enter belegen
End With
```

Die betreffenden Techniken wurden auf den vorhergehenden Seiten in den dort behandelten Beispielen bereits erläutert. Neu ist lediglich die Eigenschaft *CancelButton*, der hier der Wert »b1« zugewiesen wird. Dieser Wert entspricht dem Objektnamen einer Schaltfläche (die noch in das Formular eingefügt wird). Die Eigenschaft legt das *Button*-Steuerelement fest, welches auf das Drücken der **[ESC]**-Taste reagieren soll. Alternativ gibt es die Eigenschaft *AcceptButton*, die die Schaltfläche (Button), welche auf das Drücken der **[↵]**-Taste reagieren soll, festlegt. Hier habe ich der Eigenschaft *CancelButton* die OK-Schaltfläche zugewiesen. Dies hat den Effekt, dass ein mit der *ShowDialog()*-Methode angezeigtes Formular automatisch bei einem Mausklick auf die Schaltfläche geschlossen wird. Sie können sich also die Anweisung *Me.Close()* zum Schließen des Formulars in der *Click*-Ereignisbehandlungsroutine sparen. Trotzdem besteht die Möglichkeit, eine *Click*-Ereignisbehandlungsprozedur für die betreffende Schaltfläche zu implementieren.

Um etwas Sinnvolles mit dem Formular anzustellen, muss das Formular mit Steuerelementen ausgestattet werden. Auch diese lassen sich über Programmanweisungen zur Laufzeit zum Formular hinzufügen. Sie erzeugen eine Objektinstanz des betreffenden Steuerelements und fügen diese dem übergeordneten Elternobjekt hinzu. Die Objektvariablen zur Instantiierung des Formulars und der Schaltfläche lassen sich mit folgenden Anweisungen vereinbaren:

```
Dim oForm As New Form()      ' Formular anlegen
Dim b1 As New Button()       ' Schaltfläche
```

Den Befehl zum Erzeugen eines Formulars kennen Sie schon. Eine Schaltfläche wird also ähnlich wie ein Formular durch Aufruf des *New*-Konstruktors der Klasse *Button* erzeugt. Kennen Sie die Klassen für weitere Steuerelemente, lassen sich deren Instanzen analog erzeugen. Anschließend besteht die Möglichkeit, über die betreffende Objektvariable auf die Eigenschaften und Methoden des betreffenden Objekts zuzugreifen. Beim Formular lassen sich beispielsweise Titeltext, Position oder Abmessungen vereinbaren. Für die Schaltfläche sind dessen Abmessungen sowie die Position innerhalb des Formulars als Eigenschaften anzugeben. Außerdem soll die Schaltfläche ja noch eine Beschriftung erhalten. Die folgenden Anweisungen definieren die wichtigsten Eigenschaften der Schaltfläche:

```
With b1 ' Schaltfläche definieren
.Text = "OK" ' Bezeichnung
.Name = "But1" ' Objektname
.Size = New Size (80, 25) ' Größe (Breite x Höhe)
.Location = New Point(80, 30) ' Position (x, y) in Pixel
End With
```

Hier wurde ein *With*-Block benutzt, um direkt auf die Member der Klasse zuzugreifen. Dies spart nicht nur Schreibarbeit, sondern erlaubt dem Compiler auch, kompakteren Code zu generieren. Die Schaltflächenbeschriftung lässt sich über die *Text*-Eigenschaft anpassen. Die Eigenschaft *Name* legt den Namen des Objekts fest. Hilfreich ist diese Eigenschaft, falls Sie eine Liste von Objekten (z.B. alle Steuerelemente eines Formulars) vorliegen haben und über den Namen auf die Objekte zugreifen wollen. Die Abmessungen der Schaltfläche sind in der Eigenschaft *Size* hinterlegt. Da es sich aber um zwei Werte (Breite und Höhe in Pixel) handelt, muss der Eigenschaft ein entsprechender Datentyp *Size* zugewiesen werden. Hierzu kann auf den *New*-Konstruktor der Klasse *Size* zurückgegriffen werden:

```
Objekt.Size = New Size (80, 25)
```

Der Konstruktor erwartet zwei Argumente, wobei das erste Argument die Breite in Pixel und der zweite Parameter die Höhe in Pixel definiert. Der Konstruktor liefert ein Objekt vom Typ *Size* zurück, welches die beiden Werte als Struktur enthält. Damit sich die Klasse *Size* auch nutzen lässt, muss der Namensraum *System.Drawing* angegeben oder über *Imports* vereinbart werden (was aber bei Windows-Anwendungen in der Projektvorlage bereits passiert ist). Alternativ können Sie auch auf die beiden Eigenschaften *Height* und *Width* zurückgreifen, um die Objektanmessungen anzugeben.

Die Position (genauer: die linke obere Ecke) eines sichtbaren Objekts lässt sich entweder über die beiden Eigenschaften *Left* und *Top* oder über die Eigenschaft *Location* vereinbaren. Die beiden Angaben

```
Objekt.Left = 80
Objekt.Top = 30
```

würden das betreffende Objekt also 80 Einheiten vom linken Rand und 30 Einheiten vom oberen Rand positionieren. Alternativ lässt sich die Position des sichtbaren Objekts über die Eigenschaft *Location* vereinbaren (was in obiger Codesequenz genutzt wird). Lassen Sie diese Eigenschaft unbelegt, verwendet die Klasse eigene Initialisierungswerte beim Instantiieren. Der Eigenschaft *Location* ist ein Wert vom Datentyp *Point* zuzuweisen. Verwenden Sie den *New*-Konstruktor der Klasse *Point* für diesen Zweck und übergeben Sie beim Aufruf die X- und Y-Position als Parameter:

```
Objekt.Location = New Point(80, 30)
```

Auch diese Klasse findet sich im Namensraum *System.Drawing*.

Hinweis

Positions- und Größenangaben bei Formularelementen werden in Visual Basic 2005 in Pixel angegeben. Der Nullpunkt des Koordinatensystems liegt dabei in der linken oberen Ecke und die Position bezieht sich immer auf die linke obere Ecke des Objekts. Die *Location*-Eigenschaft eines Formulars legt dessen Position relativ zum Desktop oder zum übergeordneten Anwendungsfenster (sofern dieses existiert) fest. Bei einem Steuerelement bezieht sich die *Location*-Eigenschaft auf das Koordinatensystem des Containers (z.B. Formulars), in dem das Steuerelement untergebracht ist.

Nun muss das Steuerelement noch zum Formular hinzugefügt werden. Dies erfolgt über die *Controls.Add()*-Methode.

```
oForm.Controls.Add(b1) ' Schaltfläche hinzufügen
```

Die Methode fügt die Objektinstanz der Schaltfläche zur *Controls*-Auflistung aller Steuerelemente im Formular hinzu. Damit ist das Steuerelement mit seinen Eigenschaften zur Laufzeit im Formular vorhanden.

Die Ereignisbehandlung in diesem Beispiel

Nachdem bereits auf den vorherigen Seiten der Aufruf des Formulars mittels *ShowDialog()* besprochen wurde, bleibt nur noch die Implementierung und Aktivierung der Ereignisbehandlungsroutine für die Schaltfläche. Ein Mausklick auf die Schaltfläche könnte in einer *Click*-Ereignisbehandlungsprozedur abgefangen werden. Hierzu müssen Sie folgenden Prozedurrumpf mit den spezifischen Anweisungen zur Behandlung des Ereignisses im Code einfügen:

```
Shared Private Sub Button1(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles b1.Click
    MessageBox.Show("Hat geklappt", "Test")
End Sub
```

Die betreffende Prozedur wurde hier mit dem Namen *Button1* versehen und als *Shared* deklariert. Damit lässt sie sich ohne Instantiierung mit *New* aufrufen. Die Übergabeparameter sind gemäß den Standardkonventionen festgelegt. Das erste Argument enthält das Objekt, welches das Ereignis ausgelöst hat und im zweiten Parameter finden sich die Argumente des Ereignisses. Diese Parameter brauchen Sie allerdings hier nicht zu interessieren, übernehmen Sie einfach den Inhalt der Klammer in Ihren Quellcode. Wichtig ist allerdings die Angabe *Handles b1.Click*. Diese Anweisung vereinbart, dass die betreffende Prozedur auf das *Click*-Ereignis des Objekts *b1* reagiert. *b1* ist aber die Objektvariable der Schaltfläche. Folglich sollte die Prozedur bei jedem Mausklick auf die OK-Schaltfläche des Formulars ausgeführt werden. Im konkreten Beispiel wird nur ein Meldungsfeld durch die Prozedur angezeigt (das Schließen des Formulars erfolgt ja automatisch durch die *CancelButton*-Eigenschaft).

Damit das vom Laufzeitsystem erkannte Ereignis auch an die richtige Ereignisbehandlungsprozedur weitergereicht wird, müssen Sie in der *Main()*-Prozedur vor dem Aufruf des Formulars vereinbaren, dass eine Ereignisbehandlung stattfindet. Dies kann mit folgender Anweisung erfolgen:

```
AddHandler b1.Click, AddressOf MyForm.Button1
```

Die Anweisung *AddHandler* verknüpft ein Ereignis (hier *b1.Click*) mit einer Ereignisbehandlungsroutine (siehe *Kapitel 7*). Der *AddressOf*-Operator erstellt einen sogenannten Funktionsdelegaten, was nichts anderes als ein Verweis auf eine Funktion ist. In obiger Anweisung wird der Name der Prozedur *Button1* als Adresse benutzt. Da sich diese Prozedur in der Klasse *MyForm* befindet, muss dies im Ausdruck mit angegeben werden. Lange Rede, kurzer Sinn, mit der obigen Anweisung erreichen Sie, dass die Anwendung beim Aufruf des Formulars auch die Ereignisbehandlungsroutine für die Schaltfläche kennt.

Da bei Anwendung von *ShowDialog()* die Eigenschaft *CancelButton* ein automatisches Schließen des Formulars bewirkt, enthält die Ereignisbehandlungsprozedur nur einen Aufruf zur Anzeige eines Meldungsfelds – der Benutzer erfährt beim Testen, ob das Ereignis ausgelöst wurde. Sie können in der Ereignisbehandlungsprozedur (des Hauptformulars) die Anweisung *Me.Close()* hinzufügen, um das laufende Formular zu beenden. Dies ist sinnvoll, wenn die Anwendung noch fortgeführt oder das Formular erneut aufgerufen werden soll. Verwenden Sie dagegen *Application.Exit()* in der Ereignisbehandlungsroutine, wird die komplette Anwendung beendet.

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung implementierten Beispiels im Ordner *\Beisp\Kap09\Beispiel9_06* auf der Begleit-CD.

9.2.4 Formulare als Klassen realisieren

Das obige Beispiel enthält den kompletten Formularcode innerhalb der *Main*-Prozedur. Falls Sie Formulare dynamisch zur Laufzeit über Programmcode generieren müssen, sollten Sie deren Implementierung über Klassen vornehmen. Mit dem Wissen aus dem obigen Beispiel und den Kenntnissen aus *Kapitel 7* über objektorientierte Programmierung verfügen Sie über das benötigte Rüstzeug, um auch diesen Schritt zu bewältigen. Die nachfolgende Codesequenz zeigt die Implementierung eines Formulars in einer separaten Klasse:

```
Public Class Form1 ' Formularklasse
    Inherits System.Windows.Forms.Form ' wir erben

    Public Eingabe As String ' für Eingabewert

    Const H1 As Integer = 24 ' Höhe Elemente
```

Listing 9.4: Implementierung des Formulars als Klasse


```

Const y1 As Integer = 30      ' erste Zeile
Const y2 As Integer = 80      ' zweite Zeile
Const x1 As Integer = 20      ' erste Spalte

' Schaltflächen so anlegen, dass Ereignisse möglich sind
Friend WithEvents b1 As New Button()
Friend WithEvents b2 As New Button()

Dim l1 As New Label()         ' Labeltext
Public t1 As New TextBox()     ' Textfeld

Sub New()                     ' New-Konstruktor zum Anlegen des Formulars
    MyBase.New()              ' Formularobjekt aus Klasse
    With Me
        .Name = "Form1"
        .Text = " Benutzername"      ' Titeltext
        .Width = 300                 ' Abmessungen
        .Height = 150
        .Left = 100                  ' Position
        .Top = 200

        ' äußere Gestaltung
        .FormBorderStyle = _
            Windows.Forms.FormBorderStyle.FixedDialog

        ' Farben
        .BackColor = Color.LightSlateGray
        .ForeColor = Color.Black

        .MaximizeBox = False          ' Schaltflächen aus
        .MinimizeBox = False
        .ControlBox = True             ' Systemmenü ein
        ' nun ein Icon einbinden
        .Icon = New Icon("../Icons/Cross.ico")
    End With

    ' Label definieren
    With l1
        .Text = "Name"
        .Name = "L1"
    End With
End Sub

```

Listing 9.4: Implementierung des Formulars als Klasse (Forts.)

```

        .Size = New Size(80, H1)      ' Größe
        .Location = New Point(x1, y1) ' Position
    End With

    ' Textfeld definieren
    With t1
        .Text = "Born"
        .Name = "T1"
        .Size = New Size(156, H1)     ' Größe
        .Location = New Point(l1.Left + l1.Width + 20, y1)
    End With

    ' Schaltfläche1 definieren
    With b1
        .Text = "OK"
        .Name = "B1"
        ' Rückgabecode
        .DialogResult = Windows.Forms.DialogResult.OK '
        .Size = New Size(80, H1)      ' Größe
        .Location = New Point(x1 + 50, y2)
    End With

    ' Schaltfläche2 definieren
    With b2
        .Text = "Schließen"
        .Name = "B2"
        .DialogResult = Windows.Forms.DialogResult.Cancel
        .Size = New Size(80, H1)      ' Größe
        .Location = New Point(b1.Left + b1.Width + 20, y2)
    End With

    With Me      ' Steuerelemente zum Formular hinzufügen
        .Controls.Add(l1)      ' Label
        .Controls.Add(t1)      ' Textfeld
        .Controls.Add(b1)      ' Schaltfläche 1
        .Controls.Add(b2)      ' Schaltfläche 2
    End With
End Sub

' Ereignisbehandlungsroutinen

```

Listing 9.4: Implementierung des Formulars als Klasse (Forts.)

```

Private Sub OK_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles b1.Click
    Eingabe = Me.t1.Text      ' Ergebnis sichern
    Me.Close()               ' Formular schließen
End Sub

Private Sub Cancel_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles b2.Click
    Eingabe = "abgebrochen"   ' Abgebrochen
    Me.Close()               ' Formular schließen
End Sub
End Class

```

Listing 9.4: Implementierung des Formulars als Klasse (Forts.)

Das durch die Klasse implementierte Formular besitzt zwei Schaltflächen, ein Labelfeld und ein Textfeld (Abbildung 9.17, rechts oben). An dieser Stelle möchte ich noch einige Bemerkungen zum Programmcode geben. Die Anweisung

```
Inherits System.Windows.Forms.Form
```

vereinbart, dass die Klasse alle Member der Klasse *Forms* erbt. Das bedeutet, ein neues Formular besitzt bereits alle Member der Basisklasse.

Die Variable *Eingabe* dient zur Aufnahme der im Textfeld hinterlegten Benutzereingabe, sobald die OK-Schaltfläche angeklickt wird. Die Variable wird als *Public* vereinbart, damit das Hauptmodul über die Objektvariable des Formulars auf den Wert zugreifen kann. Die im Kopf der Klasse vereinbarten Konstanten wie *H1*, *y1* etc. legen die Höhe der Steuerelemente sowie deren Positionen im Formular fest.

```

Const H1 As Integer = 24      ' Höhe Elemente
Const y1 As Integer = 30      ' erste Zeile
Const y2 As Integer = 80      ' zweite Zeile
Const x1 As Integer = 20      ' erste Spalte

```

Die folgende Codesequenz definiert die Objektvariable für die beiden Schaltflächen und für das Textfeld sowie das Labelfeld.

```

' Schaltflächen so anlegen, dass Ereignisse möglich sind
Friend WithEvents b1 As New Button()
Friend WithEvents b2 As New Button()

Dim l1 As New Label()         ' Labeltext
Public t1 As New TextBox()     ' Textfeld

```

Die Zeilen mit den Schlüsselwörtern *Friend WithEvents* deklarieren dabei die Objektvariablen für die beiden Schaltflächen. Mit *Friend* erreichen wir, dass die Objekte in abgeleiteten Klassen bekannt sind. Das Schlüsselwort *WithEvents* veranlasst das .NET-Laufzeitsystem, die beiden Schaltflächenobjekte für die Ereignisbearbeitung vorzusehen. Dies ist hier zulässig, da der Code zur Formulardefinition innerhalb einer Klasse steht. Wir sparen uns damit die Registrierung der Prozeduren über *AddHandler* und Funktionsdelegates (siehe Kapitel 7).

Bezeichnungsfelder (englisch Labels) werden als Objekte vom Typ *Label* vereinbart und über den *New*-Konstruktor instantiiert. Das Textfeld besitzt den Typ *TextBox*, wobei hier die Variable als *Public* vereinbar wurde, damit diese auch in den Ereignisprozeduren zugreifbar ist.

Innerhalb der Klasse wird dann der *New*-Konstruktor neu definiert. Die betreffende Prozedur enthält den gesamten Code, der zum Anlegen des Formulars benötigt wird. Als Erstes wird der *New*-Konstruktor der Basisklasse aufgerufen:

```
MyBase.New()
```

Anschließend liegt eine Objektinstanz des Formulars vor. Über das *Me*-Schlüsselwort lässt sich auf die Eigenschaften und Methoden der aktuellen Formularinstanz zugreifen. Dies wird anschließend genutzt, um den Formularnamen, dessen Titeltext, die Abmessungen des Formulars und dessen Position über die betreffenden Eigenschaften festzulegen. Dies kennen Sie bereits aus den vorherigen Beispielen.

Konfiguration des Formulars über dessen Eigenschaften

In den bisherigen Beispielen des Kapitels haben Sie gesehen, dass sich das Erscheinungsbild eines Formulars und dessen Verhalten über die Eigenschaften anpassen lässt. Diese Technik wird auch hier zum Anpassen des zur Laufzeit erzeugten Formulars genutzt. Interessant ist die Eigenschaft *FormBorderStyle*, die das Aussehen des Formularrands bestimmt. Sie können diese Eigenschaft auf die Fensterstile *Fixed3D*, *FixedDialog*, *FixedSingle*, *FixedToolWindow*, *None*, *Sizable* und *SizableToolWindow* setzen. Die Enumerationskonstante *Sizable* erlaubt, dass der Benutzer die Formulargröße zur Laufzeit anpasst, während mit *FixedDialog* eine feste Größe vorgegeben wird.

Die Farben für den Hintergrund und den Vordergrund(-text) des Formulars können über die Eigenschaften *BackColor* und *ForeColor* gesetzt werden. Weisen Sie einen Wert vom Typ *Color* aus der *Color*-Enumerationskonstante zu. Dies ließe sich beispielsweise mit der folgenden Anweisung erreichen:

```
Me.BackColor = Color.AntiqueWhite
```

Die *Color*-Struktur besitzt eine ganze Reihe vordefinierter Werte, die in der Hilfe des .NET Framework dokumentiert sind. Die Eigenschaft *Opacity* (hier nicht benutzt) ist standardmäßig auf den Wert 1.0 gesetzt, das Formular ist undurchsichtig. Tragen Sie einen Wert zwischen 0.0 und 1.0 ein, ändert dies die Transparenz, d.h., je kleiner der Wert ist, umso durchsichtiger wird das Formular. Auch dies haben Sie in den vorherigen

Beispielen bereits kennen gelernt (denken Sie lediglich daran, dass die Werte im Fenster der Entwicklungsumgebung in Prozent dargestellt werden).

Über die Eigenschaften *MaximizeBox*, *MinimizeBox* und *ControlBox* können Sie die Schaltflächen *Maximieren/Wiederherstellen*, *Minimieren* und das Systemmenü ein-/ausblenden. Um ein Element auszublenden, setzen Sie die gewünschte Eigenschaft einfach auf den Wert *False*.

Über die Eigenschaft *Icon* kann das im Systemmenü des Formulars angezeigte Symbol zur Laufzeit aus einer *.ico*-Datei geladen werden. Besitzt die .NET-Anwendung kein weiteres Fenster, wird dieses Symbol auch für die Schaltfläche der Taskleiste benutzt. Das Symbol lässt sich mit der folgenden Anweisung aus einer Datei importieren:

```
.Icon = New Icon("Icons\Cross.ico")
```

Hier wird der *New*-Konstruktor der *Icon*-Klasse benutzt, um die Symboldatei aus dem angegebenen Verzeichnis zu laden und als Ressource an die *Icon*-Eigenschaft zu übergeben.

Hinweis

Bei den Symboldateien muss es sich um gültige Ressourcen handeln. Weiterhin setzt dieser Ansatz voraus, dass die Datei im angegebenen Verzeichnis mit der .NET-Anwendung weitergegeben wird. Die obige Pfadangabe erfordert, dass die Symboldatei *Cross.ico* in einem Unterordner *Icons* vorliegt. Dieser Unterordner muss im Verzeichnis untergebracht sein, in dem die *.exe*-Datei hinterlegt wird. In der Projektdatei wurde ein modifizierter relativer Pfad benutzt, der auf einen Ordner *Icons* im Projektordner verweist. Falls Sie das Beispiel auf die lokale Festplatte kopieren und dort bearbeiten, müssen Sie ggf. den Pfad anpassen. Entwerfen Sie Formulare in Visual Studio, können Sie diese Probleme vermeiden, indem Sie die Symbole als Ressourcen im Projekt einbinden und dann zur Laufzeit die Ressource zuweisen. Klicken Sie im Projektmappen-Explorer auf die Projektdatei und fügen Sie über die Kontextmenübefehle *Hinzufügen/Neues Element* eine neue Ressource als Vorlage im Projekt ein. Öffnen Sie die Ressource per Doppelklick im Designer und wählen Sie im Menü der Schaltfläche *Ressource hinzufügen* den Befehl *Vorhandene Datei hinzufügen*. Anschließend wählen Sie die betreffende *.ico*-Datei aus. Diese wird unter ihrem Namen automatisch als Ressource in die *.exe*-Datei mit eingebunden. Zur Laufzeit können Sie dann über die Anweisung *Me.Icon = My.Resources.Resource1.Born* auf die Ressource *Born* des Elements *Resource1* (der Datei *Resource1.resx*) zugreifen und der *Icon*-Eigenschaft des Formulars zuweisen. Sie finden ein entsprechendes Beispiel im Ordner *\Beisp\Kap09\Beispiel9_07b* der Begleit-CD.

Die Objektinstanzen der Steuerelemente werden über Codesequenzen wie die folgende mit Eigenschaften belegt:

```
' Textfeld definieren
With t1
    .Text = "Born"
    .Name = "T1"
```

```
.Size = New Size(156, H1)      ' Größe
.Location = New Point(l1.Left + l1.Width + 20, y1)
End With
```

Der Code zur Definition der Eigenschaften der einzelnen Steuerelemente ist mehr oder weniger eine Wiederholung des bereits in den vorherigen Beispielen gelernten. Das Steuerelement ist über seine Eigenschaften im Formular zu positionieren und in der Größe festzulegen. Die Eigenschaft *Text* legt beim Formular den Titeltext, bei Schaltflächen und Labels die Beschriftung und beim Textfeld den angezeigten Vorgabewert fest.

Lediglich bei den Schaltflächen kommt eine erwähnenswerte Neuerung hinzu. Dort habe ich die folgende Anweisung zur Definition einer Eigenschaft eingefügt:

```
B1.DialogResult = DialogResult.OK
```

Der Eigenschaft *DialogResult* der Schaltfläche *B1* wird einfach ein Wert aus der *DialogResult*-Enumeration zugewiesen. Hier ist dies die Konstante für *OK*, bei der Schaltfläche *B2* wird der Wert für *Cancel* in der Eigenschaft hinterlegt. Wird das Formular über die *ShowDialog()*-Methode aufgerufen, wartet die Anwendung, bis der Benutzer das Formular schließt. Da die Formularinstanz aber weiterhin existiert, lässt sich der Code der vom Benutzer angeklickten Schaltfläche über die Eigenschaft *DialogResult* des Formularobjekts abfragen. Das Hauptprogramm kann also prüfen, ob der Benutzer die Schaltfläche *OK* oder *Abbrechen* angeklickt hat.

Anschließend fügt die folgende Anweisungsfolge die Steuerelemente über die *Add()*-Methode der *Controls*-Auflistung hinzu.

```
With Me      ' Steuerelemente zum Formular hinzufügen
.Controls.Add(l1)      ' Label
.Controls.Add(t1)      ' Textfeld
.Controls.Add(b1)      ' Schaltfläche 1
.Controls.Add(b2)      ' Schaltfläche 2
End With
```

Kommen wir noch kurz auf die Ereignisbehandlung nach Mausklicks auf die beiden Schaltflächen zurück. Nachdem die betreffenden Objekte über das Schlüsselwort *WithEvents* deklariert wurden, lassen sich die Ereignisbehandlungsroutinen einfach innerhalb der Klasse einbauen. Hier sehen Sie den Code für die Behandlung des *Click*-Ereignisses der *OK*-Schaltfläche. Diese besitzt die Objektvariable *B1*:

```
Private Sub OK_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles B1.Click
    Eingabe = Me.T1.Text      ' Ergebnis sichern
    Me.Close()                ' Formular schließen
End Sub
```

Den Namen der Prozedur dürfen Sie dabei frei wählen. Die Argumente in Klammern sind fest vorgegeben. Wichtig ist lediglich der Teil *Handles Objekt.Ereignis*, wobei *Objekt* für den Objektnamen des betreffenden Steuerelements und *Ereignis* für den Namen des Ereignisses stehen. Die Schaltfläche mit dem Objektnamen *B1* benötigt daher für das *Click*-Ereignis die Angabe *Handles B1.Click*. Damit kann die Laufzeitumgebung die betreffenden Ereignisse eindeutig zuordnen.

Innerhalb der Prozedur wird dann der Wert des Textfelds über die *Text*-Eigenschaft gelesen und der als *Public* vereinbarten globalen Variable *Eingabe* zugewiesen. Die Zwischenspeicherung ist hilfreich, falls das Formular über (die eigentlich nicht einzusetzende) *Application.Run()*-Methode aufgerufen wurde. Sobald der Benutzer das Formular schließt, geht bei diesem Ansatz nämlich der Inhalt des Textfelds verloren. Daher sichert die *Click*-Ereignisprozedur der *OK*-Schaltfläche den Inhalt des Textfelds. Bei einem mit *ShowDialog()* angezeigten Formular ist dieser Kunstgriff aber überflüssig! Die Ereignisbehandlung *Click* der Schaltfläche *Abbrechen* weist der globalen Variable dagegen den Text »abgebrochen« zu. Natürlich können Sie auch eine leere Zeichenkette oder Sternchen etc. zuweisen. Um das Formularfenster bei allen Aufrufvarianten beim Anklicken der Schaltflächen zu schließen, wurde in beide Ereignishandler der Befehl

```
Me.Close()
```

eingbracht. Die *Close()*-Methode bezieht sich hier auf *Me*, also auf das Formular. Vieles in der Definition des Formulars als Klasse war im Grunde nur eine Wiederholung der auf den vorhergehenden Seiten behandelten Techniken.

Startprozedur zur Formularanzeige

Jetzt gilt es, die Startprozedur zu implementieren, die eine Formularinstanz anlegt und dem Benutzer anzeigt (Abbildung 9.17, oben). Nach dem Schließen des Formulars soll die Anwendung das Ergebnis der Formulareingabe in einem einfachen Dialogfeld anzeigen (Abbildung 9.17, unten). In der Prozedur *Main* des Startobjekts wird das Formular folgendermaßen vereinbart:

```
Dim oForm As New Form1()
```

Anschließend liegt eine Objektvariable *oForm* vor, über die auf die mittels *New* konstruierte Objektinstanz zugegriffen werden kann. Soll beispielsweise der Titeltext des Formulars im Hauptprogramm gesetzt werden, lässt sich dies mit folgender Anweisung erledigen:

```
oForm.Text = " Mit ShowDialog gestartet"
```

Auf ähnliche Weise ließen sich auch andere Eigenschaften des Objekts überschreiben – die Verwendung von Klassen hat schon seine Vorteile. Nach dem Aufruf des Formulars über *oForm.ShowDialog()* wertet das Hauptprogramm den vom Benutzer eingegebenen Textwert aus. Das Hauptprogramm kann hierzu über die Objekthierarchie *oForm.t1.Text* auf den Wert des Textfelds zugreifen. *oForm* ist der Objektnamen des Formulars und *t1* der Objektnamen des Textfelds. In der Eigenschaft *Text* dieses Objekts wird der Eingabewert

verwaltet. Zudem lässt sich der Code der vom Benutzer zum Schließen des Formulars verwendeten Schaltfläche mit der Anweisung `oForm.DialogResult` ermitteln. Weitere Details zur Implementierung sind dem Quellcode der betreffenden Projektdateien zu entnehmen.

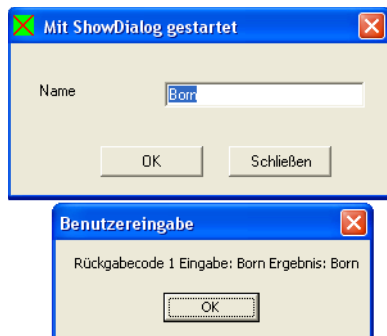


Abbildung 9.17: Dialoge des Formularbeispiels

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung realisierten Beispiels im Ordner `\Beisp\Kap09\Beispiel9_06a` auf der Begleit-CD. Das Unterverzeichnis *Icon* enthält einige Symboldateien, die sich über eine relative Pfadangabe im Systemmenü einblenden lassen.

9.2.5 ToolTips und Hilfe für Formularelemente festlegen

Viele Windows-Anwendungen bieten dem Benutzer die Möglichkeit, ToolTips zu Steuerelementen in Dialogfeldern abzurufen. Sobald der Benutzer auf ein Steuerelement zeigt, wird der Tooltipp eingeblendet. Zudem weisen manche Dialogfelder eine Schaltfläche zum Aufruf der Direkthilfe in der Titelleiste auf. Klicken Sie auf diese Schaltfläche und dann auf ein Steuerelement, wird eine kontextsensitive Hilfe für das Element eingeblendet. In .NET-Anwendungen lässt sich dies ebenfalls nutzen (Abbildung 9.18).

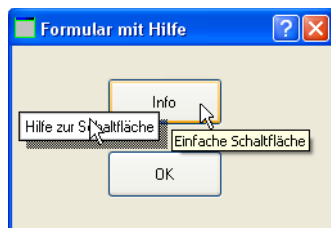


Abbildung 9.18: ToolTips und kontextabhängige Hilfe

Das .NET Framework stellt entsprechende Provider für ToolTips bereit, die sich in der Entwicklungsumgebung (Visual Studio 2005 oder Visual Basic 2005 Express Edition) oder dynamisch zur Laufzeit zum Formular hinzufügen lassen. Nachfolgend wird

beschrieben, wie Sie ein Formular in Visual Studio 2005 mit den betreffenden Optionen ausstatten.

1. Legen Sie ein Windows-Projekt mit einem Formular in der Entwicklungsumgebung an, setzen Sie die Formulareigenschaften und ergänzen Sie das Formular im Formulardesigner um die gewünschten Schaltflächen.
2. Fügen Sie als Nächstes aus der Symbolleiste *Toolbox* die beiden Elemente *HelpProvider* und *ToolTip* im Formular ein. Der Designer zeigt diese zur Laufzeit nicht im Formular sichtbaren Elemente in einem eigenen Komponentenbereich unterhalb des Formulars an (Abbildung 9.19).

Mit diesen Schritten haben Sie das Formular so vorbereitet, dass sich ToolTips und kontextsensitive Hilfe zu den einzelnen Steuerelementen hinzufügen lassen. Konkret bewirken die beiden eingefügten Steuerelemente, dass die sichtbaren Steuerelemente des Formulars über die zusätzlichen Eigenschaften *HelpProvider* und/oder *ToolTip* verfügen. Um z.B. die *ToolTip*-Eigenschaft der betreffenden Steuerelemente anzupassen, sind dann folgende Schritte erforderlich:

3. Klicken Sie im Entwurfsmodus auf das gewünschte Steuerelement im Formular und wechseln Sie zum Eigenschaftensfenster.
4. Suchen Sie im Eigenschaftensfenster die Eigenschaft *ToolTip auf ...* und tragen Sie den gewünschten Text in dieser Eigenschaft ein.

Der Name des ToolTip-Eigenschaftensfelds setzt sich dabei aus dem festen Text *ToolTip auf* und einem variablen Namen (wie *ToolTip1*) für das betreffende *ToolTip*-Steuerelement zusammen. Über diesen Namen verwaltet die Komponente die entsprechenden Tool-Tips. In Abbildung 9.19 wurde der Schaltfläche *Info* auf diese Weise der ToolTip-Text »Einfache Schaltfläche« zugewiesen.

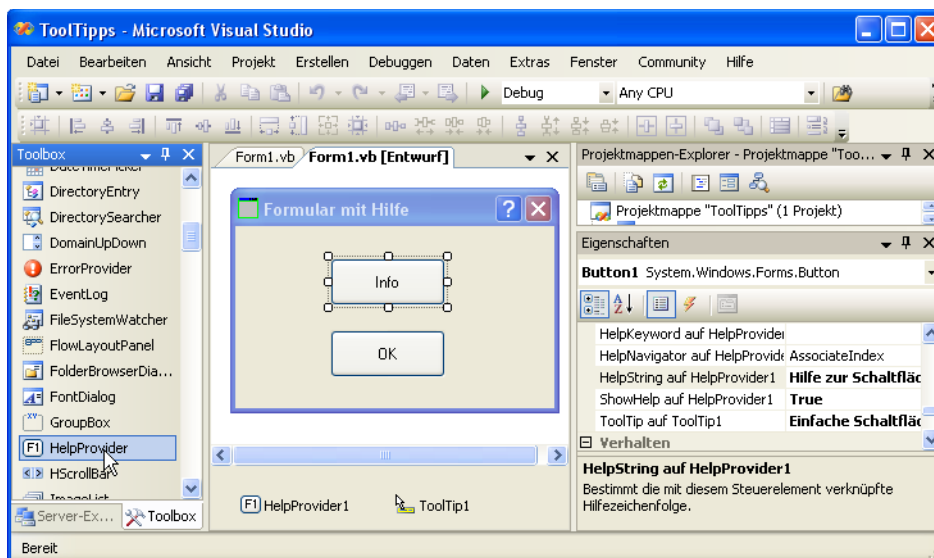


Abbildung 9.19: Entwurf des Formulars im Formulardesigner

Soll das Formular noch eine Schaltfläche *Hilfe* zum Aufrufen der Direkthilfe in der Titelleiste aufweisen? Und möchten Sie den Steuerelementen eine kontextsensitive Hilfe zuweisen, die sich nach dem Anklicken der Hilfeschaltfläche durch Anwählen des Elements abrufen lässt? Sofern Sie, wie bereits oben erläutert, das Steuerelement *HelpProvider* zum Formular hinzugefügt haben, führen Sie folgende Zusatzschritte aus:

1. Wählen Sie das Formular im Designer aus, sodass dessen Eigenschaften angezeigt werden.
2. Wechseln Sie zum Eigenschaftenfenster und setzen Sie die Werte der beiden Eigenschaften *MaximizeBox* und *MinimizeBox* auf »False«. Dies ist unbedingt erforderlich, da die Hilfeschaltfläche nur angezeigt wird, wenn die beiden Schaltflächen *Minimieren* und *Maximieren* nicht angezeigt werden.
3. Weisen Sie der Formulareigenschaft *HelpButton* den Eigenschaftswert »True« zu. Dies bewirkt, dass die Schaltfläche *Hilfe* (das ?) in der Titelleiste neben der Schaltfläche *Schließen* erscheint.
4. Klicken Sie nun im Fenster des Designers auf das Steuerelement, dem eine kontextsensitive Hilfe zuzuweisen ist.
5. Suchen Sie im Eigenschaftenfenster die Eigenschaft *HelpString auf ...* (wobei die Pünktchen hier für den Namen des Providers, z.B. *HelpProvider1*, stehen).
6. Tragen Sie im Eigenschaftsfeld den gewünschten Text ein, der in einem Pop-up-Fenster als Hilfetext erscheinen soll.

Nachdem Sie die betreffenden Schritte für jedes gewünschte Steuerelement ausgeführt haben, generiert der Formulardesigner der Entwicklungsumgebung den Code zur Anzeige der Kontexthilfe und der ToolTips.

Hinweis

Sie finden die Projektdateien dieses Windows-Beispiels im Ordner `\Beisp\Kap09\ToolTips` der Begleit-CD. Wenn Sie das Projekt übersetzen und ausführen, lassen sich sowohl Tooltips als auch die Direkthilfe für die beiden Schaltflächen des Formulars abrufen. Die Schaltfläche *Info* zeigt beim Anklicken einen Zusatzdialog an, während das Formular über die OK-Schaltfläche geschlossen wird.

ToolTips und Kontexthilfe per Code zum Formular hinzufügen

Möchten Sie ein Formular mit ToolTips oder einer Direkthilfe dynamisch per Code erzeugen? Mit dem Wissen der vorhergehenden Seiten ist es kein Problem, die betreffenden Provider zum Formularcode hinzuzufügen und die betreffenden Eigenschaften manuell zu setzen. In der Klassendefinition sind folgende Anweisungen einzufügen:

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Friend WithEvents Button1 As Button
    Friend WithEvents Button2 As Button
    ' ToolTip- und Hilfe-Provider
```

```

Friend WithEvents ToolTip1 As ToolTip
Friend WithEvents HelpProvider1 As HelpProvider
Private components As System.ComponentModel.IContainer
...
End Class

```

Die Anweisungen *ToolTip1 As ToolTip* und *HelpProvider1 As HelpProvider* vereinbaren die Variablen zur Aufnahme der entsprechenden Objektinstanzen. Da die Anweisungen mit der *WithEvent*-Anweisung versehen wurden, kann das Programm auch Ereignisse für diese Steuerelemente unterstützen.

Im *New*-Konstruktor des Formulars lassen sich dann die Eigenschaften für die ToolTips und die Kontexthilfe setzen. Die folgenden beiden Anweisungen instantiieren die betreffenden Objekte:

```

Me.ToolTip1 = New ToolTip(Me.components)
Me.HelpProvider1 = New HelpProvider()

```

Anschließend können die Instanzen über die Objektvariable *ToolTip1* und *HelpProvider1* angesprochen werden. Beide Variable sind als Eigenschaften an das Formular angebunden und lassen sich über *Me* referenzieren. Die folgende Codesequenz vereinbart dann die globalen Eigenschaften für die ToolTips wie Dauer der Anzeige, Wartezeit, bis die Anzeige erscheint etc.

```

Me.ToolTip1.AutoPopDelay = 5000 ' Dauer der Anzeige
Me.ToolTip1.InitialDelay = 1000 ' Anzeigeverzögerung
Me.ToolTip1.ReshowDelay = 500
Me.ToolTip1.ShowAlways = True

```

Um einen ToolTip zu definieren und an die Schaltfläche *Button1* anzubinden, ist folgende Anweisung erforderlich:

```

Me.ToolTip1.SetToolTip(Me.Button1, "Schließen des Dialogfelds")

```

Die Methode *SetToolTip()* erwartet im ersten Parameter das Objekt, dem der ToolTip-Text zugeordnet werden soll. Im zweiten Parameter ist der ToolTip-Text als String zu übergeben. Das Einbinden des HelpProviders ist ebenfalls mit zwei Anweisungen erledigt:

```

Me.HelpProvider1.SetHelpString(Me.Button1, "Hilfe zur Schaltfläche")
Me.HelpProvider1.SetShowHelp(Me.Button1, True)

```

Die erste Anweisung definiert den Hilfestring mittels der *SetHelpString()*-Methode, wobei als Parameter das Objekt und der Hilfestring zu übergeben sind. Anschließend ist über die *SetShowHelp()*-Methode die Anzeige der Hilfetexte freizugeben. Hierzu ist im zweiten Parameter der Wert »True« zu übergeben.

Hinweis

Weitere Details sind den entsprechenden Projektdateien zu entnehmen. Sie finden diese im Ordner \Beisp\Kap09\ToolTips1 der Begleit-CD.

9.2.6 Formulare mit dynamischen Inhalten

Als Zusammenfassung des bisher Gelernten möchte ich nun das Beispiel eines Formulars zeigen, welches dynamisch die Größe und Position ändert. Beim Aufruf enthält das Formular nur zwei Schaltflächen *Schließen* und *Erweitern >>>* (Abbildung 9.20, links).

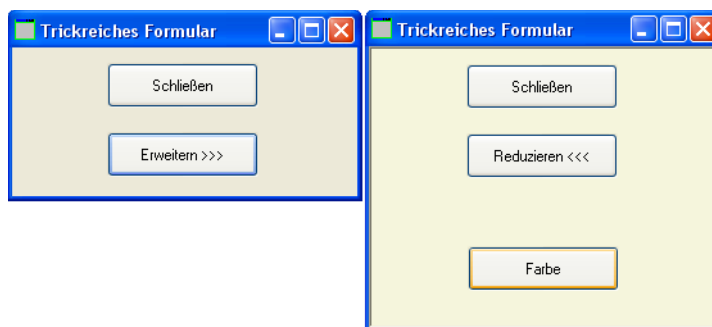


Abbildung 9.20: Veränderbares Formular

Klicken Sie auf die Schaltfläche *Erweitern >>>*, wird das Formular vergrößert und die Beschriftung der Schaltfläche auf *Reduzieren <<<* geändert (Abbildung 9.20, rechts). Im erweiterten Zustand ist eine dritte Schaltfläche *Farbe* zu sehen, die zum Umschalten der Hintergrundfarbe und des Fensterstils des Formulars dient.

Zudem ist die Reihenfolge der Schaltflächen, die beim Drücken der -Taste den Fokus erhalten, vordefiniert. Beim Aufruf des Formulars erhält die Schaltfläche *Erweitern >>>* den Fokus. Drückt der Anwender die -Taste, wechselt der Fokus zur Schaltfläche *Schließen* und eine Betätigung der -Taste setzt den Fokus auf *Farbe*.

Zur Entwurfszeit wird das Formular in der maximalen Größe und mit allen Steuerelementen im Designer der Entwicklungsumgebung angelegt. Anschließend müssen Sie die gewünschte Funktionalität in den Ereignisbehandlungsroutinen (z.B. *Load*-Ereignis des Formulars, *Click*-Ereignis der 2. Schaltfläche) unterbringen. Die Größenänderung des Formulars beim Anklicken der Schaltfläche *Erweitern >>>* lässt sich über die Formulareigenschaften *Height* und *Width* erreichen. Ähnliches gilt für die Position des Formulars (Eigenschaften *Top* und *Left*). Eine Änderung der Schaltflächenbeschriftung erfordert lediglich, dass die Eigenschaft *Text* neu definiert wird. Zum Zuweisen der Hintergrundfarbe eines Formulars wird die *BackColor*-Eigenschaft benutzt.

Das nachfolgende Listing fasst die bereits bekannten Techniken zusammen und zeigt Ausschnitte aus den Ereignisbehandlungsroutinen, um die betreffenden Formulareigenschaften dynamisch anzupassen:

```

Public Class Form1
    Const txt1 As String = "Erweitern >>>"
    Const txt2 As String = "Reduzieren <<<"
    Dim Color1 As System.Drawing.Color

    Private Sub Button2_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Button2.Click
        ' 2. Schaltfläche angeklickt - Erweitern/Reduzieren
        With Me
            If .Button2.Text = txt1 Then
                .Height = 250
                .Button2.Text = txt2
            Else
                .Height = 150
                .Button2.Text = txt1
            End If
        End With
    End Sub

    Private Sub Form1_Load(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles Me.Load
        ' Formular beim Laden anpassen
        Me.Height = 150          ' Formularhöhe
        Me.Button2.Text = txt1    ' Erweitern-Schaltfläche
        Color1 = Me.BackColor    ' merke Farbe Hintergrund
    End Sub

```

Hinweis

Sie finden die Projektdateien des Beispiels im Ordner `\Beisp\Kap09\Beispiel9_07` der Begleit-CD. Der Ordner `\Beisp\Kap09\Beispiel9_07a` enthält die Projektdateien eines abgeleiteten Beispiels, bei dem das Formular komplett über Visual-Basic-Programmcode aufgebaut wird.

9.2.7 Ein Formular mit Text- und Bildinhalten realisieren

Ein Formular mit einer OK-Schaltfläche zum Schließen reicht eigentlich, um ein benutzerspezifisches Dialogfeld zu realisieren. Der Ausgabertext kann dann mittels eines Bezeichnungsfelds (Label) im Formular angezeigt werden. Fügen Sie dann noch ein Bildelement (PictureBox) hinzu, lassen sich sogar kleine Symbole im Dialogfeld einblenden.



Abbildung 9.21: Zwei Varianten des Formulars

Wenn das betreffende Formular geschickt gestaltet wird, erhalten Sie ggf. ein universelles Anzeigedialogfeld, welches aus einer Anwendung mit wechselnden Texten und Symbolen aufgerufen werden kann (Abbildung 9.21). Die Texte sowie das anzuzeigende Bild lassen sich dabei von der rufenden Anwendung vorgeben.

Dieses einfache Formular erfordert den Einsatz von Bezeichnungsfeldern (Steuerelement *Label*) und von Bildelementen (Steuerelement *PictureBox*). Hierbei sind einige spezielle Techniken zum Einbinden der Bilder über Ressourcen oder getrennte Dateien erforderlich. Zudem soll die Anzeige des Formulars über eine benutzerdefinierte Klasse *Test* erfolgen. Um diese Vorgabe als Projekt zu realisieren, gehen Sie in folgenden Schritten vor:

1. Starten Sie die Entwicklungsumgebung und legen Sie ein neues Projekt als Windows-Anwendung an.
2. Fügen Sie über das Menü *Projekt* und den Befehl *Klasse hinzufügen* zwei neue Klassen hinzu. Benennen Sie diese als *Test.vb* und *Tools.vb* und ändern Sie den Elementnamen des Windows-Formular in *About.vb*.
3. Setzen Sie die Eigenschaften des Formulars (Größe, Titeltext) so, dass das Formular ungefähr die Größe aus Abbildung 9.22 sowie den vorgegebenen Titeltext besitzt. Die betreffenden Einstellungen lassen sich direkt im Formulardesigner (Größe) oder im Eigenschaftfenster des Formulars (Titeltext, Formularstil »FixedDialog«) vornehmen. Weisen Sie dem Systemmenü zusätzlich ein Symbol (z.B. *Born.ico*) über die betreffende Eigenschaft zu.
4. Fügen Sie eine Schaltfläche *OK* hinzu und ergänzen Sie das *Click*-Ereignis um den Befehl *Me.Close()*. Dies stellt sicher, dass das Formular beim Anklicken der Schaltfläche geschlossen wird.
5. Fügen Sie über die Toolbox ein Bildelement (*PictureBox*) im Formular ein. Die Position und Größe wählen Sie gemäß Abbildung 9.22. Weisen Sie dieser *PictureBox* dann über das Eigenschaftfenster eine Bilddatei zu. Ich habe hier das kleine Symbol *Born.ico* benutzt, welches auf der Begleit-CD im Projektordner hinterlegt ist.
6. Fügen Sie über die Toolbox ein Label-Element zum Formular hinzu. Die Position und Größe wählen Sie gemäß Abbildung 9.22.

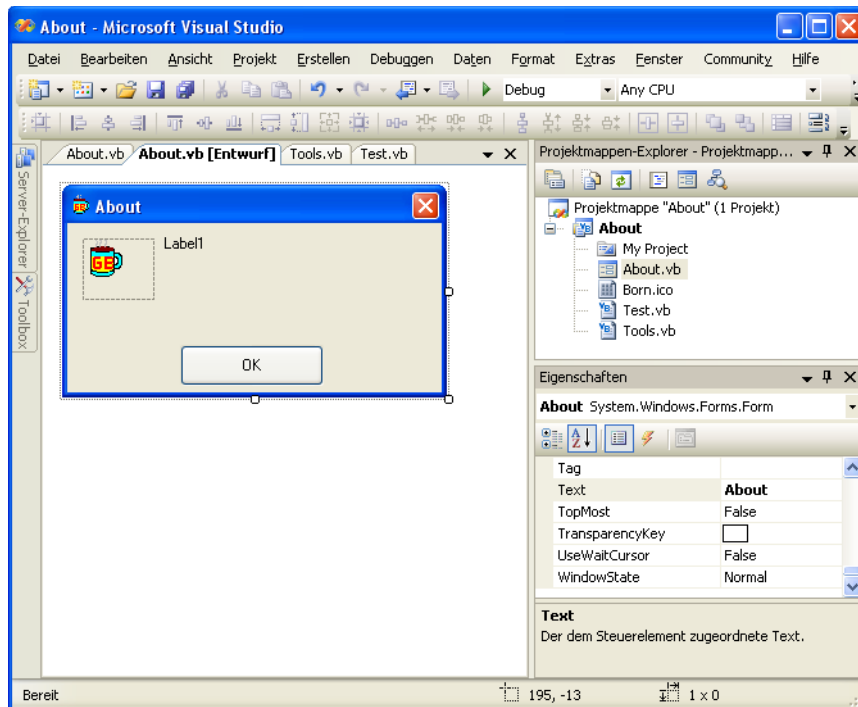


Abbildung 9.22: Entwurf des About-Formulars in der Entwicklungsumgebung

7. Wechseln Sie zum Codefenster der Klasse *Test.vb* und geben Sie den Code zum Aufruf des Formulars gemäß Abbildung 9.5 ein.
8. Klicken Sie im Projektmappen-Explorer auf das Projekt und öffnen Sie das Eigenschaftensfenster (z.B. über den Kontextmenübefehl *Eigenschaften*). Setzen Sie auf der Eigenschaftenseite *Anwendung* den Eintrag *Startobjekt* auf *Sub Main*. Dies bewirkt, dass die *Main*-Prozedur aus der Klasse *Test* beim Programmstart aktiviert wird. Weisen Sie bei Bedarf noch ein Anwendungssymbol zu, kontrollieren Sie, ob der Ausgabetyt auf *Windows-Anwendung* steht.

Schließen Sie das Eigenschaftensfenster und übersetzen Sie das Projekt. Wenn Sie anschließend das Projekt ausführen, sollte das in Abbildung 9.21 links gezeigte Formular zu sehen sein. Die Anweisungen des nachfolgend gezeigten Listings demonstrieren, wie das Formular als Objektinstanz über den *New*-Konstruktor angelegt wird. Dann lässt sich über die Objektvariable *oForm* auf die Eigenschaften des Formulars (*Text*) sowie der untergeordneten Steuerelemente (*Label1.Text*) zugreifen.

```
Shared Sub Main()
    Const Titel As String = "Borns About-Dialog V 1.0"
    Const Text As String = "Text für das Dialogfeld"
    Const Bild As String = "Tools.bmp"
```

Listing 9.5: Code zum Aufruf des About-Formulars

```

Dim oForm As About = New About() ' neues Formular

With oForm ' Greife auf Eigenschaften zu
.Text = "Borns About Dialog" ' Titeltext
.Label1.Text = "Beispiel für ein Dialogfeld," & _
    "welches als Formular realisiert wurde." & vbCrLf & _
    vbCrLf & vbCrLf & _
    "Dieser Text wurde per Programm eingetragen"
.ShowDialog() ' zeige den Dialog
.Dispose() ' Objekt verwerfen
End With
' jetzt ein zweites Dialogfeld mit geändertem Text und Bild
oForm = New About(Titel, Text, Bild)
oForm.ShowDialog()
oForm.Dispose()
Exit Sub
End Sub

```

Listing 9.5: Code zum Aufruf des About-Formulars (Forts.)

Die Techniken kennen Sie bereits aus den vorhergehenden Abschnitten. Neu ist der zweite Aufruf zur Instantiierung des *About*-Formulars, bei dem drei Argumente übergeben werden. Über diese Argumente lassen sich der Formulartitel, der Text für das Bezeichnungsfeld sowie das eingeblendete Bild übergeben. Um diese Art des Aufrufs mit mehrfach überlagerten Methoden zu unterstützen, muss die Formalklasse entsprechend erweitert werden (siehe auch *Kapitel 7*). Hierzu wird der folgende Code zur Erweiterung des *New*-Konstruktors in der Klasse des Formulars hinterlegt:

```

Public Sub New()
    MyBase.New()
    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()
End Sub

' Erweiterter New-Konstruktor
Public Sub New(ByVal Titel As String, _
    ByVal Text As String, _
    ByVal Bild As String)
    Me.New() ' Init Formular
    Dim bmp As Bitmap
    With Me
        .Text = Titel ' Formulartitel
        .Label1.Text = Text ' Label-Text
    End With
End Sub

```

Listing 9.6: Code zur Erweiterung der Formalklasse


```

    Try
        bmp = New Bitmap(Bild)
    Catch e As Exception
        MsgBox("Fehler " & Err.Number & " " & _
            Err.Description & vbCrLf & _
            "Datei " & Bild & " nicht gefunden")
        Exit Sub          ' Abbrechen
    Finally
        ' bleibt hier leer
    End Try
    .PictureBox1.Image = CType(bmp, Bitmap)
End With
End Sub

```

Listing 9.6: Code zur Erweiterung der Formalklasse (Forts.)

Der neue *New*-Konstruktor erwartet drei Argumente, von denen das erste Argument den Titeltext des Formulars, das zweite Argument den im Label auszugebenden Text und das dritte Argument den Dateinamen angibt. Im Konstruktor werden die beiden ersten Parameter den betreffenden Formulareigenschaften zugewiesen. Der dritte Parameter spezifiziert das anzuzeigende Bild, welches der *Image*-Eigenschaft des *PictureBox*-Felds zugewiesen werden soll. Allerdings benötigt diese Eigenschaft ein *Bitmap*-Objekt und keinen Dateinamen. Hierzu wird als Erstes ein *Bitmap*-Objekt erzeugt:

```

Dim bmp As Bitmap
bmp = New Bitmap(Bild)

```

Die Variable *bmp* ist entsprechend als *Bitmap* deklariert. Diese *Bitmap* lässt sich dann der *Image*-Eigenschaft der *PictureBox* zuweisen. Dabei muss die *Bitmap* aber über *CType* in den Datentyp der *Image*-Eigenschaft umgewandelt werden:

```

Me.PictureBox1.Image = CType(bmp, Bitmap)

```

Der erste Parameter in *CType* enthält das *Bitmap*-Objekt, der zweite Parameter gibt den Typ an. Allerdings gibt es noch ein Problem: Falsche Dateinamen, die als Parameter *Bild* an den Konstruktor übergeben werden, führen unweigerlich zu einem Laufzeitfehler (*Exception* bzw. *Ausnahme*, siehe Kapitel 6). Um einen auftretenden Laufzeitfehler wegen einer fehlenden Datei abzufangen, wird der Code zum Zuweisen der Bilddatei zur *PictureBox* folgendermaßen modifiziert:

```

Try          ' Hier eine fehlende Datei abfangen
    bmp = New Bitmap(Bild)
Catch e As Exception
    MsgBox ("Fehler " & Err.Number & " " & _
        Err.Description & vbCrLf & _
        "Datei " & Bild & " nicht gefunden")

```

```
Exit Sub          ' Abbrechen
End Try
.PictureBox1.Image = CType(bmp, Bitmap) ' weise zu
```

Hier kommt die Konstruktion *Try...Catch...End Try* zum Abfangen des Fehlers zum Einsatz. Der zwischen *Try* und *Catch* stehende Code wird ausgeführt. Tritt nun ein Laufzeitfehler auf, verzweigt der Programmablauf zum *Catch*-Zweig. Dort muss hinter *Catch* die abzufangende Ausnahme (Fehlerbedingung) angegeben werden. Ich habe hier die Anweisung *As Exception* benutzt, die auf jede Ausnahme (d.h. auf jeden Laufzeitfehler) reagiert. In obigem Beispiel löst die Ausnahme die Anzeige eines Meldungsfelds aus. Im Meldungsfeld wird über das *Err*-Objekt die Eigenschaft *Number* angezeigt. In *Number* steht die Fehlernummer. Alternativ ließe sich auch auf die Eigenschaft *Message* des Objekts *e* zurückgreifen, um die Fehlerursache auszugeben. Da die Fehlerursache in diesem Fall bekannt ist, wird hier zusätzlich eine Klartextmeldung mit ausgegeben. Anschließend sorgt die *Exit Sub*-Anweisung dafür, dass die Prozedur beendet wird, d.h., die hinter *End Try* folgende Zuweisung des Bildobjekts findet nicht mehr statt.

Hinweis

Die andere Möglichkeit zum Abfangen von Laufzeitfehlern bestünde in der Verwendung der *On Error Goto xx*-Anweisung, die aus früheren Visual-Basic-Versionen bekannt ist. Der Platzhalter *xx* steht für eine Marke im Programm. Diese Fehlerprüfung kann in der Hauptprozedur *Main* des Beispiels mit eingebunden werden (siehe Kapitel 6). Dadurch lassen sich Laufzeitfehler an anderer Stelle (z.B. beim Zugriff auf eine nicht eingebundene Ressource) abfangen. Dieser Ansatz wird in der Klasse *Test.vb* des Beispiels demonstriert, wobei aber die *On Error Goto ErrHandler*-Anweisung auskommentiert ist (da die Fehlerbehandlung mit *Try ... Catch ... End Try* in *About.vb* demonstriert werden soll).

Sie können die obigen Anweisungen im Codefenster eines Projekts eintragen und im Projektordner eine anzuzeigende Bilddatei (*.bmp*) hinterlegen. Wenn Sie dann das Projekt übersetzen, wird bei der Anzeige des zweiten Formulars i.d.R. trotzdem eine Fehlermeldung mit dem Hinweis erscheinen, dass die angegebene Datei nicht gefunden wurde. Der Grund liegt darin, dass die Entwicklungsumgebung die ausführbare Programmdatei *About.exe* im Ordner *bin* (bzw. in Unterordnern) hinterlegt. Standardmäßig wird die Datei *Tools.bmp* nicht mit in den Zielordner der *.exe*-Programmdatei kopiert. Sie könnten diese Datei manuell in den Ordner der *.exe*-Programmdatei kopieren, damit der Formularaufruf funktioniert. Einfacher geht es aber, wenn Sie die Grafikdatei *Tools.bmp* im Projektmappen-Explorer über den Kontextmenübefehl *Hinzufügen/Vorhandenes Element* der Projektdatei in das Projekt aufnehmen. Wenn Sie dann bei angewählter Grafikdatei im Eigenschaftenfenster den Wert der Eigenschaft *In Ausgabeverzeichnis kopieren* auf »Immer kopieren« setzen, übernimmt die Entwicklungsumgebung das Kopieren für Sie. Sie finden das Projektbeispiel im Ordner *\Beisp\Kap09\About* der Begleit-CD.

Tipp

Statt das *About*-Fenster (wie hier diskutiert) manuell als normales Windows-Formular zu entwerfen, können Sie sich das Leben aber stark erleichtern. Fügen Sie im Projekt-mappen-Explorer über die Kontextmenübefehle *Hinzufügen/Neues Element* der Projektdatei ein neues Formular im Projekt ein. Im Dialogfeld *Neues Element einfügen* wählen Sie die Vorlage *Infofeld*. Beim Schließen wird ein Element *AboutBox1.vb* im Projekt hinzugefügt. Es handelt sich um ein *About*-Dialogfeld, welches zur Laufzeit mit Informationen aus der Assembly gefüllt wird (Abbildung 9.23). Sie müssen daher ggf. über die Eigenschaften des Projekts zur Seite *Anwendung* der Projekteigenschaften gehen und über die Schaltfläche *Assemblyinformationen* den Zusatzdialog zur Eingabe der Assemblyinformationen abrufen.

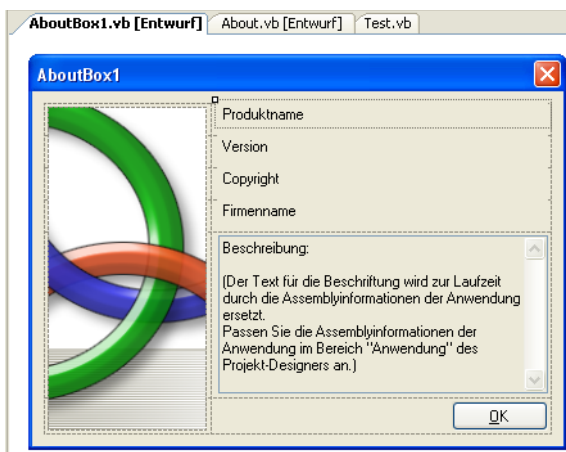


Abbildung 9.23: AboutBox-Formular der Vorlage Infofeld

Wie lässt sich der Pfad zu einer .NET-Anwendung ermitteln?

In dem oben gezeigten Programmaufruf des erweiterten *New*-Konstruktors wurde es zwar nicht genutzt, aber gelegentlich ist es hilfreich, den Pfad zur *.exe*-Datei der .NET-Anwendung zu kennen. Dann lässt sich auf Dateien oder Unterordner zurückgreifen. Die nachfolgend gezeigte Methode *GetPath()* liefert den Pfad zu dem Ordner zurück, aus dem die *.exe*-Datei mit der .NET-Anwendung gestartet wurde:

```
Imports System.Reflection
Imports System

Public Class Tools
    Shared Function GetPath() As String
        ' Extrahiere den Pfad der laufenden Anwendung
        Dim oMod As System.Reflection.Module = _
            [Assembly].GetExecutingAssembly().GetModules()(0)
```

```

Dim pfad As String
Dim len1 As Integer

len1 = oMod.Name.Length          ' Länge Dateiname
pfad = oMod.FullyQualifiedName    ' Pfad mit Dateiname
pfad = pfad.Remove(pfad.Length - len1, len1)
Return pfad
End Function
End Class

```

Im Namespace *System.Reflection* findet sich die Klasse *Module*. Die obige Anweisung ermittelt über *GetExecutingAssembly().GetModules()(0)* das erste Modul der Assembly mit dem Hauptmodul und speichert dieses Objekt in die Objektvariable *oMod*. Die Eigenschaft *FullyQualifiedName* dieses Objekts liefert den Pfad samt dem Dateinamen der .NET-Anwendung, die Eigenschaft *Name* liefert dagegen den Namen der Assembly (exe-Datei) zurück. Die Funktion subtrahiert einfach den Dateinamen *Name* von dem in der Eigenschaft *FullyQualifiedName* zurückgelieferten String ab. Das Ergebnis sollte der Pfad sein, der von der Methode an das rufende Programm zurückgegeben wird.

Hinweis

Es gibt noch die Eigenschaft *ScopeName*, die den Namen der Assembly liefert. Diese Eigenschaft lässt sich aber nicht zur Ermittlung des Pfads heranziehen. Benennen Sie die .exe-Datei um, liefert *ScopeName* weiterhin den alten Namen und die Subtraktion von *FullyQualifiedName* wird fehlerhaft.

Die andere Möglichkeit zum Ermitteln des Zugriffspfads wäre die Verwendung der Klasse *EnvironmentInfo* im Namespace *System*. Die Eigenschaft *Environment.CommandLine* der Klasse liefert den kompletten Befehl zum Aufruf der Anwendung. Sie können dann das Anführungszeichen am Pfad Anfang sowie den Programmnamen am Dateiende abschneiden und erhalten den Pfad. Sie finden ein Projektbeispiel, welches den Pfad zur ausführbaren Programmdatei sowie weitere Infos über die Anwendung liefert, im Ordner *\Beisp\Kap09\GetPath* der Begleit-CD. Die Klasse *Tools* enthält die oben gezeigte Implementierung der *GetPath()*-Funktion, während die Klasse *Tools1* die Implementierung der *GetPath()*-Funktion über die Eigenschaft *Environment.CommandLine* vornimmt. Details sind dem Quellcode der betreffenden Projektdateien zu entnehmen.

9.2.8 Willkommensdialog und Verifizieren von Formulareingaben

In einem weiteren Beispiel soll ein Formular mit einigen Textfeldern erstellt werden, dessen Eingaben beim Anklicken der *Prüfen*-Schaltfläche zu verifizieren sind (Abbildung 9.25). Die Schaltfläche *Abbrechen* erlaubt das Formular ohne Verifizierung zu schließen. Die *OK*-Schaltfläche bleibt solange gesperrt, bis die Prüfung zulässige Eingaben erkennt. Ein *LinkLabel* blendet noch einen Hyperlink zu einer Webseite im Formular ein. Das Beispiel demonstriert zudem, wie sich die Aktivierreihenfolge der Formularelemente setzen

lässt, wie eine erneute Eingabe in ein Feld direkt nach dem Verlassen dieses Feldes möglich ist bzw. wie sich Fokuswechsel per Programm verhindern lassen. Ein Textfeld dient zusätzlich zur Kennworteingabe, d.h., Eingaben werden durch * ersetzt. Zudem sollen die Eingaben in diesem Feld als Großbuchstaben an das Programm zurückgeliefert werden. Als letztes Schmäckerl soll die .NET-Anwendung zudem beim Programmstart einen Willkommen-Dialog anzeigen. Die Realisierung ist in der Entwicklungsumgebung kein größeres Problem. Gehen Sie in folgenden Schritten vor:

1. Starten Sie die Entwicklungsumgebung und legen Sie ein Projekt für eine Windows-Anwendung an. Anschließend fügen Sie ein zweites Formularelement, welches als Begrüßungsbildschirm dienen soll, zum Projekt hinzu. Sowohl Visual Basic 2005 als auch Visual Basic 2005 Express Edition verfügen über die Vorlage *Begrüßungsbildschirm*, die ein fertig gestaltetes Begrüßungsformular als Element *SplashScreen1.vb* im Projektmappen-Explorer hinterlegt.
2. Klicken Sie im Projektmappen-Explorer die Projektdatei mit der rechten Maustaste an und rufen Sie das Eigenschaftenfenster über den gleichnamigen Kontextmenübefehl auf. Anschließend wählen Sie auf der Eigenschaftenseite *Anwendung* im Listenfeld *Begrüßungsbildschirm* das betreffende Formularelement (Abbildung 9.24).

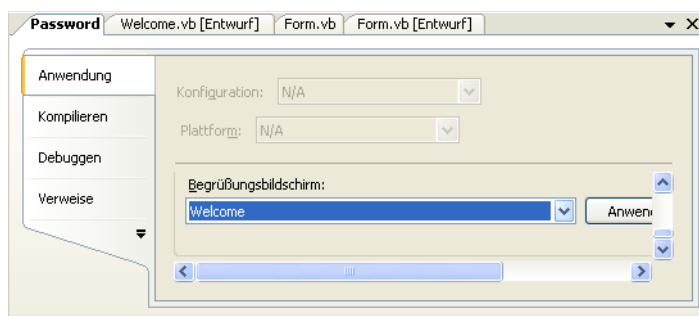


Abbildung 9.24: Festlegen des Startbildschirms

3. Haben Sie die Vorlage *Begrüßungsbildschirm* zum Projekt hinzugefügt und im Listenfeld *Begrüßungsbildschirm* eingetragen? Dieser bezieht die Informationen zum angezeigten Anwendungsnamen, zum Autor und zur Version aus den Assemblyinformationen. Klicken Sie auf der Eigenschaftenseite *Anwendung* auf die Schaltfläche *Assemblyinformationen* und passen Sie im Zusatzdialog die betreffenden Felder an. Schließen Sie den Dialog. Bei Bedarf können Sie weitere Projekteigenschaften anpassen und dann die Eigenschaftenseite schließen.
4. Öffnen Sie das Formular für den Begrüßungsbildschirm ggf. über den Projektmappen-Explorer im Formulardesigner und fügen Sie bei Bedarf zusätzliche Steuerelemente (z.B. Bezeichnungsfelder mit Versionsangaben, Bilder etc.) zum Formular hinzu. Danach schließen Sie das Formular für den Begrüßungsbildschirm im Formulardesigner.
5. Wechseln Sie im Formulardesigner zum Startformular und fügen Sie die Steuerelemente für die Textfelder, Schaltflächen, für die Label und das LinkLabel gemäß den Vorgaben in Abbildung 9.25 im Formular ein.

6. Klicken Sie auf das Formular und setzen Sie im Eigenschaftfenster den Formulartitel. Setzen Sie die ggf. Eigenschaften für *MinimizeBox* und *MaximizeBox* auf den Wert *False*.
7. Markieren Sie das dritte Textfeld für die Kennworteingabe und tragen Sie im Eigenschaftfenster in der Eigenschaft *PasswordChar* das Zeichen * ein (jede Benutzereingabe im Feld wird dann als Sternchen angezeigt). Setzen Sie die Eigenschaft *CharacterCasing* auf *Upper* (wandelt die Benutzereingabe automatisch in Großbuchstaben um).
8. Markieren Sie die linke Schaltfläche und setzen Sie deren Eigenschaft *Text* auf *OK*. Die Eigenschaft *Enabled* wird auf *False* gesetzt, um das Steuerelement zu sperren.
9. Weisen Sie der Eigenschaft *Text* der restlichen Steuerelemente die in Abbildung 9.25 gezeigten Texte zu. Das *LinkLabel* erhält in der Eigenschaft *Text* die URL einer Internetseite zugewiesen.
10. Weisen Sie der Eigenschaft *TabIndex* der Steuerelemente die Werte für die gewünschte Aktivierreihenfolge zu. Das Feld für den Namen erhält den Wert 0, das Feld für das Alter den Wert 1 und so weiter. Steuerelemente, die nicht per -Taste anwählbar sein sollen, erhalten in der Eigenschaft *TabStop* den Wert *False*.

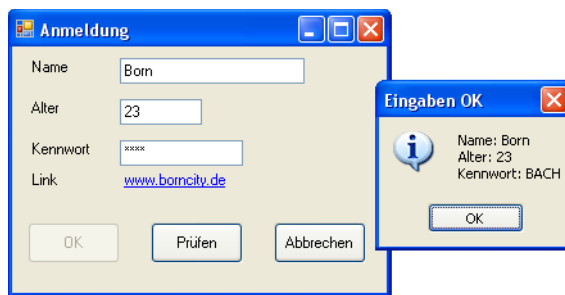


Abbildung 9.25: Formular mit Kennworteingabe und Ergebnisanzeige

Nach diesen Vorbereitungen sind die Ereignisbehandlungsprozeduren für die Anwendung zu definieren. Zur Speicherung der Werte soll die Klasse einige Variablen enthalten, die im Codefenster des Formulars eingegeben werden.

```
Dim dirty As Boolean
Dim uname As String
Dim passw As String
Dim alter As Integer
```

Das Flag *dirty* steuert, ob das Formular über die *OK*-Schaltfläche geschlossen werden kann (nur wenn der Wert auf *True* gesetzt ist). Die drei anderen Variablen nehmen die Eingabewerte des Benutzers auf.

Wenn Sie im Formulardesigner auf das Formular doppelklicken, öffnet sich das Codefenster und die Ereignisbehandlungsprozedur *Load_Form1* ist bereits vorbereitet. Diese Prozedur wird beim Laden des Formulars aufgerufen. Hier ist folgender Code einzutragen:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    dirty = True
    Me.TextBox1.Text = ""
    Me.TextBox2.Text = ""
    Me.TextBox3.Text = ""
    Me.Button1.Enabled = False ' Sperre OK
End Sub
```

Als Erstes wird das Flag *dirty* auf *True* gesetzt. Dies erlaubt in den Ereignisbehandlungsroutinen das Beenden des Formulars zu sperren. Die folgenden Anweisungen löschen die Textfelder des Formulars und die letzte Zuweisung setzt die *Enabled*-Eigenschaft der OK-Schaltfläche auf *False*, um die Schaltfläche zu sperren.

Die Ereignisbehandlungsroutine für das *Click*-Ereignis der Schaltfläche *Schließen* erhält nur eine Anweisung der Art *Me.Close()*. Doppelklicken Sie im Formulardesigner auf die OK-Schaltfläche, wird das Codefenster mit der zugehörigen Ereignisbehandlungsroutine geöffnet. Tragen Sie folgenden Code ein:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    ' OK-Schaltfläche nur wirksam wenn Prüfung ok
    If Not dirty Then
        MessageBox.Show("Eingaben OK" & vbCrLf & _
            "Name: " & uname & vbCrLf & _
            "Alter: " & alter.ToString & vbCrLf & _
            "Kennwort: " & passw & vbCrLf)
        Me.Close()
    Else
        MsgBox("Falsche oder fehlende Eingaben")
    End If
End Sub
```

Die Prozedur prüft das *dirty*-Flag. Ist dieses *False*, liegen gültige Eingaben vor. Dann gibt die Prozedur die aktuellen Werte der Variablen *uname*, *alter* und *passw* über ein Meldungsfeld aus und schließt das Formular über *Me.Close()*.

Die Ereignisbehandlung für das *Click*-Ereignis der *Prüfen*-Schaltfläche enthält etwas mehr Code. In dieser Routine werden die Eingaben auf Plausibilität geprüft, ggf. in Integerwerte gewandelt und in die lokalen Variablen *uname*, *alter* und *passw* gespeichert. Der Zugriff auf ein Textfeld ist über dessen Objektname möglich. Die folgende Anweisung überträgt den Inhalt des ersten Textfelds in die Variable *uname*:

```
uname = .TextBox1.Text
```

Die Ereignisbehandlungsprozedur prüft als Erstes, ob Eingaben in allen drei Feldern vorliegen und speichert den Namen sowie das Kennwort ab. Fehlen Eingaben, wird dies dem Benutzer durch ein Meldungsfeld signalisiert. Liegen in allen Feldern Eingaben vor, wird die Altersangabe verifiziert. Eine ungültige Zahl oder Altersangabe unter 1 Jahr und über 99 Jahre werden abgewiesen. Beachten Sie, dass die Textfelder (TextBox) nur Zeichenketten als Inhalt aufweisen. Die Werte müssen daher über Funktionen wie *CInt* in einen Zahlenwert konvertiert werden. Diese Techniken kennen Sie bereits aus dem vorhergehenden Kapitel. Nachfolgend ist der Quellcode für die komplette Ereignisbehandlungsroutine zu sehen:

```
Private Sub Button2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button2.Click
    ' Prüfen der Eingaben
    With Me
        If .TextBox1.Text <> "" Then ' Name eingegeben
            uname = .TextBox1.Text
        Else
            MsgBox("Name fehlt")
            Exit Sub
        End If
        If .TextBox3.Text <> "" Then ' Kennwort eingeben
            passw = .TextBox3.Text
        Else
            MsgBox("Kennwort fehlt")
            Exit Sub
        End If
        If .TextBox2.Text <> "" Then
            If IsNumeric(.TextBox2.Text) Then
                alter = CInt(.TextBox2.Text)
                If (alter < 1) Or (alter > 99) Then
                    MsgBox("Alter stimmt nicht")
                    dirty = True
                    Exit Sub
                Else
                    dirty = False
                    Me.Button1.Enabled = True ' Freigeben
                End If
            Else
                MsgBox("Alter ist keine Zahl")
                dirty = True
                Exit Sub
            End If
        Else
    
```



```

    MsgBox("Alter fehlt")
    dirty = True
    Exit Sub
End If
End With
MsgBox("Prüfung erfolgt")
End Sub

```

Als Besonderheit soll die Eingabe im Textfeld für den Namen (*Textbox1*) beim Verlassen des Felds, also wenn der Benutzer auf ein anderes Steuerelement klickt oder die -Taste drückt, verifiziert werden. Eine Eingabe eines oder mehrerer Leerzeichen ist dabei abzuweisen, d.h., ein Dialogfeld weist den Benutzer auf die Fehleingabe hin und der Fokus bleibt auf dem betreffenden Feld. Dies lässt sich mit einer Ereignisbehandlungsroutine realisieren, die auf das *Leave*-Ereignis des Steuerelements reagiert. Die Ereignisbehandlungsroutine enthält folgenden Code:

```

Private Sub Textbox_Exit(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles TextBox1.Leave
    ' Wird beim Verlassen des ersten Textfelds aufgerufen
    ' Verweigere Fokuswechsel, wenn nur Leerzeichen eingegeben wurden
    If Me.TextBox1.Text.Length > 0 Then ' nur bei Eingabe
        If Me.TextBox1.Text.Trim(" ") = "" Then
            Me.TextBox1.Focus()
            MsgBox("Sorry, falscher Name '" & _
                Me.TextBox1.Text & "'")
        End If
    End If
End Sub

```

Die *If*-Bedingung prüft zuerst, ob im Textfeld eine Eingabe erfolgte. Liegt keine Eingabe vor, kann der Benutzer das Feld verlassen (andernfalls würde selbst das Anklicken der *Schließen*-Schaltfläche ein *Leave*-Ereignis auslösen und ein Beenden des Formulars wäre nur nach einer Eingabe im ersten Feld möglich!). In der nächsten *If*-Bedingung wird geprüft, ob nur Leerzeichen eingegeben wurden. Hierbei habe ich auf einen Trick zurückgegriffen. Die *Trim()*-Methode kann ein vorgegebenes Zeichen am Anfang oder Ende einer Zeichenkette entfernen. Wurden nur Leerzeichen eingetippt, ist das Ergebnis eine leere Zeichenkette. Die Anweisung

```
Me.TextBox1.Focus()
```

ruft die *Focus()*-Methode auf, die den Fokus auf das zugehörige Objekt setzt. Dies sorgt dafür, dass ein Fokuswechsel unterbunden wird.

Tipp

An dieser Stelle noch ein kleiner Hinweis zur Eingabe von Ereignisbehandlungsprozeduren. Geben Sie im Codefenster einfach nur die Anweisung *Private Sub name ()* ein (*name* ist dabei beliebig) und drücken Sie die **[↵]**-Taste. Dann wird ein einfacher Prozedurrumpf erzeugt. Geben Sie am Ende der Prozedurdefinition das Schlüsselwort *Handles* ein, drücken die **[Leertaste]** und geben den Namen des Steuerelements gefolgt von einem Punkt ein. Drücken Sie die Tastenkombination **[Strg] + [Leertaste]**, erscheint ein Fenster zur Auswahl des Ereignisses (Abbildung 9.26). Abschließend können Sie noch die in den Klammern stehenden Argumente der Ereignisbehandlungsroutine aus einer anderen bereits vorhandenen Ereignisbehandlungsprozedur übertragen. Komplette Prozedurrahmen für Ereignisbehandlungsroutinen lassen sich alternativ mit der am Kapitelanfang im Abschnitt »Ereignisprozeduren in Formularen hinterlegen« Methode im Codefenster einfügen.

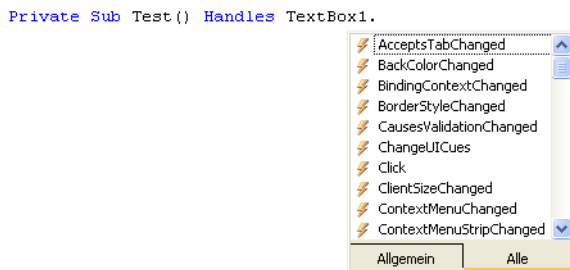


Abbildung 9.26: Abrufen des Ereignisnamens

Mit dem in diesem Beispiel gezeigten Ansatz können Sie Formulare jederzeit mit einer Prüffunktion ausstatten. Die *Enabled*-Eigenschaft erlaubt Ihnen ein Steuerelement im Formular zu aktivieren oder zu deaktivieren. Durch Zugriff auf die *Text*-Eigenschaft lässt sich die Beschriftung der meisten Schaltflächen steuern. Über weitere Eigenschaften können Sie den meisten Steuerelementen ein bestimmtes optisches Aussehen zuweisen oder deren Funktionalität anpassen. Klicken Sie auf die betreffende Eigenschaft im Eigenschaftenfenster, um Näheres über die Eigenschaften der Steuerelemente zu erfahren.

Auf das Schließen des Formulars reagieren

Auf den vorhergehenden Seiten wurde der Code der Ereignisbehandlungsroutinen vorgestellt und es wurde gezeigt, wie sich die Prüfung der Eingaben vornehmen lässt. Bei unrichtigen Eingaben soll das Schließen des Formulars über die *OK*-Schaltfläche verhindert werden. Allerdings kann der Benutzer die Funktionstaste **[F4]** drücken oder die *Schließen*-Schaltfläche des Formulars anklicken. Um eine Prüfung beim Schließen eines Formulars durchzuführen, ist es am effizientesten, den Code in der *FormClosing*-Ereignisbehandlungsroutine des Formulars einzubauen.

```

Private Sub Form_FormClosing(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.FormClosingEventArgs) _
    Handles Me.FormClosing
    ' wird beim Schließen des Formulars aufgerufen
    ' hier kann ein Schließen ggf. abgefangen werden
    If Not dirty Then
        MessageBox.Show("Name: " & uname & vbCrLf & _
            "Alter: " & alter.ToString & vbCrLf & _
            "Kennwort: " & passw, "Eingaben OK", _
            MessageBoxButtons.OK, MessageBoxIcon.Information)
        e.Cancel = False    ' Schließen erlauben
    Else
        If MessageBox.Show("Falsche oder fehlende Eingaben", _
            "Wirklich beenden?", MessageBoxButtons.YesNo, _
            MessageBoxIcon.Question) = _
            Windows.Forms.DialogResult.Yes Then
            e.Cancel = False    ' Schließen erlauben
        Else
            e.Cancel = True    ' Schließen verhindern
        End If
    End If
End Sub

```

Listing 9.7: Reagieren auf das Schließen eines Formulars

Der Ereignisbehandlungsroutine wird der Parameter *e* vom Typ *FormClosingEventArgs* übergeben. Möchten Sie das Schließen des Formulars verhindern, setzen Sie einfach die *Cancel*-Eigenschaft des *FormClosingEventArgs*-Objekts auf den Wert *True*. Nur wenn Sie die Eigenschaft auf den Wert *False* setzen, wird das Formular (z.B. als Folge des Aufrufs der *Close()*-Methode) auch geschlossen.

Ausführen eines Links

Falls Sie ein *LinkLabel*-Steuerelement im Formular eingefügt haben, lässt sich die URL des Hyperlinks in der *Text*-Eigenschaft hinterlegen. Damit sich aber beim Anklicken des Hyperlinks etwas tut, müssen Sie eine entsprechende Ereignisbehandlungsroutine implementieren.

```

Private Sub LinkLabel1_LinkClicked(ByVal sender As System.Object, _
    ByVal e As System.Windows.Forms.LinkLabelLinkClickedEventArgs) _
    Handles LinkLabel1.LinkClicked
    ' Link-Label angeklickt, Hyperlink aktivieren
    System.Diagnostics.Process.Start(LinkLabel1.Text)
End Sub

```

Listing 9.8: Handler zum Aufruf eines angeklickten Hyperlinks

In der aktuellen Implementierung wird in der *LinkClicked*-Ereignisbehandlungsroutine die *Start*-Methode mit der anzuzeigenden URL aufgerufen. Bei einer bestehenden Internetverbindung erscheint die betreffende Verweiseite im Fenster des Internet Explorer.

Hinweis

Sie finden die kompletten Projektdateien des Beispiels auf der Begleit-CD im Ordner *\Beisp\Kap09\Password*. Sobald Sie die Anwendung übersetzen, lässt sich das zugehörige .NET-Programm ausführen. Beim Start wird der Begrüßungsbildschirm für einige Sekunden ein- und dann automatisch ausgeblendet, ohne dass eine Zeile Code geschrieben wurde. Anschließend sehen Sie das Eingabeformular. Solange nicht alle Eingabefelder mit Werten belegt sind, liefert die Schaltfläche *Prüfen* einen Fehlerdialog. Erst wenn alle Eingaben getätigt wurden, gibt das Programm die OK-Schaltfläche frei. Über diese Schaltfläche rufen Sie ein Dialogfeld zur Anzeige der Eingabewerte auf. Weitere Details sind den Projektdateien zu entnehmen.

Tipp

Für Anmeldedialoge können Sie auf fertige Vorlagen zurückgreifen. Fügen Sie im Projektmappen-Explorer ein neues Element über die Vorlage *Anmeldeformular* im Projekt ein. Das Formular besitzt bereits zwei Textfelder für Benutzernamen und Kennwort sowie zwei Schaltflächen (Abbildung 9.27). Sie können die Benutzereingaben in der *Click*-Ereignisbehandlungsroutine des Formulars auswerten. Das Steuerelement für den Benutzernamen ist mit *UsernameTextBox* und das Steuerelement für das Kennwort mit *PasswordTextBox* benannt. Die *Text*-Eigenschaft der beiden Steuerelemente liefert die Benutzereingaben. Sie finden das Beispiel, welches die Benutzereingaben beim Anklicken der OK-Schaltfläche einblendet, auf der Begleit-CD im Ordner *\Beisp\Kap09\Anmeldung*.

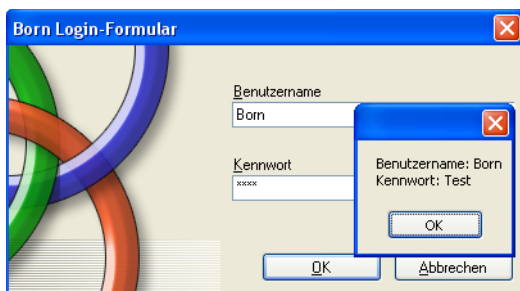


Abbildung 9.27: Formular mit Anmeldedialog

9.2.9 MaskedTextBox-Elemente

Eine Neuerung in .NET Framework 2.0 ist das *MaskedTextBox*-Element, welches maskeerte Eingaben in Textfeldern erlaubt (Abbildung 9.28).

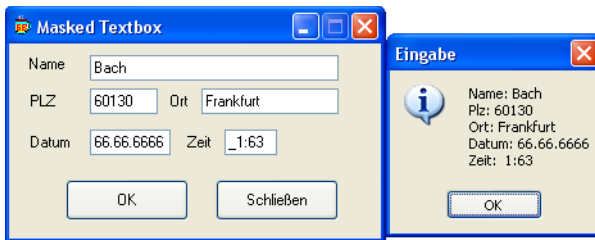


Abbildung 9.28: Formular mit MaskedTextBox-Elementen

Fügen Sie ein entsprechendes Steuerelement im Formular ein, lässt sich anschließend die Schaltfläche der Steuerelementeigenschaft *Mask* wählen. Im Zusatzdialog *Eingabeformat* können Sie Formatschablonen (Datum kurz oder lang, Zeit, Postleitzahl) für die Eingabewerte des Felds abrufen oder benutzerdefiniert eintragen. Die Formatschablonen bewirken zur Laufzeit, dass nur gültige Zeichen im Feld eingegeben werden können. Setzen Sie die Eigenschaft *BeepOnError*, lösen fehlerhafte Eingaben einen Warnton aus. Die Schablone prüft aber nur, ob korrekte Zeichen oder Ziffern im Feld eingetippt wurden. Die Validierung einer Eingabe muss durch Zuweisen eines Datentyps zur *ValidatingType*-Eigenschaft der *MaskedTextBox* und Auswerten des *TypeValidationCompleted*-Ereignisses erfolgen.

Hinweis

Sie finden das komplette Beispiel auf der Begleit-CD im Ordner `\Beisp\Kap09\MaskedTextBox`. Über die OK-Schaltfläche können Sie die Benutzereingaben in einem Dialogfeld anzeigen. Fehlerhafte Eingabe von Datum und Zeit führen zu einem Fehlerdialog. Weitere Details sind den Projektdateien zu entnehmen.

Mit den bisherigen Informationen sind Sie bereits in der Lage, einfache Formulare mit Schaltflächen und Textfeldern zu erstellen. Im nächsten Kapitel zeige ich, wie Sie weitere Steuerelemente in Formulare einbinden.

10

Weitere Steuerelemente in Formularen

Nach dem Einstieg in die Entwicklung von Formularen möchte ich in diesem Kapitel auf spezielle Fragen zum Einfügen weiterer Steuerelemente in Formularen eingehen. Sie erfahren, wie Sie zum Beispiel Optionsfelder, Kontrollkästchen, Kombinationsfelder, Bildanzeigen, Kalender oder Struktur- und Listenansichten verwenden können.

10.1 Formular zur Optionsauswahl

Kontrollkästchen und Optionsfelder erlauben dem Benutzer die Auswahl bestimmter Optionen innerhalb eines Formulars. Zudem lassen sich Schaltflächen mit Symbolen versehen bzw. in der Form wahlweise eingedrückt darstellen.

In Abbildung 10.1 ist das im Rahmen der folgenden Abschnitte entwickelte Beispielformular zu sehen. Es enthält eine Gruppe von Kontrollkästchen, die bestimmte Optionen setzen oder löschen. Wird das Kontrollkästchen *Button -> Flat* markiert, ist die mittlere Schaltfläche flach darzustellen.



Abbildung 10.1: Formular zur Optionsauswahl

Ohne Markierung des Kontrollkästchens erscheint die Schaltfläche wie die beiden benachbarten Schaltflächen. Zusätzlich wird die mittlere Schaltfläche mit einem Symbol statt eines Textes versehen. Über die drei Optionsfelder *Bild 1*, *Bild 2* und *Bild 3* lässt sich das angezeigte Bild der Schaltfläche ändern. Klickt der Benutzer auf die OK-Schaltfläche, erscheint das in Abbildung 10.1 (rechts) gezeigte Dialogfeld mit den Stati der einzelnen Optionsfelder und Kontrollkästchen. In den Bezeichnern der einzelnen Optionen sind zudem die Buchstaben der Tastaturnavigation unterstrichen (z.B. lässt sich die Option *Drucken* über die Tastenkombination **[Alt]**+**[R]** setzen oder löschen).

10.1.1 Entwurf des Formulars im Designer

Die betreffenden Steuerelemente (RadioButton, CheckBox und Button) lassen sich in der Entwicklungsumgebung sehr einfach in das Formular einfügen. Gehen wir den Entwurf des Formulars schrittweise durch:

1. Legen Sie in der Entwicklungsumgebung ein neues Projekt als Windows-Anwendung an.
2. Wechseln Sie zum Formulardesigner und setzen Sie die Eigenschaften des Formulars (z.B. Titeltext, *Minimieren*- und *Maximieren*-Schaltfläche ausgeblendet) gemäß Abbildung 10.1.

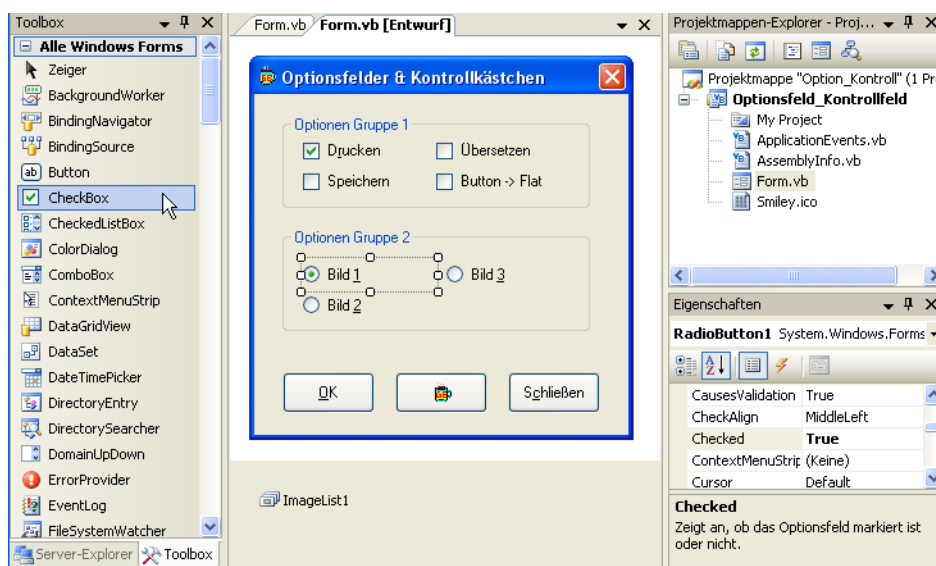


Abbildung 10.2: Formulardesign mit ImageList1-Steuerelement

3. Fügen Sie die drei in Abbildung 10.2 gezeigten Schaltflächen ein. Der linken Schaltfläche weisen Sie in der Eigenschaft *Text* die Bezeichnung »&OK« zu, die gleiche Eigenschaft setzen Sie für die rechte Schaltfläche auf »S&chließen«. Das &-Zeichen vor den einzelnen Buchstaben aktiviert diesen Buchstaben für die Tastaturnavigation. Diese Technik können Sie auch für die restlichen Steuerelemente verwenden.
4. Wählen Sie in der Toolbox das *GroupBox*-Steuerelement und fügen Sie zwei Gruppen im Formular ein. Setzen Sie die Eigenschaft *Text* für die obere Gruppe auf *Optionen Gruppe 1* und für die untere Gruppe auf *Optionen Gruppe 2*. Das Steuerelement wird durch einen Rahmen mit dem angegebenen Gruppentext angezeigt und wirkt als eine Art Container. Das *GroupBox*-Steuerelement gruppiert die darin enthaltene Steuerelemente. Bei Optionsfeldern bewirkt diese Gruppierung, dass immer nur ein Optionsfeld markiert wird (weitere getrennte Optionsfelder müssten in einer zweiten Gruppe im Formular hinterlegt werden).

5. Fügen Sie anschließend über die Toolbox vier Kontrollkästchen (Steuerelement *CheckBox*) in die obere Gruppe ein und benennen Sie diese über die Eigenschaft *Text* gemäß Abbildung 10.2. Soll ein Kontrollkästchen bereits beim Aufruf des Formulars markiert werden, setzen Sie die Eigenschaft *Checked* im Eigenschaftfenster des Steuerelements auf *True*.
6. Wählen Sie im nächsten Schritt das Steuerelement *RadioButton* in der Toolbox und fügen Sie drei Optionsfelder in der unteren Gruppe ein. Benennen Sie diese ebenfalls über die jeweils zugehörige Eigenschaft *Text* gemäß Abbildung 10.2. Zudem sollte eines der Optionsfelder die Eigenschaft *Checked=True* aufweisen, während die beiden anderen Eigenschaftsfelder den Wert *Checked=False* besitzen.

Mit diesen Vorbereitungen haben Sie bereits die Grundzüge des Formulars geschaffen (die Zuweisung eines Symbols zur mittleren Schaltfläche wird gleich behandelt). Bei Bedarf sollten Sie noch die *TabIndex*-Eigenschaft der anwählbaren Steuerelemente so justieren, dass diese in der gewünschten Reihenfolge angesprungen werden.

10.1.2 Die Ereignisprozeduren für OK und Schließen realisieren

Doppelklicken Sie im Fenster des Formulardesigners auf eine Schaltfläche, gelangen Sie direkt in der Codeansicht in die *Click*-Ereignisbehandlungsroutine des jeweiligen Steuerelements. Für die Schaltfläche *Schließen* fügen Sie lediglich eine Anweisung zum Beenden des Formulars ein.

```
Private Sub Button3_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button3.Click
    ' Schließen-Schaltfläche beendet Anwendung
    Me.Close()
End Sub
```

Die OK-Schaltfläche soll das in Abbildung 10.1 rechts gezeigte Dialogfeld mit den Status der Optionsfelder anzeigen. Sie müssen daher in der *Click*-Ereignisprozedur dieser Schaltfläche die betreffenden Anweisungen unterbringen:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    ' OK-Schaltfläche - zeige die Optionen an
    MessageBox.Show( _
        CheckBox1.Text.Replace("&c", "") & ": " & _
        CheckBox1.Checked & vbCrLf & _
        CheckBox2.Text.Replace("&c", "") & ": " & _
        CheckBox2.Checked & vbCrLf & _
        CheckBox3.Text.Replace("&c", "") & ": " & _
        CheckBox3.Checked & vbCrLf & _
        CheckBox4.Text.Replace("&c", "") & ": " & _
        CheckBox4.Checked & vbCrLf & vbCrLf & _
```



```

RadioButton1.Text.Replace("&c, """) & ": " & _
RadioButton1.Checked & vbCrLf & _
RadioButton2.Text.Replace("&c, """) & ": " & _
RadioButton2.Checked & vbCrLf & _
RadioButton3.Text.Replace("&c, """) & ": " & _
RadioButton3.Checked, _
"Einstellungen", MessageBoxButtons.OK, _
    MessageBoxIcon.Information)
End Sub

```

Ich habe in obigem Listing `MessageBox.Show()` zur Anzeige der Stati benutzt. Der Parameter `Text` wird dabei mit den auszugebenden Werten gefüllt. Um die im Designer zum Formular hinzugefügten Texte der Optionen anzuzeigen, greife ich einfach auf die `Text`-Eigenschaft des betreffenden Steuerelements zu. Mit `Me.CheckBox1.Text` erhalten Sie den Text des ersten Kontrollkästchens. Allerdings gibt es noch ein Problem. Dieser Text enthält ggf. das `&`-Zeichen für die Tastaturnavigation. Durch Anwendung der `Replace`-Methode, die das Zeichen `&` durch eine leere Zeichenkette ersetzt, lässt sich dieses Steuerzeichen aber leicht herausfiltern. Der Status eines Kontrollkästchens oder eines Optionsfelds wird einfach über die `Checked`-Eigenschaft abgefragt. Der Wert dieser Eigenschaft ist entweder `False` oder `True`, je nach Markierung.

10.1.3 Absenken der mittleren Schaltfläche steuern

Beim Markieren des Kontrollkästchens `Button -> Flat` soll die mittlere Schaltfläche flach angezeigt werden. Wird die Markierung des Kontrollkästchens gelöscht, erhält die Schaltfläche wieder die erhöhte Position zurück. Dies lässt sich sehr einfach in der Ereignisprozedur des `CheckedChanged`-Ereignisses des betreffenden Steuerelements realisieren. Diese Prozedur wird bei jeder Werteänderung aufgerufen:

```

Private Sub CheckBox4_CheckedChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles CheckBox4.CheckedChanged
    If Me.CheckBox4.Checked Then
        Me.Button2.FlatStyle = FlatStyle.Flat
    Else
        Me.Button2.FlatStyle = FlatStyle.Standard
    End If
End Sub

```

Je nach Zustand des Kontrollkästchens wird die Eigenschaft `FlatStyle` der Schaltfläche `Button2` auf `Flat` oder `Standard` gesetzt.

10.1.4 Die mittlere Schaltfläche mit einem Symbol versehen

Kommen wir jetzt zur Gestaltung der mittleren Schaltfläche. Diese soll mit einem Symbol versehen werden. Um lediglich ein Symbol zuzuweisen, sind folgende Schritte auszuführen:

1. Markieren Sie die Schaltfläche im Designer und wechseln Sie zum Eigenschaftensfenster dieses Steuerelements.
2. Klicken Sie auf die Eigenschaft *Image* (Abbildung 10.3) und öffnen Sie über die Schaltfläche dieser Eigenschaft das Dialogfeld *Ressource auswählen*.
3. Importieren Sie ggf. im Dialogfeld *Ressource auswählen* die gewünschte Bitmap-Datei als lokale Ressource oder in eine Projekt-Ressourcen-Datei.
4. Anschließend wählen Sie eine Symboldatei aus und schließen das Dialogfeld über die OK-Schaltfläche.

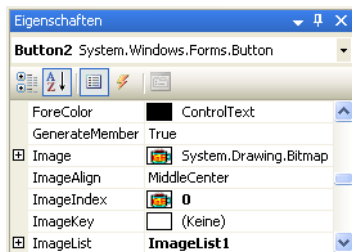


Abbildung 10.3: Zuweisen eines Symbols zu einer Schaltfläche

Tipp

Möchten Sie später das Symbol wieder entfernen, klicken Sie mit der rechten Maustaste auf das Eigenschaftensfeld *Image* und wählen im Kontextmenü den Befehl *Zurücksetzen*.

Mit diesem Schritten bindet der Designer die betreffende Datei als Ressource ein und weist diese der Schaltfläche zu. Zur Laufzeit wird die Ressource als Bitmap-Bild in der Schaltfläche angezeigt. Bei Bedarf können Sie noch die Abmessungen des Bildes über das Eigenschaftensfenster einstellen. Die Eigenschaft *Text* belassen wir hier leer. Möchten Sie Text und Symbol verwenden, tragen Sie einen Text im gleichnamigen Eigenschaftensfeld ein. Dann verwenden Sie die Eigenschaften *ImageAlign* und *TextAlign*, um Text und Bild entsprechend zu positionieren.

Im aktuellen Beispiel soll das Formular aber mit wechselnden Symbolen (abhängig vom markierten Optionsfeld) ausgestattet werden. Dies erfordert die Verwendung eines *ImageList*-Steuerelements. Ein solches Steuerelement fungiert als Container und kann verschiedene Bilddateien aufnehmen. Zudem lässt sich die *ImageList* der Schaltfläche zuordnen. Daher ist im ersten Schritt das Steuerelement *ImageList* im Formular hinzuzufügen:

1. Fügen Sie ein *ImageList*-Steuerelement aus der Toolbox in das Formular ein (ein Doppelklick darauf genügt). Das Steuerelement wird unterhalb des sichtbaren Formularausschnitts dargestellt (Abbildung 10.2)

2. Markieren Sie das Steuerelement, und klicken Sie in dessen Eigenschaftsfenster auf die Schaltfläche der Eigenschaft *Images*.
3. Jetzt erscheint ein Dialogfeld *Bildaufflistungs-Editor*, in dem Sie über die Schaltflächen *Hinzufügen* und *Entfernen* Bilder in die Image-List aufnehmen können (Abbildung 10.4).

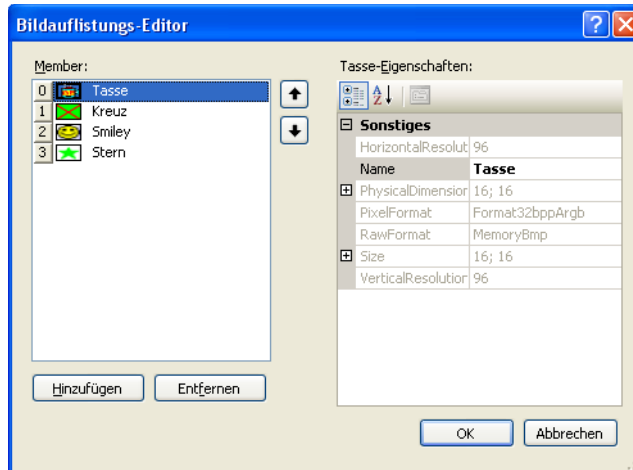


Abbildung 10.4: Dialogfeld des Bildaufflistungs-Editors

Sobald alle Bilddateien aufgeführt sind, schließen Sie das Dialogfeld über die *OK*-Schaltfläche. Das Steuerelement stellt die Bilder über Indizes von 0 bis n bereit. Nun gilt es noch, das *ImageList*-Steuerelement an die mittlere Schaltfläche anzubinden:

1. Markieren Sie im Formulardesigner erneut die mittlere Schaltfläche und wechseln Sie anschließend in das Eigenschaftsfenster dieses Steuerelements.
2. Klicken Sie auf die Eigenschaft *ImagesList* und wählen Sie im dann erscheinenden Listenfeld den Namen des *ImageList*-Steuerelements aus. Dadurch werden die Bilder für die Schaltfläche abrufbar.
3. Wählen Sie im nächsten Schritt im Listenfeld der Eigenschaft *ImageIndex* das gewünschte Bild.

Wenn alles geklappt hat, sollten die Eigenschaften gemäß Abbildung 10.3 zu sehen sein. Die Eigenschaft *Image* enthält dann das gewählte Symbol aus der *ImageList*. Übersetzen Sie das Formularbeispiel und führen Sie es aus, wird dieses die mittlere Schaltfläche mit dem Symbol zeigen.

10.1.5 Umschalten der angezeigten Symbole

Um das in der mittleren Schaltfläche angezeigte Symbol zu wechseln, müssen Sie eine geeignete Ereignisbehandlungsroutine implementieren. Hierzu bietet es sich an, das *CheckedChanged*-Ereignis der drei Optionsfelder zu verwenden. Die Ereignisbehandlungsroutine für das erste Optionsfeld besitzt folgenden Code:

```
Private Sub RadioButton1_CheckedChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles RadioButton1.CheckedChanged
    ' Wechsele das Symbol für die mittlere Schaltfläche
    Me.Button2.ImageIndex = 0
End Sub
```

Die Anweisung innerhalb der Prozedur setzt den Wert von *ImageIndex* auf das gewünschte Symbol des *ImageList*-Steuerelements. Dieses Steuerelement muss übrigens nicht mehr angegeben werden, da dieses über die Eigenschaft *ImageList* mit der Schaltfläche verbunden ist. Das Steuerelement besorgt sich also über den angegebenen Index das gewünschte Symbol.

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung realisierten Beispiels im Ordner `\Beisp\Kap10\Optionsfeld` auf der Begleit-CD. Der Visual-Basic-Code für das Formular findet sich in der Datei `Form1.vb`.

10.2 Arbeiten mit Auswahlfeldern

Zur Auswahl von Werten aus Listen können Listenfelder (*ListBox*), Kombinationsfelder (*ComboBox*) und Listenfelder mit Kontrollkästchen (*CheckedListBox*) benutzt werden. Listenfelder erlauben die Auswahl eines oder mehrerer Werte aus einer vordefinierten Liste. Bei Kombinationsfeldern kann aber immer nur ein Wert gewählt werden. Bei einer *CheckedListBox* findet sich vor jedem Eintrag ein Kontrollkästchen, welches zur Auswahl markiert werden muss. Kombinationsfelder bieten die Möglichkeit, dass der Benutzer einen Wert aus einer (aufklappenden) Liste wählt oder zusätzliche Texte frei eingeben kann. Die Handhabung dieser Steuerelementtypen soll jetzt an einem einfachen Formular demonstriert werden (Abbildung 10.5).

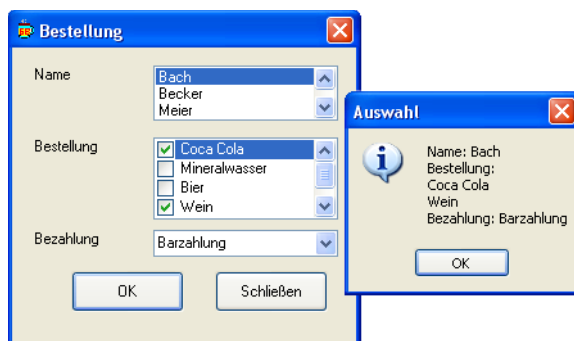


Abbildung 10.5: Formular mit Auswahlfeldern und Ergebnisanzeige

Alle drei Steuerelementtypen lassen sich über Schaltflächen der *Toolbox* im Formularlayout einfügen. *ListBox*- und *CheckedListBox*-Steuerelemente können dann eine oder mehrere Zeilen sowie eine oder mehrere Spalten aufweisen. Die Zahl der anzuzeigenden Zeilen wird dabei durch die Höhe des Steuerelements bestimmt. Ist die Eigenschaft *MultiColumn=True* gesetzt, werden die Werte in mehreren Spalten nebeneinander angezeigt. Die Spaltenbreite wird in diesem Fall über die Eigenschaft *ColumnWidth* festgelegt. Das Kombinationsfeld (hier das Feld *Bezahlung*) besteht aus einer Zeile. Die Liste mit den vorgegebenen Einträgen lässt sich vom Benutzer über eine Schaltfläche aufklappen. Über die Eigenschaft *SelectionMode* lässt sich festlegen, ob der Benutzer im Listenfeld einen oder mehrere Einträge auswählen darf. Die Vorgabewerte der Auswahlfelder lassen sich dabei sowohl beim Entwurf über die Eigenschaften als auch zur Laufzeit erzeugen.

10.2.1 Das Formular mit den Steuerelementen entwerfen

Der Entwurf des Formulars aus Abbildung 10.5 mit je einem *ListBox*-, *CheckedListBox*- und *ComboBox*-Steuerelement erfolgt mit folgenden Schritten:

1. Legen Sie in der Entwicklungsumgebung eine *Windows-Anwendung* als Projekt an und wechseln Sie anschließend zum Fenster des Formulardesigners. Passen Sie die Eigenschaften des leeren Formulars nach Ihren Wünschen an (Größe, Titelleiste, Schaltflächen, Stil der Ränder).
2. Fügen Sie aus der *Toolbox* die in Abbildung 10.5 gezeigten Steuerelemente *Button*, *Label*, *ListBox*, *CheckedListBox* und *ComboBox* im Formular ein.
3. Passen Sie die Eigenschaft *Text* der betreffenden Steuerelemente an. Beim *ComboBox*-Steuerelement sollte die Eigenschaft leer sein oder einen Vorgabewert aufnehmen. Beim *ListBox*-Steuerelement gibt es diese Eigenschaft nicht.
4. Füllen Sie die Auflistung für die Auswahlfelder mit den Vorgabebegriffen. Hierzu klicken Sie auf das Steuerelement und dann im Eigenschaftfenster auf die Eigenschaft *Items*. Sobald sich das in Abbildung 10.6 gezeigte Dialogfeld des *Zeichenfolgen-Editor* öffnet, tragen Sie die Begriffe ein und schließen dann das Dialogfeld über die *OK*-Schaltfläche.
5. Sollen die Einträge sortiert in der Liste erscheinen, setzen Sie die *Sorted*-Eigenschaft des betreffenden Steuerelements auf *True*.

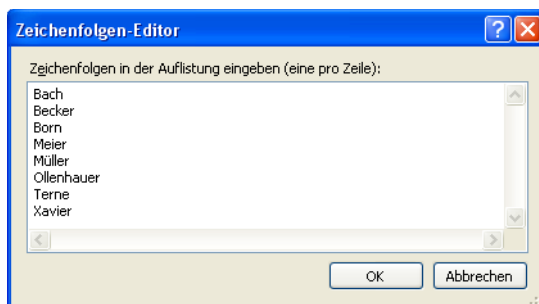


Abbildung 10.6: Vorgabewerte für Auswahlfelder

Passen Sie die Aktivierungsreihenfolge der Steuerelemente sowie deren sonstige Eigenschaften an und ergänzen Sie die Schaltfläche *Schließen* um eine *Click*-Ereignisprozedur, die über den Befehl *Me.Close()* das Formular schließt. Anschließen können Sie weitere Ereignisprozeduren ergänzen, die die Steuerelemente mit Werten füllen oder beim Anklicken der *OK*-Schaltfläche die Auswahl anzeigen (siehe folgende Abschnitte).

10.2.2 Auswahlfelder zur Laufzeit mit Werten füllen

Um ein *ListBox*-, *CheckedListBox*- oder *ComboBox*-Steuerelement zur Laufzeit mit Werten zu füllen, ist die *Add*-Methode der *Items*-Auflistung zu wählen. Die betreffenden Anweisungen lassen sich beispielsweise in der *Load*-Ereignisbehandlungsprozedur des Formulars unterbringen. Die folgenden Anweisungen verdeutlichen dies:

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
    ' Beim Laden des Formulars noch einige Einträge addieren  
    With Me  
        .ListBox1.Items.Add("Schrader")  
        .ComboBox1.Items.Add("Devisen")  
        .CheckedListBox1.Items.Add("Rotwein")  
    End With  
End Sub
```

Hier wird jedes der drei Auswahl-Steuerelemente mit einem neuen Eintrag versorgt. Um Elemente zu entfernen, können Sie die *Remove()*-Methode

```
Me.ListBox1.Items.Remove("Born")
```

benutzen. Weitere Details zu den Operatorüberladungen der Methoden entnehmen Sie bitte der Hilfe zum .NET Framework.

Hinweis

In den Formularbeispielen wird im Code über das *Me*-Objekt auf die darin enthaltenen Steuerelemente zurückgegriffen (z.B. *Me.ListBox1.Items.Add()*). Sie können bei Steuerelementen aber auch direkt den Objektnamen (z.B. *ListBox1.Items.Add()*) in den Anweisungen angeben, um auf die Eigenschaften und Methoden zuzugreifen.

10.2.3 Lesen der Benutzerauswahl

Damit bleibt noch die Auswertung der betreffenden Benutzereingaben bzw. -auswahl. In diesem Beispiel soll der Code in der *Click*-Ereignisbehandlungsprozedur der *OK*-Schaltfläche untergebracht werden. Sofern nur ein Element in einem Listenfeld zur Auswahl vorgesehen ist, lässt sich mit *.ListBox1.SelectedItem* der Wert des markierten Elements wählen. Der Name *ListBox1* steht hier für den Namen des Listenfelds. Bei einem *CheckListBox*-Steuerelement wird die Sache etwas aufwändiger. Dort kann der Benutzer

ja ein oder mehrere Kontrollkästchen auswählen. Die markierten Elemente werden von der *CheckedItems*-Auflistung zurückgeliefert. Die *Count*-Eigenschaft der *CheckItems*-Auflistung enthält die Zahl der Elemente. Ein Wert lässt sich über

```
CheckedListBox1.CheckedItems(i)
```

abrufen, wobei *i* für den Index zwischen 0 und *Count-1* steht. Bei einem *ComboBox*-Steuerelement gibt es die Möglichkeit, dass der Benutzer einen Vorgabewert auswählt oder selbst einen Wert ausgibt. Die Auswahl lässt sich über *.ComboBox1.SelectedItem* ermitteln (falls die Auswahl auf ein Element begrenzt ist). Eine Texteingabe des Benutzers wird einfach über die *Text*-Eigenschaft des Steuerelements abgefragt.

Falls eine Mehrfachauswahl des betreffenden Steuerelements freigegeben ist, müssen Sie die *SelectedItems*-Auflistung auswerten. Der folgende Codeausschnitt zeigt die Auswertung der Benutzereingaben für obiges Beispielformular:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    ' OK-Schaltfläche angeklickt, zeige Auswahl
    Dim txt As String
    Dim i As Integer
    Dim oCheck As CheckedListBox

    With Me
        ' Hole den Eintrag im Listefeld "Bestellung"
        ' entweder: .ListBox1.Items(.ListBox1.SelectedIndex) oder
        txt = "Name: " & .ListBox1.SelectedItem & vbCrLf
        txt = txt & "Bestellung: " & vbCrLf

        ' Werte jetzt das CheckedListBox-Feld aus
        For i = 0 To .CheckedListBox1.CheckedItems.Count - 1
            txt = txt & .CheckedListBox1.CheckedItems(i) & vbCrLf
        Next i
        If .ComboBox1.SelectedItem <> "" Then
            txt = txt & "Bezahlung: " & .ComboBox1.SelectedItem
        Else
            txt = txt & "Bezahlung: " & .ComboBox1.Text
        End If
        MessageBox.Show(txt, "Auswahl", _
            MessageBoxButtons.OK, MessageBoxIcon.Information)
    End With
End Sub
```

Listing 10.1: Zugriff auf Listen- und Kombinationsfelder

Hinweis

Sie finden die Projektdateien der Windows-Anwendung im Ordner `\Beisp\Kap10\Listenfeld` auf der Begleit-CD.

10.3 Arbeiten mit Einstellfeldern und Fortschrittsanzeige

Windows-Dialoge enthalten diverse Steuerelemente zum Einstellen von Werten. Die Palette reicht von einem numerischen Drehfeld (*NumericUpDown*), bei dem sich Werte eingeben oder über Schaltflächen schrittweise erhöhen/erniedrigen lassen, über *DomainUpDown*-Steuerelemente, die die Einstellung von Werten aus einer vordefinierten Liste ermöglichen, bis hin zu Schiebereglern (*TrackBar*). Zudem gibt es die Möglichkeit einer Fortschrittsanzeige (*ProgressBar*), um dem Benutzer den Status einer Operation anzuzeigen. Diese Steuerelemente sollen jetzt in einem kleinen Formular integriert und per Programm ausgewertet werden (Abbildung 10.7).

10.3.1 Einfügen der Steuerelemente in das Formular

Zum Erstellen des in Abbildung 10.7 gezeigten Formulars gehen Sie in folgenden Schritten vor:



Abbildung 10.7: Formular mit Einstellfeldern und Fortschrittsanzeigen

1. Legen Sie ein Projekt mit einem neuen Formular in der Entwicklungsumgebung an und setzen Sie die Eigenschaften des leeren Formulars nach Bedarf (Größe, Titelzeile, Schaltflächen, Stil der Ränder).
2. Fügen Sie aus der Werkzeugleiste *Toolbox* die in Abbildung 10.7 gezeigten Steuerelemente *NumericUpDown*, *DomainUpDown*, *TrackBar* und *ProgressBar* im Formular ein. Sie können die Elemente aus optischen Gründen mit *GroupBox*-Elementen einfassen. Beschriften Sie die Steuerelemente über zusätzliche *Label*-Elemente.
3. Passen Sie die Eigenschaften für die betreffenden Steuerelemente an. Bei *TrackBar*, *ProgressBar* und *NumericUpDown* sind die minimalen und maximalen Werte sowie die Schrittweiten über Eigenschaften vorzugeben.

4. Beim *DomainUpDown* lassen sich die Texteinträge der Einstellvorgaben als Auflistung über die Eigenschaft *Items* definieren. Sollen die Einträge sortiert in der Liste erscheinen, setzen Sie die *Sorted*-Eigenschaft des betreffenden Steuerelements auf *True*.

Setzen Sie ggf. die Aktivierungsreihenfolge der Steuerelemente sowie deren sonstige Eigenschaften und ergänzen Sie die Schaltfläche *Schließen* um eine *Click*-Ereignisbehandlungsprozedur, die das Formular über den Befehl *Me.Close()* schließt.

10.3.2 Beispiel um Ereignishandler ergänzen

Das Formularbeispiel soll so ergänzt werden, dass die *OK*-Schaltfläche die aktuellen Einstellungen in einem Dialogfeld anzeigt. Beim Laden sind die Werte der Steuerelemente teilweise auf Initialisierungswerte zu setzen. Ändert der Benutzer die Einstellung des Schiebereglers, ist der Wert in der Fortschrittsanzeige zu aktualisieren. Die Initialisierung der Steuerelemente erfolgt in der *Load*-Ereignisbehandlungsprozedur des Formulars mit folgendem Code:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    ' Initialisiere beim Laden
    NumericUpDown1.Value = 25 ' Drehfeld
    DomainUpDown1.Text = "(Auto)"
    DomainUpDown1.Items.Add("(Auto)") ' neuer Eintrag
    TrackBar1.Value = 5 ' Schieberegler 50%
    ProgressBar1.Value = 5
    Text = Titel
    Label5.Text = L1 & TrackBar1.Value
End Sub
```

Ein *NumericUpDown*-Steuerelement wird über die Eigenschaft *Value* mit einem Vorgabewert belegt. Bei einem *DomainUpDown*-Steuerelement definiert die *Text*-Eigenschaft den aktuell angezeigten Wert. Dieser wird in obigem Code auf *(Auto)* gesetzt. Möchten Sie einen Vorgabewert in die Wertauflistung des Steuerelements einfügen, ist die *Add*-Methode der *Items*-Auflistung zu verwenden. Die obigen Anweisungen verwenden zudem das *Label*-Steuerelement der Fortschrittsanzeige, um den aktuellen Wert als numerische Zahl anzuzeigen.

Änderungen des Schiebereglers lassen sich über das *TrackBarScroll*-Ereignis abfangen. Der folgende Code im Ereignishandler liest den aktuellen Wert (Eigenschaft *Value*) des Schiebereglers und weist diesen der gleichnamigen Eigenschaft der Fortschrittsanzeige zu. Bei beiden Steuerelementen wurden die Min/Max-Werte so gesetzt, dass die Skalierung übereinstimmt. Die Schrittweite der beiden Steuerelemente bestimmt dann, wie sich eine Änderung des Schiebereglers auf die Fortschrittsanzeige auswirkt.

```
Private Sub TrackBar1_Scroll(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles TrackBar1.Scroll
```

```
' Schieberegler
Me.ProgressBar1.Value = Me.TrackBar1.Value
Me.Label15.Text = L1 & Me.TrackBar1.Value
End Sub
```

Die erste Anweisung innerhalb der Prozedur speichert den Wert des Schiebereglers in der Fortschrittsanzeige. Die zweite Anweisung hängt den Wert des Schiebereglers als Ziffern an das *Label*-Steuerelement an. Zur Anzeige der aktuellen Einstellungen beim Anklicken der OK-Schaltfläche wird folgender Code im *Click*-Ereignis dieser Schaltfläche eingesetzt:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
' OK-Schaltfläche, Werte anzeigen
Dim txt As String = ""
txt = txt & "Hersteller (DomainUpDown): " & _
    DomainUpDown1.Text & vbCrLf
txt = txt & "Alter (NumericUpDown): " & _
    NumericUpDown1.Value & vbCrLf
txt = txt & "Auflösung (TrackBar): " & _
    TrackBar1.Value & vbCrLf
txt = txt & "Wert (ProgressBar): " & _
    ProgressBar1.Value & vbCrLf
MessageBox.Show(txt, "Werte", _
    MessageBoxButtons.OK, MessageBoxIcon.Information)
End Sub
```

Diese Anweisungen erstellen lediglich eine Zeichenkette aus den Werten der *Text*- und *Value*-Eigenschaften und zeigen diese über die *Show()*-Methode der *MessageBox*-Klasse an. Also alles bereits häufiger genutzte Techniken.

Hinweis

Sie finden die Projektdateien dieses als Windows-Anwendung realisierten Beispiels im Ordner *\Beisp\Kap10\Drehfeld* auf der Begleit-CD.

10.4 Registerkarten und Textelemente verwenden

Um längeren Text zu bearbeiten, können Sie ein einzeliges Textfeld in ein mehrzeiliges Textfeld umwandeln. Dieses lässt sich dabei mit Bildlaufleisten versehen, um im Text blättern zu können. Zusätzlich bietet das .NET Framework im Namensraum *System.Windows.Forms* ein sogenanntes *RichTextBox*-Steuerelement, mit dem sich formatierte Texte (RichText) handhaben lassen.

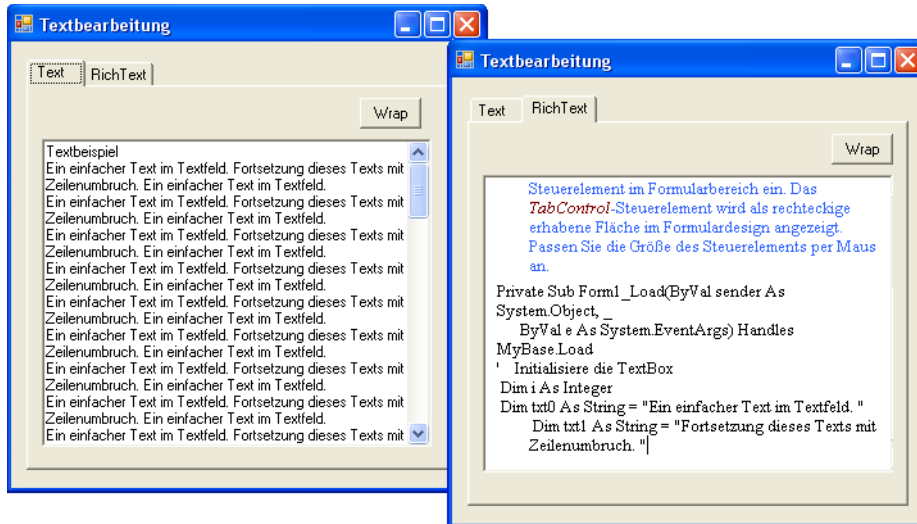


Abbildung 10.8: Dialogfeld mit zwei Registerkarten für Text und RichText

Um mehrere Steuerelemente in einem Dialogfeld unterzubringen, kommen in diesem Beispiel auch Registerkarten zum Einsatz. Die Nutzung der drei Steuerelemente *TabControl*, *TextBox* und *RichTextBox* soll jetzt in einem Beispiel demonstriert werden. Die beiden Steuerelemente zur Textanzeige werden in einem Dialogfeld auf zwei Registerkarten untergebracht. Dies ermöglicht Ihnen, das Verhalten der Steuerelemente zu untersuchen (Abbildung 10.8).

10.4.1 Einfügen eines TabControl-Steuerelements

Um Registerkarten in einem Formular nutzen zu können, müssen Sie ein *TabControl*-Steuerelement aus der Symbolleiste *Toolbox* im Formulardesign einfügen. Anschließend sind die einzelnen Registerkarten (als *TabPage*-Steuerelemente) hinzuzufügen und mit Eigenschaften zu versehen. Diese Schritte sollen exemplarisch an der oben skizzierten Beispielanwendung demonstriert werden. Als Erstes wird ein Formular mit zwei Registerkarten benötigt:

1. Erstellen Sie ein neues Projekt vom Typ einer Windows-Anwendung, wechseln Sie zum Formulardesign und passen Sie als Erstes die Abmessungen des Formulars, dessen Titelleiste und dessen restliche Eigenschaften nach Bedarf an.
2. Fügen Sie aus der *Toolbox* ein *TabControl*-Steuerelement im Formularbereich ein. Das *TabControl*-Steuerelement wird als rechteckige erhöhte dargestellte Fläche im Formulardesign angezeigt. Passen Sie die Größe des Steuerelements per Maus an und verankern Sie dieses über die *Anchor*-Eigenschaft an den Formularrändern. Dies bewirkt, dass das Steuerelement zur Laufzeit in der Größe jeweils an die Formularabmessungen angepasst wird.

Das Steuerelement enthält nach dem Einfügen im Formularlayout bereits zwei Registerkarten. Bei Bedarf können Sie aber weitere Registerkarten zum Steuerelement hinzufügen. Das *TabControl*-Steuerelement verwaltet die Registerkarten als Auflistung über die Eigenschaft *TabPage*.

1. Wechseln Sie zum Eigenschaftenfenster des *TabControl* und wählen Sie die Eigenschaft *TabPage*. In der betreffenden Zeile wird eine Schaltfläche angezeigt, die Sie anklicken müssen.
2. Die Entwicklungsumgebung öffnet nun das Dialogfeld des *TabPage*-Auflistungs-Editors zur Verwaltung der *TabPage*s (Abbildung 10.9). Klicken Sie auf die Schaltfläche *Hinzufügen*, um einen neuen Member in die Auflistung hinzuzufügen. Jeder Member erhält einen vorgegebenen Objektnamen (z.B. *TabPage1*). Einen markierten Member können Sie über die Schaltfläche *Entfernen* löschen. Die beiden Schaltflächen rechts neben der Liste *Member* erlauben die Reihenfolge der Registerkarten zu ändern.
3. Jeder Member der Auflistung stellt ein Registerkartenobjekt (*TabPage*) dar, welches wiederum Eigenschaften besitzt. Diese Eigenschaften werden sichtbar, sobald der Member markiert wurde. Tragen Sie für jede *TabPage* den für den Registerreiter vorgesehenen Text in der Eigenschaft *Text* ein. In Abbildung 10.9 wurde der zweiten Registerkarte der Text *RichText* zugewiesen, d.h., diese Beschriftung erscheint zur Laufzeit auf dem Registerreiter der Karte.

Sobald alle benötigten Registerkarten zur Auflistung hinzugefügt wurden, können Sie das Dialogfeld des Editors schließen. Das Steuerelement bietet bereits die komplette Logik, um zwischen den Registerkarten umzuschalten.

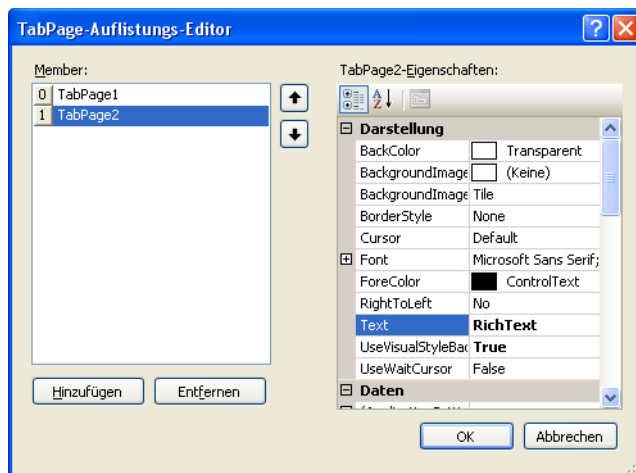


Abbildung 10.9: Dialogfeld des *TabPage*-Auflistungs-Editors

10.4.2 Ein mehrzeiliges Textfeld mit Bildlaufleiste einrichten

Im nächsten Schritt ist im Formularlayout ein mehrzeiliges Textfeld mit Bildlaufleisten auf der Registerkarte *Text* einzurichten (Abbildung 10.10). Positionieren Sie das Steuerelement so, dass noch Platz für eine Schaltfläche *Wrap* bleibt.

1. Holen Sie bei Bedarf im Formularlayout die mit *Text* beschriftete Registerkarte durch einen Klick auf den Registerreiter in den Vordergrund.
2. Fügen Sie aus der *Toolbox* ein *TextBox*-Steuerelement im Formularbereich ein. Es wird zunächst nur ein einzeiliges Textfeld angezeigt.
3. Markieren Sie das Steuerelement und wechseln Sie zum Eigenschaftfenster. Setzen Sie dort die Eigenschaft *MultiLine* auf den Wert *True*. Die so gesetzte Eigenschaft ermöglicht Ihnen nun, das Textfeld im Formularlayout auf die gewünschte Größe zu bringen.
4. Anschließend wählen Sie die *Anchor*-Eigenschaft und verankern das Steuerelement an den vier Rändern des übergeordneten Containers (damit das *Textfeld* automatisch bei Größenänderungen des übergeordneten Containers mit angepasst wird).
5. Stellen Sie anschließend noch die Eigenschaft *ScrollBars* auf den Wert *Both*, um dem Element Bildlaufleisten zuzuweisen. Diese Bildlaufleisten werden zur Laufzeit nur bei Bedarf angezeigt.

Nach diesen Vorbereitungen sollten Sie noch die Schaltfläche *Wrap* im Formularentwurf einfügen. Über diese Schaltfläche soll zur Laufzeit der automatische Zeilenumbruch im Textfeld ein- oder ausschaltbar sein.

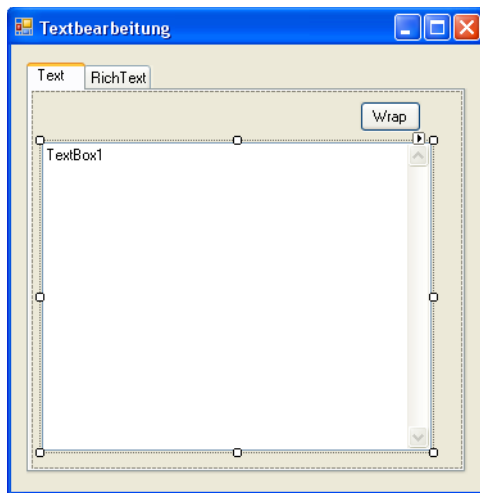


Abbildung 10.10: Registerkarte Text mit Textbox und Schaltfläche im Formularentwurf

Tipp

Eine nette Neuerung ist der SmartTag-Bereich, der bei einigen Steuerelementen (z.B. TabControl, TextBox etc.) im Formularbereich eingeblendet wird. Sobald Sie das Steuerelement im Designer anwählen, wird dann eine kleine Schaltfläche in der rechten oberen Ecke des Markierungsrahmens eingeblendet (Abbildung 10.10). Klicken Sie auf diese Schaltfläche, öffnet sich der SmartTag-Bereich und zeigt die für das Steuerelement verfügbaren Aufgaben an. Sie können also bei einem *TextBox*-Steuerelement die Eigenschaft *MultiLine* auch direkt im SmartTag-Bereich *Textaufgaben* durch Markieren des Kontrollkästchens *MultiLine* auf *True* setzen.

Text einstellen und Zeilenumbruch zur Laufzeit umschalten

Beim Laden des Formulars soll das Textfeld mit einem kurzen Beispieltext gefüllt werden. Zudem soll die Schaltfläche *Wrap* dem Benutzer das Umschalten des automatischen Zeilenumbruchs (für über den rechten Rand hinausreichende Zeilen) erlauben. Der automatische Zeilenumbruch lässt sich über die Eigenschaft *WordWrap* beeinflussen. Weisen Sie dieser Eigenschaft die Werte *True* oder *False* zu. Zur Initialisierung des Steuerelements wird das *Load*-Ereignis des Formulars benutzt. Der nachfolgende Code enthält die betreffenden Anweisungen (das *RichTextBox*-Control wird nicht mit Text initialisiert):

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
'    Initialisiere die TextBox  
Dim i As Integer  
Dim txt0 As String = "Ein einfacher Text im Textfeld. "  
Dim txt1 As String = "Fortsetzung dieses Texts mit Zeilenumbruch. "  
Dim txt As String = "Textbeispiel" & vbCrLf  
For i = 0 To 40 ' 10 Zeilen Text  
    txt = txt & txt0 & txt1 & txt0 & vbCrLf  
Next  
Me.TextBox1.Text = txt  
End Sub
```

Es wird hier eine Zeichenkette zusammengebaut und dann der Eigenschaft *Text* der *TextBox* zugewiesen. Die Konstante *vbCrLf* sorgt für einen manuellen Zeilenumbruch im Textfeld.

Die Implementierung der Ereignisbehandlungsprozedur als Reaktion auf einen Mausklick auf die Schaltfläche *Wrap* enthält folgende Anweisungen:

```
Private Sub Button3_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button3.Click  
' Wrap in TextBox umkehren  
If Me.TextBox1.WordWrap Then  
    Me.TextBox1.WordWrap = False
```

```

        Me.Button3.FlatStyle = FlatStyle.Standard
    Else
        Me.TextBox1.WordWrap = True
        Me.Button3.FlatStyle = FlatStyle.Flat
    End If
End Sub

```

In der Prozedur wird der Status der Eigenschaft *WordWrap* geprüft. Ist diese auf *True* gesetzt, schaltet der Code die Eigenschaft auf *False* und umgekehrt. Gleichzeitig wird noch die Eigenschaft *FlatStyle* der Schaltfläche zwischen *Standard* und *Flat* umgeschaltet, um den Status des automatischen Zeilenumbruchs zu signalisieren.

Hinweis

Wenn Sie das Beispiel ausführen, lassen sich die Zeilen des Textfelds über die Schaltfläche *Wrap* umbrechen. Ohne diesen Zustand reichen die Zeilen über den rechten Rand und das Textfeld erhält eine horizontale Bildlaufleiste zum Blättern. Das Textfeld stellt bereits Funktionen zum Markieren des Inhalts sowie zum Ausschneiden, Kopieren und Einfügen per Zwischenablage bereit.

10.4.3 Verwenden eines RichTextBox-Elements

Der Namespace *System.Windows.Forms* stellt noch eine zweite Variante eines Steuerelements zur Aufnahme von Texten bereit. Es handelt sich um das *RichTextBox*-Steuerelement, welches formatierten Text darstellen kann. Das Steuerelement verwendet das Rich Text Format (RTF), welches auch in den Windows-Programmen WordPad und Microsoft Word benutzt wird. Ein solches Steuerelement soll jetzt auf der Registerkarte *RichText* eingerichtet werden (Abbildung 10.11). Positionieren Sie das Steuerelement so, dass noch Platz für eine Schaltfläche *Wrap* bleibt.



Abbildung 10.11: Registerkarte RichText mit eingefügtem Text

1. Holen Sie im Formularlayout die mit *RichText* beschriftete Registerkarte durch einen Klick auf den Registerreiter in den Vordergrund.
2. Fügen Sie aus der *Toolbox* ein *RichTextBox*-Steuerelement im Formularbereich ein und passen Sie dessen Größe im Layout an.
3. Markieren Sie das Steuerelement und stellen Sie anschließend die Eigenschaft *ScrollBars* auf den Wert *Both*, um dem Element Bildlaufleisten zuzuweisen. Diese Bildlaufleisten werden zur Laufzeit nur bei Bedarf angezeigt.
4. Wählen Sie im Eigenschaftsfenster ggf. die Eigenschaften der Gruppe *Font* und setzen Sie die Eigenschaften für Schriftgrad, für Fett und Kursiv etc.

Fügen Sie, wie bereits oben beim Textfeld beschrieben, noch die Schaltfläche *Wrap* im Formularentwurf ein.

Hinweis

Über die *Toolbox* lassen sich auch Steuerelemente für horizontale (*HScrollBar*) und vertikale (*VScrollBar*) Bildlaufleisten in ein Formular einbringen. Dieser Ansatz wird hier aber nicht behandelt, da die Steuerelemente zur Textdarstellung (und weitere Steuerelemente) die Bildlaufleisten intern unterstützen. Hinweise zur Verwendung der Bildlaufleisten-Steuerelemente können Sie der Hilfe des .NET Framework entnehmen.

Das RichTextBox-Element mit Code ergänzen

Das Element soll beim Laden des Formulars mit einem kurzen Beispieltext gefüllt werden und den Zeilenumbruch über die Schaltfläche *Wrap* unterstützen. Die Anweisungen zum Umschalten des automatischen Zeilenumbruchs sind identisch mit dem Code des Textfelds, es ist lediglich der Objektname *RichTextBox1* einzusetzen. Das Gleiche gilt für die Initialisierung des *RichTextBox*-Steuerelements mit Text:

```
Me.RichTextBox1.Text = "..."
```

In diesem Beispiel wird darauf verzichtet, Code zum Formatieren der einzelnen Textabschnitte bereitzustellen. Hinweise zu diesem Thema finden Sie z.B. in Kapitel 12 sowie in der Hilfe zum .NET Framework unter dem Stichwort »RichTextBox-Klasse«.

Hinweis

Wenn Sie das Beispiel ausführen, können Sie aber einen formatierten Text aus Word oder aus der Entwicklungsumgebung per Zwischenablage in das *RichTextBox*-Steuerelement übertragen. Dort werden die Formatierungen beibehalten. Die Schaltfläche *Wrap* ermöglicht, den automatischen Zeilenumbruch am rechten Rand ein- oder auszuschalten. Sie finden die Projektdateien des Beispiels im Ordner `\Beisp\Kap10\Text` auf der Begleit-CD.

10.5 Anzeige von Bildern und Kalenderdaten

Die Anzeige von Bildern in Formularen hatten wir bereits beim *About*-Beispiel im vorherigen Kapitel angerissen. Jetzt möchte ich ein Beispiel vorstellen, welches ein Dialogfeld mit zwei Registerkarten anzeigt. Auf der Registerkarte *Bild* wird ein Bild ausgegeben. Über die beiden Schaltflächen *Vorwärts* und *Zurück* kann der Benutzer zwischen mehreren Bildern blättern (Abbildung 10.12, links). Die Bilder kommen dabei wechselweise aus einer Datei und aus einem *ImageList*-Steuerelement. Die Registerkarte *Kalender* (Abbildung 10.12, rechts) enthält ein Kalender-Steuerelement sowie zwei Felder, um das Datum oder die Zeit auszuwählen. Die Schaltfläche *Wochen* erweitert die Kalenderanzeige um die Angabe der jeweiligen Kalenderwochen. Mit der Schaltfläche *Show* lassen sich die aktuellen Werte der Steuerelemente in einem Dialogfeld abrufen.

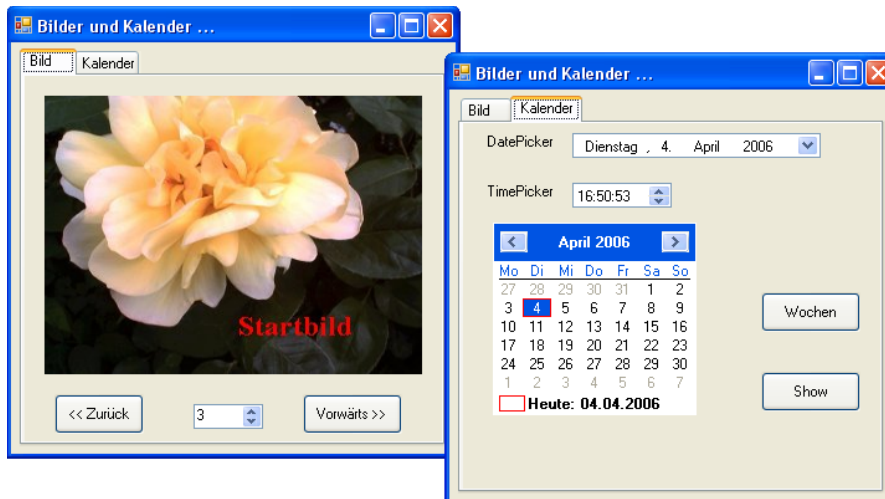


Abbildung 10.12: Anzeige von Bildern und Kalenderdaten

10.5.1 Entwurf der Registerkarte zur Bildanzeige

Zum Entwerfen des Formulars gehen Sie in der Entwicklungsumgebung in folgenden Schritten vor:

1. Legen Sie in der Entwicklungsumgebung ein neues Projekt als Windows-Anwendung an, setzen dann die Projekteigenschaften (z.B. Symbol für die Programmdatei) und wechseln zum Formularentwurf. Passen Sie die Formulareigenschaften (Titeltext, Größe) den Anforderungen an.
2. Fügen Sie über die *Toolbox* ein *ImageList*-Steuerelement zur Aufnahme der Bilder zum Formularentwurf hinzu. Das Steuerelement wird als nicht sichtbare Komponente am unteren Rand des Designfensters eingeblendet.

3. Wählen Sie anschließend die Schaltfläche der Eigenschaft *Images* und definieren Sie im Bildauflistungs-Editor die im *ImageList*-Steuerelement aufzunehmenden Bilder (Abbildung 10.13). Ein Bild lässt sich aus einer Datei mittels der Schaltfläche *Hinzufügen* aufnehmen und mittels *Entfernen* wieder löschen.
4. Fügen Sie anschließend ein *TabControl*-Steuerelement zum Formular hinzu. Benennen Sie die standardmäßig vorhandenen zwei Registerkarten (*TabPage*-Steuerelemente) über die Eigenschaft *Text* und verankern Sie das Steuerelement an den Formlarrändern. Die entsprechenden Schritte wurden bereits im vorhergehenden Beispiel erläutert.
5. Wechseln Sie zur Registerkarte *Bild* und fügen Sie ein *PictureBox*-Steuerelement hinzu. Markieren Sie das *PictureBox*-Steuerelement und weisen Sie in dessen Eigenschaftfenster über die Eigenschaft *Image* eine Bilddatei als Startbild zu. Die Übertragung der Bilder aus der *ImageList*-Komponente erfolgt später programmgesteuert.
6. Kontrollieren Sie die Eigenschaften für die Größe und die Auflösung des *PictureBox*-Steuerelements. Setzen Sie zudem den Anzeigemodus über die Eigenschaft *SizeMode*. Der Wert *AutoSize* bewirkt die Anpassung der Größe des Steuerelements an die Bildgröße, während der Wert *CenterImage* das Bild im Clientbereich zentriert. Mit *ImageStretch* passt das Steuerelement das Bild an die Größe des Clientbereichs an.
7. Fügen Sie ein *NumericUpDown*-Control sowie die beiden Schaltflächen zum Blättern zur Registerkarte hinzu. Benennen Sie die beiden Schaltflächen und setzen Sie die Eigenschaften des *NumericUpDown*-Steuerelements. Die Eigenschaft *Increment* sollte auf 1 stehen. Zusätzlich können Sie den minimalen und maximalen Wert vorgeben (obwohl dies im Beispiel per Programm erfolgt).

