

Klaus Löffelmann

Microsoft Visual Basic 2008 – Das Entwicklerbuch

Klaus Löffelmann

Microsoft Visual Basic 2008 – Das Entwicklerbuch

Microsoft®
Press

Klaus Löffelmann: Microsoft Visual Basic 2008 – Das Entwicklerbuch
Microsoft Press Deutschland, Konrad-Zuse-Str. 1, 85716 Unterschleißheim
Copyright © 2009 by Microsoft Press Deutschland

Das in diesem Buch enthaltene Programmmaterial ist mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor, Übersetzer und der Verlag übernehmen folglich keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programmmaterials oder Teilen davon entsteht. Die in diesem Buch erwähnten Software- und Hardwarebezeichnungen sind in den meisten Fällen auch eingetragene Marken und unterliegen als solche den gesetzlichen Bestimmungen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller.

Das Werk, einschließlich aller Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlags unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
11 10 09

ISBN 978-3-86645-506-1

© Microsoft Press Deutschland
(ein Unternehmensbereich der Microsoft Deutschland GmbH)
Konrad-Zuse-Str. 1, D-85716 Unterschleißheim
Alle Rechte vorbehalten

Korrektorat: Kristin Grauthoff, Lippstadt; Silvia Fehlow, Gütersloh
Fachlektorat: Ruprecht Dröge, Ratingen (<http://www.beconstructed.de>)
Satz: Silja Brands, Uta Berghoff, ActiveDevelop, Lippstadt (<http://www.ActiveDevelop.de>)
Layout: Gerhard Alfes, mediaService, Siegen (www.media-service.tv)
Umschlaggestaltung: Hommer Design GmbH, Haar (www.HommerDesign.com)
Gesamtherstellung: Kösel, Krugzell (www.KoeselBuch.de)

Für Silke Spindeler –

*warte einfach auf uns
im Restaurant am Rande
des Universums*

Inhaltsverzeichnis

Vorwort	XXVII
Einleitung	XXIX
Danksagungen	XXX
Support.....	XXXI
Teil A – Der Einstieg in Sprache und Entwicklungswerkzeuge	1
1 Einführung.....	3
Welches Vorwissen verlangt dieses Buch?	4
Welche Softwarevoraussetzungen benötigen Sie?.....	5
Wissenswertes zur Installation von Visual Studio 2008.....	7
Lauffähigkeit von Visual Studio und den erstellten Kompilaten unter den verschiedenen Betriebssystemen	8
Parallelinstallation von Visual Studio 6, Visual Studio 2002, Visual Studio 2003, Visual Studio 2005 und Visual Studio 2008	9
Die SQL-Datenbanken für Visual Studio 2008: SQL Server 2005 und SQL Server 2008	9
Der Umgang mit Web-Links in diesem Buch – http://www.activedevelop.de/vb2008	10
Die Begleitdateien zum Buch und das E-Book	11
Nützliches zu Visual Basic 2008 – www.codeclips.de	11
2 »Symbolischer Allzweckbefehlscode für Anfänger«.....	13
Visual Studio das erste Mal starten	14
Konsolenanwendungen (Console Applications).....	16
Anwendung starten.....	18
Anatomie eines (Visual Basic-)Programms	20
Programmstart mit der Main-Methode	23
Methoden mit und ohne Rückgabewerte.....	25
Funktionen mit Rückgabewerten.....	25
Deklaration von Variablen	25
Ausdrücke und Variablendefinitionen	27
Gleichzeitiges Deklarieren und Definieren von Variablen	27
Komplexe Ausdrücke	28
Boolesche Ausdrücke.....	29
Und was sind Objekte im Unterschied zu »normalen« Datentypen?	30
Ableiten von Objekten und abstrakten Objekten.....	31
Programmstrukturen.....	32

Schleifen	32
For/Next-Schleifen	32
For Each-Schleifen.....	35
Do/Loop-und While/End While-Schleifen.....	36
Exit – Vorzeitiges Verlassen von Schleifen	37
Continue – Vorzeitige Schleifenwiederholung	37
Anweisungen, die Programmcode bedingt ausführen – If/Then/Else und Select/Case.....	38
If ... Then ... Else ... ElseIf ... EndIf.....	38
Vergleichsoperatoren, die boolesche Ergebnisse zurückliefern.....	39
Select ... Case ... End Select.....	41
Vereinfachter Zugriff auf Objekte mit With/End With.....	42
Gültigkeitsbereiche von lokalen Variablen.....	42
 3 Einführung in das .NET Framework	45
Was ist .NET und aus was besteht es?	46
Was ist eine Assembly?	47
Was ist ein Namespace?	47
Was versteckt sich hinter CLR (Common Language Runtime) und CLI (Common Language Infrastructure)?	51
Was ist die FCL (Framework Class Library) und was die BCL (die Base Class Library)?	52
Was ist das CTS (Common Type System)?	53
Was ist CIL/MSIL (Microsoft-Intermediate Language) und wozu dient der JITter?.....	53
Erzwungene Typsicherheit und Deklarationszwang von Variablen	55
Namensgebung von Variablen.....	58
Und welche Sprache ist die beste?	60
 4 Der Schnelleinstieg in die Bedienung der Visual Studio-Entwicklungsumgebung (IDE)	61
Der erste Start von Visual Studio.....	62
Übernehmen von Visual Studio 2005-Einstellungen	63
Visual Studio 2005-Projekte zu Visual Studio 2008 migrieren.....	63
Die Startseite – der erste Ausgangspunkt für Ihre Entwicklungen	65
Der Visual Studio-Nachrichten-Channel	65
Anpassen der Liste der zuletzt bearbeiteten Projekte.....	66
Die IDE auf einen Blick	67
Genereller Umgang mit Fenstern in der Entwicklungsumgebung	69
Dokumentfenster	69
Toolfenster	69
Andocken von Toolfenstern	70
Arbeiten mit Toolfenstern.....	71
Navigieren durch Dateien und Toolfenster mit dem IDE-Navigator.....	72
Die wichtigsten Toolfenster	72
Projektmappen und Projekte mit dem Projektmappen-Explorer verwalten.....	73
Projektdateitypen.....	74
Alle Projektdateien anzeigen.....	74

Weitere Funktionen des Projektmappen-Explorers	75
Organisieren von Codedateien in Unterordnern	75
Dateioperationen innerhalb des Projektmappen-Explorers.....	76
Das Eigenschaftenfenster	77
Einrichten und Verwalten von Ereigniscode mit dem Eigenschaftenfenster	77
Die Fehlerliste	79
Konfigurieren von Warnungen in den Projekteigenschaften	80
Die Aufgabenliste	81
Das Ausgabefenster.....	83
Die dynamische Hilfe	85
Die Klassenansicht	87
Codeeditor und Designer.....	87
Wichtige Tastenkombinationen auf einen Blick	87
Verbesserungen an der integrierten Entwicklungs- umgebung (IDE) in Visual Studio 2008	89
Einfacheres Positionieren und Andocken von Toolfenstern.....	89
Navigieren durch Dateien mit dem IDE-Navigator	90
Wechseln zu bestimmten Dateien im Editor	90
Navigieren zwischen Toolfenstern in der IDE.....	91
Umgebungsschriftart definieren	91
Designer für Windows Presentation Foundation-Projekte	93
IntelliSense-Verbesserungen.....	95
Syntax-Tooltipps	95
Filtern beim Tippen	96
 5 Einführung in Windows Forms – Designer und Codeeditor am Beispiel.....	99
Das Fallbeispiel – Der DVD-Hüllen-Generator »Covers«	100
Das »Pflichtenheft« von Covers	101
Erstellen eines neuen Projektes	103
Gestalten von Formularen mit dem Windows Forms-Designer	105
Positionieren von Steuerelementen.....	105
Häufige Arbeiten an Steuerelementen mit Smarttags erledigen	109
Dynamische Anordnung von Steuerelementen zur Laufzeit	110
Automatisches Scrollen von Steuerelementen in Containern.....	121
Selektieren von Steuerelementen, die Sie mit der Maus nicht erreichen	122
Festlegen der Tabulatorreihenfolge (Aktivierreihenfolge) von Steuerelementen	123
Über die Eigenschaften Name, Text und Caption.....	125
Einrichten von Bestätigungs- und Abbrechen-Funktionalitäten für	
Schaltflächen in Formularen.....	127
Hinzufügen neuer Formulare zu einem Projekt.....	127
Wie geht's weiter?.....	129
Namensgebungskonventionen für Steuerelemente in diesem Buch	130
Funktionen zum Layouten von Steuerelementen im Designer.....	131
Tastaturkürzel für die Platzierung von Steuerelementen.....	133

Der Codeeditor	133
Die Wahl der richtigen Schriftart für ermüdfreies Arbeiten	133
Viele Wege führen zum Codeeditor	134
IntelliSense – Ihr stärkstes Zugpferd im Coding-Stall	135
Automatische Vervollständigung von Struktur-Schlüsselworten und Codeeinrückung	137
Fehlererkennung im Codeeditor	138
XML-Dokumentationskommentare für IntelliSense bei eigenen Objekten und Klassen	142
Hinzufügen neuer Codedateien zum Projekt	146
Code umgestalten (Refactoring)	148
Die Bibliothek der Codeausschnitte (Code Snippets Library)	151
Einstellen des Speicherns von Anwendungseinstellungen mit dem Settings-Designer...	155
Weitere Funktionen des Codeeditors.....	163
Aufbau des Codefensters	163
Automatischen Zeilenumbruch aktivieren/deaktivieren.....	163
Navigieren zu vorherigen Bearbeitungspositionen im Code	164
Rechteckige Textmarkierung	164
Gliederungsansicht	165
Suchen und Ersetzen, Suche in Dateien	166
Suchen in Dateien	167
Inkrementelles Suchen	168
Gehe zu Zeilennummern	168
Lesezeichen	169
6 Einführung in Windows Presentation Foundation.....	171
Was ist die Windows Presentation Foundation?	172
Was ist so neu an WPF?	173
25 Jahre Windows, 25 Jahre gemalte Schaltflächen	174
Wie WPF Designer und Entwickler zusammenbringt	185
XAML: Extensible Application Markup Language.....	187
Der WPF-Designer	194
Ereignisbehandlungsroutinen in WPF und Visual Basic	195
Logischer und visueller Baum.....	195
Die XAML-Syntax im Überblick	197
Ein eigenes XAMLPad	201
Zusammenfassung	206
7 Die essentiellen .NET-Datentypen.....	207
Numerische Datentypen	208
Numerische Datentypen deklarieren und definieren	209
Delegation numerischer Berechnungen an den Prozessor	210
Hinweis zur CLS-Konformität	213
Die numerischen Datentypen auf einen Blick	213
Tabellarische Zusammenfassung der numerischen Datentypen	220

Rundungsfehler bei der Verwendung von Single und Double.....	221
Besondere Funktionen, die für alle numerischen Datentypen gelten	224
Spezielle Funktionen der Fließkommatypen	228
Spezielle Funktionen des Wertetyps Decimal.....	230
Der Datentyp Char.....	230
Der Datentyp String.....	231
Strings – gestern und heute	232
Strings deklarieren und definieren	232
Der String-Konstruktor als Ersatz von String\$.....	233
Einem String Zeichenketten mit Sonderzeichen zuweisen.....	234
Speicherbedarf von Strings	235
Strings sind unveränderlich	235
Speicheroptimierung von Strings durch das Framework	236
Ermitteln der String-Länge	237
Ermitteln von Teilen eines Strings oder eines einzelnen Zeichens.....	237
Angleichen von String-Längen.....	238
Suchen und Ersetzen.....	238
Algorithmisches Auflösen eines Strings in Teile	242
Ein String-Schmankerl zum Schluss.....	243
Iterieren durch einen String	247
Stringbuilder vs. String – wenn es auf Geschwindigkeit ankommt.....	247
Der Datentyp Boolean.....	254
Konvertieren von und in numerische Datentypen	254
Konvertierung von und in Strings	255
Der Datentyp Date	255
Rechnen mit Zeiten und Datumswerten – TimeSpan	256
Bibliothek mit brauchbaren Datumsrechenfunktionen	257
Zeichenketten in Datumswerte wandeln	260
.NET-Äquivalente primitiver Datentypen	263
 8 Tipps & Tricks für das angenehme Entwickeln zuhause und unterwegs	267
Der Einsatz mehrerer Monitore	268
Und die Bildschirmschirmdarstellung auf Notebooks?.....	270
Sichern, Wiederherstellen oder Zurücksetzen aller Visual Studio-Einstellungen	270
Sichern der Visual Studio-Einstellungen.....	270
Wiederherstellen von Visual Studio-Einstellungen	272
Zurücksetzen der IDE-Einstellungen in den Ausgangszustand	274
Wieviel Arbeitsspeicher darf's denn sein?	276
Testen Ihrer Software unterwegs und zuhause mit Virtualisierungssystemen	277
Microsoft Virtual PC	278
Virtual Server	281
Hyper-V in Windows Server 2008.....	282
Hilfe zur Selbsthilfe	283

Erweitern Sie die Codeausschnittsbibliothek um eigene Codeausschnitte	288
Erstellen einer Code Snippets-XML-Vorlage	290
Hinzufügen einer neuen Snippet-Vorlage zur Snippet-Bibliothek (Codeausschnittsbibliothek)	293
Verwenden des neuen Codeausschnittes	294
Parametrisieren von Codeausschnitten.....	295
 Teil B - Umsteigen auf Visual Basic 2008	303
 9 Migrieren zu Visual Basic 2008 – Vorüberlegungen.....	305
Das Problem der Visual Basic 6.0-Migration	306
Kurzfristige Rettung mit virtuellen PCs	308
Grundsätzliches zur Umstellung von Visual Basic 6-Anwendungen auf .NET	308
Aufwandsabschätzung einer Migration.....	310
Interop Forms Toolkit – »Weiche Migration« für kleinere und modulare Projekte	311
10-teilige Webcast-Reihe zum Thema Migration.....	311
 10 Migration von Visual Basic 6-Anwendungen	313
Unterschiede in der Variablenbehandlung.....	315
Veränderungen bei primitiven Integer-Variablen – die Größen von Integer und Long und der neue Typ Short	315
Typen, die es nicht mehr gibt	316
... und die primitiven Typen, die es jetzt gibt	316
Deklaration von Variablen und Variablentypzeichen	318
Typsicherheit und Typliterale zur Typdefinition von Konstanten	320
Deklaration und Definition von Variablen »in einem Rutsch«	325
Lokaler Typrückschluss	325
Vorsicht: New und New können zweierlei in VB6 und VB.NET sein!	326
Überläufe bei Fließkommazahlen und nicht definierte Zahlenwerte	327
Alles ist ein Objekt oder »let Set be«	329
Gezieltes Freigeben von Objekten mit Using.....	330
Direktdeklaration von Variablen in For-Schleifen	333
Unterschiede bei verwendbaren Variablentypen für For/Each in VB6 und VB.NET	334
Continue in Schleifen.....	334
Gültigkeitsbereiche von Variablen	335
Globale bzw. öffentliche (public) Variablen	335
Variablen mit Gültigkeit auf Modul, Klassen oder Formularebene.....	336
Gültigkeitsbereiche von lokalen Variablen	337
Arrays	339
Die Operatoren += und -= und ihre Verwandten	340
Die Bitverschiebeoperatoren << und >>	341
Fehlerbehandlung	342
Elegantes Fehlerabfangen mit Try/Catch/Finally	344
Kurzschlussauswertungen mit OrElse und AndAlso.....	348

Variablen und Argumente auch an Subs in Klammern!	350
Namespaces und Assemblies	350
Assemblies	350
Namespaces	353
So bestimmen Sie Assembly-Namen und Namespace für Ihre eigenen Projekte	356
Verschiedene Namespaces in einer Assembly	357
Zugriff auf den Framework-System-Namespace mit Global.....	358
 11 Neues im Visual Basic 2008-Compiler	361
First and definitely not least – die Performance des Visual Basic 2008-Compilers	362
Framework Version-Targeting (Framework-Versionszielwahl) bei Projekten	364
Lokale Typrückschlüsse (Local Type Inference)	366
Generelles Einstellen von Option Infer, Strict, Explicit und Compare.....	367
If-Operator vs. IIf-Funktion.....	368
Festlegen der Projekteinstellungen für die Benutzerkontensteuerung (Windows Vista).....	368
Nullable-Typen	369
Besonderheiten bei Nullable beim Boxen.....	372
Anonyme Typen	374
Lambda-Ausdrücke.....	375
Abfrageausdrücke mit LINQ	375
Erweiterungsmethoden	375
Der eigentliche Zweck von Erweiterungsmethoden	376
Inhalte von System- oder externen Typen in Vorschaufenstern fürs	
Debuggen konfigurieren	377
Zugriff auf den .NET Framework-Quellcode beim Debuggen	378
 Teil C – Objektorientiertes Programmieren	385
 12 Einführung in die objektorientierte Programmierung.....	387
Was spricht für Klassen und Objekte?	389
Miniadresso – die prozedurale Variante.....	390
 13 Klassentreffen.....	393
Was ist eine Klasse?	394
Klassen mit New instanziieren	394
New oder nicht New – wieso es sich bei Objekten um Verweistypen handelt	396
Nothing	399
Klassen anwenden	400
Wertetypen	401
Wertetypen mit Structure erstellen	403
Wertetypen durch Zuweisung klonen.....	403
Wann Werte-, wann Referenztypen verwenden?.....	404
Konstanten vs. Felder (Klassen-Membervariablen)	405

14	Klassencode entwickeln.....	407
	Eigenschaften.....	408
	Zuweisen von Eigenschaften.....	410
	Ermitteln von Eigenschaften.....	410
	Eigenschaften mit Parametern.....	412
	Default-Eigenschaften (Standardeigenschaften).....	413
	Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage?	414
	Der Konstruktor einer Klasse – bestimmen, was bei New passiert	418
	Parametrisierte Konstruktoren	420
	Hat jede Klasse einen Konstruktor?	422
	Klassenmethoden mit Sub und Function.....	426
	Überladen von Methoden, Konstruktoren und Eigenschaften	427
	Methodenüberladung und optionale Parameter.....	429
	Gegenseitiges Aufrufen von überladenen Methoden.....	430
	Gegenseitiges Aufrufen von überladenen Konstruktoren.....	432
	Überladen von Eigenschaften-Prozeduren mit Parametern	433
	Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften – Gültigkeitsbereiche definieren	434
	Zugriffsmodifizierer bei Klassen	434
	Zugriffsmodifizierer bei Prozeduren (Subs, Functions, Properties)	435
	Zugriffsmodifizierer bei Variablen.....	435
	Unterschiedliche Zugriffsmodifizierer für Eigenschaften-Accessors	436
	Statische Komponenten	437
	Statische Methoden	438
	Statische Eigenschaften	440
	Module in Visual Basic – automatisch statische Klassen erstellen.....	440
	Delegaten und Lambda-Ausdrücke	441
	Umgang mit Delegaten am Beispiel.....	441
	Lambda-Ausdrücke	444
	Lambda-Ausdrücke am Beispiel	445
	Über mehrere Codedateien aufgeteilter Klassencode – Das Partial-Schlüsselwort	447
	Partieller Klassencode bei Methoden und Eigenschaften	447
15	Vererben von Klassen und Polymorphie	449
	Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance)	450
	Initialisierung von Member-Variablen bei Klassen ohne Standardkonstruktoren	458
	Überschreiben von Methoden und Eigenschaften.....	459
	Überschreiben vorhandener Methoden und Eigenschaften von Framework-Klassen	462
	Das Speichern von Objekten im Arbeitsspeicher.....	463
	Polymorphie	466
	Polymorphie in der Praxis.....	470
	Polymorphie und der Gebrauch von Me, MyClass und MyBase	483
	Abstrakte Klassen und virtuelle Prozeduren.....	485
	Eine Klasse mit MustInherit als abstrakt deklarieren.....	485

Eine Methode oder Eigenschaft einer abstrakten Klasse mit MustOverride als virtuell deklarieren	486
Schnittstellen (Interfaces).....	488
Unterstützung bei abstrakten Klassen und Schnittstellen durch den Editor	495
Schnittstellen, die Schnittstellen implementieren	500
Einbinden mehrerer Schnittstellen in eine Klasse.....	502
Die Methoden und Eigenschaften von Object	503
Polymorphie am Beispiel von ToString und der ListBox.....	503
Prüfen auf Gleichheit von Objekten mit Object.Equals oder dem Is/ IsNot-Operator....	506
Equals, Is und IsNot im praktischen Entwicklungseinsatz	509
Übersicht über die Eigenschaften und Methoden von Object	510
Shadowing (Überschatten) von Klassenprozeduren	510
Shadows als Unterbrecher der Klassenhierarchie	512
Sonderform »Modul« in Visual Basic	516
Singleton-Klassen und Klassen, die sich selbst instanziieren.....	516
 16 Entwickeln von Wertetypen.....	519
Erstellen von Wertetypen mit Structure am praktischen Beispiel	520
Unterschiedliche Verhaltensweisen von Werte- und Referenztypen	526
Verhalten der Parameterübergabe mit ByVal und ByRef steuern	527
Konstruktoren und Standardinstanzierungen von Wertetypen	528
Gezieltes Zuweisen von Speicherbereichen für Struktur-Member mit den StructLayout- und FieldOffset-Attributen.....	531
Performance-Unterschiede zwischen Werte- und Referenztypen.....	533
Wieso kann durch Vererben aus Object (Referenztyp) ValueType (Wertotyp) werden?	534
 17 Typenumwandlung (Type Casting) und Boxing von Wertetypen	535
Konvertieren von primitiven Typen	536
Konvertieren von und in Zeichenketten (Strings).....	538
Konvertieren von Strings mit den Parse- und ParseExact-Methoden	539
Konvertieren in Strings mit der ToString-Methode	539
Abfangen von fehlschlagenden Typkonvertierungen mit TryParse oder Ausnahmebehandlern	540
Casten von Referenztypen mit DirectCast	541
Boxing von Wertetypen und primitiven Typen	542
Übrigens: Was DirectCast nicht kann.....	545
Wann wird und wann wird nicht geboxed?	545
Wertänderung von durch Schnittstellen geboxten Wertetypen.....	545
 18 Beerdigen von Objekten – Dispose, Finalize und der Garbage Collector	549
Der Garbage Collector – die Müllabfuhr in .NET	552
Generationen	553
Die Geschwindigkeit der Objektbereitstellung	554

Finalize	556
Wann Finalize nicht stattfindet	557
Dispose	560
Unterstützung durch den Visual Basic-Editor beim Einfügen eines Disposable-Patterns.....	570
19 Eigene Operatoren für benutzerdefinierte Typen.....	571
Einführung in Operatorenprozeduren	572
Vorbereitung einer Struktur oder Klasse für Operatorenprozeduren.....	574
Implementierung von Operatoren	578
Überladen von Operatorenprozeduren.....	579
Implementierung von Vergleichsoperatoren	580
Implementierung von Typkonvertierungsoperatoren mit Operator CType	581
Implementieren von Wahr- und Falsch-Auswertungsoperatoren	583
Problembehandlungen bei Operatorenprozeduren	584
Aufgepasst bei der Verwendung von Referenztypen	584
Übersicht der implementierbaren Operatoren.....	586
20 Ereignisse und Ereignishandler.....	589
Konsumieren von Ereignissen mit WithEvents und Handles	591
Auslösen von Ereignissen	594
Der Umweg über Onxxx.....	595
Zur-Verfügung-Stellen von Ereignisparametern.....	596
Die Quelle des Ereignisses: Sender.....	597
Nähere Informationen zum Ereignis: EventArgs	598
Dynamisches Anbinden von Ereignissen mit AddHandler.....	600
Teil D – Programmieren von und mit Datenstrukturen des .NET Frameworks	609
21 Enums	611
Bestimmung der Werte der Aufzählungselemente.....	613
Bestimmung der Typen der Aufzählungselemente	614
Ermitteln des Elementtyps zur Laufzeit	614
Konvertieren von Enums	614
In Zahlenwerte umwandeln und aus Werten definieren.....	615
In Strings umwandeln und aus Strings definieren.....	615
Flags-Enum (Flags-Aufzählungen)	615
Abfrage von Flags-Aufzählungen.....	617
22 Arrays und Collections.....	619
Grundsätzliches zu Arrays.....	620
Änderung der Array-Dimensionen zur Laufzeit	621
Wertevorbelegung von Array-Elementen im Code.....	623
Mehrdimensionale und verschachtelte Arrays.....	624
Die wichtigsten Eigenschaften und Methoden von Arrays.....	625

Implementierung von Sort und BinarySearch für eigene Klassen	627
Enumeratoren	635
Benutzerdefinierte Enumeratoren durch Implementieren von IEnumerable	636
Grundsätzliches zu Auflistungen (Collections)	638
Wichtige Auflistungen der Base Class Library	642
ArrayList – universelle Ablage für Objekte	642
Typsichere Auflistungen auf Basis von CollectionBase	644
Hashtables – für das Nachschlagen von Objekten	648
Anwenden von Hashtables	648
Verwenden eigener Klassen als Schlüssel (Key)	658
Enumerieren von Datenelementen in einer Hashtable	661
DictionaryBase	662
Queue – Warteschlangen im FIFO-Prinzip	662
Stack – Stapelverarbeitung im LIFO-Prinzip	663
SortedList – Elemente ständig sortiert halten	664
23 Generics (Generika) und generische Auflistungen.....	667
Einführung	668
Generics: Verwenden einer Codebasis für verschiedene Typen	668
Lösungsansätze	669
Typengeneralisierung durch den Einsatz generischer Datentypen	671
Beschränkungen (Constraints)	674
Beschränkungen für generische Typen auf eine bestimmte Basisklasse	675
Beschränkungen auf Klassen, die bestimmte Schnittstellen implementieren	679
Beschränkungen auf Klassen, die über einen Standardkonstruktor verfügen	683
Beschränkungen auf Wertetypen	683
Kombinieren von Beschränkungen und Bestimmen mehrerer Typparameter	683
Generische Auflistungen (Generic Collections)	684
KeyedCollection – Schlüssel/Wörterbuch-Auflistung mit zusätzlichen Index-Abrufen	687
Elementverkettungen mit LinkedList(Of)	689
List(Of)-Auflistungen und Lambda-Ausdrücke	691
24 Serialisierung von Typen	697
Einführung in Serialisierungstechniken	698
Serialisieren mit SoapFormatter und BinaryFormatter	701
Flaches und tiefes Klonen von Objekten	706
Universelle DeepClone-Methode	710
Serialisieren von Objekten mit Zirkelverweisen	712
Serialisierung von Objekten unterschiedlicher Versionen beim Einsatz von BinaryFormatter oder SoapFormatter-Klassen	715
XML-Serialisierung	715
Prüfen der Versionsunabhängigkeit der XML-Datei	720
Serialisierungsfehler bei KeyedCollection	721
Workaround	725

25	Attribute und Reflection – Wenn Klassen ein Bewusstsein entwickeln.....	727
	Genereller Umgang mit Attributen	728
	Einsatz von Attributen am Beispiel von ObsoleteAttribute	729
	Die speziell in Visual Basic verwendeten Attribute	731
	Einführung in Reflection	732
	Die Type-Klasse als Ausgangspunkt für alle Typenuntersuchungen	733
	Klassenanalysefunktionen, die ein Type-Objekt bereitstellt	734
	Objekthierarchie von MemberInfo und Casten in den spezifischen Info-Typ	736
	Ermitteln von Eigenschaftswerten über PropertyInfo zur Laufzeit	737
	Erstellung benutzerdefinierter Attribute und deren Erkennen zur Laufzeit	737
	Ermitteln von benutzerdefinierten Attributen zur Laufzeit.....	741
26	Kultursensible Klassen entwickeln.....	743
	Allgemeines über Format Provider in .NET	744
	Kulturabhängige Formatierungen mit CultureInfo	745
	Vermeiden von kulturabhängigen Programmfehlern.....	747
	Formatierung durch Formatzeichenfolgen	748
	Formatierung von numerischen Ausdrücken durch Formatzeichenfolgen	748
	Formatierung von numerischen Ausdrücken durch vereinfachte Formatzeichenfolgen	751
	Formatierung von Datums- und Zeitausdrücken durch Formatzeichenfolgen.....	752
	Formatierung von Zeitausdrücken durch vereinfachte Formatzeichenfolgen.....	757
	Gezielte Formatierungen mit Format Providern.....	758
	Gezielte Formatierungen von Zahlenwerten mit NumberFormatInfo	758
	Gezielte Formatierungen von Zeitwerten mit DateTimeFormatInfo.....	759
	Kombinierte Formatierungen	760
	Ausrichtungen in kombinierten Formatierungen	761
	Angeben von Formatzeichenfolgen in den Indexkomponenten.....	762
	So helfen Ihnen benutzerdefinierte Format Provider, Ihre Programme	
	zu internationalisieren.....	763
	Benutzerdefinierte Format Provider durch IFormatProvider und ICustomFormatter	774
	Automatisch formatierbare Objekte durch Einbinden von IFormattable	777
27	Reguläre Ausdrücke (Regular Expressions).....	779
	RegExperimente mit dem RegExplorer	780
	Erste Gehversuche mit Regulären Ausdrücken	782
	Einfache Suchvorgänge	782
	Einfache Suche nach Sonderzeichen	783
	Komplexere Suche mit speziellen Steuerzeichen	784
	Verwendung von Quantifizierern	786
	Gruppen	788
	Suchen und Ersetzen	790
	Captures	791
	Optionen bei der Suche.....	794
	Steuerzeichen zu Gruppdefinitionen	795

Programmieren von Regulären Ausdrücken.....	796
Ergebnisse im Match-Objekt.....	796
Die Matches-Auflistung	797
Abrufen von Captures und Gruppen eines Match-Objektes.....	798
Regex am Beispiel: Berechnen beliebiger mathematischer Ausdrücke.....	800
Der Formelparser	802
Die Klasse ADFormularParser	802
Vererben der Klasse ADFormularParser, um eigene Funktionen hinzuzufügen.....	820
Teil E - Entwicklungsvereinfachungen in Visual Basic 2008	821
28 Eine philosophische Betrachtung der Visual Basic-spezifischen Vereinfachungen.....	823
Die VB2008-Vereinfachungen am Beispiel DotNetCopy	824
DotNetCopy mit /Autostart und /Silent-Optionen	827
Die prinzipielle Funktionsweise von DotNetCopy	829
29 Der My-Namespace	831
Formulare ohne Instanziierung aufrufen.....	833
Auslesen der Befehlszeilenargumente mit My.Application.CommandLineArgs.....	835
Gezieltes Zugreifen auf Ressourcen mit My.Resources.....	837
Anlegen und Verwalten von Ressource-Elementen.....	837
Abrufen von Ressourcen mit My.Ressources.....	838
Internationalisieren von Anwendungen mithilfe von Ressource-Dateien und dem My-Namespace	840
Vereinfachtes Durchführen von Dateioperationen mit My.Computer.FileSystem	843
Verwenden von Anwendungseinstellungen mit My.Settings.....	846
30 Das Anwendungsframework.....	849
Die Optionen des Anwendungsframeworks.....	851
Windows XP Look and Feel für eigene Windows-Anwendungen – Visuelle XP-Stile aktivieren	851
Verhindern, dass Ihre Anwendung mehrfach gestartet wird – Einzelinstanzanwendung erstellen.....	851
MySettings-Einstellungen automatisch sichern – Eigene Einstellungen beim Herunterfahren speichern.....	851
Bestimmen, welcher Benutzer-Authentifizierungsmodus verwendet wird	851
Festlegen, wann eine Anwendung »zu Ende« ist – Modus für das Herunterfahren.....	852
Einen Splash-Dialog beim Starten von komplexen Anwendungen anzeigen – Begrüßungsdialog.....	852
Eine Codedatei implementieren, die Anwendungsergebnisse behandelt	852

Teil F – Language Integrated Query – LINQ	857
31 Einführung in Language Integrated Query (LINQ).....	859
Wie »geht« LINQ prinzipiell	862
Erweiterungsmethoden als Basis für LINQ	865
Die Where-Methode.....	867
Die Select-Methode	868
Anonyme Typen	869
Typrückschluss für generische Typparameter	869
Die OrderBy-Methode	872
Sortieren nach Eigenschaften, die per Variable übergeben werden.....	874
Die GroupBy-Methode	875
Kombinieren von LINQ-Erweiterungsmethoden.....	876
Vereinfachte Anwendung von LINQ – Erweiterungsmethoden mit der LINQ-Abfragesyntax.....	878
32 LINQ to Objects	879
Einführung in LINQ to Objects.....	880
Verlieren Sie die Skalierbarkeit von LINQ nicht aus den Augen!	880
Der Aufbau einer LINQ-Abfrage.....	881
Kombinieren und verzögertes Ausführen von LINQ-Abfragen	887
Faustregeln für das Erstellen von LINQ-Abfragen.....	889
Kaskadierte Abfragen	890
Gezieltes Auslösen von Abfragen mit ToArray oderToList.....	890
Verbinden mehrerer Auflistungen zu einer neuen.....	891
Implizite Verknüpfung von Auflistungen.....	892
Explizite Auflistungsverknüpfung mit Join	894
Gruppieren von Auflistungen	894
Gruppieren von Listen aus mehreren Auflistungen.....	896
Group Join	897
Aggregatfunktionen.....	898
Zurückliefern mehrerer verschiedener Aggregationen	899
Kombinieren von gruppierten Abfragen und Aggregatfunktionen.....	899
33 LINQ to XML	901
Einführung in LINQ to XML.....	902
XML-Dokumente verarbeiten – Visual Basic 2005 im Vergleich mit Visual Basic 2008	903
XML-Literale – XML direkt im Code ablegen.....	904
Einbetten von Ausdrücken in XML-Literalen.....	905
Erstellen von XML mithilfe von LINQ	905
Abfragen von XML-Dokumenten mit LINQ to XML.....	907
IntelliSense-Unterstützung für LINQ to XML-Abfragen.....	908

34	LINQ to SQL.....	911
	Einleitung	912
	Object Relational Mapper (O/RM)	912
	Objekt-relationale Unverträglichkeit – Impedance Mismatch	913
	Microsoft SQL Server	914
	SQL Server 2008 Express Edition with Tools.....	915
	Einsatz von SQL Server Express in eigenen Anwendungen	916
	Voraussetzungen für die Beispiele dieses und des nächsten Kapitels	918
	Download und Installation von <i>SQL Server 2008 Express Edition with Tools</i>	919
	Installation der Beispieldatenbanken	928
	Anfügen (»Attachen«) der Beispieldatenbanken an die SQL Server-Instanz	930
	LINQ to SQL oder LINQ to Entities – was ist besser, was ist die Zukunft?	935
	Hat LINQ to SQL eine Zukunft?.....	936
	Entscheidungshilfe – Gegenüberstellung der wichtigsten Features von LINQ to SQL und LINQ to Entities	938
	Wie es bisher war – ADO.NET 2.0 vs. LINQ in .NET 3.5	939
	LINQ to SQL am Beispiel – Die ersten Schritte	941
	Protokollieren der generierten T-SQL-Befehle.....	951
	Verzögerte Abfrageausführung und kaskadierte Abfragen	952
	Eager und Lazy-Loading – Steuern der Ladestrategien bei 1:n-Relationen	955
	Trennen des Abfrageergebnisses vom Kontext.....	960
	Daten verändern, speichern, einfügen und löschen	961
	Datenänderungen mit SubmitChanges an die Datenbank übermitteln	961
	Einfügen von Datensätzen mit InsertOnSubmit	963
	Daten löschen mit DeleteOnSubmit.....	967
	Concurrency-Checks (Schreibkonfliktprüfung)	968
	Transaktionen	971
	TransactionScope (Transaktionsgültigkeitsbereich)	971
	Verwenden der Transaktionssteuerung des DataContext	972
	Was, wenn LINQ einmal nicht reicht	973
	LINQ-Abfragen dynamisch aufbauen	974
	Abfragen und IQueryable	975
	Dynamische Abfragen durch <i>DynamicExtension.vb</i>	976
35	LINQ to Entities – Programmieren mit dem Entity Framework.....	979
	Voraussetzungen für das Verstehen dieses Kapitels.....	981
	Technische Voraussetzungen	981
	Prinzipielle Funktionsweise eines Entity Data Model (EDM)	981
	LINQ to Entities – ein erstes Praxisbeispiel.....	983
	Nachträgliches Ändern des Entitätscontainernamens	988
	Abfrage von Daten eines Entitätsmodells	990
	Abfrage von Daten mit LINQ to Entities-Abfragen	991
	Wie Abfragen zum Datenprovider gelangen – Entity-SQL (eSQL)	993

Anpassen des Namens der Entitätenmenge.....	993
Generierte SQL-Anweisungen unter die Lupe nehmen	994
Lazy- und Eager-Loading im Entity Framework	996
Anonymisierungsvermeidung bei Abfragen in verknüpften Tabellen.....	1000
Kompilierte Abfragen	1003
Daten verändern, speichern, einfügen und löschen.....	1004
Datenänderungen mit SaveChanges an die Datenbank übermitteln.....	1004
Einfügen von verknüpften Daten in Datentabellen.....	1006
Daten aus Tabellen löschen.....	1008
Concurrency-Checks (Schreibkonfliktprüfungen).....	1010
Ausblick.....	1012
 Teil G – SmartClient-Anwendungen entwickeln.....	1013
 36 Einführung in SmartClient-Entwicklung	1015
Die Gretchenfrage: Windows Forms oder WPF?	1016
Vorüberlegungen zur Smart Client-Entwicklung	1018
Strukturieren von Daten in Windows Forms-Anwendungen	1019
 37 Programmieren mit Windows Forms.....	1031
Formulare zur Abfrage von Daten verwenden – Aufruf von Formularen aus Formularen	1032
Der Umgang mit modalen Formularen	1032
Überprüfung auf richtige Benutzereingaben in Formularen	1043
Anwendungen über die Tastatur bedienbar machen.....	1047
Definition von Schnellzugriffstasten per »&«-Zeichen.....	1048
Accept- und Cancel-Schaltflächen in Formularen	1051
Über das »richtige« Schließen von Formularen	1051
Unsichtbarmachen eines Formulars mit Hide	1052
Schließen des Formulars mit Close.....	1052
Entsorgen des Formulars mit Dispose.....	1053
Grundsätzliches zum Darstellen von Daten aus Auflistungsklassen in Steuerelementen	1053
Darstellen von Daten aus Auflistungen im ListView-Steuerelement.....	1054
Spaltenköpfe	1054
Hinzufügen von Daten.....	1055
Beschleunigen des Hinzufügens von Elementen	1055
Automatisches Anpassen der Spaltenbreiten nach dem Hinzufügen aller Elemente	1056
Zuordnen von ListViewItems und Objekten, die sie repräsentieren	1056
Feststellen, dass ein Element der Liste selektiert wurde.....	1057
Selektieren von Elementen in einem ListView-Steuerelement	1058
Verwalten von Daten aus Auflistungen mit dem DataGridView-Steuerelement	1059
Über Datenbindung, Bindungsquellen, die BindingSource-Komponente	
und warum Currency nicht unbedingt Währung heißt	1060
Erstellen einer gebundenen DataGridView mit dem Visual Basic-Designer	1062

Sortieren und Benennen der Spalten eines DataGridView-Steuerelements	1067
Programmtechnisches Abrufen der aktuellen Zeile und aktuellen Spalte.....	1072
Formatieren von Zellen.....	1073
Problemlösung für das Parsen eines Zellennwerts mit einem komplexen Anzeigeformat	1073
Verhindern des Editierens bestimmter Spalten aber Gestatten Ihrer Neueingabe	1075
Überprüfen der Richtigkeit von Eingaben auf Zellen- und Zeilenebene.....	1076
Ändern des Standardpositionierungsverhaltens des Zellencursors nach dem Editieren einer Zelle.....	1079
Entwickeln von MDI-Anwendungen	1081
Vererben von Formularen	1089
Ein erster Blick auf den Designer-Code eines Formulars	1093
Modifizierer von Steuerelementen in geerbten Formularen	1095
38 Im Motorraum von Formularen und Steuerelementen.....	1097
Über das Vererben von Form und die Geheimnisse des Designer-Codes	1098
Geburt und Tod eines Formulars – New und Dispose.....	1100
Ereignisbehandlung von Formularen und Steuerelementen.....	1104
Vom Programmstart mithilfe des Anwendungsframeworks über eine Benutzeraktion zur Ereignisauslösung.....	1105
Implementieren neuer Steuerelementereignisse auf Basis von Warteschlangenauswertungen.....	1109
Wer oder was löst welche Formular- bzw. Steuerelementereignisse wann aus?	1110
Kategorie Erstellen und Zerstören des Formulars.....	1114
Kategorie Mausereignisse des Formulars	1116
Kategorie Tastaturereignisse des Formulars	1118
Kategorie Position und Größe des Formulars	1119
Kategorie Anordnen der Komponenten und Neuzeichnen des Formulars	1120
Kategorie Fokussierung des Formulars.....	1121
Kategorie Tastaturvorverarbeitungsnachrichten des Formulars	1122
Kategorie Erstellen/Zerstören des Controls (des Steuerelements)	1124
Kategorie Mausereignisse des Controls	1124
Kategorie Tastaturereignisse des Controls	1127
Kategorie Größe und Position des Controls	1128
Kategorie Neuzeichnen des Controls und Anordnen untergeordneter Komponenten..	1129
Kategorie Tastaturnachrichtenvorverarbeitung des Controls.....	1131
Die Steuerungsroutinen des Beispielprogramms	1132
Anmerkungen zum Beispielprogramm	1137
39 Individuelles Gestalten von Elementinhalten mit GDI+	1139
Einführung in GDI+	1141
Linien, Flächen, Pens und Brushes	1144
Angabe von Koordinaten	1144
Wieso Integer- <i>und</i> Fließkommaangaben für Positionen und Ausmaße?	1145