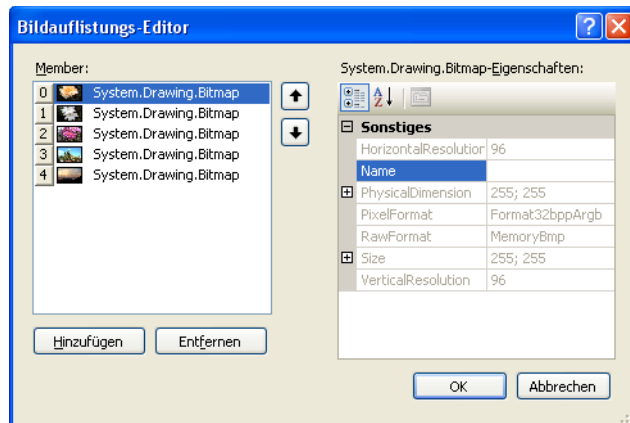


Abbildung 10.13: Dialogfeld des Bildauflistungs-Editor

Mit diesen Vorbereitungen ist das Formular mit der Registerkarte *Bild* bereits vorbereitet. Wenn Sie anschließend das Projekt übersetzen und ausführen, wird zumindest das Startbild auf der Registerkarte angezeigt. Die restlichen Funktionen werden nun programmgesteuert hinzugefügt.

Achtung

Überprüfen Sie beim Layout des Formulars die Eigenschaft *ColorDepth* des *ImageList*-Steuerelements und passen Sie diese gemäß Ihren Anforderungen an. Ist die Farbtiefe auf 256 Farben gesetzt, wird das *ImageView*-Steuerelement keine fotorealistischen Bilder anzeigen können! Die Bilddateien werden von der *ImageList* als Ressourcen mit im Projekt hinterlegt (die Entwicklungsumgebung sorgt automatisch dafür, dass die Ressourcen beim Übersetzen mit eingebunden werden). Sie können aber auch auf das *ImageList*-Steuerelement verzichten und die Bilder direkt aus Dateien einlesen (wird weiter unten diskutiert).

Programmcode zur Initialisierung

Wählen Sie die Registerkarte *Bild* an, wird das Ereignis *Enter* des betreffenden *TabPage*-Steuerelements ausgelöst. Die betreffenden Ereignisbehandlungsprozeduren sind der richtige Ort, um Initialisierungen vorzunehmen. Die Eigenschaften für Minimum und Maximum des *NumericUpDown*-Steuerelements werden in der *Enter*-Ereignisbehandlungsroutine der Registerkarte zugewiesen. Die ermöglicht Ihnen zur Laufzeit der Anwendung ggf. die Aufnahme weiterer Bilddateien in das *ImageList*-Steuerelement. Die Ereignisbehandlungsprozedur enthält folgenden Code:

```
Private Sub TabPage3_Select(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles TabPage3.Enter
    ' TabPage Bilder wählen
    Me.NumericUpDown1.Minimum = 0
    Me.NumericUpDown1.Maximum = Me.ImageList1.Images.Count - 1
End Sub
```

Die Eigenschaft *Minimum* wird fest auf 0 gesetzt, da Bildindizes des *ImageList*-Steuerelements mit diesem Wert beginnen. Der Eigenschaft *Maximum* wird die Eigenschaft *Count-1* der *ImageList.Images*-Auflistung zugewiesen. Die Anweisung begrenzt die Werte, die das Drehfeld annehmen kann, auf den Bereich der Bildindizes in der *ImageList*.

Der Startwert (Eigenschaft *Text*) des *NumericUpDown*-Steuerelements ist in der *Load*-Ereignisbehandlungsroutine des Formulars auf den Wert "3" zu setzen. Ich habe fünf Bilder in der *ImageList* eingefügt, der Wert 3 erlaubt also ein Blättern in beiden Richtungen. Die Aufnahme in die *Load*-Prozedur bewirkt, dass dieser Startwert nur einmalig beim Aufruf der Anwendung und nicht bei jedem Wechsel zur Registerkarte gesetzt wird. Zudem soll im Beispiel das Startbild aus einer Datei übernommen werden. Die betreffende *Load*-Prozedur enthält die folgenden Anweisungen:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim bmp As Bitmap
    Me.NumericUpDown1.Text = "3" ' Init UpDown-Control
    bmp = Tools.GetBmp(Tools.GetPath() & "Blume.jpg")
```

```

If Not IsNothing(bmp) Then
    Me.PictureBox1.Image = bmp
Else ' falls Bilddatei nicht existiert
    Me.PictureBox1.Image = ImageList1.Images(1)
End If
End Sub

```

Die Eigenschaft *Image* der *PictureBox* erwartet einen Wert vom Typ *Bitmap*. Um den betreffenden Datentyp komfortabel aus einer Datei zu generieren, habe ich den gesamten Code in zwei Funktionen der Klasse *Tools* verlagert. Die Funktion *GetBmp* liefert den betreffenden Datentyp und erwartet als Argument einen Pfad zur Bilddatei. Die Datei *Blume.jpg* muss im Verzeichnis der auszuführenden Anwendung hinterlegt sein. Die Funktionsmethode *GetBmp* ist dabei folgendermaßen implementiert:

```

Shared Function GetBmp(ByVal file As String) As Bitmap
' ermittele BMP aus Datei
Dim bmp As Bitmap
Try
    bmp = New Bitmap(file)      ' Bild zuweisen
Catch Err As Exception
    MsgBox("Fehler Datei" & file & " nicht gefunden")
    bmp = Nothing              ' Objekt unbelegt
' Exit Function                ' Abbrechen
End Try
Return bmp
End Function

```

Letztendlich wird mit *New Bitmap(file)* eine neue Instanz des *Bitmap*-Datentyps angelegt und als Funktionsergebnis zurückgegeben. Die restlichen Anweisungen dienen zum Abfangen von Fehlern, die hier über ein Dialogfeld gemeldet werden. Diese Details haben Sie bereits im *About*-Beispiel im vorherigen Kapitel kennen gelernt. Auch die Funktionsmethode *GetPath()* zur Ermittlung des aktuellen Pfads, aus dem die Anwendung gestartet wurde, ist dort behandelt.

Hinweis

Die *Function Exit*-Anweisung wurde hier auskommentiert, da sie zu einem Problem führt: Existiert die angegebene Datei nicht, würde die Funktion verlassen, ohne dass der Rückgabewert definiert ist. In der aktuellen Implementierung liefert die Funktion bei einer nicht vorhandenen Bilddatei den Wert »Nothing« zurück. In der *Load*-Ereignisprozedur wird der Rückgabewert überprüft. Wurde kein *Bitmap*-Objekt zurückgegeben, setzt das Programm hilfsweise ein Bild aus der *ImageList* ein. Vergessen Sie in Ihrem Projektcode diese Überprüfung, tritt ein Laufzeitfehler auf.

Programmcode zur Steuerung der Bildanzeige

Beim Klicken auf eine der Schaltflächen oder beim Betätigen des Drehfelds soll in der Bildliste geblättert werden. Dabei sind die aktuelle Bildnummer im Drehfeld zu führen und die Steuerelemente zu koordinieren. Wird der untere oder obere Bildindex erreicht, soll die Anwendung die betreffende Schaltfläche sperren. Beim Drehfeld habe ich die betreffenden Anweisungen in der *ValueChanged*-Ereignisbehandlungsroutine hinterlegt:

```
Private Sub NumericUpDown1_ValueChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles NumericUpDown1.ValueChanged
' Drehfeld - Bild aktualisieren
Dim j As Integer = CInt(Me.NumericUpDown1.Value)
With Me
    .PictureBox1.Image = .ImageList1.Images(j)
    .Button4.Enabled = True ' Schaltflächen freigeben
    .Button3.Enabled = True
    If j <= 0 Then ' an Grenze 0 festhalten
        .Button3.Enabled = False ' Schaltfläche Zurück sperren
    End If
    If j >= .ImageList1.Images.Count - 1 Then ' an Obergrenze festhalten
        .Button4.Enabled = False ' Schaltfläche Weiter sperren
    End If
End With
End Sub
```

Die Anweisung *.PictureBox1.Image = .ImageList1.Images(j)* weist dem *PictureBox*-Steuerelement ein Bild aus der *Images*-Auflistung des *ImageList1*-Steuerelements zu. Der Index *j* wird dabei aus dem *NumericUpDown1*-Steuerelement aus der Eigenschaft *Value* übernommen. Da es sich um einen *String*-Wert handelt, wurde eine Typwandlung mittels *CInt* vorgenommen. Die restlichen Anweisungen sorgen dafür, dass die Schaltflächen gesperrt werden, sobald der Wert des Drehfelds eine der Grenzen erreicht.

Der Code zur Behandlung des *Click*-Ereignisses der Schaltflächen ist ähnlich strukturiert. Hier sehen Sie die Ereignisbehandlungsprozedur für die Schaltfläche *Zurück*:

```
Private Sub Button3_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button3.Click
' Schaltfläche Zurück
Dim i As Integer = CInt(NumericUpDown1.Value)
With Me
    i -= 1
    If i <= 0 Then ' an Grenze 0 festhalten
        i = 0
    End If
End With
End Sub
```

```

.Button3.Enabled = False ' Schaltfläche sperren
End If
.PictureBox1.Image = .ImageList1.Images(i)
.NumericUpDown1.Value = i
.Button4.Enabled = True ' Schaltfläche Weiter freigeben
End With
End Sub

```

Der Wert des Drehfelds wird gelesen, in einen Integerwert konvertiert, dekrementiert und dann auf die Grenzwerte überprüft. In der letzten Anweisung wird der so ermittelte Wert in das Drehfeld zurückgeschrieben (eine Betätigung der Schaltflächen aktualisiert also auch den Wert im Drehfeld). Zudem weist die Ereignisbehandlungsroutine dem *ImageView*-Steuerelement über den laufenden Bildindex *i* ein Bild aus dem *ImageList*-Steuerelement zu. Die Ereignisbehandlung der Schaltfläche *Vorwärts* ist analog angelegt.

Hinweis

Die Projektdateien des Beispiels finden sich im Ordner *\Beisp\Kap10\ImgView* der Begleit-CD. Die Bilddateien zur Anzeige finden sich im Unterordner *\Images*. Wenn Sie den betreffenden Code um eine Funktion zum Einlesen aller Bilddateien aus einem Ordner erweitern, haben Sie quasi einen Bildbetrachter realisiert.

10.5.2 Entwurf der Registerkarte zur Kalenderanzeige

Der Namensraum *System.Windows.Forms* des .NET Framework stellt zwei Steuerelemente *MonthCalendar* und *DateTimePicker* bereit. Das Steuerelement *MonthCalendar* zeigt ein Kalenderblatt und *DateTimePicker* erlaubt das komfortable Abrufen von Datums- und Zeitwerten (Abbildung 10.14). Sofern Sie das obige Beispiel in der Entwicklungsumgebung bereits entworfen haben, können Sie die zweite Registerkarte um die betreffenden Komponenten ergänzen:

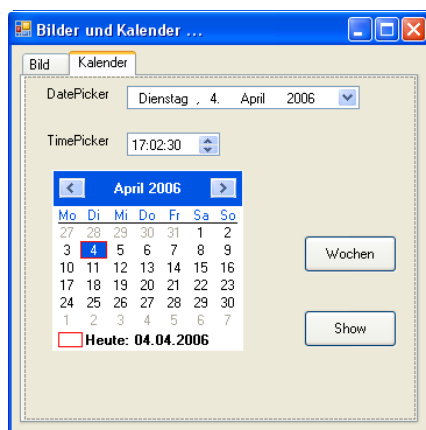


Abbildung 10.14: Registerkarte mit *MonthCalendar*- und *DateTimePicker*-Steuerelementen

1. Fügen Sie die beiden Steuerelemente *MonthCalendar* und *DateTimePicker* gemäß Abbildung 10.14 zur Registerkarte hinzu. Das *DateTimePicker*-Element ist dabei zweimal anzulegen. Ergänzen Sie dann noch die Registerkarte um zwei Schaltflächen und die benötigten Beschriftungsfelder.
2. Markieren Sie das obere *DateTimePicker*-Steuerelement und wechseln Sie zum Eigenschaftfenster. Setzen Sie die Eigenschaft *Format* dieses Elements auf den Wert *Long*. Dies erzwingt die Anzeige eines Datumswerts im Langformat mit dem Namen des Wochentags. Die Eigenschaft *ShowCheckbox* erlaubt es, ein Kontrollkästchen im Datumsfeld einzublenden und wird hier auf *False* belassen. Der gleiche Wert wird der Eigenschaft *ShowUpDown* zugewiesen. Damit wird das Steuerelement als aufklappbares Listenfeld angezeigt.
3. Wechseln Sie zum zweiten *DateTimePicker*-Steuerelement und setzen Sie dessen Eigenschaft *Format* auf den Wert *Time*, um die Anzeige eines Zeitwerts zu erzwingen. Die Eigenschaft *ShowUpDown* wird auf *True* gesetzt, um die Schaltflächen eines Drehfelds am rechten Rand einzublenden. Die Zeitwerte lassen sich dann über diese Schaltfläche anpassen (Abbildung 10.14).

Beim *MonthCalendar*-Steuerelement werden keine weiteren Eigenschaften gesetzt. Bei Bedarf können Sie im Eigenschaftfenster aber den roten Kringel um das aktuelle Datum (Eigenschaft *ShowTodayCircle*) oder die Anzeige des Tagesdatums (Eigenschaft *ShowToday*) ausblenden. Mit diesen Schritten ist das Formular mit der Registerkarte *Kalender* fertig, es bleibt nur noch, die Ereignisbehandlungsroutinen hinzufügen.

Programmcode des Beispiels

Das Kalendersteuerelement erlaubt die Anzeige der Kalenderwochen über die Eigenschaft *ShowWeekNumbers*. Die betreffende Anweisung

```
Me.MonthCalendar1.ShowWeekNumbers = _
    Not Me.MonthCalendar1.ShowWeekNumbers()
```

wird in der *Click*-Ereignisbehandlungsprozedur der Schaltfläche *Woche* hinterlegt. Durch den *Not*-Operator wird einfach der Wert bei jedem Ereignis zwischen *False* und *True* umgeschaltet. Die Anzeige der aktuellen Werte erfolgt in der *Click*-Ereignisbehandlungsprozedur der Schaltfläche *Show*. Die Ereignisbehandlungsprozedur enthält folgenden Code:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    ' Zeige Werte
    Dim txt As String = ""
    txt = "Datum: " & Me.DateTimePicker1.Text & vbCrLf
    txt = txt & "Zeit: " & Me.DateTimePicker2.Text & vbCrLf
    txt = txt & "Kalenderdatum: " & Me.MonthCalendar1.TodayDate
    MessageBox.Show(txt, "Ergebnisse", _
        MessageBoxButtons.OK, MessageBoxIcon.Information)
End Sub
```

Weitere Hinweise zu den Eigenschaften der betreffenden Steuerelemente und Details zu den verfügbaren Methoden entnehmen Sie bitte der Hilfe des .NET-Framework.

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung entwickelten Beispiels im Ordner `\Beisp\Kap10\ImgView` auf der Begleit-CD. Die vom Beispiel angezeigten Bilddateien finden sich im Unterordner `Images` und im Unterordner `\bin\Images`. In der `Load`-Prozedur des Formulars finden Sie auskommentierte Anweisungen, in denen die Dateinamen direkt in einem Feld hinterlegt wurden und die demonstrieren, wie sich die Dateien zur Laufzeit nachträglich laden ließen.

10.6 Listen- und Strukturansichten nutzen

Der Windows-Explorer und viele andere Windows-Anwendungen bieten sie: Steuerelemente, die Listenansichten (*ListView*) und Strukturansichten (*TreeView*) bereitstellen. Zudem lassen sich nebeneinander angeordnete Fenster über Teilerfelder (*Splitter*) in der Größe anpassen. Diese Steuerelemente und Teilefelder lassen sich auch in .NET Framework in Formularen nutzen. Das folgende Beispiel zeigt, wie solche Steuerelemente in ein Dialogfeld einzubinden sind.

10.6.1 Vorbereiten des Formulars mit den Steuerelementen

Bei diesem Beispiel steht zunächst der Entwurf des Formulars im Vordergrund, da die Positionierung eines Teilerfelds einige Vorüberlegungen erfordert. Das Dialogfeld soll als einfaches Formular gestaltet werden, wobei das *TreeView*-Steuerelement ca. ein Drittel der linken Hälfte und das *ListView*-Steuerelement den restlichen rechten Teil des Formulars nutzen soll. Die genauen Breiten dieser Felder brauchen Sie nur grob vorzugeben, da der Benutzer die Aufteilung ja später per Maus anpassen kann. Das Dialogfeld soll aber oberhalb der Anzeigeelemente noch freien Platz zur Aufnahme von Schaltflächen aufweisen.

Beim Entwurf eines Formulars, welches ein Teilerfeld enthalten soll, müssen Sie mit der *Dock*-Eigenschaft arbeiten. Dies bedeutet, dass sich ein Steuerelement über diese Eigenschaft am Rand eines Formulars (bzw. des umgebenden Containers) verankern lässt. Dabei treten aber Nebeneffekte auf, die Sie beim Entwurf des Formulars durch geschickte Gestaltung vermeiden lassen. In Abbildung 10.15 sehen Sie mehrere Varianten eines Formulars mit den gleichen Steuerelementen (*TreeView*, *Splitter* und *ListView*). Das oben links gezeigte Formular besitzt jedoch deutliche Ränder (was gewollt war), während beim Formular rechts oben die Steuerelemente den kompletten Fensterplatz beanspruchen (was so nicht gewollt war, aber durch die Verankerung ausgelöst wird).

Möchten Sie die in Abbildung 10.15 rechts oben gezeigte Variante benutzen, brauchen Sie keine besonderen Maßnahmen beim Entwurf zu beachten (einfach die Steuerelemente in das Formulardesign ziehen und deren Eigenschaften setzen). Um die in Abbildung 10.15 links oben gezeigte Anordnung mit Rand zu erzwingen, sind die drei Steuerelemente *TreeView*, *Splitter* und *ListView* in einem zusätzlichen *Panel*-Steuerelement unterzubringen.

Hinweis

Ein *Panel*-Steuerelement stellt eine Art unsichtbarer Container zur Aufnahme weiterer Steuerelemente dar. Die im *Panel* enthaltenen Steuerelemente beziehen sich in der *Dock*-Eigenschaft auf das Panel und nicht auf das Formular. Auch andere Eigenschaften wie *Visible* etc. des *Panel*-Elements wirken auf die enthaltenen Steuerelemente.

Das neue SplitContainer-Element

Mit dem .NET Framework 2.0 wurde auch ein neues *SplitContainer*-Steuerelement eingefügt, welches das Erzeugen von Formularen mit geteilten Bereichen stark vereinfacht. Sobald Sie das Steuerelement aus der Toolbox in das Formular einfügen, wird ein Platzhalter angezeigt, der bereits den Splitter enthält. Sie können dann in die beiden Teile des *SplitContainers* die gewünschten Steuerelemente wie *TreeView* oder *ListView* einfügen und müssen sich nicht mehr mit einem *Splitter*-Steuerelement herumschlagen.

Hinweis

Im Projektordner `\Beisp\Kap10\TreeListView_Design` finden Sie ein Projektbeispiel, welches die drei Formularvarianten *Variante1*, *Variante2* und *Variante3* enthält. *Variante3* wurde mit einem *SplitContainer* realisiert.

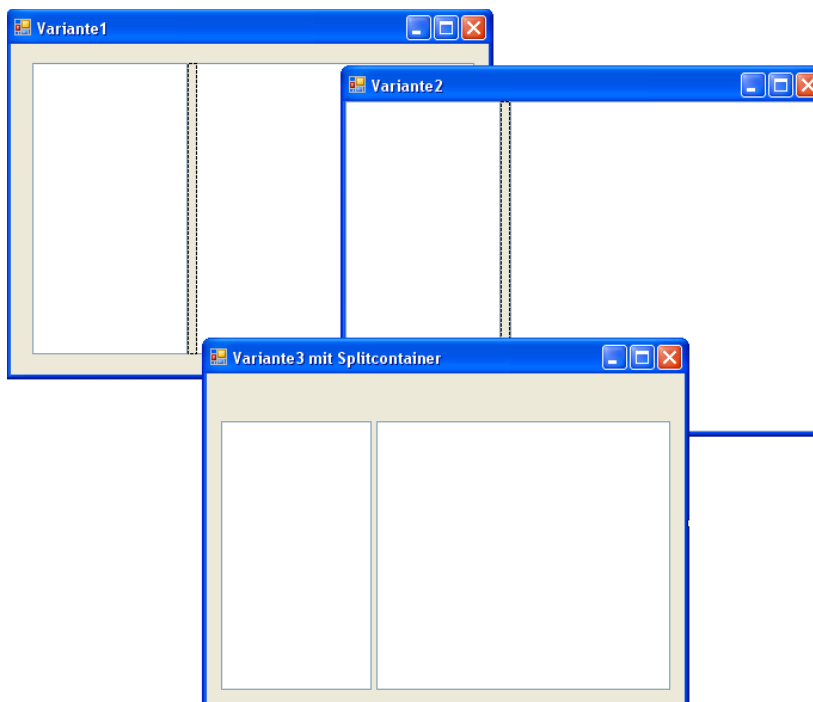


Abbildung 10.15: Unterschiedliche Formularvarianten

Schritte zum Formularentwurf

Für das erste Beispiel soll jetzt ein Dialogfeld gemäß dem in Abbildung 10.15 (rechts oben) gezeigten Layout realisiert werden:

1. Legen Sie in der Entwicklungsumgebung ein neues Projekt als Windows-Anwendung an, setzen Sie dann die Projekteigenschaften (z.B. Symbol für die Programmdatei) und wechseln zum Formularentwurf. Passen Sie die Formulareigenschaften (Titeltext, Größe) den Anforderungen an.
2. Fügen Sie über die Toolbox ein *Panel*-Steuerelement zum Formularentwurf hinzu. Das Steuerelement wird als gestreifter Rahmen im Formularlayout dargestellt. Passen Sie die Größe dieses Steuerelements so an, dass ein Rand im Formular übrig bleibt (Abbildung 10.16). Am oberen Rand belassen Sie etwas mehr Platz, da dort noch Schaltflächen untergebracht werden sollen.

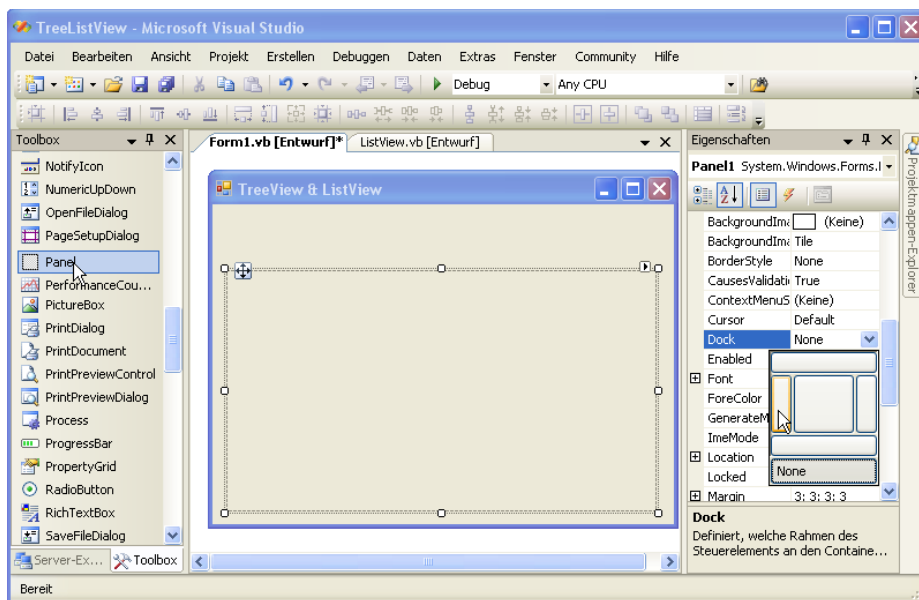


Abbildung 10.16: Designfenster und Eigenschaften

3. Fügen Sie aus der Toolbox ein *TreeView*-Steuerelement im linken Teil des *Panel* zum Formularentwurf hinzu. Die Größe des Steuerelements sollte in etwa ein Drittel der Panelbreite umfassen. Wählen Sie bei markiertem *TreeView*-Steuerelement im Eigenschaftenfenster die Eigenschaft *Dock* und setzen Sie diese auf *Left* (einfach gemäß Abbildung 10.16 auf das Feld am linken Rand klicken).
4. Ergänzen Sie im Design den rechten Teil des *Panel*-Steuerelements um ein *ListView*-Steuerelement aus der Toolbox. Die Größe des Steuerelements sollte in etwa zwei Drittel der Panelbreite umfassen (belassen Sie einen schmalen Steg zwischen den beiden Steuerelementen).

5. Fügen Sie jetzt noch ein *Splitter*-Steuerelement zum *Panel* hinzu. Das Element sollte horizontal die Größe des Teilers zwischen den beiden Steuerelementen *TreeView* und *ListView* umfassen und ca. 1 Rasterfeld breit sein. Die *Dock*-Eigenschaft dieses Elements wird automatisch auf *Left* gesetzt, d.h., der *Splitter* wird am linken Rand des *Panel* verankert.
6. Markieren Sie jetzt erneut das *ListView*-Steuerelement und setzen Sie im Eigenschaftfenster die Eigenschaft *Dock* auf *Fill* (einfach gemäß Abbildung 10.16 auf das mittlere Feld klicken).

Die Eigenschaft *Dock* legt fest, wie das Formular die Steuerelemente beim Arbeiten mit dem Teilungsfeld (Splitter) darstellen soll. Wird der Wert auf *None* gesetzt, passiert nichts. Mit dem Wert *Left* passt das Formular das Steuerelement immer am linken Rand des betreffenden Containers (Formular, Panel, Registerkarte etc.) an. Das Gleiche gilt sinngemäß für *Right*, *Top* und *Bottom*. Der Wert *Fill* der *Dock*-Eigenschaft des *ListView*-Steuerelements bewirkt, dass das betreffende Steuerelement zur Laufzeit immer den verbleibenden Rest des Containers einnimmt. Dies ist wichtig, da der Benutzer ja beim Verschieben des Teilungsfelds die Größe des zweiten beteiligten Steuerelements verändert. In diesem Beispiels sind das *TreeView*- und das *Splitter*-Steuerelement am linken Rand des *Panel* verankert. Folglich wirkt das Teilungsfeld auf das *TreeView*-Element und das *ListView*-Element muss automatisch die Größe des verbleibenden Raums einnehmen.

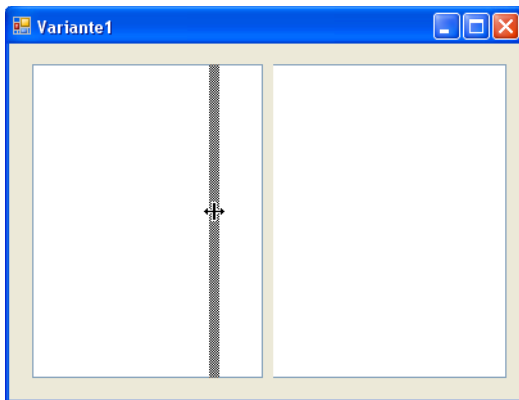


Abbildung 10.17: Fenster mit den leeren *TreeView*- und *ListView*-Elementen

Hinweis

Wenn Sie zu diesem Zeitpunkt das Projekt übersetzen und ausführen, sollte ein Fenster gemäß Abbildung 10.17 angezeigt werden. Sie können bereits das Teilerfeld zum Anpassen der Fensterbreiten nutzen. Sie finden das Projektbeispiel mit den verschiedenen Formularvarianten im Ordner `\Beisp\Kap10\TreeListView_Design` der Begleit-CD. In Abbildung 10.17 wird die Formularvariante 1 gezeigt. Bei Bedarf können Sie die beiden anderen Varianten *Variante2* und *Variante3* testen, indem Sie diese Formulare als Startobjekt in den Projekteigenschaften vorgeben und das Projekt neu übersetzen.

Dock- und Anchor-Eigenschaften für Formulargrößenanpassung

Wenn der Benutzer die Formulargröße verändert, sollten die Steuerelemente entweder ihre Position zum Formularrand beibehalten oder an die Formulargröße angepasst werden. Verwenden Sie ein *SplitContainer*-Steuerelement, könnten Sie die *Dock*-Eigenschaft auf *Fill* setzen. Dann wird der Container automatisch an die Formulargröße angepasst. Problematisch wird dies aber, wenn Sie neben dem *SplitContainer* noch weitere Steuerelemente wie Schaltflächen etc. im Formular unterbringen möchten. Der *SplitContainer* wird automatisch auf die Formulargröße erweitert, die zusätzlichen Steuerelemente liegen ebenfalls im *SplitContainer*.

Eine Lösung besteht darin, im Formularbereich ein zusätzliches *Panel*-Steuerelement zur Aufnahme der zusätzlichen Steuerelemente (Schaltflächen etc.) zu hinterlegen und die *Dock*-Eigenschaft des *Panel*-Elements an einem der Formularränder (z.B. oben) zu verankern. Fügen Sie einen *SplitContainer* zum Formularbereich hinzu und setzen Sie dessen *Dock*-Eigenschaft auf *Fill*. Dann passt das Laufzeitsystem die Containergröße für *Panel* und *SplitContainer* jeweils an den Formularabmessungen an, sorgt aber dafür, dass sich die beiden Container nicht überlappen.

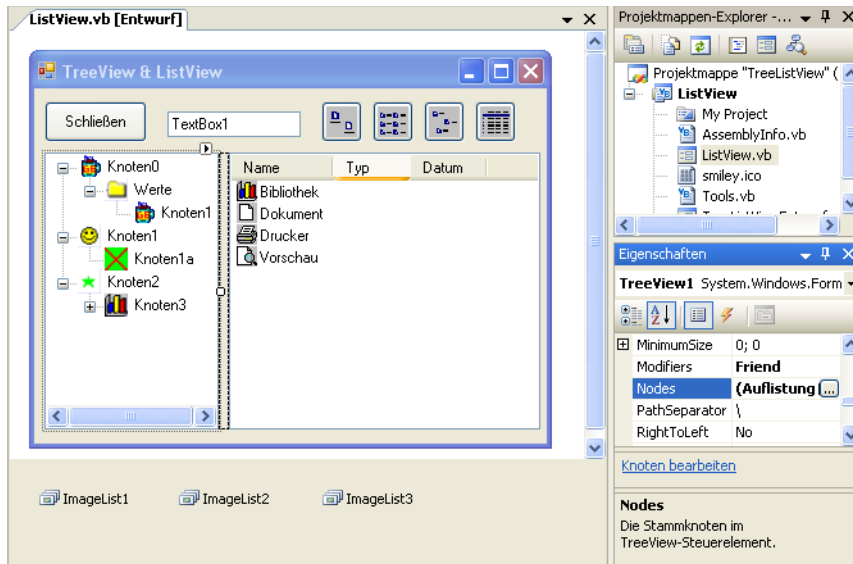
Der smartere Weg, Größenänderungen im Formular in einem *SplitContainer* (oder einem *Panel*) zu berücksichtigen, besteht in der Verwendung der bereits auf den vorhergehenden Seiten mehrfach erwähnten *Anchor*-Eigenschaft. Fügen Sie den *SplitContainer* (oder das *Panel*) im Formular ein. Anschließend wählen Sie im Eigenschaftenfenster des Steuerelements die *Anchor*-Eigenschaft. Über diese Eigenschaft lässt sich das Steuerelement selektiv bezüglich der Position des übergeordneten Containers verankern. Eine Verankerung an einem Formular bewirkt z.B., dass das Laufzeitsystem die Größe des Steuerelements bei Änderungen der Formularabmessungen mit anpasst.

Hinweis

Diese Verankerung über die *Anchor*-Eigenschaft lässt sich auf alle Steuerelemente innerhalb des Formulars anwenden. Dies stellt sicher, dass die Position des Steuerelements im Formular auch bei Größenänderungen erhalten bleibt. Das Formular *SplitContainer_Panel* des im Ordner `\Beisp\Kap10\TreeView_Design` der Begleit-CD hinterlegten Projektbeispiels zeigt, wie ein *SplitContainer_Panel* in einem Formular so hinterlegt wird, dass noch Platz für weitere Steuerelemente bleibt. Binden Sie dieses Formular als Startobjekt in den Projekteigenschaften ein, können Sie das Verhalten des Formulars bei Größenänderungen etc. studieren. Standardmäßig ist das Startobjekt auf *Form1* gesetzt, um die Klasse *TreeViewManual.vb* einzubinden. Diese Klasse enthält den Code, um das Formular direkt zur Laufzeit aufzubauen.

10.6.2 Gestalten des TreeView-Steuerelements

Das *TreeView*-Steuerelement eignet sich zur Darstellung einer Strukturansicht (eines Strukturbaums). Die Explorerleisten in Ordnerfenstern stellen ein Beispiel für eine Strukturansicht dar. Die Struktur (d.h. der Inhalt) des Steuerelements lässt sich statisch im Designer oder dynamisch zur Laufzeit festlegen. Nachdem das Steuerelement im Formulardesign eingefügt wurde, soll dieses jetzt mit einer einfachen statischen Struktur gefüllt werden (Abbildung 10.18, linkes Feld unter der Schaltfläche *Schließen*).

Abbildung 10.18: Formularlayout und Eigenschaften des *TreeView*-Steuerelements

Um die einzelnen Knoten des Strukturbaums mit Symbolen zu versehen, benötigen Sie ein *ImageList*-Steuerelement als Container. In diesem Container werden die in der Anwendung benötigten Bilddateien hinterlegt und von den Steuerelementen abgerufen:

1. Fügen Sie als Erstes ein *ImageList*-Steuerelement im Design ein und konfigurieren Sie dieses so, dass es eine Reihe von Symbolen enthält. Im Beispielpjekt (Abbildung 10.18) ist dies das *ImageList1*-Element am unteren Rand des Ansicht-Designers. Die Originalsymboldateien sind im Unterordner *\Images* des Projektordners *\Beisp\Kap10\TreeView* hinterlegt. Die Schritte zum Füllen eines *ImageList*-Steuerelements wurden im vorhergehenden Abschnitt »Entwurf der Registerkarte zur Bildanzeige« vorgestellt.
2. Zum Konfigurieren der *TreeView*-Inhalte markieren Sie das Steuerelement im Designer (Abbildung 10.18, linkes Feld unter der Schaltfläche *Schließen*) und wechseln zum Eigenschaftensfenster.
3. Wählen Sie über das Listenfeld der Eigenschaft *ImageList* das in den vorhergehenden Schritten vorbereitete *ImageList*-Steuerelement (der Name wird über ein Listenfeld zur Auswahl angeboten). Dies gewährleistet, dass das *TreeView*-Steuerelement auf die Bilddaten der *ImageList* zugreifen kann.
4. Setzen Sie in den Eigenschaften *ImageIndex* und *SelectedImageIndex* eines der Symbole aus dem *ImageList*-Steuerelement (die Auswahl erfolgt über die Listenfelder der betreffenden Eigenschaften).

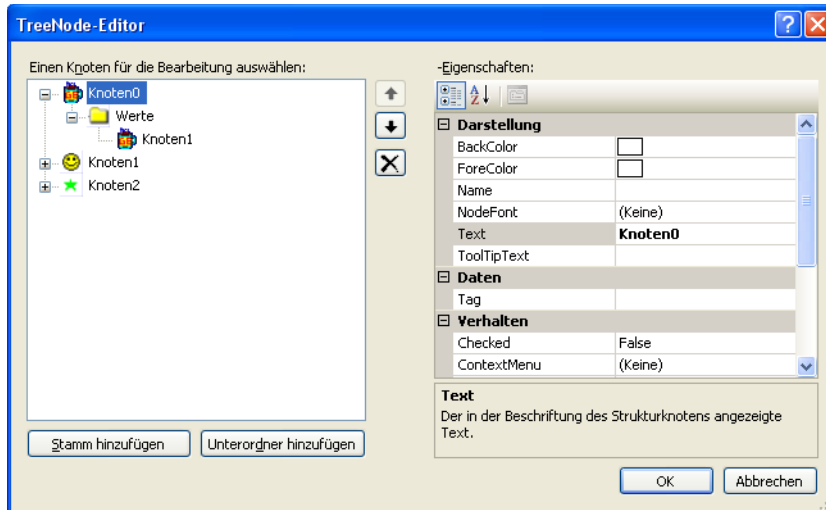


Abbildung 10.19: Konfigurierung der Struktur im TreeNode-Editor

5. Klicken Sie auf die Schaltfläche der Eigenschaft *Nodes*, um die Auflistung der Elemente (Baumstruktur mit Knoten) im TreeNode-Editor zu ergänzen.
6. Fügen Sie die gewünschten Elemente im TreeNode-Editor (Abbildung 10.19) hinzu und schließen Sie diesen über die OK-Schaltfläche.

Der TreeNode-Editor erlaubt eine sehr komfortable Konfiguration des Inhalts des *TreeView*-Steuerelements:

- Über die Schaltfläche *Stamm hinzufügen* lässt sich ein Startknoten in der Struktur hinzufügen.
- Die Schaltfläche *Unterordner hinzufügen* ergänzt den markierten Knoten oder Unterknoten um einen weiteren untergeordneten Knoten.
- Die rechts neben der Liste *Einen Knoten zur Bearbeitung auswählen* sichtbaren Schaltflächen erlauben einen gewählten Knoten in den Zweigen des Baums zu verschieben oder aus dem *TreeView* zu löschen.
- Die Bezeichnung eines markierten Knotens lässt sich über die Eigenschaft *Text* in der rechten Eigenschaftenspalte vereinbaren. Über die Eigenschaft *ToolTipText* lässt sich ein Text hinterlegen, der beim Zeigen auf den Knoten als ToolTip eingblendet wird.
- Wurde ein *ImageList*-Element in den Eigenschaften des *TreeView*-Steuerelements konfiguriert, können Sie zudem jedem Knoten zwei Symbole über die Eigenschaften *ImageIndex* und *SelectedImageIndex* zuweisen. *ImageIndex* legt das Bild für den nicht ausgewählten Knoten fest. Klickt der Benutzer auf einen Knoten, wird das in der Eigenschaft *SelectedImageIndex* gesetzte Symbol angezeigt.

Sie können beide Eigenschaften *ImageIndex* und *SelectedImageIndex* auf den gleichen Index setzen, falls sich das Symbol des Elements bei der Auswahl nicht ändern soll. Bei unterschiedlichen Symbolen in diesen Feldern können Sie bereits im *TreeNode*-Editor auf den Knoten im Strukturbaum klicken. Dann wird der Editor das zweite Symbol anzeigen. Klicken Sie auf einen anderen Knoten, erscheint für das vorher ausgewählte Element das erste definierte Symbol.

Hinweis

Sie finden ein entsprechend gestaltetes und in der Entwicklungsumgebung abrufbares Formularbeispiel im Ordner `\Beisp\Kap10\TreeView` der Begleit-CD. Der Ordner `\Beisp\Kap10\TreeViewSplitContainer` enthält das unter Verwendung eines *SplitContainer*-Steuerelements (an Stelle eines *Panel*-Elements mit *Split-Control*) realisierte Beispiel.

ImageList und TreeView per Programmcode erzeugen

Möchten Sie zur Laufzeit ein *TreeView* mit weiteren Knoten füllen? Auch dies ist mit wenigen Visual-Basic-Anweisungen möglich. Um ein *ImageList*-Steuerelement anzulegen und dann mit einigen Bildern zu füllen, sind folgende Anweisungen zu verwenden:

```
Const path As String = "F:\Beisp\Kap10\Icons\"
' Namen der Symboldateien für die ImageList
Dim file () As String = { _
    "Cross.bmp", "Smiley.bmp", "Star.bmp" }
Dim i As Integer
Friend WithEvents ImageList1 As System.Windows.Forms.ImageList
...
With Me
    .ImageList1 = New System.Windows.Forms.ImageList() ' neue Instanz
    With Me.ImageList1
        For i = 0 To file.Length-1 ' Fülle ImageList
            .Images.Add(Image.FromFile(path & file(i)))
        Next i
    End With
End With
```

Sobald die *ImageList* instantiiert und mit Symbolen gefüllt ist, kann diese an andere Steuerelemente angebunden werden. Die folgende Codesequenz instantiiert das *TreeView*-Steuerelement und weist einige Knoten zu:

```
Friend WithEvents TreeView1 As System.Windows.Forms.TreeView
...
Me.TreeView1 = New System.Windows.Forms.TreeView() ' neue Instanz

' TreeView1 - Eigenschaften für das Element festlegen
```

```
Me.TreeView1.Dock = System.Windows.Forms.DockStyle.Left ' andocken
Me.TreeView1.ImageList = Me.ImageList1 ' an ImageList binden
Me.TreeView1.ImageIndex = 0 ' 1. Bild (Standard)
Me.TreeView1.Name = "TreeView1"
Me.TreeView1.SelectedImageIndex = 2 ' 3. Bild (Standard)
' Initialisiere die Größe des Steuerelements
Me.TreeView1.Size = New System.Drawing.Size(120, 200)
Me.TreeView1.TabIndex = 0

' Knoten hinzufügen - TreeNode (Name, Bild1, Bild2)
Me.TreeView1.Nodes.AddRange( _
    New System.Windows.Forms.TreeNode() { _
        New System.Windows.Forms.TreeNode("Knoten1a", 1, 1, _
        New System.Windows.Forms.TreeNode() { _
            New System.Windows.Forms.TreeNode("Knoten2", 2, 1)}), _
        New System.Windows.Forms.TreeNode("Drucker")})

' hier noch einen Einzelknoten anhängen
Dim knoten As TreeNode = New TreeNode("Test", 2, 1)
Me.TreeView1.Nodes.Add (knoten)
```

Das Hinzufügen eines Knotens erfolgt entweder über *AddRange* oder über die *Add*-Methode. Beide Methoden erwarten *TreeNode*-Objekte, wobei *AddRange* mehrere Objekte enthalten kann. Zudem lassen sich die Objekte mit Unterknoten schachteln. Knoten lassen sich mittels der *Remove*-Methode wieder entfernen. Alle Knoten lassen sich mittels der *Clear*-Methode entfernen.

Hinweis

Details zu den Methoden finden Sie in der Hilfe des .NET Framework. Das im Ordner *\Beisp\Kap10\TreeListView_Design* hinterlegte Projekt enthält das Element *TreeListViewManual.vb*. Dieses Element enthält den Code, um das Formular mit einem *TreeView*- und dem nachfolgend beschriebenen *ListView*-Steuerelement zu versehen. Zudem kommen in diesem Beispiel die nachfolgend erwähnten Methoden *Clear* und *Remove* zum Entfernen von Knoten zum Einsatz.

Auswerten des markierten Knotens

Selektiert der Benutzer einen Knoten im *TreeView*-Element, löst dies ein *AfterSelect*-Ereignis aus. Innerhalb der Ereignisbehandlungsroutine lässt sich dann der selektierte Knoten über die *SelectedNode*-Eigenschaft ermitteln. Die Eigenschaft *Text* dieses Knotens liefert dessen Name. Der folgende Code zeigt diesen Knotennamen in einem Dialogfeld:

```
Private Sub TreeView1_AfterSelect(ByVal sender As System.Object, _
    ByVal e As System.Windows.Forms.TreeViewEventArgs) _
```



```
Handles TreeView1.AfterSelect
MsgBox ("Auswahl :" & Me.TreeView1.SelectedNode.Text)
End Sub
```

Das Meldungsfeld erscheint aber auch beim Start der Anwendung, da dann der Knoten der Wurzel selektiert ist.

Hinweis

Die Anweisungen zur Ereignisbehandlung finden Sie im Codefenster des Elements *Listview.vb*, wenn Sie das im Ordner `\Beisp\Kap10\TreeListView` hinterlegte Projekt in der Entwicklungsumgebung öffnen. Das Beispiel enthält Code, um den selektierten Knoten in einem Textfeld des Formulars einzublenden. Das im Ordner `\Beisp\Kap10\TreeListViewSplitContainer` gespeicherte Projekt besitzt die gleiche Funktionalität, verwendet aber einen *SplitContainer* zur Aufnahme des *TreeView*-Steuerelements.

Entfernen von Knoten aus dem TreeView

Zur Laufzeit lassen sich einzelne Knoten über die *Remove*-Methode der *TreeView.Nodes*-Auflistung löschen. Um alle Knoten des *TreeView*-Elements aus der *Nodes*-Auflistung zu entfernen, ist dagegen die *Clear*-Methode zu verwenden.

```
' Aktuell selektierter Knoten löschen
Me.TreeView1.Nodes.Remove(Me.TreeView1.SelectedNode)
Me.TreeView1.Nodes.Clear() ' Alle Knoten löschen
```

Sie finden ein Beispiel, welches diese beiden Methoden in der Ereignisbehandlungsroutine der *Löschen*-Schaltfläche nutzt, im Projekt, welches im Ordner `\Beisp\Kap10\TreeListViewDel` hinterlegt ist.

10.6.3 Entwurf des ListView-Steuerelements

Das *ListView*-Steuerelement bietet die Möglichkeit, Symbole und Text in verschiedenen Anordnungen (Große Symbole, Kleine Symbole, Liste, Details) darzustellen. Diese Darstellung kennen Sie aus dem Windows-Explorer. Auf den vorherigen Seiten haben Sie ein solches Steuerelement im Formular eingefügt. Nun gilt es diese Steuerelement im Designer zu konfigurieren. Dabei werden die Spalten der Detaildarstellung sowie die anzuzeigenden Einträge (Text und Symbol) festgelegt:

1. Das Steuerelement unterstützt die Anzeige von Symbolen in einer großen und einer kleinen Darstellung. Fügen Sie deshalb zwei *ImageList*-Steuerelemente zum Formularentwurf hinzu. Im ersten *ImageList*-Steuerelement fügen Sie Symbole in großer Darstellung (z.B. 32x32 oder 48x48 Pixel) hinzu. In der zweiten *ImageList* kommen die gleichen Symbole in kleiner Darstellung (z.B. 16x16) hinzu.
2. Markieren Sie anschließend das *ListView*-Steuerelement im Ansicht-Designer der Entwicklungsumgebung und wechseln Sie zum Eigenschaftfenster. Stellen Sie die Eigenschaft *View* auf die gewünschte Größe (z.B. *List* oder *Details*). Über die Eigen-

schaft *Checkboxes* lässt sich festlegen, ob Kontrollkästchen vor den Einträgen angezeigt werden (Wert auf *True* setzen). Die Eigenschaft *LabelEdit=True* erlaubt dem Benutzer zur Laufzeit die Symboltitel anzupassen. Mit der Eigenschaft *LabelWrap* erlauben Sie einen Textumbruch bei der Anzeige großer Symbole.

3. Wählen Sie die Eigenschaft *LargeImageList* und weisen Sie dieser den Namen der *ImageList* mit den großen Symbolen zu. Bei Bedarf können Sie die Eigenschaft *ImageSize* dieser Symbole einstellen.
4. Wiederholen Sie den letzten Schritt für die Eigenschaft *SmallImageList*, weisen dieser aber den Namen der *ImageList* mit den kleinen Symbolen zu (und passen Sie ggf. die Eigenschaft *ImageSize* dieser Symbole an).

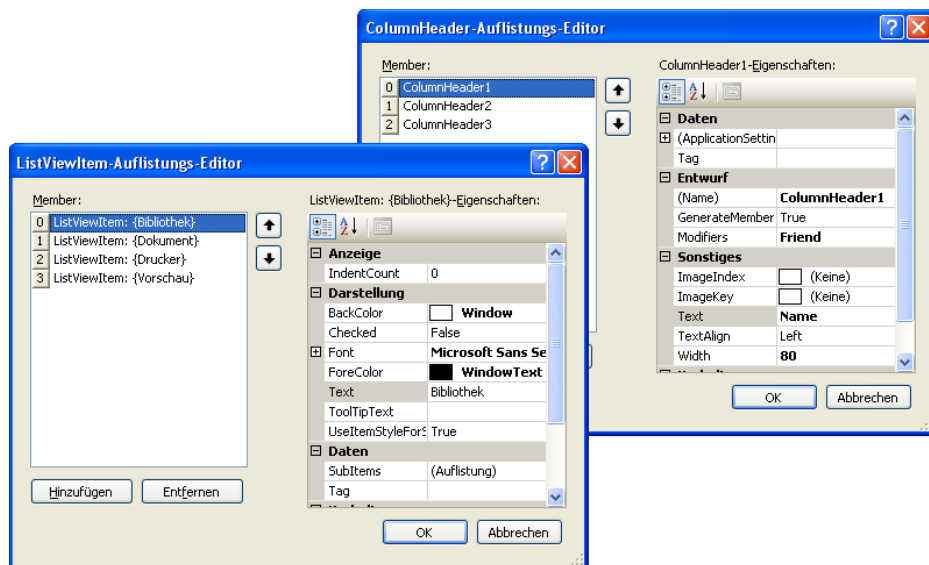


Abbildung 10.20: Editoren zum Einfügen der Spalten und der Einträge im ListView-Element

5. Klicken Sie auf die Schaltfläche der Eigenschaft *Columns* des *ListView*-Elements und geben Sie im dann erscheinenden Dialogfeld *ColumnHeader-Auflistungs-Editor* (Abbildung 10.20, rechts oben) die Eigenschaften für die Spaltenköpfe der *Details*-Darstellung ein. Die einzelnen Spalten lassen sich über die Liste *Member* anwählen, die Spaltentitel werden über die Eigenschaft *Text* definiert. Die Spaltenbreite ist im Feld *Width* einzutragen. Soll im Spaltentitel ein Symbol eingeblendet werden, können Sie das Symbol über die Eigenschaft *ImageIndex* aus einem *ImageList*-Element übernehmen. Nachdem die Eigenschaften definiert wurden, schließen Sie das Dialogfeld über die *OK*-Schaltfläche.
6. Klicken Sie im Eigenschaftenfenster des *ListView*-Steuerelements auf die Schaltfläche der Eigenschaft *Items*. Tragen Sie im Dialogfeld *ListViewItems-Auflistungs-Editor* die Elemente ein, die im Steuerelement anzuzeigen sind (Abbildung 10.20, links unten). Für jeden Eintrag lässt sich der Titel (Eigenschaft *Text*) und ein Symbol (Eigenschaft *ImageIndex*) vorgeben. Über die Schaltflächen *Hinzufügen* und *Entfernen* lassen sich die Einträge der Auflistung pflegen. Schließen Sie das Dialogfeld über die *OK*-Schaltfläche.

Zudem können Sie anschließend im Formularentwurf noch vier Schaltflächen (außerhalb des *Panel*-Elements) hinzufügen, mit denen sich später die Darstellung zwischen Große Symbole, Kleine Symbole, Liste und Details umschalten lässt (Abbildung 10.21).

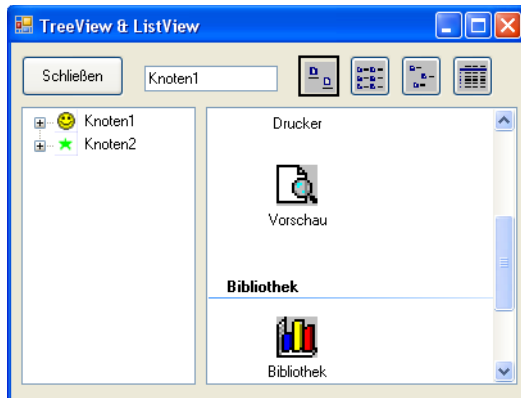


Abbildung 10.21: Beispiel mit ListView-Element

Tipp

Das Formular in diesem Beispiel besitzt eine Schaltfläche zum Maximieren des Fensters. Möchten Sie, dass das *TreeView*- und *ListView*-Steuerelement automatisch an die Fenstergröße angepasst werden? Entweder Sie verankern das *Panel*-Steuerelement an den vier Seiten des Formulars (siehe auch vorherige Seiten). Oder Sie arbeiten mit verschachtelten Containern, die Sie über die *Dock*-Eigenschaft auf *Fill* setzen.

Sollen die Elemente des *ListView*-Steuerelements gruppiert werden? Dann wählen Sie die Schaltfläche der Eigenschaft *Groups* des *ListView*-Elements. Im Dialogfeld *List-ViewGroup-Auflistungs-Editor* können Sie über die *Hinzufügen*-Schaltfläche einen oder mehrere Gruppeneinträge definieren. Die Gruppennamen werden über die Eigenschaft *Header* festgelegt. Anschließend müssen Sie über die Schaltfläche der *Items*-Eigenschaft des *ListView*-Elements das Dialogfeld *ListViewItem-Auflistungs-Editor* öffnen. Wählen Sie einen Eintrag in der Liste *Member* an und ordnen Sie diesen über die Eigenschaft *Group* (in der Kategorie *Verhalten*) einer der definierten Gruppen zu. Zur Laufzeit wird das *ListView*-Steuerelement den betreffenden Eintrag in der Gruppe darstellen.

Wenn Sie den Formularentwurf nun übersetzen und ausführen, sollte bereits ein Dialogfeld mit dem *TreeView*-Block und dem *ListView*-Bereich erscheinen. Der *ListView*-Bereich weist die von Ihnen definierten Elemente auf. In Abbildung 10.21 sehen Sie ein von mir gestaltetes Formular mit einem *ListView*-Element, welches große Symbole anzeigt. Dabei wird auch eine Gruppierung der Symbole vorgenommen.

Neue Einträge hinzufügen

Sie können mittels der *Add*-Methode der *Items*-Auflistung neue Elemente zum Steuerelement hinzufügen. Um einen neuen Eintrag in die Liste aufzunehmen, ist die folgende Anweisung anzuwenden:

```
Me.ListView1.Items.Add("Test", 2)
```

Der erste Parameter definiert den anzuzeigenden Symboltitel, der zweite Parameter definiert den Bildindex in die *ImageList*-Steuerelemente, aus denen die Symbole bezogen werden.



Abbildung 10.22: Spaltenanzeige im ListView-Element

Bei dem im Designer realisierten obigen Beispiel gibt es noch ein Problem: In der Darstellung *Details* wird nur die erste Spalte mit dem Text und dem Symbol gefüllt. Die restlichen Spalten müssen mit Unterelementen (Subitems) gefüllt werden. Hierzu wird die *Add*-Methode der *SubItems*-Auflistung benutzt. Die folgende Schleife ergänzt alle Einträge im Steuerelement um zwei weitere Spalten:

```
Dim i As Integer
' Hinzufügen von Daten zum ListView-Element
With Me ' Add-Methode: 1. Wert = Text, 2. Wert = Bildindex
' Ergänze jetzt die Untereinträge für die Details-Spalten
For i = 0 To .ListView1.Items.Count - 1
    ListView1.Items(i).SubItems.Add("Nr. " & i.ToString)
    ListView1.Items(i).SubItems.Add(Now().ToString)
Next
End With
```

Der erste Eintrag erhält eine laufende Nummer zugewiesen, der zweite Eintrag enthält das Datum (Abbildung 10.22).

Umschalten des Darstellungsmodus und Gruppendarstellung unterdrücken

Der Darstellungsmodus lässt sich zur Laufzeit über die *View*-Eigenschaft umschalten. Die folgende Anweisung erzwingt die Darstellung großer Symbole:

```
Me.ListView1.View = View.LargeIcon
```

Die Eigenschaft kann dabei auf eine der in der Klasse *View* vereinbarte Konstante (*View.LargeIcon*, *View.SmallIcon*, *View.List*, *View.Details*) gesetzt werden. Das Steuerelement übernimmt selbst die korrekte Darstellung der Symbole. Voraussetzung ist lediglich, dass Sie die korrekten *ImageList*-Daten zugeordnet haben.

Haben Sie Gruppen im *ListView*-Steuerelement definiert, werden diese in Abhängigkeit vom Darstellungsmodus eingeblendet. Die folgende Anweisung unterdrückt aber die Anzeige der betreffenden Gruppen:

```
Me.ListView1.ShowGroups = False
```

Weisen Sie der Eigenschaft den Wert *True* zu, werden definierte Gruppen wieder angezeigt.

Wie wird ein Mausklick auf den Spaltenkopf festgestellt?

Im Darstellungsmodus *Details* enthält die Listendarstellung mehrere Spaltenköpfe. Klicken Sie in einem Windows-Ordnerfenster auf einen der Spaltenköpfe, wird die Liste nach dem betreffenden Kriterium sortiert. Um auf das Anklicken eines Spaltenkopfs zu reagieren, müssen Sie eine *ColumnClick*-Ereignisbehandlungsprozedur für dieses Objekt einrichten. Die nachfolgenden Zeilen zeigen, wie sich dies realisieren lässt:

```
Private Sub ListView1_Spalte1(ByVal sender As Object, _
    ByVal e As ColumnClickEventArgs) Handles ListView1.ColumnClick
    ' Anklicken der Spalte - Sortierung umschalten
    With Me
        If e.Column = 0 Then ' Erster Spaltenkopf in Detailansicht
            If .ListView1.Sorting = SortOrder.Ascending Then
                .ListView1.Sorting = SortOrder.Descending
            Else
                .ListView1.Sorting = SortOrder.Ascending
            End If
        End If
    End With
End Sub
```

Beim Aufruf der Prozedur übergibt das System das Objekt *e* der angeklickten Spalte als Parameter. Die Eigenschaft *Column* des Objekts *e* gibt die Spaltennummer an, die angeklickt wurde. Bei einem Klick auf die erste Spalte wird die Sortierung in obiger Prozedur wahlweise auf- oder absteigend vorgenommen. Dies lässt sich über die Eigenschaft *Sorting* einstellen.

Auf eine Auswahl im ListView reagieren

Wählt der Benutzer einen Eintrag in der Listendarstellung, lässt sich dies abfragen. Das *ListView*-Steuerelement löst ein *SelectedIndexChanged*-Ereignis aus. Der folgende Code zeigt ein Beispiel für eine solche Ereignisbehandlungsprozedur:

```
Private Sub ListView1_SelectedIndexChanged( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles ListView1.SelectedIndexChanged
    Dim txt As String = ""
    Dim i As Integer
    For i = 0 To Me.ListView1.SelectedItems.Count - 1
        txt = txt & Me.ListView1.SelectedItems(i).Text & " "
    Next
    Me.Label1.Text = txt
End Sub
```

Die markierten Elemente finden sich in der *SelectedItems*-Auflistung des Steuerelements. Da es mehrere gewählte Elemente geben kann, habe ich die *Count*-Eigenschaft dieser Auflistung abgefragt und verwende eine Schleife zur Bearbeitung aller markierten Elemente. Hier werden die Elemente in einem Label im Formular angezeigt.

Hinweis

Sie finden mehrere Projektbeispiele im Ordner *\Beisp\Kap10* der Begleit-CD. Das Element *ListView.vb* des im Unterordner *TreeListView* gespeicherten Projektbeispiels demonstriert, wie sich ein *TreeView*- und ein *ListView*-Element einsetzen lassen. Das Projektbeispiel im Unterordner *TreeListViewSplitContainer* zeigt, wie ein *SplitContainer*-Element zur Aufnahme der *TreeView*- und *ListView*-Elemente benutzt werden kann. Das im Unterordner *TreeListViewDel* hinterlegte Projekt enthält zusätzlich Code, um Knoten aus dem *TreeView*-Element zu löschen (siehe Erläuterungen auf den vorhergehenden Seiten). Und das Modul *TreeListViewManual.vb* des im Unterordner *TreeListView_Design* gespeicherten Projekts demonstriert, wie sich *TreeView*- und *ListView*-Elemente zur Laufzeit zu einem Formular hinzufügen lassen.

Wenn Sie im Projekt ein neues Formular hinzufügen, können Sie die Vorlage *Explorer-Fenster* im Dialogfeld *Neues Element hinzufügen* wählen. Diese Vorlage erstellt ein Fenster, welches neben einem *TreeView*- und *ListView*-Steuerelement zusätzlich eine Menü- und eine Symbolleiste aufweist. Sie brauchen dann nur noch die gewünschte Funktionalität zum betreffenden Formularelement hinzuzufügen.

10.6.4 Einsatz des Webbrowser-Steuerelements

Im .NET 2.0-Framework ist auch ein Webbrowser-Steuerelement enthalten, welches die Grundfunktionalität des Internet Explorer zum Abrufen und Anzeigen von Webseiten bereitstellt. Der Einsatz des Steuerelements soll jetzt an einem einfachen Beispiel demonstriert werden. Ein Formular enthält ein Textfeld zur Eingabe der Web-Adresse

der abzurufenden Seite, eine Schaltfläche zum Abrufen der gewünschten Seite sowie das Webbrowser-Steuerelement (Abbildung 10.23). Zusätzlich gibt es noch mehrere Schaltflächen, um zwischen besuchten Seiten vor und zurück zu blättern, um das Laden einer Seite abubrechen und um die vom Benutzer im Internet Explorer eingestellte Homepage abzurufen. Zum Realisieren des Beispiels gehen Sie in folgenden Schritten vor:

1. Erzeugen Sie ein neues Windows-Projekt in der Entwicklungsumgebung und fügen Sie die in Abbildung 10.23 sichtbaren Steuerelemente (*Label*, *TextBox*, *Button* und *Webbrowser*) zum Formular hinzu.
2. Passen Sie die Eigenschaften der Steuerelemente gemäß Ihren Wünschen an und ergänzen Sie anschließend den Code zur Behandlung des *Click*-Ereignisses für die Schaltfläche.

Für das *Label*- und die *Button*-Steuerelemente setzen Sie z.B. die *Text*-Eigenschaft auf die anzuzeigenden Texte. Das *Label*-Steuerelement verankern Sie über die *Anchor*-Eigenschaft am linken und oberen Rand, während das *Button*-Steuerelement der Schaltfläche *Wechseln zu* am oberen und rechten Rand verankert wird. Das Textfeld für die Adresseingabe können Sie am linken, oberen und rechten Rand verankern. Beim *Webbrowser*-Steuerelement ist die *Anchor*-Eigenschaft so zu setzen, dass das Element am linken, unteren und rechten Rand verankert wird. Die Schaltflächen unterhalb des Textfelds können Sie am linken und oberen Rand verankern. Dies bewirkt, dass das *WebBrowser*-Element sowie das Textfeld für die Adresseingabe bei Änderungen der Formulargröße mit angepasst werden, während das Label und die Schaltflächen die ursprüngliche Größe beibehalten.

Klickt der Benutzer auf die Schaltfläche *Wechseln zu*, soll die angegebene Seite im *Webbrowser*-Steuerelement angezeigt werden. Hierzu stellt die Komponente die *Navigate*-Methode bereit, die eine als Argument übergebene Adresse erwartet. Die Methode versucht die betreffende Seite abzurufen. Dabei ist es egal, ob als Adresse eine URL zu einer Webseite (z.B. *www.microsoft.com*) oder ein Verweis auf ein lokales Laufwerk (z.B. *C:*) übergeben wird. Bei Internetadressen ist lediglich Voraussetzung, dass eine Verbindung zum Internet existiert. Der Code zur Behandlung des *Click*-Ereignisses wird dadurch auf eine Anweisung reduziert.

```
Me.WebBrowser1.Navigate(Me.TextBox1.Text)
```

Der *Navigate*-Methode wird hier der Wert der *Text*-Eigenschaft des *TextBox*-Steuerelements übergeben. Ist dort eine gültige Adresse hinterlegt, kann das Steuerelement die Seite abrufen und anzeigen. Die Navigationsschaltflächen im Formular bedienen sich verschiedener Methoden, die durch das *WebBrowser*-Steuerelement bereitgestellt werden.

- *GoBack()*: Diese Methode veranlasst den Browser, zur vorhergehenden Seite (sofern möglich) zurückzukehren. Die Eigenschaft *CanGoBack* zeigt an, ob eine Vorgängerseite existiert.
- *GoForward()*: Bewirkt, dass der Browser (sofern möglich) zur Folgeseite blättert. Die Eigenschaft *CanGoForward* zeigt an, ob eine Folgeseite existiert.
- *GoHome()*: Diese Methode ruft die vom Benutzer im Internet Explorer eingestellte Homepage in der Anzeige auf.
- *GoHome()*: Diese Methode ruft die vom Benutzer im Internet Explorer eingestellte Homepage in der Anzeige auf.



Abbildung 10.23: Webbrowser-Steuerelement zur Anzeige von Webseiten

Details zu den vom Steuerelement bereitgestellten Methoden und Eigenschaften finden Sie in der Hilfe zum .NET Framework.

Hinweis

Das Beispiel wurde sehr einfach gehalten. Ein Klick auf die Schaltflächen *Zurück* und *Vorwärts* aktualisiert z.B. nicht die im Textfeld angezeigte Adresse. Theoretisch ließe sich die Anweisung `Me.TextBox1.Text = Me.WebBrowser1.Uri.ToString` verwenden, um die URL der aktuellen Seite in das Textfeld einzublenden. Bei praktischen Versuchen zeigt sich jedoch, dass das Steuerelement in der Eigenschaft *Uri* nicht immer die Adresse der aktuell angezeigten Seite liefert. Ein in der .NET-Framework-Hilfe zum Download angebotenes Beispiel enthält ebenfalls nichts zur Aktualisierung der Adressanzeige. Die Projektdateien des obigen Beispiels finden Sie im Ordner `\Beisp\Kap10\Browser` auf der Begleit-CD.

10.6.5 Verwenden des PropertyGrid-Steuerelements

Möchten Sie dem Benutzer zur Laufzeit einer Anwendung den Zugriff auf die Eigenschaften von Steuerelementen gewähren? Dann lässt sich das *PropertyGrid*-Steuerelement für diesen Zweck einsetzen (Abbildung 10.24).



Abbildung 10.24: PropertyGrid-Steuerelement zur Anzeige von Eigenschaften

Sie können das Steuerelement aus der Toolbox zu einem Formular hinzufügen. Anschließend ist der Eigenschaft *SelectedObject* des Steuerelements der Name eines anderen Steuerelements zuzuweisen. Die Namen der verfügbaren Objekte werden in einem Listefeld eingeblendet. Das *PropertyGrid*-Steuerelement zeigt dann zur Laufzeit die Eigenschaften des verbundenen Steuerelements an. Ändert der Benutzer die Eigenschaften, wirkt sich dies sofort auf das betreffende Objekt aus (in Abbildung 10.24 wurde z.B. die *Text*-Eigenschaft der Schaltfläche im *PropertyGrid*-Steuerelement angepasst).

Hinweis

Sie finden ein Beispiel zur Verwendung des *PropertyGrid*-Steuerelements im Ordner *\Beisp\Kap10\PropertyGrid* auf der Begleit-CD. Das betreffende Formular-Element enthält keinerlei benutzerspezifischen Code, die Funktionalität zum Anzeigen bzw. Anpassen der Beschriftung der Schaltfläche wird durch das *PropertyGrid*-Steuerelement bereitgestellt.

10.6.6 Container-Steuerelemente zur Layout-Kontrolle

Ändern Formulare ihre Größe, möchte man in der Regel die im Formular enthaltenen Steuerelemente in der Größe und/oder in der Position anpassen. .NET 2.0 stellt daher (neben den bereits besprochenen Steuerelementen wie der *GroupBox* für Options- und Kontrollfelder, dem *Panel*, dem *SplitContainer* sowie dem *TabControl*) weitere Container-Steuerelemente zur Verbesserung der Layout-Kontrolle in Formularen bereit. So ist es bei Formularen hilfreich, wenn bei Größenänderung die Position und die Abmessungen der im Formular enthaltenen Steuerelemente angepasst werden.

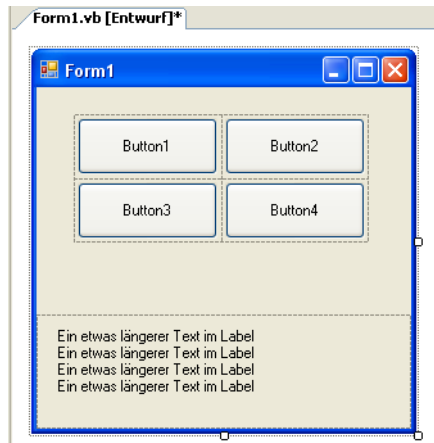


Abbildung 10.25: TableLayoutPanel und FlowLayoutPanel zur Layout-Kontrolle

Sie können die Steuerelemente über die *Dock*- und *Anchor*-Eigenschaft am Formularrand verankern oder dynamisch zur Laufzeit das Layout des Formulars anpassen. Ab .NET 2.0 steht aber mit den Steuerelementen *TableLayoutPanel* und das *FlowLayoutPanel* eine komfortablere Möglichkeit bereit. In Abbildung 10.25 wurden diese beiden Steuerelemente verwendet, um Steuerelemente aufzunehmen. Das *TableLayoutPanel*-Steuerelement wurde im Formularentwurf über die *Anchor*-Eigenschaft an den vier Seiten des Formulars verankert und nimmt vier Schaltflächen auf. Diese wurden in den einzelnen Tabellenzellen eingefügt und über die *Anchor*-Eigenschaft mit den Zellrändern verankert. Die Verankerung über *Anchor* erlaubt übrigens die freie Positionierung im Formular, während *Dock=Top* das *TableLayoutPanel* diekt an den oberen Rand des Formulars in der Formularbreite anordnet.

Das *FlowLayoutPanel*-Steuerelement wurde über die *Dock*-Eigenschaft am unteren Formularrand verankert und nimmt mehrere *Label*-Steuerelemente mit einem Beispielttext auf. Über die *Padding*-Eigenschaft des Steuerelements lässt sich eine Art »innerer Rand« definieren, der den Abstand der enthaltenen Steuerelement bestimmt. Die im *FlowLayoutPanel* hinterlegten *Label*-Steuerelemente sind ebenfalls über die *Anchor*-Eigenschaft mit dem Container verankert. Die im Projekt gewählten Einstellungen bewirken zur Laufzeit, dass die Schaltflächengröße dynamisch an der Formulargröße angepasst wird. Das *FlowLayoutPanel*-Steuerelement ordnet die enthaltenen *Label*-Steuerelemente abhängig von der Formularbreite nebeneinander oder untereinander an. Die gewünschte Flussrichtung des *FlowLayoutPanel* (horizontal oder vertikal) lässt sich über die Eigenschaft *FlowDirection* vorgeben. Statt den Inhalt zu umbrechen, kann das Steuerelement den Inhalt aber auch abschneiden.

Hinweis

Sie finden ein Beispielprojekt mit einem Formular, welches die beiden Steuerelemente als Container enthält, im Ordner `\Beisp\Kap10\TabellenControl` auf der Buch-CD. Das Beispiel benutzt keinen zusätzlichen Code, so dass die Schaltflächen ohne Funktion sind. Trotzdem können Sie zur Laufzeit das Verhalten der Container-Steuerelemente studieren, indem Sie die Formulargröße verändern. Weitere Hinweise zu den jeweiligen Steuerelementen finden Sie in der Hilfe.

Damit möchte ich dieses Kapitel schließen. Sie haben die wichtigsten Steuerelemente und deren Handhabung kennen gelernt. Hinweise zu den jeweiligen Eigenschaften und Methoden finden Sie in der Hilfe des .NET-Framework. Im nächsten Kapitel dreht sich alles um die Gestaltung von Menüs, Symbol- und Statusleisten.

11

Menüs und weitere Leisten

In diesem Kapitel wird gezeigt, wie sich Anwendungsfenster mit Menü-, Symbol- und Statusleisten ausstatten lassen. Microsoft hat in .NET 2.0 eine Reihe neuer, als Strip-Controls bezeichnete, Elemente eingeführt. Nachfolgend wird gezeigt, wie Sie die neuen Elemente einsetzen und wie sich ggf. die älteren .NET 1.x-Controls weiter verwenden lassen. Sie finden auch Hinweise, um die betreffenden Funktionen zur Laufzeit über Visual Basic-Anweisungen zu realisieren.

11.1 Menüs einbinden

Windows-Anwendungen werden typischerweise mit einer eigenen Menüleiste versehen, über deren Einträge sich Befehle abrufen lassen. Das Anwendungsfenster stellt dabei nichts anderes als ein Formular dar, in dem die betreffende Menüleiste hinterlegt wurde. Nachfolgend finden Sie die notwendigen Informationen, um .NET-Anwendungen mit einem Fenster samt Menüleiste auszustatten.

11.1.1 Ein Menü in .NET-Formularen erstellen

Menüs lassen sich interaktiv im Formularedesigner der Entwicklungsumgebung entwerfen. Sie benötigen ein Formular, in dem Sie das betreffende Steuerelement aus der Toolbox einfügen. Anschließend lassen sich die Menüstruktur und dann die Reaktionen auf die einzelnen Befehle definieren.

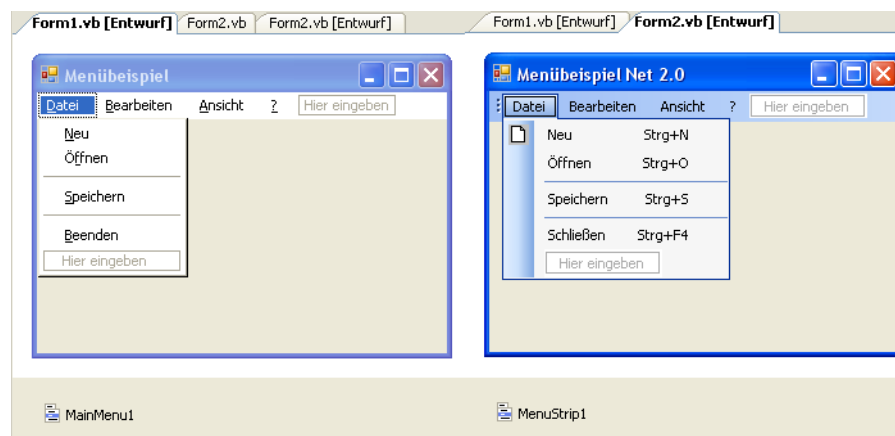


Abbildung 11.1: Formularentwurf mit eingefügten Menü-Steuerelementen

Hinweis

Das .NET Framework 2.0 stellt für Menüs, Kontextmenüs, Symbol- und Statusleisten einen Satz an neuen »Strip«-Steuerelementen bereit, die sich im Design an moderne Microsoft-Anwendungen (z.B. Windows XP, Office 2003) anlehnen. Aus Kompatibilitätsgründen werden die alten .NET 1.x-Steuerelemente, die Menüs, Kontextmenüs, Symbol- oder Statusleisten im alten Windows-Stil abbilden, aber weiter unterstützt. Laden Sie Projekte älterer Visual-Studio-Versionen in Visual Studio 2005 bzw. der Visual Basic 2005 Express Edition, werden die Steuerelemente nach der Konvertierung des Projekts im Formular angezeigt. Allerdings sind die alten .NET 1.x-Controls (mit Ausnahme des *MainMenu*-Steuerelements) standardmäßig nicht mehr in der Toolbox enthalten. Um die alten Steuerelemente noch zu nutzen, müssen Sie diese entweder aus dem Programmcode heraus ansprechen oder manuell in die Toolbox einbinden.

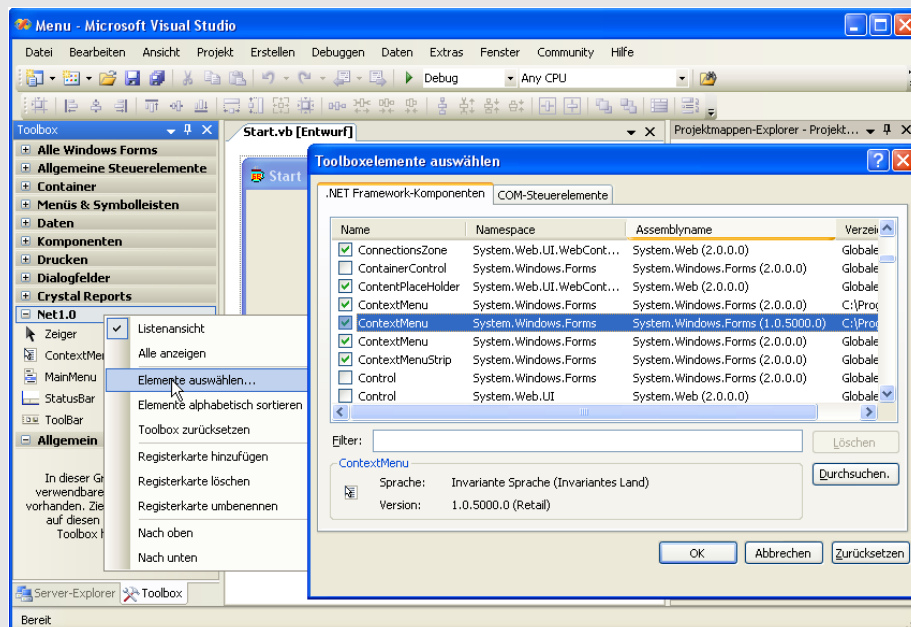


Abbildung 11.2: Alte Controls als Toolboxelemente aufnehmen

Zum Einbinden fügen Sie in der Toolbox zuerst über den Kontextmenübefehl *Registerkarte hinzufügen* eine neue Gruppe (z.B. mit dem Namen »NET 1.x«) ein. Anschließend wählen Sie den Kontextmenübefehl *Elemente auswählen* dieser Gruppe (Abbildung 11.2, Hintergrund). Im Dialogfeld *Toolboxelemente auswählen* (Abbildung 11.2, Vordergrund) suchen Sie auf der Registerkarte *.NET Framework-Komponenten* die Einträge für die fehlenden Komponenten. Markieren Sie deren Kontrollkästchen und schließen Sie das Dialogfeld über die OK-Schaltfläche. Dann sollten die alten Controls in der neuen Gruppe als Toolboxelemente erscheinen. Über den Kontextmenübefehl *Toolbox zurücksetzen* der Toolbox lässt sich der Auslieferungszustand wiederherstellen (benutzerdefinierte Steuerelemente werden entfernt). Bei neuen Projekten empfiehlt sich aber die Verwendung der .NET 2.0 Strip-Controls. Nachfolgend gehe ich, sofern möglich, auf beide Varianten der Controls ein.

Um ein Formular mit einem Menü zu versehen, müssen Sie zuerst die Entscheidung treffen, ob Sie noch das alte *MainMenu*-Steuerelement (Abbildung 11.1, links) oder doch lieber das neue *.NET-2.0-MenuStrip*-Steuerelement (Abbildung 11.1, rechts) mit dem modernen Design verwenden. Der zweite Vorteil (neben dem neuen Design, welches zu den Standard Symbol- und Statusleisten passt) besteht bei diesem Element darin, dass sich im Menü auch Symbole, ComboBox- und TextBox-Elemente einfügen lassen. Zum Einfügen des Menüs in einem Formular gehen Sie in folgenden Schritten vor:

1. Legen Sie in der Entwicklungsumgebung ein neues Projekt als Windows-Anwendung an. Wechseln Sie anschließend zum Formulardesigner und setzen Sie die gewünschten Eigenschaften des Formulars (z.B. Titeltext, die Schaltflächen zum Minimieren/Maximieren eingeblendet, etc.).

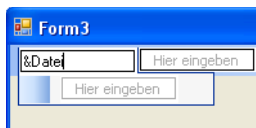


Abbildung 11.3: Ergänzen der Menüstruktur

2. Fügen Sie nun aus der Toolbox das Steuerelement *MenuStrip* oder *MainMenu* im Formulardesign ein. Das Steuerelement wird unterhalb des Formulars als nicht sichtbare Komponente im Komponentenbereich des Formulardesigns eingeblendet (Abbildung 11.1). Im Formularentwurf wird die Menüleiste durch den Text »Hier eingeben« angedeutet.
3. Klicken Sie im Formularentwurf in der Menüleiste auf den Text »Hier eingeben«. Der Designer blendet zwei weitere Platzhalter »Hier eingeben« ein (Abbildung 11.3). Der rechte Platzhalter steht für den jeweils rechts stehenden Menüeintrag. Der zweite unterhalb des Menüeintrags erscheinende Platzhalter symbolisiert das aufgeklappte Menü (Abbildung 11.3). Tippen Sie einen Namen für den betreffenden Menüeintrag ein und bestätigen Sie dies durch Anklicken eines weiteren Eintrag bzw. durch Drücken der -Taste (wechselt zum darunter liegenden Platzhalter) oder -Taste (wechselt zum rechts neben dem Eintrag liegenden Platzhalter).
4. Ergänzen Sie auf diese Weise die Einträge für die komplette Menüleiste, indem Sie auf die jeweiligen Platzhalter klicken. In Abbildung 11.4 habe ich die Einträge *Datei*, *Bearbeiten*, *Ansicht* und *?* gewählt. Achten Sie bei der Wahl der Menübezeichnungen darauf, dass Sie mit den Windows-Konventionen konform bleiben.
5. Im nächsten Schritt gilt es, die einzelnen Menüs mit Befehlen und ggf. Untermenüs zu ergänzen. Klicken Sie im Designer auf den betreffenden Menüeintrag. Der Designer hebt das Menü hervor und blendet den ersten Platzhalter »Hier eingeben« für den betreffenden Menübefehl ein. Klicken Sie auf diesen Platzhalter und geben Sie den Menübefehl ein. Sobald Sie die -Taste drücken, wechselt der Designer zum nächsten Menübefehl.

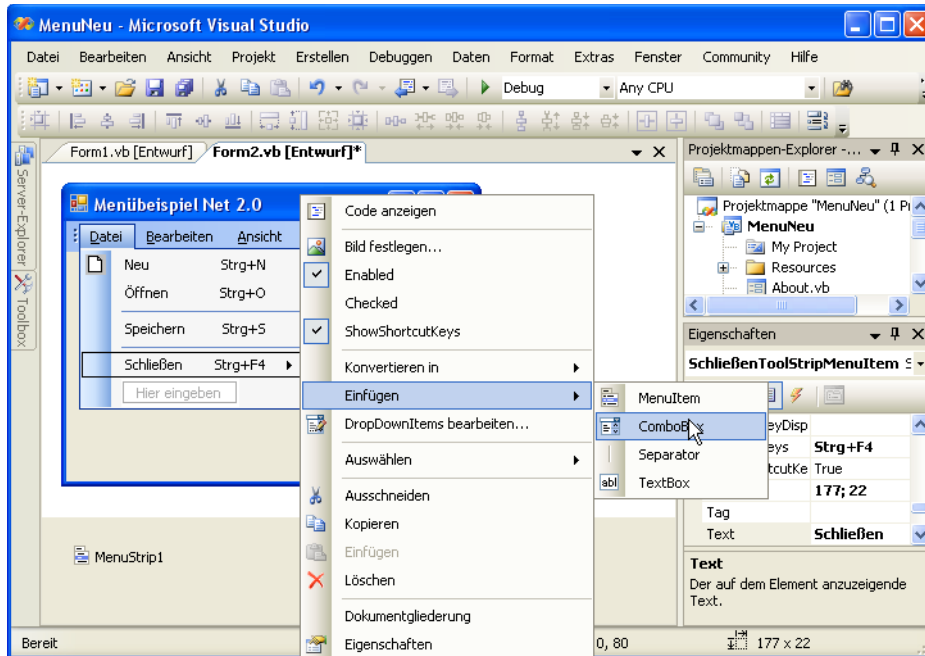


Abbildung 11.4: Entwurf der Menüleiste im Ansicht-Designer

Die Schritte zum Definieren eines Menüs sind also recht einfach. Hier noch einige Tipps, wie sich spezielle Eigenschaften für Menübefehle realisieren lassen:

- Um ein Untermenü zu einem Menübefehl einzufügen, klicken Sie auf den rechts vom betreffenden Menübefehl angezeigten Platzhalter »Hier eingeben«. Dann können Sie den Untermenübefehl eintragen.
- Möchten Sie einen waagerechten Trennstrich als Separator zwischen zwei Menübefehlen einfügen? In Abbildung 11.4 enthält das Menü *Datei* zwei solcher Trennzeichen, um Menübefehle zu Gruppen zusammenzufassen. Dann tippen Sie einfach ein Minuszeichen als Menütext ein. Die Alternative: Klicken Sie mit der rechten Maustaste auf den Menübefehl, der unterhalb des Trennzeichens liegen soll. Wählen Sie anschließend im Kontextmenü den Befehl *Einfügen/Separator* (beim *MenuStrip*-Steuerelement) bzw. *Trennzeichen einfügen* (beim *MainMenu*-Steuerelement).
- Bei Menüs und Menüeinträgen zeigt Windows die Abkürzungstasten zur Tastaturnavigation durch unterstrichene Buchstaben an (siehe z.B. Menü *Datei* in Abbildung 11.4). Diese Abkürzungstasten für die Tastaturnavigation werden über den Namen des Menüeintrags definiert, indem Sie dem betreffenden Buchstaben ein &-Zeichen voranstellen (z.B. *&Datei*, *&Neu* etc.).
- Windows-konforme Menüs weisen häufig zusätzliche Abkürzungstasten auf (z.B. *[Strg]+[V]* für den Befehl *Einfügen* im Menü *Bearbeiten*). Um eine solche Abkürzungstaste zuzuweisen, markieren Sie den Befehl und wählen den gewünschten Wert über das Listefeld der Eigenschaft *Shortcut* (beim *MainMenu*-Steuerelement) bzw. über

das Menü der Eigenschaft *ShortcutKeys* (beim *MenuStrip*-Steuerelement) des Eigenschaftfensters (Abbildung 11.4). Über die Eigenschaft lässt sich eine Liste mit vorgegebenen Tastenkombinationen öffnen und die gewünschte Tastenkombination auswählen.

- Möchten Sie einem Menüeintrag ein Bild zuweisen (nur beim *MenuStrip*-Steuerelement), wählen Sie diesen mit der rechten Maustaste an und klicken im Kontextmenü auf den Befehl *Bild festlegen*. Anschließend lässt sich in einem Zusatzdialog *Ressource auswählen* eine lokale Bilddatei oder eine Ressource zum Menü hinzufügen.
- Um Trennzeichen oder Menübefehle im Entwurf zu löschen, wählen Sie diese mit der rechten Maustaste an und klicken im Kontextmenü auf den Befehl *Löschen*. Diese Operation lässt sich über die Schaltfläche *Rückgängig* der Symbolleiste wieder zurücknehmen.

Anschließend können Sie das Projekt bereits übersetzen und ausführen lassen. Das Menü sollte dann im Fenster eingeblendet werden und Sie können die einzelnen Menüeinträge per Maus oder Tastatur abrufen. Wie Sie die Menüeinträge an Funktionen anbinden, die bei Anwahl des jeweiligen Befehls auszuführen sind, wird im nächsten Abschnitt besprochen.

11.1.2 Menübefehle mit Funktionen belegen

Das Steuerelement übernimmt automatisch die Verwaltung der Menüstruktur, d.h., sobald Sie die Struktur des Menüs im Ansicht-Designer entworfen haben, kann der Benutzer zur Laufzeit die Menübefehle anwählen, Untermenüs öffnen oder schließen etc. Sie müssen allerdings in einem zweiten Schritt dafür sorgen, dass die Menüeinträge auch mit entsprechenden Funktionen belegt werden (z.B. der Befehl *Öffnen* sollte einen *Öffnen*-Dialog zeigen). Dies geschieht, indem die betreffende Funktion über Code in den *Click*-Ereignisbehandlungsroutinen der einzelnen Menübefehle hinterlegt wird. Beachten Sie dabei, dass gemäß den Windows-Konventionen Aktionen nur den Befehlen innerhalb eines geöffneten Menüs, niemals aber den Einträgen der Menüleiste zugewiesen werden dürfen.

1. Um einem Menübefehl eine Funktion zuzuweisen, öffnen Sie das Menü im Ansicht-Designer und wählen anschließend den Menübefehl per Doppelklick an.
2. Die Entwicklungsumgebung wechselt zum Codefenster und fügt automatisch den Prozedurrumpf einer Ereignisbehandlungsroutine für das betreffende Menüelement ein. Ergänzen Sie den Prozedurrumpf dieser Ereignisbehandlungsroutine um die Befehle zur Implementierung der gewünschten Funktion (Abbildung 11.5).

Um beispielsweise über den Befehl *Beenden* im Menü *Datei* eine Anwendung zu beenden, fügen Sie in der *Click*-Ereignisbehandlungsroutine dieses Befehls die folgende Anweisung ein:

```
Application.Exit()
```

Dieser Befehl ruft die *Exit*-Methode des *Application*-Objekts auf. Die Methode bewirkt, dass das .NET Framework die laufende Anwendung beendet.

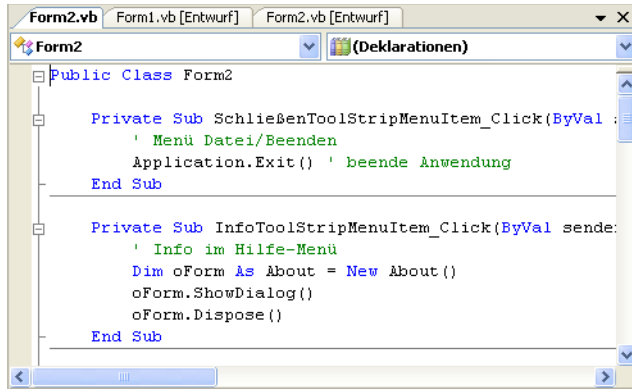


Abbildung 11.5: Ereignisbehandlungsroutinen für Menübefehle im Codefenster

In Abbildung 11.5 ist der Code der *Click*-Ereignisbehandlungsroutine (Prozedur *InfoToolStripMenuItem_Click()*) des Befehls *Info* des Menüs ? zu sehen. In der ersten Anweisung wird eine neue Formularinstanz angelegt, die dann über die *ShowDialog()*-Methode anzuzeigen ist. Sobald der Benutzer das Formular schließt, sorgt die *Dispose()*-Methode in der Ereignisbehandlungsroutine dafür, dass das Formularobjekt freigegeben wird. Allerdings ist ein Aufruf der *Dispose()*-Methode nicht zwingend erforderlich, da der Garbage Collector die Formularinstanz nach dem Schließen des Formulars aus dem Speicher entfernt.

Die betreffenden Anweisungen haben Sie bereits in den vorhergehenden Kapiteln kennen gelernt. Der obige Code stellt also keine Neuerung dar. Auf diese Weise können Sie beliebige Funktionalitäten an ein Menü anbinden.

Hinweis

Ich habe ein kleines Projektbeispiel entwickelt und im Ordner `\Beisp\Kap11\Menu` auf der Begleit-CD hinterlegt. Laden Sie die Datei *Menu.sln* in der Entwicklungsumgebung und übersetzen Sie diese. Die Anwendung zeigt ein Startformular *Start.vb* mit Schaltflächen, über welche Sie ein mit dem alten *MainMenu*-Steuerelement realisiertes (*Form1.vb*) oder ein mittels des neuen *MenuStrip*-Steuerelements (*Form2.vb*) realisiertes Formular aufrufen können. Innerhalb des jeweiligen Menüs sind einige Befehle mit Funktionen belegt. Sie können daher die Verwendung beider Menüvarianten studieren.

Ein Menü per Code einfügen

Sie können Menüs auch direkt zur Laufzeit über entsprechende Anweisungen zum Formular hinzufügen. Dies ist mit wenigen Anweisungen (z.B. unter Verwendung des alten *MainMenu*-Steuerelements) möglich.

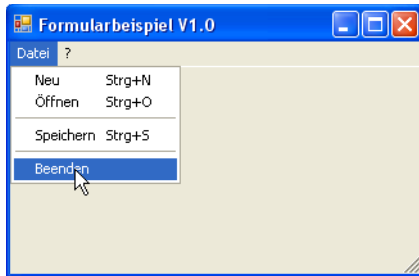


Abbildung 11.6: Beispielformular mit geöffnetem Menü

Der Programmcode zum Erzeugen des Formulars (Abbildung 11.6) sowie zum Gestalten des Menüs wird am besten in einer eigenen Klasse hinterlegt, die folgendermaßen deklariert ist:

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    ' Deklarationen der Variablen
    ...
    Public Sub New()
        MyBase.New()
        ...
    End Sub
End Class
```

Dies kennen Sie bereits aus den Formularbeispielen der vorhergehenden Kapitel. Die Objektvariablen für das Hauptmenü sowie für jeden Menüeintrag (auch die Trennzeichen) werden über folgende Anweisungssequenz im Deklarationsteil der Klasse vereinbart:

```
Friend WithEvents MainMenu1 As MainMenu
Friend WithEvents MenuItem1 As MenuItem
Friend WithEvents MenuItem2 As MenuItem
Friend WithEvents MenuItem3 As MenuItem
Friend WithEvents MenuItem4 As MenuItem
...
Friend MenuItem14 As MenuItem ' keine Ereignisbehandlung
Friend MenuItem15 As MenuItem
```

Die beiden letzten Einträge enthalten kein Schlüsselwort *WithEvents*, da bei Trennzeichen keine Ereignisbehandlung erforderlich ist. Anschließend kann innerhalb der Prozedur des *New*-Konstruktors das Menü samt seinen Einträgen konstruiert werden. Die folgende Anweisungssequenz instantiiert das neue Formular sowie die Objektvariablen für das Menü:

```
Public Sub New()
    MyBase.New() ' Neues Formular

    Me.MainMenu1 = New MainMenu() ' Hauptmenü
    Me.MenuItem1 = New MenuItem() ' Untermenüeinträge
    Me.MenuItem2 = New MenuItem()
    ..
End Sub
```

Für jeden deklarierten Menüeintrag ist eine entsprechende Objektinstanz anzulegen. Das Hauptmenü wird anschließend über die *AddRange()*-Methode der *MenuItems*-Collection an das Formularobjekt gebunden:

```
Me.MainMenu1.MenuItems.AddRange(
    New MenuItem() {Me.MenuItem1, Me.MenuItem2})
```

Hier besitzt das Hauptmenü die beiden Einträge *MenuItem1* und *MenuItem2*. Die Eigenschaften (Name etc.) dieser beiden Objekte werden über deren Objektnamen definiert. Dabei ist es auch erforderlich, die Untermenüeinträge mit anzugeben. Daher werden im Programmcode des *New*-Konstruktors erst die Eigenschaften der jeweiligen Menüeinträge vereinbart. Die folgende Sequenz vereinbart einen Befehl:

```
'MenuItem4 = Untermenü "Neu" für das Menü "Datei"
Me.MenuItem4.Index = 0
Me.MenuItem4.Shortcut = Shortcut.CtrlN ' Tastenkürzel
Me.MenuItem4.Text = "&Neu" ' Beschriftung
```

Die Eigenschaft *Index* wird mit fortlaufenden Werten zwischen 0 und n-1 belegt und gibt die Reihenfolge der Menüeinträge im jeweiligen Menü an. Über die Eigenschaft *Text* lässt sich die Menübezeichnung vereinbaren. Für ein Trennzeichen ist die Eigenschaft *Text* auf den Wert "-" zu setzen. Ein &-Zeichen im Namen des Menüeintrags definiert die Abkürzungstaste, die dem im Menünamen unterstrichenen Buchstaben entspricht. Diese Abkürzungstaste (hier z.B. **Alt** + **N**) wirkt nur bei einem geöffnetem Menü. Soll eine zusätzliche Abkürzungstaste definiert werden, die die Auswahl des Befehls auch bei nicht geöffnetem Menü erlaubt, verwenden Sie die Eigenschaft *Shortcut*. Dieser wird eine benannte Konstante der Art *CtrlN*, *AltF11* etc. zugewiesen.

Hinweis

Die Einträge der *Shortcut*-Enumeration sind in der Hilfe dokumentiert. Ob ein solches Tastenkürzel zur Laufzeit im Menü eingeblendet wird, lässt sich über die Eigenschaft *ShowShortcut* steuern. Mit dem Wert *True* erscheint das Tastenkürzel, der Wert *False* unterdrückt die Anzeige. Ein Menüeintrag lässt sich übrigens über die Eigenschaft *Visible* sperren, indem Sie den Wert auf *False* setzen.

Sobald die Eigenschaften eines Menüeintrags dem Objekt zugeordnet sind, können Sie daran gehen, den betreffenden Hauptmenüzweig zu erzeugen. Die folgende Anweisungssequenz definiert ein Hauptmenü mit den Einträgen *Datei* und ?:

```
Me.MenuItem1.Index = 0 'MenuItem1 = Menü "Datei"
Me.MenuItem1.MenuItems.AddRange _
    (New MenuItem() _
    {Me.MenuItem4, Me.MenuItem5, Me.MenuItem14, _
    Me.MenuItem6, Me.MenuItem15, Me.MenuItem7})
Me.MenuItem1.Text = "&Datei" ' Menübezeichnung

Me.MenuItem2.Index = 1 'MenuItem2 = ?-Menü
Me.MenuItem2.MenuItems.AddRange(New MenuItem() {Me.MenuItem3})
Me.MenuItem2.Text = "&?"
```

Hier wird die *AddRange()*-Methode der *MenuItems*-Auflistung benutzt, um die Einträge des Menüs einer Auflistung hinzufügen. Die Auflistung wird über *New MenuItem()* erzeugt, die Elemente sind dann der Auflistung in geschweiften Klammern {...} zuzuweisen. Die Eigenschaft *Text* definiert die Menübezeichnung.

Hinweis

Das betreffende Menüelement, zu dem ein Untermenü erzeugt werden soll, wird immer über den Objektnamen (hier *Me.MenuItem1* für den ersten Eintrag im Hauptmenü) identifiziert. Benötigen Sie ein Untermenü zu einem Menübefehl? Dann wenden Sie die *AddRange()*-Methode einfach auf das betreffende Objekt an. Um im Menü *Datei* den Befehl *Neu* mit einem Untermenü (z.B. *Leeres Dokument*) zu versehen, ist die *AddRange()*-Methode aus obiger Befehlssequenz sinngemäß auf das Objekt *Me.MenuItem4* anzuwenden.

Der Ansicht-Designer der Entwicklungsumgebung vergibt beim *MainMenu*-Steuerelement die Objektnamen für Steuerelemente und Menüeinträge nach einem automatischen Schema (Steuerelement-Bezeichnung plus eine fortlaufende Nummer). Angaben wie *MenuItem1*, *TextBox1*, *ToolBarButton1* etc. sind nicht sonderlich transparent (obwohl ich diese Bezeichnungen in vielen Beispielen aus »Faulheit« beibehalten habe). Sie können aber die Objektnamen über die Eigenschaft *Name* des betreffenden Menüelements anpassen. Der Objektnamen *MenuItemDatei* ist wohl etwas aussagekräftiger als *MenuItem1*. Dies wurde in einigen der folgenden Beispiele genutzt.

Um das Formular mit dem Menü zu erzeugen, müssen Sie die in den bisherigen Beispielen benutzte Befehlssequenz etwas erweitern. Der nachfolgende Codeausschnitt zeigt die betreffenden Befehle:

```
Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
Me.ClientSize = New System.Drawing.Size(320, 161)
Me.Menu = Me.MainMenu1 ' Menüobjekt an Formular binden
```

```
Me.Name = "Form1"
Me.Text = "Menübeispiel"
```

Neu ist die Eigenschaft *Menu*, der die Objektadresse des Objekts *MainMenu1* zugewiesen wird. Mit diesen Anweisungen haben Sie bereits ein funktionsfähiges Menü erzeugt, welches sich zur Laufzeit öffnen und bedienen lässt. Um nun noch Befehle an die einzelnen Menüeinträge anzubinden, fügen Sie einfach entsprechende Ereignisbehandlungsroutinen hinzu. In obigen Anweisungen wurde der Befehl *Beenden* über das Objekt *MenuItem7* spezifiziert. Die nachfolgende Ereignisbehandlungsroutine reagiert auf das Anklicken des Befehls *Beenden* im Menü *Datei* und schließt die Anwendung:

```
Private Sub MenuItem7_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MenuItem7.Click
    Application.Exit() ' Menü Datei/Beenden -> Anwendung schließen
End Sub
```

Die Ereignisbehandlungsroutine wird über das Schlüsselwort *Handles* an das *Click*-Ereignis des betreffenden Objekts gebunden. Innerhalb der Prozedur wird einfach die *Exit*-Methode des *Application*-Objekts aufgerufen. Diese Techniken kennen Sie aber bereits aus den vorhergehenden Kapiteln. Die Gestaltung von Menüs ist also im Grunde nichts Neues.

Hinweis

Ich habe ein Beispielprojekt, in dem ein Formular mit Menüleiste direkt per Quellcode erzeugt wird, im Ordner `\Beisp\Kap11\MenuCode` auf der Begleit-CD hinterlegt. Wenn Sie das Projekt laden und ausführen, erhalten Sie ein leeres Formular mit einem einfachen Menü (Abbildung 11.6). Die Befehle *Datei/Neu*, *Datei/Beenden* und *?/Info* sind dabei mit Ereignisbehandlungsroutinen belegt. Den Code zum Erzeugen des Menüs finden Sie im Modul *Menu.vb*.

11.1.3 Menüs mit Häkchen und Punkten

In einigen Windows-Anwendungen ermöglichen Menüeinträge, eine Option ein- und wieder auszuschalten. Die betreffenden Menüeinträge werden bei aktivierter Option mit einem Häkchen versehen (Abbildung 11.7). Der Befehl *Statusleiste* im Menü *Ansicht* des Windows-Ordnerfensters stellt ein solches Beispiel dar. Darüber hinaus gibt es den Fall, dass mehrere Befehle eines Menüs zu einer Optionsgruppe zusammengezogen werden. Durch Anwahl eines Befehls der Gruppe lässt sich eine Option setzen. Die aktuell gesetzte Option wird im Menü durch einen dem betreffenden Befehl vorangestellten kleinen Punkt symbolisiert (Abbildung 11.7). Diese Optionen können Sie im Ansicht-Designer über Eigenschaften der jeweiligen Menüelemente vereinbaren:

- Möchten Sie, dass ein Menüeintrag über ein Häkchen (Abbildung 11.7) als markiert dargestellt wird? Dann wählen Sie diesen Eintrag im Designer und setzen die Eigenschaft *Checked* auf den Wert *True*.

- Soll eine Gruppe von Befehlen die Auswahl einer Option erlauben, wobei die aktuell angewählte Option durch einen Punkt symbolisiert (Abbildung 11.7) wird? Dann setzen Sie die Eigenschaften *Checked* und *RadioCheck* des markierten Eintrags auf den Wert *True*. Allerdings steht diese Eigenschaft nur beim älteren *MainMenu*-Steuerelement zur Verfügung. Beim neuen *.NET 2.0-MenuStrip*-Steuerelement müssten Sie mit Häkchen arbeiten oder ggf. einen Punkt als Symbol im Menü ein- bzw. ausblenden.

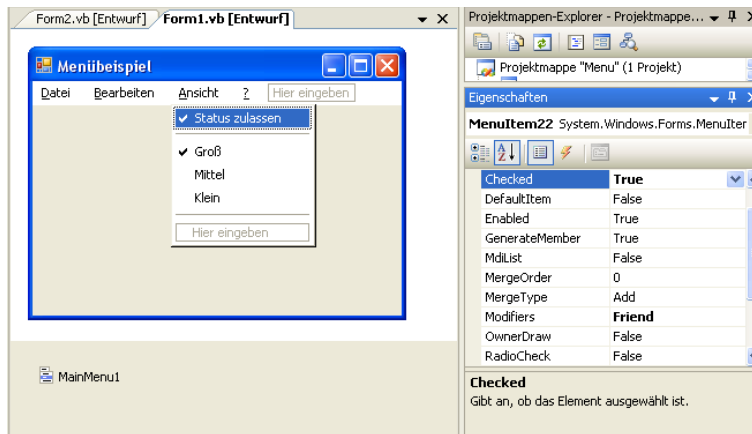


Abbildung 11.7: Menüeinträge mit Punkt und Häkchen sowie deren Eigenschaften

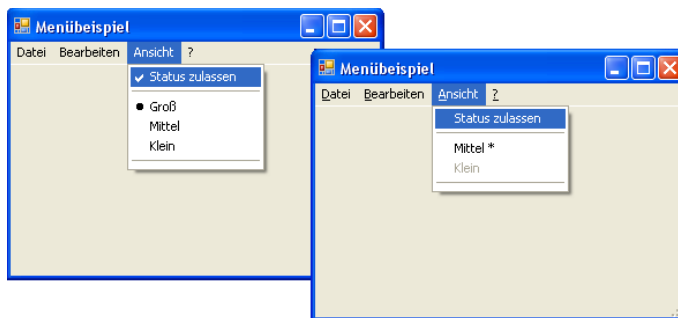


Abbildung 11.8: Ansichten des Menüs

Die Steuerung, um das Häkchen bei der Anwahl des Befehls ein- oder auszublenden, muss per Programm erfolgen. Auch das Umsetzen der Option bei Anwahl eines anderen Menübefehls einer Optionsgruppe erfordert entsprechende Programmanweisungen. Die folgende Codesequenz zeigt die Ereignisbehandlungsroutine für einen solchen Befehl. Hier wird beim Befehl *Status zulassen* wahlweise ein Häkchen ein- oder ausgeblendet. Zudem bewirkt die Anwahl des Befehls, dass ein Menübefehl *Gross* im gleichen Menü ausgeblendet und der Befehl *Klein* gesperrt wird. Gleichzeitig passt die Ereignisbehandlungsroutine den Menütext des Menüs *Mittel* dynamisch an (Abbildung 11.8).

Nachfolgend sehen Sie den Code der *Click*-Ereignisbehandlungsroutine des Befehls *Status zulassen* im Menü *Ansicht*:

```

Private Sub MenuItem22_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MenuItem22.Click
' Status anzeigen -> Zustand umkehren & Menü anpassen
Me.MenuItem22.Checked = Not Me.MenuItem22.Checked
If Me.MenuItem22.Checked = False Then
    Me.MenuItem17.Visible = False ' Befehl "Groß" ausblenden
    Me.MenuItem18.Text = "Mittel *" ' neuer Text
    Me.MenuItem19.Enabled = False ' Befehl "Klein" sperren
Else
    Me.MenuItem17.Visible = True ' Befehl "Groß" einblenden
    Me.MenuItem18.Text = "Mittel" ' neuer Text
    Me.MenuItem19.Enabled = True ' Befehl "Klein" freigeben
End If
End Sub

```

Letztendlich setzt die Ereignisbehandlungsroutine die betreffenden Eigenschaften der Menüsteuerelemente auf die gewünschten Werte. Die Ereignisbehandlung, um dem Befehl *Mittel* bei Anwahl einen Punkt als Markierung zuzuweisen, sieht folgendermaßen aus:

```

Private Sub MenuItem18_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MenuItem18.Click
' Befehl "Mittel" - Markierung auf Befehl umsetzen
Me.MenuItem18.Checked = True ' Punkt einblenden
Me.MenuItem18.RadioCheck = True
Me.MenuItem17.Checked = False ' Punkt ausblenden
Me.MenuItem17.RadioCheck = False
Me.MenuItem19.Checked = False ' Punkt ausblenden
Me.MenuItem19.RadioCheck = False
End Sub

```

In der Prozedur werden die Eigenschaften *Checked* und *RadioCheck* der jeweiligen Menüeinträge auf den gewünschten Wert gesetzt. In realen Projekten müssen Sie anschließend noch den Code, der die Auswirkungen des Befehl umsetzt, einfügen.

Hinweis

Das komplette Projektbeispiel findet sich im Ordner *\Beisp\Kap11\Menu* auf der Begleit-CD. Wenn Sie das Formular (Modul *Form1.vb*) mit dem alten Menüstil aufrufen, enthält dieses im Menü *Ansicht* einige Befehle mit Häkchen und Punkt. Im Modul *Form2.vb* kommt dagegen das *MenuStrip*-Steuerelement zum Einsatz, welches die *RadioCheck*-Eigenschaft nicht unterstützt. Daher werden in diesem Formular die Menübefehle mit einem Häkchen markiert. Der Quellcode der Ereignisbehandlungsroutinen zeigt, wie sich das Setzen oder Löschen der Markierung steuern lässt.

Hinweis (Fortsetzung)

Im Ordner `\Beisp\Kap11\MenuOption` der Begleit-CD finden Sie das modifizierte *List-View*-Beispiel aus dem vorhergehenden Kapitel.

Es wird das alte *MainMenu*-Steuerelement zur Menüdarstellung benutzt. Die Darstellung der Symbolgröße im *List-View*-Zweig lässt sich neben den Schaltflächen auch über die Befehle des Menüs *Ansicht* umschalten. Der Befehl *Status zulassen* des Menüs *Ansicht* blendet den Befehl *Gross* ein/aus und sperrt den Befehl *Klein* bei Bedarf.

Menüs mit Symbolen

Manche Windows-Anwendungen zeigen in Menüs vor den Namen der Befehle auch kleine Symbole. Verwenden Sie das neue *MenuStrip*-Steuerelement, lassen sich solche Symbole direkt über den Kontextmenübefehl *Bild einfügen* der Eigenschaft *Image* des *ToolStripMenuItem*-Controls zuweisen (siehe vorhergehende Abschnitte). Das alte .NET 1.x-Control *MenuItem* unterstützt das Einbinden von Texten und Symbolen in Menüs leider nicht. Sie haben aber die Möglichkeit, über die Eigenschaft *OwnerDraw* zu steuern, wie die Menüs gezeichnet werden. Mit der Standardeinstellung *OwnerDraw=False* wird das Zeichnen der Menüelemente durch die .NET Framework-Klassen erledigt. Bei *OwnerDraw=True*, wird bei Anwahl des betreffenden Menüelements ein *OwnerDraw*-Ereignis ausgelöst und in der zugehörigen Ereignisbehandlungsroutine kann ein Icon im betreffenden Element gezeichnet werden. Da dieses Thema den geplanten Ansatz dieses Buches sprengt, möchte ich auf im Internet angebotene Lösungen verweisen. Eine von Markus Palme implementierte und unter der GNU General Public License bereitgestellte Klassenbibliothek findet sich unter folgender URL:

www.activevb-archiv.de/VZ-Markus/palmbytes/content/dotnet/dl/DotNetUI

Interessierte Leser finden ein kleines Beispiel im Ordner `\Beisp\Kap11\IconMenu` der Begleit-CD. Zur Implementierung dieses Menüs benutzt das Beispiel zwei spezielle Klassen *MenuIcon* (übernimmt die Ausgabe von Menüeinträgen mit oder ohne Symbol) und *IconMenuItem* (erzeugt das Hauptmenü). Der Quellcode der Beispiele wurde so gehalten, dass er sich möglichst an die bisher diskutierten Techniken zur Menügestaltung anlehnt.

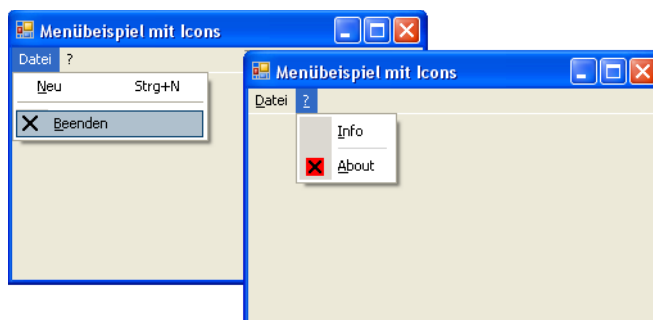


Abbildung 11.9: Menüs mit Symbolen

Die Projektdatei *IconMenu.vb* verwendet nur die Klasse *IconMenuItem*, um in einem Menü Befehle mit und ohne Symbole einzublenden. Dabei wurde der Eintrag *Neu* im Menü *Datei* bewusst über das .NET Framework-Element *MenuItem* erzeugt, während der Eintrag *Beenden* mit der erweiterten Klasse *IconMenuItem* gezeichnet wird. Das Beispiel *IconMenu1.vb* verwendet zusätzlich die Klasse *MenuIcon* zur Konstruktion der Menüleiste. Details sind dem Quellcode der beiden Beispiele zu entnehmen.

11.2 Kontextmenüs nutzen

.NET Framework unterstützt die Gestaltung und Verwaltung von Kontextmenüs über Steuerelemente. Dies erlaubt es Ihnen, Kontextmenüs interaktiv im Fenster *Ansicht-Design* zu entwerfen, an andere Steuerelemente anzubinden oder dynamisch per Programm zu generieren. Ähnlich wie bei Menüelementen unterstützt .NET 2.0 zwei Varianten des Steuerelements. Das aus Kompatibilitätsgründen (zu .NET 1.x) noch vorhandene *ContextMenu*-Steuerelement erzeugt konventionelle Kontextmenüs, die in der Optik zum *MainMenu*-Steuerelement passen. Allerdings müssten Sie das entsprechende Steuerelement manuell in der Toolbox einbinden (siehe Anmerkung im Abschnitt »Ein Menü in .NET-Formularen erstellen«). Für neue Projekte sollten Sie das neue *ContextMenuStrip*-Element verwenden, welches Menüs im Design der *MenuStrip*-Elemente generiert. Da sich der Code zur Verwaltung des *ContextMenuStrip*-Elements etwas von den Anweisungen zur Verwaltung des *ContextMenu*-Elements unterscheidet, wird nachfolgend nur die Verwendung des neuen *ContextMenuStrip*-Elements besprochen.

11.2.1 Ein Kontextmenü zum Formular hinzufügen

Wie Sie Kontextmenüs erzeugen, an Steuerelemente binden und mit Befehlen versehen, wird an dem in Abbildung 11.10 gezeigten Formular demonstriert. Dieses enthält ein *Label*- und ein *RichTextBox*-Steuerelement. Klickt der Benutzer mit der rechten Maustaste auf eines der Steuerelemente, erscheint ein Kontextmenü. Beim *Label*-Steuerelement wird das Menü dynamisch zur Laufzeit generiert. Wählt der Benutzer einen Befehl im Kontextmenü aus, soll ein Dialogfeld mit einer Meldung erscheinen bzw. beim Befehl *Beenden* ist die Anwendung zu schließen. Um das Kontextmenü mit dem neuen *ContextMenuStrip*-Element zu implementieren, gehen Sie in folgenden Schritten vor:



Abbildung 11.10: Beispiel mit zwei Kontextmenüs

1. Legen Sie in der Entwicklungsumgebung ein neues Projekt als Windows-Anwendung an und setzen Sie die Eigenschaften des Formulars gemäß Abbildung 11.10.
2. Fügen Sie gemäß Abbildung 11.10 je ein *Label*- und ein *RichTextBox*-Steuerelement zum Formular hinzu. Setzen Sie die Eigenschaft *MultiLine* des *RichTextBox*-Steuerelements auf *true* und klicken Sie auf die Schaltfläche der Eigenschaft *Lines*. Dann können Sie im Auflistungseditor mehrere Textzeilen für das Steuerelement vorgeben.
3. Fügen Sie nun aus der Toolbox das *ContextMenuStrip*-Steuerelement im Formular-design ein. Das Steuerelement wird als nicht sichtbare Komponente unterhalb des Designbereichs platziert.
4. Benennen Sie die Eigenschaft *Name* des markierten *ContextMenuStrip*-Steuerelements im Eigenschaftenbereich um. Dieser Schritt ist zwar nicht notwendig, erleichtert aber später im Programmcode die Identifikation des Kontextmenüs. Ich habe hier *RichTextContextMenu2* gewählt. In der Praxis empfehlen sich aber abgekürzte Namen wie *cmiRichText*, um die Namen der *Click*-Methoden nicht zu lang werden zu lassen. Falls Sie sich für eine Umbenennung entscheiden, sollten Sie diese übrigens sofort nach dem Erstellen des Kontextmenüeintrags durchführen. Die Objektnamen werden nämlich für die automatisch erzeugten Namen der Ereignismethoden benutzt. Benennen Sie ein Objekt später um, wirkt sich dies nicht auf die Namen der Ereignismethoden aus, d.h., Sie müssen deren Namen manuell korrigieren!

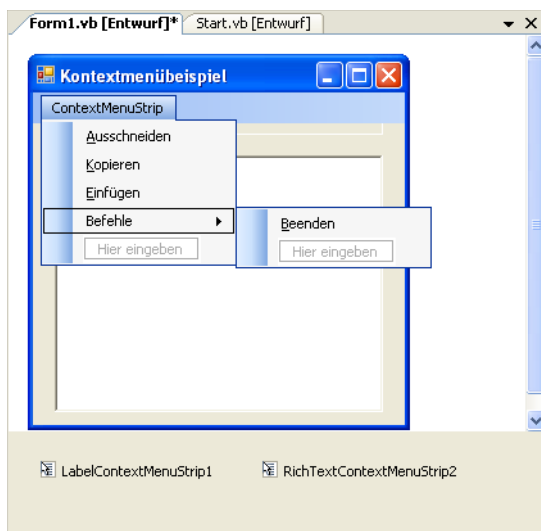


Abbildung 11.11: Entwurf des Kontextmenüs

5. Sobald das *ContextMenuStrip*-Steuerelement im Designer markiert ist, blendet dieser den Text *Kontextmenü* am oberen Rand der Titelleiste des Formulars ein. Klicken Sie auf diesen Text und definieren Sie die Befehle für das Kontextmenü des Steuerelements *RichTextContextMenu2* gemäß Abbildung 11.11. Die Vorgehensweise ist dabei identisch mit dem Entwurf eines Menüs, d.h., Sie haben die Details im vorhergehenden Abschnitt bereits kennen gelernt.

6. Fügen Sie aus der Toolbox ein zweites *ContextMenuStrip*-Steuerelement im Formular-
design ein und benennen Sie dieses über die Eigenschaft *Name* des Eigenschaftens-
fensters in *LabelContextMenuStrip1* um. Definieren Sie anschließend den Befehl *Been-*
den für dieses Kontextmenü (die restlichen Befehle werden später dynamisch
hinzugefügt).
7. Jetzt gilt es noch, die beiden Kontextmenüs an die Steuerelemente anzubinden. Mar-
kieren Sie das *Label*-Steuerelement, wechseln Sie zu dessen Eigenschaftensfenster und
wählen Sie über das Listenfeld der Eigenschaft *ContextMenuStrip* den Eintrag
LabelContextMenuStrip1.
8. Wiederholen Sie den vorhergehenden Schritt für das *RichTextBox*-Steuerelement,
indem Sie dieses markieren und dessen Eigenschaft *ContextMenuStrip* auf den Ein-
trag *RichTextContextMenuStrip2* setzen.

Mit diesen Schritten haben Sie zwei Kontextmenüs mit Befehlen definiert und an die Steuerelemente gebunden. Wenn Sie anschließend das Projekt in der Entwicklungsumgebung übersetzen und ausführen, wird das Formular die Steuerelemente zeigen. Klickt der Benutzer mit der rechten Maustaste auf die Steuerelemente, erscheint das zugehörige Kontextmenü mit den vordefinierten Befehlen.

Hinweis

Beim alten .NET 1.x *ContextMenu*-Steuerelement verwenden Sie die gleichen Schritte, wobei dort die Eigenschaft *ContextMenu* auf den Namen des *ContextMenu*-Steuerelements zu setzen ist.

Statische Kontextmenübefehle mit Funktionen belegen

Das Anbinden eines statisch im Designer definierten Kontextmenüeintrags an einen Befehl oder eine Funktion ist denkbar einfach. Sie müssen lediglich eine *Click*-Ereignisbehandlungsroutine für den betreffenden Befehl im Codeteil definieren. Gehen Sie folgendermaßen vor:

1. Markieren Sie das gewünschte *ContextMenuStrip*-Steuerelement (bzw. *ContextMenu* bei Verwendung der alten Controls) im Komponentenbereich des Formulardesigners, um den Platzhalter *Kontextmenü* im Formularentwurf einzublenden.
2. Öffnen Sie das Kontextmenü im Designer, indem Sie auf Platzhalter *Kontextmenü* klicken. Anschließend wählen Sie den gewünschten Menüeintrag mit einem Doppelklick im Designfenster an.
3. Die Entwicklungsumgebung wechselt zum Codefenster und generiert automatisch den Rumpf der *Click*-Ereignisbehandlungsroutine. Fügen Sie in der Ereignisbehandlungsroutine des *Click*-Ereignisses die gewünschten Befehle ein.

In dem hier gezeigten Beispiel soll bei Anwahl der Befehle ein Meldungsfeld erscheinen, welches einen Hinweis auf den gewählten Befehl gibt. Bei Anwahl des Befehls *Beenden* soll das Programm terminieren. Die nachfolgende Codesequenz zeigt die Anweisungen für die Befehle *Einfügen* und *Befehle/Beenden* des *RichTextBox*-Kontextmenüs:

```
Private Sub MenuItem3_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MenuItem3.Click
' Kontextmenü Einfügen
    MessageBox.Show("Einfügen", "RichTextBox")
End Sub

Private Sub MenuItem5_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MenuItem5.Click
' Kontextmenü Befehle/Beenden
    Application.Exit() ' Beenden der Anwendung
End Sub
```

Die Prozedurköpfe werden von der Entwicklungsumgebung automatisch generiert. Die in den Prozeduren enthaltenen Anweisungen wurden bereits mehrfach in den vorhergehenden Beispielen benutzt und sollten Ihnen bekannt sein.

Ein Kontextmenü per Programm erweitern

Um zur Laufzeit das Kontextmenü zu erzeugen, müssen Sie die entsprechenden Anweisungen an einer geeigneten Stelle hinterlegen. Hierzu wählt man in der Regel Ereignisbehandlungsprozeduren, in die dann die Anweisungen eingefügt werden. In obigem Beispiel soll das Kontextmenü des *Label*-Steuerelements dynamisch erweitert werden. Der Einfachheit halber habe ich den betreffenden Code im *Load*-Ereignis des Formulars untergebracht. Bei realen Anwendungen können Sie die betreffenden Anweisungen in anderen Ereignisbehandlungsroutinen hinterlegen (z.B. im *Popup*-Ereignis etc.). Die Prozedur des *Load*-Ereignisses enthält im aktuellen Beispiel folgenden Code, um neue Einträge für ein *ContextMenuStrip*-Element zu erzeugen:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
' Kontextmenü des Labels mit Befehlen Rot, Blau und Gelb
' erweitern und an Ereignisbehandlungsroutinen binden.

' zuerst die Ereignis-Handler für "Rot/Blau" und
' "Gelb" als Objekte definieren
    Dim e1 As New System.EventHandler(AddressOf Label_Ktx_2Clicked)
    Dim e2 As New System.EventHandler(AddressOf Label_Ktx_3Clicked)

' Jetzt fügen wir 2 Einträge zum Kontextmenü hinzu
    Me.LabelContextMenuStrip1.Items.Add("Rot")
    Me.LabelContextMenuStrip1.Items.Add("Blau")

' Kontextmenü-Handler für Befehle "Rot" und "Blau" zuweisen
    AddHandler LabelContextMenuStrip1.Click, e1
```

```
' Hier folgt noch ein dritter Eintrag, der kein Bild aufweist und
' beim Einfügen bereits an einen Kontextmenü-Handler gebunden wird.
Me.LabelContextMenuStrip1.Items.Add("Gelb", Nothing, e2)
End Sub
```

Das *LabelContextMenuStrip1.Items*-Objekt enthält die Auflistung aller Kontextmenüeinträge als Objekte. Über die *Add*-Methode des *LabelContextMenuStrip1.Items*-Objekts lässt sich ein Kontextmenüeintrag zum Menü hinzuzufügen. Die Methode kann dabei mit variablen Parametern aufgerufen werden. Die Einträge *Rot* und *Blau* werden definiert, indem lediglich der Name des Befehls als Argument übergeben wird.

Ein Problem stellt noch die Anbindung dieser dynamisch erzeugten Menüeinträge an die Ereignisbehandlungsroutinen dar. In obigem Listing werden zwei Varianten benutzt. Die beiden Ereignishandler *e1* und *e1* werden als Objekte vereinbart:

```
Dim e1 As New System.EventHandler(AddressOf Label_Ktx_2Clicked)
Dim e2 As New System.EventHandler(AddressOf Label_Ktx_3Clicked)
```

Die Anweisung

```
AddHandler LabelContextMenuStrip1.Click, e1
```

bindet dann das *Click*-Ereignis des Kontextmenüs an den im Objekt *e1* definierten Ereignishandler (hier *Label_Ktx_2Clicked*). Allerdings wird dieses Ereignis bei jedem Mausklick auf einen Kontextmenübefehl aufgerufen. In der aktuellen Implementierung prüft die Ereignisbehandlungsroutine daher, ob der gewünschte Befehl gerade selektiert wurde (ist beim Anklicken der Fall) und reagiert entsprechend:

```
Private Sub Label_Ktx_2Clicked(ByVal sender As Object, _
    ByVal e As System.EventArgs)
' Diese Routine wird aufgerufen, wenn der Benutzer einen
' der dynamisch angelegten Kontextmenüeinträge anklickt.
' Prüfe, welcher dynamische Befehl im Kontextmenü des
' Labels angeklickt wurde
If Me.LabelContextMenuStrip1.Items(1).Selected Then _
    MessageBox.Show("Rot angeklickt", "Label")
If Me.LabelContextMenuStrip1.Items(2).Selected Then _
    MessageBox.Show("Blau angeklickt", "Label")
End Sub
```

Hier wird mit einer *If*-Abfrage einfach geprüft, ob der betreffende Menüpunkt selektiert war. Um einen dynamisch erzeugten Kontextmenüeintrag direkt beim Einfügen in das Menü einer Ereignisbehandlungsroutine zuzuweisen, können Sie den betreffenden Ereignishandler direkt innerhalb der *Add*-Methode angeben:

```
Me.LabelContextMenuStrip1.Items.Add("Gelb", Nothing, e2)
```

Der erste Parameter des Aufrufs enthält den Namen des Befehls, während der Ereignishandler im dritten Parameter übergeben wird. Der zweite Parameter dient zur Aufnahme eines Bildobjekts, welches im Kontextmenü einzublenden ist. Hier wurde kein Bild zugeordnet, der Parameter wird auf *Nothing* gesetzt. Falls Sie im Programm auf die Einträge des Kontextmenüs zugreifen möchten, können Sie dies über die *Items*-Auflistung tun. Mit

```
Me.LabelContextMenuStrip1.Items(1).Text = "Befehl1"
```

würde beispielsweise der zweite Kontextmenüeintrag des Objekts *Me.LabelContextMenuStrip1* adressiert. Die Anweisung passt den Befehlsnamen über die Eigenschaft *Text* an.

Hinweis

Sie finden ein komplettes Projektbeispiel im Ordner `\Beisp\Kap11\Kontextmenu`. Laden Sie die *.sln*-Datei in der Entwicklungsumgebung, um die Details zu studieren. Das Beispiel enthält im Modul *Form1.vb* ein Kontextmenü, welches per *ContextMenuStrip*-Steuerelement implementiert wurde. Die Kontextmenüs im Modul *Form2.vb* wurden dagegen mit dem älteren *ContextMenu*-Steuerelement realisiert. Wird das Projekt ausgeführt, erscheint das Formular *Start.vb*, in dem sich über Schaltflächen wählen lässt, ob die neue oder die alte Variante anzuzeigen ist. Für das alte *ContextMenu*-Control ist eine geringfügig modifizierte Befehlsfolge erforderlich, um die Menüeinträge zum Steuerelement hinzuzufügen. Sie finden im Projektordner `\Beisp\Kap11\KontextmenuCode` die Datei *KontextMenu.vb*. Deren Quellcode zeigt, wie sich ein Kontextmenü mittels des alten *ContextMenu*-Steuerelements aufbauen lässt.

11.3 Symbolleisten verwenden

Neben Menüleisten bieten Symbolleisten mit ihren Schaltflächen und ggf. weiteren Steuerelementen die zweite Möglichkeit zum Abrufen von Anwendungsfunktionen. Das .NET Framework stellt in der Formalklasse entsprechende Unterklassen zur Erzeugung von Symbolleisten bereit. Die neue .NET-2.0 *ToolStrip*-Klasse erzeugt Symbolleisten im Stil moderner Windows-Anwendungen, die im Design zu den Menüleisten der *MenuStrip*-Klasse passen. Sie können bei Leisten aus der *ToolStrip*-Klasse Schaltflächen, Menüs, Textfelder, Kombinationsfelder etc. hinzufügen. Aus Kompatibilitätsgründen wird aber auch die alte *ToolBar*-Klasse des .NET 1.x Framework noch unterstützt. Um dieses Steuerelement im Formulardesigner nutzen zu können, müssen Sie das Element manuell in der Toolbox einfügen (siehe Kapitelanfang).

11.3.1 Symboleleisten zum Formular hinzufügen

Das Hinzufügen einer Symbolleiste zu einem Formular ist in der Entwicklungsumgebung mit wenigen Handgriffen erledigt. Sie brauchen im Designer nur das betreffende *ToolStrip*-Steuerelement zum Formular hinzuzufügen und die Schaltflächen bzw. Steuerelemente der Leiste zu definieren. Dies lässt sich alles interaktiv erledigen. Allerdings benötigen Sie etwas Zusatzwissen, um bestimmte Einstellungen (z.B. Schaltflächensymbole) zu setzen oder die Schaltflächen an Befehle anzubinden. Zunächst möchte ich Ihnen die Schritte zum Einfügen von Symbolleisten und Steuerelementen in eine Anwendung erläutern.

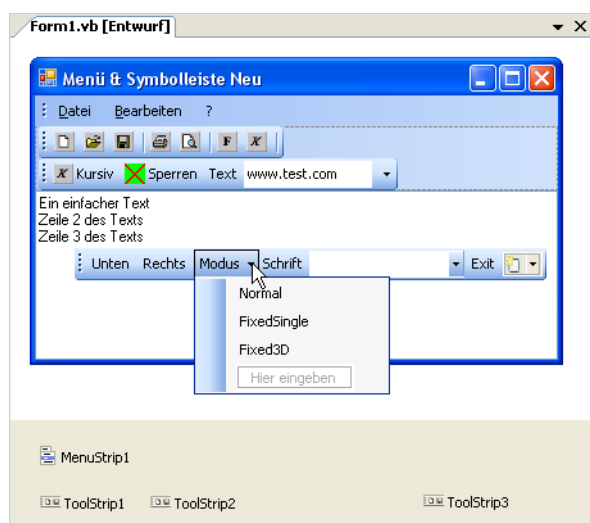


Abbildung 11.12: Beispielfenster mit drei Symbolleisten

In dem in Abbildung 11.12 gezeigten Fenster sind drei Symbolleisten mit verschiedenen Schaltflächen und Steuerelementen enthalten. Klicken Sie auf die Schaltflächen der obersten Symbolleiste (Symbolleiste 1), erscheint jeweils ein einfacher Dialog mit dem Namen der Schaltfläche. Die rechte Schaltfläche dieser Leiste ist als sogenannte *ToolStripDropDownButton*-Schaltfläche realisiert und zeigt ein Menü zur Auswahl einer Option. Je nach Auswahl ändert sich das in der Schaltfläche angezeigte Symbol. Die zweite Symbolleiste (Symbolleiste 2) mit den beiden Schaltflächen *Kursiv* und *Sperren* demonstriert, wie sich Text und Symbole in Schaltflächen verwenden lassen. Über die Schaltfläche *Sperren* lässt sich die Schaltfläche *Kursiv* wahlweise ausblenden oder deaktivieren. Ein bei Auswahl der Schaltfläche angezeigtes Dialogfeld erlaubt das Ausblenden über die *Ja*-Schaltfläche, während die *Nein*-Schaltfläche die Schaltfläche *Kursiv* schrittweise deaktiviert und wieder freigibt. Das Textfeld *Text* besteht eigentlich aus zwei Elementen, einem Label und einem Textfeld. Verliert das Textfeld den Fokus, wird die eingetippte URL in einem Zusatzdialog angezeigt. Beide Symbolleisten sind so im Formular eingefügt, dass der Benutzer diese zur Laufzeit im oberen Bereich des Formulars verschieben kann (nebeneinander oder untereinander anordnen). Die dritte Symbolleiste (Symbolleiste 3) wird zur Laufzeit am unteren Fensterrand andockt. Der Benutzer kann die Leiste aber über die beiden Schaltflächen *Unten* und *Rechts* wahlweise am unteren oder rechten

Fensterrand verankern. Die als *ToolStripDropDownButton* realisierte dritte Schaltfläche *Modus* öffnet ebenfalls ein Menü, über welches sich Befehle anwählen lassen. Je nach angewähltem Befehl ändert sich der in der Schaltfläche angezeigte Text. Das Kombinationsfeld erlaubt einen Text einzugeben oder einen Eintrag aus der Liste zu wählen. Bei Änderungen des Werts wird dieser über ein Dialogfeld eingeblendet. Die mit *Exit* bezeichnete Schaltfläche beendet die Anwendung.

Zur Vorbereitung des Beispiels können Sie ein bereits bestehende Projekt verwenden. Oder Sie legen ein neues Projekt als Windows-Anwendung in der Entwicklungsumgebung an. Sie sollten das Formular mit einer Menüleiste am oberen Rand versehen. Die Techniken zur Gestaltung einer Menüleiste wurde in den vorhergehenden Abschnitten beschrieben. Nun gilt es, die Symbolleisten im Formular einzufügen und danach mit Steuerelementen (Schaltflächen etc.) zu ergänzen.

Im Prinzip reicht es, das betreffende Steuerelement für die Symbolleiste aus der Toolbox in den Formularentwurf zu übernehmen und dann die Steuerelemente der Leiste zu ergänzen. Allerdings sollten Sie noch eine Besonderheit des *ToolStrip*-Steuerelements kennen. Wird das Steuerelement nur im Formular eingefügt, kann der Benutzer dieses standardmäßig nicht mehr per Maus an verschiedene Positionen ziehen. Daher empfiehlt es sich, die Symbolleisten ggf. in einen *ToolStripContainer* einzubetten. Dies erlaubt dem Benutzer die Symbolleiste zur Laufzeit innerhalb des Containers zu positionieren (Leisten untereinander oder nebeneinander). Im aktuellen Beispiel sollen die beiden am oberen Bildrand sichtbaren Schaltflächen in einem *ToolStripContainer* hinterlegt werden, während die dritte Symbolleiste direkt dem Formular zugeordnet ist. Nachfolgend finden Sie die Schritte, um die drei Symbolleisten zum Formular hinzuzufügen:

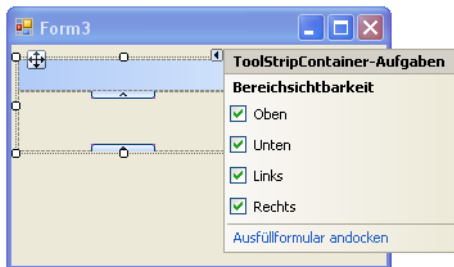


Abbildung 11.13: Einfügen eines *ToolStripContainer*-Steuerelements

1. Fügen Sie aus der Toolbox ein *ToolStripContainer*-Steuerelement im Formular hinzu (Abbildung 11.13). Legen Sie ggf. über die Kontrollkästchen des eingeblendeten Menüs die Bereichssichtbarkeit fest und ob das Ausfüllformular anzudocken ist.
2. Setzen Sie die *Dock*-Eigenschaft des Steuerelements auf den gewünschten Wert (z.B. auf *Fill*, wenn der Container das gesamte Formular ausfüllen soll). Dann lassen sich die im Container eingefügten Symbolleisten zur Laufzeit im Container frei verschieben.

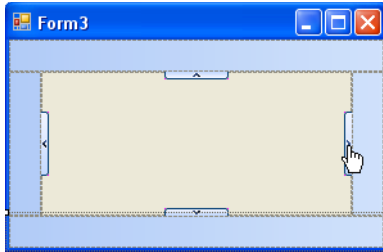


Abbildung 11.14: Anzeige der Bereiche des ToolStripContainers

Bei Bedarf können Sie die Bereiche am Rand des Containers, an denen die Symbolleisten andockt werden können, ein- oder ausblenden (Abbildung 11.14). Hierzu brauchen Sie nur auf die Klickbereiche des Containers zu klicken. Die Bereiche werden als blaue Streifen eingeblendet.

3. Stellen Sie sicher, dass der *ToolStripContainer* als Steuerelement im Formular angewählt ist und fügen Sie jetzt aus der Toolbox die Anzahl der gewünschten *ToolStrip*-Steuerelemente für die Symbolleisten hinzu. Im aktuellen Beispiel sind dies zwei Symbolleisten, die am oberen Rand des Formulars unterhalb der Menüleiste sichtbar sind.
4. Für die dritte Symbolleiste wählen Sie das Formular per Mausklick an und fügen danach ebenfalls ein *ToolStrip*-Steuerelement im Formular ein. Durch die vorherige Auswahl des Formulars wird das Steuerelement diesem und nicht dem *ToolStripContainer* zugeordnet (im Beispiel soll das Verhalten der Symbolleisten mit und ohne *ToolStripContainer* demonstriert werden). Die dritte Symbolleiste ist mitten im Formularbereich sichtbar.
5. Fügen Sie jetzt noch ein mehrzeiliges Textfeld im Formular ein und setzen Sie dessen *Dock*-Eigenschaft auf *Fill*. Klicken Sie das *Textfeld*-Steuerelement mit der rechten Maustaste an und wählen Sie im Kontextmenü den Befehl *In den Hintergrund*. Dies ist wichtig, damit die dritte Symbolleiste zur Laufzeit sichtbar wird.
6. Anschließend können Sie den jeweiligen Symbolleisten Schaltflächen und andere Steuerelemente zuweisen (siehe folgender Abschnitt).
7. Sollen Schaltflächensymbole zur Laufzeit angepasst werden, empfiehlt sich zudem die Aufnahme eines *ImageList*-Steuerelements, aus dem die Icons zugewiesen werden können. Das Steuerelement wird als nicht sichtbare Komponente im Designer unterhalb des Formulars eingeblendet. Fügen Sie anschließend die für die Symbolleiste benötigten Bilddateien zur *ImageList* hinzu. Die betreffenden Schritte haben Sie bereits in den vorhergehenden Kapiteln kennen gelernt. Falls Sie nur Schaltflächen mit Texten benutzen, kann auf den Schritt zur Gestaltung der *ImageList* verzichtet werden.

Mit diesen Schritten haben Sie das Formular mit drei Symbolleisten, einer Menüleiste und einem Textfeld ausgestattet. Wenn Sie das Beispiel ausführen, lassen sich die beiden im Container hinterlegten Symbolleisten zur Laufzeit vom Benutzer per Maus zu den Docking-Rändern (oben, oben nebeneinander, links, rechts, unten) ziehen. Die dritte, frei schwebend angeordnete, Symbolleiste ist dagegen nicht verschiebbar (diese soll später per Code an einem Rand andockt werden).

Tipp

Haben Sie irrtümlich eine Symbolleiste in einem *ToolStripContainer* hinterlegt und möchten Sie diese aus dem Container in das Formular verschieben? Klicken Sie auf die Symbolleiste und übertragen Sie diese durch die Tastenkombination **Strg** + **X** in die Zwischenablage. Danach lässt sich das Formular auswählen und die Symbolleiste mit **Strg** + **V** aus der Zwischenablage einfügen. Auf die gleiche Weise können Sie eine Symbolleiste aus einem Formular in einen *ToolStripContainer* übertragen. Beachten Sie aber, dass Sie nach diesem Schritt ggf. die Ereignisbehandlungsroutinen für die Elemente der Symbolleiste neu erstellen müssen.

Die Symbolleiste lässt sich in verschiedenen »Farben« (Office-Blau, Windows-Grau) darstellen. Die Farbe zur Darstellung des Leistenhintergrunds passen Sie über die Eigenschaft *RenderMode* an. Der Wert *System* verwendet die Farben des (Windows-) Systems, während *Professional* auf den Office-Stil setzt.

Hinzufügen von Schaltflächen und Elementen zur Symbolleiste

Das Hinzufügen von Elementen (Schaltflächen, Textfelder etc.) zur Symbolleiste ist in .NET 2.0 sehr einfach. Das Ganze lässt sich interaktiv im Designer vornehmen.

1. Klicken Sie im Fenster *Ansicht-Designer* auf die zu bearbeitende Symbolleiste. Der Designer blendet dann eine stilisierte Schaltfläche *ToolStripButton* hinzufügen am rechten Rand der Leiste ein (Abbildung 11.15).
2. Öffnen Sie das Menü der betreffenden Schaltfläche und wählen Sie einen Befehl, um das entsprechende Element zur Leiste hinzuzufügen.
3. Anschließend passen Sie ggf. die Eigenschaften des neu hinzugefügten Elements gemäß den Anforderungen an (siehe folgende Erläuterungen).
4. Wiederholen Sie diese Schritte, um alle in der Symbolleiste benötigten Steuerelemente (Schaltflächen, Textfelder etc.) aufzunehmen.
5. Passen Sie bei Bedarf die Reihenfolge der Elemente an, indem Sie diese per Mausklick markieren und dann bei gedrückter linker Maustaste vertikal innerhalb der Leiste verschieben. Sobald Sie die linke Maustaste loslassen, wird das Element an der aktuellen Position eingefügt.

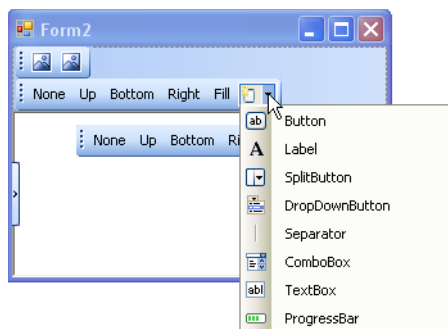


Abbildung 11.15: Einfügen von Elementen in die Symbolleiste

Um eine optische Trennung zwischen zwei Symbolleistenelementen zu erzwingen, fügen Sie ein *Separator*-Element ein. Der Befehl *Button* erlaubt Ihnen Schaltflächen (mit Text und/oder Bild) zur Leiste hinzuzufügen. Mit *Label* lässt sich ein einfacher Text ablegen (z.B. hilfreich zur Beschriftung einer *ComboBox* oder einer *Textbox*). Der Labeltext wird über die *Text*-Eigenschaft des betreffenden Elements definiert. Eine *Textbox* dient zur Eingabe von Texten. Menüs lassen sich mit den Elementen *SplitButton* und *DropDownButton* gestalten. Sie können das Element nach dem Einfügen per Maus anklicken. Sobald sich das Menü öffnet, lassen sich die Menübefehle direkt eintragen. Zudem können Sie über den Kontextmenübefehl *Einfügen* Separatoren, Textfelder und Kombinationsfelder einbringen. Alternativ lässt sich im Eigenschaftsfenster die Schaltfläche der Eigenschaft *DropDownItems* wählen. Im Elementauflistungs-Editor des betreffenden Steuerelements lassen sich dann Menübefehle, Separatoren, Textfelder, Kombinationsfelder etc. zum DropDown-Menü hinzufügen.

Ein irrtümlich eingefügtes Element können Sie mit der rechten Maustaste anklicken und über den Kontextmenübefehl *Löschen* wieder entfernen. Das Kontextmenü enthält zudem den Befehl *Konvertieren in*, über dessen Untermenübefehle Sie das Element in andere Steuerelemente konvertieren können.

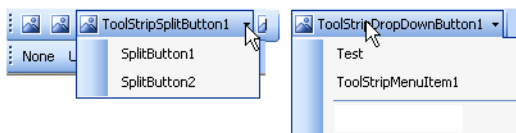


Abbildung 11.16: SplitButton und DropDownButton in einer Symbolleiste

Hinweis

Der Unterschied zwischen *SplitButton* und *DropDownButton* wird erst zur Laufzeit sichtbar. Beim *SplitButton* hat der Benutzer zur Laufzeit die Möglichkeit, die Schaltfläche des betreffenden Elements anzuklicken, um den Standardbefehl abzurufen. Zusätzlich kann er über die *Split*-Schaltfläche des Elements das Menü öffnen und dann einen Befehl wählen (Abbildung 11.16, links). Bei der Auswahl des *DropDownButton*-Elements öffnet sich dagegen immer ein Menü zur Auswahl der Befehle (Abbildung 11.16, rechts).

Übersetzen und starten Sie das Projekt, sind die Schaltflächen und Elemente der Symbolleiste bereits wählbar. Eventuell definierte ToolTipp-Infos lassen sich abrufen. Wie Sie Befehle an die Schaltflächen und Steuerelemente anbinden, erfahren Sie in den folgenden Abschnitten. Vorher möchte ich noch einige Bemerkungen zu den Eigenschaften bzw. zur Gestaltung der einzelnen Schaltflächen/Symbolleistenelemente machen.

Text und/oder Symbol für Symbolleistenelemente vorgeben

Schaltflächen, *SplitButton*- und *DropDownButton*-Elemente können wahlweise durch einen Text, durch ein Bild oder sowohl durch einen Text als auch ein Symbol dargestellt werden. Die Auswahl der Darstellungsmodi erfolgt am einfachsten über den Kontextmenübefehl *DisplayStyle* des betreffenden Elements. Der Befehl wirkt auf die gleichna-

mige Eigenschaft des Steuerelements. Sobald Sie diesen Kontextmenübefehl anwählen, erscheint ein Untermenü mit den verfügbaren Optionen (Abbildung 11.17).

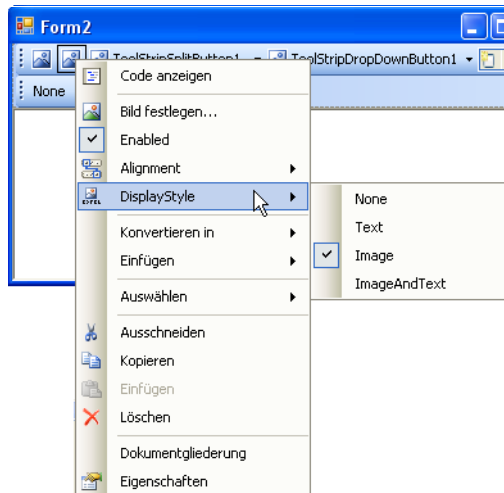


Abbildung 11.17: Anzeigestil von Elementen in der Symbolleiste vorgeben

Der Befehl *None* unterdrückt die Darstellung eines Symbols sowie des Texts. Mit dem Befehl *Text* erreichen Sie, dass das betreffende Element mit einem Text in der Symbolleiste dargestellt wird. Dies ist bei Textschaltflächen oder bei Menüs (*SplitButton* oder *DropDownButton*) häufig der Fall. Der Befehl *Image* bewirkt die Darstellung des Steuerelements in der Symbolleiste mit einem Symbol. Möchten Sie dem Element sowohl einen Text als auch ein Bild zuweisen, müssen Sie den Wert für *DisplayStyle* auf *ImageAndText* setzen.

Erzwingt der *DisplayStyle* eine Textanzeige, blendet der Designer standardmäßig den Namen des Steuerelements in der Symbolleiste ein. Sie können den anzuzeigenden Text aber über die *Text*-Eigenschaft des betreffenden Steuerelements anpassen.

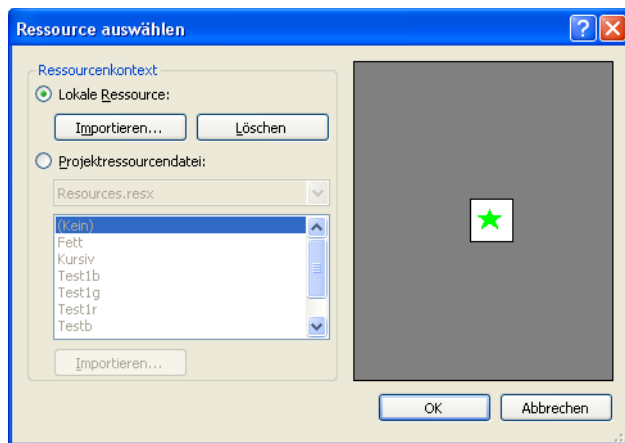


Abbildung 11.18: Einbinden von Symbolen als Ressourcen

Wurde ein *DisplayStyle* gewählt, der die Anzeige eines Symbols beinhaltet, müssen Sie dem Element ein entsprechendes Bild zuweisen. Hierzu klicken Sie das Element mit der rechten Maustaste an und wählen den Kontextmenübefehl *Bild festlegen* (Abbildung 11.17). Sobald das Dialogfeld *Ressource auswählen* erscheint (Abbildung 11.18), können Sie die Option *Lokale Ressource* markieren und die Schaltfläche *Importieren* wählen. Dann lässt sich eine Symboldatei direkt von der Festplatte aufnehmen. Alternativ lässt sich die Option *Projektressourcendatei* wählen. Dann können Sie auf die im Projekt bereits importierten Ressourcen zurückgreifen, um Bilder zuzuweisen.

Tipp

Die Schaltfläche *Importieren* blendet einen *Öffnen*-Dialog ein, der nur Symboldateien mit den Grafikformaten *.bmp*, *.gif*, *.png*, *.wmf* und *.jpeg* aufweist. Sie können aber auch *.ico*-Dateien importieren. Stellen Sie hierzu im Dialogfeld *Öffnen* den Dateityp auf »Alle Dateien (*.*)«, damit die Dateien des betreffenden Dateityps mit aufgelistet werden. Beim Zuweisen von Symbolen zu den Schaltflächen sollten Sie darauf achten, dass diese bereits in den Bildabmessungen der Größe der Schaltflächen entsprechen. Sind die Bilddateien zu klein, sieht dies optisch schlecht aus. Weiterhin empfiehlt es sich, bei den Symbolhintergründen mit einer transparenten Farbe zu arbeiten (lässt sich z.B. in Visual Studio 2005 beim Bearbeiten der Bitmapdateien angeben). Dies verhindert »Ränder« in den Schaltflächen, die durch unterschiedliche Farben zwischen Symbolhintergrund und Schaltfläche entstehen.

Bemerkung zu weiteren Eigenschaften der Steuerelemente

Sie können die Schaltflächen und Elemente der Symbolleiste per Maus im Fenster des Ansicht-Designers anklicken. Dann werden deren Eigenschaften im Eigenschaftensfenster aufgelistet. Über diese Eigenschaften lässt sich das Aussehen und Verhalten des Steuerelements beeinflussen.

- Die Eigenschaften *IMAGEALIGN* und *TextAlign* legen fest, wie der Text oder das Bild im Element auszurichten ist (linksbündig, zentriert etc.). Bei Anwahl der Eigenschaft erscheint eine Palette mit den verfügbaren Optionen.
- Das Steuerelement lässt sich über die Eigenschaft *Visible* ein-/ausblenden und über die Eigenschaft *Enabled* sperren. Standardmäßig sind die Werte mit *True* freigegeben.
- Besitzt die Eigenschaft *CheckOnClick* den Wert *true*, wird die das Element (z.B. eine Schaltfläche) beim Anklicken markiert dargestellt. Der Status lässt sich dann über die Eigenschaften *Checked* und *CheckState* auswerten.
- Ist dem Element ein Symbol zugewiesen, findet sich dieses (als Bitmap) in der Eigenschaft *Image*. Über die Eigenschaft *ImageScaling* lässt sich vorgeben, ob das Symbol die gesamte Fläche des betreffenden Symbolleistenelements ausfüllen soll.
- Einen eventuell benötigten *ToolTipText*-Text tragen Sie in der gleichnamigen Eigenschaft ein. Der Text wird beim Zeigen auf das Element als ToolTip eingeblendet.

Weitere Hinweise zu den anderen Eigenschaften der Elemente finden Sie in der Fußzeile des Eigenschaftensfensters, wenn Sie die Eigenschaft anwählen.

11.3.2 Code zur Anbindung der Elemente an Funktionen

Nachdem Sie Schaltflächen oder andere Steuerelemente zu den Symbolleisten hinzugefügt haben, muss die gewünschte Funktionalität über Ereignisbehandlungsprozeduren implementiert werden. Abbildung 11.19 zeigt das Fenster des Beispiels mit drei Symbolleisten und deren Elementen. Beim Anklicken der oberen Schaltflächen wird ein Dialogfeld mit einer Meldung geöffnet. Die Schaltfläche *Sperren* der zweiten Symbolleiste erlaubt die linke Schaltfläche *Kursiv* einzublenden oder zu sperren. Im Textfeld *Text* lässt sich ein beliebiger Text eingeben. Und über die Schaltflächen der untersten Leiste lässt sich diese am unteren oder rechten Rand andocken.

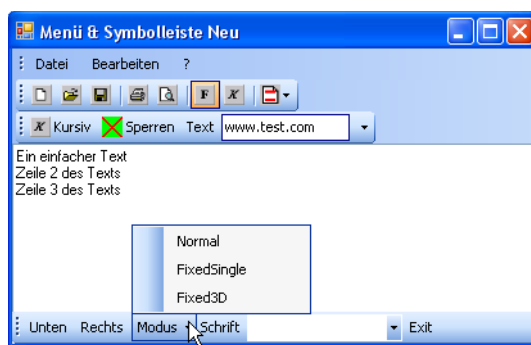


Abbildung 11.19: Formularentwurf mit Menü und eingefügten Symbolleisten

Nachfolgend finden Sie einige Hinweise zur Implementierung verschiedener Funktionen dieses Beispiels.

Auf das Anklicken einer Schaltfläche reagieren

Schaltflächen werden in der Regel Befehle zugeordnet, die beim Anklicken auszuführen sind. Klickt der Benutzer auf die Schaltfläche einer Symbolleiste, löst dies im .NET Framework aber ein *Click*-Ereignis für die komplette Symbolleiste aus. Wählen Sie im Fenster *Ansicht-Designern* das Symbol einer Schaltfläche per Doppelklick an, gelangen Sie direkt in die *Click*-Ereignisbehandlungsroutine der betreffenden Schaltfläche (die Entwicklungsumgebung generiert direkt den Prozedurrumpf der Ereignisbehandlungsroutine). Sie können dort die Anweisungen zur Implementierung der gewünschten Funktion hinterlegen. Bei den meisten Schaltflächen der Symbolleisten des Beispiels öffnet die betreffende Ereignisbehandlungsroutine nur ein Dialogfeld mittels der *MessageBox.Show()*-Methode. Hier sehen Sie den Code der *Click*-Ereignisbehandlungsroutine für die Schaltfläche *Neu* der obersten Symbolleiste:

```
Private Sub ToolStripButton1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ToolStripButton1.Click
    ' Neu-Schaltfläche angeklickt
    MessageBox.Show("Neu-Schaltfläche")
End Sub
```

Die mit dem Buchstaben *F* gekennzeichnete Schaltfläche der oberen Symbolleiste wird bei jedem Mausklick zwischen markiert und unmarkiert umgeschaltet. Hierzu wurde der Wert der Eigenschaft *CheckOnClick* auf *true* gesetzt. Alternativ könnte dieser Status auch per Code in einer Ereignisbehandlungsroutine über die Anweisung

```
Me.ToolStripButton6.Checked = true
```

gesetzt werden. *ToolStripButton6* ist dabei der Objektname der Schaltfläche, während *Checked* die betreffende Eigenschaft ist, die sich auf *true* oder *false* setzen lässt. Der Status der Schaltfläche lässt sich über die Anweisungen

```
Me.ToolStripButton6.Checked  
Me.ToolStripButton6.CheckState
```

abfragen. Die Eigenschaft *Checked* liefert *true* oder *false*, während *CheckState* die Werte *checked*, *unchecked* oder *indeterminate* (abhängig von den Einstellungen) zurückgibt. Die Ereignisbehandlungsroutine der betreffenden Schaltfläche wertet im aktuellen Beispiel diese Stati aus und zeigt das Ergebnis in einem Dialogfeld an.

Der folgende Codeausschnitt zeigt die *Click*-Ereignisbehandlungsroutine für die Schaltfläche *Sperren* der zweiten Symbolleiste.

```
Private Sub ToolStripButton9_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles ToolStripButton9.Click  
    ' Schaltfläche Sperren Leiste 2  
    Dim tmp As DialogResult  
    With Me  
        tmp = MessageBox.Show("Ausblenden = Ja, Sperren = Nein", _  
            "Schaltfläche Kursiv anpassen", _  
            MessageBoxButtons.YesNo, _  
            MessageBoxIcon.Question)  
        If tmp = System.Windows.Forms.DialogResult.Yes Then  
            .ToolStripButton8.Visible = False ' ausblenden  
        Else  
            .ToolStripButton8.Visible = True ' anzeigen  
            ' Schaltfläche wahlweise sperren/zulassen  
            If .ToolStripButton8.Enabled Then  
                .ToolStripButton8.Enabled = False  
            Else  
                .ToolStripButton8.Enabled = True  
            End If  
        End If  
    End With  
End Sub
```

Beim Anklicken fragt ein Dialogfeld, ob die Schaltfläche *Kursiv* der Symbolleiste auszu-blenden oder zu sperren ist. Je nach dem von *MessageBox.Show* zurückgelieferten Wert werden dann die Eigenschaften *Enabled* und *Visible* der Schaltfläche gesetzt.

Anpassen des Symbols einer Schaltfläche

Die oberste Symbolleiste besitzt eine *DropDownButton*-Schaltfläche mit einem Symbol, die bei Anwahl ein Menü einblendet (Abbildung 11.20). Je nach gewähltem Menübefehl soll sich das in der Schaltfläche angezeigte Symbol ändern.

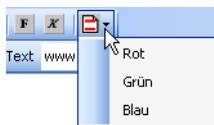


Abbildung 11.20: Menü zum Anpassen des Schaltflächensymbols

Für diesen Zweck muss der *Image*-Eigenschaft des betreffenden Elements eine Bitmap-Struktur zugewiesen werden. Im aktuellen Beispiel finden sich die betreffenden Symbole in einem *ImageList*-Steuerelement. Zur Designzeit wurde daher jeder Befehl im Menü der *DropDownButton*-Schaltfläche per Doppelklick angewählt. In der *Click*-Ereignisbehandlungsroutine des Befehls wird dann das gewünschte Symbol zugewiesen. Das folgende Codefragment zeigt die Ereignisbehandlungsroutine des Befehls *Rot*:

```
Private Sub RotToolStripMenuItem_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles RotToolStripMenuItem.Click
    ' Rot im Menü gewählt - Image DropDown-Button ändern
    Me.ToolStripDropDownButton2.Image = Me.ImageList1.Images.Item(12)
End Sub
```

Es wird einfach ein Bild aus der *Item*-Auflistung der *Images*-Eigenschaft der *ImageList* gelesen und der betreffenden Eigenschaft zugewiesen. Die laufenden Indizes der Bilder können Sie im Bildauflistungs-Editor ermitteln.

Schaltflächentext dynamisch anpassen

Die dritte Symbolleiste des Beispiels enthält das *ToolStripDropDown*-Element *Modus*, welches ein Menü öffnet. Je nach dem im Menü gewählten Befehl soll der Text *Modus* in *Normal*, *Fixed3D* etc. geändert werden.

```
Private Sub NormalToolStripMenuItem_Click_1( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles NormalToolStripMenuItem.Click
    ' DropDownMenü = Normal in Leiste 3
    Me.ToolStripDropDownButton1.Text = "Normal"
```



```

    MessageBox.Show("Normal gewählt")
End Sub

```

Hierzu wird der anzuzeigende Text in der *Click*-Ereignisbehandlungsroutine der Eigenschaft *Text* zugewiesen. Hier sehen Sie die Ereignisbehandlungsroutine für den Befehl *Normal*, die zusätzlich noch ein Meldungsfeld einblendet, um dem Benutzer eine Rückmeldung bei Anwahl des Befehls zu geben.

Andocken der dritten Symbolleiste

Die dritte Symbolleiste des Beispiels wurde bewusst nicht in einen *ToolStripContainer* eingebaut, sondern direkt dem Formular zugewiesen. Damit lässt sich diese Symbolleiste direkt über die *Dock*-Eigenschaft an den Seiten des Formulars andocken. Die Schaltflächen *Unten* und *Rechts* der dritten Symbolleiste besitzen folgenden Code in der *Click*-Ereignisbehandlungsroutine:

```

Private Sub ToolStripButton10_Click1( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ToolStripButton10.Click
    ' Schaltfläche "Rechts"
    Me.ToolStrip3.Dock = DockStyle.Right
End Sub

Private Sub ToolStripButton11_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ToolStripButton11.Click
    ' Schaltfläche "Unten"
    Me.ToolStrip3.Dock = DockStyle.Bottom
End Sub

```

Je nach aufgerufener Ereignisbehandlungsroutine wird die *Dock*-Eigenschaft auf *DockStyle.Right* oder *DockStyle.Bottom* gesetzt. Über *DockStyle.None* ließe sich die Symbolleiste wieder im Formular schwebend anordnen, während *DockStyle.Fill* die Leiste auf die Größe des freien Bereichs erweitert.

Hinweis

Die beiden oberen, im Container hinterlegten, Schaltflächen können vom Benutzer per Maus innerhalb dieses Containers verschoben werden. Dies ist eine Eigenschaft des *ToolStripContainers*. Setzen Sie die Eigenschaft *AllowItemReorder* der Symbolleiste auf *True*, kann der Benutzer zur Laufzeit die Anordnung der Schaltflächen bzw. Elemente bei gedrückter **[Alt]**-Taste ändern. Als Erweiterung des Beispiels ließen sich nun Funktionen zur Drag&Drop-Behandlung der Symbolleiste implementieren. Die schwebend angeordnete dritte Symbolleiste ließe sich dann per Maus zu beliebigen Positionen im Fenster ziehen.

Eingaben in Textfeldern und Kombinationsfeldern auswerten

Das Beispiel enthält in der zweiten Symbolleiste ein Textfeld *Text*, in dem der Benutzer Eingaben vornehmen kann. Das Kombinationsfeld *Schrift* in der dritten, unteren Symbolleiste erlaubt dem Benutzer sowohl einen in der Liste vorgegebenen Eintrag für die Schriftart zu wählen als auch einen freien Text einzutippen. Zur Auswertung von Benutzereingaben in solchen Steuerelementen gibt es verschiedene Ansätze. Wählt der Benutzer einen Listeneintrag im *ToolStripComboBox*-Steuerelement, löst dies ein *SelectedIndexChanged*-Ereignis aus. Die folgende Ereignisbehandlungsroutine zeigt den Code, mit dem die Benutzerauswahl angezeigt wird:

```
Private Sub ToolStripComboBox1_SelectedIndexChanged( _
    ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles ToolStripComboBox1.SelectedIndexChanged
    ' Index in Kombinationsfeld der Leiste geändert
    MessageBox.Show("Listenfeld Schrift: " & _
        Me.ToolStripComboBox1.Text)
End Sub
```

Im aktuellen Fall gibt die Ereignisbehandlungsroutine nur den vom Benutzer ausgewählten Wert in einem Dialogfeld aus. Der Wert kann direkt über die *Text*-Eigenschaft des *ToolStripComboBox*-Steuerelements zurückgelesen werden.

Schwieriger wird es aber, sobald der Benutzer Eingaben in einem Textfeld (*TextBox*) oder in einem *ToolStripComboBox*-Steuerelement vornehmen kann. Sie könnten dann zwar auf jede Änderung des Werts im betreffenden Steuerelement reagieren und eine *TextChanged*-Ereignisbehandlungsroutine aufrufen. Das Problem besteht aber darin, dass Sie dann nicht wissen, ob die Benutzereingabe abgeschlossen ist. Die andere Möglichkeit wäre, auf den Verlust des Fokus über eine *LostFocus*-Ereignisbehandlungsroutine zu reagieren. Dies setzt aber voraus, dass der Benutzer ein anderes Steuerelement in der Anwendung anklickt (um den Fokus zu wechseln). Man könnte auch, ähnlich wie bei der *Adresse*-Symbolleiste des Internet Explorer, eine eigene Schaltfläche (*Wechseln*) zur Annahme des Werts in der Symbolleiste hinterlegen. Im aktuellen Beispiel wurde das Problem eleganter gelöst – die Anwendung wertet die Benutzereingabe erst beim Drücken der -Taste aus. Das Erkennen der -Taste macht jedoch einen kleinen Kniff erforderlich. Das Beispiel benutzt die *KeyPressed*-Eigenschaft des betreffenden Steuerelements, um auf Tasteneingaben zu reagieren. Innerhalb der Ereignisbehandlungsroutine lässt sich dann der Tastencode auswerten. Wird die -Taste erkannt, ist der Wert des betreffenden Elements anzuzeigen. Das folgende Codefragment zeigt die Ereignisbehandlungsroutine des Kombinationsfelds aus dem Beispielformular:

```
Private Sub ToolStripComboBox1_KeyUp( _
    ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyEventArgs) _
    Handles ToolStripComboBox1.KeyUp
    ' Benutzereingabe in Textfeld Leiste 3 mit Enter-Taste abgeschlossen?
```

Listing 11.1: Erkennen der -Taste in einer Ereignisbehandlungsroutine

```

If e.KeyCode = Keys.Enter Then ' Bei Enter Text anzeigen
    MessageBox.Show("Listenfeld Schrift: " & _
        Me.ToolStripComboBox1.Text)
End If
End Sub

```

Listing 11.1: Erkennen der -Taste in einer Ereignisbehandlungsroutine (Forts.)

Beim Aufruf wird der Routine im Parameter *e* der Code der gedrückten Taste übergeben. Der Code lässt sich über die *KeyCode*-Eigenschaft direkt abfragen. Im aktuellen Beispiel prüft eine *If*-Abfrage, ob die -Taste gedrückt wurde und zeigt dann den aktuellen Wert der *Text*-Eigenschaft des Steuerelements in einem Dialogfeld an.

Symbolleisten ein- oder ausblenden

Ähnlich wie bei anderen Steuerelementen können Sie auch Symbolleisten über die *Visible*-Eigenschaft ein- oder ausblenden. Im aktuellen Beispiel kann der Benutzer für die Menü- und die drei Symbolleisten ein Kontextmenü öffnen. Über die angebotenen Befehle lässt sich jede der drei Symbolleisten wahlweise ein- oder ausblenden (Abbildung 11.21). Die angezeigten Befehle wechseln dabei in Abhängigkeit vom Status der Symbolleiste.

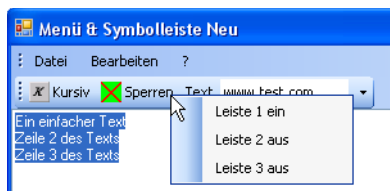


Abbildung 11.21: Symbolleisten ein- oder ausblenden

Zur Implementierung dieser Funktionalität wurde in diesem Beispiel ein kleiner Trick benutzt. Im Formular wurde ein *ContextMenuStrip*-Steuerelement mit den drei in Abbildung 11.21 gezeigten Befehlen hinzugefügt. Danach wurde dieses *ContextMenuStrip*-Steuerelement über die Eigenschaft *ContextMenuStrip* an die Steuerelemente zur Anzeige der Menü- und Symbolleisten gebunden. Dadurch wird das gleiche Kontextmenü für alle Leisten benutzt. Bei Anwahl eines Kontextmenübefehls ist dann nur noch die zugehörige *Click*-Ereignisbehandlungsroutine aufzurufen, die dann den anzuzeigenden Befehl anpasst und die Menüleiste ein- oder ausblendet. Nachfolgend ist der Code für den Befehl *Leiste 1 ein* zu setzen:

```

Private Sub Leiste1AusToolStripMenuItem_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles Leiste1AusToolStripMenuItem.Click
    ' Symbolleiste 1 ein-/ausblenden
    If Me.ToolStrip1.Visible Then
        Me.ToolStrip1.Visible = False
    End If
End Sub

```

```

        Me.Leiste1AusToolStripMenuItem.Text = "Leiste 1 ein"
    Else
        Me.ToolStrip1.Visible = True
        Me.Leiste1AusToolStripMenuItem.Text = "Leiste 1 aus"
    End If
End Sub

```

In der Ereignisbehandlungsroutine wird die *Visible*-Eigenschaft der betreffenden Leiste ausgewertet. Dann wird der Wert der Eigenschaft invertiert und der Kontextmenübefehl über die *Text*-Eigenschaft des betreffenden Menübefehls angepasst.

Hinweis

Sie finden die Dateien des kompletten Projektbeispiels im Ordner `\Beisp\Kap11\SymbolBar` auf der Begleit-CD. Das Element *Form1.vb* enthält das komplette Beispiel samt dem auf den vorherigen Seiten erläuterten Code. Im Element *Form2.vb* ist ein Formular mit einigen Symbolleisten hinterlegt. Dieses Formular dient zur Demonstration des Entwurfs von Symbolleisten. Die untere Symbolleiste enthält dabei verschiedene Schaltflächen, um die möglichen Docking-Varianten einer Leiste am Formularrand zu zeigen. Die *Click*-Ereignisbehandlungsroutinen der Schaltflächen enthalten den Code, um die *Dock*-Eigenschaft der Symbolleiste zu setzen. Um das Formular zu testen, müssen Sie die Projekteigenschaften so anpassen, dass *Form2* als Startobjekt benutzt wird.

11.3.3 Verwenden des alten ToolBar-Steuerelements

Möchten Sie das alte .NET 1.x Control *ToolBar* zum Erzeugen von Symbolleisten einsetzen, gehen Sie ähnlich wie beim *ToolStrip*-Control vor. Fügen Sie das Steuerelement aus der Toolbox im Formulardesign ein und passen Sie dessen Eigenschaften über das Eigenschaftenfenster an.

- Die Eigenschaften *Anchor* und *Dock* definieren, wo die Symbolleiste im umgebenden Steuerelement verankert sein soll. Sie sollten die Leiste über *Anchor* ggf. am oberen, linken Rand verankern. Mit der *Dock*-Eigenschaft lässt sich die Leiste am oberen (Top), unteren (Bottom), linken (Left) oder rechten (Right) Rand des umgebenden Elements (meist des Formulars) andocken. Ist die *Dock*-Eigenschaft ungleich *None*, passt .NET Framework die Leiste bei Größenänderungen des Fensters automatisch an.
- Mit der Eigenschaft *Appearance* bestimmen Sie, wie die Schaltflächen in der Symbolleiste anzuzeigen sind. Der Wert *Normal* legt fest, dass die Schaltflächen optisch hervorgehoben werden. Mit der Einstellung *Flat* lässt sich eine Symbolleiste mit flachen Schaltflächen realisieren. Erst beim Zeigen mit der Maus auf eine Schaltfläche wird diese optisch hervorgehoben.
- Die Eigenschaft *BorderStyle* erlaubt Ihnen über die Werte *None*, *FixedSingle* und *Fixed3D* die Symbolleiste mit oder ohne Rand darzustellen.

- Die Eigenschaft *ButtonSize* ermöglicht Ihnen die Größe der Schaltflächen innerhalb der Symbolleiste zu fixieren. Verzichten Sie auf die Anpassung dieser Werte, legt der Designer die Größe der Schaltflächen automatisch fest. Die Abmessungen der Symbolleiste werden automatisch in der Eigenschaft *Size* geführt.
- Die Anordnung von Texten in Schaltflächen, die ebenfalls ein Symbol enthalten, wird über die Eigenschaft *TextAlign* gesteuert. Sie können den Text rechts vom Symbol (Right) oder unterhalb des Symbols (Underneath) anordnen.
- Die Eigenschaft *Wrappable* lässt sich auf *True* oder *False* setzen. Beim Wert *True* kann .NET Framework den Inhalt der Symbolleiste zur Laufzeit in mehrere Zeilen umbrechen, falls das Fenster schmaler als der benötigte Platz zur Anzeige ist.
- Mit den beiden Eigenschaften *Visible* und *Enabled* steuern Sie, ob eine Symbolleiste sichtbar ist bzw. gesperrt wird.
- Über die Eigenschaft *ImageList* weisen Sie dem Steuerelement den (Objekt-) Namen der ggf. im Formular eingefügten *IMAGELIST* zu. Über dieses *ImageList* kann später den Schaltflächen ein Symbol zugewiesen werden.

Im Eigenschaftsfenster finden Sie weitere Eigenschaften, um das Verhalten der Symbolleiste zu beeinflussen. Klicken Sie notfalls auf den betreffenden Eintrag, um die zugehörige Beschreibung abzurufen.

Achtung

Die Schaltflächen der *ToolBar*-Symbolleiste lassen sich jeweils mit einem Symbol und/oder einem Text versehen. Die Anordnung der Texte wird durch die Eigenschaft *TextAlign* (Right oder Underneath) geregelt. Dies beeinflusst aber die Größe der Schaltflächen, die bei hinterlegten Texten entweder breiter oder höher werden. Entwerfen Sie im Formulardesigner eine Symbolleiste mit solchen gemischten Schaltflächen, wird deren Größe zur Laufzeit automatisch angepasst. Probleme gibt es aber, falls Sie die Eigenschaft *ButtonSize* des Symbolleistensteuerelements manuell im Eigenschaftsfenster anpassen. Dann verwendet das Laufzeitsystem diese Vorgaben. Wurde die Schaltflächengröße auf die Symbolabmessungen abgestimmt, wird ein zusätzlich hinzugefügter Text nicht mehr angezeigt (dies gilt selbst dann, wenn Sie auf ein Symbol verzichten und nur einen Buchstaben zuordnen; .NET Framework reserviert Platz für das nicht vorhandene Symbol). Passen Sie die Breite in *ButtonSize* so an, dass Text und Symbol angezeigt werden, sehen Schaltflächen, die nur ein Symbol enthalten, ziemlich unschön aus (die Breite der Schaltflächen wird für alle Schaltflächen gleich gesetzt). Falls Ihnen dies im Designer einmal passiert, löschen Sie den Wert der Eigenschaft *ButtonSize* im Eigenschaftsfenster. Der Designer hinterlegt dann automatisch einen »Standardwert« im Feld dieser Eigenschaft, die automatische Größenanpassung bei den Schaltflächen dieser Symbolleiste funktioniert wieder.

Tipp

Haben Sie Schaltflächen nur mit Symbolen belegt und werden diese Symbole in der rechten unteren Ecke der Schaltfläche angezeigt? In diesem Fall sollten Sie die Eigenschaft *TextAlign* vom Wert *Right* auf *Underneath* umsetzen, um die Symbole waagrecht in der Schaltfläche zu zentrieren. Haben Sie Schaltflächen mit Text und Symbolen in der Symbolleiste eingefügt und die Eigenschaft *ButtonSize* des *ToolBar*-Steuerelements manuell auf die Größe des Symbols abgestimmt, kann es passieren, dass der zugehörige Text nicht mehr angezeigt wird. Dann sollten Sie die Eigenschaft über den Kontextmenübefehl *Zurücksetzen* auf den Standardwert zurücksetzen.

Hinzufügen der Schaltflächen zu einer Symbolleiste

Das Hinzufügen von Elementen (Schaltflächen) zur Symbolleiste erfolgt als Auflistung. Auch dies lässt sich im *ToolBarButton-Auflistungs-Editor* bewältigen.

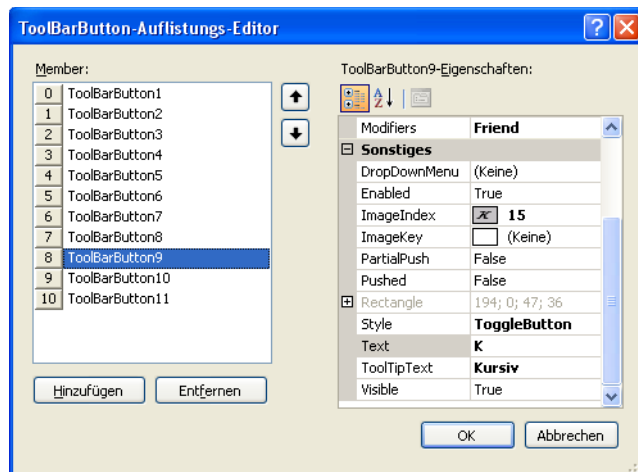


Abbildung 11.22: ToolBarButton-Auflistungs-Editor

1. Klicken Sie im Eigenschaftensfenster des *ToolBar*-Steuerelements auf die Schaltfläche der Eigenschaft *Buttons*, um den *ToolBarButton-Auflistungs-Editor* zu öffnen (Abbildung 11.22).
2. Klicken Sie im Fenster des Editors auf die Schaltfläche *Hinzufügen* und legen Sie anschließend die Eigenschaften für die Schaltfläche fest.
3. Wiederholen Sie den letzten Schritt, um alle in der Symbolleiste erzeugten Schaltflächen in die Auflistung aufzunehmen.
4. Passen Sie bei Bedarf die Reihenfolge der Schaltflächen über die beiden rechts von der Liste *Member* befindlichen Schaltflächen an und schließen Sie das Dialogfeld des Editors über die *OK*-Schaltfläche.

Dann sollten die Schaltflächen der Symbolleiste bereits im Formularentwurf zu sehen sein. Übersetzen und starten Sie das Projekt, sind die Schaltflächen der Symbolleiste anklickbar. Eventuell definierte ToolTip-Infos lassen sich abrufen. Die Eigenschaften der Schaltflächen werden im Fenster des *ToolBarButton-Auflistungs-Editors* (Abbildung 11.15) geändert. Markieren Sie hierzu die gewünschte Schaltfläche in der Liste *Member* und passen Sie die Eigenschaft in der Eigenschaftenliste an.

- In der Eigenschaft *Tag* können Sie eine Zahl oder einen Text hinterlegen, über den sich die Schaltfläche eindeutig innerhalb der Symbolleiste identifizieren lässt. Dies ist beispielsweise hilfreich, wenn Sie die angeklickte Schaltfläche in einer Ereignisbehandlungsroutine ermitteln müssen. .NET Framework löst beim Anklicken einer Schaltfläche nur ein *Click*-Ereignis für die komplette Symbolleiste aus. Die Eigenschaft *Name* können Sie auf dem Vorgabewert belassen. Dies ist die Eigenschaft, unter der das Objekt angesprochen wird.
- Die Schaltfläche lässt sich über die Eigenschaft *Visible* ein-/ausblenden und über die Eigenschaft *Enabled* sperren. Standardmäßig sind die Werte mit *True* freigegeben.
- Soll der Schaltfläche ein Symbol zugewiesen werden, wählen Sie dessen Index über die Eigenschaft *ImageIndex*. Diese Eigenschaft wird nur dann Werte aufweisen, wenn der Symbolleiste ein *ImageList*-Steuerelement zugeordnet wurde.
- Soll die Schaltfläche mit einem Beschriftungstext versehen werden, tragen Sie diesen in der Eigenschaft *Text* ein. Es wird dann der für die Symbolleiste vereinbarte Font zur Darstellung benutzt.
- Einen eventuell benötigten *ToolTip*-Text tragen Sie in der gleichnamigen Eigenschaft ein. Der Text wird beim Zeigen auf die Schaltfläche als ToolTip eingeblendet.
- Die vom Control in Symbolleisten unterstützten Schaltflächenvarianten werden über die Eigenschaft *Style* vorgegeben. Der Wert *PushButton* erzeugt eine Schaltfläche, die nach dem Anklicken in die Ruhelage »zurückspringt«. Mit *ToggleButton* erhalten Sie eine Schaltfläche, die zwei Stati (Ruhelage und eingedrückt) erlaubt. Wählen Sie den Wert *Separator*, wird eine Trennlinie zwischen den Schaltflächen generiert. Der Wert *DropDownButton* erzeugt eine Schaltfläche mit einem Dropdown-Pfeil am rechten Rand. Über den Pfeil lässt sich ein Menü öffnen, in dem der Benutzer weitere Befehle abrufen kann.
- Haben Sie die Eigenschaft *Style* auf den Wert *DropDownButton* gesetzt, lässt sich der Schaltfläche ein Kontextmenü über die Eigenschaft *DropDownMenu* zuweisen. Das Kontextmenü muss dabei als Steuerelement im Formular vorliegen. Der Benutzer kann zur Laufzeit das Menü über den Dropdown-Pfeil der Schaltfläche öffnen und die Befehle des Kontextmenüs abrufen.

Die Verwaltung der den *DropDownButton*-Schaltflächen zugeordneten Menüs erfolgen durch die *ToolBarButton*-Klasse des .NET Framework. Sobald Sie das Menü im Designer definiert (siehe vorhergehende Abschnitte) und der Schaltfläche zugewiesen haben, können Sie das Projekt übersetzen und ausführen. Dann lassen sich die Menüs der *DropDownButton*-Schaltfläche bereits öffnen und die Befehle anklicken. Um sinnvolle Aktionen an diese Menüs zu binden, müssen Sie die entsprechenden Ereignisbehandlungsroutinen implementieren. Dies haben Sie bereits weiter oben im Abschnitt »Abbildung« kennen gelernt. Auf den folgenden Seiten finden Sie weitere Hinweise zu diesem Thema.

Hinweis

Im Ordner `\Beisp\Kap11\SymbolBarAlt` der Buch-CD finden Sie die Projektdateien, die das alte *ToolBar*-Steuerelement aus .NET 1.x verwenden. Um das Formular dieses Beispiels in der Entwicklungsumgebung zu bearbeiten, müssten Sie aber das alte *ToolBar*-Steuerelement manuell zur Toolbox hinzufügen (siehe Kapitelanfang).

11.3.4 Eine Symbolleiste per Programm erzeugen

Falls Sie zur Laufzeit eine neue Symbolleiste per Programmcode einfügen und die Schaltflächen auf gleichem Weg definieren möchten, ist dies über die *ToolStrip*-Klasse möglich. Hierzu soll das vorherige Beispiel etwas modifiziert werden. Das Formular besitzt beim Start nur noch eine Menü- und eine Symbolleiste – die restlichen Symbolleisten wurden in der Entwicklungsumgebung entfernt. Klickt der Benutzer auf die Schaltfläche *Neu* der sichtbaren Symbolleiste, soll eine neue Symbolleiste mit drei Schaltflächen im *ToolStripContainer* eingeblendet werden. Wählt der Benutzer die erste Schaltfläche dieser neuen Symbolleiste, erscheint lediglich ein Meldungsfeld. Über die zweite Schaltfläche der Symbolleiste lässt sich diese an den unteren Rand des Fensters verschieben und die dritte Schaltfläche schiebt die Symbolleiste wieder in den *ToolStripContainer* des Fensters zurück. Die Anwendung soll dabei die jeweils nicht nutzbaren Schaltflächen der neu erzeugten Symbolleiste sperren. Das nachfolgende Codefragment zeigt die Implementierung dieses Ansatzes:

```
Public Class Form1
    Friend WithEvents ToolBar1 As ToolStrip
    Friend WithEvents ToolBarButton1 As ToolStripButton
    Friend WithEvents ToolBarButton2 As ToolStripButton
    Friend WithEvents ToolBarButton3 As ToolStripButton
    ...
    Private Sub ToolStripMenuItem2_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles ToolStripMenuItem2.Click
        ' Befehl Neu im Menü
        MessageBox.Show("Neu")
    End Sub

    Private Sub ToolStripButton1_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles ToolStripButton1.Click
        ' Schaltfläche Neu der Symbolleiste
        ' -> Erzeuge eine neue Symbolleiste am unteren Rand
        With Me
            ' Symbolleiste erzeugen
```



```

.ToolBar1 = New ToolStrip()      ' Symbolleiste aufbauen
With ToolBar1
    .Text = "Symbolleiste"      ' Titel
    .Dock = DockStyle.Bottom    ' Verankerung unten
    .Visible = True             ' einblenden
End With

' Vereinbare drei neue Schaltflächenobjekte
.ToolBarButton1 = New ToolStripButton()
.ToolBarButton2 = New ToolStripButton()
.ToolBarButton3 = New ToolStripButton()

' Schaltfläche 1 mit Text und Symbol
With .ToolBarButton1
    .Text = "Nr. 1"
    .Name = "Button1"          ' Objektname
    .DisplayStyle = ToolStripItemDisplayStyle.ImageAndText
    .Image = ImageList1.Images.Item(1)
End With

With .ToolBarButton2
    .Text = "Nr. 2"
    .Name = "Button2"
    .ToolTipText = "Zum unteren Formularrand verschieben"
End With

With .ToolBarButton3
    .Text = "Nr. 3"
    .Name = "Button3"
    .ToolTipText = "In Symbolleisten-Container zurück"
    .Enabled = False           ' Sperren, da momentan nutzlos
End With

' ToolBar1 - Schaltflächen zur Symbolleiste hinzufügen
.ToolBar1.Items.Add(.ToolBarButton1)
.ToolBar1.Items.Add(.ToolBarButton2)
.ToolBar1.Items.Add(.ToolBarButton3)
End With

' Zum ToolStripContainer oder zum Formular hinzufügen
ToolStripContainer1.TopToolStripPanel.Controls.Add(ToolBar1)
'Controls.Add(ToolBar1)
End Sub

```

```
' ### - Ereignisbehandlung der Symbolleisten-Schaltflächen
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ToolBarButton1.Click
    ' Befehl Nr.1 in Benutzer-Symbolleiste
    MessageBox.Show("Schaltfläche Nr.1 angeklickt")
End Sub

Private Sub Button2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ToolBarButton2.Click
    ' Befehl Nr.2 in Benutzer-Symbolleiste
    Controls.Add(ToolBar1) ' Symbolleiste zum Formular schieben
    Me.ToolBarButton3.Enabled = True ' freigeben
    Me.ToolBarButton2.Enabled = False ' sperren
End Sub

Private Sub Button3_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ToolBarButton3.Click
    ' Befehl Nr.3 in Benutzer-Symbolleiste
    ' Symbolleiste in ToolStripContainer zurückschieben
    ToolStripContainer1.TopToolStripPanel.Controls.Add(ToolBar1)
    Me.ToolBarButton2.Enabled = True ' freigeben
    Me.ToolBarButton3.Enabled = False ' sperren
End Sub
...
End Class
```

Letztendlich wiederholt sich hier, was Sie bereits mehrfach beim Erstellen der Formulare kennen gelernt haben. Sie müssen die Objektvariablen der Elemente vereinbaren, wobei über *WithEvents* zusätzlich eine Ereignisbehandlung initiiert wird. Dann sind die Objekte zu instantiieren, die Eigenschaften zu setzen und bei Auflistungen verwenden Sie Methoden wie *Add* oder *AddRange*, um die Objekte hinzuzufügen.

Hinweis

Den Code des kompletten Beispiels finden Sie in der Datei *Form1.vb* des Projektbeispiels, welches im Ordner *\Beisp\Kap11\SymbolBarCode* der Begleit-CD hinterlegt ist.

Soll das alte *ToolBar*-Steuerelement dynamisch zur Laufzeit im Formular eingefügt werden, lässt sich ein ähnlicher Ansatz verwenden. Sie müssen lediglich die *ToolBar*-Klasse und die *ToolBarButton*-Elemente neu instantiieren und dem Container mittels der *Add()*-Methode hinzufügen. Anschließend sind die Ereignisbehandlungsroutinen zur Reaktion auf das Anklicken von Schaltflächen zu implementieren. Klickt der Benutzer auf eine Schaltfläche, löst dies aber ein *Click*-Ereignis für das *ToolBar*-Steuerelement aus.

Hinweis (Fortsetzung)

Der betreffenden Ereignisbehandlungsroutine wird ein Parameter *e* vom Typ *System.Windows.Forms.ToolBarButtonClickEventArgs* übergeben. Sie können dann die Eigenschaft *e.Tag* verwenden, um die angeklickte Schaltfläche herauszufinden. Enthält die Symbolleiste eine *DropDownButton*-Schaltfläche mit einem Menü? Dann fügen Sie für jedes Menüelement eine separate *Click*-Ereignisbehandlungsroutine ein. Aus Platzgründen wird an dieser Stelle auf den Abdruck eines entsprechenden Codeabschnitts verzichtet. Sie finden aber ein Projektbeispiel im Ordner `\Beisp\Kap11\SymbolBarCodeAlt` der Begleit-CD. Laden Sie das Projektbeispiel über die *.sln*-Datei in der Entwicklungsumgebung. In der Datei *SymbolBarCodeAlt.vb* findet sich der komplette Code zum Erzeugen einer *ToolBar*-Symbolleiste in einem Formular.

11.4 Eine Statusleiste verwenden

Windows-Anwendungen mit Dokumentbereichen, die über Fenster mit Menü- und/oder Symbolleisten verfügen, bieten in der Regel auch eine Statusleiste zur Anzeige von Informationen. Die Statusleiste erscheint dabei am unteren Fensterrand. In der Entwicklungsumgebung lassen sich Statusleisten standardmäßig über das neue *StatusStrip*-Element einfügen und konfigurieren. Die ältere *StatusBar*-Klasse aus .NET 1.x lässt sich nur dann im Formulardesigner nutzen, wenn Sie das betreffende Element manuell zur Toolbox hinzugefügt haben (siehe Kapitelanfang).

11.4.1 Entwurf einer Anwendung mit einer Statusleiste

Statusleisten lassen sich in der Entwicklungsumgebung mit wenigen Handgriffen als *StatusStrip*-Element in das Formular eines Anwendungsfensters einfügen. Der Designer generiert dabei den benötigten Code selbsttätig.

1. Legen Sie in der Entwicklungsumgebung eine neue Windows-Anwendung an und weisen Sie dem ersten Formular die gewünschten Eigenschaften zu (z.B. Titelleiste und Abmessungen). Ergänzen Sie das Formular um weitere Steuerelemente (Menü- und Symbolleisten etc.).
2. Fügen Sie ein *StatusStrip*-Steuerelement aus der Toolbox zum Formular hinzu und passen Sie ggf. dessen Eigenschaften (z.B. *AllowItemReorder*, erlaubt das Neuordnen der Elemente zur Laufzeit, *LayoutStyle*, beeinflusst die Anordnung der Elemente in der Statusleiste etc.) an.
3. Klicken Sie auf die im *StatusStrip*-Steuerelement eingeblendete *DropDownButton*-Schaltfläche des Designers (Abbildung 11.23) und wählen Sie im eingeblendeten Menü das einzufügende Steuerelement (*StatusLabel*, *ProgressBar*, *DropDownButton* oder *SplitButton*).
4. Wechseln Sie zum Eigenschaftfenster und setzen Sie die Eigenschaften des betreffenden Steuerelements. Wiederholen Sie diesen Vorgang so lange, bis alle in der Statusleiste gewünschten Elemente hinterlegt sind.

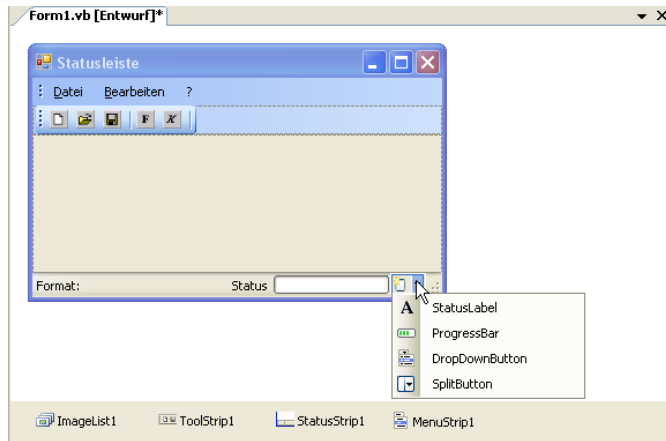


Abbildung 11.23: Entwurf des StatusBar-Beispiels

Bei einem *StatusLabel*-Steuerelement lässt sich der anzuzeigende Text über die *Text*-Eigenschaft vorgeben. Soll das Steuerelement als Link fungieren, hinterlegen Sie den URL der aufzurufenden Seite ebenfalls in der *Text*-Eigenschaft. Zusätzlich müssen Sie die Eigenschaft *IsLink* des Steuerelements auf den Wert *true* setzen.

Die Breite der Steuerelemente wird von verschiedenen Einstellungen bestimmt. Standardmäßig ist die Eigenschaft *AutoSize* auf den Wert *true* voreingestellt. Der Wert erlaubt dem Element, die Größe automatisch nach dem Inhalt festzulegen. Bei einem *StatusLabel* bestimmt dann beispielsweise der anzuzeigende Text, wie breit das Element ist. Um zu verhindern, dass bei der Anzeige unterschiedlicher Texte die Breite des Elements schwankt, können Sie die Eigenschaft *AutoSize* auf den Wert *false* setzen. Dann lässt sich die Breite über die *Size*- bzw. die *Width*-Eigenschaft des Elements einstellen. Setzen Sie die *Spring*-Eigenschaft auf *true*, passt das *StatusStrip*-Element die Breite des Steuerelements so an, dass dieses den verbleibenden Platz in der Leiste ausfüllt.

Möchten Sie ein Symbol in der Statusleiste einblenden, klicken Sie das Steuerelement an und wählen den Kontextmenübefehl *Bild festlegen*. Anschließend lässt sich ein Bitmap-Bild über eine Ressource oder aus einer Datei der Eigenschaft *Image* zuweisen. Über die Eigenschaft *ImageAlign* kann die Anordnung des Symbols beeinflusst werden und über *ImageScaling* lässt sich die Symbolgröße auf das Element abstimmen. Der Wert der Eigenschaft *DisplayStyle* steuert, ob Text und/oder Bild sichtbar sein sollen.

Die Darstellung des Statusleistenelements lässt sich über den Wert der Eigenschaft *BorderStyle* beeinflussen. Über *RaisedOuter* oder *Sunken* lässt sich der Bereich des Panels optisch hervorheben.

Bei einem *ProgressBar*-Steuerelement können Sie die Eigenschaften *Minimum*, *Maximum* und *Step* setzen, um den Wertebereich sowie die Schrittweite vorzugeben. *SplitButton*- und *DropDownButton*-Elemente lassen sich wie bei einer Symbolleiste gestalten. Sobald die Steuerelemente in der Statusleiste hinterlegt sind, können Sie die Anwendung bereits übersetzen. Das Formular wird die Statusleiste mit den statischen Informationen der Steuerelemente anzeigen.

Ereignisbehandlungsroutinen zur Aktualisierung der Statusleiste

Im aktuellen Beispiel wurden mehrere *StatusLabel*-Elemente und eine Fortschrittsanzeige hinterlegt. Das erste *StatusLabel*-Element dient nur zur Anzeige eines statischen Texts »Format«. Zudem ist diesem Steuerelement ein Symbol zugeordnet, welches sich wahlweise über die Befehle *Symbol ein* bzw. *Symbol aus* des *Bearbeiten*-Menüs ein – oder ausblenden lässt. Die Schaltflächen *F* und *K* werden im »Toggle«-Modus betrieben, d.h., ein Mausklick aktiviert oder deaktiviert die jeweilige Schaltfläche. Bei aktivierten Schaltflächen werden die Texte »Fett« und »Kursiv« im zweiten *StatusLabel*-Feld der Statusleiste eingeblendet. Ein Mausklick auf die Schaltfläche *Neu* der Symbolleiste bewirkt eine Änderung in der Fortschrittsanzeige. Klickt der Benutzer auf das als Hyperlink angelegte dritte Label, soll die Anwendung die betreffende Webseite im Internet Explorer abrufen. Dies ist durch geeignete Ereignisbehandlungsroutinen zu realisieren.

Im aktuellen Beispiel werden die Werte für Minimum und Maximum der Fortschrittsanzeige in der *Load*-Ereignisbehandlungsroutine gesetzt. Beim Anklicken der Schaltfläche *Neu* der Symbolleiste wird in der *Click*-Ereignisbehandlungsroutine dieses Steuerelements der *Value*-Wert des Fortschrittsbalkens in der Fortschrittsanzeige um 10 erhöht. Nachfolgend ist der Code der Ereignisbehandlungsroutine zu sehen:

```
Private Sub ToolStripButton1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ToolStripButton1.Click
    ' Befehl Neu - Progressbar ändern
    If Me.ToolStripProgressBar1.Value < _
        Me.ToolStripProgressBar1.Maximum Then
        Me.ToolStripProgressBar1.Value = _
            Me.ToolStripProgressBar1.Value + 10
    Else
        Me.ToolStripProgressBar1.Value = _
            Me.ToolStripProgressBar1.Minimum
    End If
End Sub
```

Beim Zuweisen eines Werts müssen Sie darauf achten, dass dieser nicht außerhalb der angegebenen Grenzen liegt.



Abbildung 11.24: StatusBar-Beispiel

Das Ein- oder Ausblenden des *F*-Symbols im ersten *StatusLabel*-Steuerelement erfolgt über dessen *DisplayStatus*-Anweisung mit den Befehlen

```
Me.ToolStripStatusLabel1.DisplayStyle = _  
    ToolStripItemDisplayStyle.ImageAndText  
Me.ToolStripStatusLabel1.DisplayStyle = _  
    ToolStripItemDisplayStyle.Text
```

Die jeweilige Anweisung wurde in die *Click*-Ereignisbehandlungsroutinen der entsprechenden Befehle des Menüs *Bearbeiten* hinterlegt. Um beim Anklicken des als Link ausgeführten Labels die zugehörige Verweiseite aufzurufen, muss in dessen *Click*-Ereignisbehandlungsroutine ein Aufruf der folgenden Art stehen:

```
System.Diagnostics.Process.Start(ToolStripStatusLabel4.Text)
```

Hier wird die *Start*-Methode des *Process*-Objekts mit dem in der Eigenschaft *Text* hinterlegten Link aufgerufen. Dies bewirkt die Anzeige der Folgeseite im Internet Explorer. Um den Status der beiden Schaltflächen *F* und *K* der Symbolleiste im zweiten *StatusLabel*-Element einzublenden, muss dessen *Text*-Eigenschaft in den *Click*-Ereignisbehandlungsroutinen entsprechend gesetzt werden. Hier ist der Code für die *Click*-Ereignisbehandlung der Schaltfläche *F* zu sehen:

```
Private Sub ToolStripButton6_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles ToolStripButton6.Click  
    ' Schaltfläche Fett - blende Text in Statusbar ein  
    If Me.ToolStripButton6.Checked Then  
        Fett = " Fett "  
    Else  
        Fett = "      "  
    End If  
    ' Format in Statusfeld einblenden  
    Me.ToolStripStatusLabel1.Text = Fett & Kursiv  
End Sub
```

Die Routine weist der Variablen *Fett* einen Text oder mehrere Leerzeichen zu (je nach Status der Schaltfläche, der über die *Checked*-Eigenschaft ausgewertet wird). Anschließend wird der Inhalt der Variablen *Fett* und der in einer zweiten Ereignisbehandlungsroutine gesetzten Variablen *Kursiv* der *Text*-Eigenschaft des betreffenden *StatusLabel*-Elements zugewiesen.

Hinweis

Sie finden das Projektbeispiel im Ordner `\Beisp\Kap11\StatusBar` auf der Begleit-CD. Im Unterverzeichnis *Images* finden Sie zwei Dateien *MiniCross.ico* und *Book.ico*, die mit den Abmessungen 16x18 Pixel entworfen wurden.

11.4.2 Eine StatusBar-Beispielanwendung

Abschließend möchte ich noch ein kleines Anwendungsbeispiel vorstellen, welches das Verhalten der Statusleiste demonstriert und einige der in den vorhergehenden Abschnitten vermittelten Techniken zusammenfasst. Ein Anwendungsfenster enthält ein *TextBox*-Steuerelement, eine Menüleiste, eine Symbolleiste und eine Statusleiste (Abbildung 11.25).

Über das Menü *Datei* sowie über die Schaltflächen der Symbolleiste lassen sich verschiedene Befehle abrufen, wobei die meisten Elemente nur ein Dialogfeld abrufen. Die Funktion *Neu* löscht den Inhalt des Textfelds. Im Menü *Bearbeiten* finden sich zwei Befehle, um den Inhalt des Textfelds zu löschen oder mit einem Text zu füllen. Das Menü *Ansicht* enthält Befehle, um die Statusleiste zwischen einer einfachen Darstellung und einer erweiterten Variante mit eingeblendetem Symbol bzw. Datum umzuschalten. Zudem finden Sie Befehle, um die Statusanzeige zu löschen bzw. zu aktualisieren.

Die Statusleiste zeigt die Zahl der Zeichen sowie die Zahl der Zeilen im Textfeld. Die Zeilenanzahl wird durch einen Zeilenumbruch am rechten Rand des Fensters nicht beeinflusst, es werden nur die durch Absatzmarken getrennten Zeilen ermittelt.

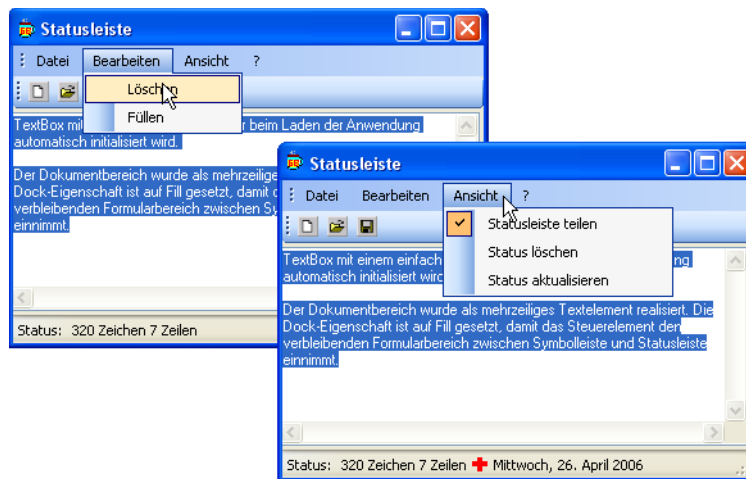


Abbildung 11.25: Fenster des Anwendungsbeispiels

Die Steuerelemente wurden statisch im Fenster des Ansicht-Designers im Formular eingetragen. In der *Load*-Ereignisbehandlungsprozedur werden die Eigenschaften der Textbox sowie der Statusleiste definiert. Hierbei trägt das Programm in der Statusleiste im rechten Label das aktuelle Datum in der *Text*-Eigenschaft ein. Über die *Visible*-Eigenschaft wird die Anzeige aber ausgeblendet. In der Ereignisbehandlungsroutine des Befehls *Ansicht/Statusleiste teilen* muss dann die *Visible*-Eigenschaft auf *true* gesetzt werden, um das Datum in der Statusanzeige einzublenden. Der Code enthält eine Prozedur *FillText*, die das Steuerelement mit dem Text belegt. Diese Prozedur wird über den Menübefehl *Bearbeiten/Füllen* aufgerufen. Die Aktualisierung der Statusanzeige erfolgt über eine eigene Prozedur *UpdateStatus*, die folgende Anweisungen enthält:

```
Private Sub UpdateStatus()  
    Dim Zeilen As Integer ' Zeichenzahl  
    Dim Zeichen As Integer ' Zeilen (ohne Wrap)  
    Dim txt As String  
  
    Zeichen = Me.TextBox1.TextLength  
    Zeilen = Me.TextBox1.Lines().GetUpperBound(0) + 1  
  
    txt = Zeichen.ToString & " Zeichen " & _  
        Zeilen.ToString & " Zeilen"  
  
    ' Statusleiste aktualisieren  
    Me.ToolStripStatusLabel2.Text = txt  
    If Me.StatusleisteTeilenToolStripMenuItem.Checked Then  
        Me.ToolStripStatusLabel4.Text = "Zeit: " & _  
            System.DateTime.Today.ToLongDateString()  
        Me.ToolStripStatusLabel4.Visible = True  
    Else  
        Me.ToolStripStatusLabel4.Visible = False ' 2. Label einblenden  
    End If  
End Sub
```

Die in der Statusleiste angezeigte Zeichenzahl wird über `Me.TextBox1.TextLength` ermittelt. Die Zahl der Textzeilen im Textfeld muss über einen Trick berechnet werden. Sie können das `Lines`-String-Array mit der `GetUpperBound(0)`-Methode auf den höchsten Index abfragen. Der um 1 erhöhte Wert gibt die Zahl der Textzeilen (die durch harte Zeilenumbrüche gekennzeichnet sind) an. Schaltet der Benutzer die `WordWrap`-Eigenschaft des Textfelds auf `True`, bewirkt dies eventuelle Zeilenumbrüche am rechten Fensterrand. Diese »weichen« Zeilenumbrüche werden aber nicht in der Zeilenzahl berücksichtigt.

Hinweis

Weitere Details zur Implementierung der Funktionen in den Ereignisbehandlungsroutinen finden Sie in den VB-Dateien des Projektbeispiels, welches im Ordner `\Beisp\Kap11\StatusBar1` auf der Begleit-CD hinterlegt ist.

11.4.3 Nutzen des alten StatusBar-Controls

Möchten Sie bei bestehenden Projekten die alte `StatusBar`-Klasse weiter nutzen und haben Sie das betreffende Element in der Toolbox manuell hinzugefügt? Dann lässt sich ein `StatusBar`-Steuerelement per Toolbox im Formular einfügen. Allerdings ergeben sich einige Unterschiede zum neuen `StatusBarStrip`-Steuerelement.

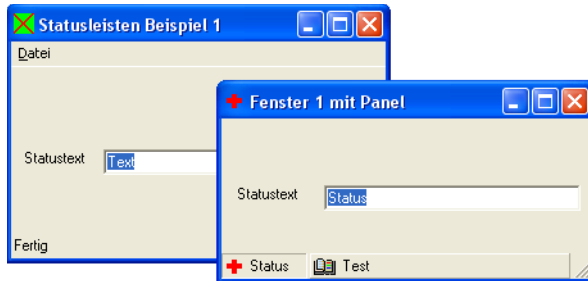


Abbildung 11.26: Fenster des StatusBar-Beispiels

Sie können über das *StatusBar*-Steuerelement eine einfache Statusleiste erzeugen, die den gesamten unteren Fensterrand einnimmt und lediglich Texte darstellen kann (Abbildung 11.26, linkes Fenster). Die zweite Variante der Statusleiste benutzt sogenannte Panels, wobei jedem Panel ein eigener Anzeigestil zugeordnet werden kann (Abbildung 11.26, rechtes Fenster). Panels unterteilen die Statusleiste in mehrere Abschnitte, die Text und/oder Symbole anzeigen können. Um eine einfache Statusleiste in einem Formular bereitzustellen, fügen Sie das *StatusBar*-Steuerelement zum Formularentwurf hinzu. Dann lässt sich über die *Text*-Eigenschaft des Steuerelements ein Text in der Statusleiste anzeigen. Zur Laufzeit können Sie den Text über folgende Anweisung anpassen:

```
Me.StatusBar1.Text = "Fertig ..."
```

Entsprechend lässt sich auch lesend auf die betreffende *Text*-Eigenschaft des Steuerelements zugreifen.

Eine StatusBar-Statusleiste mit mehreren Panels ausstatten

Um eine Statusleiste um mehrere Panels zu erweitern und diesen Text und/oder Symbole zuzuweisen, sind folgende zusätzliche Schritte erforderlich:

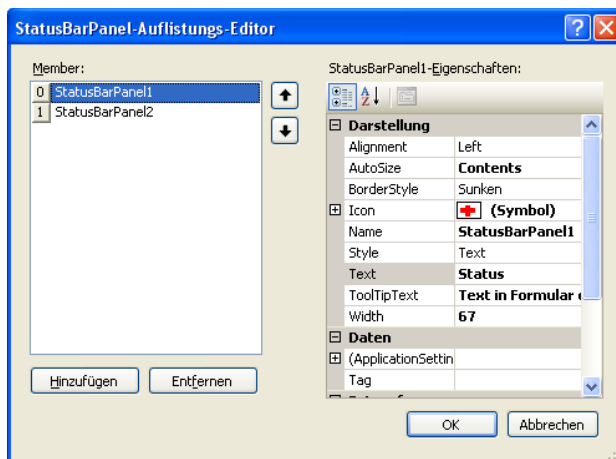


Abbildung 11.27: Konfigurierung der Panels im StatusBarPanel-Auflistungs-Editor

1. Wählen Sie das im Formulardesign eingefügte *StatusBar*-Steuerelement im Fußbereich des Formulars an und wechseln Sie zu dessen Eigenschaftenfenster.
2. Setzen Sie die *ShowPanels*-Eigenschaft auf *True*. Mit dem Wert *False* wird dagegen nur eine einfache Statusleiste angezeigt.
3. Klicken Sie im Eigenschaftenfenster des *StatusBar*-Steuerelements auf die Schaltfläche der Eigenschaft *Panels*, um den *StatusBarPanel*-Auflistungs-Editor aufzurufen und die Panels interaktiv zu konfigurieren.
4. Im *StatusBarPanel*-Auflistungs-Editor fügen Sie neue Panels über die Schaltfläche *Hinzufügen* in die Auflistung ein. Anschließend setzen Sie die Eigenschaften des Panel in diesem Dialogfeld (Abbildung 11.27). Bestehende Member können Sie in der linken Spalte markieren und dann die Eigenschaften in der rechten Spalte setzen. Den Panels lassen sich Text und Symbole, ein Stil und ein ToolTip über verschiedene Eigenschaften zuweisen.

Die Eigenschaften eines Panels bestimmen, wie dieses angezeigt wird. Nachfolgend noch einige kurze Hinweise, was beim Festlegen der Eigenschaften zu beachten ist:

- Eigenschaft *Alignment*: Legt über die Werte *Left*, *Right* und *Center* fest, wie der Inhalt eines Panel (Text und Bild) anzuordnen ist.
- Eigenschaft *AutoSize*: Der Wert erlaubt dem Panel, die Größe automatisch nach dem Inhalt festzulegen. Mit *None* wird die Eigenschaft abgeschaltet. Zusätzlich sind die Werte *Spring* und *Contents* zulässig. Objekte, deren *AutoSize*-Eigenschaft auf *StatusBarPanelAutoSize.Contents* gesetzt ist, haben Vorrang vor den Panel-Bereichen, die auf den *StatusBarPanelAutoSize.Spring*-Wert fest gelegt sind. Beansprucht beispielsweise ein mit *Content* belegtes Panel den Platz, wird ein mit *Spring* ausgezeichnetes benachbartes Panel gekürzt. Die Breite (in Pixel) eines Panel lässt sich über die Eigenschaft *Width* festlegen.
- Eigenschaft *BorderStyle*: Definiert über die Werte *None*, *Raised* und *Sunken* die Anzeige des Panels. Über *Raised* und *Sunken* lässt sich der Bereich des Panels optisch hervorheben.
- Eigenschaft *Icon*: Über diese Eigenschaft können Sie dem Panel ein Symbol (ico-Datei) zuordnen. Wichtig ist dabei, dass das Symbol in den zum Panel passenden Abmessungen vorliegt. Andernfalls werden Bildteile abgeschnitten. Das Symbol wird direkt aus einer Datei übernommen (und in VB.NET in einer Ressourcendatei hinterlegt).
- Eigenschaft *Text*: Über diese Eigenschaft können Sie dem Panel einen statischen Text zuordnen. Dieser wird ggf. rechts neben dem Symbol angezeigt.
- Eigenschaft *ToolTipText*: Über diese Eigenschaft lässt sich dem Panel ein ToolTip-Text zuweisen. Dieser wird zur Laufzeit angezeigt, sobald der Benutzer auf das Panel zeigt.

Die Eigenschaft *Style* sollten Sie auf dem Eintrag *Text* belassen, damit das Steuerelement die Anzeige übernimmt.

Hinweis

Bei vielen Steuerelementen ist die Eigenschaft *OwnerDraw* vorhanden. Wird diese Eigenschaft auf *True* gesetzt, müssen Sie die Zeichenoperationen selbst in einer *OwnerDraw*-Ereignisbehandlungsroutine durchführen. Dies erlaubt zwar spezifische Anpassungen, führt aber über den Ansatz dieses Buches hinaus. Im Ordner `\Beisp\Kap12\IconMenu\Source` finden Sie die bereits erwähnte Erweiterungsklasse von Markus Palme, die diesen Ansatz demonstriert.

Zur Laufzeit kann direkt über die Objektnamen des Panel auf dessen *Text*-Eigenschaft mit folgender Anweisung zugegriffen werden:

```
StatusBarPanel1.Text = "Fertig ..."
```

Der einzige Unterschied zur Aktualisierung der Textanzeige eines *StatusBar*-Steuerelements besteht in der Verwendung eines anderen Objektnamens für das Panel (hier *StatusBarPanel1*). Abschließend möchte ich noch auf einen speziellen Punkt eingehen. Panel-Elemente lassen sich per Mausklick anwählen, was ein *StatusBar_Panel*-Ereignis auslöst. Der nachfolgende Code zeigt die betreffende Ereignisbehandlungsroutine:

```
Private Sub StatusBar1_PanelClick(ByVal sender As System.Object, _
    ByVal e As System.Windows.Forms.StatusBarPanelClickEventArgs) _
    Handles StatusBar1.PanelClick
    ' Klick auf Statusleiste, filtere Panel
    If e.StatusBarPanel Is StatusBarPanel1 Then
        MessageBox.Show("Bitte einen Text eingeben", "Panel1", _
            MessageBoxButtons.OK, MessageBoxIcon.Information)
    ElseIf e.StatusBarPanel Is StatusBarPanel2 Then
        ' BorderStyle 2. Panel beim Klick anheben/absenken
        With StatusBarPanel2
            If.BorderStyle = StatusBarPanelBorderStyle.Raised Then
                .BorderStyle = StatusBarPanelBorderStyle.Sunken
            Else
                .BorderStyle = StatusBarPanelBorderStyle.Raised
            End If
            MessageBox.Show("Wir prüfen doch noch!!!", "Panel2", _
                MessageBoxButtons.OK, MessageBoxIcon.Hand)
        End With
    End Sub
```

Das *Click*-Ereignis bezieht sich auf das *StatusBar*-Objekt. Sie müssen daher in der Ereignisbehandlungsroutine ermitteln, welches Panel angewählt wurde. Hierzu wird der Parameter *e* der Prozedur benutzt. Die Eigenschaft *StatusBarPanel* des im Parameter *e* übergebenen Objekts enthält den Objektnamen des betreffenden Steuerelements. Über die *Is*-Klausel lässt sich daher direkt prüfen, ob das Objekt einem vorgegebenen Objekt-

namen entspricht. Trifft dies zu, erscheint in obigem Code ein Dialogfeld. Beim zweiten Panel wird zudem die *BorderStyle*-Eigenschaft zwischen *Sunken* und *Raised* hin- und hergeschaltet.

Hinweis

Dieses Beispiel benutzt also den Objektnamen und nicht den *Tag*-Wert (wie beim *ToolBar*-Beispiel weiter oben) zur Ermittlung des angeklickten Container-Elements in der Ereignisbehandlungsroutine. Sie finden ein Projektbeispiel, welches das alte *StatusBar*-Steuerelement verwendet, im Ordner `\Beisp\Kap11\StatusBarAlt`. Dieses Beispiel enthält daher zwei Formulare (*Form1.vb* enthält eine einfache Statusleiste, während in *Form2.vb* mehrere *Panel*-Steuerlemente zur Aufnahme der Steuerelemente verwendet werden). Das Projekt lässt sich über die *.sln*-Datei in der Entwicklungsumgebung laden. Tragen Sie vor dem Übersetzen des Projekts das gewünschte Formular als Startobjekt in den Projekteigenschaften (Seite *Anwendung*) ein, um das Verhalten des jeweiligen Formulars zu studieren. Einzelheiten zur Implementierung entnehmen Sie dem Code der beiden Formulare.

Mit diesen Erläuterungen möchte ich das Kapitel schließen. Mit den bisherigen Ausführungen verfügen Sie über das benötigte Wissen, um eigene Anwendungen mit Menü-, Symbol- und Statusleisten auszustatten. Weitere Details finden Sie in der Hilfe zum .NET Framework. Im nachfolgenden Kapiteln geht es um die Verwendung von Dialogen (z.B. Laden oder Speichern).

12

Standarddialoge und MDI-Anwendungen

In diesem Kapitel wird gezeigt, wie sich Standarddialoge zum Öffnen, Speichern etc. in .NET-Anwendungen einbinden lassen. Weitere Themen sind das Gestalten von MDI-Anwendungen und der Einsatz verschiedener Anwendungstechniken.

12.1 Standarddialoge nutzen

Zum Öffnen und Speichern von Dateien, zur Auswahl der Schriftart oder zur Anzeige einer Farbpalette bietet Windows verschiedene Standarddialoge an (zum Drucken gilt Ähnliches, siehe *Kapitel 16*). Das .NET Framework unterstützt die Verwendung solcher Dialoge über eigene Klassen.

12.1.1 Eine Beispielanwendung für Standarddialoge

Die Verwendung einiger Standarddialoge soll am Beispiel einer einfachen Anwendung (Mini-Editor, Abbildung 12.1) demonstriert werden. Diese bietet ein Anwendungsfenster mit Menüleiste, einer Symbolleiste mit einigen Schaltflächen, einem mehrzeiligen Textfeld und einer Statusleiste, in der die Zahl der Zeichen und Zeilen sowie der Änderungsstatus angezeigt werden. Über die Schaltfläche *Neu* (bzw. den gleichnamigen Befehl im Menü *Datei*) lässt sich der Inhalt des Textfelds löschen. Die Schaltfläche *Öffnen* (bzw. der gleichnamige Befehl im Menü *Datei*) öffnet das Dialogfeld *Öffnen* zur Auswahl einer Datei (wobei verschiedene Dateitypen *.txt*, *.ini*, *.reg*, *.vb*, *.vbs*, *.log*) unterstützt werden. Über die Schaltfläche *Speichern* (bzw. der gleichnamige Befehl im Menü *Datei*) wird der Inhalt des Textfelds in der aktuellen Datei gesichert. Bei einem neuen Dokument soll der *Speichern unter*-Dialog erscheinen. Das Gleiche gilt bei Anwahl des Befehl *Speichern unter* im Menü *Datei*, wobei dieser immer den *Speichern unter*-Dialog zeigt.

Das Beispiel eignet sich gut, um weitere Techniken, die Sie bei der Entwicklung realer Anwendungen benötigen, zu demonstrieren. Daher stellt das Menü *Bearbeiten* den Befehl *Rückgängig* zur Verfügung, mit dem sich die letzte Änderung zurücknehmen lässt. Im Menü *Format* finden Sie die Befehle *Schriftart...* (erlaubt die Auswahl der Schriftartoptionen), *Zeilenumbruch* (schaltet den weichen Zeilenumbruch am rechten Rand ein/aus), *Farben* (erlaubt die Auswahl der Schriftfarbe) und *Pfad in Titel* (blendet den Pfad der geladenen Datei in der Titelzeile ein/aus). Zudem muss in der Anwendung sicher gestellt werden, dass Dokumentänderungen beim Laden eines neuen Dokuments oder beim Beenden der Anwendung noch vom Benutzer gesichert werden. In den nachfolgenden Abschnitten lernen Sie, wie sich diese Anwendung realisieren lässt. Ein großer Teil der Funktionalität lässt sich bereits mit dem Wissen aus den vorhergehenden Kapiteln und Abschnitten realisieren. So wurde die Gestaltung des Formulars mit Menü-, Symbol- und Statusleiste im vorhergehenden Kapitel behandelt. Ein Textfeld mit der

Eigenschaft *WordWrap* ist in Kapitel 11 (»Registerkarten und Textelemente verwenden«) zu finden. Gehen wir die Aufgabe schrittweise an.

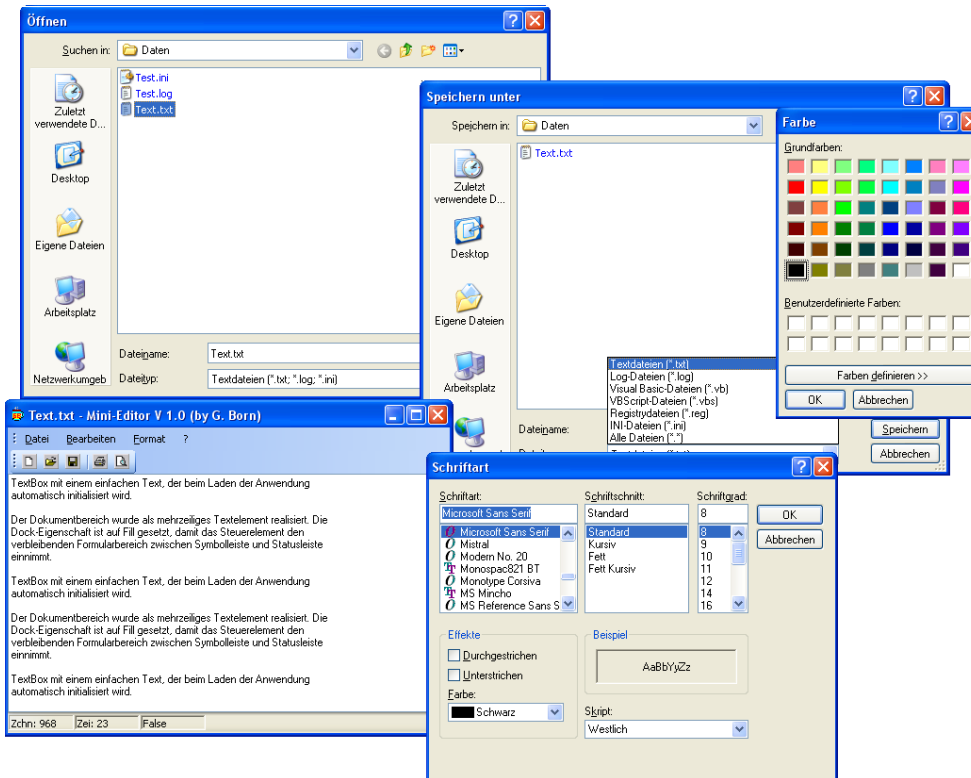


Abbildung 12.1: Dialogbeispiel Mini-Editor mit Windows-Dialogen

12.1.2 Erstellen des Anwendungsformulars

Der Entwurf der sichtbaren Komponenten des Anwendungsfensters ist in der Entwicklungsumgebung kein Problem. Im Grunde können Sie sogar auf das zuletzt entwickelte Projektbeispiel zurückgreifen und dieses anpassen.

1. Legen Sie in der Entwicklungsumgebung ein neues Windows-Projekt an oder laden Sie eine bestehende Projektdatei.
2. Statten Sie das Formular mit einer Menüleiste gemäß Abbildung 12.2 aus und fügen Sie zusätzlich eine Symbolleiste mit den in der Abbildung gezeigten Schaltflächen hinzu. Die Statusleiste ist mit drei *StatusLabel*-Elementen zu versehen.

Bei diesem Beispiel werden die neuen .NET-2.0-Strip-Steuerlemente (*MenuStrip*, *ToolStrip*, *StatusBarStrip*) eingesetzt. Die Steuerelemente sind in Abbildung 12.2 im Fußbereich des Formulardesigns zu sehen. Das Setzen der Eigenschaften der betreffenden Steuerelemente wurde bereits im vorherigen Kapitel behandelt. Allerdings gibt es noch eine Neuerung, die kurz erwähnt werden soll. Um bei den *StatusLabel*-Elementen eines

StatusBarStrip-Steuerelements eine abgesenkte Darstellung zu erreichen, muss die Eigenschaft *BorderStyle* auf *Sunken* gesetzt werden. Damit diese Eigenschaft aber Auswirkungen hat, muss zusätzlich noch die Eigenschaft *BorderSides* auf *All* eingestellt werden. Zudem habe ich bei den *StatusLabel*-Elementen die Eigenschaft *AutoSize* auf *false* gesetzt und eine feste Breite zugewiesen.

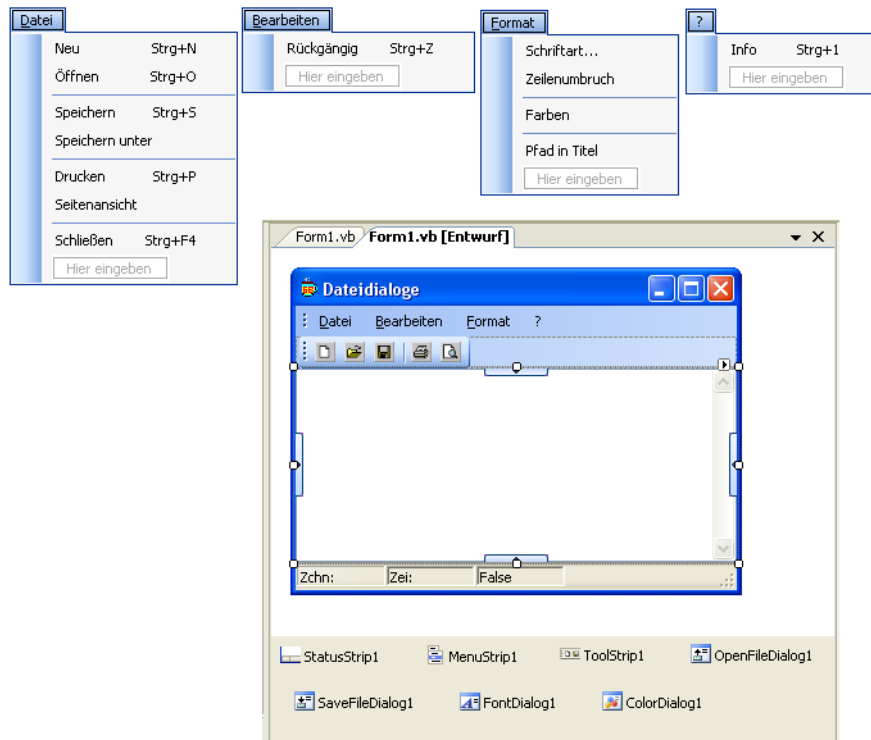


Abbildung 12.2: Formularentwurf und Menüstruktur

3. Fügen Sie ein *TextBox*-Steuerelement zum Formularentwurf hinzu. Setzen Sie die Eigenschaften folgendermaßen: *MultiLine* = *True* (erlaubt mehrere Zeilen), *BorderStyle* = *Fixed3D*, *AcceptsReturn* = *True* (erlaubt bei Eingaben), *AcceptsTab* = *True* (unterstützen), *AllowDrop* = *True* (Ziehen in Textfeld unterstützen) und *Dock* = *Fill* (bewirkt, dass das Element den verbleibenden Platz im Anwendungsfenster einnimmt). Über die Eigenschaft *MaxLength* können Sie ggf. die maximal unterstützte Zeichenzahl setzen (die Grenze liegt bei 64 KB).
4. Fügen Sie abschließend aus der *ToolBox* die Steuerelemente *OpenFileDialog*, *SaveFileDialog*, *FontDialog* und *ColorDialog* dem Entwurf hinzu. Diese Elemente sind nicht sichtbar und werden zur Anzeige der Dialoge benötigt. Der Designer generiert dann automatisch den Code zur Deklaration und Instantiierung der Objekte. Die Objekte lassen sich anschließend im Code über die Objektnamen (z.B. *DateiDialog1*) referenzieren. Alternativ können Sie die Objektdeklaration und -instantiierung manuell im Programmcode vornehmen und somit auf das Einfügen der betreffenden (nicht sichtbaren) Steuerelemente verzichten.

Sobald das Design des Anwendungsfensters steht, können Sie zur Codeansicht des Formulars wechseln und die Ereignisbehandlungsroutinen für angewählte Menübefehle oder Schaltflächen der Symbolleiste implementieren.

Hinweise zur Anpassung des Codes

Die Anwendung muss sich einmal merken, ob der Dokumentinhalt durch den Benutzer verändert wurde. Zudem sollte der Name der Dokumentdatei bekannt sein. Daher sind auf der Ebene der *Form1*-Klasse zwei Variable zur Aufnahme dieser Werte zu vereinbaren.

```
Dim docChanged As Boolean = False      ' Dokument geändert
Dim file As String = ""                ' Dokumentname
```

Die Initialisierungswerte weisen das neue Dokument als unverändert aus, ein Dateiname existiert nicht. Weiterhin wird eine Prozedur *UpdateStatus* (ähnlich wie im *StatusBar1*-Beispiel des vorherigen Kapitels) zur Anzeige der Zeichen- und Zeilenzahl in der Statusleiste benutzt. Zudem soll der Mini-Editor den Dateinamen wahlweise mit dem Pfad in der Titelseite ausgeben. Die *UpdateStatus*-Prozedur sieht dann folgendermaßen aus:

```
Private Sub UpdateStatus()
    ' Hilfsprozedur zum Aktualisieren der Anzeige
    ' Aktualisiert Titelseite und Statusleiste
    Const Title As String = " - Mini-Editor V 1.0 (by G. Born)"
    Dim Zeilen As Integer ' Zeichenzahl
    Dim Zeichen As Integer ' Zeilen (ohne Wrap)

    ' Zeichen- und Zeilenzahl ermitteln
    Zeichen = Me.TextBox1.TextLength
    Zeilen = Me.TextBox1.Lines().GetUpperBound(0) + 1
    ' Statusleiste (3 'Panels') aktualisieren
    ' Zeichenzahl, Zeilenzahl, Änderungsstatus
    Me.ToolStripStatusLabel1.Text = "Zchn: " & Zeichen.ToString
    Me.ToolStripStatusLabel2.Text = "Zei: " & Zeilen.ToString
    Me.ToolStripStatusLabel3.Text = docChanged.ToString

    ' Titelleiste aktualisieren
    ' bei neuen Dokumenten Anzeige "Neu - ...."
    ' sonst den Dateinamen einblenden
    ' dabei zwischen kurzem Dateinamen und
    ' langem Dateinamen mit Pfad unterscheiden
    ' Anpassung über Menü Format/Pfad in Titel
    If file = "" Then
        Me.Text = "Neu" & Title ' neues Dokument
```

```

Else ' kurzer Dateiname oder kompletter Pfad?
    If Me.PfadInTitelToolStripMenuItem.Checked Then
        Me.Text = file & Title ' Dateiname
    Else ' kompletten Pfad anzeigen
        Me.Text = IO.Path.GetFileName(file) & Title
    End If
End If

' Sperre Schaltfläche Rückgängig, wenn Text unverändert
If docChanged = True Then
    Me.RückgängigToolStripMenuItem.Enabled = True ' freigeben
Else
    Me.RückgängigToolStripMenuItem.Enabled = False ' sperren
End If
End Sub

```

Der Code zur Aktualisierung der Anzeige in der Statusleiste birgt kaum Neues. Es wird einfach die *Text*-Eigenschaft des jeweiligen *StatusLabel*-Steuerelements gesetzt. Im dritten Element wird der Status des Flags *docChanged* zu Kontrollzwecken eingeblendet. Neu ist allerdings die Aktualisierung der Titelzeile des Anwendungsfensters. Bei einem neuen Dokument (dann ist die Variable *file* mit einer leeren Zeichenkette belegt) erscheint ein Text »Neu – ...«. Falls ein Dateiname vorliegt, lässt sich dieser direkt in der Titelleiste hinterlegen (es wird der Inhalt der Variablen *file* sowie die Konstante *Title* der Eigenschaft *Text* des *Form*-Objekts, hier über *Me* referenziert, zugewiesen). Dieser Dateiname beinhaltet aber den kompletten Pfad. Falls der Benutzer die Option im Menü *Format/Pfad in Titel* abschaltet, muss die Pfadangabe unterdrückt werden. Hierzu habe ich in obiger Prozedur die *GetFileName*-Methode der Klasse *Path* (im Namensraum *System.IO.Path*) benutzt. Dieser Methode wird eine Pfadangabe samt Dateiname übergeben. Als Ergebnis erhalten Sie den Dateinamen ohne Pfad zurück.

Der Code zur Verwaltung des Menüeintrags *Format/Pfad in Titel* wird in der Ereignisprozedur des Menüelements hinterlegt und sieht folgendermaßen aus:

```

Private Sub PfadInTitelToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles PfadInTitelToolStripMenuItem.Click
    ' Menübefehl Pfadanzeige Titelzeile ein/aus
    With Me
        .PfadInTitelToolStripMenuItem.Checked = _
            Not .PfadInTitelToolStripMenuItem.Checked
    End With
    UpdateStatus() ' Titelleiste aktualisieren
End Sub

```

Letztendlich wird nur der Status *Checked* des Steuerelements bei jedem Mausklick invertiert (umschalten zwischen *True* und *False*). Ein Aufruf der *UpdateStatus*-Prozedur bewirkt dann die Aktualisierung der Titelzeile.

Nach diesen Vorbereitungen haben Sie bereits eine kleine Anwendung, die nach dem Übersetzen die betreffenden sichtbaren Steuerelemente enthält. Sie können Texte im Textfeld eintippen und sogar mit der Zwischenablage arbeiten (wird alles durch das Steuerelement bereitgestellt). Jetzt gilt es, die gewünschte Funktionalität zum Laden und Speichern von Textdateien sowie die restlichen Optionen hinzuzufügen.

Code zur Implementierung des Befehls Neu

Die Prozedur, die bei Anwahl des Befehls *Neu* (Menü *Datei* oder Schaltfläche oder Tastenkürzel) ausgeführt wird, soll den Inhalt des Textfelds löschen. Enthält das Dokumentfenster ungesicherte Änderungen, soll das in Abbildung 12.3 gezeigte Dialogfeld erscheinen. Der Benutzer kann dann entscheiden, wie zu verfahren ist (mit *Abbrechen* geht es zum Dokumentfenster zurück, *Nein* verwirft das Dokument und *Ja* öffnet einen *Speichern unter*-Dialog). Die Prozedur *NewDoc* wurde daher mit folgenden Anweisungen implementiert:

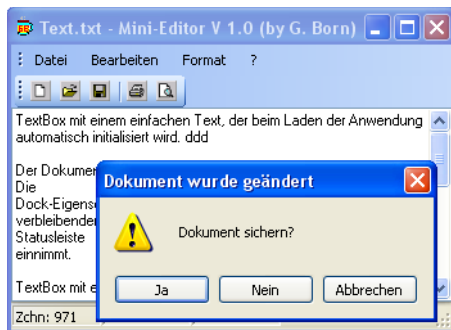


Abbildung 12.3: Dialog bei ungesicherten Änderungen

```
Private Sub NewDoc()  
    ' Hilfsprozedur, Aufruf für "Neues Dokument"  
    If SaveDirtyDoc() Then ' Änderungen sichern?  
        Me.TextBox1.Text = "" ' Textfeld löschen  
        file = "" ' kein Dateiname bekannt  
        docChanged = False ' Dokument unverändert  
        UpdateStatus() ' Statusleiste aktualisieren  
    End If  
End Sub
```

Letztendlich setzt diese Prozedur den Inhalt des *TextBox1*-Steuerelements über die Eigenschaft *Text* zurück. Zusätzlich werden noch die Flags für den Änderungsstatus und den Dateinamen auf die Initialisierungswerte gesetzt und dann die Aktualisierungsprozedur für Titel- und Statusleiste aufgerufen. Der Aufruf der Funktion *SaveDirtyDoc()*

innerhalb der *If*-Struktur stellt sicher, dass dem Benutzer bei ungesicherten Änderungen der Dialog aus Abbildung 12.3 gezeigt wird. Weiterhin übernimmt die Funktion *SaveDirtyDoc()* ggf. die Sicherung der Änderungen. Nur wenn die Funktion den Wert *True* zurückliefert, dürfen die Änderungen verworfen werden.

Diese Funktion *SaveDirtyDoc()* wird auch beim Aufruf der Funktion *OpenDoc()* benutzt und ist folgendermaßen implementiert:

```
Private Function SaveDirtyDoc() As Boolean
' Veranlasst eine Sicherung eines geänderten Dokuments
' Gibt True zurück, falls keine Änderungen mehr vorliegen
If docChanged = True Then    ' aktuelle Änderungen gesichert?
    ' dem Benutzer den "Dokument sichern"-Dialog zeigen
    tmp = MessageBox.Show("Dokument sichern?", _
        "Dokument wurde geändert", _
        MessageBoxButtons.YesNoCancel, _
        MessageBoxIcon.Warning)
    Select Case tmp
        Case System.Windows.Forms.DialogResult.Yes
            SaveDoc()          ' Dokument speichern
            If docChanged = True Then
                Return False ' Fehlschlag (Dialog wurde abgebrochen)
            Else
                Return True  ' gesichert
            End If
        Case System.Windows.Forms.DialogResult.No
            Return True      ' keine Sicherung
        Case System.Windows.Forms.DialogResult.Cancel
            Return False     ' Abbrechen
    End Select
Else
    Return True             ' keine ungesicherten Änderungen
End If
End Function
```

Über die *If*-Abfrage wird über das Flag *docChanged* geprüft, ob ungesicherte Änderungen vorliegen. Trifft dies nicht zu, terminiert die Funktion mit dem Wert *True* (und die rufende Anwendung darf das Dokument verwerfen). Bei ungesicherten Änderungen blendet die Funktion den in Abbildung 12.3 gezeigten Benutzerdialog ein. Die Auswahl *Abbrechen* veranlasst die Funktion den Wert *False* zurückzugeben (die Anwendung darf das Dokument nicht verwerfen). Mit der Auswahl *Nein* gibt die Funktion den Wert *True* zurück (Benutzer wünscht keine Sicherung, Dokument darf verworfen werden). Die Auswahl der *Ja*-Schaltfläche veranlasst die Funktion die Prozedur *SaveDoc* zum Sichern aufzurufen (die Prozedur wird weiter unten vorgestellt). Nach einer Sicherung ist der Funktionswert *True* zurückzugeben (rufende Anwendung darf dann das gesicherte

Dokument verwerfen). Leider kann der Benutzer den von *SaveDoc* angezeigten *Speichern unter*-Dialog über *Abbrechen* verlassen und die Änderung umgehen. Daher prüft eine zweite *If*-Anweisung nach Abarbeitung des *SaveDoc*-Aufrufs, ob *docChanged* wirklich *False* ist. Wurde die Sicherung nicht durchgeführt, terminiert die Funktion mit dem Rückgabewert *False* (und die Anwendung darf das Dokument nicht verwerfen). Die Kapselung der betreffenden Anweisungen in einer Funktion *SaveDirtyDoc* bewirkt, dass sich die Prüfung in *NewDoc*, *OpenDoc* etc. auf eine Anweisungszeile reduziert.

12.1.3 Öffnen-Dialog zur Anwendung hinzufügen

Die Implementierung eines *Öffnen*-Dialogs ist mit wenigen Handgriffen erledigt. Wurde im Formular ein *OpenFileDialog*-Steuerelement eingefügt, enthält der Code bereits die Anweisungen, um die Objektvariable samt Ereignisbehandlung zu deklarieren und zu instantiieren. Dann können Sie direkt über den Objektnamen des Steuerelements die Eigenschaften setzen. Alternativ lässt sich das Dialogobjekt mit folgenden Anweisungen instantiieren:

```
Dim OpenFileDialog1 As OpenFileDialog ' Objekt deklarieren
OpenFileDialog1 = New OpenFileDialog()
```

Anschließend lässt sich der folgende Code benutzen, um den Dialog anzuzeigen und den gewählten Dateinamen in eine Variable *file* zu speichern:

```
With OpenFileDialog1
    .AddExtension = True ' Dateiextension autom. einfügen
    .Filter = "Text (*.txt)|*.txt|" ' Dateiextensionen
    .ShowDialog() ' Dialog anzeigen
    file = .FileName ' Dateiname zurückgeben
    .Dispose() ' Objekt verwerfen
End With
```

Der Wert *True* in der *AddExtension*-Eigenschaft bewirkt, dass die Dateinamenerweiterung automatisch anhand des gewählten Dateityps ergänzt wird, falls der Benutzer nur den Dateinamen im Dialogfeld eintippt. Die Eigenschaft *Filter* spezifiziert, welche Dateinamenerweiterung das Dialogfeld *Öffnen* im Listefeld *Dateityp* bereitstellt (Abbildung 12.4). Die Methode *ShowDialog()* veranlasst die Anzeige des Dialogs. Sobald der Benutzer das Dialogfeld *Öffnen* schließt, lässt sich der gewählte Dateiname (samt Pfad) über die Eigenschaft *FileName* abfragen. Wurde das Dialogfeld über die Schaltfläche *Abbrechen* beendet, enthält die Eigenschaft eine leere Zeichenkette. Mit der *Dispose*-Methode wird das Dialogfeld verworfen.

Hinweis

Das *Dialog*-Objekt liefert über die Eigenschaft *DialogResult* einen Hinweis, ob das Dialogfeld über die Schaltfläche *Abbrechen* verlassen wurde. Ich benutze aber die Prüfung auf einen leeren Dateinamen, da der komplette Code zur Verwaltung des *Dialog*-Objekts in eine Funktion verlagert wurde und die rufende Anwendung nicht an die Eigenschaft *DialogResult* herankommt. Bei Bedarf können Sie die Funktion ja anders implementieren.

Verwenden Sie die Eigenschaft *InitialDirectory*, um beim Öffnen des Dialogfelds (*Open*, *SaveAs*) einen Ordner vorzugeben. Die Eigenschaft *MultiSelect* = *True* erlaubt dem Benutzer mehrere Dateien zu markieren. Dann lassen sich die Dateinamen über die *FileNames*-Auflistung abfragen. Diese und weitere Eigenschaften sind in der Hilfe des .NET Framework dokumentiert.

Im Beispiel habe ich den kompletten Code zum Abwickeln des Öffnen-Aufrufs in eine Funktion *OpenF()* verlagert. Diese wird dann beim Auftreten eines *Click*-Ereignisses der Schaltflächen *Öffnen* oder des Menüs *Datei/Öffnen* über die Ereignisbehandlungsroutine und die benutzerspezifische Prozedur *OpenDoc* aufgerufen.

Hinweise zur Gestaltung des Filterkriteriums

Der Ausdruck der Eigenschaft *Filter* besitzt für jeden Dateityp zwei durch das Zeichen | getrennte Kriterien. Der erste Teilausdruck gibt den angezeigten Klartextnamen an, der zweite Teilausdruck definiert den intern zur Anzeige benutzten Filter.

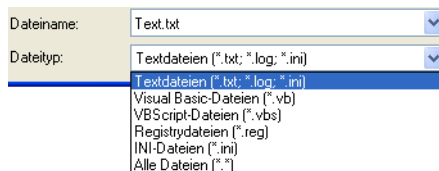


Abbildung 12.4: Dateitypen in einem Dateidialog

Der folgende Ausdruck für *Filter* erzeugt die in Abbildung 12.4 gezeigten Einträge im Listenfeld *dateityp*:

```
.Filter = "Textdateien (*.txt; *.log; *.ini)|*.txt;*.log;*.ini|" & _
         "Visual Basic-Dateien (*.vb)|*.vb|" & _
         "VBScript-Dateien (*.vbs)|*.vbs|" & _
         "Registrydateien (*.reg)|*.reg|" & _
         "INI-Dateien (*.ini)|*.ini|" & _
         "Alle Dateien (*.*)|*.*"
```

Tipp

In der Filtereigenschaft für das Dialogfeld *Öffnen* dürfen durchaus mehrere Dateinamenerweiterungen in einer Zeile zusammengefasst werden. Dies ist beim Kriterium »Textdateien (*.txt;*.log;*.ini) | *.txt;*.log;*.ini | « der Fall. Beim Filterkriterium für das Dialogfeld *Speichern unter* dürfen Sie aber nur einen Dateityp pro Eintrag verwenden, da sonst die Dateinamenerweiterung beim Speichern nicht mehr eindeutig ist. Aus diesem Grunde habe ich in der Beispielanwendung den Ausdruck für die Filterkriterien in zwei globalen Konstanten *rFilter* (für Lesen) und *wFilter* (für Schreiben) im Klassenkopf deklariert.

Einlesen einer Textdatei in das TextBox-Steuerelement

Im Vorgriff auf das Kapitel zur Dateibehandlung möchte ich kurz erläutern, wie sich die ausgewählte Textdatei lesen lässt. Lesende Zugriffe auf Dateien erfolgen über die *StreamReader*-Klasse, die sich im Namensraum *System.IO* befindet. Die folgende Sequenz definiert eine Instanz des *StreamReader*-Objekts. Dabei wird der Dateiname beim Aufruf des *New*-Konstruktors als Parameter übergeben.

```
Dim oFile As IO.StreamReader ' Dateivariablen deklarieren
oFile = New IO.StreamReader(file) ' Datei öffnen
txt = oFile.ReadToEnd ' lese die Textdatei in txt ein
```

Anschließend lässt sich die *ReadToEnd*-Methode benutzen, um den kompletten Inhalt der Textdatei zu lesen. Für das Beispiel habe ich die *OpenDoc*-Prozedur um eine Fehlerbehandlung (bei nicht existierenden Dateien) sowie um Aufrufe der Funktionen *SaveDirtyDoc* (sichert geänderte Dokumente) und *OpenF* (ermittelt den Dateinamen) ergänzt. Die Details sind nachfolgendem Codeausschnitt zu entnehmen:

```
Private Sub OpenDoc()
' Hilfsprozedur, Aufruf beim Öffnen des Dokuments
If SaveDirtyDoc() Then ' Änderungen sichern?
file = OpenF() ' Zeige Öffnen-Dateidialog
If file <> "" Then ' hat der Benutzer eine Datei gewählt?
Dim oFile As IO.StreamReader ' Dateivariablen deklarieren
Try ' Laufzeitfehler abfangen
oFile = New IO.StreamReader(file) ' Datei öffnen
' lese die Textdatei in das TextBox-Element ein
Me.TextBox1.Text = oFile.ReadToEnd
Catch e As Exception ' Fehlerzweig
MessageBox.Show(e.Message, "Fehler beim Öffnen", _
MessageBoxButtons.OK, MessageBoxIcon.Error)
Beep() ' akustische Warnung
Exit Sub ' Prozedur verlassen
```

Listing 12.1: Einlesen einer Textdatei

```

End Try
oFile.Close()      ' Datei schließen
docChanged = False ' Dokumentinhalt ungeändert
UpdateStatus()     ' Statusleiste aktualisieren
End If
End If
End Sub

```

Listing 12.1: Einlesen einer Textdatei (Forts.)

12.1.4 Realisierung der Speichern-Funktion

Die Anweisungen zum Speichern des Dokumentinhalts sind ähnlich wie beim Öffnen eines Dokuments. Daher werden die Details hier etwas knapper erläutert.

Anzeige des Speichern unter-Dialogs

Im ersten Schritt ist dem Benutzer das Dialogfeld *Speichern unter* anzuzeigen, dies wird im Beispiel mit folgender Funktion *SaveF()* realisiert:

```

Private Function SaveF() As String
    ' Hilfsfunktion zum Speichern unter
    ' gibt den Dateinamen oder "" zurück
    Dim oDlg As SaveFileDialog      ' Objekt deklarieren
    oDlg = New SaveFileDialog()     ' neue Instanz holen
    oDlg.AddExtension = True       ' Dateiextension autom. einfügen
    oDlg.Filter = wFilter          ' Filter für Dateiextensionen
    oDlg.ShowDialog()              ' Dialog anzeigen
    Return oDlg.FileName           ' Dateiname zurückgeben
    oDlg.Dispose()                 ' Objekt verwerfen
End Function

```

Der obige Code ist bereits so gehalten, dass er auch direkt, ohne *SaveFileDialog*-Steuerelemente in einer Quelldatei eingebunden werden kann. Falls Sie im Designer ein *SaveFileDialog*-Steuerelement im Formular eingefügt haben, sollten Sie dieses über die *Name*-Eigenschaft mit *oDlg* benennen. Dann können die beiden ersten Anweisungen zum Deklarieren der Objektvariablen *oDlg* und zum Anlegen der Objektinstanz über *New* entfallen (der Designer generiert den betreffenden Code bereits automatisch). Ansonsten gelten meine Ausführungen zum *OpenFileDialog*-Steuerelement auch sinngemäß für den *SaveFileDialog*.

Speichern des Dokuments

Um einen Text in eine bereits bestehende Textdatei zu speichern, sind die folgenden Anweisungen ausreichend:

Namespace System

```

...
Dim oFile As IO.StreamWriter      ' Dateivariablen deklarieren
oFile = New IO.StreamWriter(file) ' Datei öffnen
oFile.WriteLine(txt)              ' speichere Text in Datei
oFile.Close()

```

Die ersten beiden Anweisungen deklarieren eine Objektvariable *oFile* und erzeugen eine Objektkonstante (die auf die angegebene Datei verweist). Dann lässt sich die *WriteLine()*-Methode des *StreamWriter*-Objekts zum Schreiben des im Argument übergebenen Texts benutzen. Die *Close*-Methode schließt danach die Datei.

Hinweis

Zum Schreiben der Texte können Sie alternativ zu *WriteLine()* auch die *Write()*-Methode verwenden. Die Methoden unterscheiden sich nur insofern, als *WriteLine()* am Ende noch einen Zeilenumbruch einfügt.

Bei einer neuen Datei müssen Sie eine leicht modifizierte Anweisungsfolge verwenden, die die neue Datei anlegt.

```

Dim oStream As IO.StreamWriter ' Streamobjekt
' Öffne neue Datei und lege diese an
oFile = New IO.FileStream(file1, IO.FileMode.Create)
oStream = New IO.StreamWriter(oFile) ' Stream-Objekt holen
oStream.WriteLine(txt)               ' speichere Text in Datei
oStream.Close()                     ' Datei schließen

```

Der *FileStream*-Konstruktor erwartet im ersten Parameter den Dateinamen samt Pfad. Der zweite Parameter muss eine Konstante für den *Open*-Modus aufweisen. Ich habe hier *FileMode.Create* benutzt, damit ggf. auch eine bestehende Datei überschrieben werden kann (falls der Benutzer dies im *Speichern unter*-Dialog so gewählt hat). Dann wird eine Instanz der *StreamWriter*-Klasse geöffnet, wobei dieser die Objektvariable der neuen Datei übergeben wird. Dies erlaubt es über die *WriteLine()*-Methode des *oStream*-Objekts auf die neue Datei schreibend zuzugreifen.

In der Beispielanwendung wurde der obige Code in zwei Prozeduren eingebettet und um Funktionen zur Fehlerbehandlung erweitert. Die Prozedur *SaveDoc* speichert den aktuellen Text in eine bestehende Datei. Stellt die Prozedur fest, dass noch kein Dateiname existiert, wird einfach die zweite *SaveAsDoc*-Prozedur aufgerufen. Diese Prozedur öffnet über die Funktion *SaveF()* einen *Speicher unter*-Dialog und fordert den Dateinamen an. Danach wird die angegebene Datei angelegt und der Dokumentinhalt gesichert. Durch die Aufteilung in diese beiden Prozeduren lassen sich diese sehr einfach über die *Click*-Ereignisbehandlungsroutinen der Menübefehle *Datei/Speichern*, *Datei/Speichern unter* und der Schaltfläche *Speichern* aufrufen.

Hinweis

Die Prozeduren *SaveDoc* und *SaveAsDoc* übernehmen zusätzliche Verwaltungsaufgaben. So muss das *docChanged*-Flag bei jedem Speichern auf *False* gesetzt werden. Bei einer neuen Datei ist der beim Speichern benutzte Dateiname zu protokollieren. Da ein Aufruf des *Speichern unter*-Dialogs vom Benutzer über die Schaltfläche *Abbrechen* beendet werden kann, verwendet *SaveAsDoc* intern die Variable *file1* für den Dateinamen und weist dann den Wert später der globalen Variablen *file* zu. Im Beispielcode schließt die betreffende Prozedur die Datei, sobald deren Inhalt gelesen bzw. gesichert wurde. Vergessen Sie beispielsweise den *Close*-Aufruf beim Lesen, lässt sich die Datei später nicht mehr sichern (da diese noch vom Lesen in Zugriff ist). In realen Anwendungen kann es erforderlich sein, die Datei während der Bearbeitung geöffnet zu halten (um den Schreibzugriff durch andere Benutzer zu unterbinden). Dann müsste das Öffnen und Schließen zentral verwaltet werden. Details der Implementierung entnehmen Sie bitte dem auf der CD enthaltenen Beispiel (Datei *Form1.vb* im Ordner *\Beisp\Kap12\Dateidialoge*).

12.1.5 Anzeige eines Font-Dialogs

Zur Auswahl der Schriftart und der zugehörigen Eigenschaften bietet Windows das Dialogfeld *Schriftart* (Abbildung 12.5). In dem von mir benutzten Beispiel soll sich dieses Dialogfeld über den Befehl *Format/Schriftart* öffnen lassen. Die Benutzerauswahl wirkt sich dann auf den in der TextBox angezeigten Inhalt aus. Dies erfordert nur wenige Zeilen Programmcode, die ich hier in der Ereignisbehandlungsroutine des *Click*-Ereignisses des Menüelements hinterlegt habe.

```
Private Sub StatusleisteTeilenToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles StatusleisteTeilenToolStripMenuItem.Click
    ' Menübefehl Schriftart angeklickt
    FontDialog1.ShowColor = True      ' Farbauswahl erlauben
    FontDialog1.AllowScriptChange = False ' Kein Skript-Feld
    If FontDialog1.ShowDialog() <> _
        System.Windows.Forms.DialogResult.Cancel Then
        TextBox1.Font = FontDialog1.Font ' zuweisen
        TextBox1.ForeColor = FontDialog1.Color
    End If
End Sub
```

Die beiden ersten Anweisungen steuern über die Eigenschaften, ob eine Farbauswahl (Schriftfarbe) und die Option *Skript* (erlaubt die Änderung der Schriftart im Kombinationsfeld *Skript*) zulässig ist. Die *ShowDialog()*-Methode zeigt das Dialogfeld an. Die Methode gibt das Ergebnis *DialogResult* zurück. Ist dieses ungleich *Cancel*, lässt sich das *Font*-Objekt über die Eigenschaft *Font* des *FontDialog1*-Objekts zurücklesen. Das *Font*-Objekt stellt die einzelnen Optionen wie Schriftart, Schriftgrad, Farbe etc. über Eigenschaften bereit. In obi-

ger Prozedur wird der Wert aber direkt der Eigenschaft *Font* des *TextBox1*-Objekts zugewiesen – die Benutzerauswahl wirkt sich also sofort im Textfeld aus.

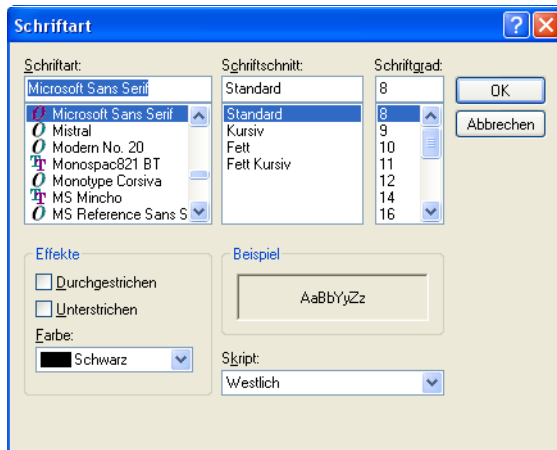


Abbildung 12.5: Dialog zur Auswahl der Schriftart

Die obige Prozedur setzt voraus, dass das Steuerelement *FontDialog1* im Formularentwurf eingefügt wurde. Dann generiert der Designer automatisch den Code zur Deklaration und Instantiierung. Andernfalls fügen Sie die Anweisung

```
Friend WithEvents ColorDialog1 As System.Windows.Forms.ColorDialog
```

im Deklarationsteil der Klasse und im *New*-Konstruktor die folgende Anweisung ein:

```
Me.FontDialog1 = New System.Windows.Forms.FontDialog()
```

Ich habe hier die Langformen gewählt, die den benutzten Namespace enthalten. Falls der betreffende *Imports*-Befehl im Programmkopf hinterlegt sind, lässt sich auch die Kurzform verwenden.

12.1.6 Anzeige des Dialogs zur Farbauswahl

Ähnlich wie Schriftarten lassen sich Farben unter Windows mittels eines Dialogfelds *Farbe* abrufen (Abbildung 12.6). Der Benutzer kann eine Farbe wählen. Wird das Dialogfeld über die *OK*-Schaltfläche geschlossen, gibt dieser einen Farbwert (RGB-Wert) zurück. Die betreffende Funktionalität lässt sich in .NET Framework über eine eigene Klasse *ColorDialog* ansprechen. Das hier entwickelte Beispiel erlaubt den Aufruf des Dialogfelds über den Befehl *Format/Farben*. Die Anwendung färbt anschließend den Text der *TextBox* in der gewählten Farbe ein. Auch hier habe ich den Code in die *Click*-Ereignisbehandlungsroutine des Menüelements hinterlegt.



Abbildung 12.6: Dialog zur Farbauswahl

```
Private Sub StatusAktualisierenToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles StatusAktualisierenToolStripMenuItem.Click
    ' Menübefehl Farben angeklickt
    If ColorDialog1.ShowDialog <> _
        System.Windows.Forms.DialogResult.Cancel Then _
        TextBox1.ForeColor = ColorDialog1.Color
End Sub
```

Listing 12.2: Textfarbe über *Farbe*-Dialog wählen und zuweisen

12.1.7 Rückgängig machen einer TextBox-Änderung

Die Anwendung erlaubt über den Befehl *Bearbeiten/Rückgängig* bzw. über die Tastenkombination **Strg**+**Z** die Rücknahme der letzten Änderung im Textfeld. Die Implementierung ist recht einfach, die *TextBox*-Klasse unterstützt dies bereits.

```
If Me.TextBox1.CanUndo Then _
    Me.TextBox1.Undo() ' Rückgängig machen
```

Über die *CanUndo*-Eigenschaft lässt sich prüfen, ob das Steuerelement die Rücknahme der letzten Änderung unterstützt. Trifft dies zu, lässt sich die Änderung über die *Undo*-Methode aufheben.

Hinweis

Im Beispiel wurde der leicht modifizierte Code in der *Click*-Ereignisbehandlungsroutine *RückgängigToolStripMenuItem_Click* des Menüelements hinterlegt. Die Routine soll den Befehl *Rückgängig* im Menü *Bearbeiten* sperren, falls nichts mehr rücknehmbar ist. Allerdings habe ich festgestellt, dass das Steuerelement immer *CanUndo* = *True* zurückmeldet, da zurückgenommene Änderungen nach einem zweiten Aufruf von *Undo* wieder vorhanden sind (gleiches Verhalten wie beim Windows-Editor).

12.1.8 Schließen bei ungesicherten Änderungen verhindern

Beenden Sie eine Anwendung wie den Windows-Editor, Microsoft Word etc. und enthält das Dokumentfenster noch ungesicherte Dokumentänderungen, erscheint eine Sicherheitsabfrage, die Ihnen Gelegenheit zum Sichern der Änderungen gibt. Diese Sicherheitsabfrage wurde bereits weiter oben im Zusammenhang mit dem Laden neuer Dokumente besprochen. Im ersten Schritt reicht es, einen Aufruf der *SaveDirtyDoc()*-Funktion in die *Click*-Ereignisbehandlungsroutine des Befehls *Datei/Beenden* zu platzieren.

```
If SaveDirtyDoc() Then Application.Exit() ' Anwendung beenden
```

Leider gibt es noch einen Haken. Eine Anwendung lässt sich auch über die Schaltfläche *Schließen* in der rechten oberen Ecke des Fensters sowie über das Systemmenü beenden. Um auch dies abzufangen, müssen Sie auf das *FormClosing*-Ereignis reagieren. Dieses wird beim Schließen des Anwendungsformulars ausgelöst. Nachfolgend sehen Sie den Code der Variante, die im aktuellen Beispiel hinterlegt ist:

```
Private Sub Form1_FormClosing(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.FormClosingEventArgs) _
    Handles Me.FormClosing
    ' Beim Schließen des Formulars
    If SaveDirtyDoc() Then ' geändertes Dokument sichern?
        e.Cancel = False ' Beenden zulassen
    Else
        e.Cancel = True ' Beenden sperren
    End If
End Sub
```

Der Ereignisbehandlungsroutine wird ein *System.ComponentModel.CancelEventArgs*-Objekt übergeben. Dessen *Cancel*-Eigenschaft lässt sich verwenden, um das Beenden zu steuern. Wird *Cancel* in der Ereignisbehandlungsroutine auf *False* gesetzt, terminiert die Anwendung. Ändern Sie den Wert aber auf *True*, bricht die Anwendung die Terminierung ab. In der Beispielanwendung bewirkt der obige Code, dass der Benutzer beim Beenden der Anwendung Gelegenheit zur Sicherung der Änderungen bekommt.

Hinweis

Nach diesen etwas umfangreicheren Ausführungen möchte ich das Beispiel schließen. Die Dialoge zum Drucken oder zum Anzeigen der Vorschau erfordern etwas umfangreicheren Code (siehe *Kapitel 16*). Sie finden das komplette Projektbeispiel im Ordner *\Beisp\Kap12\Dateidialoge* der Begleit-CD. Die Implementierungsdetails entnehmen Sie der Datei *Form1.vb* in diesem Ordner.

Auf der CD finden Sie noch ein weiteres Beispiel im Ordner *\Beisp\Kap11\RTF-App*, wobei die Datei *Form1.vb* den Code der Anwendung enthält. Dieses Beispiel wurde aus *Dateidialoge* abgeleitet. Das komplette Projekt wurde kopiert; anschließend habe ich das *TextBox1*-Steuerelement durch ein *RichTextBox*-Steuerelement ersetzt.

Hinweis (Fortsetzung)

Da die *Name*-Eigenschaft des neuen Steuerelements im Eigenschaftenfenster auf *TextBox1* korrigiert wurde, lässt sich der bestehende Code weitgehend 1:1 weiterverwenden. Lediglich der Rumpf der *TextChanged*-Ereignisbehandlungsprozedur musste neu erstellt und dann der Code der alten Prozedur eingefügt werden.

12.2 Verwenden des FolderBrowser-Dialogs

Zur Auswahl von Ordnern benutzen Windows-Anwendungen häufig den *BrowseForFolder*-Dialog des Betriebssystems. Dieser Dialog lässt sich in .NET-Anwendungen über das *FolderBrowser*-Steuerelement aufrufen. Die Verwendung soll jetzt an einem einfachen Beispiel demonstriert werden (Abbildung 12.7).

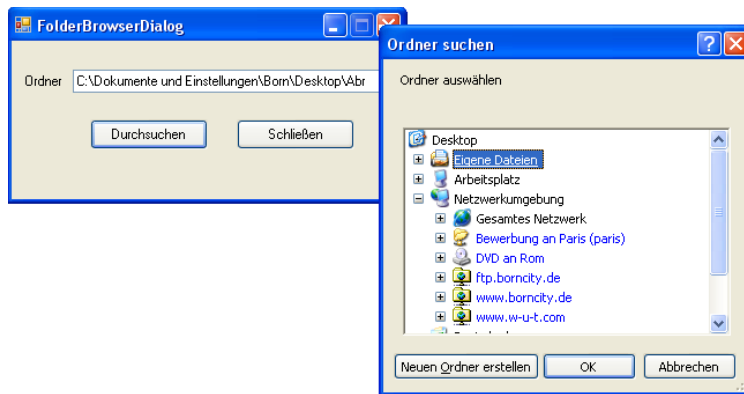


Abbildung 12.7: Beispiel mit Dialogfeld *Ordner suchen*

Klickt der Benutzer im Beispielformular auf die Schaltfläche *Durchsuchen*, öffnet sich ein Dialogfeld *Ordner suchen*, in dem der Pfad zum Desktop bereits voreingestellt ist. Der Benutzer kann dann im Dialogfeld zu einem Ordner navigieren und/oder einen neuen Ordner anlegen. Beim Schließen des Dialogfelds wird der ausgewählte Pfad in das Textfeld des Formulars übertragen. Wurde die Schaltfläche *Abbrechen* angeklickt oder hat der Benutzer eine FTP-Adresse im Zweig *Netzwerkumgebung* gewählt, erscheint dagegen ein Dialogfeld mit einer entsprechenden Fehlermeldung. Die Implementierung dieses Beispiels ist in der Entwicklungsumgebung relativ einfach und mit wenigen Schritten erledigt.

1. Legen Sie in der Entwicklungsumgebung ein neues Beispielprojekt mit einem einfachen Formular an und fügen Sie die Schaltflächen *Durchsuchen*, *Schließen* sowie ein Textfeld hinzu.
2. Fügen Sie aus der Toolbox ein *FolderBrowserDialog*-Steuerelement zum Formular hinzu. Das Steuerelement wird am unteren Rand des *Ansicht-Design*-Fensters eingeblendet.
3. Setzen Sie anschließend die Eigenschaften des *FolderBrowserDialog*-Steuerelements und fügen Sie danach den benötigten Code zu den Ereignisbehandlungsroutinen der Schaltflächen hinzu.

Die Eigenschaften des *FolderBrowserDialog*-Steuerelements können Sie im Eigenschaftsfenster der Entwicklungsumgebung oder direkt im Programmcode setzen. Der Dialogfeldtitel wird über die Eigenschaft *Description* bestimmt, während die Eigenschaft *RootFolder* den Startpfad (z.B. Desktop) des Dialogfelds vorgibt. Falls Sie den Pfad im Code definieren, achten Sie darauf, dass nur Werte der Enumeration *System.Environment.SpecialFolder.Desktop* verwendet werden. Die Eigenschaft *ShowNewFolderButton* legt mit dem Wert *true* fest, dass die Schaltfläche *Neuer Ordner* im Dialogfeld eingeblendet wird. Ein vom Benutzer ausgewählter Pfad wird in der Eigenschaft *SelectedPath* hinterlegt.

Die *Click*-Ereignisprozedur der *Schließen*-Schaltfläche des Beispiels enthält nur die Anweisung *Me.Close()* zum Schließen des Formulars. Der Code in der *Click*-Ereignisprozedur der *Durchsuchen*-Schaltfläche ist nachfolgendem Listing zu entnehmen:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    ' Durchsuchen-Schaltfläche angeklickt
    ' Rufe FolderBrowserDialog auf
    With Me.FolderBrowserDialog1
        ' hier können die Eigenschaften gesetzt werden
        .Description = "Ordner auswählen" ' Dialogfeld-Titel
        .ShowNewFolderButton = True ' Neue Ordner anlegbar
        ' voreingestellter Ordner
        .RootFolder = System.Environment.SpecialFolder.Desktop
        .SelectedPath = ""
        ' Dialog anzeigen
    If .ShowDialog() = Windows.Forms.DialogResult.OK Then
        If .SelectedPath <> "" Then ' gibt es einen Pfad?
            Me.TextBox1.Text = .SelectedPath ' Pfad zuweisen
        Else
            MessageBox.Show("Kein Pfad gefunden," & vbCrLf & _
                "Eventuell eine FTP-Adresse gewählt?", _
                "Fehler bei Ordnerauswahl", _
                MessageBoxButtons.OK)
        End If
    Else
        MessageBox.Show("Kein Pfad ausgewählt," & vbCrLf & _
            "Abbrechen gewählt?", _
            "Fehler bei Ordnerauswahl", _
            MessageBoxButtons.OK)
    End If
    End With
End Sub
```

Listing 12.3: Nutzen des FolderBrowser-Diialogs

In der Prozedur werden die Eigenschaften des Steuerelements gesetzt (der Code zur Instantiierung wird automatisch durch die Entwicklungsumgebung beim Einfügen des Steuerelements generiert). Anschließend bewirkt der Aufruf der *ShowDialog*-Methode die Anzeige des Steuerelements. Die Methode gibt den Wert *Windows.Forms.DialogResult.OK* zurück, falls der Benutzer den Dialog über die OK-Schaltfläche schließt. In diesem Fall weist der Programmcode den gewählten Pfad dem Textfeld im Formular zu. Allerdings gibt es noch eine Besonderheit zu beachten. Falls der Benutzer eine FTP-Adresse in der Netzwerkumgebung gewählt hat, ist der Pfad leer. Daher fängt der Code diesen Fall ab und zeigt einen Fehlerdialog. Das Gleiche gilt, falls der Benutzer das Dialogfeld über die *Abbrechen*-Schaltfläche verlässt.

Hinweis

Sie finden die Projektdateien dieses Beispiels im Ordner `\Beisp\Kap12\FolderBrowser-Dialog` der Begleit-CD.

12.3 MDI-Anwendungen erstellen

Viele Windows-Anwendungen unterstützen das Multiple Document Interface (MDI), d.h., Sie können mehrere Dokumente in einem Anwendungsfenster verwalten. Das Anwendungsfenster fungiert dabei als ein übergeordnetes MDI-Formular, welches dann die untergeordneten MDI-Fenster (Unterfenster) verwaltet. Nachfolgend möchte ich an einem sehr einfachen Beispiel zeigen, wie Sie eine solche Anwendung realisieren können.

Abbildung 12.8 zeigt zwei Varianten des Anwendungsfensters des in diesem Abschnitt zu entwickelnden MDI-Beispiels. Die Anwendung kann, wie im Beispiel *Dateidialoge*, Textdateien laden und speichern. Im Menü *Datei* finden Sie Befehle zum Anlegen neuer Dokumente, zum Öffnen, Speichern und Schließen (die Befehle zum Drucken und zur Seitenvorschau sind noch nicht implementiert). Zusätzlich unterstützen die drei linken Schaltflächen der Symbolleiste das Anlegen, Öffnen und Speichern von Dokumenten. Jedes Dokument wird dabei in einem eigenen Fenster geöffnet. Die Fenster lassen sich über die Befehle des Menüs *Fenster* anordnen und in den Vordergrund holen. Die Verwaltung der Befehle mit den Namen der Dokumentfenster erfolgt dabei automatisch durch die MDI-Steuerung. Maximieren Sie ein Dokumentfenster, wird der Fenstertitel ebenfalls durch die MDI-Steuerung mit in der Titelleiste des Anwendungsfensters eingeblendet. Die Befehle wie *Speichern*, *Speichern unter* und *Schließen* wirken sich immer auf das aktuelle Dokumentfenster aus. Mit *Alle schließen* lassen sich alle geöffneten Dokumentfenster schließen. Im Menü *Bearbeiten* finden Sie Befehle wie *Ausschneiden*, *Kopieren* und *Einfügen*, die den Datenaustausch markierter Bereiche mit der Zwischenablage unterstützen. Die Befehle *Löschen* und *Alles markieren* erlauben markierte Bereiche zu löschen bzw. das gesamte Dokument zu markieren. Im Menü *Format* erlaubt der Befehl *Schriftart* die Formatierung markierter Textbereiche.

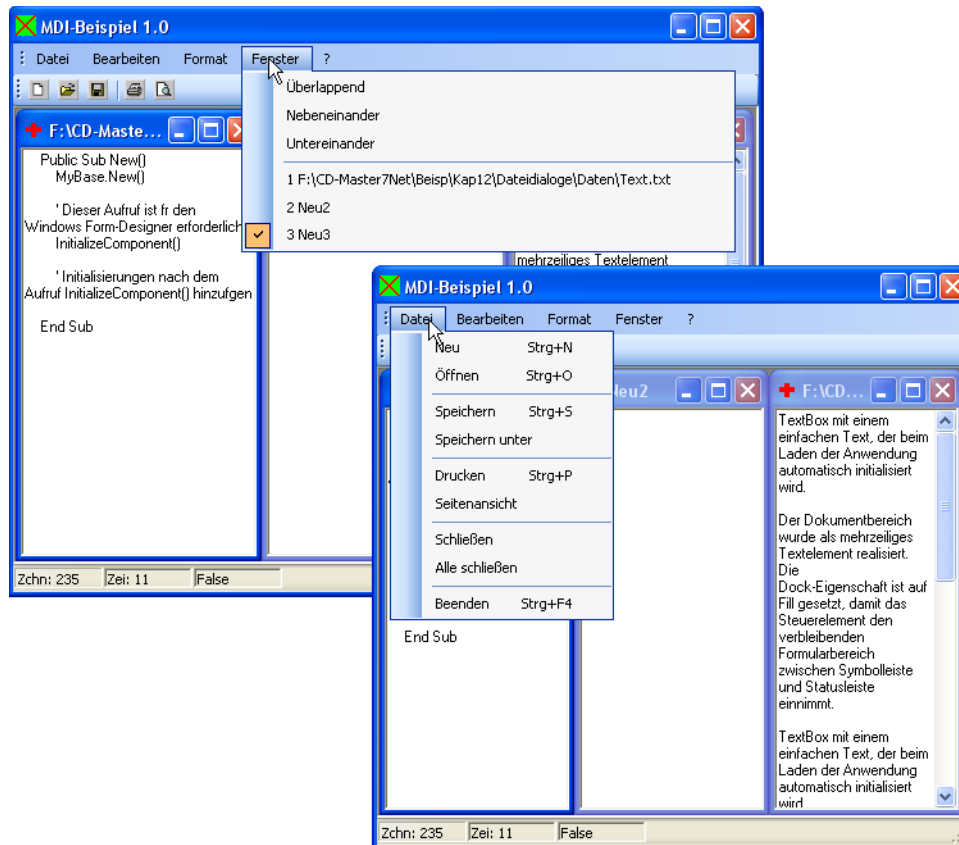


Abbildung 12.8: Fenster des MDI-Beispiels und Menüvarianten

Die Anwendung wurde mit einem *RichTextBox*-Steuerelement versehen, um die betreffende Funktionalität bereitzustellen. Dadurch können Sie per Zwischenablage formatierte Texte übernehmen. Zusätzlich habe ich die Lese-/Schreibroutinen der vorhergehenden Beispiele so modifiziert, dass auch einfache RTF-Dateien gelesen bzw. gesichert werden können. Das Beispiel bietet daher eine gute Gelegenheit, weitere Programmier-techniken für Anwendungen zu diskutieren.

Um die Anwendung zu implementieren, sind allerdings verschiedene Schritte erforderlich, die typisch für MDI-Anwendungen sind. Nachfolgend werden die Schritte zur Implementierung der Beispiel MDI-Anwendung in der Entwicklungsumgebung besprochen.

- Legen Sie im ersten Schritt ein neues Projekt als Windows-Anwendung an und entwerfen Sie das Formular für das Hauptfenster (MDI-Parent). Dieses Fenster enthält alle sichtbaren Steuerelemente wie Menü-, Symbol- und Statusleisten, die sich nicht auf das Dokumentfenster beziehen.
- In einem zweiten Schritt ist dann ein weiteres Formular (MDI-Client) im Projekt hinzuzufügen, welches die Steuerelemente des Dokumentfensters aufnimmt.

Nach diesen Vorbereitungen der visuellen Komponenten müssen Sie den Code zur Implementierung der Programmfunktionalität zu den beiden Formularen hinzufügen. Der Code wird dabei je nach Funktionalität im MDI-Client oder in der Hauptanwendung hinterlegt. Dies werde ich weiter unten noch näher beleuchten.

12.3.1 Vorbereitungen für das übergeordnete Formular

Das übergeordnete Formular, welches als Fenster der Anwendung die MDI-Dokumente aufnimmt, lässt sich im Designer mit wenigen Handgriffen realisieren. Wichtig ist dabei nur, dass Sie bestimmte Randbedingungen beachten.

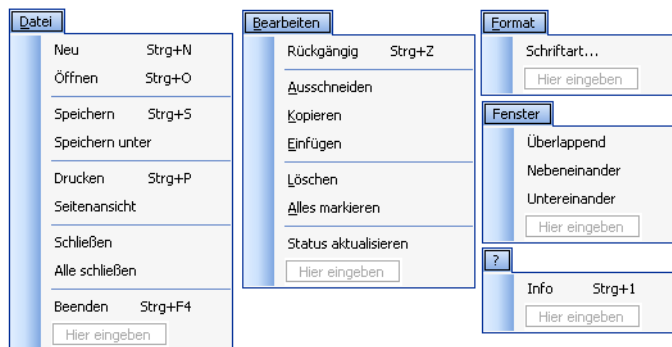


Abbildung 12.9: Menüs des Hauptfensters

1. Legen Sie in der Entwicklungsumgebung ein neues Windows-Projekt an und setzen Sie die Eigenschaften für das Formular. Bei Bedarf können Sie die *WindowState*-Eigenschaft auf *Maximized* einstellen, um das Fenster beim Start der Anwendung auf den Vollbildmodus zu vergrößern.
2. Setzen Sie abschließend die *IsMDIContainer*-Eigenschaft im Eigenschaftenfenster des Formulars auf *True*. Dies ist wichtig, da das Formular dadurch als MDI-Container für untergeordnete Fenster gekennzeichnet wird.
3. Fügen Sie anschließend eine Menüleiste zum Hauptformular hinzu und konfigurieren Sie in dieser Menüleiste die in Abbildung 12.9 gezeigten Einträge. Sie können dabei das alte *MenuBar*- oder das neue *MenuStrip*-Steuerelement verwenden. Wichtig ist, dass die Menüleiste die Einträge *&Datei* und *&Fenster* aufweist und dass im Menü *Datei* der Befehl *&Neu* und ggf. der Befehl *&Schließen* enthalten ist. Bei der Benennung der Menüelemente sollten Sie auf Konsistenz mit den Windows-Konventionen achten. Der Befehl *Neu* wird zum Anlegen neuer Dokumentfenster benötigt, der Befehl *Schließen* erlaubt ein Dokumentfenster zu schließen (obwohl dies auch über die Schaltfläche *Schließen* in der rechten oberen Ecke des Dokumentfensters möglich ist). Die Schritte zur Konfigurierung eines Hauptmenüs haben Sie bereits in den vorhergehenden Beispielen kennen gelernt.

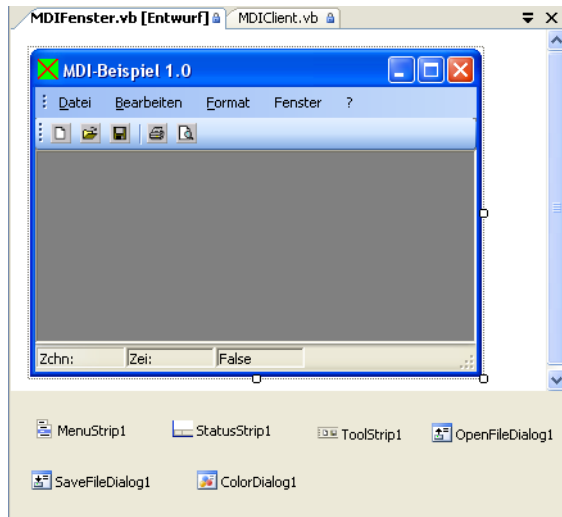


Abbildung 12.10: Formularentwurf des Hauptfensters

4. Wechseln Sie nun zum Menüelement mit dem Namen *&Fenster* und markieren den Eintrag in der Menüleiste. Verwenden Sie das alte *MenuBar*-Steuerelement und setzen Sie in dessen Eigenschaftfenster den Wert der Eigenschaft *MDIList* auf *True*. Beim neuen *MenuStrip*-Steuerelement müssen Sie dagegen das komplette Steuerelement wählen und dann zu dessen Eigenschaftfenster wechseln. Stellen Sie dort den Wert der Eigenschaft *MdiWindowListItem* über das Listefeld auf den Objektnamen des Menüs *Fenster* (im aktuellen Beispiel ist dies der Objektname *FensterToolStripMenuitem*). Dieser Zwischenschritt ist wichtig, da diese beiden Eigenschaften das Menü *Fenster* an die Verwaltung der MDI-Client-Fenster anbinden. Zur Laufzeit blendet das .NET Framework dann im Menü *Fenster* automatisch die Tittleistentexte der geöffneten MDI-Clients ein. Der Benutzer kann dann zwischen den Fenstern umschalten.
5. Fügen Sie anschließend eine Symbolleiste und die Statusleiste zum Hauptformular hinzu. Statten Sie die Symbol- und Statusleiste mit den in Abbildung 12.10 gezeigten Elementen aus und fügen Sie die Steuerelemente für die Dialoge zum Formular hinzu. Diese Schritte haben Sie bereits in den vorhergehenden Beispielen kennen gelernt.

Mit diesen Schritten haben Sie das MDI-Anwendungsfenster gestaltet und Sie können zum Entwurf des MDI-Clients übergehen.

Hinweis

Im Vergleich zu bisherigen Projekten gibt es drei Abweichungen. Das Hauptformular enthält kein Steuerelement zur Aufnahme des Dokuments. Beim Formular muss die Eigenschaft *IsMDIContainer*-Eigenschaft auf *True* gesetzt werden. Und Sie benötigen ein Menü, dessen *MdiWindowListItem*-Eigenschaft (bzw. die *MDIList*-Eigenschaft des alten *StatusBar*-Steuerelements) auf *True* zu setzen ist.

Tipp

Im aktuellen Beispiel wurde das MDI-Parent-Formular manuell in der Entwicklungsumgebung entworfen. Wenn Sie im Projektmappen-Explorer die Projektdatei mit der rechten Maustaste anklicken und die Kontextmenübefehle *Hinzufügen/Neues Element* wählen, erscheint das Dialogfeld *Neues Element*. Dort finden Sie eine komplette Vorlage *Übergeordnetes MDI-Formular*, die ein Formular mit einer bereits vorhandenen Menü- und eine Symbolleiste erzeugt (Abbildung 12.11).

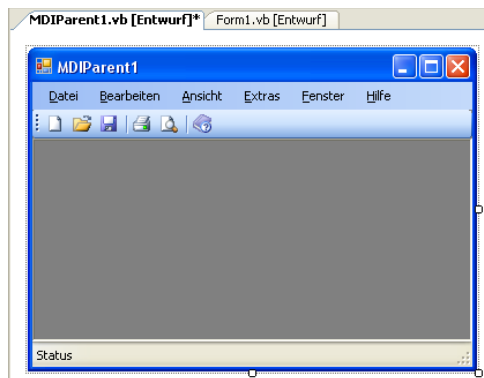


Abbildung 12.11: Fertiges Formular für das übergeordnete MDI-Parent-Fenster

12.3.2 Das untergeordnete MDI-Formular erstellen

Um die neuen Dokumente in MDI-Client-Fenstern anzeigen zu können, benötigen Sie ein zweites (Client) Formular.

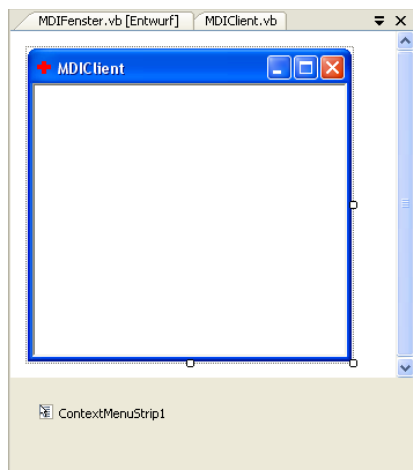


Abbildung 12.12: Formularentwurf des Client-Fensters

1. Klicken Sie im Projektmappen-Explorer der Entwicklungsumgebung mit der rechten Maustaste auf den Menüeintrag *Projekt* und wählen Sie im Kontextmenü den Befehl *Hinzufügen/Neues Element*. Wählen Sie im angezeigten Dialogfeld die Kategorie *Windows Form*, geben ggf. den Dateinamen vor und schließen dann das Dialogfeld.
2. Wechseln Sie zum Fenster des Client-Formulars und setzen Sie dessen Eigenschaften. Sie können beispielsweise die *WindowState*-Eigenschaft auf *Maximized* einstellen, um ein neues Dokumentfenster maximiert zu öffnen.
3. Ziehen Sie anschließend ein *RichTextBox*-Steuerelement aus der *ToolBox* in den Formularentwurf und setzen Sie die *Dock*-Eigenschaft auf *Fill* an.
4. Fügen Sie noch ein *ContextMenuStrip*-Steuerelement zum Formular hinzu und weisen Sie diesem einen Befehl *Schriftart* zum Aufruf des betreffenden Dialogs zu.

Mit diesen Schritten haben Sie das MDI-Formular angelegt. Das MDI-Formular wird wegen der *Dock*-Eigenschaft *Fill* vollständig vom *RichTextBox*-Steuerelement ausgefüllt (selbst wenn später die Größe des Formulars geändert wird).

Nach diesen Vorbereitungen haben Sie bereits eine lauffähige Anwendung, die sich übersetzen und ausführen lässt. Diese wird aber nur ein leeres MDI-Anwendungsfenster zeigen. Sie benötigen zusätzliche Funktionen, um neue Dokumentfenster anzulegen, Dokumente zu laden, zu speichern und ggf. zu bearbeiten. Nachfolgend finden Sie die Beschreibung des Codes, um die Funktionen der oben skizzierte Anwendung zu implementieren.

Tipp

Der Programmcode lässt sich dabei in der Klasse des Hauptformulars sowie in der Klasse des MDI-Clients hinterlegen. Alle Prozeduren, die zur Verwaltung der Anwendung dienen (z.B. Ereignisbehandlung für Menüelemente, Schaltflächen etc.) kommen in die Klasse der Anwendung. Prozeduren, die sich auf ein Dokument beziehen (z.B. Speichern von Daten, Ändern im Dokument etc.) sollten in die Klasse des MDI-Clients verlagert werden. Dies führt nicht nur zu einer transparenteren Programmstruktur, sondern erleichtert auch die Implementierung der betreffenden Funktionen – im Client kann beispielsweise direkt über *Me* auf ein Steuerelement im Dokumentfenster zugegriffen werden.

12.3.3 Hinzufügen des Programmcodes für neue Dokumentfenster

Um ein neues Dokument anzulegen, müssen Sie eine geeignete Funktion implementieren, die über die *Click*-Ereignishandler des Menüelements *Neu* und ggf. über die Schaltfläche *Neu* der Symbolleiste aufgerufen wird. Ich habe die betreffende Prozedur *DocNeu* genannt und einen Prozeduraufruf in den *Click*-Ereignishandlern des Menüelements und der Schaltfläche *Neu* der Symbolleiste hinterlegt. Der Code findet sich in der Klasse der Hauptanwendung. Die Ereignisbehandlungsprozedur der Schaltfläche *Neu* beziehungsweise des Menüeintrags *Neu* ruft die Prozedur *NewDoc()* auf. Die Prozedur *DocNeu* enthält folgenden Code:

```
Private Sub NewDoc()  
    'Erstellt ein neues Dokumentfenster samt Dokument  
    Dim NewDoc As MDIClient = CreateDocument("")  
    NewDoc.UpdateStatus()    ' Statusleiste aktualisieren  
End Sub
```

In der ersten Anweisung wird eine neue Objektvariable *NewDoc* vom Typ *MDIClient* deklariert. Der Typ wird durch den Klassennamen bestimmt (den ich über die *Name*-Eigenschaft des MDI-Client-Formulars eingestellt habe). Der betreffenden Objektvariable wird dann über die Funktion *CreateDocument()* eine neue Instanz eines MDI-Clients zugewiesen. Dabei wird als Parameter eine leere Zeichenkette übergeben (damit *CreateDocument* den Fenstertitel setzen kann). Die zweite Zeile ruft die benutzerspezifische Methode *UpdateStatus* auf. Diese Methode aktualisiert die Statuszeile sowie die Titelzeile des Dokumentfensters (mit dem Dokumentnamen und der Zeichen- und Zeilenzahl). Ich habe diese Methode als Prozedur in der Klasse *MDIClient* des Client-Formulars hinterlegt. Wird die Methode über die Objektvariable *NewDoc* aufgerufen, bezieht sie sich auf die zugehörige Instanz.

Der Code zum Anlegen eines neuen Dokumentfensters befindet sich in der Prozedur *CreateDocument*, die in der Klasse des Hauptformulars gespeichert ist (eine Verlagerung in die Klasse des Clients ist nicht möglich, da dieser ja beim Aufruf noch nicht existiert). Die Funktionsprozedur enthält folgende Anweisungen:

```
Private Function CreateDocument(ByVal name As String) As MDIClient  
    ' Erstellt ein MDI-Fenster mit einem leeren Dokument  
    ' Falls beim Öffnen einer Datei aufgerufen, wird  
    ' der Dateiname im Fenstertitel hinterlegt, sonst "Neu x"  
    ' Dieser Trick ist erforderlich, da sonst der Menübefehl  
    ' im Menü "Fenster" nicht korrekt aktualisiert wird.  
    Dim NewMDIChild As New MDIClient()  
    NewMDIChild.MdiParent = Me  
    DocumentCount += 1  
    If name = "" Then ' neues leeres Fenster  
        NewMDIChild.Text = "Neu" & DocumentCount.ToString  
    Else  
        ' Dateiname in Titel  
        NewMDIChild.Text = name  
    End If  
    NewMDIChild.Show()    ' Anzeigen  
    Return NewMDIChild    ' Objekt zurückgeben  
End Function
```

In der ersten Zeile wird eine neue Instanz des MDI-Clients erzeugt und der lokalen Objektvariablen *NewMDIChild* zugewiesen. Die zweite Anweisung hinterlegt in der Eigenschaft *MdiParent* eine Referenz auf die Adresse des Elternformulars (Me). Dies erlaubt den Zugriff aus den Prozeduren des Clients auf die Parent-Anwendung (die

Objektreferenz lässt sich über *MdiParent* abfragen). Der Aufruf der *Show*-Methode bewirkt die Anzeige des Formulars als neues Dokumentfenster im Fenster der MDI-Anwendung.

Die restlichen Anweisungen dienen zur Verwaltung. Die Variable *DocumentCount* zählt die geöffneten MDI-Dokumente und wird um 1 erhöht. Über die Eigenschaft *Text* der *NewMDIChild*-Objektvariablen lässt sich auf die Titelzeile der neuen Dokumentinstanz zugreifen. Wurde beim Aufruf der Prozedur eine leere Zeichenkette übergeben, weist die Prozedur dem Fenster den Text »Neu« kombiniert mit der laufenden Dokumentnummer als Dokumenttitel zu. Beim Öffnen einer Datei in einem neuen Fenster wird der Dateiname als Parameter übergeben und in die Titelzeile des Dokumentfensters geschrieben. Die letzte Anweisung liefert das Objekt an das rufende Programm zurück.

12.3.4 Weitere Verwaltungsfunktionen für die Fenster

Mit den bisherigen Anweisungen erlaubt die MDI-Anwendung neue leere Dokumentfenster anzulegen. Die Umschaltung zwischen den Fenstern über das Menü *Fenster* wird bereits durch die *Form*-Klasse des .NET Framework unterstützt. Auch das Schließen eines Dokumentfensters ist über die dem Fenster zugeordnete Schaltfläche *Schließen* möglich. Sie können aber weitere Funktionen zum Schließen eines Fenster über das Menü *Datei* oder zum Anordnen der Fenster über das Menü *Fenster* etc. implementieren.

Fenster übereinander oder nebeneinander anordnen

Die Beispielanwendung bietet im Menü *Fenster* Befehle, um die MDI-Dokumentfenster im Fenster der Parent-Anwendung kaskadierend, horizontal und vertikal nebeneinander anzuordnen. Die Anordnung der Fenster lässt sich über die *LayoutMdi*-Methode des Hauptformulars verändern. Der Methode ist lediglich der gewünschte Layoutmodus als Parameter zu übergeben. Nachfolgend sehen Sie den Code, der in den *Click*-Ereignisbehandlungsroutinen der betreffenden Menüeinträge hinterlegt wurde:

```
Private Sub ÜberlappendToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles ÜberlappendToolStripMenuItem.Click
    ' Menü Fenster/Überlappend
    Me.LayoutMdi(MdiLayout.Cascade) ' Fenster überlappend
End Sub

Private Sub NebeneinanderToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles NebeneinanderToolStripMenuItem.Click
    ' Menü Fenster/Nebeneinander
    Me.LayoutMdi(MdiLayout.TileVertical) ' Fenster nebeneinander
End Sub

Private Sub UntereinanderToolStripMenuItem_Click( _
```

```

ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles UntereinanderToolStripMenuItem.Click
' Menü Fenster/Untereinander
Me.LayoutMdi(MdiLayout.TileHorizontal) ' Fenster untereinander
End Sub

```

Schließen der Dokumentfenster

Sollen die Dokumentfenster per Menü geschlossen werden, müssen Sie lediglich einen entsprechenden Befehl in den *Click*-Ereignisbehandlungsroutinen der betreffenden Menüelemente eintragen. Im aktuellen Beispiel habe ich die eigentlichen Befehle aber in Prozeduren verlagert, die in der Klasse des Hauptformulars untergebracht sind. Die Prozedur zum Schließen des aktuell angewählten Dokumentfensters besitzt folgenden Inhalt:

```

Private Sub CloseDoc()
' Schließe aktuelles Dokumentfenster samt Dokument
Me.ActiveMdiChild.Close()
End Sub

```

Über die Objekthierarchie *Me.ActiveMdiChild* lässt sich das aktive MDI-Fenster abfragen. Die vom Hauptformular bereitgestellte *Close*-Methode schließt das Formular. Möchten Sie alle Dokumentfenster schließen, muss die Prozedur die Auflistung der MDI-Dokumente durchlaufen und jeweils die *Close()*-Methode aufrufen. Die folgende Prozedur zeigt dies:

```

Private Sub CloseAllDoc()
' Schließe alle Dokumentfenster samt Dokumenten
Dim MDIChild As Form() = Me.MdiChildren ' MDI-Client-Auflistung
Dim i As Integer
For i = 0 To MDIChild.Length - 1
    MDIChild(i).Close() ' Schließen des Fensters
Next i
End Sub

```

In der ersten Anweisung wird eine Objektvariable vom Typ *Form* vereinbart, der dann über die *Me.MdiChildren*-Eigenschaft des Hauptformulars die Auflistung aller Unterfenster zugewiesen wird. Anschließend lassen sich alle Elemente der Auflistung in einer einfachen *For*-Schleife abrufen. Über den Schleifenindex kann dann direkt ein Element der Auflistung in der Objektvariablen *MDIChild* angesprochen werden. Für das betreffende Objekt wird die *Close()*-Methode angewandt.

Schließen geänderter Dokumente verhindern

Beim Schließen von Dokumenten mit ungesicherten Änderungen sollte ein Dialog dem Benutzer Gelegenheit zum Sichern geben. Diese Technik hatte ich bereits im Beispiel *DateiDialoge* diskutiert. Bei einer MDI-Anwendung muss die betreffende Ereignisbehandlungsroutine in der Klasse des Clients untergebracht werden. Ich habe die im Bei-

spiel *DateiDialoge* vorgestellte Ereignisbehandlungsroutine *Fenster_Close* des *MyBase.Closing*-Ereignisses sowie die Funktion *SaveDirtyDoc()* aus diesem Projekt in die Klasse des MDI-Clients direkt übertragen. Die Funktion *SaveDirtyDoc()* wurde lediglich in einem Punkt geändert. Das *RichTextBox*-Steuerelement bietet eine Eigenschaft *Modified*. Hier wird diese Eigenschaft anstelle der Variablen *docChanged* zur Kennung geänderter Dokumente benutzt.

Die *Click*-Ereignisbehandlungsroutine des Befehls *Beenden* des Menüs *Datei* ruft die Prozedur *AppExit* auf. Diese muss ebenfalls berücksichtigen, dass ungesicherte Dokumente noch zu speichern sind. In der Routine werden alle Fenster der MDI-Clients in einer Schleife durchlaufen und über die *Close*-Methode geschlossen. Liegen ungesicherte Änderungen vor, zeigt die Methode einen Sicherheitsdialog und gibt dem Benutzer die Möglichkeit zum Speichern des Dokumentfensters. Dabei gibt es aber leider noch den Sonderfall, dass der Benutzer bei einem ungesicherten Dokument die Schaltfläche *Abbrechen* des Sicherheitsdialogs wählt. In diesem Fall wird das Fenster mit den ungesicherten Daten nicht geschlossen und die Methode *Application.Exit()* darf nicht aufgerufen werden. Hierzu wird in der Schleife nach dem Aufruf der *Close*-Methode die Eigenschaft *MDIChild(i).Visible* geprüft. Ist das Fenster nach dem Ausführen der *Close*-Methode sichtbar, wird ein lokales Flag gesetzt und die *Exit*-Methode nicht ausgeführt. Die Details sind nachfolgendem Codefragment zu entnehmen:

```
Private Sub AppExit()
    ' wird beim Schließen aufgerufen
    ' durch Schließen der Dokumentfenster kann im der MDIClient-Klasse
    ' eine Prozedur Closing implementiert werden, die ungesicherte
    ' Dokumente speichert.
    Dim MDIChild As Form() = Me.MdiChildren ' MDI-Client-Auflistung
    Dim i As Integer
    Dim bad As Boolean = False

    For i = 0 To MDIChild.Length - 1
        MDIChild(i).Close() ' Schließen des Fensters
        If MDIChild(i).Visible Then bad = True ' Fenster noch offen!
    Next i
    If bad Then ' ist noch ein Dokumentfenster geöffnet ?
        MessageBox.Show("Es ist noch ein Dokument offen", _
            "Fehler beim Schließen", MessageBoxButtons.OK)
    Else
        Application.Exit() ' Anwendung beenden
    End If
End Sub 'Beenden
```

Listing 12.4: Beenden einer MDI-Anwendung

Hinweis

Sie finden das komplette Projektbeispiel im Ordner `\Beisp\Kap12\MDI-Anwendung` auf der Begleit-CD. In `MDIFenster.vb` sind alle Codeteile, die sich mit der Verwaltung des Projekts befassen, hinterlegt. Im Element `MDIClient.vb` finden Sie das Client-Formular samt dem Code zur Verwaltung der Fenster. Der Ordner `\Beisp\Kap12\MDI-Anwendung1` auf der Begleit-CD enthält ein ähnliches Projektbeispiel, welches aber noch mit den älteren *MenuBar*-, *StatusBar*- und *Symbolbar*-Steuerelementen aufgebaut ist und lediglich nach Visual Basic 2005 portiert wurde. Der Code innerhalb der Ereignisbehandlungsroutinen unterscheidet sich daher geringfügig vom Beispiel *MDI-Anwendung*. Für neue Projekte sollten Sie auf die modernen *MenuStrip*-, *ToolStrip*- und *StatusStrip*-Steuerelemente zurückgreifen, da nur diese voll im Ansicht-Designer unterstützt werden.

12.3.5 Weitere Programmiertechniken für die Anwendung

Das Beispiel beinhaltet noch einige spezifische Erweiterungen, die bisher nicht in Beispielen behandelt wurden. An dieser Stelle möchte ich kurz auf die betreffenden Aspekte eingehen.

Laden von RTF-Dokumentdateien

Das *RichTextBox*-Steuerelement unterstützt formatierte Textdaten. Ohne Codeteile haben Sie bereits die Möglichkeit, ein in der Windows-Anwendung WordPad erstelltes formatiertes Dokument per Zwischenablage in das Steuerelement zu übertragen. Dann werden die Formatierungen mit übernommen. Das *RichTextBox*-Steuerelement erlaubt zudem das Einlesen von Textdateien und Dateien, die im RTF-Format gesichert wurden. Hierzu bietet die betreffende Klasse eine *LoadFile*-Methode.

Hinweis

Das RTF-Format wurde von Microsoft zum Austausch von Textdokumenten definiert und speichert die Texte samt Formatierung im ANSI-Zeichencode. Anwendungen wie WordPad, Microsoft Word etc. verwenden das RTF-Format. Allerdings liegen verschiedene Versionen des RTF-Formats vor und Microsoft Word benutzt umfassendere Befehle. Sie sollten zum Test daher nur mit WordPad erstellte Dokumente verwenden. Andernfalls kann es zu Laufzeitfehlern wegen nicht unterstützter Formate kommen.

Zum Laden von Dokumentdateien habe ich daher die *OpenDoc*-Routine des *DateiDialoge*-Beispiels leicht modifiziert. Die Prozedur *OpenDoc* ist in der Klasse des Hauptformulars hinterlegt und wird von den *Click*-Ereignisbehandlungsroutinen der Menü- und Symbolleiste aufgerufen.