Wie viel Platz habe ich zum Zeichnen?	1145
Das gute, alte Testbild und GDI+ im Einsatz sehen!.....	1146
Flimmerfreie, fehlerfreie und schnelle Darstellungen von GDI+-Zeichnungen	1155
Zeichnen ohne Flimmern	1155
Programmtechnisches Bestimmen der Formulargröße.....	1159
Was Sie beim Zeichnen von breiten Linienzügen beachten sollten.....	1161
 40 Entwickeln von Steuerelementen für Windows Forms	1169
Neue Steuerelemente auf Basis vorhandener Steuerelemente implementieren	1171
Der Weg eines Steuerelements vom Klassencode in die Toolbox.....	1173
Implementierung von Funktionslogik und Eigenschaften.....	1175
Steuern von Eigenschaften im Eigenschaftenfenster.....	1177
Entwickeln von konstituierenden Steuerelementen.....	1180
Anlegen eines Projekts für die Erstellung einer eigenen Steuerelement-Assembly.....	1181
Initialisieren des Steuerelements	1186
Methoden und Ereignisse delegieren.....	1188
Implementieren der Funktionslogik	1189
Implementierung der Eigenschaften.....	1191
Erstellen von Steuerelementen von Grund auf.....	1192
Ein Label, das endlich alles kann	1193
Vorüberlegungen und Grundlagenerarbeitung	1194
Klasseninitialisierungen und Einrichten der Windows-Darstellungsstile des Steuerelements.....	1195
Zeichnen des Steuerelements	1197
Größenbeeinflussung durch andere Eigenschaften	1200
Implementierung der Blink-Funktionalität	1203
Designercode-Generierung und Zurücksetzen von werteerbenden Eigenschaften mit ShouldSerializeXXX und ResetXXX	1205
Designer-Reglementierungen	1207
 Teil H – Entwickeln von WPF-Anwendungen	1211
 41 Wichtige Steuerelemente zum Aufbau von WPF-Anwendungen.....	1213
Einführung.....	1214
Weitergeleitete Ereignisse (Routed Events).....	1215
Weitergeleitete Befehle (Routed Commands).....	1224
Eigenschaften der Abhängigkeiten	1228
Eingaben.....	1231
Schaltflächen.....	1233
Bildlaufleisten und Schiebereglер	1235
Steuerelemente für die Texteingabe.....	1239
Das Label-Element.....	1243
Menüs	1244
Werkzeugeleisten (Toolbars)	1249
Zusammenfassung	1252

42	Layout.....	1253
	Das StackPanel.....	1254
	Das DockPanel.....	1256
	Das Grid.....	1260
	Das GridSplitter-Element.....	1266
	Das UniformGrid	1269
	Das Canvas-Element.....	1270
	Das Viewbox-Element	1271
	Text-Layout	1273
	Das WrapPanel.....	1277
	Standard-Layout-Eigenschaften.....	1278
	Width- und Height-Eigenschaft	1278
	MinWidth-, MaxWidth-, MinHeight- und MaxHeight-Eigenschaft.....	1278
	HorizontalAlignment- und VerticalAlignment-Eigenschaft.....	1279
	Margin-Eigenschaft.....	1279
	Padding-Eigenschaft	1280
	Zusammenfassung.....	1281
43	Grafische Grundelemente	1283
	Grundlagen	1284
	Die Grafik-Auflösung	1291
	Die grafischen Grundelemente	1293
	Rechteck und Ellipse.....	1294
	Einfache Transformationen.....	1296
	Die Linie	1297
	Die Polylinie	1298
	Das Path-Element.....	1302
	Hit-Testing mit dem Path-Element.....	1304
	Zusammenfassung.....	1307
	Teil I – Anwendungsausführung parallelisieren	1309
44	Einführung in die Technik des Threading.....	1311
45	Threading-Techniken	1315
	Gleichzeitige Codeausführung mithilfe eines Thread-Objektes	1316
	Starten von Threads	1318
	Grundsätzliches über Threads.....	1318
	Synchronisieren von Threads	1319
	Synchronisieren der Codeausführung mit SyncLock	1319
	Mehr Flexibilität in kritischen Abschnitten mit der Monitor-Klasse	1322
	Synchronisieren von beschränkten Ressourcen mit Mutex.....	1326
	Weitere Synchronisierungsmechanismen	1328

Verwenden von Steuerelementen in Threads	1331
Managen von Threads.....	1333
Starten eines Threads mit Start.....	1333
Vorübergehendes Aussetzen eines Threads mit Sleep – Statusänderungen im Framework bei Suspend und Resume.....	1333
Abbrechen und Beenden eines Threads.....	1334
Datenaustausch zwischen Threads durch Kapseln von Threads in Klassen.....	1338
Der Einsatz von Thread-Klassen in der Praxis.....	1341
Verwenden des Thread-Pools	1349
Thread-sichere Formulare in Klassen kapseln.....	1354
Threads durch den Background-Worker initiieren	1357
Threads durch asynchrone Aufrufe von Delegaten initiieren	1360
Stichwortverzeichnis	1363

Vorwort

Die Familie der BASIC-Dialekte hat sich über die Zeit stark gewandelt. Doch seit Bill Gates BASIC für den Altair entworfen hat, ist das Ziel aller BASIC-Dialekte stets dasselbe geblieben, nämlich State-of-the-Art-Entwicklungen für Jedermann zu ermöglichen. Heutzutage gibt Ihnen Visual Basic .NET die Stärke einer erstklassigen Programmiersprache und hilft Ihnen, produktiv bei Ihren Entwicklungsaufgaben zu sein. Ich verstehe mich als großer Anwalt dieser Sprache, und sehe mit Begeisterung der vor Ihnen liegenden Reise entgegen, auf der Sie mithilfe dieses Buchs Ihre Visual Basic-Expertise ausbauen.

Seit 4 Jahren arbeite ich nun für das Microsoft Visual Basic Team. Ich begann meine Arbeit als Program Manager der Visual Basic-Entwicklungsumgebung und habe mich vom ersten Tag an in deren Funktionsvielfalt verliebt. Von allen Features, die diese bietet, ist mir IntelliSense nach wie vor das liebste. Als ich meinen Job begann, ging ich völlig in der Herausforderung auf, wie man sich der Entwicklungsumgebung noch einfacher nähern und sie produktiver gestalten könnte. Eine der wichtigsten Innovationen im Visual Basic 2008-Release war LINQ (Language INtegrated Query). Ein Großteil meiner Arbeit bestand darin, LINQ durch IntelliSense, Tooltips, Farbgebung, Formatierungen und weitere Elemente einfach anwendbar zu machen. Es machte ungeheueren Spaß, diese Features zu entwickeln, und ich hoffe, dass es Ihnen Spaß machen wird, sie zu verwenden, wenn Sie sich durch die LINQ-Kapitel dieses Buchs arbeiten. Vergessen Sie dabei nicht die XML-Abschnitte, die Ihnen zeigen, dass Visual Basic 9 die beste Sprache ist, mit XML zu entwickeln.

Ich habe Klaus durch eine interessante E-Mail-Diskussion über Schlüsselwort-Entscheidungen des Teams in der Visual Basic Sprache kennengelernt. Klaus hat ein scharfes Auge fürs Detail und ein solides Wissen der Sprache und der Entwicklungsumgebung. Er verfügt darüber hinaus über einen unverwechselbaren Schreibstil, der sowohl unterhält und Sie zusätzlich zum Weiterlesen anregt. Dieses Buch verwendet einen großartigen, ganzheitlichen Ansatz, Ihnen die verschiedenen Aspekte des Programmierens in Visual Basic näher zu bringen, und dazu gehören die Entwicklungsumgebung, Werkzeuge und natürlich die Sprache gleichermaßen. Beim Programmieren in Visual Basic geht es um genau diese Gesamterfahrung. Ganz gleich, ob Sie Neueinsteiger oder schon erfahren im Umgang mit den verschiedensten Versionen von Visual Basic .NET sind – dieses Buch bietet für Jeden etwas.

*Lisa Feigenbaum
Program Manager
Microsoft Visual Basic*

Englischer Originaltext

The Basic family of languages has evolved over time. However, since Bill Gates designed Basic to program against the Altair, the goal of these languages has stayed the same: to make state-of-the-art programming approachable. Visual Basic .NET today provides the power of a first class general purpose programming language, and helps you be productive in your development tasks. I am a huge advocate of the language, and am excited for the journey that lies ahead as you grow your Visual Basic expertise with this book.

I've been on the Microsoft Visual Basic team for over 4 years. I first joined as the VB IDE Program Manager and immediately fell in love with the feature set. Of all the IDE features, IntelliSense continues to be my favorite. As soon as I started, I immediately became consumed with the challenge of how to improve the IDE to make Visual Basic programming even more approachable and productive. One of the most important innovations in the Visual Studio 2008 release was LINQ (Language INtegrated Query). Much of my work in this release was in making LINQ easy to use, through IntelliSense, tool tips, colorization, formatting, and more. It was a lot of fun to design these features, and I hope you enjoy using them as you work through the LINQ sections of this book. In addition, make sure to explore the XML section, where you will see how Visual Basic 9 is the best language for programming with XML.

I first met Klaus through an interesting email discussion about keyword choices in the Visual Basic Language. Klaus has a keen eye to details and a solid understanding of the language and IDE. He also has a distinctive way of writing that is both entertaining, and intrigues you to read on. This book takes a great, holistic approach to various aspects of programming Visual Basic, including the IDE, tooling, and of course the language. Programming in Visual Basic is about that entire experience. Whether you are new to programming, or proficient with multiple versions of Visual Basic .NET, this book has got something for you.

*Lisa Feigenbaum
Program Manager
Microsoft Visual Basic*

Einleitung

Ich erinnere mich noch genau an meine erste Begegnung mit einem Computer. Es war im Physikraum meiner alten Schule, dem Gymnasium Schloß Overhagen, und mein damaliger Mathelehrer Karsten Richter hatte sich inmitten der 80er Jahre gegen jeden Widerstand durchgesetzt und es tatsächlich geschafft, einen TI/994a von Texas Instruments mit allem notwendigen Zubehör zu beschaffen. Deswegen konnte das Schloß Overhagen als eine der ersten Schulen der Stadt Lippstadt einen Oberstufenkurs anbieten, bei dem viele Lehrer noch nicht einmal *ungefähr* wussten, welche Inhalte er vermitteln sollte: Informatik. Karsten Richter hingegen wusste genau, was er tat. Mit einer Mischung aus Ehrfurcht und Begeisterung hingen wir an Karstens Lippen, während er sich redlich bemühte, uns in dessen unglaublich behäbige und holprige aber deswegen nicht minder interessante Programmiersprache einzuführen. Die Rede war von BASIC. Meine Begeisterung war dermaßen groß, dass meine Eltern sich von meinem ständigen Generve mit der Bitte nach einem eigenen Computer mit einem eben solchen freikaufen. Das Bemerkenswerte dabei: Das BASIC meines ersten Computers war nicht nur von Microsoft, es kannte auch schon so moderne Techniken wie dynamische Programmierelemente und einen durch einen Garbage Collector kontrollierten »Objektspeicher«, bei dem man sich nicht um das Aufräumen nicht mehr benötigter Variablen kümmern musste. Es war ein Commodore 64. Das ist jetzt 23 Jahre her, und in BASIC Software zu entwickeln übte vom ersten Moment an einen solch immensen Reiz aus, dass ich ein gutes Jahr später mit zweien meiner Mitschüler an einem ersten Buch mitarbeitete – dem Grafikbuch zum Commodore 128. Und auch hier ging es wieder überwiegend um eines: um BASIC. Seitdem ist viel Wasser die Lippe heruntergeflossen, und Computersysteme wie BASIC-Dialekte kamen und gingen. Atari ST mit GfA- sowie Omikron-BASIC, ein 8086 mit GW-Basic (*Ghee Wiz!*, etwa: *Potz Blitz!*), der erste 286er mit QuickBasic, und dann, auf meinen 386SX war ich stolz wie Oskar, als ich mein allererstes »Hello World« unter Windows 3.0 verfassen durfte. Und womit schrieb ich es? – Mit *Visual Basic*, damals in seiner ersten Version, die ganze drei Disketten zur Installation benötigte. Es folgten in den darauffolgenden Jahren die Versionen 2 bis 6, zu 1, 3 und 6 ließ man mich wieder Bücher schreiben. Dann kam mit Visual Studio 2002 der großartige Sprung in die .NET-Welt, es folgten 2003 und 2005 auch mit den entsprechenden Büchern dazu, und damit sind wir in der Neuzeit angekommen. Nach einer dermaßen langen Zeit könnte man meinen, es wäre irgendwann langweilig geworden; ist es aber nicht. Microsoft hat es immer wieder geschafft, mit neuen Technologien Nachwuchs für den Beruf des Softwareentwicklers zu gewinnen und uns inzwischen alte Hasen bei der Stange zu halten. Und auch mit Visual Basic 2008 ist meine Begeisterung ungebrochen. Im Gegenteil: Wir ITler sind ja im Gegensatz zu früher sogar gesellschaftsfähig geworden ;-) – die Motivation, täglich neu auf Entdeckungsreise zu gehen ist höher denn je, und das jüngste Ergebnis liegt vor Ihnen, über 1.400 Seiten gefüllt mit, so hoffe ich, nützlichem Wissen, Tipps und Tricks rund um Visual Basic 2008, dem .NET Framework und ein kleines bisschen auch SQL Server 2008. Wie sagte Lisa in ihrem Vowort so treffend? *Heutzutage gibt Ihnen Visual Basic .NET die Stärke einer erstklassigen Programmiersprache und hilft Ihnen, produktiv bei Ihren Entwicklungsaufgaben zu sein.* Ich hoffe, dass Ihnen dieses Buch ganz aktiv dabei helfen kann!

Danksagungen

Die Ankündigung, diese Seitengrenze zu durchbrechen, hat mir in meinem nicht vollständig IT-dominierten Bekanntenkreis (»Teufelszeug!« – *Zitat Momo W.*) bereits einiges an Reaktionen beschert, von denen ein langgezogenes »Oh-Kay« in Kombination mit einem sich von mir weg drehenden Blick noch eine der unspektakulärsten war. Und ich gebe zu: 1.400 Seiten sind schon grenzwertig, in jeder Beziehung. Es ist das, was man mit Word so gerade noch textverarbeiten kann, ohne die Übersicht zu verlieren; es ist das, was noch ohne größere Probleme als Buch verarbeitet und gebunden werden kann; es ist das, was man noch so gerade in die Badewanne mitnehmen kann, ohne einen Badezimmersunami zu riskieren. Es ist aber auch das, in dem Sie sehr viele Infos zum Thema finden und eine riesige Sammlung an hoffentlich gutem Insiderwissen, die ich ohne unser ActiveDevelop-Team und ohne meine Freunde nie hinbekommen hätte. Mehr Engagement von seinem eigenen Team und seinen Freunden kann man sich nicht wünschen! Erschwerend kam hinzu, dass wir just in dem Moment, in dem das vor Ihnen liegende Buch endlich zur Produktion anstand, auch die beiden ASP.NET-Bücher unseres Kollegen Holger Schwichtenberg in Produktion hatten, dessen umfassendes und fundiertes Wissen stets dazu führt, dass seine Bücher umfangstechnisch die Grenze der Badewannentauglichkeit erreichen. Aber auch diese beiden, weit über 1000 Seiten umfassenden Bücher haben wir in dem Zeitrahmen, in dem es sie zu produzieren galt, rechtzeitig hinbekommen und das in erster Linie durch den aufopfernden Einsatz unserer Setzerinnen Silja Brands und Uta Berghoff. Deswegen verspreche ich hier vor Zeugen noch für Dezember 2008 allen Beteiligten eine große Kick-Off-SingStar-Party, und möchte mich damit ganz ganz herzlich bei ihnen bedanken.

Silja Brands und Uta Berghoff waren es also, die teils bis in die späten Abendstunden und mehrere komplette Wochenenden im Büro harrten, den typografischen Satz durchführten, alle Mitschaffenden koordinierten und Korrekturen einpflegten, während ihre Lebenspartner sich zuhause langweilen mussten und den dafür Verantwortlichen verfluchten. Ihnen gilt mein besonderer Dank.

Bedanken möchte ich mich auch bei Jürgen Heckhuis und Uwe Thiemann, die sich aufopfernd um die Portierung der vorhandenen Beispiele und das Neuerstellen der Grafiken unter Visual Studio 2008 und Windows Vista gekümmert haben. Dass es die vielen zusätzlichen Infos in Form von IntelliLinks rechtzeitig und in der großen Fülle noch ins Buch geschafft haben, ist nicht zuletzt Jürgens Verdienst. Bei unserem Chef-Entwickler Andreas Belke möchte ich mich auch bedanken. Er hat mit mir noch zu manch später Stunde vor dem Bildschirm ausgehalten, und die vielen Beispiele für LINQ to SQL/Entities/XML mit erarbeitet, kritisch begutachtet und debuggt.

Unsere beiden Korrekturleserinnen Kristin Grauthoff und Silvia Fehlow mussten in den vergangenen Wochen auch hart ran. Sie lasen nicht weniger als 3000 Seiten, und sorgten dafür, dass unser manchmal holpriges Deutsch und unsere streckenweise recht kreative Grammatik allgemeinverständliche Züge annehmen konnten.

Bedanken möchte ich mich auch bei meinem Lektor bei Microsoft Press, Thomas Braun-Wiesholler. Ohne seinen Einsatz, mir benötigte Software und Infos zum Thema zu verschaffen, sowie Ansprechpartner zu vermitteln, die ich im Bedarfsfall um Rat fragen konnte, hätte ich das Ziel so nicht erreicht.

Mein Kumpel Angela Wördehoff war in der Zeit des Buchschreibens Leidensgenossin, denn auch sie schrieb zu dieser Zeit, nämlich ihre Diplomarbeit. Dank ihr weiß ich jetzt nicht nur über Krankenkassensysteme der ganzen Welt Bescheid, sondern auch, dass sie es immer schaffen wird, mich auf ihre ganz unverwechselbare Art dazu zu bringen, alles zu geben und durchzuhalten. Vielen, vielen Dank dafür!

Mit seinem umfassenden Wissen stand mir auch mein guter Freund Ruprecht Dröge als Fachlektor dieses Buchs und als mein persönlicher Motivationscoach stets zur Seite, der sich, obwohl durch eine böse, schmerzhafte Verletzung gehandicapt, dennoch immer wieder aufraffte, die vielen Seiten dieses Buchs zu lesen, und mir mit wertvollen Tipps und Hinweisen hilfreich zur Seite stand.

Und schließlich geht ein dickes Dankeschön an meine Eltern. Ohne sie wär ich schließlich nicht hier. Mein Vater ist der beste Organisator und Krisenmanager und meine Mutter die beste REFA-Software-Vertrieblerin dieser Welt!

Support

Es wurden alle Anstrengungen unternommen, um die Korrektheit dieses Buchs und des Begleitinhalts sicherzustellen.

Microsoft Press stellt unter der folgenden Internetadresse eventuell notwendige Korrekturen zu veröffentlichten Büchern zur Verfügung:

<http://www.microsoft-press.de/support.asp>

Sollten Sie Anmerkungen, Fragen oder Ideen zu diesem Buch haben, senden Sie diese bitte an eine der folgenden Adressen von Microsoft Press:

Postanschrift:

Microsoft Press

Betreift: Visual Basic 2008 – Das Entwicklerbuch

Konrad-Zuse-Straße 1

85716 Unterschleißheim

E-Mail:

presscd@microsoft.com

Beachten Sie bitte, dass unter der oben angegebenen Adresse kein Produktsupport geleistet wird. Supportinformationen zum .NET Framework, zu Visual Basic .NET oder Visual Studio finden Sie auf der Microsoft-Produktsupportseite unter:

<http://www.microsoft.com/germany/support>

Teil A

Der Einstieg in Sprache und Entwicklungswerkzeuge

In diesem Teil:

Einführung	3
»Symbolischer Allzweckbefehlscode für Anfänger«	13
Einführung in das .NET Framework	45
Der Schnelleinstieg in die Bedienung der Visual Studio-Entwicklungsumgebung (IDE)	61
Einführung in Windows Forms – Designer und Codeeditor am Beispiel	99
Einführung in Windows Presentation Foundation	171
Die essentiellen .NET-Datentypen	207
Tipps & Tricks für das angenehme Entwickeln zuhause und unterwegs	267

Kapitel 1

Einführung

In diesem Kapitel:

Welches Vorwissen verlangt dieses Buch?	4
Wissenswertes zur Installation von Visual Studio 2008	7
Der Umgang mit Web-Links in diesem Buch – http://www.activedevelop.de/vb2008	10
Die Begleitdateien zum Buch und das E-Book	11
Nützliches zu Visual Basic 2008 – www.codeclips.de	11

An der berühmten Eier legenden Wollmilchsau haben sich Erfinder und Tüftler der unterschiedlichsten Fachgebiete schon immer die Zähne ausgebissen – es jedem und allen recht zu machen, ist einfach ein Ding der Unmöglichkeit. Ich denke, dass das gleichermaßen für Fachbuchautoren (nicht nur) zum Thema Visual Basic gilt, jedenfalls muss ich das denken, wenn ich mir Amazon-Rezensionen zu verschiedenen .NET-Büchern zu Gemüte führe. Insbesondere die Kritiken zu meinem eigenen Buch zeigen mir, dass viele Leser genau das gut finden, was andere kritisieren – und bei einer solchen Konstellation ist natürlich guter Rat teuer.

Doch nicht nur aufgrund von Amazon- oder Buch.de-Kritiken, sondern auch durch viele Leserbriefe, derer ich erstaunlich viele als Reaktion auf das letzte 2005er Visual Basic Buch bekam, merke ich, dass Eines unisono bemängelt wurde: Der Übergang in Sachen »Schwierigkeitsgrad der Themen«, also von halbe Kraft auf volle Kraft voraus war vielen wohl zu abrupt – viele Leser des vorherigen Buchs vermissten einfach einen schwierigkeitssteigernden Teil, der die Brücke zwischen erstem einfachen Beispiel und der schon sehr viel anspruchsvolleren objektorientierten Programmierung schlug.

Aus diesem Grund gibt es in der Neufassung nun einen erweiterten Einführungsteil, in dem ich mir große Mühe geben werde, mehr sowohl auf die Sprachgrundlagen von BASIC an sich als auch des Visual Basic-Dialektes, aber auch ausreichend auf die Werkzeuge der IDE einzugehen. Und da wir gerade bei Amazon-Kritiken sind: natürlich immer in dem blumigen Stil, den Sie von mir gewohnt sind ... ;-)

Welches Vorwissen verlangt dieses Buch?

Im Gegensatz zum Vorgänger dieses Buchs sind die Anforderungen, die dieses Buch an Sie, den Leser, stellt, viel geringer geworden. Warum? Ganz einfach: Während das Vorgängerbuch sich an Softwareentwickler gerichtet hatte, die bereits fundierte Programmiererfahrung mit Basic-Derivaten oder anderen Programmiersprachen hatten, setzt dieses Werk ein paar Level tiefer an.

Dennoch wird dieses Buch eher nicht in der Lage sein, ein regelrechtes Einsteigerbuch zu ersetzen. Zwar werden Sie auf den nächsten Seiten auch Grundlegendes zur Sprache BASIC finden, doch werden Sie feststellen, dass das alleine schon aus Platzgründen in einem sehr kompakten und gerafften Stil erfolgt. Schaden wird es also überhaupt nicht, schon Erfahrungen beispielsweise in der Skriptprogrammierung oder mit der Programmierung von Makros wie z.B. mit Word oder Excel zu haben. Und dann wird Ihnen das Studium der folgenden Abschnitte sicherlich leicht fallen und auch dafür sorgen, dass Sie sich vielleicht vor Jahren Gelerntes noch einmal in Ihre Erinnerung zurückrufen können und Sie bei der Lektüre der einzelnen Abschnitte über die Sprachkonstrukte Aha-Effekte im Sinne von »Ach ja, so ging das nochmal!« erleben.

Falls Sie sich allerdings noch nie in ihrem Leben sich mit dem Thema Softwareentwicklung beziehungsweise Programmierung auseinander gesetzt haben, so könnte die vorherige Lektüre eines entsprechenden Einsteigerbuchs unter Umständen von Vorteil oder vielleicht sogar notwendig sein. Der Fokus dieses Buchs liegt nämlich nicht in erster Linie auf dem Erlernen der Sprache Basic selbst. Bei diesem Buch dreht es sich vor allem um das Kennenlernen von Visual Basic 2008 und dem .NET Framework.

Welche Softwarevoraussetzungen benötigen Sie?

Um *alle* Beispiele in diesem Buch nachzuvollziehen, benötigen Sie eine aktuelle Version von Visual Studio 2008 in der Professional Edition und zwar mit dem Service Pack 1. Und es wird Sie freuen zu hören, dass Sie diese Version von Microsoft direkt, und zwar unter dem IntelliLink **A0101** beziehen können. Auf der beiliegenden DVD finden Sie übrigens die Express-Editionen der 2008er .NET-Programmiersprachen und des SQL Server 2008 Express Edition direkt zum Einsatz bereit – 90% der Beispiele in diesem Buch können Sie damit nachvollziehen, und obendrein können Sie diese Versionen unbegrenzt und kostenlos verwenden.

90-Tage-Version bedeutet übrigens, dass Sie diese Version von Visual Studio immerhin 3 Monate kostenlos und *ohne Einschränkungen* verwenden können. Nur: Nach Ablauf dieser Probefrist müssen Sie sich entscheiden, wie es weitergeht. Empfehlen kann ich den Lesern unter Ihnen, die auch nach Ablauf dieser Probefrist weiterhin professionell mit Visual Studio 2008 arbeiten möchten, auf jeden Fall den Kauf der Professional Edition zumindest aber der Standardversion von Visual Studio 2008 in Erwägung zu ziehen. Infos zum Kauf und zur Preisgestaltung von Visual Studio Professional finden Sie beispielsweise unter dem IntelliLink **A0102**. Mehr zu IntelliLinks erfahren Sie im Abschnitt »Der Umgang mit Web-Links in diesem Buch – <http://www.activedevelop.de/vb2008>« ab Seite 10.

Die preiswerte Alternative: die Express Editions

Doch es gibt auch eine sehr viel preiswertere Alternative, die wahrscheinlich die bessere für die Hobbyisten oder die Gelegenheitsentwickler unter Ihnen darstellt. Microsoft bietet nämlich mit den so genannten Express Editions abgespeckte Versionen von Visual Studio an, die jeweils nur eine Programmiersprache beinhalten. So gibt es beispielsweise die C# Express Edition, die C++ Express Edition und natürlich auch die Visual Basic Express Edition – alle Versionen befinden sich auf der beiliegenden Buch-DVD.

Aber um es noch einmal zu wiederholen: Deutlich empfehlenswerter für eine professionelle Softwareentwicklung ist *Visual Studio 2008 Professional*, und deswegen sollten Sie auf lange Zeit gesehen dessen Einsatz in Erwägung ziehen. Die Benutzeroberfläche von Visual Studio fördert Ihre Produktivität ungemein, und es wäre wie »Porsche fahren mit angezogener Handbremse«, wenn Sie nur mit dem Framework SDK oder der Express Edition von Visual Basic 2008 größere Entwicklungen durchführen wollten.

Für wirkliche Profis – Das MSDN-Abo

Professionell Software entwickeln zu können, bedeutet nicht nur, das bestmögliche Entwicklungswerkzeug einzusetzen, denn das macht nur einen Teil des Entwicklungsgeschehens aus. Sie müssen natürlich auch dafür Sorge tragen, dass Ihre Anwendungen auf den verschiedenen Betriebssystemen laufen und mit anderen Software-Installationen zusammenarbeiten. Genau dazu hat Microsoft das *Microsoft Developer Network-Abonnement (MSDN-Subscription)* geschaffen: Gegen einen bestimmten Betrag, der sich nach Ausbaustufe und Laufzeit richtet, erhalten Sie ein, zwei oder sogar drei Jahre lang, neben einem einmaligen Grundstock des Status Quo aller Betriebssysteme, Server und Softwarepakete auf DVD, monatliche Aktualisierungen in Form von DVDs, die Installationspakete der jeweils neuen oder aktualisierten Softwareprodukte aus dem Hause Microsoft enthalten, und die Sie im Rahmen von Entwicklungsprojekten nutzen können.

Nur so können Sie sicherstellen, dass eine Software auch in allen möglichen Umgebungskonstellationen zuverlässig arbeiten wird. Obendrein haben Sie die Möglichkeit, die entsprechenden Images und benötigten Installations-Keys direkt von der MSDN-Seite herunterzuladen, wie in der folgenden Abbildung zu sehen.

The screenshot shows a Microsoft Internet Explorer window displaying the MSDN Subscriptions website. The URL in the address bar is <https://msdn.microsoft.com/en-us/subscriptions/securedownloads/default.aspx>. The page title is "Download - Home page - Windows Internet Explorer". The main content area is titled "Downloads" and lists various Microsoft products available for download. The products listed include:

- MSDN Library Express for Visual Studio 2008 Service Pack 1 (x86, x64 WoW) - EXE (German)
- MSDN Library for Visual Studio 2008 (x86 and x64 WoW) - DVD (German)
- MSDN Library for Visual Studio 2008 Service Pack 1 (x86) - DVD (German)
- Visual Studio 2008 Express Editions (x86 and x64 WoW) - DVD (German)
- Visual Studio 2008 Express Editions with Service Pack 1 (x86, x64 WoW) - DVD (German)
- Visual Studio 2008 Professional Edition (x86 and x64 WoW) - DVD (German)
- Visual Studio 2008 Service Pack 1 (x86, x64 WoW) - DVD (German)
- Visual Studio Team System 2008 Architecture Edition (x86 and x64 WoW) - DVD (German)
- Visual Studio Team System 2008 Database Edition (x86 and x64 WoW) - DVD (German)
- Visual Studio Team System 2008 Development Edition (x86 and x64 WoW) - DVD (German)
- Visual Studio Team System 2008 Team Foundation Server Service Pack 1

 The left sidebar shows a navigation tree with categories like Applications, Business Solutions, Designer Tools, Developer Tools, and Visual Studio.

Abbildung 1.1 Mit Hilfe eines MSDN-Abos können Sie sich die für wichtigen Images für fast alle Microsoft Produkte direkt vom MSDN-Server herunterladen und für Entwicklungszwecke nutzen.

Grundsätzliches zum Thema *MSDN-Abos* erfahren Sie unter dem IntelliLink **A0103**. Mehr über MSDN-Abos in Verbindung mit Visual Studio 2008 finden Sie unter dem IntelliLink **A0104**.

Wissenswertes zur Installation von Visual Studio 2008

Eigentlich müssen Sie nichts Außergewöhnliches beachten, wenn Sie Visual Studio auf Ihrem Computer zum Laufen bringen wollen.¹ Die folgende Tabelle zeigt Ihnen die ideale Hardwarevoraussetzung, die ein reibungsloses Arbeiten mit Visual Studio 2008 ermöglicht.

WICHTIG Orientieren Sie sich bei dieser Liste lieber nicht an der angegebenen Mindestkonfiguration nach Microsoft-Spezifikation, die eher humoristischen Anforderungen genügt, sondern besser an der »Wohlfühl-Rundum-Sorglos«-Konfiguration, mit der das Arbeiten Spaß macht, und die nicht wegen langer Wartezeiten zu ständiger Nervosität, Hypertonie, Tachykardie oder Schlaflosigkeit auf Grund zu hohen Kaffeekonsums führt. Ein paar Euro in ein wenig Hardware investiert, können Ihre Turn-Around-Zeiten beim Entwickeln drastisch verbessern!

Denken Sie auch daran, dass es sich bei vielen der heutigen Prozessoren, die als Standardkonfiguration von Computern verbaut werden, automatisch um so genannte Dual-Core-Prozessoren (bzw. sogar Quad-Core-Prozessoren) handelt. Solche Prozessoren vereinen zwei (bei Quad-Core-Prozessoren sogar vier) Prozessorkerne unter einem Dach. Ganz einfach ausgedrückt bedeutet das: befindet sich ein Dual-Core-Prozessor in Ihrem Computer, dann verfügt Ihr Computer quasi über *zwei* völlig unabhängig voneinander arbeitende Prozessoren. Und bei Quad-Core-Prozessoren sind das sogar dann vier an der Zahl. Nun gibt es für die Ausstattung von Entwicklungsrechnern folgendes in Erwägung zu ziehen: Bei der Entwicklung von so genannten Multi-Threading-Anwendungen – das sind Anwendungen, bei denen verschiedene Programmteile quasi gleichzeitig ablaufen können – gibt es auf Single-Core-Prozessoren unter Umständen völlig andere Verhaltensweisen dieser parallel laufenden Programmteile als auf Dual- oder Quad-Core-Prozessoren, auf denen Programme ja tatsächlich gleichzeitig laufen können, da sie eben von zwei unabhängigen Prozessorkernen ausgeführt werden. Dem gegenüber stehen Single-Core-Computer, auf denen Multi-Threading-Anwendungen nur scheinbar gleichzeitig laufen – hier wird zwei scheinbar gleichzeitig laufenden Multi-Threading-Programmteilen Prozessorzeit im ständigen Wechsel zur Verfügung gestellt. Sie können auf einem Single-Core-Prozessor also unter Umständen gar nicht alle unterschiedlichen Aspekte einer Multi-Threading-Anwendung erfassen, nachstellen und testen. Deswegen ist es auf jeden Fall empfehlenswert, dass Ihr Entwicklungsrechner ebenfalls über mindestens zwei unabhängige arbeitende Kerne verfügt, damit die Anwendungen, die Sie entwickeln, auch später für Multi-Core-Prozessoren gerüstet sind.

Komponente	Mindestkonfiguration laut Microsoft	Ideal-Konfiguration (erfahrungsgemäß)
Prozessor	1,4 GHz	Dual Core Prozessor mit ausreichend großem Second Level Cache, zum Beispiel Intel Core 2 Duo E6600, Core 2 Quad E6600 oder AMD Phenom X4 9950.
Ram	256 MByte (Windows XP SP2) bzw. 512 MByte (Windows Vista)	1 GByte unter Windows XP SP2 bzw. 2 GByte unter Windows Vista
Festplattengröße (incl. MSDN-Hilfe) – also freier Speicherplatz.	5,4 GByte	5,4 GByte
HINWEIS: Denken Sie daran, dass gerade bei Notebooks noch ausreichend freier Speicher für Systemdateien, insbesondere für den Suspend-Modus noch bis zu 6 GBytes freier Speicher verbleiben muss.		►

¹ ... und Sie zuvor noch keine Beta-Version, eine CTP-Version oder einen Release-Kandidaten von Visual Studio 2008 auf Ihrem Rechner installiert hatten. Sollte das jedoch der Fall gewesen sein, lesen Sie UNBEDINGT vorher den folgenden Abschnitt!

Komponente	Mindestkonfiguration laut Microsoft	Ideal-Konfiguration (erfahrungsgemäß)
DVD-Laufwerk	Wird benötigt	Wird benötigt
Video	800 x 600 256 Farben	1280 x 1024 True Color; oder mindestens Dual-Monitor-Support mit zwei Mal 1024 x 768 bei True Color. Für den Einsatz unter Windows Vista achten Sie bitte auf eine DirectX 9.0-fähige Grafikkarte mit mindestens 128 MB Grafikkartenspeicher! ²

Tabelle 1.1 Die Minimal- und Idealvoraussetzungen an die Hardware für ein reibungsloses Arbeiten mit Visual Studio 2008

Lauffähigkeit von Visual Studio und den erstellten Kompilaten unter den verschiedenen Betriebssystemen

Mit Visual Studio 2008 erstellte Programme laufen in jedem Fall unter Windows XP SP2, Windows Vista, Windows Server 2003 SP2 sowie Windows Server 2008. Diese Programme laufen *nicht* auf Windows 95 und Windows NT, und unter Windows 98 oder Windows Millennium steht Ihnen nicht der komplette Funktionsumfang des Framework 2.0 zur Verfügung (Framework 2.0-Programme laufen aber grundsätzlich unter diesen beiden älteren Betriebssystemen nach Installation der entsprechenden Service Packs entgegen vieler Meinungen schon). Sie benötigen allerdings *mindestens* eine der zahlreichen Windows XP-Editionen mit dem Service Pack 2, Windows 2003 Server (natürlich auch R2), Windows 2008 Server oder eine der Windows Vista-Versionen (aber nicht die Starter-Edition), um die Entwicklungsumgebung (entweder die kostenlose Express Edition oder eine der Visual Studio-Vollversionen) installieren zu können. Die Installation der Entwicklungsumgebung unter Windows 2000 ist leider nicht mehr möglich.

Framework Version Targeting (Versionsbestimmung des Frameworks für Kompilate)

Wichtig ist in diesem Zusammenhang zu wissen, dass Sie mit Visual Studio 2008 bestimmen können, mit welcher Framework-Version die von Ihnen entwickelte Anwendung funktionieren soll. Sie können also bestimmen, dass das Programm, das Sie gerade entwickeln, das Framework 2.0, das Framework 3.0 oder eben das Framework 3.5 verwenden soll (Auf Neudeutsch spricht man dabei vom so genannten *Framework Version Targeting*).

² Kein Budget für einen zweiten Bildschirm, aber Sie haben einen Laptop? Nutzen Sie Ihren Laptop als Zweitbildschirm. Mehr dazu gibt unter dem IntelliLink **A0110**.

HINWEIS Dabei ist wiederum wichtig zu wissen, dass Programme, die noch unter Windows 2000 SP4 laufen sollen, höchstens auf das Framework 2.0 abzielen können. Die Installation des Microsoft Framework 3.0 oder 3.5 ist unter Windows 2000 leider nicht mehr möglich (und das ist der eigentliche Grund, weswegen Sie für dieses Betriebssystem keine Programme mehr schreiben können, die auf diese Framework-Versionen abzielen).

Unter Windows XP hingegen lassen sich beide neuen Framework-Versionen installieren. Voraussetzung dafür ist allerdings, dass das Service Pack 2 für Windows XP zuvor eingerichtet wurde.

Und jetzt noch eine letzte Anmerkung in diesem Zusammenhang: Windows Vista sowie Windows 2008 Server haben sowohl das Framework 2.0 als auch das Framework 3.0 standardmäßig an Bord. Das Framework in der Version 3.5 muss hingegen auch auf diesen modernen Betriebssystemen individuell installiert werden.

Mehr zum Thema Framework-Targeting finden Sie in Kapitel 11 im gleichnamigen Abschnitt.

WICHTIG Sowohl für Visual Studio 2008 als auch für das .NET-Framework 3.5 steht zu Drucklegung dieses Buchs das erste ServicePack bereits bereit. Bitte denken Sie daran, dieses im Bedarfsfall zu installieren, und denken Sie auch an die richtige Laufzeitversion auf den Rechnern Ihrer Kunden (also die mit ServicePack 1), wenn Sie Ihre Software ausliefern. Die richtige Laufzeitversion des .NET-Framework, das das Service Pack 1 bereits enthält, finden Sie unter dem IntelliLink **A0105**. Das deutsche Sprachpaket zum SP1 finden Sie unter dem IntelliLink **A0106**.

Das ISO-Image für das Service Pack 1 von Visual Studio 2008 finden Sie unter dem IntelliLink **A0107**.