```
Private Sub OpenDoc()  
    ' Aufruf zum Öffnen des Dokuments  
    Dim file As String  
    file = OpenF()      ' Dateidialog anzeigen
```

```

If file <> "" Then ' Datei gewählt
    ' Erzeuge MDI-Fenster mit Leerdokument
    Dim MyDoc As MDIClient = CreateDocument(file)
    MyDoc.OpenDocument(file) ' Lade Textdatei
    MyDoc.UpdateStatus() ' Statuszeile aktualisieren
End If
End Sub

```

Diese Prozedur verwendet die bereits vorgestellte Hilfsprozedur *OpenF* zur Abfrage des Dateinamens über den *Öffen-Dateidialog*. Diese Prozedur wurde gegenüber dem vorherigen Beispiel nur insoweit modifiziert, als die Eigenschaften *.CheckFileExists = True* und *.CheckPathExists = True* gesetzt sind. Dann verifiziert das Dialogfeld, dass die vom Benutzer angegebene Datei im betreffenden Pfad existiert. Wählt der Benutzer einen Dateinamen, ruft die Prozedur *OpenDoc* die Funktion *CreateDocument()* auf, um ein neues leeres Dokumentfenster zu erzeugen. Dabei wird bereits der Dateiname mit als Parameter übergeben, damit dieser als Fenstertitel im Client-Fenster sowie im Menü *Fenster* eingetragen wird. Anschließend gelangt die Prozedur *OpenDocument* der Klasse *MDIClient* zum Einsatz, die das Fenster mit den Dateidaten füllt. Die Prozedur *OpenDocument* erwartet den Dateinamen samt Pfad und besitzt folgende Anweisungen:

```

Public Sub OpenDocument(ByVal File As String)
    'Dokument öffnen -> Textdatei oder RTF-Format?
    If IO.Path.GetExtension(File).ToLower = ".rtf" Then
        OpenRTFDocument(File) ' Lade RTF-Dokumentdatei
    Else
        OpenTxtDocument(File) ' Lade Txt-Dokumentdatei
    End If
    Me.File = File ' Filename merken
    Me.UpdateStatus()
End Sub

```

Über die Methode *GetExtension* des Namensraums *System.IO.Path* lässt sich die Dateinamenerweiterung (z.B. *.rtf*) aus dem Pfad ermitteln. Bei der Erweiterung *.rtf* wird anschließend die *OpenRTFDocumentFile*-Prozedur aufgerufen. Liegt eine andere Erweiterung vor, kommt die Prozedur *OpenTxtDocument* zum Einsatz. Die letztgenannte Prozedur stammt aus dem Projekt *DateiDialoge*, wobei die Befehle zur Anzeige des Dateidialogs in die Prozedur *OpenDocument* verlagert wurden. Die Aufteilung in zwei Prozeduren wurde gewählt, da die Methode zum Einlesen von RTF-Dateien bei Textdateien einen Fehler lieferte. Die Prozedur *OpenRTFDocumentFile* besitzt folgenden Inhalt:

```

Public Sub OpenRTFDocument(ByVal File As String)
    ' RTF-Dokument öffnen
    Try ' Laufzeitfehler abfangen
        Me.RichTextBox1.LoadFile(File) ' einlesen
    Catch e As Exception ' Fehlerzweig
    End Try
End Sub

```

```

        MessageBox.Show(e.Message, "Fehler beim Öffnen", _
        MessageBoxButtons.OK, MessageBoxIcon.Error)
        Beep()           ' ggf. akustische Warnung
        Exit Sub         ' Prozedur verlassen
    End Try
    Me.RichTextBox1.Modified = False ' Inhalt geändert
End Sub

```

Eine *Try...Catch*-Anweisung fängt Laufzeitfehler ab. Das Einlesen der RTF-Datei erfolgt mittels der *LoadFile*-Methode des *RichTextBox*-Steuerelements. Der Methode ist lediglich der Dateiname zu übergeben. Dann öffnet diese die Datei und liest deren Inhalt in die *Text*-Eigenschaft des Steuerelements ein. Der letzte Befehl in der Prozedur setzt die *Modified*-Eigenschaft des *RichTextBox*-Steuerelements auf *False* zurück (beim Laden der Datei wird dieser auf *True* gestellt).

Speichern eines (RTF-)Dokuments

Das Speichern eines Dokuments erfolgt mit einem ähnlichen Schema wie das Laden. In den *Click*-Ereignisbehandlungsroutinen der Menü- und Symbolleistenelemente werden die beiden Prozeduren *SaveDoc* und *SaveAsDoc* aufgerufen. Diese Prozeduren finden sich in der Klasse des Hauptformulars. Beide Prozeduren ermitteln eine Referenz auf den aktuellen MDI-Client und rufen dann eine Hilfsprozedur (*SaveDocument* bzw. *SaveAsDocument*) auf. Die Prozedur *SaveDoc* besitzt dabei folgende Anweisungen:

```

Private Sub SaveDoc()
    ' Aufruf beim Speichern des Dokuments
    ' benutzt die SaveDocuments-Methode der MDIClient-Klasse
    ' Hole Referenz zum MDI-Client
    Dim MyDoc As MDIClient = CType(Me.ActiveMdiChild, MDIClient)
    MyDoc.SaveDocument() ' Speichern
End Sub

```

Der erste Befehl erzeugt die Objektreferenz auf den Client, der zweite aktiviert die Hilfsprozedur in der über die *MyDoc*-Instanz referenzierten MDI-Client-Klasse. Die Hilfsprozedur *SaveDocument* prüft nun, ob dem Dokument bereits eine Datei zugewiesen wurde. Trifft dies zu, wird das Dokument entweder über die Prozeduren *SaveTextDocument* oder *SaveRTFDocument* gesichert. Andernfalls wird in den *SaveAsDocument*-Programmteil verzweigt, um das neue Dokument unter einem angegebenen Namen zu speichern. Die Anweisungen in *SaveTextDocument* entsprechen weitgehend dem Code, der im Beispiel *DateiDialoge* in der Prozedur *SaveDoc* zum Einsatz gelangte. Die Prozedur *SaveRTFDocument* erwartet einen Dateinamen als Argument und übernimmt das Speichern eines bestehenden (und eines neuen) RTF-Dokuments. Hier kommen folgende Anweisungen zum Einsatz:

```

Public Sub SaveRTFDocument(ByVal File As String)
    ' Dokument als RTF speichern (für Save und SaveAs)

```

```

Try ' Laufzeitfehler abfangen
    Me.RichTextBox1.SaveFile(File) ' speichern
Catch e As Exception ' Fehlerzweig
    MessageBox.Show(e.Message & " " & File, _
        "Fehler beim Speichern", _
        MessageBoxButtons.OK, MessageBoxIcon.Error)
    Beep() ' akustische Warnung
    Exit Sub ' Prozedur verlassen
End Try
Me.RichTextBox1.Modified = False ' gesichert
End Sub

```

Das Sichern erfolgt über die *SaveFile*-Methode des *RichTextBox*-Steuerelements. Auch hier setzt die Prozedur die *Modified*-Eigenschaft des Objekts auf *False* zurück.

Hinweis

Die Programmlogik zum Speichern neuer Dokumentdateien entspricht dem bisher bereits diskutierten. Es wurden die gleiche Logik wie bei *SaveDoc* benutzt, wobei die Prozeduren mit dem Zusatz *..AS..* im Namen gekennzeichnet sind. Der interne Code der Prozeduren wurde aus dem Beispiel *Dateidialoge* abgeleitet und um den bereits oben erwähnten Zweig zur Sicherung von RTF-Daten ergänzt. Details können Sie den Beispieldateien *MDIFenster.vb* und *MDIClient.vb* im Ordner *\Beisp\Kap12\MDI-Anwendung* entnehmen.

Aktualisierung des Status beim Wechsel zu einem Dokument

Wechselt der Benutzer zu einem MDI-Dokument, müssen die Angaben in der Statusleiste des Anwendungsfensters aktualisiert werden. Die eigentliche Aktualisierung erfolgt in der Prozedur *UpdateStatus*, die hier in der Klasse des Clients hinterlegt wurde. Dadurch kann in der Prozedur direkt über das Schlüsselwort *Me* auf den Inhalt des Client-Formulars und damit auf das *RichTextBox*-Steuerelement zugegriffen werden. Um auf Elemente des Hauptanwendungsfensters zuzugreifen, lässt sich die Objektreferenz aus der *MdiParent*-Eigenschaft heranziehen. Die folgende Anweisung hinterlegt diese Referenz in einer Objektvariablen:

```
Dim MainWin As MDIFenster = CType(Me.MdiParent, MDIFenster)
```

Der *CType*-Operator bewirkt, dass der korrekte Typ *MDIFenster* (dies ist der von mir gesetzte Objektname des Hautformulars bzw. der zugehörigen Klasse) verwendet wird. Nachfolgend sehen Sie den kompletten Code der Prozedur *UpdateStatus*. Es lässt sich erkennen, wie elegant der Zugriff auf die Objekteigenschaften von Client und Parent abläuft, wenn die entsprechenden Referenzen in Objektvariablen liegen.

```

Friend Sub UpdateStatus()
    ' Aktualisiert Titelzeile und Statusleiste des Hauptfensters

```

```

Dim Zeilen As Integer ' Zeichenzahl
Dim Zeichen As Integer ' Zeilen (ohne Wrap)

Dim MainWin As MDIFenster = CType(Me.MdiParent, MDIFenster)
' Dateiname in Titelleiste MDI-Fenster
If File <> "" Then Me.Text = File
Zeichen = Me.RichTextBox1.Text.Length
Zeilen = Me.RichTextBox1.Lines().GetUpperBound(0) + 1
' Label in Statusleiste aktualisieren
MainWin.ToolStripStatusLabel1.Text = "Zchn: " & Zeichen.ToString
MainWin.ToolStripStatusLabel2.Text = "Zei: " & Zeilen.ToString
MainWin.ToolStripStatusLabel3.Text = _
Me.RichTextBox1.Modified.ToString
If Me.RichTextBox1.CanUndo Then ' Liegen Änderungen vor?
' Menü Rückgängig freigeben, sonst sperren
MainWin.RückgängigToolStripMenuItem.Enabled = True
Else
MainWin.RückgängigToolStripMenuItem.Enabled = False
End If
End Sub

```

Hinweis

In der Prozedur *UpdateStatus* sind die Objektnamen von Menüelementen nach den dort hinterlegten Menübefehlen benannt (z.B. *RückgängigToolStripMenuItem*). Falls dort Umlaute im Objektnamen auftauchen, sollten Sie diese Umlaute in der Objektbezeichnung (Eigenschaft *Name* des betreffenden Menüelements) ändern.

Die Prozedur *UpdateStatus* wird bereits beim Anlegen eines neuen Dokuments (Prozedur *NewDoc*), beim Laden einer Datei (Prozedur *OpenDoc*), beim Speichern (Prozeduren *SaveDoc*, *SaveAsDoc*) sowie bei jeder Änderung im Dokumentfenster (Aufruf über das *TextChanged*-Ereignis des *RichTextBox1*-Steuerelements) aufgerufen. Dies löst aber ein Problem nicht: Sind mehrere Dokumente geladen, sollte die Statusleiste beim Wechsel zu einem Dokument aktualisiert werden (diese zeigt ja Zeichenzahl, Änderungsstatus etc. an). Hier kann das *Enter*- (oder *Activated*-) Ereignis des *RichTextBox1*-Steuerelements benutzt werden. Diese in der Klasse des Clients hinterlegte Ereignisbehandlungsprozedur besitzt folgenden Inhalt:

```

Private Sub RichTextBox1_Activ(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Enter
' wird bei jedem Wechsel zu einem Fenster aufgerufen
UpdateStatus() ' aktualisiere Status
End Sub

```

Die Signatur ist durch *MyBase.Enter* universell definiert, Sie können dadurch die Prozedurdeklaration für eigene Projekte direkt übernehmen.

Unterstützung der Zwischenablage (Ausschneiden, Kopieren, Einfügen)

Das *RichTextBox*-Steuerelement unterstützt, wie andere Steuerelemente, automatisch den Austausch markierter Texte mit der Windows-Zwischenablage. Im Beispiel habe ich aber die Befehle *Ausschneiden*, *Kopieren* und *Einfügen* im Menü *Bearbeiten* hinzugefügt. Daher werden *Click*-Ereignisbehandlungsroutinen für die betreffenden Menüeinträge benötigt. Diese sind in der Klasse des Hauptformulars zu hinterlegen. Da die Zugriffe aber auf das Dokumentfenster erfolgen, muss in der Ereignisbehandlungsroutine die Adresse des aktiven MDI-Clients ermittelt werden. Für den Befehl *Ausschneiden* enthält die Ereignisbehandlungsroutine folgende Anweisungen:

```
Private Sub MenuItemCut_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MenuItemCut.Click
    ' Menü Bearbeiten/Ausschneiden
    Dim MyDoc As MDIClient = CType(Me.ActiveMdiChild, MDIClient)
    MyDoc.RichTextBox1.Cut() ' in Zwischenablage
End Sub
```

Die erste Anweisung innerhalb der Prozedur vereinbart eine Objektvariable *MyDoc* und weist dieser die Referenz auf die Instanz des aktiven MDI-Fensters über *ActiveMdiChild* zu. Dann wird die *Cut*-Methode des *RichTextBox*-Objekts aufgerufen, die den markierten Dokumentteil ausschneidet und im aktiven Format in die Zwischenablage überträgt. Zum Kopieren eines markierten Dokumentausschnitts verwenden Sie folgende Anweisung mit der *Copy*-Methode:

```
MyDoc.RichTextBox1.Copy()
```

Der Inhalt der Zwischenablage lässt sich über die folgende Anweisung im Steuerelement an der aktuellen Cursorposition einfügen (es wird vorausgesetzt, dass die Zwischenablage gültige Textdaten für das Steuerelement enthält).

Um einen markierten Textteil über einen Menübefehl zu löschen, weisen Sie der *SelectedRtf*-Eigenschaft einfach eine leere Zeichenkette zu.

```
MyDoc.RichTextBox1.SelectedRtf = "" ' Lösche markierten Inhalt
```

Soll der gesamte Dokumenttext markiert werden (z.B. über einen Menübefehl), tragen Sie folgende Anweisung in der betreffenden Ereignisbehandlungsroutine ein:

```
MyDoc.RichTextBox1.SelectAll() ' alles markieren
```

Die *SelectAll*-Methode wird vom Steuerelement bereitgestellt. Die Funktion *Rückgängig* wird beim *RichTextBox*-Objekt ebenfalls über eine *Undo*-Methode unterstützt.

Formatierung der RTF-Dokumentdaten im Steuerelement

Im Beispiel *Dateidialoge* wurde gezeigt, wie sich die Schriftart oder die Schriftfarbe für den gesamten Inhalt eines *TextBox*-Objekts setzen lässt. Das *RichTextBox*-Steuerelement erlaubt aber die Formatierung markierter Bereiche eines Dokuments mit Font-Attributen und Farben. Im MDI-Beispiel lässt sich die Formatierung eines markierten Textausschnitts über den Befehl *Schriftart* im Menü *Format* ändern. Die *Click*-Ereignisbehandlungsprozedur dieses Menüelements bietet einen Dialog *Schriftart* zur Auswahl der Font-Attribute an und besitzt folgenden Aufbau:

```
Private Sub MenuItemFont_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MenuItemFont.Click
    ' Menübefehl Schriftart angeklickt
    Dim MyDoc As MDIClient = CType(Me.ActiveMdiChild, MDIClient)
    Dim MyFont As FontDialog = New FontDialog() ' für Fontdialog
    With MyFont
        .ShowColor = True ' Farbauswahl erlauben
        .AllowScriptChange = False ' Kein Skript-Feld
        .Font = MyDoc.RichTextBox1.SelectionFont ' Font merken
        .Color = MyDoc.RichTextBox1.SelectionColor ' Farbe merken
    If .ShowDialog() <> DialogResult.Cancel Then
        With MyDoc.RichTextBox1
            .SelectionFont = MyFont.Font ' zuweisen
            .SelectionColor() = MyFont.Color
        End With
    End If
    End With
End Sub
```

Die vom Benutzer markierten Dokumentteile des *RichTextBox*-Objekts werden über die Eigenschaften *SelectionFont* und *SelectionColor* verwaltet. Eine Eigenschaft umfasst alle Schriftattribute (Schriftart, Schriftgrad, Auszeichnungen wie fett, kursiv etc.), während *SelectionColor* die Schriftfarbe definiert. Die obige Sequenz ermittelt die aktuellen Formate und weist diese dem *FontDialog*-Objekt zu. Dadurch zeigt das Dialogfeld *Schriftart* beim Aufruf die aktuellen Formateinstellungen an. Sobald der Benutzer das Dialogfeld schließt, werden die neuen Werte dieses Objekts dem markierten Dokumentbereich zugewiesen – das *RichTextBox*-Objekt passt die Formatierung entsprechend an.

Wie bekommt man das Kontextmenü?

An dieser Stelle noch einige Hinweise zur Frage, wie sich das *RichTextBox*-Steuerelement mit einem Kontextmenübefehl *Schriftart* versehen lässt. Die erste Idee, im Fenster *Ansicht-Design* ein *ContextMenu*-Steuerelement im Fenster der Hauptanwendung unterzubringen, den Befehl zu konfigurieren und eine Ereignisbehandlungsroutine zu implementieren, bringt Sie nicht weiter. Der Ereignishandler wird niemals aufgerufen. Sie müssen das *ContextMenuStrip*-Steuerelement (oder *ContextMenu-Element*) vielmehr im

Client-Fenster unterbringen, mit den gewünschten Befehlen konfigurieren und anschließend an das *RichTextBox*-Steuerelement anbinden. Sobald Sie dann den Ereignishandler implementiert haben, wird der Befehl zur Laufzeit angeboten. Der Code innerhalb der Ereignisbehandlungsroutine gleicht dem oben vorgestellten Ansatz:

```
Private Sub ContextMenuStrip1_Opening(ByVal sender As System.Object, _
    ByVal e As System.ComponentModel.CancelEventArgs) _
    Handles ContextMenuStrip1.Opening
    ' Kontextmenübefehl Schriftart angeklickt, formatiere Markierung
    Dim MyFont As FontDialog = New FontDialog() ' Fontdialoginstanz
    With MyFont
        .ShowColor = True ' Farbauswahl erlauben
        .AllowScriptChange = False ' Kein Skript-Feld
        .Font = Me.RichTextBox1.SelectionFont ' Font merken
        .Color = Me.RichTextBox1.SelectionColor ' Farbe merken
        If .ShowDialog() <> DialogResult.Cancel Then
            With Me.RichTextBox1
                .SelectionFont = MyFont.Font ' zuweisen
                .SelectionColor() = MyFont.Color
            End With
        End If
    End With
End Sub
```

Die Anweisungen berücksichtigen lediglich, dass in der Klasse des Clients ein direkter Zugriff auf die Objekte des Formulars über *Me* möglich ist.

Hinweis

Sie finden das komplette Beispiel, wie bereits oben erwähnt, im Ordner *\Beisp\Kap11\MDI-Anwendung* auf der Begleit-CD. Laden Sie die *sln*-Datei in die Entwicklungsumgebung, lässt sich das Beispiel samt Quellcode und Formulardesign untersuchen.

Mit diesen Erläuterungen möchte ich das vorliegende Kapitel schließen. Mit den bisherigen Ausführungen verfügen Sie über das benötigte Wissen, um eigene Anwendungen mit Benutzeroberflächen zu erstellen. Einige Themen (wie beispielsweise das Drucken von Dokumenten) werden in den nachfolgenden Kapiteln angesprochen.

Teil 4

Programmiertechniken

13

.NET-Basisfunktionen für Windows

Dieses Kapitel beleuchtet, wie Sie mittels der .NET-Framework-Klassenbibliothek bestimmte Basisfunktionen, z.B. zum Zugriff auf die aus Windows übergebenen Argumente, den Aufruf der Shell, das Suspendieren eines Prozesses, den Zugriff auf Umgebungsvariablen oder Registrierungseinträge etc. realisieren können.

13.1 Zugriff auf Programmargumente

Gelegentlich ist es hilfreich, einem Programm bestimmte Argumente (Schalter oder Dateinamen) mitgeben zu können. Dies gilt insbesondere für Konsoleanwendungen – die Technik zur Parameterübergabe lässt sich aber auch bei Windows-Anwendungen nutzen. Denken Sie nur an Anwendungen, bei denen eine Dokumentdatei auf deren Symbol gezogen wird. Dann startet Windows die Anwendung und übergibt dieser automatisch das zu bearbeitende Dokument. Der Zugriff auf die Parameter der Befehlszeile lässt sich innerhalb der .NET-Anwendung über verschiedene Ansätze realisieren, die ich nachfolgend kurz vorstellen möchte.

13.1.1 Parameterübernahme als Argumente der Main-Prozedur

In der einfachsten Variante können Sie die *Main*-Prozedur der Anwendung so formulieren, dass diese die beim Aufruf übergebenen Parameter als Argumente erwartet. Dies lässt sich mit folgendem Prozedurrumpf erreichen:

```
Class Test
  Shared Sub Main(ByVal CmdArgs() As String)
    ...
  End Sub
End Class
```

Statt einer leeren Klammer () wird bei der Prozedur *Sub* das Argument *CmdArgs()* angegeben. Es handelt sich dabei um ein Feld vom Typ *String*, in die das .NET Framework die von Windows übergebenen Argumente zur Laufzeit überträgt. Anschließend können Sie innerhalb der Prozedur *Main* ganz normal, wie bei anderen Prozeduren gewohnt, auf die einzelnen Elemente zugreifen. Das folgende kleine Programm liest die der Anwendung beim Aufruf übergebenen Parameter ein, setzt diese in einer Zeichenkette mit einer laufenden Nummerierung der Argumente zusammen und zeigt das Ergebnis in einem Dialogfeld an:

```

'*****
' File/Projekt: Parameter
' Autor: G. Born www.borncity.de
' Listet die beim Start übergebenen Argumente
' in einem Meldungsfeld auf.
'*****
Option Strict

Class Test
Shared Sub Main(ByVal CmdArgs() As String)
    Const Title As String = "Argumente"
    Dim ArgNum As Integer ' Index Argumente
    Dim Txt As String = "" ' Hilfstext

    Txt = "Argumente" & vbCrLf

    For ArgNum = 0 To UBound(CmdArgs)
        ' Argumente in Text zusammensetzen
        Txt = Txt & "Nr. " & ArgNum & _
            " = " & CmdArgs(ArgNum) & vbCrLf
    Next ArgNum

    ' Zeige das Ergebnis an
    MsgBox(Txt,vbOkOnly Or vbInformation,Title)
End Sub
End Class

```

Listing 13.1: Code zur Anzeige der Programmparameter

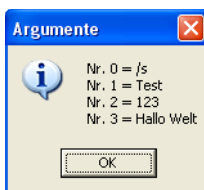


Abbildung 13.1: Anzeige der beim Programmaufruf übergebenen Argumente

Die einzelnen Parameter werden in der Startprozedur in einer einfachen *For*-Schleife gelesen und mit einer laufenden Nummerierung versehen in die Ausgabevariable *Txt* hinterlegt. Jedes Argument wird durch die Konstante *vbCrLf* in eine eigene Zeile umbrochen. Dies erlaubt Ihnen, die Auswirkungen verschiedener Eingaben beim Programmaufruf zu testen. Sie sehen sofort, ob eine Eingabe als ein Parameter übernommen oder in mehrere Parameter separiert wird. Abbildung 13.1 zeigt ein vom Beispielprogramm ermitteltes Ergebnis.

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung realisierten Beispiels im Ordner `\Beisp\Kap13\Parameter` auf der Begleit-CD. Sie können die Windows-Anwendung aus dem Fenster der Eingabeaufforderung aufrufen und die Parameter zum Testen mit übergeben (z.B. `Parameter.exe /s "Hallo Text" 123`). Alternativ besteht die Möglichkeit, eine Verknüpfung zur betreffenden Anwendung anzulegen und die Argumente zum Aufruf im Feld *Ziel* der Registerkarte *Verknüpfung* einzutragen. Sofern Sie das Beispiel jedoch in der Entwicklungsumgebung im *Debug*-Modus ausführen, lassen sich die als Argumente zu übergebenden Werte direkt in der Entwicklungsumgebung angeben. Klicken Sie im Projektmappen-Explorer die Projektdatei mit der rechten Maustaste an und wählen Sie den Kontextmenübefehl *Eigenschaften*. Anschließend können Sie die Werte auf der Eigenschaftenseite *Debuggen* im Feld *Befehlszeilenargumente* eintragen. In Abbildung 13.2 sehen Sie ein Beispiel für eine solche Eingabe. Um den Begriff »Hallo Welt« als einen einzigen Parameter an die Anwendung zu übergeben, müssen Sie die Wörter in Anführungszeichen stellen. Dies ist insbesondere bei langen Dateinamen wichtig, die Leerzeichen enthalten. Missachten Sie diese Regel, wird das lesende Programm keinen korrekten Dateinamen aus den Parametern ermitteln können.

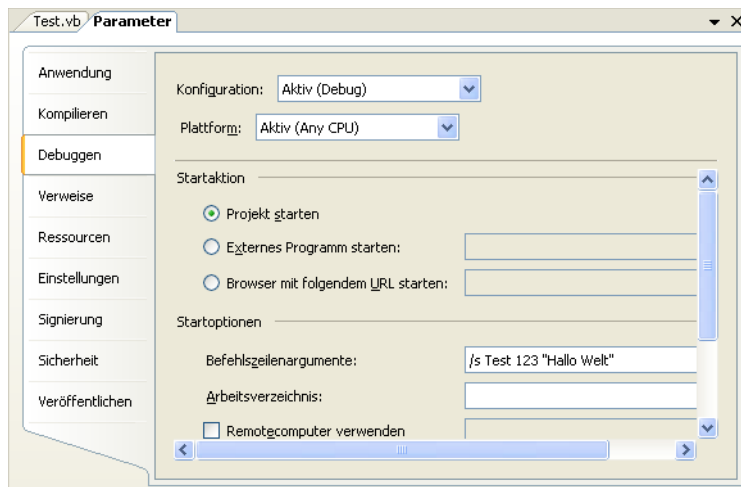


Abbildung 13.2: Befehlszeilenargumente festlegen

13.1.2 Parameter über Zusatzmethoden abfragen

Der oben gezeigte Ansatz lässt sich aber nicht immer nutzen. So ist es erforderlich, als Startroutine des Projekts *Sub Main* zu verwenden (da diese die *CmdArgs*-Auflistung als Parameter besitzt). Was ist aber, wenn ein Formular als Startroutine im Projekt benötigt wird? Oder Sie möchten vielleicht wissen, unter welchem Namen und aus welchem Pfad die Anwendung aufgerufen wurde. Müssen Sie sicherstellen, dass der Benutzer lange Dateinamen mit oder ohne Anführungszeichen als Parameter übergeben kann? Dann

sollten Sie zu alternativen Methodenaufrufen greifen, um die Aufrufparameter zu ermitteln (Abbildung 13.3).

Um die Aufrufparameter als Zeichenkette zu lesen, können Sie in einem VB-Programm die *Visual-Basic-Command-Funktion* oder die *CommandLine*-Eigenschaft des Namensraums *System.Environment* benutzen. Der Vorteil dieses Ansatzes besteht darin, dass Sie nun den kompletten String mit den Argumenten analysieren können. Ist nur ein Dateiname als Argument vorgesehen, ließe sich die gesamte Zeichenkette als Dateiname verwenden (egal, ob Leerzeichen im Dateinamen enthalten sind oder nicht). Ein zweiter Vorteil des hier skizzierten Ansatzes besteht darin, dass Sie direkt im Programmcode auf die Funktion bzw. die Eigenschaft zugreifen können – Sie benötigen die oben gezeigte Variante der *Main*-Prozedur nicht.

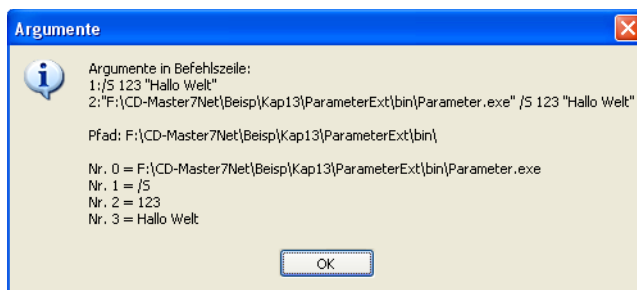


Abbildung 13.3: Modifizierte Parameteranzeige

Benötigen Sie die einzelnen Argumente doch, lässt sich die durch *Command* zurückgegebene Zeichenkette mit folgender Anweisung in Leerzeichen separieren:

```
Dim args() As String = command.Split(" ".ToCharArray)
```

Die *Split()*-Methode benutzt hier ein Leerzeichen, um die Befehlszeile in einzelne Argumente zu trennen. Das Ergebnis wird anschließend dem Feld *args* zugewiesen. Um auch den Namen sowie den Pfad der aufgerufenen Anwendung abzufragen, lässt sich die *GetCommandLineArgs()*-Methode des Namensraums *System.Environment* verwenden. Diese liefert ein Array mit Zeichenketten zurück. Das erste Feldelement enthält dabei den Anwendungsnamen samt Pfad. Dieser Parameter wird beispielsweise unter »Nr. 0« in Abbildung 13.3 angezeigt. Um den Pfad zur Anwendung zu ermitteln, könnten Sie also anstelle der in den vorhergehenden Kapiteln benutzten *Tools.GetPath()*-Methode folgende Anweisungen verwenden:

```
MsgBox (MyPath(CmdArgs(0))
...
Shared Function MyPath(ByVal name As String) As String
    Return Microsoft.VisualBasic.Left(_
        name, InstrRev(name, "\")) ' liefert den Pfad
End Function
```

Die Funktion *MyPath* verwendet die Funktionen *Left* und *InStrRev* der Visual-Basic-Kompatibilitätsklasse, um den Pfad samt Namen von rechts nach links nach dem ersten \-Zeichen zu durchsuchen und dann dieses Ende abzuschneiden. Als Argument ist der komplette Pfadname zu übergeben.

Hinweis

Die *Application.ExecutablePath*-Eigenschaft des Namespace *System.Windows.Forms* liefert übrigens für Windows-Anwendungen auch den kompletten Namen der aufrufenden Anwendung samt Pfad. Der Wert könnte ebenfalls zur Ermittlung des Pfades genutzt werden, indem Sie die *Path.GetDirectoryName()*-Methode im Namespace *System.IO* zum Abtrennen des Namens verwenden: *MyPath = Path.GetDirectoryName(Application.ExecutablePath) & "\"*.

Sie finden die Dateien des Projektbeispiels im Ordner *\Beisp\Kap13\ParameterExt* auf der Begleit-CD. Das Projekt enthält ein leeres Formular und die Anwendung zeigt beim Start die Aufrufparameter an. Der Code zur Analyse dieser Aufrufparameter wurde in die *Load*-Ereignisbehandlungsroutine des Formulars integriert. Weitere Details sind dem Quellcode des Moduls *Form1.vb* zu entnehmen.

13.2 Rückgabe eines Statuscodes

Wenn eine .NET-Anwendung terminiert, liefert diese den Process-Exit-Code 0 an das hostende Betriebssystem zurück. Sie haben aber die Möglichkeit, eine beliebige Ganzzahl zwischen 0 und 255 an den aufrufenden Prozess zurückzuliefern. Hierzu müssen Sie nur *Main* als Funktion implementieren und den gewünschten Wert über die *Return*-Anweisung zurückgeben. Die folgenden Anweisungen zeigen, wie sich so etwas realisieren lässt:

```
'*****  
' File/Projekt: SetExitCode  
' Autor: G. Born www.borncity.de  
' Wertet die als Argumente übergebene Zahl aus  
' und gibt diese als Exit-Code zurück.  
'*****  
  
Class Test  
    Shared Function Main(ByVal CmdArgs() As String) As Integer  
        Dim Args As String = Command ' Argumente  
        Dim code As Integer = -1  
  
        If IsNumeric(Args) Then code = CInt(Args)  
  
        If code >= 0 and code < 256 Then
```

Listing 13.2: Datei *SetExitCode.vb* gibt einen Fehlercode zurück


```

        MsgBox("Argument " & Args & vbCrLf & _
            "Aufrufpfad " & Application.ExecutablePath, _
            vbOKOnly Or vbExclamation, "Erfolg")
    Return code
Else
    MsgBox("Falsches Argument " & Args, _
        vbOKOnly Or vbExclamation, "Fehler")
    Return 0
End if
End Function
End Class

```

Listing 13.2: Datei *SetExitCode.vb* gibt einen Fehlercode zurück (Forts.)

Der Benutzer kann beim Aufruf einen Wert zwischen 0 und 255 als Argument in der Befehlszeile angeben. Das Programm liest die komplette Argumentzeile über *Command* ein. Dann wird das Argument auf Plausibilität überprüft (z.B. Wert zwischen 0 und 255) und bei numerischen Werten als Ergebnis in einem Dialog angezeigt. Fehlerhafte Argumente meldet das Programm ebenfalls in einem Dialogfeld. Nach einer Konvertierung der Zeichenkette in einen numerischen Wert wird dieser als Argument über die *Return*-Anweisung zurückgegeben.

Hinweis

Dieses Feature ist für Konsolenanwendungen ganz praktisch, die einen Statuscode an Stapelverarbeitungsdateien zurückliefern sollen. Der Exit-Code lässt sich in einer BAT-Datei über den Befehl `ERRORLEVEL` auswerten. Sie finden die Projektdateien des Beispiels im Ordner `\Beisp\Kap13\SetExitCode` auf der Begleit-CD. Zum Testen des Programms können Sie die im Unterordner `\Bin` enthaltene WSH-Skriptdatei *Test.vbs* verwenden. Sofern der Windows Script Host (WSH) auf der Maschine installiert ist, ruft das Skript die Datei *SetExitCode.exe* mit dem Parameter 13 auf, wartet, bis das Programm wieder terminiert, liest den Exit Code des Betriebssystems zurück und zeigt diesen in einem Dialogfeld an. Wie Sie den Exit Code aus einer .NET-Anwendung heraus auswerten, wird im Abschnitt »Warten, bis ein Prozess sich beendet hat und den Exit-Code abfragen« weiter unten erläutert.

13.3 Reagieren auf Anwendungsereignisse

Beim Starten einer Anwendung, beim Anlegen neuer Instanzen, beim Beenden der Anwendung, bei Änderungen in der Verfügbarkeit von Netzwerkverbindungen etc. werden Ereignisse ausgelöst, auf die in Ereignisbehandlungsroutinen reagiert werden kann. Nachfolgend erhalten Sie eine kurze Einführung, wie sich die Anwendungsereignisse in der Entwicklungsumgebung nutzen lassen und wie die Ereignisbehandlungsroutinen erzeugt werden.

13.3.1 Freigabe der Anwendungsereignisse

Soll auf Anwendungsereignisse (z.B. der Start der Anwendung) reagiert werden, muss ein *ApplicationEvents*-Modul im Projekt hinzugefügt werden. Hierzu gehen Sie in folgenden Schritten vor:

1. Klicken Sie im Fenster des Projektmappen-Explorers den Projekteintrag (hier *AppEvents*) mit der rechten Maustaste an und wählen Sie im Kontextmenü den Befehl *Eigenschaften*.
2. Wählen Sie die Eigenschaftenseite *Anwendung*, blättern Sie über die vertikale Bildlaufleiste zum Ende der Seite und klicken Sie auf die Schaltfläche *Anwendungsereignisse anzeigen* (Abbildung 13.4).

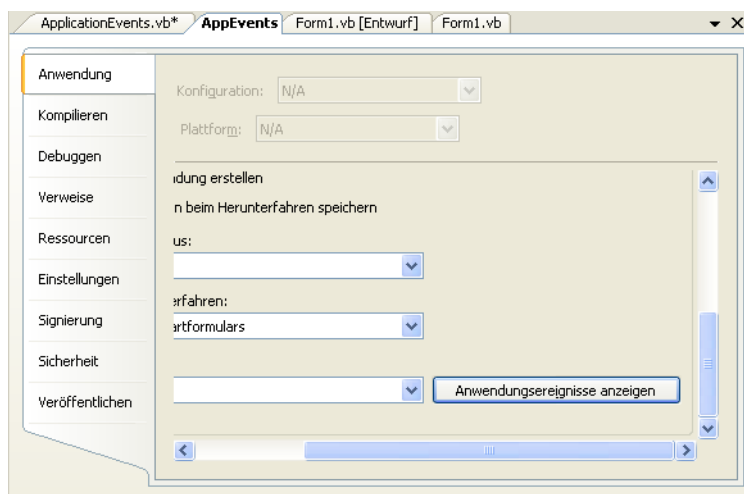


Abbildung 13.4: Anzeigen der Anwendungsereignisse

Die Entwicklungsumgebung blendet dann ein Modul *ApplicationEvents.vb* im Fenster des Projektmappen-Explorers ein.

13.3.2 Einfügen der MyApplication-Ereignisbehandlungsroutinen

Um den Prozedurrumpf der verschiedenen *MyApplication*-Ereignisbehandlungsroutinen einzufügen und den Code zu ergänzen, gehen Sie in folgenden Schritten vor:

1. Wählen Sie im Fenster des Projektmappen-Explorers das Modul *ApplicationEvents.vb* per Doppelklick an.
2. Wechseln Sie zum Fenster der Codeansicht des Moduls *ApplicationEvents.vb* und stellen Sie das linke Listenfeld auf den Wert »(MyApplication-Ereignisse)« (Abbildung 13.5, links).
3. Wählen Sie im rechten Listenfeld des Codefensters den Namen des gewünschten Ereignisses (Abbildung 13.5, rechts).

4. Sobald die Entwicklungsumgebung den Prozedurrumpf der Ereignisbehandlungsroutine im Codefenster eingefügt hat, ergänzen Sie den Code dieser Prozedur um die gewünschten Anweisungen.

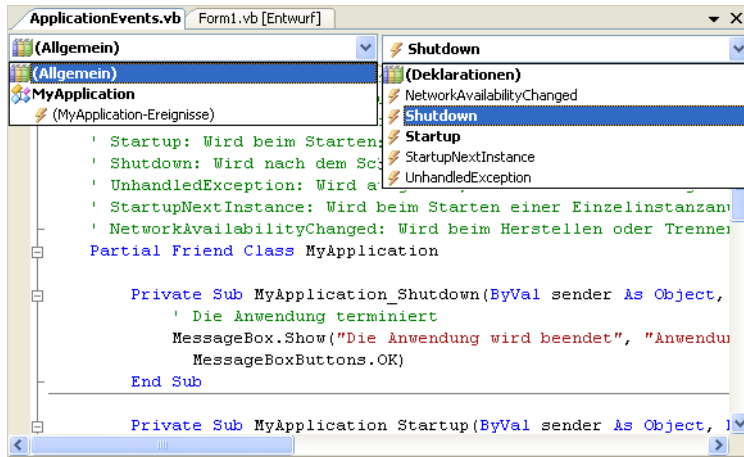


Abbildung 13.5: Einfügen der MyApplication-Ereignisbehandlungsroutinen

In Abbildung 13.5 ist die *MyApplication*-Ereignisbehandlungsroutine für das *Shutdown*-Ereignis zu sehen, welches beim Beenden der Anwendung ausgelöst wird. Im aktuellen Beispiel zeigt die Prozedur lediglich einen Dialog an.

Hinweis

Sie finden ein Projektbeispiel im Ordner `\Beisp\Kap13\AppEvent` auf der Begleit-CD. In diesem Beispiel sind die *MyApplication*-Ereignisbehandlungsroutinen für den Anwendungsstart und das Anwendungsende im Modul *ApplicationEvents.vb* implementiert.

13.4 Aufrufe externer Anwendungen

Gelegentlich kommt es vor, dass eine .NET-Anwendung externe Programme unter Windows aufrufen muss. Hierzu stehen verschiedene Möglichkeiten zur Verfügung. Der folgende Abschnitt skizziert die Vor- und Nachteile dieser Ansätze.

13.4.1 Arbeiten mit der Shell-Funktion

Die Microsoft-Visual-Basic-Kompatibilitätsklasse *Microsoft.VisualBasic* bietet die *Shell*-Funktion, mit der sich externe Anwendungen aufrufen lassen. Diese Methode erwartet beim Aufruf vier Parameter:

```
Shell (cmd_calc, AppWinStyle.NormalFocus, Wait, Timeout)
```

Im ersten Parameter *cmd_calc* wird der Name der auszuführenden Programmdatei (z.B. "Notepad.exe") ggf. samt einem Pfad aufgeführt. Der zweite optionale Parameter *AppWinStyle* definiert den Fensterstil für die betreffende Anwendung. Die verfügbaren Fensterstile liegen als Enumeration vor, die Namen sind in nachfolgender Tabelle aufgeführt. Sie sehen, dass sich eine Anwendung auch mit einem versteckten Fenster starten lässt.

Enumerationswert	Beschreibung
<i>AppWinStyle.Hide</i>	Fenster ausblenden, Fenster erhält aber den Fokus
<i>AppWinStyle.NormalFocus</i>	Normales Fenster, welches den Fokus erhält
<i>AppWinStyle.MinimizedFocus</i>	Fenster minimiert öffnen und Fokus zuteilen
<i>AppWinStyle.MaximizedFocus</i>	Maximiertes Fenster mit Fokus
<i>AppWinStyle.NormalNoFocus</i>	Normales Fenster ohne Zuweisung des Fokus
<i>AppWinStyle.MinimizedNoFocus</i>	Fenster minimieren und keinen Fokus zuweisen

Tabelle 13.1: Fensterstile

Lassen Sie den Parameter für den Fensterstil weg (z.B. *Shell ("Calc.exe", True)*), wird ein normales Fenster geöffnet, welches den Fokus erhält. Der dritte optionale Parameter *Wait* lässt sich auf *True* oder auf *False* setzen. Beim Wert *False* startet die Funktion die externe Komponente und übergibt dann die Kontrolle wieder an das rufende Programm. Möchten Sie, dass die Funktion so lange wartet, bis das externe Programm wieder beendet wird, ist der dritte Parameter *Wait* auf den Wert *True* zu setzen. Im vierten optionalen Parameter *Timeout* können Sie eine Wartezeit in Millisekunden hinterlegen. Wurde im dritten Parameter *Wait* der Wert *True* übergeben, gibt die Funktion *Shell* die Kontrolle nach Ablauf der Wartezeit an die aufrufende Anwendung zurück (egal, ob die extern gestartete Komponente bereits terminiert ist oder nicht). Um also den Windows-Rechner zu starten, lassen sich folgende Anweisungen verwenden:

```
Shell ("Calc.exe")
Shell ("Calc.exe", AppWinStyle.NormalFocus)
Shell ("Calc.exe", AppWinStyle.NormalFocus, True)
```

In der letzten Variante wartet *Shell*, bis der Windows-Rechner terminiert. In allen Befehlen wurde kein Pfad angegeben, d.h., Windows durchsucht das Windows-Verzeichnis, das Verzeichnis *System* sowie den Ordner, aus dem die .NET-Anwendung gestartet wurde. Befindet sich die Anwendung in einem anderen Ordner, müssen Sie den Pfad mit angeben. Benötigt die Anwendung bestimmte Parameter beim Aufruf, können Sie diese mit im ersten Parameter übergeben. Die Funktion *Shell* kann einen Wert zurückliefern. Die Anweisung

```
Id = Shell ("Calc.exe")
```

bewirkt, dass die Funktion *Shell* die sogenannte Prozess-ID der gestarteten Anwendung zurückliefert. Bedingung ist aber, dass der Prozess noch aktiv ist (d.h., der dritte Parameter muss auf *False* gesetzt sein, oder die *Timeout*-Angabe bewirkt trotz laufender

Anwendung eine Rückkehr zum rufenden Programm). Ist der extern gestartete Prozess bereits wieder beendet, liefert *Shell* den Wert 0 zurück.

Hinweis

Die Prozess-ID ist hilfreich, um ein Anwendungsfenster mit der *AppActivate*-Funktion in den Vordergrund zu holen (siehe auch die folgenden Seiten). Weil die *Shell*-Funktion bei einem beendeten Prozess immer den Wert 0 zurückliefert, kann die Funktion nicht zur Ermittlung des Exit-Codes einer Anwendung verwendet werden. Eine Lösung für dieses Problem findet sich auf den folgenden Seiten.

Was ist beim Aufruf von DOS-Anwendungen zu beachten?

Falls Sie eine Anwendung auf Konsoleebene ausführen möchten, können Sie den betreffenden Dateinamen (z.B. *Edit.com*) angeben. Bei DOS-Befehlen wie *Dir* klappt dies aber nicht, da es keine Datei *Dir.com* bzw. *Dir.exe* gibt. Vielmehr muss über *Shell* der Kommandoprozessor des Betriebssystems aufgerufen und dann der Befehl ausgeführt werden. Bei Windows 9x wird die Datei *Command.com* als Kommandoprozessor verwendet, während die Windows NT-Architektur (Windows NT, 2000, XP etc.) die Datei *cmd.exe* als bevorzugten Kommandoprozessor einsetzt. Um nicht jeweils zwei verschiedene Programmvarianten pflegen zu müssen, gibt es aber die Möglichkeit, die Umgebungsvariable *%Comspec%* im auszuführenden Befehl anzugeben. Vereinbarungsgemäß speichert Windows in dieser Umgebungsvariable den Namen des Befehlsprozessors. Einziges Problem ist, dass die *Shell*-Funktion Platzhalter für Umgebungsvariable nicht automatisch expandiert. Daher muss die Funktion *ExpandEnvironmentVariables()* aus dem Namensraum *System.Environment* aufgerufen werden. Die folgenden Anweisungen öffnen das Fenster der Eingabeaufforderung und führen einen *dir*-Befehl aus:

```
Const cmd_dos AS String = "%comspec% /k dir *.*"
...
id = Shell (ExpandEnvironmentVariables(cmd_Dos), _
            AppWinStyle.NormalFocus, True)
```

Der Schalter */k* in der Befehlszeile bewirkt, dass das Fenster der Eingabeaufforderung nach Ausführung des Befehls geöffnet bleibt. Soll das Konsolenfenster automatisch geschlossen werden, verwenden Sie den Schalter */c*.

Ein Anwendungsbeispiel

Die oben beschriebenen Informationen werden jetzt in einem Beispiel angewandt. Die .NET-Anwendung fragt in Benutzerdialogen nach, ob der Windows-Rechner, der Editor *Notepad*, sowie ein DOS-Befehl auszuführen sind. Bestätigt der Benutzer die Dialoge über die *Ja*-Schaltfläche, wird die betreffende Anwendung gestartet. Zur Kontrolle zeigt das Programm dann die von *Shell* zurückgelieferte Prozess-ID an. Weitere Details sind dem nachfolgenden Listing zu entnehmen:

```
'*****
' File/Projekt: RunExe
' Autor: G. Born www.borncity.de
' Rufe externen Anwendungen auf - demonstriert die
' Anwendung der Shell-Funktion.
'*****
Option Strict On

Imports System.Environment      ' für ExpandEnvironment...

Public Class Test1

    Shared Sub Main()
        Const title As String = "RunExe-Beispiel"
        Const cmd_edit As String = "Notepad.exe"
        Const cmd_calc As String = "Calc.exe"
        Const cmd_dos As String = "%comspec% /k dir *.*"

        Dim id As Integer      ' Prozess ID oder Exit-Code

        ' Rufe den Rechner auf und warte, bis Programm beendet wird
        If MsgBox("Starte den Rechner und warte bis Calc beendet wird", _
            vbYesNo Or vbQuestion, title) = vbYes Then
            Shell(cmd_calc, AppWinStyle.NormalFocus, True)
        End If

        ' Rufe den Rechner auf und warte 3 Sekunden bis zum Abbruch
        If MsgBox("Calc beendet, starte erneut und warte 3 Sekunden", _
            vbYesNo Or vbQuestion, title) = vbYes Then
            id = Shell(cmd_calc, AppWinStyle.NormalFocus, True, 3000)
        End If

        ' Starte den Editor, arbeite aber weiter
        If MsgBox("Starte den Editor, wartet aber nicht", _
            vbYesNo Or vbQuestion, title) = vbYes Then
            id = Shell(cmd_edit, AppWinStyle.MaximizedFocus, False)
            MsgBox("Prozess-ID des Editors: " & id)
        End If

        ' Starte einen DOS-Befehl
```

Listing 13.3: Beispiel zum Aufruf externer Anwendungen

```

If MsgBox("Öffne das Fenster der Eingabeaufforderung", _
    vbYesNo Or vbQuestion, title) = vbYes Then
    id = Shell(ExpandEnvironmentVariables(cmd_dos), _
        AppWinStyle.NormalFocus, True)
    MsgBox("Fenster der Eingabeaufforderung geschlossen")
End If

' Zeige das Ergebnis an
MsgBox("Fertig")
End Sub
End Class

```

Listing 13.3: Beispiel zum Aufruf externer Anwendungen (Forts.)

Hinweis

Sie finden das als Windows-Anwendung realisierte Projektbeispiel im Ordner `\Beisp\Kap13\RunExe` auf der Begleit-CD.

13.4.2 Kontrolle externer Prozesse über die Klasse Process

Die Visual Basic *Shell*-Funktion erlaubt zwar externe Anwendung zu starten. Allerdings gibt es einige Einschränkungen. Die Funktion gibt zwar einen Prozess-ID-Code zurück, liefert aber bei einem terminierten Prozess immer den Wert 0 als Exit-Code. Möchten Sie den von einem externen Programm beim Terminieren zurückgelieferten Exit-Code ermitteln? Zur Kontrolle von externen Prozessen enthält das .NET Framework die Klasse *Process* im Namensraum *System.Diagnostics*. Diese Klasse stellt sowohl Methoden, die den Aufruf externer Anwendungen erlauben, als auch Methoden, um Prozesse auch wieder zu beenden und um Eigenschaften wie den Exit-Code abzufragen, bereit. Um eine Instanz dieser Klasse anzulegen, haben Sie mehrere Möglichkeiten. Sie können beispielsweise im Fenster *Ansicht-Designer* ein *Process*-Steuerelement aus der Toolbox abrufen und dann im Formular hinzufügen. Anschließend können Sie auf das Steuerelement klicken und im Eigenschaftenfenster die Eigenschaften anpassen. So lässt sich die *Name*-Eigenschaft auf den Namen des Prozesses setzen. Über die Eigenschaft *StartInfo* erhalten Sie Zugriff auf die Member dieser Struktur mit den Einzeleigenschaften. Dort können Sie die Start- und Ausführungsoptionen für den Prozess setzen (z.B. der Name der zu startenden Anwendung wird in *FileName* hinterlegt). Alternativ lässt sich auch eine Instanz der betreffenden Klasse per Code über die folgenden Anweisungen erzeugen:

```

Imports System.Diagnostics      ' für Process
...
Dim edit_proc As Process = New Process()

```

Der *New*-Konstruktor der Klasse legt eine Instanz an und weist diese der Objektvariablen *edit_proc* zu. Anschließend können Sie über diese Objektvariable die Eigenschaften für den Prozess definieren und dann den Prozess starten. Die folgende Anweisungsfolge

legt die Eigenschaften für einen Prozess fest und startet dann das angegebene Programm:

```
Dim calc_proc As Process = New Process()  
With calc_proc          ' Prozesseigenschaften setzen  
    .EnableRaisingEvents = False ' keine Events  
    With .StartInfo      ' jetzt die Startinfo festlegen  
        .FileName = "Calc.exe"    ' Dateiname  
        .Arguments = ""  
        .WindowStyle = ProcessWindowStyle.Normal  
        .UseShellExecute = True    ' Windows-Shell mit Verben  
    End With  
    .Start()              ' Prozess starten  
    MsgBox(cmd_calc & " gestartet, ID " & .Id)  
End With
```

Die erste Anweisungszeile erzeugt eine neue Objektinstanz und weist diese der Objektvariablen *calc_proc* zu. Der Aufruf der *Start()*-Methode in der drittletzten Anweisungszeile bewirkt, dass der angegebene Prozess in den Speicher geladen und ausgeführt wird. Vorher müssen Sie aber die Eigenschaften des Prozess-Objekts über die *StartInfo*-Eigenschaft definieren. Diese Eigenschaft des *Process*-Objekts besitzt verschiedene Member, die sich wiederum als Eigenschaften abrufen lassen:

- *FileName*: Diese Eigenschaft in *StartInfo* ist unbedingt erforderlich und legt den Namen der auszuführenden Anwendung fest. Sie können dabei den Namen einer ausführbaren *.exe*-Datei (z.B. *Calc.exe*) eintragen und ggf. um eine Pfadangabe erweitern. Zusätzlich besteht die Möglichkeit, auch den Namen einer Dokumentdatei (z.B. *C:\Text\Brief.doc*) anzugeben (*UseShellExecute* muss dann auf *True* gesetzt sein, *Verb* muss den auszuführenden Befehl enthalten). Bei *UseShellExecute=False* können nur ausführbare Dateien aufgerufen werden.
- *Arguments*: Legt optional die an den startenden Prozess zu übergebenden Argumente fest.
- *UseShellExecute*: Wird diese Eigenschaft auf *True* gesetzt, verwendet die *Start()*-Methode die Shell des Windows-Betriebssystems zur Ausführung des Prozesses. Dann werden alle für den Dateityp registrierten Befehle unterstützt. Ist der Wert auf *False* gesetzt, führt die laufende Anwendung den Prozess aus. Dann können über weitere Eigenschaften die Standard-Ein-/Ausgabestreams umgeleitet werden.
- *Verb*: Gibt das beim Shell-Aufruf zu benutzende Verb (z.B. *open*) an. Es muss sich um ein gültiges Verb für den Dateityp handeln. Die von der Shell unterstützten Verben lassen sich über die *Verbs*-Auflistung abfragen.

Eine Übersicht über alle Member (d.h. Eigenschaften) von *StartInfo* erhalten Sie im Eigenschaftfenster der Entwicklungsumgebung, wenn Sie ein eingefügtes *Process*-Steuerelement im Ansicht-Designer anklicken. Die Hilfe zum .NET Framework enthält weitere Informationen zur Klasse *Process* und listet alle *StartInfo()*-Member auf.

Hinweis

Gemäß der Hilfe können Sie die *Start()*-Methode auch in einer vereinfachten Fassung aufrufen. Mit *Process.Start("IExplore.exe", "http://www.microsoft.com")* wird z.B. der Internet Explorer gestartet und eine Webseite abgerufen. Die Methode liefert dann ein *Process*-Objekt zurück, über dessen Eigenschaften sich die Prozess-ID ermitteln lässt.

Obige Sequenz führt die angegebene Anwendung über die *Start()*-Methode aus. Schlägt der Aufruf fehl, wird ein Laufzeitfehler ausgelöst (den Sie über *Try...Catch...End Try* abfangen können). Bei erfolgreichem Aufruf zeigt die Sequenz anschließend die dem Prozess zugeordnete Prozess-ID-Kennung in einem Dialogfeld an. Hierzu wird einfach die Eigenschaft *ID* der *Process*-Objektvariable abgefragt.

Ein Dokument über Process aufrufen

Der Aufruf einer Dokumentdatei über die *Start()*-Methode des *Process*-Objekts erfordert, dass Sie die Eigenschaft *UseShellExecute* auf *True* setzen und in der Eigenschaft *Verb* den Namen eines gültigen Verbs hinterlegen. Die nachfolgende Sequenz demonstriert diesen Ansatz:

```
Dim doc_proc As Process = New Process()
With doc_proc ' Prozesseigenschaften setzen
    .EnableRaisingEvents = False ' keine Events
    With .StartInfo ' jetzt die Startinfo festlegen
        .FileName = "C:\Test.txt" ' Textdatei
        .Arguments = ""
        .verb = "open" ' Öffnen verwenden
        .WindowState = ProcessWindowState.Normal
        .UseShellExecute = True ' Windows-Shell mit Verben
    End With
    Try ' Laufzeitfehler abfangen
        .Start() ' starten
    Catch ex As Exception
        MsgBox(ex.ToString, _
            vbOkOnly Or vbCritical, _
            "Fehler beim Aufruf")
    End Try
```

Ich habe in dieser Sequenz Laufzeitfehler beim Ausführen der *Start()*-Methode mittels *Try ... Catch ... End Try* abgefangen.

Einen laufenden Prozess beenden

Sofern Sie die Prozess-ID eines laufenden (externen) Prozesses kennen, können Sie diese über die Methoden des *Process*-Objekts beenden. Die Prozess-ID lässt sich über die

Eigenschaft *ID* des *Process*-Objekts abfragen. Zum Beenden der Anwendung empfiehlt die Hilfe zum .NET Framework den Zugriff auf die *CloseMainWindow()*-Methode. Diese sendet eine Aufforderung zum Schließen an die betreffende Anwendung. Damit erhält diese Gelegenheit, den Benutzer zum Sichern von Daten aufzufordern und terminiert sich selbst. Eine nicht reagierende Anwendung kann dagegen mit der *Kill()*-Methode zwangsweise beendet werden. Dabei gehen allerdings alle ungesicherten Daten verloren. Die folgenden Anweisungen demonstrieren dies:

```
edit_proc.Kill          ' beende den Prozess
doc_proc.CloseMainWindow ' beende den Prozess
```

Die Unterschiede im Verhalten können Sie am nachfolgend vorgestellten Beispiel studieren.

Warten, bis ein Prozess sich beendet hat und den Exit-Code abfragen

Gelegentlich ist es hilfreich, wenn die Anwendung auf das Beenden eines externen Prozesses wartet (z.B. nach dem Aufruf der *CloseMainWindow()*-Methode). Zudem kann über das *Process*-Objekt der Prozess-Exit-Code der externen Anwendung ausgewertet werden. Die folgende Sequenz nutzt dies, indem die Anwendung *SetExitCode1.exe* aufgerufen wird. Diese Anwendung liefert den beim Aufruf übergebenen numerischen Wert als Prozess-Exit-Code an das Betriebssystem zurück.

```
' Starte die .NET-Anwendung mit einem Parameter
Dim Net_proc As Process = New Process()
With Net_proc          ' Prozesseigenschaften setzen
    .EnableRaisingEvents = False ' keine Events
With .StartInfo        ' jetzt die Startinfo festlegen
    .FileName = "c:\SetExitCode1.exe"
    .Arguments = "24"          ' Argumente
    .WindowState = ProcessWindowState.Minimized
End With
.Start()
MsgBox (cmd_Net & " gestartet , ID " & .id)
' Jetzt warten wir darauf, dass der Prozess terminiert
Do Until .HasExited
    Thread.Sleep (200)        ' warte 200 ms
Loop
MsgBox (cmd_Net & " beendet, Exit Code " & .ExitCode)
End With
```

Die Befehle zum Starten des externen Prozesses kennen Sie von den Codefragmenten der vorhergehenden Seiten. Als Neuerung wird der Wert 24 als Argument übergeben. Die Anwendung *SetExitCode1.exe* wird diesen Wert als Fehlercode zurückliefern. Bevor dieser Code über die Eigenschaft *ExitCode* des *Process*-Objekts abgefragt werden kann, ist sicherzustellen, dass der Prozess auch wirklich terminiert ist. Diese Information lässt sich über die *HasExited*-Eigenschaft ermitteln. Ein Wert *True* birgt die Bestätigung. In obi-

ger Sequenz habe ich eine *Do...Loop*-Schleife für diesen Test benutzt. Die Schleife wird so lange durchlaufen, bis *HasExited* den Wert *True* annimmt.

Hinweis

Alternativ können Sie auch eine leicht modifizierte Variante zum Aufruf der *Start()*-Methode verwenden. Die folgende Codesequenz übergibt der Methode im ersten Parameter den Namen der auszuführenden Anwendung, während der zweite Parameter die Argumente für diese Anwendung enthält.

```
Dim newProc As Diagnostics.Process
newProc = Diagnostics.Process.Start( _
    MyPath() & cmd_NET, "28")
newProc.WaitForExit() ' warte auf Prozessende
If newProc.HasExited Then _
    MsgBox ("Rückgabecode: " & newProc.ExitCode.ToString)
```

Bei dieser Sequenz wird die *Start()*-Methode als Funktion verwendet, die einen Prozess-ID-Code zurückliefert. Über diese ProzessID kann die *WaitForExit()*-Methode aufgerufen werden. Diese unterbricht den Programmablauf, bis der zugehörige Prozess terminiert. Danach lässt sich der Prozess-Exit Code über die Eigenschaft *ExitCode* abfragen.

Verzögerungen in .NET-Anwendungen

An dieser Stelle möchte ich noch eine weitere Information geben. Normalerweise hätten die folgenden Anweisungen zur Implementierung der Warteschleife gereicht:

```
Do Until .HasExited : Loop
```

Das Ganze ist recht kompakt und passt in eine Zeile. Der Pferdefuß dieser Lösung: Das Programm durchläuft die Schleife ohne Unterbrechung – die CPU-Auslastung wird unter Windows auf 100 Prozent ansteigen. Windows sorgt zwar dafür, dass andere Prozesse über den Prozess-Scheduler Laufzeit erhalten. Aber die CPU-Auslastung ist unnötig und verschlechtert die Reaktionszeiten für andere Programme.

Die CPU-Auslastung sinkt enorm, wenn Sie die .NET-Framework-Anwendung beim Durchlaufen der Schleife kurzzeitig suspendieren. In der Zeit kann Windows die Rechenzeit anderen laufenden Prozessen zuweisen. In Visual Basic .NET lässt sich hierzu die *Sleep()*-Methode der Klasse *Thread* (Namespace *System.Threading*) verwenden. Die Anweisung

```
Thread.Sleep (200)
```

suspendiert den laufenden Thread (da wir hier nicht mit Neben-Threads arbeiten, ist dies der Haupt-Thread der Anwendung) für n Millisekunden. Die Wartezeit in Millisekunden wird der Methode dabei als Argument übergeben. In obiger Anweisung habe ich 200 Millisekunden gewählt. Dies ist kurz genug, um schnell auf das Beenden zu reagieren, entlastet aber den Rechner trotzdem.

Beim Beenden eines laufenden Prozesses können Sie auch einen zweiten Ansatz benutzen, um auf das Prozessende zu warten, die .NET-Anwendung aber zu suspendieren:

```
doc_proc.CloseMainWindow ' Prozess zum Beenden auffordern  
doc_proc.WaitForExit()   ' warten, bis Prozess wirklich terminiert
```

In der obigen Sequenz wird der Prozess über *CloseMainWindow* zum Terminieren aufgefordert. Der nachgeschaltete Aufruf der *WaitForExit()*-Methode suspendiert das laufende .NET-Programm so lange, bis der angegebene Prozess terminiert. Optional lässt sich die Wartezeit in Millisekunden als Parameter an *WaitForExit* übergeben. Dann geht die Kontrolle nach Ablauf der Wartezeit an den rufenden Prozess zurück, auch wenn die betreffende Komponente nicht terminiert ist.

Laufende Anwendungen erneut aktivieren

Die Fenster der unter Windows laufenden Anwendungen werden auf dem Desktop angezeigt oder minimiert in der Taskleiste dargestellt. Genau ein Fenster besitzt den Fokus und wird im Vordergrund angezeigt. Um ein anderes Anwendungsfenster zu aktivieren und damit in den Vordergrund zu schalten, lässt sich die *AppActivate*-Funktion aus Visual Basic 2005 aufrufen. Der Funktion kann entweder eine Prozess-ID oder der Titel des gewünschten Fensters übergeben werden.

```
AppActivate ("Rechner") ' über Fenstertitel  
AppActivate (edit_proc.id) ' über Prozess-ID
```

Die Process-ID eignet sich vor allem, wenn Sie die externe Anwendung aus der .NET-Anwendung gestartet und Zugriff auf die Objektvariable *Process* haben. Über den Fenstertitel lassen sich beliebige Fenster aufrufen. Der Titel kann dabei ein Teilausdruck sein, der den Anfang oder das Ende eines Fenstertitels festlegt. Probleme gibt es aber mit Anwendungen, die im Fenster variable Titel (z.B. Dokumentnamen) aufführen. Dann ist der Aufruf über die Prozess-ID der bessere Weg.

Hinweis

AppActivate wirkt jedoch nur mit Windows-Fenstern und kann nicht bei Konsoleanwendungen benutzt werden. In der Hilfe des .NET Framework sind die Eigenschaften und Methoden der Komponente *Process* ausführlich dokumentiert. Dort finden Sie auch Hinweise, wie sich mittels der *Processes*-Auflistung die Liste der laufenden Prozesse abfragen und auswerten lässt.

Ein Anwendungsbeispiel

Zur Demonstration der oben beschriebenen Techniken habe ich zwei kleine Anwendungsbeispiele erstellt. Die in Abbildung 13.6 gezeigten Anwendung zeigt ein Formular, welches auf der linken Seite Schaltflächen zum Starten verschiedener Anwendungen aufweist. Anschließend kann die Anwendung über die am rechten Rand freigegebene Schaltfläche wieder beendet werden.

Für eine laufende Anwendung wird die Prozess-ID in einem Textfeld angezeigt. Bei *.NET-App* blendet die Prozesssteuerung zudem den Prozess-Exit-Code der betreffenden Anwendung nach deren Beendigung ein.



Abbildung 13.6: Fenster der Prozesssteuerung

Hinweis

Sie finden dieses als Windows-Anwendung realisierte Projekt im Ordner `\Beisp\Kap13\RunProcess` auf der Begleit-CD. Details sind dem Quellcode der Formulardatei *Form1.vb* zu entnehmen. Das Beispiel benötigt die externe Datei *SetExitCode1.exe* im Ordner, in dem auch die Datei *RunProcess.exe* abgelegt ist. Eine weitere Anwendung findet sich im Ordner `\Beisp\Kap13\RunProcess1` auf der Begleit-CD. Diese Anwendung fragt den Benutzer über eine Sequenz von Dialogen, ob bestimmte Aktionen durchzuführen sind. Dabei startet das Programm den Windows-Rechner, den Windows-Editor und eine .NET-Framework-Anwendung *SetExitCode1.exe*. Zudem wird ein Textdokument automatisch in der für *.txt*-Dateien registrierten Anwendung geladen. Hierbei wird der Pfad und der Name des Textdokument in der *FileName*-Eigenschaft der *StartInfo*-Struktur hinterlegt. Da die Interaktion mit dem Textdokument über ein Verb (Drucken, Laden etc.) der Shell erfolgen muss, wertet das Programm die *Verbs()*-Auflistung der *StartInfo*-Struktur aus und bietet dem Benutzer die Verben an. Anschließend wird das Verb über die *UseShellExecute()*-Methode mittels der Shell ausgeführt. Mit dem Verb »open« lässt sich die Textdatei beispielsweise im standardmäßig für *.txt*-Dateien registrierten Windows-Editor laden. Bei jedem Schritt wird die Prozess-ID oder ggf. der Prozess-Exit-Code angezeigt. Anschließend lassen sich die Anwendungsfenster wahlweise in den Vordergrund holen und dann schrittweise ferngesteuert schließen. Dabei kommen die weiter oben beschriebenen Techniken zum Einsatz. Beim Test ist zu beachten, dass das Dialogfeld gelegentlich den Fokus verliert und als Symbol in der Taskleiste erscheint (z.B. wenn eine Anwendung im Vordergrund geladen wird). Laufzeitfehler werden in diesem Beispiel (aus Aufwandsgründen) mit der *On Error Goto Fehler*-Anweisung zentral bearbeitet. In einer Implementierung für den produktiven Einsatz empfiehlt es sich aber, die einzelnen Fehlerursachen (z.B. fehlende Anwendung, fehlende Dokumentdatei) über *Try .. Catch .. End Try*-Blöcke abzufangen. Details zur Implementierung sind dem Quellcode zu entnehmen.

13.5 Einbinden externer COM-Komponenten

Das .NET Framework erlaubt Ihnen, unter Windows auf bestehende COM-Komponenten zuzugreifen. Dadurch ist es beispielsweise kein Problem, Funktionen von Word, Excel, der Windows-Shell etc. in .NET-Anwendungen zu nutzen. In diesem Abschnitt möchte ich diesen Aspekt etwas beleuchten, indem die Funktionen der Windows-Shell (Datei *Shell32.dll*) aus einer Anwendung heraus aufgerufen werden.

Hinweis

Windows benutzt intern die Datei *Shell32.dll*, die eine Reihe interessanter Methoden bietet, die sich extern aufrufen lassen. Sobald eine Referenz auf die Shell eingerichtet ist, lassen sich daher auch in .NET-Anwendungen Ordnerfenster, die Systemsteuerung, die Eigenschaftenfenster für Datum/Uhrzeit, die Taskleiste und mehr abrufen. Die Vorgehensweise wird nachfolgend an einem Beispiel demonstriert.

13.5.1 Spätes Binden (Late Binding) an eine COM-Anwendung

COM-Anwendungen werden in der Registrierung mit einem eindeutigen Namen (ID) registriert. Die Windows-Shell ist z.B. unter dem Namen »Shell.Application« registriert. Eine Anwendung kann daher zur Laufzeit eine Objektreferenz über diesen registrierten Namen einrichten. In Visual Basic 2005 wäre dies mit folgenden Anweisungen möglich:

```
Dim Shell As Object  
Shell = CreateObject ("Shell.Application")
```

Die erste Anweisung vereinbart eine Variable vom Typ *Object*. Die *CreateObject*-Funktion erzeugt anschließend einen Verweis auf das angegebene COM-Objekt und weist diesen Verweis der Objektvariablen zu. Da das Ganze erst zur Laufzeit erfolgt, spricht man von später Bindung (Late Binding). Der Vorteil besteht darin, dass sich bei dieser Art der Anbindung mit den registrierten COM-Namen wie »Shell.Application« arbeiten lässt. Der Nachteil liegt darin, dass spätes Binden langsamer abläuft. Zudem können Fehler auftreten, wenn die angegebene Komponente zur Laufzeit nicht mehr vorliegt. Das spätes Binden funktioniert auch nicht, wenn *Option Strict On* gesetzt ist.

13.5.2 Frühes Binden (Early Binding) an eine COM-Anwendung

Die Alternative beim Zugriff auf COM-Anwendungen besteht darin, die Bindung bereits bei der Übersetzung vorzunehmen. Dann lässt sich bereits bei der Übersetzung prüfen, ob die Schnittstelle korrekt implementiert ist. Zudem können alle Referenzen auf die Methoden und Eigenschaften der COM-Komponente beim Übersetzen mit aufgelöst werden. Diese als frühes Binden (Early Binding) bezeichnete Technik wird durch Visual Basic 2005 bzw. .NET Framework unterstützt. Die Befehle zum Anlegen einer Objektvariable, die auf die Windows-Shell verweist, sehen dann so aus:

```
Dim Shell As Shell32.ShellClass
Shell = New Shell32.ShellClass()
```

In der ersten Zeile wird wieder die Objektvariable *Shell* vereinbart. Diesmal wird jedoch ein spezieller Datentyp *Shell32.ShellClass* zugewiesen. Es muss sich dabei um den Datentyp handeln, der die externe COM-Komponente repräsentiert. Die zweite Anweisung legt dann eine Instanz dieses Objekts (COM-Komponente) an und lädt dieses in den Speicher. Anschließend lassen sich die Methoden und Eigenschaften dieser COM-Komponente aus .NET-Anwendungen heraus aufrufen. Die Methode des frühen Bindens funktioniert auch, falls *Option Strict On* gesetzt ist. Allerdings setzt dies voraus, dass .NET Framework über einen COM-Wrapper auf die betreffende COM-Komponente zugreifen kann. Dieser COM-Wrapper wird durch die Entwicklungsumgebung automatisch im Verzeichnis *\Bin* mit der ausführbaren Anwendung als DLL-Datei bereitgestellt. Um bei der Übersetzung auf die COM-Komponente zugreifen zu können, muss eine Referenz im Projekt eingerichtet werden. Dies ist mit folgenden Schritten möglich:

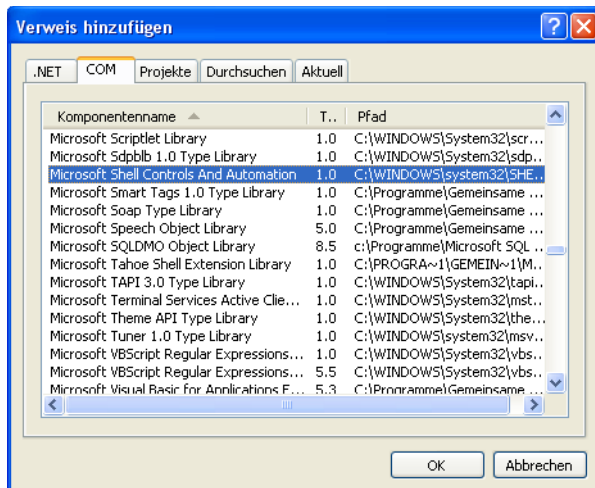


Abbildung 13.7: Einbinden von COM-Komponenten

1. Klicken Sie im Fenster des Projektmappen-Explorers mit der rechten Maustaste auf den Projekteintrag und wählen Sie im Kontextmenü den Befehl *Verweis hinzufügen*.
2. Im dann angezeigten Dialogfeld *Verweis hinzufügen* wählen Sie die Registerkarte *COM* (Abbildung 13.7), selektieren die gewünschte COM-Komponente (für die Windows-Shell ist dies die Komponente *Microsoft Shell Controls And Automation*) mit einem Mausklick und bestätigen dies über die *OK*-Schaltfläche.

Falls die COM-Komponente noch nicht registriert ist, können Sie die Datei über die Schaltfläche *Durchsuchen* auch direkt aus einem Verzeichnis auf der Festplatte referenzieren. Mit diesem Schritt kennt die Entwicklungsumgebung die Komponente und Sie können diese im Code nutzen. Die Hilfe zum .NET Framework enthält weitere Informationen zu COM-Wrappern.

13.5.3 Ordnerfenster und Dialogfeld Ausführen öffnen

Sobald eine Instanz der Windows-Shell geladen und die Referenz in einem Objekt hinterlegt ist, lässt sich auf die Methoden dieser Instanz zugreifen:

- *Open(<pfad>)*: Diese Methode öffnet den im Parameter <pfad> angegebenen Ordner in der Windows-Ordneransicht.
- *Explore(<pfad>)*: Öffnet den im Parameter <pfad> angegebenen Ordner im Windows-Explorer.
- *FileRun(<pfad>)*: Öffnet das Dialogfeld *Ausführen*, welches im Startmenü unter dem gleichnamigen Befehl zur Verfügung steht.

Die folgenden Anweisungen demonstrieren die Anwendung dieser Methoden. Dabei wird die späte Bindung benutzt, um eine Objektinstanz über *CreateObject* anzulegen. Anschließend lassen sich die Dialoge über die von der Shell angebotenen Methoden öffnen.

```
Dim Shell As Object = CreateObject ("Shell.Application")
Shell.Open("C:\")      ' Öffne ein Ordnerfenster
Shell.Explore("C:\")
Shell.FileRun()
```

Hinweis

Sie finden die Projektdateien eines als Windows-Anwendung realisierten Programmbeispiels im Ordner `\Beisp\Kap13\ShellInterop` auf der Begleit-CD. Beim Projekt wurde die Einstellung auf *Option Strict Off* gesetzt, da es sich sonst nicht übersetzen lässt. Details sind dem Quellcode zu entnehmen. Das Beispiel fragt in mehreren Dialogen ab, ob Ordnerfenster oder das Dialogfeld *Ausführen* verwendet werden sollen.

13.5.4 Shell-Aufrufe unter Verwendung eines COM-Wrappers

Möchten Sie bereits zur Übersetzungszeit eine Bindung an das Windows *Shell*-Objekt erzwingen, müssen Sie eine Objektvariable mit dem Typ der Windows-Shell instantiieren. Nachfolgend sehen Sie ein Programmfragment mit den betreffenden Anweisungen:

```
Option Strict On
Class Test
  Shared Sub Main()
    Dim Shell As Shell132.ShellClass = New Shell132.ShellClass ()
    Shell.Open("C:\")      ' Öffne ein Ordnerfenster
    Shell.Explore("C:\")
    Shell.FileRun()
  End Sub
End Class
```


Mit diesem Ansatz kann die Einstellung *Option Strict On* benutzt werden, da die Objektvariable *Shell* als vom Typ *Shell32.ShellClass* deklariert und anschließend über den *New*-Konstruktor instantiiert wird.

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung realisierten Programmbeispiels im Ordner `\Beisp\Kap13\ShellInterop1` auf der Begleit-CD. Details sind dem Quellcode zu entnehmen.

13.5.5 Ein weiteres Beispiel für Shell-Aufrufe

In einem neuen Beispiel soll der Umgang mit weiteren Shell-Aufrufen untersucht werden. Für dieses Projekt wurde in der Entwicklungsumgebung ein Formular zum Aufrufen der Funktionen entworfen (Abbildung 13.8).

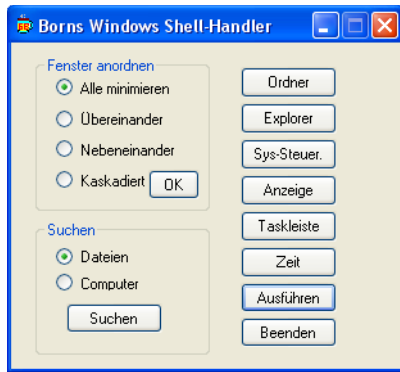


Abbildung 13.8: Formular zum Abrufen von Windows Shell-Funktionen

Die einzelnen Funktionen lassen sich über Schaltflächen und Optionsfelder anwählen. Dabei gelangen weitere Methoden der Windows-Shell zum Einsatz. Zur Manipulation der Fenster auf dem Desktop werden folgende Methoden benutzt:

- *TileVertically*: Ordnet alle auf dem Desktop angezeigten Fenster vertikal an.
- *TileHorizontally*: Ordnet alle auf dem Desktop angezeigten Fenster horizontal an.
- *CascadeWindows*: Die Fenster auf dem Desktop überlappend anzeigen.
- *MinimizeAll*: Alle auf dem Desktop angezeigten Fenster minimieren.
- *UndoMinimizeAll*: Diese Methode müsste eigentlich *UndoAll* heißen, denn sie nimmt die Wirkung des aus der obigen Gruppe aufgeführten Befehls wieder zurück. Bei *MinimizeAll* werden die Fenster in der alten Größe wiederhergestellt. Bei *CascadeWindows* passiert das Gleiche.

Die betreffenden Funktionen werden in diesem Beispiel über die Optionsfelder der Gruppe *Fenster anordnen* selektiert und dann über den Handler der OK-Schaltfläche aufgerufen. Neben den hier und weiter oben aufgeführten Methoden der Windows-Shell benutzt das Beispiel noch folgende:

- *ControlPanellItem*: Wird die Methode mit einer leeren Zeichenkette als Argument aufgerufen, öffnet sie das Fenster der Systemsteuerung. Alternativ lässt sich der Name einer CPL-Datei übergeben, um den betreffenden Eigenschaftendialog zu öffnen. Mit *Shell.ControlPanellItem("Desk.cpl")* wird das Dialogfeld *Eigenschaften von Anzeige* aufgerufen.
- *TrayProperty*: Öffnet das Eigenschaftensfenster der Taskleiste.
- *SetTime*: Öffnet das Eigenschaftensfenster für das Datum und die Uhrzeit.
- *FindFiles*: Öffnet das Fenster zum Suchen nach Dateien.
- *FindComputer*: Öffnet das Fenster zum Suchen nach Computern.

Neben diesen Methoden gibt es noch weitere, die im Beispiel aber nicht benutzt werden. Sie können diese Funktionen beispielsweise im Objektbrowser der Entwicklungsumgebung anzeigen lassen (Abbildung 13.9). Rufen Sie hierzu die Seite des Objektbrowsers über die Schaltfläche der Symbolleiste auf. Anschließend wählen Sie im Listenfeld *Durchsuchen* den Befehl *Benutzerdefinierter Komponentensatz bearbeiten*. Im gleichnamigen Dialogfeld müssen Sie dann auf der Registerkarte *COM* den Eintrag *Microsoft Shell Controls and Automation* per Mausklick wählen und dann mittels der Schaltfläche *Hinzufügen* in die Liste ausgewählter Komponenten übertragen. Schließen Sie das Dialogfeld über die OK-Schaltfläche. Wenn Sie danach das Listenfeld *Durchsuchen* des Objektbrowsers auf *Benutzerdefinierte Komponenten* stellen, wird der Eintrag »Shell32« eingeblendet und Sie können dessen Member inspizieren.

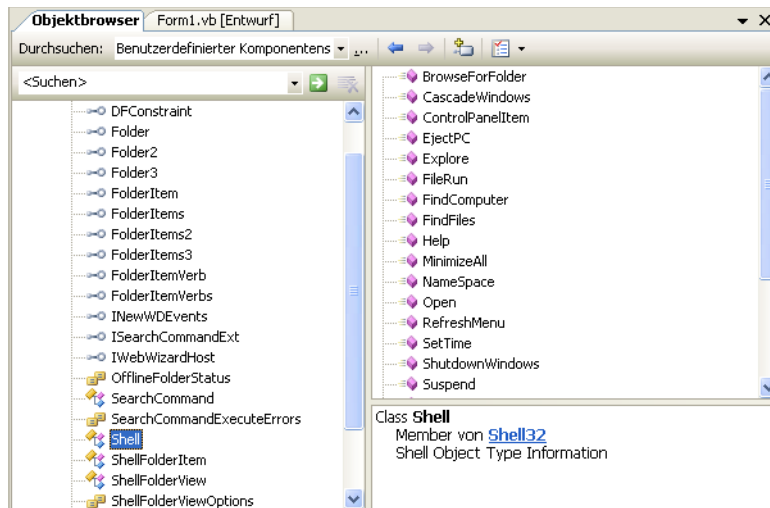


Abbildung 13.9: Methoden der Windows-Shell

Hinweis

Wenn Sie über die *Shell*-Funktion von Visual Basic 2005 das Windows-Modul *Control.exe* aufrufen, lässt sich durch geschickte Angabe von Parametern fast jede beliebige Registerkarte der Eigenschaftsfenster der Systemsteuerung öffnen. Die Zeichenkette "*Control.exe Main.cpl,@1,0*" enthält den Befehl, um die Registerkarte zur Einstellung der Tastatur zur Anzeige zu bringen. Durch Variation der CPL-Datei sowie der angegebenen Parameter lassen sich weitere Registerkarten abrufen. Eine Dokumentation der Parameter finden Sie z.B. unter http://www.winfaq.de/faq_html/tip0564.htm. Sie finden das komplette Projekt im Ordner *\Beisp\Kap13\WinShell* auf der Begleit-CD. Laden Sie die Datei *WinShell.sln*, um das Projekt in der Microsoft-Entwicklungsumgebung zu testen. Details zur Implementierung entnehmen Sie dem Quellcode der Datei *Form1.vb*.

13.6 Das NotifyIcon-Steuerelement verwenden

Manche Windows-Anwendungen besitzen die Möglichkeit, sich im Infobereich der Taskleiste (auch als Tray-Bereich oder Notify-Bereich bezeichnet) zu verankern. Beim Minimieren des Anwendungsfensters verschwindet dieses komplett vom Desktop und auch die Schaltfläche in der Taskleiste wird ausgeblendet. Stattdessen erscheint ein Anwendungssymbol im Statusbereich der Taskleiste. Mit dem *NotifyIcon*-Steuerelement bietet das .NET Framework eine einfache Möglichkeit, um Windows-Anwendungen mit dieser Fähigkeit auszustatten. Dies soll an einem einfachen Beispiel demonstriert werden.

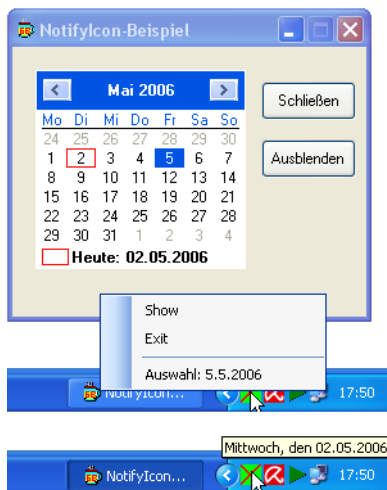


Abbildung 13.10: Menüs und Anzeigen des NotifyIcon-Beispiels

Die betreffende Anwendung meldet sich beim Start mit einem kleinen Fenster, in dem ein Kalenderblatt eingeblendet ist (Abbildung 13.10). Über die Schaltfläche *Schließen* wird die Anwendung beendet. Die Schaltfläche *Ausblenden* bewirkt, dass das Fenster verschwindet und ein Symbol im Statusbereich der Taskleiste erscheint. Zeigt der Benut-

zer per Maus auf das Symbol, wird das aktuelle Datum als QuickInfo eingeblendet. Ein Klick mit der rechten Maustaste auf das Symbol blendet ein Kontextmenü im Statusbereich der Taskleiste ein. Der untere Menübefehl zeigt das im Kalenderblatt vom Benutzer per Mausklick markierte Datum. Der Befehl *Exit* beendet die Anwendung und mit *Show* lässt sich das Anwendungsfenster auf dem Desktop einblenden. Ist das Anwendungsfenster sichtbar, verschwindet das Symbol im Infobereich der Taskleiste.

13.6.1 Entwurf des Formulars für das NotifyIcon-Beispiel

Um eine Windows-Anwendung mit einem NotifyIcon auszustatten bzw. um das oben skizzierte Beispiel zu realisieren, müssen Sie zuerst das Formular für die Anwendung entwerfen. Gehen Sie in folgenden Schritten vor:



Abbildung 13.11: Entwurf des NotifyIcon-Beispiels

1. Legen Sie in der Entwicklungsumgebung ein neues Windows-Projekt mit einem Formular an und setzen Sie ggf. die gewünschten Projekteigenschaften.
2. Ergänzen Sie das Formular um ein *MonthCalendar*-Steuerelement und fügen Sie die beiden Schaltflächen *Schließen* sowie *Ausblenden* im Formular ein (Abbildung 13.11).
3. Fügen Sie ein *ContextMenuStrip*-Steuerelement zum Formular hinzu und definieren Sie die in Abbildung 13.11 sichtbaren drei Menüeinträge.
4. Fügen Sie ein *NotifyIcon*-Steuerelement im Formular ein und setzen Sie dessen Eigenschaften.

Der *ContextMenuStrip*-Eigenschaft des Fensters ist über das zugehörige Listenfeld der Name des *ContextMenuStrip*-Steuerelements zuzuweisen. Dies stellt sicher, dass beim Klick auf das Symbol im Statusbereich der Taskleiste ein Kontextmenü geöffnet wird. Weiterhin müssen Sie der Eigenschaft *Icon* des *NotifyIcon*-Steuerelements ein Symbol zuweisen. Dieses Symbol ist später zur Laufzeit im Infobereich der Taskleiste bei ausgeblendetem Fenster sichtbar. Der als QuickInfo beim Zeigen auf das Symbol eingeblendete Text wird über die *Text*-Eigenschaft des Steuerelements definiert.

Um zu verhindern, dass der Benutzer ein Formular zur Laufzeit in der Größe anpasst, setzen Sie dessen Eigenschaft *FormBorderStyle* auf den Wert *FixedDialog*. Soll ein normaler Fensterrand verwendet werden, besteht die Möglichkeit, die Eigenschaften *MinimumSize* und *MaximumSize* vorzugeben (z.B. auf den Wert der Eigenschaft *Size* setzen). Dann kann der Benutzer die Formulargröße nur noch zwischen den angegebenen Eigenschaftswerten variieren. Wenn Sie dann noch die *Maximieren*-Schaltfläche über den Eigenschaftswert *MaximizeBox = false* ausblenden, kann der Anwender das Fenster nur noch minimieren oder schließen.

13.6.2 Code für die Ereignisbehandlungsroutinen

Die Funktionalität des Beispiels wird durch entsprechende Anweisungen in den Ereignisbehandlungsroutinen implementiert. Beim Anklicken der Schaltfläche *Schließen* oder des Kontextmenübefehls *Exit* terminiert das Programm. In den betreffenden Ereignisbehandlungsroutinen wird eine *Application.Exit()*-Anweisung untergebracht. Alternativ können Sie die *Close()*-Methode des Formularobjekts (z.B. *Me.Close()*) zum Schließen der Anwendung verwenden.

Ausblenden des Formulars

Klickt der Benutzer auf die Schaltfläche *Ausblenden*, muss das Formular vom Desktop verschwinden und als Symbol im Infobereich der Taskleiste auftauchen. Hierzu werden in der *Click*-Ereignisbehandlungsroutine der Schaltfläche folgende Anweisungen hinterlegt:

```
Private Sub Button2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button2.Click
    ' Schaltfläche, um Formular auszublenden
    Me.Visible = False           ' Formular ausblenden
    Me.NotifyIcon1.Visible = True ' Icon im Tray einblenden
End Sub
```

Hier wird die Eigenschaft *Visible* des Formulars auf den Wert *false* gesetzt, so dass dieses vom Desktop verschwindet. Gleichzeitig ist die *Visible*-Eigenschaft des *NotifyIcon*-Steuerelements auf *true* zu stellen, damit das Symbol der Anwendung im Statusbereich der Taskleiste erscheint. Beim Start der Anwendung wird diese Eigenschaft in der *Load*-Ereignisbehandlungsroutine des Formulars auf *false* gesetzt, um die Anzeige des Symbols im Infobereich zu unterdrücken.

Einblenden des aktuellen Datums beim Zeigen auf das Symbol

Zeigt der Benutzer auf das im Statusbereich der Taskleiste sichtbare Symbol, soll das aktuelle Datum mit dem Wochentag als *QuickInfo* eingeblendet werden. Das Zeigen auf das Symbol löst ein *MouseMove*-Ereignis aus, welches sich zur Anzeige der gewünschten Informationen nutzen lässt. Die *QuickInfo* wird selbsttätig eingeblendet, in der Ereignisbehandlungsroutine muss lediglich die *Text*-Eigenschaft des *NotifyInfo*-Steuerelements entsprechend gesetzt werden. Hierzu werden folgende Anweisungen verwendet:

```
Private Sub NotifyIcon1_MouseMove(ByVal sender As Object, _  
    ByVal e As System.Windows.Forms.MouseEventArgs) _  
    Handles NotifyIcon1.MouseMove  
' beim Zeigen auf das Icon - Datum einblenden  
Dim txt As String  
txt = WeekdayName(Weekday(Me.MonthCalendar1.TodayDate))  
txt = txt & ", den "  
txt = txt & Me.MonthCalendar1.TodayDate  
Me.NotifyIcon1.Text = txt  
End Sub
```

Über *WeekdayName* wird der Name des Wochentags (kulturabhängig) aus dem mit *Weekday* ermittelten Wochentag berechnet. Das aktuelle Datum liefert die *TodayDate*-Eigenschaft des *MonthCalendar*-Steuerelements.

Einblenden des gewählten Datums im Kontextmenü

Der Benutzer kann im Kalendersteuerelement beliebige Datumswerte anklicken. Das zuletzt markierte Datum soll beim Öffnen des Kontextmenüs im untersten Befehl eingeblendet werden. Hierzu lässt sich das *Paint*-Ereignis des Kontextmenüs verwenden. Dort wird der *Text*-Eigenschaft des betreffenden Kontextmenübefehls der betreffende Datumswert mit folgendem Code zugewiesen:

```
Private Sub DatumToolStripMenuItem_Paint(ByVal sender As Object, _  
    ByVal e As System.Windows.Forms.PaintEventArgs) _  
    Handles DatumToolStripMenuItem.Paint  
' wird beim Anzeigen des Kontextmenüs aufgerufen  
' blende selektiertes Datum im untersten Menübefehl ein  
DatumToolStripMenuItem.Text = "Auswahl: " & _  
    Day(Me.MonthCalendar1.SelectionStart).ToString() & "." & _  
    Month(Me.MonthCalendar1.SelectionStart).ToString() & "." & _  
    Year(Me.MonthCalendar1.SelectionStart).ToString()  
End Sub
```

Der vom Benutzer markierte Datumswert wird über die *SelectionStart*-Eigenschaft des *MonthCalendar*-Steuerelements geliefert. Die obigen Anweisungen benutzen die *Day*-, *Month*- und *Year*-Funktionen, um die betreffenden Informationen formatiert als Menübefehl darzustellen.

Anzeige des ausgeblendeten Formulars

Wählt der Benutzer den Kontextmenübefehl *Show* des *NotifyIcon*-Steuerelements, sorgt die *Click*-Ereignisbehandlungsroutine dafür, dass das Formular wieder sichtbar wird und dass das Symbol im Infobereich verschwindet.

```
Private Sub ShowToolStripMenuItem_Click( _
    ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles ShowToolStripMenuItem.Click
    ' Kontextmenü "Show" Anzeigen des Formulars
    Me.Activate()           ' in den Vordergrund
    Me.Visible = True       ' Formular sichtbar
    Me.NotifyIcon1.Visible = False ' Icon im Tray ausblenden
End Sub
```

Der Aufruf der *Activate()*-Methode des Formulars bewirkt, dass dieses in den Vordergrund geholt wird. Um das Formular sichtbar zu machen, wird dessen *Visible*-Eigenschaft auf *true* gesetzt. Gleichzeitig wird die *Visible*-Eigenschaft des *NotifyIcon*-Steuerelements auf *false* eingestellt, um das Symbol im Infobereich auszublenden.

Hinweis

Sie finden die Projektdateien des kompletten Beispiels im Ordner `\Beisp\Kap13\TrayIcon` der Begleit-CD. Laden Sie das Projekt in der Entwicklungsumgebung, lassen sich die Details des Quellcodes im Modul *Form1.vb* nachsehen.

13.7 Zugriff auf Systeminformationen

Das .NET Framework bietet über die Klasse *Environment* die Möglichkeit, eine Reihe an Informationen über die Umgebung abzufragen. So können Sie die Betriebssystemversion, die Version des .NET Framework, den aktuell angemeldeten Benutzer, den NET-BIOS-Namen der Maschine und vieles mehr ermitteln. In einem einfachen Beispiel möchte ich die Techniken vorstellen, mit denen sich die betreffenden Informationen abrufen lassen.

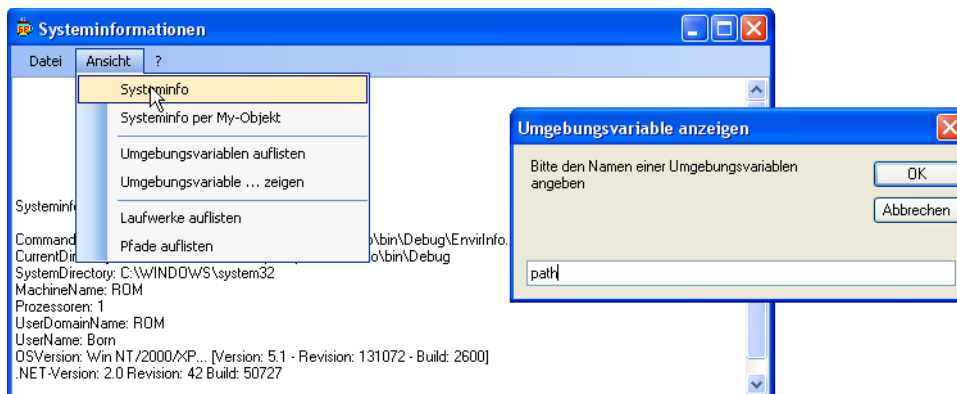


Abbildung 13.12: Abrufen verschiedener Systeminformationen

Zur Demonstration habe ich die verschiedenen Techniken zum Abrufen der Informationen in eine kleine als *Systeminformationen* bezeichnete Anwendung gepackt. Diese wurde als Projekt in der Entwicklungsumgebung realisiert und besteht aus einem Formular mit einer Menüleiste sowie einem *TextBox*-Steuerelement zur Anzeige der abgerufenen Daten. Über das Menü *Datei* lässt sich die Anwendung beenden. Zudem gibt es dort den Befehl *Neu*, um den Inhalt des *TextBox*-Steuerelements zu löschen. Das Menü *Ansicht* bietet dem Benutzer verschiedene Optionen zum Abrufen unterschiedlicher Systemdaten. Dies reicht von Angaben über die Betriebssystemversion über die Ermittlung aller Umgebungsvariablen samt ihren Werten bis zur Anzeige aller Laufwerke und Pfade zu wichtigen Systemordnern (Abbildung 13.12).

Hinweis

Sie finden das Projekt mit den Projektdateien auf der Begleit-CD zum Buch im Ordner `\Beisp\Kap13\EnvirInfo`. Laden Sie das Projekt über die *.sln*-Datei in die Microsoft-Entwicklungsumgebung und starten Sie die Anwendung, lassen sich die Systeminformationen über die Befehle des Menüs *Ansicht* abrufen.

13.7.1 Abrufen der Umgebungseinstellungen

Über den Befehl *Ansicht/Systeminfo* zeigt das Beispielprogramm die Umgebungseinstellungen wie Betriebssystemversion, das aktuelle Verzeichnis, das Systemverzeichnis oder die Informationen über Benutzername und NetBIOS-Maschinenname. Auf diese Informationen lässt sich über die *Environment*-Klasse zugreifen. Sobald der Zugriffspfad auf die Klasse *System.Environment* mit *Imports* vereinbart wurde, können Sie über *Environment* auf die Member zugreifen. Einige dieser Eigenschaften werden dabei als Zeichenketten zurückgeliefert. Andere Eigenschaften stellen ihrerseits Objekte dar, über den Eigenschaften die gewünschten Informationen extrahiert werden können. Das folgende Codefragment zeigt die Prozedur, die von der Ereignisbehandlungsroutine des Menübefehls *Ansicht/Systeminfo* aufgerufen wird. Die Prozedur ermittelt die Informationen und schreibt diese in das Textfeld *TextBox1* des Formulars.

```
Private Sub ShowSysInfo()  
    ' zeige alle Umgebungseinstellungen an  
    Dim tmp As String = "Systeminformationen aus Environment-Klasse" _  
        & vbCrLf & vbCrLf  
    ' stelle jetzt alle Informationen zusammen  
    tmp = tmp & "CommandLine: " & Environment.CommandLine & vbCrLf & _  
        "CurrentDirectory: " & Environment.CurrentDirectory & vbCrLf & _  
        "SystemDirectory: " & Environment.SystemDirectory & vbCrLf & _  
        "MachineName: " & Environment.MachineName & vbCrLf & _  
        "Prozessoren: " & Environment.ProcessorCount & vbCrLf & _  
        "UserDomainName: " & Environment.UserDomainName & vbCrLf & _  
        "UserName: " & Environment.UserName & vbCrLf
```

Listing 13.4: Ermitteln von Systemdaten über die *Environment*-Klasse


```

If Environment.OSVersion.Platform = PlatformID.Win32NT Then
    tmp = tmp & "OSVersion: Win NT/2000/XP... ["
ElseIf Environment.OSVersion.Platform = PlatformID.Win32Windows Then
    tmp = tmp & "OSVersion: Win 9x [" & vbCrLf
End If
tmp = tmp & "Version: " & Environment.OSVersion.Version.Major & _
    "." & Environment.OSVersion.Version.Minor & _
    " - Revision: " & Environment.OSVersion.Version.Revision & _
    " - Build: " & Environment.OSVersion.Version.Build & "]" & vbCrLf

tmp = tmp & ".NET-Version: " & Environment.Version.Major & "." & _
    Environment.Version.Minor & " Revision: " & _
    Environment.Version.Revision & " Build: " & _
    Environment.Version.Build & vbCrLf
Me.TextBox1.Text = tmp
End Sub

```

Listing 13.4: Ermitteln von Systemdaten über die Environment-Klasse (Forts.)

In einigen Anweisungen sehen Sie, wie die Eigenschaft als Objekt interpretiert und über dessen Eigenschaft auf die gewünschten Werte zugegriffen wird (z.B. *env.OSVersion.Version.Major*).

Verwendung des My-Objekts zum Ermitteln von Systeminformationen

Bereits in *Kapitel 7* wurde auf das neu in .NET 2.0 eingeführte *My*-Objekt hingewiesen. Dieses Objekt stellt einen schnellen Zugriff auf verschiedene Systemdaten bereit. Dies soll im Beispielprogramm genutzt werden, um bestimmte Systeminformationen abzurufen. Mit *My.Computer.Name* lässt sich z.B. der Netzwerkname des aktuellen Rechners ermitteln, während sich der verfügbare Arbeitsspeicher mittels *My.Computer.Info.TotalPhysicalMemory* ermitteln lässt. Der nachfolgende Codeausschnitt zeigt die Anweisungen der Prozedur *ShowSysInfoMy*, die über die *Click*-Ereignisbehandlungsroutine des Befehls *Ansicht/Systeminfo per My-Objekt* aufgerufen wird:

```

Private Sub ShowSysInfoMy()
    ' zeige SystemInfos über My-Objekt an
    Dim tmp As String = "Systeminformationen per My-Objekt" & _
        vbCrLf & vbCrLf
    Dim wert As Double
    Dim i As System.IO.DriveInfo ' für Laufwerksauflistung

    ' stelle jetzt die Informationen zusammen
    tmp = tmp & "MachineName: " & My.Computer.Name & vbCrLf
    wert = My.Computer.Info.TotalPhysicalMemory / 1024.0 / 1024.0

```

Listing 13.5: Ermitteln von Systemdaten über das My-Objekt

```

tmp = tmp & "Arbeitsspeicher (vorhanden): " & _
    Microsoft.VisualBasic.Format(wert, "###.00 MB") & vbCrLf
wert = My.Computer.Info.AvailablePhysicalMemory / 1024.0 / 1024.0
tmp = tmp & "Arbeitsspeicher (verfügbar): " & _
    Microsoft.VisualBasic.Format(wert, "###.00 MB") & vbCrLf
wert = My.Computer.Info.TotalVirtualMemory / 1024.0 / 1024.0
tmp = tmp & "Virtueller Speicher (vorhanden): " & _
    Microsoft.VisualBasic.Format(wert, "###.00 MB") & vbCrLf
wert = My.Computer.Info.AvailableVirtualMemory / 1024.0 / 1024.0
tmp = tmp & "Virtueller Speicher (verfügbar): " & _
    Microsoft.VisualBasic.Format(wert, "###.00 MB") & vbCrLf
' Peripherie-Infos
tmp = tmp & "Maustyp: " & My.Computer.Mouse.GetType.ToString _
    & vbCrLf & "Mausrad vorhanden: " & _
    My.Computer.Mouse.WheelExists.ToString & vbCrLf & _
    "Maustasten getauscht: " & _
    My.Computer.Mouse.ButtonsSwapped.ToString & vbCrLf & _
    "NumLock-Taste gedrückt: " & _
    My.Computer.Keyboard.NumLock.ToString & vbCrLf & _
    "ScrollLock-Taste gedrückt: " & _
    My.Computer.Keyboard.ScrollLock.ToString & vbCrLf & _
    "CapsLock-Taste gedrückt: " & _
    My.Computer.Keyboard.CapsLock.ToString & vbCrLf & _
    "Strg-Taste gedrückt: " & _
    My.Computer.Keyboard.CtrlKeyDown.ToString & vbCrLf & _
    "Alt-Taste gedrückt: " & _
    My.Computer.Keyboard.AltKeyDown.ToString & vbCrLf & _
    "Umschalt-Taste gedrückt: " & _
    My.Computer.Keyboard.ShiftKeyDown.ToString & vbCrLf & vbCrLf
' Betriebssystem-Infos
tmp = tmp & "Betriebssystem: " & My.Computer.Info.OSFullName _
    & vbCrLf & _
    "-Plattform: " & My.Computer.Info.OSPlatform & vbCrLf & _
    "-Version: " & My.Computer.Info.OSVersion & vbCrLf & _
    "Netzwerk vorhanden: " & _
    My.Computer.Network.IsAvailable.ToString & vbCrLf & _
    "Benutzername: " & My.User.Name & vbCrLf & _
    "Benutzer authentifiziert: " & My.User.IsAuthenticated _
    & vbCrLf & vbCrLf
' Filesystem-Infos

```

Listing 13.5: Ermitteln von Systemdaten über das My-Objekt (Forts.)

```

tmp = tmp & "Aktuelles Verzeichnis: " & _
    My.Computer.FileSystem.CurrentDirectory & vbCrLf

tmp = tmp & "Drives: " ' Laufwerke auflisten

For Each i In My.Computer.FileSystem.Drives
    tmp = tmp & i.ToString & " " ' Laufwerksbuchstabe geht immer
Next

' Jetzt Laufwerksdaten für Laufwerke ermitteln
tmp = tmp & vbCrLf
For Each i In My.Computer.FileSystem.Drives
    If i.IsReady Then ' Lw bereit, hole weitere Daten
        tmp = tmp & i.ToString & " Volume: " & i.VolumeLabel & _
            " Typ: " & i.DriveType & " -> Format: " & i.DriveFormat
        wert = i.TotalSize / 1024.0 / 1024.0 / 1024.0
        tmp = tmp & vbCrLf & " -> Kapazität: " & _
            Microsoft.VisualBasic.Format(wert, "0.## GB ")
        wert = i.AvailableFreeSpace / 1024.0 / 1024.0 / 1024.0
        tmp = tmp & vbCrLf & " Freie Kapazität: " & _
            Microsoft.VisualBasic.Format(wert, "0.## GB ")
    End If
Next
tmp = tmp & vbCrLf & "Spezialverzeichnisse" & vbCrLf
tmp = tmp & "AllUsersApplicationData: " & _
    My.Computer.FileSystem.SpecialDirectories.AllUsersApplicationData _
    & vbCrLf
tmp = tmp & "CurrentUserApplicationData: " & _
    My.Computer.FileSystem.SpecialDirectories.CurrentUserApplicationData
    & vbCrLf
tmp = tmp & "Desktop: " & _
    My.Computer.FileSystem.SpecialDirectories.Desktop & vbCrLf
tmp = tmp & "MyDocuments: " & _
    My.Computer.FileSystem.SpecialDirectories.MyDocuments & vbCrLf
tmp = tmp & "MyMusic: " & _
    My.Computer.FileSystem.SpecialDirectories.MyMusic & vbCrLf
tmp = tmp & "MyPictures: " & _
    My.Computer.FileSystem.SpecialDirectories.MyPictures & vbCrLf
tmp = tmp & "ProgramFiles: " & _
    My.Computer.FileSystem.SpecialDirectories.ProgramFiles & vbCrLf
tmp = tmp & "Programs: " & _

```

Listing 13.5: Ermitteln von Systemdaten über das My-Objekt (Forts.)

```

    My.Computer.FileSystem.SpecialDirectories.Programs & vbCrLf
tmp = tmp & "Temp: " & _
    My.Computer.FileSystem.SpecialDirectories.Temp & vbCrLf

Me.TextBox1.Text = tmp
End Sub

```

Listing 13.5: Ermitteln von Systemdaten über das My-Objekt (Forts.)

Die Funktion ermittelt eine Reihe an Systemdaten und zeigt diese im Textfeld der Anwendung an. Zur Formatierung der Kapazitätsangaben wird die Visual-Basic-Funktion *Format* verwendet. Bei der Abfrage der Laufwerkseigenschaft ist wichtig, dass über die Eigenschaft *IsReady* geprüft wird, ob überhaupt ein Medium zugreifbar ist. Findet diese Prüfung nicht statt, löst die Abfrage von Medieneigenschaften bei Wechseldatenträgern bei einem fehlenden Medium einen Laufzeitfehler aus. Details sind dem obigen Codeauszug zu entnehmen. Interessant ist, dass die Abfrage dieser Informationen über das My-Objekt wesentlich länger als der Zugriff über die *Environment*-Klasse dauert. Zudem stehen über das My-Objekt nicht alle in der *Environment*-Klasse verfügbaren Eigenschaften zur Verfügung.

13.7.2 Umgebungsvariable anzeigen

Die in Windows definierten Umgebungsvariablen werden im Environment-Bereich des laufenden Prozesses gespiegelt und lassen sich über die *GetEnvironmentVariables()*-Methode der Klasse *Environment* als Auflistung abrufen. Da eine Umgebungsvariable immer aus einem Namen und dem Wert besteht, haben die Microsoft-Entwickler den Rückgabewert der Methode als *Dictionary*-Typ implementiert. Sie müssen also die Variable zur Aufnahme der Auflistung folgendermaßen vereinbaren:

```
Dim env As IDictionary
```

Die einzelnen Elemente eines Dictionary lassen sich dann über *obj.key* (Schlüssel oder hier der Name der Umgebungsvariablen) und über *obj.value* (Wert) hinterfragen. In der folgenden Codesequenz wird eine *For Each*-Schleife benutzt, um alle Elemente des Dictionary zu durchlaufen. Dann werden die Schlüssel und die Werte in eine Zeichenkettenvariable *tmp* übertragen. Da der Datentyp von *i.Key* und *i.Value* keine Zeichenkette ist, erfolgt wegen der Angabe *Option Strict On* eine explizite Typkonvertierung mittels *CType*.

```

Private Sub ShowEnv()
    ' Umgebungsvariablen anzeigen
    Dim i As DictionaryEntry
    Dim env As IDictionary
    Dim tmp As String = "Umgebungsvariablen" & vbCrLf & vbCrLf
    env = GetEnvironmentVariables() ' Hole Umgebungsvariable
    For Each i In env                ' über alle Dictionary-Einträge
        tmp = tmp & CType(i.Key, String) ' Schlüssel
        tmp = tmp & " Wert: "
    Next i
End Sub

```

```

    tmp = tmp & CType(i.Value, String) & " Wert"
    tmp = tmp & vbCrLf
Next
Me.TextBox1.Text = tmp    ' In Textbox
End Sub

```

In der letzten Anweisung schreibt das Programm die ermittelten Ergebnisse in das Textfeld des Formulars. Der obige Code zeigt den *Click*-Ereignishandler für den Menübefehl *Ansicht/Umgebungsvariablen auflisten*.

Möchten Sie dagegen den Wert einer speziellen Umgebungsvariablen abfragen, können Sie direkt auf die *GetEnvironmentVariable()*-Methode der Klasse *Environment* zugreifen. Die Methode erwartet als Parameter den Namen der Umgebungsvariablen und liefert deren Wert zurück.

```
wert = GetEnvironmentVariable(name)
```

Existiert die angegebene Umgebungsvariable nicht, liefert die Methode eine leere Zeichenkette zurück. Im Beispielprogramm verwendet der Ereignishandler des Menübefehls *Ansicht/Umgebungsvariable ... zeigen* einen *InputBox*-Dialog (der *Microsoft.VisualBasic*-Kompatibilitätsklasse) zur Abfrage des Namens der Umgebungsvariable.

13.7.3 Alle logischen Laufwerke auflisten

Die Liste aller im System bekannten logischen Laufwerke lässt sich über die Methode *GetLogicalDrives* abfragen. Die Methode liefert ein *String*-Array mit den Werten zurück. Das folgende Codefragment zeigt die Ereignisbehandlungsroutine des Menübefehls *Ansicht/Laufwerke Auflisten*:

```

Private Sub MenuItemDrives_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MenuItemDrives.Click
' Menü Ansicht/Laufwerke auflisten
Dim i As Integer
Dim drives() As String
Dim tmp As String = "Logische Laufwerke" & vbCrLf & vbCrLf
drives = env.GetLogicalDrives    ' Hole Auflistung
For i = 0 To drives.Length - 1
    tmp = tmp & drives(i) & vbCrLf
Next i
Me.TextBox1.Text = tmp    ' In Textfeld
End Sub

```

Das *String*-Array wird über eine *For*-Schleife durchlaufen. In dieser Schleife werden die Laufwerkskennbuchstaben in eine temporäre Variable übertragen. Das Ergebnis wird zum Schluss in das Textfeld des Formulars gestellt.

13.7.4 Pfade der Systemordner ermitteln

Windows benutzt eine Reihe von Ordnern zum Verwalten der verschiedenen Benutzerdaten (z.B. *Programs*, *Personal*, *Favorites* etc.). Möchten Sie auf einen solchen Ordner zugreifen, benötigen Sie den zugehörigen Pfad. Diese lassen sich aber über die Werte der *GetSpecialFolder*-Auflistung der Klasse *Environment.SpecialFolder* abfragen. Diese Auflistung enthält die Werte aller Spezialordner als numerische Werte. Falls Sie den Namen eines Spezialordners kennen, können Sie dessen Name und Pfad mit den folgenden Anweisungen ermitteln:

```
name = GetName(GetType(Environment.SpecialFolder), _
                SpecialFolder.DesktopDirectory)
pfad = GetFolderPath(SpecialFolder.DesktopDirectory)
```

Die erste Anweisung benutzt die *GetName()*-Methode der Klasse *Enum*, um den Namen des angegebenen Werts zu ermitteln (siehe auch den Abschnitt »Zugriff auf Enumeration mit Schleifen« in Kapitel 7). Die zweite Anweisung verwendet die *GetFolderPath()*-Methode der *Environment*-Klasse, um den Pfad zu ermitteln. Als Argument erwartet die Methode einen Wert aus der *Environment.SpecialFolder*-Auflistung. Um alle Einträge der *SpecialFolder*-Auflistung der Klasse *Environment* abzurufen, lässt sich eine Schleife verwenden:

```
For Each i In GetValues(GetType(Environment.SpecialFolder))
...
Next i
```

Diese Technik ist in Kapitel 7 in dem bereits erwähnten Abschnitt beschrieben. Die Methode *GetValues()* wird über die per *Imports*-Anweisung importierte Struktur *System.Enum* bereitgestellt. Die folgende Codesequenz stellt die Ereignisbehandlungsroutine des Menübefehls *Ansicht/Pfade auflisten* dar:

```
Private Sub MenuItemPath_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MenuItemPath.Click
' Menü Ansicht/Pfade auflisten
Dim i As Environment.SpecialFolder
Dim name As String
Dim tmp As String = "Systempfade" & vbCrLf & vbCrLf
For Each i In GetValues(GetType(Environment.SpecialFolder))
    name = GetName(GetType(Environment.SpecialFolder), i)
    tmp = tmp & CType(i, Integer).ToString & " : " ' Integer-Wert
    tmp = tmp & name & vbTab ' als Environment.SpecialFolder-Wert
    tmp = tmp & GetFolderPath(i) & vbCrLf ' Pfad holen
Next i
Me.TextBox1.Text = tmp ' In Textfeld
End Sub
```

Die *For*-Schleife bewirkt, dass wirklich alle Ordner der *SpecialFolders*-Auflistung bearbeitet werden. In der Schleife verwende ich verschiedene Techniken, um den numerischen Wert, den Namen des Spezialordners sowie dessen Pfad anzuzeigen.

Hinweis

Weitere Details zur Anwendung *Systeminformationen* entnehmen Sie bitte dem Quellcode der bereits Eingangs erwähnten Projektdatei. Schreibzugriffe auf Umgebungsvariablen wurden übrigens nicht implementiert. Falls Sie dies beabsichtigen, verwenden Sie die neu in .NET Framework 2.0 aufgenommene *SetEnvironmentVariable()*-Methode mit dem Parameter *EnvironmentVariableTarget*. Die Methode erstellt, ändert oder löscht eine Umgebungsvariable, die im aktuellen Prozess oder im Registrierungsschlüssel des Windows-Betriebssystems gespeichert ist.

13.8 Zugriff auf die Registrierung

Das .NET Framework bietet den Zugriff auf die Windows-Registrierung über die Klasse *Registry* im Namensraum *Microsoft.Win32* an. Dort finden Sie sowohl Methoden, um Registrierungseinträge zu lesen, als auch um Einträge zu schreiben, zu verändern und Werte oder Schlüssel zu entfernen. Dies setzt natürlich voraus, dass das Programm unter einem Benutzerkonto läuft, welches die notwendigen Berechtigungen zum Zugriff auf die Registrierung besitzt. Nachfolgend möchte ich an einigen einfachen Beispielen zeigen, wie sich auf die Registrierung zugreifen lässt.

13.8.1 Registrierungseintrag schreiben, lesen und löschen

Zum Zugriff auf die Registrierung stellt das .NET Framework eine Reihe an Methoden über die Klasse *Registry* bereit. Um diese Methoden in einem Programm nutzen zu können, sollten Sie als Erstes den Namensraum mit *Imports Microsoft.Win32* vereinbaren. Dann können Sie anschließend die Member der Klassen in Kurzform im Programmcode angeben.

Achtung

Wenn Sie ein neues Projekt anlegen, vergibt die Entwicklungsumgebung automatisch einen Namespace für die Anwendung, die aus dem Projektnamen abgeleitet wird. Dies führt dann aber ggf. zu Konflikten mit den Namensräumen bzw. Klassen, die Sie im Programmcode zu nutzen gedenken. Beim Erstellen des Beispiels hatte ich ein Projekt mit dem Namen *Registry* in der Entwicklungsumgebung angelegt und wunderte mich anschließend, dass Eingaben wie *Registry.ClassesRoot.OpenSubKey()* als fehlerhaft bemängelt wurden. Erst als ich den Namespace in den Projekteigenschaften umbenannt hatte, war das Problem behoben. Die Alternative wäre, die Namensräume beim Zugriff immer komplett anzugeben oder mit Aliasnamen zu arbeiten. Bei Registrierungszugriffen wäre der vollqualifizierte Name z.B. *Microsoft.Win32.Registry.ClassesRoot.OpenSubKey()*. Wird ein Aliasname mit *Imports Reg = Microsoft.Win32.Registry* vereinbart, könnte die Kurzform *Reg.ClassesRoot.OpenSubKey()* verwendet werden.

Einen Schlüssel anlegen

Um einen neuen Schlüssel anzulegen, verwenden Sie die *CreateSubKey()*-Methode der Klasse *RegistryKey*.

```
Dim oKey As RegistryKey  
  
oKey = Registry.ClassesRoot.CreateSubKey(".1Gborn")
```

Die obige Sequenz vereinbart eine Objektvariable vom Typ *RegistryKey*, der anschließend der neu angelegte Schlüssel zugewiesen werden soll. Zum Zugriff auf die Hauptschlüssel *HKCR*, *HKCU*, *HKLM* etc. muss einer der Member der *Registry*-Klasse (*ClassesRoot*, *CurrentConfig*, *ClassesRoot*, *CurrentUser*, *DynData*, *LocalMachine*, *PerformanceData*, *Users*) angegeben werden. Beachten Sie, dass je nach Betriebssystem nicht alle diese Hauptzweige vorhanden sind. *CreateSubKey* bezieht sich dann auf diesen Hauptschlüssel. Der Unterschlüssel ist als Argument zu übergeben, wobei auch eine Schlüsselhierarchie vereinbart werden kann. Die obige Angabe legt den Unterschlüssel *.1Gborn* im Zweig *HKEY_Classes_Root* an. Mit der Angabe *.1Gborn/Text* im Argument würden ggf. die Unterschlüssel *.1Gborn* und der Unterunterschlüssel *Text* erzeugt.

Einen Schlüssel öffnen

Das Öffnen eines Schlüssels ist über die *OpenSubKey()*-Methode der Klasse *Registry* möglich, wobei im Prinzip das Gleiche wie beim Anlegen neuer Schlüssel gilt.

```
Dim oKey As RegistryKey  
  
oKey = Registry.ClassesRoot.OpenSubKey(".1Gborn")
```

In dieser Sequenz wird auf den angegebenen Schlüssel zugegriffen. Die Referenz wird dann als Objekt in *oKey* gespeichert. Soll der Schlüssel mit Werten belegt werden, muss als zweites Argument der Wert *True* übergeben werden – der Wert *False* sperrt Schreibzugriffe, es lassen sich weder Werte noch Unterschlüssel hinzufügen.

```
oKey = Registry.ClassesRoot.OpenSubKey(key, True)
```

Voraussetzung beim Aufruf ist, dass der Schlüssel existiert. Anderenfalls wird ein Laufzeitfehler ausgelöst.

Schlüssel schließen

Zum Schließen eines geöffneten Schlüssels ist nur die Anweisung *obj.Close()* zu verwenden, wobei *obj* der Name der Objektvariablen mit der Referenz zum Schlüssel ist.

Werte schreiben

Das Schreiben von Werten in einen geöffneten Unterschlüssel ist ebenfalls kein größeres Thema. Sie verwenden die *SetValue()*-Methode der Klasse *RegistryKey*.


```
Dim oKey As RegistryKey

oKey = Registry.ClassesRoot.OpenSubKey(".1GBorn")
oKey.SetValue(valname, value)
```

Die obige Sequenz vereinbart eine Objektvariable vom Typ *RegistryKey*, öffnet den Schlüssel und legt unter dem im Argument *valname* angegebenen Namen den Wert *value* ab. Sofern Sie anstelle von *valname* eine leere Zeichenkette "" übergeben, wird der Standardwert des Schlüssels geschrieben.

Die Methode zum Ablegen eines Werts in der Registrierung erwartet im ersten Parameter den Namen des Werts. Diese Werte können, je nach Betriebssystem, neben dem Namen verschiedene Datentypen wie *String*, *DWORD*, *Binary* etc. aufweisen. Welcher Datentyp dem Wert beim Anlegen zugewiesen wird, hängt vom zweiten, an die Methode übergebenen Parameter ab. Die Methode bestimmt den Typ des Registrierungswerts automatisch aus dem Datentyp des zweiten Arguments. Wenn Sie eine Zeichenkette übergeben, wird diese auch als *String* im Schlüssel hinterlegt. Die folgende kurze Codesequenz demonstriert, wie sich drei Werte schreiben lassen:

```
Const key As String = ".1GBorn"
Const valname As String = "Test"
Const value As String = "Version 1.0"
Const valname1 As String = "DWORD"
Const value1 As Integer = 12
Const valname2 As String = "Binär"
Dim value2() As Byte = {&HC, &HD, &H1, &H22, &HFF} ' für Binary
oKey = Registry.ClassesRoot.CreateSubKey(key)
oKey.SetValue("", "123") ' Standardwert
oKey.SetValue(valname, value) ' Stringwert
oKey.SetValue(valname1, value1) ' DWORD
oKey.SetValue(valname2, value2) ' Binary
```

Durch geschickte Wahl des Datentyps können Sie also beeinflussen, wie Werte in der Registrierung abgebildet werden (Abbildung 13.13).

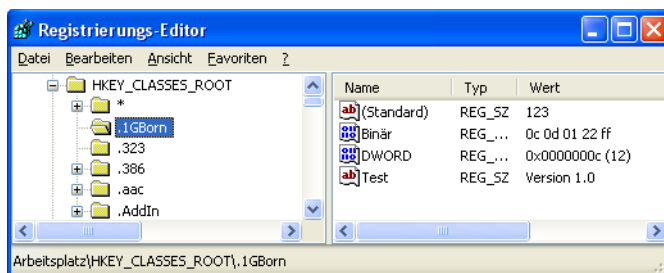


Abbildung 13.13: Verschiedene Werte im Schlüssel .1GBorn

Lesen von Werten

Um Werte aus einem Schlüssel auszulesen, deren Namen Sie kennen, lässt sich die *GetValue()*-Methode der Klasse *RegistryKey* verwenden:

```
Dim oKey As RegistryKey
Dim oWert As Object

oKey = Registry.ClassesRoot.OpenSubKey(".1Gborn")
oWert = oKey.GetValue("")
```

Nachdem der Schlüssel geöffnet wurde, liest die betreffende Methode den als Argument übergebenen Wert. Im aktuellen Wert ist der Name ein Leerstring, folglich wird der Standardwert des Schlüssels gelesen. Um den Wert *Test* zu lesen, ist die Anweisung so zu modifizieren:

```
oWert = oKey.GetValue("Test")
```

Leider gibt es beim Lesen der Werte noch zwei unangenehme Dinge, die beim Einsatz zu beachten sind. Existiert der zu lesende Wert nicht, löst die Methode einen Laufzeitfehler aus, den Sie abfangen sollten. Das zweite Problem: Das von *GetValue* zurückgelieferte Ergebnis ist vom Typ *Object* (damit Werte unterschiedlichen Typs zurückgelesen werden können). *String*- und *DWORD*-Werte lassen sich ggf. mit der *ToString()*-Methode in Zeichenketten wandeln und anzeigen. Beim Lesezugriff auf einen *Binary*-Wert enthält die Variable vom Typ *Object* aber ein Byte-Array mit den Werten. Sie müssen daher dieses Array auswerten. Die folgende Sequenz zeigt, wie sich die Werte des Felds abrufen lassen:

```
Dim oBin() As Byte      ' Hilfsarray zum Zurücklesen
...
oBin = oKey.GetValue(valname2) ' Binary-Wert
For Each i In oBin      ' Byte-Array bearbeiten
    MsgBox (i.ToString)
Next i
```

Allerdings ist die Vorgabe *Option Strict Off* erforderlich, da sonst keine implizierten Typkonvertierungen zwischen *Object* und dem Byte-Array möglich sind (die *GetValue()*-Methode liefert bei *Option Strict On* einen Fehler).

Lesen von Werten und Schlüsseln

Um Werte aus einem Schlüssel zu entfernen, wenden Sie die *DeleteValue()*-Methode der Klasse *RegistryKey* an:

```
oKey.DeleteValue(name)
```

Der Platzhalter *name* steht hier für den Namen des zu löschenden Werts, *oKey* muss auf den betreffenden Schlüssel verweisen. Der Standardwert des Schlüssels wird aber nicht gelöscht.

Soll ein Schlüssel gelöscht werden, muss dieser leer sein. Dann muss der übergeordnete Schlüssel geöffnet werden. Über die *DeleteSubKey()*-Methode wird der Schlüssel entfernt:

```
Registry.ClassesRoot.DeleteSubKey(key)
```

Die obige Anweisung legt den Hauptschlüssel *HKEY_Classes_Root* fest und löscht dann den in *key* angegebenen Unterschlüssel. Soll ein kompletter Zweig samt Inhalt gelöscht werden, ist die Methode *DeleteSubKeyTree* anzuwenden:

```
Registry.ClassesRoot.DeleteSubKeyTree(key)
```

Auch hier wird wieder der Hauptschlüssel angegeben, um einen Unterschlüssel zu entfernen.

Achtung

Wenden Sie beim Löschen von Werten und Schlüsseln erhöhte Vorsicht an. Durch Eingriffe in die Registrierung kann die Funktionalität von Windows beeinträchtigt werden.

Beispiel: Setzen und Löschen von Einträgen

An einem einfachen Beispiel soll der Umgang mit den obigen Methoden demonstriert werden. Die Anwendung benutzt eine Folge von Dialogfeldern zur Benutzerführung (Abbildung 13.14). Beim Ablauf kann der Benutzer einen Schlüssel *.1GBorn* im Zweig *HKEY_Classes_Root* der Registrierung anlegen. Der Name wurde so gewählt, dass der Schlüssel am Anfang der Liste auftaucht, sich also sehr schnell im Registrierungseditor identifizieren lässt (Abbildung 13.13). Anschließend schreibt das Programm verschiedene Werte, liest diese zurück und entfernt dann Werte und Schlüssel wieder aus der Registrierung. Durch diesen Ansatz reduziert sich der Code auf das Wesentliche. Laufzeitfehler werden zentral über die *On Error Goto Fehler*-Anweisung abgefangen.

Hinweis

Sie finden die Projektdateien der Windows-Anwendung im Ordner *\Beisp\Kap13\Registry* auf der Begleit-CD. Der Code befindet sich in der Datei *Registry.vb*. Im Verzeichnis *\Beisp\Kap13\Registry1* finden Sie ein zweites Projekt mit der Datei *Registry1.vb*. Diese Datei legt die Schlüsselhierarchie *HKEY_Classes_Root\.1GBorn\txt* an, schreibt einige Werte und löscht dann den Schlüssel *.1GBorn* über *DeleteSubKeyTree*. Details sind dem Quellcode zu entnehmen, Erläuterungen zu den verwendeten Methoden finden Sie auf den vorhergehenden Seiten.

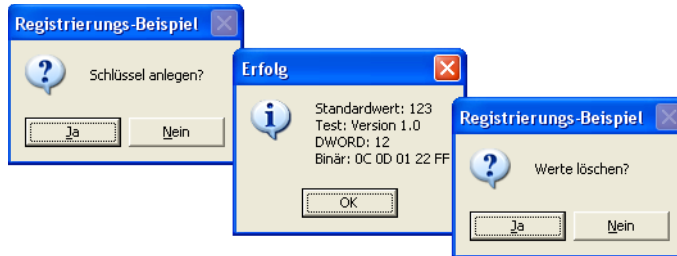


Abbildung 13.14: Teildialoge zur Benutzerführung

13.8.2 Unterschlüssel und Werte auflisten

Sofern Sie die Namen von Unterschlüsseln oder Werten nicht kennen, lässt sich nicht über die obigen Funktionen darauf zugreifen. Sie haben aber die Möglichkeit, den Inhalt eines vorgegebenen Schlüssels über eine Auflistung, getrennt nach Unterschlüsseln und Werten, abzufragen. Dies soll an einer kleinen Anwendung demonstriert werden (Abbildung 13.15).

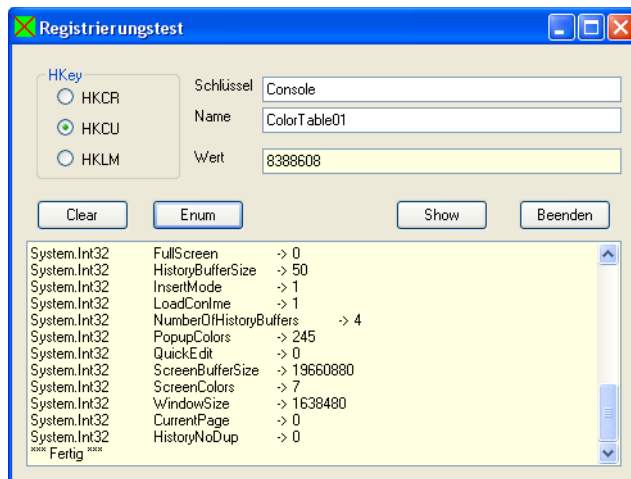


Abbildung 13.15: Registrierungseinträge lesen und auflisten

Über die Optionen der Gruppe *HKey* lässt sich wählen, welcher der Hauptregistrierungswege zu benutzen ist. Geben Sie im Feld *Schlüssel* einen Wert ein und ergänzen Sie ggf. das Feld *Name*. Über die Schaltfläche *Show* lässt sich der Wert des angegebenen Namens im Feld *Wert* abrufen. Die Schaltfläche *Enum* zeigt dagegen den Inhalt des im Feld *Schlüssel* angegebenen Registrierungszweigs an. Mit *Clear* lässt sich das Textfeld im unteren Teil des Formulars löschen.

Das Auflisten aller Schlüssel eines Unterschlüssels ist über die *GetSubKeyNames()*-Methode möglich.

```
oSubKeys = okey.GetSubKeyNames()
```

Hier wird vorausgesetzt, dass *okey* auf einen gültigen geöffneten Schlüssel verweist. Die Methode gibt ein String-Array zurück, welches hier der Variablen *oSubKeys* zugewiesen wird. Die Namen der Unterschlüssel lassen sich in folgender Schleife auswerten:

```
Dim txt As String = ""
For Each i In oSubKeys ' Alle Unterschlüssel
    txt = txt & "Schlüssel " & i.ToString() & vbCrLf
Next i
MsgBox (txt)
```

Sollen statt der Unterschlüssel die Namen der im Schlüssel hinterlegten Werte abgefragt werden, ist die *GetValueNames()*-Methode zu verwenden:

```
oValues = oKey.GetValueNames()
```

Diese liefert ebenfalls ein String-Array zurück, welches die Namen aller in *okey* hinterlegten Werte enthält. Zur Auswertung ließe sich folgende *For*-Schleife benutzen:

```
For Each i In oSubKeys ' Alle Werte
    MsgBox ("""" & i & """: " & okey.GetValue(i))
Next i
```

Der Schleifenindex *i* gibt den Namen des Werts an. Bei der Anzeige im Dialogfeld wird dieser Name in doppelte Anführungszeichen gesetzt, um den Standardwert als "" anzuzeigen. Dies wird in obiger Sequenz durch die Zeichenfolgen """" erreicht. Die beiden äußeren Anführungszeichen signieren eine Zeichenkette. Das innere Anführungszeichen muss in Visual Basic .NET verdoppelt werden, damit der Compiler dies als in der Zeichenkette einzufügendes Anführungszeichen erkennt.

Leider gibt es noch ein Problem bei dem oben gezeigten einfachen Ansatz. Ein Wert kann verschiedene Datentypen aufweisen. *GetValue()* liefert daher immer einen Wert vom Typ *Object* zurück. Bei *String*-Werten lässt sich dieser direkt in eine Zeichenkette konvertieren (*ToString*). Gleiches gilt für *DWORD*-Werte oder *REG_EXPAND_SZ*-Werte, wobei bei letzterem Datentyp die Platzhalter wie *%WinDir%* bereits automatisch expandiert werden. Probleme werfen die Datentypen *Binary* (Binär) und *REG_MULTI_SZ* auf, bei denen der zurückgelieferte Objektwert ein *Byte*-Array (Binary) oder ein *String*-Array (*REG_MULTI_SZ*) enthält. Dann müssen die einzelnen Werte des jeweiligen Arrays explizit aus dem Objekt extrahiert werden. Die folgende Codesequenz zeigt meine Implementierung des Programmblocks, der die Werte eines Schlüssels in einem TextBox-Steuerelement des Formulars hinterlegt. Neben der reinen Programmlogik habe ich noch eine Fehlerbehandlung über *Try..Catch End Try* hinzugefügt, um auftretende Fehler abzufangen.

```
Try
    oValues = oKey.GetValueNames() ' Auflistung der Werte
Catch ex As Exception
    MessageBox.Show("Der Schlüssel " & key & " _
        lässt sich nicht lesen", _
```

```

        "Fehler", MessageBoxButtons.OK, _
        MessageBoxIcon.Stop)
    Exit Sub
End Try
For Each i In oValues          ' alle Werte der Auflistung
    kTyp = oKey.GetValue(i).GetType().ToString ' Typ ermitteln
    wert = wert & kTyp & vbTab          ' Typ und ...
    If i = "" Then                    ' ... Name einblenden
        wert = wert & "(Standard) " & vbTab
    Else
        wert = wert & i & " " & vbTab
    End If

Try    ' jetzt die Werte nach Datentypen auswerten
Select Case kTyp
    Case "System.Byte[]"          ' Binary-Wert als Byte-Array
        oBin = oKey.GetValue(i)    ' in Byte-Array lesen
        wert = wert & "->" & vbCrLf ' Umbruch in Folgezeile
        For Each j In oBin 'Bin    ' Byte-Array bearbeiten
            If Hex(j).Length < 2 Then ' bei einer Ziffer 0 vorsetzen
                wert = wert & "0" & Hex(j) & " "
            Else
                wert = wert & Hex(j) & " "
            End If
        Next j
        wert = wert & vbCrLf          ' Zeilenumbruch
    Case "System.String[]"          ' Multi-String-Eintrag
        oStr = oKey.GetValue(i)    ' in String-Array
        wert = wert & vbCrLf          ' Umbruch in Folgezeile
        For Each j In oStr          ' String-Array bearbeiten
            ' zeilenweise in Ausgabestring mit vorgesetztem "-> "
            wert = wert & vbTab & vbTab & "-> " & j & vbCrLf
        Next j
        wert = wert & vbCrLf          ' Zeilenumbruch
    Case Else
        ' String, DWORD oder Multi_Expand implementiert,
        ' andere exotische Win 2K/XP Typen werfen eine Ausnahme
        wert = wert & "-> " & CType(oKey.GetValue(i), String) & vbCrLf
    End Select
Catch ex As Exception
    MessageBox.Show("Sorry, Fehler beim Lesen eines Werts" & vbCrLf & _
        "Typ " & kTyp & " nicht korrekt implementiert" & _

```

```

        vbCrLf & ex.Message, "Fehler", _
        MessageBoxButtons.OK, MessageBoxIcon.Warning)
    End Try
    Me.TextBoxList.AppendText(wert) ' Zwischeninfo anzeigen
    wert = "" ' lösche Inhalt
Next i

```

Die Programmsequenz benutzt zwei Variable *oBin* und *oStr*, um die Array-Elemente von Binärwerten und Multi-Strings auszuwerten. Sobald die Daten als Zeichenkette vorliegen, werden diese mittels der *AppendText()*-Methode des *TextBox*-Steuerelements in die Anzeige geschrieben. Dies hat den Vorteil, dass der Anwender sofort die bereits bearbeiteten Daten sieht.

Hinweis

Sie finden die Projektdateien des als Windows-Anwendung implementierten Beispiels im Ordner `\Beisp\Kap13\RegistryEnum`. Nach der Übersetzung öffnet dieses das in Abbildung 13.15 gezeigte Formular. Details zur Implementierung entnehmen Sie bitte dem Quellcode des Beispiels. Beachten Sie, dass *Option Strict Off* gesetzt ist, da das Programm implizite Typkonvertierungen zwischen dem von *GetValue* gelieferten *Object*-Typ und den Byte- bzw. String-Arrays benutzt (z.B. `oBin = oKey.GetValue(i)`). Im gleichen Verzeichnis finden Sie übrigens noch eine *.reg*-Datei für Windows 2000/XP, die einen Schlüssel *HKCR\1.Gborn* mit verschiedenen Datentypen in der Registrierung einträgt (einfach die *.reg*-Datei per Doppelklick importieren).

13.8.3 Remote-Registrierungszugriffe

Das .NET Framework unterstützt neben der lokalen Registrierung auch Remote-Registrierungszugriffe auf die Registrierung anderer Computer in einem Netzwerk. Im folgenden Beispiel möchte ich die auf der vorherigen Seite entwickelte Anwendung so umstellen, dass diese auch Registrierungsschlüssel auf anderen Maschinen auslesen kann. Das Programm benutzt das in Abbildung 13.16 gezeigte Anwendungsformular, welches jetzt zusätzlich noch ein Feld zur Eingabe des Maschinennamens aufweist. Geben Sie nichts in diesem Feld ein, beziehen sich alle Anfragen auf die lokale Registrierung. Ein Maschinennamen wie *Prag* bewirkt, dass das Programm einen Remotezugriff auf die betreffende Maschine startet. Ist der betreffende Schlüssel zugreifbar, listet das Programm die angefragten Informationen auf. Andernfalls wird der Zugriff abgewiesen und Sie erhalten einen Fehlerdialog angezeigt.

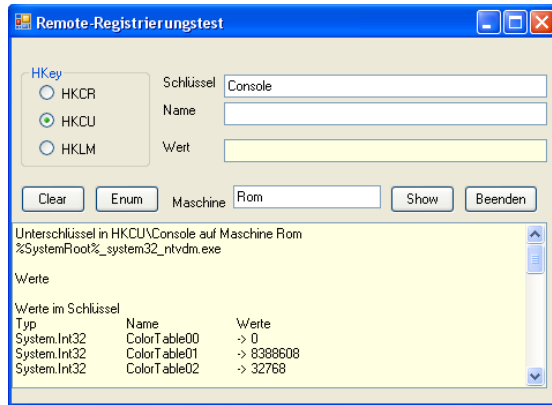


Abbildung 13.16: Variante zum Remote-Zugriff

Die Änderungen gegenüber dem obigen Beispiel sind marginal. In der betreffenden Click-Ereignisbehandlungsroutine der jeweiligen Schaltfläche muss der Handler erst eine Verbindung zum Remote-Computer herstellen. Hierzu kommt die *OpenRemoteBaseKey()*-Methode zum Einsatz. Die Methode erwartet im ersten Parameter eine Enumerationskonstante für die Remote-Registry-Struktur (Hive) (z.B. *RegistryHive.ClassesRoot*, *RegistryHive.CurrentUser*, *RegistryHive.LocalMachine* etc.) und im zweiten Argument den Namen des Remote-Computers. Die Methode liefert einen Wert vom Typ *RegistryKey* zurück, der dann auf den Hauptzweig der betreffenden Maschine verweist. Existiert der Zweig oder die Maschine nicht, löst dies einen Laufzeitfehler aus. Im nächsten Schritt ist dann der gewünschte Unterschlüssel auf der Maschine mit *OpenSubKey* zu öffnen. Im Gegensatz zu den vorherigen Beispielen wird dabei nicht ein Objekt der lokalen Hauptzweige (z.B. *Registry.ClassesRoot*) sondern die mit *OpenRemoteBaseKey*-Key initialisierte Objektvariable verwendet. Die folgende Sequenz zeigt die relevanten Codeteile – ich habe an dieser Stelle einmal auf die Behandlung von Laufzeitfehlern verzichtet:

```
Dim oRemKey As RegistryKey ' Remote-Objekt
Dim okey As RegistryKey

' Remote-Verbindung zu HKCR auf Rechner Prag herstellen
oRemKey = RegistryKey.OpenRemoteBaseKey( _
    RegistryHive.ClassesRoot, "Prag")
' Jetzt den Schlüssel .txt in HKCR öffnen
okey = oRemKey.OpenSubKey(".txt", False)
' hier können jetzt Befehle zum Lesen der Werte folgen
...
```

Sobald die Remote-Verbindung steht und der gewünschte Schlüssel geöffnet wurde, können Sie die weiter oben in den Beispielen beschriebenen Methode nutzen, um Werte und Unterschlüssel zu lesen oder zu verändern (falls das Zugriffsflag bei *OpenSubKey* auf *True* gesetzt war und das Programm die benötigten Privilegien besitzt).

Tipp

.NET Framework bietet Ihnen weitere Funktionen zum Zugriff auf die Registrierung. So lassen sich z.B. Sicherheitsberechtigungen auswerten oder anpassen. Konsultieren Sie ggf. die Hilfe zum .NET Framework, und sich über Details zu informieren. Das Beispiel zum Remote-Zugriff finden Sie im Ordner `\Beisp\Kap13\RemoteRegistry` auf der Begleit-CD. Laden Sie die `.sln`-Datei in der Entwicklungsumgebung, um den Code der Datei `Form1.vb` zu studieren. Die Übersetzung muss, wegen der Typkonvertierung, mit *Option Strict Off* erfolgen. Die betreffenden Methodenaufrufe zum Zugriff auf die Registrierung finden sich in den Ereignisbehandlungsroutinen der Schaltflächen `Show` und `Enum`.

Voraussetzung für den Zugriff auf die Registrierung einer fremden Maschine ist, dass das Programm die erforderlichen Zugriffsprivilegien besitzt (was z.B. der Fall ist, wenn Sie es unter dem Konto »Administrator« ausführen). Wichtig ist vor allem, dass sowohl auf der Quell- als auf der betreffenden Zielmaschine der Remote-Registrierungsdienst laufen muss (Windows 2000, Windows XP Professional und folgende Versionen sind bereits entsprechend vorbereitet). Das Beispiel wird also unter Windows XP Home Edition nicht funktionieren. Für einen ersten Test empfiehlt es sich, als Maschine den Netzwerknamen des lokalen Rechners zu verwenden. Ob Remote-Zugriffe auf die Registrierung der betreffenden Maschine zulässig sind, können Sie im Registrierungseditor `RegEdit.exe` über den Befehl *Mit Netzwerkregistrierung verbinden* des Menüs *Datei* testen (gibt es dort Konflikte beim Zugriff auf die Registrierung der Maschine, wird auch die .NET-Anwendung nicht funktionieren).

13.8.4 Zugriff auf INI-Dateien

Zum Zugriff auf INI-Dateien bietet .NET Framework keine Unterstützung. Es gibt im Internet aber einige Klassen, die über API-Aufrufe so etwas für .NET bieten (z.B. unter www.mentalis.org). Eine von der erwähnten Internetseite heruntergeladene Klasse finden Sie auf der Begleit-CD im Ordner `\Beisp\Kap13\INIReader\INIReader`. Der Visual-Basic-Code findet sich im Unterordner `\vb`. Sie können das Projekt im Ordner `\Beisp\Kap13\INIReader` laden und ausführen. Das Projekt ist als Konsolenanwendung implementiert. Beim Ausführen erzeugt das Programm eine INI-Datei (`test.ini`) auf dem Laufwerk C:\ und listet dann deren Inhalte auf Konsoleebene auf. Details entnehmen Sie bitte dem Programmcode sowie der dem Beispiel beiliegenden Hilfe.

13.9 Zugriff auf Windows-API-Funktionen

Das .NET Framework unterstützt auch den Zugriff auf die Windows API-Funktionen. Allerdings sollten Sie diese Technik nur sehr sparsam einsetzen, da einerseits fast alles bereits durch die .NET-Framework-Klassenbibliothek abgedeckt wird. Zudem ist die Übergabe der Daten an die Windows-API nicht ganz unkritisch – da es bei fehlerhaften Übergabeparameter im schlimmsten Fall zum Systemabsturz kommt. Außerdem wird unmanaged Code ausgeführt, was zu Leistungseinbußen und dem Verlust der Systemunabhängigkeit führt. Trotzdem möchte ich die Technik für interessierte Leser kurz skizzieren.

13.9.1 Deklaration einer API-Funktion

Die Deklaration einer API-Funktion ist recht einfach. Fügen Sie – am besten in einer eigenen Klasse – die Anweisungen mit der Deklaration der Schnittstelle der API-Funktion ein. Die folgenden Anweisungen deklarieren die API-Funktion zum Aufruf der Windows-internen *MsgBox*-Funktion:

```
Declare Auto Function ExMsgBox Lib "user32.dll" Alias "MessageBox" _
    (ByVal hWnd As Integer, ByVal txt As String, _
    ByVal caption As String, ByVal Typ As Integer) As Integer
```

Die *Declare*-Anweisung leitet die Definition ein. Das Schlüsselwort *Auto* veranlasst die CLR, die Zeichenfolge entsprechend dem Methoden- oder Aliasnamen gemäß den Regeln der Common Language Runtime zu konvertieren. Dann müssen Sie angeben, ob es sich um eine *Function* oder eine *Sub* handelt. Darauf folgt der Name der betreffenden Funktion oder Prozedur. Der Name steht im Quellcode der Anwendung dann zur Verfügung.

Mit dem Schlüsselwort *Lib* wird auf die betreffende Bibliothek hingewiesen, in der die API-Funktion hinterlegt ist. Der Bibliotheksname muss hinter *Lib* folgen. Sie brauchen üblicherweise kein Pfadangaben zu verwenden, falls die DLLs im Windows-Verzeichnis hinterlegt sind. Die meisten API-Funktionen weisen einen Alias-Namen auf, unter dem Sie in der Bibliothek aufzurufen sind.

Abschließend folgt die Parameterliste, die wie bei Visual Basic als *ByVal* deklariert wird und für jeden Parameter den Datentyp aufweisen muss. Leider gibt es das Problem, dass die .NET-Framework-Datentypen nicht immer mit den API-Datentypen übereinstimmen. Sie können dann die Umsetzung mit dem Schlüsselwort *MarshalAs* veranlassen. Die folgende Deklaration zeigt die Anwendung:

```
Declare Function CDTray Lib "winmm.dll" Alias "mciSendStringA" ( _
    <MarshalAs(UnmanagedType.LPStr)> ByVal cmd As String, _
    <MarshalAs(UnmanagedType.LPStr)> ByVal lpstrReturn As String, _
    ByVal Len As Integer, ByVal callback As Integer) As Long
```

Die obige API-Deklaration stellt die Funktion *CDTray* bereit, über die sich die CD-Schublade ansteuern lässt. Zwei Parameter sind über *MarshalAs* ausgezeichnet und werden auf den nicht verwalteten Datentyp *LPStr* umgesetzt.

13.9.2 Anwenden eines API-Aufrufs

Ein kleines Beispiel bietet über ein Dialogfeld die in Abbildung 13.17 gezeigten Schaltflächen, mit denen sich die CD-Schublade ein-/ausfahren lässt. Zudem lassen sich ein Meldungsfeld und ein Ton ausgeben.



Abbildung 13.17: Dialogfeld zum Abrufen der Funktionen

Die nachfolgende Codesequenz zeigt die Ereignisbehandlungsroutinen für die gezeigten Schaltflächen. Der Aufruf der API-Funktionen, die in der Klasse *WinAPI* hinterlegt sind, erfolgt wie der Aufruf anderer Methoden einer Klasse.

```
Private Sub Button_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles ButtonExit.Click, ButtonBeep.Click, _
    ButtonAus.Click, ButtonTrayEin.Click, ButtonMsg.Click
    Const Title As String = "Bitte eine Zahl (0-4) eingeben"
    Const Text As String = "0: OK-Schaltfläche" & vbCrLf & _
        "1: Hand" & vbCrLf & _
        "2: Fragezeichen" & vbCrLf & _
        "3: Ausrufezeichen" & vbCrLf & _
        "4: Stern"
    Dim code() As Int64 = {WinAPI.MB_OK, _
        WinAPI.MB_ICONHAND, _
        WinAPI.MB_ICONQUESTION, _
        WinAPI.MB_ICONEXCLAMATION, _
        WinAPI.MB_ICONASTERISK}

    Dim tmp As String ' Hilfsvariable
    Dim idx As Integer

    If sender Is Me.ButtonExit Then
        Application.Exit() ' beenden
    ElseIf sender Is Me.ButtonBeep Then
        ' Benutzer darf Beep-Code eingeben
        tmp = InputBox(Text, Title, "0")
        If tmp <> "" Then ' Abbrechen abfangen
            If IsNumeric(tmp) Then ' Fehleingaben abfangen
                idx = CInt(tmp) ' in Integer
                If idx >= 0 And idx < 5 Then
                    WinAPI.ExBeep(code(idx)) ' Beep über API-Aufruf ausgeben
                End If
            End If
        End If
    End If
```

```
End If
End If
ElseIf sender Is Me.ButtonAus Then
' CD-Laufwerk ausfahren
WinAPI.CDTray("Set CDAudio Door Open Wait", "", 0, 0)
ElseIf sender Is Me.ButtonTrayEin Then
' Laufwerk einfahren
WinAPI.CDTray("Set CDAudio Door Closed Wait", "", 0, 0)
ElseIf sender Is Me.ButtonMsg Then
' MsgBox per API
WinAPI.ExMsgBox(0, "Es hat geklappt", "Meldungsfeld", _
vbOKOnly Or MsgBoxStyle.Information)
End If
End Sub
```

Zum Ein-/Ausfahren der CD-Schublade sind lediglich die entsprechenden Befehle an die API zu übergeben. Die Methode *ExBeep* ist eine Ergänzung zur Visual Basic-Funktion *Beep*. Mit *Beep* können Sie nur einen Ton ausgeben, während *ExBeep* einen Parameter akzeptiert, der eine, einem bestimmten Windows-Ereignis zugeordnete, Audiodatei abspielt. Bei Anwahl der Schaltfläche wird vom Benutzer in einem *InputBox*-Dialog ein Wert zwischen 0 bis 4 angefordert. Je nach Eingabe wird die Methode mit einer der für die *MessageBeep*-API definierten Konstanten (*MB_OK* bis *WinAPI.MB_ICONASTERISK*) aufgerufen. Die Konstanten sind in der Klasse *WinAPI* definiert. Sofern Sie in der Systemsteuerung über das Symbol *Sounds und Multimedia* die Registerkarte *Sounds* aufgerufen und den Ereignissen getrennte Klangdateien zugewiesen haben, werden in Abhängigkeit von der eingegebenen Zahl verschiedene Sounddateien abgespielt.

Hinweis

Sie finden das Projektbeispiel auf der Begleit-CD im Ordner *\Beisp\Kap13\API-Aufrufe*. Laden Sie das Beispiel in die Entwicklungsumgebung. Die API-Deklaration findet sich in der Datei *WinAPI.vb*, die Aufrufe der Methode sind in *Form1.vb* enthalten.

Damit möchte ich dieses Kapitel schließen. Sie haben eine Reihe von Techniken kennen gelernt, die sich beim Arbeiten mit .NET Framework gewinnbringend einsetzen lassen. Das nächste Kapitel widmet sich Themen rund um das Arbeiten mit Dateien, Verzeichnissen und Laufwerken.

14

Dateioperationen in .NET

In *Kapitel 12* haben Sie im Rahmen des Beispiels »Dateidialoge« bereits eine Einführung in den Umgang mit Dateidialogen erhalten und auch erfahren, wie sich Textdateien lesen und schreiben lassen. Zudem hatte ich in einem weiteren MDI-Beispiel den Zugriff auf RTF-Dokumentdateien angerissen. .NET Framework bietet aber wesentlich umfangreichere Funktionen für Dateioperationen. Auf den folgenden Seiten erhalten Sie eine Einführung in die grundlegenden Methoden zum Zugriff auf Laufwerke, Ordner und Dateien.

14.1 Auflisten aller Laufwerke samt Eigenschaften

Das Beispiel zur Anzeige der Umgebungsinformationen aus *Kapitel 13* demonstriert, wie sich die im System enthaltenen logischen Laufwerke ermitteln lassen. Alternativ könnten Sie die Methode *GetLogicalDrives()* der Klasse *Directory* im Namensraum *System.IO* benutzen, um ein *String*-Array mit den verfügbaren logischen Laufwerksnamen zu ermitteln. Das folgende Beispiel ermittelt die lokal verfügbaren Laufwerke und deren Eigenschaften (Abbildung 14.1) aber auf anderen Wegen.



Abbildung 14.1: Anzeige der logischen Laufwerke samt deren Eigenschaften

Beim Aufruf zeigt das Programm die logischen Laufwerke samt deren Eigenschaften in einem Textfeld an. Über das Menü *Datei/Neu* lässt sich die Anzeige löschen und über *Ansicht/Laufwerke auflisten* wieder aktualisieren. Dies gibt Ihnen ggf. die Möglichkeit, Medien in Wechsellaufwerke einzulegen und abzufragen.

14.1.1 Wie lässt sich der Laufwerkstyp ermitteln?

Die erste Frage besteht darin, wie sich der Typ eines Laufwerks ermitteln lässt. In .NET 2.0 ist dies sehr einfach geworden, denn das *My*-Objekt liefert über *My.Computer.FileSystem.Drives* eine Auflistung aller auf dem lokalen Rechner gefundenen Laufwerke. Die folgende Codesequenz listet alle auf dem lokalen Rechner enthaltenen Laufwerke auf:

```
Dim oDrive As System.IO.DriveInfo
For Each oDrive In My.Computer.FileSystem.Drives
...
Next oDrive
```

Die Variable *oDrive* enthält bei jedem Schleifendurchlauf eines der von der *Drives*-Auflistung zurückgelieferten Laufwerksobjekte. Über dieses Objekt kann auf die jeweiligen Laufwerksinformationen zugegriffen werden. Die Eigenschaft *DriveType* liefert den Laufwerkstyp als Konstante (z.B. *DriveType.CDRom*, *DriveType.Fixed*). Um den Laufwerkstyp im Klartext einzublenden, lässt sich der Eigenschaftswert z.B. in einer *Case*-Struktur auf Übereinstimmung mit den betreffenden Konstanten prüfen:

```
Select Case oDrive.DriveType
    Case System.IO.DriveType.CDRom      ' Laufwerkstyp
                                        ' Typ: CD/DVD
    ...
    Case System.IO.DriveType.Fixed      ' Typ: Festplatte
    ...
    Case System.IO.DriveType.Ram        ' Typ: Ram-Disk
    ...
    Case System.IO.DriveType.Removable  ' Typ: Wechselmedium
    ...
    Case System.IO.DriveType.Network    ' Typ: Netzlaufwerk
    ...
    Case Else                          ' unbekannter Typ
    ...
End Select
```

Listing 14.1: Ermitteln des Laufwerkstyps

14.1.2 Weitere Laufwerkseigenschaften abfragen

Weitere Eigenschaften des im Laufwerk enthaltenen Mediums wie Laufwerksname (Eigenschaft *VolumeName*), das benutzte Dateisystem (Eigenschaft *DriveFormat*) oder die Kapazitäten stehen ebenfalls direkt als Eigenschaftswerte über das betreffende Objekt der *Drives*-Auflistung bereit. Diese Eigenschaften lassen sich aber nur dann abfragen, wenn ein Datenträger im Laufwerk eingelegt ist. Andernfalls löst eine Abfrage der Eigenschaften einen Laufzeitfehler aus. Über die *IsReady*-Eigenschaft des Laufwerks kann das Programm aber prüfen, ob das Medium vorhanden ist. Ein Rückgabewert *True* signalisiert einen eingelegten Datenträger. Im Beispiel wurde der komplette Code zum

Ermitteln der Laufwerke und deren Eigenschaften in eine Prozedur *ShowDriveInfos* gepackt. Der folgende Codeausschnitt zeigt die Prozedur *ShowDriveInfos*:

```
Private Sub ShowDriveInfos()  
    ' Laufwerke auflisten  
    Dim oDrive As System.IO.DriveInfo ' für Laufwerksauflistung  
    Dim frei, ges As Double  
  
    Dim txt As String = "Logische Laufwerke" & vbCrLf & vbCrLf  
  
    ' jetzt die Eigenschaften der Laufwerke ermitteln  
    For Each oDrive In My.Computer.FileSystem.Drives  
        With oDrive  
            txt = txt & "Laufwerk " & .Name & " "  
            Select Case .DriveType ' Laufwerkstyp  
                Case System.IO.DriveType.CDRom  
                    txt = txt & "Typ: CD/DVD" & vbCrLf  
                Case System.IO.DriveType.Fixed  
                    txt = txt & "Typ: Festplatte" & vbCrLf  
                Case System.IO.DriveType.Ram  
                    txt = txt & "Typ: Ram-Disk" & vbCrLf  
                Case System.IO.DriveType.Removable  
                    txt = txt & "Typ: Wechselmedium" & vbCrLf  
                Case System.IO.DriveType.Network  
                    txt = txt & "Typ: Netzlaufwerk" & vbCrLf  
                Case Else  
                    txt = txt & "Typ: unbekannt" & vbCrLf  
            End Select  
            ' jetzt weitere Eigenschaften ermitteln  
            If .IsReady Then ' Medium eingelegt  
                txt = txt & "Name: " & .VolumeLabel & vbCrLf  
                txt = txt & "Dateisystem: " & .DriveFormat & vbCrLf  
                txt = txt & "Kapazität: " & FormGB(CType(.TotalSize, Long)) _  
                    & vbCrLf  
                ges = CType(.TotalSize, Long) ' Gesamtkapazität  
                frei = CType(.AvailableFreeSpace, Long) ' freie Kapazität  
                txt = txt & "Belegt: " & FormGB(ges - frei) & vbCrLf  
                txt = txt & "Frei: " & FormGB(CType(frei, Long)) & vbCrLf  
            Else  
                txt = txt & "Laufwerk nicht bereit" & vbCrLf  
            End If  
        End With  
    End For  
End Sub
```

Listing 14.2: Ermitteln der Laufwerke und deren Eigenschaften


```

End With
txt = txt & vbCrLf & vbCrLf ' Leerzeile einfügen
Next oDrive
Me.TextBox1.Text = txt      ' In Textfeld
End Sub

```

Listing 14.2: Ermitteln der Laufwerke und deren Eigenschaften (Forts.)

Die Prozedur liest die *Drives*-Auflistung und ermittelt dann den Laufwerkstyp. Dann wird geprüft, ob das Laufwerk zum Lesen weiterer Eigenschaften bereit ist. Trifft dies zu, liest die Prozedur nacheinander verschiedene Eigenschaften. Die Kapazitätsangaben werden über eine Hilfsfunktion *FormGB* so aufbereitet, dass Angaben in Megabyte oder Gigabyte angezeigt werden.

```

Private Function FormGB(ByVal wert As Double) As String
    ' wandele den Wert in Megabyte oder Gigabyte um
    Const kb As Double = 1024.0
    Dim tmp As Double
    tmp = wert / kb / kb / kb
    If tmp < 1.0 Then ' kleiner 1GB in Megabyte
        Return (tmp * kb).ToString("#0.00 (MB)")
    Else
        Return tmp.ToString("0.00 (GB)")
    End If
End Function

```

Die Funktion erwartet einen Wert vom Typ *Double* und liefert einen fertig formatierten Text zurück. Die *ShowDriveInfos*-Prozedur wird im Beispielprogramm sowohl in der *Load*-Ereignisprozedur als auch beim *Click*-Ereignis des Menübefehls *Ansicht/Laufwerke auflisten* aufgerufen. Weitere Einzelheiten sind dem Quellcode des Beispiels zu entnehmen.

Hinweis

Sie finden das komplette Projekt auf der Begleit-CD im Ordner *\Beisp\Kap14\Drive-Info*. Laden Sie *.sln*-Datei in die Microsoft-Entwicklungsumgebung und übersetzen Sie das Projekt zum Testen.

14.2 Zugriff auf Dateien und Ordner

Eine .NET-Anwendung kann über ein Laufwerk auf die enthaltenen Dateien und Ordner zugreifen. Dabei ist aber sicherzustellen, dass das betreffende Laufwerk bereit ist (dies kann über die oben beschriebene *IsReady*-Eigenschaft geschehen). Haben Sie einen existierenden Ordner ausgewählt, können Sie die darin enthaltenen Dateien und Unterordner auflisten. Nachfolgend lernen Sie die betreffenden Techniken zum Umgang mit dem Inhalt eines Ordners kennen.

14.2.1 Unterordner und Dateien auflisten

In einem einfachen Beispiel soll der Zugriff auf einen Ordner demonstriert werden. Über einen *BrowseForFolder*-Dialog des Menüs *Datei/Öffnen* kann der Benutzer einen Ordner des Dateisystems auswählen. Nach dem Schließen des Dialogfelds lassen sich über die Befehle des Menüs *Ansicht* die im Ordner enthaltenen Unterordner und Dateien in einem Textfeld anzeigen (Abbildung 14.2).

Im *Click*-Ereignis des Menüs *Datei/Öffnen* wird der Dialog zur Auswahl eines Ordners hinterlegt. Durch die Auswahl ist sichergestellt, dass der angegebene Ordner auch tatsächlich existiert. Zur Auswahl des Ordners kommt das bereits in Kapitel 12 vorgestellte Steuerelement *FolderBrowserDialog* zum Einsatz. Der vom Benutzer ausgewählte Ordner wird in der globalen Variable *folder* hinterlegt. Zum Zugriff auf den Inhalt eines Ordners bietet das .NET Framework die Klasse *Directory* mit verschiedenen Methoden:

- *GetFileSystemEntries*: Diese Methode liefert alle Unterordner und Dateien des als *String*-Parameter übergebenen Verzeichnisses zurück. Sie müssten dann die Ordner aus dem *String*-Array herausfiltern.
- *GetDirectories*: Über diese Methode lässt sich eine Auflistung aller Unterordner in einem als *String*-Parameter übergebenden Ordner abrufen. Als Ergebnis wird ein *String*-Array zurückgegeben, welches alle Ordnerpfade enthält.

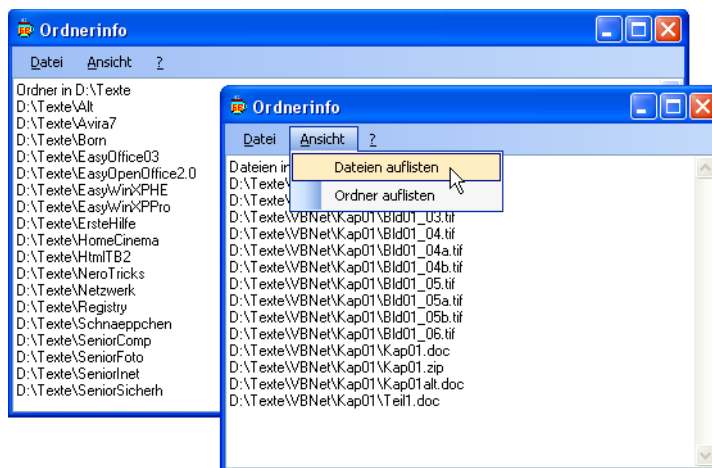


Abbildung 14.2: Auflisten von Ordnern und Dateien

Die folgende Prozedur wird vom *Click*-Ereignis des Menüs *Ansicht/Ordner auflisten* aufgerufen und schreibt alle Ordnernamen in ein Textfeld:

```
Private Sub ShowFolderInfos()  
    ' Ordner auflisten  
    Dim oFolder As String          ' Ordnerobjekt  
    Dim txt As String = "Ordner in " & folder & vbCrLf
```

```

For Each oFolder In GetDirectories(folder) ' hole Ordnerauflistung
    txt = txt & oFolder & vbCrLf          ' Name einfügen
Next oFolder
Me.TextBox1.Text = txt                  ' in Textfeld
End Sub

```

Die Variable *oFolder* wird vom Typ *String* deklariert. In einer *For Each*-Schleife werden die einzelnen Einträge des von *GetDirectories()* zurückgegebenen String-Array bearbeitet. Die Schleifenvariable *oFolder* enthält dann direkt den Namen des Ordners samt Pfad.

Hinweis

Über die Methode *GetParent()* können Sie das übergeordnete Verzeichnis ermitteln. Dies bietet die Möglichkeit, den Ordernamen zu extrahieren, indem Sie die Differenz der beiden Zeichenketten bestimmen. Die betreffende Technik wird auf den folgenden Seiten im Beispiel zur Anzeige der Attribute genutzt.

Möchten Sie anstelle der Ordner die im Verzeichnis enthaltenen Dateien anzeigen? Dann verwenden Sie statt der *GetDirectories()*-Methode die *GetFiles()*-Methode. Diese liefert ein *String*-Array mit den Namen der im Ordner gefundenen Dateien zurück. Die nachfolgende Prozedur *ShowFileInfos* verwendet eine *For Each*-Schleife, um die Dateien aufzulisten. Anschließend wird die Dateiliste in ein Textfeld geschrieben.

```

Private Sub ShowFileInfos()
    ' Dateien auflisten
    Dim oFile As String          ' Dateiobjekt
    Dim txt As String = "Dateien in " & folder & vbCrLf

    For Each oFile In GetFiles(folder) ' hole Dateiauflistung
        txt = txt & oFile & vbCrLf      ' Name einfügen
    Next oFile
    Me.TextBox1.Text = txt            ' in Textfeld
End Sub

```

Hinweis

Sie finden das als Windows-Anwendung realisierte Projekt im Ordner *\Beisp\Kap14\FolderInfo* auf der Begleit-CD. Sobald Sie das Projekt über die *.sln*-Datei laden und übersetzen, sollte sich die Anwendung ausführen lassen. Weitere Details entnehmen Sie bitte dem betreffenden Quellcode.

14.2.2 Zugriff auf Ordner- und Dateieigenschaften

Über die Objekte des Dateisystems können Sie auch auf die Eigenschaften (Attribute, Datumswerte, Größe etc.) von Dateien und Ordnern zugreifen. Zur Demonstration wird das vorherige Beispiel etwas weiter entwickelt werden.

Der oben benutzte Ansatz mit *GetDirectories(path)* bzw. *GetFiles(path)* liefert leider ein *String*-Array zurück, d.h., Sie müssten erst ein Dateiobjekt erstellen, um an dessen Eigenschaften heranzukommen. Daher benutze ich jetzt einen anderen Ansatz, indem Methoden wie *GetDirectories()* auf ein Verzeichnis angewandt werden und die Auflistung an eine Variable vom Typ *DirectoryInfo* zurückgeliefert wird. Dann lässt sich über die betreffende Objektvariable direkt auf die Eigenschaften zugreifen. Die im vorherigen Beispiel benutzte Prozedur *ShowFolderInfo* ist daher folgendermaßen zu modifizieren:

```
Private Sub ShowFolderInfos()
' Ordner auflisten
Dim oFolder As New DirectoryInfo(folder)
Dim oFolders As DirectoryInfo() = _
    oFolder.GetDirectories() ' Verzeichnisse
Dim oItem As DirectoryInfo
Dim txt As String = "Ordner in " & folder & vbCrLf
txt = txt & "Name" & vbTab & vbTab & "Datum angelegt" & vbCrLf

For Each oItem In oFolders ' hole Ordnerauflistung
    txt = txt & oItem.Name & vbTab ' Name einfügen
    txt = txt & oItem.CreationTime & vbCrLf ' Datum
Next oItem
Me.TextBox1.Text = txt ' in Textfeld
End Sub
```

Die Objektvariable *oFolder* nimmt nun die Verzeichnisinformation auf, die über *DirectoryInfo(folder)* geliefert wird. Über die Objektvariable lässt sich dann auf die *GetDirectories()*-Methode zugreifen. Diese Methode liefert jetzt alle Verzeichnisse im angegebenen Ordner zurück. Die Informationen sind nun aber Objekte und keine einfachen Zeichenketten. In einer Schleife lässt sich jedes Element der Auflistung bearbeiten, über den Objektnamen des Schleifenindex kann auf die Objekteigenschaften des Ordners zugegriffen werden. Die Eigenschaft *Name* liefert den Ordnernamen und *CreationTime*-Eigenschaft das Datum, zu dem das Verzeichnis angelegt wurde.

Um die Eigenschaften von Dateien in einem Ordner abzufragen, lässt sich ein ähnlicher Ansatz wählen. Sie müssen die jeweilige Datei als Objekt definieren, um dann auf deren Eigenschaften (*Name* = Dateiname, *Length* = Dateigröße, *LastWriteTime* = letzte Änderung etc.) zugreifen zu können. Die betreffende Prozedur enthält die folgenden Anweisungen:

```
Private Sub ShowFileInfos()
' Dateien auflisten - hole Auflistung
Dim oFolder As New DirectoryInfo(folder)
```

```

Dim oFiles As FileInfo() = oFolder.GetFiles() ' Dateiobjekt
Dim oFile As FileInfo

Dim txt As String = "Dateien in " & folder & vbCrLf
txt = txt & "Name" & vbTab & "Größe (Bytes)" & vbTab & _
    "Datum geändert" & vbTab & "Hauptattribut" & vbCrLf

For Each oFile In oFiles ' hole Dateiaufstellung
    txt = txt & oFile.Name & vbTab ' Name einfügen
    txt = txt & oFile.Length & vbTab ' Größe einfügen
    txt = txt & oFile.LastWriteTime & vbTab ' Änderungsdatum
    txt = txt & "Attribute: " & Hex(oFile.Attributes) & vbTab
    txt = txt & vbCrLf
Next oFile
Me.TextBox1.Text = txt ' in Textfeld
End Sub

```

Die in *oFile.Attribute* zurückgelieferten Attribute sind als Bitfeld codiert und werden über die Visual-Basic-Funktion *Hex* in einen Hexadezimalwert konvertiert. Durch Maskieren mit den Attributkonstanten des .NET Framework lassen sich die Attribute decodieren (siehe auch den Abschnitt »Ein Anwendungsbeispiel zur Attributanzeige« weiter unten).

Hinweis

Sie finden das modifizierte Projektbeispiel im Ordner `\Beisp\Kap14\FolderInfoExt` auf der Begleit-CD. Sobald Sie das Projekt über die *.sln*-Datei laden und übersetzen, sollte sich die Anwendung ausführen lassen. Weitere Details entnehmen Sie bitte dem betreffenden Quellcode.

14.2.3 Ist der Datenträger/Ordner leer?

Die Prüfung, ob ein Datenträger oder ein Ordner leer ist, erfordert einige Kniffe. Visual Basic bietet die *Dir*-Funktion, die den Inhalt des angegebenen Pfads als Verzeichnis auflistet. Die folgenden Anweisungen zeigen den Einsatz dieser Funktion:

```

Status = Dir (A:\*.*")
Status1 = Dir (A:\*.*",16)

```

Die oberste Zeile prüft das angegebene Laufwerk auf beliebige Dateien. In der zweiten Zeile habe ich im zweiten Parameter den Wert 16 (entspricht der Konstanten *FileAttribute.Directory*) übergeben. Dann wird nur nach Verzeichnissen gesucht.

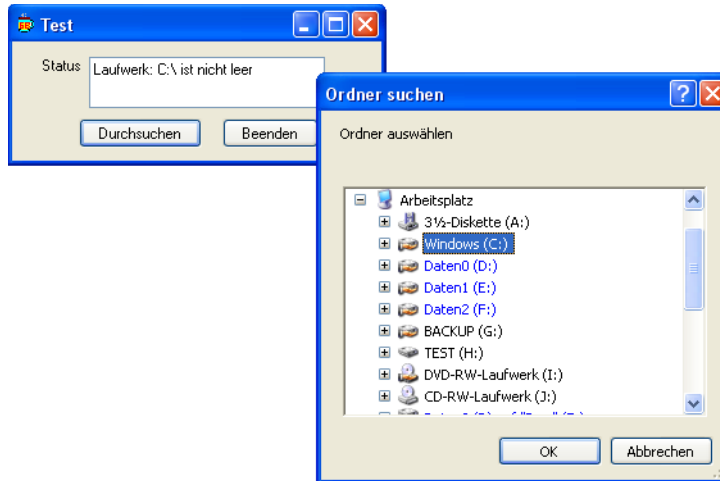


Abbildung 14.3: Überprüfen von Laufwerken und Ordnern

Diese Prüfung funktioniert sowohl bei Laufwerken als auch bei Ordnern. Das in Abbildung 14.3 gezeigte Beispiel stellt über ein Dialogfeld eine Schaltfläche *Durchsuchen* bereit. Über diese Schaltfläche lässt sich der rechts unten gezeigten *FolderBrowser*-Dialog zur Auswahl eines Laufwerks oder eines Ordners öffnen. Das Beispiel prüft dann, ob das ausgewählte Objekt leer ist und meldet den Status im Textfeld des Formulars. Die Prüfung, ob das Objekt leer ist, erfolgt mit der Funktion *IsEmpty*, die nachfolgende Anweisungen enthält:

```
Private Function IsEmpty(ByVal item As String) As Boolean
' Prüfe, ob das Objekt leer ist
Try
    If Dir(item, FileAttribute.Directory) <> "" Then
        Return False ' Ordner vorhanden
    Exit Function
    Else
        If Dir(item & ".*") <> "" Then
            Return False ' Dateien vorhanden
        Exit Function
    End If
    Return True ' nichts gefunden
End If
Catch ex As Exception
    MessageBox.Show("Laufwerk nicht bereit", "Fehler", _
        MessageBoxButtons.OK, MessageBoxIcon.Error)
    Return False
End Try
End Function
```

Die Funktion prüft erst, ob Unterverzeichnisse im angegebenen Objekt existieren. Danach wird auf Dateien getestet. Wird ein Eintrag gefunden, liefert die Funktion *False* zurück, bei einem leeren Objekt wird *True* zurückgegeben.

Hinweis

Bei Bedarf können Sie die Funktion ja so anpassen, dass diese bei einem Laufwerk ohne Medium einen Wert -1, bei leerem Laufwerk den Wert 0 und bei Laufwerken mit Daten den Wert 1 zurückgibt. Dann lässt sich auch der im aktuellen Beispiel enthaltene Fall, dass bei fehlendem Medium der Hinweis auf einen leeren Datenträger eingeblendet wird, abfangen.

Der *Click*-Ereignishandler der Schaltfläche *Durchsuchen* enthält den Code zum Aufruf des *FolderBrowserDialog*-Dialogs sowie die Anweisungen zur Prüfung, ob das gewählte Objekt leer ist.

```
Private Sub ButtonDrive_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ButtonDrive.Click
    ' Ordner auswählen mit FolderBrowserDialog
    Dim folder As String = ""
    folder = BrowseFolder() ' hole Laufwerksbuchstabe oder Ordner
    If folder <> "" Then ' Ist der Pfad leer?
        If folder.EndsWith(":") Then ' ist es ein Laufwerk?
            If IsEmpty(folder) Then
                Me.TextBox1.Text = "Laufwerk: " & folder & " ist leer"
            Else
                Me.TextBox1.Text = "Laufwerk: " & folder & " ist nicht leer"
            End If
        Else ' ein ist ein Ordner
            If IsEmpty(folder) Then
                Me.TextBox1.Text = "Ordner: " & folder & vbCrLf & " ist leer"
            Else
                Me.TextBox1.Text = "Ordner: " & folder & vbCrLf & _
                    & " ist nicht leer"
            End If
        End If
    End If
End Sub
```

Anhand der Endung des Texts im gewählten Element entscheidet die Prozedur, ob es sich um ein Laufwerk oder einen Ordner handelt. Diese Unterscheidung ist zwar für die Funktion *IsEmpty* nicht erforderlich. Ich verwende den Test aber, um den Ausgabertext an das ausgewählte Objekt anzupassen.

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap14\EmptyDrive` der Begleit-CD. Sobald Sie das Projekt über die `.sln`-Datei laden und übersetzen, sollte sich die Anwendung ausführen lassen. Der Programmcode zum Aufruf des `FolderBrowserDialog`-Dialogs wurde in die Funktion `BrowseFolder` ausgelagert. Innerhalb dieser Funktion wird das vom Benutzer ausgewählte Laufwerk bzw. das Verzeichnis ermittelt und als Wert zurückgegeben. Bei Ordnern schließt die Funktion den Pfad zudem mit dem Zeichen »\« ab – andernfalls klappt der Aufruf der `IsEmpty`-Funktion bei Ordnern nicht. Weitere Details entnehmen Sie dem betreffenden Quellcode.

14.3 Datum und Dateiattribute lesen/setzen

Ordner und Dateien besitzen einmal Datumswerte (Datum der Erstellung, der letzten Änderung, des letzten Zugriffs). Zudem werden diesen Elementen auch Attribute (Versteckt, System etc.) zugeordnet. Das .NET Framework bietet verschiedene Möglichkeiten, auf die Datumswerte und die Attribute von Ordnern und Dateien zuzugreifen. Dies ermöglicht Ihnen, diese Werte sowohl zu lesen als auch zu setzen.

14.3.1 Ein Anwendungsbeispiel zur Attributanzeige

Das Ganze möchte ich jetzt an einem einfachen Beispiel demonstrieren. Eine Anwendung ermittelt beim Starten den eigenen Namen, den Startordner der `exe`-Datei sowie das übergeordnete Verzeichnis. Zudem soll diese Anwendung noch eine Testdatei `Test.txt` im Startordner anlegen. Dann werden die Datumswerte und Attribute dieser Elemente (Anwendung, Testdatei und Ordner) in einem Formular in einem Textfeld aufgelistet (Abbildung 14.4, links oben). Klickt der Benutzer auf die Schaltfläche *Attribute setzen*, ändert die Anwendung das Attribut der Testdatei `Test.txt` auf `ReadOnly` (Abbildung 14.4, rechts unten). Zudem werden die Datumswerte (`DateCreated` etc.) auf den 1.1.1999 zurückgesetzt. Über die Schaltfläche *Attribute anzeigen* lässt sich die Anzeige aktualisieren. Die Schritte zur Realisierung dieses Anwendungsbeispiels werden in den nachfolgenden Abschnitten besprochen.

Die Textdatei bei Bedarf anlegen

Beim Starten der Anwendung wird die `Load`-Ereignisbehandlungsroutine des Formulars benutzt, um die Testumgebung bereitzustellen. Bei diesem Schritt soll die Datei `Test.txt` im Ordner, aus dem die Anwendung startet, bereitgestellt werden. Existiert diese Datei bereits, muss diese neu angelegt werden, um deren Datumswerte (Erzeugt, letzter Zugriff, letzte Änderung) auf neue Werte zu setzen.

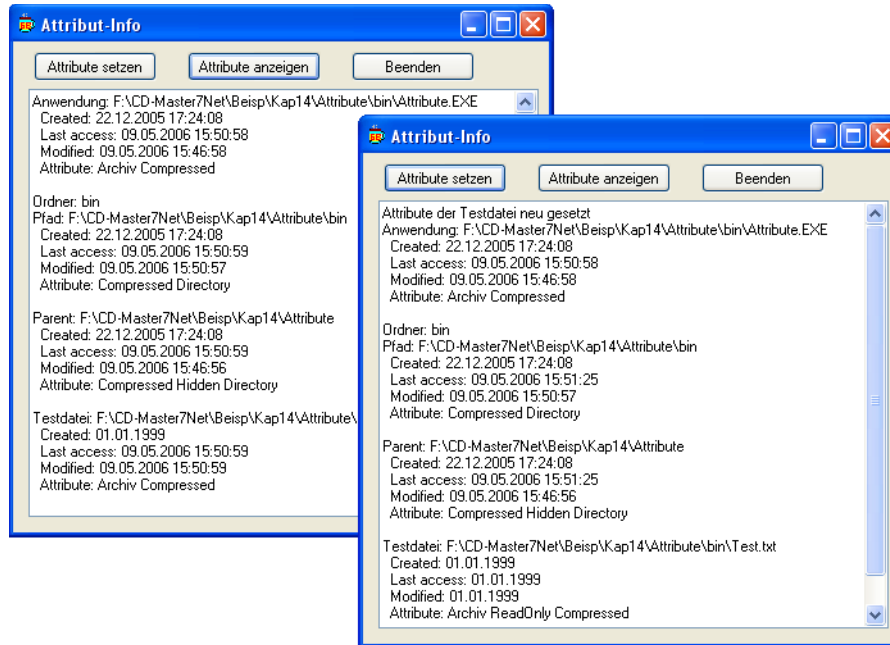


Abbildung 14.4: Anzeigen und Setzen von Attribut- und Datumswerten

Die Prüfung, ob eine Datei existiert, lässt sich über die *Exists()*-Methode vornehmen. Die Methode wird sowohl von der Klasse *Directory* (für Ordner) als auch von der Klasse *Files* (für Dateien) bereitgestellt. Die Anweisung

```
IO.File.Exists(file)
```

liefert den Wert *False*, falls es die in *file* angegebene Datei nicht gibt. Die nachfolgend gezeigte *Load*-Ereignisbehandlungsroutine des Formulars wird bei jedem Programmstart durchlaufen und prüft, ob die Datei vorhanden ist. Ist die Datei vorhanden, setzt der Aufruf der *SetAttr*-Funktion das Schreibschutzattribut zur Vorsicht zurück. Danach wird einfach eine Datei mit neuem Dateidatum erzeugt. Die Befehle zum Anlegen einer Textdatei wurden bereits in früheren Kapiteln diskutiert.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    ' erzeuge Hilfsdatei und zeige Eigenschaften
    ' der Anwendung, der Hilfsdatei und der Ordner
    Dim oStream As IO.StreamWriter ' Lege eine neue Datei an
    Dim oFile As IO.FileStream
    Dim file As String = GetPath() & "\" & TestFile ' Testdatei

    Try ' Versuch, auf CD-ROM zu schreiben, abfangen
        If IO.File.Exists(file) Then
```

```
SetAttr(file, FileAttribute.Normal) ' Attribute zurücksetzen
End If
' Testdatei mit neuen Datumswerten anlegen
oFile = New IO.FileStream(file, IO.FileMode.Create)
oStream = New IO.StreamWriter(oFile) ' Stream
oStream.WriteLine("Ein Testtext") ' schreiben
oStream.Close() ' schließen
oFile.Close()
Catch ex As Exception
    MessageBox.Show(ex.ToString, "Fehler", _
        MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try

Me.TextBox1.Text = ShowInfos() ' Infos holen
Me.TextBox1.SelectionStart = 0 ' Markierung löschen
Me.TextBox1.SelectionLength = 0 ' "
End Sub
```

Neu sind die beiden letzten Anweisungszeilen der Prozedur, die die Markierung des Inhalts des Textfelds über die Eigenschaften *SelectionStart* und *SelectionLength* aufheben.

Decodierung der Attributbits

Die Attribute einer Datei lassen sich über die *GetAttributes()*-Methode abfragen:

```
attribute = System.IO.File.GetAttributes(file)
```

Die Prozedur liefert einen Wert vom Typ *FileAttributes* zurück. Dieser enthält die einzelnen Attributwerte als Bitfeld. Zum Dekodieren dieses Bitfelds habe ich die Funktion *ShowAttributes* erstellt. Diese erwartet den Attributwert vom Typ *Integer* und liefert die decodierten Attribute als Zeichenkette zurück. Daher muss der betreffende Wert mit *CInt* vor dem Aufruf der Funktion in den Typ *Integer* konvertiert werden. Die einzelnen Dateiattribute lassen sich durch Maskierung mit den Bits für die jeweiligen Attribute herausfinden. Für das Schreibschutzattribut sieht die Anweisung folgendermaßen aus:

```
If (attrib And IO.FileAttributes.ReadOnly) > 0 Then
    ' Schreibschutzattribut ist gesetzt
End If
```

Die betreffenden Attributkonstanten sind in *System.IO.FileAttributes* vereinbart. Nachfolgend sehen Sie die Funktion, die einige Attribute auswertet und bei gesetzten Attributbits einen Text der Art »Archiv«, »ReadOnly« etc. zusammenbaut und anschließend zurückliefert.

```

Private Function ShowAttributes(ByVal attrib As Integer) As String
' Decodiere den als Integer übergebenen Attributwert
Dim txt As String = ""      ' Ergebnisstring

' Jetzt die Attribute ermitteln und Zeichenkette zusammenbauen
If (attrib And IO.FileAttributes.Archive) > 0 Then _
    txt = txt & "Archiv "
If (attrib And IO.FileAttributes.ReadOnly) > 0 Then _
    txt = txt & "ReadOnly "
If (attrib And IO.FileAttributes.Compressed) > 0 Then _
    txt = txt & "Compressed "
If (attrib And IO.FileAttributes.Encrypted) > 0 Then _
    txt = txt & "Encrypted "
If (attrib And IO.FileAttributes.Hidden) > 0 Then _
    txt = txt & "Hidden "
If (attrib And IO.FileAttributes.System) > 0 Then _
    txt = txt & "System "
If (attrib And IO.FileAttributes.Directory) > 0 Then _
    txt = txt & "Directory "
Return txt
End Function

```

Anzeige der Dateieigenschaften

Zur Anzeige der jeweiligen Dateieigenschaften benutzt das Beispiel die Funktion *Show-Infos*, die eine Zeichenkette mit den im Textfeld angezeigten Informationen liefert. Im Prozedurkopf werden die benötigten Variablen deklariert. Bei einigen dieser Variablen wird gleich ein Wert zugewiesen. Die Eigenschaft *Application.ExecutablePath* liefert den Namen der Anwendungsdatei, die hier der Variablen *file* zugewiesen wird. Der aktuelle Pfad der Anwendung wird über die benutzerspezifische *GetPath()*-Funktion ermittelt. In der Implementierung des Beispiels habe ich die Anweisungsfolge

```
path = System.IO.Path.GetDirectoryName(Application.ExecutablePath)
```

benutzt, da die betreffenden Namespaces und Bibliotheken im Projekt in Benutzung sind. Das übergeordnete Verzeichnis eines Ordners lässt sich über die *GetParent()*-Methode der Klasse *Directory* ermitteln. Anschließend benutzt die Prozedur verschiedene Methoden wie *GetCreationTime(file)*, um die Datumsangaben der Datei abzufragen. Die Methode *GetAttributes (file)* liefert die Dateiattribute. Diese werden hier in den Typ *Integer* konvertiert und zur Decodierung an die Funktion *ShowAttributes* übergeben. Weitere Details können Sie dem folgenden Listing entnehmen, welches die verschiedenen Ansätze zeigt, um Datums- und Attributeigenschaften von Dateien und Ordnern zu ermitteln. Immer wenn ein Objekt vorliegt, wird direkt auf die Eigenschaften zugegriffen. Liegt die Information zu einem Element als Zeichenkette vor, wird eine Methode zum Abrufen der Eigenschaften der jeweiligen Klasse benutzt:

```
Private Function ShowInfos() As String
' Zeige die Informationen des Ordners, des übergeordneten Ordners
' und der Anwendungsdatei im Textfeld
Dim txt As String = ""
Dim file As String = Application.ExecutablePath ' Anwendung
Dim path As String = GetPath() ' aktueller Pfad
Dim test As String = path & "\" & TestFile ' Testdatei
Dim parent As IO.DirectoryInfo = GetParent(path) ' Parent-Verz.

' erst die Dateieigenschaften auswerten
txt = "Anwendung: " & file & vbCrLf
txt = txt & " Created: " & GetCreationTime(file) & vbCrLf
txt = txt & " Last access: " & GetLastAccessTime(file) & vbCrLf
txt = txt & " Modified: " & GetLastWriteTime(file) & vbCrLf
txt = txt & " Attribute: " & ShowAttributes _
    (CInt(System.IO.File.GetAttributes(file))) ' Attribute holen
txt = txt & vbCrLf & vbCrLf

' jetzt die Eigenschaften des aktuellen Ordners in "path"
Dim ordner As String
txt = txt & "Ordner: " & _
    path.Substring(parent.FullName.Length + 1) & vbCrLf
txt = txt & "Pfad: " & path & vbCrLf
txt = txt & " Created: " & GetCreationTime(path) & vbCrLf
txt = txt & " Last access: " & GetLastAccessTime(path) & vbCrLf
txt = txt & " Modified: " & GetLastWriteTime(path) & vbCrLf
txt = txt & " Attribute: " & ShowAttributes _
    (CInt(System.IO.File.GetAttributes(path))) ' Attribute holen
txt = txt & vbCrLf & vbCrLf

' jetzt das übergeordnete Verzeichnis bearbeiten
txt = txt & "Parent: " & parent.FullName & vbCrLf
txt = txt & " Created: " & parent.CreationTime & vbCrLf
txt = txt & " Last access: " & parent.LastAccessTime & vbCrLf
txt = txt & " Modified: " & parent.LastWriteTime & vbCrLf
txt = txt & " Attribute: " & ShowAttributes _
    (CInt(parent.Attributes))
txt = txt & vbCrLf & vbCrLf

' jetzt die Eigenschaften der Testdatei ermitteln
```

Listing 14.3: Anzeige der Dateiinformationen

```

txt = txt & "Testdatei: " & test & vbCrLf
txt = txt & "  Created: " & GetCreationTime(test) & vbCrLf
txt = txt & "  Last access: " & GetLastAccessTime(test) & vbCrLf
txt = txt & "  Modified: " & GetLastWriteTime(test) & vbCrLf
txt = txt & "  Attribute: " & ShowAttributes _
    (CInt(System.IO.File.GetAttributes(test)))
txt = txt & vbCrLf & vbCrLf
Return txt
End Function

```

Listing 14.3: Anzeige der Dateiinformatoren (Forts.)

Datums- und Attributwerte setzen

Das Ändern des Datumswerts und des Attributs erfolgt in der Prozedur des *Click*-Ereignisses der Schaltfläche *Attribute setzen*. Zum Anpassen der Datumswerte kommen die Methoden *SetCreationTime()*, *SetLastAccessTime()* und *SetLastWriteTime()* zum Einsatz. Diese erwarten im ersten Parameter den Dateinamen und im zweiten Parameter den Datumswert. Zum Ändern eines Attributwerts wird die Visual-Basic-*SetAttr*-Funktion benutzt, die im ersten Argument den Dateinamen und im zweiten Argument den Attributwert erwartet. Das Bitmuster für das Attribut wird durch OR-Verknüpfung der einzelnen Attributkonstanten gebildet. Weitere Details sind nachfolgendem Codeauszug zu entnehmen. Hintergrundinformationen zu den einzelnen Methoden und Funktionen liefert die Hilfe zum .NET Framework.

```

Private Sub ButtonSet_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ButtonSet.Click
' Setze Attribute und Eigenschaften, zeige neu an
Dim file As String = GetPath() & "\" & TestFile ' Testdatei
Dim datum As Date = #1/1/1999#

SetAttr(file, FileAttribute.ReadOnly) ' Schreibschutz aufheben!!!

SetCreationTime(file, datum)          ' Dateidatum
SetLastAccessTime(file, datum)
SetLastWriteTime(file, datum)

SetAttr(file, FileAttribute.ReadOnly Or FileAttribute.Archive)

Me.TextBox1.Text = "Attribute der Testdatei neu gesetzt" & _
    vbCrLf & ShowInfos() ' Infos holen
Me.TextBox1.SelectionStart = 0 ' Markierung löschen
Me.TextBox1.SelectionLength = 0 ' "
End Sub

```

Listing 14.4: Setzen der Dateiattribute

Hinweis

Sie finden das komplette Projektbeispiel im Ordner `\Beisp\Kap14\Attribute` auf der Begleit-CD. Sobald Sie das Projekt über die `.sln`-Datei laden und übersetzen, sollte sich die Anwendung ausführen lassen. Da die Anwendung eine Testdatei anlegt und deren Eigenschaften ändert, lässt sich das Beispiel aber nicht von CD-ROM ausführen.

14.3.2 SetFileDate-Anwendung

Gelegentlich kommt es vor, dass die Datumseigenschaften und Attribute von Dateien und/oder Ordnern auf einen bestimmten Wert zu setzen sind. Während Attribute über den Eigenschaftendialog des gewählten Elements anpassbar sind, bietet Windows keine Funktion zum Ändern der Datumswerte. Mit der kleinen Anwendung *SetFileDate* lässt sich dieses Problem beheben.

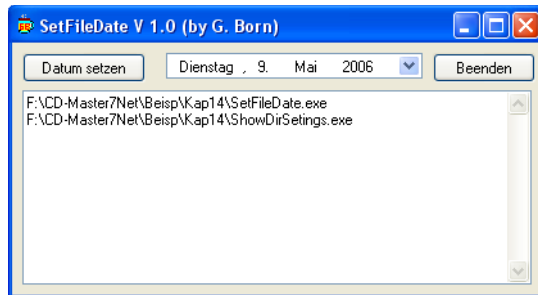


Abbildung 14.5: *SetFileDate*-Anwendung

Hinweis

Zum Starten müssen Sie die gewünschten Dateien und/oder Ordner per Drag&Drop zum Symbol der Programmdatei ziehen. Das Programm übernimmt beim Start dann die übergebenen Dateinamen. Auf die Implementierung eines Dateidialogs habe ich in diesem Beispiel verzichtet.

Die Anwendung zeigt nach dem Start alle gewählten Elemente in einem Textfeld (Abbildung 14.5). Über den *DateTime*-Picker können Sie komfortabel ein neues Datum wählen. Sobald Sie auf die Schaltfläche *Datum setzen* klicken, werden die Datumseigenschaften der gewählten Dateien und Ordner umgesetzt. Das Programm fasst einige der in den vorhergehenden Beispielen vermittelten Techniken zusammen.

Die Prozedur zur Behandlung des *Load*-Ereignisses fragt über *GetCommandLineArgs* (siehe Kapitel 13) die beim Aufruf übergebenen Argumente ab. Anschließend wird die Auflistung an Ordnern und Dateien in der globalen Feldvariablen *files* hinterlegt. Die Variable *count* enthält die Zahl der Einträge. Die Schleife schreibt die Namen der Elemente in das Textfeld des Formulars.

```

Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    ' zeige Dateien und Ordner
    Dim tmp As String = ""
    Dim i As Integer

    files = GetCommandLineArgs()      ' Dateien aus Befehlszeile
    count = files.GetUpperBound(0)    ' Dateien zählen

    If count > 0 Then
        For i = 1 To count
            tmp = tmp & files(i) & vbCrLf
        Next i
        Me.TextBox1.Text = tmp          ' Dateien anzeigen
    Else
        Me.TextBox1.Text = _
            "Setzt die Datumswerte selektierter Dateien" & vbCrLf & _
            "Es wurden keine Dateien übergeben" & vbCrLf & _
            "Bitte Dateien per Drag&Drop zur Anwendung ziehen"
    End If
    Me.TextBox1.SelectionStart = 0      ' Markierung löschen
    Me.TextBox1.SelectionLength = 0     ' "
End Sub

```

Das Umsetzen der Datumswerte für alle Einträge erfolgt in der *Click*-Ereignisbehandlungsprozedur der Schaltfläche *Datum setzen*. Die Prozedur liest den Inhalt des *DateTimePicker*-Steuerelements und benutzt eine *For*-Schleife, um die in *files* hinterlegten Dateien und Ordner zu durchlaufen. Dabei setzt das Programm die Datumseinträge über die betreffenden, bereits oben benutzten Methoden.

```

Private Sub ButtonSet_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ButtonSet.Click
    ' Setze Attribute und Eigenschaften, zeige neu an
    ' anpassen der Dateieigenschaften der Anwendung
    Dim datum As Date = CDate(Me.DateTimePicker1.Text) ' Datum
    Dim i As Integer
    If count > 0 Then
        For i = 1 To count
            Try
                SetCreationTime(files(i), datum)      ' Dateidatum
                SetLastAccessTime(files(i), datum)
                SetLastWriteTime(files(i), datum)
                Me.TextBox1.Text = Me.TextBox1.Text & _

```

```

        "Datei " & files(i) & " geändert" & vbCrLf

Catch ex As Exception
    Me.TextBox1.Text = Me.TextBox1.Text & _
        "*** Warnung: Der Prozess kann nicht auf die Datei " _
        & vbCrLf & files(i) & vbCrLf & _
        "zugreifen, da sie verwendet wird." & vbCrLf
End Try
Next i

Me.TextBox1.Text = Me.TextBox1.Text & _
    vbCrLf & "Datumswerte neu gesetzt" & vbCrLf ' Status anzeigen
Else
    Me.TextBox1.Text = "Keine Dateien gewählt"
End If
Me.TextBox1.SelectionStart = 0 ' Markierung löschen
Me.TextBox1.SelectionLength = 0 ' "
End Sub

```

Hinweis

Sie finden die Dateien des Projektbeispiels im Ordner `\Beisp\Kap14\SetFileDate` auf der Begleit-CD. Sobald Sie das Projekt über die `.sln`-Datei laden und übersetzen, sollte sich die Anwendung ausführen lassen. Weitere Details zur Implementierung entnehmen Sie bitte den Quelldateien.

14.4 Verzeichnisse und Laufwerke bearbeiten

In den vorherigen Kapiteln haben Sie bereits mehrere Ansätze kennen gelernt, mit denen sich der Pfad zum Ordner, aus dem die `exe`-Datei gestartet wurde, ermitteln lässt. .NET Framework bietet zudem Methoden, um das aktuelle Verzeichnis abzufragen oder zu setzen, zu einem Laufwerk zu wechseln und mehr.

14.4.1 Aktuelles Verzeichnis abfragen

Das aktuelle Verzeichnis wird beim Programmstart gesetzt. Dies ist das Verzeichnis, auf welches die Anwendung zugreift, um Daten zu lesen. Der Ordner lässt sich einmal über die Visual Basic-Funktion `CurDir` ermitteln.

```
ThisDir = CurDir() & "\"
```

Alternativ können Sie die `GetCurrentDirectory()`-Methode der Klasse `Directory` im unten angegebenen Namensraum zur Abfrage des Verzeichnisses verwenden.


```
ThisDir = System.IO.Directory.GetCurrentDirectory() & "\"
```

Welchen Ansatz Sie verwenden, bleibt Ihnen überlassen. *CurDir* ist Bestandteil der Sprache Visual Basic und wird über die Kompatibilitätsklasse *Microsoft.VisualBasic* bereitgestellt.

Achtung

An dieser Stelle noch der Hinweis, dass das aktuelle Verzeichnis nichts mit dem Ordner zu tun hat, aus dem die .NET-Anwendung gestartet wurde. Wenn Sie beispielsweise die .NET-Anwendung über eine Verknüpfung aufrufen, wird das aktuelle Verzeichnis auf das Arbeitsverzeichnis der Verknüpfung gesetzt. Ist diese Angabe leer, wird das Verzeichnis der Verknüpfungsdatei verwendet. Möchten Sie von einer Anwendung auf Dokumentdateien zugreifen, können Sie diese Dateien im Ordner der Anwendung oder in einem Unterverzeichnis hinterlegen. Anschließend benutzen Sie die von mir bereits vorgestellte *GetPath*-Funktion, um den Pfad zur Programmdatei zu ermitteln. Dann ist sichergestellt, dass Sie auch Unterordner dieses Verzeichnisses per Programm adressieren können.

14.4.2 Aktuelles Verzeichnis wechseln

Zum Wechseln des aktuellen Verzeichnisses können Sie die Visual-Basic-Funktion *ChDir* benutzen.

```
ChDir (CurDir() & "\Test")
```

Die obige Anweisung setzt das neue Verzeichnis auf den mit *CurDir* ermittelten Unterordner *\Test*. Sie sollten aber eine Fehlerüberprüfung vornehmen, ob der Unterordner wirklich existiert.

14.4.3 Aktuelles Laufwerk wechseln/ermitteln

Zum Ermitteln des aktuellen Laufwerks können Sie eine benutzerspezifische Funktion verwenden, die einfach den Laufwerksbuchstaben aus dem von *CurDir* zurückgelieferten Pfad extrahiert.

```
Shared Function GetDrive() As String
    ' Extrahiere aktuelles Laufwerk
    Dim tmp as String = CurDir() ' aktuelles Verzeichnis
    If tmp.Substring(1,1) = ":" Then
        tmp = tmp.SubString(0,1) ' Laufwerksbuchstabe abtrennen
    Else
        tmp = ""
    End If
    Return tmp
End Function
```

Die obige Funktion gibt entweder den Laufwerksbuchstaben oder eine leere Zeichenkette zurück.

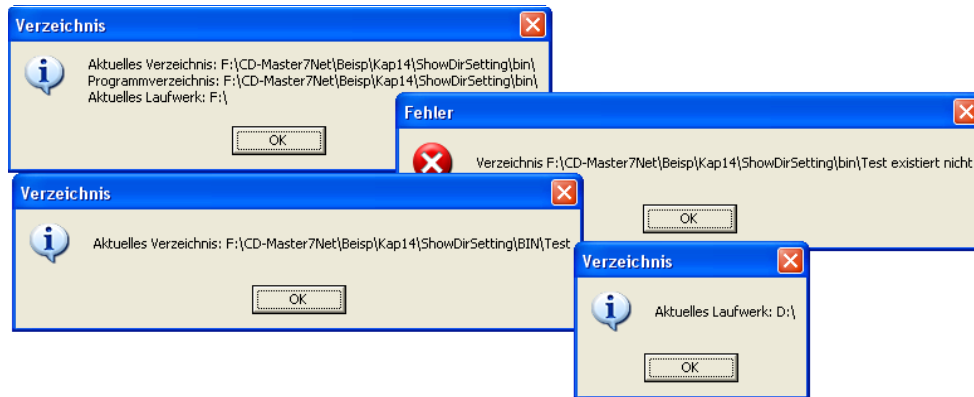


Abbildung 14.6: Dialoge des Beispielprogramms

Um das Laufwerk zu wechseln, verwenden Sie die Funktion *ChDrive* der Visual-Basic-Kompatibilitätsklasse.

```
ChDrive ("D")
```

Die obige Anweisung setzt das Laufwerk *D:* als aktuelles Laufwerk. Anschließend können Sie über *ChDir* das aktuelle Verzeichnis dieses Laufwerks anpassen.

Das folgende Listing demonstriert den Umgang mit den obigen Funktionen. Das Programm ermittelt die Pfade, zeigt diese an und setzt dann das aktuelle Laufwerk und das aktuelle Verzeichnis. Beim Programmablauf informieren Dialoge über die aktuellen Einstellungen (Abbildung 14.6).

```
'*****
' File/Projekt: ShowDirSettings
' Autor: G. Born www.borncity.de
'
' Zeigt und setzt das aktuelle Verzeichnis.
'*****
Option Strict On

Imports System.Reflection      ' für Pfad

Public Class Text
    Shared Sub Main ()
        Dim MyDir As String
        Dim ThisDir As String
```

Listing 14.5: Datei ShowDirSettings.vb zum Anpassen des aktuellen Ordners

```

Dim MyDrive As String

ThisDir = CurDir() & "\" ' aktuelles Verzeichnis
' oder mit System.IO.Directory.GetCurrentDirectory()
MyDir = Getpath()      ' Programmverzeichnis
MyDrive = GetDrive()   ' Aktuelles Laufwerk

MsgBox ("Aktuelles Verzeichnis: " & ThisDir & _
        vbCrLf & "Programmverzeichnis: " & MyDir & _
        vbCrLf & "Aktuelles Laufwerk: " & MyDrive & ":\", _
        vbOkOnly Or vbInformation, "Verzeichnis")
' Versuche über aktuelles Verzeichnis in \Test zu wechseln
Try
    ChDir ("Test")      ' in Unterverzeichnis
Catch ex As Exception
    MsgBox ("Verzeichnis " & ThisDir & "Test existiert nicht", _
            vbOkOnly Or vbCritical, "Fehler")
End Try

' Versuche über Programmverzeichnis in \Test zu wechseln
Try
    ChDir (MyDir & "Test") ' in Unterverzeichnis
    MsgBox ("Aktuelles Verzeichnis: " & CurDir(), _
            vbOkOnly Or vbInformation, "Verzeichnis")
Catch ex As Exception
    MsgBox ("Verzeichnis " & MyDir & "Test existiert nicht", _
            vbOkOnly Or vbCritical, "Fehler")
End Try

' Versuche aktuelles Laufwerk zu wechseln
Try
    ChDrive ("D")      ' Laufwerk D
    MsgBox ("Aktuelles Laufwerk: " & GetDrive() & ":\", _
            vbOkOnly Or vbInformation, "Verzeichnis")
Catch ex As Exception
    MsgBox ("Laufwerk D:\ existiert nicht", _
            vbOkOnly Or vbCritical, "Fehler")
End Try
End Sub
' #### Hilfsroutinen

```

Listing 14.5: Datei ShowDirSettings.vb zum Anpassen des aktuellen Ordners (Forts.)

```

Shared Function GetDrive() As String
    ' Extrahiere aktuelles Laufwerk
    Dim tmp As String = CurDir()    ' aktuelles Verzeichnis
    If tmp.Substring(1,1) = ":" Then
        tmp = tmp.SubString(0,1)    ' Laufwerksbuchstabe abtrennen
    Else
        tmp = ""
    End If
    Return tmp
End Function
Shared Function GetPath() As String
    ' Extrahiere den Pfad der laufenden Anwendung
    Dim oMod As System.Reflection.Module = _
        [Assembly].GetExecutingAssembly().GetModules()(0)
    Dim pfad As String
    Dim len1 As Integer
    len1 = oMod.Name.Length          ' Länge Dateiname
    pfad = oMod.FullyQualifiedName    ' Pfad mit Dateiname
    pfad = pfad.Remove(pfad.Length - len1, len1)
    Return pfad
End Function
End Class

```

Listing 14.5: Datei ShowDirSettings.vb zum Anpassen des aktuellen Ordners (Forts.)

Hinweis

Sie finden die Projektdateien im Ordner `\Beisp\Kap14\ShowDirSettings` der Begleit-CD. Der obige Code findet sich im Modul `ShowDirSettings.vb`. Sobald Sie die `.sln`-Projektdatei laden und das Projekt übersetzen, lässt sich die Anwendung ausführen. Die Anwendung prüft, ob der Unterordner `\Test` im Verzeichnis mit der ausführbaren Programmdatei vorhanden ist. Fehlt dieser Ordner, wird ein Fehlerdialog ausgelöst. Weitere Details zur Implementierung entnehmen Sie bitte den Quelldateien.

14.5 Kopieren, löschen, verschieben, umbenennen

Zum Manipulieren von Dateien und Ordnern können Sie die Funktionen der Visual Basic-Kompatibilitätsklasse nutzen. Dies kommt Umsteigern von älteren Visual-Basic-Versionen entgegen. Alternativ können Sie aber auch auf die Methoden des .NET Framework zurückgreifen. Nachfolgend werden beide Ansätze an einfachen Beispielen demonstriert.

14.5.1 Verwenden der Visual-Basic-Funktionen

Ein Verzeichnis legen Sie direkt mit der Visual-Basic-Funktion *MkDir* an.

```
MkDir ("C:\MyTest")
```

Zum Löschen dieses Verzeichnisses lässt sich die *RmDir*-Funktion verwenden.

```
RmDir ("C:\MyTest")
```

Beachten Sie aber, dass das Verzeichnis leer sein muss. Andernfalls wird ein Laufzeitfehler ausgelöst. Das Umbenennen eines Ordners oder einer Datei können Sie mit der Visual-Basic-Funktion *Rename* vornehmen, indem Sie in den beiden Parametern den alten und neuen Namen vorgeben.

```
Rename (MyDir & "MyTest", MyDir & "MyTest1")
```

Zum Kopieren von Dateien oder Ordnern lässt sich die Visual-Basic-Funktion *FileCopy* benutzen. Auch hier sind als Parameter die Namen der alten und neuen Datei samt Pfad anzugeben.

```
FileCopy(MyDir & "MyTest1\Test.txt", _  
        MyDir & "MyTest1\T1\Test1.txt")
```

Falls die Operation nicht möglich ist, wird ein Laufzeitfehler ausgelöst. Das Löschen von Dateien können Sie mittels der Visual Basic-Funktion *Kill* ausführen. Der Aufruf

```
Kill ("C:\Test\*.txt")
```

löscht alle Textdateien (.txt) im angegebenen Ordner. Das folgende Listing demonstriert den Einsatz dieser Befehle, indem es im Ordner, aus dem das Programm gestartet wird, einen Unterordner *Test* anlegt und diesen mit einem Unterverzeichnis versieht. Dann wird eine Textdatei im Unterordner *\Test* erzeugt, diese ist in den Unterordner *\T1* zu kopieren. Zudem wird der Unterordner umbenannt. Anschließend beginnt das Programm mit dem Löschen der angelegten Verzeichnisse und Dateien. Der Ablauf wird durch Dialogfelder begleitet, so dass Sie die einzelnen Schritte im Windows-Ordnerfenster kontrollieren können. Nach erfolgreichem Programmablauf sollten die Unterordner samt Dateien verschwunden sein.

```
'*****  
' File/Projekt: FileHandling  
' Autor: G. Born www.borncity.de  
'  
' Zeigt das Anlegen, Löschen und Manipulieren von Dateien  
' und Ordnern mittels VB-Funktionen.  
' Die Anwendung legt die Dateien/Ordner im aktuellen
```

Listing 14.6: Beispielcode zur Demonstration von Dateioperationen

```
' Verzeichnis an - läuft also nicht auf CD/DVD.
'*****
Option Strict On

Imports System.Reflection      ' für Pfad

Public Class Text
    Shared Sub Main ()
        On Error Goto Fehler      ' Fehlerbehandlung

        Dim MyDir As String
        Dim ThisDir As String
        Dim MyDrive As String

        MyDir = Getpath()          ' Programmverzeichnis
        Mkdir (MyDir & "MyTest")    ' \MyTest anlegen
        Mkdir (MyDir & "MyTest\T1") ' \T1 anlegen

        MsgBox ("Verzeichnis \MyTest und .\T1 angelegt", _
            vbOkOnly Or vbInformation, "Erfolg")

        Rename (MyDir & "MyTest", MyDir & "MyTest1") ' umbenennen

        MsgBox ("Verzeichnis \MyTest in \MyTest1 umbenannt", _
            vbOkOnly Or vbInformation, "Erfolg")

        CreateTxtFile(MyDir & "MyTest1\Test.txt") ' Blinddatei

        MsgBox ("Textdatei im Verzeichnis \MyTest1 angelegt", _
            vbOkOnly Or vbInformation, "Erfolg")

        FileCopy(MyDir & "MyTest1\Test.txt", _
            MyDir & "MyTest1\T1\Test1.txt") ' kopieren

        MsgBox ("Textdatei in Verzeichnis \MyTest1\T1 kopiert", _
            vbOkOnly Or vbInformation, "Erfolg")

        Kill (MyDir & "MyTest1\T1\*.*) ' Lösche Dateien
        Kill (MyDir & "MyTest1\*.*)    ' "
        Rmdir (MyDir & "MyTest1\T1")   ' Lösche Verzeichnis
```

Listing 14.6: Beispielcode zur Demonstration von Dateioperationen (Forts.)

```

MsgBox ("Verzeichnis \MyTest1 sollte leer sein", _
        vbOkOnly Or vbInformation, "Erfolg")

Rmdir (MyDir & "MyTest1")      ' Lösche Verzeichnis \MyTest1

MsgBox ("Fertig", vbOkOnly Or vbInformation, "Erfolg")

Exit Sub
Fehler:
    MsgBox (err.Description, vbOkOnly Or vbCritical, "Fehler")
End Sub

' ### Hilfsroutinen
Shared Sub CreateTxtFile(ByVal file As String)

    Dim oStream As IO.StreamWriter  ' Lege eine neue Datei an
    Dim oFile As IO.FileStream

    oFile = New IO.FileStream(file, IO.FileMode.Create)
    oStream = New IO.StreamWriter(oFile)  ' Stream
    oStream.WriteLine("Ein Testtext")     ' schreiben
    oStream.Close()                      ' Schließen
    oFile.Close()

End Sub

Shared Function GetPath() As String
' Extrahiere den Pfad der laufenden Anwendung
    Dim oMod As System.Reflection.Module = _
        [Assembly].GetExecutingAssembly().GetModules()(0)
    Dim pfad As String
    Dim len1 As Integer

    len1 = oMod.Name.Length      ' Länge Dateiname
    pfad = oMod.FullyQualifiedName  ' Pfad mit Dateiname
    pfad = pfad.Remove(pfad.Length - len1, len1)
    Return pfad
End Function
End Class

```

Listing 14.6: Beispielcode zur Demonstration von Dateioperationen (Forts.)

Hinweis

Sie finden die Projektdateien dieses Beispiels im Ordner `\Beisp\Kap14\DateiOperationenVB` auf der Begleit-CD. Der hier gezeigte Quellcode findet sich in der Projektdatei `Filehandling.vb`. Das Beispiel legt einen Unterordner mit Testdateien im Verzeichnis mit dem Quellcode an. Sie können das Beispiel also nicht von CD/DVD ausführen. Weiterhin sollte sichergestellt sein, dass das Verzeichnis `\MyTest` noch nicht existiert. Da das Programm im Ablauf durch verschiedene Dialogfelder unterbrochen wird, können Sie in einem Ordnerfenster den Einfluss der einzelnen Operationen verfolgen. Achten Sie beim Testen darauf, dass Sie die Datei nur mit Vorsicht und in einem eigenen Ordner der Festplatte ausführen. Andernfalls besteht bei Modifikationen des Programmcodes die Gefahr, dass Sie sich unbeabsichtigt Dateien löschen.

14.5.2 Dateibearbeitung per File- und Directory-Klasse

Die bereits in den weiter oben besprochenen Beispielen benutzten Klassen *File* und *Directory* des .NET Framework besitzen ebenfalls Methoden, mit denen sich Dateien und Ordner auf Dateisystemebene handhaben lassen. Nachfolgend möchte ich in einem einfachen Beispiel demonstrieren, wie sich Ordner und Dateien anlegen, kopieren, verschieben, umbenennen oder löschen lassen.

Die betreffenden Klassen stehen im Namensraum *System.IO* zur Verfügung und können mit der Anweisung *Imports System.IO* als Import vereinbart werden. Anschließend lässt sich direkt auf die Klassen und deren Methoden zugreifen. Die Prüfung, ob eine Datei oder ein Ordner bereits im Zielverzeichnis existiert, lässt sich über die *Exists()*-Methode bewerkstelligen. Die Methode wird in beiden Klassen bereitgestellt und liefert den Wert *True*, falls das Element gefunden wurde.

```
If Directory.Exists("C:\MyTest1") Then ...
If File.Exists("C:\MyTest1\Text.txt") Then ...
```

Die beiden obigen Anweisungen prüfen, ob ein Ordner bzw. eine Datei vorhanden ist und leiten dann eine Aktion ein. Die folgende Befehlssequenz prüft beispielsweise, ob ein Ordner bereits vorhanden ist und löscht diesen gegebenenfalls:

```
MyDir = GetPath() ' aktueller Programmordner
If Directory.Exists(MyDir & "MyTest1") Then _
    Directory.Delete(MyDir & "MyTest1", True)
```

Das Löschen des Ordners erfolgt über die *Delete()*-Methode der Klasse *Directory*. Der Methode ist im ersten Parameter der Pfad zum Ordner zu übergeben. Ein optionales zweites Argument lässt sich auf *True* setzen, um das Löschen eines nicht leeren Ordners zu erzwingen.

Bei der Klasse *File* existiert eine gleichnamige Methode, mit der sich eine nicht geöffnete Datei löschen lässt. Als Parameter ist der Dateiname samt Pfad zu übergeben.


```
File.Delete(path & "\\Test.txt")
```

Seien Sie vorsichtig bei Anwendung der *Delete()*-Methode, da diese die Daten nicht in den Papierkorb verschiebt, sondern direkt löscht. Die beiden Klassen unterstützen auch das Anlegen von Ordnern und Dateien. Das Anlegen eines neuen Verzeichnisses reduziert sich auf eine Zeile:

```
Directory.CreateDirectory(path)
```

Als Parameter ist der Pfad und der Verzeichnisname zu übergeben. Über die *Exists()*-Methode lässt sich ggf. vorher prüfen, ob das Verzeichnis bereits existiert. Zum Anlegen einer Datei kommen folgende Anweisungen zum Einsatz:

```
If Not File.Exists(MyDir & "MyTest1\\Test.txt") Then
    oFile = File.Create(MyDir & "MyTest1\\Test.txt")
    oFile.Close() ' Schließen, damit Datei frei wird
End If
```

Die erste Zeile prüft, ob die Datei noch nicht existiert. Dann wird die *Create()*-Methode angewandt. Die Methode erwartet den Dateinamen samt Pfad als Argument. Die Methode liefert ein *FileStream*-Objekt zurück, über welches sich schreibend oder lesend auf den Inhalt der Datei zugreifen lässt. Falls Sie die Datei später noch kopieren oder löschen müssen, ist diese zwingend zu schließen. Daher habe ich in obiger Sequenz sofort einen Aufruf der *Close()*-Methode angehängt.

Das Umbenennen und Verschieben wird in der Klasse *Directory* über die *Move()*-Methode unterstützt:

```
Directory.Move("C:\\MyTest", "C:\\MyTest1")
```

Die obige Anweisung benennt den Ordner *MyTest* in *MyTest1* um. Zum Kopieren eines Ordners liefert die Klasse *Directory* leider keine Methode. Möchten Sie ein Verzeichnis kopieren, müssen Sie die *CopyDirectory()*-Methode der *FileSystem*-Klasse verwenden:

```
My.Computer.FileSystem.CopyDirectory(("C:\\MyTest ", "C:\\MyTest1")
```

Um Dateien zu kopieren, wenden Sie einfach die *Copy()*-Methode der Klasse *File* an:

```
File.Copy("C:\\MyTest\\Test.txt", "C:\\MyTest\\T1\\Test1.txt")
```

Die obige Anweisung kopiert die Textdatei *Test.txt* in den Unterordner *T1*, wobei der Dateiname in *Test1.txt* geändert wird. Das nachfolgende Listing demonstriert den Umgang mit den Methoden der beiden Klassen:

```
'*****
' File/Projekt: FileHandlingNET
' Autor: G. Born www.borncity.de
'
' Zeigt das Anlegen, Löschen und Manipulieren von Dateien
' und Ordern unter Verwendung der Klassen File und Directory.
' Benötigt Zugriffsrechte auf die betreffenden Ordner/Dateien,
' lässt sich also auch nicht von CD/DVD ausführen.
'*****
Option Strict On

Imports System.IO          ' für Dateizugriffe
Imports System.Reflection  ' für Pfad

Public Class Text
    Shared Sub Main ()
        On Error GoTo Fehler ' Fehlerbehandlung

        Dim oFile As FileStream ' für die Testdatei

        Dim MyDir As String
        Dim path As String      ' Pfade
        Dim path1 As String

        MyDir = GetPath()       ' Programmverzeichnis
        path = MyDir & "MyTest"
        path1 = path & "\T1"

        ' aufräumen, falls Zielordner 2 bereits existiert
        ' Ordner anlegen, falls er noch nicht existiert
        If Directory.Exists(MyDir & "MyTest1") Then _
            Directory.Delete(MyDir & "MyTest1", True) ' zwangsweise löschen

        ' Ordner anlegen, falls er noch nicht existiert
        If Not Directory.Exists(path) Then _
            Directory.CreateDirectory(path) ' \MyTest anlegen

        If Not Directory.Exists(path1) Then _
            Directory.CreateDirectory(path1) ' \T1 anlegen
```

Listing 14.7: Code zur Dateibearbeitung

```

MsgBox("Verzeichnis \MyTest und \T1 angelegt", _
      vbOKOnly Or vbInformation, "Erfolg")

Directory.Move(path, path & "1") ' Ordner umbenennen

MsgBox("Verzeichnis \MyTest in \MyTest1 umbenannt", _
      vbOKOnly Or vbInformation, "Erfolg")

If Not File.Exists(MyDir & "MyTest1\Test.txt") Then
' erzeuge die Testdatei
  oFile = File.Create(MyDir & "MyTest1\Test.txt")
  oFile.Close() ' Schließen, damit Datei frei wird
  MsgBox("Textdatei im Verzeichnis \MyTest1 angelegt", _
        vbOKOnly Or vbInformation, "Erfolg")
Else
  MsgBox("Textdatei im Verzeichnis \MyTest1 bereits vorhanden", _
        vbOKOnly Or vbInformation, "Hinweis")
End If

path = MyDir & "MyTest1" ' umbenennen
path1 = path & "\T1"

File.Copy(path & "\Test.txt", _
  path & "\T1\Test1.txt") ' Datei kopieren

MsgBox("Textdatei in Verzeichnis \MyTest1\T1 kopiert", _
      vbOKOnly Or vbInformation, "Erfolg")

Directory.Delete(path1, True) ' Lösche Unterordner

MsgBox("Ordner \T1 in \MyTest1 gelöscht", _
      vbOKOnly Or vbInformation, "Erfolg")

File.Delete(path & "\Test.txt") ' Lösche Datei

MsgBox("Datei \Test.txt in \MyTest1 gelöscht", _
      vbOKOnly Or vbInformation, "Erfolg")

Directory.Delete(path, True) ' Lösche Ordner

```

Listing 14.7: Code zur Dateibearbeitung (Forts.)

```

MsgBox("Alles gelöscht, fertig", _
    vbOKOnly Or vbInformation, "Erfolg")

Exit Sub
Fehler:
    MsgBox(Err.Description, vbOKOnly Or vbCritical, "Fehler")
End Sub

' ##### Hilfsroutinen

Shared Function GetPath() As String
' Extrahiere den Pfad der laufenden Anwendung
Dim oMod As System.Reflection.Module = _
    [Assembly].GetExecutingAssembly().GetModules()(0)
Dim pfad As String
Dim len1 As Integer

len1 = oMod.Name.Length          ' Länge Dateiname
pfad = oMod.FullyQualifiedName    ' Pfad mit Dateiname
pfad = pfad.Remove(pfad.Length - len1, len1)
Return pfad
End Function
End Class

```

Listing 14.7: Code zur Dateibearbeitung (Forts.)

Hinweis

Sie finden die Projektdateien des Beispiels samt der Datei *FileHandlingNET.vb* im Ordner *\Beisp\Kap14\DateiOperationenNET* auf der Begleit-CD. Sie können die *.sln*-Datei in der Entwicklungsumgebung laden und dann die Details des Quellcode studieren. Beachten Sie, dass sich die auf der Beispiel-CD im Ordner *\bin* hinterlegte *.exe*-Datei nur von Festplatte ausführen lässt, da während des Ablaufs ein Unterordner *MyTest* im Verzeichnis der Programmdatei angelegt wird.

14.6 Zugriffe auf Dateiinhalte

Neben der Handhabung ganzer Dateien ist der Zugriff auf Dateiinhalte häufiger gefragt. In den vorherigen Kapiteln haben Sie bereits gelernt, wie sich Dateidialoge zur Dateiauswahl aufrufen lassen. Zudem habe ich im Rahmen von Beispielen bereits ganz kurz skizziert, wie sich Textdateien lesen und schreiben lassen. Nachfolgend möchte ich noch etwas auf dieses Thema eingehen.

14.6.1 Allgemeine Hinweise zu Dateizugriffen

Für Dateizugriffe bietet das .NET Framework verschiedene Varianten. Einige haben Sie bereits in früheren Beispielen kennen gelernt. So bietet die Klasse des *RichTextBox*-Steuerelements Methoden, um Dateien im Text- oder RTF-Format zu speichern oder zu laden (siehe Beispiele »RTF-App« und »MDI-Anwendung« in *Kapitel 12* sowie im Beispiel *Dump.vb* auf den folgenden Seiten). Auch das Laden von Bilddateien (Bitmaps und Icons) wird von einigen Steuerelementen unterstützt.

Im .NET Framework sind für Dateizugriffe aber die Klassen *FileStream*, *BinaryReader*, *BinaryWriter*, *StreamReader* und *StreamWriter* vorgesehen. In diesem Abschnitt möchte ich auf die Möglichkeit zum Dateizugriff über die Klasse *FileStream* zu sprechen kommen. Dabei wird demonstriert, wie sich anschließend auf Textdateien mittels der Klassen *StreamReader/StreamWriter* oder auf Binärdateien mittels der Klassen *BinaryReader/BinaryWriter* zugreifen lässt. Den Zugriff auf Textdateien haben Sie bereits in Beispielen kennen gelernt (siehe Beispiel »Dateidialoge« in *Kapitel 12*).

Dateizugriffe mittels der .NET Framework-Klassen werden (in der Regel) in zwei Stufen durchgeführt. Den Zugriff (Anlegen, Öffnen etc.) auf die Datei regelt ein *FileStream*-Objekt. Das Lesen oder Schreiben der Daten erfolgt dann über ein weiteres Objekt der bereits erwähnten Klassen (z.B. *StreamReader*), welches diverse Methoden bereitstellt.

14.6.2 Eine neue Datei anlegen und mit Text füllen

Das Anlegen einer Datei kann über das *FileStream*-Objekt der Klasse *FileStream* im Namensraum *System.IO* erfolgen. Eine Datei lässt sich im einfachsten Fall mit der folgenden Anweisung erzeugen:

```
Dim oFile As IO.FileStream = New IO.FileStream("C:\Test.dat", IO.FileMode.Create)
```

Der *FileStream*-Konstruktor der Klasse *FileStream* erwartet bei obigem Aufruf im ersten Argument den Namen samt Pfad der Datei. Das zweite Argument enthält eine *FileMode*-Konstante (siehe folgende Tabelle), die festlegt, wie die Datei zu öffnen ist.

Konstante	Bemerkung
<i>FileMode.Append</i>	Erlaubt, Daten an eine vorhandene Datei anzufügen, bei fehlender Datei wird diese erzeugt und die Daten können geschrieben werden
<i>FileMode.Create</i>	Legt eine neue Datei an, eine existierende Datei wird überschrieben
<i>FileMode.CreateNew</i>	Es wird eine neue Datei angelegt. Existiert die Datei bereits, löst dies eine Ausnahme (Laufzeitfehler) aus.
<i>FileMode.Open</i>	Die bereits existierende Datei wird geöffnet. Der Benutzer muss eine Leseberechtigung besitzen.
<i>FileMode.OpenOrCreate</i>	Es wird das Öffnen versucht. Existiert die Datei nicht, wird sie angelegt.
<i>FileMode.Truncate</i>	Eine bereits vorhandene Datei wird geöffnet, dann aber auf die Länge 0 gekürzt

Tabelle 14.1: Dateimodi für FileStream-Aufrufe

Je nach Modus wird vorausgesetzt, dass der Benutzer die entsprechende Berechtigung zum Lesen oder Schreiben für die Datei besitzt. Fehlt die Berechtigung, löst dies eine Ausnahme aus.

Hinweis

Die Klasse unterstützt weitere Varianten des *FileStream*-Konstruktors, die zusätzliche Parameter zum Festlegen der Berechtigungen sowie des *Share*-Modus erlauben. Einzelheiten finden Sie in der Hilfe des .NET Frameworks.

Sobald das *FileStream*-Objekt existiert, lässt sich mit Methoden wie *StreamWriter()* oder *StreamReader()* auf den Inhalt der Datei zugreifen. Das nachfolgende Listing zeigt, wie sich eine Textdatei mittels der hier beschriebenen Methoden anlegen und dann über *StreamWriter()* mittels der *WriteLine()*-Methode mit zehn Zeilen Text füllen lässt. Die *WriteLine()*-Methode bewirkt, dass jede Zeile mit einem Zeilenumbruch abgeschlossen wird. Verwenden Sie stattdessen die *Write()*-Methode, werden keine Zeilenwechsel eingefügt.

```
'*****
' File/Projekt: TxtFileCreate
' Autor: G. Born www.borncity.de
'
' Zeigt das Anlegen einer Datei und das Schreiben eines
' Textes über das FileStream-Objekt. Benutzt StreamWriter
' zum Füllen der Datei.
'
' Achtung: Legt eine Textdatei CreateTest.txt im
' Unterverzeichnis \Test an, läuft daher nicht von
' CD-ROM. Falls Sie den Ordner der CD kopieren, ist
' die Testdatei bereits vorhanden. Dann liefert das
' Programm den Hinweis, dass CreateTest.txt existiert.
' Löschen Sie die Textdatei.
'*****
Option Strict On

Imports System.Reflection      ' für Pfad

Public Class Text
    Shared Sub Main ()
        On Error Goto Fehler      ' Fehlerbehandlung

        Const SubFolder As String = "Test"
        Const FileName As String = "CreateTest.txt"
```

Listing 14.8: Code zum Erzeugen einer Textdatei

```

Dim i As Integer

' Variable für Datei initialisieren
Dim file As String = _
    GetPath() & SubFolder & "\" & FileName
Dim oFile As IO.FileStream ' regelt Dateizugriff
Dim oStream As IO.StreamWriter ' regelt Zugriff auf Inhalte

If Not IO.File.Exists(file) Then ' gibt es die Datei?
    ' Nein -> neu anlegen
    oFile = New IO.FileStream(file, IO.FileMode.Create)
    oStream = New IO.StreamWriter(oFile) ' Stream öffnen

    For i = 0 To 10 ' 10 Zeilen schreiben
        oStream.WriteLine(i.ToString & ": Ein Testtext.")
    Next i
    oStream.Close() ' Stream schließen
    oFile.Close() ' Datei schließen
    MsgBox ("Datei " & file & " erzeugt", _
        vbOkOnly Or vbInformation, "Erfolg")
Else
    MsgBox ("Datei " & file & " existiert bereits", _
        vbOkOnly Or vbInformation, "Abbruch")
End If
Exit Sub

Fehler: ' Fehler abfangen
MsgBox (err.Description, vbOkOnly Or vbCritical, "Fehler")
End Sub

' #### Hilfsroutinen
Shared Function GetPath() As String
    ' Extrahiere den Pfad der laufenden Anwendung
    Dim oMod As System.Reflection.Module = _
        [Assembly].GetExecutingAssembly().GetModules()(0)
    Dim pfad As String
    Dim len1 As Integer

    len1 = oMod.Name.Length ' Länge Dateiname
    pfad = oMod.FullyQualifiedName ' Pfad mit Dateiname

```

Listing 14.8: Code zum Erzeugen einer Textdatei (Forts.)

```

        pfad = pfad.Remove(pfad.Length - len1, len1)
        Return pfad
    End Function
End Class

```

Listing 14.8: Code zum Erzeugen einer Textdatei (Forts.)

Hinweis

Sie finden die Projektdateien samt der Datei *TxtFileCreate.vb* mit dem Beispielcode im Ordner *\Beisp\Kap14\ TxtFileCreate* auf der Begleit-CD. Das Beispiel setzt voraus, dass im Verzeichnis mit der Programmdatei der Unterordner *\Test* vorhanden ist. Wenn Sie dann das Programm ausführen, wird eine Textdatei *CreateTest.txt* im Unterordner *\Test* angelegt.

14.6.3 Den Inhalt einer Textdatei lesen

Im nächsten Schritt möchte ich jetzt zeigen, wie Sie den Inhalt einer Textdatei lesen können. Dies haben Sie zwar bereits in früheren Beispielen genutzt, das Einlesen soll jetzt aber zeilenweise passieren. Zum Lesen wird die oben erzeugte Beispieldatei *CreateTest.txt* benutzt. Deren Inhalt wird zeilenweise gelesen, in einer Zeichenkette hinterlegt und dann über ein Dialogfeld angezeigt (Abbildung 14.7).

Das Anlegen der zum Öffnen der Datei und zum Lesen des Streams erforderlichen Objektinstanzen gleicht dem obigen Ansatz:

```

Dim oFile As IO.FileStream = _
    New IO.FileStream("C:\Test.txt", IO.FileMode.Open)
Dim oStream As IO.StreamReader = _
    New IO.StreamReader(oFile)

```

In der ersten Anweisung wird das *FileStream*-Objekt zum Zugriff auf die Datei vereinbart. Als Parameter werden der Dateiname samt Pfad sowie der Zugriffsmodus im Konstruktor angegeben. Hier habe ich den Zugriffsmodus *FileMode.Open* benutzt, was zum Öffnen reicht. Die zweite Anweisung wendet die *StreamReader()*-Methode auf das zurückgegebene *FileStream*-Objekt an.

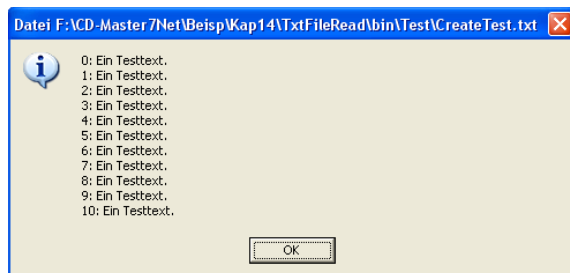


Abbildung 14.7: Anzeige des Dateiinhalts

Das Einlesen des Dateiinhalts soll anschließend zeilenweise bis zum Dateiende erfolgen. Hierzu lässt sich die *ReadLine()*-Methode benutzen, die immer eine Zeile der Datei bis zum nächsten Zeilenwechsel einliest.

```
Do
    tmp = oStream.ReadLine()      ' lese Zeile
    If IsNothing(tmp) = "" Then Exit Do ' Dateiende erreicht?
    txt = txt & tmp & vbCrLf      ' in Ausgabepuffer
Loop Until False
```

Die *ReadLine()*-Methode liefert den gelesenen Text als *String* zurück. Wurde das Dateiende erreicht, wird eine leere Zeichenkette (entspricht dem VB-Wert *Nothing*) zurückgegeben. Ich habe in der obigen Sequenz eine Endlosschleife benutzt, die über *Exit Do* verlassen wird. Innerhalb der Schleife prüft eine *If*-Anweisung, ob noch Daten gelesen werden und überträgt diese Daten ggf. in eine Zeichenkette. Da *ReadLine()* das Zeichen für den Zeilenumbruch entfernt, wird hier bei jeder Zeile ein expliziter Zeilenumbruch über *vbCrLf* an die Zeichenkette angehängt.

Hinweis

Die *StreamReader()*-Methode erlaubt Ihnen über einen optionalen zweiten Parameter die in der Textdatei benutzte Zeichencodierung zu berücksichtigen. Die Anweisung:

```
Dim oStream As IO.StreamReader = _
    New IO.StreamReader(oFile, Text.Encoding.GetEncoding(1251))
```

setzt den Zeichencode 1251. Ein Wert *True* anstelle eines Zeichencodes weist den Reader an, die Codierung selbst zu ermitteln. Neben *ReadLine()* bietet die *StreamReader*-Klasse weitere Methoden. *Read()* liest ein Zeichen oder die angegebene Zeichenzahl ein und gibt diese entweder als Integerwert über die Funktion bzw. in einem beim Aufruf als Parameter übergebenen Puffer zurück. *ReadBlock()* liest die angegebene Zahl an Zeichen in einem Puffer zurück und *ReadToEnd()* liest alle Zeichen bis zum Dateiende. Details zu den Aufrufen liefert die Hilfe des .NET Framework.

Das nachfolgende Listing zeigt den kompletten Beispielcode, welcher die vorhandene Datei *CreateTest.txt* im Unterordner *\Test* einliest und deren Inhalt als Dialogfeld anzeigt (Abbildung 14.7):

```
'*****
' File/Projekt: TxtFileRead
' Autor: G. Born www.borncity.de
'
' Liest eine Textdatei über das FileStream-Objekt
' unter Verwendung von StreamReader.
'*****
```

Listing 14.9: Die Datei *TxtFileRead.vb* liest eine Textdatei

```

Option Strict On

Imports System.Reflection      ' für Pfad

Public Class Text
    Shared Sub Main ()
        On Error Goto Fehler    ' Fehlerbehandlung

        Const SubFolder As String = "Test"
        Const FileName As String = "CreateTest.txt"

        Dim tmp As String = ""
        Dim txt As String = ""

        ' Variable für Datei initialisieren
        Dim file As String = _
            GetPath() & SubFolder & "\" & FileName
        Dim oFile As IO.FileStream    ' regelt Dateizugriff
        Dim oStream As IO.StreamReader ' regelt Zugriff auf Inhalte

        If IO.File.Exists(file) Then ' gibt es die Datei?
            ' Ja -> versuche zu Öffnen und dann zu Lesen
            oFile = New IO.FileStream(file, IO.FileMode.Open)
            oStream = New IO.StreamReader(oFile) ' Stream öffnen

            ' jetzt lesen, bis Dateiende erreicht ist
            Do
                tmp = oStream.ReadLine()    ' lese Zeile
                If IsNothing(tmp) Then Exit Do ' Dateiende erreicht?
                txt = txt & tmp & vbCrLf    ' in Ausgabepuffer
            Loop Until False

            ' Ergebnis anzeigen
            MsgBox (txt, vbOkOnly Or vbInformation, "Datei " & file)
            oStream.Close()    ' Stream schließen
            oFile.Close()      ' Datei schließen
        Else
            MsgBox ("Datei " & file & " existiert nicht", _
                vbOkOnly Or vbInformation, "Abbruch")
        End If
    End Sub
End Class

```

Listing 14.9: Die Datei TxtFileRead.vb liest eine Textdatei (Forts.)

```

Exit Sub

Fehler: ' Fehler abfangen
        MsgBox (err.Description, vbOkOnly Or vbCritical, "Fehler")
End Sub

' ##### Hilfsroutinen
Shared Function GetPath() As String
' Extrahiere den Pfad der laufenden Anwendung
    Dim oMod As System.Reflection.Module = _
        [Assembly].GetExecutingAssembly().GetModules()(0)
    Dim pfad As String
    Dim len1 As Integer

    len1 = oMod.Name.Length          ' Länge Dateiname
    pfad = oMod.FullyQualifiedName   ' Pfad mit Dateiname
    pfad = pfad.Remove(pfad.Length - len1, len1)
    Return pfad
End Function
End Class

```

Listing 14.9: Die Datei TxtFileRead.vb liest eine Textdatei (Forts.)

Hinweis

Sie finden Projektdateien sowie die Datei *TxtFileRead.vb* mit dem Beispielcode im Ordner *\Beisp\Kap14\TxtFileRead* auf der Begleit-CD. Sie können die ausführbare Programmdatei von der CD-ROM starten, da der Unterordner *\Test* mit der erforderlichen Datei existiert.

14.6.4 Textdatei lesen und verändern

In einem weiteren Beispiel soll demonstriert werden, wie sich eine Textdatei lesen und dann um weitere Zeilen erweitern lässt. An die weiter oben angelegte Textdatei sollen einige Zeilen angehängt werden (Abbildung 14.8). Zur Unterscheidung der Dateien habe ich der hier benutzten Testdatei den neuen Namen *TestMe.txt* gegeben.

Das Programm ist so angelegt, dass es beim Start erkennt, ob die Testdatei existiert. Falls nicht, soll über eine Hilfsprozedur *MakeFile* die Datei *ReadMe.txt* mit einigen Textzeilen generiert werden. Danach wird die Datei neu geöffnet, das Programm hängt die Zeilen an, schließt die Datei und verwendet die Hilfsprozedur *ShowFile* zur Anzeige des Datei-inhalts. Die beiden Hilfsprozeduren sind so implementiert, dass die Streams und die *FileStream*-Objekte der Dateien vor dem Beenden geschlossen werden. Dies stellt sicher, dass sich die Datei zum Anhängen der Datensätze öffnen lässt.

Der Code zum Öffnen der bestehenden Textdatei und zum Anhängen einer Zeile, enthält nur wenige Anweisungen. Geöffnet wird die Datei über die folgende Anweisung:

```
oFile = New IO.FileStream(file, IO.FileMode.Append)
```

Der Dateimodus *FileMode.Append* erzeugt ggf. eine neue Datei, erlaubt aber nicht das Lesen. Da im Beispiel vorher die Datei *MakeFile* aufgerufen wurde, liegt die Textdatei bereits vor. *Append* bewegt den Schreibzeiger des *FileStream*-Objekts an das Ende der Datei. Nun muss noch eine Instanz des *StreamWriter*-Objekts erzeugt werden:

```
Dim oStreamOut As IO.StreamWriter
oStreamOut = New IO.StreamWriter(oFile)
```

Anschließend lassen sich die Textzeilen mit einer simplen Sequenz an Schreibweisungen in die Datei ablegen:

```
oStreamOut.WriteLine("Angehängte Zeile 1.")
oStreamOut.WriteLine("Angehängte Zeile 2.")
oStreamOut.WriteLine("Angehängte Zeile 3.")
oStreamOut.Close()           ' schließen
oFile.Close()
```

Die letzten beiden Zeilen schließen bereits den *StreamWriter* und den *StreamFile*-Zugriff auf die Datei. An dieser Stelle entfällt das Listing aus Platzgründen, Sie können den Quellcode direkt von der CD laden.

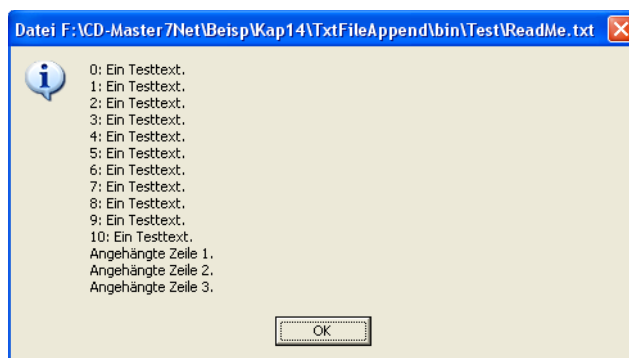


Abbildung 14.8: Anzeige des erweiterten Dateiinhalts

Hinweis

Sie finden die Projektdateien mit der Datei *TxtFileAppend.vb* im Ordner *\Beisp\Kap14\TxtFileAppend* auf der Begleit-CD. Kopieren Sie die ausführbare Programmdatei von der CD-ROM in ein Verzeichnis der Festplatte und legen Sie den Unterordner *\Test* an. Beim ersten Start der Anwendung wird die fehlende Testdatei *ReadMe.txt* automatisch erzeugt.

14.6.5 Textdatei lesen und beschreiben

Im folgenden Beispiel möchte ich noch demonstrieren, wie sich in einem Programm gleichzeitig auf eine Textdatei lesend und schreibend zugreifen lässt.

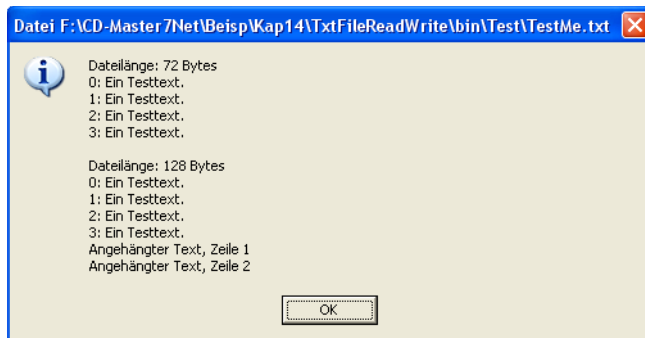


Abbildung 14.9: Anzeige der beiden Dateiinhalte

Das Programm liest den Inhalt einer Testdatei, hängt anschließend einige Zeilen an das Dateiende an und liest den Dateiinhalt erneut. Das Ergebnis beider Vorgänge wird in einem Dialogfeld angezeigt (Abbildung 14.9). Die Anweisungen zum Öffnen der Datei kennen Sie bereits aus den vorherigen Beispielen. Hier der Codeausschnitt der aktuellen Anwendung:

```
Const SubFolder As String = "Test"
Const FileName As String = "TestMe.txt"

Dim txt As String = ""

' Variable für Datei initialisieren
Dim file As String = _
    GetPath() & SubFolder & "\" & FileName
Dim oFile As IO.FileStream ' regelt Dateizugriff
Dim oStreamIn As IO.StreamReader ' regelt Zugriff auf Inhalte
Dim oStreamOut As IO.StreamWriter
```

Die Prozedur *GetPath()* stellt sicher, dass sich die Unterverzeichnisse auf den Ordner mit der Programmdatei beziehen. Zudem habe ich hier zwei *Stream*-Objektvariable für die Ein- und Ausgabe vereinbart. Bevor die eigentlichen Programmfunktionen durchlaufen werden, überprüft das Programm, ob die Testdatei im Unterordner *\Test* vorhanden ist. Fehlen die Elemente, werden der Unterordner und/oder die Testdatei mit folgender Befehlsfolge angelegt:

```
' Ordner \Test anlegen, falls er noch nicht existiert
If Not IO.Directory.Exists(GetPath() & SubFolder) Then _
    IO.Directory.CreateDirectory(GetPath() & SubFolder)
If Not IO.File.Exists(file) Then MakeFile(file)
```

Die Anweisung zum Anlegen eines Ordners wurde bereits weiter oben im Rahmen eines Beispiels vorgestellt. Die Prozedur *MakeFile* stammt aus dem vorhergehenden Beispiel, wurde aber so modifiziert, dass nur noch vier Textzeilen in die neue Datei gespeichert werden. Ist die Testumgebung eingerichtet, öffnet das Programm die angegebene Datei über eine neue *StreamFile*-Instanz:

```
oFile = New IO.FileStream(file, IO.FileMode.Open,  
                          IO.FileAccess.ReadWrite)
```

Hier wird der Dateimodus *FileMode.Open* benutzt, da die Datei zum Lesen und Schreiben geöffnet werden soll. Im dritten Argument findet sich noch der Parameter für den Zugriffsmodus, der hier auf *FileAccess.ReadWrite* gesetzt wird. Um nur zu Lesen (*FileAccess.Read*) oder nur zu Schreiben (*FileAccess.Write*) setzen Sie die hier angegebenen Konstanten ein.

Im nächsten Schritt versucht die Anwendung den Inhalt der bestehenden Datei einzulesen und in einer Variablen *txt* zur späteren Verwendung zu speichern. Hierzu werden die folgenden Anweisungen benutzt:

```
If oFile.CanRead Then ' Lässt sich die Datei lesen?  
    oStreamIn = New IO.StreamReader(oFile) ' Reader anlegen  
Else ' Hier wird eine Ausnahme geworfen ....  
    Throw New Exception ("Panik, kann Datei nicht lesen!!!")  
End if  
  
' Ermittle Dateilänge, hole Dateiinhalt und speichere in txt  
txt = "Dateilänge: " & oFile.Length.ToString & " Bytes" & vbCrLf  
txt = txt & oStreamIn.ReadToEnd() & vbCrLf ' alles lesen
```

Die erste Neuerung besteht in der Verwendung der *CanRead()*-Methode der *FileStream*-Klasse. Diese Methode prüft, ob die Datei zum Lesen geöffnet ist (was hier zwangsweise durch den gesetzten Zugriffsmodus der Fall ist). Lässt sich die Datei im Lesezugriff öffnen, wird eine Instanz der *StreamReader*-Klasse zum Lesen erzeugt und der Objektvariablen *oStreamIn* zugewiesen. Eine ähnliche Programmsequenz führt eine Prüfung über die *CanWrite()*-Methode durch, ob sich die Datei schreiben lässt:

```
If oFile.CanWrite Then  
    oStreamOut = New IO.StreamWriter(oFile) ' Writer anlegen  
Else ' Hier wird eine Ausnahme geworfen ....  
    Throw New Exception ("Panik, kann nicht in die Datei schreiben!!!")  
End if
```

Trifft dies zu, legt das Programm eine neue Instanz des *StreamWriter*-Objekts an und weist dies der Textdatei *oStreamOut* zu.

Eine Neuerung ist die in beiden Sequenzen im *Else*-Zweig der Abfrage benutzte Anweisung *Throw New Exception ("...")*. Mit dem Schlüsselwort *Throw* wird eine Ausnahme

geworfen. Die Anweisung bewirkt, dass das Programm mit einem Laufzeitfehler terminiert. Der Fehler wird dabei in einem Dialogfeld im Klartext angezeigt.

Nach diesen Vorbereitungen ist die Datei geöffnet und es steht je eine *StreamReader*- und eine *StreamWriter*-Instanz zur Verfügung, über die lesend bzw. schreibend auf den Dateiinhalt zugegriffen werden kann. Der Variablen *txt* wird dann im ersten Schritt die Dateilänge über die Eigenschaft *Length* sowie der Dateiinhalt über die Methode *ReadToEnd()* zugewiesen.

In einem weiteren Schritt sollen einige Textzeilen an das Dateieinde angehängt werden. Die Positionierung des Dateizeiger an das Dateieinde mittels der Anweisung

```
oFile.Position = oFile.Length
```

kann an dieser Stelle aber entfallen, da der Lesezeiger nach dem Einlesen am Ende der aktuell geöffneten Datei steht. Die folgenden Anweisungen verwenden die *WriteLine()*-Methode, um einige Textzeilen an die Datei anzuhängen:

```
oStreamOut.WriteLine ("Angehängter Text, Zeile 1")
oStreamOut.WriteLine ("Angehängter Text, Zeile 2")
oStreamOut.Flush()
```

Durch Aufruf der *Flush()*-Methode wird das Auslagern der Dateipuffer in die Datei erzwungen. Um anschließend den Inhalt der geänderten Datei einzulesen und an die Variable *txt* anzuhängen, ist die folgende Codesequenz erforderlich:

```
oFile.Position = 0           ' Zeiger an Anfang
txt = txt & oStreamIn.ReadToEnd() ' alles lesen
```

Die erste Anweisung positioniert den Schreibseizeiger an den Dateianfang. Dann wird einfach der gesamte Inhalt der Datei über die *ReadToEnd()*-Methode des *StreamReader*-Objekts gelesen und der Variablen zugewiesen.

Nach Ausgabe der Ergebnisse sollte das Programm noch die *StreamReader/StreamWriter*-Objekte der geöffneten Datei schließen. Die *Close()*-Methoden der betreffenden Klassen schließen die Streams respektive die geöffnete Datei und geben die zugeordneten Ressourcen frei:

```
oStreamIn.Close()           ' StreamReader schließen
' oStreamOut.Close() wegen Laufzeitfehler auskommentiert
oFile.Close()
```

Hier wurde die *Close()*-Methode der *StreamOut*-Instanz auskommentiert, da die Anweisung einen Laufzeitfehler auslöst. Offenbar kann nur die jeweils zuletzt mit *New* angelegte Instanz von *StreamWriter/StreamReader* mit *Close()* geschlossen werden (bei *BinaryReader* und *BinaryWriter* ist dies anders, siehe auch folgendes Beispiel).

Hinweis

Sie finden das Projekt einschließlich der Datei *TxtFileReadWrite.vb* mit dem gesamten Code des Beispiels im Ordner *\Beisp\Kap14\TxtFileReadWrite* auf der Begleit-CD. Kopieren Sie den Projektordner von der CD-ROM in ein Verzeichnis der Festplatte, laden Sie die *.sln*-Datei in die Entwicklungsumgebung und übersetzen Sie das Projekt. Beim ersten Start erzeugt sich die Anwendung den fehlenden Unterordner *\Test* sowie die Testdatei automatisch.

14.6.6 Lesen und Schreiben binärer Daten

In mit der *StreamFile*-Klasse geöffnete Dateien lassen sich auch Binärdaten schreiben bzw. Sie können aus solchen Dateien Binärdaten lesen. Das Ganze soll an einem einfachen Beispiel demonstriert werden. Ein Programm enthält intern ein *Integer*-Array mit verschiedenen Werten:

```
Dim Werte() As Integer = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Der Inhalt dieses Felds soll in einer Binärdatei gesichert werden. Danach wird der Feldinhalt gelöscht und die Werte für alle Elemente sind aus der Datei zurückzulesen. Die Originalwerte sowie das zurückgelesene Ergebnis ist in einem Dialogfeld anzuzeigen (Abbildung 14.10).

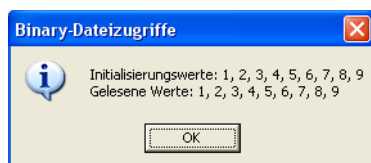


Abbildung 14.10: Anzeige der Originalwerte und gespeicherten Daten

Der Zugriff auf die Datei erfolgt wieder über ein *FileStream*-Objekt. Die Deklarationen der betreffenden Objektvariablen sehen folgendermaßen aus:

```
Dim oFile As IO.FileStream      ' regelt Dateizugriff
Dim oStreamOut As IO.BinaryWriter ' regelt Zugriff auf Inhalte
Dim oStreamIn As IO.BinaryReader
```

Die Deklaration des *FileStream*-Objekts ist identisch mit den Anweisungen der vorherigen Beispiele. Lediglich bei den *Stream*-Objektvariablen kommen die *BinaryWriter*-Klasse bzw. die *BinaryReader*-Klasse zum Einsatz.

Um die Datei zum Schreiben zu öffnen, wird zuerst das *FileStream*-Objekt instantiiert. Dann muss eine Instanz des *BinaryWriter*-Objekts angelegt werden:

```
oFile = New IO.FileStream(file, IO.FileMode.OpenOrCreate)
oStreamOut = New IO.BinaryWriter(oFile) ' Writer anlegen
```


Die Befehle entsprechend weitgehen dem, was Sie bereits auf den vorhergehenden Seiten beim *TextWriter*-Objekt kennen gelernt haben. Einziger Unterschied, die *BinaryWriter*-Klasse bietet leicht modifizierte Methoden zum Schreiben. Hier sehen Sie den Code zum Speichern der Werte und zum Löschen der Feldinhalte:

```
For i = 0 To count
    oStreamOut.Write (Werte(i))      ' Schreibe Wert
    Werte(i) = -1                    ' Element löschen
Next i
```

Die Schleife bewirkt, dass bei jedem Durchlauf ein Element des Feldes gesichert und dann auf den Wert -1 zurückgesetzt wird.

Hinweis

Das *BinaryWriter*-Objekt bietet verschiedene Methoden, mit denen sich unterschiedliche Datentypen sichern lassen. Details finden Sie in der Hilfe zum .NET Framework.

Um anschließend den Inhalt der Datendatei erneut in das Feld einzulesen, muss eine Objektinstanz der *BinaryReader*-Klasse mit der folgenden Anweisung angelegt werden:

```
oStreamIn = New IO.BinaryReader(oFile)
```

Hier verweist die *BinaryReader*-Instanz auf den *FileStream* der noch geöffneten Datei, die über die Objektvariable *oFile* als Argument übergeben wurde. Bevor die Werte gelesen werden können, ist der Positionszeiger über die Eigenschaft *Position* auf den Dateianfang zu setzen:

```
oFile.Position = 0
```

Die nachfolgende Schleife liest dann schrittweise die Werte der Datei als Integerwerte und weist diese den Feldelementen zu:

```
For i = 0 To count
    Werte(i) = oStreamIn.ReadInt32()    ' Lese Wert
Next i
```

Innerhalb der Schleife kommt die *ReadInt32*-Methode zum Einsatz. Die *BinaryReader*-Klasse bietet weitere Varianten der *Read()*-Methode, die unterschiedliche Datentypen zurückliefern. Die verwendete Methode muss auf die in der Datei enthaltenen Daten abgestimmt werden.

Zum Abschluss sind die geöffneten *BinaryReader*-, *BinaryWriter*- und *FileStream*-Instanzen über die *Close()*-Methode zu schließen:

```
oStreamIn.Close()
oStreamOut.Close()
oFile.Close()
```

Das folgende Listing zeigt den Quellcode des kompletten Beispiels. Dieser enthält noch Anweisungen zum Anlegen eines eventuell fehlenden Unterverzeichnisses, zur Ermittlung des Programmpfads sowie zur Anzeige der zu schreibenden und der gelesenen Werte.

```
'*****
' File/Projekt: BinaryReadWrite
' Autor: G. Born www.borncity.de
'
' Zeigt das binäre Lesen und Schreiben in eine Datei.
' Verwendet das FileStream-Objekt mit BinaryReader
' und BinaryWriter.
'*****
Option Strict On

Imports System.IO          ' für StreamFile etc.
Imports System.Reflection  ' für Pfad

Public Class Text
    Shared Sub Main ()
        On Error Goto Fehler ' Fehlerbehandlung

        Const SubFolder As String = "Test"
        Const FileName As String = "Daten.dat"

        Dim Werte() As Integer = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
        Dim i As Integer
        Dim tmp As String = ""

        ' Obergrenze Array-Index
        Dim count As Integer = Werte.GetUpperBound(0) - 1

        ' Variable für Datei initialisieren
        Dim file As String = _
            GetPath() & SubFolder & "\" & FileName
        Dim oFile As IO.FileStream ' regelt Dateizugriff
        Dim oStreamOut As IO.BinaryWriter ' regelt Zugriff auf Inhalte
        Dim oStreamIn As IO.BinaryReader

        ' Unterordner anlegen, falls er noch nicht existiert
        If Not IO.Directory.Exists(GetPath() & SubFolder) Then _
            IO.Directory.CreateDirectory(GetPath() & SubFolder)
```

Listing 14.10: Code zum binären Schreiben/Lesen

```

' Datei öffnen oder anlegen
oFile = New IO.FileStream(file, IO.FileMode.OpenOrCreate)
oStreamOut = New IO.BinaryWriter(oFile) ' Writer anlegen

tmp = "Initialisierungswerte: " & ShowArray(Werte, count)

For i = 0 To count
    oStreamOut.Write (Werte(i))          ' Schreibe Wert
    Werte(i) = -1                        ' Element löschen
Next i

' jetzt versuchen wird das Feld neu einzulesen
oStreamIn = New IO.BinaryReader(oFile) ' Reader anlegen

oFile.Position = 0                      ' an Dateianfang
For i = 0 To count
    Werte(i) = oStreamIn.ReadInt32()    ' Lese Wert
Next i

tmp = tmp & "Gelesene Werte: " & ShowArray(Werte, count)

oStreamIn.Close()                      ' schließen
oStreamOut.Close()
oFile.Close()

MsgBox (tmp, vbOkOnly Or vbInformation, "Binary-Dateizugriffe")

Exit Sub

Fehler: ' Fehler abfangen
MsgBox (err.Description, vbOkOnly Or vbCritical, "Fehler")
End Sub

' ##### Hilfsroutinen
Shared Function GetPath() As String
' Extrahiere den Pfad der laufenden Anwendung
    Dim oMod As System.Reflection.Module = _
        [Assembly].GetExecutingAssembly().GetModules()(0)
    Dim pfad As String

```

Listing 14.10: Code zum binären Schreiben/Lesen (Forts.)

```

        Dim len1 As Integer

        len1 = oMod.Name.Length          ' Länge Dateiname
        pfad = oMod.FullyQualifiedName  ' Pfad mit Dateiname
        pfad = pfad.Remove(pfad.Length - len1, len1)
        Return pfad
    End Function

    Shared Function ShowArray(ByVal wert() As Integer, ByVal count As Integer) As String
    ' Werte des Feldes in eine Zeichenkette schreiben
        Dim i As Integer
        Dim txt As String = ""
        For i = 0 To count
            txt = txt & wert(i).ToString
            If i < count Then txt = txt & ", "      ' Separator
        Next i
        Return txt & vbCrLf
    End Function
End Class

```

Listing 14.10: Code zum binären Schreiben/Lesen (Forts.)

Hinweis

Sie finden das Projekt samt der hier gezeigten Datei *BinaryReadWrite.vb* mit dem gesamten Code des Beispiels im Ordner *\Beisp\Kap14\BinaryReadWrite* auf der Begleit-CD. Kopieren Sie den Projektordner von der CD-ROM in ein Verzeichnis der Festplatte, laden Sie die *.sln*-Datei in der Entwicklungsumgebung und übersetzen Sie das Projekt. Beim ersten Start legt die Anwendung den fehlenden Unterordner samt Datei automatisch an. Der Dateinhalt lässt sich übrigens mit dem nachfolgend vorgestellten Programm *Dump.exe* anzeigen (Abbildung 14.11). Weitere Informationen zu erweiterten Methoden der *FileStream*-Klasse sowie der untergeordneten Klassen zum Lesen und Schreiben finden Sie in der Hilfe zum .NET Framework.

14.6.7 Anzeige von Dateien als Hexadezimaldump

Möchten Sie gelegentlich den Inhalt binärer Dateien ansehen? Die vom vorhergehenden Beispielprogramm geschriebene Datei *Daten.dat* lässt sich zwar im Windows-Editor öffnen, dort werden aber keine vernünftigen Werte angezeigt. Ähnliches gilt für den Inhalt von Bibliotheks- oder Programmdateien. Sobald eine Datei Binärwerte aufweist, benötigen Sie spezielle Werkzeuge, die den Inhalt als Strom von Binärdaten lesen und als Sequenz von Bytefolgen in Hexadezimalnotation ausgeben. Diese sogenannten Dump-Programme werden gelegentlich als Free- oder Shareware angeboten. Mit dem bisherigen Wissen ist es kein Problem mehr, eine solche .NET-Anwendung mit den betreffende Funktion in Visual Basic zu realisieren.

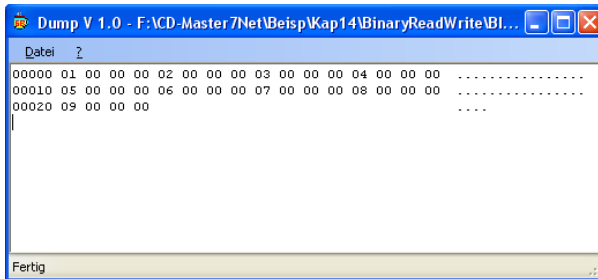


Abbildung 14.11: Anzeige des Dateiinhalts als Hexadezimal-Dump

Das von mir implementierte Beispiel erlaubt dem Benutzer, eine beliebige Datei per Drag&Drop zum Symbol der *.exe*-Programmdatei zu ziehen. Wird die Datei über dem Symbol der Programmdatei losgelassen, startet die Anwendung, ermittelt den Dateinamen, öffnet die Datei und listet dann deren Inhalt als Hexdump in einem Fenster auf (Abbildung 14.11). Wird die Programmdatei dagegen mit einem Doppelklick gestartet, kann der Benutzer eine beliebige Datei über das Menü *Datei/Öffnen* auswählen. Deren Inhalt wird anschließend im Fenster als Hexdump ausgegeben. Die Zahlen in der linken Spalte des Anzeigefensters geben den Offset innerhalb der Datei als Hexadezimalwert an. Daran schließen sich 16 Datenbytes als Hexadezimalzahlen gefolgt von der ANSI-Darstellung der Werte an. Nicht darstellbare Zeichencodes werden durch einen Punkt wiedergegeben. Die Statuszeile des Programmfensters informiert Sie über den Status der Ausgabe, der Dateiname wird dynamisch in der Titelseite des Fensters hinterlegt. Im Menü *Datei* finden Sie zudem den Befehl *Speichern unter*, mit dem Sie den Dump zur Archivierung und zum Drucken als Textdatei sichern können.

Das Programm benutzt die in den Beispielen der vorherigen Kapitel vermittelten Techniken. Die Ansätze zum Anzeigen von Dialogen oder zur Verwaltung von Menüs kennen Sie bereits. Die *Load*-Ereignisprozedur überprüft mittels der *GetCommandLineArgs*-Methode (siehe auch Kapitel 13), ob ein Dateiname als Parameter beim Aufruf übergeben wurde:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    ' Beim Laden des Formulars ggf. Aufrufparameter auswerten
    Dim CmdArgs() As String
    Me.Text = title
    CmdArgs = GetCommandLineArgs() ' lese Argumente
    If CmdArgs.GetUpperBound(0) > 0 Then
        file = CmdArgs(1) ' hole Dateiname
        Me.ToolStripStatusLabel1.Text = "Datei dumpen"
        ShowDump()
    Else
        Me.ToolStripStatusLabel1.Text = "Warte auf Dateiauswahl"
    End If
End Sub
```

Trifft dies zu, wird eine Statusmeldung in der Statusleiste angezeigt und dann die *ShowDump*-Prozedur zur Anzeige des Dumps aufgerufen. Die Ereignisbehandlungsroutine des Befehls *Öffnen* zeigt das Dialogfeld *Öffnen* zur Dateiauswahl, legt den vom Benutzer gewählten Dateinamen in der globalen Variablen *file* ab und startet dann die Anzeige über *ShowDump*.

Die Anweisungen der Prozedur *ShowDump* zum Einlesen der Binärdaten sowie zur Aufbereitung als Hexdump finden Sie in nachfolgendem Listing. Die Prozedur öffnet die angegebene Datei im *BinaryReader*-Modus. Dann wird die Datei in Sequenzen zu je 16 Byte gelesen. Die in der Feldvariablen *Buf()* enthaltenen Bytewerte sind dann über die Hilfsprozedur *ShowHex* in Hexadezimalzahlen mit führenden Nullen zu konvertieren und auszugeben. Zur Pufferung der Textausgabe wird eine lokale Variable *TextBuf* benutzt, deren Inhalt anschließend in das *RichTextBox*-Steuerelement übertragen wird.

```
Private Sub ShowDump()  
    ' Dumpe den Dateiinhalt  
    Dim count, rest As Long  
    Dim i, j As Integer  
    Dim Buf(16) As Byte          ' Hilfspuffer  
    Dim tmp, tmp1 As String  
    Dim oFile As IO.FileStream   ' regelt Dateizugriff  
    Dim oStream As IO.BinaryReader ' regelt Zugriff auf Inhalte  
  
    Me.ToolStripStatusLabel1.Text = "Bitte warten ..."  
    Try  
        If IO.File.Exists(file) Then ' gibt es die Datei?  
            ' Ja -> versuche zu Öffnen und dann zu Lesen  
            oFile = New IO.FileStream(file, IO.FileMode.Open)  
            oStream = New IO.BinaryReader(oFile) ' Stream öffnen  
            Me.Text = title & "- " & file  
            TextBuf = "" ' Lösche Zwischenpuffer  
            ' Datei lesen  
            Do While oStream.BaseStream.Position < oStream.BaseStream.Length  
                tmp1 = " "  
                tmp = MyHex(count, 5)  
                rest = oStream.BaseStream.Length - oStream.BaseStream.Position  
                If rest >= 16 Then  
                    Buf = oStream.ReadBytes(16)  
                    For i = 0 To 15  
                        ' lese Byte  
                        tmp = tmp & MyHex(Buf(i), 2) ' in Hex  
                        If Buf(i) > &H1F Then  
                            tmp1 = tmp1 & Chr(Buf(i))  
                        Else
```

```

        tmp1 = tmp1 & "."
    End If
Next i
TextBuf = TextBuf & tmp & tmp1 & vbCrLf
count += 16
Else ' Beackere die Restbytes
    Buf = oStream.ReadBytes(CInt(rest))
    For i = 0 To CInt(rest) - 1
        ' lese Byte
        tmp = tmp & MyHex(Buf(i), 2) ' in Hex
        If Buf(i) > &H1F Then
            tmp1 = tmp1 & Chr(Buf(i))
        Else
            tmp1 = tmp1 & "."
        End If
    Next i

    For i = CInt(rest) To 15 ' mit Blanks auffüllen
        ' lese Byte
        tmp = tmp & " " ' Restzeile füllen
    Next i
    TextBuf = TextBuf & tmp & tmp1 & vbCrLf
    Exit Do ' fertig
End If
Loop
Me.RichTextBox1.Text = TextBuf ' in Steuerelement
Me. ToolStripStatusLabel1.Text = "Fertig"
oStream.Close() ' Schließen
oFile.Close() ' "
Else
    Me. ToolStripStatusLabel1.Text = "Fehler: Datei " & file & " nicht gefunden"
End If
Catch ex As Exception
    MessageBox.Show(ex.Message, "Fehler", _
        MessageBoxButtons.OK, MessageBoxIcon.Error)
    Me. ToolStripStatusLabel1.Text = "Warte auf Dateiauswahl"
End Try
End Sub

```

Hinweis

An dieser Stelle noch einige Hinweise zur Übernahme der Dump-Ergebnisse in das *RichTextBox*-Steuerelement. Der Ansatz, die Teilergebnisse mit einer Anweisung der Art *Me.RichTextBox.Text = Me.RichTextBox.Text & tmp & tmp1 & vbCrLf* zu übernehmen, spart zwar die Puffervariable *TextBuf*. Die bei diesem Ansatz erforderlich werdenden Kopieroperationen kosten aber sehr viel Laufzeit, wodurch die Anwendung selbst zum Laden sehr kleiner Dateien viel Zeit benötigt. Möchten Sie den Speicherplatzverbrauch durch die Variable *TextBuf* vermeiden, hängen Sie die Teilergebnisse mittels der *AppendText()*-Methode des *RichTextBox*-Steuerelements an: *Me.RichTextBox.AppendText (tmp & tmp1 & vbCrLf)*. Dies hat den zweiten Vorteil, dass der Benutzer sofort sieht, wie die Teilergebnisse als Dump in der Anzeige einlaufen. Bei umfangreichen Dateien benötigt das Programm trotzdem recht lange zur Anzeige. Sie können dies optimieren, indem Sie bei der Ausgabe nur eine »Seite« füllen und dem Benutzer eine Funktion zum Scrollen nach unten bieten. Bei jedem Scroll-Vorgang wird einfach eine weitere Seite gelesen und angezeigt. Der Lesezugriff lässt sich auch optimieren, indem Sie mehr als 16 Bytes in den Puffer einlesen.

Zum Schreiben der Textdatei öffnet die Prozedur des betreffenden Ereignishandlers einen *Speichern unter*-Dialog, liest den Dateinamen und benutzt die *WriteFile()*-Methode des *RichTextBox*-Steuerelements zum Sichern im Textformat:

```
RichTextBox1.SaveFile(sFile, RichTextBoxStreamType.PlainText)
```

Dies stellt sicher, dass alle Informationen aus der Anzeige in der Datei abgelegt werden. Verwenden Sie dagegen die *StreamWriter()*-Methode zum Sichern, werden (zumindest habe ich dies festgestellt) die Zeilenumbrüche (Code 0AH 0DH) durch das Zeichen 0AH ersetzt. Der Windows-Editor kann solche Dateien nicht mehr korrekt anzeigen. Weitere Details sind dem Quellcode des Beispiels zu entnehmen.

Hinweis

Das Projekt findet sich im Ordner *\Beisp\Kap14\Dump* auf der Begleit-CD. Laden Sie die *.sln*-Datei in die Entwicklungsumgebung und übersetzen Sie das Projekt. Danach lässt sich Dump auf beliebige Dateien anwenden. Einzige Voraussetzung ist lediglich, dass die anzuzeigende Datei nicht im Zugriff durch Windows oder eine Anwendung ist.