Parallelinstallation von Visual Studio 6, Visual Studio 2002, Visual Studio 2003, Visual Studio 2005 und Visual Studio 2008

Um es ganz kurz und prägnant zu sagen: Alle in der Überschrift genannten Versionen von Visual Studio können problemlos parallel nebeneinander installiert werden. Das gilt allerdings nur für Windows XP SP2 als Betriebssystemplattform, denn sowohl Visual Studio 2002 als auch Visual Studio 2003 lassen sich leider nicht unter Vista zum Laufen bringen. (Visual Studio 6 läuft hingegen mit Einschränkungen – wer hätte das gedacht! Deaktivieren Sie dazu unbedingt und schon vor der Installation die erweiterte Benutzerkontensteuerung von Windows Vista. Wichtiges zu VB6-Anwendungen unter Vista erfahren Sie auch in Kapitel 9.)

Die SQL-Datenbanken für Visual Studio 2008: SQL Server 2005 und SQL Server 2008

Mit den neuen Features LINQ to SQL bzw. LINQ to ADO.NET Entities (ab SP1 von Visual Studio bzw. Visual Basic 2008 verfügbar) wird die Zusammenarbeit mit SQL Server-Datenbanken nicht nur ganz erheblich vereinfacht, sondern aus Codesicht auch noch super elegant. Natürlich müssen Sie dazu entweder im Netzwerk oder auf derselben lokalen Maschine eine Instanz der zahlreichen SQL Server-Versionen verfügbar machen.

Grundsätzlich gibt es hier wieder die Unterscheidung zwischen Express und »erwachsenem« Produkt. Dazu soviel: Außer den Express Editions und der Workgroup Edition lässt sich SQL Server nur auf einem Server-Betriebssystem (also Windows Server 2003 oder Windows Server 2008) installieren. Einzige Ausnahme bildet die Enterprise-Edition: Sie gibt es in der Entwicklerversion auch zur Installation auf einem lokalen Entwicklerrechner mit XP oder Vista und heißt deswegen bezeichnenderweise SQL Server Developer

Edition. Sie entspricht komplett einer Enterprise-Edition, und ist, falls Sie sie nicht sowieso zusammen mit Visual Studio 2008 oder Ihrem MSDN-Abo bekommen haben, für kleines Geld bei Microsoft zu haben. Sie dürfen mit ihr aber, wie schon gesagt, nur entwickeln und sie nicht als Plattform für produktive Datenbankanwendungen verwenden.

Visual Studio 2008 installiert im Bedarfsfall SQL Server 2005 Express Edition mit dem Service Pack 2 mit, wenn Sie es wünschen. Zur Drucklegung dieses Buchs ist bereits SQL Server 2008 in der Express Edition verfügbar; diese Version wird übrigens, sofern Sie mögen, von den 2008er C#- bzw. VB-Express-Editionen (aber nur von denen mit SP1!) mitinstalliert.

Sie können SQL Server 2005 und SQL Server 2008-Instanzen bedenkenlos nebeneinander laufen lassen. Das gilt ebenfalls für die Express-Editionen. Während einer SQL Server 2008-Installation könnte es nur dann Probleme geben, wenn sich auf dem Zielcomputer bereits ein so genanntes Management Studio Express zur Administrierung einer SQL Server 2005 Express Version befindet. Doch dieses können Sie vor einer Installation auch problemlos deinstallieren, da Sie mit dem Management Studio von SQL Server 2008 natürlich auch SQL Server 2005-Instanzen managen können.

HINWEIS Das Management Studio Express für die SQL Server Express Edition gibt es seit der 2008er Version nicht mehr einzeln zum Herunterladen. Stattdessen verwenden Sie die Version *SQL Server 2008 Express Edition with Tools* oder *with Advanced Services*, die beide ein *Management Studio 2008 Basic* an Board haben. Übrigens:

TIPP Kapitel 34 hält eine Detailbeschreibung bereit, die Ihnen zeigt, wie Sie *SQL Server 2008 Express Edition with Tools* auf Ihrem Entwicklungsrechner zum Laufen bringen, und liefert viele weitere Details, wie beispielsweise dem Einrichten der *AdventureWorks*-Beispieldatenbank in der SQL 2008 Express-Instanz.

Der Umgang mit Web-Links in diesem Buch – <http://www.activedevelop.de/vb2008>

Papier ist geduldig, und im Gegensatz zu einem Computermonitor vielseitig einsetzbar. Leider macht Papier unter Umständen aber auch ungeduldig, nämlich dann, wenn es Web-Links als Information transportiert, denn es bietet keine *Ausschneide-* und *Einfügen*-Funktionalität (jedenfalls keine, die mit Computern kompatibel wäre), um einen Link einfach in die Adresszeile eines Internet-Browsers zu kopieren. Aus diesem Grund finden Sie in diesem Buch im Fließtext keine absoluten Web-Links sondern nur Referenzkennzahlen, die sich wiederum auf einer bestimmten Web-Seite auflösen lassen. Diese Web-Seite lautet:

<http://www.activedevelop.de/vb2008>

Sollte dieses Buch also beispielsweise für weiterführende Informationen auf eine Web-Seite verweisen, werden Sie statt des kompletten Links (der dank PHP, ASP oder ähnlichen Technologien nahezu unzählige, zumeist kryptische Buchstabenkombinationen enthält) nur eine Referenz auf einen *IntelliLink* (toller Begriff, was?) finden. So könnte eine Passage also folgendermaßen lauten:

»Mehr zum Thema Silk-Dämpfung finden Sie unter dem IntelliLink **A0108** – Sie können dort einiges Interessantes über die Entstehung von Materie und dunkler Materie erfahren.«

In diesem Fall geben Sie den URL <http://www.activedevelop.de/vb2008> in den Internet-Browser Ihrer Wahl ein, suchen den entsprechenden Link in der Liste, die nach Buchteil, Kapitel und Seite sortiert ist und finden so recht zügig den Weg zur Webseite, auf die eigentlich verwiesen sein soll – in diesem Beispiel zu den Alpha-Centauri-Folgen mit Professor Harald Lesch (die angesprochene Folge ist im Übrigen vom 14.09.2005).

WICHTIG Und lieber Herr Professor Lesch: Ich bekenne hier öffentlich, dass ich ein großer Fan von Alpha Centauri bin, und ich hoffe sehr, dass bald wieder mehr von Ihnen auf B3 zu sehen ist! ☺

Diese Vorgehensweise hat nicht nur den Vorteil, dass Sie nicht ellenlange Links in die Adresszeile des Browsers eintippen müssen, und dabei wie so oft die Wahrscheinlichkeit von Tippfehlern recht groß ist, sondern Links auch gepflegt und damit aktuell gehalten werden können.

Falls Ihnen selber ein Link auffällt, der nicht mehr aktuell ist, können Sie mich gerne jederzeit mit einer E-Mail an info@activedevelop.de darüber informieren.

Die Begleitdateien zum Buch und das E-Book

Neben den Verzeichnissen auf der beiliegenden DVD, die die Visual Studio Express-Editions beinhalten, finden Sie auf der DVD weitere Verzeichnisse, die auch sämtliche Begleitdateien sowie eine E-Book-Version dieses Buchs beinhaltet. Die PDF-Dateien des Vorgängerbuches »Visual Basic 2005 – Das Entwicklerbuch« können Sie sich zusammen mit den Beispieldateien unter dem IntelliLink **A0111** herunterladen. Es enthält z.B. noch wertvolle Informationen über ADO.NET, die aus Platzgründen nicht auf die 2008er-Version dieses Buches umgearbeitet werden konnten.

BEGLEITDATEIEN Kopieren Sie diese Dateien einfach in ein Verzeichnis Ihrer Wahl, damit Sie die dort enthaltenen Projekte an den Stellen, an denen es sinnvoll erscheint, anpassen, mit ihnen herumspielen und ausprobieren können. Wann immer Beispiele im Buch auftauchen, die zur Verdeutlichung von Zusammenhängen auf längere Codeabschnitte zurückgreifen, sehen Sie einen wie in diesem Beispiel gezeigten Hinweis auf die Begleitdateien sowie die Nennung des Unterverzeichnisses, in dem Sie das für das Beispiel benötigte Projekt finden.

Nützliches zu Visual Basic 2008 – www.codeclips.de

Natürlich kann ein Buch niemals so aktuell wie das Internet sein. Und sowohl Verlag als auch Autoren sind immer bemüht, neben dem Buch einen zusätzlichen Info-Service für den Leser zu bieten. Als Tipp: Die Seite <http://www.codeclips.de> sollten Sie deswegen regelmäßig besuchen.

Vielleicht finden Sie dort das eine oder andere nützliche Werkzeug, Lernvideo oder Codebeispiel, mit dem Sie sich Ihre tägliche Entwicklungsarbeit erleichtern können.

Kapitel 2

»Symbolischer Allzweckbefehlscode für Anfänger«

In diesem Kapitel:

Visual Studio das erste Mal starten	14
Konsolenanwendungen (Console Applications)	16
Anatomie eines (Visual Basic-)Programms	20
Programmstart mit der Main-Methode	23
Methoden mit und ohne Rückgabewerte	25
Deklaration von Variablen	25
Ausdrücke und Variablendefinitionen	27
Und was sind Objekte im Unterschied zu »normalen« Datentypen?	30
Programmstrukturen	32
Schleifen	32
Anweisungen, die Programmcode bedingt ausführen – If/Then/Else und Select/Case	38
Vereinfachter Zugriff auf Objekte mit With/End With	42
Gültigkeitsbereiche von lokalen Variablen	42

Merkwürdige Kapitelüberschrift, meinen Sie? Aber warum denn? Denn schließlich ist das die wörtliche Übersetzung von *beginners all-purpose symbolic instruction code*. Und nun nehmen Sie einmal jeden einzelnen Anfangsbuchstaben, und setzen ihn zu einem Wort zusammen – heraus kommt: *BASIC*. Als es 1964 von John George Kemeny und Thomas Eugene Kurtz am Dartmouth College entwickelt wurde (von den beiden stammt im Übrigen auch die Namensgebung), hatte es mit der Programmiersprache, wie wir sie heute als Visual Basic 2008 kennen, nur wenig, sehr, sehr wenig zu tun. Und von Objektorientierung ist BASIC damals so weit entfernt gewesen, wie Columbus am Ende seiner berühmten Entdeckungsreise von Hinterindien.

Dennoch gibt es immer noch grundlegende Sprachelemente in Form von grundsätzlichen Vorgehensweisen bei der Variablen Deklaration und der Anwendung von Strukturbefehlen, die immer noch sehr »basicesque« in der ursprünglichen Definition von BASIC sind, und über diese grundsätzlichen Sprachelemente wird Ihnen dieses Kapitel alles Wissenswerte erzählen.

Und nun verdrehen Sie nicht die Augen, und sagen, »Och, nö, kenn ich doch alles schon!«, denn: Im ungünstigsten Fall kennen Sie wirklich bereits alles, was Sie hier im Folgenden beschrieben finden, und Sie klopfen sich nach dem Kapitel auf die Schulter, loben sich in Form von »Jo – ich bin gut, ich mach sofort mit Objektorientierung weiter!«. Und dann widmen Sie sich hoch motiviert diesen sehr viel anspruchsvoller Themen; oder Sie lesen die folgenden Absätze und ertappen sich vielleicht doch beim manchmaligen »Wie, das geht auch?«-Sagen!

Doch eines sollte auch nicht unerwähnt bleiben: Dieses Kapitel soll kein Einsteigerbuch in Form von ausführlichen Erklärungen des Themas sein, und es fängt sicherlich auch nicht bei Adam und Eva an. Programmieren an sich sollte Ihnen nicht fremd sein, und die folgenden Abschnitte sollen Ihnen die Sprache Visual Basic zu »Rekapitulationszwecken« einerseits in Erinnerung rufen, Ihnen andererseits aber auch Unterschiede zu BASIC-Dialekten aufzeigen, mit denen Sie vielleicht bislang gearbeitet haben – und all das in möglichst kompakter Form. Damit scheidet dieses Kapitel aber eigentlich für einen Zweck schon aus: Nämlich Visual Basic ohne Vorkenntnisse von der Pike auf zu lernen.

Visual Studio das erste Mal starten

Heutzutage in Visual Basic zu programmieren bedeutet, dass Sie sicherlich 99,999% der Zeit Ihrer Arbeit in Visual Studio verbringen. Die übrige Zeit sind Sie damit beschäftigt, Codedateien aus anderen Projekten zu suchen und sie in Ihrem aktuellen Projekt einzubinden oder Visual Studio nach einem Absturz neu zu starten, was aber seit der 2008er Version glücklicherweise äußerst selten geworden ist.

Die IDE – die integrierte Entwicklungsumgebung – von Visual Studio 2008 stellt Ihnen die Hilfsmittel in einer Oberfläche zur Verfügung, mit der Sie Ihre Programme gestalten. Das sind nach Wichtigkeit sortiert:

- Der Visual Basic 2008-Compiler,¹ der dann aktiv wird, wenn Sie den Compile-Vorgang per Befehl (Befehle im Menü *Erstellen* oder entsprechende Symbolleistenklicks).
- Der Visual Studio Editor, der Sie mit Syntax-Highlighting, IntelliSense und anderen Highlights beim Editieren der Befehlszeilen für Ihr Programm unterstützt.
- Verschiedene Designer mit ihren entsprechenden Toolfenstern, die Sie beim Aufbauen von Formularen oder anderen visuellen Objekten unterstützen.
- Der Projektmappen-Explorer, der die Codedateien verwaltet und organisiert, aus denen Ihr Projekt besteht.

Beim ersten Start von Visual Studio 2008 landen Sie allerdings nicht direkt in dieser integrierten Entwicklungsumgebung, sondern Sie sehen im Vorfeld erst folgenden Dialog:



Abbildung 2.1 Beim ersten Start des Programms bestimmen Sie, wie die Entwicklungsumgebung voreingestellt werden soll

In diesem Dialog bestimmen Sie, mit welchen Voreinstellungen die Entwicklungsumgebung konfiguriert werden soll. Vorschlagsweise wählen Sie an dieser Stelle *Allgemeine Entwicklungseinstellungen* aus.

¹ Kurz zum Hintergrund: Programme, die Sie schreiben, werden vom Compiler in einen Zwischencode, den so genannten MSIL übersetzt, der Plattform-unabhängig dann zur Laufzeit in Prozessorcode übersetzt wird. Mehr zu diesem Thema erfahren Sie im nächsten Kapitel. Der Vollständigkeit halber: Visual Studio stellt natürlich auch weitere Compiler für C++ oder C# bereit, doch soll uns das in diesem Zusammenhang weniger interessieren. Mit Visual Basic Express startet übrigens ein abgespecktes Visual Studio, das tatsächlich nur den Visual Basic-Compiler enthält. Es steht Ihnen aber natürlich frei, dieses mit Visual C# oder C++ Express zu ergänzen. Den IntelliLink auf die Express-Download-Seiten finden Sie unter [A0201](#).

TIPP *Allgemeine Entwicklungseinstellungen* ist die Einstellung, mit der die meisten Visual Studio 2005 und 2008 Installationen voreingestellt werden. Visual Basic-Entwicklungseinstellungen enthalten spezielle Anpassungen, bei denen Fensterlayout, Befehlsmenüs und Tastenkombinationen mehr auf das schnelle Erreichen spezieller Visual Basic-Befehle angepasst sein sollen. Der Dialog zum Anlegen eines neuen Projektes ist beispielsweise nur auf Visual Basic-Projekte angepasst, um nicht zu sagen, beschnitten: Einige Optionen, wie das automatische Anlegen einer Projektmappe sind schon beim Anlegen eines neuen Projektes automatisch ausgeblendet – Sie haben die Möglichkeit, Projekte namenlos zu erstellen und erst am Ende der Entwicklungssitzung unter einem bestimmten Namen zu speichern. Das gilt auch für die Befehle, die Sie über Pulldown-Menüs abrufen können: »Angepasst« bedeutet, dass viele Funktionen, die auch Visual Basic-Projekte betreffen könnten, einfach ausgeblendet sind. Probieren Sie die für Sie am besten geeignete Variation aus. Falls Sie mit der hier gewählten Voreinstellung später nicht mehr zufrieden sind, finden Sie im nächsten Kapitel die notwendigen Hinweise, wie Sie Einstellungen von Visual Studio zurücksetzen können.

Konsolenanwendungen (Console Applications)

Wenn Sie für Anwender programmieren, dann entwickeln Sie mit großer Wahrscheinlichkeit Programme, die die grafische Benutzeroberfläche von Windows verwenden. Solche Anwendungen nennen sich im .NET-Jargon »Windows Forms-Anwendungen« oder kurz: »WinForms-Anwendungen«. Für den Anwender ist das sicherlich der zurzeit einfachste Weg, ein Programm zu bedienen – für Lernzwecke hingegen ist dieser Projekttyp nicht unbedingt der geeignete, da das aufwändige Drumherum von Windows mit all seinen grafischen Elementen wie Schaltflächen, Fenstern, der Maussteuerung etc. vom Erlernen des jeweiligen Sprachelements ablenkt.

Unter .NET können Sie jedoch auch einen weiteren Projekttyp verwenden – die Älteren unter uns werden sich an ihn noch gerne erinnern –, und auch Administratoren von Servern wird er selbst in der heutigen Zeit kein Unbekannter sein. Diese so genannten Konsolenanwendungen sind Programme, die mit einer sehr minimalistischen Benutzeroberfläche an den Start gehen: Solche Programme werden direkt an der Windows Eingabeaufforderung gestartet und auch ausschließlich durch die Tastatur gesteuert. Die einzige Schnittstelle zwischen Benutzer und Programm ist eine zeichenorientierte Bildschirmausgabe und die Tastatur.

Für die folgenden Abschnitte werden wir ausschließlich Konsolenanwendungen entwerfen. Sie erlauben – wie eingangs schon erwähnt – den Blick aufs Wesentliche. Und die folgende Schritt-für-Schritt-Anwendung zeigt, wie Sie eine Konsolenanwendung anlegen:

1. Wählen Sie aus dem Menü *Datei* den Befehl *Neu* und weiter den Unterbefehl *Projekt*. Visual Studio zeigt anschließend einen Dialog, wie Sie ihn auch in Abbildung 2.8 erkennen können.
2. Unter *Projekttypen* klappen Sie *Visual Basic* auf und wählen Windows aus.
3. Unter *Vorlagen* wählen Sie *Konsolenanwendung*, etwa wie in Abbildung 2.2 zu sehen.²
4. Geben Sie einen Namen für Ihr neues Projekt ein. Falls Sie möchten, dass ein ausdrückliches Projektmappen-Verzeichnis angelegt wird, entfernen Sie das Häkchen der entsprechenden Option.

² Visual Basic Express bietet Ihnen an dieser Stelle nicht ganz so viele Optionen; Sie legen ein neues Projekt aber in etwa genau so an, wenn Sie das Verhalten von Visual Basic Express im Umgang mit Projektmappen ändern: Wählen Sie dazu den Befehl *Extras/Optionen*, und entfernen Sie unter *Projekte und Projektmappen* im Zweig *Allgemein* die Option *Neue Projekte beim Erstellen speichern*.

HINWEIS Visual Studio ist dafür ausgelegt, dass Ihre Projekte sehr komplex und umfangreich werden können. So umfangreich, dass bei einem einzigen Projekt die Übersichtlichkeit einerseits verloren gehen kann, andererseits die Aufteilung eines Projektes in Teilprojekte einige dieser Teilprojekte auch für andere Projekte wieder verwendbar macht. Aus diesem Grund legen Sie nicht nur Projekte, sondern in der Regel auch eine Projektmappe an (Visual Basic Express hat hier leicht andere Voreinstellungen, beherrscht aber Projektmappen grundsätzlich auch). In dieser Projektmappe befindet sich im einfachsten Fall also nur ein Projekt – Ihre Windows- oder Konsolenanwendung. Das Projektmappen-Verzeichnis macht dann Sinn, wenn Sie mehrere Projekte in Ihrer Projektmappe erwarten: das können neben Ihrer eigentlichen Hauptanwendung beispielsweise auch weitere Klassenbibliotheken oder ganz andere Projekttypen wie beispielsweise Webservices sein. Das Basisverzeichnis enthält in dem Fall lediglich die »Solution«-Datei – übrigens der englische Name für Projektmappe –, und diese trägt die Endung *.sln*.

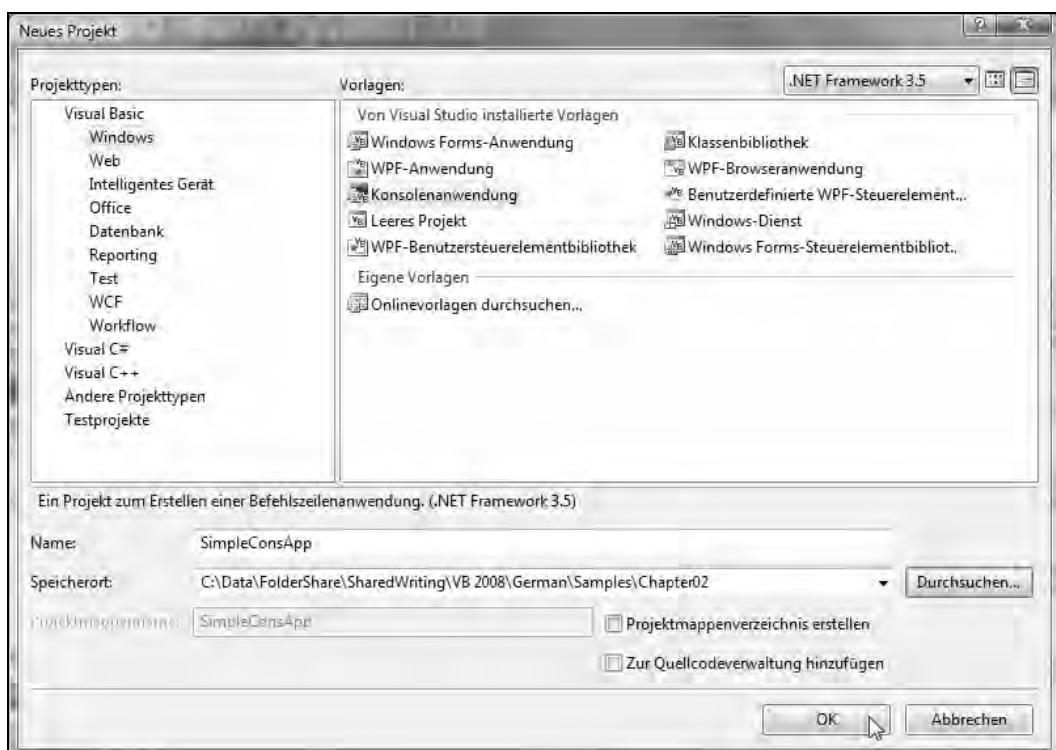


Abbildung 2.2 Mit diesem Dialog, den Sie mit Datei/Neu/Projekt erreichen, erstellen Sie ein neues Projekt für eine Konsolenanwendung

5. Bestimmen Sie mit *Durchsuchen* den *Pfad*, unter dem Sie Ihr Projekt abspeichern möchten.
6. Klicken Sie anschließend auf *OK*.

Im Anschluss daran befinden Sie sich im Code Editor von Visual Basic, in dem Sie zwischen den Befehlszeilen Sub Main und End Sub nun die folgenden Zeilen einfügen, sodass Ihr Ergebnis anschließend wie in Abbildung 2.3 aussehen sollte.

```
Sub Main()
    Dim geburtsjahr As Date
    Dim alter As Integer

    Console.Write("Bitte geben Sie ihr Geburtsdatum ein (dd.mm.yyyy): ")
    geburtsjahr = CDate(Console.ReadLine())
    alter = (Now.Subtract(geburtsjahr)).Days \ 365
    Console.WriteLine("Sie sind {0} Jahre alt", alter)
    Console.ReadKey()    End Sub
```

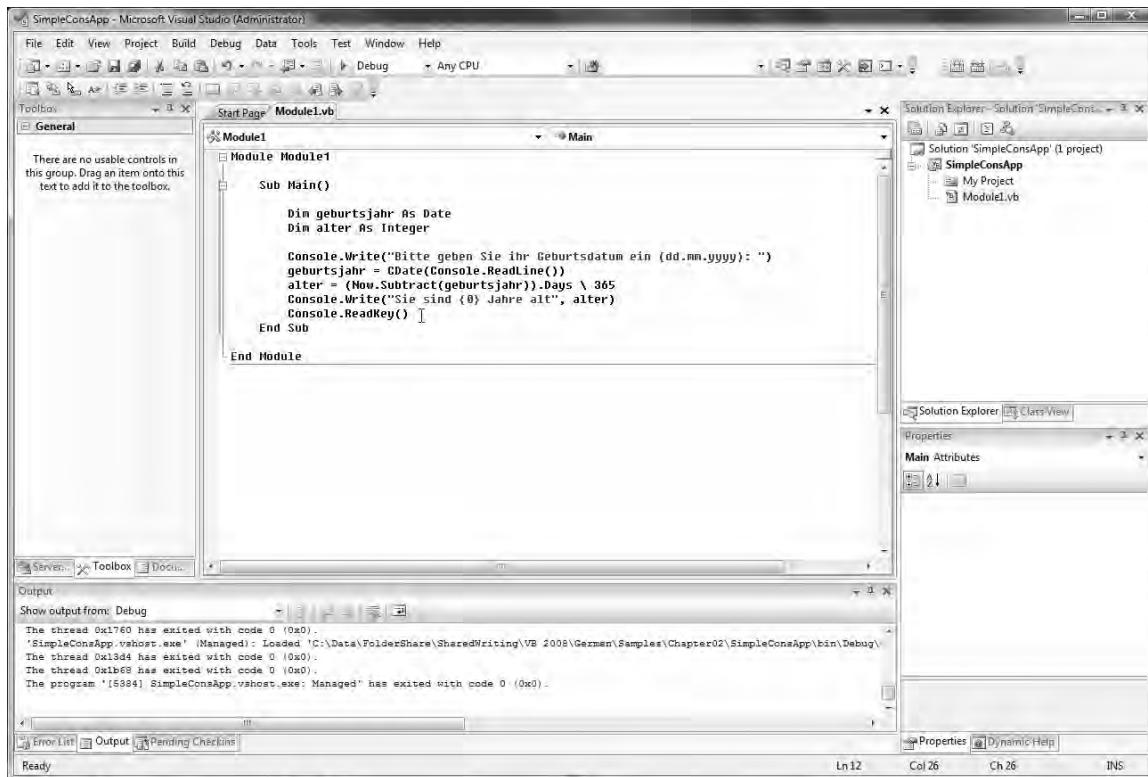


Abbildung 2.3 Die IDE von Visual Studio 2008 nach dem Anlegen einer neuen Konsolenanwendung und der Nacharbeitung einiger Codezeilen

Anwendung starten

Wenn Sie diese kleine Anwendung starten, indem Sie entweder **F5** drücken, das Startsymbol in der Symbolleiste anklicken () oder aus dem Menü *Debuggen* den Befehl *Debuggen starten* wählen, fordert sie Sie auf, Ihr Geburtsdatum einzugeben. Und hier sehen Sie, wie sehr sich Konsolenanwendungen von Ihren gewohnten Windows-Anwendungen unterscheiden: Lediglich über die Tastatur erfolgt die Interaktion mit dem Programm, sowie durch die Anzeige von Text auf dem Bildschirm – Abbildung 2.8 gibt Ihnen einen Eindruck von dieser (unserer ersten!) typischen Konsolenanwendung.



Abbildung 2.4 Eine typische Konsolenanwendung: Interaktion mit dem Benutzer erfolgt nur über die Tastatur und über reine Textausgaben

WICHTIG Die Geschwindigkeit, mit der eine Anwendung (ganz gleich ob Konsolenanwendung oder Windows Forms-Anwendung) abläuft, wenn Sie sie auf die gerade beschriebene Weise starten, entspricht lange nicht der, wie sie außerhalb der Visual Studio-Benutzerumgebung laufen würde. Sie haben nämlich innerhalb der VS-IDE die Möglichkeit, die Anwendung mithilfe des automatisch angehängten Debuggers auf Fehler zu untersuchen. Dabei können Sie beispielsweise an bestimmten Zeilen Haltepunkte setzen (mit **F9**, wenn Sie mit dem Cursor auf der entsprechenden Zeile stehen), um Ihr Programm, wenn es einen Haltepunkt erreicht, automatisch anzuhalten und dann bestimmte Programmzustände zur Laufzeit zu untersuchen und sie Zeile für Zeile (**F11**) oder in Prozedurenschritten (**F10**) abzuarbeiten. Dass Ihnen all diese Möglichkeiten zur Verfügung gestellt werden können, kostet Zeit bei der Programmausführung, wobei es keine Rolle spielt, ob Sie sie nun konkret nutzen oder nicht.

Möchten Sie Ihre Anwendung in der Geschwindigkeit laufen sehen, die sie unter normalen Bedingungen erreichen würde, starten Sie sie mit **Strg F5**, oder wählen Sie aus dem Menü *Debuggen* den Befehl *Starten ohne Debuggen*. Die Debugger-Funktionalität steht Ihnen dann natürlich nicht zur Verfügung.

TIPP Sie können sich übrigens die gesamte Debugging-Funktionalität auch durch eine Symbolleiste zur Verfügung stellen lassen. Dazu klicken Sie einfach mit der rechten Maustaste in der Visual Studio IDE auf einen freien Bereich innerhalb einer Symbolleiste, um das Kontextmenü zu öffnen. Wählen Sie hier *Debuggen*, um die entsprechende Symboleiste darzustellen, so wie in Abbildung 2.5 zu sehen.

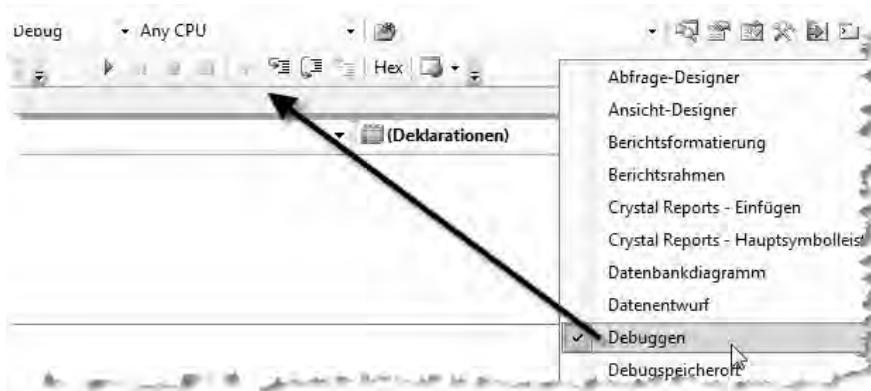


Abbildung 2.5 Öffnen Sie das Kontextmenü mithilfe der rechten Maustaste über dem freien Bereich einer Symbolleiste, können Sie weitere Symbolleisten anzeigen. Die Symbolleiste Debuggen stellt Ihnen einfachen Zugang zu vielen Debugging-Funktionen bereit.

Anatomie eines (Visual Basic-)Programms

Die ersten Datenträger in der EDV waren nicht magnetischer Natur, also keine Disketten und keine Festplatten, es waren Lochstreifen. Lochstreifen liefen durch einen Lochstreifenleser und anhand der Anordnung der Löcher fanden so die zuvor gestanzten Bits und Bytes ihren Weg zurück in den Computer. Wussten Sie aber, dass die ersten lochstreifenähnlichen Gebilde in der Textilbranche zum Einsatz kamen? Dabei übrigens nicht in Form der ersten frühen Computer, und mit der Aufgabe, dort vielleicht Umsätze oder Kundeninformationen zu speichern, sie fanden vielmehr ihren Einsatz in Webstühlen. Auf hintereinander angeordneten Holzplättchen wurden Informationen für die Steuerung des Webstuhls festgehalten.

Und streng genommen entsprach schon diese Vorgehensweise auch der, wie man in der heutigen Zeit festhält, was ein Computer machen soll, und damit der Anatomie eines Programms, wie Sie es auch heute entwickeln:

1. Sie benötigen etwas, dass Sie verarbeiten – nämlich die Daten (Wolle).
2. Sie benötigen Vorschriften, wie etwas verarbeitet wird – die Programmanweisungen (das Strickmuster).

Natürlich ist das eine starke Vereinfachung dessen, was Sie an Möglichkeiten haben, Ihre Programme zu gestalten, bzw. es lässt nicht unbedingt darauf schließen, welche unglaublichen und unfassbar vielen Möglichkeiten das sind. Auch unsere erste Anwendung schabt nur leicht die Spitze des Eisberges an, und deswegen finden Sie im Folgenden auch noch mal eine etwas getunte Version des ersten Beispiels, die Ihnen die verschiedenen Aspekte der Anatomie eines typischen Visual Basic Programms (wenn es sich auch immer noch nur um eine Konsolenanwendung handelt), auf der folgenden Seite auch in visueller Form, ein wenig näher bringen soll.

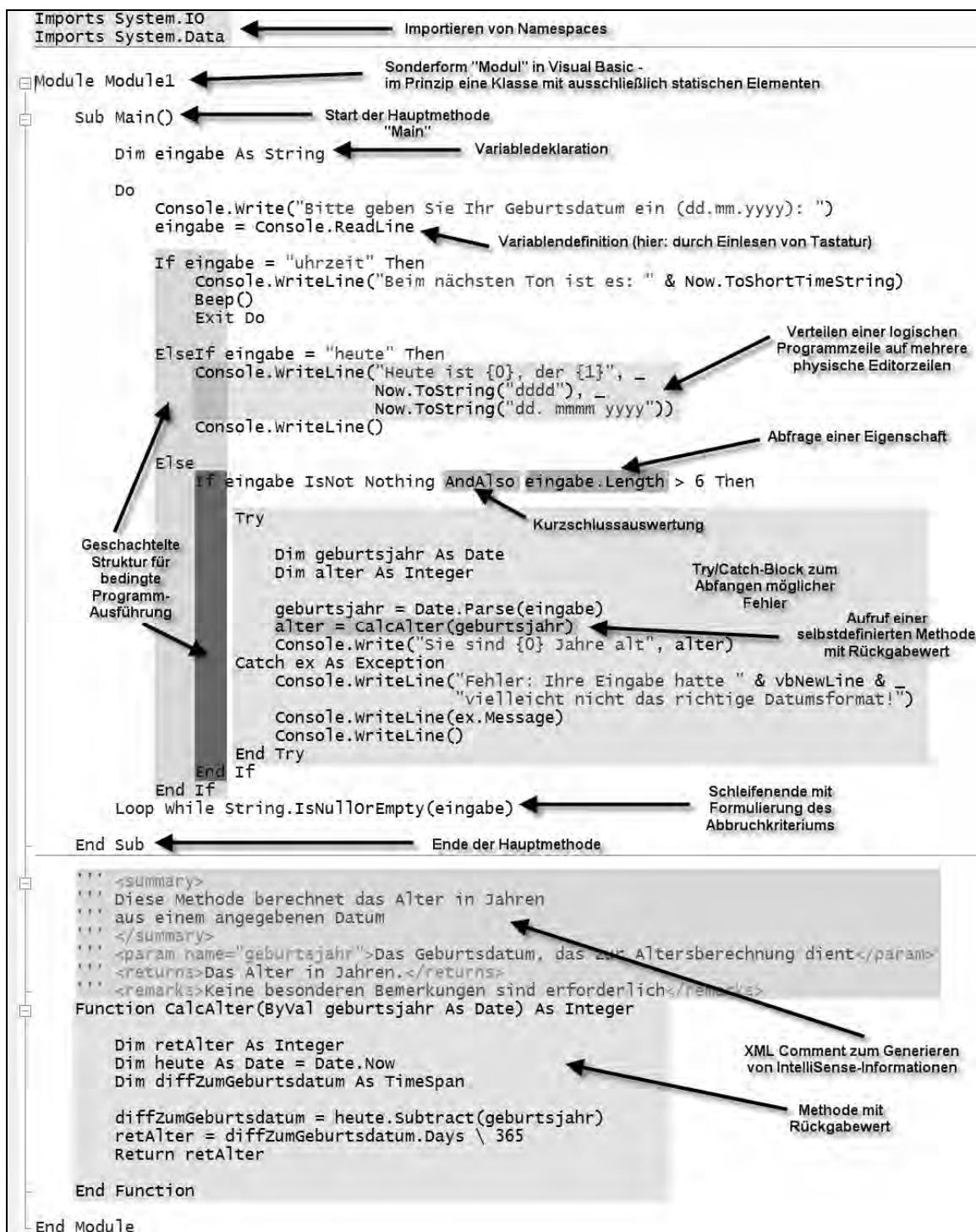


Abbildung 2.6 Die Anatomie einer kleinen Visual Basic-Anwendung visualisiert

Diese Grafik zeigt auch: Es stimmt zwar, dass eine Anwendung zum einen aus Daten und zum anderen aus Programmanweisungen besteht, doch gerade was die Programmanweisungen anbelangt, gibt es offensichtlich eine ganze Reihe unterschiedlicher Strukturen. Doch die sind sicherlich für Sie leichter zu verstehen, wenn Sie das Beispielprogramm auch einmal in Aktion gesehen haben, deswegen: Bevor wir uns mit der internen Funktionsweise des Programms beschäftigen und uns den einzelnen Komponenten der Sprache BASIC³ widmen, werfen wir erstmal einen Blick auf dessen Bedienung:

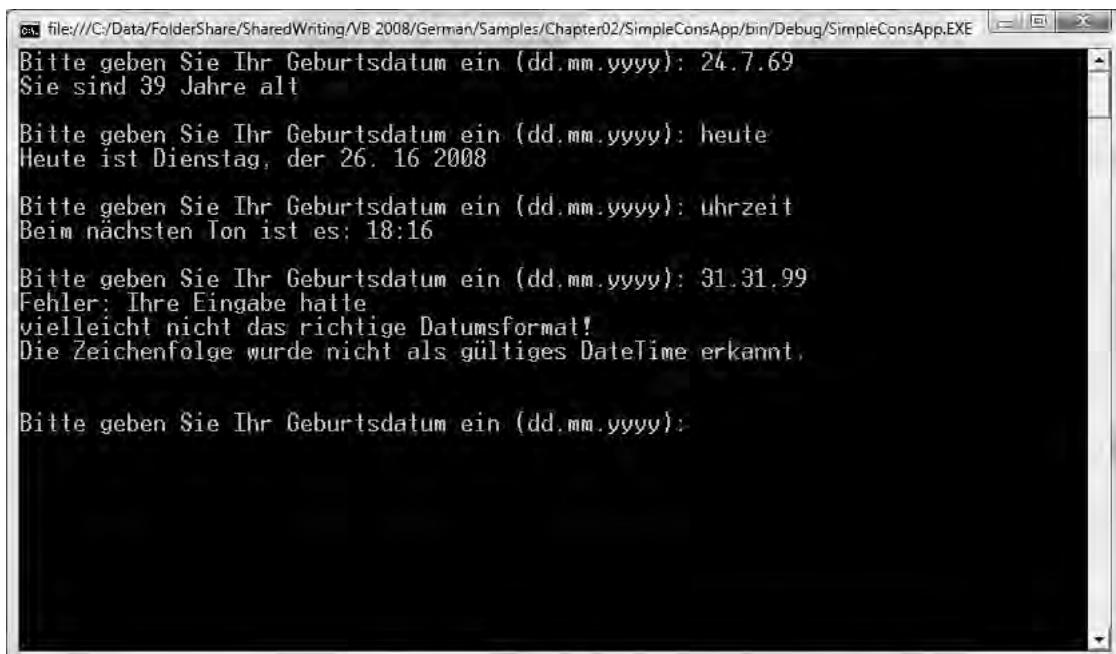


Abbildung 2.7 Die »Professional Edition« unseres Beispiels hat sogar noch ein paar mehr Features in petto, als die erste »Light«-Version

Selbsterklärend, oder? Nach dem Start der Anwendung geben Sie entweder ein Datum (am besten ein Geburtsdatum) oder einen der Befehle *heute* oder *uhrzeit* ein. Das Programm reagiert wie in Abbildung 2.8 zu sehen darauf. Falls Sie ein Datum mit ungültigem Format eingeben, wird dieser Fehler erkannt, abgefangen und das Programm zeigt Ihnen einen entsprechenden Hinweis.

Und wie funktioniert diese zugegebun recht einfache Anwendung nun intern? Auch recht einfach, und die Grafik hier auf der linken Seite gibt Ihnen auch einen – zugegeben – etwas alternativen Eindruck, mit welchen Elementen diese Anwendung aufgebaut wird.