14.7 Nutzen der FileSystemWatcher-Klasse

Über Objektinstanzen der *FileSystemWatcher*-Klasse können Änderungen im Dateisystem überwacht werden. Ein *FileSystemWatcher* löst ein Ereignis aus, wenn eine Datei oder ein Ordner angelegt, gelöscht, umbenannt oder geändert wurde. Nachfolgend wird an einem Beispiel skizziert, wie sich die *FileSystemWatcher*-Klasse für diesen Zweck einsetzen lässt.

14.7.1 Beispiel: Überwachen eines Ordners auf Textdateien

In einem einfachen Beispiel soll der Einsatz der *FileSystemWatcher*-Klasse demonstriert werden. Die .NET-Anwendung ist als Konsolenapplikation implementiert und zeigt beim Start den *BrowserFolderDialog*-Dialog an. Der Benutzer kann über diesen Dialog den zu überwachenden Ordner auswählen oder neu anlegen. Sobald der *FileSystemWatcher* den zu überwachenden Ordner kennt und aktiv ist, erscheint eine Textmeldung im Fenster der Eingabeaufforderung. Der Benutzer kann anschließend den gewählten Ordner unter Windows öffnen und dort Textdateien anlegen, umbenennen, löschen oder deren Inhalt ändern. Jede dieser Aktionen veranlasst eine entsprechende Meldung im Fenster der Eingabeaufforderung (Abbildung 14.12).

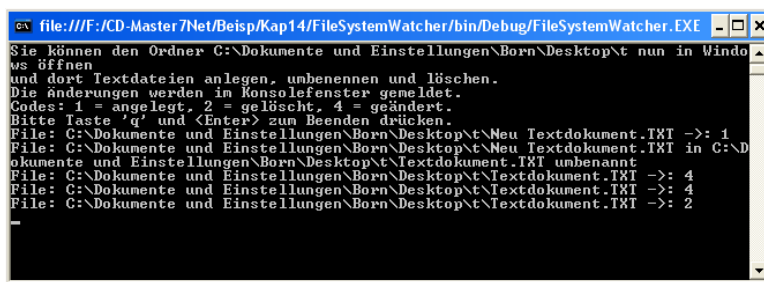


Abbildung 14.12: Meldungen, als Reaktion auf Ereignisse des *FileSystemWatcher*

14.7.2 Hinweise zur Implementieren des Beispiels

Eine Instanz der *FileSystemWatcher*-Klasse lässt sich über die Anweisung

```
Dim watcher As New System.IO.FileSystemWatcher()
```

erzeugen. Über die Eigenschaften des betreffenden Objekts lässt sich dann der *FileSystemWatcher* konfigurieren. Der zu überwachende Ordner wird beispielsweise über die Eigenschaft *Path* zugewiesen. Um die Überwachung auf spezielle Dateitypen zu begrenzen, lässt sich ein entsprechender Filterwert in der Eigenschaft *Filter* hinterlegen. Das *FileSystemWatcher*-Objekt löst Ereignisse für jede Änderung im überwachten Ordner aus. Sie können die zu überwachenden Ereignisse aber zusätzlich über die Eigenschaft *NotifyFilter* einschränken, indem Sie dieser verschiedene Konstanten zuweisen. Das folgende Codefragment zeigt, wie sich die Eigenschaften setzen lassen:

```
watcher.Path = "C:\Test\" ' zu überwachender Ordner
watcher.Filter = "*.txt" ' Begrenze auf Textdateien
' Überwache: LastAccess, LastWrite und Umbenennen
watcher.NotifyFilter = NotifyFilters.LastAccess Or _
    NotifyFilters.LastWrite Or _
    NotifyFilters.FileName Or _
    NotifyFilters.DirectoryName)
watcher.EnableRaisingEvents = True ' Überwachung einschalten
```

Über die Eigenschaft *EnableRaisingEvents* wird gesteuert, ob die Ereignisse weitergeleitet werden oder nicht. Für die verschiedenen Ereignisse (*Changed*, *Created*, *Deleted* und *Renamed*) lässt sich dann eine Ereignisbehandlungsroutine mittels der *AddHandler*-Anweisung registrieren. Die Ereignisse *Changed*, *Created* und *Deleted* benutzen dabei die gleiche Aufrufstruktur, können also auf eine Ereignisbehandlungsroutine geleitet werden. Für das *Rename*-Ereignis ist aber eine separate Ereignisbehandlungsroutine erforderlich. Die folgende Codesequenz registriert die Handler für die erwähnten Ereignisse:

```
' Ereignishandler registrieren
AddHandler watcher.Changed, AddressOf OnChanged
AddHandler watcher.Created, AddressOf OnChanged
AddHandler watcher.Deleted, AddressOf OnChanged
AddHandler watcher.Renamed, AddressOf OnRenamed
```

Die beiden Ereignisbehandlungsroutinen *OnChanged* und *OnRename* besitzen folgende Prozedurrümpfe:

```
Private Shared Sub OnChanged(ByVal source As Object, _
                             ByVal e As FileSystemEventArgs)

End Sub

Private Shared Sub OnRenamed(ByVal source As Object, _
                              ByVal e As RenamedEventArgs)

End Sub
```

Im ersten Argument wird der Name samt Pfad des betreffenden Dateisystemobjekts übergeben. Beim *Changed*-Ereignis liefert das zweite Argument einen Code mit dem Hinweis auf die Änderung (z.B. 1 = Element erzeugt, 2 = Element gelöscht, 4 = Element geändert). Je nach ausgeführter Operation können aber mehrere Ereignisse auftreten. Nachfolgendes Listing zeigt den Quellcode des kompletten Beispiels, welches Anweisungen zur Anzeige des *BrowserFolderDialog*-Dialogs sowie zur Ausgabe von Benutzermeldungen im Fenster der Eingabeaufforderung enthält:

```
'*****
' File/Projekt: FileSystemWatcher
' Autor: G. Born www.borncity.de
'
' Demonstriert die Verwendung eines FileSystemWatchers.
' Der Benutzer kann über ein Dialogfeld einen Ordner
' öffnen oder neu anlegen. Danach lässt sich der Ordner
' unter Windows öffnen. Sobald dort Textdateien angelegt,
' geändert, umbenannt oder gelöscht werden, wird dies im
```

Listing 14.11: Code des FileSystemWatcher-Beispiels

```

' Fenster der Eingabeaufforderung gemeldet.
'*****
Option Strict On

Imports System.IO
Imports System.Windows.Forms

Class Test
    Public Shared Sub Main()
        Run()
    End Sub

    Private Shared Sub Run()
        Dim folder As String = ""
        Dim oDlg As New System.Windows.Forms.FolderBrowserDialog
        ' Ordner auswählen mit FolderBrowserDialog
        With oDlg
            ' hier können die Eigenschaften gesetzt werden
            .Description = "Ordner auswählen" ' Dialogfeld-Titel
            .ShowNewFolderButton = True       ' neue Ordner anlegbar
            ' voreingestellter Ordner
            .RootFolder = System.Environment.SpecialFolder.Desktop
            ' Dialog anzeigen
            If .ShowDialog() = Windows.Forms.DialogResult.OK Then
                folder = .SelectedPath ' Laufwerksbuchstabe
            End If
        End With

        If folder = "" Then Exit Sub ' Beenden, da kein Pfad

        ' Jetzt FileSystemWatcher neu aufsetzen
        Dim watcher As New System.IO.FileSystemWatcher()
        watcher.Path = folder
        ' Überwache: LastAccess, LastWrite und Umbenennen
        watcher.NotifyFilter = (NotifyFilters.LastAccess Or _
                                NotifyFilters.LastWrite Or _
                                NotifyFilters.FileName Or _
                                NotifyFilters.DirectoryName)
        ' Begrenze auf Textdateien
        watcher.Filter = "*.txt"
    End Sub
End Class

```

Listing 14.11: Code des FileSystemWatcher-Beispiels (Forts.)

```

' Ereignishandler registrieren
AddHandler watcher.Changed, AddressOf OnChanged
AddHandler watcher.Created, AddressOf OnChanged
AddHandler watcher.Deleted, AddressOf OnChanged
AddHandler watcher.Renamed, AddressOf OnRenamed

' Überwachung einschalten
watcher.EnableRaisingEvents = True

' Hier warten wir, bis der Benutzer die Konsole beendet
Console.WriteLine("Sie können den Ordner {0}" & _
    " nun in Windows öffnen", folder)
Console.WriteLine("und dort Textdateien anlegen," & _
    " umbenennen und löschen.")
Console.WriteLine("Die Änderungen werden im Konsolenfenster" _
    & " gemeldet.")
Console.WriteLine("Codes: 1 = angelegt, 2 = gelöscht, " & _
    "4 = geändert.")
Console.WriteLine("Bitte Taste 'q' und <Enter> zum " & _
    "Beenden drücken.")
While Chr(Console.Read()) <> "q"c
End While
watcher.Dispose() ' Objektinstanz freigeben
End Sub

' Definiere die Event handler
Private Shared Sub OnChanged(ByVal source As Object, _
    ByVal e As FileSystemEventArgs)
' Beim Ändern, Anlegen, Löschen von Dateien
Console.WriteLine("File: " & e.FullPath & " ->: " & e.ChangeType)
End Sub

Private Shared Sub OnRenamed(ByVal source As Object, _
    ByVal e As RenamedEventArgs)
' Beim Umbenennen einer Datei
Console.WriteLine("File: {0} in {1} umbenannt", _
    e.OldFullPath, e.FullPath)
End Sub
End Class

```

Listing 14.11: Code des FileSystemWatcher-Beispiels (Forts.)

Hinweis

Sie finden die Projektdateien dieses als Konsoleanwendung implementierten Beispiels im Ordner `\Beisp\Kap14\FileSystemWatcher` der Begleit-CD. Laden Sie die `.sln`-Datei in der Entwicklungsumgebung, lässt sich das Beispiel ausführen. Das Beispiel im Ordner `\Beisp\Kap14\FileSystemWatcher1` der Begleit-CD ist als Windows-Anwendung implementiert. Hier wurde ein `FileSystemWatcher`-Steuerelement dem Formular hinzugefügt. Über die Schaltfläche *Ordner wählen* lässt sich der Dialog zur Auswahl bzw. zum Anlegen des Ordners aufrufen. Die Schaltfläche *Ordner ändern* erzeugt beim ersten Aufruf eine Textdatei im Zielordner. Ist die Zieldatei bereits vorhanden, wird deren Inhalt geändert. Diese beiden Ereignisse werden im Textfeld des Anwendungsfensters gemeldet. Details sind dem Quellcode des Beispiels zu entnehmen. Weitere Informationen zur `FileSystemWatcher`-Klasse finden Sie in der Hilfe zum .NET Framework.

14.8 Zugriff auf die EventLog-Dateien

Windows verwaltet intern Protokolldateien (Event-Logs), in die Anwendungen oder das System Meldungen bei bestimmten Ereignissen eintragen können. Administratoren können diese Protokolleinträge über die Ereignisanzeige inspizieren. Nachfolgend wird gezeigt, wie aus .NET-Anwendungen auf die Ereignisprotokollierung schreibend und lesend zugegriffen werden kann.

14.8.1 Bemerkungen zur Ereignisprotokollierung

Ein Benutzer kann über das Symbol *Ereignisanzeige* die vom System verwalteten Protokolle abrufen und analysieren. In Windows XP lässt sich beispielsweise in der Systemsteuerung das Symbol *Verwaltung* anwählen. Der dann geöffnete Ordner *Verwaltung* enthält ein Symbol zum Aufrufen der Ereignisanzeige. Die Ereignisse werden dabei standardmäßig in die Kategorien *Anwendung*, *Sicherheit* und *System* unterteilt und vom System in eigenen Protokollen verwaltet. Nach Anwahl einer Ereigniskategorie listet die Ereignisanzeige die im Protokoll hinterlegten Einträge auf (Abbildung 14.13).

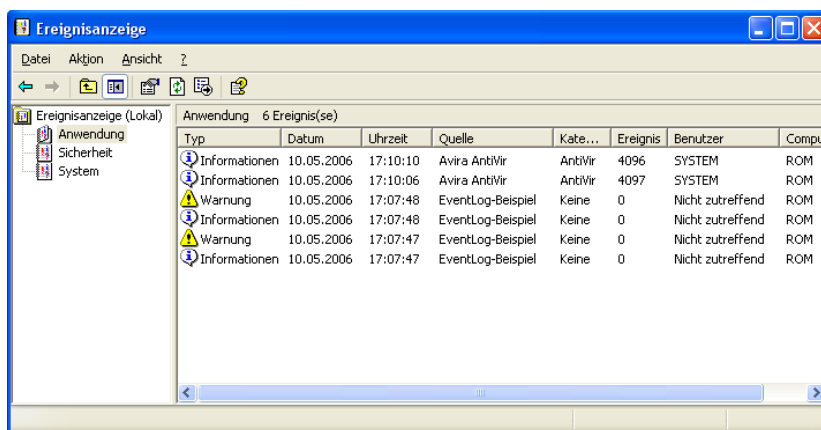


Abbildung 14.13: Ereignisanzeige mit Protokollen und Einträgen

Jedem Eintrag kann ein Typ (z.B. *Warnung*, *Information*, *Fehler*) zugeordnet werden. Abhängig vom Typ wird dann ein Symbol und der Klartext in der betreffenden Spalte eingeblendet. In der Rubrik *Quelle* wird der Name der Anwendung gezeigt, die den Eintrag im Ereignisprotokoll hinterlegt hat. Das System protokolliert automatisch Datum und Uhrzeit, den zugehörigen Computer und optional auch den Benutzer sowie einen Ereigniscode mit.

14.8.2 Das EventLog-Element verwenden

Die Klasse *EventLog* bietet eine Reihe von Membern rund um die Ereignisprotokollierung. Um eine Instanz dieser Klasse zu erzeugen, haben Sie verschiedene Möglichkeiten. Sie können beispielsweise eine Instanz direkt im Quellcode erzeugen und dann das Objekt zum Zugriff auf die Ereignisprotokollierung nutzen.

```
Dim myLog As New EventLog()' Instanz von EventLog erzeugen
myLog.Source = "MyBorn"    ' Zuordnung zur Quelle
```

Die erste Zeile erzeugt eine neue Instanz des *EventLog*-Objekts, der dann in der zweiten Anweisungszeile über die Eigenschaft *Source* ein Wert für die Quelle zugewiesen wird. Bei Windows-Anwendungen können Sie im Ansicht-Designer auch ein *EventLog*-Steuerelement aus der Toolbox zum Formular hinzufügen. Das Steuerelement wird am unteren Rand des Designers eingeblendet. Dann haben Sie die Möglichkeit, die Eigenschaften des Steuerelements interaktiv im Eigenschaftfenster der Entwicklungsumgebung zu setzen (Abbildung 14.14).

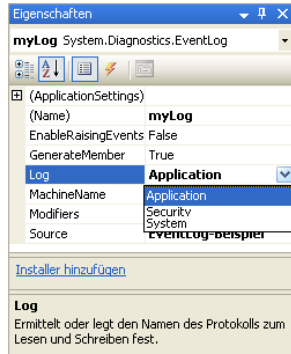


Abbildung 14.14: Eigenschaften des *EventLog*-Steuerelements

In Abbildung 14.14 wurde die Eigenschaft *Source* auf »EventLog-Beispiel« gesetzt, während sich über die Eigenschaft *Log* der Name des Protokolls, in welches Ereignismeldungen zu hinterlegen sind, wählen lässt.

So können Sie eine Quelle unter einer eigenen Kategorie registrieren

Die Quelle muss der Ereignisprotokollierung bekannt sein. Die Registrierung wird ggf. beim ersten Aufruf einer Methode zum Schreiben in die Protokolldateien automatisch durchgeführt. Sie können aber prüfen, ob die Quelle bereits registriert ist. Zudem lässt sich der Quelle ein eigenes Protokoll zuweisen. Dies ist mit folgenden Anweisungen möglich:

```
' Prüfe, ob Quelle "MyBorn" vorhanden, falls nein, anmelden
If Not System.Diagnostics.EventLog.SourceExists("MyBorn") Then _
    System.Diagnostics.EventLog.CreateEventSource("MyBorn", _
        "MyTestLog")
```

Über die Methode *SourceExists()* wird geprüft, ob die betreffende Quelle bereits registriert ist. Liefert die Methode den Wert *false*, wird die Quelle in der zweiten Anweisungszeile mittels der Methode *CreateEventSource()* einmalig registriert. Im ersten Parameter erwartet die Methode dabei den Namen der Quelle (hier »MyBorn«). Typischerweise wird man als Quelle den Namen der Anwendung verwenden. Der zweite Parameter der *CreateEventSource()*-Methode erlaubt die Protokollkategorie anzugeben. Im aktuellen Code wird ein neues Protokoll *MyTestLog* angelegt. Diese Kategorie taucht später in der Ereignisanzeige in der linken Spalte auf.

Hinweis

Standardmäßig protokollieren Anwendungen Ereignismeldungen in der Kategorie *Anwendung* (intern unter dem Namen *Application* geführt). Die *Log*-Eigenschaft im Eigenschaftenfenster listet daher standardmäßig nur die in Abbildung 14.14 sichtbaren Protokolle auf. Haben Sie eine eigene Protokollkategorie über die *CreateEventSource()*-Methode angelegt, taucht deren Name ebenfalls im Listenfeld *Log* im Eigenschaftenfenster des Steuerelements auf. Sie können dann diese Protokollkategorie dem betreffenden *EventLog*-Steuerelement zuweisen.

Die Windows-Ereignisanzeige bietet leider keine Möglichkeit, um eine Protokollkategorie zu löschen – Sie können nur den Inhalt der aktuell gewählten Protokolldatei löschen. Falls Sie beim Experimentieren mit dem *EventLog*-Steuerelement unbeabsichtigt eine neue Protokollkategorie angelegt haben, verbleibt diese quasi als »Leiche« im System. Um das System zu bereinigen, lässt sich die neu angelegte Protokollkategorie aber mit einem Eingriff in die Registrierung entfernen. Starten Sie die Registrierungseditor *Regedit.exe* und wählen Sie den Schlüssel *HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\Eventlog*. Dort finden Sie neben den standardmäßig vorhandenen Unterschlüsseln *Application*, *Security* und *System* auch einen weiteren Unterschlüssel mit dem Namen der neuen Protokollkategorie. Löschen Sie den Unterschlüssel der neuen Protokollkategorie (z.B. den Schlüssel *MyTestLog*).

Möchten Sie den Eintrag für eine in der Ereignisanzeige registrierte Quelle entfernen (z.B. weil sich der Name der Quelle geändert hat)? Eine Quelle wird bei der erstmaligen Registrierung unter dem betreffenden Namen als Unterschlüssel im Schlüssel der Protokollkategorie hinterlegt (z.B. *Application\MyBorn*). Zum Entfernen der registrierten Quelle müssen Sie den Unterschlüssel (z.B. *MyBorn*) im Schlüssel der Protokollkategorie suchen und löschen. Die *.evt*-Protokolldateien werden standardmäßig im Windows-Unterordner *\system32\config* hinterlegt. Möchten Sie die Protokolldatei der entfernten Kategorie aus diesem Unterordner löschen? Dann führen Sie einen Neustart des Systems aus, damit dieses die betreffende Protokolldatei freigibt. Anschließend lässt sich die *.evt*-Protokolldatei unter einem Administratorenkonto löschen.

14.8.3 Schreiben in das Ereignisprotokoll

Liegt eine Instanz der *EventLog*-Klasse als Objekt vor, lässt sich über dessen Methoden direkt in das Ereignisprotokoll schreiben. Die folgende Codesequenz setzt die Eigenschaften *Source* und *Log* des Objekts und schreibt dann einen Eintrag unter Verwendung der *WriteEntry()*-Methode in die Protokolldatei.

```
myLog.Source = "EventLog-Beispiel"  
myLog.Log = "Application"  
  
myLog.WriteEntry("Born: 1. Benutzer-Eintrag ", _  
                EventLogEntryType.Information)
```

Die Methode *WriteEntry()* erwartet im ersten Argument den zu schreibenden Text, während optional im zweiten Parameter noch eine *EventLogEntryType*-Konstante für den Typ des Ereignisses (z.B. *Information*, *Warning* etc.) übergeben werden darf. Der protokollierte Texteintrag lässt sich später in der Windows-Ereignisanzeige abrufen, indem Sie den Eintrag per Doppelklick anwählen.

Hinweis

Hinweise zu weiteren Varianten beim Aufruf der *WriteEntry()*-Methode finden Sie in der Hilfe zum .NET Framework. Um alle Einträge eines Ereignisprotokolls zu löschen, lässt sich die Methode *Clear()* des *EventLog*-Objekts aufrufen.

14.8.4 Lesen des Ereignisprotokolls

Über Instanzen der *EventLog*-Klasse lassen sich die Einträge des Ereignisprotokolls auch lesen. Die folgende Codesequenz demonstriert diesen Ansatz:

```
Dim entry As EventLogEntry  
myLog.Source = "EventLog-Beispiel"  
myLog.Log = "Application"  
For Each entry In myLog.Entries ' durchlaufe Ereignisprotokoll  
    If entry.Source = myLog.Source Then ' filtere Quelle  
        txt = entry.Source & _  
              entry.TimeGenerated & _  
              entry.Message & vbCrLf  
    End If  
Next
```

In einer Schleife werden alle Einträge der *Entries*-Auflistung des *EventLog*-Objekts durchlaufen. Welches Protokoll auszuwerten ist, legt die *Log*-Eigenschaft dieses Objekts fest. Die Schleifenvariable *entry* ist als *EventLogEntry* definiert und erlaubt direkt den Zugriff auf die Eigenschaften des jeweiligen Protokolleintrags. In der obigen Codesequenz filtert eine *If*-Anweisung nur solche Einträge, deren Eigenschaft *Source* des Pro-

tokolleintrags mit der gleichnamigen Eigenschaft des *EventLog*-Objekts übereinstimmen. Der Protokolltext lässt sich über die Eigenschaft *Message* abrufen, während *TimeGenerated* den Protokollzeitpunkt angibt.

14.8.5 Hinweise zur Implementieren eines Beispiels

Zur Demonstration des schreibenden und lesenden Zugriffs auf die Ereignisprotokollierung wird ein einfaches Beispiel als Windows-Anwendung implementiert. Die Anwendung zeigt ein Formular mit einem Textfeld und vier Schaltflächen (Abbildung 14.15). Über die Schaltfläche *Ereignis schreiben* werden bei jedem Mausklick jeweils zwei Einträge im Protokoll *Application* unter dem Namen der Quelle *EventLog-Beispiel* hinterlegt. Die hierzu benötigten Eigenschaften *Log* und *Source* wurden über das Eigenschaftsfenster des *EventLog*-Steuerelements *myLog* definiert – die Eigenschaften müssen also im Code nicht mehr gesetzt werden. Sobald die Einträge geschrieben wurden, meldet die Anwendung dies durch eine kurze Nachricht im Textfeld des Formulars. Die Schaltfläche *Ereignisse auflisten* bewirkt, dass in der zugehörigen *Click*-Ereignisbehandlungsroutine das Protokoll *Application* der Ereignisanzeige ausgelesen wird. Anschließend filtert das Programm alle Anwendungsereignisse der Quelle »EventLog-Beispiel« heraus und zeigt deren Eigenschaften *Source*, *TimeGenerated* und *Message* im Textfeld. Die Schaltfläche *Protok. löschen* löscht, nachdem der Benutzer eine Sicherheitsabfrage bestätigt hat, die Einträge der *Application*-Protokolldatei der Ereignisanzeige.

Hinweis

Sie finden die Projektdateien dieses Beispiels im Ordner `\Beisp\Kap14\EventLog` der Begleit-CD. Laden Sie die *.sln*-Datei in der Entwicklungsumgebung und führen Sie das Beispiel aus. Weitere Informationen zu den Membern der *EventLog*-Klasse finden Sie in der Hilfe zum .NET Framework.

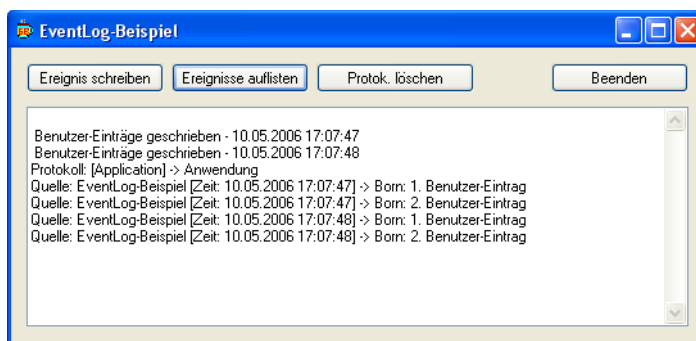


Abbildung 14.15: Beispiel zur Nutzung des *EventLog*-Steuerelements

Mit diesem Beispiel möchte ich das Kapitel zum Thema Dateizugriffe schließen. Sie haben jetzt einen Überblick über die wichtigsten Ansätze zur Handhabung von Dateioperationen erhalten. Zudem beherrschen Sie die Grundlagen, um auf Dateien oder auf die Ereignisprotokolle lesend und schreibend zuzugreifen.

15

Datenbankfunktionen nutzen

In diesem Kapitel erhalten Sie eine Einführung in die Nutzung der Datenbankfunktionen von ADO.NET. Sie lernen, wie sich SQL-Datenbanken anlegen und verwalten lassen und wie Sie aus einer .NET-Anwendung auf solche Datenbanken bzw. andere Datenquellen zugreifen können.

15.1 ADO.NET-Grundlagen

Zum Zugriff auf Datenbanken oder andere Datenquellen (z.B. Excel-Dateien, XML-Dokumente etc.) stellt Microsoft ADO.NET bereit. Bei ADO.NET handelt es sich um mehrere Klassen, die Methoden zum Zugriff auf externe Daten bieten. ADO.NET fungiert dabei als Zwischenschicht, die über austauschbare Datenanbieter (Provider) den konsistenten Zugriff auf verschiedene Datenquellen erlaubt (Abbildung 15.1).

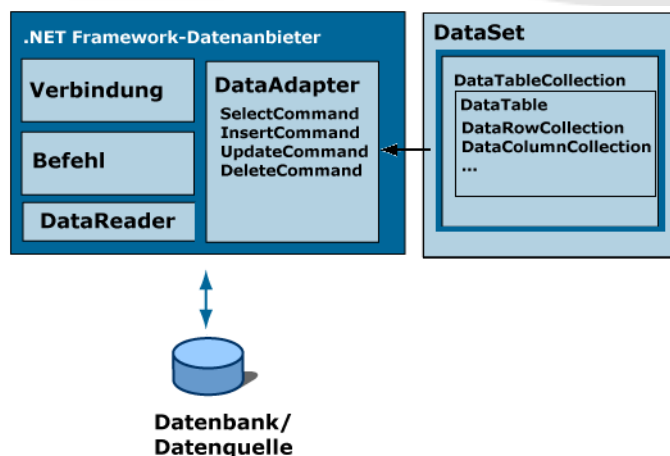


Abbildung 15.1: Zugriff auf Daten über ADO.NET

Als Datenquellen sind im Microsoft SQL-Server verwaltete Datenbanken denkbar. Über OLE DB- und ODBC-Datenanbieter (Provider) sind aber auch Zugriffe auf weitere Datenquellen (z.B. Access-Datenbanken, XML-Dateien, Excel-Arbeitsmappen oder strukturierte Textdateien im CSV-Format) möglich. Zum Zugriff auf Datenquellen stehen zwei Komponenten zur Verfügung:

- *.NET-Framework-Datenanbieter*: Es handelt sich dabei um Komponenten, die einen direkten Zugriff auf die Daten externer Datenquellen erlauben. Ein *Connection*-Objekt stellt die Verbindung zur Datenquelle her. Über das *Command*-Objekt lassen sich

Befehle an die Datenquelle senden, um Daten anzufordern, zurückzugeben, zu ändern oder gespeicherte Prozeduren in Datenbanken aufzurufen. Das *DataReader*-Objekt stellt einen *Stream* von Daten der Datenquelle bereit und das *DataAdapter*-Objekt bildet die Brücke zwischen *DataSet*-Objekt und Datenquelle.

- *DataSet*: Die Instanzen der *DataSet*-Klasse stellen die Daten, unabhängig von der Datenquelle, im Arbeitsspeicher bereit. Das *DataSet*-Objekt enthält eine Auflistung von (mindestens einem, meist aber mehreren) *DataTable*-Objekten, über die sich die Daten lesen und schreiben lassen. Jedes *DataTable*-Objekt besteht aus (Daten-) Zeilen und Spalten (ist also eine Tabelle). Die Datenzeilen repräsentieren die Datensätze, während die Spalten die Felder eines aus der Datenquelle extrahierten Datensatzes darstellen. Felder können dabei Schlüssel (Indizes) umfassen.

Über die einzelnen Klasseninstanzen kann eine .NET-Anwendung also auf deren Methoden zugreifen, um eine Verbindung zu einer Datenquelle aufzunehmen und Daten anzufordern. Der eigentliche Verbindungsteil zwischen ADO.NET und der Datenquelle wird dann durch die Datenanbieter (Provider) realisiert. Durch Wechsel des Providers lässt sich ohne Änderung der Programmlogik auf unterschiedliche Datenquellen zugreifen. Lediglich die Verbindungseinstellungen zur Kontaktaufnahme mit der Datenquelle sind anzupassen.

15.1.1 Wann werden DataReader und DataSet verwandt?

Zum Abrufen von Daten aus der Datenquelle müssen Sie sich also zwischen der Verwendung eines *DataSet* oder eines *DataReader* entscheiden. Ein *DataSet* sollten Sie in folgenden Fällen verwenden:

- Um lokale Dateien der Anwendung zwischenspeichern und zu bearbeiten. Das *DataSet*-Objekt fungiert quasi wie ein Cache im Arbeitsspeicher.
- Beim Ausführen umfangreicher Datenverarbeitungsschritte, die längere Zeit benötigen. Während der Bearbeitung der Daten wird die Verbindung zur Datenquelle freigegeben, so dass andere Clients auf die Datenquelle zugreifen können.

Zudem ist ein *DataSet* bei einer Datenanbindung an Steuerelemente innerhalb eines Formulars oder beim Erstellen einer Beziehung zwischen Daten aus mehreren Quellen ganz hilfreich. Benötigen Sie lediglich Daten von der Datenquelle (lesender Zugriff), steht mit dem *DataReader* eine leistungsfähigere Alternative zur Verfügung. Setzen Sie den *DataReader* z.B. ein, wenn Sie nur Daten (z.B. einer Abfrage) zur Anzeige benötigen.

15.1.2 Wie werden die Klassen in Anwendungen genutzt?

Zum Zugriff auf ADO.NET-Funktionen müssen Sie entsprechende Instanzen der Klassen in der Anwendung erzeugen. Anschließend können über die betreffenden Objekte Zugriffe auf die von der jeweiligen Klasseninstanz bereitgestellten Methoden erfolgen. Dabei haben Sie zwei Möglichkeiten:

- Sie können, wie auf den folgenden Seiten dieses Kapitels skizziert, den *DataSet*-Designer und die Assistenten zur Konfigurierung von Datenquellen verwenden, um Funktionen zur Anzeige von Datenbanktabellen in Formularen zu hinterlegen.

- Die Alternative besteht darin, die Funktionen zum Zugriff auf Datenbanktabellen direkt im Programmcode unter Verwendung von Instanzen der ADO.NET-Klassen zu realisieren.

Der DataSet-Designer erlaubt zwar mit einigen Mausklicks die benötigten Formulare zu erstellen und den Zugriff auf Datenquellen einzurichten. Der DataSet-Designer fügt die benötigten Steuerelemente (DataSet, TableAdapter, BindingSource etc.) zum Formular hinzu und generiert auch den Programmcode zur Verbindung der betreffenden Objekte. Der Ansatz hat aber aus meiner Sicht den Nachteil, dass die Funktionsweise recht untransparent ist (es ist manchmal recht aufwändig, die zum Zugriff auf die Daten erforderlichen Einstellungen in den verschiedenen Eigenschaftsfenstern aufzuspüren). Zudem musste ich feststellen, dass in den Designern zum Erzeugen des Codes noch einige Fehler enthalten sind. Es kommt immer wieder vor, dass die Entwicklungsumgebung beim Übersetzen eines Projekts oder bei der Anzeige von Formularen Fehler meldet, die nach dem Beenden der Umgebung und erneutem Laden des Projekts auf wundersame Weise verschwunden sind. Entscheiden Sie sich für den direkten Zugriff auf die ADO.NET-Klassen, müssen Sie die betreffenden Klasseninstanzen direkt im Programmcode quasi »zu Fuß« über entsprechende Visual Basic-Anweisungen erstellen und auch für die Implementierung der Programmlogik sorgen. Oft ist dies aber der aus Sicht des Programmierers transparentere Ansatz. In den nachfolgenden Abschnitten skizziere ich beide Ansätze.

Hinweis

Um aus Visual Basic auf die ADO.NET-Klassen zugreifen zu können, muss der Namensraum *System.Data* mit einer *Imports*-Anweisung im Projekt eingebunden werden. Dies wird automatisch durch die Entwicklungsumgebung eingestellt, sobald ein Designer in ADO.NET verwendet wird. Die Hilfe zum .NET Framework enthält unter dem Stichwort »ADO.NET« umfangreiche Informationen über die Architektur von ADO.NET sowie über die verfügbaren Klassen und deren Member.

15.2 Datenbanken anlegen und verwalten

Mit Visual Studio 2005 lässt sich die Microsoft SQL Server Standard Edition installieren. Benutzer von Visual Basic 2005 Express Edition haben dagegen die Möglichkeit, den SQL Server 2005 Express Edition zu installieren. In diesem Abschnitt wird kurz skizziert, wie sich SQL-Datenbanken direkt aus Visual Studio 2005 bzw. Visual Basic 2005 Express Edition erstellen und pflegen lassen. Weiterhin erfahren Sie, wie Sie Datenverbindungen zu anderen Datenquellen (z.B. Access-Datenbanken, Excel-Dateien etc.) einrichten können.

15.2.1 Verwalten der Datenverbindungen

Verbindungen zu SQL-Datenbanken des Microsoft SQL Server (oder der Express Edition) lassen sich direkt aus der Entwicklungsumgebung verwalten. Visual Studio stellt hier das Fenster des Server-Explorer bereit, während in Visual Basic 2005 Express Edition der Datenbank-Explorer als Fenster verfügbar ist (Abbildung 15.2).

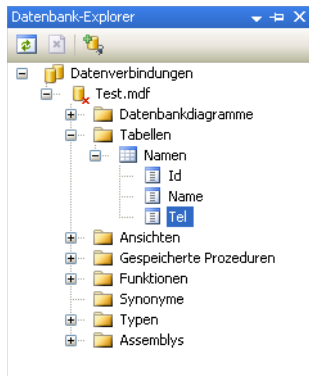


Abbildung 15.2: Zugriff auf SQL-Datenbanken

Über das Symbol *Datenverbindungen* lassen sich alle Verbindungen zu Datenquellen (SQL-Server-Datenbanken, Access-Datenbanken, Excel-Tabellen etc.) anzeigen. Zudem können Sie neue Verbindungen und sogar SQL-Datenbanken neu anlegen. Expandieren Sie den Zweig *Datenverbindungen* und wählen Sie das Symbol einer Verbindung zu einer Datenbank an, listet das Fenster des Datenbank-Explorers (bzw. des Server-Explorers) die in der betreffenden Datenquelle gefundenen Datenbankelemente auf. Zudem erlaubt Ihnen der Datenbank-Explorer (bzw. der Server-Explorers) die Struktur einer Datenbanktabelle zu ändern und auf die Inhalte von Datenbanktabellen zuzugreifen.

Achtung

In Visual Basic 2005 Express Edition lässt sich nur die separat zu installierende SQL-Server 2005 Express Edition nutzen, um auf SQL-Server-Datenbankdateien zuzugreifen. Verwenden Sie Visual Studio 2005, sollte sich auch die jeweils im Paket mitgelieferte Microsoft SQL-Server-Version installieren lassen. Anschließend lassen sich im Server-Explorer und über die DataSet-Designer Verbindungen zu Microsoft-SQL-Server-Datenbanken herstellen. Es hat aber den Anschein, dass es in Visual Studio 2005 noch einen Fehler gibt. Ist auf dem System ein SQL-Server 2005 Express Edition installiert, wird bei der Visual Studio 2005-Installation der im Paket enthaltene Microsoft SQL-Server nicht korrekt eingerichtet. Dann bietet das Fenster des Server-Explorers nur den Zugriff auf die SQL-Server-Datenbankdateien an. Dies ist in den im Buch benutzten Abbildungen der Fall, da beide Versionen der Entwicklungsumgebung auf dem System installiert waren. Falls Sie mit Microsoft SQL-Server arbeiten, sehen die Dialoge etwas anders als hier gezeigt aus.

15.2.2 Eine neue SQL-Server-Datenbankdatei anlegen

Sofern noch keine SQL-Server-Datenbankdatei vorhanden ist, können Sie diese gleich aus der Entwicklungsumgebung heraus anlegen und mit Daten füllen. Die Entwicklungsumgebung bietet Ihnen dabei mehrere Ansätze, um eine *.mdf*-Datei (dies ist eine SQL-Server-Datenbankdatei) zu erzeugen. Sie können die *.mdf*-Datei als neue SQL-Server-Datenbankdatei direkt dem Projekt hinzufügen (siehe unten). Oder Sie benutzen das

Fenster des Datenbank-Explorers (bzw. des Server-Explorers), um eine neue SQL-Server-Datenbank mit folgenden Schritten anzulegen:

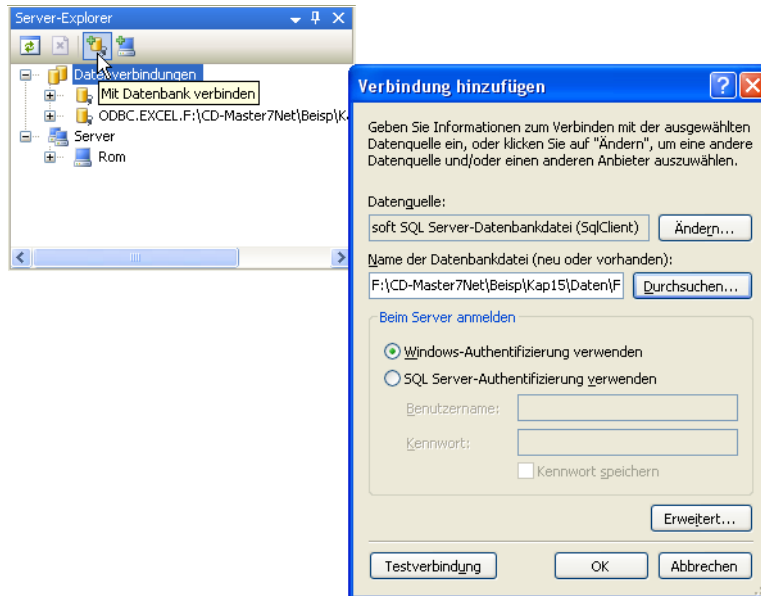


Abbildung 15.3: Eine Datenbankverbindung anlegen

1. Klicken Sie im Fenster *Server-Explorer* (bzw. *Datenbank-Explorer*) auf die Schaltfläche *Mit Datenbank verbinden* (Abbildung 15.3, links oben). Oder wählen Sie das Symbol *Datenverbindungen* mit der rechten Maustaste an und wählen Sie den Kontextmenübefehl *Verbindung hinzufügen*.
2. Im Dialogfeld *Verbindung hinzufügen* (Abbildung 15.3, rechts unten) tippen Sie den Namen der neuen Datenbankdatei (z.B. *Firma.mdf*) im Feld *Name der Datenbankdatei (neu oder vorhanden)* ein. Über die Schaltfläche *Durchsuchen* lässt sich dabei der Pfad zum Zielordner, in dem die Datenbankdatei hinterlegt wird, auswählen.
3. Als Datenquelle belassen Sie die Einstellung auf »Microsoft SQL-Server Datenbankdatei«. Schließen Sie das Dialogfeld über die OK-Schaltfläche.
4. Die Sicherheitsabfrage der Entwicklungsumgebung mit dem Hinweis, dass die Datenbankdatei nicht vorhanden ist und eine neue erstellt werden soll, bestätigen Sie über die *Ja*-Schaltfläche.

Die Entwicklungsumgebung erzeugt dann die Datenbankdatei im Zielordner und hinterlegt gleichzeitig eine Verbindung zu dieser Datenquelle unter dem betreffenden Datenbanknamen (z.B. *Firma*) im Fenster des *Server-Explorers* bzw. *Datenbank-Explorers*. Über diesen Verbindungseintrag können Sie zukünftig eine Verbindung zur Datenquelle aufbauen und dann deren Daten ansehen bzw. manipulieren. Die obige Schrittfolge hat den Vorteil, dass Sie sich nicht mit der Verbindung zum SQL-Server herumschlagen müssen und den Speicherort der Datenbankdatei bestimmen können.

Anlegen einer neuen SQL-Server-Datenbankdatei im Projekt

Ist die SQL-Server 2005 Express Edition vorhanden? Dann haben Sie die Möglichkeit, eine neue SQL Server-Datenbankdatei zur Aufnahme lokaler Daten anzulegen und direkt dem Projekt zuzuordnen. Hierzu gehen Sie in folgenden Schritten vor:

1. Klicken Sie in der Entwicklungsumgebung im Fenster des Projektmappen-Explorers die Projektdatei mit der rechten Maustaste an und wählen Sie im Kontextmenü den Befehl *Hinzufügen/Neues Element*.
2. Wählen Sie im Dialogfeld *Neues Element* den Eintrag *SQL-Datenbank* aus. Korrigieren Sie ggf. den im Feld *Name* eingeblendeten Datenbanknamen und bestätigen Sie die Vorgaben über die *Hinzufügen*-Schaltfläche (Abbildung 15.4, links unten).
3. Sobald der Assistent zum Konfigurieren der Datenquelle startet (Abbildung 15.4, rechts oben), wählen Sie die gewünschten Datenbankobjekte (Tabellen, Ansichten) etc. aus. Hierzu markieren Sie das betreffende Kontrollkästchen in der Liste der angezeigten Datenbankobjekte.

Sobald Sie das Dialogfeld des Assistenten über die *Fertig stellen*-Schaltfläche schließen, wird die neue leere SQL-Server-Datenbankdatei in der Projektmappe unter dem angegebenen Namen erzeugt. Anschließend sehen Sie im Fenster des Projektmappen-Explorers einen Eintrag für die neue SQL-Server-Datenbankdatei. Zudem wurde vom Assistenten ein *DataSet*-Element zur Projektmappe hinzugefügt (über dieses Element erfolgt später der Zugriff auf die Datenbankobjekte). Sie können anschließend, wie im folgenden Abschnitt beschrieben, die Tabellen der SQL-Datenbank im Fenster des Server-Explorer (bzw. im Datenbank-Explorer) definieren und ggf. mit Daten füllen.

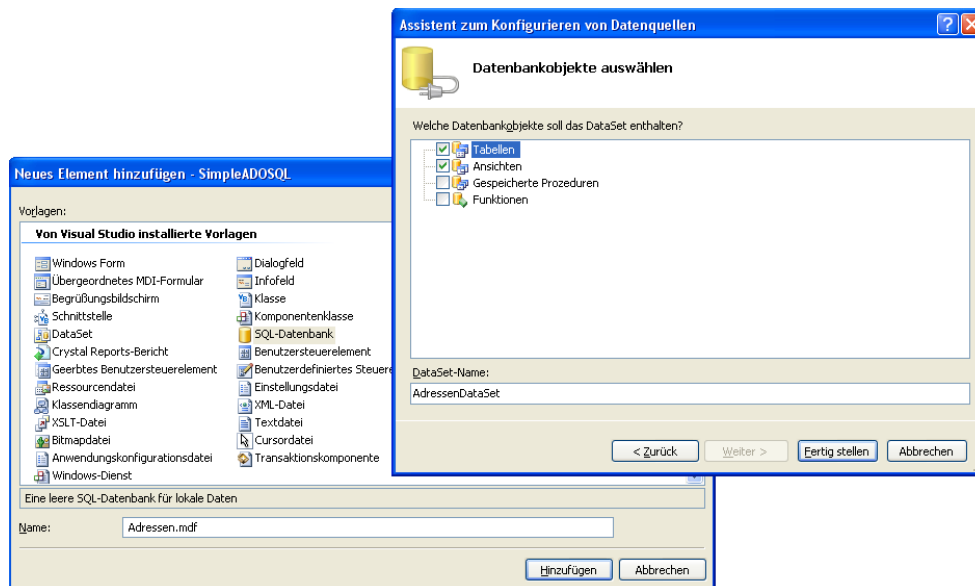


Abbildung 15.4: Hinzufügen einer neuen SQL-Datenbank und Konfigurieren der Datenquelle

Hinweis

Um eine im Projekt eingebundene Datenbankdatei schnell im Server-Explorer (bzw. Datenbank-Explorer) zu öffnen, klicken Sie deren Symbol im Projektmappen-Explorer mit der rechten Maustaste an und wählen den Kontextmenübefehl *Öffnen*. Die Entwicklungsumgebung richtet dann eine Verbindung im Server-Explorer bzw. Datenbank-Explorer ein.

15.2.3 Hinzufügen von Tabellen zur Datenbank

Sobald die Verbindung zur (neuen) SQL-Server-Datenbankdatei in der Entwicklungsumgebung steht, lassen sich Tabellen hinzufügen und deren interne Strukturen (Felder) definieren.

1. Expandieren Sie im Fenster des Server-Explorer bzw. Datenbank-Explorer im Zweig *Datenverbindungen* den Eintrag für die neu angelegte Datenbank.
2. Klicken Sie das Symbol *Tabellen* mit der rechten Maustaste an und wählen Sie den Kontextmenübefehl *Neue Tabelle hinzufügen* (Abbildung 15.5, links oben). Die Entwicklungsumgebung erzeugt dann eine neue leere Tabelle und blendet das Fenster zur Definition der Tabellenstruktur ein.
3. Klicken Sie ggf. auf den Registerreiter des Fensters zur Definition der Tabellenstruktur, um die Tabelleneigenschaften im Eigenschaftfenster anzuzeigen. Der Wert der Eigenschaft (*Name*) wird von der Entwicklungsumgebung mit dem Vorgabewert »Tabelle x« belegt. Sie können dort den gewünschten Tabellennamen (z.B. »Kunde«) eintragen (Abbildung 15.5, rechts unten).
4. Klicken Sie danach im Fenster zur Definition der Tabellenstruktur auf die erste freie Zeile der Spalte *Spaltenname* und tippen Sie den Feldnamen für das gewünschte Tabellenfeld ein.
5. Anschließend legen Sie im Listefeld *Datentyp* der zugehörigen Zeile den Felddatentyp für das gewünschte Tabellenfeld fest. Der Microsoft SQL-Server unterstützt verschiedene Datentypen, die sich über das Listefeld abrufen lassen.
6. Abschließend können Sie über das Kontrollkästchen der Spalte *NULL zulassen* festlegen, ob das Feld bei einem Datensatz leer sein darf.
7. Ist eine Zeile im Fenster zur Definition der Tabellenstruktur markiert, blendet die Entwicklungsumgebung zusätzlich das Fenster *Spalteneigenschaften* ein. Dort können Sie ggf. einzelne Eigenschaften (z.B. die Feldlänge) anpassen.
8. Wiederholen Sie die obigen Schritte, um die Felder der Tabelle zu definieren. Klicken Sie mit der rechten Maustaste auf eine Zeile (oder deren Zeilenkopf), finden Sie im Kontextmenü Befehle, um die Zeile mit der Felddefinition zu löschen oder um eine neue Zeile einzufügen. Feldnamen lassen sich korrigieren, indem Sie diese in der Rubrik *Spaltenname* per Maus anklicken und dann überschreiben.
9. Sind alle Felder der Tabelle definiert, sollten Sie ein Feld als Primärschlüssel kennzeichnen. Hierzu klicken Sie den Zeilenkopf des Feldes mit der rechten Maustaste an und wählen den Kontextmenübefehl *Primärschlüssel festlegen*.

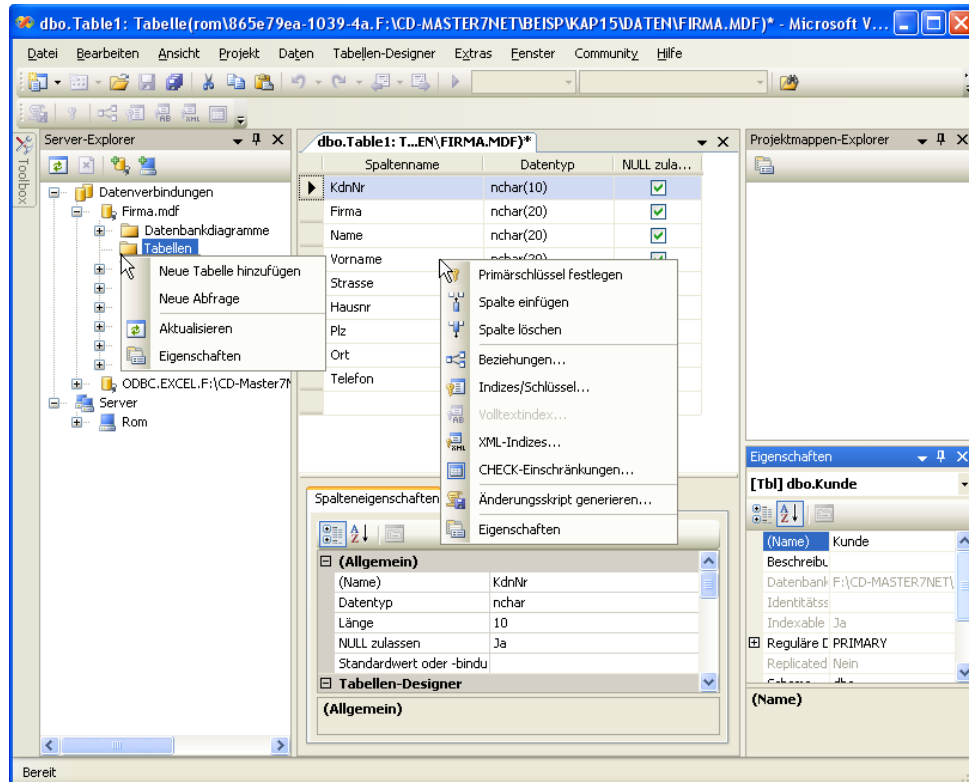


Abbildung 15.5: Tabelle in der Datenbank anlegen

Sobald die Tabellenstruktur definiert ist, können Sie diese über die Schaltfläche <Tabellenname> speichern in der Datenbank ablegen. Die Entwicklungsumgebung blendet anschließend den Tabellennamen sowie die definierten Felder im Zweig *Tabellen* des Datenbank-Explorers ein. Sie können dann das Fenster zur Definition der Tabellenstruktur in der Entwicklungsumgebung schließen. Wählen Sie den Eintrag der Tabelle später im Datenbank-Explorer (bzw. im Server-Explorer) per Doppelklick an, wird das Fenster mit der Tabellenstruktur erneut angezeigt.

Hinweis

Eine Datenbanktabelle besteht aus Feldern, die einen Feldnamen und einen Felddatentyp aufweisen. Die Daten werden dann in Form von Datensätzen in der Tabelle hinterlegt. Ein Datensatz besteht dabei aus den in der Felddefinition spezifizierten Feldern. Dabei kann ein Feld nur solche Daten aufnehmen, die dem Felddatentyp entsprechen. Beim Tabellenentwurf lassen sich als Schlüssel (Index) über Felder definieren. Ein Schlüssel hält den Inhalt des oder der Felder in sortierter Form vor und erlaubt den gezielten Zugriff auf die Datensätze der Tabelle. Der als Primärindex ausgezeichnete Schlüssel besitzt die Einschränkung, dass die Tabelle für das Feld eindeutige Daten aufweist, d.h., es können keine zwei Datensätze mit dem gleichen Wert im betreffenden Feld auftreten.

15.2.4 Tabellendaten anzeigen, eingeben und bearbeiten

Sobald die Tabelle mit der Tabellenstruktur in der Datenbank hinterlegt wurde, können Sie Daten in die Tabelle eintragen und später anzeigen bzw. ändern.

1. Klicken Sie den Eintrag für die neue Tabelle im Datenbank-Explorer (bzw. im Server-Explorer) mit der rechten Maustaste an und wählen Sie den Kontextmenübefehl *Tabellendaten anzeigen* (Abbildung 15.6).
2. Im Fenster der Entwicklungsumgebung wird dann das Tabellenblatt mit den Feldnamen als Spaltenköpfe eingeblendet. Die einzelnen Zeilen entsprechen den Datensätzen der Tabelle. Sie können dann auf die Felder der einzelnen Datensätze klicken und Werte eintragen.

Die Daten werden übernommen, sobald Sie die Eingabe über die **[↵]**-Taste bestätigen. Nicht gespeicherte Werte erkennen Sie durch einen im betreffenden Feld eingeblendeten kleinen roten Kreis mit einem weißen Ausrufezeichen. Drücken Sie die **[Esc]**-Taste, wird die Eingabe im aktuellen Feld verworfen. Enthält ein Feld den Wert »NULL« im Tabellenblatt, signalisiert dies, dass dem Feld im betreffenden Datensatz noch kein Wert zugewiesen wurde. Die unterste Zeile der Tabelle steht für einen neuen Datensatz und enthält in allen Feldern den Wert »NULL«. Tragen Sie in den Feldern dieses Datensatzes Daten ein, fügt die Entwicklungsumgebung automatisch einen neuen Datensatz zur Tabelle hinzu.

Am unteren Rand des Fensters mit der Anzeige des Tabellenblatts finden Sie die Navigationsschaltflächen. Diese erlauben Ihnen in den Datensätzen der Tabelle zu blättern. Klicken Sie mit der rechten Maustaste auf eine Stelle der Tabelle oder auf den Spalten- bzw. Zeilenkopf des Datenblatts, öffnet sich ein Kontextmenü. Dort finden Sie verschiedene Befehle, um z.B. zum SQL-Modus zu wechseln. Bei einem Mausklick auf einen Zeilenkopf wird der betreffende Datensatz markiert. Mit dem Kontextmenübefehl *Löschen* lässt sich dann der so markierte Datensatz entfernen.

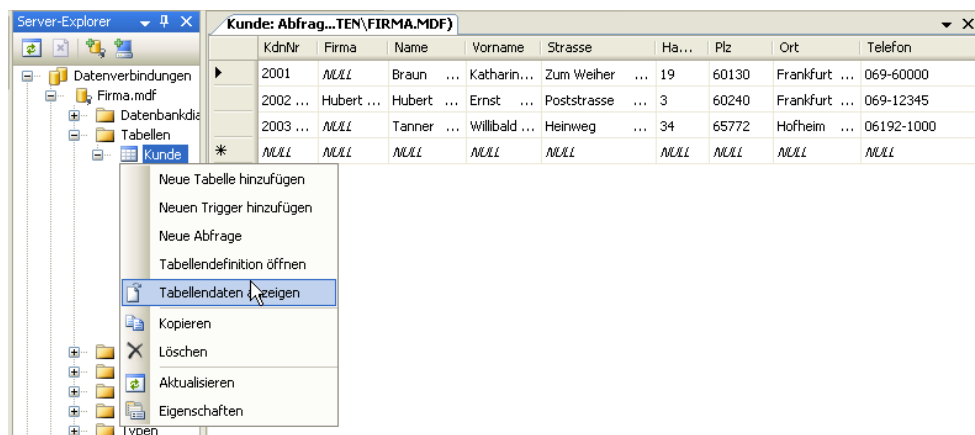


Abbildung 15.6: Tabellendaten anzeigen, eingeben und bearbeiten

15.2.5 Verbindung mit Datenquellen herstellen

.NET unterstützt den Zugriff auf unterschiedliche Datenquellen. Neben SQL-Datenbanken können Sie daher auch auf Access-Datenbanken, auf Excel-Tabellen, auf dBase-Dateien etc. zugreifen. Voraussetzung ist lediglich, dass die betreffende Datenquelle unterstützt und in der Verbindung konfiguriert wurde. Verbindungen zu anderen Datenquellen (z.B. Access-Datenbanken) lassen sich dabei in der Entwicklungsumgebung direkt im Fenster des Server-Explorers (bzw. des Datenbank-Explorers) einrichten. Hierzu verwenden Sie folgende Schritte:

1. Klicken Sie im Fenster des Server-Explorers (bzw. im Datenbank-Explorer) mit der rechten Maustaste auf das Symbol *Datenverbindungen* und wählen Sie im Kontextmenü den Befehl *Verbindung hinzufügen*.
2. Klicken Sie im Dialogfeld *Verbindung hinzufügen* (Abbildung 15.7, links oben) auf die Schaltfläche *Ändern...*, um eine andere Datenquelle auszuwählen.

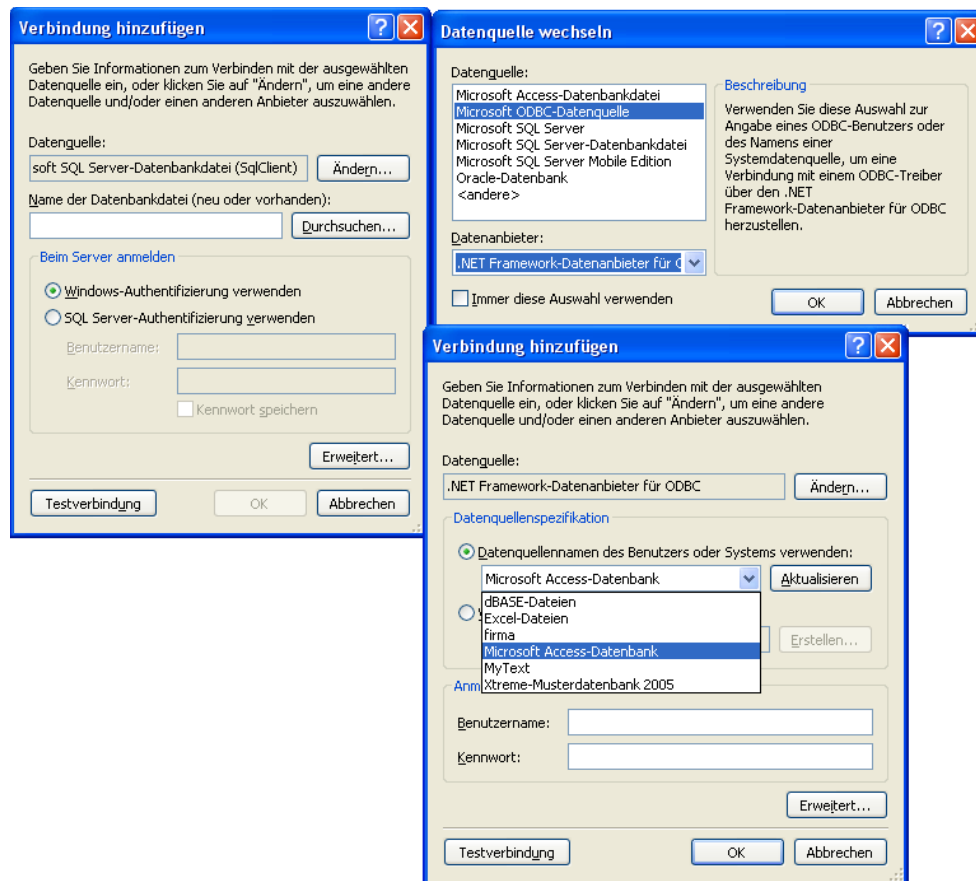


Abbildung 15.7: Einrichten einer Verbindung zu anderen Datenquellen

3. Im Dialogfeld *Datenquelle wechseln* (Abbildung 15.7, rechts oben) finden Sie dann die vom System (bzw. von .NET) unterstützten Datenquellen. Wählen Sie die gewünschte Datenquelle und schließen Sie das Dialogfeld über die OK-Schaltfläche.
4. Im dann angepassten Dialogfeld *Verbindung hinzufügen* müssen Sie den Datenquellenennamen über das betreffende Listefeld auswählen (Abbildung 15.7, rechts unten).
5. Über die Schaltfläche *Testverbindung* lässt sich prüfen, ob ein Zugriff auf die Datenquelle möglich ist. Die OK-Schaltfläche schließt das Dialogfeld und richtet die Verbindung zur Datenquelle ein.

Im Dialogfeld *Datenquelle wechseln* finden Sie die vom System unterstützten Datenquellen. .NET stellt dabei verschiedene Datenanbieter (Provider) zum Zugriff auf Datenquellen bereit. Neben den Datenanbietern zum Zugriff auf den Microsoft SQL-Server oder auf Microsoft-SQL-Server-Datenbankdateien ist auch ein Provider zur Verbindungsaufnahme mit Microsoft-Access-Datenbanken vorhanden. Den Eintrag »Microsoft ODBC-Datenquelle« können Sie verwenden, um auf unterschiedliche Datenquellen wie Access-Datenbanken, Excel-Tabellen, CSV- und Textdateien etc. zuzugreifen.

Wählen Sie den Eintrag *<andere>* (Abbildung 15.7, rechts oben) in der Liste der Datenquellen an, lässt sich der Datenanbieter (Datenbank-Provider) im Listefeld *Datenanbieter* frei wählen. Neben den SQL-Providern sollten auch .NET-Datenanbieter für ODBC- und OLE-DB-Zugriffe aufgelistet werden.

Die im Dialogfeld *Verbindung hinzufügen* zur Verbindungsaufnahme angebotenen Optionen hängen vom gewählten Datenanbieter ab. Wählen Sie den ODBC-Datenanbieter aus, muss eine ODBC-Verbindung über den ODBC-Datenquellenadministrator definiert worden sein (siehe folgender Abschnitt). In dieser Verbindung findet der Datenanbieter die Informationen zum Zugriff auf die Datenquelle. Bei einem OLE-DB-Provider muss der OLE-DB-Anbieter sowie der Pfad zur Datenquelle im Dialogfeld *Verbindung hinzufügen* eingetragen werden.

Definition einer Benutzer-DSN für eine ODBC-Verbindung

ODBC-Verbindungen greifen auf die Verbindungsdaten zurück, die über den ODBC-Datenquellen-Administrator definiert wurden. Ein Administrator kann die Verbindungsdaten für eine ODBC-Verbindung mit folgenden Schritten konfigurieren:

1. Wählen Sie im Fenster der Windows-Systemsteuerung das Symbol *Datenquellen (ODBC)* per Doppelklick an. Ist das Symbol in der Systemsteuerung nicht zu sehen, wählen Sie das Symbol *Verwaltung* per Doppelklick. Im Ordnerfenster *Verwaltung* finden Sie das Symbol *Datenquellen (ODBC)*.
2. Wählen Sie im Fenster des dann angezeigten ODBC-Datenquellen-Administrators die Registerkarte *Benutzer-DSN* und klicken Sie anschließend auf die Schaltfläche *Hinzufügen* der Registerkarte (Abbildung 15.8, links oben).
3. Im Dialogfeld *Neue Datenquelle erstellen* wählen Sie einen zur Datenquelle passenden Treiber aus (Abbildung 15.8, rechts oben). Dies kann z.B. der Microsoft-Access-Treiber für Zugriffe auf Access-Datenbanken sein. Oder Sie verwenden den Microsoft-Excel- bzw. dBase-Treiber für Zugriffe auf .xls- oder .dbf-Dateien. Mit dem Microsoft-Text-Treiber lässt sich auf Daten zugreifen, die in Text- oder CSV-Dateien strukturiert abgelegt sind. Schließen Sie das Dialogfeld über die *Fertig stellen*-Schaltfläche.

4. Der Aufbau des dann angezeigten Dialogfelds *ODBC <Treiber> Setup* hängt vom gewählten Treiber ab. In Abbildung 15.8, unten, ist die Variante mit dem Texttreiber zu sehen. Tragen Sie als Erstes einen Namen für die DSN im Feld *Datenquellenname* ein. Dieser Name taucht später in der Entwicklungsumgebung beim Konfigurieren der Verbindung über ODBC-Datenanbieter auf. Über die Schaltfläche *Verzeichnis auswählen* (bzw. *Arbeitsmappe auswählen* etc.) kann dann der Pfad zur Datenquellendatei eingestellt werden. Ist die Schaltfläche gesperrt, müssen Sie die Markierung des Kontrollkästchens *Aktuelles Verzeichnis verwenden* löschen. Je nach Treiber lassen sich dann zusätzliche Einstellungen (z.B. Version der Excel- oder dBase-Datei) vorgeben. Die Schaltfläche *Optionen* erweitert ggf. das Dialogfeld und Sie können spezifische Einstellungen vornehmen (z.B. bei Textdateien lässt sich das Dateiformat eingrenzen).
5. Nach dem Setup der Verbindung ist das Dialogfeld über die *OK*-Schaltfläche zu schließen. Anschließend können Sie den ODBC-Datenquellen-Administrator ebenfalls über die *OK*-Schaltfläche schließen.

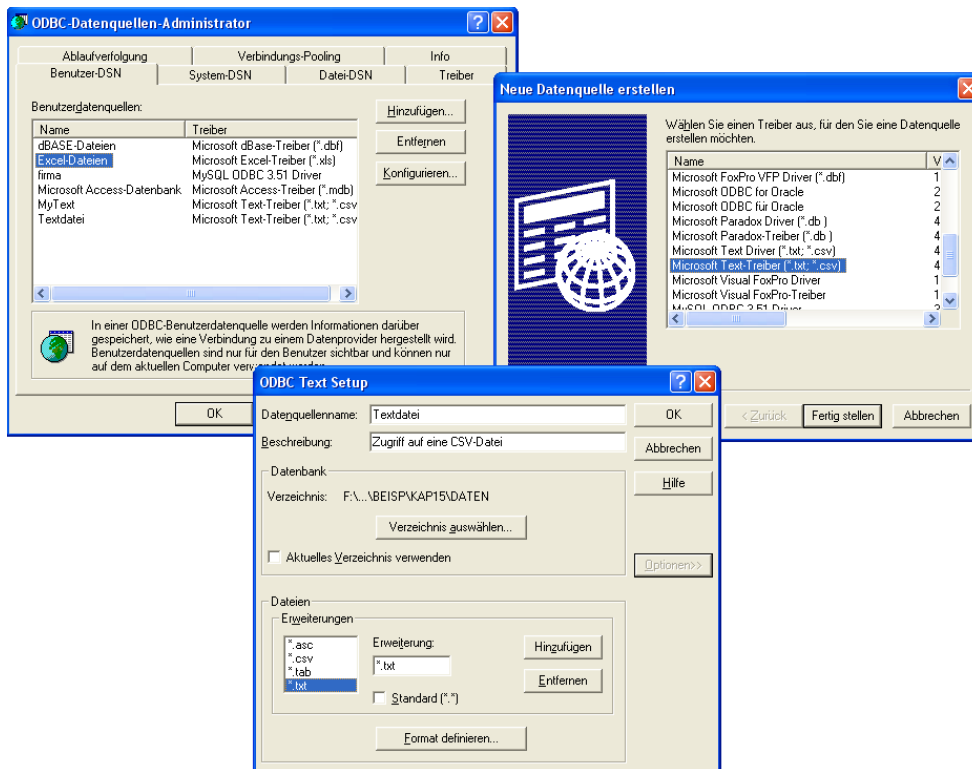


Abbildung 15.8: Einrichten einer Benutzer-DSN für eine ODBC-Verbindung

Der ODBC-Datenquellen-Administrator legt eine Data-Source-Name(DSN)-Verbindung als Datei in einem von Windows verwalteten Verzeichnis ab. In dieser Datei sind alle zum Zugriff auf die betreffende Datenquelle benötigten Verbindungsdaten gespeichert. Je nach Datenquelle ist dabei der genaue Name der Datenquellendatei (z.B. bei Excel Arbeitsmappen oder bei Access Datenbankdateien) aufgeführt. Bei dBase- oder Textda-

teien wird dagegen nur das Verzeichnis mit den Datenquellendateien hinterlegt. Wird eine solche ODBC-Verbindung als Datenquelle im Server-Explorer oder im Datenbanken-Explorer eingerichtet, listet das betreffende Fenster bei Anwahl des Symbols *Tabellen* dann alle im Verzeichnis gefundenen dBase- oder Textdateien als Tabellen auf.

Tipp

Erlaubt die ODBC-Verbindung bei einem bestimmten Datenquellentyp nur das Verzeichnis festzulegen? Um Probleme beim gezielten Zugriff auf Textdateien oder dBase-Datenbankdateien zu vermeiden, empfiehlt es sich, diese ggf. in separaten Unterordnern abzulegen. Dann lässt sich für jeden Unterordner eine separate ODBC-Verbindung anlegen.

15.2.6 SQL-/Access-Datenbankdatei in das Projekt einbinden

Sofern Sie in der .NET-Anwendung mit einer SQL Server- oder einer Access-Datenbankdatei arbeiten, empfiehlt es sich, die betreffende Datenquelle direkt im Projekt einzubinden (funktioniert aber nur bei installierter SQL-Server 2005 Express Edition). Dann erstellt die Entwicklungsumgebung automatisch die benötigten Elemente (z.B. die *DataSet.xsd*-Datei) und kopiert die Datenbankdatei beim Erstellen des Projekts in das Zielverzeichnis der Anwendungsdatei.

1. Klicken Sie im Fenster des Projektmappen-Explorers das Symbol der Projektdatei mit der rechten Maustaste an und wählen Sie im Kontextmenü die Befehle *Hinzufügen/Vorhandenes Element*.
2. Im Dialogfeld *Vorhandenes Element* stellen Sie den Wert des Listenfelds *Dateityp* auf »Datendateien (*.xsd; *.xml; *.mdf; *.mdb)«. Anschließend suchen Sie das Zielverzeichnis mit der Datenbankdatei, wählen diese per Mausklick an und schließen das Dialogfeld über die Schaltfläche *Hinzufügen*.
3. Sobald der Dialog des Assistenten zur Konfigurierung der Datenquelle erscheint (siehe Abbildung 15.16 weiter unten) markieren Sie die Kontrollkästchen der Datenbankelemente (z.B. *Tabellen*), die in das DataSet-Schema zu übernehmen sind.

Sobald Sie den Dialog des Assistenten über die *Fertig stellen*-Schaltfläche schließen, richtet der Assistent eine Verbindung zur Datenbank im Projekt ein, fügt die Datenbankdatei zum Projekt hinzu und erzeugt auch ein als *<Datenbankname>DataSet.xsd* bezeichnetes Element im Projektmappen-Explorer. Das *DataSet.xsd*-Element enthält ein Schema, welches die Typinfos zu den Datenbankobjekten (Tabellen, Ansichten etc.) enthält. Das aus diesem Schema abgeleitete *DataSet*-Objekt dient zur Laufzeit als Cache (fungiert wie eine lokale Datenbank), welches die für von der Datenbank bzw. deren Objekten (z.B. Tabellen) gelesenen Daten im Arbeitsspeicher hält. Auf den folgenden Seiten wird gezeigt, wie sich aus der .NET-Anwendung auf das DataSet zugreifen lässt.

Hinweis

Ein leeres *DataSet*-Element lässt sich im Projekt einfügen, indem Sie im Kontextmenü der Projektdatei die Befehle *Hinzufügen/Neues Element* anklicken. Wählen Sie dann im Dialogfeld *Neues Element* die Vorlage *DataSet* und klicken Sie auf die *Hinzufügen*-Schaltfläche. Ein Doppelklick auf den neuen *DataSet*-Eintrag im Projektmappen-Explorer öffnet diesen im *DataSet*-Designer. Ziehen Sie anschließend eine Tabelle einer bereits eingerichteten Datenverbindung aus dem Fenster des Server-Explorers (bzw. Datenbank-Explorers) in den *DataSet*-Designer, richtet dieser das *DataSet* zur Verwaltung dieser Tabellendaten ein.

15.3 Zugriff auf Datenquellen aus Formularen

Sie können in .NET-Anwendungen über verschiedene Ansätze (z.B. über selbst geschriebenen Programmcode) auf die Tabellen der Datenquellen zugreifen. Um lediglich Daten einer Tabelle in einem Formular anzuzeigen oder einfache Bearbeitungsfunktionen für die Daten bereitzustellen, können Sie auf die Designer der Entwicklungsumgebung zugreifen. Diese ermöglichen Anzeigeelemente im Formular einzufügen und generieren automatisch den Code, um Datenbanktabellen an diese Elemente anzubinden. Nachfolgend wird an einigen Beispielen skizziert, wie sich diese Funktionen nutzen lassen.

15.3.1 Darstellung einer Datenbanktabelle per Formular

In einem ersten Beispiel soll skizziert werden, wie sich eine Datenbanktabelle direkt auf ein Formular abbilden lässt. Dabei soll das Formular eine Navigationsleiste zum Blättern in den Datensätzen bieten und auch das Bearbeiten und Speichern von Datensätzen zulassen.

1. Legen Sie ggf. ein neues Windows-Projekt mit einem Formular in der Entwicklungsumgebung an. Das Formular soll zur Anzeige der Tabellendaten dienen.
2. Erstellen Sie ggf. eine SQL-Server-Datenbankdatei, die Sie mit einer Tabelle und den notwendigen Daten ergänzen. Alternativ können Sie auf eine bereits bestehende Datenquelle (SQL-Datenbankdatei, Access-Datenbankdatei etc.) aufsetzen. Nehmen Sie die Datenbankdatei bzw. die Datei der Datenquelle im Projekt mit auf.

Die erforderlichen Schritte zum Anlegen neuer Projekte, zum Erstellen von Datenbanken oder zum Entwerfen von Tabellen sowie zum Hinzufügen von Daten wurden auf den vorhergehenden Seiten beschrieben. Abbildung 15.9 zeigt das Fenster der Entwicklungsumgebung mit dem leeren Formular, dem geöffneten Server-Explorer, in dem die Verbindung zur neuen Datenbank eingeblendet wird und den Projektmappen-Explorer. Die SQL-Server-Datenbankdatei ist als Projektelement zu sehen. Beim Hinzufügen der Datenbank zum Projekt wird auch gleich ein *DataSet*-Element generiert und im Projekt eingefügt. Dieses Element stellt die Mechanismen zum Zugriff auf die Tabellen der Datenquelle/Datenbank bereit.

Hinweis

In Abbildung 15.9 ist zusätzlich noch ein separates Fenster *Datenquellen* sichtbar. Falls das Fenster in Ihrer Entwicklungsumgebung fehlt, lässt es sich über den Befehl *Datenquellen anzeigen* im Menü *Daten* der Entwicklungsumgebung einblenden. Dieses Fenster wird benötigt, um auf sehr einfache Weise Anzeigeelemente zur Abbildung einer Datenbanktabelle im Formular zu generieren.

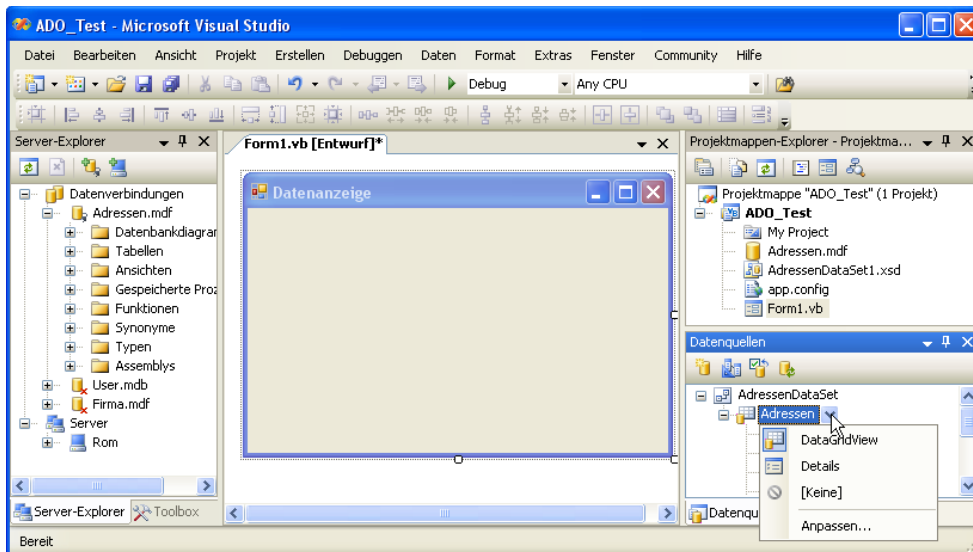


Abbildung 15.9: Projektelemente zum Zugriff auf Datenbanktabellen

Erstellen des Datenzugriffformulars

Nachdem das Projekt durch Einbinden der Datenbank (bzw. Datenquelle) vorbereitet ist und über ein leeres Formular verfügt, soll dieses für den Datenzugriff auf die Tabellendaten hergerichtet werden. Zur Anzeige lässt sich eine Art Tabellendarstellung in Form eines *DataGridView*-Steuerelements nutzen. Dieses listet mehrere Datensätze sowie die Felder der Tabelle auf. Eine Alternative besteht darin, die Tabellendaten auf Textfelder im Formular abzubilden. Zur Navigation innerhalb der Datensätze wird dann eine Navigationsleiste benutzt, die an die betreffende Datenquelle anzubinden ist. Das Hinzufügen der Anzeigeelemente zum Formular sowie das Konfigurieren der Datenquellenanbindung können Sie komplett dem Designer der Entwicklungsumgebung überlassen.

1. Holen Sie den Ansicht-Designer mit dem zur Anzeige der Tabellendaten vorgesehenen Formular in den Vordergrund.
2. Klicken Sie ggf. auf den Registerreiter *Datenquellen*, um das betreffende Fenster in den Vordergrund zu holen.
3. Expandieren Sie im Fenster *Datenquellen* den *DataSet*-Zweig und klicken Sie auf die Schaltfläche des Knotens mit der Datenbanktabelle (in Abbildung 15.9 ist dies der Knoten *Adressen*, der die Datenbanktabelle abbildet).

4. Wählen Sie im eingeblendeten Menü die Anzeigevariante *DataGridView* oder *Details*, die bestimmt, welches Steuerelement im Formular einzufügen ist.
5. Anschließend ziehen Sie den Knoten mit dem Tabellennamen aus dem *DataSet*-Zweig des Fensters *Datenquellen* per Maus zum im *Ansicht-Designer*-Fenster eingeblendeten Formular.

Sobald Sie die Maustaste loslassen, fügt der Designer ein Anzeige-Steuerelement sowie eine Navigationsleiste im Formular ein und ergänzt dieses um die erforderlichen Elemente zur Anbindung an die Tabelle.

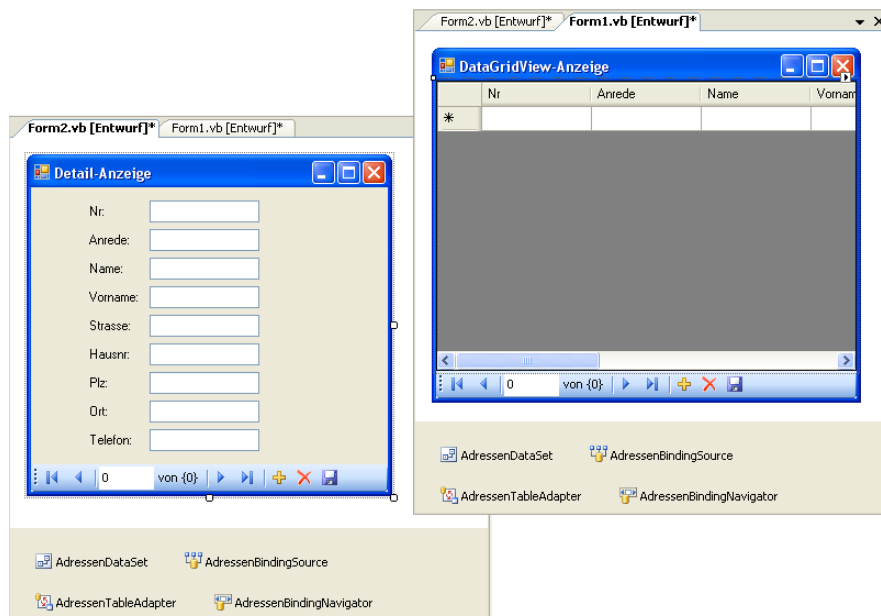


Abbildung 15.10: Formularentwürfe mit eingeblendeten Anzeigeelementen

In Abbildung 15.10 sehen Sie beide Formularvarianten mit der aus Textfeldern bestehenden Detailanzeige und der Variante mit einem *DataGridView*. Sie müssen dann die Navigationsleiste anklicken und über deren *Dock*-Eigenschaft am unteren Formularrand verankern. Beim *DataGridView*-Steuerelement empfiehlt es sich, die *Dock*-Eigenschaft auf »Fill« zu setzen, um das Formular auszunutzen.

Der Pfiff bei diesem Ansatz besteht darin, dass der Designer anschließend alle erforderlichen Komponenten erzeugt, um die Tabellendaten zur Laufzeit im Formular anzuzeigen. Der Designer fügt hierzu nicht nur ein *BindingSource*-Element zur Anbindung an die Datenbank im Formular ein. Es werden auch ein *DataSet*-Element (fungiert als Daten-cache im Arbeitsspeicher), ein *TableAdapter*-Element (zum Zugriff auf die Tabellendaten) sowie ein *BindingNavigator*-Element (stellt die Navigationsleiste) im Formular hinterlegt. Gleichzeitig werden diese Elemente automatisch so konfiguriert, dass Zugriffe auf die Datenbanktabelle möglich sind. Weiterhin fügt der Designer automatisch Code in der *Load*-Ereignisbehandlungsroutine des Formulars sowie in der *Click*-Ereignisbehandlungsroutine der *Speichern*-Schaltfläche der Navigationsleiste ein.

```

Public Class Form1
    Private Sub AdressenBindingNavigatorSaveItem_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles AdressenBindingNavigatorSaveItem.Click
        Me.Validate()
        Me.AdressenBindingSource.EndEdit()
        Me.AdressenTableAdapter.Update(Me.AdressenDataSet.Adressen)
    End Sub
    Private Sub Form1_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load
        ' TODO: Diese Codezeile lädt Daten in die Tabelle
        ' "AdressenDataSet.Adressen". Sie können sie bei
        ' Bedarf verschieben oder entfernen.
        Me.AdressenTableAdapter.Fill(Me.AdressenDataSet.Adressen)
    End Sub
End Class

```

Listing 15.1: Vom Designer generierter Code zum Zugriff auf die Datenbanktabelle

Über den Aufruf der *Fill()*-Methode des *TableAdapter*-Objekts werden die Daten der Tabelle über das *DataSet*-Objekt eingelesen. In der *SaveItem_Click*-Ereignisbehandlungsprozedur des Navigationselements sorgt ein Aufruf der *Update()*-Methode dafür, dass die im Steuerelement geänderten Daten über das *DataSet*-Objekt zur Datenbanktabelle zurückgeschrieben werden.

Tipps zum Anpassen der Tabelleneigenschaften beim DataSet

Ein *DataSet*-Objekt wirkt als Cache im Speicher, der die Daten der Datenbanktabelle verwaltet. Das zugehörige *DataSet*-Schema beschreibt daher alle Tabellen und deren (Feld-) Elemente, die durch das *DataSet*-Objekt verwaltet werden. Anpassungen müssen daher über die Eigenschaften des *DataSet* vorgenommen werden. Soll z.B. ein neuer Datensatz in einer Tabelle eingefügt werden, muss bei einem als Primärindex ausgewiesenen Tabellenfeld sichergestellt sein, dass der betreffende Feldwert im neuen Datensatz eindeutig ist. Während Microsoft Access z.B. so etwas wie AutoFelder (der Wert wird automatisch erzeugt) unterstützt, bietet die SQL-Server-Datenbank beim Entwurf der Tabellenstruktur keine entsprechende Eigenschaft für Felder. Um beim Anlegen neuer Datensätze einer Tabelle eindeutige Werte für ein Schlüsselfeld zu generieren, müssen Sie die entsprechenden Vorgaben im *DataSet*-Schema vornehmen. So lässt sich im Schema die *AutoIncrement*-Eigenschaft des betreffenden Tabellenfeldes auf *true* setzen.

1. Wählen Sie den im Projektmappen-Explorer angezeigten Eintrag für das *DataSet*-Element per Doppelklick an. Dann wird das Fenster des *DataSet*-Designers mit den im *DataSet* definierten Tabellen eingeblendet (Abbildung 15.11, linkes Fenster).
2. Markieren Sie in der Tabelle des *DataSet*-Designers das gewünschte Tabellenfeld per Maustaste. Dann werden die Feldeigenschaften im Eigenschaftenfenster angezeigt.

3. Setzen Sie die Eigenschaften des betreffenden Felds auf die gewünschten Werte und schließen Sie danach die Anzeige des *DataSet*-Designers.

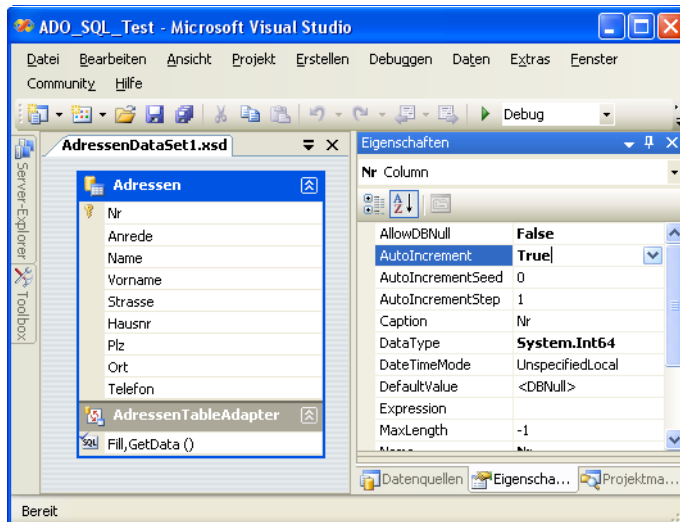


Abbildung 15.11: Tabelleneigenschaften im DataSet-Objekt anpassen

In Abbildung 15.11 wurde in dem in der rechten Spalte sichtbaren Eigenschaftenfenster die Eigenschaft *AutoIncrement* des Feldes *Adressen.Nr* auf *True* gesetzt. Der Startwert *AutoIncrementSeed* beginnt mit 0 und die Schrittweite *AutoIncrementStep* ist auf 1 gesetzt. Damit versucht die beim Einfügen neuer Datensätze aufgerufene *Insert()*-Methode des im *DataSet* enthaltenen *TableAdapter* den Wert des Felds *Nr*, beginnend bei 0, zuzuweisen. Der Wert wird schrittweise um 1 erhöht, bis sichergestellt ist, dass für den Primärinde ein eindeutiger Wert im Feld des neuen Datensatzes eingefügt wird.

In den Feldeigenschaften eines *DataSet*-Schemas können Sie auch den Spaltentitel (Eigenschaft *Caption*), die maximale Felddlänge (Eigenschaft *MaxLength*), die Zulässigkeit von NULL-Werten für Felder (Eigenschaft *AllowDBNull*) etc. anpassen. Hinweise zu den Eigenschaften finden Sie in der .NET-Framework-Hilfe, wenn Sie im Index nach *DataColumn*-Klasse und dann nach *Eigenschaften* nachschlagen.

Hinweis

Sie finden das Projektbeispiel im Ordner `\Beisp\Kap15\ADO_SQL_Test` auf der Begleit-CD. Das Beispiel ist als Windows-Anwendung realisiert und beinhaltet eine kleine SQL-Server-Datenbankdatei mit Adressdaten im Programmordner. Die Anwendung zeigt beim Start ein einfaches Formular, über dessen Schaltflächen Sie wahlweise ein Dateneingabeformular mit *DataGridView*-Anzeige oder eine *Detail*-Anzeige abrufen können. Beide Formulare bieten eine Navigationsleiste, über deren Schaltfläche *Speichern* Sie im Formular an den Datensätzen vorgenommene Änderungen in die Datenbank speichern können.

Achtung

So komfortabel der Ansatz zum Formularentwurf über den DataSet-Designer ist, gibt es doch einige Probleme. So sind beispielsweise die Änderungsmöglichkeiten der Formulardaten sehr begrenzt. Klicken Sie auf einen im Formular angezeigten Tabellenwert und fügen Sie am Ende des Texts ein neues Zeichen ein, löst dies beim Wechsel in eine neue Zeile/Spalte u.U. einen Fehler aus. Es wird gemeldet, dass die Textlänge der betreffenden Spalte den *MaxLength*-Werte überschreitet. Die Ursache: Beim Eintippen des neuen Zeichens in die Zelle des DataGridView arbeitet die Anwendung im Einfügen-Modus. Dadurch ändert sich die Länge des hinterlegten Texts. Durch beim Anzeigen automatisch angehängte Leerzeichen wird dann der Text länger als die für das Feld festgelegte *MaxLength*-Einstellung. Vermeiden lässt sich dies nur, indem der Benutzer darauf achtet, nach dem Einfügen eines Texts die entsprechende Anzahl an Leerzeichen am Ende des Texts manuell zu löschen (oder Sie implementieren eine Funktion in den Ereignisbehandlungsroutinen der betreffenden Steuerelemente, die die Textlänge der Benutzereingaben auf die *MaxLength*-Eigenschaft des Feldes abstimmt). Eine weitere Einschränkung: Vergessen Sie die *Speichern*-Schaltfläche der Navigationsleiste anzuklicken, wird auch keine *Update()*-Methode ausgeführt und die Änderungen gehen verloren. Bei konkreten Projekten müssten Sie daher in der *FormClosing*-Ereignisbehandlungsroutine die oben gezeigten Anweisungen zum Update der Änderungen hinterlegen. Im aktuellen Beispiel habe ich dies für das Formular mit der *DataGridView*-Anzeige manuell implementiert. Ob geänderte Werte vorliegen, lässt sich übrigens durch den Aufruf der *HasChanges()*-Methode des *DataSet* ermitteln. Der unangenehmste Effekt tritt beim Testen in der Entwicklungsumgebung auf. Bei jedem Start der Anwendung kopiert die Entwicklungsumgebung auch die im Projekt eingeschlossene Datenbankdatei aus der Projektmappe in den Programmordner der *.exe*-Datei. Eventuell in einer vorherigen Sitzung an der Datenbanktabelle vorgenommene Änderungen werden dann überschrieben. Beim Testen sieht es dann vordergründig so aus, als ob die Änderungen der vorherigen Sitzungen verloren gegangen wären, bzw. als ob die *Update()*-Methode nicht funktioniert. Sie können aber das Kopieren der SQL-Datenbankdatei in das Zielverzeichnis der *.exe*-Datei verhindern. Klicken Sie im Projektmappen-Explorer den Eintrag der Projektdatei an. Anschließend setzen Sie im Eigenschaftfenster die Eigenschaft *In Ausgabeverzeichnis kopieren* auf den Wert »Kopieren, wenn neuer«. Dann wird die im Projektordner enthaltene Datenbankdatei nur in den Ausgabepfad kopiert, wenn Sie manuell Änderungen vorgenommen haben.

15.3.2 Tabelle manuell an ein DataGridView binden

Die Verwendung des *DataSet*-Designers zum Erstellen eines Formulars mit Elementen zur Anzeige von Tabellendaten ist zwar recht komfortabel. Der Designer erzeugt alle erforderlichen Anzeigeelemente im zugehörigen Formular und konfiguriert auch deren Eigenschaften. Oft ist es aber so, dass einzelne Steuerelemente direkt im Ansicht-Designer in Formulare einzufügen sind. Dann sollten Sie wissen, wie sich die Steuerelemente manuell im Ansicht-Designer an eine Datenbanktabelle anbinden lassen. In einem weiteren Beispiel soll jetzt ein *DataGridView*-Steuerelement manuell in ein Formular aufgenommen und dann an eine in einer SQL-Server Datenbankdatei *Adressen.mdf* hinterlegte Tabelle gebunden werden.

1. Legen Sie in der Entwicklungsumgebung ein neues Projekt unter Verwendung einer Windows-Anwendung an.
2. Anschließend fügen Sie eine neue SQL-Server-Datenbankdatei zum Projekt hinzu, definieren eine Tabellenstruktur zur Aufnahme von Adressdaten und fügen anschließend einige Daten zur Tabelle hinzu. Alternativ können Sie eine bestehende Datenbankdatei zum Projekt hinzufügen. Die erforderlichen Schritte wurden am Kapitelanfang sowie im vorhergehenden Abschnitt skizziert.
3. Öffnen Sie das Formular im Formular designer und fügen Sie aus der Toolbox ein *DataGridView*-Steuerelement zum Formularbereich hinzu.
4. Sobald das Steuerelement eingefügt wurde, erscheint das Fenster *DataGridView-Aufgaben* im Vordergrund (Abbildung 15.12). Passen Sie die Markierung der Kontrollkästchen *Hinzufügen aktivieren*, *Löschen aktivieren* etc. gemäß Ihren Anforderungen an. Ein markiertes Kontrollkästchen bewirkt, dass das *DataGridView*-Steuerelement die betreffenden Funktionen bereitstellt.
5. Wählen Sie im Listenfeld *Datenquelle auswählen* des Fensters *DataGridView-Aufgaben* die im Projekt angewählte Datenbank als Datenquelle.

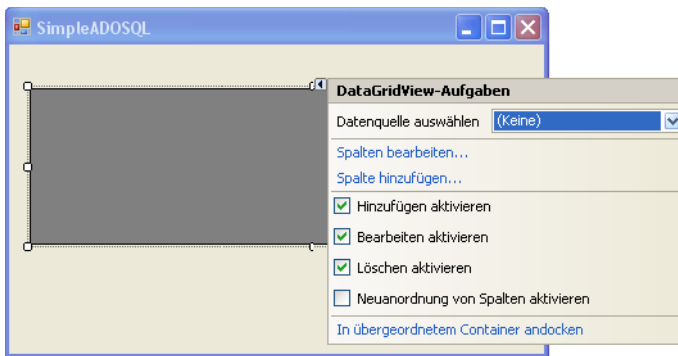


Abbildung 15.12: Aufgaben des DataGridView festlegen

Sobald Sie im Ansicht-Designer auf eine Stelle außerhalb des Fensters *DataGridView-Aufgaben* klicken, verschwindet die Anzeige und Sie sehen im unteren Bereich des Ansicht-Designers die automatisch eingefügten Steuerelemente *AdressenDataSet*, *AdressenBindingSource* und *AdressenTableAdapter*. Gleichzeitig sollte bereits die Spaltenstruktur im *DataGridView*-Steuerelement des Formulars sichtbar werden (Abbildung 15.13).

Die Funktion *DataGridView-Aufgaben* der Entwicklungsumgebung stellt automatisch die benötigten Eigenschaften für die einzelnen Steuerelemente ein. Weiterhin wird in der *Load*-Ereignisbehandlungsroutine des Formulars die Anweisung

```
Me.AdressenTableAdapter.Fill(Me.AdressenDataSet.Adressen)
```

hinterlegt. Der Vorspann *<Adressen>* in obiger Anweisung resultiert aus der hier im Beispiel verwendeten Datenbankdatei *Adressen.mdf*. Diese Anweisung ruft die *Fill()*-Methode des *TableAdapter*-Steuerelements auf. Als Argument wird dabei das Tabellenobjekt *DataSet.Adressen* übergeben (*Adressen* steht hier für den Namen der Tabelle).

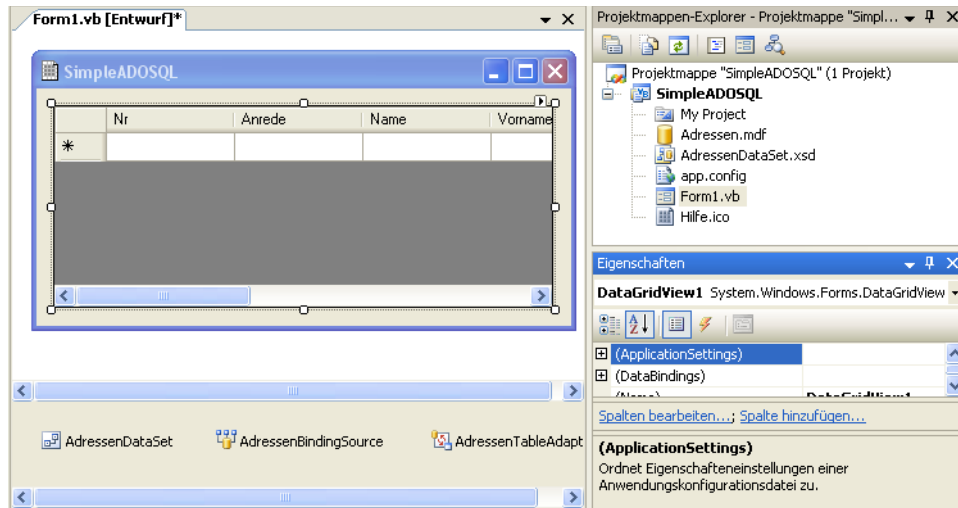


Abbildung 15.13: Formularentwurf mit DataGridView und Eigenschaftenfenster

Bei Bedarf können Sie noch die Abmessungen des *DataGridView*-Steuerelements im Ansicht-Designer anpassen. Sofern Sie die *Dock*-Eigenschaft des Steuerelements auf *Fill* setzen, nimmt das *DataGridView* den gesamten Bereich des Formulars ein. Ist das Steuerelement markiert, finden sich im Eigenschaftenfenster zudem die Hyperlinks *Spalten bearbeiten* und *Spalten hinzufügen*, die Zusatzdialoge öffnen. Dort können Sie die Felder der Tabelle den Spalten des *DataGridView*-Steuerelements zuordnen oder die Spaltenüberschriften anpassen. Wird dagegen das *TableAdapter*-Steuerelement im Ansicht-Designer gewählt, können Sie im Eigenschaftenfenster den Hyperlink *Abfragen im DataSet-Designer bearbeiten* wählen. Der *DataSet*-Designer zeigt die stilisierte Tabellenstruktur sowie das *TableAdapter*-Element. Wählen Sie dieses Element, lassen sich dessen Eigenschaften (z.B. der SQL-Befehl zur Datenabfrage) im Eigenschaftenfenster anzeigen bzw. modifizieren. Sobald Sie das Beispiel in der Entwicklungsumgebung übersetzen und ausführen lassen, erscheint bereits das Formular mit dem *DataGridView*-Steuerelement. In dem zum *DataGridView* zugehörigen Datenblatt werden dann die Datensätze der gewählten Tabelle eingeblendet. Sie können zudem die Daten direkt manipulieren – diese werden aber nicht gespeichert (da die betreffenden Anweisungen zum Aufruf der *Update()*-Methode noch fehlen).

Hinweis

Sie finden das als Windows-Anwendung realisierte Projektbeispiel samt der Datenbankdatei im Ordner `\Beisp\Kap15\SimpleADOSQL` auf der Begleit-CD.

Ergänzen des Formulars um ein BindingNavigator-Steuerelement

Zur Navigation innerhalb der Daten einer Datenbanktabelle lässt sich die bereits auf den vorhergehenden Seiten benutzte Navigationsleiste verwenden. Hierzu müssen Sie das *BindingNavigator*-Steuerelement aus der Toolbox zum Formular hinzufügen und an die Tabelle anbinden.

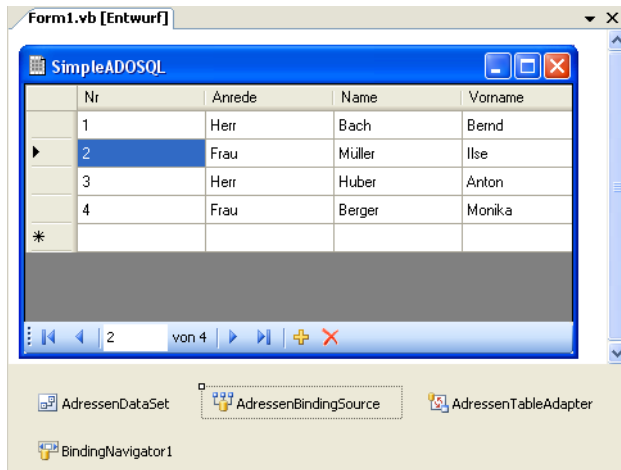


Abbildung 15.14: Tabellenanzeige mit Navigationsleiste (Vordergrund)

1. Öffnen Sie das auf den vorhergehenden Seiten erstellte Formular mit dem *DataGridView*-Steuerelement im Formulardesigner und fügen Sie ein *BindingNavigator*-Steuerelement aus der Toolbox zum Formular hinzu.
2. Stellen Sie sicher, dass das unterhalb des Formulars sichtbare *BindingNavigator*-Steuerelement markiert ist und setzen Sie dessen *Dock*-Eigenschaft auf *Bottom*. Dies stellt sicher, dass das Steuerelement am unteren Formlarrand verankert wird.
3. Setzen Sie anschließend die Eigenschaft *BindingSource* des *BindingNavigator*-Steuerelements auf den Wert der Datenquelle. Das betreffende Listefeld wird den Wert des *BindingSource*-Objekts, welches die Anbindung an die Datenbanktabelle übernimmt, zur Anwahl anbieten.

Wenn Sie dann das Projekt übersetzen, wird der Inhalt der Tabelle im *DataGridView*-Steuerelement aufgelistet. Über die am unteren Rand eingeblendete Navigationsleiste (Abbildung 15.14) kann der Benutzer dann zwischen den Datensätzen blättern.

Hinweis

Sie finden das erweiterte Projektbeispiel im Ordner `\Beisp\Kap15\SimpleADOSQL-Navi` auf der Begleit-CD. Wenn Sie den Inhalt des Ordners auf die Festplatte kopieren und dann das Beispiel in der Entwicklungsumgebung laden, lässt es sich ausführen. Beachten Sie aber, dass die zwei in der Navigationsleiste enthaltenen Schaltflächen, um neue Datensätze hinzuzufügen oder bestehende Datensätze zu löschen, nur auf die Anzeige im *DataGridView* wirken. Um die Datensätze bei Änderungen in der Datenbank zu aktualisieren, müssten Sie in den Ereignisbehandlungsroutinen der Navigations-Schaltflächen Anweisungen zum Zurückspeichern der Änderungen in die Datenbanktabelle hinzufügen. Oder Sie nehmen eine eigene Schaltfläche *Speichern* in der Navigationsleiste auf und versehen deren *Click*-Ereignisbehandlungsroutine mit dem entsprechenden Code. Die Anweisungen zum Aufruf der *Update()*-Methode wurden auf den vorhergehenden Seiten im Beispiel *ADO_SQL_Test* vorgestellt.

15.3.3 So binden Sie beliebige Steuerelemente beim Entwurf

In den beiden obigen Beispielen wurde gezeigt, wie einfach das Binden eines *DataGrid-View*-Steuerelements an eine Datenquelle sowie die Erweiterung um eine Navigationsleiste ist. .NET 2.0 erlaubt Ihnen aber beliebige Steuerelemente bereits zur Entwurfszeit an Datenquellen zu binden. Nachfolgend wird die prinzipielle Vorgehensweise am Beispiel einer Access-Datenbankdatei skizziert. Diese enthält die Tabelle *Users* mit den Feldern *Id*, *Name* und *PersID*. Diese Datenbank soll zur Entwurfszeit in das Projekt eingebunden und die Tabellenfelder an Textfelder eines Formulars gebunden werden (Abbildung 15.15).

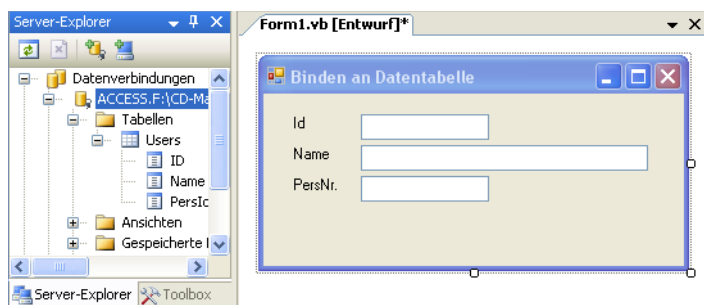


Abbildung 15.15: Tabellenstruktur und Formular zur Darstellung der Tabellendaten

Zuerst muss eine Verbindung zur Datenquelle eingerichtet werden. Die Entwicklungsumgebung bietet dabei zwei Ansätze zum Anbinden der Datenquelle an das Projekt. Die einfachste Variante besteht darin, die Datenbankdatei direkt im Projekt aufzunehmen (dies wurde bereits in den früheren Beispielen des Kapitels skizziert).

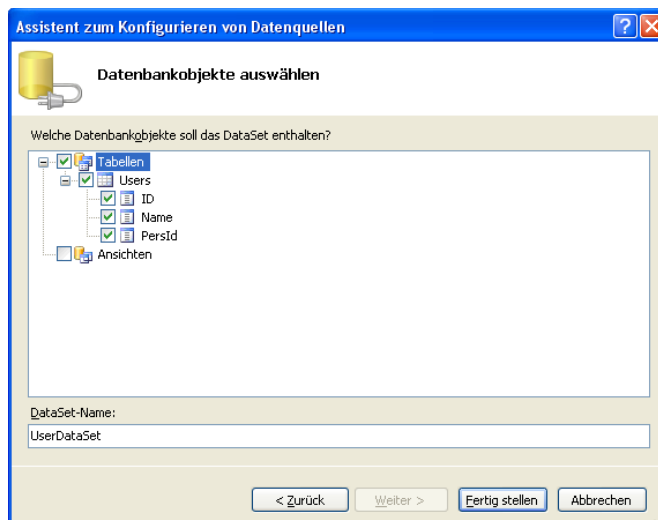


Abbildung 15.16: Auswahl des Datenbankobjekts für das DataSet

Dieser Ansatz hat den Vorteil, dass die Entwicklungsumgebung beim Erstellen der Anwendung die Datenbankdatei mit in das Programmverzeichnis übernimmt und bei der Weitergabe eines Projekts berücksichtigt. Gleichzeitig legt der Assistent zur Konfiguration der Datenquelle bereits die benötigten Verbindungselemente im Projekt an und nimmt auch die notwendigen Einstellungen vor. Hier nochmals die erforderlichen Schritte, demonstriert am Beispiel einer Microsoft Access-Datenbank:

1. Legen Sie ein Windows-Projekt mit dem in Abbildung 15.15 gezeigten Formular an und fügen Sie die Steuerelemente (Label und Textfelder) hinzu.
2. Falls die Datenbankdatei noch nicht existiert, legen Sie diese im Fenster des Server-Explorer (bzw. Datenbank-Explorer) an, fügen die Tabelle hinzu und ergänzen diese ggf. mit einigen Daten. Die notwendigen Schritte sind am Kapitelanfang skizziert. Im aktuellen Projektbeispiel wird eine bereits bestehende Access-Datenbankdatei verwendet.
3. Klicken Sie im Projektmappen-Explorer das Symbol der Projektdatei mit der rechten Maustaste an und wählen Sie im Kontextmenü den Befehl *Hinzufügen/Vorhandenes Element*. Anschließend wählen Sie im Dialogfeld *Vorhandenes Element hinzufügen* die Datenbankdatei per Doppelklick aus. Wird keine Datenbankdatei angezeigt, stellen Sie den Filter *Dateityp* des Dialogfelds auf *Datendateien*.
4. Die Entwicklungsumgebung fügt die Datenbankdatei zum Projekt hinzu. Weiterhin wird ein *DataSet*-Objekt erzeugt. Dieses Objekt enthält den Namen der Datenbank (z.B. *<Datenbankname>DataSet*) und fungiert als Datencache im Arbeitsspeicher. Jetzt wird ein Assistent zur Konfigurierung der Datenquelle gestartet (Abbildung 15.16). Dem Assistenten wurden bereits beim Aufruf die Verbindungsdaten zur Datenquelle von der Entwicklungsumgebung mitgeteilt. Sie brauchen daher nur noch im letzten Dialogschritt die in der Anwendung benötigten Datenbankobjekte anzugeben. Hierzu markieren Sie die Kontrollkästchen der Tabellen bzw. Tabellenfelder, die das *DataSet*-Objekt der Anwendung zur Verfügung stellen soll. Schließen Sie den Dialog des Assistenten über die *Fertig stellen*-Schaltfläche.

Im Projekt sollten dann sowohl die Datenbank als auch ein als *<Datenbankname>DataSet.xsd* bezeichnetes Element im Projektmappen-Explorer zu sehen sein.

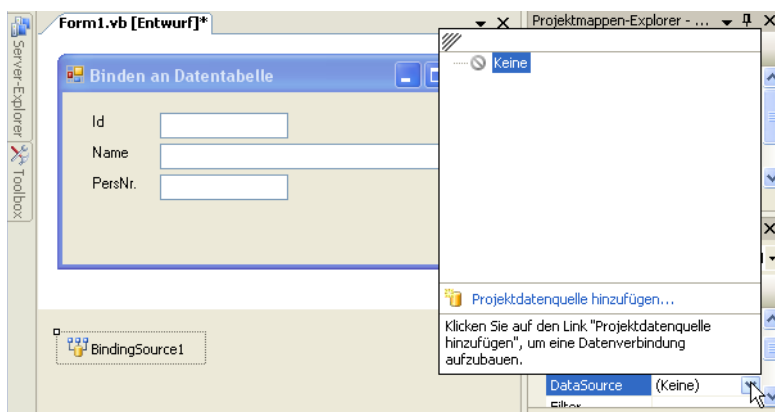


Abbildung 15.17: Projektdatenquelle der BindingSource festlegen

Falls die Datenbank nicht im Projekt eingebunden werden soll bzw. falls Sie lediglich eine Verbindung zur Datenbank (z.B. im Server-Explorer) einrichten möchten, sollten Sie folgende Vorgehensweise verwenden:

1. Legen Sie ein Windows-Projekt mit dem in Abbildung 15.15 gezeigten Formular an und fügen Sie die Steuerelemente (Label und Textfelder) hinzu. Erzeugen Sie ggf. die Datenbankdatei mit den benötigten Tabellen und fügen Sie die Daten hinzu.
2. Anschließend sollten Sie im Fenster des Server-Explorer (bzw. im Datenbank-Explorer) eine Verbindung zur Datenbank definieren. Die entsprechenden Schritte wurden am Kapitelanfang skizziert.

Nun gilt es, das Projekt an die Datenquelle zu binden. Sie können für beide Varianten die folgende Vorgehensweise wählen:

1. Markieren Sie das Steuerelement im Formular (z.B. ein Textfeld) und expandieren Sie in dessen Eigenschaftsfenster den (*DataBindings*)-Knoten. Klicken Sie danach auf die Schaltfläche der *Text*-Eigenschaft.

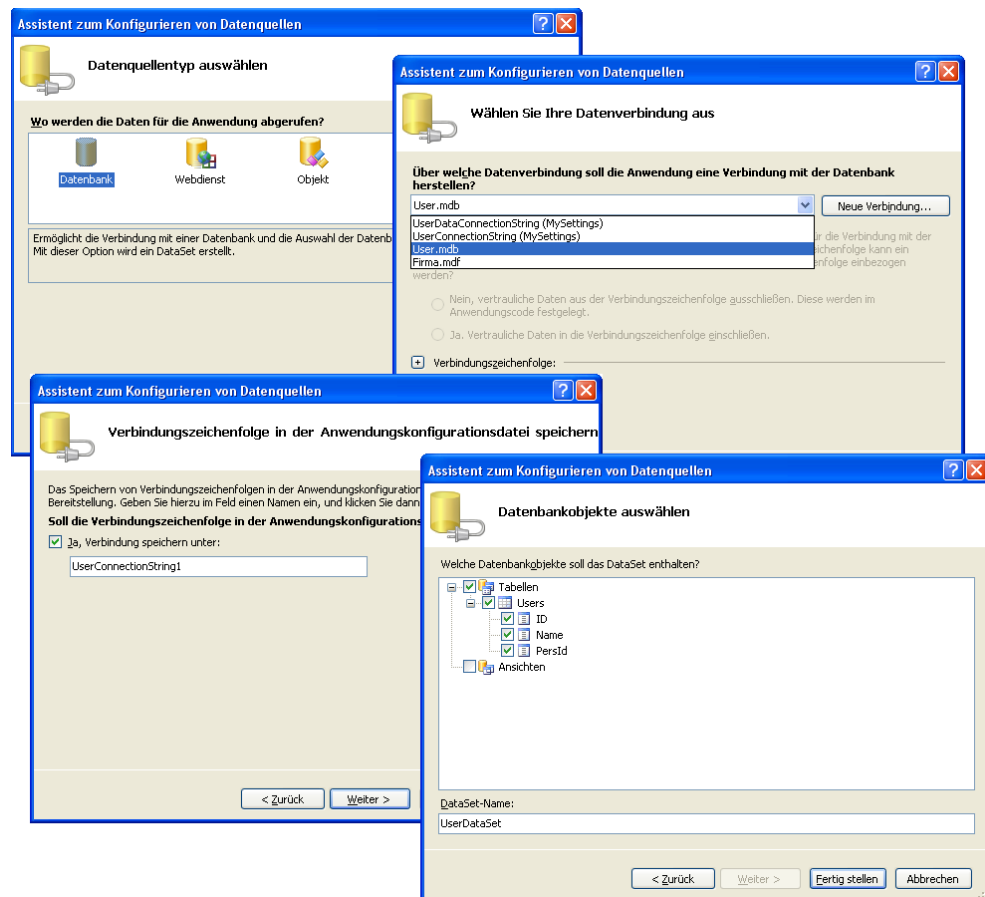


Abbildung 15.18: Konfigurieren der Datenquelle

2. Sobald Sie die Schaltfläche der Eigenschaft wählen, öffnet die Entwicklungsumgebung das Fenster des DataSource-UI-Typ-Editor (Abbildung 15.17). Ist noch keine Datenquelle konfiguriert, ist das Fenster noch leer. Klicken Sie auf den Hyperlink *Projektdatenquelle hinzufügen* und warten Sie, bis der Assistent zum Konfigurieren von Datenquellen erscheint.
3. Wählen Sie im Dialogfeld *Datenquellentyp auswählen* die Option *Datenbank* und klicken Sie auf die Schaltfläche *Weiter* (Abbildung 15.18, links oben).
4. Wählen Sie im Dialogfeld *Wählen Sie Ihre Datenverbindung aus* über das Listefeld die von Ihnen eingerichtete Datenbankverbindung und klicken Sie auf die Schaltfläche *Weiter* (Abbildung 15.18, rechts oben). Bei Bedarf können Sie über die Schaltfläche *Neue Verbindung* auch eine neue Datenbankverbindung erstellen.
5. Belassen Sie im Folgedialog die Markierung der Option *Ja, Verbindung speichern unter* und den im Textfeld eingetragenen Namen (Abbildung 15.18, links unten). Der Assistent speichert dann die Verbindungszeichenfolge für die Datenquelle in der Anwendungskonfigurationsdatei. Sie können den Wert dieser Variablen jederzeit über die Projekteigenschaften (Seite *Einstellungen*) ansehen und verändern. Zudem besteht die Möglichkeit, der Eigenschaft *DataSource* des *BindingSource*-Objekts eine neue Datenquelle zuzuweisen. Klicken Sie auf die Schaltfläche *Weiter*.
6. Wählen Sie im letzten Dialogschritt *Datenbankobjekte auswählen* (Abbildung 15.18, rechts unten) die gewünschten Tabellen für das DataSet aus und klicken Sie auf die Schaltfläche *Fertig stellen*.

Mit dieser Schrittfolge wird das *DataSet*, der Tabellenadapter für die Datenverbindung und das *BindingSource*-Objekt mit der Verbindung zur Datenquelle zum Formular hinzugefügt.

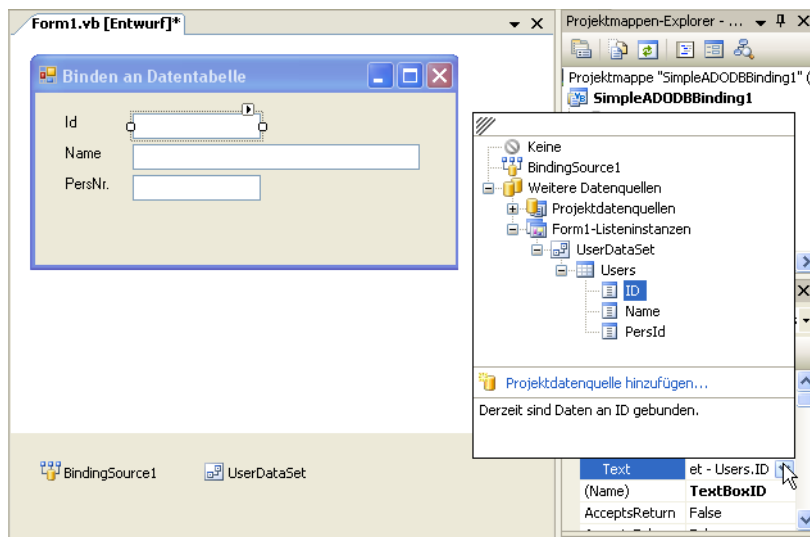


Abbildung 15.19: Binden eines Steuerelements an ein Tabellenfeld der Datenquelle

7. Falls bereits ein *BindingSource*-Objekt konfiguriert wurde oder eine Datenquelle im *DataSource*-UI-Typ-Editor aufgeführt ist, können Sie den Zweig der *BindingSource* expandieren und den Namen des Felds auswählen, an welches das Steuerelement gebunden werden soll (Abbildung 15.19).
8. Wiederholen Sie diese Schritte für alle Steuerelemente, die an Tabellenfelder gebunden werden sollen.

Bei diesen Schritten fügt der *DataSource*-UI-Typ-Editor nicht nur die benötigten Objekte zum Formular hinzu, sondern es wird in der *Load*-Ereignisbehandlungsprozedur eine Anweisung zum Aufruf der *Fill()*-Methode, mit dem der *TableAdapter* mit Daten gefüllt wird, hinterlegt. Um im Formular zwischen den Datensätzen navigieren zu können, lässt sich noch ein *BindingNavigator*-Steuerelement aus der Toolbox zum Formular hinzufügen und über die *Dock*-Eigenschaft am unteren Rand andocken. Setzen Sie die Eigenschaft auf den Namen des *BindingSource*-Steuerelements, welches die Bindung an die Datenbank übernimmt.

Hinweis

Sie finden zwei Projekte, die diese Technik nutzen, auf der Begleit-CD. Beim Projekt im Ordner `\Beisp\Kap15\SimpleADODBBinding` ist die Access-Datenbank *User.mdb* Bestandteil des Projekts. Die Datenbank wird daher beim Erzeugen des Projekts in das Programmverzeichnis kopiert. Beim Projekt im Ordner `\Beisp\Kap15\SimpleADODBBinding1` befindet sich die Datenbank in der Projektmappe, wurde aber nicht zum Projekt hinzugefügt. Es muss daher eine (ODBC-)Verbindung zur Datenbank im Server-Explorer (bzw. im Datenbank-Explorer) angelegt werden. Anschließend ist das *BindingSource*-Steuerelement über die *DataSource*-Eigenschaft auf die Datenquelle zu setzen. Dadurch wird der Verbindungsstring an den Pfad zur Datenbank angepasst. Erst danach kann das Projekt übersetzt und getestet werden.

15.4 Datenbankzugriffe mit ADO.NET per Programm

Microsoft stellt über ADO.NET die notwendigen Funktionen zum Zugriff auf Datenbanken oder Datenquellen bereit. Sie können, wie auf den vorhergehenden Seiten dieses Kapitels skizziert, den *DataSet*-Designer und die Assistenten zur Konfigurierung von Datenquellen verwenden, um Funktionen zur Anzeige von Datenbanktabellen in Formularen zu hinterlegen. Oder Sie verwenden den nachfolgend gezeigten Ansatz, die Funktionen zum Zugriff auf Datenbanktabellen direkt im Programmcode unter Nutzung der ADO.NET-Funktionen zu realisieren.

15.4.1 Auslesen einer Tabelle aus einer Access-Datenbank

In einem ersten Beispiel soll nun untersucht werden, wie sich eine Datenbanktabelle abfragen lässt. Als Datenbank wird jetzt eine lokale Access-Datenbank *UserData.mdb* benutzt, die im Unterordner `\Daten` des Programmordners hinterlegt ist. Diese Daten-

bank enthält nur eine Tabelle mit dem Namen *Users*. Abbildung 15.20 zeigt einen Auszug aus dieser Tabelle mit den gespeicherten Daten sowie den Feldnamen.

Users: Abfrage(F:\...\UserData.mdb)			
	ID	Name	PersId
►	1	Born	123
	2	Brown	234
	3	Miller	343
	4	Bush	345
	11	Born G	12355

Abbildung 15.20: Struktur der Tabelle Users mit Daten

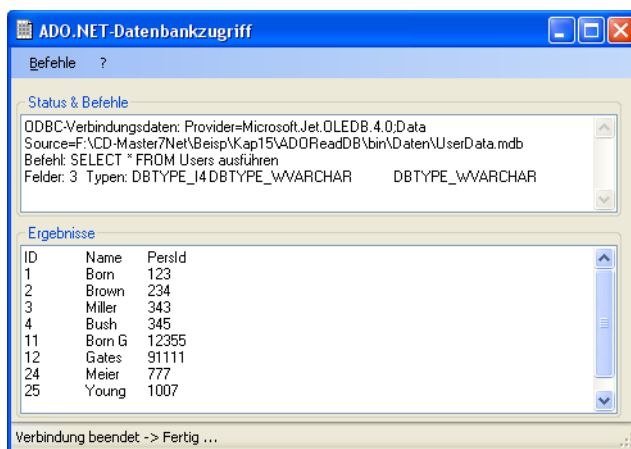


Abbildung 15.21: Anzeige des Tabelleninhalts

Das Feld *ID* der Tabelle *Users* ist dabei ein Autofeld, welches einen laufenden Index führt. Die beiden Felder *Name* und *PersID* sind vom Datentyp *String*. Die Datenbank soll nun in einem sehr einfachen Programm über den OLEDB-Provider »Microsoft.Jet.OLEDB.4.0« abgefragt werden. Die Daten sind in einem Formular als Text auszugeben. Zudem soll das Programm zu Kontrollzwecken die an den Datenbank-Provider übergebenen Befehle anzeigen (Abbildung 15.21). Das Formular wird mit zwei Textfeldern realisiert, wobei das obere den Status und die Befehle anzeigt. Im unteren Textfeld werden die Ergebnisse der Tabelle dargestellt. Das Formular lässt sich im Formulardesigner interaktiv entwerfen. Die Menüleiste bietet im Menü *Befehle* die Einträge *Neu* (löscht die Textfelder) und *Access-Datenbank abrufen* (fragt die Datenbank neu ab und zeigt die Daten).

Der Zugriff auf die Tabelle der Datenbank läuft gemäß der obigen Darstellung in mehreren Schritten ab. Der zum Aufbau der Verbindung und zum Abrufen der Daten benötigte Code wurde in der *Click*-Ereignisbehandlungsroutine des Menübefehls *Access-Datenbank abrufen* hinterlegt. Um mit der Datenbank arbeiten zu können, muss zuerst eine Verbindung über den Datenbankprovider aufgebaut werden. Im Programm wird daher die Verbindungszeichenfolge benötigt, über die der Provider eine Verbindung zur Datenquelle aufbauen kann. Falls der Pfad zur Datenquelle bekannt ist, kann die Verbindungszeichenfolge direkt unter Verwendung von einer Pfadangabe erstellt werden.

Befindet sich die Datenbankdatei beispielsweise im Unterordner *\Daten* des aktuellen Programmordners, ließe sich zunächst der Programmpfad zur Datenquelle ermitteln.

```
Dim file As String = GetPath() & "Daten\UserData.mdb"
```

Die anwendungsspezifische Funktion *GetPath()* ermittelt den Programmpfad zur Programmdatei und hängt den relativen Pfad zur Datenbankdatei an. Anschließend wird der Befehl zum Öffnen der Verbindung zwischen Datenbank und dem .NET-Datenbankprovider vereinbart. Bei einer Access-Datenbank ließe sich z.B. folgender Befehl verwenden:

```
Dim con As String = _  
    "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & file
```

Hinter dem Schlüsselwort *Provider* ist die Angabe des Datenbankproviders erforderlich. Hier wird der Microsoft Jet-Datenbankprovider benutzt. Das Schlüsselwort *Data Source* leitet dann die Angabe über die Lage der Datenbank ein. Hier werden Pfad und Dateiname der Datenbankdatei aus der Variable *file* übernommen. Mit diesen Verbindungsinformationen kann nun eine neue Instanz des *OleDb.OleDbConnection*-Objekts angelegt werden:

```
Dim oDB As OleDb.OleDbConnection = New OleDb.OleDbConnection(con)  
oDB.Open()
```

Die *Open*-Methode öffnet dann die Verbindung zur Datenquelle. Existiert die angegebene Datenbank bzw. Datei nicht oder enthält der *Connection*-Befehl Fehler, löst dies einen Laufzeitfehler aus.

Hinweis

Falls Sie mit einer anderen Datenbank arbeiten möchten, müssen Sie nur den *Connection*-String anpassen und an Stelle des *OleDb.OleDbCommand*-Objekts ggf. ein anderes Objekt (z.B. das *SqlCommand*-Objekt) verwenden.

```
Dim oCon As SqlConnection = _  
    New SqlConnection ("server=localhost;uid=sa;" & _  
        "pwd=xx;database=northwind")  
oCon.Open() ' öffnen  
...  
oCon.Close() ' schließen
```

Die Anweisungen zur Abfrage der Daten aus der Datenbanktabelle bleiben dagegen gleich.

Steht die Verbindung zur Datenbank, ist ein Befehl zur Abfrage der Daten zu vereinbaren. Die Abfragen werden dabei als SQL-Befehle formuliert. Die folgende Anweisung erzeugt über die *CreateCommand*-Methode ein *OleDbCommand*-Objekt, welches dann über die Eigenschaft *CommandText* mit dem auszuführenden SQL-Befehl versehen wird.

```
Dim oCmd As OleDb.OleDbCommand = oDB.CreateCommand()
oCmd.CommandText = "SELECT * FROM Users"
```

Zusätzlich bietet dieses Objekt die *ExecuteReader*-Methode, mit der sich dann die Abfrage ausführen und das Ergebnis in einer *OleDb.OleDbDataReader*-Objektvariablen hinterlegen lässt.

```
Dim oReader As OleDb.OleDbDataReader = oCmd.ExecuteReader()
```

Die Objektvariable *oReader* (Instanz der Klasse *OleDbDataReader*) enthält nach dem Ausführen der *ExecuteReader*-Methode die Tabelle samt Strukturinformationen. Die Eigenschaft *FieldCount* liefert die Zahl der Felder in der Tabelle. Die einzelnen Spalten (Felder) einer Tabelle lassen sich über Indexwerte von 0 bis *count-1* ansprechen. Die Methode *GetName(i)* des *oReader*-Objekts liefert den Namen des betreffenden Feldes. Die Methode *GetDataTypeName(i)* des gleichen Objekts gibt den internen Datentyp des Datenbankfelds zurück.

Hinweis

Beim Zugriff auf den Microsoft SQL Server benutzen Sie die folgenden Anweisungen, um das *Reader*-Objekt anzulegen:

```
Dim oSqlCommand As SqlCommand = _
    New SqlCommand("Select * From Users", oCon)
Dim oReader as SqlDataReader = oSqlCommand.ExecuteReader()
```

Sobald das *Reader*-Objekt bereitsteht, können Sie auf die Datensätze zugreifen.

Die *Read()*-Methode des *oReader*-Objekts erlaubt jeweils zum nächsten Datensatz zu gehen. Ein Zugriff auf die Felder eines Datensatzes ist über die *Item()*-Auflistung möglich. Diese Auflistung liest die Daten der Felder eines Datensatzes im internen Format der Datenbank. Alternativ können Sie diese Daten auch direkt über verschiedene Methoden des *oReader*-Objekts wie *GetByte()*, *GetBytes()*, *GetDecimal()* etc. einlesen. Dann müssen die Datentypen der Felder aber mit dem Typ der Funktion übereinstimmen, da andernfalls falsche Ergebnisse geliefert werden. Näheres zu diesen Methoden finden Sie in der Hilfe zum .NET Framework.

Die folgende Schleife zeigt, wie alle Sätze einer Tabelle durchlaufen werden. Dabei werden die Felder der Tabelle über die *Item()*-Methode gelesen, über *ToString()* gewandelt und dem Textfeld angehängt:

```
Do While oReader.Read() ' Datensätze durchlaufen
    For i = 0 To CInt(oReader.FieldCount) - 1
        Me.TextBox2.AppendText(oReader.Item(i).ToString & vbTab)
    Next i
Loop
```

Abschließend werden der Reader sowie die geöffnete Verbindung über den Aufruf der *Close()*-Methode geschlossen:

```
oReader.Close()
oDB.Close()
```

Nachfolgend finden Sie den gesamten Code der *Click*-Ereignisbehandlungsroutine des Befehls *Befehle/Access-Datenbank abrufen*, der eine Verbindung zur Datenbank aufbaut, die Daten der Tabelle *Users* über einen SQL-Befehl abrufen und dann die Ergebnisse in Textfeldern anzeigt:

```
Private Sub MenuItemMDB_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MenuItemMDB.Click
' Access-Datenbankverbindung aufbauen und Werte zeigen
' Benötigt die Datenbank im angegebenen Verzeichnis
Dim i As Integer      ' für Schleifen
Dim file As String = GetPath() & "Daten\UserData.mdb"
Dim con As String = _
    "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & file

' Verbindungsobjekt instantiieren
Dim oDB As OleDb.OleDbConnection = New OleDb.OleDbConnection(con)
oDB.Open()          ' Öffnen der Verbindung zur Datenbank

Me.StatusBar1.Text = "Verbindung geöffnet ..."
Me.TextBox1.Text = " OLEDB-Verbindungsdaten: " & con & vbCrLf

' Befehl zur Datenbankabfrage
Dim oCmd As OleDb.OleDbCommand = oDB.CreateCommand()
oCmd.CommandText = "SELECT * FROM Users"

Me.StatusBar1.Text = "Daten abrufen ..."

' Befehl mit dem OleDbDataReader ausführen
Dim oReader As OleDb.OleDbDataReader = oCmd.ExecuteReader()
Me.TextBox1.AppendText("Befehl: " & oCmd.CommandText & _
    " ausführen" & vbCrLf)
Me.TextBox1.AppendText("Felder: " & oReader.FieldCount & vbTab)

Me.TextBox2.Text = "" ' Ergebnisfeld leeren
Me.StatusBar1.Text = "Daten lesen ..."
Me.TextBox1.AppendText("Typen: ")

For i = 0 To CInt(oReader.FieldCount) - 1 ' Feldnamen & Typen
    Me.TextBox1.AppendText(oReader.GetDataTypeName(i) & vbTab)
' Spaltenköpfe schreiben
```



```

Me.TextBox2.AppendText(oReader.GetName(i) & vbTab)
Next i
Me.TextBox1.AppendText(vbCrLf) ' Zeilenumbrüche einfügen
Me.TextBox2.AppendText(vbCrLf)

Do While oReader.Read() ' Datensätze durchlaufen
    For i = 0 To CInt(oReader.FieldCount) - 1
        Me.TextBox2.AppendText(oReader.Item(i).ToString & vbTab)
    Next i
    Me.TextBox2.AppendText(vbCrLf)
Loop

oReader.Close()
oDB.Close()
Me.StatusBar1.Text = "Verbindung beendet -> Fertig ..."
End Sub

```

Hinweis

Die Projektdateien finden Sie im Ordner `\Beisp\Kap15\ADOReadDB` auf der Begleit-CD. Das Unterverzeichnis `\Bin` enthält einen Unterordner `\Daten` mit der benötigten Access-Datenbank. In diesem Projekt wird der absolute Pfad zur Datenquelle über die `GetPath()`-Funktion ermittelt. Dann werden Pfad und Dateiname zur Verbindungszeichenfolge hinzugefügt. In .NET ist aber noch ein alternativer Ansatz möglich, um die Verbindungszeichenfolge zum Zugriff auf die Datenbankdatei zu erzeugen. Sobald Sie eine Datenbank in ein Projekt einbinden, wird vom Assistenten zum Konfigurieren der Datenquelle eine Anwendungsvariable mit der Verbindungszeichenfolge in den Anwendungseigenschaften hinterlegt. Die Eigenschaften sind übrigens im Element `Settings.settings` des Projektmappen-Explorers gespeichert. Die Projektdateien im Ordner `\Beisp\Kap15\ADOReadDB1` der Begleit-CD demonstrieren, wie sich die Verbindungszeichenfolge aus der Anwendungsvariablen `AccessDatenbank` zum Verbindungsaufbau nutzen lässt. Weitere Details entnehmen Sie dem Quellcode des betreffenden Beispiels.

15.4.2 Daten programmgesteuert in einem DataGridView anzeigen

In einem weiteren Beispiel soll die Tabelle der Datenbank in einer `DataGridView`-Anzeige ausgegeben werden (Abbildung 15.22). Das betreffende Formular erlaubt in einem Textfeld die Eingabe einer SQL-Anweisung, um Daten aus der Tabelle `Users` der Access-Datenbank abzurufen. Daher wird das `DataGridView`-Steuerelement nicht (wie weiter oben gezeigt) direkt an die Tabelle angebunden, sondern programmgesteuert mit dem Ergebnis der Abfrage gefüllt.

Die Tabelle *Users* besitzt dabei die bereits oben erwähnten Felder *ID*, *Name* und *PersId*. Beim Programmstart wird der SQL-Befehl `SELECT * FROM Users` zum Abrufen aller Felder der Tabelle mit allen Datensätzen im Textfeld eingeblendet und die *DataGridView*-Anzeige ist leer. Sie können nun bei Bedarf den SQL-Befehl modifizieren, um die anzuzeigenden Daten zu filtern. Die Angabe `SELECT ID, Name FROM Users Where ID > 3` bewirkt z.B., dass nur Daten für zwei Felder angezeigt werden. Zudem werden alle Datensätze ausgefiltert, bei denen der Wert des Felds *PersId* größer 3 ist. Die Schaltfläche *Reset-SQL* fügt den obigen SQL-Befehl `SELECT * FROM Users` erneut im Textfeld ein – Sie können also jederzeit die komplette Tabelle erneut abrufen.

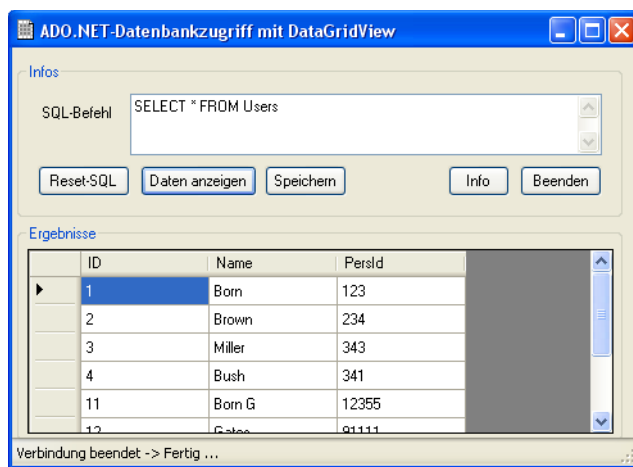


Abbildung 15.22: Anzeige von Daten in einem DataGridView

Die Schaltfläche *Daten anzeigen* veranlasst das Programm, eine Verbindung zur Datenbank (hier die Access-Datenbank *UserData.mdb* im Unterverzeichnis *\Daten*) über den Microsoft-Jet-DB-Treiber herzustellen und über diese OLEDB-Verbindung die Daten des SQL-Befehls abzurufen. Die Anwendung ist dabei so realisiert, dass der komplette Datensatz an das *DataGridView*-Element weitergereicht wird und dort zur Anzeige gelangt. Sie können die Schaltfläche *Daten anzeigen* durchaus mehrfach anklicken, um die Anzeige durch Lesen der Daten zu aktualisieren. Variieren Sie den SQL-Befehl, werden die geänderten Daten angezeigt – wobei aber die Struktur des *DataGridView* erhalten bleibt. Nicht relevante Spalten werden durch Einträge eines Werts »(Null)« als ungültig markiert. Die Statusleiste des Fensters informiert Sie übrigens über den Ablauf einer Abfrage.

Im *DataGridView*-Fenster lassen sich die angezeigten Daten der Spalten *Name* und *PersId* editieren. Sie können beispielsweise einen Wert ändern. Klicken Sie auf den Zeilenkopf, wird dieser durch ein kleines Dreieck markiert. Drücken Sie die `[Entf]`-Taste, wird der Datensatz aus der *DataGridView*-Anzeige entfernt. Der letzte Eintrag im *DataGridView* ist ein Leersatz. Klicken Sie auf diese Zeile und geben Sie Daten ein, wird ein neuer Datensatz in der Tabelle angelegt. Mittels der Schaltfläche *Store* lassen sich im *DataGridView*-Element vorgenommene Änderungen in der Tabelle der Datenbank speichern.

Zur Implementierung der Anwendung wurde in der Entwicklungsumgebung ein Projekt als Windows-Anwendung erzeugt und das standardmäßig vorhandene Formular mit den in Abbildung 15.22 gezeigten Schaltflächen, dem Textfeld zur Anzeige/Eingabe der SQL-Befehle, einem *StatusStrip*-Element mit einem *ToolStripStatusLabel*-Steuerelement sowie mit einem *DataGridView*-Element ergänzt. Der Code zum Zugriff auf die Tabelle der Datenbank, zum Füllen des *DataGridView*-Elements und zur Aktualisierung der Datenbanktabelle wurde in den *Click*-Ereignisbehandlungsprozeduren der Schaltflächen hinterlegt. Das *Click*-Ereignis der Schaltfläche *Reset* speichert lediglich den SQL-Befehl im Textfeld und enthält folgende Anweisung:

```
Me.TextBox2.Text = "SELECT * FROM Users"
```

Der gleiche Wert wird der Textbox in der *Load*-Ereignisbehandlungsprozedur des Formulars zugewiesen. Die Ereignisbehandlungsroutine für die Schaltfläche *Daten anzeigen* wurde aus dem vorherigen Beispiel abgeleitet. Die folgenden Anweisungen erstellen eine Instanz des *OleDbConnection*-Objekts und legen den SQL-Befehl fest:

```
Dim con As String = _  
    "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & file  
Dim sCmd As String = "SELECT * FROM Users"  
Dim oDB As OleDb.OleDbConnection = New OleDb.OleDbConnection(con)
```

Die Verbindung zur Datenbank wird über die Anweisung *oDB.Open()* geöffnet. Anschließend gilt es, die Abfrage zu gestalten und das *DataGridView*-Steuerelement an die bei der Abfrage zurückgelieferte Tabelle zu binden. Den folgenden Befehl zum Erstellen einer neuen Instanz eines an die geöffnete Datenbank angebundenen *OleDbCommand*-Objekts kennen Sie bereits aus dem vorherigen Beispiel:

```
Dim oCmd As OleDb.OleDbCommand = New OleDb.OleDbCommand(sCmd, oDB)
```

Die Anbindung der Daten an das *DataGridView* erfolgt über einen Datenadapter, der in Form eines *OleDbDataAdapter*-Objekts bereitgestellt wird und ein *DataSet* enthält. Hierzu werden im Programmkopf die beiden folgenden Objektvariablen deklariert:

```
Dim oDataSet As New DataSet("User")  
Dim oAdapter As OleDb.OleDbDataAdapter
```

Dann wird eine neue Instanz des *OleDbDataAdapter*-Objekts erstellt, wobei das *OleDbCommand*-Objekt als Parameter zu übergeben ist. Aus diesem Objekt ermittelt der Adapter die Abfrage:

```
oAdapter = New OleDb.OleDbDataAdapter(oCmd)  
oAdapter.Fill(oDataSet, "User")
```

Die zweite Anweisung benutzt die *Fill*-Methode, um das *DataSet* zu füllen und mit dem Namen "User" zu versehen. Jetzt braucht das *DataSet* nur noch an das Steuerelement *DataGrid* gebunden zu werden:

```
Me.DataGrid1.DataSource = oDataSet.Tables("User")
```

Hierzu wird der *DataSource*-Eigenschaft des Steuerelements einfach eine Objektreferenz zugewiesen. Diese Objektreferenz wird über *oDataSet.Tables* bereitgestellt. Als Parameter ist der Name der Tabelle zu übergeben (dies erlaubt mehrere Tabellen pro *DataSet* zu verwalten).

Update der Datenbanktabelle aus dem DataGridView

Zur Aktualisierung der Datenbanktabelle mit den Änderungen im *DataGridView* müssen alle Änderungen aus dem *DataSet* per *Update*-Kommando in die Datenbank zurückgeschrieben werden. Das größte Problem ist dabei, den korrekten SQL-Befehl für das Update zu ermitteln. Glücklicherweise bietet ADO.NET über das *OleDb.OleDbCommandBuilder*-Objekt die Möglichkeit, einen SQL-Befehl zur Aktualisierung der Datenbank automatisch aus den (im *DataAdapter*-Objekt gespeicherten *DataTable*-Objekt) vorgenommenen Änderungen erstellen zu lassen. Der nachfolgende Code zeigt den Inhalt der *Click*-Ereignisbehandlungsprozedur der Schaltfläche *Store*:

```
Private Sub ButtonUpdate_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ButtonUpdate.Click
    ' Access-Tabelle "Users" aus DataGrid aktualisieren
    oDB.Open()           ' Öffnen der Verbindung zur Datenbank
    ' Befehl vom CommandBuilder automatisch erstellen lassen
    Dim oCmd As OleDb.OleDbCommandBuilder = New _
        OleDb.OleDbCommandBuilder(oAdapter)
    ' vom Builder erstellter SQL-Befehl anzeigen
    Me.TextBox2.Text = oCmd.GetUpdateCommand.CommandText
    Try ' DataSet in Tabelle zurückschreiben
        oAdapter.Update(oDataSet, "User")
        Me.StatusBar1.Text = "Daten aktualisiert ..."
    Catch ex As Exception
        MessageBox.Show(ex.Message, "Fehler beim Update", _
            MessageBoxButtons.OK, MessageBoxIcon.Information)
    End Try
    oDB.Close()         ' schließen
End Sub
```

Das vom *OleDbCommandBuilder* erstellte Kommando ist über *oCmd* an den *OleDbDataAdapter* angebunden. Dann genügt der Aufruf der *Update*-Methode des *OleDbDataAdapter*-Objekts, um die Änderungen in die Datenbank zurückzuschreiben.