³ Während Microsofts Produktnamen wie in »Visual Basic« tatsächlich ein kleingeschriebenes »Basic« enthalten, wird, wenn von der Sprache selbst die Rede ist, »BASIC« wirklich großgeschrieben.

Programmstart mit der Main-Methode

Jede Konsolenanwendung startet mit einer bestimmten Methode, die (nicht nur) in Visual Basic den Namen *Main* trägt und (nur in Visual Basic) Anweisungen enthält, die zwischen Sub und End Sub eingeklammert sind. Eine Sub in Visual Basic definiert also eine Methode ohne Rückgabewerte, und mindestens diese eine Methode *Main* benötigt eine Konsolenanwendung, damit Windows weiß, wo Ihr Programm startet. Methoden mit Rückgabewerten werden in Visual Basic .NET übrigens mit Function definiert, ihr Rückgabewert dann mit Return zurückgegeben – der Abschnitt nach diesem Kasten erklärt das ein wenig genauer.

Die Geschichte bedeutender Programmiersprachen ... und die Geschichte von Sub Main

Wir begegnen, wenn wir heute in der Geschäftswelt von Programmiersprachen sprechen, eigentlich nur zwei verschiedenen Konzepten: Der prozedural-strukturierten sowie der objektorientierten Programmierung⁴. Prozedural bedeutet, man kann, wie es der nächste Abschnitt deutlich macht, Code wiederverwenden bzw. mehrfach benutzen, da er von verschiedenen Punkten innerhalb des Programms aufgerufen werden kann, und stets an die Stelle zurückkehrt, von der aus er aufgerufen wurde. Ein anderes Wort für Methode ist übrigens Prozedur, und in einer solchen wird wieder verwendbarer Code platziert, weswegen sich diese Art des konzeptionellen Aufbaus eben prozedurale Programmierung nennt.

Der Terminus strukturierte Programmierung ergibt sich aus bestimmten Strukturkonstrukten, wie zum Beispiel If/Then/Else- oder Switch-Konstrukten. Und die objektorientierte Programmierung setzt da noch eine Abstraktionsebene drauf, doch das soll uns in diesem schönen grauen Kasten vorerst nicht interessieren. Uns soll hier beschäftigen: Wo nahm alles seinen Anfang?

Angefangen hat allerdings alles zu einer Zeit, zu der Entwickler noch die Prozessoren der damaligen Maschinen sozusagen zu Fuß in direktem Maschinencode programmieren mussten. Einem gewissen Herrn Backus ging das Entwickeln auf diese Weise irgendwann so sehr auf die Nerven, dass er mit der Bitte, ein Hilfsprogramm entwickeln zu dürfen, an seine Vorgesetzten herantrat. Der liebe Herr Backus arbeitete zu dieser Zeit – 1958 übrigens – auf einem 704 Mainframe-Computer, und seine Vorgesetzten waren wie er selbst Angestellte des Unternehmens, das diesen Computer baute: Die *International Business Machines Cooperation*, besser bekannt unter ihrem abgekürzten Namen IBM.

Dieses Hilfsprogramm sollte dabei helfen, Computer mit Befehlen zu programmieren, die ein wenig mehr an die menschliche Sprache angelehnt sein sollten und die es darüber hinaus auch erlaubten, direkt mathematische Funktionen auszuwerten und zu berechnen. Der hochtrabende Name dieses Projektes: *The IBM Mathematical Formula Translating System*. Diesen Namen jedoch ständig auszusprechen hätte das Projekt vermutlich erst ein Jahr später fertig werden lassen, sodass John Warner Backus sich kurzerhand für einen Kosenamen aus einer Vermischung einzelner Silben entschied, und geboren war eine der ersten Hochsprachen, nämlich ForTran – natürlich hier nur zur Verdeutlichung der Silbenherkunft mit großem »T« geschrieben.



⁴ Aspektorientierte und dynamische Programmierung werden gerade während diese Zeilen entstehen immer mehr zu einem Thema; durchgesetzt haben sich aber beide bislang nicht, obwohl man, während sich bei der aspektorientierten Programmierung sicherlich einige Spezialanwendungen in einem Bruchteil der OOP-Zeit lösen lassen, die Entwicklung bei der dynamischen Programmierung sicherlich auch für viele kommerzielle Anwendungsfälle im Auge behalten sollte.

Alle Hochsprachen dieser Zeit verfolgten das so genannte imperative Konzept, bei dem Befehle nacheinander abgehandelt wurden. Strukturen gab es eigentlich noch gar keine, geschweige denn Prozeduren. Die wenigen Abweichungen vom normalen Programmverlauf ergaben sich, wenn zum Beispiel Sprungziele berechnet wurden – eine GOT0-Anweisung gab es immerhin –, oder man eines von drei angebbaren Sprungzielen mit einer IF-Anweisung erreichen konnte, je nachdem, ob der zu untersuchende Ausdruck kleiner, gleich oder größer als 0 war. Mit jeder weiteren Entwicklung evolvierten die Programmiersprachen ganz allmählich über verschiedene Fortran- und Algol-Versionen, bis sie schließlich bei CPL (*Combined Program Language*, etwa: *kombinierte Programmiersprache*) landeten, einer Programmiersprache, die versuchte das Beste aus Fortran und Algol, den Sprachen für wissenschaftliche und Cobol, der Sprache für finanzmathematische Anwendungen zu vereinen. Dummerweise war das Ergebnis ein Monster, das zwar ideale Voraussetzungen hatte, selbst als Plattform für einen Sprachcompiler zu dienen, aber eben viel zu mächtig und träge war.

Martin Richards von der Cambridge University nahm sich deshalb der CPL an (von der man auch sagt, die Abkürzung sei von *Cambridge Program Language* abgeleitet worden; denn in Cambridge wurde diese Programmiersprache auch entwickelt⁵). Er speckte sie ab und schuf damit die Sprache BCPL, die aber wiederum nur auf Mainframe-Computern richtig einsetzbar war, jetzt aber immerhin als Plattform für die weitere Compilerentwicklung taugte.

Übrigens: BCPL war die erste Programmiersprache, die Sprachstrukturen {
 naja, {
 sagen wir:
 { deren Ansätze }
 }
 }
 erstmalig mit geschweiften Klammern kennzeichnete.

Ken Thompson und Dennis Ritchie waren es dann, die sich in den berühmten Bell Laboratories so etwa gegen 1969 das Ziel setzten, diese BCPL tatsächlich auch auf Mini-Computern zum Laufen zu bekommen. Da die damaligen Minicomputer natürlich auch Minimalisten in Sachen Hardware waren, war das ein ehrgeiziges Ziel, zu dem es nur einen Weg gab: Alles an Elementen musste dazu aus der Sprache verschwinden, was nicht absolut überlebenswichtig war, und die Jungs taten genau das, und noch mehr – denn sie machten in ihrem Sparbemühen auch vor dem Namen nicht halt: Aus BCPL wurde so »B«, der – Sie ahnen es vielleicht – Papa von C und Opa von C++ – und letzten Endes auch C#. Übrigens: Im Tutorial zur B-Implementierung von B. W. Kerninghan und Murray Hill heißt es gleich auf Seite 1 direkt unter dem Abschnitt 2 »Generelles Layout von Programmen«:

```
main( ) {  

  statements  

}  

newfunc(arg1, arg2) {  

  statements  

}
```

⁵ Um ganz genau zu sein: CPL war ein Gemeinschaftsprojekt der Cambridge University und der Computerabteilung der Londoner Universität, was wiederum zu Behauptungen führte, dass die ursprüngliche Abkürzung *Cambridge Programming Language* aus diesem Grund in *Combined Programming Language* abgeändert wurde und weniger aus der Kombination zweier Computersprachwelten.

```
fun3(arg) {  
    more statements  
}
```

Und weiter:

»Alle B-Programme bestehen aus einer oder mehreren Funktionen, die den Funktionen oder Unterrouitinen eines Fortran-Programms entsprechen, beziehungsweise den Prozeduren von PL/I. Bei Main [im Beispiel] handelt es sich um eine solche Funktion, und in der Tat müssen alle B-Programme über eine Main-Funktion verfügen. Die Ausführung eines Programms beginnt bei der ersten Anweisung von Main und endet für gewöhnlich bei der letzten. Main sorgt normalerweise dafür, dass weitere Funktionen aufgerufen werden – sowohl die des Programms als auch andere aus Bibliotheken. Wie in [modernem] Fortran besteht mit der Übergabe von Parametern [bzw. Argumenten] eine Möglichkeit der Kommunikation...«

Und nun wissen Sie nicht nur ein wenig mehr über die Evolution bedeutender Programmiersprachen der Neuzeit, sondern auch um die historische Bedeutung der Main-Funktion, die es schon in B vor über 35 Jahren gegeben, und deren Name sich als Symbolbezeichner für den Einsprungspunkt zum Start eines Programms bis heute gehalten hat.

Methoden mit und ohne Rückgabewerte

Eine Methode Main, wie Sie sie gerade kennengelernt haben, hat natürlich einen Sonderstatus innerhalb einer Anwendung – denn sie braucht mindestens einen benutzerdefinierten Typ, in dem sie »liegen« kann und definiert nur durch ihren Namen den Startpunkt der Anwendung. Ein benutzerdefinierter Typ ist dabei eigentlich eine Klasse – in Visual Basic gibt es aber auch den Sonderfall einer Klasse, ein Modul. Ein Modul ist im Prinzip nichts weiter als eine Klasse, die über Methoden verfügt, die immer da sind – man nennt das auch *statische* Methoden.

Funktionen mit Rückgabewerten

Generell benötigen Sie Methoden, um Ihre Anwendungen einerseits übersichtlicher, aber andererseits auch, um bestimmten Code mehrfach verwendbar machen zu können. Umso mehr Sinn macht es dann, wenn Methoden Parameter, also andere Daten entgegen nehmen können (zur Kommunikation zwischen einer Methode und der aufrufenden Entität – wie vorhin gegen Ende des grauen Kasten zitiert), Berechnungen mit diesen Parametern anstellen und in Abhängigkeit dieser Eingabedaten und der entwickelten Algorithmen ein Rückgabergebnis liefern. In der Grafik auf Seite 21 sehen Sie ein solches Beispiel in der Methode CalcAlter; dieser Methode wird ein Geburtsdatum als Parameter übergeben,

Deklaration von Variablen

Unsere Anwendung *deklariert* als erste Zeile ihrer Main-Methode eine so genannte Variable, in der es für die Dauer ihres »Lebens« Daten vorhalten kann.

Die Zeile

```
Dim eingabe As String
```

legt fest, dass die Variable `eingabe` Inhalte vom Typ `string` aufnehmen kann.

`String` bedeutet dabei, dass `eingabe` Zeichenketten (engl. *character strings*) oder mit anderen Worten, beliebigen Text aufnehmen kann. `String` stellt also den Typen dar. Würden Sie als Typ `Integer` angeben, dann könnte die Variable `eingabe` Zahlen zwischen ungefähr -2 Milliarden und 2 Milliarden speichern, aber auch nur ganze Zahlen, keine Brüche. Und mit `Double` als Typ auch Brüche, deren Wertebereich allerdings mit größerer Genauigkeit kleiner wird, weswegen man auch von Fließkommazahlen spricht, da sich das Dezimalkomma verschieben kann.

WICHTIG In .NET (nicht nur in Visual Basic .NET) ist die Typsicherheit übrigens sehr wichtig, denn die Verwendung falscher Typen ist – wenn eine Programmiersprache das zulässt – eine sehr häufige Fehlerquelle. Wenn Sie ein Datum beispielsweise im deutschen Datumsformat ausschreiben und sortieren, dann ist dabei z.B. der 11.10.2002 größer als der 10.11.2005 – denn die 11 ist größer als die 10. Als Datumswert betrachtet ist das zweite Datum natürlich »größer« als das erste, da der jüngere Wert. Das Sortieren als Zeichenkette führt also zu einem völlig anderen Ergebnis als das Sortieren nach Datum – Typsicherheit wird deshalb in jeder .NET-Sprache großgeschrieben.

Die verschiedenen Typen, die Sie im .NET Framework und damit auch in Ihren Visual Basic-Programmen verwenden können, finden Sie übrigens im nächsten Kapitel beschrieben.

HINWEIS Dass Variablen in Visual Basic mit dem Schlüsselwort `Dim` »vertypisiert« werden – man nennt diesen Vorgang korrekterweise *Deklarieren von Variablen* – hat übrigens historische Gründe. In den ersten BASIC-Versionen war es ursprünglich möglich, Variablen »einfach so« zu verwenden, ohne zuvor etwas zu Typ und über zukünftige Namenskonsistenz zu definieren. Variablen wurden erst zur Laufzeit des Programms aus dem Kontext ihrer Zuweisung angelegt und typisiert. Das galt aber nicht für so genannte Arrays, Variablenfelder also, die man mithilfe eines oder mehrerer Indexe verwenden konnte, um beispielsweise innerhalb von Schleifen mit Zählvariablen auf die einzelnen Feldelemente zugreifen zu können (und natürlich auch immer noch kann – denn Arrays sind weiterhin zentraler Bestandteil aller .NET-Sprachen). Hier war es schon damals erforderlich, die Dimensionen und Grenzen eines Arrays (Ober-, Untergrenze, Anzahl Dimensionen) zu definieren, und deswegen leitet sich der Befehl für das Einrichten der Dimensionen eines Arrays bis heute auch für die Deklaration normaler Variablen ab.

Durch die reine Deklaration einer Variablen hat sie allerdings noch keinen Wert, bzw. falls es etwas wie einen Grundzustand der Variablen gibt, wurde ihr dabei automatisch dieser Wert zugewiesen – das ist der Wert 0 für alle numerischen Datentypen, `false` für boolesche Variablen und *nichts* für Zeichenketten. Das »nichts« dabei wirklich ein Wert bzw. ein Variablenzustand sein kann (und der Computer damit philosophisch betrachtet einen Gegensatz in sich selbst schafft, nämlich: »der Zustand ist nichts«), sei nur am Rande erwähnt und einfach hingenommen. Übrigens: Sie finden zu nichts wirklich gute Grundlagen in Wikipedia. Entschuldigung – ich meinte natürlich: Sie finden zu »Nichts« wirklich gute Grundlagen in Wikipedia.

Die eigentliche Definition einer Variablen erfolgt durch die Zuweisung eines Ausdrucks – der nächste Abschnitt »Ausdrücke und Variablendefinitionen« weiß mehr darüber.

Ausdrücke und Variablendefinitionen

Sowohl für numerische Berechnungen als auch für Datumsberechnungen oder Zeichenkettenoperationen ist es wichtig, die Funktionsweise von Ausdrücken verstanden zu haben. Im Beispiel in der Grafik auf Seite 21 ist die Zeile

```
Eingabe = Console.ReadLine()
```

ein String-Ausdruck. Die Methode `ReadLine` liefert hier eine Zeichenkette zurück (in diesem Fall eine, die der Benutzer über die Tastatur eingegeben hat), und weist das Ergebnis an die Variable `Eingabe` zu.

Ein anderes Beispiel für einen Ausdruck aus dem Beispiel in der Grafik auf Seite 21 ist die statische Parse-Methode des `Date`-Datentyps, und mit der Zuweisung des Ausdrucks

```
geburtsjahr = Date.Parse(eingabe)
```

wird die Zeichenkette, die das Datum repräsentieren soll, in einen echten Datumstyp umgewandelt – denn wir erinnern uns: Der richtige Typ an dieser Stelle ist relevant; ein Computer sortiert eine Zeichenkette, die zufälligerweise einen Datumswert darstellt, anders als ein Datum (siehe auch Kasten »Wichtig« auf Seite 26). Deswegen müssen wir den `Date`-Datentyp anweisen, die Zeichenkette zu parsen (Zeichen für Zeichen zu durchlaufen und dabei zu analysieren), und zu versuchen, die Zeichen als Datum zu interpretieren und entsprechend umzuwandeln.

Gleichzeitiges Deklarieren und Definieren von Variablen

Deklaration und Definition von Variablen kann übrigens in einem Rutsch erfolgen. Anstelle also

```
Dim weihnachten08 As Date  
weihnachten08 = #12/24/2008#
```

können Sie auch schreiben:

```
Dim weihnachten08 As Date = #12/24/2008#
```

HINWEIS Der `As`-Part hinter einer Variablen-deklaration ist übrigens innerhalb von Prozeduren (zum Beispiel innerhalb von Methoden, die mit `Sub` oder `Function`) definiert werden, mit Visual Basic 2008 nicht mehr erforderlich, wenn Deklaration und Definition in einem Rutsch erfolgen. Sobald die Option *lokaler Typrückschluss* (mehr dazu im entsprechenden Abschnitt in Kapitel 11) aktiviert ist, kann der Visual Basic-Compiler bei einer Ausdruckszuweisung während einer Deklaration aufgrund des Typs des Ausdrucks den Typ für die Deklaration erkennen. Anstelle also Definition und Deklaration einer *String*-Variablen mit

```
Dim süßeZoe as String = "Zoe Dröge"
```

reicht es auch

```
Dim süßeZoe = "Zoe Dröge"
```

zu schreiben. Der Compiler erkennt nämlich, dass der konstante Ausdruck »Zoe Dröge« ein String ist und zieht bei der »Vertypisierung« den entsprechenden Rückschluss.

Komplexe Ausdrücke

Das Ergebnis bzw. der Rückgabewert einer Methode kann natürlich auch als Operand eines Operators dienen und so die Rückgabewerte mehrerer Methoden miteinander in einem komplexen Ausdruck kombinieren. Nur der einfacheren Verständlichkeit halber ist die Methode aus dem Ausgangsbeispiel

```
Function CalcAlter(ByVal geburtsjahr As Date) As Integer
    Dim retAlter As Integer
    Dim heute As Date = Date.Now
    Dim diffZumGeburtsdatum As TimeSpan
    diffZumGeburtsdatum = heute.Subtract(geburtsjahr)
    retAlter = diffZumGeburtsdatum.Days \ 365
    Return retAlter
End Function
```

so formuliert, dass ein Ergebnis in mehreren Zwischenschritten errechnet wird. Natürlich könnte man den Wert, der sich in der vorletzten Zeile in `retAlter` befindet, und der mit `Return` zurückgegeben wird, nicht nur an einem Stück berechnen, sondern auch zurückgeben. Das sähe dann so aus:

```
Function CalcAlter(ByVal geburtsjahr As Date) As Integer
    Return Date.Now.Subtract(geburtsjahr).Days \ 365
End Function
```

Das Ergebnis ist natürlich dasselbe.

Numerische Ausdrücke zu verstehen, fällt nicht sonderlich schwer – in jeder 10. Klasse sind wir mit ihnen spätestens das erste Mal konfrontiert worden. Zeichenkettenausdrücke verhalten sich da ähnlich. Wenn wir verstehen, dass Deklaration, Ausdrucksberechnung und anschließende Definition von

```
Dim dbl As Double
dbl = 5
dbl = dbl + 5 * 10
```

für `dbl` 100 ergeben (Hierarchieregel: »Potenz vor Klammer vor Punkt vor Strich« wird bei der Auswertung natürlich beachtet!), dann können wir auch einfach nachvollziehen, wieso

```
Dim str As String  
str = "-> Angela Wördehoff <-"  
str = str.Substring(3, 16)
```

»Angela Wördehoff« als Wert für die Variable str ergibt, und wieso

```
Dim übermorgen As Date  
übermorgen = Date.Now  
übermorgen = übermorgen.AddDays(2)
```

immer zwei Tage ab heute ergibt.

Boolesche Ausdrücke

Numerische Ausdrücke, Ausdrücke, die Datumswerte berechnen und String-Ausdrücke sind vergleichsweise einfach zu lesen und verstehen – zu sehr sind sie an typische Formeln von Kurvendiskussionen angelehnt, die wir alle in der 9. oder 10. Klasse kennengelernt haben.

Wie würden Sie allerdings den folgenden Ausdruck lesen?

```
Dim var = 5 = 5
```

Ist diese Zeile gültig, und falls ja, von welchem Typ ist var und welchen Wert trägt var?

Zunächst: Ja, diese Codezeile ist gültig. Zum zweiten: Sie definiert eine Variable vom Typ Boolean. Variablen dieses Typs haben keinen außergewöhnlich großen Zahlenbereich, denn sie können nur zwei Zustände wiedergeben: True und False.

Und jetzt betrachten wir die Formel da oben noch mal ein wenig analytischer: $5 = 5$ ist eine wahre Aussage. Wahr heißt auf englisch True, das Ergebnis des Ausdrucks $5=5$ ist also True. Da der Gleichheitsoperator (nicht der Zuweisungsoperator, der eine Variable definiert: gleiches Operatorzeichen, anderer Kontext, andere Bedeutung!) immer ein boolsches Ergebnis zurückliefert, wird die Variable var aus diesem Grund auch durch lokalen Typrückschluss als boolean definiert, und würden Sie anschließend Ihr Ergebnis ausdrucken, zum Beispiel durch die Anweisung

```
Console.WriteLine(var.ToString)
```

ergäbe das auch den »Wert« True.

Boolesche Variablen bzw. boolesche Ausdrücke sind deswegen so wichtig, weil sie als Argumente in Anweisungen für die bedingte Programmabarbeitung oder auch zum Testen eines Schleifenabbruchkriteriums eingesetzt werden. Bei einer If-Abfrage zum Beispiel wird der zwischen If und EndIf stehende Codeblock dann ausgeführt, wenn das Ergebnis des hinter If stehenden booleschen Ausdrucks True ist, ansonsten eben nicht. Und gibt es in diesem Konstrukt noch einen Else-Zweig, dann wird dieser ausgeführt, wenn das Ergebnis False ist – der Abschnitt »If ... Then ... Else ... ElseIf ... EndIf« ab Seite 38 zeigt nochmals detaillierter, wie das funktioniert.

Und was sind Objekte im Unterschied zu »normalen« Datentypen?

Objekte sind Dinge. Irgendetwas. Was man anfassen oder sonst wie beschreiben kann. Und das ist beim Programmieren nicht anders. Objekte sind damit eines der abstraktesten Gebilde, neben den Threads vielleicht so das abstrakteste Gebilde, was man bei der Programmierung kennt.

Um zu erklären, was ein Objekt ist, beschreibt man am Besten, was es nicht ist. Eine Integer-Variable beispielsweise ist kein Objekt. Eine Datums-Variable auch nicht. Eine boolsche Variable nicht. Ein Datentyp namens Point, in der eine Position zum Zeichnen festgelegt wird, ist auch kein Objekt. Das Gebilde, auf dem man zeichnet, der Inhalt eines Fensters, einer PictureBox oder ein Druckerkontext, ist hingegen ein Objekt. Ein Pinsel, mit dem gezeichnet wird, ist ein Objekt. Eine Schaltfläche ist ein Objekt. Eine TextBox ist ein Objekt. Ein Tooltip ist eines, eine TCP/IP-Verbindung kann auch ein Objekt sein, ein Objekt nämlich, das diese Verbindung steuert. Beim String wird es schwierig. Ist eine Zeichenkette ein Objekt? Prinzipiell ja, aber per Definition nicht.

Sie merken: Objekte und primitive Datentypen haben schon etwas gemeinsam. Beide speichern Daten und beide reglementieren den Zugang zu diesen Daten. Aber Objekte sind in der Regel komplexer, sie brauchen mehr Speicherplatz und sie verfügen über mehrere Methoden, um etwas mit diesem Objekt zu machen (eine TextBox fokussieren, eine Verbindung aufbauen, eine Schaltfläche in den Vordergrund holen). Sie verfügen auch über komplexere Eigenschaften (den Text einer TextBox bestimmen, die Puffergröße einer TCP/IP-Verbindung bestimmen, eine Schaltfläche ein- und ausschalten, und damit ihren »Eingeschaltet«-Zustand ändern). Und: Objekte können Ereignisse auslösen (Eine Schaltfläche wurde geklickt, der Text einer TextBox geändert, in einer offenen Verbindung wurden Daten empfangen).

Einfache Variablentypen wie Integer, Double, Date oder Boolean kann man deswegen direkt nach der Deklarierung verwenden. Die .NET Framework-Infrastruktur – die Common Language Runtime (CLR) um genau zu sein – kümmert sich unter anderem darum, dass bei diesen so genannten primitiven Datentypen der entsprechende Speicherplatz auf dem Prozessorstack⁶ reserviert ist. Eine Variable in Ihrem Programm ist also mit einer Speicheradresse auf dem Prozessorstack verbunden.

Komplexere Objekte passen da nicht drauf. Eines vielleicht – aber für Ihr Vorhaben einen Bilderorganizer zu programmieren, werden Sie wohl mehr als nur ein Objekt gleichzeitig im Speicher halten müssen. Und deswegen muss für diese komplexen Objekte Speicher reserviert werden. Dieser zu reservierende Speicher nennt sich in .NET- Sprech übrigens *Managed Heap*. Managed Heap, weil Sie sich nicht sorgen müssen, dass andere Datenstrukturen den Speicherbereich für Ihre Bilder verletzen können. Und weil Sie sich auch nicht darum kümmern müssen, den Speicherbereich wieder freizugeben, wenn Sie das Objekt nicht mehr benötigen.

⁶ Ein spezieller Speicherbereich zum Ablegen kurzfristiger Informationen, auf den der Prozessor bzw. ein laufender Prozess mithilfe des Prozessors extrem schnell zugreifen kann.

Ableiten von Objekten und abstrakten Objekten

Objekte unter .NET kennen aber noch eine Besonderheit: Sie bauen aufeinander auf.

Das ist mit Objekten im täglichen Leben nicht anders: Ein Gefäß zum Beispiel ist ein abstraktes Objekt, das wir vielleicht generell klassifizieren aber nicht konkretisieren können. Eine Milchbüte ist sicherlich ein Gefäß, aber ein Gefäß muss nicht zwangsläufig eine Milchbüte sein. Ein Gefäß kann auch eine Flasche oder ein Eimer oder ein Fass sein. Die Schnittmenge bestimmter Eigenschaften macht all diese Objekte zu einem Gefäß, und deswegen kann man sagen, dass sämtliche flüssigkeithaltende Objekte wie Milchbüte, Flasche, Fass oder Eimer von Gefäß abgeleitet sind.

Bei der Programmierung funktioniert das ebenfalls. Sie können die Vorlage für ein Objekt erstellen – so etwas nennt man übrigens Klasse – aber diese Vorlage nur für weitere Vorlagen verwenden. Und nur die Unterschiede fließen dann in die Ableitungen ein, und machen – durch weitere Eigenschaften und Methoden – aus einer abstrakten Klasse Gefäß eben die Klasse Flasche oder die Klasse Milchbüte oder die Klasse Fass. Und dann benutzen Sie diese neue Vorlage, um ein konkretes Fass ins Leben zu rufen – es zu instanzieren. Sie instanzieren dann ein Objekt aus einer Klasse.

Auch da gibt es wieder eine Analogie, Förmchen und Sandkuchen. Ein Förmchen ist eine Klasse, das Instanziieren der Sand, den Sie vom Sandhaufen, also vom »Sand Heap«, nehmen. Sie haben nur ein Förmchen, also eine Klasse, können aber durch das Einfüllen des Sandes in das Förmchen unzählige Objekte herstellen. In .NET-Sprech würde man sagen: »Sie reservieren auf dem Managed Heap Speicher, damit eine Klasse in ein Objekt instanziert werden kann⁷.

Das ist übrigens auch der Grund, weswegen Objekte zum Neuanlegen – zum Instanziieren – das New-Schlüsselwort benötigen. Wenn Sie also eine neue Schaltfläche einem Windows-Formular hinzufügen wollen, dann sind die folgenden Zeilen dafür notwendig:

```
'Neues Button-Objekt instanzieren
Dim t As New System.Windows.Forms.Button

'Text-Eigenschaft setzen
t.Text = "Neue Schaltfläche"

'Dem Formular hinzufügen
meinFormular.Controls.Add(t)

'Schaltfläche fokussieren
meinFormular.Focus()
```

Klassen und Objekte sind ein so umfangreiches Thema, dass ihnen ein ganzer Buchteil gewidmet ist. Mit dieser ungefähren Vorstellung wird es Ihnen aber zunächst erstmal leichter fallen, in den vielen Beispielen bis dort hin Klassen und Objekte verwenden zu können und sie vor allen Dingen von primitiven Datentypen zu unterscheiden.

⁷ In »Sandkasten-Sprech« übersetzt würde das heißen: »Sie nehmen Sand aus dem Sandhaufen, damit ein Förmchen daraus einen Sandkuchen bilden kann.«

Programmstrukturen

Das Abarbeiten von Befehlen ist neben der Fähigkeit eines Programms, Daten mit Variablen zu speichern, seine zweite große Aufgabe. Damit ein Programm aber eben nicht wie ein Webstuhl einfach hintereinander vordefinierte Anweisungen umsetzt, sondern auch flexibel auf besondere Zustände innerhalb seiner Datenstrukturen reagieren kann, bzw. eine Reihe von Anweisungen beispielsweise bis zum Eintreten einer bestimmten Bedingung immer wieder kontrolliert zu wiederholen in der Lage ist, gibt es besondere Programmstrukturen, in Form von Schleifenkonstrukten und Konstrukten für die bedingte Programmabarbeitung.

Zu letzterem gehört strenggenommen auch das Abfangen von Fehlern mit Try/Catch, da hier innere Programmstrukturen ebenfalls nur bedingt ausgeführt werden, nämlich dann, wenn innerhalb eines Try-Blocks eine Ausnahme aufgetreten ist. Die folgenden Abschnitte liefern die Details:

Schleifen

Schleifen dienen dazu, ihre inneren Befehle in der Grafik auf Seite 21 sehen Sie ein Beispiel für eine Schleife in Form eines Do/Loop-Konstrukts, das dafür sorgt, dass der Anwender wiederholt Geburtsdatumsangaben bzw. Befehlseingaben durchführen kann, und zwar sooft, bis er an der Eingabeaufforderung einfach drückt, die Variable Eingabe also leer bleibt und die Abbruchbedingung für die Schleife erreicht ist, die hinter dem Loop-Schlüsselwort formuliert wurde.

Neben Do/Loop-Schleifen kennt Visual Basic .NET übrigens auch For/Next-Schleifen, die alle inneren Anweisungen so oft wiederholen, bis eine Zählvariable einen bestimmten Wert über- oder unterschritten hat, For Each/Next-Schleifen, die alle inneren Anweisungen so oft wiederholen, wie es bestimmte Elemente in einer Objektauflistung gibt und While/End While-Schleifen, die ihre inneren Anweisungen so oft wiederholen, wie eine bestimmte Abbruchbedingung noch gilt, die direkt hinter While angegeben wird. Gibt es übrigens keine Abbruchbedingung, spricht man von einer Endlosschleife – das Schleifeninnere läuft dann solange, bis Sie Ihre Stromrechnung nicht mehr bezahlen. (Wobei das strenggenommen auch schon wieder eine Abbruchbedingung darstellt.)

For/Next-Schleifen

Mithilfe einer For/Next-Schleifenkonstruktion können Sie erreichen, dass eine Reihe von Anweisungen des umklammernden Codeblocks so oft wie angegeben wiederholt werden. Die Syntax für eine solche Konstruktion lautet:

```
For zählvariable [As datentyp] = start To end [ Step schrittweite ]
  [ Anweisungen ]
  [ Exit For ]
  [ Continue For ]
Next [ zählvariable ]
```

Dabei ist zählvariable der For-Anweisung zentraler Bestandteil des gesamten Schleifenkonstruktions und deswegen erforderlich. datentyp bestimmt, von welchem numerischen Typ zählvariable sein soll, kann aber

auch weggelassen werden, wenn lokaler Typrückschluss eingeschaltet ist (was es standardmäßig ist) oder wenn diese Variable bereits zuvor definiert war.

start und ende sind erforderliche Angaben und bestimmen, wo die Zählung beginnt und wo die Zählung endet. Falls durch schrittweite nicht angegeben, wird die Schleife sofort wiederholt, wie es die ganzzahlige Differenz von ende und start angibt. schrittweite kann aber optional angegeben werden, und stellt den Wert dar, um den zählvariable mit jeder durchlaufenen Schleife erhöht wird.

Zwischen For und Next können anschließend beliebig viele Anweisungen stehen, die mit der angegebenen Anzahl von Wiederholungen ausgeführt werden.

Die Anweisung Exit For ist optional, und in der Regel selbst in einem konditionalen If-Block eingebettet. Sie überträgt die Steuerung aus der For-Schleife, beendet also die Schleife, auch wenn zählvariable ihren ende-Wert noch nicht erreicht hat.

Die Angabe von Next ist erforderlich, denn es beendet die Definition der For-Schleife.

Verwenden Sie eine For/Next-Struktur am besten, wenn ein Satz von Anweisungen mit einer festgelegten Anzahl von Wiederholungen ausgeführt werden soll. Eine While/End While-Schleife oder eine Do/Loop-Schleife (siehe unten) eignen sich besser, falls Sie nicht im Voraus wissen, wie oft die Anweisungen in der Schleife auszuführen sind. Wenn die Schleife jedoch eine festgelegte Anzahl von Wiederholungen durchlaufen soll, eignet sich eine For/Next-Schleife am besten, denn Sie bestimmen die Anzahl der Iterationen (Durchläufe, Wiederholungen) zu Beginn der Schleife.

HINWEIS Der Wert von schrittweite kann positiv oder negativ sein. Entsprechend müssen natürlich die Start- und Endwerte der Schleife angepasst werden, damit es eine sinnvolle Anzahl an Schleifeniterationen geben kann. Falls kein Wert für schrittweite angegeben ist, wird standardmäßig 1 verwendet.

Folgende weitere Regeln gelten für For-Schleifen:

- **Verwendbare Datentypen:** Der Datentyp von zählvariable ist am besten Integer, kann jedoch ein beliebiger Typ sein, der die Operatoren für größer oder gleich ($>=$), kleiner oder gleich ($<=$), Addition (+) und Subtraktion (-) unterstützt. Es kann sogar ein benutzerdefinierter Typ sein, sofern er alle diese Operatoren unterstützt.

Die Ausdrücke start, ende und schrittweite ergeben normalerweise den Typ Integer, können jedoch einen beliebigen Datentyp ergeben, der implizit in den Typ von zählvariable umgewandelt werden kann. Sollten Sie für zählvariable einen benutzerdefinierten Typ verwenden, müssen Sie den CType-Konvertierungsoperator definieren (siehe Kapitel 19 – »Operatorenprozeduren«), um die Typen von start, ende oder schrittweite in den Typ von zählvariable zu konvertieren.

- **Deklaration der Zählvariable:** Wenn zählvariable nicht außerhalb dieser Schleife deklariert wurde, müssen Sie sie in der For-Anweisung deklarieren – durch lokalen Typrückschluss müssen Sie dabei standardmäßig den Typ nicht angeben. In diesem Fall ist der Gültigkeitsbereich (siehe »Gültigkeitsbereiche von lokalen Variablen« ab Seite 42) von zählvariable der Rumpf der Schleife. Sie können zählvariable jedoch nicht sowohl außerhalb als aus auch innerhalb der Schleife deklarieren.

- **Bestimmung der Anzahl von Iterationen:** Visual Basic wertet die Iterationswerte start, ende und schrittweite nur einmal aus, und zwar vor Beginn der Schleife. Wenn der Anweisungsblock ende oder schrittweite ändert, wirken sich diese Änderungen nicht auf die Iteration der Schleife aus.

- **Schachteln von Schleifen:** Sie können For-Schleifen schachteln, indem Sie eine Schleife in eine andere einfügen. Jede Schleife erfordert jedoch eine eindeutige schrittweite-Variable. Die folgende Konstruktion ist gültig.

```
For i As Integer = 1 To 10
    For j As Integer = 1 To 10
        For k As Integer = 1 To 10
            ' Insert statements to operate with current values of i, j, and k.
        Next k
    Next j
Next i
```

Sie können auch unterschiedliche Arten von Steuerungsstrukturen in anderen Steuerungsstrukturen schachteln. Weitere Informationen finden Sie unter »Geschachtelte Steuerungsstrukturen«.

TIPP Sie können die zählvariable bei einer Next-Anweisung optional angeben. Dies verbessert die Lesbarkeit des Programms unter Umständen bei häufigen und tiefen Verschachtelungen. Sie müssen dieselbe Variable angeben, die in der entsprechenden For-Anweisung vorhanden ist. Aber:

HINWEIS Wenn eine Next-Anweisung einer äußeren Schachtelungsebene vor der Next-Anweisung einer inneren Ebene auftritt, meckert der Compiler das an, und Sie sehen eine entsprechende Meldung in der Fehlerliste. Der Compiler kann diesen Überlappungsfehler einerseits nur erkennen, wenn Sie in jeder Next-Anweisung die entsprechende Zählvariable angeben, nimmt aber andererseits die jeweils richtige Zählvariable an, wenn Sie keine explizit benennen. Die letzte Variante ist deswegen oftmals die bessere.

Endlosschleifen

Mit Exit For kann auch ein Zustand vermieden werden, der eine so genannte Endlosschleife darstellt. Dabei handelt es sich um eine Schleife, die mit einer extrem großen oder unendlichen Anzahl von Wiederholungen ausgeführt werden kann. Wenn Sie eine solche Bedingung feststellen, können Sie Exit For verwenden, um die Schleife zu verlassen.

Verhalten

- **Schleifenbeginn:** Visual Basic wertet start, ende und schrittweite, wie oben schon erwähnt, nur einmal aus, und zwar zu Beginn der Ausführung der For/Next-Schleife. Dann wird der Wert von start an zählvariable zugewiesen. Bevor der Anweisungsblock ausgeführt wird, wird zählvariable mit ende verglichen. Wenn zählvariable bereits den Endwert überschritten hat, wird die For-Schleife beendet und die Steuerung an die Anweisung nach der Next-Anweisung übergeben. Andernfalls wird der Anweisungsblock ausgeführt.
- **Iterationen der Schleife:** Visual Basic erhöht bei jedem Auftreten der Next-Anweisung zählvariable um den Wert von schrittweite und kehrt dann zur For-Anweisung zurück. Die Variable zählvariable wird dann abermals mit ende verglichen, und wieder wird je nach Ergebnis der Block ausgeführt oder die Schleife beendet. Dieser Vorgang wird fortgesetzt, bis ende von zählvariable wertmäßig überschritten wird oder die Programmsteuerung zuvor auf eine Exit For-Anweisung trifft.

- Beenden der Schleife. Die Schleife wird erst beendet, wenn ende von zählvariable wertmäßig überschritten wurde. Wenn zählvariable gleich ende ist, wird die Schleife fortgesetzt. Der Vergleich, der bestimmt, ob der Block ausgeführt werden soll, lautet zählvariable <= ende, für den Fall, dass schrittweite positiv ist, und zählvariable >= ende, wenn schrittweite negativ ist.
- Ändern von Iterationswerten. Falls Sie den Wert von zählvariable während eines Schleifendurchlaufs ändern, wird unter Umständen das Lesen und Debuggen des Codes erschwert. Das Ändern des Werts von start, ende oder schrittweite wirkt sich nicht auf die Iterationswerte aus, die zu Beginn der Schleife bestimmt wurden.

Im folgenden Beispiel werden geschachtelte For/Next-Strukturen mit unterschiedlichen Werten für schrittweite veranschaulicht.

```
Dim words, digit As Integer
Dim thisString As String = ""
For words = 10 To 1 Step -1
    For digit = 0 To 9
        thisString &= CStr(digit)
    Next digit
    thisString &= " "
Next words
```

For Each-Schleifen

Mithilfe einer For/Each-Schleife wiederholen Sie eine Reihe von Anweisungen für jedes Element in einer Auflistung. Die generelle Anwendung von For/Each lautet:

```
For Each element [ As datentyp ] In auflistung
    [ Anweisungen ]
    [ Exit For ]
    [ Anweisungen ]
    [ Continue For ]
Next [ element ]
```

Die Variable `element` stellt dabei die Variable dar, die zum Durchlaufen der Elemente der Auflistung verwendet. Sie brauchen diese Variable nicht bei `Next` erneut anzugeben. `Anweisungen` beschreibt innerhalb der Struktur eine oder mehrerer Anweisungen zwischen `For Each` und `Next`, die für jedes Element in `auflistung` ausgeführt werden.

Die Anweisung `Exit For` ist optional und in der Regel selbst in einem konditionalen `If`-Block eingebettet. Sie überträgt die Steuerung aus der `For`-Schleife, beendet also die Schleife, auch wenn `zählvariable` ihren `ende`-Wert noch nicht erreicht hat.

Die Angabe von `Next` ist erforderlich, denn es beendet die Definition der `For`-Schleife.

Verwenden Sie eine `For Each/Next`-Schleife, wenn Sie für jedes Element einer Auflistung oder eines Arrays einen Satz von Anweisungen ausführen lassen wollen. Im Gegensatz dazu eignet sich eine `For/Next`-Anweisung besser, wenn jede Iteration einer Schleife einer Zählvariablen zugeordnet und der Anfangs- und Endwert der Variablen bestimmt werden kann. Für eine Auflistung ist das Konzept von Anfangs- und

Endwert jedoch bedeutungslos, und Sie kennen nicht unbedingt die Anzahl der Elemente in der Auflistung. In diesem Fall ist eine For Each/Next-Schleife besser geeignet.

HINWEIS Mehr zu Auflistungen finden Sie im Kapitel 22; dort wird in diesem Zusammenhang auch noch intensiver auf die Funktionsweise von For Each-Schleifen eingegangen.

Das folgende Beispiel zeigt, wie mithilfe einer For Each-Schleife durch die Steuerelementauflistung eines Formulars iteriert und die Hintergrundfarbe für alle Steuerelemente dabei auf Blau gesetzt wird.

```
'Allen Steuerelementen eines Formulars einen
'blauen Hintergrund verpassen
For Each steuerelement As Control In meinFormular.Controls
    steuerelement.BackColor = Color.Blue
Next
```

Do/Loop-und While/End While-Schleifen

Diese Schleifentypen wiederholten einen Anweisungsblock, entweder solange eine boolesche Bedingung True ist, bzw. bis die Bedingung True wird. Dabei wird im Zusammenhang mit Do das Schlüsselwort Until verwendet, um anzusehen, dass eine Bedingung True werden muss, während Sie While verwenden, um anzusehen, die Schleife laufen zu lassen, solange *wie* eine Bedingung True ist.

While und Until können dabei sowohl hinter dem Schleifenbeginn (Do) als auch beim Schleifenende (Loop) stehen. Der Zeitpunkt, zu dem die entsprechenden Bedingungen geprüft werden, wird damit auf den Schleifenbeginn oder auf das Schleifenende festgelegt.

Die Do While/Loop-Schleife entspricht prinzipiell der While/End While-Schleife. Es ergeben sich für die Platzierung der Abbruchbedingungen der Schleife folgende Möglichkeiten:

```
Do { While | Until } condition
    [ statements ]
    [ Exit Do ]
    [ statements ]
Loop
```

oder

```
Do
    [ statements ]
    [ Exit Do ]
    [ statements ]
Loop { While | Until } condition
```

oder

```
While condition
    [ statements ]
    [ Exit While ]
    [ statements ]
End While
```

Sowohl bei While/End While als auch bei Do/Loop dienen boolesche Variablen bzw. boolesche Ausdrücke als Abbruchbedingungen. Die folgenden Codezeilen zeigen ein paar Beispiele für den Einsatz dieser Schleifentypen.

```
Dim locCount As Integer

'Raufzählen.
Do While locCount < 10
    locCount += 1
Loop

'locCount ist jetzt 10; wieder runterzählen.
Do
    locCount -= 1
Loop Until locCount = 0

'locCount ist jetzt 0; wieder raufzählen.
While locCount < 10
    locCount += 1
End While

'locCount ist wieder 10; und wieder bis 0 runterzählen.
Do Until locCount = 0
    locCount -= 1
Loop
```

Exit – Vorzeitiges Verlassen von Schleifen

Die Exit-Anweisung überträgt die Steuerung direkt an die erste Anweisung nach dem Schleifenende – bei einer For-Schleife setzt die Programmausführung also nach der Next-Anweisung fort, bei einer Do-Schleife hinter Loop. Möglicherweise möchten Sie eine Schleife beenden, wenn Sie eine Bedingung feststellen, die das Fortsetzen des Durchlaufs unnötig oder unmöglich macht, z.B. ein fehlerhafter Wert oder eine Auforderung zum Beenden. Wenn Sie eine Ausnahme in einer Try/Catch/Finally-Anweisung abfangen, können Sie beispielsweise am Ende des Finally-Blocks Exit For verwenden.

Sie können eine beliebige Anzahl von Exit-Anweisungen an jeder Stelle einer Schleife einfügen. Exit wird häufig nach der Auswertung einer Bedingung verwendet, z.B. in einer If/Then/Else-Struktur.

Continue – Vorzeitige Schleifenwiederholung

Sie können auch eine Schleife vorzeitig »loopen« lassen – in einer For/Next-Schleife also in bestimmten Fällen das Next quasi vorziehen. Das funktioniert mit Continue. Sie können Continue natürlich auch bei allen anderen Schleifentypen einsetzen.

Anweisungen, die Programmcode bedingt ausführen – If/Then/Else und Select/Case

Der Boolean-Datentyp wird in der Regel bei der Auswertung von Entscheidungen benötigt. Mit ihm können Sie in Abhängigkeit seines Wertes steuern, ob Programmcode ausgeführt wird oder nicht. Dafür verwenden Sie die If-, Case- [Is], While- oder Until-Anweisungen.

Die IIf-Funktion steuert zwar nicht den Programmablauf, sollte aber auf Grund ihrer Ähnlichkeit zur If-Anweisung ebenfalls in diesem Zusammenhang erwähnt werden. Sie liefert ein Funktionsergebnis auf Grund des booleschen Wertes, der ihr übergeben wird. Ist der übergebene Wert True, wird das erste mögliche Funktionsergebnis zurückgeliefert, ist er False, das zweite.

If ... Then ... Else ... Elself ... EndIf

Die If-Anweisung haben Sie höchstwahrscheinlich schon hunderte Male angewendet und wissen deshalb aus dem Effeff, wie sie funktioniert. Der Vollständigkeit halber möchte ich sie dennoch ein wenig genauer unter die Lupe nehmen:

In der einfachsten Form wird bei der If-Anweisung der Code ausgeführt, der zwischen If und End If positioniert wird, wenn der hinter If stehende boolesche Ausdruck True wird. Obwohl Basic seit Jahren im Einsatz ist, gibt es immer noch Entwickler, die das Konzept von Vergleichen mit booleschen Ausdrücken nicht verinnerlicht und Schwierigkeiten beim Verständnis folgender Konstrukte haben:

```
locBoolean = True
If locBoolean Then
    'Wird nur ausgeführt, wenn locBoolean True ist.
End If
```

Einige Entwickler verstehen nicht, wieso hier nicht der folgende Ausdruck zum Einsatz kommen muss:

```
locBoolean = True
If locBoolean = True Then
    'Wird nur ausgeführt, wenn locBoolean True ist.
End If
```

Tatsache ist, das der Ausdruck

```
locBoolean = True
```

in dieser Bezeichnung keine Besonderheit der If-Anweisung ist, sondern im Prinzip eine ganz normale Funktion. Wenn locBoolean den Wert True hat, ist der gesamte Ausdruck natürlich ebenfalls wieder True. If macht nichts weiter, als den dahinter stehenden booleschen Wert zu untersuchen und die nachstehenden Anweisungen nur dann auszuführen, wenn der Wert True war. Aus diesem Grund braucht der Wert nicht noch zusätzlich durch Eingreifen des Programmierers überprüft zu werden; das wäre redundant. Denn das Gleichheitszeichen ist hier ja der Vergleichsoperator! Wenn man locBoolean durch seinen augenblicklichen Wert ersetzt, lautet die Überprüfung

```
If True = True Then ' (Wenn wahr wahr ist, dann)
```

was man ziemlich selbstsicher durch If True Then ... oder If locBoolean Then ... ersetzen kann (wenn es wahr ist, dann).

Es ist in Basic (nicht nur in Visual Basic) aber in der Tat verwirrend, dass Zuweisungsoperator und Vergleichsoperator mit denselben Zeichen angewandt werden. Der Ausdruck

```
Dim locBoolean = "Klaus" = "Uwe"
```

ist aber natürlich gültig. Das erste Gleichheitszeichen fungiert hier als Zuweisungsoperator, das zweite als boolescher Vergleichsoperator. In diesem Beispiel nimmt locBoolean den Wert False an, weil die Zeichenkette »Klaus« nicht »Uwe« entspricht. Der Vergleichsoperator hat vor dem Zuweisungsoperator die höhere Priorität. Andernfalls käme es bei diesem Beispiel auch zu einem Typkonvertierungsfehler.

Aber auch der andere Weg ist nicht wirklich überzeugend, C++ benutzt beispielsweise = für die Zuweisung, == für den Vergleich. Das ist aber alles andere als intuitiv. Allein 5% – 8% aller Fehler in C++ Programmen geht nämlich auf diese Verwechslung zurück. Und das ist viel häufiger als die Anzahl der Fehler, die durch die Falschbenutzung des = Zeichens in VB passieren.

Dem If-Codeblock kann auch ein Else-Codeblock folgen, der ausgeführt wird, wenn der boolesche Ausdruck hinter If den Wert False annahm. Darüber hinaus können Sie mit ElseIf weitere Auswertungen in das If-Konstrukt einschieben. Der Codeblock hinter dem letzten Else-Codeblock wird, falls vorhanden, nur dann ausgeführt, wenn keine der Bedingungen der einzelnen If- bzw. ElseIf-Sektionen True ergaben.

Ein Beispiel:

```
locString1 = "Klau's, und lass Dich nicht erwischen"
locString2 = "Klaus*"
locBoolean = (locString1 Like locString2) ' ergibt False

If locBoolean Then
    'Schachteln geht natürlich auch:
    If locString2 = "Klaus" Then
        Console.WriteLine("Namen gefunden!")
    Else
        Console.WriteLine("Keinen Namen gefunden!")
    End If
ElseIf Now = #12:00:00 PM# Then
    Console.WriteLine("Mittag!")
ElseIf Now = #12:00:00 AM# Then
    Console.WriteLine("So spät noch auf?")
Else
    Console.WriteLine("Es ist irgendwann sonst oder locString1 war nicht wie locString1...")
End If
```

Vergleichsoperatoren, die boolesche Ergebnisse zurückliefern

Visual Basic kennt die folgenden so genannten Vergleichsoperatoren, die zwei Ausdrücke miteinander vergleichen und ein boolesches Ergebnis zurückliefern:

- Ausdruck1 = Ausdruck2: Prüft auf gleich; liefert True zurück, wenn beide Ausdrücke gleich sind.
- Ausdruck1 > Ausdruck2: Prüft auf größer; liefert True zurück, wenn Ausdruck1 größer als Ausdruck2 ist.
- Ausdruck1 < Ausdruck2: Prüft auf kleiner; liefert True, wenn Ausdruck1 kleiner als Ausdruck2 ist.
- Ausdruck1 >= Ausdruck2: Prüft auf größer/gleich; liefert True zurück, wenn Ausdruck1 größer oder gleich Ausdruck2 ist.
- Ausdruck1 <= Ausdruck2: Prüft auf kleiner/gleich; liefert True, wenn Ausdruck1 größer oder gleich Ausdruck2 ist.
- Ausdruck1 <> Ausdruck2: Prüft auf ungleich; liefert True, wenn Ausdruck1 nicht Ausdruck2 entspricht.
- Ausdruck1 Is [Ausdruck2|Nothing]: Prüft auf Gleichheit eines Objektverweises (nur auf Referenztypen anwendbar); liefert True zurück, wenn Ausdruck1 auf den gleichen Datenspeicherbereich wie Ausdruck2 zeigt. Wenn Ausdruck1 keinem Speicherbereich zugewiesen ist (definierte Objektvariable aber kein instanziertes Objekt), liefert der Vergleich durch Is mit Nothing den booleschen Wert True zurück.
- Ausdruck1 IsNot [Ausdruck2|Nothing]: Prüft auf Ungleichheit eines Objektverweises (nur auf Referenztypen anwendbar); liefert True zurück, wenn Ausdruck1 auf einen anderen Datenspeicherbereich wie Ausdruck2 zeigt. Wenn Ausdruck1 auf einen gültigen Speicherbereich mit Instanzdaten verweist, liefert der Vergleich durch IsNot mit Nothing den booleschen Wert True zurück.
- String1 Like String2: Prüft auf Ähnlichkeit zweier Strings; ein Mustervergleich kann den Vergleich flexibilisieren. Bei vorliegender Gleichheit der beiden Strings nach bestimmten Regeln⁸ wird True zurückgeliefert, anderenfalls False.

Die folgenden Codezeilen demonstrieren den Einsatz der Vergleichsoperatoren:

```
Dim locString1 As String = "Uwe"
Dim locString2 As String = "Klaus"

locBoolean = (locString1 = locString2)      ' Ergibt False.
locBoolean = (locString1 > locString2)       ' Ergibt True.
locBoolean = (locString1 < locString2)       ' Ergibt False.
locBoolean = (locString1 >= locString2)      ' Ergibt True.
locBoolean = (locString1 <= locString2)      ' Ergibt False.
locBoolean = (locString1 <> locString2)     ' Ergibt True.
locBoolean = (locString1 Is locString2)       ' Ergibt False.

locString2 = "Uwe"
String.Intern(locString2)                  ' Ergibt jetzt True, da beide
locBoolean = (locString1 Is locString2)     ' Stringobjekte auf einen Bereich zeigen.

locString1 = "Klau's, und lass Dich nicht erwischen"
locString2 = "Klau*"
locBoolean = (locString1 Like locString2)    ' Ergibt True.
```

⁸ Genauereres erfahren Sie in der MSDN unter dem Schlagwort *Like-Operator*.

Select ... Case ... End Select

Sie können, wie im vorherigen Abschnitt zu sehen war, `ElseIf` zur Optionsanalyse verwenden, wenn Sie mehrere boolesche Ausdrücke in einem Rutsch prüfen und darauf mit der Ausführung entsprechender Programmcodes reagieren müssen. Mit einem `Select`-Konstrukt geht das allerdings sehr viel eleganter. `Select` bereitet einen Ausdruck für einen Vergleich mit booleschem Ergebnis vor; der eigentliche Vergleich findet dann durch verschiedene `Case`-Anweisungen, aber mindestens eine `Case`-Anweisung statt, hinter denen jeweils ein entsprechendes Vergleichsargument vom gleichen Typ (oder implizit konvertierbar) folgen muss. `Case Else` kann optional für die Ausführung von Anweisungen herangezogen werden, wenn keine der hinter `Case` angegebenen Bedingungen zutraf. `End Select` schließt das Konstrukt ab. Andererseits führt `Select` auch keine weiteren Überprüfungen durch, wenn einmal eine Bedingungsprüfung positiv verlief.

Bei der Bedingungsprüfung prüft `Case` ohne Zusatz auf Gleichheit. Durch Verwenden des `Is`-Schlüsselwortes können Sie auch andere Vergleichsoperatoren verwenden. Das folgende Beispiel soll den Umgang verdeutlichen:

```
Dim locString1 as String = "Miriam"

Select Case locString1
    Case "Miriam"
        Console.WriteLine("Treffer")
    Case Is > "M"
        Console.WriteLine("Name kommt nach 'M' im Alphabet")

    Case Is < "M"
        Console.WriteLine("Name kommt vor 'M' im Alphabet")

    Case Else
        Console.WriteLine("Name beginnt mit 'M'")

    'case like "Miri"
    'Das funktioniert nicht!!!

End Select
```

Allerdings werden hier Vergleichsoperation und bedingte Ausführung in einem Abwasch gemacht, sodass das folgende Konstrukt nicht funktioniert:

```
'Das funktioniert so nicht!!!
Select Case locBoolean

    case: Console.WriteLine("War wahr!")

End Select
```

Der Compiler meckert hier zu Recht.

Vereinfachter Zugriff auf Objekte mit With/End With

Mit With/End With können Sie eine Reihe von Anweisungen für ein bestimmtes Objekt ausführen, ohne den Namen des Objekts immer wieder neu nennen zu müssen. Wenn der vollqualifizierte Name für die entsprechende Objektvariable sehr lang ist, kann so durch die Verwendung von With/End With nicht nur Tipparbeit gespart sondern auch die Leistung verbessert werden. Außerdem senken Sie obendrein das Risiko einer falschen Schreibweise eines seiner Elemente.

Wenn Sie beispielsweise mehrere verschiedene Eigenschaften eines einzelnen Objekts verwenden möchten, fügen Sie die Anweisungen für die Eigenschaftenzuweisungen in eine With/End With-Struktur ein. Sie müssen dann nicht in jeder Eigenschaftenzuweisung auf das Objekt verweisen, sondern es genügt ein einziger Verweis auf das Objekt, indem Sie dem Eigenschaftennamen lediglich einen Punkt voranstellen:

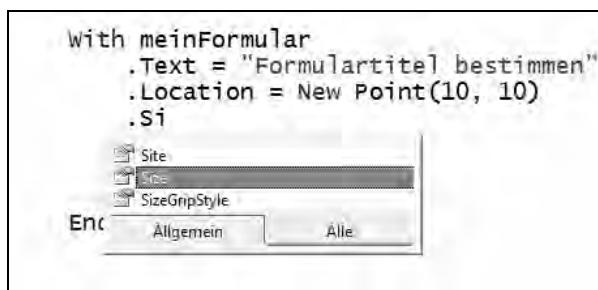


Abbildung 2.8 In einem With/End With-Block greifen Sie mit dem Punkt auf die Liste der Methoden und Eigenschaften des Objektes zu, das hinter With angegeben wurde

HINWEIS Denken Sie daran, dass auch With/End With eine Struktur bildet, und damit Variablen die zwischen With und End With definiert werden, nur in diesem Gültigkeitsbereich gelten. Der folgende Abschnitt hält mehr zu diesem Thema bereit.

Gültigkeitsbereiche von lokalen Variablen

Bei Variablen, die auf Procedurebene deklariert wurden (also innerhalb von Subs, Functions oder – dazu kommen wir später noch – auch Property-Prozeduren), gibt es Auswirkungen auf ihren Gültigkeitsbereich. Während in Visual Basic 6 beispielsweise noch Variablen innerhalb von Prozeduren überall ab dem Zeitpunkt ihrer Deklaration gültig waren, sind sie das in VB.NET ab dem Zeitpunkt ihrer Deklaration nur innerhalb der Struktur, in der sie definiert sind. »Struktur« in diesem Zusammenhang bedeutet im Prinzip irgendetwas, was Code in irgendeiner Form einklammern kann – das können beispielsweise If/Then/Else-Abfragen, For/Next- oder Do/Loop-Schleifen aber auch With-Blöcke sein. Es gilt dabei die Regel: Jede Struktur-anweisung, die dazu führt, dass der Editor den dazwischen platzierten Code einrückt, begrenzt automatisch den Gültigkeitsbereich von Variablen, die in diesem Block definiert sind. Ein Beispiel zeigt Abbildung 2.9.

```

Option Strict On

Module Module1

Sub Main()

    For zähler As Integer = 0 To 10
        Dim fünfGefunden As Boolean
        If zähler = 5 Then
            fünfGefunden = True
        End If
    Next
    If fünfGefunden Then
        Debug.Print("Die Zahl 5 kam in der Zahlenreihe vor!")
    End If
End Sub

End Module

```

Abbildung 2.9 Variablen sind nur in dem Strukturblock gültig, in dem sie deklariert wurden

Wie in der Abbildung zu sehen, versucht das Programm beim zweiten Mal auf fünfGefunden zuzugreifen, nachdem der Strukturbereich und damit auch der Gültigkeitsbereich der Variable verlassen wurde – und das führt zu einem Fehler.

Wollten Sie fünfGefunden in jedem Strukturblock zugreifbar machen, müssten Sie folgende Änderungen vornehmen:

```

Sub Main()

    Dim fünfGefunden As Boolean

    For zähler As Integer = 0 To 10
        'Dim fünfGefunden As Boolean
        If zähler = 5 Then
            fünfGefunden = True
        End If
    Next

    If fünfGefunden Then
        Debug.Print("Die Zahl 5 kam in der Zahlenreihe vor!")
    End If
End Sub

```

Die Änderungen sind hier im Listing durch Fettschrift gekennzeichnet.

Diese Regel für Gültigkeitsbereiche führt dazu, dass Variablen innerhalb mehrerer Strukturblöcke mehrfach unter gleichem Namen deklariert werden können, wie das folgende Beispiel zeigt:

```

Sub Main()
    For zähler As Integer = 0 To 10
        Dim fünfGefunden As Boolean
        If zähler = 5 Then
            fünfGefunden = True
        End If
    Next

```

```

Next
For zähler As Integer = 0 To 10
    Dim fünfGefunden As Boolean
    If zähler = 5 Then
        fünfGefunden = True
    End If
Next
End Sub

```

Hier wird die Variable `fünfGefunden` innerhalb einer Prozedur zweimal deklariert – dennoch meldet Visual Basic keinen Fehler, weil sich beide Deklarationen in unterschiedlichen Gültigkeitsbereichen befinden.

Allerdings:

```

Option Strict On

Module Module1
    Sub Main()
        For zähler As Integer = 0 To 10
            Dim fünfGefunden As Boolean
            If zähler = 5 Then
                fünfGefunden = True
            End If
        For zweiterZähler As Integer = 0 To 10
            Dim Fünfgefunden As Boolean
                If zähler = 5 Then
                    fünfGefunden = True
                End If
            Next
        Next
    End Sub
End Module

```

Abbildung 2.10 Variablen eines übergeordneten Gültigkeitsbereichs dürfen solche eines untergeordneten nicht verbergen

Das Deklarieren von Variablen gleichen Namens in einem Gültigkeitsbereich, der einen anderen kapselt, funktioniert natürlich nicht, denn Variablen in einem übergeordneten Gültigkeitsbereich sind ohnehin immer von einem untergeordneten aus zugreifbar. Eine entsprechende Fehlermeldung würde, falls Ihnen das passiert, dann so ausschauen, wie in Abbildung 2.10 zu sehen.

HINWEIS Mehr zu Gültigkeitsbereichen anderer Elemente und in anderen als lokalen Programmgebieten erfahren Sie in Kapitel 14.

Kapitel 3

Einführung in das .NET Framework

In diesem Kapitel:

Was ist .NET und aus was besteht es?	46
Was ist eine Assembly?	47
Erzwungene Typsicherheit und Deklarationszwang von Variablen	55
Namensgebung von Variablen	58
Und welche Sprache ist die beste?	60

.NET bzw. .NET Framework gehören in der IT-Branche zu den mit am Meisten nicht verstandenen Begriffen. Und daran ist Microsoft nicht ganz unschuldig, denn offensichtlich haben die Marketingmeister diesen Begriff seinerzeit ebenfalls so gründlich nicht verstanden, dass sie ihn für alles und jeden Zweck verwendet haben, der ihnen nur in den Sinn kommen konnte, als sich die erste Version des .NET Frameworks seiner ersten Einsatzfertigkeit näherte.

So hieß die erste Version von Windows Server 2003 noch .NET-Server, und das Einzige, was man hier in Sachen .NET finden konnte, war ein vorinstalliertes .NET Framework. Das UI (*User Interface*, etwa *Benutzerschnittstelle*) des Betriebssystems selbst stützte sich mit keiner Codezeile auf dieses Framework, und außer, dass das Framework in der damaligen Version 1.1 vorinstalliert war, existierte nur der Internet Informations Server, der auf Basis dieser Framework-Version das Hosten von ASP.NET-Internet-Sites zuließ. Immerhin.

Fragte man zu der damaligen Zeit, was .NET eigentlich sei und aus welchen Komponenten es bestünde, bekam man alle möglichen Antworten, und nicht unbedingt welche, die der Wahrheit sehr nahe kamen.

OK – dann wollen wir damit mal beginnen, was also was ist .NET?

Was ist .NET und aus was besteht es?

.NET ist eine Sammlung verschiedener Technologien, die auf einer gemeinsamen Infrastruktur beruhen, und die Softwareentwicklern die Möglichkeit bieten, sichere, stabile, einfach zu pflegende, auf mehreren Plattformen laufbare Software sowohl Webbrowser-orientiert als auch als SmartClient zu entwickeln.

Es gibt eine ganze Menge Begriffe im Zusammenhang mit dem .NET Framework, mit denen gerade auf Fachtagungen, Entwicklerkonferenzen und unter Entwicklerkollegen gerne herumgeworfen wird – oft ohne dass den Werfern die Bedeutung so richtig klar zu sein scheint. Selbst erfahrenen Programmierern erschließt sich der Hintergrund bestimmter .NET-Fachbegriffe auf Anhieb nicht ohne weiteres, und die Erfahrung zeigt, dass es auch eine ganze Menge Entwickler gibt, die bestenfalls über Halbwissen verfügen, und von denen man entweder nur Halbwahrheiten oder sogar völlig falsche Informationen bekommt.

Und dann gibt es natürlich noch eine besonders erwähnenswerte Eigenschaft der ganzen .NET Framework-Plattform, die das Kompilieren von Anwendungen betrifft. Im Gegensatz zu herkömmlichen Programmiersprachen produziert nämlich keiner der .NET-Sprachen-Compiler, und damit auch nicht der Visual Basic Compiler, direkt vom Prozessor ausführbare Maschinenbefehle, sondern einen Zwischencode, der erst zur Laufzeit der Anwendung in nativen Prozessorcode umgesetzt wird. Damit werden Anwendungen einerseits plattformunabhängig und andererseits können Optimierungen auf das jeweilige System vorgenommen werden, was nicht möglich wäre, würde das Programm schon im Vorfeld in Maschinensprache übersetzt.

Die folgenden Abschnitte sollen deshalb die wichtigsten .NET-Fachbegriffe und Technologien vorstellen und kurz erklären. Und keine Angst: Auch wenn einen die vielen Abkürzungen und Akronyme anfangs verwirren, so ist das dahinter stehende Konzept genial, schlüssig, und im Grunde genommen auch ganz einfach zu verstehen. Und spätestens, wenn Sie die folgenden Abschnitte gelesen haben, werden Sie einen roten Faden erkennen, der sich elegant durch das ganze .NET-Konzept zieht.

Was ist eine Assembly?

Wenn Sie ein Programm unter Windows schreiben, dann wandelt ein Compiler im einfachsten Fall Ihren Quellcode schlussendlich in eine ausführbare .EXE-Datei um. Bei größeren Projekten bietet es sich an, Funktionen, die von einzelnen Teilprogrammen immer wieder verwendet werden, in so genannten DLLs¹ zusammenzufassen und von außen aufzurufen – einfach um Redundanz durch doppelte Funktionen zu vermeiden und letztendlich Platz zu sparen. Eine DLL kann man sich als eine Art Bibliothek (daher auch der Name) vorstellen, die zwar alleine nicht lauffähig ist, die in ihr enthaltenen Funktionen aber beliebig vielen anderen EXE-Dateien zur Verfügung stellt.

Unter .NET funktioniert das im Großen und Ganzen genauso. Ein entscheidender Unterschied ist jedoch, wie die .EXE-Dateien bzw. DLLs tatsächlich vorliegen (dazu liefert der kommende *JITter*-Abschnitt tiefere Einblicke), und mit welchem Oberbegriff diese bezeichnet werden: nämlich als *Assembly*.

Eine ausführbare .NET-EXE-Datei ist eine Assembly. Eine .NET-DLL ist auch eine Assembly. Eine Assembly ist im Grunde genommen also nichts weiter als eine direkt (EXE) oder indirekt (DLL) ausführbare Einheit, die Programmcode enthält. Eine Assembly ist also zunächst die kleinste auslieferbare Einheit einer .NET Programmierung.

Hinter dem Konzept von Assemblies verstecken sich zugegebenermaßen noch kompliziertere Konzepte und weitaus mehr Möglichkeiten. Doch für den täglichen Umgang mit .NET-Technologien ist die hier gegebene Beschreibung völlig ausreichend.

Was ist ein Namespace?

Namespaces dienen in erster Linie zur thematischen Ordnung von Klassen innerhalb von Assemblies. Namespaces haben überhaupt keinen Einfluss auf den Namen einer Assembly sondern nur auf den vollqualifizierten Namen einer Klasse oder Struktur *innerhalb* einer Assembly. Namespaces haben keinen Einfluss auf Dateinamen (anders als Assemblies), sie sind also eine abstrakte Größe.

Die Definition von Namespaces dient also in erster Linie genau zu dem, wozu Kapitel in einem Buch dienen. Würden die einzelnen Abschnitte eines Buchs nur »lose im Raum« stehen, litte die Übersicht darunter ganz gewaltig. Genauso verhält es sich bei Objekten. Der Name des Objekts `AdressenDetails` beispielsweise sagt nur ungefähr etwas darüber aus, welchem Zweck es dient. Befindet sich das Objekt im Namespace `MeineFibu.Lieferanten.AdressenDetails`, so ist eine schon größere Ordnung hergestellt – das Wiederfinden des »richtigen« Objektes wird einfacher und auch die Möglichkeit der Existenz eines weiteren Objektes namens `AdressenDetails` ist ohne Mehrdeutigkeitsprobleme denkbar, wenn es sich in einem anderen Namespace wie beispielsweise `MeineFibu.Kunden.AdressenDetails` befindet.

Namespaces können sich durchaus über mehrere Assemblies erstrecken. Es also denkbar, zwei Klassen des Namespace `MeineFibu.Lieferanten` in einer und zwei weitere Klassen des gleichen Namespace in einer anderen Assembly unterzubringen. Namespaces und Assemblies sind, was das anbelangt, voneinander völlig unabhängig.

¹ *Dynamic Link Libraries*, etwa: *dynamisch verbindbare Bibliotheken*.

Die FCL, die alle Klassen des Frameworks enthält (genaue Erklärung folgt), macht von Namespaces regen Gebrauch. Klassen für die Formularsteuerung befinden sich beispielsweise im Namespace `System.Windows.Forms`. Dieser Namespace ist aber nicht nur in der Assembly `System.Windows.Forms` definiert, in der sich die meisten und wichtigsten Objekte für die Formularsteuerung befinden. Hier ist es nur durch die Entscheidung der Microsoft-Programmierer so geschehen, dass Namespace und Assembly gleiche Namen tragen. Das muss aber, wie schon gesagt, nicht so sein: Andere Objekte des gleichen Namespaces (`System.Windows.Forms`) sind beispielsweise auch in der Assembly `System.dll` vorhanden.

HINWEIS Namespaces ermöglichen zwar einerseits das »saubere« Definieren von Klassen mit gleichem Klassennamen; dabei kann es unter Umständen aber auch passieren, dass ein bereits importierter Namespace eine Klasse oder Struktur des System-Namespace verbirgt und Sie deswegen nicht mehr darauf zugreifen können. Das `Global`-Schlüsselwort sorgt dann für Abhilfe. Genaueres zu diesem Thema finden Sie im entsprechenden Abschnitt in Kapitel 10 im Abschnitt »Zugriff auf den Framework-System-Namespace mit Global«.

Einbinden von Namespaces und Assemblies in Codedateien und Projekten am Praxisbeispiel

Und nachdem wir uns die Theorie um Namespaces und Assemblies einverleibt haben, schauen wir uns an, wie das Ganze in der Praxis ausschaut. Dazu folgendes Szenario: Wir erstellen eine Konsolen-Anwendung, (*Datei/Neu/Projekt* und aus dem Dialog, der jetzt erscheint, unter *Visual Basic* und *Windows* die Vorlage *Konsolenanwendung* auswählen. Nennen Sie das Beispielprojekt etwa *NamespaceDemo*) von der aus wir eine `MessageBox`, also ein typisches Windows-Nachrichtenfeld aufrufen wollen.

Wenn Sie versuchen, den Namen der `MessageBox`-Klasse einzugeben, quittiert Ihnen der Compiler diesen Versuch mit einer Fehlermeldung, etwa wie in der nachstehenden Grafik zu erkennen:

```

Module Module1
    Sub Main()
        MessageBox.Show("Ein Nachrichtenfeld anzeigen")
    End Sub
End Module

```

The screenshot shows a code editor window with the following VB.NET code:

```

Module Module1
    Sub Main()
        MessageBox.Show("Ein Nachrichtenfeld anzeigen")
    End Sub
End Module

```

A tooltip appears over the word `MessageBox` with the text "Der Name 'MessageBox' wurde nicht deklariert." (The name 'MessageBox' was not declared.)

Abbildung 3.1 Eine Methode kann nur auf die Objekte zugreifen, die seine eigene Assembly selbst definiert oder die in Assemblies liegen, auf die sie verweist

Dass der Compiler `MessageBox` nicht kennt liegt einfach daran, dass die `MessageBox`-Klasse in einer Assembly des Frameworks definiert wird, die standardmäßig nicht in Konsolenanwendungen eingebunden ist. Ihre Aufgabe ist es in solchen Fällen deswegen, eine Referenz auf die Assembly zu setzen, die die entsprechende Klasse enthält – in unserem Fall ist das die Assembly `System.Windows.Forms.dll`.

1. Fahren Sie dazu mit dem Mauszeiger auf den Projektnamen (*nicht* Projektmappen-Namen; Sie fahren bei einer Ein-Projekt-Projektmappe also auf den *zweiten* Eintrag von oben!).
2. Klicken Sie die rechte Maustaste, um das Kontextmenü zu öffnen.

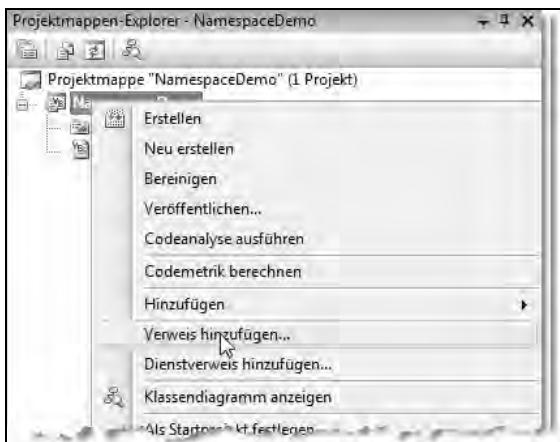


Abbildung 3.2 Sie fügen Ihrem Projekt einen Verweis auf eine zusätzliche Assembly hinzu, indem Sie im Projektmappe-Explorer das Kontextmenü öffnen und Verweis hinzufügen anklicken...

3. Wählen Sie den Eintrag *Verweis hinzufügen...* aus.
4. Im Dialog, der jetzt erscheint, wählen Sie auf der ersten Registerkarte den Eintrag *System.Windows.Forms* aus, etwa wie in der folgenden Abbildung zu sehen.

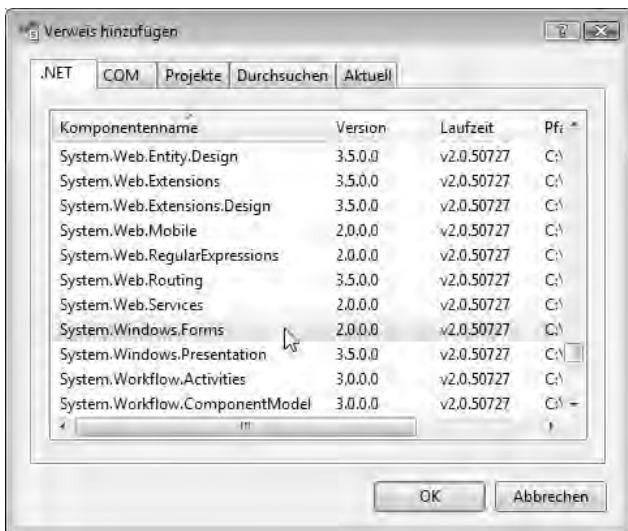


Abbildung 3.3 In diesem Dialog wählen Sie die Assembly aus, auf die Ihr Projekt (und damit die aktuelle Assembly) verweisen soll

5. Klicken Sie auf OK, um den Dialog zu verlassen.

An der Fehlermeldung hat sich nichts geändert, aber immerhin haben Sie die Möglichkeit, auf die MessageBox-Klasse zuzugreifen. Der Namespace vieler Klassen lautet nämlich oftmals genau so, wie die Assembly selbst – auf jedem Fall ist das in unserem Beispiel so. Wenn Sie also den MessageBox-Klassennamen und seinen vollqualifizierten Namen austauschen (das ist der, der die Namespace-Angabe mit enthält), ist die Fehlermeldung schon einmal ausgemerzt, wie die folgende Abbildung zeigt:

```

Module Module1
    Sub Main()
        System.Windows.Forms.MessageBox.Show("Ein Nachrichtenfeld anzeigen")
    End Sub
End Module

```

Abbildung 3.4 Mit dem vollqualifizierten Namen können Sie eine Klasse in einer per Verweis referenzierten Assembly *immer* erreichen!

Natürlich würden Sie sich die Finger wund tippen, müssten Sie beim Verwenden einer Methode der Klassen einer externen Assembly ständig deren vollqualifizierten Namen ausschreiben, zumal Sie in der Regel nicht nur eine Klassen sondern viele Klassen einer eingebundenen Assembly verwenden.

Die Imports-Anweisung erlaubt es deswegen, einen Namespace für eine Codedatei global zu importieren. Wie diese funktioniert, zeigt die folgende Abbildung.

```

Imports System.Windows.Forms

Module Module1
    Sub Main()
        MessageBox.Show("Ein Nachrichtenfeld anzeigen")
    End Sub
End Module

```

Abbildung 3.5 Mit der Imports-Anweisung binden Sie einen Namespace global für eine Codedatei ein

Aber auch das kann auf Dauer zu einer zusätzlichen Tippbelastung führen, deswegen kennt Visual Basic auch die Möglichkeit, Namespaces global für ein Projekt einzubinden.

1. Dazu öffnen Sie abermals, wie unter Abbildung 3.2 gezeigt, das Kontextmenü des Projektes.
2. Wählen Sie aber dieses Mal den Menüpunkt *Eigenschaften* aus.
3. Im Dialog, den Visual Studio jetzt als Registerdokument öffnet, wählen Sie die Registerkarte *Verweise* aus. Auf dieser Registerkarte finden Sie einerseits die Verweise auf alle Assemblies, die Ihrem Projekt hinzugefügt wurden, und deren Klassen sie deswegen verwenden können, sowie andererseits eine Liste mit den vorhandenen und vorimportierten Namespaces, die diese Assemblies zur Verfügung stellen. Die Assemblies, vor denen Sie ein Häkchen sehen, sind vorimportiert – auf deren Klassen können Sie also ohne weiteres zugreifen. Damit ersparen Sie sich die entsprechenden Imports-Anweisungen am Anfang der betroffenen Codedateien.

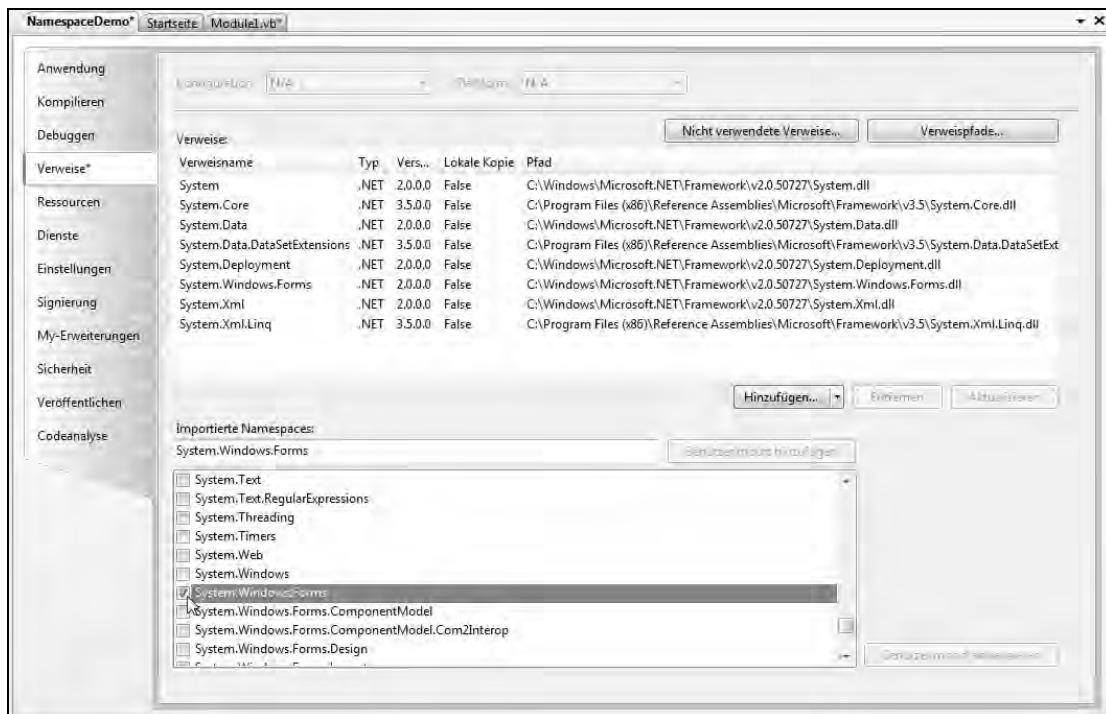


Abbildung 3.6 In den Projekteigenschaften finden Sie auf der Registerkarte *Verweise* eine Referenz aller Assemblies sowie der Namespaces, die diese Assemblies zur Verfügung stellen. Die angeklickten Namespaces werden automatisch vorimportiert

HINWEIS Da das Konzept von Assemblies und Namespaces insbesondere für VB6-Umsteiger oft Verständnisschwierigkeiten bereitet, finden Sie im Kapitel 10 weitere Informationen und Beispiele zu diesem Thema.