Hinweis

Sie finden die Projektdateien dieses Beispiels im Ordner `\Beisp\Kap15\ADOReadDBGrid` auf der Begleit-CD. Das Unterverzeichnis `\Bin` enthält den Unterordner `\Daten` mit der benötigten Access-Datenbank. Um das Beispiel auszuführen, müssen Sie den kompletten Projektordner in einen Ordner der Festplatte kopieren. Andernfalls lässt sich die Access-Datenbank nicht öffnen bzw. beschreiben. Sie können ein *DataSet*-Steuerelement direkt im Ansicht-Designer zum Formular hinzufügen. In diesem Fall kann die Instantiierung des *DataSet*-Objekts im Code entfallen. Im Ordner `\Beisp\Kap15\ADOReadDBGridExt` ist ein entsprechendes Beispiel enthalten – die betreffende Anweisungszeile wurde im Code von *Form1.vb* auskommentiert. Zudem wird die Verbindungszeichenfolge bei diesem Projekt aus einer Anwendungsvariablen mit *My.Settings.UserDataConnectionString* gelesen und die Datenbankdatei ist direkt im Projekt eingebunden. Weitere Details entnehmen Sie bitte dem Quellcode der betreffenden Beispiele. Hinweise, mit welchen Befehlsstrings Sie eine Anwendung an einen SQL Server binden, finden Sie in der Hilfe zu ADO.NET.

15.4.3 DataSet an Steuerelemente binden und navigieren

Häufig wird man kein komplettes *DataGrid*-Element in einem Anwendungsformular unterbringen, sondern die benötigten Informationen über einzelne Textfelder anzeigen. Dann besteht die Notwendigkeit, das *DataSet*-Objekt so an das betreffende Steuerelement anzubinden, dass der Inhalt des gewünschten Feldes der Datentabelle angezeigt wird. Das zweite Problem besteht darin, dass das Formular eine Möglichkeit zur Navigation innerhalb der Datensätze bieten muss. Einen entsprechenden Ansatz, wie sich dies mit den Designern realisieren lässt, findet sich auf den vorherigen Seiten des Kapitels. Sie können diese Funktion aber auch direkt im Programmcode unter Verwendung verschiedener Klassen aus ADO.NET realisieren. Die Vorgehensweise möchte ich an einem einfachen Beispiel demonstrieren. Ein Formular soll immer einen Datensatz der Tabelle *Users* aus der Access-Datenbank `\Daten\UserData.mdb` anzeigen. Über Schaltflächen hat der Benutzer die Möglichkeit, zwischen den jeweiligen Datensätzen zu blättern (Abbildung 15.23).

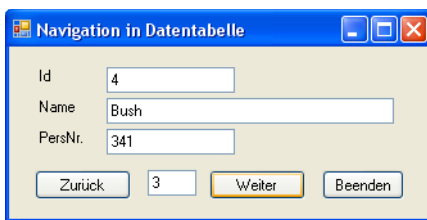


Abbildung 15.23: Anzeige von Daten in einem Formular

Auf eine Aktualisierung der Eingaben in der Datenbank habe ich hier verzichtet. Sie können den auf der vorherigen Seite gezeigten *Update*-Ansatz verwenden.

Datenbankverbindung vorbereiten und DataSet anlegen

In einem ersten Schritt werden die nachfolgend aufgeführten Variablen global in der Klasse deklariert. Dies hat den Vorteil, dass sich aus den einzelnen Ereignisbehandlungsroutinen darauf zugreifen lässt.

```
Dim file As String = GetPath() & "Daten\UserData.mdb"
Dim con As String = _
    "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & file
Dim sCmd As String = "SELECT * FROM Users"
Dim oDB As OleDb.OleDbConnection = New OleDb.OleDbConnection(con)
Dim oDataSet As New DataSet("User")
Dim oAdapter As OleDb.OleDbDataAdapter
Dim oBind As Binding ' Binding-Manager
```

Diese Deklarationen kennen Sie bereits, es werden die Variablen zur Aufnahme der Verbindungsinformationen, ein *DataSet*-Objekt sowie ein *DataAdapter*-Objekt deklariert und teilweise instantiiert. Neu ist lediglich die *oBind*-Variable, die für den Binding-Manager benötigt wird (siehe unten).

Die Verbindungsaufnahme mit der Datenbank erfolgt in der *Load*-Ereignisbehandlungsroutine des Formulars. Dort wird auch das *DataSet* an den *DataAdapter* gebunden.

```
oDB.Open() ' Öffnen der Verbindung zur Datenbank
Try
    ' Befehl zur Datenbankabfrage erstellen
    Dim oCmd As OleDb.OleDbCommand = New OleDb.OleDbCommand(sCmd, oDB)
    oAdapter = New OleDb.OleDbDataAdapter(oCmd) ' Adapter mit Abfrage
    oAdapter.Fill(oDataSet, "User") ' Fülle DataSet mit Tabellendaten
Catch ex As Exception
    MessageBox.Show(ex.Message, "Fehler beim Laden", _
        MessageBoxButtons.OK, MessageBoxIcon.Information)
End Try
```

Diese Schritte kennen Sie bereits aus den vorhergehenden Beispielen.

Steuerelemente an DataSet-Elemente binden

Sobald die Verbindung zur Datenbank steht und das *DataSet*-Objekt mit Daten gefüllt ist, lässt sich die Anbindung an die einzelnen Steuerelemente vornehmen. Hierbei muss jedem Steuerelement das Feld der Datenbanktabelle zugewiesen werden, welches zur Anzeige vorgesehen ist. Hierzu wird die *Add()*-Methode der *DataBindings*-Eigenschaft des betreffenden Steuerelements benutzt. Diese fügt der Auflistung ein *DataSet*-Objekt hin.

```
Me.TextBoxID.DataBindings.Add("Text", _
    oDataSet.Tables("User"), "ID")
```

```
Me.TextBoxName.DataBindings.Add("Text", _
    oDataSet.Tables("User"), "Name")
Me.TextBoxPers.DataBindings.Add("Text", _
    oDataSet.Tables("User"), "PersID")
```

Die obige Codesequenz bindet das *DataSet* mit dem Namen »User« an die drei angegebenen Steuerelemente. Der erste Parameter ist hier auf »Text« gesetzt, um die Werte in die *Text*-Eigenschaft des Steuerelements zu übertragen. Die *Add()*-Methode erwartet im zweiten Parameter den *Tables*-Wert des *DataSet*-Objekts. Die gewünschte Tabelle des *DataSet* wird über einen Index in *Tables* angegeben. Hier wird immer die Tabelle mit dem Namen »User« benutzt. Da die Datenquelle mehrere Felder umfassen kann, wird im dritten Parameter der *Add()*-Methode der Name des gewünschten Felds übergeben. Die obige Sequenz bindet also alle Felder der *User*-Tabelle an die jeweiligen Steuerelemente.

Navigation in den Datensätzen der Datenquelle

Bleibt zum Schluss noch die Frage, wie sich in den Datensätzen der Datenquelle (hier der *DataSet*-Tabelle) bewegen lässt. Beim Aufruf der *Fill()*-Methode wird die Tabelle im *DataSet* ja gefüllt und beim Binden erscheint der erste Datensatz in der Anzeige. Die Navigation in Datensätzen erfolgt in ADO.NET über den Binding-Manager. Dieser wird folgendermaßen deklariert:

```
Dim oBind As Binding = Me.TextBoxID.DataBindings.Item("Text")
```

Hier wird die Objektvariable *oBind* vom Typ *Binding* deklariert und dann mit dem Element *Item("Text")* der *DataBindings*-Auflistung belegt. Das Element *Text* der *DataBindings*-Auflistung des betreffenden Steuerelements wurde gemäß obigen Ausführungen an das *DataSet*-Objekt angebunden. Die Positionierung auf einen Datensatz erfolgt über die Eigenschaft *Position* der *BindingManagerBase*-Eigenschaft des *Binding*-Objekts. Die folgende Anweisung setzt den Datensatzzeiger auf den dritten Datensatz:

```
oBind.BindingManagerBase.Position = 2
```

Im Beispielprogramm habe ich zwei *Click*-Ereignisbehandlungsprozeduren für die Schaltflächen *Zurück* und *Weiter* angelegt, die die *Position*-Eigenschaft um 1 erhöhen oder erniedrigen. Die unterste Position ist auf 0 (1. Datensatz) fixiert, die oberste Grenze wird über die Eigenschaft *Count* der *BindingManagerBase*-Eigenschaft festgelegt. Die folgende Codesequenz zeigt die *Click*-Ereignisbehandlungsprozeduren der beiden Schaltflächen:

```
Private Sub ButtonDown_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ButtonDown.Click
    ' Ein Datensatz runter
    If oBind.BindingManagerBase.Position > 0 Then
        oBind.BindingManagerBase.Position -= 1 ' erhöhen
    End If
    Me.TextBoxCnt.Text = oBind.BindingManagerBase.Position.ToString
```

```

End Sub

Private Sub ButtonUp_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ButtonUp.Click
    ' Ein Datensatz rauf
    If oBind.BindingManagerBase.Position < _
        oBind.BindingManagerBase.Count Then
        oBind.BindingManagerBase.Position += 1 ' erhöhen
    End If
    Me.TextBoxCnt.Text = oBind.BindingManagerBase.Position.ToString
End Sub

```

Die letzte Anweisung der jeweiligen Prozedur schreibt den Wert der aktuellen Position in das Textfeld, welches den Datensatz anzeigt. Alles in allem ist das programmgesteuerte Navigieren in Datensätzen also kein größeres Problem.

Hinweis

Das komplette Projektbeispiel finden Sie auf der Begleit-CD im Ordner *\Beisp\Kap15\ADOREadNavi*. Das Unterverzeichnis *\Bin* enthält den Unterordner mit einer Access-Datenbank. Weitere Details entnehmen Sie dem Quellcode des betreffenden Beispiels. Möchten Sie das *DataSet*-Element bereits zur Designzeit zum Formular hinzufügen und die Textfelder über die *DataBinding.Text*-Eigenschaft an dieses Element binden? Im Ordner *\Beisp\Kap15\ADOREadNaviNeu* finden Sie ein entsprechend modifiziertes Projektbeispiel. Im Projekt wurde die Datenbank direkt eingebunden und ein *BindingSource*-Steuerelement stellt die Verbindung zwischen Datenbank und *DataSet*-Element her. Dies erlaubt dann die Verbindungszeichenfolge zur Laufzeit aus einer Anwendungsvariablen *My.Settings.UserDataConnectionString* zu lesen und die Datenbankdatei zu öffnen. In diesem Projekt wurde zusätzlich ein *ToolStrip*-Steuerelement am unteren Formarrand verankert und mit Navigationsschaltflächen versehen (Abbildung 15.24). Diese Steuerelemente wurden übrigens aus einem *BindingNavigator*-Steuerelement in das *ToolStrip*-Steuerelement kopiert. Die Details zur Implementierung und Anbindung an die zur Designzeit eingefügten Steuerelemente entnehmen Sie bitte dem Quellcode der betreffenden Beispiele. Hinweise, mit welchen Befehlsstrings Sie eine Anwendung an einen SQL Server binden, finden Sie in der Hilfe zu ADO.NET.



Abbildung 15.24: Modifiziertes Formular zur Anzeige von Daten

15.4.4 Erstellen von Berichten

ADO.NET stellt mit dem *ReportViewer*-Steuerelement eine Komponente zur Berichtserstellung in Windows-Anwendungen bereit. Nachfolgend wird an einem einfachen Beispiel skizziert, wie sich ein Bericht auf Basis einer Datenbanktabelle erstellen lässt.

Vorbereiten des Projekts für den Bericht

Ein Bericht lässt sich direkt in einem Windows-Formular integrieren, d.h., beim Aufruf des Formulars wird der Bericht durch die betreffenden Steuerelemente generiert. Aus diesem Grund benötigen Sie ein Projekt, welches bestimmte Elemente enthält. Hierzu führen Sie folgende Schritte aus:

1. Legen Sie ein neues Windows-Projekt mit einem leeren Formular (zur Anzeige des Berichts) in der Entwicklungsumgebung an.
2. Passen Sie ggf. die Projekteinstellungen sowie die Eigenschaften des Formulars (Titel, Abmessungen etc.) an.
3. Danach verwenden Sie im Projektmappen-Explorer den Kontextmenübefehl *Hinzufügen der Projektdatei*, um ein *DataSet*-Element zum Projekt hinzuzufügen.

Sie haben dabei die Möglichkeit, ein leeres *DataSet*-Element über den Kontextmenübefehl *Hinzufügen/Neues Element* im Projekt einzufügen und diesem danach manuell eine Datenverbindung sowie Datenbankobjekte (z.B. Tabellen) zuzuweisen. Die Schritte wurden am Kapitelanfang im Abschnitt »Einbinden beliebiger Datenquellen in das Projekt« skizziert. Soll die Datenbankdatei Bestandteil des Projekts sein, empfiehlt es sich aber, diese (als neue oder bestehende Datenbank) einfach im Projekt aufzunehmen (siehe z.B. Abschnitte »Anlegen einer SQL-Datenbankdatei im Projekt« bzw. »SQL-/Access-Datenbankdatei in das Projekt einbinden« weiter oben). Dann fügt der betreffende Assistent beim Einbinden der Datenbankdatei automatisch ein *DataSet*-Element mit den ausgewählten Datenbankobjekten zum Projekt hinzu.

Entwerfen des Berichts

Das ReportWriter-Steuerelement greift zum Erstellen und Anzeigen auf eine Berichtsvorlage zurück. In einem eigenen Schritt müssen Sie daher in der Entwicklungsumgebung den Bericht entwerfen. Hierzu gehen Sie in folgenden Schritten vor:

1. Klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf das Symbol der Projektdatei und wählen Sie im Kontextmenü die Befehle *Hinzufügen/Neues Element*.
2. Im Dialogfeld *Neues Element* wählen Sie die Vorlage *Bericht* (Abbildung 15.25), passen ggf. den Text im Feld *Name* an und schließen das Dialogfeld über die *Hinzufügen*-Schaltfläche.
3. Anschließend wählen Sie das neu eingefügte *Berichts*-Element im Fenster des Projektmappen-Explorers per Doppelklick an. In der Entwicklungsumgebung wird dann ein Fenster mit der (noch leeren) Berichtsvorlage eingeblendet.
4. Sie müssen nun die gewünschten Elemente wie statische Texte, Felder zum Einblenden von Datenbankwerten, Berechnungen etc. in die Berichtsseite einfügen.

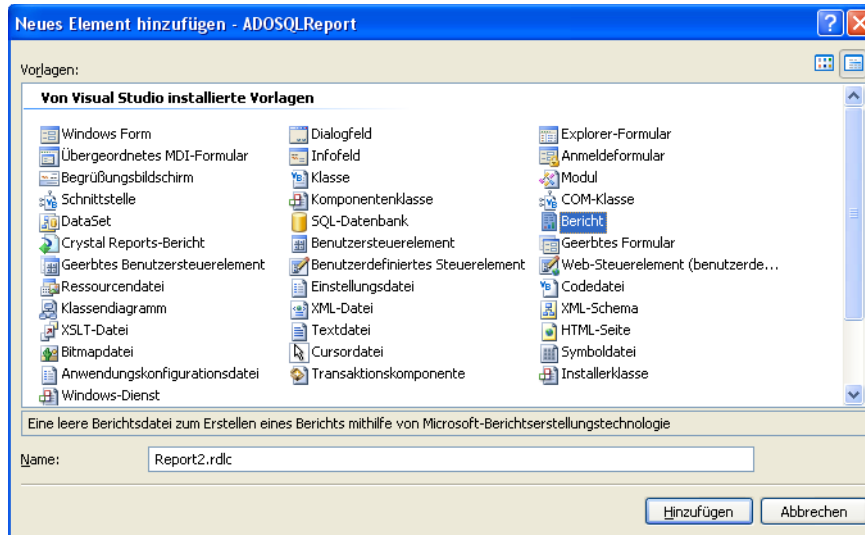


Abbildung 15.25: Einfügen einer Berichtsvorlage zum Projekt

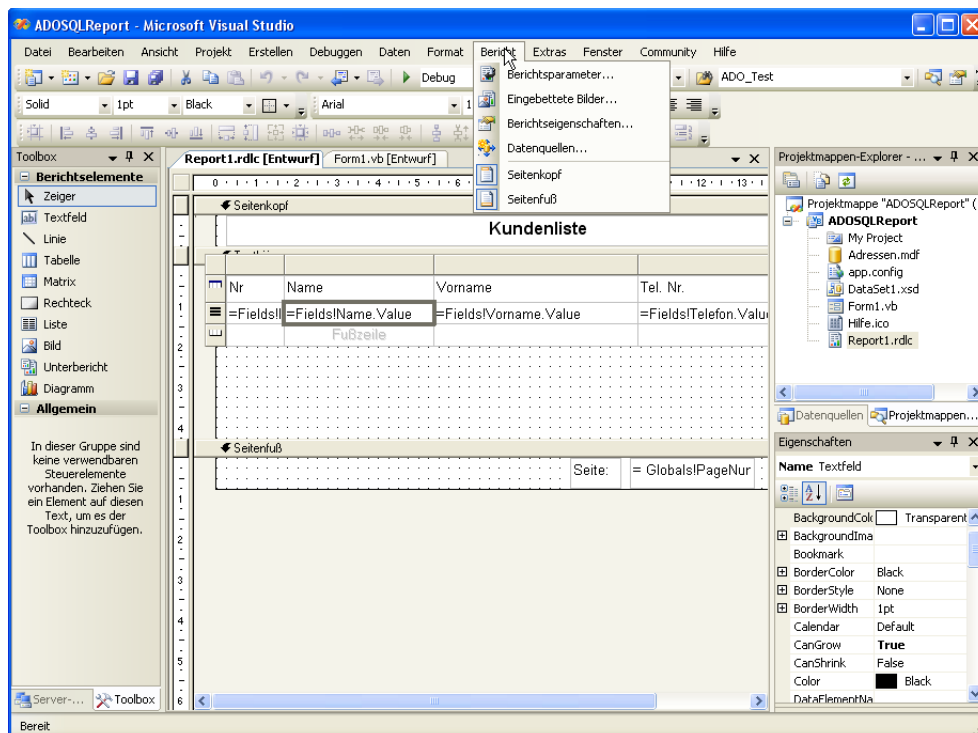


Abbildung 15.26: Entwurf des Berichts im Designer

Abbildung 15.26 zeigt ein Beispiel für eine solche Berichtssseite im Fenster der Entwicklungsumgebung. Ein Bericht besteht dabei aus dem Seitenbereich sowie einem Seitenkopf und einem Seitenfuß. Standardmäßig wird nur der Seitenbereich im Entwurfsfenster eingeblendet. Um den Seitenkopf bzw. -fuß anzeigen und mit Elementen gestalten zu können, müssen Sie das Menü *Bericht* im Fenster der Entwicklungsumgebung öffnen und die Befehle *Seitenkopf* bzw. *Seitenfuß* anklicken. Anschließend können Sie aus der Toolbox die gewünschten statischen Berichtselemente wie Linien, Rechtecke, Bilder, Textfelder, Tabellen, Listen etc. im Berichtsentwurf einfügen. Statische Texte lassen sich z.B. in eingefügten Textfeldern eintippen. Die Schaltflächen der Symbolleisten *Layout*, *Berichtsformatierung* und *Berichtsrahmen* erlauben dabei die betreffenden Elemente zu formatieren (z.B. Schriftgrade von Textfelder, Rahmen um Elemente) oder im Formular auszurichten.

Im Seitenkopf können Sie beispielsweise die Überschrift für den Bericht, ein Firmenlogo etc. hinterlegen. Dynamische Inhalte des Berichts werden über Feldverweise eingebunden. Diese Feldverweise können sich auf Datenbanktabellen oder auf Funktionen (z.B. Seitennummerierung) beziehen.

1. Zum Einfügen der Inhalte von Datenbanktabellen im Bericht wählen Sie einfach den Registerreiter *Datenquellen* (dann wird das betreffende Fenster mit den konfigurierten Datenquellen eingeblendet). Falls der Registerreiter nicht sichtbar ist, müssen Sie im Fenster der Entwicklungsumgebung den Befehl *Datenquellen anzeigen* im Menü *Daten* anwählen.
2. Expandieren Sie im Fenster *Datenquellen* das gewünschte *DataSet*, so dass die darin verwalteten Tabellen oder Datenbankobjekte samt ihren Feldern sichtbar werden.
3. Dann ziehen Sie einfach das Symbol eines Feldelements per Maus aus dem Fenster *Datenquellen* zum Designbereich des Berichts.

Sobald Sie die linke Maustaste loslassen, wird das betreffende Feld im Berichtsentwurf als Verweis eingefügt. In Abbildung 15.27 wurde eine Tabellenstruktur im Seitenbereich des Berichtsentwurfs hinterlegt. In dessen Feldern lassen sich z.B. die Feldverweise auf die Tabellenelemente einfügen.

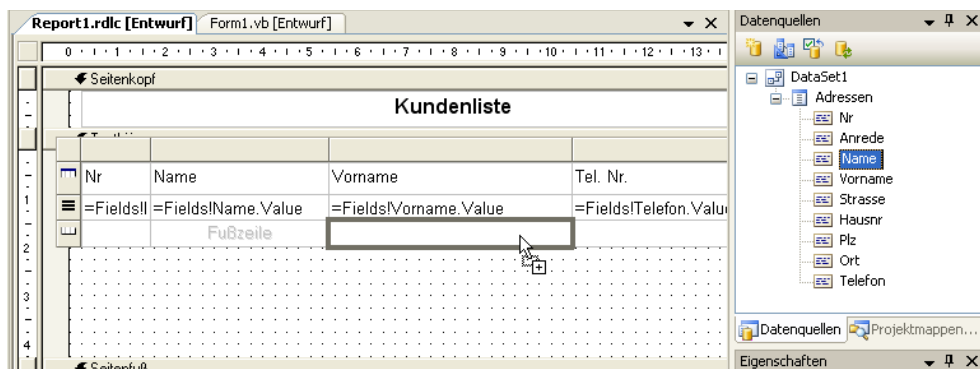


Abbildung 15.27: Einfügen von Datenbankfeldern im Bericht

Möchten Sie den Bericht mit einer Seitennummerierung oder anderen Informationen versehen? Der Designer stellt verschiedene Funktionen bereit, mit denen sich unterschiedliche Daten im Bericht einfügen lassen. Einige Elemente wie z.B. Seitennummern müssen aber zwingend im Seitenkopf oder -fuß hinterlegt werden (andernfalls erscheint beim Einfügen eine Fehlermeldung). Zum Einfügen einer Seitennummer im Seitenfuß gehen Sie z.B. folgendermaßen vor:

1. Fügen Sie zuerst ein Element (z.B. ein Textfeld) als Container zur Aufnahme des Funktionsergebnisses aus der Toolbox in den gewünschten Bereich ein.
2. Klicken Sie das Platzhalterelement (z.B. das Textfeld) per Maus an, so dass der Markierungsrahmen zu sehen ist.
3. Klicken Sie mit der rechten Maustaste in den Markierungsrahmen, um das Kontextmenü zu öffnen. Wählen Sie im Kontextmenü den Befehl *Ausdruck* (Abbildung 15.28).
4. Im Dialogfeld *Ausdruck bearbeiten* lässt sich nun in der linken unteren Liste eine Kategorie und dann in der mittleren Liste ein Ausdruck (z.B. »PageNumber«) wählen.
5. Übertragen Sie den Ausdruck mittels der *Hinzufügen*-Schaltfläche in das obere Textfeld des Dialogs und achten Sie darauf, dass dem Ausdruck ein Gleichheitszeichen vorangestellt ist.

Sobald Sie das Dialogfeld über die OK-Schaltfläche schließen, wird der Ausdruck als Feldbezeichner im Platzhalter (z.B. Textfeld) im Formular eingefügt.

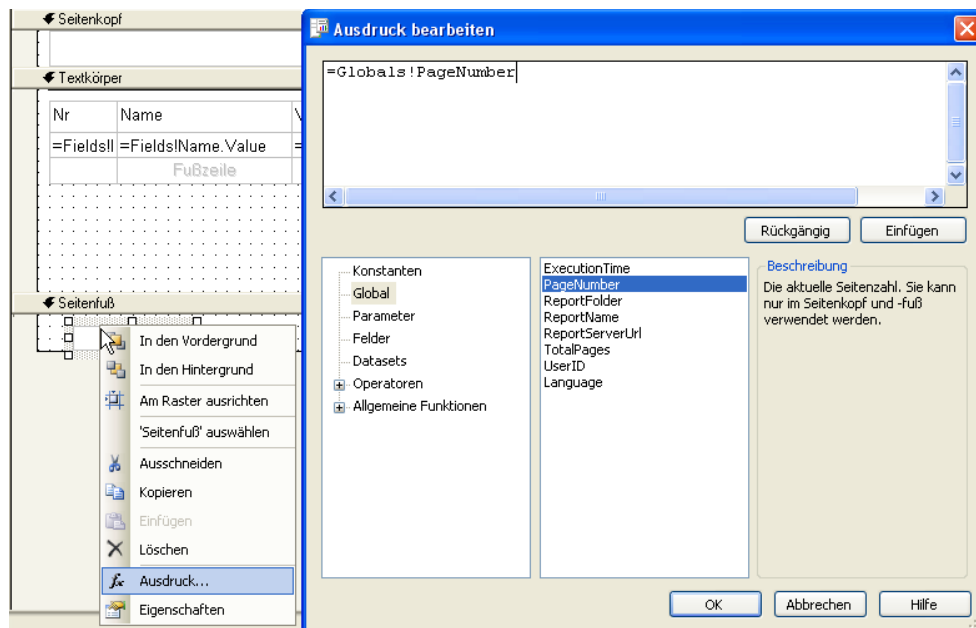


Abbildung 15.28: Einfügen von Ausdrücken im Bericht

Ist der Entwurf des Berichts fertig gestellt, können Sie dessen Design-Fenster schließen und mit der Gestaltung des Formulars zur Berichtsanzeige beginnen.

Formular mit ReportViewer ergänzen

Zur Darstellung des Berichts in der .NET-Anwendung lässt sich ein Formular mit einem eingefügten *ReportViewer*-Steuerelement verwenden. Zum Einfügen und Konfigurieren des Steuerelements im Formular sind folgende Schritte erforderlich:

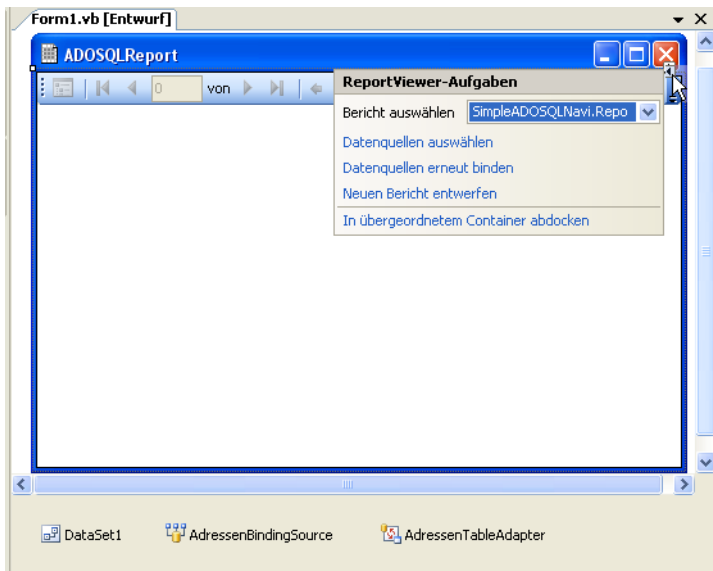


Abbildung 15.29: Einbinden des *ReportViewer*-Steuerelements im Formular

1. Öffnen Sie das noch leere Formular durch einen Doppelklick auf den Eintrag des Formulars im Projektmappen-Explorer. Das Formular wird dann im Fenster des Ansicht-Designers eingeblendet.
2. Fügen Sie aus der Toolbox ein *ReportViewer*-Steuerelement in den Formularbereich ein. Passen Sie die Formularabmessungen ggf. an und setzen Sie die *Dock*-Eigenschaft des *ReportViewer*-Steuerelements auf den Wert *Fill*. Dies bewirkt, dass das mit einer Navigationsleiste und einem Darstellungsbereich für den Bericht versehene Steuerelement an die Formulargröße angepasst wird.
3. Klicken Sie in der rechten oberen Ecke des Formularfensters auf die Schaltfläche, um den SmartTag-Bereich des *ReportViewer*-Steuerelements zu öffnen (Abbildung 15.29) und wählen Sie im Listenfeld *Bericht auswählen* den Eintrag für den gerade erstellten Berichtsentwurf.

Tipp

Sie können im Smarttagbereich des *ReportViewer*-Elements auch den Befehl *In übergeordnetem Container andocken* wählen, um das Steuerelement auf die Größe des Formulars abzustimmen.

Mit diesem Schritt werden automatisch Instanzen der im Bericht verwendeten Datenquellen erstellt. Gleichzeitig wird der Code generiert, um ein *DataSet*-Objekt, eine *TableAdapter*-Komponente und ein *BindingSource*-Objekt für die im Bericht verwendeten Datenquelle zu instantiieren. Die *Load*-Ereignisbehandlungsroutine enthält z.B. die beiden folgenden Anweisungen:

```
Me.AdressenTableAdapter.Fill(Me.DataSet1.Adressen)
Me.ReportViewer1.RefreshReport()
```

Die erste Anweisung bewirkt den Aufruf der *Fill()*-Methode, die den *TableAdapter* mit den im *DataSet* gespeicherten Tabellendaten füllt. In der zweiten Anweisungszeile bewirkt ein Aufruf der *RefreshReport()*-Methode die Anzeige des Berichts im *ReportViewer*-Steuerelement. Sobald das Steuerelement über den Smarttagbereich an den Bericht gebunden wurde, können Sie das Projekt übersetzen und ausführen. Dann wird das Formular angezeigt und der Bericht aus den Daten der Datenquelle generiert (Abbildung 15.30). Über die Schaltflächen der Symbolleiste können Sie den Bericht ggf. drucken, im Seitenlayout anpassen oder als Excel-Arbeitsblatt bzw. als PDF-Datei speichern.



Abbildung 15.30: Anzeige des Berichts im Formular

Hinweis

Sie finden die Projektdateien eines einfachen Berichtsbeispiels im Ordner `\Beisp\Kap15\ADOSQLReport` auf der Begleit-CD. Im Projekt wurde eine einfache SQL Server-Datenbankdatei *Adressen.mdf* eingebunden und über ein *DataSet*-Objekt der Anwendung bereitgestellt. Weiterführende Informationen zum Erstellen von Berichten oder zur Anwendung der in Visual Studio mitgelieferten Crystal-Report-Elemente finden Sie in der Hilfe.

Damit möchte ich die Einführung in ADO.NET schließen. Auf den vorhergehenden Seiten haben Sie eine Fülle an Informationen zum Arbeiten mit ADO.NET kennen gelernt. Trotzdem konnte ich nur einen Bruchteil des Funktionsumfangs anreißen. Weitere Details können Sie der Hilfe zum .NET Framework oder weiterführender Literatur zu ADO.NET entnehmen.

16

Spezielle Themen und Techniken

In diesem Kapitel werden spezielle Themen zur Anwendungsentwicklung für .NET wie Zugriffe auf XML-Daten, Drucken, Grafikprogrammierung etc. vorgestellt.

16.1 Zugriff auf XML-Daten in .NET

XML (Extended Markup Language) ist ein Standard zur Beschreibung und Speicherung von Daten. Speziell wenn es um Fragen des Datenaustauschs zwischen Anwendungen geht, kommt man häufig an XML nicht mehr vorbei. Die .NET-Architektur selbst benutzt XML in vielen Bereichen und bietet Klassen, um direkt auf XML-Daten lesend und schreibend zuzugreifen.

16.1.1 XML im Schnellüberblick

XML stellt eine allgemeine Beschreibungssprache für Daten dar, die in Textform in Dateien oder im Speicher hinterlegt werden. Dabei benutzt XML eine Beschreibungsstruktur, die das Lesen und Schreiben der Daten sehr stark erleichtert. Ein typisches (hier von mir für Testzwecke entworfenen) XML-Dokument besitzt folgende Struktur:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<root>
  <person>
    <name alter="22">Born</name>
    <vorname>Klaus</vorname>
    <gehalt Note="Euro">1200</gehalt>
  </person>
  <person>
    <name alter="42">Bach</name>
    <vorname>Hans</vorname>
    <gehalt Note="Euro">3200</gehalt>
  </person>
</root>
```

Die erste Zeile enthält eine Art Prolog, an der ein XML-Leseprogramm (Parser) erkennt, dass es sich um eine XML-Datei handelt. Die *encoding*-Angabe legt die Zeichencodierung fest und ist erforderlich, wenn beispielsweise Umlaute oder Sonderzeichen in der Datei vorkommen. Ohne *encoding*-Angabe benutzen die meisten XML-Parser den UTF-8-Zeichensatz (entspricht *encoding="utf-8"*). An die erste Signaturzeile können sich noch Zeilen

mit so genannten Processing Instructions (PI) anschließen. Dies sind Hinweise für das bearbeitende Werkzeug auf weiterführende Informationen wie Definitionsdaten in Form von DTDs oder Schemata etc. (habe ich in obigem Beispiel weggelassen). Danach folgen die so genannten XML-Elemente. Dies sind Schlüsselwörter, die in spitzen Klammern hinterlegt sind (z.B. `<root>`). XML-Elemente werden nach einem bestimmten Schema codiert:

- XML-Elemente bestehen immer aus einem einleitenden Tag (z.B. `<name>`) und einem abschließenden Tag (`</name>`). Zwischen diesen Tags können der Wert eines Elements (z.B. `<name>Bach</name>` oder untergeordnete XML-Elemente `<person><size>178</size></person>`) hinterlegt werden. Einige XML-Elemente besitzen keinen Wert, sind also leer. Dann kann der abschließende Tag entfallen und es wird eine verkürzte Schreibweise in der Form `<name />` benutzt.
- Die Schreibweise eines XML-Elements muss zwischen einleitendem und abschließendem Tag eingehalten werden. Insbesondere werden Groß- und Kleinbuchstaben unterschieden. Die Angabe `<name>Bach</name>` ist daher etwas anderes als `<Name>Bach</Name>` und die Angabe `<Name>Bach</name>` ist schlicht falsch, da Anfangs- und Endtag nicht in der Schreibweise zusammenpassen.
- Innerhalb des einleitenden Tags eines XML-Elements lassen sich noch so genannte Attributwerte einfügen. Diese Attribute erlauben einen Wert an das Element anzuhängen (z.B. `<size unit="cm">...</size>`).

Der Wert eines XML-Elements kann aus einfachem Text bestehen (z.B. `<name>Bach</name>`), es können aber auch komplexere Daten wie Binärdaten einer Bilddatei, Programmanweisungen etc. hinterlegt sein. Damit der XML-Parser diesen Teil nicht auswertet und auf die XML-Syntax hin überprüft, werden solche Daten mit einer CDATA-Anweisung (z.B. `<![CDATA [ Daten... ]]>`) als Wert im Tag gekapselt. Zusätzlich ist es möglich, Platzhalter (Entitäten) im XML-Dokument zu definieren. Kommentare, die nicht durch den XML-Parser ausgewertet werden, sind in folgender Notation im XML-Dokument zu hinterlegen:

```
<!-- Kommentarzeile -->
```

Wichtig ist es, einen Kommentar mit `<!--` einzuleiten und mit `-->` abzuschließen. Sie können eine XML-Datei nach den obigen Kriterien mit XML-Elementen füllen. Damit sichergestellt ist, dass verschiedene Anwendungen diese Daten auch lesen können, muss die Struktur der XML-Dokumentdatei bestimmten Anforderungen genügen. Ein Kriterium ist, dass XML-Daten »wohlgeformt« sein müssen. Ein als wohlgeformt klassifiziertes XML-Dokument genügt verschiedenen Regeln (es muss die Regeln der XML-Spezifikation erfüllen). Nachfolgend finden Sie eine Kurzübersicht über diese Regeln:

- Jede XML-Datei beginnt mit dem bereits erwähnten Prolog `<?xml ...?>`. Die im Dokument verwendeten Zeichen müssen dem in der XML-Signatur angegebenen (oder standardmäßig verwendeten) *encoding*-Schema entsprechen. Fehlt das *encoding*-Attribut, lassen sich z.B. keine Umlaute für die Werte verwenden.
- Die XML-Elemente sind nach den oben beschriebenen Regeln aufzubauen (z.B. einleitender und abschließender Tag ist vorhanden). Attributwerte sind in Anführungszeichen `'...'` oder doppelte Anführungszeichen `"..."` zu stellen.

- Die Tags der XML-Elemente müssen korrekt geschachtelt werden, Groß-/Kleinschreibung ist zu beachten. Werte, die Zeichen wie <, > oder & enthalten, sind als sogenannte Entities (zB. > für >) zu codieren. Andernfalls würde der XML-Parser diese Zeichen als Beginn oder Ende eines Tags bzw. Beginn eines Entity interpretieren, was zu Fehlern führt.
- Jede XML-Datei muss einen sogenannten *root*-Element aufweisen, welches auf die Signaturzeile folgt und alle anderen XML-Elemente einschließt.

XML-Parser weisen nicht wohlgeformte XML-Dokumente mit einer entsprechenden Meldung zurück. Zur Anzeige einer wohlgeformten XML-Datei eignen sich neben speziellen XML-Editoren auch der Internet Explorer (ab Version 5.0.1) oder eine Entwicklungsumgebung wie Visual Studio bzw. Visual Basic 2005 Express Edition. Sobald Sie eine XML-Dokumentdatei in der Entwicklungsumgebung laden, zeigt diese die Struktur der XML-Datei im internen XML-Editor an. Die Tag-Namen sowie die Werte der XML-Elemente und Attribute werden farbig hervorgehoben (Abbildung 16.1).

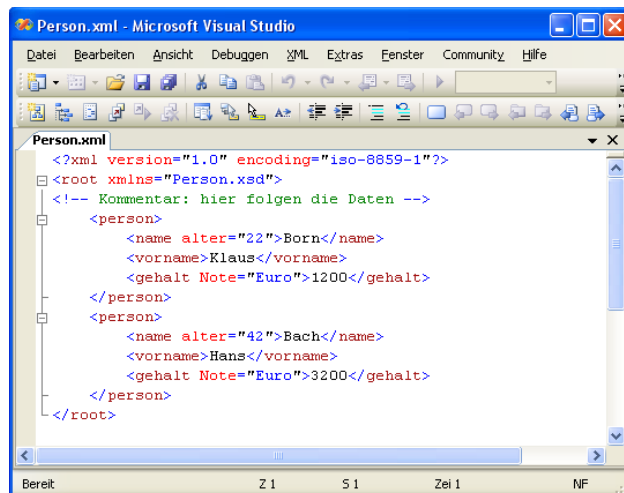


Abbildung 16.1: Anzeige der XML-Daten in Visual Studio

XML-Dokumente können neben Wohlgeformtheit auch das Kriterium der Gültigkeit erfüllen. Dann muss zum Dokument aber eine Art Strukturbeschreibung (Schema oder DTD) existieren, welche die zulässigen Elemente, deren Attribute und die Reihenfolge der Elemente im XML-Dokument festlegt. XML sieht mehrere Varianten an Schemabeschreibungen vor:

- Die älteste Variante (definiert in der XML 1.0-Spezifikation) besteht in einem Verweis auf eine sogenannte DTD (Document Type Definition). Diese DTD beschreibt explizit jeden Elementtyp und die mögliche Schachtelung der XML-Elemente. Allerdings sind DTDs etwas mühsam zu lesen und zu pflegen
- Moderner ist der Ansatz der *XML-Schemata*, die XML-Elemente zur Beschreibung der Struktur von XML-Dokumenten verwenden. Unter dem Begriff *XSD-Schemata* existiert mittlerweile ein W3C-Standard zur Beschreibung von XML-Dokumentstrukturen.

Die Entwicklungsumgebung (Visual Studio bzw. Visual Basic 2005 Express Edition) unterstützt XML-Dokumente und XSD-Schemata. Die Überprüfung einer XML-Dokumentdatei auf Wohlgeformtheit erfolgt automatisch und ein Test auf Gültigkeit ebenfalls, sofern ein Schema existiert.

Hinweis

Weitergehende Hinweise zu XML finden Sie in der Hilfe zum .NET Framework (z.B. unter dem Stichwort »XML, Anzeigen von Daten«). Eine detailliertere Einführung in XML finden Sie beispielsweise in meinem im Markt&Technik-Verlag erschienenen Titel »Jetzt lerne ich XML«, ISBN 3-8272-6854-0.

Wie kann ich XML-Dokumente und XSD-Schemata erstellen?

Zum Erstellen von XML-Dokumentdateien können Sie auf den XML-Editor der Entwicklungsumgebung zurückgreifen. Dieser unterstützt die Eingabe der XML-Elemente und achtet darauf, dass diese den Regeln für wohlgeformte XML-Dokumente entsprechen. Um XML-Dokumente zu erstellen bzw. zu pflegen, haben Sie in der Entwicklungsumgebung mehrere Möglichkeiten.

- Wählen Sie in der Entwicklungsumgebung im Menü *Datei* den Befehl *Neu/Datei* und im Dialogfeld *Neue Datei* den Eintrag *XML-Datei*. Dann erzeugt die Entwicklungsumgebung eine neue XML-Dokumentdatei, die nur die Zeile mit dem Prolog enthält. Die XML-Dokumentdatei lässt sich später in einem beliebigen Zielordner speichern.
- Alternativ können Sie die neue XML-Datei direkt dem Projekt zuordnen und automatisch in der Projektmappe hinterlegen lassen. Hierzu klicken Sie im Fenster des Projektmappen-Explorers mit der rechten Maustaste auf das Symbol der Projektdatei und wählen den Kontextmenüeintrag *Einfügen/Neues Element*. Anschließend wählen Sie im Dialogfeld *Neues Element* die Vorlage *XML-Datei*.

Sobald Sie das Dialogfeld über die *Hinzufügen*-Schaltfläche schließen, wird eine aus der Zeile `<?xml ... ?>` bestehende leere XML-Struktur angelegt. Sie können anschließend die XML-Elemente direkt im Fenster eintippen (Abbildung 16.2, unten rechts). Sobald Sie einen Starttag der Art `<test>` eintippen, wird automatisch der Endtag hinzugefügt. Sie können dann zwischen den Tags den Wert eintragen. Attributwerte tippen Sie direkt ein, Kommentare ebenso. Der Editor hebt die einzelnen Elemente der XML-Struktur farbig hervor.

- Bereits bestehende XML-Dateien fügen Sie dem Projekt hinzu, indem Sie im Fenster des Projektmappen-Explorers mit der rechten Maustaste auf das Symbol der Projektdatei klicken und den Kontextmenüeintrag *Einfügen/Vorhandenes Element* wählen.
- Über die Schaltfläche *Öffnen* der Symbolleiste oder über den gleichnamigen Befehl des Menüs *Datei* lässt sich der gleichnamige Dialog aufrufen. In diesem Dialog können Sie eine XML-Datei auswählen. Beim Schließen des Dialogs über die *Öffnen*-Schaltfläche wird die Datei im XML-Editor der Entwicklungsumgebung geladen und als XML-Struktur eingeblendet.

Die geladenen XML-Dokumente werden direkt im XML-Editor der Entwicklungsumgebung angezeigt und lassen sich ebenfalls direkt bearbeiten.

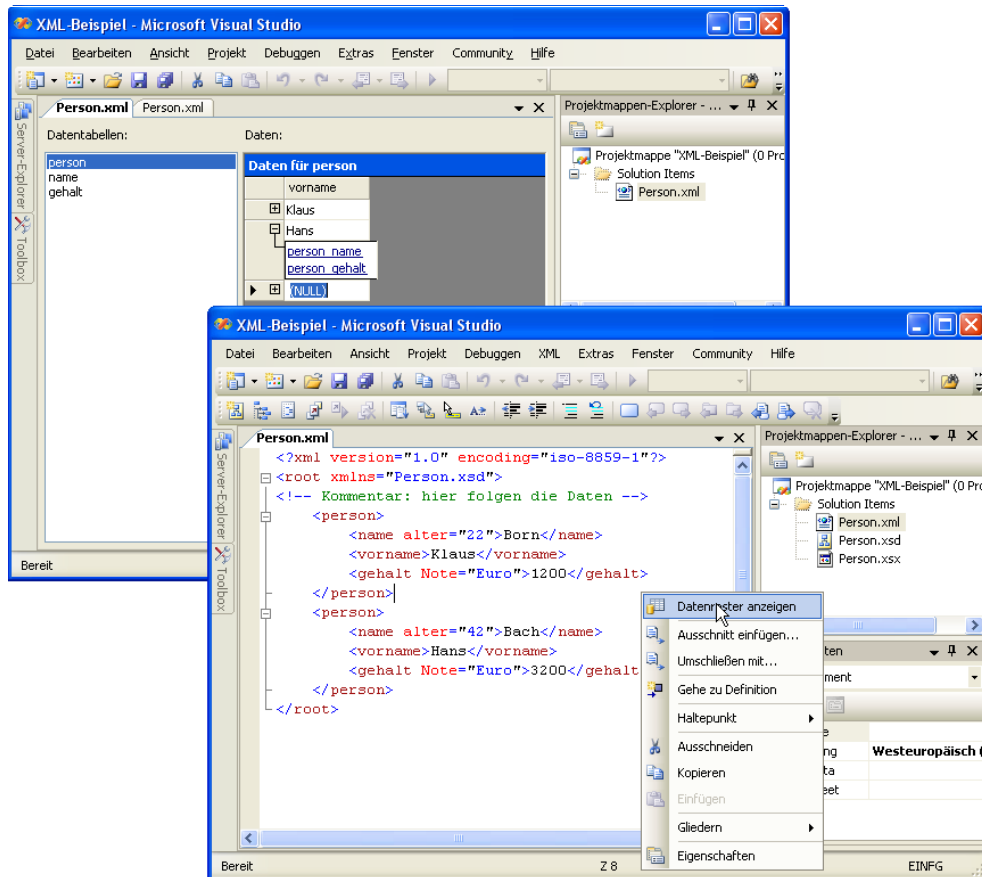


Abbildung 16.2: Ansichten beim Erstellen eines XML-Dokuments in Visual Studio .NET

Klicken Sie das Fenster mit der Codeansicht mit der rechten Maustaste an, lässt sich über den Kontextmenübefehl *Datenraster anzeigen* eine modifizierte Darstellung (Abbildung 16.2, Hintergrund) abrufen. Beim Datenraster werden die bereits definierten XML-Elemente in der linken Spalte *Datentabellen* aufgelistet. Klicken Sie ein Element an, zeigt die mittlere Spalte *Daten* den Inhalt dieses Elements an. Sie können dann einen neuen Knoten einfügen, indem Sie auf die unterste Zeile der Tabelle klicken und den Wert eintippen. Ein Pluszeichen im Zeilenkopf signalisiert, dass das XML-Element Attribute besitzt. Über das Pluszeichen lässt sich eine Unterstruktur mit den Attributnamen in der zweiten Spalte einblenden. Klicken Sie auf die als Hyperlink ausgeführten Attributnamen, öffnet sich eine Tabellenstruktur, in der Sie die Attributwerte sehen bzw. anpassen können. Eine dann in der Titelleiste der Tabelle eingeblendete Schaltfläche *Navigiert zu den übergeordneten Zeilen zurück* erlaubt zur Darstellung des XML-Elements zurückzugehen. Zur Anzeige der XML-Struktur gelangen Sie, indem Sie mit der rechten Maustaste auf das Dokumentfenster klicken und den Kontextmenübefehl *Code anzeigen* wählen.

Hinweis

Der Darstellungsmodus *Datenraster anzeigen* lässt sich aber erst aufrufen, wenn die XML-Datei ein Wurzelement und ein Datenelement aufweist. Ist dies nicht der Fall, zeigt die Entwicklungsumgebung einen Fehler an.

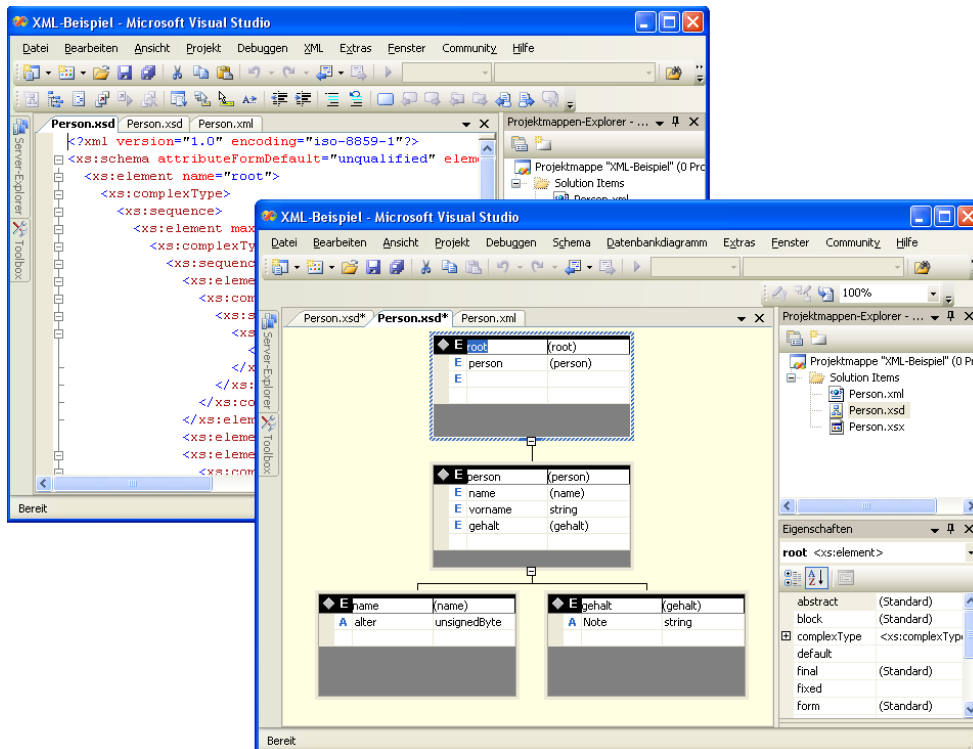


Abbildung 16.3: Ansichten eines XSD-Schemata in Visual Studio

Der XML-Editor der Entwicklungsumgebung kann aber noch mit zwei weiteren netten Funktionen aufwarten. Ist ein XML-Dokument geöffnet, wird das Menü **XML** im Fenster der Entwicklungsumgebung eingeblendet. Über den Befehl *Schema erstellen* weisen Sie den Editor an, automatisch ein XSD-Schemata aus den XML-Daten zu generieren. Das Schema wird dann in einem getrennten Fenster im XML-Code angezeigt (Abbildung 16.3, oben). Klicken Sie dann das Codefenster mit der rechten Maustaste an, lässt sich im Kontextmenü der Befehl *Ansicht-Designer* wählen. Der Editor zeigt dann eine grafische Darstellung des XSD-Schemata mit XML-Struktur (Abbildung 16.3, unten). Ist der Ansicht-Designer aktiv, lässt sich die Darstellung über den Befehl *Zoom* des Menüs *Datenbankdiagramm* vergrößern oder verkleinern. Im Menü *Schema* finden Sie zudem den Befehl *Layout*, über dessen Untermenü sich die Struktur horizontal oder vertikal anordnen lässt. Das Menü weist den Befehl *Hinzufügen* mit einem Untermenü auf, über dessen Befehle Sie neue Elemente zum XML-Schema hinzufügen können. Weiterhin können Sie XML-Elemente über die Toolbox einfügen und deren Eigenschaften im Eigenschaftenfenster pflegen.

Hinweis

Visual Studio unterstützt weitere XML-Funktionen, die in der Hilfe (z.B. unter dem Stichwort »XML-Editor«) beschrieben werden. Für die folgenden Beispiele finden Sie auf der Begleit-CD fertige XML-Dokumente. Im Ordner `\Beisp\Kap16\XML-Beispiel` der Begleit-CD finden Sie z.B. eine Projektmappe mit einer XML-Dokumentdatei, der auch ein XSD-Schema zugeordnet ist.

16.1.2 Eine Datenbanktabelle in XML exportieren

In einem ersten einfachen Beispiel soll die Tabelle *Users* der Access-Datenbank *UserData.mdb* (aus Kapitel 15) gelesen und in eine XML-Datei umgesetzt werden. Hierzu lässt sich über ADO.NET eine Verbindung benutzen, um über einen *DataAdapter* ein *DataSet* zu füllen. Anschließend wird die *WriteXML()*-Methode des *DataSet*-Objekts zum Speichern verwendet. Das Beispielprogramm verwendet diese Methode, um die Daten der Tabelle *User* aus der Access-Datenbank *UserData.mdb* in die XML-Datei *UserData.xml* des Programmverzeichnisses zu exportieren. Da die Access-Datenbank im Projekt eingebunden wurde, kopiert die Entwicklungsumgebung diese ebenfalls beim Erstellen in das Programmverzeichnis. Der Programmcode besteht im Wesentlichen aus den Ansätzen, die in den ADO.NET-Beispielen des vorherigen Kapitels entwickelt wurden. Zum Sichern der Daten mittels der *WriteXML()*-Methode eines *DataSet*-Objekts sind nur wenige Anweisungen erforderlich. Zuerst ist eine gültige Verbindung zur Datenbank einzurichten (hier wird der *OleDB*-Treiber genutzt).

```
Dim oDB As OleDb.OleDbConnection = New OleDb.OleDbConnection( _
    My.Settings.UserDataConnectionString)
oDB.Open() ' Verbindung öffnen
```

Dann sind ein *DataSet*-Objekt sowie ein *DataAdapter*-Objekt anzulegen und der *DataAdapter* ist mit den Daten der Abfrage zu füllen:

```
Dim sql As String = "SELECT * FROM Users" ' SQL-Query
Dim oDataSet As New DataSet ("User") ' neues DataSet anlegen
Dim oAdapter = New OleDb.OleDbDataAdapter(sql, oDB)
oAdapter.Fill(oDataSet) ' DataSet an DataAdapter zuweisen und füllen
```

Im Parameter *sql* zum Anlegen des *DataAdapter* steht ein SQL-Befehl *SELECT * FROM Users*. Die Objektvariable *oDB* verweist auf die geöffnete Verbindung zur Datenbank. Die *Fill()*-Methode überträgt die Daten in das *DataSet*. Anschließend reduziert sich das Speichern in eine XML-Datei auf folgende Zeile:

```
oDataSet.WriteXML (outFile, XmlWriteMode.WriteSchema)
```

Der Parameter *outFile* enthält den Namen der Ausgabedatei, der zweite Parameter definiert den Ausgabemodus für die XML-Daten. Hier wurde die *WriteXML*-Methode angewiesen, ein XML-Schema zu sichern (dies ist eine XML-Datei, die neben den eigentlichen

Daten noch ein Schema mit der Beschreibung der XML-Elemente enthält – d.h., es werden Strukturinformationen und Daten gespeichert).

Hinweis

Die Projektdateien des Beispiels finden Sie im Ordner `\Beisp\Kap16\ADODBXMLExport` der Begleit-CD. Zum Testen der Anwendung müssen Sie das Programmverzeichnis mit der Anwendung und der Datenbankdatei in einen Ordner der Festplatte kopieren. Das Programm zeigt beim Ablauf den Status in Dialogfeldern. Zudem enthält es noch Anweisungen zur Behandlung von Laufzeitfehlern (Ausnahmebehandlung). Weitere Details sind dem Quellcode zu entnehmen. Einzelheiten zur `WriteXML()`-Methode finden Sie in der Hilfe zum .NET Framework.

16.1.3 XML-Daten per DataSet einlesen

Das Einlesen von XML-Daten geht mit der `ReadXML()`-Methode eines `DataSet`-Objekts sehr einfach. Dies wird jetzt an einer kleinen Anwendung zur Anzeige der XML-Daten in einem `DataGridView`-Element demonstriert (Abbildung 16.4).

ID	Name	PersId
1	Born	123
2	Brown	234
3	Miller	343
4	Bush	345
11	Born G	12355
12	Gates	91111
24	Meier	777
25	Young	1007

Abbildung 16.4: Anzeige von XML-Daten in einem DataGridView

Das Programm öffnet über die `Load`-Ereignisbehandlungsroutine beim Start automatisch die angegebene XML-Datei (hier die Datei `UserData.xml` im Unterordner `\Daten` des Programmverzeichnisses). Der eigentliche Programmcode zum Lesen der XML-Datei sowie zur Anzeige der Ergebnisse im DataGridView umfasst nur wenige Zeilen.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    ' Beim Laden des Formulars Daten einlesen
    Dim oDataSet As New DataSet("User") ' DataSet instantiieren

    Try
        ' lese jetzt XML-Datei per ReadXML-Methode in DataSet
```

Listing 16.1: Einlesen einer XML-Datei über ein DataSet und Anzeige in DataGridView

```

oDataSet.ReadXml(GetPath() & "Daten\UserData.xml")

' Ordne jetzt das DataSet dem DataGridView-Element zu
' Wichtig: Bei der Tables-Auflistung muss ein Name angegeben
' werden (z.B. "Table" oder 0 für die erste Tabelle)
Me.DataGridView1.DataSource = oDataSet.Tables("Table")

Catch ex As Exception
    ' Irgend etwas hat nicht geklappt
    MsgBox(ex.Message, vbOKOnly Or vbCritical, "Fehler")
    Exit Sub
End Try
End Sub

```

Listing 16.1: Einlesen einer XML-Datei über ein DataSet und Anzeige in DataGridView (Forts.)

Sobald das *DataSet*-Objekt instantiiert wurde, lässt sich dessen die *ReadXML()*-Methode aufrufen. Die Methode ist mehrfach überladen und unterstützt verschiedene Aufrufvarianten. Hier wird eine Variante benutzt, der lediglich der Dateiname samt Pfad als Argument zu übergeben ist. Anschließend wird der *DataSource*-Eigenschaft des *DataGridView* die *Tables*-Eigenschaft des *DataSet*-Objekts zugewiesen. Diese Eigenschaft liefert eine Auflistung zurück. Daher ist zu beachten, dass entweder ein Index (z.B. der Wert 0 für die erste Tabelle) oder der Name des betreffenden XML-Elements (hier "Table") unterhalb des Wurzelknotens als Argument angegeben wird. Andernfalls wird nichts im *DataGridView*-Element angezeigt.

Hinweis

Sie finden die Projektdateien des Beispiels im Ordner `\Beisp\Kap16\ADOREadXML-DataGridView` der Begleit-CD. Im Programmverzeichnis `\Bin` ist ein Unterordner `\Daten` mit der benötigten XML-Datei hinterlegt.

In .NET Framework 1.x wurde das *DataGrid*-Steuerelement zur Anzeige von Tabellendaten benutzt. In .NET Framework 2.0 stellt die Toolbox dagegen das *DataGridView*-Steuerelement für diesen Zweck zur Verfügung. Allerdings sind die beiden Steuerelemente nicht zueinander kompatibel und entsprechende Anwendungen lassen sich auch nicht konvertieren. Aus Kompatibilitätsgründen wird die *DataGrid*-Klasse nach wie vor in .NET Framework 2.0 unterstützt. Im Ordner `\Beisp\Kap16\ADOREadXMLDataGrid` der Begleit-CD finden Sie noch eine Lösung, die unter Verwendung des *DataGrid*-Steuerelements realisiert wurde. Dieses Steuerelement hat beim Einlesen von XML-Dateien den Vorteil, dass Navigationselemente in der Anzeige eingeblendet werden, über die zwischen übergeordneter Darstellung (kollaborierte XML-Tabelle) und Detailansicht (Anzeige der XML-Daten als Datentabelle) umgeschaltet werden kann. Details sind dem Quellcode des Elements *Form1.vb* zu entnehmen.

16.1.4 Zugriff auf XML-Dateien per DOM

Das .NET Framework erlaubt Ihnen neben ADO.NET auch andere Ansätze, um auf XML-Dokumente zuzugreifen. Ein vom W3C-Gremium vorgestellter Standard ist das Document Object Model (DOM), welches einen Satz an Objekten samt Methoden und Eigenschaften zur Bearbeitung von XML-Dokumenten aus Programmen festlegt. Auch das .NET Framework stellt entsprechende Klassen bereit, die einen Zugriff auf XML-Dokumente über DOM erlauben.

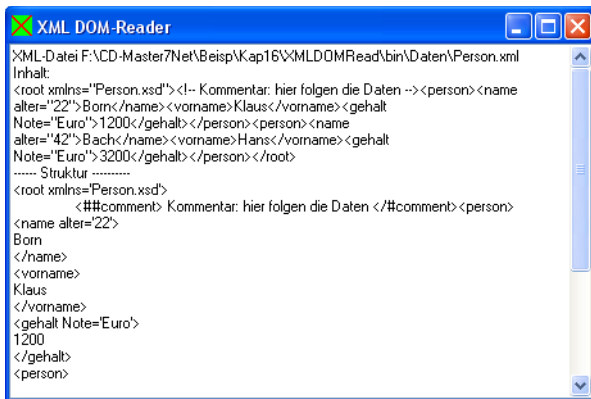


Abbildung 16.5: Anzeige der gelesenen XML-Daten

Dies soll an einem einfachen Beispiel demonstriert werden. Das Programm liest die Datei *Person.xml*, die auch einen Kommentar und einen Verweis auf die Schemadatei enthält, aus dem Unterordner *\Daten* ein. Anschließend wird der komplette XML-Dokumentinhalt ungefiltert in einem Textfeld angezeigt. Danach durchläuft das Programm alle XML-Elemente und gibt diese samt Werten und Attributen im Textfeld aus (Abbildung 16.5). Auf das saubere Einrücken der Tags und das Ergänzen aller abschließenden Tags wurde an dieser Stelle verzichtet, da ja nur der Zugriff auf Werte und Attributwerte demonstriert werden soll.

Die Klasse zum Zugriff auf die XML-Daten ist im Namensraum *System.XML* hinterlegt. Sie sollten einen Import auf diesen Namensraum vereinbaren:

```
Imports System.XML
```

Zudem muss in der Entwicklungsumgebung ein Verweis auf die entsprechende Bibliothek *System.Xml* eingebunden sein. Die Programmanweisungen zum Zugriff auf die XML-Dokumentdatei per DOM sind recht einfach. Zuerst vereinbaren Sie eine neue Instanz des *XmlDocument*-Objekts, welches die Daten aufnehmen soll:

```
Dim oXMLDoc As XmlDocument = New XmlDocument() ' XML-DOM-Instanz
```

Dann können Sie mittels der *Load()*-Methode dieses Objekts auf die XML-Dokumentdatei zugreifen und diese laden:

```
oXMLDoc.Load("C:\\Personen.xml")
```

Als Parameter ist der Name der Dokumentdatei an die Methode zu übergeben. Anschließend steht die komplette XML-Struktur im Objekt *oXMLDoc* bereit. XML-Elemente und Attribute werden dabei über folgende Datentypen abgebildet:

```
Dim oElement As XmlNode      ' Knoten mit einem XML-Element
Dim oAttribut As XmlAttribute ' Knoten mit Attribut
```

Das Document-Objekt-Modell betrachtet alles innerhalb der XML-Dokumentdatei als »Knoten«. Ein Kommentar, ein XML-Element, ein Attributwert etc. kann alles ein Knoten sein. Dieser Knoten wird über einen *XmlNode*-Datentyp abgebildet. Über die Member des *XMLDocument*-Objekts (Eigenschaften und Methoden) können Sie auf die Knoten zugreifen und Werte bzw. Attributwerte abfragen. Mit der Anweisung

```
Dim oRoot As XmlNode = oXMLDoc.DocumentElement
```

lässt sich der erste Dokumentknoten (die Wurzel der XML-Dokumentdatei) lesen und der Variablen *oRoot* zuweisen. Die *XmlNode*-Klasse enthält dann Member, um auf die Struktur des XML-Dokuments zuzugreifen. Die folgende Tabelle listet einige dieser Member (hier handelt es sich um Eigenschaftswerte) auf:

Member	Bemerkung
<i>Attributes</i>	Eigenschaft, die eine Auflistung aller Attribute des aktuellen Elements liefert
<i>ChildNodes</i>	Eigenschaft, liefert die Auflistung der untergeordneten Knoten des aktuellen Knotens
<i>FirstChild</i>	Die Eigenschaft liefert den ersten Kindknoten des aktuellen Elements
<i>HasChildNodes</i>	Die Eigenschaft liefert einen Wert <i>True</i> oder <i>False</i> , der angibt, ob der aktuelle Knoten einen Kindknoten besitzt
<i>InnerText</i>	Die Eigenschaft liefert den XML-Text, der im Knoten enthalten ist (bei einem XML-Node sind dies die Knoten, die als Wert im XML-Element enthalten sind)
<i>InnerXml</i>	Die Eigenschaft liefert das Markup (die Struktur), das im Objekt enthalten ist (bei einem XML-Node sind dies die Knoten, die als Wert im XML-Element enthalten sind) als XML-Struktur
<i>LastChild</i>	Die Eigenschaft liefert den letzten Unterknoten eines Knotens
<i>Name</i>	Die Eigenschaft liefert den Namen des aktuellen Knotens (kann ein Tag-Name sein)
<i>NextSibling</i>	Die Eigenschaft liefert den nächsten benachbarten Knoten der aktuellen Ebene
<i>NodeType</i>	Die Eigenschaft liefert den Knotentyp zurück. Text ist ein Knoten, der ein XML-Element mit Textwerten enthält.
<i>OuterXml</i>	Die Eigenschaft liefert den kompletten XML-Text, der im Objekt enthalten ist (bei <i>oXMLDoc</i> ist dies die XML-Dokumentstruktur)
<i>ParentNode</i>	Die Eigenschaft liefert den übergeordneten Knoten des aktuellen Knotens
<i>Value</i>	Eine Eigenschaft, die den Wert zurückgibt. Bei Elementknoten wird der Wert des Elements zurückgegeben.

Tabelle 16.1: Hilfreiche Member der XmlNode-Klasse

Im Beispielprogramm wird eine rekursiv aufrufbare Prozedur *CrossNodes*, der ein Knoten übergeben wird, um die Struktur des XML-Dokuments zu durchlaufen, benutzt. Der Erstaufwurf erfolgt in der *Load*-Ereignisbehandlungsprozedur des Formulars, nachdem die XML-Datei geladen wurde:

```
CrossNodes(oRoot)
```

Es wird das Wurzelement, welches über *DocumentElement* ermittelt wurde, als Knoten übergeben. Die Prozedur besitzt folgende Anweisungen:

```
Private Sub CrossNodes(ByVal node As XmlNode)
' Bearbeite Knotenebene (rekursiver Ansatz)
' Childs vorhanden? -> Nein-> Knoten anzeigen
If Not IsNothing(node) Then ShowNode(node) ' Knoten anzeigen

If node.HasChildNodes Then      ' Gibt es Child-Knoten ?
    node = node.FirstChild      ' hole ersten Knoten
    While Not IsNothing(node)    ' Child-Knoten bearbeiten ?
        CrossNodes(node)        ' rekursiver Aufruf
        node = node.NextSibling ' nächster Knoten der Ebene
    End While
End If
End Sub
```

Besitzt der im Parameter neu übergebene Knoten den Wert *Nothing*, ruft die Prozedur die Prozedur *ShowNode* auf, die die Ausgabe übernimmt. Besitzt der übergebene Knoten weitere »Kindknoten« (Prüfung mit *HasChildNodes*), liest die Prozedur diese über *FirstChild* und *NextSibling* und ruft sich mit diesen Knoten rekursiv auf. Mit dem rekursiven Ansatz wird der Baum durchlaufen. Die Aufbereitung der Ausgabe erfolgt dann in der Prozedur *ShowNode*, die folgenden Inhalt hat:

```
Private Sub ShowNode(ByVal node As XmlNode)
' Versuche, den Inhalt eines Elements anzuzeigen
' Der abschließende Tag bei geschachtelten Knoten fehlt aber.
If Not node.HasChildNodes Then
    ' Wert des Elements holen
    If CType(node, XmlNode).Name = "#text" Then ' Element-Wert
        Me.TextBox1.AppendText(CType(node, XmlNode).Value & _
            vbCrLf & "</" & node.ParentNode.Name & ">" & vbCrLf)
    Else
        Me.TextBox1.AppendText(Strings.Chr(9) & "<#" & node.Name & ">" & _
            node.Value & "</" & node.Name & ">")
    End If
Else ' Knoten mit Attributen
```

```

Me.TextBox1.AppendText("<" & node.Name)
If XmlNodeType.Element = node.NodeType Then
    Dim map As XmlNamedNodeMap = node.Attributes
    Dim attrnode As Object
    For Each attrnode In map ' Attribute
        Me.TextBox1.AppendText(" " & _
            CType(attrnode, XmlNode).Name & "=" & _
            CType(attrnode, XmlNode).Value & "'")
    Next
End If
Me.TextBox1.AppendText(">" & vbCrLf)
End If
End Sub

```

Besitzt ein Knoten weitere Unterknoten, wird nur dessen Inhalt (der Wert) und der abschließende Tag (über *node.ParentNode.Name*) ausgegeben. Der Typ des Knotens lässt sich mit *CType(node, XmlNode).Name* abfragen. Wird der String »#text« geliefert, ist ein Textwert im Knoten hinterlegt, der sich direkt über *CType(node, XmlNode).Value* ermitteln lässt. Den Namen des Elements ermittelt das Programm dann über *node.ParentNode.Name*. Handelt es sich nicht um einen Textwert, wird einfach der Name und dessen Wert über "<#" & *node.Name* & ">" & *node.Value* & "</" & *node.Name* & ">" in einen String gewandelt und ausgegeben (auf diese Weise werden z.B. Kommentare angezeigt). Erkennt die Prozedur, dass es sich um ein Blatt im Strukturbaum handelt, wird der Knotenname mit "<" & *node.Name* ausgegeben. Dann sorgt eine *For Each*-Schleife dafür, dass alle Attribute eines XML-Elements gelesen und mit

```

CType(attrnode, XmlNode).Name & "=" & _
CType(attrnode, XmlNode).Value & "'"

```

in einen Ausgabestring übergeben werden. Danach wird die abschließende Klammer und ein Zeilenumbruch zum Text hinzugefügt. Die Ausgabe erfolgt hier mit *Me.TextBox1.AppendText* in ein Textfeld.

Hinweis

Die Dateien dieses Projektbeispiels finden Sie im Ordner *\Beisp\Kap16\XMLDOM-Read* auf der Begleit-CD. Die *Load*-Ereignisbehandlungsprozedur enthält die Anweisungen zum Laden der XML-Datei sowie zum Aufruf der Hilfsprozeduren. Die Details entnehmen Sie dem Quellcodelist. Beim Ausführen erwartet das Programm einen Unterordner *\Daten* mit der Datei *Personen.xml*.

16.1.5 Zugriff auf die XML-Daten per XMLTextReader

Obwohl der Umgang mit dem DOM-Ansatz vielen Programmierern bekannt ist, besitzt dieser einen entscheidenden Nachteil: Die gesamte XML-Dokumentstruktur wird in den

Speicher geladen und bearbeitet. Bei einer kleinen XML-Beispieldatei ist dies kein Problem, aber ein XML-Dokument mit vielen tausend Knoten bringt Leistungsdefizite zum Vorschein. Konkret: Je größer die XML-Datei, umso langsamer geht die Bearbeitung vonstatten. Ein Ansatz zur Lösung ist SAX (Simple API for XML), eine Schnittstelle, die Funktionen zum sequentiellen Lesen von XML-Knoten bereitstellt. Programme, die auf dem .NET Framework aufsetzen, müssen sich mit solchen Feinheiten nicht herumschlagen. Die Klassenbibliothek stellt einen XMLReader als Klasse bereit, der ähnlich wie SAX ein Vorwärtslesen im XML-Datenstrom ohne Zwischenspeicherung zulässt. Dadurch können Sie auch große XML-Dokumente in .NET-Anwendung schnell und effizient analysieren.

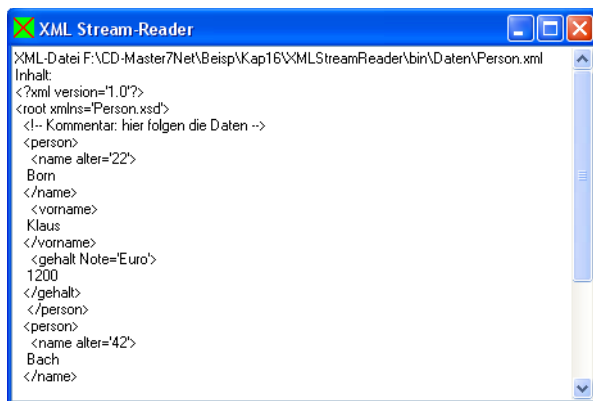


Abbildung 16.6: Darstellung des XMLReader-Beispiels

Das Ganze soll ebenfalls an Hand einer kleinen Beispielanwendung demonstriert werden. Diese liest die im Unterverzeichnis `\Daten` enthaltene XML-Dokumentdatei `Personen.xml` ein, analysiert die Knoten und gibt dann das Ergebnis in einem Textfeld als simulierte XML-Struktur aus (Abbildung 16.6). Allerdings wurde der Fokus bei der Entwicklung des Beispiels weniger auf die optische Aufbereitung der Daten als auf die Demonstration der Funktionalität gelegt. Daher erfolgt das Einrücken der untergeordneten XML-Elemente auch nicht sauber, da die *Depth*-Eigenschaft offenbar einen fehlerhaften Level für den abschließenden Tag eines Elements zurückliefert. In realen Anwendungen wird es selten darauf ankommen, Elemente formatiert darzustellen. Vielmehr soll das Programm Werte und Attributwerte lesen, was im Beispiel auch geleistet wird.

Beim Aufruf wird die XML-Dokumentdatei im *Load*-Ereignis des Formulars geladen und analysiert. Die Ergebnisse werden direkt in ein Textfeld geschrieben und stehen dem Benutzer zur Kontrolle zur Verfügung. Der Programmcode zum Instantiieren des Readers sowie zur Analyse der XML-Knoten ist im Vergleich zum vorherigen Beispiel recht einfach. Mit der folgenden Anweisung wird ein *StreamReader*-Objekt deklariert und instantiiert:

```
Dim oXML As StreamReader = New StreamReader("C:\Test.xml")
```

Dies kennen Sie bereits aus dem Kapitel zur Dateibearbeitung. Der eigentliche *XMLReader* wird dann auf dieses *Stream*-Objekt aufgesetzt:

```
Dim oXmlRead As XmlTextReader = New XmlTextReader(oXML)
```

Sobald diese Instanz vorliegt, lässt sich die XML-Dokumentstruktur in einer simplen Schleife durchlaufen:

```
Do While oXmlRead.Read () ' über alle Knoten
  Select Case oXmlRead.NodeType
    Case XmlNodeType.Element ' Element gefunden
      MsgBox ("Element: " & oXmlRead.Name)
      If oXmlRead.HasAttributes Then ' gibt es Attribute
        While oXmlRead.MoveToNextAttribute() ' alle Attribute
          MsgBox (oXmlRead.Name & "=" & oXmlRead.Value & "")
        End While
      End IF
    Case XmlNodeType.Text ' Wert ist Text
      MsgBox ("Wert: " & oXmlRead.Value)
    Case Else
      ' nicht implementierte Typen
    End Select
  Loop
```

Die *Read()*-Methode liest das nächste XML-Strukturelement im Datenstrom ein. Ist das Ende erreicht, wird der Wert *False* zurückgeliefert und die Schleife terminiert. Über die Eigenschaften der Objektinstanz *oXmlRead* lässt sich dann das gelesene Strukturelement analysieren. Der Knotentyp (z.B. ein XML-Element, ein Textwert eines Elements etc.) kann über die Eigenschaft *NodeType* bestimmt werden. Ich habe in obigem Codeausschnitt eine *Select Case*-Struktur zur Analyse des Knotentyps angedeutet. Ein Knoten vom Typ »Element« stellt den einleitenden XML-Tag eines Elements dar, dessen Name mit *oXmlRead.Name* abgefragt werden kann. Bei einem Element lässt sich über die *HasAttributes*-Eigenschaft feststellen, ob Attributwerte hinterlegt sind. Diese lassen sich über die *MoveToNextAttribute()*-Methode des in *oXmlRead* referenzierten *XMLReader*-Objekts in einer Schleife abrufen. Der Attributname wird dann über *oXmlRead.Name* und der Wert über *oXmlRead.Value* ermittelt. Das Beispielprogramm fügt diese Detailinformationen zu einer Zeichenkette zusammen und hängt diese mittels der *TextBox.Append()*-Methode an den Inhalt des Textfelds an. Nachfolgend sehen Sie den Code der kompletten *Load*-Ereignisbehandlungsroutine des Beispiels:

```
Private Sub Form_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
  ' Beim Laden des Formulars Daten einlesen und bearbeiten
  Dim txt As String = ""
  Dim file As String = GetPath() & "Daten\Person.xml"
  Dim oXML As StreamReader
  Dim oXmlRead As XmlTextReader
```

Listing 16.2: Code zum Einlesen und Aufbereiten der XML-Datei

```

txt = "XML-Datei " & file & vbCrLf & "Inhalt: " & vbCrLf

Try
    oXML = New StreamReader(file)      ' öffne XML-Datei
    oXmlRead = New XmlTextReader(oXML) ' neue XMLReader-Instanz

    Do While oXmlRead.Read ()          ' über alle Knoten
        txt = txt & Ins(oXmlRead.Depth) ' einrücken
        Select Case oXmlRead.NodeType
            Case XmlNodeType.Element    ' Element gefunden
                txt = txt & "<" & oXmlRead.Name
                If oXmlRead.HasAttributes Then ' gibt es Attribute
                    While oXmlRead.MoveToNextAttribute() ' alle Attribute
                        txt = txt & " " & oXmlRead.Name & "=" & _
                            oXmlRead.Value & "'"
                    End While
                End IF
                txt = txt & ">" & vbCrLf      ' abschließen

            Case XmlNodeType.Text        ' Wert ist Text
                txt = txt & oXmlRead.Value & vbCrLf

            Case XmlNodeType.CDATA
                txt = txt & "<![CDATA[{0}]]>" & oXmlRead.Value & vbCrLf

            Case XmlNodeType.ProcessingInstruction
                txt = txt & "<?" & oXMLRead.Name & _
                    oXmlRead.Value & "?>" & vbCrLf

            Case XmlNodeType.Comment
                txt = txt & "<!--" & oXmlRead.Value & "-->" & vbCrLf

            Case XmlNodeType.XmlDeclaration
                txt = txt & "<?xml version='1.0'?" & vbCrLf

            Case XmlNodeType.Document
                ' nicht implementiert

            Case XmlNodeType.DocumentType
                txt = txt & "<!DOCTYPE " & oXmlRead.Name & "[" & _

```

Listing 16.2: Code zum Einlesen und Aufbereiten der XML-Datei (Forts.)

```

        oXmlRead.Value & "]" & vbCrLf

    Case XmlNodeType.EntityReference
        txt = txt & oXmlRead.Name

    Case XmlNodeType.EndElement
        txt = txt & "</" & oXmlRead.Name & ">" & vbCrLf

    Case Else
        'txt = txt & "***Wertetyp nicht implementiert***"
    End Select
Loop
Me.TextBox1.AppendText(txt)
Catch ex As Exception
    MsgBox(ex.Message, vbOKOnly Or vbCritical, "Fehler")
Exit Sub
End Try
Me.TextBox1.AppendText("***Fertig***")
End Sub

```

Listing 16.2: Code zum Einlesen und Aufbereiten der XML-Datei (Forts.)

Ich habe in der *Select Case*-Struktur verschiedene Node-Typen implementiert. Details zur kompletten Programmstruktur können Sie dem Quellcode entnehmen.

Hinweis

Sie finden die Projektdateien des betreffenden Beispiels im Ordner `\Beisp\Kap16\XMLStreamReader` der Begleit-CD. Der Quellcode enthält noch einige Hilfsroutinen (z.B. zur Ermittlung des Pfads etc.). Das Beispiel benötigt den Unterordner `\Daten` mit der XML-Datei `Personen.xml`.

Aus Platzgründen konnten in der obigen XML-Einführung nur einige Aspekte angerissen werden. Sie haben beispielsweise die Möglichkeit, XML-Dokumente über DOM programmgesteuert um XML-Elemente zu erweitern. .NET Framework bietet eine *XmlTextWriter*-Klasse zum Speichern von XML-Daten. Selbst Variablen oder Datenstrukturen lassen sich über die Klasse *XmlSerializer* des Namensraums *System.Xml.Serialization* speichern und wieder einlesen. Detailliertere Informationen zu den Membern des *XmlReaders*, zu der erwähnten *XmlSerializer*-Klasse und zu anderen XML-Klassen finden Sie in der Hilfe zum .NET Framework.

16.2 Anwendungseinstellungen und Hilfe

In den vorhergehenden Kapiteln haben Sie eine Vielzahl von Techniken kennen gelernt, um .NET-Anwendungen zu erstellen. In diesem Abschnitt möchte ich noch zeigen, wie sich dynamische Anwendungseinstellungen nutzen lassen. Weiterhin wird kurz angerissen, wie Sie Hilfedateien in Anwendungen integrieren können.

16.2.1 Verwalten von Formular- und Anwendungseinstellungen

Standardmäßig legen Sie die Eigenschaften von Formularen und Steuerelementen (z.B. Beschriftungen mittels der *Text*-Eigenschaft) im Eigenschaftenfenster des betreffenden Elements fest. Oder die Eigenschaften werden zur Laufzeit im Programmcode umgesetzt. Gelegentlich besteht aber der Wunsch, ggf. einige Eigenschaften ohne Eingriff in den Programmcode zu ändern. So könnte ein Administrator oder ein Anwender mit den entsprechenden Berechtigungen beispielsweise die Beschriftungen in einem Formular lokalisieren wollen. Zur Lösung dieses Problems stellt .NET Framework eine Möglichkeit bereit, Anwendungseigenschaften in Konfigurationsdateien auszulagern. Diese, im XML-Format hinterlegten, Konfigurationsdateien lassen sich vom Benutzer bzw. Administrator ohne Eingriff in den Quellcode anpassen. .NET-Anwendungen lesen die Konfigurationsdateien beim Programmstart und weisen die Eigenschaftenwerte dann den jeweiligen Elementen dynamisch zu. Weiterhin kann auch aus dem Quellcode auf solche Anwendungseinstellungen zugegriffen werden (z.B. auf Pfadangaben).

Hinweis

Gegenüber dem .NET Framework 1.x ergeben sich bei der Version 2.0 aber einige Abweichungen. In .NET Framework 1.x standen für Steuerelemente dynamische Eigenschaften (Eintrag *Dynamic Properties* im Eigenschaftenfenster einzelner Steuerelemente) zur Verfügung. Während die betreffende Klasse aus Kompatibilitätsgründen noch vorhanden ist, wurde die Benutzeroberfläche für dynamische Eigenschaften aus der Entwicklungsumgebung entfernt. Stattdessen lassen sich solche Werte jetzt über Anwendungseinstellungen verwalten. Nachfolgend wird der Ansatz von .NET Framework 2.0 zur Verwaltung dynamischer Einstellungen besprochen.

So lassen sich dynamisch Formulareinstellungen konfigurieren

Möchten Sie bestimmte Eigenschaften von Formularen oder Steuerelementen (z.B. Beschriftungen) automatisch in Konfigurationsdateien auslagern und von der .NET-Anwendung dynamisch verwalten lassen? Dann müssen Sie die Anwendungseinstellungen für die betreffenden Steuerelemente anpassen. Dies soll jetzt an einem einfachen Formular demonstriert werden (Abbildung 16.7).

Dieses besitzt zwei Schaltflächen, wobei eine ein Dialogfeld mit einer Auflistung von Anwendungseigenschaften einblendet, während die andere Schaltfläche zum Beenden der Anwendung benutzt wird. Die Beschriftung der Schaltflächen sowie der Formulartitel sollen dabei über Konfigurationsdateien anpassbar sein. Zudem werden im Formular noch die Anwendungseinstellungen über ein *PropertyGrid*-Steuerelement eingeblendet. Der Benutzer kann daher die Anwendungseinstellungen direkt in diesem Steuerelement als Eigenschaften sehen bzw. anpassen.



Abbildung 16.7: Formular mit dynamischen Eigenschaften

Wie lassen sich nun aber der Formulartitel oder die Beschriftung der Schaltflächen bereits im Ansicht-Designer über Anwendungseigenschaften definieren? Vielleicht ist Ihnen beim Formularentwurf bereits der Eintrag (*Application Settings*) im Eigenschaftensfenster einzelner Steuerelemente aufgefallen? Um eine Formular- oder Steuerelementeigenschaft in der Entwicklungsumgebung dynamisch über eine Anwendungseinstellung zuzuweisen, gehen Sie in folgenden Schritten vor:

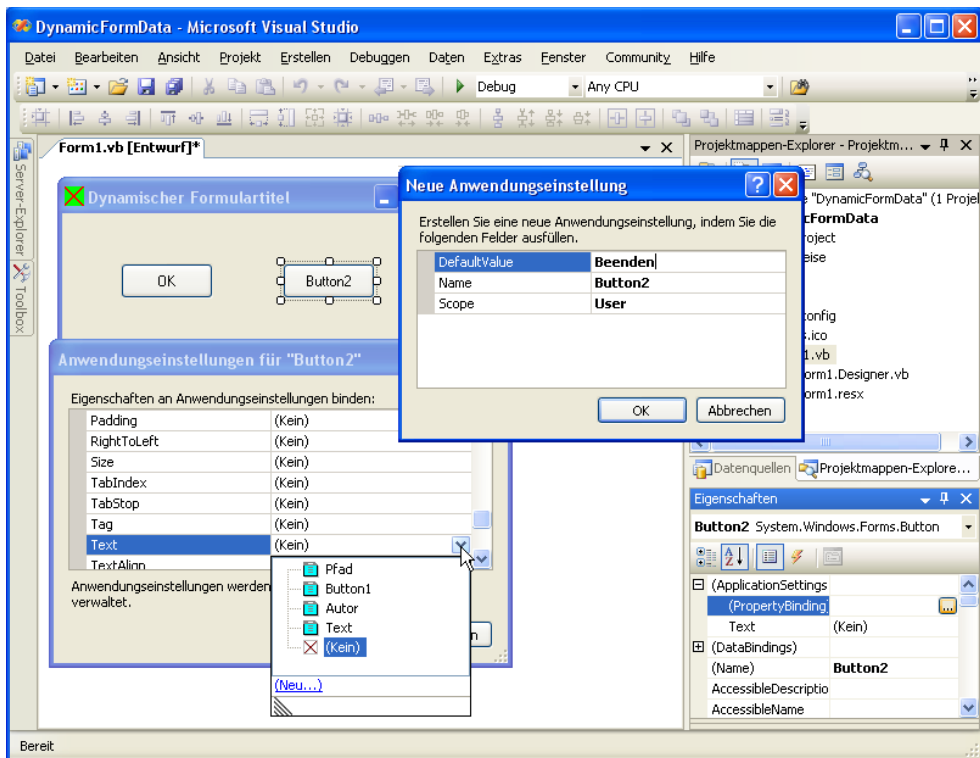


Abbildung 16.8: Festlegen von dynamischen Eigenschaften in der Entwicklungsumgebung

1. Wählen Sie in der Entwicklungsumgebung das Formular oder das Steuerelement im Ansicht-Designer an, damit dessen Eigenschaftsfenster angezeigt wird. Expandieren Sie anschließend im Eigenschaftsfenster den Zweig (*Application Settings*). Dann wird u.a. die Eigenschaft (*Property Binding*) angezeigt (Abbildung 16.8, rechts unten). Klicken Sie auf die Eigenschaftenzeile und dann auf die Schaltfläche der Eigenschaft (*Property Binding*).
2. Sobald das Dialogfeld *Anwendungseinstellungen für <Name>* erscheint (Abbildung 16.8, links unten im Hintergrund), suchen Sie in der Liste der Eigenschaften den gewünschten Eintrag. Soll beispielsweise die Beschriftung einer Schaltfläche oder der Formulartitel definiert werden, ist die Eigenschaft *Text* in der Liste zu suchen. Klicken Sie in der Zeile der betreffenden Eigenschaft auf die Schaltfläche zum Öffnen des Listenfelds.
3. Im Listenfeld werden ggf. bereits definierte Anwendungseinstellungen eingeblendet. Durch Anklicken eines Namens lässt sich der betreffenden Eigenschaft der für den Namen definierte Wert zuweisen. Klicken Sie auf den Eintrag (*kein*), wird die Zuweisung der Anwendungseinstellung aufgehoben.
4. Um einen neuen Eintrag anzulegen, klicken Sie auf den im Listenfeld eingeblendeten Hyperlink *Neu*. Im Dialogfeld *Neue Anwendungseinstellung* (Abbildung 16.8, Vordergrund) tippen Sie in der Zeile *Name* einen Bezeichner für die betreffende Anwendungseinstellung ein. In Abbildung 16.8 wurde der Bezeichner »Button2« gewählt, da dieser zum Objektnamen der betreffenden Schaltfläche passt. Anschließend stellen Sie in der Zeile *DefaultValue* den neuen Wert der betreffenden Anwendungseinstellung ein. Bezieht sich die Anwendungseinstellung auf eine Texteingenschaft, ist ein Text (z.B. der Beschriftungstext für die Schaltfläche) einzutragen. Bei anderen Eigenschaften stellt das Dialogfeld ggf. ein Listenfeld bereit, über das sich die zulässigen Werte (z.B. Wahrheitswerte *true*, *false*) abrufen lassen.

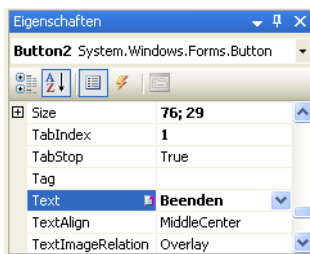


Abbildung 16.9: Markierung einer dynamischen Eigenschaft

Schließen Sie die geöffneten Dialoge über die OK-Schaltfläche, wird der betreffende Wert als Anwendungseinstellung in einer Konfigurationsdatei gespeichert. Gleichzeitig weist die Entwicklungsumgebung der Eigenschaft des betreffenden Steuerelements über die (*Property Binding*)-Eigenschaft den dynamischen Wert zu. Wenn Sie im Eigenschaftsfenster des betreffenden Formulars oder Steuerelements zur jeweiligen Eigenschaft blättern, ist dies mit einem kleinen »Fähnchen« versehen (Abbildung 16.9). Gleichzeitig wird der dynamisch definierte Wert der Anwendungseinstellung für die Eigenschaft eingeblendet. Sie können den Wert später direkt im Eigenschaftsfenster ändern. Die Entwicklungsumgebung schreibt dann die Änderung automatisch in die Konfigurationsdatei. Sobald Sie das Projekt mit dem so erstellten Formular übersetzen und ausführen, werden die dyna-

misch vergebenen Anwendungseinstellungen gelesen und den betreffenden Formular- und Steuerelementeigenschaften zugewiesen. Durch Anpassen der Konfigurationsdatei (siehe folgende Seiten) können Sie die Werte der Anwendungseinstellungen jederzeit und ohne Eingriff in den Quellcode oder Neuübersetzen des Projekts anpassen.

So setzen Sie allgemeine Anwendungseinstellungen

Möchten Sie allgemeine Anwendungseinstellungen (z.B. Pfadangaben) in einem Projekt vereinbaren? Dies lässt sich über die Seite *Einstellungen* der Projekteigenschaften veranlassen.

1. Klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf den Eintrag der Projektdatei und wählen Sie den Kontextmenübefehl *Eigenschaften*.
2. Rufen Sie die Seite *Einstellungen* ab (Abbildung 16.10). Dann werden Ihnen die bereits definierten Anwendungseinstellungen (auch für Steuerelemente vereinbare Einstellungen) aufgelistet und Sie können Einstellungen anpassen, löschen oder hinzufügen.

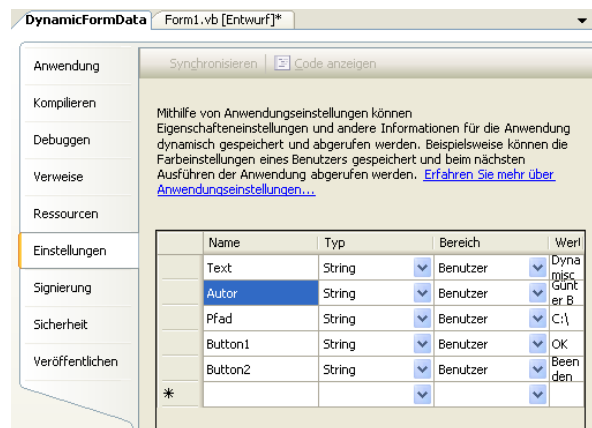


Abbildung 16.10: Definition der Anwendungseinstellungen

Zum Korrigieren bestehender Einstellungen klicken Sie die betreffenden Zellen in der Tabellenstruktur an. In der Rubrik *Name* der Tabellenstruktur können Sie den Namen der Einstellung ändern. Das Listenfeld der Spalte *Typ* erlaubt den Typ des Anwendungswerts (z.B. String, Byte etc.) festzulegen und über die Spalte *Bereich* lässt sich zwischen den Werten Benutzer und Anwendung umschalten. Die Spalte *Wert* weist dann den eigentlichen Wert der Anwendungseinstellung auf. Um eine neue Einstellung aufzunehmen, fügen Sie einfach einen Namen, den Wert und die Vorgaben für Typ und Bereich in der untersten Zeile ein. Löschen lässt sich eine Einstellung, indem Sie den Zeilenkopf mit der rechten Maustaste anwählen und den Kontextmenübefehl *Einstellung entfernen* wählen.

Hinweis

Sie können auch im Projektmappen-Explorer der Entwicklungsumgebung den Eintrag *My Projekt* per Doppelklick anwählen. Dann sollte die Eigenschaftenseite *Einstellungen* ebenfalls geöffnet werden.

Zugriff auf die Anwendungseinstellungen per Programmcode

Anwendungseinstellungen, die sich auf Steuerelemente beziehen, werden dynamisch durch das System verwaltet und automatisch im Programmcode zur Anzeige des Formulars berücksichtigt. Möchten Sie im Code auf einzelne Anwendungseinstellungen zugreifen (z.B. um einen Pfad zu lesen), lässt sich dies über das *My*-Objekt sehr leicht erledigen. Die folgende Programmsequenz greift auf verschiedene Anwendungseinstellungen zu und zeigt diese an:

```
Const title As String = "Anwendungseinstellungen"
Dim txt As String = ""
txt = "Autor: " & My.Settings.Autor & vbCrLf
txt = txt & "Pfad: " & My.Settings.Pfad & vbCrLf
txt = txt & "Text: " & My.Settings.Text & vbCrLf
txt = txt & "Button1: " & My.Settings.Button1 & vbCrLf
txt = txt & "Button2: " & My.Settings.Button2 & vbCrLf
MessageBox.Show(txt, Title)
```

Der Zugriffspfad für die Einstellungen lautet dabei immer *My.Settings.<Name>*, wobei *<Name>* für den Namen der betreffenden Einstellung steht.

Dem Benutzer Zugriff auf die Anwendungseinstellungen erlauben

Möchten Sie dem Benutzer das Anpassen der Anwendungseinstellungen innerhalb der .NET-Anwendung erlauben? Dies lässt sich sehr einfach über ein Formular mit eingefügtem *PropertyGrid*-Steuerelement realisieren. Fügen Sie das betreffende Steuerelement über die Toolbox in das Formular ein. Anschließend müssen Sie das Steuerelement zur Laufzeit an die Anwendungseinstellungen binden. Hierzu empfiehlt es sich, in der *Load*-Ereignisbehandlungsroutine des Formulars die folgende Anweisung zu hinterlegen:

```
Me.PropertyGrid1.SelectedObject = My.Settings
```

Die *SelectedObjekt*-Eigenschaft des Steuerelements gibt an, welche Eigenschaften aufzulisten sind. Hier wird der Wert auf *My.Settings* gesetzt, so dass das Steuerelement sämtliche Anwendungseinstellungen einblendet.

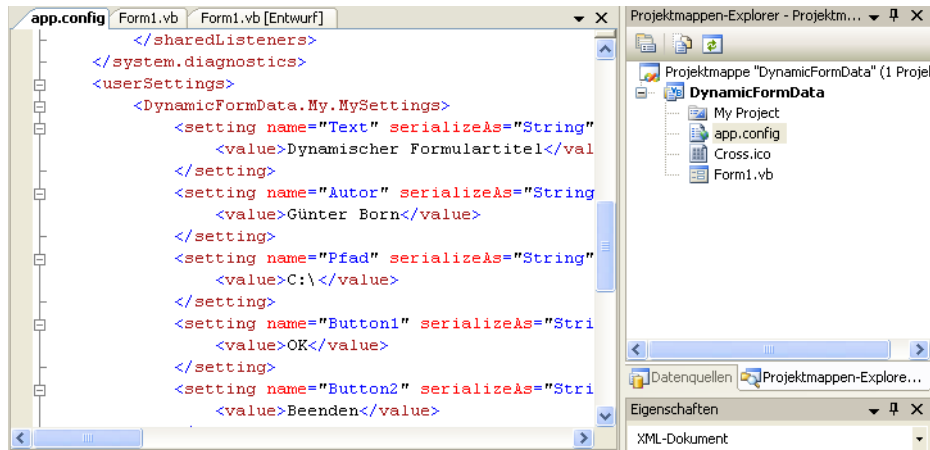


Abbildung 16.11: Anzeige der Anwendungseinstellungen als XML-Seite

Hinweis

Die Konfigurationsdaten werden von der Entwicklungsumgebung in der XML-Datei `<Name>.config` im Anwendungsordner hinterlegt. `<Name>` steht dabei für den Namen der Anwendungsdatei (hier *DynamicFormData.exe*). Diese XML-Datei lässt sich über den Projektmappe-Explorer der Entwicklungsumgebung über den Eintrag *app.config* im XML-Editor öffnen (Abbildung 16.11). Die Anwendungseinstellungen befinden Sie im Knoten *userSettings*. Falls Sie in Ihrer .NET-Anwendung keine Funktionen zum Ändern der Anwendungseinstellungen bereitstellen, lässt sich ein beliebiger XML-Editor zur Pflege der *.config*-Datei verwenden. Voraussetzung ist allerdings, dass der Editor die XML-Datei im »UTF-8«-Code schreiben kann. Beim Windows-Editor Notepad muss z.B. im Dialogfeld *Speichern* unter das Listenfeld *Codierung* auf den Wert »UTF-8« gesetzt werden. Wählen Sie eine andere Codierung, schreibt der Editor eine Bytesequenz (z.B. EF, BB, BF) als Codesignatur an den Anfang der Konfigurationsdatei. Dadurch wird die XML-Datei ungültig und ein Zugriffsversuch beim Start der Anwendung löst einen Laufzeitfehler aus. Im Ordner `\Beisp\Kap16\ConfigEdit` der Begleit-CD findet sich eine kleine .NET-Anwendung, die das Bearbeiten der Konfigurationsdatei ermöglicht. Es handelt sich um eine Variante der in Kapitel 12 entwickelten Anwendung *DateiDialoge*, bei der ich den Dateifilter für den Dateidialog auf **.config* gesetzt und die Funktionalität auf die Befehle *Öffnen*, *Speichern* und *Speichern unter* reduziert habe. Nach dem Start des im Unterordner `\Bin` hinterlegten Programms *ConfigEdit.exe* erscheint ein Fenster mit einem Textfeld. Die im Editor geladenen Anwendungseinstellungen lassen sich im Textfeld editieren und dann speichern. Dabei wird die korrekte Kodierung im »UTF-8«-Format berücksichtigt. Die Befehle zum Öffnen und Speichern von *.config*-Dateien lassen sich über die Menü- und Symbolleiste abrufen. Details zur Implementierung des Editors entnehmen Sie dem Quellcode der Projektdatei *Form1.vb*.

Hinweis (Fortsetzung)

Sie finden die Projektdateien des Beispiels *DynamicFormData* im Ordner `\Beisp\Kap16\DynamicFormData` auf der Begleit-CD. Die Eigenschaft *Text* passt den Formulartitel an, während die Eigenschaften *Button1* und *Button2* die Beschriftung der Schaltflächen beeinflussen. Die restlichen Eigenschaften sind als Beispiel definiert. Über die linke mit OK beschriftete Schaltfläche lassen sich die Werte der Anwendungseinstellungen in einem Dialogfeld auflisten. Zudem werden die Eigenschaften in einer Eigenschaftenseite des Formulars aufgeführt. Passen Sie dort einen der Werte für *Text*, *Button1* und *Button2* an, wirkt sich dies auf die Beschriftung des Formulars bzw. der Steuerelemente aus. Weitere Details sind dem Quellcode der Datei *Form1.vb* zu entnehmen.

16.2.2 Hilfedateien einbinden

Die bisherigen Beispiele boten nur eine Hilfe in Form eines kurzen *About*-Dialogs oder über ein einfaches Meldungsfeld mit ein paar Textzeilen. Zudem habe ich das Einbinden von Kontexthilfen für Schaltflächen über den *HelpProvider* in *Kapitel 9* kurz behandelt. Bei umfangreicheren Anwendungen möchten Sie aber sicherlich dem Benutzer eine Hilfeseite bereitstellen, die er über einen Eintrag im Hilfemenü oder über Schaltflächen abrufen kann (Abbildung 16.12).

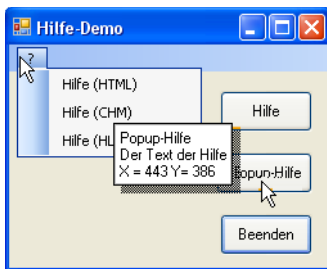


Abbildung 16.12: Hilfe anzeigen

Der Aufruf der Hilfe ist mit dem .NET Framework kein Problem. Sobald Sie die Klassen für die Gestaltung von Formularen im Projekt einbinden, steht auch die Klasse *Help* aus dem Namensraum *System.Windows.Forms* zur Verfügung. Diese Klasse erlaubt Ihnen über die *ShowHelp()*-Methode eine Hilfedatei aufzurufen. Die nachfolgende Codesequenz zeigt Ihnen die Ereignisbehandlungsroutine für das *Click*-Ereignis eines Menüeintrags und einer Schaltfläche:

```
Private Sub ItemH2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles ButtonH2.Click, _
    HilfeHTMLToolStripMenuItem.Click
    ' Benutzer hat Hilfe angefordert
    Help.ShowHelp(Me, "Hilfe.htm") ' Hilfedatei im HTML-Format
End Sub
```

Die *ShowHelp()*-Methode erwartet zwei Parameter. Im ersten Parameter ist eine Referenz auf das übergeordnete Steuerelement, welches die Hilfe bindet, zu übergeben. Hier habe ich *Me* eingesetzt, um das Hilfefenster an das Formular zu binden. Im zweiten Parameter ist der Name samt Pfad einer Hilfedatei einzusetzen. Im obigem Beispiel wurde einfach der Name einer HTML-Dokumentdatei eingetragen, die im Ordner der Programmdatei hinterlegt ist. Bei Bedarf können Sie den aktuellen Pfad über die bereits häufig eingesetzte Funktion *GetPath ()* (der von mit entwickelten *Tools*-Klasse) ermitteln und um einen relativen Pfad zur Hilfedatei ergänzen. Alternativ können Sie auch *hlp*-Dateien (z.B. *Help.ShowHelp(Me, "Hilfe.hlp")*) oder *chm*-Dateien (z.B. *Help.ShowHelp(Me, "Hilfe.chm")*) im zweiten Parameter mit angeben.

Hinweis

HTML-Dateien lassen sich mit einfachen Bordmitteln (z.B. mit dem Visual Web Developer oder notfalls sogar mit dem Texteditor der Entwicklungsumgebung) erstellen. Diese Art von Dateien eignen sich sehr gut, um dem Benutzer eine Übersicht über die Programmfunktionen zu bieten. Solche HTML-Dateien lassen sich mit dem HTML-Help-Compiler in *.chm*-Dateien überführen. Laut Hilfe ist dieser Compiler in Visual Studio enthalten – allerdings scheint Microsoft dessen Auslieferung vergessen zu haben. Sie können dieses Programm aber auch kostenlos aus dem Internet von Microsofts Webseiten (www.microsoft.com) beziehen. Suchen Sie einfach nach dem Begriff »HTML Workshop« oder nach »hcrtf«. Dateien im Windows *hlp*-Format werden dagegen durch den älteren Microsoft Help-Workshop erstellt. Dieses Programm ist bei vielen Microsoft-Werkzeugen dabei und lässt sich in älteren Fassungen aus dem Internet von der Microsoft-Webseite herunterladen. Auf der Begleit-CD finden Sie im Ordner *\Beisp\Kap16\Hilfedateien* eine Datei im Adobe-Acrobat-PDF-Format, die das Erstellen von Hilfeseiten mit diesem Help-Workshop und Microsoft Word skizziert. Zudem enthält der Ordner einige Hilfedateien im *.html*- *.chm*- und *.hlp*-Format.

Das .NET Framework bietet noch eine Möglichkeit, eine Popup-Hilfe zu verwenden. Die Methode *ShowPopup()* wird ebenfalls von der Klasse *Help* bereitgestellt. Die folgende Ereignisbehandlungsroutine demonstriert, wie sich diese Hilfe nutzen lässt:

```
Private Sub ButtonPop_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ButtonPop.Click
    ' Popup-Hilfe
    Dim x As Integer = Me.Left + Me.ButtonPop.Left - 30 ' Position
    Dim y As Integer = Me.Top + Me.ButtonPop.Top + 60
    Dim Punkt As System.Drawing.Point = New System.Drawing.Point(x, y)
    Dim PopupText As String = "Popup-Hilfe" & vbCrLf & _
        "Der Text der Hilfe" & vbCrLf & _
        "X = " & x.ToString & " Y= " & y.ToString
    Help.ShowPopup(Me, PopupText, Punkt)
End Sub
```


Der Methode ist im ersten Parameter das Handle des übergeordneten Steuerelements (hier *Me* für das Formular) zu übergeben. Der zweite Parameter enthält den im Popup-Fenster anzuzeigenden Text und der dritte Parameter definiert den Anzeigepunkt des Popup-Fensters als *Point*-Struktur.

Hinweis

Die Projektdateien des Beispiels finden Sie im Ordner `\Beisp\Kap16\Hilfe` auf der Begleit-CD. Die Hilfedateien liegen als Kopie im Ordner `\Bin` bei. Details zum Code finden Sie in der Datei *Form1.vb*.

16.3 Drucken in .NET-Anwendungen

Das Drucken aus .NET-Anwendungen ist ein eigenes Thema, da es keinen simplen *Print*-Befehl (z.B. zum Drucken des Inhalts einer *TextBox*) gibt. Allerdings bietet das .NET Framework einige Klassen, die eine Druckausgabe unterstützen (auch wenn es noch verschiedene Unzulänglichkeiten, z.B. beim Drucken mehrseitiger Dokumente mit Text und Grafik gibt). Um Ihnen zumindest die Ausgabe einfacher Texte zu ermöglichen, möchte ich die prinzipielle Vorgehensweise beim Drucken demonstrieren. Der folgende Abschnitt ist dabei so organisiert, dass Sie in einem ersten Beispiel quasi die aufs Essentielle reduzierte Möglichkeit zum Drucken sowie zur Anzeige der Seitenvorschau kennen lernen. In einem zweiten Beispiel habe ich die bereits in *Kapitel 12* entwickelte Minianwendung zum Umgang mit *Dateidialogen* um Funktionen zum Drucken und zur Seitenansicht erweitert. Dort dreht sich es sich um die Frage, wie sich beispielsweise mehrseitige Dokumente zu Papier bringen lassen.

16.3.1 Hinweise zum Drucken

Der Druckvorgang wird über im Namensraum *System.Drawing.Printing* bereitgestellte Dienste unterstützt. In diesem Namensraum findet sich die Klasse *PrintDocument*, welche die Druckeigenschaften bestimmt und die Druckaufbereitung eines Dokuments übernimmt. Dieses Objekt lässt sich im Ansicht-Designer aus der Toolbox als (nicht sichtbares) Steuerelement im Komponentenbereich des Formulardesigners einbinden. Alternativ können Sie eine Instanz im Programmcode erstellen. Die Klasse *Margins* stellt Eigenschaften zur Verwaltung der Seitenränder und Druckoptionen bereit.

Vor der reinen Druckausgabe ist noch etwas »Verwaltungskram« zu erledigen. In Windows ist der Benutzer es gewohnt, dass vor dem eigentlichen Ausdruck ein Druckdialog zur Auswahl der Druckoptionen (Auswahl des Druckers, Ausgabe in eine Datei, Seitenzahl etc.) angezeigt wird. Sie können das Steuerelement *PrintDialog* per Designer im Formular hinterlegen. Dann wird das betreffende Objekt zur Laufzeit automatisch instantiiert. Oder Sie instantiiieren das Objekt manuell im Programmcode. Über einen Aufruf der *ShowDialog()*-Methode dieses Objekts lässt sich der *Drucken*-Dialog anzeigen. Beendet der Benutzer diesen Dialog über die *Drucken*-Schaltfläche, muss die Anwendung die *Print()*-Methode des *PrintDialog*-Objekts aufrufen, um die Druckausgabe anzustoßen.

Die eigentliche Druckausgabe erfolgt in einer Ereignisbehandlungsroutine, die das *PrintPage*-Ereignis des *PrintDocument*-Objekts abfängt. Dort muss der Code hinterlegt werden, der den Dokumentausdruck über Aufrufe von *e.Graphics.DrawString* steuert. Dort ist auch die Programmlogik zur Aufteilung der Druckausgabe auf mehrere Seiten unterzubringen.

Neben diesen essentiellen Klassen stehen noch einige weitere Dienste für Zusatzdialoge zur Verfügung. Die meisten Anwendungen bieten eine Option *Seitenansicht*, bei der ein Abbild der Druckausgabe in einem Vorschauenfenster angezeigt wird. Das .NET Framework stellt für dieses Ansinnen eine eigene Klasse *PrintPreviewDialog* bereit, die sich entweder manuell im Programmcode oder automatisch als nicht sichtbares Steuerelement über den Ansicht-Designer im Formular instantiieren lässt. Die Instanz dieser Klasse bedient sich zur Laufzeit einfach der Methoden des *PrintDocument*-Objekts, um quasi einen virtuellen Druckvorgang anzustoßen. Die Ausgaben werden abgefangen und im Anzeigedialog als Seitenvorschau ausgegeben. Über den Dialog *Seite einrichten* kann der Benutzer bei den meisten Anwendungen ggf. die Seitenausrichtung sowie die Seitenränder festlegen. Auch so etwas lässt sich in .NET Framework über die *PageSetUp*-Klasse erledigen. Nachfolgend wird gezeigt, wie Sie die obigen Klassen zum Drucken nutzen können.

16.3.2 Druckfunktionen implementieren, so geht's

Die Implementierung einer Druckfunktion samt Seitenansicht und einer Option zum Einstellen der Seiteneigenschaften soll jetzt an einem sehr einfachen Beispiel demonstriert werden. Die Anwendung stellt dem Benutzer das in Abbildung 16.13 im Entwurf gezeigte Formular bereit. In einem Textfeld lässt sich ein einfacher Text (der aber nie mehr als eine Seite umfasst) eintippen. Die Schaltfläche *Seite einrichten* stellt einen entsprechenden Dialog zur Auswahl des Seitenformats (Hoch-/Querformat, Seitenränder) bereit. Über die Schaltfläche *Drucken* wird der Dialog zur Auswahl des Druckers aufgerufen und dann die Druckausgabe angestoßen. Die Schaltfläche *Vorschau* erzeugt eine Seitenansicht des eingegebenen Texts.

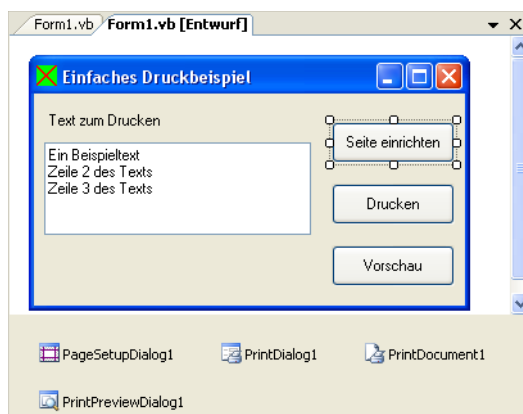


Abbildung 16.13: Formularentwurf der Beispielanwendung

In Abbildung 16.13 sehen Sie bereits die für das Beispiel benötigten, im Formular selbst nicht sichtbaren, Steuerelemente *PageSetupDialog1* (Seite einrichten), *PrintDialog1* (für den Druckdialog und das Drucken) und *PrintPreviewDialog1* (für die Seitenansicht). Zudem ist das *PrintDocument1*-Steuerelement zur Instantiierung der Klasse *PrintDocument* im Komponentenbereich des Ansicht-Designers aufgeführt.

Hinweis

Die Funktionen des Beispiels wurden so gewählt, dass die Implementierung sehr einfach wird (alles ist auf den Kern des absolut Notwendigen reduziert – der kurze Text erfordert z.B. keine Seitensteuerung bei der Druckausgabe). Zudem soll bei der Implementierung im Projekt möglichst viel Verwaltungsarbeit vom Codegenerator des Designers erledigt werden.

Die Implementierung des Beispiels in der Entwicklungsumgebung (Visual Studio 2005 bzw. Visual Basic 2005 Express Edition) erfolgt mit folgenden Schritten:

1. Legen Sie in der Entwicklungsumgebung ein neues Projekt als Windows-Anwendung an und ergänzen Sie das Formular um die in Abbildung 16.13 gezeigten sichtbaren Steuerelemente (Textfeld, Schaltflächen). Setzen Sie die Eigenschaften dieser Steuerelemente (z.B. die *Text*-Eigenschaft) gemäß den in der Abbildung gezeigten Vorgaben.
2. Fügen Sie im Formulardesigner je ein Steuerelement der Typen *PrintDocument* (zur Abwicklung der Druckausgabe), *PrintDialog* (zur Anzeige des Druckdialogs), *PrintPreview* (für die Seitenansicht) und *PageSetupDialog* (zur Anzeige des Dialogs *Seite einrichten* – Abbildung 16.18) zum Formular hinzu.

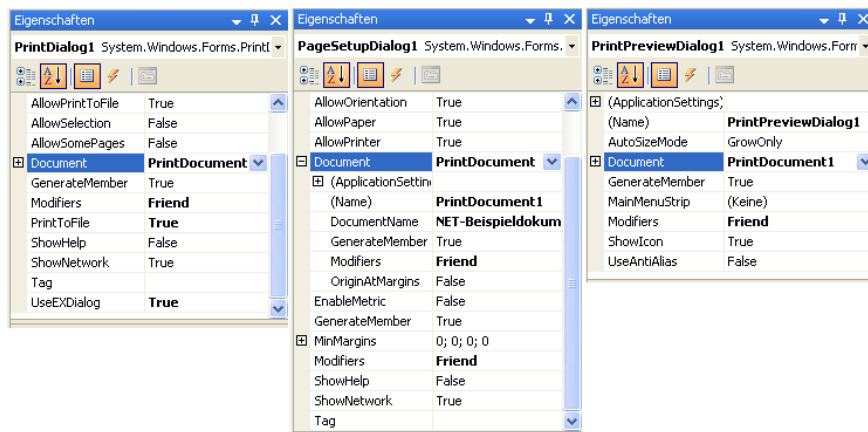


Abbildung 16.14: Eigenschaften der Steuerelemente

3. Klicken Sie im Designbereich auf das Symbol des unterhalb des Formulars angezeigten Steuerelements *PrintDialog*. Wechseln Sie zum Eigenschaftensfenster des Steuerelements und setzen Sie die Eigenschaft *Document* auf den Namen des *PrintDocument*-Steuerelements (Abbildung 16.14, links). Wurde das *PrintDocument*-Steuerelement im Formulardesign eingefügt, wird dessen Name (meist *PrintDocument1*) bereits im Listefeld der Eigenschaft zur Auswahl angezeigt.

4. Setzen Sie die restlichen Eigenschaften des *PrintDialog*-Steuerelements. Die Eigenschaft *AllowPrintToFile* ist auf *True* zu setzen, falls die Option zum Drucken in Dateien im Dialogfeld erscheinen soll. Wird die Eigenschaft *PrintToFile* auf *True* gesetzt, wird das Kontrollkästchen *Ausgabe in Datei umleiten* standardmäßig markiert. Die Eigenschaften *AllowSelection* und *AllowSomePages* bestimmen, ob die Optionsfelder zum Drucken einzelner Seiten oder zum Drucken markierter Bereiche im Dialogfeld freigegeben werden. Der Wert *True* in den letzten beiden Eigenschaften macht aber nur Sinn, falls Sie die betreffenden Funktionen auch in der Programmlogik der Druckausgabe implementieren.

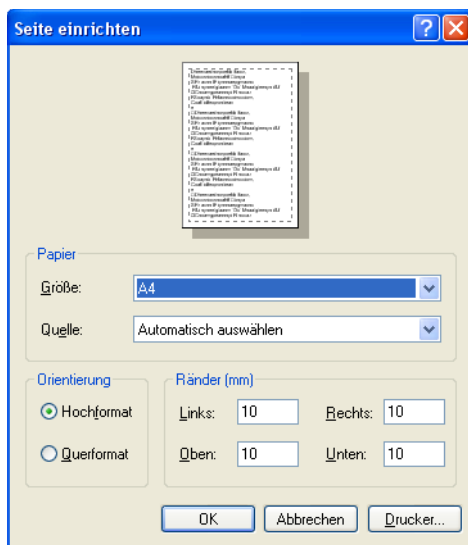


Abbildung 16.15: Dialogfeld *Seite einrichten*

5. Markieren Sie das Steuerelement *PageSetupDialog* und setzen Sie dessen Eigenschaften gemäß Abbildung 16.14 (Mitte). Standardmäßig sind die Eigenschaften für *AllowMargins*, *AllowOrientation*, *AllowPaper* und *AllowPrinter* auf *True* gesetzt, um die betreffenden Optionen im Dialogfeld freizugeben (Abbildung 16.15). Falls Sie den Knoten *Document* expandieren, können Sie in der Eigenschaft *PrintName* einen Text hinterlegen. Dieser Text legt den Dokumenttitel im Druckauftrag fest und wird in der Druckerwarteschlange des Druckmanagers für den betreffenden Auftrag angezeigt.
6. Markieren Sie das Steuerelement *PrintPreviewDialog* und setzen Sie dessen Eigenschaften *Document* auf den Namen des *PrintDocument*-Steuerelements (Abbildung 16.14, rechts). Dies stellt sicher, dass das Steuerelement zur Laufzeit auf die Druckfunktionen zugreifen kann.

Nach dieser Vorbereitung ist noch der Code in den *Click*-Ereignisbehandlungsroutinen der drei Schaltflächen zu ergänzen (ohne diesen Code wird sich bei Anwahl der Schaltflächen wenig tun).

Code zum Einrichten der Seite

Der Dialog zum Einrichten der Seite (Abbildung 16.15) wird in der *Click*-Ereignisbehandlungsroutine des entsprechenden Bedienelements aufgerufen. Sofern kein *PageSetupDialog*-Steuerelement im Design eingefügt wurde, setzen Sie als Erstes die folgende Anweisung in der Ereignisbehandlungsroutine ein:

```
Dim PageSetupDialog1 As New PageSetupDialog()
```

Im aktuellen Beispiel fehlt diese Zeile, da die Entwicklungsumgebung das Objekt *PageSetupDialog1* wegen des eingefügten Steuerelements beim Instantiieren des Formulars automatisch mit generiert. Die Anzeige des Dialogfelds *Seite einrichten* erfolgt mit der Anweisung

```
Me.PageSetupDialog1.ShowDialog()
```

Das wäre es im Grunde gewesen, da das *PageSetupDialog*-Objekt über die Eigenschaft *Document* mit dem *PrintDocument*-Objekt verbunden ist. Damit kann das *PageSetupDialog*-Objekt vom Benutzer vorgenommene Seiteneinstellungen in den Eigenschaften des *PrintDocument*-Objekts anpassen. Leider gibt es in der zugehörigen Klasse einen Fehler, der zu falschen Seiteneinstellungen führt, falls metrische Einheiten (mm) für die Seitenränder benutzt werden. Intern benutzt .NET Framework Werte (z.B. für Seitenränder), die in 1/100 Inch (Zoll) angegeben werden. Beim Aufruf des Dialogfelds *Seite einrichten* erfolgt jedoch keine korrekte Umrechnung der Inch-Angaben in Millimeter (statt der Standardvorgabe von 1 Inch für die Seitenränder werden im Dialogfeld dann 10 mm angegeben). Ändert der Benutzer Einstellungen für die Seitenränder, werden diese fehlerhaft in Inch zurückgerechnet. Daher muss diese Umrechnung bei einem auf metrische Einheiten eingestellten System »zu Fuß« korrigiert werden. Konkret sind vor dem Aufruf des Dialogfelds die Werte für die Seitenränder um den Faktor 2,54 zu erhöhen und nach dem Schließen um den Faktor 2,54 zu verringern.

Die *Click*-Ereignisbehandlungsroutine der Schaltfläche *Seiten einrichten* enthält daher etwas Zusatzcode. Zuerst wird über *Globalization.RegionInfo.CurrentRegion* die Ländereinstellung gelesen und der Variablen *met* zugewiesen. Dann kontrolliert das Programm in einer *If*-Anweisung, ob die Ländereinstellungen ein metrisches Einheitensystem verwenden. In diesem Fall müssen die in der Eigenschaft *DefaultPageSettings.Margins* hinterlegten Werte für die Seitenränder um den Faktor 2,54 erhöht werden. Anschließend aktiviert die Prozedur über *ShowDialog()* die Anzeige des Dialogfelds *Seite einrichten*. In der *If*-Anweisung wird geprüft, ob der Benutzer den Dialog über die *OK*-Schaltfläche verlassen hat. In diesem Fall werden alle Randeinstellungen der *Margins*-Eigenschaft um den Faktor 2,54 reduziert.

```
Private Sub ButtonPage_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ButtonPage.Click
    ' Schaltfläche "Seite einrichten" angeklickt

    ' Hole die aktuellen Regionseinstellungen
    Dim met As Globalization.RegionInfo = _
```

```
Globalization.RegionInfo.CurrentRegion

With Me.PrintDocument1.DefaultPageSettings.Margins
  If met.IsMetric Then ' metrisches System ?
    ' ### Bug: Korrigiere Umrechnung Inch/Millimeter
    .Left = CInt(.Left * 2.54)
    .Right = CInt(.Right * 2.54)
    .Top = CInt(.Top * 2.54)
    .Bottom = CInt(.Bottom * 2.54)
  End If
  ' Seite einrichten-Dialog anzeigen
  If Me.PageSetupDialog1.ShowDialog() = _
    System.Windows.Forms.DialogResult.OK Then
    If met.IsMetric Then
      ' ### Bug: Korrigiere Umrechnung Inch/Millimeter
      .Left = CInt(.Left / 2.54)
      .Right = CInt(.Right / 2.54)
      .Top = CInt(.Top / 2.54)
      .Bottom = CInt(.Bottom / 2.54)
    End If
  End If
End With
End Sub
```

Listing 16.3: Anzeige des Dialogfelds Seite einrichten mit Umrechnung der Einheiten

Achtung

Der obige Code wird mit den aktuellen Versionen des .NET Framework funktionieren. Falls Microsoft später einen Service-Pack herausgibt, der die Klasse korrigiert, müssen Sie die betreffenden Codeteile zur Korrektur des Umrechnungsfehlers natürlich löschen. Allerdings bleibt festzuhalten, dass dieser Fehler bereits seit der Version 1.0 des .NET Framework existiert und bisher nicht korrigiert wurde. Vermutlich wird Microsoft aus Kompatibilitätsgründen auch nichts anderes übrig bleiben, als die korrekte Umrechnung dem Entwickler der Anwendung zu überlassen.

Die vom Benutzer vorgenommenen Seiteneinstellungen bleiben während der Sitzung erhalten. Um nicht beim ersten Aufruf des Dialogfelds *Seite einrichten* mit auf 25,4 mm eingestellten Seitenrändern zu starten (dies entspricht der Standardvorgabe von 1 Zoll der betreffenden Klasse), werden die aktuellen Seitenabmessungen im aktuellen Beispiel in der *Load*-Ereignisbehandlungsroutine auf 0 gesetzt.

Drucken-Dialog hinzufügen und Druckausgabe anstoßen

Die Befehle zur Anzeige des *Drucken*-Dialogs (Abbildung 16.16) sowie zum Anstoßen der Druckausgabe sind in der *Click*-Ereignisbehandlungsroutine der Schaltfläche *Drucken* zu hinterlegen.

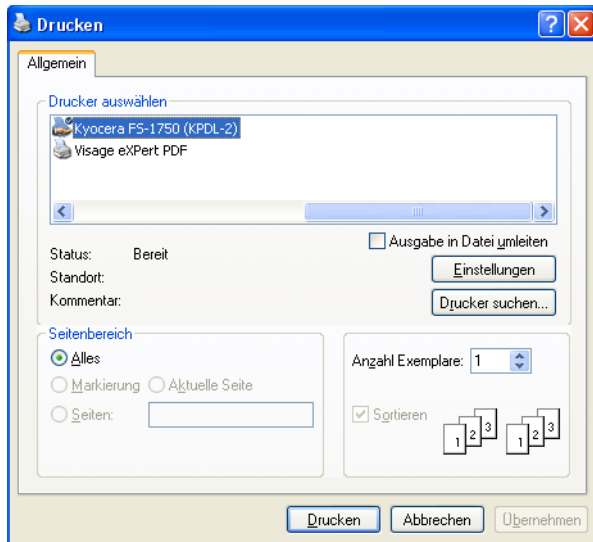


Abbildung 16.16: Dialogfeld DRUCKEN

Sofern beim Formularentwurf kein *PrintDialog*- und kein *PrintDocument*-Steuerelement im Designer eingefügt wurde, können Sie diese auch implizit instantiieren:

```
Dim PrintDialog1 As New PrintDialog()
Dim PrintDocument1 As New PrintDocument
PrintDialog1.Document = PrintDocument1
```

Die letzte Anweisung weist die *PrintDocument*-Objektinstanz der Eigenschaft *Document* des *PrintDialog*-Objekts zu. Damit kann das Objekt *PrintDocument* auf die aktuellen, vom Benutzer gewählten und im *PrintDialog*-Objekt hinterlegten, Einstellungen zugreifen. Dadurch ist dem *PrintDocument*-Objekt z.B. bekannt, welcher Drucker vom Benutzer ausgewählt wurde. Im aktuellen Beispiel ist der obige Code nicht mehr erforderlich, da die benötigten Steuerelemente sowie die Zuordnung der Eigenschaften im Ansicht-Designer im Formular eingefügt wurden.

Anschließend lässt sich der *Drucken*-Dialog (Abbildung 16.16) mittels der *ShowDialog*-Methode des *PrintDialog*-Objekts anzeigen:

```
If (Me.PrintDialog1.ShowDialog() = _
    System.Windows.Forms.DialogResult.OK) Then
    Me.PrintDocument1.Print()      ' Druckausgabe anstoßen
End If
```

Im hier gezeigten Code wird in einer *If*-Anweisung geprüft, ob der Benutzer das Dialogfeld über die *Drucken*-Schaltfläche verlässt. In diesem Fall ist der Rückgabewert der *PrintDialog.ShowDialog()*-Methode auf *DialogResult.OK* gesetzt. Dann lässt sich die Druckausgabe durch Aufruf der *Print()*-Methode des *PrintDocument*-Objekts starten.

Hinweis

Die Angabe *Me* im Code zeigt, dass die im Formular eingefügten Steuerelemente *PrintDialog1*- und *PrintDocument1* benutzt werden. Visual Basic erlaubt Ihnen in einer Ereignisbehandlungsprozedur den Bezeichner *Me* wegzulassen und nur mit dem Objektnamen zu arbeiten. Haben Sie die Objekte manuell im Code instantiiert, verwenden Sie natürlich die im Programm deklarierten Objektnamen.

Implementieren der *PrintPage*-Ereignisbehandlung

Mit den obigen Vorbereitungen haben Sie zwar ein Formular, welches funktionierende Schaltflächen aufweist. Aber beim Anklicken der *Drucken*-Schaltfläche wird sich nichts tun. Das *PrintDocument*-Objekt »weiß einfach nicht«, was es drucken soll. Daher löst das *PrintDocument*-Objekt beim Drucken *PrintPage*-Ereignisse aus. In der zugehörigen Ereignisbehandlungsroutine erfolgt dann die eigentliche Ausgabe der Druckdaten. In obigem Beispiel habe ich dafür gesorgt, dass nur ein kurzer Text, der auf jeden Fall auf eine Seite passt, auszugeben ist. Dann wird der Code der Ereignisbehandlungsroutine sehr kompakt:

```
Private Sub PrintDocument1_PrintPage(ByVal sender As Object, _  
    ByVal e As System.Drawing.Printing.PrintPageEventArgs) _  
    Handles PrintDocument1.PrintPage  
    ' Wickelt das Drucken ab - hier: Inhalt der Textbox  
    ' mit Arial 12 Pkt, Farbe Schwarz ausgeben  
    Dim leftMargin As Single = e.MarginBounds.Left  
    Dim topMargin As Single = e.MarginBounds.Top  
  
    ' Ausgabe: Text, Schriftart, Pinselfarbe, x,y, Format)  
    e.Graphics.DrawString(Me.TextBox1.Text, _  
        New Font("Arial", 12), Brushes.Black, _  
        leftMargin, topMargin, New StringFormat())  
    e.HasMorePages = False    ' fertig  
End Sub
```

Der Prozedur wird im Argument *e* ein *PrintPageEventArgs*-Objekt übergeben, welches Methoden und Eigenschaften bereitstellt. Der Ausdruck erfolgt über die *Graphics.DrawString()*-Methode des als Argument übergebenen *PrintPageEventArgs*-Objekts. Als erster Parameter sind der Methode die auszudruckenden Daten (hier Text) zu übergeben. Zusätzlich benötigt die Methode noch die Angaben über den beim Drucken zu verwendenden Font sowie die Zeichenfarbe. Der zweite Parameter legt dabei die Schriftart fest, die ich im Beispiel mit einer neuen *Font*-Instanz auf eine 12 Punkt Arial-Schriftart einstelle.

Der dritte Parameter definiert die Druckfarbe (*Brushes.Black*) des Zeichenobjekts (hier wurde die Farbe *Schwarz* für den Pinsel gewählt). Die nächsten beiden Parameter legen die X- und Y-Position, bezogen auf die linke obere Ecke der Seite, fest. Die Seitenränder werden dabei vom Objekt *e* als Eigenschaften angegeben. Hier habe ich diese Angaben übernommen, da es die erste Zeile der Seite ist. Bei mehreren Zeilen müssen Sie die Y-Koordinate jeweils erhöhen, um den Text zeilenweise untereinander zu drucken. Der letzte Parameter gibt an, dass die Druckausgabe die Daten als Text formatieren soll.

Nach dem Aufruf der *Graphics.DrawString*-Methode lässt sich dem *PrintDocument*-Objekt mitteilen, ob noch Daten zur Ausgabe anstehen. Da wir hier den gesamten Text des *TextBox*-Steuerelements gedruckt haben, wird die Eigenschaft *HasMorepages* auf *False* gesetzt, d.h., die Druckausgabe wird beendet. Beim Wert *True* würde die Ereignisprozedur vom *PrintDocument*-Objekt erneut aufgerufen und könnte weitere Daten drucken.

16.3.3 Realisierung der Seitenansicht

Die Implementierung der Funktion zur Seitenansicht (Abbildung 16.17) setzt voraus, dass Sie das Steuerelement *PrintPreviewDialog* entweder im Ansicht-Designer im Formular hinterlegt oder das betreffende Objekt über folgende Anweisungen im Code instantiiert haben:

```
Dim dlg As New PrintPreviewDialog() ' hole neue Vorschau-Instanz
dlg.Document = PrintDocument      ' Dokument an Klasse binden
dlg.ShowDialog()
```

Die erste Zeile instantiiert das *PrintPreviewDialog*-Objekt. Die zweite Zeile bindet die Instanz des *PrintDocument*-Objekts über die Eigenschaft *Document* an die Druckausgabe. Dadurch kann die Seitenansicht die Druckfunktionen von *PrintDocument* mitbenutzen. Die letzte Anweisung in obiger Sequenz ruft die Anzeige des Dialogfelds *Seitenansicht* auf. Im aktuellen Beispiel reduziert sich der Code der *Click*-Ereignisbehandlungsprozedur der Schaltfläche *Vorschau* auf diese Zeile:

```
Me.PrintPreviewDialog1.ShowDialog()
```

Die Instantiierung des *PrintPreviewDialog*-Objekts sowie die Zuweisung der *Document*-Eigenschaft wird durch vom Designer automatisch generierten Code beim Instantiieren des Formulars erledigt.

Der Benutzer kann im Dialogfeld über verschiedene Schaltflächen die Zoomdarstellung variieren. Bei mehrseitigen Dokumenten ließe sich auch eine Miniaturansicht der betreffenden Seiten einblenden (sofern das Drucken mehrerer Seiten unterstützt wird).

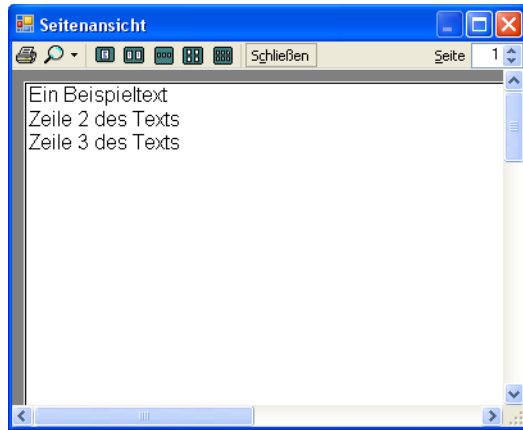


Abbildung 16.17: Dialogfeld Seitenansicht mit einem Dokument

Hinweis

Sie finden die Projektdateien des Beispiels im Ordner `\Beisp\Kap16\SimplePrint` auf der Begleit-CD. Details zum Code finden Sie in der Datei `Form1.vb`.

16.3.4 Mehrseitiges Drucken geht auch

In den wenigsten Fällen werden sich Ihre Anwendungen darauf beschränken, nur einige Zeilen Text auszugeben. Sobald ein Dokument etwas umfangreicher wird, müssen Sie sich um das Thema »Drucken mehrerer Seiten« kümmern. Die in Abbildung 16.18 gezeigte Anwendung wurde aus dem bereits in *Kapitel 12* entwickelten Beispiel zur Anzeige der Dateidialoge abgeleitet. Dort stand noch die Implementierung der Funktionen zum Drucken und zur Anzeige der Seitenansicht aus. Dies soll jetzt mit den gerade vermittelten Kenntnissen nachgeholt werden. Dabei lernen Sie auch gleich die Überlegungen kennen, wie sich eine Druckausgabe eventuell auf mehrere Seiten aufteilen lässt:

1. Legen Sie in der Entwicklungsumgebung ein neues Projekt als Windows-Anwendung an oder verwenden Sie ein bestehendes Projekt. Ergänzen Sie das Projekt um die benötigte Funktionalität. Hier habe ich einfach das Beispiel *DateiDialoge* aus *Kapitel 12* kopiert und die Projektdatei umbenannt.
2. Fügen Sie im Formulardesigner des Anwendungsformulars je ein Steuerelement der Typen *PrintDocument* (zur Abwicklung der Druckausgabe), *PrintDialog* (zur Anzeige des Druckdialogs), *PrintPreview* (für die Seitenansicht) und *PageSetupDialog* (zur Anzeige des Dialogs *Seite einrichten*) hinzu.
3. Legen Sie die Eigenschaften der Steuerelemente gemäß meinen Ausführungen im vorherigen Beispiel fest.

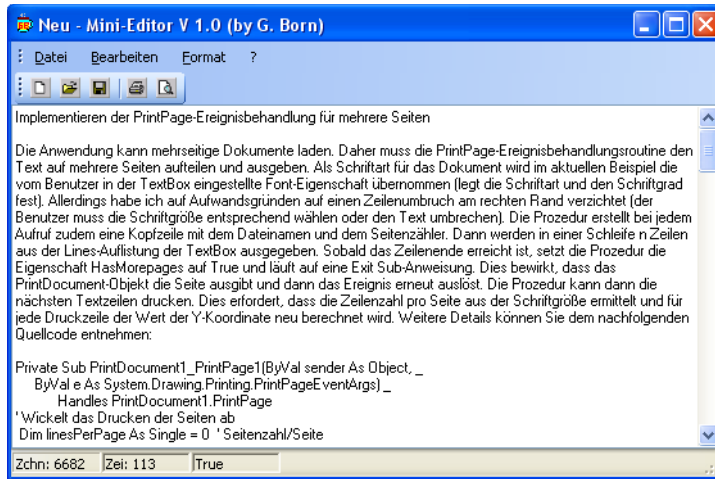


Abbildung 16.18: Fenster der Beispielanwendung

Nach dieser Vorbereitung müssen Sie den Code zur Anzeige der Steuerelemente, die Anweisungen zum Anstoßen der Druckausgabe sowie die Befehle zur Aufbereitung der Daten beim Drucken implementieren. Sie können im ersten Ansatz den Code aus den *Click*-Ereignisbehandlungsroutinen des vorherigen Druckbeispiels verwenden und in die Prozeduren *Print_Doc* und *ViewDoc* übertragen. Fügen Sie zudem die komplette *PrintPage*-Ereignisbehandlungsprozedur des vorherigen Beispiels im Codebereich des Formulars ein. Zudem sollten Sie das Beispiel noch um einen Menüeintrag *Seite einrichten* im Menü *Datei* erweitern und die *Click*-Ereignisbehandlungsroutine mit dem im obigen Beispiel gezeigten Code ergänzen.

Hinweis

Die Beschreibung des Codes zum Hinzufügen eines Dialogs *Seite einrichten* spare ich mir daher (die Implementierung im aktuellen Beispiel gleicht dem Ansatz der vorhergehenden Seiten). Bei den beiden Prozeduren zum Aufruf der Seitenvorschau (*ViewDoc*) sowie zum Anstoßen des Druckens (*Print_Doc*) werden zusätzlich zwei Variable *page = 0* und *zeile = 0* gesetzt, um einen Seitenzähler und einen Lesezeiger in die *Lines*-Auflistung der TextBox zu initialisieren.

Implementieren der PrintPage-Ereignisbehandlung für mehrere Seiten

Die Anwendung kann mehrseitige Dokumente laden. Daher muss die *PrintPage*-Ereignisbehandlungsroutine den Text auf mehrere Seiten aufteilen und ausgeben. Als Schriftart für das Dokument wird im aktuellen Beispiel die vom Benutzer in der TextBox eingestellte *Font*-Eigenschaft übernommen (legt die Schriftart und den Schriftgrad fest). Allerdings habe ich aus Aufwandsgründen auf einen Zeilenumbruch am rechten Rand verzichtet (der Benutzer muss die Schriftgröße entsprechend wählen oder den Text umbrechen). Die Prozedur erstellt bei jedem Aufruf zudem eine Kopfzeile mit dem Dateinamen und dem Seitenzähler. Dann werden in einer Schleife n Zeilen aus der *Lines*-

Auflistung der *TextBox* ausgegeben. Sobald das Zeilenende erreicht ist, setzt die Prozedur die Eigenschaft *HasMorepages* auf *True* und läuft auf eine *Exit Sub*-Anweisung. Dies bewirkt, dass das *PrintDocument*-Objekt die Seite ausgibt und dann das Ereignis erneut auslöst. Die Prozedur kann dann die nächsten Textzeilen drucken. Dies erfordert, dass die Zeilenzahl pro Seite aus der Schriftgröße ermittelt und für jede Druckzeile der Wert der Y-Koordinate neu berechnet wird. Weitere Details können Sie dem nachfolgenden Quellcode entnehmen:

```
Private Sub PrintDocument1_PrintPage1(ByVal sender As Object, _
    ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
    Handles PrintDocument1.PrintPage
    ' Wickelt das Drucken der Seiten ab (unterstützt mehrere Seiten)
    Dim linesPerPage As Single = 0 ' Seitenzahl/Seite
    Dim yPos As Single = 0 ' y-Pos. Druckzeile
    ' aktueller Zeilenzähler der Seite, starte mit 2
    Dim count As Integer = 2 ' wegen Kopfzeile + Leerzeile
    Dim leftMargin As Single = e.MarginBounds.Left ' linker Rand
    Dim topMargin As Single = e.MarginBounds.Top ' oberer Rand
    Dim line As String = Nothing ' Puffer für Text aus TextBox

    Dim printFont As Font ' Schriftart zum Drucken
    Dim maxCount As Integer ' Zeilenzahl im Dokument (TextBox)

    page += 1 ' Seitenzähler erhöhen

    If file <> "" Then ' Kopfzeile vorbereiten
        line = "Datei: " & file & " Seite: " & page.ToString
    Else
        line = "Neues Dokument Seite: " & page.ToString
    End If

    printFont = TextBox1.Font ' Schriftart aus TextBox.Font
    maxCount = TextBox1.Lines().GetUpperBound(0) ' TextBox.Text-Zeilen

    ' Zeilenzahl pro Seite berechnen
    linesPerPage = e.MarginBounds.Height / _
        printFont.GetHeight(e.Graphics)

    ' jetzt die Kopfzeile ausgeben
    ' damit die Zeile auch bei langen Dateinamen auf Seite passt
    e.Graphics.DrawString(line, printFont, Brushes.Black, _
        leftMargin, topMargin, New StringFormat())
```

Listing 16.4: Druckausgabe für mehrere Seiten

```

While count < linesPerPage      ' Jetzt die Zeilen lesen und drucken
  If zeile >= maxCount Then
    e.HasMorePages = False      ' fertig
    Exit Sub                    ' Beenden
  End If
  line = Me.TextBox1.Lines(zeile) ' hole Textzeile
  ' Textzeile ausgeben - überlange Zeilen werden abgeschnitten
  yPos = topMargin + count * printFont.GetHeight(e.Graphics)
  e.Graphics.DrawString(line, _
    printFont, Brushes.Black, _
    leftMargin, yPos, New StringFormat())
  count += 1 ' Zeilen pro Seite gedruckt
  zeile += 1 ' Index in Lines()-Auflistung erhöhen
End While
e.HasMorePages = True ' es gibt noch Zeilen
End Sub

```

Listing 16.4: Druckausgabe für mehrere Seiten (Forts.)

Hinweis

Die Projektdateien des Beispiels finden Sie im Ordner `\Beisp\Kap16\Drucken`. Sobald Sie eine Textdatei entsprechender Länge laden oder Text eintippen, lässt sich über die Seitenvorschau verifizieren, dass der Inhalt auf mehrere Druckseiten aufgeteilt wird. Details zur Implementierung der Funktionen finden Sie im Code der Datei `Form1.vb`. Im Ordner `\Beisp\Kap16\DruckenExt` finden Sie eine etwas modifizierte Variante, in der zur Anzeige der Seitenvorschau eine eigene Komponentenkasse `PreviewDocument.vb` benutzt wird. Diese Klasse überschreibt einige `PrintDocument`-Methoden. Sinnvoll ist ein solcher Ansatz aber nur, falls Sie unterschiedliche Prozeduren für Druckausgabe und Seitenansicht benötigten. Schauen Sie sich den Code dieser Klasse hinsichtlich der Details an.

16.4 Grafikprogrammierung in .NET

.NET Framework stellt über die `Graphics`-Klasse eine Programmierschnittstelle GDI+ bereit. Damit lassen sich Grafiken oder Text in Windows-Forms (bzw. im Formular enthaltene Steuerelemente) zeichnen. Zudem kann die Schnittstelle benutzt werden, um Bilder als Objekte zu bearbeiten. Nachfolgend wird an einigen Beispielen der Einsatz verschiedener Methoden der `Graphics`-Klasse zum Zeichnen demonstriert.

16.4.1 Einstieg in das Zeichnen mit der Graphics-Klasse

Der Zugriff auf die GDI+ Programmierschnittstelle erfolgt über die Methoden der *Graphics*-Klasse. Diese stellt diverse Zeichenmethoden bereit und kapselt die GDI+-Zeichenoberfläche. Als Zeichenoberfläche wird meist die Oberfläche eines Formulars oder eines Steuerelements benutzt. Über verschiedene Einstellungen lässt sich dabei die gesamte Oberfläche des Objekts oder ein Ausschnitt zum Zeichnen verwenden. Dabei gibt es zwei Möglichkeiten, an eine *Graphics*-Zeichenoberfläche heranzukommen.

- Einmal lässt sich der Code zur Implementierung der Zeichenfunktionen in der *Paint*-Ereignisbehandlungsroutine des betreffenden Formulars oder Steuerelements unterbringen. Der Ereignisbehandlungsroutine wird ein Argument *e* vom Typ *PaintEventArgs* übergeben, welches über die *Graphics*-Eigenschaft den Zugriff auf die Zeichenoberfläche ermöglicht.
- Die Alternative besteht darin, dass Sie per Programm das *Graphics*-Zeichenobjekt des betreffenden Formulars oder Steuerelement mittels der *CreateGraphics()*-Methode instantiieren. Dann lässt sich über dieses Objekt auf den Zeichenbereich zugreifen.

Beide Ansätze werden in den nachfolgenden Beispielen skizziert. Sobald Sie über ein *Graphics*-Objekt verfügen, können Sie auf dessen Methoden zugreifen, um im Zeichenbereich etwas zu tun. Der Ablauf für eine Zeichenoperation ist innerhalb eines Programms dabei weitgehend unabhängig von der gewählten Zeichenmethode und lässt sich in folgenden Schritten beschreiben:

- Erzeuge einen Zeichenbereich als Objekt der *Graphics*-Klasse gemäß den beiden oben beschriebenen Ansätzen.
- Erzeuge die für die Zeichenoperation benötigten Ressourcenobjekte (Pinsel, Brush, Text). Ein *Pen*-Objekt dient zum Zeichnen von einfarbigen Linien oder Flächenelementen, während ein *Brush*-Objekt z.B. beim Zeichnen von Mustern zum Einsatz kommt. Zum Ausgeben von Texten benötigen Sie ein *Font*-Objekt und die *TextRenderer*-Klasse.
- Rufe die vom *Graphics*-Objekt bereitgestellte Methode zum Ausführen der Zeichenoperation auf. Je nach Methode sind verschiedene Parameter mit den Ressourcenobjekten, Koordinatenangaben etc. zu übergeben.

Alle Zeichenoperationen finden dabei ausschließlich innerhalb des vom *Graphics*-Objekt definierten Zeichenbereich statt. Positionen innerhalb des Zeichenbereichs werden in Bildpunkten (Pixel) angegeben, wobei der Koordinatennullpunkt in der linken oberen Ecke des Zeichenbereichs liegt. Sobald die Zeichenoperation abgeschlossen ist und die erzeugten Ressourcenobjekte nicht mehr benötigt werden, sind diese über die *Dispose()*-Methode nach Möglichkeit wieder frei zu geben. Die *Graphics*-Klasse stellt dabei eine Vielzahl an Methoden zum Zeichnen, zum Füllen von Formen und zur Bearbeitung von Bitmap-Grafiken (Bildbearbeitung) bereit.

Name	Bemerkung
<i>DrawLine()</i>	Erlaubt Linien einer vorgegebenen Stärke in der angegebenen Farbe zu zeichnen
<i>DrawEllipse()</i>	Zeichnet Kreise und Ellipsen in ein umrahmendes Rechteck, wobei Linienfarbe und -dicke vorgegeben werden

Name	Bemerkung
<i>DrawArc()</i>	Zeichnet Kreisbögen mit der vorgegebenen Linienfarbe und -dicke
<i>DrawRectangle()</i>	Zeichnet ein Rechteck mit der vorgegebenen Linienfarbe und -stärke
<i>DrawPolygon()</i>	Zeichnet ein durch mehrere Koordinatenpunkte vorgegebenes Polygon (Kurvenzug) in der vorgegebenen Linienfarbe und -stärke
<i>DrawBezier()</i>	Zeichnet eine durch vier Koordinatenpunkte vorgegebene Bèzier-Spline-Kurve in der vorgegebenen Linienfarbe und -stärke
<i>DrawString()</i>	Zeichnet die angegebene Textzeichenfolge an der angegebenen Position mit dem angegebenen <i>Brush</i> -Objekt und <i>Font</i> -Objekt in der gewählten Farbe
<i>DrawImage()</i>	Zeichnet eine Bitmap- oder Metafile-Grafik an der angegebenen Position und in der vorhandenen Größe
<i>FillEllipse()</i>	Zeichnet einen gefüllten Kreis oder eine gefüllte Ellipse mit der angegebenen Farbe
<i>FillRectangle()</i>	Zeichnet ein gefülltes Rechteck mit der angegebenen Farbe
<i>FillPolygon()</i>	Zeichnet ein Polygon und füllt dessen Fläche
<i>FillPath()</i>	Füllt das Innere eines Grafikpfads aus
<i>FillClosedCurve()</i>	Füllt das Innere einer geschlossenen Kurve und erlaubt unregelmäßig geformte Flächen zu zeichnen

Tabelle 16.2: Von der Graphics-Klasse bereitgestellte Methoden

Weiterhin stellt die *TextRenderer*-Klasse eine *DrawText()*-Methode zur Ausgabe von Texten im Zeichenbereich bereit. Über die *Clear()*-Methode lässt sich zudem der Zeichenbereich wieder löschen. Je nach Methode gibt es dabei verschiedene Aufrufvarianten hinsichtlich der möglichen Parameter. Die Details lassen sich in der Hilfe zum .NET Framework nachschlagen. Die Verwendung verschiedener Methoden lernen Sie in den nachfolgend gezeigten Beispielen kennen.

16.4.2 Zeichnen in der Paint-Ereignisbehandlungsroutine

Der Code zur Implementierung der Zeichenfunktionen lässt sich in der *Paint*-Ereignisbehandlungsroutine des betreffenden Formulars oder Steuerelements unterbringen. Dieser Prozedur wird vom System als zweites Argument die Variable *e* vom Typ *System.Windows.Forms.PaintEventArgs* übergeben. Diese Instanz liefert über die *Graphics*-Eigenschaft ein *Graphics*-Objekt zurück, welches eine Ausgabe an den Grafikpuffer zulässt. Um einen Zugriff auf den *Graphics*-Zeichenbereich zu erhalten, lässt sich dann folgende Anweisung verwenden:

```
Dim g As Graphics = e.Graphics
```

Es wird hier die *Graphics*-Eigenschaft des übergebenen *PaintEventArgs*-Objekts gelesen und einer Variablen (hier *g*) zugewiesen. Anschließend können Sie über diese Objektvariable auf die Methoden der *Graphics*-Klasse zugreifen. Die *Graphics*-Klasse stellt dabei gemäß obiger Tabelle eine Vielzahl an Methoden zum Zeichnen (*DrawArc()*, *DrawBezier()*, *DrawEllipse()*, *DrawImage()*, *DrawLine()*, *DrawPolygon()*, *DrawRectangle()* und

DrawString(), zum Füllen von Formen (*FillClosedCurve()*, *FillEllipse()*, *FillPath()*, *FillPolygon()* und *FillRectangle()*) sowie weitere Methoden zur Bildbearbeitung bereit. Je nach verwendeter Methode werden dabei unterschiedliche Argumente übergeben.

Um beispielsweise eine Linie zu zeichnen, müssen Sie zuerst ein *Pen*-Objekt (quasi der Pinsel zum Zeichnen) definieren. Dieses Objekt legt über Eigenschaften die Zeichenfarbe sowie die Stiftstärke fest und wird mit folgender Anweisung vereinbart:

```
Dim mypen As Pen = New Pen(Color.Blue, 3)
```

Die Anweisung erzeugt hier einen roten Zeichenstift mit drei Pixel Stiftbreite. Um dann eine Linie zu zeichnen, lässt sich die *DrawLine()*-Methode aufrufen:

```
g.DrawLine(mypen, 30, 30, 150, 100)
```

Die Methode erwartet neben dem *Pen*-Objekt im ersten Argument noch weitere Parameter mit den Koordinaten (X,Y) des Anfangs- und des Endpunkts als Integerwerte. Um mit dem Pen ein Rechteck zu zeichnen, lässt sich die *DrawRectangle()*-Methode verwenden:

```
g.DrawRectangle(mypen, 10, 10, 100, 150)
```

Auch hier erwartet die Methode neben dem *Pen*-Objekt die Koordinaten des Anfangs- und des Endpunkts.

Um beispielsweise einen Text auszugeben, kommt die *DrawText()*-Methode zum Einsatz. Diese wird allerdings von der *TextRenderer*-Klasse bereitgestellt und erwartet ein *Graphics*-Zeichenobjekt als ersten Parameter.

```
TextRenderer.DrawText(e.Graphics, "Ein Text", _  
    fnt, New Point(100, 150), Color.Aqua)
```

Im zweiten Argument wird der zu zeichnende Text übergeben, während der dritte Parameter ein *Font*-Objekt spezifiziert. Dieses lässt sich mit:

```
Dim fnt As New Font("Arial", 25)
```

vereinbaren und definiert die Schriftart sowie den Schriftgrad. Als vierter Parameter erwartet die Methode den Anfangspunkt (X,Y) zur Textausgabe, während der letzte Parameter die Textfarbe angibt. Alternativ lässt sich auf die *DrawString()* des *Graphics*-Objekts zurückgreifen, die z.B. mit folgenden Parametern aufgerufen werden kann:

```
g.DrawString("Noch ein Text", fnt, Brushes.Blue, 50, 200)
```

Im ersten Argument steht der auszugebende Text, während der zweite Parameter ein *Font*-Objekt mit der Schriftart und der -größe übergibt. Der dritte Parameter definiert ein *Brush*-Objekt mit der Schriftfarbe und in den beiden letzten Argumenten wird das Koordinatenpaar für den Anfangspunkt der Ausgabe spezifiziert.

Das *Paint*-Ereignis wird immer dann aufgerufen, wenn sich etwas an der Zeichenoberfläche des Objekts ändert (z.B. Ändern der Fenstergröße, Fenster wird neu angezeigt etc.). Zum Zeichnen müssen in der *Paint*-Ereignisbehandlungsroutine verschiedene Objekte jeweils neu erzeugt werden. Diese belegen aber Ressourcen im Arbeitsspeicher. Ist die Zeichenoperation abgeschlossen und benötigen Sie die erzeugten Objekte nicht mehr? Sofern die zugehörige Klasse eine *Dispose()*-Methode unterstützt, sollten Sie diese aufrufen, um das selbst erzeugte Objekt und damit die belegten Ressourcen freizugeben. Dies könnte folgendermaßen aussehen:

```
mypen.Dispose() ' Pen freigeben
fnt.Dispose()   ' Font-Objekt freigeben
g.Dispose()     ' Grafikbereich freigeben
```

Die *Dispose()*-Methode eines Objekts dürfen Sie dann nicht aufrufen, wenn dieses nicht im Programm erzeugt wurde (z.B. bei vom Laufzeitsystem vordefinierte Muster). Geben Sie ein solches Objekt über *Dispose()* trotzdem frei, löst dies beim nächsten Zugriffsversuch (z.B. beim erneuten Instanzieren) einen Laufzeitfehler aus.

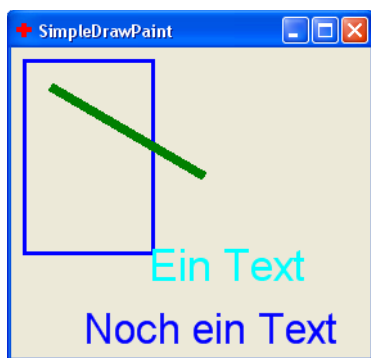


Abbildung 16.19: Grafikausgaben in einem Formular

Die folgende Programmsequenz zeigt eine *Paint*-Ereignisbehandlungsroutine, die eine Linie, ein Rechteck und zwei Texte auf der Oberfläche eines Formulars zeichnet. Das Ergebnis dieser Anweisungen ist in Abbildung 16.19 zu sehen.

```
Private Sub Form1_Paint1(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.PaintEventArgs) _
    Handles Me.Paint
    ' Paint-Ereignisbehandlung
    Dim g As Graphics = e.Graphics ' erzeuge Zeichenbereich
    Dim mypen As Pen               ' für Zeichenstift definieren
    Dim fnt As New Font("Arial", 25) ' Schriftart Arial, 25 Punkt
    mypen = New Pen(Color.Blue, 3)  ' Stift blau, 3 Punkt dick
    g.DrawRectangle(mypen, 10, 10, 100, 150) ' zeichne ein Rechteck
```

Listing 16.5: Zeichenausgabe in der *Paint*-Ereignisbehandlungsprozedur

```

mypen = New Pen(Color.Green, 7) ' Stift grün, 10 Punkt dick
g.DrawLine(mypen, 30, 30, 150, 100) ' dicke grüne Linie zeichnen

' jetzt einen Text ausgeben
TextRenderer.DrawText(e.Graphics, "Ein Text", _
    fnt, New Point(100, 150), Color.Aqua)

' Text mit DrawString() ausgeben
g.DrawString("Noch ein Text", fnt, Brushes.Blue, 50, 200)

mypen.Dispose() ' Pen freigeben
fnt.Dispose()   ' Font-Objekt freigeben
g.Dispose()     ' Grafikbereich freigeben
End Sub

```

Listing 16.5: Zeichenausgabe in der Paint-Ereignisbehandlungsprozedur (Forts.)

Hinweis

Sie finden die Projektdateien dieses Beispiels im Ordner `\Beisp\Kap16\Simple-DrawPaint` auf der Begleit-CD. Details zu den einzelnen im Beispiel benutzten Methoden finden Sie in der Hilfe zum .NET Framework, wenn Sie auf der Indexseite den Methodennamen angeben.

Tipp

Möchten Sie den Code einer *Paint*-Ereignisbehandlungsroutine debuggen? Dann tritt das Problem auf, dass beim Erreichen eines Haltepunkts beim Wechsel zwischen den Fenstern der Anwendung und der Entwicklungsumgebung weitere *Paint*-Ereignisse auslöst werden. Das Debuggen wird erleichtert, wenn Sie die Eigenschaft *TopMost* des Formulars auf den Wert *true* setzen (ist bei den hier gezeigten Beispielen der Fall). Dann bleibt das Formular nach dem Aufruf im Vordergrund und Sie können den Code der *Paint*-Ereignisbehandlungsroutine durch Anklicken der *Debugger*-Schaltflächen des im Hintergrund sichtbaren Fensters der Entwicklungsumgebung ablaufen lassen. Um ein neues *Paint*-Ereignis auszulösen, minimieren Sie das Formular und stellen es über die Taskleiste wieder her.

16.4.3 Zeichenbereich per Programm holen und zeichnen

Soll in der Anwendung gezielt, z.B. bei Anwahl einer Schaltfläche, etwas in einen Zeichenbereich ausgegeben werden? Dann muss der *Graphics*-Zeichenbereich im Programm mit folgenden Anweisungen definiert werden:

```
Dim g As System.Drawing.Graphics ' für Zeichenbereich
g = Me.PictureBox1.CreateGraphics() ' Grafikbereich auf PictureBox
```

Die erste Zeile vereinbart die Objektvariable *g* für den Zeichenbereich. In der zweiten Zeile wird die *CreateGraphics()*-Methode eines Formulars oder Steuerelements aufgerufen. Die Methode liefert das *Graphics*-Objekt an die Variable *g* zurück. Anschließend können Sie die Methoden des *Graphics*-Objekts zum Zeichnen verwenden. Die folgenden beiden Anweisungen erzeugen ein *Pen*-Objekt und zeichnen dann ein Rechteck im Zeichenbereich:

```
mypen = New Pen(Color.Blue, 3) ' Stift blau, 3 Punkt dick
g.DrawRectangle(mypen, 10, 10, 100, 150) ' zeichne ein Rechteck
```

Der Code entspricht den Anweisungen aus dem vorhergehenden Beispiel. An dieser Stelle vielleicht noch eine Bemerkung über den Zeichenbereich. Im vorhergehenden Beispiel wurde der komplette Bereich des Formulars zum Zeichnen benutzt. Im aktuellen Beispiel habe ich jedoch ein (leeres) *PictureBox*-Steuerelement als Container für den Zeichenbereich verwendet. Dies hat den Vorteil, dass Zeichenoperationen, die außerhalb der Umrisse des Steuerelements liegen, automatisch durch eine Clipping-Funktion unterdrückt werden. Es lassen sich also Steuerelemente wie Schaltflächen zur Bedienung der Anwendung im Formular positionieren und Sie können sicher sein, dass sich Zeichenoperationen nur auf den Container (hier das *PictureBox*-Element) beziehen.

Hinweis

Die Größe eines Objekts (Formular, Steuerelement) und damit auch der Zeichenbereich lässt sich über die *ClientSize*-Eigenschaft einschränken. Der Eigenschaft ist ein Wert vom Typ *Size* (Breite, Höhe) zuzuweisen. Soll die Größe des Objekts erhalten bleiben, der Zeichenbereich aber begrenzt werden, weisen Sie der *Clip*-Eigenschaft des *Graphics*-Objekts einen entsprechenden *Region*-Wert zu (siehe folgende Seiten).

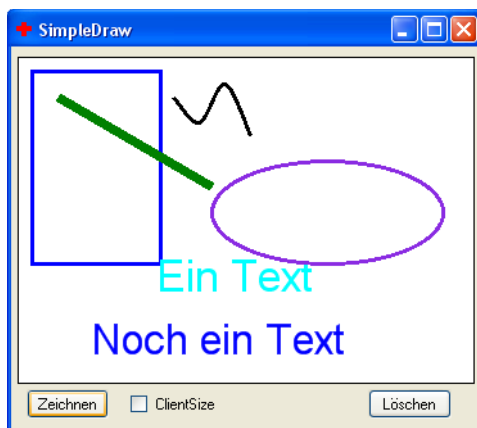


Abbildung 16.20: Grafikausgaben in einer PictureBox

In Abbildung 16.20 ist ein solches Formular zu sehen. Ein *PictureBox*-Steuerelement mit weißem Hintergrund dient als Zeichenfläche. Die *Zeichnen*-Schaltfläche bewirkt das Zeichnen einer Linie, eines Rechtecks, einer Kurve, einer Ellipse und zweier Texte. Die Schaltfläche *Löschen* erlaubt den Zeichenbereich zu löschen. Ein markiertes Kontrollkästchen *ClientSize* reduziert die Größe der *PictureBox* im Formular, um den Zeichenbereich zu begrenzen.

Die Ausgabe der Zeichenobjekte passiert in der nachfolgend vorgestellten Prozedur *Zeichnen*. Die Prozedur wird über die *Click*-Ereignisbehandlungsroutine der Schaltfläche *Zeichnen* aufgerufen.

```
Sub Zeichnen()
    Dim g As System.Drawing.Graphics ' für Zeichenbereich
    Dim mypen As Pen                 ' für Zeichenstift definieren
    Dim fnt As New Font("Arial", 25) ' Schriftart Arial, 25 Punkt
    Static pts(3) As Point           ' für Punkte

    g = Me.PictureBox1.CreateGraphics() ' Grafikber. auf PictureBox
    mypen = New Pen(Color.Blue, 3)      ' Stift blau, 3 Punkt dick
    g.DrawRectangle(mypen, 10, 10, 100, 150) ' zeichne ein Rechteck

    mypen = New Pen(Color.Green, 7)     ' Stift grün, 10 Punkt dick
    g.DrawLine(mypen, 30, 30, 150, 100) ' dicke grüne Linie zeichnen
    ' jetzt einen Text ausgeben
    TextRenderer.DrawText(g, "Ein Text", _
        fnt, New Point(100, 150), Color.Aqua)

    ' jetzt Text mit DrawString() ausgeben
    g.DrawString("Noch ein Text", fnt, _
        Brushes.Blue, 50, 200)

    ' jetzt einen dünnen Kreis zeichnen
    mypen = New Pen(Color.Aqua, 3)
    g.DrawEllipse(mypen, 150, 80, 180, 80)
    ' ein Kurve zeichnen
    mypen = New Pen(Color.Black, 3)
    pts(0) = New Point(120, 30)
    pts(1) = New Point(140, 50)
    pts(2) = New Point(160, 20)
    pts(3) = New Point(180, 60)
    g.DrawCurve(mypen, pts)
```

Listing 16.6: Zeichenausgabe in das *PictureBox*-Steuerelement

```

mypen.Dispose() ' Pen freigeben
fnt.Dispose()   ' Font freigeben
g.Dispose()     ' Grafikbereich freigeben
hasDrawn = True ' Zeichenbereich gefüllt
End Sub

```

Listing 16.6: Zeichenausgabe in das PictureBox-Steuerelement (Forts.)

Gegenüber dem obigen Beispiel wurde in dieser Prozedur noch ein Aufruf zum Zeichnen einer Ellipse aufgenommen.

```
g.DrawEllipse(mypen, 150, 80, 180, 80)
```

Die *DrawEllipse()*-Methode erwartet neben dem *Pen*-Objekt noch zwei Koordinatenpaare, die das umschließende Rechteck der Ellipse angeben. Eine Kurve lässt sich über die *DrawCurve()*-Methode zeichnen. Diese erwartet neben dem *Pen*-Objekt im ersten Parameter ein Feld mit Koordinatenangaben für die betreffenden Punkte:

```
g.DrawCurve(mypen, pts)
```

Die *Löschen*-Schaltfläche im Formular setzt den Zeichenbereich über die *Clear()*-Methode wieder zurück. Als Argument erwartet die Methode einen Farbwert. Die betreffende Prozedur *Clear_Draw()* mit den nachfolgenden Anweisungen wird aus der *Click*-Ereignisbehandlungsroutine der *Löschen*-Schaltfläche aufgerufen:

```

Sub Clear_Draw()
    ' Lösche den Zeichenbereich
    Dim g As System.Drawing.Graphics ' für Zeichenbereich
    g = Me.PictureBox1.CreateGraphics() ' auf PictureBox
    g.Clear(Color.White) ' Löschen des Zeichenbereichs
    hasDrawn = False      ' Zeichenbereich wieder leer
End Sub

```

Listing 16.7: Löschen des Zeichenbereichs

Jetzt gilt es noch ein kleines Problem zu lösen: Die Zeichenoperationen werden beim Anklicken der *Zeichnen*-Schaltfläche durchgeführt. Vergrößert der Benutzer später das Formular, ändert sich auch die Größe des Zeichenbereichs. Da aber die *Paint*-Ereignisbehandlungsroutine nicht implementiert wurde, bleibt der Inhalt des Zeichenbereichs unverändert – d.h., die bereits gezeichneten Objekte erscheinen abgeschnitten. Der Benutzer erwartet aber, dass die Zeichnung nach der Anpassung der Fenstergröße komplett sichtbar ist. Lösen lässt sich das Problem, indem Sie die entsprechenden Zeichenfunktionen aus der *Paint*-Ereignisbehandlungsroutine heraus anstoßen. Die Alternative: Sie verwenden eines der Ereignisse *Resize*, *ResizeBegin* und *ResizeEnd*, die bei Änderungen der Formulargröße ausgelöst werden. Im aktuellen Beispiel bewirkt die *ResizeEnd*-Ereignisbehandlungsroutine den erneuten Aufruf der Prozedur *Zeichnen*. Da der Zeichenbereich bei Größenänderungen des Formulars noch leer sein kann, wird vor dem

Aufruf der Prozedur geprüft, ob ein internes Flag *hasDraw* den Wert *true* besitzt. Das Flag wird beim Laden des Formulars, beim Löschen des Zeichenbereichs sowie beim Zeichnen aktualisiert.

Die Größe des *PictureBox*-Steuerelements wird in der *Click*-Ereignisbehandlungsroutine des Kontrollkästchens *ClientSize* angepasst. Bei markiertem Kontrollkästchen reduziert die Prozedur die Größe der *PictureBox*. Wird die Markierung wieder gelöscht, ist die *PictureBox* wieder so zu vergrößern, dass das Formular weitgehend ausgefüllt wird. Um Formularänderungen zu berücksichtigen, wird die Größe dynamisch aus den aktuellen Formularabmessungen bestimmt. Hierzu ermittelt die Anwendung in der *Load*-Ereignisbehandlungsroutine des Formulars die Größendifferenz zwischen Formular und *PictureBox*. Die Differenzwerte werden in den Variablen *dx* bzw. *dy* hinterlegt. Anschließend wird in den Zweigen einer *If*-Anweisung der *ClientSize*-Wert reduziert bzw. auf die Ursprungsgröße zurückgesetzt. Der Quellcode der *Click*-Ereignisbehandlungsroutine des Kontrollkästchens *ClientSize* ist in nachfolgendem Listing zu sehen:

```
Private Sub CheckBox1_CheckedChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles CheckBox1.CheckedChanged
    ' Kontrollkästchen "ClientSize" angeklickt
    ' Ändere die ClientSize-Werte der PictureBox,
    ' um den Zeichenbereich zu verändern
    If Me.CheckBox1.Checked Then ' ClientSize einschränken
        Me.PictureBox1.ClientSize = _
            New Size(Me.Size.Width - dx - 100, _
                Me.Size.Height - dy - 100)
    Else
        ' voller Bereich
        Me.PictureBox1.ClientSize = _
            New Size(Me.Size.Width - dx, _
                Me.Size.Height - dy)
    End If
    If hasDrawn Then ' ist bereits etwas gezeichnet?
        ' ### Achtung: Refresh muss aufgerufen werden,
        ' ### da sonst GDI-Zeichenmethode ggf. mit der
        ' ### Anpassung der ClientSize-Einstellung
        ' ### noch nicht fertig ist, wenn das Zeichnen
        ' ### beginnt -> dann bleibt alter Inhalt sichtbar
        Me.PictureBox1.Refresh() ' neu aufbauen
        Zeichnen() ' und neu zeichnen
    End If
End Sub
```

Listing 16.8: Anpassen der *ClientSize*-Eigenschaft

Bezüglich des Quellcodes der Ereignisbehandlungsroutine sind zwei Dinge anzumerken. Beim Formularentwurf wurde die *MinSize*-Eigenschaft des Formulars so gesetzt, dass der Zeichenbereich der PictureBox immer genügend Raum für eine Verkleinerung um 100 x 100 Pixel besitzt. Daher wurde auf eine Prüfung auf negative Werte bei der Reduzierung des *ClientSize*-Werts verzichtet. Zudem gibt es noch ein Problem zu lösen. Das Zurücksetzen des *ClientSize*-Werts auf die ursprünglichen Abmessungen bewirkt kein Neuzeichnen der Zeichenobjekte (da ja keine *Paint*-Ereignisbehandlung implementiert wurde). Die Idee, einfach nach einer *ClientSize*-Änderung noch die *Zeichnen()*-Prozedur aufzurufen, funktioniert nicht. Die Funktionsaufrufe werden über die Routinen der GDI+-Klasse an Windows weitergeleitet. Dort werden die Zeichenausgaben ggf. vor der *ClientSize*-Änderung durchgeführt. Erst ein Aufruf der *Refresh()*-Methode des *PictureBox1*-Steuerelements löst dieses Problem. Sie können dies sehr leicht prüfen, indem Sie die betreffende Anweisung auskommentieren.

Hinweis

Sie finden die Projektdateien dieses Beispiels im Ordner `\Beisp\Kap16\SimpleDraw` auf der Begleit-CD. Details können Sie dem Quellcode des Moduls *Form1.vb* entnehmen. Hinweise zu den betreffenden Methoden sowie den verschiedenen Möglichkeiten zur Überladung der Operatoren finden Sie in der Hilfe zum .NET Framework.

Clipping beim Zeichnen anwenden

Auf den vorhergehenden Seiten wurde bereits erwähnt, dass das Objekt die Zeichenfläche bestimmt und dass sich der Zeichenbereich über die *ClientSize*-Eigenschaft einschränken lässt. Soll die Größe des Zeichenbereichs eines Objekts unverändert bleiben, aber bestimmte Zeichenfunktionen sind auf eine Region zu begrenzen? Dann ist eine Clipping-Funktion recht hilfreich, die alle Zeichenoperationen außerhalb der Region unterdrückt. Hierzu bietet die *Graphics*-Klasse den Member *Clip*. Über die Werte dieser Eigenschaft lässt sich der Zeichenbereich des betreffenden Objekts begrenzen. Die Anwendung ist sehr einfach.

```
g = Me.PictureBox1.CreateGraphics()
g.Clip = New Region(New Rectangle(10, 10, 100, 80))
```

Die erste Anweisungszeile erzeugt ein *Graphics*-Objekt mittels der *CreateGraphics()*-Methode. Anschließend wird der *Clip*-Eigenschaft des *Graphics*-Objekts ein neuer *Region*-Wert mit dem Zeichenbereich zugewiesen. Hier wird der *Region*-Wert über einen *Rectangle*-Wert definiert. Anschließend können Zeichenoperationen nur noch innerhalb dieses Clippbereichs erfolgen.

Der Pfiff bei der ganzen Sache liegt darin, dass keine Änderungen am betreffenden Steuerelement erfolgen. Es wird lediglich der Clippbereich der Zeichenfläche eingeschränkt. Verwerfen Sie das *Graphics*-Objekt über *Dispose()* und erzeugen Sie dieses über *CreateGraphics()* neu, gilt wieder der ursprüngliche Clippbereich.

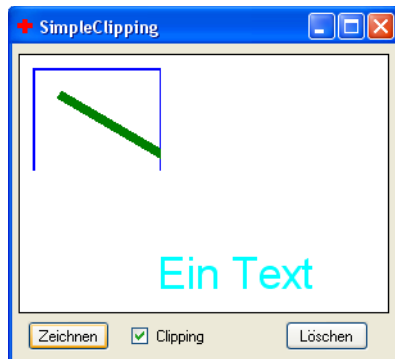


Abbildung 16.21: Grafikausgaben in einer PictureBox mit Clip-Bereich

Abbildung 16.21 zeigt ein aus den vorherigen Beispielen abgeleitetes Formular. Über die Schaltfläche *Zeichnen* lässt sich die Zeichenfunktion anstoßen, während *Löschen* den Zeichenbereich zurücksetzt. Ist das Kontrollkästchen *Clipping* markiert, setzt die entsprechende Zeichenroutine einen Clipbereich.

Bei Anwendung der *Clip*-Eigenschaft sind aber einige Dinge zu beachten. Einmal bewirkt das Setzen der *Clip*-Region keine sichtbare Änderung im Zeichenbereich (bereits gezeichnete Inhalte bleiben im Zeichenbereich erhalten). Sie müssen die *Clear()*-Methode aufrufen, um ggf. die Auswirkung der *Clip*-Einstellungen sehen zu können. Weiterhin fällt in Abbildung 16.21 auf, dass offenbar ein Text außerhalb des Clipbereichs liegt. Dieser Text wird hier über die *DrawText()*-Methode der *TextRenderer*-Klasse ausgegeben. Der Methode wird zwar das *Graphics*-Objekt zum Zeichnen übergeben. Dessen *Clip*-Eigenschaft bleibt aber unberücksichtigt.

Hinweis

Sie finden die Projektdateien dieses Beispiels im Ordner `\Beisp\Kap16\SimpleClipping` auf der Begleit-CD. Um die Wirkung des *Clipping*-Kontrollkästchens besser studieren zu können, wird in dessen *Click*-Ereignisbehandlungsroutine der Zeichenbereich gelöscht und dann die Prozedur zum Zeichnen aufgerufen. Wenn Sie die betreffenden Anweisungen auskommentieren, lässt sich die Wirkungsweise des Clipping im Zeichenbereich nur erkennen, wenn Sie die *Löschen*-Schaltfläche betätigen oder die Formulargröße ändern. Details können Sie dem Quellcode des Moduls *Form1.vb* entnehmen.

16.4.4 Beispiel: Anwendung weiterer Zeichenfunktionen

Das folgende Beispiel benutzt ein Formular mit zwei Registerkarten (Abbildung 16.22) und demonstriert die Anwendung weiterer Zeichenmethoden.

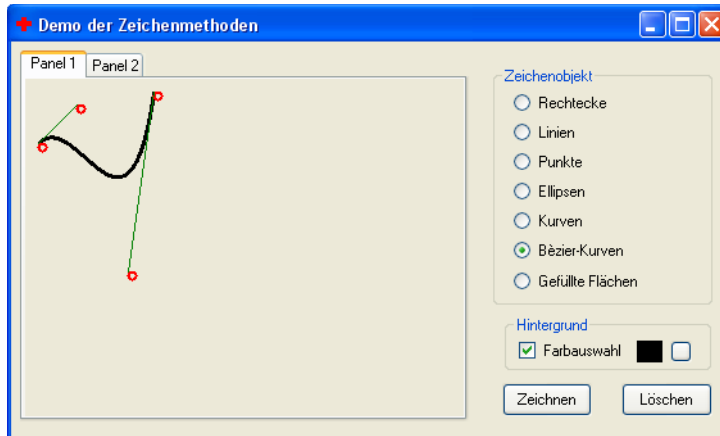


Abbildung 16.22: Demonstration von Zeichenfunktionen

Über Optionsfelder lassen sich verschiedene Zeichenfunktionen abrufen. Die Zeichenoperationen werden ausgeführt, sobald ein *Paint*-Ereignis (z.B. beim Wechsel zwischen den beiden Registerkarten) ausgelöst wird. Weiterhin kann der Benutzer die *Zeichnen*-Schaltfläche anwählen, um die Zeichenfunktion der aktuell gewählten Zeichenoption anzuzeigen. Die Funktionen benutzen dabei den Zeichenbereich der aktuellen Registerkarte. Werden mehrere Optionen nacheinander angewählt und jeweils über die *Zeichnen*-Schaltfläche bestätigt, führt dies zu einer Überlagerung der Zeichenelemente. Der Zeichenbereich lässt sich aber über die *Löschen*-Schaltfläche zurücksetzen. Standardmäßig wird der Zeichenbereich dabei auf die Hintergrundfarbe des Steuerelements gesetzt (diese wurde im Designer über die Eigenschaft *BackColor* vorgegeben). Markiert der Benutzer das Kontrollkästchen *Farbauswahl*, lässt sich eine individuelle Farbe für das Löschen des Zeichenbereichs wählen. Zum Einstellen der Farbe wird der *Color*-Dialog über die Schaltfläche der Gruppe aufgerufen. Die zuletzt gewählte Farbe wird über ein Farbfeld eingeblendet.

Farbauswahl und Realisierung eines Farbfelds

Die Farbe lässt sich vom Benutzer über den *Color*-Dialog auswählen. Der betreffende Wert findet sich in der *Color*-Eigenschaft des *ColorDialog1*-Objekts. Im Beispiel wird die Farbe in einer Variablen *farbe_sel* gespeichert und in einem Farbfeld angezeigt. Als Farbfeld wurde ein *PictureBox*-Element verwendet, dessen *BackColor*-Eigenschaft auf die vom Benutzer gewählte Farbe gesetzt wird. Die nachfolgend gezeigte *Click*-Ereignisbehandlungsroutine enthält die betreffenden Anweisungen zum Abfragen der Farbe und zur Anzeige im Farbfeld:

```
Private Sub Button3_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button3.Click
    ' Schaltfläche zur Farbauswahl gewählt
    ' -> ColorDialog zur Farbauswahl zeigen
    If ColorDialog1.ShowDialog() = _
```

Listing 16.9: Prozedur zur Farbauswahl und -anzeige

```

        Windows.Forms.DialogResult.OK Then
            farbe_sel = ColorDialog1.Color      ' Farbe merken
            Me.PictureBox1.BackColor = farbe_sel ' und anzeigen
        End If
    End Sub

```

Listing 16.9: Prozedur zur Farbauswahl und -anzeige (Forts.)

Zeichnen von Flächen

Die Option *Gefüllte Flächen* des Beispiels zeichnet einige gefüllte Rechtecke und Kreise. Hierzu wird ein *Brush*-Objekt mit folgenden Anweisungen erzeugt:

```

Dim mybrush As Brush ' unifarbener Pinsel
mybrush = New SolidBrush(Color.AliceBlue)

```

Sobald das *Brush*-Objekt definiert wurde, lässt sich beispielsweise die *FillRectangle()*-Methode mit folgender Anweisung aufrufen:

```
g.FillRectangle(mybrush, 10, 10, 200, 60)
```

Neben dem *Brush*-Objekt erwartet die Methode in dieser Aufrufvariante die Koordinaten (X,Y) zweier diagonalen Punkte, die ein Rechteck definieren. Um eine gefüllte Ellipse oder einen Kreis zu zeichnen, wird die *FillEllipse()*-Methode benutzt:

```
g.FillEllipse(mybrush, 50, 75, 120, 70)
```

Auch hier werden neben dem *Brush*-Objekt zwei Koordinatenpaare erwartet, die das umschließende Rechteck des Kreises oder der Ellipse beschreiben. Neben einfarbigen Pinseln lassen sich dem *Brush*-Objekt auch Muster zuweisen.

```

mybrush = New Drawing2D.HatchBrush( _
    Drawing2D.HatchStyle.DiagonalCross, Color.Red)

```

Hier wird über die *HatchBrush()*-Klasse des Namespaces *System.Drawing.Drawing2D* ein Muster mit diagonalen Kreuzen in der Farbe Rot zugewiesen. Das *Brush*-Objekt lässt sich dann in *FillEllipse()* oder *FillRectangle()* einsetzen. Über die *FillPie()*-Methode können auch Kreisausschnitte gezeichnet werden.

```
g.FillPie(mybrush, 150, 120, 100, 100, -45, 310)
```

Das Koordinatenpaar für die linke obere Ecke wird im zweiten und dritten Parameter übergeben. Der vierte und fünfte Parameter beschreiben die Breite und die Höhe des umschließenden Rechtecks. Der *startAngle*-Wert im sechsten Parameter beschreibt den zwischen der x-Achse und der ersten Seite des Kreisausschnitts im Uhrzeigersinn gemessenen Winkel in Grad. Der zwischen dem *startAngle*-Parameter und der zweiten Seite des Kreisausschnitts im Uhrzeigersinn gemessene Winkel in Grad wird im letzten Parameter erwartet.

Zeichnen von Bézier-Splines

Bézier-Splines sind Kurven, die durch vier Koordinatenpaare beschrieben werden. Um eine solche Kurve zu zeichnen, lässt sich folgende Codesequenz verwenden:

```
Dim start As Point = New Point(10, 50)
Dim point1 As Point = New Point(40, 20)
Dim point2 As Point = New Point(80, 150)
Dim _end As Point = New Point(100, 10)

mypen = New Pen(Color.Black, 3)
g.DrawBezier(mypen, start, point1, point2, _end)
```

Es werden die vier Koordinatenpunkte und ein *Pen*-Objekt definiert. Dann wird die *DrawBezier()*-Methode aufgerufen.

Zeichnen von Punkten

Über die Option *Punkte* der Registerkarte lässt sich ein Punktmuster im Zeichenbereich der betreffenden Registerkarte ausgeben. Allerdings besitzt die *Graphics*-Klasse keine Zeichenmethode zur Ausgabe von Punkten. Um dennoch einzelne Punkte auf der Zeichenfläche auszugeben, lässt sich die *FillRectangle()*-Methode verwenden. Diese lässt sich folgendermaßen aufrufen:

```
g.FillRectangle(mybrush, x, y, 1, 1)
```

Der zweite und dritte Parameter beim Methodenaufruf gibt die Koordinate für den auszugebenden Punkt an. Um einen Punkt zu zeichnen, sind der vierte und fünfte Parameter auf 1 (Pixel) zu setzen. Durch eine Folge von Aufrufen mit variablen Koordinaten lassen sich beliebige Rechtecke der Größe 1x1 Pixel (also Punkte) im Zeichenbereich ausgeben.

Hinweis

Sie finden die Projektdateien des kompletten Beispiels im Ordner `\Beisp\Kap16\Zeichnen` der Begleit-CD. Details sind dem Code des jeweiligen Moduls *Form1.vb* zu entnehmen. Beschreibungen zu den Methodenaufrufen finden Sie in der Hilfe zum .NET Framework.

16.4.5 Linienstile und Muster beim Zeichnen verwenden

In den bisherigen Beispielen wurde gezeigt, wie sich über Methoden der *Graphics*-Klasse Linien oder gefüllte Flächen im Zeichenbereich ausgeben lassen. Mit den gleichen Methoden lassen sich aber auch mit Mustern gefüllte Flächen oder Linien mit unterschiedlichen Stilen (strichpunktierte Linien, Pfeile etc.) zeichnen. Um beispielsweise eine gestrichelte Linie in der vorgegebenen Farbe und mit einer einstellbaren Dicke zu zeichnen, sind nur wenige Anweisungen erforderlich. Die folgenden Anweisungen vereinbaren ein *Graphics*-Objekt mit einem Zeichenbereich eines *PictureBox1*-Steuerelements sowie ein *Pen*-Objekt mit einer Linienstärke von 5 Pixel und der Farbe Schwarz:

```
Dim g As System.Drawing.Graphics ' für Zeichenbereich
Dim mypen As Pen
g = Me.PictureBox1.CreateGraphics() ' auf PictureBox
mypen = New Pen(Color.Black, 5) ' Pinsel schwarz, 5 Pkt.
```

Dies kennen Sie bereits aus den vorhergehenden Beispielen. Um den Linienstil anzupassen, ist der *DashStyle*-Eigenschaft des *Pen*-Objekts lediglich ein betreffender Wert vom Typ *Drawing2D.DashStyle* zuzuweisen. Sobald Sie den Typ angeben, zeigt die Entwicklungsumgebung bereits die definierten Konstanten für die Linienstile an. Die folgende Anweisung definiert einen Linienstil *DashDot* (strichpunktierte Linie). Die zweite Anweisung verwendet die *DrawLine()*-Methode, um die Linie zu zeichnen.

```
mypen.DashStyle = Drawing2D.DashStyle.DashDot ' Linienstil
g.DrawLine(mypen, 10, 10, 200, 10) ' Zeichnen
```

Möchten Sie die Enden von Linien mit besonderen Abschlüssen versehen, müssen Sie den Eigenschaften *StartCap* und *EndCap* nur die betreffenden Werte zuweisen. Die beiden folgenden Anweisungen zeigen, wie dies funktioniert:

```
mypen.StartCap = Drawing2D.LineCap.ArrowAnchor ' Linienende Start
mypen.EndCap = Drawing2D.LineCap.DiamondAnchor ' Linienende Ende
```

Sobald Sie den Begriff *Drawing2D.LineCap* im Codefenster eintippen, werden die in der Klasse vereinbarten Konstanten für die Eigenschaften eingeblendet. *ArrowAnchor* zeichnet beispielsweise einen Pfeil als Linienabschluss. Als Fazit bleibt also festzuhalten, dass Sie zum Variieren der Linienstile lediglich die Eigenschaften *DashStyle*, *StartCap* und *EndCap* anpassen müssen.

Ähnliches gilt für das Zeichnen von Mustern bei gefüllten Flächen. In den bisherigen Beispielen wurde beim Aufruf der entsprechenden Methoden (z.B. *FillRectangle()*) ein unifarbene *Brush*-Objekt (*SolidBrush*) übergeben. Erzeugen Sie stattdessen ein *Brush*-Objekt über die Klassen des Namespace *System.Drawing.Drawing2D*. Über die Klassen lassen sich vordefinierte Muster (Kreuze, Ziegel etc.) anzeigen. Dies wurde bereits im vorhergehenden Beispiel genutzt (siehe Abschnitt »Zeichnen von Flächen«). Die folgenden Anweisungen fassen dies nochmals zusammen. Es wird ein *Graphics*-Objekt erzeugt, welches den Zeichenbereich eines *PictureBox*-Steuerelements bereitstellt. Dann wird ein *Brush*-Objekt mit einem roten Kreuzmuster instantiiert. Ein Aufruf der *FillRectangle()*-Methode zeichnet dann ein Rechteck mit dem betreffenden Muster.

```
Dim g As System.Drawing.Graphics ' für Zeichenbereich
Dim mybrush As Brush ' Pinsel
g = Me.PictureBox1.CreateGraphics() ' auf PictureBox
' jetzt Musterpinsel erzeugen (rotes Kreuzmuster)
mybrush = New Drawing2D.HatchBrush(Drawing2D.HatchStyle.DiagonalCross, Color.Red)
g.FillRectangle(mybrush, 10, 10, 200, 40)
```

Abbildung 16.23 zeigt zwei Formulare, in denen Muster sowie verschiedene Linienstile unter Verwendung der obigen Methoden gezeichnet wurden.

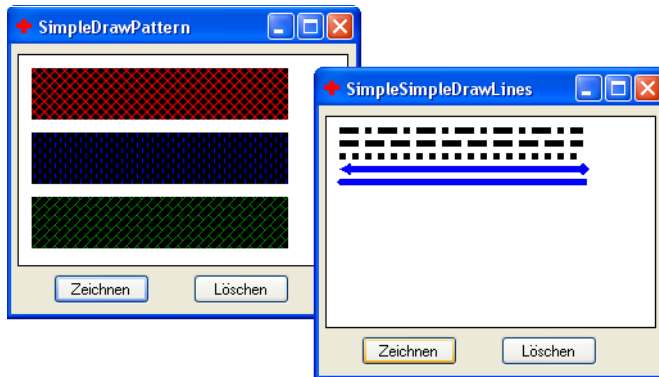


Abbildung 16.23: Zeichnen von Mustern und Linienvarianten

Hinweis

Sie finden die Projektdateien des kompletten Beispiels zum Zeichnen von Mustern im Ordner `\Beisp\Kap16\SimpleDrawPattern` der Begleit-CD. Der Ordner `\Beisp\Kap16\SimpleDrawLines` enthält ein Beispiel, welches gestrichelte Linien und Linien mit Pfeilen an den Enden anzeigt. Beide Beispiele wurden vom *SimpleDraw*-Projekt abgeleitet. Details sind dem Code in der Prozedur *Zeichnen* des jeweiligen Moduls *Form1.vb* zu entnehmen. Beschreibungen zu den Methodenaufrufen finden Sie in der Hilfe zum .NET Framework.

16.4.6 Zeichenfunktionen für Bitmaps und Metagrafiken

Die *Graphics*-Klasse bietet weitere Methoden, wie *DrawImage()*, mit denen sich Bitmap- oder Metagrafiken in einen Zeichenbereich ausgeben lassen.

Abbildung 16.24 zeigt ein Beispielformular, in dem über die *Zeichnen*-Schaltfläche ein Farbverlauf, ein über eine Grafikdatei erzeugtes Muster sowie ein Bild im *.jpeg*-Format angezeigt werden. Das Bild wird dabei zusätzlich als gedrehte Kopie und über einen *Clip*-Bereich beschnitten angezeigt. Über die folgenden Anweisungen wird ein Grafikobjekt erzeugt:

```
Dim g As System.Drawing.Graphics ' für Zeichenbereich
g = Me.PictureBox1.CreateGraphics() ' auf PictureBox
```

Um eine Fläche mit einem Farbverlauf zu zeichnen, kommt die *FillRectangle()*-Methode zum Einsatz. Statt aber ein *Brush*-Objekt mit einem einfarbigen Pinsel zu übergeben, wird ein Farbverlauf definiert. Ein *Brush*-Objekt mit einem Farbverlauf lässt sich über die *LinearGradientBrush()*-Methode der *Drawing2D*-Klasse definieren. Die folgenden Anweisungen zeigen die notwendigen Schritte:

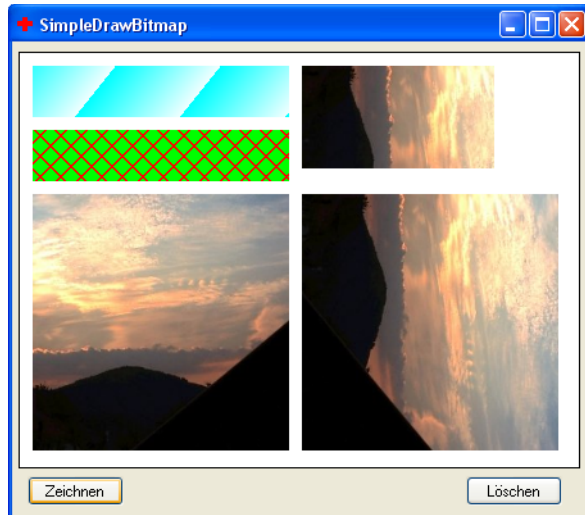


Abbildung 16.24: Demonstration der Bitmap- Zeichenfunktionen

```
Dim mybrush As Brush          ' Pinsel
Dim p1 As New Point(0, 0)     ' Startpunkt Farbverlauf
Dim p2 As New Point(50, 40)   ' Endpunkt Farbverlauf
' Definiere Brush mit Farbverlauf über lineare Gradienten
mybrush = New Drawing2D.LinearGradientBrush(p1, p2, _
      Color.Aqua, Color.Azure)
g.FillRectangle(mybrush, 10, 10, 200, 40) ' Rechteck zeichnen
```

Die *LinearGradientBrush()*-Methode erwartet in den beiden ersten Parametern zwei *Point*-Werte, die die Start- und Endkoordinaten des linearen Gradienten für den Farbverlauf definieren. Gleiche X-Werte in den beiden Punkten erzeugen einen vertikalen Farbverlauf, während gleiche Y-Werte einen horizontalen Farbverlauf definieren. Durch unterschiedliche X- und Y-Werte in den Koordinaten lässt sich ein schräger Farbverlauf (z.B. diagonal) realisieren. Entspricht die X- oder Y-Differenz der beiden Punkte der Breite bzw. Höhe des Zeichenbereichs, wird der Farbverlauf komplett dargestellt. Ist die Differenz größer, wird nur ein Ausschnitt des Farbverlaufs gezeigt. Bei einem Differenzwert, der kleiner als der Zeichenbereich ist, wiederholt sich der Verlauf und wird als Muster dargestellt. Die Ausgabe des Farbverlaufs erfolgt z.B. durch Aufruf der bereits mehrfach benutzten Methoden zum Zeichnen gefüllter Flächen (z.B. *FillRectangle()*-Methode).

In einigen der vorhergehenden Beispiele wurde bereits gezeigt, dass Pinsel auch Muster aufweisen können. Neben Farbverläufen und den in der *Drawing2D*-Klasse definierten Mustern können *Brush*-Objekte auch beliebige Texturen als Muster aufweisen. Dann wird diese Textur beim Zeichnen mit dem *Brush*-Objekt verwendet. Eine solche Textur lässt sich als Bitmap-Grafik über die Klasse *TextureBrush* als *Brush*-Objekt erzeugen. Die folgende Anweisung benutzt eine in einem *ImageList*-Steuerelement hinterlegte Grafik, um ein mit einer Textur gefülltes *Brush*-Objekt zu erzeugen:

```
mybrush = New TextureBrush(ImageList1.Images(0))
```

Soll die Textur direkt aus einer Bitmap-Datei gelesen werden, ist folgende Variante beim Aufruf der *TextureBrush*-Methode zu verwenden:

```
mybrush = New TextureBrush(New Bitmap("GrafikBig.bmp"))
```

Die *Bitmap()*-Methode wertet den als Argument übergebenen Pfad aus, liest die Bitmap-Datei ein und wandelt das Ganze in ein *Bitmap*-Objekt um. Dieses Objekt wird dann als Argument dem *New*-Konstruktor der Klasse *TextureBrush()* übergeben. Der Konstruktor erzeugt das *Brush*-Objekt, welches der Objektvariablen zugewiesen wird. Falls Sie diesen Ansatz verwenden, ist sicherzustellen, dass die Bitmap-Datei im angegebenen Pfad vorhanden ist. Andernfalls tritt ein Laufzeitfehler auf. Im obigen Beispiel muss die Bitmap-Datei also im Ordner mit den Programmdateien hinterlegt sein.

Hinweis

Aus Gründen der Übersichtlichkeit wurde bei den Beispielprogrammen meist auf die Absicherung von Laufzeitfehlern verzichtet. In realen Anwendungen sollten Sie aber eine Laufzeitfehlerbehandlung über einen *Try ... Catch .. End Try*-Block einfügen, um z.B. fehlende Dateien abzufangen.

Sobald das *Brush*-Objekt vorliegt, lässt sich mit den *Fill*-Methoden der *Graphics*-Klasse zeichnen. Die folgende Anweisung zeichnet ein Rechteck mit dem mit der Textur gefüllten *Brush*-Objekt:

```
g.FillRectangle(mybrush, 10, 60, 200, 40)
mybrush.Dispose() ' freigeben
```

Da *Brush*-Objekte bei größeren Bitmaps sehr viel Speicherplatz belegen, sollte diese nach Verwendung möglichst früh wieder freigegeben werden. Dies erfolgt hier in der zweiten Anweisungszeile durch Aufruf der *Dispose()*-Methode des *Brush*-Objekts.

Soll eine Bitmap oder eine Grafik im Metafileformat im Zeichenbereich angezeigt werden, lässt sich die *DrawImage()*-Methode des *Graphics*-Objekts verwenden. Die folgende Anweisung definiert ein *Bitmap*-Objekt und weist diesem eine *.jpg*-Grafikdatei zu. Die *Bitmap* kapselt die Grafikdaten und stellt verschiedene Methoden und Eigenschaften bereit. So lässt sich die *Bitmap* über *DrawImage()* zeichnen. Im ersten Parameter erwartet die Methode das *Bitmap*-Objekt, während die folgenden Parameter die Koordinatenpaare für das umgebende Rechteck spezifizieren:

```
Dim bm As Bitmap = New Bitmap("Abend.jpg")
g.DrawImage(bm, 10, 110, 200, 200)
```

Sobald Sie über ein *Bitmap*-Objekt verfügen, können Sie auf dessen Methoden zugreifen. Die folgenden Anweisungen rotieren ein *Bitmap*-Objekt um 90 Grad (ohne Spiegeln) und zeichnen das Ergebnis erneut im Zeichenbereich:

```
' Bitmap rotieren und anzeigen
bm.RotateFlip(RotateFlipType.Rotate90FlipNone)
g.DrawImage(bm, 220, 110, 200, 200)
```

Neben der *RotateFlip()*-Methode stellt das *Bitmap*-Objekt weitere Methoden zur Manipulation der Bitmap bereit. Mit *Clone()* lässt sich beispielsweise eine Kopie der Bitmap anfertigen und einer anderen Variablen zuweisen. Mit *Save()* lässt sich eine Bitmap in einem Stream speichern. Die Anweisung

```
bm.Save(sFile)
```

sichert die Bitmap in einer Datei, wobei der im Argument übergebene Dateiname über die Extension das Grafikformat (*.bmp*, *.gif*, *.jpg*, *.png*) festlegt. Über die Methoden *SetPixel()* und *GetPixel()* ist schreibender bzw. lesender Zugriff auf einzelne Bildpunkte der Bitmap möglich.

Hinweis

Sie finden die Projektdateien des hier diskutierten Beispiels im Ordner *\Beisp\Kap16\SimpleDrawBitmap* der Begleit-CD. Die angezeigten Grafikdateien befinden sich im Unterordner *\Bin*, in dem auch die Programmdatei von der Entwicklungsumgebung hinterlegt wird. Im Ordner *\Beisp\Kap16\SimpleBitmapRotateConvert* ist ein zweites Projekt hinterlegt, welches das Laden von Bitmapdateien erlaubt. Die geladene Bitmap wird dann der *Image*-Eigenschaft des *PictureBox*-Steuerelements zur Anzeige zugewiesen. Dies hat den Vorteil, dass eine automatische Skalierung der Bildgröße durch das Steuerelement erfolgt. Die Bitmap lässt sich über eine Schaltfläche *Rotieren* schrittweise um 90 Grad nach rechts drehen und über die Schaltfläche *Speichern* in eine Datei speichern. Dabei sind verschiedene Zielformate zulässig. Die Anwendung kann z.B. benutzt werden, um Fotos anzusehen, zu drehen und in andere Bitmap-Formate zu konvertieren. Details zur Implementierung entnehmen Sie dem Quellcode. Weitere Informationen zu den Methodenaufrufen finden Sie in der Hilfe zum .NET Framework, wenn Sie den Methodennamen auf der Indexseite eintippen.

16.4.7 Anfertigen von Screenshots einer Anwendung

Zum Abschluss möchte ich noch ein kleines Beispiel vorstellen, welches das Anfertigen von Bildschirmschnappschüssen (ScreenShots) über das *Graphics*-Objekt erlaubt (Abbildung 16.25). Klickt der Benutzer die Schaltfläche *ScreenShot* an, wird der Inhalt des eigenen Fensters in eine Bitmap kopiert und dann über ein *PictureBox*-Steuerelement im Fenster angezeigt. Über die Schaltfläche *Löschen* lässt sich die Anzeige des Dokumentbereichs leeren, während die Schaltfläche *Laden* einen Dialog zum Laden von Grafikdateien öffnet. Wählt der Benutzer eine Grafikdatei aus, wird diese geladen und über das *PictureBox*-Steuerelement angezeigt. Die Schaltfläche *Speichern* erlaubt den Inhalt des Dokumentbereichs in eine Grafikdatei zu speichern.

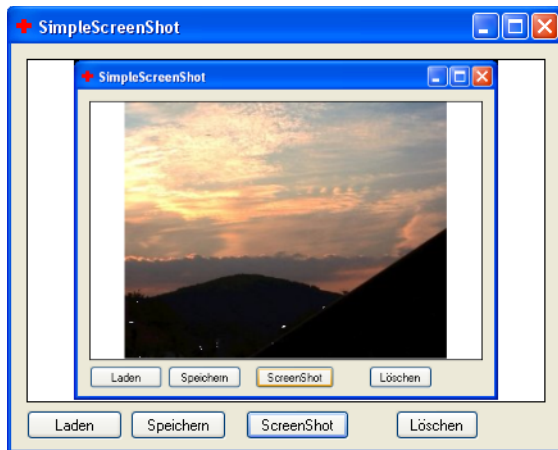


Abbildung 16.25: ScreenShot-Demo

Das Programm wurde aus den vorhergehenden Beispielen abgeleitet. Neu hinzugekommen ist im Wesentlichen die *Click*-Ereignisbehandlungsroutine der Schaltfläche *ScreenShot*. Um einen ScreenShot des aktuellen Fensters einer Anwendung anzufertigen, lässt sich die neu in .NET Framework 2.0 aufgenommene *DrawToBitmap()*-Methode einsetzen. Die Methode ist zwar nur zur Unterstützung der .NET-Framework-Infrastruktur vorgesehen und wird von Steuerelementinstanzen bereitgestellt. Mittels der Methode lässt sich aber ein Screenshot mit einem Dreizeiler anfertigen und in einem *PictureBox*-Steuerelement anzeigen.

```
bm = New Bitmap(w, h)           ' Bitmap-Objekt für Screenshot
Me.DrawToBitmap(bm, Rectangle.FromLTRB(0, 0, w, h))
Me.PictureBox1.Image = bm       ' anzeigen
```

Die erste Zeile erzeugt ein neues Bitmap-Objekt und weist dieses der Variablen *bm* zu. Als Parameter werden die Breite und die Höhe der Bitmap in Pixel erwartet. Hier wurden die beiden Variablen *w* und *h* über *Me.Width* und *Me.Height* auf die Abmessungen des Formulars gesetzt. Anschließend wird die *DrawToBitmap*-Methode des Formularobjekts aufgerufen, die im ersten Parameter das zum Zeichnen benötigte Bitmap-Objekt erwartet. Der zweite Parameter muss eine *Rectangle*-Struktur enthalten, die den zu kopierenden Bildschirmausschnitt des Controls beschreibt. In der letzten Zeile wird die so erzeugte Bitmap einfach der *Image*-Eigenschaft eines *PictureBox*-Steuerelements zur Anzeige zugewiesen. Das folgende Listing zeigt den kompletten Code der *Click*-Ereignisbehandlungsroutine der Schaltfläche *ScreenShot*:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    ' Screenshot des Formularfensters anfertigen
    Dim w As Integer = Me.Width      ' Breite des Formulars
```

Listing 16.10: Code zum Anfertigen eines ScreenShots mit Anzeige

```
Dim h As Integer = Me.Height      ' Höhe des Formulars

If bm IsNot Nothing Then          ' ist Bitmap leer ?
    bm = Nothing                  ' Objekt freigeben
End If

bm = New Bitmap(w, h)             ' Bitmap-Objekt für Screenshot

' Screenshot vornehmen und in Bitmap schreiben
Me.DrawToBitmap(bm, Rectangle.FromLTRB(0, 0, w, h))
Me.PictureBox1.Image = bm         ' in PictureBox anzeigen
Me.Button2.Enabled = True         ' Schaltfläche Löschen freigeben
End Sub
```

Listing 16.10: Code zum Anfertigen eines Screenshots mit Anzeige (Forts.)

Hinweis

Sie finden die Projektdateien des Beispiels im Ordner `\Beisp\Kap16\SimpleScreenShot` der Begleit-CD. Im Unterordner `\Bin` finden sich einige Grafikdateien, die sich über die Schaltfläche *Laden* in die Anzeige holen lassen. Details zur Implementierung entnehmen Sie dem Quellcode.

Auf den vorhergehenden Seiten haben Sie eine Fülle an Informationen zum Programmieren von .NET-Anwendungen unter Verwendung von Visual Basic kennen gelernt. Mit diesem Wissen lassen sich bereits leistungsfähige .NET-Anwendungen erstellen. Trotzdem konnte nur ein Bruchteil des .NET-Framework-Funktionsumfangs behandelt werden. Insbesondere Informationen zu weiterführenden Themen wie Threading, Zugriffsberechtigungen, Sicherheit etc. führen über das Ziel dieses Buches hinaus. Schlagen Sie ggf. in der Hilfe zum .NET Framework (Stichwort »Fortgeschrittene .NET-Framework-Entwicklung« oder »Sicherheit in .NET Framework«) oder zur Entwicklungsumgebung nach, um Informationen zu speziellen Punkten zu erhalten.

Teil 5

Anhang

A Die Begleit-CD

Die in diesem Buch besprochenen Beispiele finden Sie auf der beiliegenden Begleit-CD im Ordner `\Beisp`. Für jedes Kapitel wurde ein separater Unterordner (z.B. `\Kap02`) benutzt, der dann die Projektordner mit den jeweiligen Projekten aufweist. Der Ordner `\KapDiv` enthält noch einige im Buch nicht besprochene Beispiele, die bestimmte Techniken nutzen.

Kopieren Sie den kompletten Ordner `\Beisp` in ein Verzeichnis der Festplatte, um die Beispiele in der Entwicklungsumgebung (Visual Studio 2005 oder Visual Basic 2005 Express Edition) laden und übersetzen zu können. Tipp: Sofern Sie mit Windows-Versionen arbeiten, die vor Microsoft Windows XP erschienen sind, denken Sie daran, das Schreibschutzattribut für den Ordner `\Beisp` und die darin enthaltenen Unterordner bzw. Dateien aufzuheben.

Die Datenbankbeispiele erfordern teilweise, dass der SQL-Server 2005 Express Edition installiert ist und dann DSN-Einträge auf Datenquellen unter Windows eingerichtet wurden. Näheres erfahren Sie in den betreffenden Buchabschnitten. Informationen zu Visual Studio 2005, .NET 2.0, Downloads (z.B. Visual Basic 2005 Express Edition) finden Sie auf den Microsoft-Internetseiten unter www.microsoft.com/germany/msdn/default.mspx.

Die Begleit-CD zum Buch wurde von Verlag und Autor auf Basis »AS-IS« zusammengestellt und liegt dem Buch kostenlos bei. Trotz größter Sorgfalt können Fehler (insbesondere in der beiliegenden Software) nicht ausgeschlossen werden. Verlag und Autor übernehmen weder eine Gewährleistung noch eine Haftung für den Inhalt des Mediums. Die Nutzung erfolgt auf eigene Gefahr. Falls Sie auf offensichtliche Fehler in der Zusammenstellung der CD stoßen, ist der Autor für einen Hinweis über den Verlag dankbar. Eine defekte CD reklamieren Sie bitte beim Verlag.

Stichwortverzeichnis

Symbole

- &= 143
- &H 201
- &-Zeichen 410
- (Application Settings)-Eigenschaft 717
- (Enter)-Taste auswerten 485
- (Property Binding)-Eigenschaft 718
- *= 143
- += 143
- .NET
 - Entwicklungsumgebung 23
 - Menü erstellen 455
 - Überblick 19
- .NET-Anwendung suspendieren 558
- .NET-Datentypen 104
- .NET-Framework 19–20
 - Common Language Runtime 21
 - installieren 30
 - Klassenbibliothek 22
 - Konfigurierung 31
- .NET-Framework Redistributable 21
- .NET-Framework SDK 21
- .NET-Framework-Datenanbieter 653
- .NET-Schnittstellen nutzen 300
- .resx-Datei 39
- .sln-Datei 39
- .vb-Datei 39
- /= 143
- < 151
- <- 151
- <> 151
- <param>-Elemente 95
- <remarks>-Element 95
- <returns>-Element 95
- <summary>-Element 95
- = 143
- = 151
- > 151
- >= 151
- ^= 143
- GetFileSystemEntries-Methode 597

A

- Abkürzungstasten
 - in Formularen 410
- AboutBox
 - Assemblyinformationen festlegen 398
- Abstrakte Basisklassen 285
- AcceptButton
 - Eigenschaft 376
- Access
 - Tabelle lesen 679
- ActiveMdiChild-Eigenschaft 531
- Add()-Methode 303
- Add()-Methode (Controls) 378
- AddHandler 292, 379
- Addition 149
- Add-Methode 417
- AddRange()-Methode 462
- AddressOf 288
- AddressOf-Operator 379
- ADO
 - ReportViewer verwenden 692
- ADO.NET 653
 - Access
 - Tabelle lesen 679
 - Architektur 653
 - DataSet 654
 - einbinden 655
- ADOREadDB 684
- ADOREadDB1 684
- ADOREadNavi 691
- ADOREadNaviNeu 691
- ADOREadXMLDataGrid 707
- ADOREadXMLDataGridView 707
- AfterSelect-Ereignis (TreeView) 443
- Aktuelles Laufwerk 612
- Aktuelles Verzeichnis
 - abfragen 611
 - wechseln 612
- AllowDrop 353
- alte Steuerelemente
 - in Toolbox einbinden 456

- an das Dateiende positionieren 634
- Anchor 353
- Anchor-Eigenschaft 367, 439
- And 150
- AndAlso 150
- Ansicht-Designer 350
- Anweisung
 - Throw New Exception 634
- Anweisungen 91
 - mehrere pro Zeile 92
 - über mehrere Zeilen 92
- Anwendung
 - aktivieren 559
 - bereitstellen 77
 - erstellen 63
 - extern aufrufen 550
 - Fenster in den Vordergrund holen 559
 - MyApplication-
 - Ereignisbehandlungsroutine einfügen 549
 - Parameterübergabe 545
 - Prozess Exit Code 547
 - schließen überwachen 520
 - suspendieren 558
- Anwendungseinstellungen
 - anpassen 720
 - per Code lesen 720
 - setzen 719
 - verwalten 716
- Anwendingsergebnisse 548
 - freigeben 549
- Anwendungsframework 46
- Anwendungsmodale Dialoge 341
- Anwendungspfad ermitteln 398, 547
- Anwendungsprogramm
 - Symbol zuweisen 47
- Anwendungsstart
 - Ereignis 548
- Anwendungstyp
 - Konsole oder Windows-Anwendung 46
- Anzeigemodus
 - PictureBox-Steuerelement 429
- API-Aufruf 591
 - anwenden 589
- API-Funktion
 - aufrufen 588
 - deklarieren 589
- AppActivate
 - Funktion 559
- AppendText
 - Methode 586
- AppendText()-Methode (RichTextBox) 643
- Application.ExecutablePath-Eigenschaft 547
- Application.Exit 379
- Application.Run 370
- ApplicationEvents 549
- ApplicationEvents.vb 550
- Arbeitsspeicher ermitteln 572
- Argumente
 - benannte 219
- Array 121
 - deklarieren 121
 - Größe anpassen 132
 - Methode
 - BinarySearch() 185
 - Clear 182
 - Clone() 181
 - Copy() 181
 - CopyTo() 181
 - Sort 189
 - Methode IndexOf() 188
- ArrayList-Klasse 303
- Assembly 23
- Assemblyinformationen 47
 - festlegen 398
- Assemblymanifest 23
- Assemblyname 46
- Attribute 609
 - Dateien 603
 - decodieren 605
- Attributkonstanten 605
- Attributwert
 - setzen (Datei) 608
- Auf Beenden warten 557
- auf Tasteneingabe warten 124
- Auflistung
 - CheckedItems 418
 - Items (ListBox) 417
- Aufrufparameter
 - ermitteln 545
- Ausgabe
 - Formatschablone 212
- Ausnahme 396
 - werfen 634
- Ausnahmen-Assistenten 224
- Auto 589
- AutoIncrement-Eigenschaft 669
- AutoSize 353

B

BackColor 353
BackColor-Eigenschaft 383, 748
Basisklasse
 abstrakte 285
Basisklassen 234
Bedingte Anweisungen 192
Befehle
 an Menü anbinden 459
Begleit-CD 761
Begrüßungsbildschirm 400
Begrüßungsmeldung 126
Beispiel
 DateiDialoge 517
 RTF-App 520
 SimpleDraw 746
 Wochentage ausgeben 122
Beispiel7_10 305
Beispiel7_10a 307
Beispiel7_10b 310
benannte Konstante 97
Benutzer
 Interaktion 319
benutzerdefinierte Datentypen 135
Benutzer-DSN 663
Benutzerführung
 mit MsgBox() 338
Benutzername
 abfragen 571
Bereich
 Seitenkopf einblenden 694
Bereitstellen
 von Anwendungen 77
Bericht
 entwerfen 692
 Seitenbereich 694
Berichte
 erstellen 692
Betriebssystemversion
 abfragen 571
Bezeichner
 Regeln 103
 Sonderzeichen 104
Bezeichnungsfeld 367
 Text ändern 367
Bézier-Splines zeichnen 750
Bild
 anzeigen 428
Bildanzeige 428

Bildbetrachter 433
Bildlaufleisten-Steuerelement 427
Binärdaten
 lesen/schreiben 635
BinaryReader-Objekt 635
BinarySearch() 186
BinaryWriter-Objekt 635
Binding-Manager 690
BindingNavigator-Steuerelement 673
Bitmap()-Methode 754
Bitmap-Funktionen 752
Bitmap-Objekt 754
BrowseForFolder 521
Brush-Objekt 751
Button 366
ByRef 216
ByVal 216

C

c (Datentyp Zeichen) 166
Call
 by Reference 217
 by Value 217
CancelButton
 Eigenschaft 376
Cancel-Eigenschaft 520
CancelEventArgs-Objekt 520
CanRead()-Methode (FileStream) 633
CanUndo-Eigenschaft 519
CanWrite-Methode (FileStream) 633
Carriage-Return 323
Case Else 195
Catch 229
CBool 146
CByte 146
CChar 146
CDate 146
CDBl 146
CDec 146
CD-Tray
 ein-/ausfahren 591
CenterParent 360
CenterScreen 360
Char
 Wert zuweisen 166
Chars 158
Checkbox 410
CheckedChanged-Ereignis 414
Checked-Eigenschaft 464

- CheckedItems-Auflistung 418
 - CheckedListBox
 - Benutzerauswahl lesen 417
 - mit Werten füllen 417
 - Chr() 332
 - ChrW() 332
 - CIL-Code 21
 - CInt 146
 - Class-Anweisung 96
 - Clear()-Methode 744
 - Click 356
 - Click-Ereignis 378
 - ClickOnce-Technologie 77
 - ClientSize-Eigenschaft 742
 - Clipboard
 - unterstützen 538
 - Clip-Eigenschaft 746
 - Clipping
 - anwenden 746
 - CLng 146
 - Clone() 158
 - Clone()-Methode 755
 - Close()-Methode 368, 386
 - CloseMainWindow()-Methode 557
 - Close-Methode (Datei) 620
 - Close-Methode (Formular) 531
 - Close-Methode (StreamReader) 634
 - CLR-Debugger 74
 - CLS 22
 - CObj 146
 - Codefenster 55
 - Collection-Klasse
 - nutzen 307
 - ColorDialog-Klasse 518
 - Color-Struktur 383
 - ComboBox
 - Benutzerauswahl lesen 417
 - mit Werten füllen 417
 - Wert ermitteln 418
 - COM-Komponenten
 - einbinden 561
 - Command-Funktion 546
 - CommandLine-Eigenschaft 546
 - Command-Objekt (ADO.NET) 653
 - Common Language Runtime (CLR) 21
 - Common Language Specification 22
 - Common Type System 22, 105
 - Compare() 158
 - CompareInfo 165
 - CompareInfo-Klasse 164
 - CompareOrdinal() 158
 - CompareTo() 158
 - COM-Wrapper 562
 - verwenden 563
 - Concat() 158
 - Connection-Objekt (ADO.NET) 653
 - Console
 - Ein-/Ausgaben 319
 - Const-Anweisungen
 - wo einsetzen 98
 - ContextMenu-Steuerelement 469
 - ContextMenuStrip-Eigenschaft 470
 - ContextMenuStrip-Steuerelement 469–470
 - Continue 199
 - ControlBox-Eigenschaft 384
 - Controls 378
 - Copy() 158
 - CopyDirectory()-Methode 620
 - Copy-Methode 538
 - Copy-Methode (Datei) 620
 - CopyTo() 158
 - CreateCommand-Methode 681
 - CreateGraphics()-Methode 737, 742
 - Create-Methode (Datei) 620
 - CreateObject-Funktion 561, 563
 - CreateSubKey-Methode 579
 - CreationTime-Eigenschaft (Ordner) 599
 - CShort 146
 - CSng 146
 - CStr 146
 - CTS 22, 105
 - primitive Datentypen 105
 - CType 146, 396, 575
 - CType-Funktion 166
 - CultureInfo-Klasse 165
 - CurrentDirectory
 - abfragen 571
 - Cut-Methode 538
- D**
- DashStyle-Eigenschaft 751
 - DataAdapter-Objekt (ADO.NET) 654
 - DataBindings-Eigenschaft (Steuerelement) 689
 - DataGridView-Steuerelement 667, 672, 684
 - DataReader-Objekt (ADO.NET) 654
 - DataSet 654
 - an Steuerelemente binden 688

- in Datensätzen navigieren 688
- DataSet-Designer 654, 669
- DataSet-Element 668
 - im Projekt einfügen 666
- DataSource-Eigenschaft 687
- DataSource-UI-Typ-Editor 678
- DataTable-Objekt 654
- Datei
 - anlegen 620, 624
 - Attribute
 - lesen/ändern 603
 - BinaryReadWrite.vb 639
 - Create-Modus 624
 - Datumswerte
 - lesen/setzen 603
 - DbgCLR.exe 75
 - dotnetfx.exe 31
 - existiert 619
 - existiert sie 603
 - Filehandling.vb 619
 - FileHandlingNET.vb 623
 - Form1.vb 520
 - IconMenu.vb 468
 - IconMenu1.vb 468
 - in .exe-Verzeichnis kopieren 397
 - KontextMenu.vb 473
 - lesen 514
 - lesen und schreiben 632
 - ListView.vb 449
 - löschen 616, 619
 - Menu.sln 460
 - Registry1.vb 582
 - RunPrpcess.exe 560
 - schreiben 515
 - ShowDirSettings.vb 615
 - TxtFileAppend.vb 631
 - TxtFileCreate.vb 627
 - TxtFileRead.vb 630
 - TxtFileReadWrite.vb 635
 - umbenennen 616
 - Wetter.vb 315–316
 - WinAPI.vb 591
 - Zugriffskonstanten 624
 - zum Lesen geöffnet 633
- Dateiattribute
 - setzen 603
- Dateieigenschaften
 - anzeigen 606
- Dateien
 - auflisten 597
 - Eigenschaften 599
- Dateiende
 - positionieren 634
- Dateifilter
 - setzen 512
- Dateifunktionen
 - in Visual Basic 616
- Dateigröße 599
- Dateihandhabung
 - über File-Klasse 619
- Dateiinhalt
 - anzeigen (Dumpen) 639
- Dateilänge 634
- Dateimodi 625
- Dateiname
 - aus Pfad abspalten 509
- Dateisignatur (Textdateien) 721
- Dateisystem
 - ermitteln 594
- Dateizugriffe
 - allgemeine Hinweise 624
- Datenanbieter 663
- Datenbank
 - aktualisieren 687
 - im Projekt anlegen 658
 - in Datensätzen navigieren 688
 - Tabelle aktualisieren 684
 - Tabelle in Formular darstellen 666
 - Tabellen hinzufügen 659
- Datenbankdatei
 - in Projekt einbinden 665
- Datenbank-Explorer 34
- Datenquellen
 - Verbindung herstellen 662
- Datensatz
 - entfernen 661
- Datensätze
 - Navigation 690
- Datenstruktur vereinbaren 135
- Datenträger
 - leer 600
- Datentyp
 - Boolean 106–107
 - Byte 106–107
 - Char 106–107
 - Decimal 106, 108
 - Double 106–107
 - für Konstante 101

- Integer 106–107
- Long 106–107
- MsgBoxResult 339
- Object 105–106, 108
 - Nachteile beim impliziten Einsatz 112
- SByte 107
- Short 106–107
- Single 106–107
- String 106, 108
- UInt16 107
- UInt32 107
- UInt64 107
- Variant 105
- Datentypen 104, 108
 - Beispiele 108
 - des CTS 105
 - primitive 105
 - umwandeln 146
 - Visual Basic 2005 106
- DateTimePicker-Steuerelement 433
- Datum
 - Eingabefeld 434
 - Jahr ermitteln 127
 - Monat ermitteln 127
 - Tag ermitteln 127
 - Wochentag ermitteln 127
- Datumswert
 - setzen (Datei) 608
- Day 127
- DayOfWeek 127
- DbgCLR.exe 74
- Debuggen
 - Ausführen bis Cursor 66
 - in Visual Studio 2005 64
- Debugger
 - Aufrufliste anzeigen 73
 - aus Laufzeitumgebung aufrufen 74
 - Einzelschritt 65
 - Funktionen zur Ablaufsteuerung 64
 - Haltepunkt
 - Bedingungen 67
 - Haltepunkt setzen 66
 - nächste Anweisung festlegen 66
 - starten 64
 - Stop-Anweisung 67
 - Variable
 - Wert setzen/überwachen 71
 - Werte anzeigen 70
 - Zeilennummer ablesen 69
- DefaultValue 718
- Deklaration
 - einer Konstanten 97
- Delegate 288
- Delegaten 379
- DeleteKey()-Methode 581
- Delete-Methode (Ordner) 619
- DeleteSubKey()-Methode 582
- Desktop
 - Fenster
 - anordnen 564
- Dezimalformat 325
- Dialog
 - Speichern unter 515
- Dialoge
 - erstellen 364
- Dialogfeld
 - Ausführen öffnen 563
 - Farben 518
 - Hilfeschaltfläche 387
 - Kontextabhängige Hilfe 387
 - ListViewGroup-Auflistungs-Editor 446
 - modal/nichtmodal 370
 - Ressource auswählen 363
 - Schriftart 517
 - Voreinstellungen 539
 - Seite
 - einrichten 728
- DialogResult
 - Eigenschaft 385
- Dictionary 575
- DictionaryEntry-Struktur 306
- Dim 112
- DirectCast 146
- Directory.CreateDirectory 620
- Directory.Delete 619
- DirectoryInfo()-Methode 599
- Dir-Funktion 600
- Dispose()-Methode 740
- Do ... Loop Until-Schleife 198
- Do Until ... Loop-Schleife 197
- Do While ... Loop-Schleife 196
- Dock 353
- Dock-Eigenschaft 435
- Document Object Model (DOM) 708
- Document Type Definition 701
- Dokument
 - beim Schließen sichern 520

- externes
 - starten 556
 - schließen verhindern 531
 - speichern 515
 - Do-Loop-Schleife 196
 - DOM 708
 - DomainUpDown-Steuerelement 419
 - DOS-Anwendung
 - aufrufen 552
 - Dot Net 19
 - DoubleClick 356
 - DrawBezier()-Methode 750
 - DrawCurve()-Methode 744
 - DrawEllipse()-Methode 744
 - DrawImage()-Methode 752, 754
 - DrawLine()-Methode 739
 - DrawRectangle()-Methode 739
 - DrawText()-Methode 738–739
 - DrawToBitmap()-Methode 756
 - Drehfeld 419
 - DriveInfo. 596
 - DriveType-Eigenschaft 594
 - Druckausgabe
 - anstoßen 730
 - Druckdialog
 - anzeigen 724
 - Drucken 724
 - einfaches Beispiel 725
 - Grundlagen 724
 - mehrseitig 733
 - Seitenansicht anzeigen 725
 - Drücken
 - der (Enter)-Taste auswerten 485
 - Drucken-Dialog 730
 - Drucken-Projekt 736
 - DSN
 - einrichten 663
 - DTD 701
 - Dump-Programm 639
 - Dynamic Properties 716
 - dynamische Eigenschaften (Steuerelemente) 716
- E**
- Eigenschaft
 - (Application Settings) 717
 - (Property Binding) 718
 - AcceptButton 376
 - AcceptsReturn (TextBox) 507
 - AcceptsTab (TextBox) 507
 - ActiveMdiChild 531
 - Alignment (Panel) 501
 - AllowDBNull (DataSet) 670
 - AllowDrop (TextBox) 507
 - AllowItemReorder 484
 - AllowMagins (PageSetupDialog) 727
 - AllowOrientation (PageSetupDialog) 727
 - AllowPaper (PageSetupDialog) 727
 - AllowPrinter (PageSetupDialog) 727
 - AllowSelection (PrintDialog) 727
 - AllowSomePages (PrintDialog) 727
 - AllowToPrintFile (PrintDialog) 727
 - AlwaysCreate (Setup) 83–84
 - Anchor 439
 - Anchor (ToolBar-Symbolleiste) 487
 - Appearance (ToolBar-Symbolleiste) 487
 - Arguments (StartInfo) 555
 - Author (Setup) 81
 - AutoIncrement (DataSet) 670
 - AutoSize 495
 - AutoSize (Panel) 501
 - BackColor 383
 - BackgroundImage 363
 - BackGroundImageStyle 363
 - Beschreibung abrufen 354
 - BindingSource 674
 - BorderSides 507
 - BorderStyle 507
 - BorderStyle (Panel) 501
 - BorderStyle (Statusbar) 495
 - BorderStyle (ToolBar-Symbolleiste) 487
 - ButtonSize (ToolBar-Symbolleiste) 488
 - Cancel 520
 - CancelButton 376
 - CanGoBack 450
 - CanGoForward 450
 - CanUndo 519
 - Caption (DataSet) 670
 - CharacterCasing 401
 - Checked 411
 - Checked (Menü) 464
 - CheckOnClick 480
 - Clip() 746
 - ColorDepth 430
 - Columns 445
 - CommandLine 546
 - ContextMenu 470
 - ContextMenuStrip 470
 - ControlBox 384

- DataBindings 689
- DataSource 687
- Dateien/Ordner 599
- DefaultPageSettings.Margins 728
- Description (FolderBrowseDialog) 522
- DialogResult 385, 513
- DisplayStyle 495
- DisplayStyle (Symbolleiste) 480
- Dock 437–439
- Dock (ToolBar-Symbolleiste) 487
- Document (PrintDialog) 726
- DriveFormat 594
- DriveType 594
- DropDownMenu 490
- dynamische (Steuerelemente) 716
- Enabled 401, 405
- Enabled (Schaltfläche) 480, 490
- Enabled (ToolBar-Symbolleiste) 488
- EnableRaisingEvents
 - (FileSystemWatcher) 645
- EndCap 751
- ExitCode 558
- ExitCode (Process) 557
- FieldCount (ADO.NET) 682
- FileName (StartInfo) 555
- Filter (FileSystemWatcher) 644
- Filter (Open/SaveAs) 512
- FlatStyle 367, 412
- FlowDirection 453
- Font 427
- ForeColor 383
- Format (DateTimePicker) 434
- FormBorderStyle 365, 383
- FullyQualifiedName 399
- Groups (ListView) 446
- HasExited (Process) 557
- HasMorepages 732, 735
- Header (TreeView-Gruppe) 446
- Height 377
- Icon 359, 384
- Icon (Panel) 501
- ID (Process) 556
- Image 413–414
- Image (Schaltfläche) 480
- ImageAlign 413, 495
- ImageIndex 415, 440
- ImageIndex (ToolBar-Schaltfläche) 490
- ImageIndex (TreeView) 441
- ImageList 440
- ImageList (ToolBar-Symbolleiste) 488
- Images (ImageList) 414
- ImageScaling 480, 495
- implementieren 243
- In Ausgabeverzeichnis kopieren 397
- Index (Menü) 462
- InitialDirectory (Dialog) 513
- IsMDIContainer (Formular) 525
- IsReady 575
- IsReady (Laufwerk) 594
- Items 416, 445
- LargeImageList 445
- Left 377
- Length (Datei) 634
- Length (Felder) 133
- Lines (RichTextBox) 469
- Location 375, 377
- MaximizeBox 364, 384
- MaximumSize 568
- MaxLength (DataSet) 670
- MaxLength (TextBox) 507
- MDIList 526
- MdiParent 529, 536
- MdiWindowListItem (MDI-Fenster) 526
- Menu 464
- Message 397
- MinimizeBox 364, 384
- MinimumSize 568
- Modified 532
- Multiline 424
- MultiSelect (Dialog) 513
- Name 399
- newline 323
- Nodes 441
- NodeType (XML-Reader) 713
- NotifyFilter (FileSystemWatcher) 644
- Now 127
- Number 397
- Opacity 365, 383
- OwnerDraw (Menüs) 467
- Panels (StatusBar) 501
- PasswordChar 401
- Path (FileSystemWatcher) 644
- Position (BindingManager) 690
- PrintName 727
- PrintToFile (PrintDialog) 727
- ProductName (Setup) 81
- RadioButton (Menü) 465
- ReadOnly 249

- RenderMode 477
- RootFolder (FolderBrowseDialog) 522
- ScrollBars 424, 427
- SelectedImageIndex 440
- SelectedImageIndex (TreeView) 441
- SelectedObject 452
- SelectedPath (FolderBrowseDialog) 522
- SelectedRtf 538
- SelectionColor 539
- SelectionFont 539
- SelectionMode 416
- Shortcut (Menü) 458, 462
- ShortcutKeys (Menü) 459
- ShowCheckbox 434
- ShowIcon 364
- ShowInTaskbar 364
- ShowNewFolderButton
 (FolderBrowseDialog) 522
- ShowShortcut (Menü) 462
- ShowToday 434
- ShowTodayCircle 434
- ShowUpDown 434
- ShowWeekNumbers 434
- Size 377
- Size (ToolBar-Symbolleiste) 488
- SizeMode 429
- SmallImageList 445
- Sorted 416, 420
- Sorting (ListView) 448
- Spring 495
- StartCap 751
- StartInfo 554
- StartInfo (Process) 555
- StartPosition 360
- StatusBarPanel 502
- Style (ToolBar-Schaltfläche) 490
- Symbolleiste (ToolBar) 487
- TabIndex 375
- Tabindex 401
- TabPage 423
- TabStopp 401
- Tag (ToolBar-Schaltfläche) 490
- Text 354
- Text (Menü) 462
- Text (Panel) 501
- Text (ToolBar-Schaltfläche) 490
- TextAlign 413
- TextAlign (ToolBar-Symbolleiste) 488
- ToolTip (ToolBar-Schaltfläche) 490
- ToolTipText (Schaltfläche) 480
- ToolTipText (Panel) 501
- Top 377
- TopMost 741
- TransparencyKey 365
- UserName 128
- UseShellExecute (StartInfo) 555
- ValidatingType 408
- Verb (StartInfo) 555
- View (ListView) 444
- Visible (Menü) 462
- Visible (Schaltfläche) 480
- Visible (ToolBar-Schaltfläche) 490
- Visible (ToolBar-Symbolleiste) 488
- VolumeName 594
- Width 377
- Width (Panel) 501
- WindowState 364
- WordWrap 425, 427
- Wrappable (ToolBar-Symbolleiste) 488
- WriteOnly 249
- Zugriffsberechtigungen für Accessoren
 250
- Eigenschaftfenster 35, 55
 - Datum/Uhrzeit öffnen 565
 - öffnen 565
 - Visual Studio 353
- Eigenschaft-MdiChildren 531
- Eingabe
 - ist es eine Zahl? 211
- Eingabeaufforderung
 - Farben 328
 - Fenster geöffnet lassen 124
 - Fensterabmessungen 328
 - Taste auslesen 124
- Einzelinstanzanwendung erstellen 46
- Element 95
- Else-Anweisung 193
- EmptyDrive 603
- Enabled 353
- EndCap-Eigenschaft 751
- EndsWith 158
- Entwicklungsumgebung
 - XML-Editor 702
- Enum
 - Methode
 - GetName 205
 - GetValues 207
- Enumeration 200
 - IsDefined()-Methode 206

- Typ angeben 201
- Werte angeben 201
- Zugriffsmodifizierer 200
- Enumerationsmember 201
- Enum-Klasse 205
- Environment.SpecialFolder-Auflistung 577
- Environment-Klasse 571
- Equals() 159
- Ereignis
 - beim Anwendungsstart 548
 - beim Terminieren 548
 - Closing 520
 - Enter 430
 - Resize 744
- Ereignisanzeige
 - Protokoll löschen 651
 - Protokollkategorie löschen 650
- Ereignisbehandlung 379
- Ereignisbehandlungsprozedur 356
 - Formular laden 401
- Ereignisprotokoll
 - Eintrag schreiben 651
 - Einträge lesen 651
- Ereignisprotokollierung 648
- Ereignisprozedur
 - im Formular einfügen 356
- Ereignisse 290
 - anwendungsbezogen 548
 - auslösen 290
- Err.Description 227
- Err.Erl 227
- Err.Number 227
- Err.Source 227
- Err-Objekt 227
- ERRORLEVEL 548
- Eurozeichen
 - Probleme 326
- Event 291
- EventLog 648
 - Beispiel 652
 - Eintrag schreiben 651
 - Einträge lesen 651
 - Element verwenden 649
 - Quelle registrieren 649
- Exception 224, 396
 - auslösen 228
 - erzeugen 634
- Exklusiv-Oder 150
- ExecutablePath-Eigenschaft 547
- ExecuteReader-Methode (ADO.NET) 682
- Existiert Datei 603
- Exists()-Methode (Datei/Ordner) 619
- Exists()-Methode (Dateien/Ordner) 604
- Exit
 - Do 209
 - For 209
 - Sub 209
- Exit-Anweisung 209
- ExitCode-Eigenschaft 557
- Exit-Methode 459
- Explore (Windows-Shell) 563
- Explorer-Fenster 449
- Exponentialformat 325
- Extended Markup Language 699
- externe Anwendung
 - aufrufen 550
- F**
- Farbauswahl-Dialog 518
 - anzeigen 518
- Farbe
 - bei Konsoleausgaben 328
 - Symbolleiste ändern 477
- Farben
 - setzen (Formular) 383
- Farbtiefe
 - bei ImageList 430
- Fehlerbehandlung 395
- Fehlerobjekt 227
- Feld
 - deklarieren 121
 - GetLength-Methode 133
 - GetUpperBound-Methode 133
 - Größe anpassen 132
 - größten Indexwert berechnen 133
 - kopieren 178
 - Length-Eigenschaft 133
 - neu initialisieren 182
 - redimensionieren 132
 - sortieren 189
 - suchen in 185
- Felder 121
 - initialisieren 125
 - mehrdimensionale 130
- Feldvariable
 - deklarieren 121
- Fenster
 - anordnen (MDI) 530

- kaskadieren (Desktop) 564
- kein Symbol in Taskleiste 364
- mit Menü 455
- wiederherstellen (Desktop) 564
- zum Suchen nach Computern öffnen 565
- zum Suchen nach Dateien öffnen 565
- Fensterstile 383, 551
- Festkommaformat 325
- File.Delete 619
- FileAccess.ReadWrite 633
- FileAttributes-Konstanten 605
- FileMode-Konstante 624
- FileNames-Auflistung (Dialog) 513
- FileOpenDialog 512
- FileStream-Objekt 624
- FileSystemWatcher 644
 - Beispiel 644
- FileSystemWatcher-Steuerelement 648
- Fill()-Methode 669, 672
- FillEllipse()-Methode 749
- Fill-Methode (OleDbDataAdapter) 687
- FillPie()-Methode 749
- FillRectangle()-Methode 749, 751
- Finally 229
- Flächen zeichnen 749
- FlowLayoutPanel-Steuerelement 452
- Flush()-Methode (Datei) 634
- Focus()-Methode 404
- Fokuswechsel verhindert 400
- FolderBrowserDialog 597
- FolderBrowserDialog-Steuerelement 521
- FolderInfo 598
- FolderInfoExt 600
- Font-Dialog
 - anzeigen 517
- For Each Next-Schleife 206
- ForeColor 353
- ForeColor-Eigenschaft 383
- Format() 159
- Format()-Methode 326
- Formatbezeichner 324
 - benutzerdefiniert 324
- Format-Funktion 326
- Formatierung
 - RTF-Text 539
 - Stellenzahl
 - festlegen 326
- Formatschablone 212
- Formatzeichenfolgen 324
- FormBorderStyle-Eigenschaft 383
- FormClosing 356
- FormClosing-Ereignis 520
- FormClosing-Ereignisbehandlungsroutine 405
- Formdesigner 349
- FormStartPosition 360
- Formular
 - Abbrechen-Schaltfläche vereinbaren 376
 - ableiten 372
 - About-Dialog 392
 - Aktivierreihenfolge 399
 - Aktivierungsreihenfolge der Steuerelemente 375
 - als Klasse 379
 - anlegen 349
 - auf Schließen reagieren 405
 - ausblenden 568
 - Beispiel 354
 - Beschriftung
 - einbinden 366
 - Bilddatei
 - einbinden 395
 - dynamisch anpassen 391
 - dynamische Eigenschaften 716
 - Eigenschaft
 - FormBorderStyle 383
 - Size 377
 - Eigenschaften
 - ändern 352
 - Eigenschaften anpassen 358
 - Eigenschaften beim Laden ändern 360
 - Eigenschaften zur Laufzeit ändern 361
 - einblenden 569
 - Eingabe prüfen 399
 - Ereignisprozeduren 356
 - erneut einblenden 371
 - Fokuswechsel verhindert 400
 - für Datenzugriff erstellen 667
 - Größenänderung verhindern 365, 568
 - Hintergrundbild 363
 - in Datensätzen navigieren 688
 - Kennworteingabe 400
 - Komponentenbereich 388
 - Kontexthilfe
 - manuell hinzufügen 389
 - Kontrollkästchen 409
 - löschen (Formulardesigner) 373
 - mit Ereignisverarbeitung 376

- mit Fortschrittsanzeige 419
 - mit Listenfild entwerfen 416
 - mit ListView-Steuerelement 435
 - mit Schaltfläche 376
 - mit Schieberegler 419
 - mit Splitter-Steuerelement 435
 - mit Text und Bild 392
 - neu anlegen (Formulardesigner) 372
 - Optionsfeld 409
 - per Code erzeugen 375
 - positionieren 377
 - Rückgabecodes 366
 - Schaltfläche
 - erzeugen 376
 - Minimieren
 - ausblenden 384
 - Schaltflächen
 - einbinden 366
 - Schaltflächen Minimieren/Maximieren
 - ausblenden 364
 - schließen 368, 386
 - Schließen-Schaltfläche 376
 - Show()-Methode 374
 - später ein-/ausblenden 368
 - Steuerelement
 - einbinden (Formulardesigner) 351
 - Steuerelemente
 - löschen 352
 - markieren 352
 - Stile 365
 - Systemmenü
 - ausblenden 384
 - Symbol 359, 384
 - Tastaturnavigation aktivieren 410
 - Textfeld
 - einbinden 366
 - Titel setzen 354
 - Tooltip
 - für Steuerelemente 387
 - ToolTipp
 - per Code hinzufügen 389
 - transparent 365
 - Transparenzwerte 365
 - Unterformular aufrufen 374
 - Varianten 362
 - zweites in Visual Studio hinzufügen 372
 - Formulardesigner 60, 350
 - Designansicht 351
 - zur Codeansicht wechseln 351
 - Formulare 349
 - Formularklasse 379
 - For-Next-Schleife 198
 - Fortschrittsanzeige 419
 - Fortsetzungszeilen 92
 - Friend 97, 102, 118
 - Frühes Binden 561
 - an COM-Anwendung 561
 - Function 210
 - Funktion
 - Beep 591
 - ChDir 612
 - ChDrive. 613
 - Command 546
 - CurDir 611
 - ExpandEnvironmentVariables 552
 - Get 245
 - GetBmp 431
 - Hex() 202
 - IsEmpty 601
 - IsMissing 221
 - IsNumeric 211
 - Kill 616
 - MkDir 616
 - Now() 127, 130
 - Rename 616
 - Rmdir 616
 - Rückgängig 519
 - SaveDirtyDoc() 510
 - Funktionen 209
 - zur Typkonvertierung 146
- G**
- Generic-Elemente 310
 - geschachtelte Schleifen 198
 - gestrichelte Linien 752
 - Get-Anweisung 245
 - GetAttributes()
 - Methode 606
 - GetAttributes-Methode 605
 - GetCommandLineArgs()-Methode 546
 - GetCurrentDirectory()-Methode 611
 - GetDirectories()-Methode 598–599
 - GetDirectories-Methode 597
 - GetDirectoryName()-Methode 547
 - GetEncoding 628
 - GetEnumerator() 159
 - GetEnvironmentVariable()-Methode 576
 - GetEnvironmentVariables()-Methode 575

- GetExtension
 - Methode (Pfad) 534
- GetFileName-Methode 509
- GetFiles()-Methode 598
- GetFolderPath()-Methode 577
- GetLength-Methode 133
- GetLogicalDrives()-Methode 576, 593
- GetName()-Methode 577
- GetName-Methode 205
- GetParent() 598
- GetParent()-Methode 606
- GetPath 547
- GetPath-Funktion 606
- GetPath-Methode 398
- GetPixel()-Methode 755
- GetSpecialFolder-Auflistung 577
- GetSubKeyNames()-Methode 583
- GetType-Methode 205
- GetUpperBound()-Methode 133
- GetValue()-Methode 581
- GetValueNames()-Methode 584
- GetValues-Methode 207
- Gleitkomma-Division 149
- Gliederung
 - nutzen 95
- Globalization.RegionInfo.CurrentRegion 728
- GoBack 450
- GoForward 450
- GoHome 450
- GotFokus 356
- Goto-Anweisung 195
- Grafikprogrammierung 736
- Graphics.DrawString()-Methode 731
- Graphics-Klasse 737
- Größe
 - eines Feldes anpassen 132
 - Steuerelement ändern 742
- GroupBox-Steuerelement 410
- Grundlagen
 - objektorientierte Programmierung 233
- Gruppe
 - in ListView definieren 446
- H**
- Haltepunkt
 - Bedingungen 67
 - setzen 66
 - Trefferzahl vorgeben 68
- Handles 378
- HasAttributes-Eigenschaft (XMLReader) 713
- HasChanges()-Methode 671
- HasExited-Eigenschaft 557
- HashTable-Klasse 305
- Height
 - Eigenschaft 377
- HelpProvider 388
- Hexadezimal 201
- Hilfe
 - einbinden 722
 - nutzen 85
- Hilfdateien 723
- Hilfe-Projekt 724
- Hilfeschaltfläche
 - implementieren 387
 - in Dialogfeld 387
- HScrollBar 427
- HTMLHelp-Compiler 723
- I**
- IComparable-Schnittstelle 300
- Icon-Eigenschaft 384
- IDictionary 575
- IDictionary-Schnittstelle 305
- If-Anweisung 192
- If-Then-Else-Anweisung 193
- If-Then-ElseIf-Anweisung 194
- ICollection-Schnittstelle 303
- ImageList
 - füllen (per Code) 442
- ImageList-Steuerelement 413, 428
- ImageView-Steuerelement 428
- Imports-Anweisung 110
- IndexOf 164
- IndexOf() 159
- IndexOfAny() 159
- Infobereich
 - der Taskleiste 566
- Inherits 382
- INI-Datei
 - Zugriff 588
- Initialisierungswert 113
- InputBox-Methode 345
- Insert() 159
- Insert()-Methode (DataSet) 670
- Installationsart 79

Installationspaket
 installieren 84
 Installer
 Dateien/Ordner hinzufügen 82
 Integer-Division 149
 Interfaces 236, 293
 Internet Explorer
 aufrufen 556
 IsDefined()-Methode 206
 IsMDIContainer-Eigenschaft 525
 IsMissing 221
 IsNot 154
 IsNumeric 211, 330
 IsReady-Eigenschaft 575
 IsReady-Eigenschaft (Laufwerk) 594
 Item()-Auflistung (ADO.NET) 682
 Items-Auflistung 417

J

Jahr
 aus Datum ermitteln 127
 Join() 159

K

Kalender
 anzeigen 428
 gewähltes Datum ermitteln 569
 Kalenderanzeige 433
 Kapselung 233
 Kennworteingabe 400
 Kennwortfeld 401
 Kennzeichenleiste 66
 KeyDown 356
 KeyValuePair()-Methode 302
 Kill()-Methode 557
 Klasse
 Array 178
 Button 376
 ColorDialog 518
 CompareInfo 164
 Console 319
 CultureInfo 165
 Delegate 288
 Directory 593, 597
 Eigenschaften 233
 Environment 570
 Felder 233
 für Vererbung
 sperrern 283

Graphics 737
 Help 722
 IconMenuItem 467
 in DLL überführen 314
 in separaten Dateien 314
 Margins 724
 Member
 überladen 257
 MenuIcon 467
 Methoden 233
 mit Eigenschaften 243
 mit Klassenvariablen 240
 PageSetUp 725
 PrintDialog 724
 PrintDocument 724
 PrintPreviewDialog 725
 Shell32.ShellClass 564
 Size 377
 StringBuilder 177
 System.Array 182
 System.Environment 323
 TextureBrush() 754
 Thread 558
 über DLL einbinden 314
 Vererbung 265
 View 448
 wie erreiche ich dieses 236
 XmlSerializer 715
 Klassen
 was ist das 233
 Klassenbibliothek
 Klasse identifizieren 236
 Klassen-Designer 61
 Klassendiagramm
 anzeigen 61
 Klassenmember 102
 Klassenmitglieder 57
 Klassenvariablen 240
 Kombinationsfeld
 Eingabe auswerten 485
 Kommandoprozessor 552
 Kommentar 93
 Kommentaranweisung 93
 Konfigurations-Manager 44
 Konsole
 Benutzereingaben 328
 Read 328
 ReadLine 328

- Konsoleausgaben
 - in Farbe 328
- Konsolenfenster
 - Eigenschaften 328
- Konstante 97
 - CompareOptions.IgnoreCase 165
 - Datentyp festlegen 101
 - mehrere pro Zeile deklarieren 101
 - newline 103
 - vbCrLf 103, 323
 - vbDefaultButton1 340
 - vbExclamation 335
 - vbOkOnly 103
 - vbTab 323
 - Zugriffsberechtigungen 101
- Konstanten
 - Gültigkeit 99
- Konstruktor 252
 - Vererbung 268
- Kontextmenü 468
 - an Steuerelement anbinden 470
 - Befehl anbinden 470
 - dynamisch erweitern 471
 - hinzufügen 468
 - im Programm erzeugen 473
- Kontrollkästchen 409
 - markieren 411
- Koordinatensystem
 - Nullpunkt 378
- Kurzeinführung
 - in Visual Studio 2005 33
- L**
 - Label 366, 392
 - langpack.exe 31
 - LastIndexOf() 159
 - LastIndexOfAny() 159
 - LastWriteTime-Eigenschaft (Datei) 599
 - Late Binding 561
 - Laufwerk
 - aktuelles ermitteln 612
 - setzen 612
 - bereit 594
 - Eigenschaften 594
 - Kapazität abfragen 595
 - Laufwerke auflisten 576, 593
 - Laufwerkstyp
 - ermitteln 594
 - Konstanten 594
 - Laufzeitfehler 224
 - per Programm erzeugen 634
 - Laufzeitfehlerbehandlung 395
 - LayoutMdi-Methode 530
 - leere Zeichenkette 169
 - Leerzeile
 - erzeugen (Konsole) 320
 - Left-Eigenschaft 377
 - Length 158, 169
 - Length-Eigenschaft 133, 169
 - Lib 589
 - LinearGradientBrush()-Methode 752
 - Line-Feed 323
 - Linie
 - zeichnen 739
 - Linien
 - mit Pfeilen 752
 - Linienenden
 - anpassen 751
 - Linienstil
 - anpassen 751
 - Linienstile 750
 - Link
 - ausführen 406
 - LinkClicked-Ereignis 407
 - LinkLabel 399
 - ausführen 406
 - ListBox
 - Benutzerauswahl lesen 417
 - mit Werten füllen 417
 - Listing
 - Kommentaranweisung 93
 - ListView
 - Auswahl abfragen 449
 - Darstellungsmodus 448
 - Eigenschaft
 - Checkboxes 445
 - Columns 445
 - Items 445
 - LabelEdit 445
 - LargeImageList 445
 - SmallImageList 445
 - View 445
 - Eintrag hinzufügen 447
 - Einträge hinzufügen 445
 - Gruppendarstellung aus 448
 - Klick auf Spaltenkopf abfangen 448
 - Spalten setzen 445
 - ListViewGroup-Auflistungs-Editor 446

ListView-Steuerelement
 füllen 444
 Load 356
 Load()-Methode 708
 LoadFile-Methode 533, 535
 Location-Eigenschaft 377
 Logische Operatoren 149
 Löschen
 in Papierkorb 620
 von Dateien 616
 LostFocus-Ereignis 485
 LostFokus 356

M

MainMenu-Steuerelement 457
 Main-Prozedur
 mit Parameterübernahme 543
 managed Code 21
 Margins-Eigenschaft 728
 Maschinennamen
 abfragen 571
 MaskedTextBox 408
 MaximizeBox-Eigenschaft 384
 MaximumSize 353
 MDI-Anwendung 540
 Fenster schließen 530
 Fenster verwalten 530
 übergeordnetes Formular erstellen 525
 untergeordnetes Formular erstellen 527
 MDI-Anwendungen 523
 MdiChildren-Eigenschaft 531
 MDI-Client 524
 MDI-Dokument
 Aktivierung erkennen 536
 neues Dokument 528
 MDI-Dokumentfenster
 schließen 531
 MDI-Fenster
 kaskadieren 530
 nebeneinander anordnen 530
 MDIList-Eigenschaft (Menüelement) 526
 MDI-Parent 524
 MdiParent-Eigenschaft 529, 536
 Me 270, 358
 Me.ActiveMdiChild 531
 Mehrdimensionale Felder 130
 Mehrfachvererbung 235
 Member 57, 135, 233
 freigegebene 235
 Menü 455
 Eintrag sperren 462
 erstellen 455
 Funktionen anbinden 459
 manuell erstellen 460
 mit Symbolen 459, 467
 Trennzeichen einfügen 458
 Menübefehl
 mit Punkt 465
 Menübeispiel 460
 Menu-Eigenschaft 464
 Menüeintrag
 mit Häkchen 464
 MessageBoxButtons-Enumeration 343
 MessageBoxIcon-Enumeration 343
 MessageBox-Klasse 342
 Methode
 Add (ListBox) 417
 Append() 177
 AppendText (RichTextBox) 643
 Bitmap() 754
 CacadeWindows (Windows Shell) 564
 CanRead (FileStream) 633
 CanWrite (FileStream) 633
 Clear() 651, 738, 744
 Clone() 755
 Close (Datei) 620
 Close (Formular) 368, 386, 531
 CloseMainWindow (Process) 557
 ControlPanellItem (Windows Shell) 565
 Copy (Datei) 620
 Copy (RichTextBox) 538
 Create (Datei) 620
 CreateEventSource() 650
 CreateGraphics() 737, 742
 Cut (RichTextBox) 538
 Delete (Ordner) 619
 DeleteSubKeyTree 582
 Dispose() 740
 DrawArc() 738
 DrawBezier() 738, 750
 DrawCurve() 744
 DrawEllipse() 737, 744
 DrawImage() 738, 752, 754
 DrawLine() 737, 739
 DrawPolygon() 738
 DrawRectangle() 738–739
 DrawString() 738–739
 DrawText() 738–739

- DrawToBitmap() 756
- EndWith() 164
- ExBeep 591
- Exists (Datei/Ordner) 619
- Exit 459
- FillClosedCurve() 738
- FillEllipse() 738, 749
- FillPath() 738
- FillPie() 749
- FillPolygon() 738
- FillRectangle() 738, 749
- FindComputer (Windows Shell) 565
- FindFiles (Windows Shell) 565
- Flush (Datei) 634
- Format 326
- GetAttributes 605
- GetAttributes (Datei) 606
- GetCommandLineArgs 546
- GetCreationTime() 606
- GetDataTypeName 682
- GetDirectories 597
- GetDirectories() 598
- GetEnvironmentVariable() 576
- GetEnvironmentVariables() 575
- GetExtension 534
- GetFiles 598
- GetFileSystemEntries 597
- GetFolderPath 577
- GetLength (Felder) 133
- GetLogicalDrives 576
- GetLogicalDrives() 593
- GetName 205
- GetName (ADO.NET) 682
- GetParent() 598, 606
- GetPath() 398
- GetPixel() 755
- GetType 111, 205
- GetUpperBound 133
- GetValues 207
- GetValues() 577
- Graphics.DrawString() (Drucken) 731
- implementieren 251
- IndexOfAny() 164, 166
- InputBox 345
- Kill (Process) 557
- LayoutMdi 530
- LinearGradientBrush() 752
- LoadFile 535
- MinimizeAll (Windows Shell) 564
- Move (Ordner) 620
- OpenRemoteBaseKey() 587
- PadLeft() 168
- PadRight() 168
- Past (RichTextBox) 538
- Position (Datei) 634
- Print() 724
- Raise() 228
- Read 332
- Read() (XMLReader) 713
- ReadLine 328
- ReadXML() (DataSet) 706
- Refresh() 746
- RotateFlip() 755
- Save() 755
- SaveFile 536
- SelectAll 538
- SetCreationTime 608
- SetEnvironmentVariable() 578
- SetHelpString 390
- SetLastAccessTime 608
- SetLastWriteTime 608
- SetPixel() 755
- SetShowHelp 390
- SetTime (Windows Shell) 565
- SetToolTip 390
- Show 370, 530
- ShowDialog 370
- ShowDialog() 512
- ShowHelp() 722
- ShowPopup() 723
- Sleep 558
- SourceExists() 650
- Start 407
- StartsWith() 164
- StreamReader 627
- StreamWriter 625
- TileHorizontally (Windows Shell) 564
- TileVertically (Windows Shell) 564
- toLower() 164
- toString 111
- toUpper() 164
- TrayProperty (Windows Shell) 565
- TrimEnd 170
- TrimStart 170
- Undo 519
- UndoMinimizeAll (Windows Shell) 564
- vor Überschreiben schützen 284
- WaitForExit (Process) 559

Write() 110, 319
 WriteEntry() 651
 WriteLine() 110, 320
 Microsoft CLR-Debugger 74
 Microsoft Help-Workshop 723
 Mini-Editor 505
 Befehl
 Neu 510
 Öffnen 512
 Pfad in Titel 509
 Formular entwerfen 506
 Minimieren-Schaltfläche
 ausblenden 364
 MinimizeBox-Eigenschaft 384
 MinimumSize 354
 Mod 149
 Modified-Eigenschaft 532
 Modul-Anweisung 96
 Module 235
 Modulo-Division 149
 Monat aus Datum ermitteln 127
 Month 127
 MonthCalendar-Steuerelement 433
 MouseClick 356
 MouseMove 356
 Move-Methode (Ordner) 620
 MoveToNextAttribute()-Methode
 (XMLReader) 713
 MsgBox
 anwendungsmodal 341
 Stil
 spezielle Konstante 341
 systemmodal 341
 MsgBox() 334
 Rückgabewert auswerten 337
 Rückgabewerte 338
 Schaltflächen wählen 337
 Symbole
 Konstante 335
 MsgBoxResult 339
 Multi Document Interface (MDI)
 Anwendungen erstellen 523
 MultiCast-Delegates 293
 Multiple Document Interface 523
 Multiplikation 149
 Muster 750
 Muster zeichnen 750
 MyApplication-
 Ereignisbehandlungsroutinen 549

MyBase 270
 MyClass 276
 My-Objekt 312, 572
 MyPath-Funktion 547

N

Namensraum 110
 für newline 124
 System 319
 System.Collections 303
 System.Data 655
 System.Delegate 288
 System.Diagnostics 73, 554
 System.Drawing 377
 System.Environment 571
 System.Windows.Forms 375
 Namensraum System.Drawing.Printing 724
 Namensraum System.IO.Path 509
 Namensräume 236
 im Projekt importieren 49
 Namespace
 Namenskonflikte 578
 System.Reflection 399
 System.XML 708
 Namespaces 236
 Klasse identifizieren 236
 Ndoc 96
 Negation 149
 Neu
 Auflistungen mit Generic-Elementen 310
 Continue 199
 Eigenschaften mit verschiedenen
 Zugriffsberechtigungen 250
 Generic-Elemente 310
 IsNot 154
 Konsolenfenster
 Eigenschaften 328
 MaskedTextBox 408
 MenuStrip-Steuerelement 457
 My-Objekte 312
 Nullable-Wertetypen 139
 Operatoren überladen 262
 Partial-Typen 313
 Schlüsselwort Global 202
 SmartTag-Bereich 425
 SplitContainer 436
 TryCast()-Methode 147
 XML-Kommentare 94
 New 156

New-Konstruktor 252
newline 124
newline-Eigenschaft 323
Not 149
Nothing 155
NotifyIcon 566
 Projekt entwerfen 567–568
NotInheritable 283
Now 127
Nullable-Wertetypen 139
Nullpunkt
 Koordinatensystem (Formulare) 378
NumericUpDown-Steuerelement 419

O

Objekt 233
 Eigenschaften
 zugreifen 237
 FileStream 624
 Instanz erzeugen 238
 Methode
 Get 245
 Set 245
 Stream 624
Objektbrowser 58
Objektinstanz
 Bezug durch Me 270
objektorientierte Programmierung 233
ODBC-Datenanbieter 663
ODBC-Verbindung
 einrichten 663
Oder 150
Öffnen-Dialog 512
OleDb.OleDbConnection-Objekt 681
OleDbCommand-Objekt 681
On Error 226
On Error Goto 226
OnError 228
OnError1 228
Opacity
 Eigenschaft 383
Open (Windows-Shell) 563
OpenFileDialog-Steuerelement 512
OpenRemoteBaseKey()-Methode 587
OpenSubKey()-Methode 579
Operator
 Is 152–153
 Like 152
Operatoren 148
 arithmetische 149
 logische 149
 Priorität 154
 Relationale 151
 überladen 262
 Zeichenfolgen 152
Option
 Explicit 119
 Strict 119
Option Compare 48
Option Explicit 48
Option Strict 48
Optional 221
optionale Argumente 221
Optionsfeld 409
 markieren 411
Or 150
Ordner
 anlegen 620
 Dateien
 auflisten 597
 Eigenschaften 599
 existiert 619
 leer 600
 löschen 616, 619
 Unterordner
 auflisten 597
Ordnerhandhabung
 über Directory-Klasse 619
Ordnernamen
 ermitteln 599
OrElse 150
Overridable 277
Overrides 277
OwnerDraw-Eigenschaft 467, 502
OwnerDraw-Option (Menü) 467

P

PadLeft() 159
PadRight() 159, 169
PageSetUp-Klasse 725
Paint 356
Paint-Ereignis 740
 Code debuggen 741
PaintEventArgs 737
PaintEventArgs-Objekt 738
Panels-Auflistung 500
Panels-Eigenschaft (StatusBar) 501

- Panel-Steuerlement 436
- ParamArray 222
- Parameter
 - auswerten 543
- Parameterübergabe
 - ByVal 216
 - Ref 216
- Partial-Typen 313, 355
- Paste-Methode 538
- Path.GetDirectoryName()-Methode 547
- Pen
 - definieren 739
- Pfad
 - Dateiname
 - abspalten 509
 - der Anwendung ermitteln 547
 - ermitteln 398
 - ermitteln (Programm) 547
- Pfeile zeichnen 752
- PictureBox 392
- PictureBox-Steuerlement 429
- Plattform 48
- Position-Methode (Datei) 634
- Potenzieren 149
- Preserve 132
- Primärschlüssel
 - festlegen 659
- Primitive Typen 155
- Print()-Methode 724, 731
- PrintDialog
 - Eigenschaft
 - Document 726
- PrintDialog-Klasse 724
- PrintDocument-Klasse 724
- PrintPage-Ereignis 725, 731, 734
- PrintPreviewDialog
 - Klasse 725
- Private 102, 118
- Process
 - auf Beenden warten 557
 - Dokument aufrufen 556
- Process-Steuerlement 554
- Programm
 - Control.exe 566
 - Dump 639
- Programmpfad
 - ermitteln 398
- ProgressBar-Steuerlement 419
- Projekt 39
 - anlegen/öffnen 40
 - DataSet-Element einfügen 666
 - Datei in Ausgabeverzeichnis kopieren 397
 - Datenbankdatei einbinden 665
 - Einstellungen anpassen 44
 - Komponenten hinzufügen 50
 - RunProcess 560
 - Verweise hinzufügen 59
- Projekt-Designer 45–46
 - Eigenschaftenseiten 46
- Projekteigenschaft
 - Einstellungen 49
 - Ressourcen 49
 - Signierung 49
 - Verweise 49
- Projekteigenschaften 44, 50
 - Anwendung 46, 48
 - Debuggen 49
- Projektmappe
 - Element löschen 352
 - Elemente löschen/umbenennen 51
- Projektmappen-Explorer 55
- Projektmappen-Explorers 34
- Projekttypen 43
- Projektvorlage erzeugen 52
- PropertyGrid 451
- Protected 102, 118, 278
- Protected Friend 102
- Protokolldatei
 - löschen 650
- Provider 663
- Prozedur
 - Argumente
 - benannte 219
 - optionale Argumente 221
 - Parameter klammern 215
 - Set 245
- Prozeduren 213
- Prozentformat 325
- Prozess
 - auf beenden warten 559
 - beenden 556
 - kontrollieren 554
- Prozess-ID 551
 - abrufen 556
- Public 97, 102, 118
- Punkte zeichnen 750

Q

Quellcode 93

R

RadioButton 410
 RadioButton-Eigenschaft 465
 Raise()-Methode 228
 RaiseError 229
 RaiseEvent 291
 Read()-Methode 332, 628
 Read()-Methode (ADO.NET) 682
 Read()-Methode (XMLReader) 713
 ReadBlock()-Methode 628
 ReadInt32-Methode 636
 ReadLine()-Methode 328, 628
 ReadToEnd()-Methode 628
 ReadToEnd-Methode 514
 ReadXML()-Methode (DataSet) 706
 Redim 132
 Refresh()-Methode 746
 Registerkarte
 hinzufügen/löschen 423
 konfigurieren 423
 Registerkarten 422
 Reihenfolge ändern 423
 Registrierung
 Schlüssel
 anlegen 579
 auflisten 583
 hinzufügen 578
 löschen 581
 öffnen 579
 schließen 579
 Wert hinzufügen 578
 Wert lesen 581
 Wert löschen 581
 Wert schreiben 579
 Werte
 auflisten 583
 Zugriff 578
 Registrierungszugriff
 Remote 586
 RegistryKey
 Close()-Methode 579
 OpenSubKey()-Methode 579
 Reihenfolge
 der Steuerelemente 375
 Relationale Operatoren 151

Remote

 Registrierungszugriff 586
 Remove() 159
 Remove()-Methode 303, 306
 RemoveHandler 293
 RenderMode (Symbolleiste) 477
 Replace() 159
 ReportViewer-Steuerelement 696
 reservierte Schlüsselwörter (Visual Basic)
 104
 Ressource
 auswählen 363
 Return 210
 RichText 421
 RichTextBox
 Bereiche formatieren 539
 Inhalt selektieren 538
 Kontextmenü 539
 Steuerelement 422
 RichTextBox-Steuerelement 422
 RichText-Steuerelement 426
 Rmdir-Funktion 616
 RotateFlip()-Methode 755
 RTF-Dokument
 Bereiche formatieren 539
 speichern 535
 RTF-Dokumente
 laden 533
 Rückgängig machen 519
 Run()-Methode 370

S

Save()-Methode 755
 SaveFile-Methode 536
 SAX (Simple API for XML) 712
 Schaltfläche
 .ico-Datei zuweisen 480
 absenken 412
 Beschriftung ändern 361
 Click-Ereignisprozedur 356
 Eigenschaft
 Text 377
 Eigenschaften 376
 Eigenschaften (Symbolleiste) 480
 erzeugen 376
 flach darstellen 409
 mit Symbol 413
 Symbol anpassen 483
 Symbol wechseln 414

- Symbol zuweisen (Symbolleiste) 480
- Symbol zuweisen (ToolBar-Symbolleiste) 490
- Text und Bild ausrichten 413
- Text zuweisen (ToolBar-Symbolleiste) 490
- ToolStripDropDownButton 474
- Varianten bei ToolBar-Symbolleisten 490
- Schaltfläche (ToolBar)
 - Eigenschaften 490
- Schleife 196
 - Do ... Loop Until 198
 - Do ... Loop While 197
 - Do Until ... Loop 197
 - Do While ... Loop 196
 - Enumeration bearbeiten 204
 - For Each Next 206
 - For-Next 198
 - geschachtelte 198
- Schleifenindex 198
 - Schrittweite 199
- Schließen
 - überwachen 520
 - ungesicherte Änderungen abfangen 520
- Schlüssel
 - anlegen (Registrierung) 579
 - löschen (Registrierung) 581
 - öffnen (Registrierung) 579
 - schließen (Registrierung) 579
- Schlüsselwort
 - AddressOf 288
 - AS 101
 - Const 98
 - Delegate 288
 - Function 210
 - Get 245
 - Global 202
 - Imports 110
 - Me 270, 358
 - MustInherit 285
 - MustOverride 285
 - MyBase 270
 - MyClass 276
 - New 238, 241
 - NotInheritable 283
 - Optional 221
 - Overridable 277
 - Overrides 277
 - ParamArray 222
 - Protected 278
 - Return 210
 - Shadows 278
 - Static 224
 - Then 192
 - Throw 634
 - To 198
 - WithEvents 383
- Schlüsselwörter (Visual Basic 2005) 104
- Schnittstellen 236, 293
 - .NET 300
- Schriftartenauswahl 517
- Schriftattribute 539
- Schrittweite
 - Schleifenindex 199
- ScreenShot anfertigen 755
- Seite einrichten 728
- Seitenvorschau 732
- SelectAll-Methode 538
- Select-Case-Anweisung 194
- SelectedNode-Eigenschaft (TreeView) 443
- SelectedObjekt-Eigenschaft 720
- SelectedRtf-Eigenschaft 538
- SelectionColor-Eigenschaft 539
- SelectionFont-Eigenschaft 539
- SelectionLength-Eigenschaft 605
- SelectionStart-Eigenschaft 605
- Set 245
- SetAttr-Funktion 604, 608
- SetEnvironmentVariable()-Methode 578
- SetFileDate 611
- SetFileDate-Anwendung 609
- SetHelpString-Methode 390
- SetPixel()-Methode 755
- SetShowHelp-Methode 390
- SetToolTip-Methode 390
- Setup
 - Eigenschaft
 - AlwaysCreate 83–84
 - Author 81
 - InstallAllUser 81
 - Manufacturer 81
 - mit Windows-Installer 80
 - Startmenü-Verknüpfung definieren 83
- Setup.exe 77, 79
- SetValue()-Methode 579
- Shadows 278
- Shared 118, 235

- Share-Modus (Datei) 625
- Shell
 - Ordner öffnen 563
- Shell.Application 561
- Shell-Funktion 550
- Shortcut-Eigenschaft (Menü) 462
- Shortcut-Enumeration 462
- Show()-Methode 374
- ShowDialog()-Methode 238, 370, 724
- ShowGroups 448
- ShowHelp()-Methode 722
- Show-Methode 343, 370, 530
- ShowPanels-Eigenschaft (Statusbar) 501
- ShowPopup()-Methode 723
- ShowShortcut-Eigenschaft 462
- SimpleADODBBinding 679
- SimpleBitmapRotateConvert 755
- SimpleClipping 747
- SimpleDraw-Beispiel 746
- SimpleDrawBitmap 755
- SimpleDrawLines-Beispiel 752
- SimpleDrawPattern-Beispiel 752
- SimplePrint-Projekt 733
- SimpleScreenShot 757
- Sleep()-Methode 558
- SmartTag 696
- SmartTag-Bereich 425
- Sort()-Methode 303
- sortieren
 - Felder 189
- Spätes Binden 561
 - an COM-Anwendung 561
- SpecialFolder-Auflistung (Environment) 577
- Speichern
 - von Dokumenten 515
- Speichern unter
 - Dialog 515
- Split() 159
- SplitContainer 436
- Spring-Eigenschaft 495
- SQL-Datenbank
 - anlegen 656
- Stammnamespace 46
- Standarddialoge 505
 - Beispiel 505
- Standardformatbezeichner 324
- StartCap-Eigenschaft 751
- Startformular 46
- StartInfo-Eigenschaft (Process) 555
- Start-Methode 407
- StartsWith() 159
- Static 118
- StatusBar 497
 - Panels 500
- Statusbar
 - Beispiel 498
- StatusBar-Control 499
- StatusBar-Steuerelement 501
- Statusbereich
 - Infoanzeige beim Zeigen auf Symbol 568
- Statuscode
 - zurückgeben 547
- StatusLabel 494
- Statusleiste 494
 - aktualisieren 496, 500
 - einfügen 494
 - mit Symbol 501
 - Panel
 - aktualisieren 502
 - Panels
 - einblenden 501
 - ShowPanels-Eigenschaft 501
- Steuerelement
 - an DataSet binden 689
 - an Datenquelle binden 671
 - an Formularrändern verankern 367
 - beim Entwurf an Datenquellen binden 675
 - BindingNavigator 673
 - CheckedListBox 415
 - ColorDialog 507
 - ContextMenu 468–469
 - ContextMenuStrip 469–470
 - DataBindings-Eigenschaft 689
 - DateTimePicker 433
 - Dock-Eigenschaft 435
 - DomainUpDown 419
 - Eigenschaften
 - ändern 352
 - FlowLayoutPanel 452
 - FolderBrowserDialog 521
 - FontDialog 507
 - Größe ändern 352, 742
 - Größe festlegen 377
 - GroupBox 410
 - ImageList 413, 428
 - ImageView 428
 - Label 392–393

- ListBox 415
- ListView
 - füllen 444
- MainMenu 457
- MenuStrip 457
- MonthCalendar 433
- NotifyIcon 566
- NumericUpDown 419
- OpenFileDialog 507
- PageSetupDialog 726–727, 733
- Panel 436
- PictureBox 393
- Position festlegen 377
- positionieren 377
- PrintDialog 726, 733
- PrintDocument 726, 733
- PrintPreview 726, 733
- PrintPreviewDialog 727
- Process 554
- ProgressBar 419
- PropertyGrid 451
- RichTextBox 422
- SaveFileDialog 507
- sperrern 401
- StatusBar 501
- TabControl 422
- TableLayoutPanel 452
- TextBox 422
- TrackBar 419
- TreeView 439
- Vordergrundfarbe 353
- zum Formular addieren 378
- Steuerelement.verschieben 352
- Steuerelemente
 - an Datenbank binden 688
 - ausrichten 352
 - ComboBox 415
 - Container zur Layout-Kontrolle 452
 - dynamische Eigenschaften 716
 - PictureBox 392
- Steuerelement-OpenFileDialog 512
- Steuerelements
 - DataGridView 667
- Stop-Anweisung 67
- Stream-Objekt 624
- StreamReader()-Methode 627
- StreamReader-Klasse 514
- StreamWriter()-Methode 625
- String
 - Eigenschaft
 - Length 169
 - in Char-Array umwandeln 175
 - Methode
 - Join() 175
 - Remove 170
 - Replace 170
 - Split() 175
 - Substring 170
 - ToCharArray() 175
 - Trim 170
 - TrimEnd 170
 - TrimStart 170
 - mit Leerzeichen auffüllen 169
- StringBuilder 177
- String-Klasse
 - Member 158
- Structure-Anweisung 135
- Strukturen 135
- Substring() 159
- Subtraktion 149
- Suche
 - in unsortierten Feldern 187
- suchen
 - in Feldern 185
- suchen (in Feldern) 186
- Symbol
 - in Schaltfläche entfernen 413
- SymbolBarAlt 491
- SymbolBarCodeAlt 494
- Symboldateien
 - Download 48
 - entwerfen 48
- Symbolleiste 473
 - andocken 484, 487
 - Befehl an Schaltfläche anbinden 481
 - Bild/Text für Schaltfläche 478
 - ein-/ausblenden 486
 - Elemente hinzufügen 477
 - Elementposition ändern 484
 - entwerfen 474
 - Farbe anpassen 477
 - in ToolStripContainer übertragen 477
 - per Programm erzeugen 491
 - Schaltfläche
 - Text fehlt in ToolBar-Symbolleiste 488
 - Schaltfläche flach darstellen (ToolBar) 487

- Schaltflächen hinzufügen 477
- Schaltflächentext anpassen 483
- Separator einfügen 478
- Separator in ToolBar einfügen 490
- Steuerelement
 - DisplayStyle 478
- Steuerelemente
 - Eigenschaften 480
- ToolBar
 - Schaltflächen hinzufügen 489
- ToolBar-Schaltflächenbreite
 - automatisch anpassen 489
- ToolBar-Schaltflächensymbol zentrieren 489
- System.Boolean 106
- System.Byte 106
- System.Char 106
- System.Console 319
- System.Decimal 106
- System.Double 106
- System.Drawing.Drawing2D 751
- System.Int16 106
- System.Int32 106
- System.Int64 106
- System.Object 106
- System.Single 106
- System.String 106
- SystemDirectory abfragen 571
- Systeminformationen abfragen 571
- Systeminfos
 - per My-Objekt abfragen 572
- Systemmenü
 - ausblenden 364
 - Symbol 384
 - Symbol zuweisen 359
- Systemmodale Dialoge 341
- Systemordner
 - Pfade
 - auflisten 577
- Systemsteuerung
 - öffnen 565
- Systemverzeichnis
 - abfragen 571
- T**
- TabControl-Steuerelement 422
- Tabelle
 - Eigenschaften anpassen 669
 - in DataGridView darstellen 667, 671
- Tabellen
 - Daten anzeigen/eingeben 661
- Tabellenfeld
 - AutoIncrement-Eigenschaft 669
- TableAdapter-Steuerelement 672
- TableLayoutPanel-Steuerelement 452
- TabPage-Eigenschaft 423
- Tag aus Datum ermitteln 127
- Taskleiste, Eigenschaftenfenster öffnen 565
- Tastaturnavigation
 - aktivieren 410
 - bei Steuerelementen 409
 - in Menüs 458
- Tasteneingabe
 - warten auf 124
- Teilfenster
 - verwalten 37
- Text 354
 - formatieren 539
 - mit Leerzeichen auffüllen 169
- Textausgabe
 - Formatierung 322
 - Zeilenwechsel 322
- Textbox 366
- TextBox-Steuerelement
 - Eigenschaften 507
- Textdatei
 - lesen 514, 627
 - verändern 630
- Text-Eigenschaft 377
- Textelemente 422
- Textfeld
 - an Datenbanktabelle binden 675
 - Eingabe auswerten 485
 - Eingabe verifizieren 399
 - Großbuchstaben 401
 - Kennworteingabe 401
 - Markierung
 - aufheben 605
 - mehrzeilig 424
 - mit Bildlaufleiste 424
 - RichText-Variante 426
- TextRenderer-Klasse 739
- Then 192
- Thread, Sleep 558
- Throw 634
- Titelzeile aktualisieren 509
- ToCharArray() 159
- TodayDate-Eigenschaft 569

- ToLower() 159
 - ToolBar 473
 - Schaltflächen hinzufügen 489
 - ToolBar
 - Steuerelement 487
 - ToolBarButton-Auflistungs-Editor 489
 - ToolBar-Symbolleiste
 - Eigenschaften 487
 - umbrechen lassen 488
 - ToolBox
 - Registerkarte hinzufügen 456
 - Toolbox 34
 - alte Steuerelemente einbinden 456
 - Controls einbinden 456
 - zurücksetzen 456
 - Toolboxelemente auswählen 456
 - ToolStripContainer 475
 - ToolStripDropDownButton-Schaltfläche 474
 - ToolStrip-Steuerelement 475
 - Tooltip
 - für Formulare 387
 - ToolTipp 388
 - Top-Eigenschaft 377
 - TopMost-Eigenschaft 741
 - ToString() 159
 - ToString()-Methode 321
 - TotalPhysicalMemory 572
 - ToUpper() 159
 - TrackBarScroll-Ereignis 420
 - TrackBar-Steuerelement 419
 - Transparenz
 - Formular 383
 - TreeView 442
 - TreeView_Design 443
 - TreeViewDel 444
 - TreeViewManual.vb 443
 - TreeViewSplitContainer 442
 - TreeNode-Editor 441
 - TreeView
 - Add-Methode 443
 - füllen (per Code) 442
 - Knoten auswerten 443
 - Knoten hinzufügen 443
 - Knoten löschen 444
 - Stamm hinzufügen 441
 - TreeView-Steuerelement 439
 - Trennzeichen einfügen (Menü) 458
 - Trim 169
 - Trim() 159
 - TrimEnd() 160
 - TrimStart() 160
 - Try 229
 - Try ... Catch ... End Try 556
 - Try Catch Finally 229
 - Try...Catch...Finally-Block 196
 - TryCast()-Methode 147
 - Try-Catch 397
 - TryCatch 232
 - TxtFileReadWrite 635
 - Typ
 - IDictionary 575
 - RegistryKey 579–580
 - Typbezeichner 104
 - TypeOf 297
 - TypeValidationCompleted-Ereignis 408
 - Typkonvertierung 146
 - Typname 102
- U**
- Überladung von Operatoren 262
 - Umbenennen
 - Ordner/Datei 616
 - Umgebungsvariable
 - auflisten 575
 - umwandeln
 - Datentyp 146
 - Und 150
 - Undo 519
 - Undo-Methode 519
 - Unicode UTF-8 722
 - Uninstall-Eintrag 79
 - Updateart 79
 - Update-Methode 687
 - UserDomainName
 - abfragen 571
 - UserName 128
 - userSettings 721
- V**
- Variable
 - als Array deklarieren 121
 - in Klassen 116
 - statisch 118
 - Wertzuweisung 113
 - Variablen 112
 - Gültigkeit beeinflussen 117
 - Initialisierungswert 113
 - wo deklarieren? 113

- Variablendeklaration bei Klassen 115
- vbAbort 339
- vbAbortRetryIgnore 337
- vbApplicationModal 342
- vbCancel 339
- vbCritical 335
- vbCrLf-Konstante 323
- vbDefaultButton1 340
- vbDefaultButton2 340
- vbDefaultButton3 340
- vbExclamation 335
- vbIgnore 339
- vbInformation 335
- vbMsgBoxRight 342
- vbMsgBoxRtlReading 342
- vbMsgBoxSetForeground 342
- vbNo 339
- vbOK 339
- vbOKCancel 337
- vbOKOnly 337
- vbQuestion 335
- vbRetry 339
- vbRetryCancel 337
- vbSystemModal 342
- vbTab-Konstante 323
- vbYes 339
- vbYesNo 337
- vbYesNoCancel 337
- Verbundzuweisungen 143
- Vererbung 234, 265
 - Member erweitern 274
 - Member überschreiben 274
 - mit Konstruktoren 268
 - sperrern 283
- Verknüpfung
 - Symbol festlegen 83
- Veröffentlichungsort 79
- Verweis 236
 - entfernen 60
- Verweise im Projekt einfügen 49
- Verweistypen 155
- Verzeichnis
 - aktuelles 611
 - anlegen 616
 - löschen 616
 - übergeordnetes, ermitteln 598
 - umbenennen 616
 - wechseln 612
- Verzweigungen 192
- Visible 354
- Visible-Eigenschaft 462
- Visual Basic
 - Anweisungen 91
 - Class 96
 - Dateifunktionen 616
 - Datentypen 104
 - debuggen 64
 - Fortsetzungszeilen 92
 - Grundlagen 91
 - Kommentaranweisung 93
 - Konstante 97
 - Modul 96
 - Programmrahmen 96
 - Projekt anlegen 40
- Visual Basic 2005
 - Datentypen 106
 - Express Edition 25
 - installieren 29
 - Überblick 33
 - Schlüsselwörter 104
- Visual Studio
 - anlegen 349
 - anpassen 85
 - Anwendung ausführen 63
 - Codefenster 55
 - Debuggen 64
 - deinstallieren 29
 - Eigenschaftsfenster 55
 - Fenster im Überblick 54
 - installieren 26
 - Kurzeinführung 33
 - Projekt 39
 - Projekt anlegen 40
 - Projekttypen 43
 - Überblick 35
 - Varianten 24
 - Verweis entfernen 60
 - warten/reparieren 29
- Visual Studio 2005 19
- vollqualifizierte Namen (Bezeichner) 236
- Vordergrundfarbe
 - des Steuerelements 353
- Vorlage
 - Dialogfeld 364
 - Infocfeld 398
 - Windows-Anwendung 350
- VScrollBar 427

W

Währungsbetrag
 formatieren 325
 Währungsformat 325
 WaitForExit()-Methode 558–559
 Warten
 in einer Anwendung 557
 Webbrowser-Steuerelement 449
 Welcome-Programm 126
 Wert
 lesen (Registrierung) 581
 löschen (Registrierung) 581
 schreiben (Registrierung) 579
 Typ
 umwandeln 146
 umwandeln 146
 Werte
 auflisten (Registrierung) 583
 Wertetypen 155
 Werttypen 155
 Width-Eigenschaft 377
 Windows
 Shell nutzen 561
 WindowsDefaultBounds 360
 WindowsDefaultLocation 360
 WindowsForms-Designer 349
 Windows-Installer 80
 WindowState-Eigenschaft 525
 With-Anweisung 191
 WithEvents 291
 Wochentag
 aus Datum ermitteln 127
 Write
 Platzhalter 322
 Write()-Methode 319
 Platzhalterzeichen 114
 WriteFile-Methode (RichTextBox) 643
 WriteLine()
 formatierte Ausgabe 322
 WriteLine()-Methode 320
 Platzhalterzeichen 114
 WriteXML-Methode (ADO.NET) 705

X

XDR-Schemata 701
 XML 699
 Attributwerte 700
 Einführung 699
 Elemente 700

leeres Element 700
 Schema erstellen 704
 XML-Datei
 lesen 706
 XML-Daten
 speichern 705
 XML-Dokument
 erstellen 702
 validieren 702
 XmlDocumentRead 711
 XML-Editor 702
 XML-Kommentardatei 96
 XML-Kommentare 94
 XmlNode
 Member 709
 XMLStreamReader 715
 XMLTextReader 711
 XOr 150
 XSD-Schema
 erstellen 702
 XSD-Schemata 701

Y

Year 127

Z

Zahlenausgabe
 Formatierung 322
 Zeichenbereich
 Größe 742
 im Programm holen 741
 löschen 744
 Zeichenfolgen-Editor 416
 Zeichenfunktionen
 anwenden 747
 Zeichenkette
 bearbeiten 168
 suchen in 163
 Teile herauslösen 170
 Zeichenketten 158
 Beispiel 171
 Konkatenation 111
 Vergleich 161
 zerlegen 175
 Zeichenkettenvergleich
 länderspezifisch 164
 Zeichenkettenverknüpfung 320

Stichwortverzeichnis

- Zeichnen
 - Bézierkurven 750
 - Bitmap/Metagrafiken 752
 - Flächen 749
 - in Paint-Ereignisbehandlungsroutine 738
 - Linienstile 750
 - Muster 750
 - Punkte 750
- Zeichnen-Beispiel 750
- Zeilennummern
 - ablesen 69
 - im Codefenster einblenden 69
- Zeilenumbruch steuern (Textfeld) 425
- Zeilenwechsel 322
 - erzeugen (Konsole) 320
- Zeit
 - Eingabefeld 434
- Zugriffsberechtigungen 101
- Zugriffsmodus
 - Datei 633
- Zuweisung
 - einfach 143
- Zwischenablage
 - unterstützen 538