Was versteckt sich hinter CLR (Common Language Runtime) und CLI (Common Language Infrastructure)?

Die *Common Language Runtime* besteht unter anderem aus der Basis-Assembly des .NET Frameworks (die Assembly selbst nennt sich *mscorlib.dll*). Sie bildet den unteren Layer, sozusagen das Fundament, auf dem alle anderen Objekte des Frameworks ruhen. Diese Assembly wird als *Base Class Library* (s. u.) bezeichnet. Diese BCL ist aber nur ein Teil der CLR, denn letztere hat weitere, genauso wichtige Aufgaben. So stellt sie beispielsweise die *JITter*-Funktionalität (s. u.), die dafür sorgt, dass aus .NET-Assemblies, die zunächst in der *Intermediate Language* vorliegen (*M[S]IL*)², kurz vor der Ausführung in nativen Assembler-Code kompiliert werden (*JIT* steht für *Just in Time*, etwa: *genau rechtzeitig*).

² Einige Microsoftler verwenden die Abkürzung *MSIL*, einige *MIL* und einige nur *IL*.

Eine ebenfalls sehr wichtige Rolle übernimmt die CLR in Form der .NET Framework-eigenen »Müllabfuhr«, dem so genannten *Garbage Collector*, den sie ebenfalls implementiert. Anders als bei C oder C++ müssen Sie sich bei der Entwicklung mit dem Framework nicht um das Aufräumen Ihres Datenmülls kümmern, um Objekte also, die Sie verwendet haben, und die Sie ab einem bestimmten Zeitpunkt nicht mehr benötigen. Der Garbage Collector stellt selbständig fest, ob ein Objekt in ihrem Programmcode noch benötigt (referenziert) wird und entsorgt es, wenn es nicht mehr verwendet wird.

Schließlich sorgt die CLR mithilfe ihrer *Execution Engine*, die sich in der DLL *mscoree.dll* verbirgt, dass Ihre .NET-Programme überhaupt zur Ausführung kommen. Aufgabe der Execution Engine ist es, die vergleichsweise neue .NET-Technologie zusammen mit dem Vorgang des Just-in-Time-Kompilierens an die Startvorgänge eines herkömmlichen Programms unter Windows anzupassen.

Die CLR selbst basiert auf der so genannten CLI (der *Common Language Infrastructure*) – »der Spezifikation eines internationalen Standards für das Erstellen von Entwicklungs- und Programmausführungsumgebungen«, einem Standard, der definiert, dass verschiedene Programmiersprachen (respektive der Code, den Sie generieren) und verschiedene Bibliotheken nahtlos zusammenarbeiten können. Dieses Konzept ermöglicht es, dass Sie unter .NET Projekte entwickeln und dabei mit verschiedenen Programmiersprachen gleichzeitig arbeiten können. So wäre es beispielsweise denkbar, finanzielle Programmabibliotheken in Visual Basic zu formulieren und diese von Ihrer eigentlichen Windows-Anwendung aufzurufen, die Sie in C# entwickelt haben.

Was ist die FCL (Framework Class Library) und was die BCL (die Base Class Library)?

Erst einmal der Ursprung für einen Haufen falscher Erklärungen. Es gibt Experten, die meinen, die FCL und BCL seien dasselbe. Und ist das richtig? Nein. Andere differenzieren schon detaillierter und erklären die BCL zur Oberkategorie der CLR. Haben sie Recht? Nein.

Die BCL ist *Teil* der Common Language Runtime, und sie enthält alle Basisobjekte, die Sie während Ihres Entwicklungsalltags benötigen. Sie definiert u. a. alle primitiven Datentypen und implementiert damit das *Common Type System* (s. u.) in Form von verwendbaren Objekten und Typen, von denen sich die meisten im System-Namespace bzw. in der Assembly *mscorlib.dll* befinden. Objekte und Methoden aus der BCL werden Sie beim Entwickeln unter .NET wohl am häufigsten verwenden. Egal, ob Sie eine Variable eines primitiven Datentyps verwenden, große Datenmengen in Arrays speichern, mit Zeichenketten hantieren oder reguläre Ausdrücke verwenden – die dafür benötigten Klassen und Methoden befinden sich alle in der BCL.

Die Base Class Library ist weitestgehend plattformunabhängig formuliert; sie ist standardisiert (denn sie basiert auf der durch die ECMA zertifizierten CLI), und deswegen ist ihr Quellcode gegenwärtig in der Version 1.0 im Rahmen der freien CLI-Implementierung namens *Rotor* auch frei verfügbar und vergleichsweise leicht portierbar. Lauffähige Implementierungen der CLI gibt es derzeit unter MacOS, FreeBSD und – natürlich – Windows.³

³ Das Projekt »MONO« unter Linux geht übrigens noch einen ganzen Schritt weiter und implementiert eine komplette FCL, die in vielen Bereichen sogar kompatibel zur Microsoft FCL ist.

Übrigens: Das Team, das für die Implementierung der Base-Class-Bibliotheken gemäß der CLI bei Microsoft zuständig ist, nennt sich selbst immer noch das *Base-Class-Library-Entwicklungsteam*. Dieses Team hat aber nichts mit der Implementierung etwa von Datenbankfunktionen, der Windows Forms-Implementierung oder anderen Funktionsbereichen am Hut, die quasi »erst auf der CLR« (und damit auf der BCL) liegen.

Das heißt im Klartext:

- Die BCL ist das CLI-Pendant unter Windows und damit Teil der CLR.
- Die FCL fasst *alle* .NET-Funktionsbereiche unter einem Namen zusammen.

Was ist das CTS (Common Type System)?

Das CTS bildet eine Richtlinie für die Implementierung von Datentypen und Datentypkonzepten unter der CLI. Um es mit den bisher vorgestellten Akronymen zu sagen: Die BCL der CLR setzt das CTS unter Beachtung der CLI um (wenn das kein Satz zum Angeben ist! Vor der Benutzung im Rahmen erotischer Kontakte zur Darstellung der eigenen Fähigkeiten ist aber dringend abzuraten).

Im Klartext heißt das: Das Common Type System definiert, wie Typen deklariert, verwendet und in der CLR gemanagt werden, und sie spielt darüber hinaus einen wichtigen Part bei der Integration der verschiedenen .NET-Sprachen durch die CLR. Sie realisiert das durch unbedingte Typsicherheit (sie können nicht ohne weiteres einer Integervariablen eine Zeichenkette zuweisen) und garantiert eine hohe Performance bei der Codeausführung. Schließlich definiert sie einen festen Satz an Richtlinien, denen die verschiedenen .NET-Sprachen folgen müssen, und sie stellt damit sicher, dass Objekte, die in einer einen .NET-Sprache entwickelt worden sind, sich in einer anderen .NET-Sprache verwenden lassen.

Dies ist ein gewaltiger Fortschritt gegenüber früheren Technologien von Microsoft (wie z.B. COM) zum Datenaustausch zwischen unterschiedlichen Programmen und Programmierumgebungen. Wenn früher ein Visual Basic Programm (unter VB 6.0) eine Zeichenfolge (string) an eine Anwendung übergeben wollte, die beispielsweise in C/C++ geschrieben worden war (was sofort der Fall war, wenn man in VB Funktionen von Windows nutzen wollte), musste man sehr genau verstehen, was man tat – eine Zeichenfolge in C ist etwas anderes als eine Zeichenfolge in VB, sonst kam es zu merkwürdigen Effekten bis hin zum Absturz der Anwendung.

Durch das Konzept des Umsetzens dieser Richtlinien ergibt sich »ganz nebenbei« ein weiterer, wirklich nicht unwesentlicher Vorteil, den Sie im folgenden Abschnitt beschrieben finden.

Was ist CIL/MSIL (Microsoft-Intermediate Language) und wozu dient der JITter?

Auch bedingt durch das Konzept und die Reglementierung, die das CTS vorschreibt, übersetzen die Compiler der verschiedenen .NET-Programmiersprachen ihren Quellcode nicht direkt in Maschinensprache, der vom Prozessor eines Computers verstanden wird. Vielmehr werden Ihre Programme zunächst in eine Zwischensprache umgewandelt – in die so genannte *Common Intermediate Language* oder kurz: *CIL*. »Zwischensprache« deshalb, weil sie einerseits schon abstrakter und eine Ebene höher als eine Prozessor-Maschinensprache angesiedelt ist, andererseits dennoch viel prozessornäher als eine vollwertige Hochsprache

konzipiert ist, wie beispielsweise Visual Basic oder C#. Auch die CIL beschreibt wieder in Anlehnung an die CLI das grundsätzliche Konzept und stellt Technologieträger dar; die konkrete Technologie, die von Microsoft zur Auslieferung für die Windowsplattform kommt, nennt sich *MSIL* (*Microsoft Intermediate Language*).

Eine Assembly enthält also in der Regel keine Befehle, die ein Prozessor direkt verstehen könnte, sondern Programmcode, aus dem erst der so genannte *Just-in-Time-Compiler* (der JITter) zur Laufzeit das eigentliche Maschinenprogramm erzeugt. Das hört sich zunächst nach einem möglichen Performance-Problem an, ist es aber nicht.⁴ Der JITter ist so optimiert, dass er nur jeweils die Methoden einer Klasse kompiliert, die als nächstes benötigt werden. Und einiges an Zeit, das zunächst durch das Kompilieren der Methoden geopfert werden muss, fahren Ihre Programme schlussendlich wieder ein, da der JITter viel effizienteren Code als jeder andere Compiler erzeugen kann, denn:

Der JITter kennt die Maschine, auf der der zu kompilierende Code laufen wird. Ein herkömmlicher Compiler kennt sie nicht, denn die Maschine, auf der eine Anwendung entwickelt und auch schon kompiliert wird, ist üblicherweise eine ganz andere als die, auf der diese laufen wird. Der JITter kann also auf die Besonderheiten eines Prozessors eingehen. Erkennt er beispielsweise, dass sich im Computer, auf dem eine .NET-Anwendung laufen soll, ein Pentium-4-Prozessor vorhanden ist, kann er den zu erzeugenden Code für diesen Prozessor optimieren. Ein herkömmlicher Compiler jedoch muss immer Maschinencode erzeugen, der nur durchschnittlich gut optimiert ist, bei dem aber gewährleistet ist, dass er auf allen kompatiblen Prozessoren (P3, P4, Athlon, etc.) funktioniert. Es gibt eine Vielzahl weiterer Optimierungsmöglichkeiten für den JITter, auf die ich an dieser Stelle nicht näher eingehen will.

HINWEIS Eine Ausnahme bildet übrigens die FCL selbst. Sie ist vorkompliiert, sodass nicht die Notwendigkeit besteht, auch noch das komplette Framework (oder zumindest die Teile, die eine Anwendung benötigt) bei jedem Start der Anwendung zu *JITten*. Damit könnte der Eindruck entstehen, dass die FCL, anders als Ihre eigenen Programme, vielleicht nicht optimal auf Ihr System »eingestellt« ist – denn wäre sie schon bei Microsoft hausintern vorkompliiert, könnte sie ja unmöglich auf Ihr System optimiert sein. Dem ist aber nicht so, denn: Möglicherweise ist Ihnen im Laufe Ihrer Arbeit mit .NET schon aufgefallen, dass die Installation des einige zig Megabyte großen Frameworks selbst auf einem flotten Rechner zwar immer noch schnell vorstatten geht, aber auffällig länger als erwartet dauert. Das liegt daran, dass die FCL bei ihrer Installation nicht nur in die entsprechenden Verzeichnisse *kopiert*, sondern quasi dort hinein *kompliiert* wird. Die Installationsdateien der FCK im Framework Setup liegen nämlich als MSIL vor. Erst wenn Sie sie auf einem Rechner installieren, werden sie in nativen Maschinencode übersetzt und dann – optimiert auf Ihren Rechner – in den Zielverzeichnissen eingerichtet.

Und einen weiteren Vorteil hat diese Vorgehensweise, denn: Programme, die Sie unter dem .NET Framework entwickeln, werden prozessorunabhängig. Eine Anwendung, die Sie mit dem Zieltyp *Any* erstellen (etwa: *zielt ab auf eine beliebige durch .NET unterstützte Architektur*) läuft auch prozessorunabhängig. Entwickeln Sie also ein Projekt mit dem Zieltyp *Any*, wird es beispielsweise unter einem 32-Bit-Windows XP laufen, aber auf einem 64-Bit-Windows Vista auch mit der vollen 64-Bit-Unterstützung.

⁴ O.k., na gut, sagen wir: In den seltensten Fällen. O.k., na gut, sagen wir: In der Regel nicht.

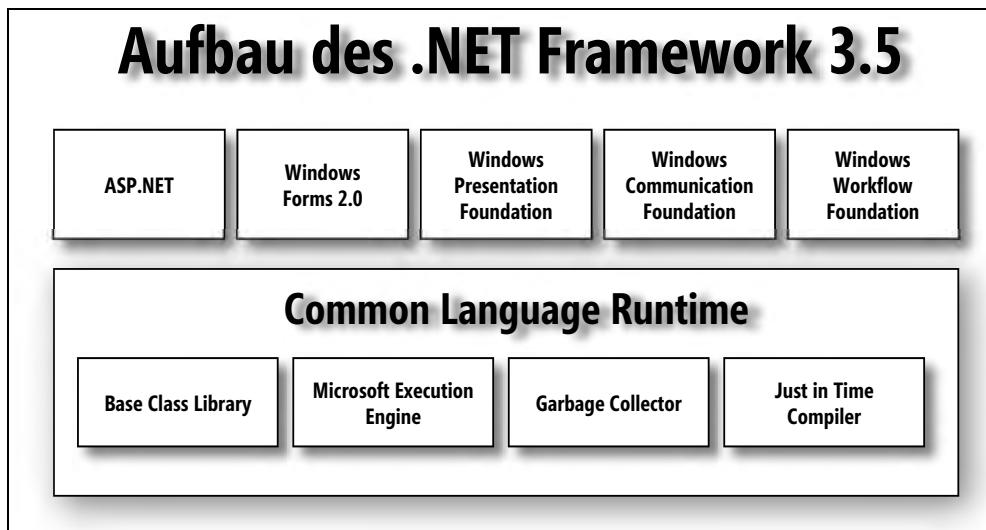


Abbildung 3.7 Diese Grafik stellt den Aufbau des .NET Framework bildlich dar

Erzwungene Typsicherheit und Deklarationszwang von Variablen

Visual Basic bietet die Möglichkeit, Programme zu entwickeln, die weder typsicher sind noch einen Variablen-deklarationszwang verlangen (Option Strict Off sowie Option Explicit Off). Ich kann verstehen, dass Microsoft die Entscheidung, diese »Eigenarten« bis in die aktuelle Visual Basic-Version zu belassen, höchstwahrscheinlich aus Kompatibilitätsgründen getroffen hat. Meine Meinung zu diesem Thema ist allerdings sehr rigoros: Ich lehne diese »Features« absolut ab, denn sie kosten enorm viel Programmausführungszeit und führen darüber hinaus zu schwer findbaren Fehlern. Zwar kann es sinnvoll sein, Objekte, deren Typ Sie nicht kennen, erst zur Laufzeit zu untersuchen und zu manipulieren, allerdings bietet das Framework dazu ein viel leistungsfähigeres Werkzeug mit der so genannten *Reflection* an. Über dieses Thema können Sie sich in Kapitel 25 informieren. Es gibt allerdings bestimmte Szenarien, bei denen das so genannte Late-Binding durchaus Sinn ergeben kann – wenn beispielsweise bei der Web-Entwicklung Objekte angesprochen werden müssen, die zur Laufzeit nicht bekannt sein *können*. Beim Late-Binding können Sie Aufrufe von Methoden durchführen, die der Compiler zur Entwurfszeit nicht auflösen kann, weil er zu dieser Zeit noch nicht wissen kann, was für ein Objekttyp sicher wirklich hinter einer Objektvariablen der Vererbungshierarchie verbirgt. Solche Konstrukte sollten aber die Ausnahme bleiben und vor allen Dingen gesondert ausgezeichnet werden. Dem trägt Microsoft sowohl in Visual Basic als auch in C# mit der nächsten Version des .NET Frameworks (4.0) mit *dynamischen Objekten* Rechnung. Mehr zu Klassenhierarchien erfahren Sie übrigens in Kapitel 15.

BEGLEITDATEIEN

Sie finden die Codedateien zu diesem Beispiel im Verzeichnis

...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 03\\OptionSamples

Öffnen Sie dort die entsprechende Projektmappendatei (.SLN-Datei).

```

Option Explicit Off
Option Strict Off

Public Class frmMain
    Private Sub btnShowWeekday_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnShowWeekday.Click

        Dim locDateVariable
        Dim locWeekday As String

        'Datumseingabe auslesen
        locDateVariable = txtDate.Text

        'In Wochentag umwandeln
        locWeekday = Weekday(locDateVariable)

        '"Nullen" vermeiden
        locWeekday += 1

        'Wochentagsnummer ausgeben
        txtWeekday.Text = locWeekday

    End Sub

End Class

```

Wenn Sie dieses Programm starten, scheint zunächst alles in Ordnung zu sein:

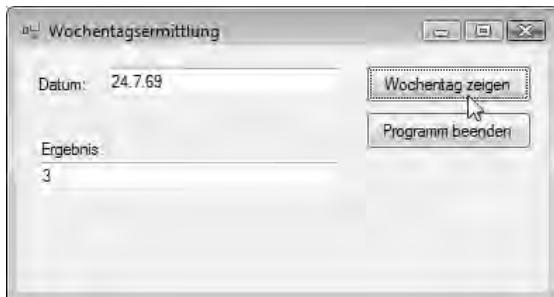


Abbildung 3.8 Dieses Programm »scheint« nur zu funktionieren – in Wahrheit verbergen sich in den paar Zeilen Code zwei dicke Fehler!

Das Programm erlaubt die Eingabe eines Datums und zeigt im Ergebnisfeld die Wochentagsnummer an. Doch stimmt das Ergebnis? Es stimmt nicht. Denn ganz gleich, welches Datum Sie eingeben, es kommt immer das gleiche Ergebnis, nämlich »3« heraus. Und woran liegt das? Nun, zum einen gibt es einen Tippfehler bei einer Variablen. Diese muss nämlich `locDateVariable` und nicht `locDateVaraible` heißen. Durch die fehlende Deklarationserzwingung deklariert der Compiler Variablen, die er noch nicht kennt, einfach selbst, sodass es jetzt zwei Variablen ähnlichen Namens gibt. Hätten Sie `Option Explicit` auf `On` geschaltet, wäre dieser Fehler nicht passiert, bzw. das Programm hätte sich von Anfang an gar nicht starten lassen.

Der zweite Fehler: `locWeekDay` wurde versehentlich als `String` und nicht als `Integer` definiert. Selbst wenn Sie den ersten Fehler behoben hätten, würde das Programm Ihnen jetzt einen Laufzeitfehler zeigen, den Sie auch erst einmal wieder beheben müssten. Sie sehen, dass auch fehlende Typsicherheit viel Arbeit produziert.

ren kann. Wenn Sie sich durch Option Strict On selbst dazu zwingen, dass Sie beispielsweise keiner String-Variablen einen Wert vom Typ Integer zuweisen können, vermeiden Sie solche Fehler im Vorfeld.

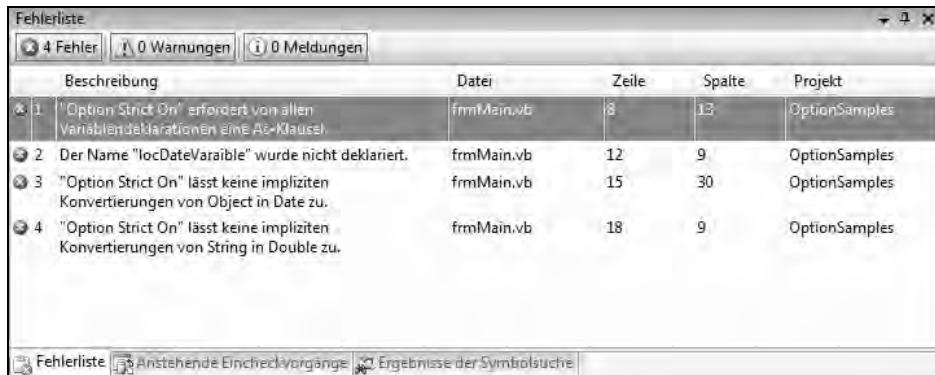


Abbildung 3.9 Bescheid wissen über Fehler heißt, Debugging zu vermeiden. Nach diesen Fehlern hätten Sie ohne Option Explicit und Option Strict wahrscheinlich eine ganze Weile gesucht!

Diesen ganzen Zeitaufwand hätten Sie sich also sparen können, hätten Sie von Anfang an die entsprechenden Optionen eingeschaltet. Die Fehlerliste sähe dann aus wie in Abbildung 3.9.

Natürlich brauchen Sie in Ihren Projekten nicht zu Beginn jeder Codedatei die entsprechenden Anweisungen zu schreiben, sondern können dieses gewünschte Verhalten entweder für das gesamte Projekt oder für alle zukünftigen Projekte voreinstellen:

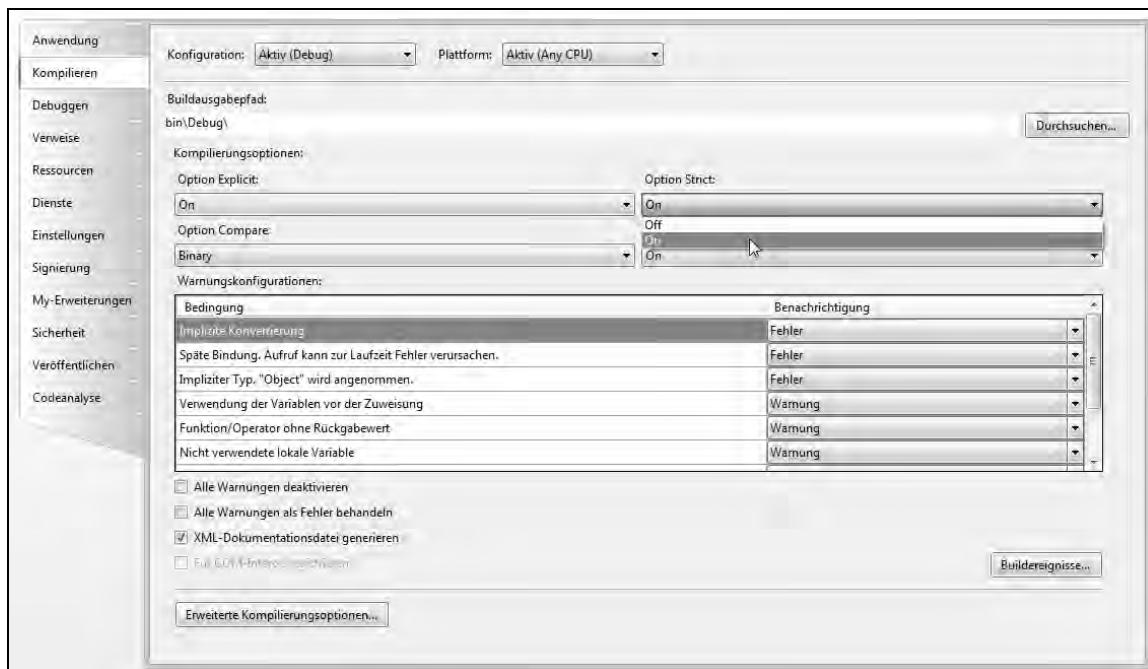


Abbildung 3.10 Auf dieser Registerkarte der Eigenschafteneinstellungen definieren Sie das Option-Verhalten global für das gesamte Projekt

- Um diese Regelung global für das ganze Projekt einzustellen, rufen Sie das Kontextmenü des Projektes im Projektmappen-Explorer auf. Wählen Sie anschließend *Eigenschaften*. Auf der Registerkarte *Kompilieren* stellen Sie das Verhalten für *Option Explicit* und *Option Strict* global ein.
- Um die Regelung für alle zukünftigen Projekte automatisch voreinzustellen, wählen Sie aus dem Menü *Extras* den Menüpunkt *Optionen*. Nehmen Sie die entsprechenden Einstellungen im Bereich *Projekte und Projektmappen/VB-Standard* vor (siehe Abbildung 3.11).

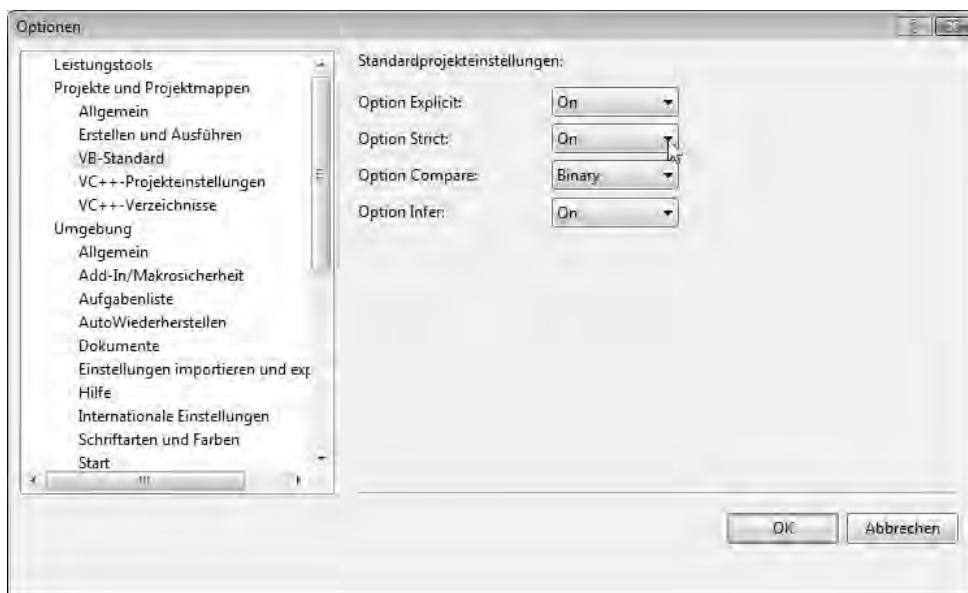


Abbildung 3.11 Mit diesem Dialog bestimmen Sie das Option-Verhalten für zukünftige Projekte

Namensgebung von Variablen

Sie werden bemerkt haben, dass ich Variablen in allen bisherigen Beispielen in der Regel nach einem bestimmten Schema benannt habe. Die Richtlinien von Microsoft besagen, dass man das eigentlich nicht mehr machen sollte. Da ein einfaches Zeigen mit der Maus auf eine Variable genügt, um ihren Typ zu erfahren, sei dieses Vorgehen überflüssig geworden.

```
Dim locDateVariable As Date
Dim locWeekday As String

locDateVariable = DateTime.Now
[Dim locDateVariable As Date]
```

Abbildung 3.12 Ein einfaches »Daraufzeigen« mit der Maus reicht aus, um dem Codeeditor den Typ einer Variablen zu entlocken

Dennoch haben sich in Programmiererkreisen Standards herauskristallisiert, und so ist es bei C#, J# und auch C++ vielerorts üblich, dass klassenglobale Variablen (die so genannten *Member-Variablen*) mit einem kurzen Präfix gekennzeichnet werden, Variablen, die an Prozeduren übergeben werden, mit kleinen Buch-

stablen beginnen und Konstanten wie auch Prozedurennamen mit großen Buchstaben beginnen. Zusammengesetzte Wörter werden durchgekoppelt, die einzelnen Wörter beginnen aber mit einem Großbuchstaben. Das folgende Beispiel zeigt einen typischen C#-Codeausschnitt aus der CLR.⁵

```
// Aus dem CLI-Source-Code
[Serializable()] public sealed class StringBuilder {

    // Klassenvariablen
    //
    internal int m_CurrentThread = InternalGetCurrentThread();
    internal int m_MaxCapacity = 0;
    internal String m_StringValue = null;
    // Statische Konstanten
    //
    internal const int DefaultCapacity = 16;
    .

    .
    .

    // Hängt ein Zeichen an das Ende dieses StringBuilder-Objektes an.
    // Die Kapazität wird im Bedarfsfall angepasst.
    public StringBuilder Append(char value, int repeatCount) {
        if (repeatCount==0) {
            return this;
        }
        if (repeatCount<0) {
            throw new ArgumentOutOfRangeException("repeatCount",
                Environment.GetResourceString("ArgumentOutOfRange_NegativeCount"));
        }

        int tid;
        String currentString = GetThreadSafeString(out tid);

        int currentLength = currentString.Length;
        int requiredLength = currentLength + repeatCount;

        if (requiredLength < 0)
            throw new OutOfMemoryException();

        if (!NeedsAllocation(currentString,requiredLength)) {
            currentString.AppendInPlace(value, repeatCount,currentLength);
            ReplaceString(tid,currentString);
            return this;
        }

        String newString = GetNewString(currentString,requiredLength);
        newString.AppendInPlace(value, repeatCount,currentLength);
        ReplaceString(tid,newString);
        return this;
    }
}
```

⁵ Die komplette CLR-Implementierung der CLI (SSCLI alias »Rotor«) erhalten Sie über den IntelliLink **A0301**.

So stellt Microsoft in seiner Knowledge Base selbst ein Verzeichnis einer Namenskonvention zur Verfügung, das der Microsoft Consultant Service für Programmierarbeiten im alten Visual Basic (Prä-.NET) Umfeld benutzt (hat). Mehr dazu gibt's unter dem IntelliLink **A0302**.

Nun berücksichtigt Visual Basic (leider?) die Unterscheidung der Groß-/Kleinschreibung nicht. Aus diesem Grund habe ich mich dazu entschlossen, Member-Variablen mit dem Präfix »my« beginnen zu lassen. Lokale Variablen (solche, die nur in Prozeduren oder Codeblöcken verwendet werden) beginnen mit »loc«. Ansonsten bezeichne ich die Variablen nicht mit ihrem Typ (also beispielsweise `intIrgendwas` oder `strEineZeichenkette`) – mit einer Ausnahme: Windows-Formularvariablen beginnen in meinem Code grundsätzlich mit dem Präfix »frm«; Windows-Steuerelementvariablen beginnen grundsätzlich mit drei Buchstaben, die sie ebenfalls eindeutig umschreiben (Ausnahme: bei *Frame*-Steuerelementen verwende ich den kompletten Namen als Präfix, um sie vom Formular unterscheiden zu können). Das ist aber nur meine persönliche Konvention. Es gibt aber keine zwingende Vorschrift, Objektvariablen, Eigenschaften, Methoden oder Ereignisse auf eine bestimmte Weise zu benennen – Sie können das halten, wie Sie wollen. Denken Sie aber daran, dass es IntelliSense bei ausgedruckten Listings in Papierform nicht gibt!

Und welche Sprache ist die beste?

Bei der Benennung von Klassen, Methoden, Variablen, Eigenschaften, etc. stellt sich schnell die Frage, welche Sprache (gesprochene, nicht Programmiersprache) man am besten als Grundlage verwendet. Klar ist: Wenn Sie in einem Team mit internationalem Anspruch arbeiten, dann sollten Sie Englisch als Ihre Basissprache verwenden. Für die einfachere Verständlichkeit bei größeren Projekten könnte Deutsch die bessere Grundlage sein, gerade wenn Sie Entwickler in Ihrem Team haben, die des Englischen nicht so mächtig sind.

Allerdings: Wenn es darum geht, wieder verwendbare Komponenten wie beispielsweise Benutzersteuerelemente zu entwickeln, würde ich Englisch selbst dann vorziehen. Es ergibt keinen Sinn, mit einem Mischmasch an Sprachen zu arbeiten, wenn eine Basisklasse aus dem Framework beispielsweise auf der englischen Sprache basiert, Sie sie aber um Methoden und Eigenschaften ergänzen, deren Namen auf dem Deutschen basieren. Und wenn man sich einmal unsicher ist, wie ein deutscher Ausdruck im Englischen zu benennen ist: Leo hilft unter <http://dict.leo.org>.

Hier im Buch werden Sie nur bei einfachen Beispielen, die der Demonstration dienen, deutsche Benennungen finden. Bei Komponenten, die Sie auch für andere Projekte wieder verwenden können, oder bei vollwertigen Anwendungen habe ich mich für die englische Sprache als Basis entschieden.

Kapitel 4

Der Schnelleinstieg in die Bedienung der Visual Studio-Entwicklungsumgebung (IDE)

In diesem Kapitel:

Der erste Start von Visual Studio	62
Die Startseite – der erste Ausgangspunkt für Ihre Entwicklungen	65
Die IDE auf einen Blick	67
Genereller Umgang mit Fenstern in der Entwicklungsumgebung	69
Dokumentfenster	69
Toolfenster	69
Die wichtigsten Toolfenster	72
Wichtige Tastenkombinationen auf einen Blick	87
Verbesserungen an der integrierten Entwicklungs- umgebung (IDE) in Visual Studio 2008	89
Designer für Windows Presentation Foundation-Projekte	93
IntelliSense-Verbesserungen	95

Die IDE – die integrierte Entwicklungsumgebung – ist der Platz in Visual Studio, wo Sie ganz sicher die meiste Zeit verbringen werden – es ist die Benutzeroberfläche, die alle Editoren, Designer, Assistenten und sonstige Werkzeugfenster und Symbolleisten zu deren Steuerung zur Verfügung stellt. Und dabei ist klar: Je besser Sie die wichtigsten Funktionen und Eigenarten dieser mächtigen IDE kennen, desto mehr Zeit werden Sie bei der täglichen Entwicklung Ihrer Softwareprojekte sparen.

Eben weil die Visual Studio-IDE in so vielen verschiedenen kleinen Symbolleisten und Fenstern stattfindet, kann es passieren, dass Sie den Überblick verlieren oder sich die Arbeit unnötig schwer machen, da Sie vielleicht mit einem geeigneten Werkzeug, das Sie einfach nur noch nicht kennen, schneller zum Ziel kämen, als Sie es bislang gemacht haben. Und dann ist es wichtig zu wissen, wie Sie die vielen Fenster und Werkzeugeisten passend auf Ihre Bedürfnisse zuschneiden und gegebenenfalls auch wieder in die Ausgangsstellung zurückgelangen.

Der erste Start von Visual Studio

Falls Sie das zweite Kapitel bereits gelesen haben, liegt der erste Start von Visual Studio bereits hinter Ihnen, und den folgenden Dialog werden Sie dann nicht mehr zu sehen bekommen. Dennoch soll der erste Start von Visual Studio 2008 an dieser Stelle der Vollständigkeit halber noch mal erwähnt werden, auch im Hinblick auf die Unterschiede zur Vorgängerversion Visual Studio 2005. Auf den ersten Blick sind Unterschiede zum Vorgänger von Visual Studio 2008 kaum sichtbar. Den ersten Start vollziehen Sie genau so, wie Sie es beim Vorgänger gemacht haben. Visual Studio zeigt Ihnen einen Dialog, in dem Sie die Voreinstellungen der Benutzeroberfläche bestimmen können, etwa wie in Abbildung 4.1 zu sehen:



Abbildung 4.1 Beim ersten Start des Programms bestimmen Sie, wie die Entwicklungsumgebung voreingestellt werden soll

In diesem Dialog bestimmen Sie, mit welchen Voreinstellungen die Entwicklungsumgebung konfiguriert werden soll. Wenn Sie es gewohnt sind und waren, mit Visual Studio 2002, 2003 und 2005 in der Standardeinstellung zu arbeiten, wählen Sie an dieser Stelle *Allgemeine Entwicklungseinstellungen* aus.

TIPP *Allgemeine Entwicklungseinstellungen* ist die Einstellung, mit der die meisten Visual Studio 2005 und 2008 Installationen voreingestellt werden. Visual Basic-Entwicklungseinstellungen enthalten spezielle Anpassungen, bei denen Fensterlayout, Befehlsmenüs und Tastenkombinationen mehr auf das schnelle Erreichen spezieller Visual Basic-Befehle angepasst sein sollen. Der Dialog zum Anlegen eines neuen Projektes ist beispielsweise nur auf Visual Basic-Projekte angepasst, um nicht zu sagen, beschnitten: Einige Optionen, wie das automatische Anlegen einer Projektmappe sind schon beim Anlegen eines neuen Projektes automatisch ausgeblendet – Sie haben die Möglichkeit, Projekte namenlos zu erstellen und erst am Ende der Entwicklungssitzung unter einem bestimmten Namen zu speichern. Das gilt auch für die Befehle, die Sie über Pulldown-Menüs abrufen können: »Angepasst« bedeutet, dass viele Funktionen, die auch Visual Basic-Projekte betreffen könnten, einfach ausgeblendet sind.

Probieren Sie die für Sie am besten geeignete Variation aus. Falls Sie mit der hier gewählten Voreinstellung später nicht mehr zufrieden sind, beherzigen Sie das im folgenden Absatz Gesagte.

Übernehmen von Visual Studio 2005-Einstellungen

Wenn Visual Studio 2005 und Visual Studio 2008 auf demselben Computer installiert sind, können Sie beim ersten Starten von Visual Studio 2008 die meisten Einstellungen von Visual Studio 2005 übernehmen. Codeausschnitte (auch bekannt unter dem Namen Code Snippets) können hingegen nicht *automatisch* übernommen werden und müssen für die Verwendung in Visual Studio 2008 manuell neu installiert werden. Wenn Visual Studio 2005 und Visual Studio 2008 nicht auf demselben Computer installiert sind, können Sie Ihre Visual Studio 2005-Einstellungen immer noch manuell für die Verwendung in Visual Studio 2008 überführen. In diesem Fall exportieren Sie die Einstellungen in eine Datei, indem Sie aus dem Menü *Extras* den Menüpunkt *Einstellungen importieren oder exportieren* wählen, und die weiteren Anweisungen befolgen. Genaueres zu diesem Thema finden Sie im Tipp & Tricks-Kapitel dieses Buchteils.

Visual Studio 2005-Projekte zu Visual Studio 2008 migrieren

Grundsätzlich hat Visual Studio 2008 andere Projektdateien- und Projektmappendateiaufbauten als Visual Studio 2005. Das heißt erst einmal: Ein Projekt, das Sie mit Visual Basic 2005 (oder einer noch früheren Version) erstellt haben, können Sie nicht direkt in Visual Studio 2008 öffnen. Visual Basic 2005 und Visual Studio 2008 können aber auf ein und demselben Rechner problemlos koexistieren, und wenn Visual Studio 2005 zusammen mit der 2008er Version auf einem Rechner installiert ist, haben Sie folgende Optionen:

- Sobald Sie eine Projekt- oder Projektmappendatei im Explorer doppelklicken, starten Sie damit automatisch die Instanz der »richtigen« (also der dem Projekt zugehörigen) Version von Visual Studio. Ge-regelt wird das dadurch, dass die entsprechende Dateiendung nicht mehr einer bestimmten Visual Studio-Version sondern einem so genannten *Visual Studio Version Selector* zugewiesen ist, der sich die aufzurufende Projektdatei anschaut, analysiert und dann entscheidet, welche Version von Visual Studio gestartet werden soll.



Abbildung 4.2 Nach dem Öffnen eines Visual Basic 2005-Projektes startet sofort der Migrationsassistent, mit dem Sie das Projekt in das neue Projektformat konvertieren können

- Wenn Sie eine Projekt- oder eine Projektappendatei in Visual Studio 2008 öffnen, die in Visual Studio 2005 erstellt wurde, wird der Migrationsassistent aktiv, der es Ihnen ermöglicht, Ihr vorhandenes Projekt in das neue Format zu migrieren (siehe Abbildung 4.2).

HINWEIS Nach der Konvertierung eines Projektes zielt das in das Visual Basic 2008-Format konvertierte Projekt auf das .NET Framework 2.0 – Sie haben hier also weiterhin »nur« die Möglichkeit, alte .NET 2.0-Funktionalität in Ihrem Projekt weiterzuverwenden, aber dafür ist das Kompilat Ihres Projektes auch in der Lage, ältere Windows 2000-Clients¹ zu bedienen, die mit den neueren Framework-Versionen nicht ausgestattet werden können.

Visual Basic 2008 erlaubt es jedoch, auch gegen unterschiedliche Framework-Versionen zu entwickeln – der nächste Absatz hält dazu genauere Informationen bereit. Möchten Sie Ihr konvertiertes Projekt umstellen, sodass Sie auch die Funktionalitäten der neueren 3.0 bzw. 3.5 Framework-Versionen nutzen können, ändern Sie die Projekteinstellungen, wie im Abschnitt »Framework Version-Targeting (Framework-Versionszielwahl) bei Projekten« von Kapitel 11 beschrieben.

Mehr zum Thema Migration, gerade wenn es um die Migration von VB6-Anwendungen geht, finden Sie in Kapitel 9 in Teil B.

¹ Oder, mit eingeschränkter Funktionalität, sogar alte Windows 98-Clients – doch das sei wirklich nur der Vollständigkeit halber erwähnt, denn schon aus Sicherheitsaspekten sollten Sie Ihren Kunden gegenüber fairerweise gar nicht mehr erwähnen, dass Ihre Software theoretisch auch unter Windows 98 lauffähig ist.

Die Startseite – der erste Ausgangspunkt für Ihre Entwicklungen

Jedes Mal, wenn Sie Visual Studio 2008 gestartet haben, begrüßt Sie die IDE mit der Startseite, etwa wie in Abbildung 4.3 zu sehen. Die Startseite zeigt Ihnen nicht nur Ihre zuletzt bearbeiteten Projekte, die Sie direkt per Mausklick auf den jeweiligen Eintrag in der IDE öffnen können, sondern sie liefert – Internetanbindung vorausgesetzt – auch aktuelle Infos rund um Visual Studio, SQL Server und das .NET Framework.



Abbildung 4.3 Nach dem Start von Visual Studio 2008 begrüßt Sie die Startseite, auf der Sie die zuletzt bearbeiteten Projekte öffnen können, und die Ihnen aktuelle Infos rund um Visual Studio liefert

In der linken, oberen Ecke der Startseite finden Sie die letzten Projekte, die Sie bearbeitet haben. Ein einfacher Mausklick genügt, um das entsprechende Projekt zu öffnen. Sie können auch direkt von hier aus neue Projekte anlegen oder Projekte öffnen, die sich nicht in der Liste der zuletzt verwendeten Projekte befinden.

Der Visual Studio-Nachrichten-Channel

Dominierend auf der Startseite ist der Visual Studio-Nachrichten-Channel, der Sie, sofern Sie über eine funktionsfähige Internetverbindung verfügen, immer mit aktuellen Informationen rund um Visual Studio, SQL Server und das .NET Framework versorgt. Damit verwandte Themen werden Sie hier sicherlich ebenfalls finden.

Welche Informationen hier genau angezeigt werden, richtet sich übrigens nach dem eingestellten Nachrichtenchannel, den Sie jederzeit ändern können, wenn Ihnen ein besserer bekannt ist. Dazu rufen Sie mit *Extras | Optionen* den *Optionen*-Dialog von Visual Studio auf und wählen *Start* im Zweig *Umgebung*.²

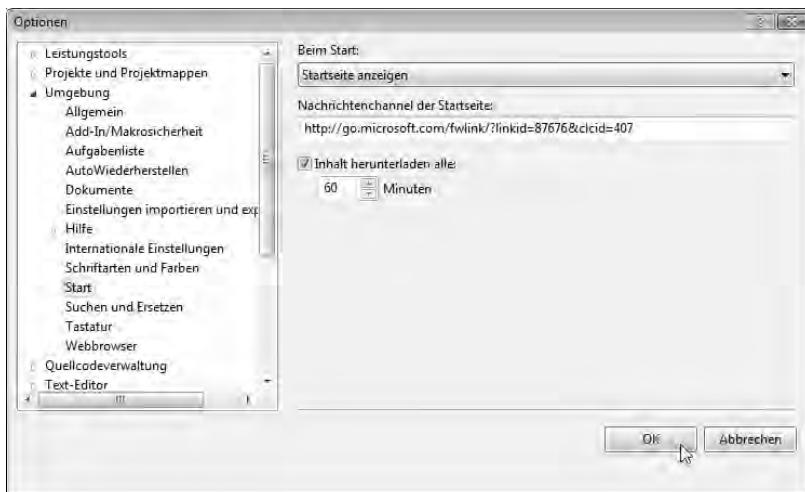


Abbildung 4.4 Mit dieser Registerkarte konfigurieren Sie den verwendeten Nachrichtenchannel der Startseite

Unter *Nachrichtenchannel der Startseite* geben Sie den URL Ihres favorisierten Nachrichtenchannels ein.

Anpassen der Liste der zuletzt bearbeiteten Projekte

In einigen Fällen können sich ältere Projekte, die Sie nicht mehr bearbeiten, als störend in der Liste erweisen. Falls Sie mit dem *Registry Editor* von Windows vertraut sind, dann – und nur dann – können Sie unter *HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\9.0\ProjectMRUList* die Liste der Projektdateien einsehen³ und gegebenenfalls ändern. Das Eintragsformat der Dateien gestaltet sich folgendermaßen:

```
File1:REG_SZ:C:\Pfad zur gewünschten Projektmappe\test1.sln
File3:REG_SZ:C:\Pfad zur gewünschten Projektmappe\test3.sln
File2:REG_SZ:C:\Pfad zur gewünschten Projektmappe\test2.sln
File4:REG_SZ:C:\Pfad zur gewünschten Projektmappe\test4.sln
```

Beachten Sie, dass die Einträge nicht notwendigerweise in numerischer Reihenfolge aufgelistet sind.

WICHTIG Die Einträge müssen lückenlos nummeriert sein, damit alle Projekte angezeigt werden. Es reicht also nicht aus, einen Eintrag aus der Liste zu entfernen, Sie müssen auch dafür sorgen, dass die entstehende Lücke in der Nummerierung ausgeglichen wird. Am einfachsten ist es, die Angaben des letzten Eintrags der Liste in den zu löschen Eintrag zu übertragen und dann den letzten Eintrag der Liste zu löschen.

² Im Bedarfsfall müssen Sie bei einigen Visual Studio-Versionen erst im Dialog das hier im Dialog nicht zu sehende Kontrollkästchen *Alle Einstellungen anzeigen* anklicken, um Zugriff auf alle Optionen nehmen zu können.

³ Für Visual Studio 2005 übrigens 8.0 statt 9.0, also *HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\8.0\ProjectMRUList*.

Die Projekte, die hier verzeichnet sind, beziehen sich sowohl auf die Projekte der Startseite als auch auf die Projekte, die Sie sehen, wenn Sie aus dem Menü *Datei* den Menüpunkt *Zuletzt geöffnete Projekte* auswählen.

Die Liste der zuletzt geöffneten Dateien können Sie übrigens ebenfalls bearbeiten. Verfahren Sie dazu wie oben beschrieben, verwenden Sie jedoch statt des oben zuletzt genannten Schlüssels den Schlüssel *HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\9.0\FileMRUList*.

Ein kleiner Tipp für die ganz Eiligen: Verschieben Sie die nicht mehr benötigten Projekte aus Ihrem Dokumenten-Ordner, Visual Studio 2008, Projects (oder wo immer Sie sie abgelegt haben) in ein anderes Verzeichnis (z.B. »Projektarchiv«), klicken Sie zum Öffnen des Projektes dann auf die Liste der Startseite. Visual Studio fragt dann nach, ob Sie dieses, nicht mehr zu findende Projekt aus der Liste entfernen wollen.

Die IDE auf einen Blick

Auf der nächsten Seite finden Sie eine Grafik, die Ihnen die wichtigsten Elemente der IDE demonstriert. Um es ganz klar zu sagen: Sollte Ihre persönliche IDE-Konfiguration schon jetzt oder nach einer Weile so ausschauen, wie die IDE-Konfiguration in der Grafik auf der nächsten Seite – stoppen Sie sofort, was auch immer Sie machen. Rudern Sie zurück. Setzen Sie die komplette IDE (übrigens mit dem Befehl *Fenster / Fensterlayout zurücksetzen*) in ihren Ausgangszustand zurück, denken Sie darüber nach, welche der IDE-Elemente Sie wirklich am häufigsten benötigen, und bringen Sie diese gemäß den Erklärungen der nächsten Abschnitte in den Vordergrund.

BEGLEITDATEIEN

Um sich mit allen Elementen in Visual Studio .NET vertraut machen zu können, benötigen Sie ein Projekt zum »Spielen«. Da Sie das in den folgenden Beispielen verwendete Projekt noch nie verwendet haben, steht es natürlich noch nicht in der Liste – aber so können es wie in den folgenden Schritten beschrieben aus dem Ordner der Beispieldateien laden. Mehr zu den Beispieldateien finden Sie im entsprechenden Abschnitt in Kapitel 1 beschrieben.

- Wählen Sie aus dem Menü *Datei* den Menüpunkt *Projekt/Projektmappe öffnen*.
- In der Datei-Auswahl, die jetzt erscheint, wählen Sie das Laufwerk und Verzeichnis, in dem Sie die Begleitdateien installiert haben. Wählen Sie dort den Ordner
`...\VB 2008 Entwicklerbuch\A - Einführung\Kapitel 04\Ereignisdemo`
- Doppelklicken Sie auf den Dateinamen *Ereignisse.sln*, um das Programm zu laden.
- Wählen Sie anschließend im Menü *Erstellen* den Menüpunkt *Ereignisse neu erstellen*.

Die folgende Grafik dient lediglich der Orientierungshilfe, als Ausgangspunkt sozusagen für die Abschnitte, die sich im Detail mit den wichtigsten Elementen beschäftigen. Falls Sie bemerken, dass Ihnen der Platz auf dem Bildschirm zu eng wird, ziehen Sie es wirklich in Erwägung, einen Monitor mit einer größeren Auflösung oder besser, da noch mehr Platz und obendrein billiger, einen zweiten Monitor für den Mehrmonitorbetrieb Ihres Computers anzuschaffen (Kapitel 8 liefert mehr Informationen dazu).

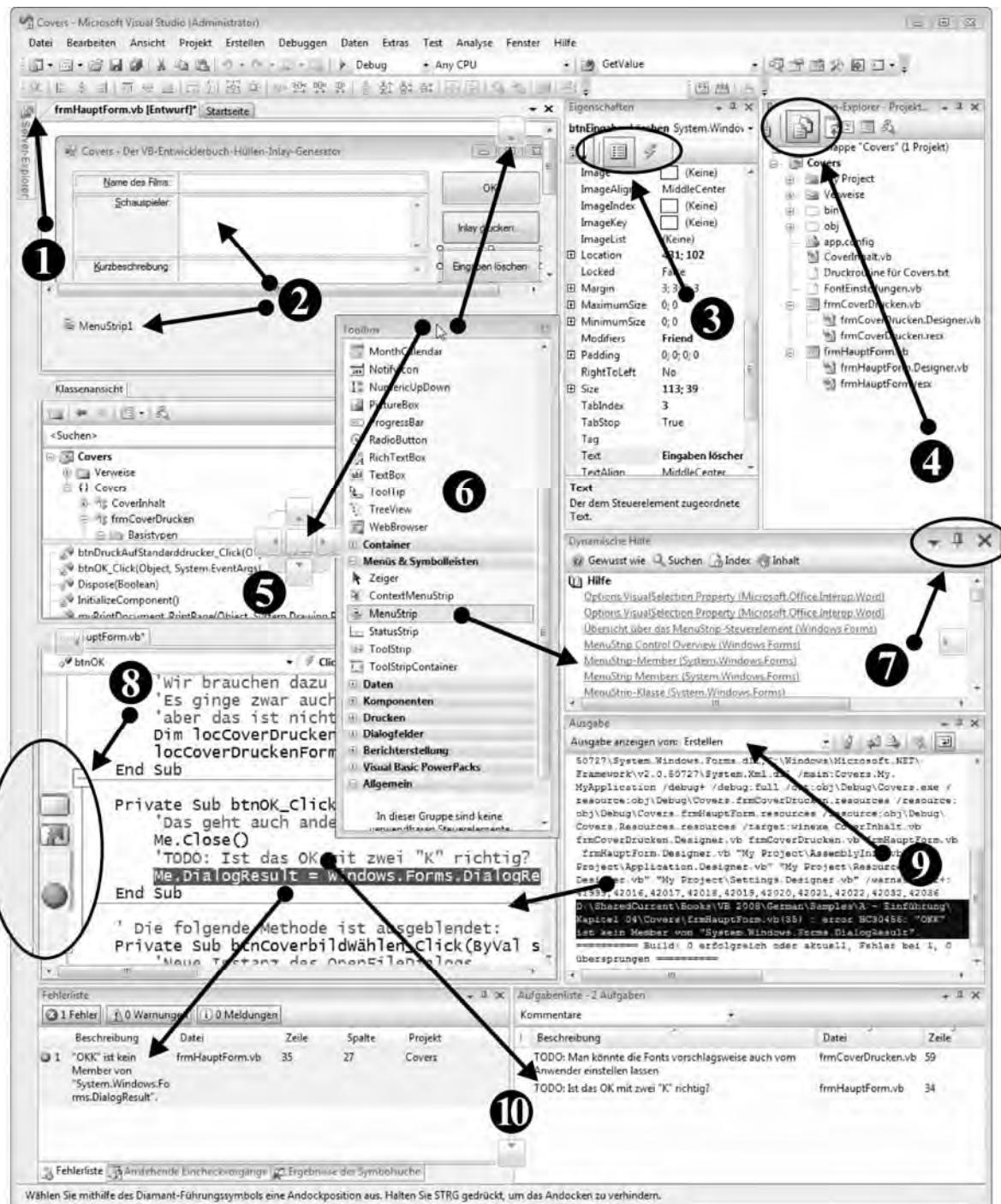


Abbildung 4.5 Die Abbildung zeigt die wichtigsten Features der Visual Studio IDE (Die Erklärung zu dieser Abbildung finden Sie ab Seite 69, Abschnitt »Toolfenster«)

TIPP Die meisten Grafikkarten, die heutzutage in modernen Computern oder Notebooks verbaut werden, erlauben ohnehin den Anschluss eines zweiten Monitors und die entsprechende Erweiterung des Desktops auf diesen. Die zusätzliche Investition für einen zweiten Monitor wird sich schnell rechnen, denn glauben Sie mir: Eine Auflösung von 1024 x 768 Punkten auf nur einem verfügbaren Monitor führt fast zwangsläufig zu einer Überladung Ihres Sichtfeldes; Ihre Konzentration wird entweder durch das ständige Auf- und Zuklappen der benötigten Toolfenster oder durch viele zu kleine Toolfenster stark nachlassen! Außerdem verlieren Sie durch das ständige Navigieren zwischen den Elementen zu viel Zeit.

Genereller Umgang mit Fenstern in der Entwicklungsumgebung

Dass die Benutzeroberfläche der Entwicklungsumgebung ungewöhnlich viele Möglichkeiten bietet, werden Sie seit Ihren ersten Experimenten mit Visual Studio bemerkt haben. Der Nachteil der Entwicklungsumgebung ist: Sie hält so viele Elemente parat, dass Sie leicht die Übersicht verlieren können. Der Vorteil: Die Entwicklungsumgebung können Sie auf Ihre eigenen Bedürfnisse zuschneiden, wie sonst kaum ein anderes Windowsprogramm. Schauen Sie sich die einzelnen Bereiche der Oberfläche ein wenig genauer an. Abbildung 4.5 zeigt Ihnen die wichtigsten Elemente der Visual Studio-IDE auf einen Blick.

Dokumentfenster

Die Fenster der Entwicklungsumgebung werden durch so genannte *Registerkartengruppen* verwaltet. Davon gibt es zwei verschiedene Arten: Zum einen die, die den eigentlichen Inhalt Ihrer Projektdateien (Code, Formulare) sowie die Startseite, Projekteigenschaften und u. U. bestimmte Werkzeuge wie die Klassenansicht eines Projektes abbilden. Diese werden *Dokumentfenster* genannt. Dokumentfenster werden dynamisch erstellt, wenn Sie Dateien oder andere Elemente öffnen oder erstellen. Die Liste der geöffneten Dokumentfenster wird im Menü *Fenster* angezeigt.

Die Möglichkeiten zur Verwaltung von Dokumentfenstern hängen weitestgehend vom Schnittstellenmodus ab, der unter *Allgemein, Umgebung* im Dialog *Optionen* ausgewählt wurde. Sie können wahlweise im Modus *Dokumente im Registerkartenformat* (Standard) oder im Modus *Multiple Document Interface* (MDI – etwa: Mehrfache Dokumentschnittstelle) arbeiten. Experimentieren Sie einfach mit diesen Einstellungen, um eine Ihren Bedürfnissen und Ihrem Geschmack entsprechende Umgebung für die Dokumentbearbeitung zu erstellen.

Toolfenster

Toolfenster werden im Menü *Ansicht* aufgelistet und sind durch die aktuelle Anwendung und deren Add-Ins definiert. Sie können in der IDE unterschiedlich konfiguriert werden:

- Automatisch ein- oder ausgeblendet (in der Abbildung mit ① gekennzeichnet)
- Im Registerformat und verknüpft mit anderen Toolfenstern
- Angedockt an die Ränder der IDE
- Nicht angedockt (»schwebend«) (in der Abbildung mit ② gekennzeichnet)

- Zur Anzeige als Dokumentfenster (in der Abbildung mit ❸ gekennzeichnet)
- Zur Anzeige auf anderen Monitoren

Zusätzlich können Sie gleichzeitig mehrere Instanzen bestimmter Toolfenster anzeigen lassen. So können Sie z. B. mehrere Fenster eines Webbrowsers anzeigen. Wählen Sie zum Erstellen einer neuen Toolfensterinstanz im Menü *Fenster* den Menübefehl *Neues Fenster* aus. Außerdem können Sie festlegen, wie sich die Betätigung der Schaltflächen *Schließen* und *Automatisch im Hintergrund* auf eine Gruppe zusammen angedockter Toolfenster auswirkt.

WICHTIG Sie können nahezu jedes Toolfenster als Dokumentfenster behandeln (dazu öffnen Sie das Kontextmenü des jeweiligen Toolfensters durch Rechtsklick auf den Fenstertitel und wählen anschließend *Dokument* im Registerkartenformat); Sie können allerdings kein echtes Dokumentfenster wie den Quellcode einer Codedatei oder die Designer-Darstellung eines Formulars als Toolfenster darstellen.

Andocken von Toolfenstern

Im Modus *Dokumente im Registerkartenformat* können Sie festlegen bzw. verhindern, dass die Dokumentfenster angedockt werden können, indem Sie im Kontextmenü des Fenstertitels die Option *Andockbar* aktivieren bzw. deaktivieren. Im MDI⁴-Modus sind Dokumentfenster nicht andockbar.

TIPP Bei einigen Dokumentfenstern innerhalb der IDE handelt es sich in Wirklichkeit um Toolfenster, für die die Andockfunktion deaktiviert wurde. Sie können diese Fenster dennoch andocken, indem Sie im Menü *Fenster* die Option *Andockbar* auswählen.

Um Fenster entweder an eine der Seiten der IDE oder in anderen Registerkartengruppen anzudocken, verfahren Sie folgendermaßen. Orientieren Sie sich dabei bitte an der Toolbox (bezeichnet mit ❶) in Abbildung 4.5.

- Klicken Sie mit der Maus auf den Fenstertitel des Toolfensters, das Sie neu positionieren möchten, und halten Sie die Maustaste dabei fest.
- Sobald Sie beginnen, das Fenster zu verschieben, blendet die IDE Positionshilfen ein, wie Sie sie auch in der Abbildung erkennen können.
- Um ein Toolfenster an einer Seite der IDE anzudocken, ziehen Sie es auf eine der vier Positionshilfen, die an den vier Seiten der IDE zu finden sind. In der Abbildung erkennen Sie eine dieser Positionshilfen, die durch den von der Titelzeile der Toolbox nach oben laufenden Pfeil markiert ist.
- Um ein Toolfenster neben oder in einer Registerkartengruppe anzuordnen, bewegen Sie die Registerkarte in Richtung einer Registerkartengruppe (oder eines anderen Toolfensters, aus der dann nach dem Loslassen automatisch eine Registerkartengruppe wird). Die IDE bildet anschließend eine weitere Positionshilfe ein – immer in der Nähe der jeweiligen Registerkartengruppe – die aus fünf Symbolen besteht. Die äußeren Symbole erlauben das Anordnen des Toolfensters an der jeweiligen Seite der Registerkartengruppe (oder des anderen Toolfensters); das innere Symbol dient dem Zweck, das Toolfenster der Registerkartengruppe hinzuzufügen (oder ein einzelnes Toolfenster, das Sie ansteuern, zu einer Registerkartengruppe werden zu lassen).

⁴ MDI: *Multi Document Interface*. Visual Studio stellt dabei ein umgebendes Hauptfenster dar, und *alle* anderen Dokumentfenster lassen sich hierin gleichberechtigt (ohne Registerkartenaufteilung) anordnen und organisieren.

Das Verschieben und Andocken von Toolfenstern läuft im Prinzip immer noch genau so einfach ab, wie schon in Visual Studio 2005, erfuhr in Visual Studio 2008 aber ein weiteres Mal eine Überarbeitung in der grafischen Unterstützung des »Landesplatzes« des Fensters: Beim Ziehen mit der Maus wird ein so genanntes Diamant-Führungssymbol angezeigt. Indem Sie bei gedrückter Maustaste den Mauszeiger auf eines der Pfeilsymbole führen und dann loslassen, können Sie bequem die Zielposition bestimmen (siehe Abbildung 4.6).

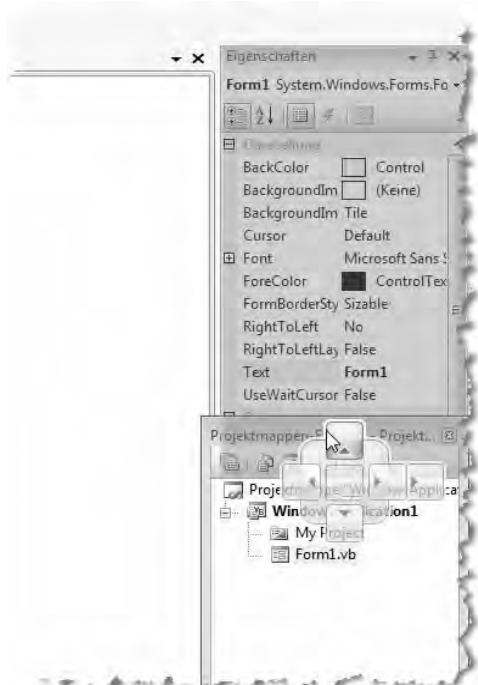


Abbildung 4.6 Mithilfe des Diamant-Führungssymbols lassen sich Fenster schnell und bequem positionieren

Arbeiten mit Toolfenstern

- Um ein Toolfenster einer Registerkartengruppe zu aktivieren, klicken Sie einfach auf die entsprechende »Lasche« in der Registerkartengruppe.
- Möchten Sie, dass ein Toolfenster nur im Bedarfsfall aufgeklappt wird, wenn Sie es benötigen, und sich anschließend wieder schließt, wenn Sie seinen Umgebungsreich verlassen, klicken Sie auf das Heft-zwecken-Symbol, wie Sie es in Abbildung 4.5. mit 7 bezeichnet erkennen können.
- Wenn Sie ein Toolfenster versehentlich geschlossen haben, finden Sie die wichtigsten Fenster zur erneuten Aktivierung durch Menübefehle im Menü *Ansicht*. Beachten Sie dabei auch den Menüpunkt *Weitere Fenster*, der Zugriff auf die weniger wichtigen Fenster bereithält.
- Das Fensterlayout während des Debuggens eines Programmes ist unabhängig vom Fensterlayout der IDE im Entwurfsmodus. Sobald Sie Ihre Anwendung in der IDE starten, schaltet Visual Studio auf das Fensterlayout des Debug-Modus. Änderungen, die Sie an den Toolfensterkonfigurationen vornehmen, wirken sich nicht auf die Fensterkonfiguration aus, die gilt, wenn Sie sich im Entwurfsmodus befinden, und umgekehrt. Spezielle Toolfenster zum Debuggen finden Sie übrigens im Menü *Debuggen | Fenster*.

Navigieren durch Dateien und Toolfenster mit dem IDE-Navigator

Um im Codeeditor zu einer bestimmten geöffneten Datei zu navigieren, egal wann diese zuletzt verwendet wurde, können Sie den IDE-Navigator verwenden. Die Funktionsweise des IDE-Navigators ist in etwa dieselbe, wie die des Task-Switchers in Windows selbst, den Sie mit der Tastenkombination **Alt** **→** bedienen. Der IDE-Navigator ist nicht über Menüs verfügbar. Sie können ihn über die Tastenkombinationen **Strg** **→** bzw. **Strg** **↑** **→** bedienen – je nachdem, in welcher Reihenfolge Sie durch die Dateien navigieren möchten (siehe Abbildung 4.7).



Abbildung 4.7 Den IDE-Navigator rufen Sie mit **Strg** **→** oder **Strg** **↑** **→** ins Leben

Drücken Sie bei gehaltener **Strg**-Taste – für die umgekehrte Richtung halten Sie zusätzlich die **↑**-Taste gedrückt – die Tabulatortaste **→** so oft, bis die gewünschte Datei ausgewählt ist oder klicken Sie diese direkt mit der Maus an.

TIPP Alternativ klicken Sie in der oberen rechten Ecke des Editors auf die Schaltfläche *Aktive Dateien* neben der Schaltfläche *Schließen*, und wählen Sie die gewünschte Datei aus der Liste aus.

Mit dem IDE-Navigator können Sie auch zwischen den Toolfenstern navigieren, die in der IDE geöffnet sind. Je nachdem, in welcher Reihenfolge Sie navigieren möchten, können Sie die Tastenkombinationen **Alt** **F7** oder **Alt** **↑** **F7** verwenden.

Die wichtigsten Toolfenster

Beim Stöbern durch die IDE werden Sie festgestellt haben, wie viele Werkzeuge Ihnen Visual Studio für den Produktiveinsatz anbietet – zu viele, um sie alle an dieser Stelle ausführlich darzustellen. Doch die wichtigsten Toolfenster, die, die Sie mit Abstand am häufigsten benötigen, finden Sie in den folgenden Abschnitten beschrieben.

Projektmappen und Projekte mit dem Projektmappen-Explorer verwalten

Mithilfe des Projektmappen-Explorers navigieren Sie in Ihrem Projekt (in Abbildung 4.5 unter ④ zu sehen). Er zeigt eine Liste aller Dateien, aus denen Ihr Projekt bzw. Ihre Projektmappe besteht sowie eine Liste mit Verweisen auf alle Assemblies oder andere Projekte der Projektmappe, die ein Projekt verwendet. Die wichtigsten Funktionen zum Managen Ihres Projektes erreichen Sie, indem Sie das Kontextmenü aufrufen, wenn Sie sich mit dem Mauszeiger über dem Projektmappen-Explorer befinden.

Eine Projektmappe kann verschiedene Projekte enthalten. Dabei ist es egal, ob diese Projekte sich gegenseitig benötigen und aufrufen oder ob sie voneinander völlig unabhängig sind. Sie definieren eines der Projekte als Startprojekt, indem Sie den Projektnamen mit der rechten Maustaste anklicken und den Menüpunkt *als Startprojekt festlegen* auswählen.

HINWEIS Bitte verwechseln Sie diese Funktion nicht mit der Funktion *Startobjekt*, die Sie über die Eigenschaftenseite eines Projektes erreichen (Kontextmenü eines Projektes im Projektmappenexplorer), und mit deren Hilfe Sie bestimmen, welches Objekt innerhalb Ihres *Startprojektes* das *Startobjekt* sein soll.

Über das Kontextmenü einer Projektmappe oder eines Projektes erreichen Sie Funktionen, um ...

- ... die Projektmappe/das Projekt zu erstellen – dabei werden nur veränderte Dateien der Projektmappe/des Projektes neu kompiliert.
- ... einen Build – also das resultierende Kompilat eines Compilerdurchlaufs in Form von Assemblies, ausführbaren oder anderen Dateien – komplett zu bereinigen. Wenn Sie einen Build bereinigen, werden alle Zwischen- und Ausgabedateien gelöscht, sodass nur die Projekt- und Komponentendateien übrig bleiben. Anschließend können aus den Projekt- und Komponentendateien neue Instanzen der Zwischen- und Ausgabedateien erstellt werden.
- ... die Projektmappe/das Projekt *neu* zu erstellen – dabei werden alle Dateien der Projektmappe neu kompiliert.
- ... den Konfigurationsmanager für eine Projektmappe aufzurufen, um Abhängigkeiten, Prioritäten, Plattformmeigenarten und Ähnliches festzulegen.
- ... ein neues oder vorhandenes Projekt oder Element der Projektmappe/dem Projekt hinzuzufügen.
- ... das Startprojekt der Projektmappe festzulegen.
- ... das Projekt im Debug-Modus oder im Einzelschrittmodus ablaufen zu lassen (dazu muss das Kontextmenü der Projektmappe ausgewählt worden sein).
- ... eine neue Instanz eines Projektes im Debug-Modus zu starten (dazu muss das Kontextmenü des Projektes ausgewählt worden sein).
- ... das (gesamte) Projekt zu speichern.
- ... die Projektmappe zur Quellcodeverwaltung (*Visual Source Safe*) hinzuzufügen, bzw. entsprechende Dateien des Projektes, die unter Quellcodeverwaltung stehen, ein- und auszuchecken.
- ... das Projekt/die Projektmappe umzubenennen.
- ... die Eigenschaften für die Projektmappe abzurufen.

Projektdateitypen

Die wichtigsten Projektdateitypen in Visual Basic sind (ASP.NET-Projekte einmal außen vor gelassen):

Symbol	Typ	Aufgabe
	Projektmappe (Solution)	Beinhaltet die Zusammenstellung der Projektappendateien sowie die globalen Einstellungen der Projektmappe.
	Visual Basic-Projekt	Beinhaltet die Zusammenstellung eines Visual Basic-Projektes sowie die globalen Einstellungen für das Projekt.
	Formulardatei	Stellt die Quellcodedatei einer Klasse dar, die von <i>System.Windows.Forms</i> abgeleitet wurde, damit ein Formular verwaltet und mit dem Designer von Visual Studio .NET bearbeitet werden kann.
	Visual Basic-Klassendatei	Stellt die Quellcodedatei einer Visual Basic-Klasse, eines Moduls oder einer <i>Assembly-Info</i> dar.

Tabelle 4.1 Die wichtigsten Projektdateien

TIPP Wenn Sie per Doppelklick auf eine Formulardatei nicht den Designer, sondern direkt das entsprechende Codefenster anzeigen lassen möchten, öffnen Sie das Kontextmenü, indem Sie mit der rechten Maustaste auf die betroffene Datei klicken, und wählen anschließend den Eintrag *Öffnen mit*. Im Dialog, der jetzt gezeigt wird, wählen Sie *Microsoft Visual Basic-Editor* per Mausklick aus und klicken anschließend auf die Schaltfläche *Als Standard*.

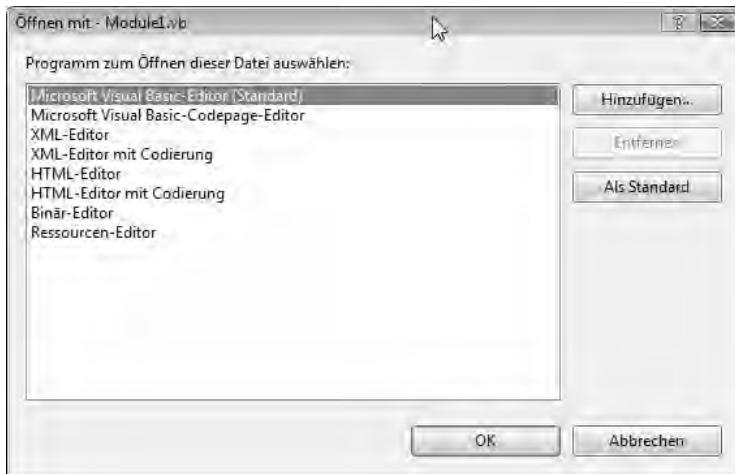


Abbildung 4.8 Mit diesem Dialog, den Sie durch Öffnen des Kontextmenüs einer Datei Ihres Projektes erhalten, bestimmen Sie, welcher Editor standardmäßig verwendet werden soll

Alle Projektdateien anzeigen

Einige Projektelemente eines Visual Basic-Projektes verfügen über mehrere Dateien, von denen standardmäßig nur die wichtigsten im Projektmappen-Explorer dargestellt werden. Dieses Ausblenden von für den Entwickler anfangs oft weniger wichtigen Projektdateien sieht man besonders schön bei Formular-Klassen.

Im »Rohzustand«, wenn also nicht alle Dateien eines Projektes angezeigt werden, sieht man nach dem Erstellen eines neuen Formulars lediglich eine einzelne Klassendatei, die obendrein noch völlig leer ist – gerade einmal die Klassendefinition ist dort zu finden.

»Unter der Haube« spielt sich jedoch schon eine ganze Menge ab, und »unter der Haube« meint in diesem Fall: Es gibt weitere Dateien, die zum Formular (und damit auch zum Projekt) gehören.

Mit dem in Abbildung 4.5 unter ④ mit dem Pfeil gekennzeichneten Symbol können Sie alle Dateien eines Projektes ein- und ausblenden.

Weitere Funktionen des Projektmappen-Explorers

Die folgende Tabelle erklärt kurz die weiteren Funktionen, die sich durch die Symbole des Projektmappen-Explorers aufrufen lassen:

Symbol	Aufgabe
	<p>Stellt die Eigenschaftenseite einer Projektmappe oder eines Projektes dar.</p> <p>HINWEIS: Anders als noch in Visual Basic.NET 2003 werden die Eigenschaften eines Projektes anschließend als Dokumentfenster in der aktuellen Registerkartengruppe für Dokumentfenster dargestellt und auch als solche behandelt.</p> <p>TIPP: Wenn Sie den Projektmappen-Explorer frei schwebend konfiguriert haben – beispielsweise um ihn auf einem zweiten Monitor darzustellen und so mehr Platz für Designer und Codeeditor zu haben – müssen Sie auf Grund einer »Unzulänglichkeit« die Eigenschaftenseite einer Projektmappe zweimal aufrufen, da sich beim ersten Aufruf die Selektion innerhalb des Projektmappen-Explorers verstellt. Dem Entwicklungsteam ist dieser Bug bekannt – offensichtlich aus Kompatibilitätsgründen zur Visual Studio Extensibility kann er aber nicht ohne weiteres behoben werden. Der IntelliLink A0401 verrät mehr.</p>
	<p>Schaltet um zwischen der eingeschränkten und der vollständigen Projektdateienanzeige. Bei der vollständigen Projektdateienanzeige sehen Sie ausnahmslos alle dem Projekt zugeordneten Dateien – beispielsweise wird der vom Designer automatisch erzeugte Code zur Erstellung eines Formulars dann unterhalb der eigentlichen Formularklasse eingeblendet. Die vollständige Anzeige erlaubt ebenfalls einen Blick in die von einem Projekt referenzierten Assemblies (die Sie benötigen, wenn Sie bestimmte Funktionalitäten des Frameworks oder eines anderen Projektes benötigen), die Sie anderenfalls nicht sehen würden.</p>
	Aktualisiert die Projektdateienliste im Projektmappenexplorer.
	Ruft den Codeeditor auf und stellt die zuvor im Projektmappen-Explorer ausgewählte Codedatei dort dar.
	Ruft den Formular-Designer auf und stellt die zuvor im Projektmappen-Explorer ausgewählte Formulardatei dort dar.
	Falls Sie eine Klassendatei im Projektmappen-Explorer ausgewählt haben, gelangen Sie mit dieser Funktion zum visuellen Klassendesigner.

Tabelle 4.2 Die wichtigsten Projektdateien

Organisieren von Codedateien in Unterordnern

Visual Studio interessiert es nicht, ob eine Projektdatei, die Sie für ein Projekt benötigen, innerhalb des gleichen Verzeichnisses oder in einem Unterverzeichnis des Projektes liegt.



Abbildung 4.9 In umfangreichen Projekten helfen Unterverzeichnisse, dass die Übersicht über die vielen Codedateien eines Projektes nicht verloren geht

TIPP Deswegen der Tipp: Machen Sie von Ordnern zur Organisation von Einheiten Ihres Projektes ruhig intensiv Gebrauch – Sie werden sich besser in größeren Projekten zurechtfinden. Legen Sie mithilfe des Projektmappen-Explorers Unterverzeichnisse an, in denen Sie die Codedateien zusammenfassen, die thematisch zusammengehören. Nutzen Sie dabei vor allem die Möglichkeiten von partiellen Klassen, mit deren Hilfe Sie den Code einer Klasse über mehrere Quellcodedateien verteilen können. Kapitel 14 erklärt mehr zum Thema partielle Klassen und dem Modifizierer Partial.

Abbildung 4.9 zeigt Ihnen, wie u.a. Klassen in Unterverzeichnissen in größeren Projekten zum leichteren Navigieren im Code organisiert sein können:

Dateioperationen innerhalb des Projektmappen-Explorers

Falls Sie übrigens einmal in die Lage kommen sollten, Code oder sonstige Dateien kopieren, verschieben oder löschen zu müssen – dazu müssen Sie Visual Studio nicht verlassen. Zwar bietet Ihnen der Projektmappen-Explorer keine so komfortable Funktionalität wie der Windows-Explorer von XP oder gar der Windows Vista Explorer; aber Sie müssen Visual Studio immerhin nicht verlassen ...

Über das Kontextmenü der Codedatei eines Projektes können Sie die Funktionen *Verschieben* oder *Kopieren* abrufen. Nachdem Sie die Datei auf diese Weise markiert haben, klicken Sie entweder das Projekt oder einen Ordner innerhalb des gleichen oder eines anderen Projektes an – wiederum mit der rechten Maustaste – und wählen aus dem Kontextmenü, das anschließend erscheint, *Einfügen*. Tastenkürzel funktionieren übrigens im Projektmappen-Explorer ähnlich gut wie im Windows-Explorer.

WICHTIG Wenn Sie Formulare kopieren oder verschieben, sollten Sie ohnehin nicht den Windows-Explorer sondern den Projektmappen-Explorer verwenden. Nur mit ihm stellen Sie implizit sicher, dass Sie alle zum Formular dazugehörigen Dateien »verwischen«. Das gilt unter Umständen ebenfalls für andere Projektdateien wie beispielsweise typisierte DataSets.

Das Eigenschaftenfenster

Mithilfe des Eigenschaftenfensters – in Abbildung 4.5 unter ❸ zu sehen – definieren Sie Eigenschaften von Elementen. Elemente in diesem Zusammenhang sind nicht nur Steuerelemente innerhalb des Designers – der Gültigkeitsbereich des Eigenschaftenfensters erstreckt sich auf weite Bereiche der Visual Studio IDE. Natürlich werden Sie dieses Fenster in der Regel dann (und auch deswegen recht oft) verwenden, um Eigenschaften von Steuerelementen zu bestimmen, die Sie im Formular-Designer bearbeiten. Sie benötigen das Eigenschaftenfenster allerdings auch, um Eigenschaften anderer Projektmappen-Elemente zu bestimmen: So legen Sie beispielsweise auch Parameter wie den Namen eines zu installierenden Programms, den Produkthersteller oder die Setup-Version Ihrer Anwendung in Setup- und Bereitstellungsprojekten mit dem Eigenschaftenfenster fest.

Einrichten und Verwalten von Ereigniscode mit dem Eigenschaftenfenster

Das Eigenschaftenfenster verwenden Sie ebenfalls, um die einem mit dem Designer editierbaren Objekt zugeordneten Ereignisse visuell zu bearbeiten. Sie schalten mit den Symbolen, die durch den von Punkt ❸ abgehenden Pfeil in Abbildung 4.5 markiert sind, zwischen den Ereignissen und den Eigenschaften eines Objektes um.

Ereignisbehandlungsroutinen für Steuerelemente werden im Code bereitgestellt. Klickt der Anwender beispielsweise auf eine Schaltfläche, wird – so vorhanden – der Code der Ereignisbehandlungsroutine für diese Schaltfläche ausgeführt.

Sie haben die Möglichkeit, diese Ereignisbehandlungsroutinen ausschließlich durch den Codeeditor festzulegen – sollten aber aus Gründen der Bequemlichkeit und der Schnelligkeit schon auf das Eigenschaftenfenster zurückgreifen, denn Sie können es für mehrere Aufgaben in Sachen Ereignisse verwenden:

- Als Navigationshilfe: Sämtliche Ereignisbehandlungsroutinen des Objektes, die bereits definiert wurden, finden Sie in der Liste wieder. Ein Doppelklick auf das entsprechende Ereignis reicht aus, um zur entsprechenden Ereignisbehandlungsroutine im Codeeditor zu gelangen.

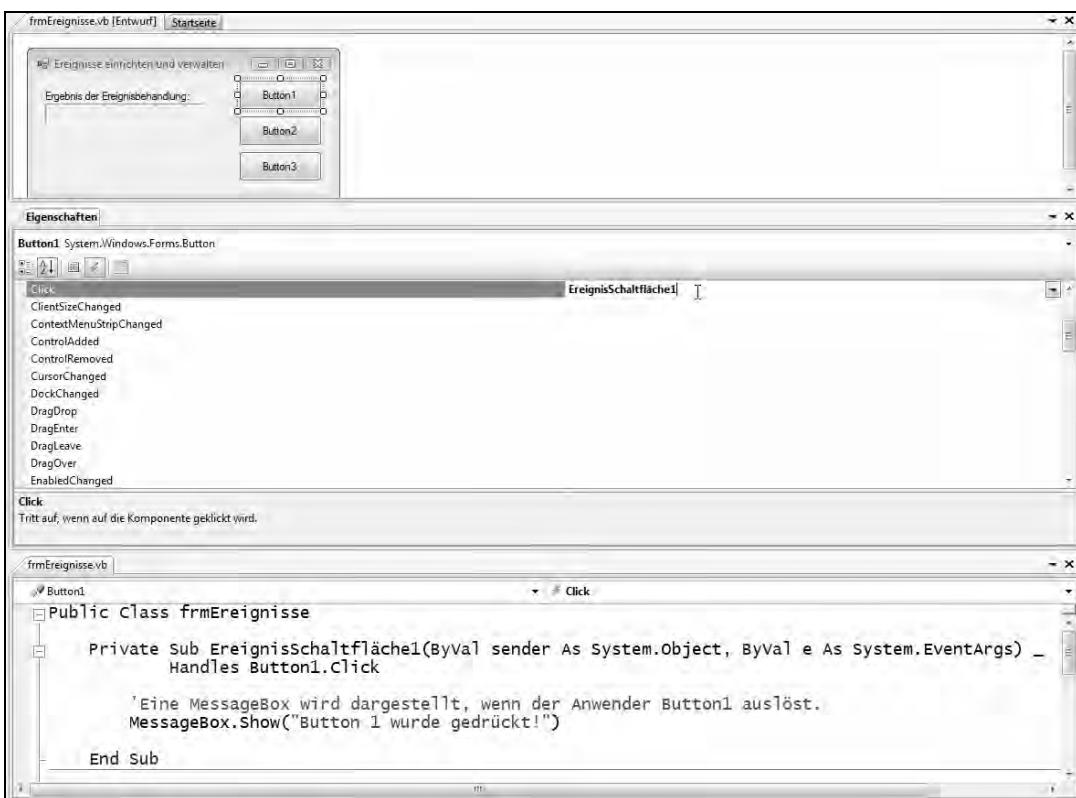


Abbildung 4.10 Der Anwender markiert die Schaltfläche mit dem Namen Button1 (oberes Drittel) und gibt den Namen der Ereignisbehandlungsroutine (mittleres Drittel) ein. Die Behandlungsroutine trägt dann diesen Namen (Ereignisschaltfläche1) und behandelt (Handles) das entsprechende Ereignis (Click) des Objektes (Button1).

- Zum Erstellen einer Ereignisbehandlungsroutine mit einem Standardnamen: Dazu doppelklicken Sie einfach auf ein Ereignis; die IDE fügt anschließend automatisch einen entsprechenden Funktionsrumpf (eine so genannte »Stub«) in die Codedatei ein, und benennt diese als eine Kombination aus Ereignisnamen und Steuerelementnamen.
- Zum Erstellen einer benannten Ereignisbehandlungsroutine: Dazu geben Sie den gewünschten Funktionsnamen der Ereignisbehandlungsroutine neben dem Ereignisnamen ein und drücken anschließend . Die IDE fügt nun automatisch einen entsprechenden Stub in die Codedatei mit dem von Ihnen vorgegebenen Namen ein (siehe Abbildung 4.10).

TIPP Falls Sie sich wundern, wieso das Eigenschaftenfenster in Abbildung 4.10 als Dokument erscheint: Im Abschnitt »Toolfenster« auf Seite 69 erfahren Sie, wie Sie Toolfenster als Dokument einer Dokumentenregisterkartengruppe hinzufügen können.

- Zum Zuweisen der gleichen Ereignisbehandlungsroutine an mehrere Ereignisse. Anders als noch in Visual Basic 6.0 kann eine einzelne Ereignisbehandlungsroutine durchaus mehrere Ereignisse behandeln, wenn diese signaturkompatibel sind (also die gleichen Parameter beim Aufrufen zur Übergabe anbieten). In diesem Fall weisen Sie mit dem Eigenschaftenfenster einem Ereignis eine Ereignisbehandlungsroutine zu: Klappen Sie die Aufklappliste neben dem entsprechenden Ereignis auf, und Sie sehen in der Liste die Namen der signaturkompatiblen Ereignisbehandlungsroutinen. Wählen Sie eine, die dieses Ereignis (ebenfalls) behandeln soll, aus der Liste aus (siehe Abbildung 4.14).

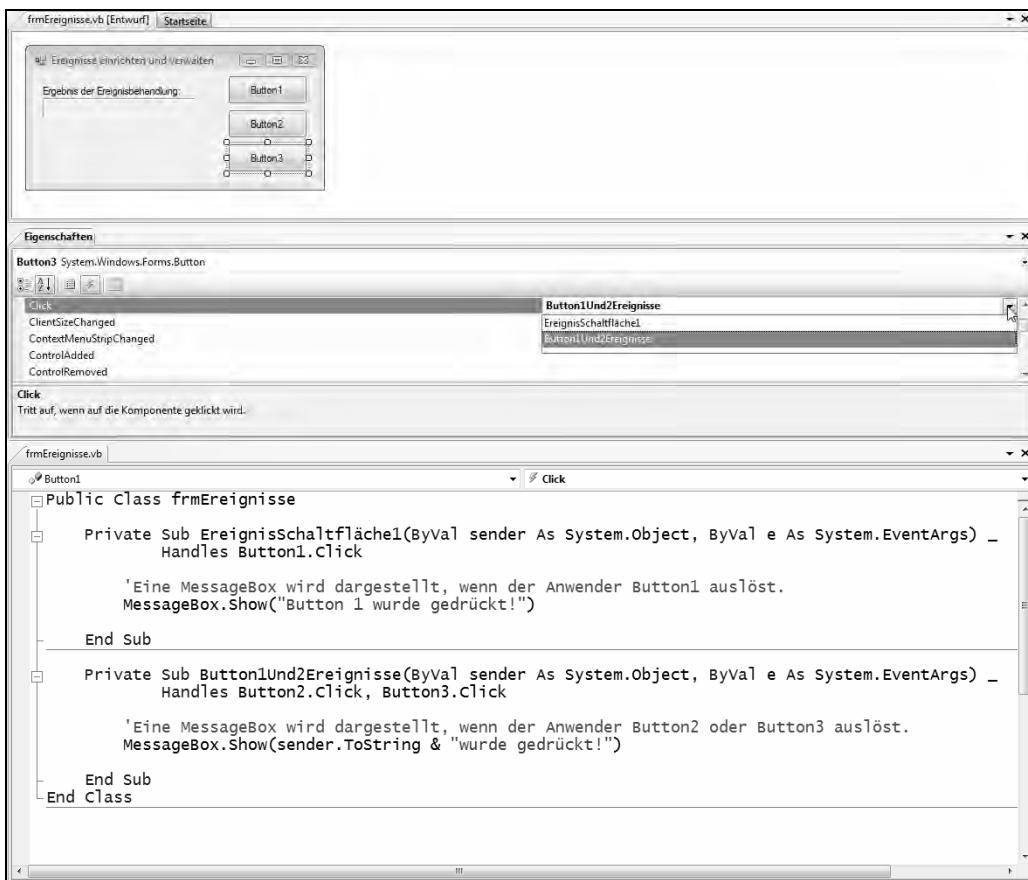


Abbildung 4.11 Der Anwender markiert die Schaltfläche mit dem Namen Button3 (oberes Drittel) und wählt den Namen der Ereignisbehandlungsroutine (mittleres Drittel) aus. Die vorhandene Behandlungsroutine behandelt dann dieses Ereignis ebenfalls (unteres Drittel).

Die Fehlerliste

Visual Basic verfügt über einen so genannten Background-Compiler. Dieser Compiler läuft im Hintergrund, und er überprüft ständig Ihre vorgenommenen Änderungen und Ergänzungen des Programmcodes auf syntaktische Richtigkeit. Der Vorteil: Sie müssen nicht einen bei großen Projekten u.U. schon recht lange dauernden Compilerlauf anstoßen, um die Flüchtigkeitsfehler im Code zu finden – Sie sehen sie sofort, direkt nach der Eingabe einer neuen Zeile. Die Fehlerliste hilft Ihnen dabei, die Übersicht über Fehler im

Programm nicht zu verlieren. Und um den Krieg der Sprachen weiter anzufachen, können wir Visual Basic Entwickler durchaus selbstbewusst behaupten, dass dieser Background-Compiler seinen Dienst um Klassen (verstehen Sie?) besser verrichtet als sein Pendant in C#.

Sie sehen die Fehlerliste in Abbildung 4.5 mit ⑩ markiert (das linke Fenster). Die Fehler, die in der Fehlerliste erscheinen, haben übrigens eine direkte Verbindung zum Code (und der Codedatei), der den Fehler verursacht: Ein Doppelklick auf den Fehler bringt Sie an die Stelle im Codeeditor, an der die IDE den Fehler vermutet.

Die drei verschiedenen Meldungstypen der Fehlerliste

Es gibt übrigens drei verschiedene Meldungstypen in der Fehlerliste, deren Anzeige Sie nach Belieben konfigurieren können:

- **Fehler:** Richtige Fehler verhindern, dass eine Anwendung vor Behebung des Fehlers mit der zugrunde liegenden Codequelle überhaupt gestartet werden kann. Sie müssen den Fehler korrigieren, um das Programm (oder die Assembly) lauffähig zu machen.
- **Warnungen:** Eine Warnung ist ein Hinweis für Sie, dass etwas eigentlich nicht so läuft, wie es sollte; dieses Etwas ist allerdings nicht so schwerwiegend, dass es ein Funktionieren der Anwendungen völlig blockieren würde. Sie können den Schweregrad von Warnungen (welche Ereignisse führen zu Warnungen, welche Warnungen sollen wie Fehler behandelt werden) im Übrigen für jedes Projekt festlegen. Der folgende Abschnitt zeigt, wie es geht.
- **Meldungen:** Geben Ihnen zusätzliche Hinweise und Informationen, die aber in der Regel die Funktionsfähigkeit einer Anwendung nicht beeinflussen.

Welche der Kategorien angezeigt werden, bestimmen Sie übrigens mit den gleich lautenden Schaltflächen am oberen Rand des Fensters. Klicken Sie auf eine der Schaltflächen, um eine Kategorie auszublenden; klicken Sie abermals auf die Schaltfläche, um eine Kategorie wieder darstellen zu lassen.

Konfigurieren von Warnungen in den Projekteigenschaften

Welche Zustände Ihres Codes Warnungen oder Fehler generieren, lässt sich für jedes Projekt individuell einstellen. Anders als noch in Visual Studio 2003 werden die Eigenschaften eines Projektes als Dokumentfenster in der entsprechenden Registerkartengruppe dargestellt.

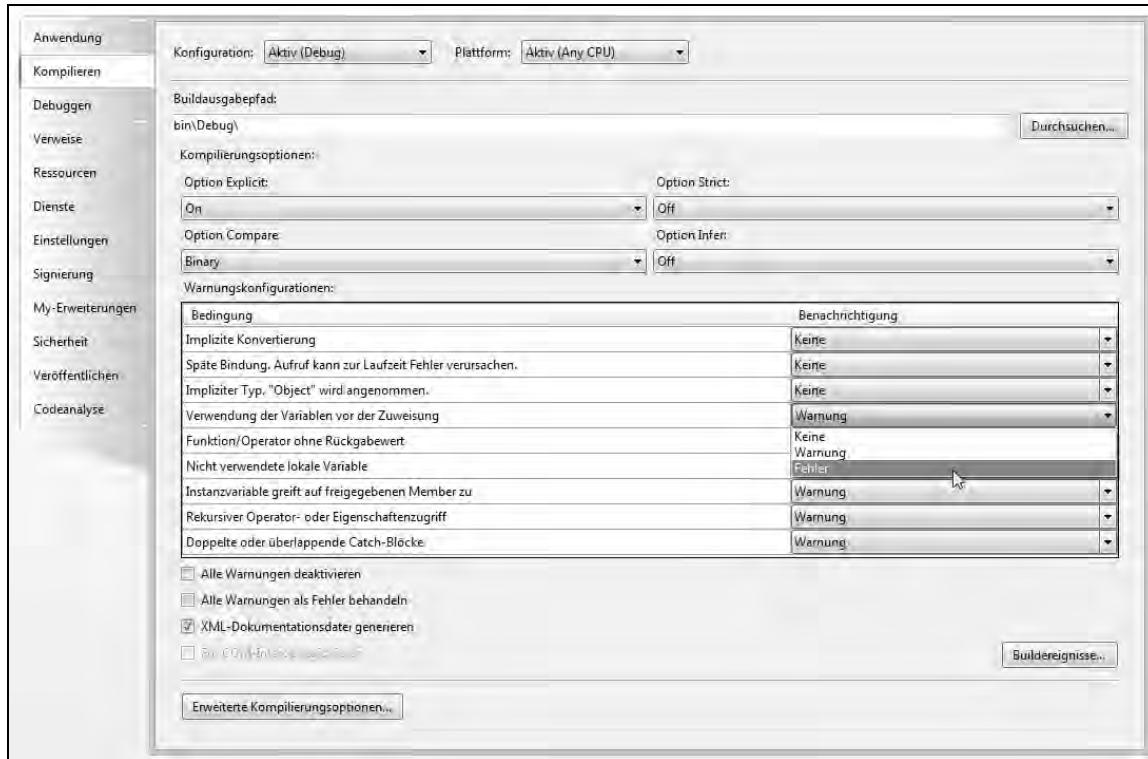


Abbildung 4.12 Mit dem Register *Kompilieren* stellen Sie das Fehlermeldungsverhalten eines Projektes ein

Um die Eigenschaften eines Projektes zu öffnen, rufen Sie das Kontextmenü des entsprechenden Projektes mit der rechten Maustaste auf und wählen anschließend *Eigenschaften*. Sie sehen nun einen Dialog, wie er etwa auch in Abbildung 4.12 zu sehen ist. Dort können Sie dann die verschiedenen Meldungstypen individuell konfigurieren.

Die Aufgabenliste

Die Aufgabenliste verhält sich in Visual Basic 2008 ähnlich wie die Fehlerliste, nur mit dem Unterschied, dass dort Punkte erscheinen, die Sie selber eintragen.

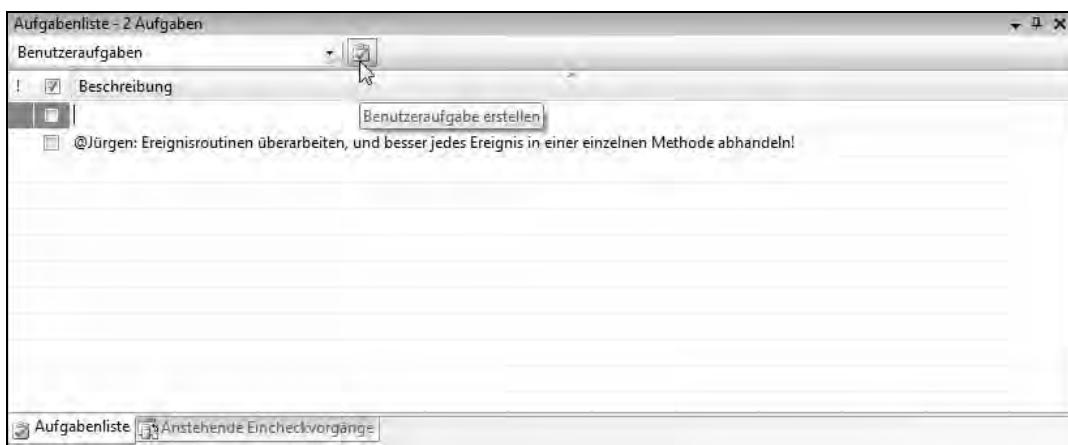


Abbildung 4.13 Die Aufgabenliste zeigt in Abhängigkeit der Auswahl in der Aufklappliste entweder benutzerdefinierte, manuell hinzugefügte Aufgaben oder durch Kommentare im Code eingefügte

Meldungen, die in der Aufgabenliste erscheinen, können durch zwei verschiedene Methoden dort hineinge-
langen.

- Durch das manuelle Hinzufügen einer Aufgabe zur Aufgabenliste. Dazu wählen Sie aus der Aufklapp-
liste den Punkt Benutzeraufgaben und klicken Sie anschließend auf das rechts daneben stehende Sym-
bol, etwa wie in Abbildung 4.13 zu sehen.
- Durch Einfügen von Kommentaren innerhalb des Listings. In Abbildung 4.5 sehen Sie unter Punkt ③
einen Pfeil, der vom Codefenster in das Aufgabenfenster reicht. Sobald Sie einen Kommentar im Code-
listing, wie dort zu sehen, einfügen, der mit bestimmten Schlüsselworten beginnt, sehen Sie diesen
Kommentar in der Aufgabenliste – vorausgesetzt Sie haben zuvor in der Aufgabenliste aus der Auf-
klapliste den Punkt *Kommentare* gewählt.

Die Schlüsselwörter, auf die das Aufgabenfenster sozusagen »reagiert«, lassen sich übrigens nach Belieben konfigurieren. Wählen Sie dazu aus dem Menü *Extras* den Menüpunkt *Optionen*. Im Bereich *Umge-
bung/Aufgabenliste* konfigurieren Sie vorhandene Schlüsselwörter (so genannte »Tokens«) oder richten weitere ein. Abbildung 4.14 zeigt, wie es geht. Neben der Einrichtung der Schlüsselworte können Sie auch deren Priorität festlegen. Dazu wählen Sie aus der Aufklapliste die *Priorität*, die eine Codezeile mit dem entsprechenden Token automatisch bekommen soll, wenn sie in die Aufgabenliste eingetragen wird. Hohe oder niedrige Prioritäten in der Liste werden anschließend mit einem entsprechenden Symbol zur besseren Wiedererkennung gekennzeichnet.

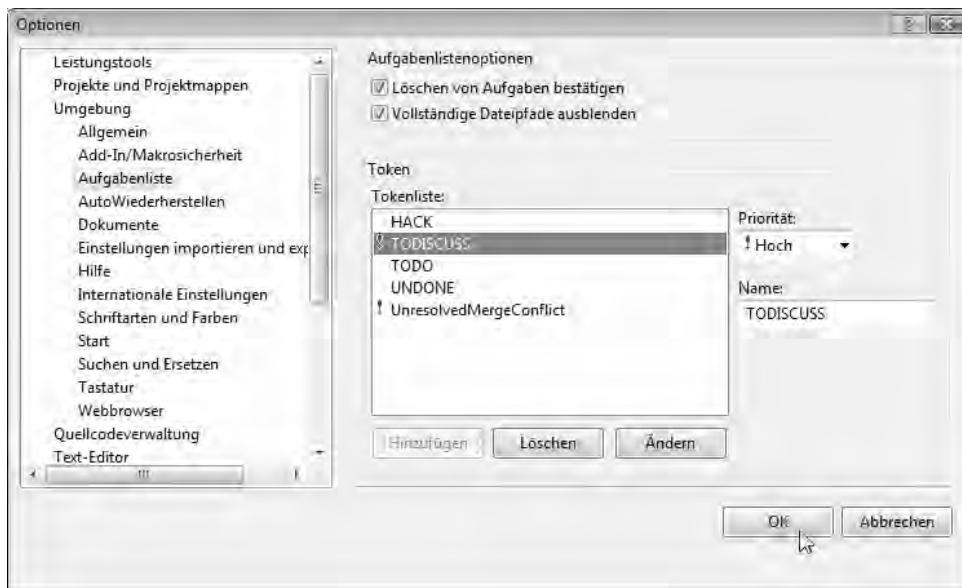


Abbildung 4.14 Zusätzliche Schlüsselwörter für die Aufnahme von Aufgaben in die Aufgabenliste durch Codekommentare lassen sich individuell konfigurieren

Tipps für die Aufgaben- und die Fehlerliste

Sowohl die Aufgaben- als auch die Fehlerliste enthalten Listenelemente, die Sie mithilfe der Spaltenköpfe sortieren können. Klicken Sie dazu auf einen Spaltenkopf, um die Liste nach der Spalte zu sortieren. Klicken Sie ein zweites Mal auf den gleichen Spaltenkopf, wird die Liste nach dieser Spalte in absteigender Reihenfolge sortiert.

Navigieren mit der Aufgaben- und der Fehlerliste

Mithilfe beider Fenster können Sie im Codeeditor navigieren. Möchten Sie zu einem in der Aufgabenliste ausgewiesenen Kommentar im Code gelangen, doppelklicken Sie einfach auf den entsprechenden Listeneintrag. Das gleiche gilt für Fehler, die Sie in der Fehlerliste sehen.

Das Ausgabefenster

Das Ausgabefenster in Visual Studio 2008 erfüllt zwei Funktionen. Zum einen gibt es während des Komplierens eines Projektes oder einer Projektmappe die Meldungen des Compilers aus. Zum anderen zeigt es bestimmte Statusmeldungen während des Programmablaufs an oder erlaubt auch, dass Ihre eigenen Programme beispielsweise mit einer Anweisung wie `Debug.Print` eigene Ausgaben im Ausgabefenster vornehmen. Welche Ausgabetypen im Ausgabefenster dargestellt werden, bestimmen Sie mit der Aufklappliste *Ausgabe anzeigen von*.

Ausgabe anzeigen von Erstellen

Abbildung 4.15 zeigt Ihnen beispielhaft die Ausgabe eines Projekt-Builds, in dem (mit Absicht natürlich) zwei Compiler-Fehler aufgetreten sind.

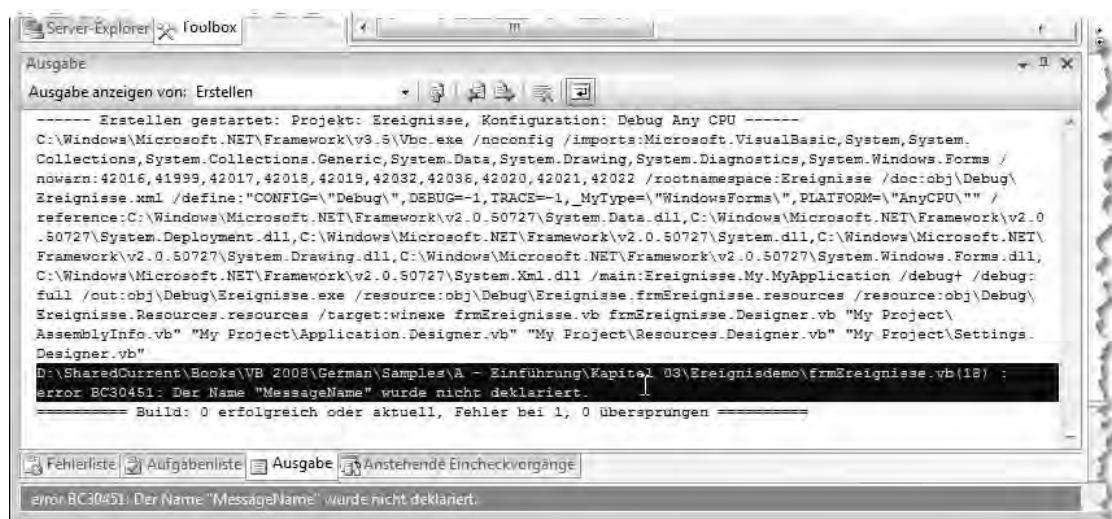


Abbildung 4.15 Dieses Beispiel zeigt die Build-Protokollierung (Ausgabe von Erstellen) eines Visual Basic-Projektes mit normaler Ausführlichkeit

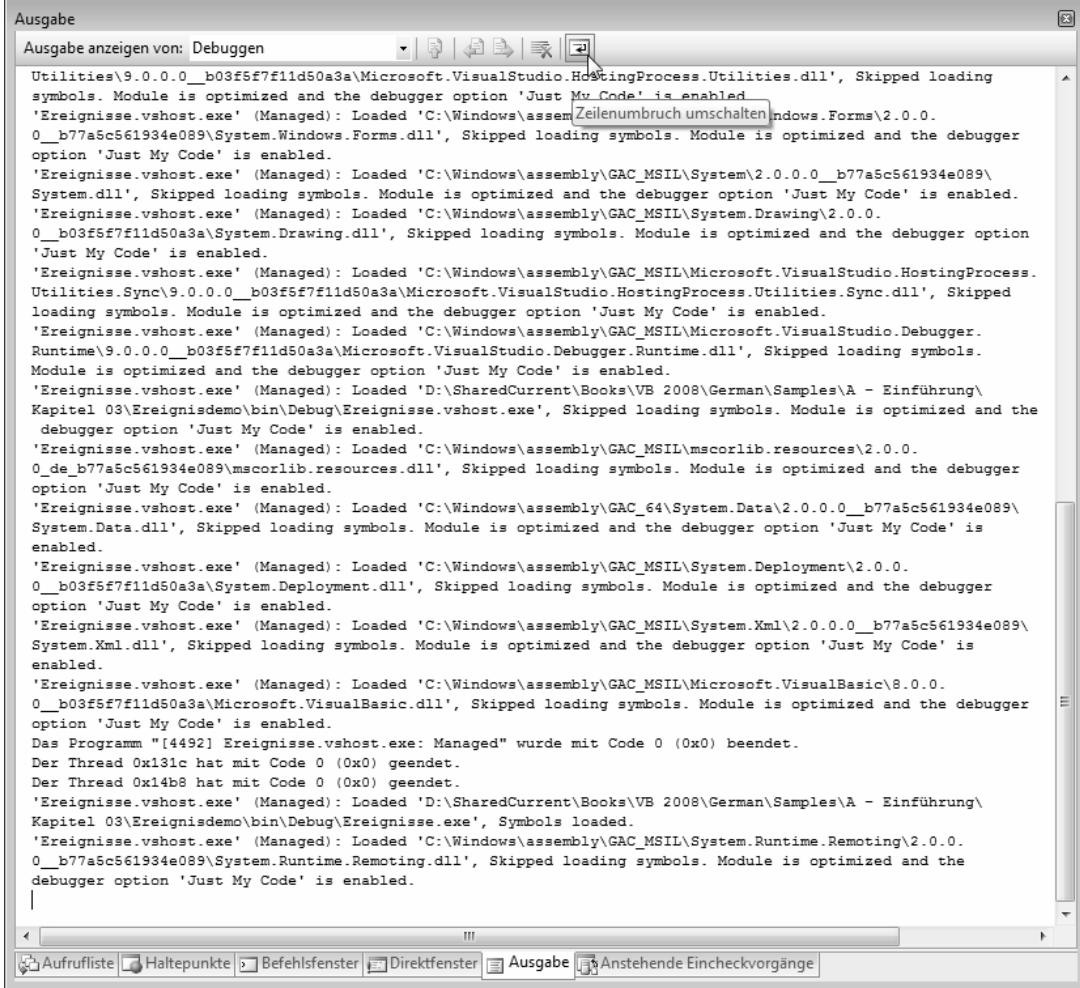
Mit den Symbolen neben der Aufklappliste am oberen Fensterrand haben Sie die Möglichkeit, bestimmte Funktionalitäten abzurufen, die Ihnen das Ausgabefenster zur Verfügung stellt. Die folgende Tabelle zeigt, welche Funktionen es gibt:

Symbol	Aufgabe
	Wenn Sie in das Ausgabefenster auf eine Compiler-Fehlermeldung klicken, können Sie dieses Symbol als Navigationshilfe verwenden und zur entsprechenden Zeile im Code springen, an der der Fehler aufgetreten ist. Alternativ bringt Sie ein Doppelklick auf die entsprechende Compiler-Fehlermeldung ebenfalls zur dazugehörigen Quellcodetextstelle. Abbildung 4.5 verdeutlicht das auch unter Punkt ③.
	Falls es im Ausgabefenster mehr als eine Meldung wie beispielsweise Fehlermeldungen des Compilers gibt, können Sie mit diesem Symbol zur vorherigen Meldung navigieren.
	Falls es im Ausgabefenster mehr als eine Meldung wie beispielsweise Fehlermeldungen des Compilers gibt, können Sie mit diesem Symbol zur nächsten Meldung navigieren.
	Löscht den Inhalt des Ausgabefensters.
	Schaltet den Zeilenumbruch ein. In diesem Fall werden eigentlich einzeilige Meldungen in neue Zeilen umbrochen, falls der Platz im Fenster nicht ausreicht. Der horizontale Schiebebalken wird dann natürlich nicht mehr benötigt und verschwindet (siehe auch Tooltip folgender Grafik).

Tabelle 4.3 Symbole, mit der Sie erweiterte Funktionalitäten des Ausgabefensters steuern

Ausgabe anzeigen von Debuggen

Diese Ausgaben stehen Ihnen nur zur Verfügung, wenn Ihr Projekt im Debug-Modus gestartet wird bzw. beispielsweise durch das Setzen eines Haltepunktes unterbrochen wurde. Abbildung 4.16 zeigt eine typische Ausgabe einer Windows Forms-Anwendung nach dem Start.



```

Ausgabe
Ausgabe anzeigen von: Debuggen
Utilities\9.0.0.0_b03f5f7f11d50a3a\Microsoft.VisualStudio.HostingProcess.Utilities.dll', Skipped loading
symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
'Ereignisse.vhost.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_MSIL\System\2.0.0.0_b77a5c561934e089\System.dll', Skipped loading symbols. Module is optimized and the debugger
option 'Just My Code' is enabled.
'Ereignisse.vhost.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_MSIL\System.Drawing\2.0.0.0_b03f5f7f11d50a3a\System.Drawing.dll', Skipped loading symbols. Module is optimized and the debugger option
'Just My Code' is enabled.
'Ereignisse.vhost.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_MSIL\Microsoft.VisualStudio.HostingProcess.Utilities.Sync\9.0.0.0_b03f5f7f11d50a3a\Microsoft.VisualStudio.HostingProcess.Utilities.Sync.dll', Skipped
loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
'Ereignisse.vhost.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_MSIL\Microsoft.VisualStudio.Debugger.Runtime\9.0.0.0_b03f5f7f11d50a3a\Microsoft.VisualStudio.Debugger.Runtime.dll', Skipped loading symbols.
Module is optimized and the debugger option 'Just My Code' is enabled.
'Ereignisse.vhost.exe' (Managed): Loaded 'D:\SharedCurrent\Books\VB 2008\German\Samples\A - Einführung\Kapitel 03\Ereignisdemo\bin\Debug\Ereignisse.vhost.exe', Skipped loading symbols. Module is optimized and the
debugger option 'Just My Code' is enabled.
'Ereignisse.vhost.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_MSIL\mscorlib.resources\2.0.0.0_de_b77a5c561934e089\mscorlib.resources.dll', Skipped loading symbols. Module is optimized and the debugger
option 'Just My Code' is enabled.
'Ereignisse.vhost.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_64\System.Data\2.0.0.0_b77a5c561934e089\System.Data.dll', Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is
enabled.
'Ereignisse.vhost.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_MSIL\System.Deployment\2.0.0.0_b03f5f7f11d50a3a\System.Deployment.dll', Skipped loading symbols. Module is optimized and the debugger
option 'Just My Code' is enabled.
'Ereignisse.vhost.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_MSIL\System.Xml\2.0.0.0_b77a5c561934e089\System.Xml.dll', Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is
enabled.
'Ereignisse.vhost.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_MSIL\Microsoft.VisualBasic\8.0.0.0_b03f5f7f11d50a3a\Microsoft.VisualBasic.dll', Skipped loading symbols. Module is optimized and the debugger
option 'Just My Code' is enabled.
Das Programm "[4492] Ereignisse.vhost.exe: Managed" wurde mit Code 0 (0x0) beendet.
Der Thread 0x131c hat mit Code 0 (0x0) geendet.
Der Thread 0x14b8 hat mit Code 0 (0x0) geendet.
'Ereignisse.vhost.exe' (Managed): Loaded 'D:\SharedCurrent\Books\VB 2008\German\Samples\A - Einführung\Kapitel 03\Ereignisdemo\bin\Debug\Ereignisse.exe', Symbols loaded.
'Ereignisse.vhost.exe' (Managed): Loaded 'C:\Windows\assembly\GAC_MSIL\System.Runtime.Remoting\2.0.0.0_b77a5c561934e089\System.Runtime.Remoting.dll', Skipped loading symbols. Module is optimized and the
debugger option 'Just My Code' is enabled.

```

Abbildung 4.16 Dieses Beispiel zeigt Meldungen im Ausgabefenster (Auszage von Debuggen) eines Visual Basic-Projektes nach dem Start im Debug-Modus

Die dynamische Hilfe

Das Fenster »dynamische Hilfe« bietet Ihnen Schlagwörter zum aktuellen Kontext an, mit deren Hilfe Sie sich per Mausklick das entsprechende Thema in der Hilfe anzeigen lassen können. Abbildung 4.5 zeigt die Verbindung von aktuellem Kontext im Toolfenster (Punkt ⑥ zu Punkt ⑦) zur dynamischen Hilfe mit einem

Pfeil. Diese Verbindungsherstellung funktioniert natürlich auch in anderen Zusammenhängen – sehr flexibel und stets brauchbar gerade dann, wenn Sie im Editor arbeiten.

TIPP Der aktuelle Kontext spiegelt sich in der dynamischen Hilfe wider, wann immer Visual Studio ein Schlüsselwort oder einen Objektnamen im Visual Basic-Editor erkannt hat und sich der Cursor darauf befindet. Bei Ereignissen funktioniert das natürlich nur dann, wenn Sie sich mit dem Cursor im eigentlichen Ereignisnamen befinden – und der »eigentliche Ereignisname« befindet sich hinter dem `Handles`-Schlüsselwort.

Im Zusammenhang mit IntelliSense ist die dynamische Hilfe das ideale Werkzeug, sich schnell über die Methoden, Eigenschaften und anderen Elemente eines bestimmten Objektes bzw. dessen Klasse zu informieren.

HINWEIS

Visual Basic 2008 Express Edition verfügt leider nicht über die dynamische Hilfe.

Performance-Einbrüche im Editor verursacht durch die dynamische Hilfe

Wenn der Editor gegen Ende einer umfangreichen Quellcodedatei auf nicht so leistungsfähigen Maschinen zu langsam wird, könnte die dynamische Hilfe der Grund dafür sein. Schließen Sie das Fenster der dynamischen Hilfe einfach, und Sie können anschließend Ihre Quellcodedatei in der gewohnten Geschwindigkeit bearbeiten.

Anpassen des Inhalts der dynamischen Hilfe

Sie können den Inhalt der dynamischen Hilfe anpassen. Dazu rufen Sie mit *Extras | Optionen* den Optionsdialog von Visual Studio auf.

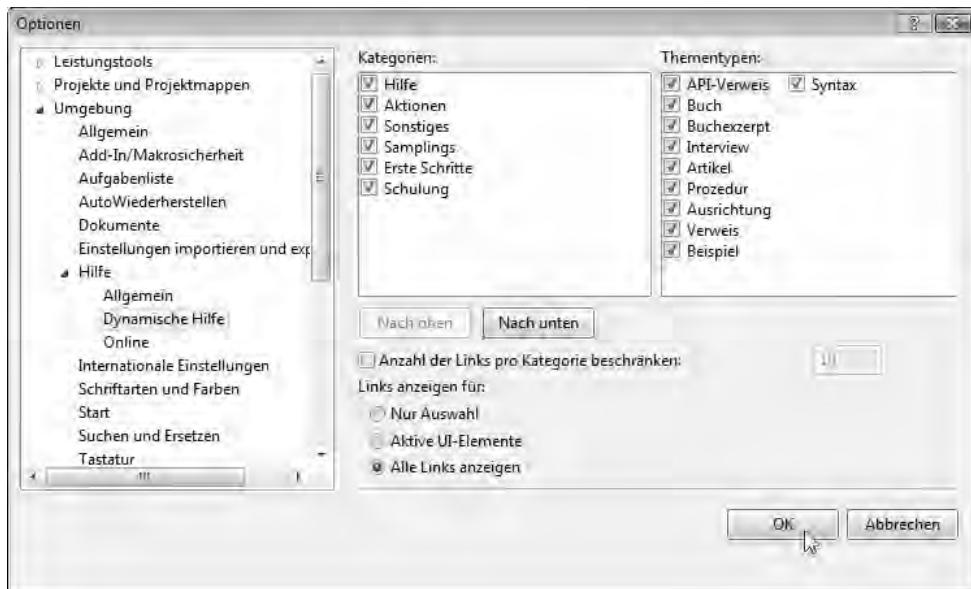


Abbildung 4.17 Hier konfigurieren Sie, wie umfangreich die dynamische Hilfe Hilfethemen finden soll

Unter *Umgebung/Hilfe/Dynamische Hilfe* finden Sie die Registerkarte (siehe Abbildung), mit der Sie die Themen bestimmen können, die im Kontext angezeigt werden sollen.

TIPP Wenn Sie Performance-Einbußen im Editor feststellen, die durch die dynamische Hilfe verursacht werden (siehe auch vorheriger Absatz), Sie aber dennoch nicht komplett auf die dynamische Hilfe verzichten möchten, wählen Sie hier so wenig wie möglich Kategorien und Thementypen aus und beschränken Sie sich auf das Wesentliche. Auf diese Weise erhöhen Sie die Performance in der Regel schon ausreichend, um zügig weiterarbeiten zu können.

Die Klassenansicht

Die Klassenansicht dient zur aufgeteilten Ansicht Ihres Projektes auf Klassenbasis und kann Ihnen auch bei der Navigation im Projekt behilflich sein. Die Klassenansicht teilt die Elemente eines Projektes hierarchisch ein – Sie müssen nur auf das jeweilige Pluszeichen vor einem Element klicken, um eine Ebene zu öffnen. Ein Doppelklick auf das entsprechende Element bringt Sie anschließend zur entsprechenden Definition im Quellcode. In Abbildung 4.5 finden Sie unter ❸ ein Beispiel für das Klassenansicht-Toolfenster, das in diesem Fall als Dokumentfenster in einer Registerkartengruppe eingefügt wurde.

Codeeditor und Designer

Bei diesen Werkzeugen hat sich seit Visual Studio 2003 eine ganze Menge getan. So gibt es beispielsweise beim Designer eine ausgeklügelte Positionierungshilfe; der Codeeditor glänzt, wie in Abbildung 4.5 unter ❸ in der Vergrößerung zu sehen, mit zusätzlichen Orientierungshilfen zum Speicherzustand und anderen Infos. Diesen beiden wichtigen Werkzeugen sei daher ein eigenes – das anschließende – Kapitel gewidmet.

Wichtige Tastenkombinationen auf einen Blick

Falls Sie, wie am Anfang des Kapitels beschrieben wurde, die allgemeinen Entwicklungseinstellungen gewählt haben, verwenden Sie für häufige Aufgaben die folgende Tabelle, um mit Tastaturskürzeln entsprechende Befehle aufzurufen. Falls Sie Ihr Befehlsmapping für die Tastatur geändert haben – Sie können die Befehle wie in Abbildung 4.18 zu sehen, anpassen, indem Sie aus dem Menü *Extras* den Befehl *Optionen* wählen, und unter *Umgebung* den Unterpunkt *Tastatur* selektieren. Wählen Sie den Befehl aus der Liste aus, bestimmen Sie, wo die Tastenkombination verwendet werden soll, klicken Sie in das Feld *Tastenkombination drücken*, drücken Sie die neue Kombination auf der Tastatur, und klicken Sie anschließend auf *Zuweisen*, um die neue Tastenkombination zuzuordnen.

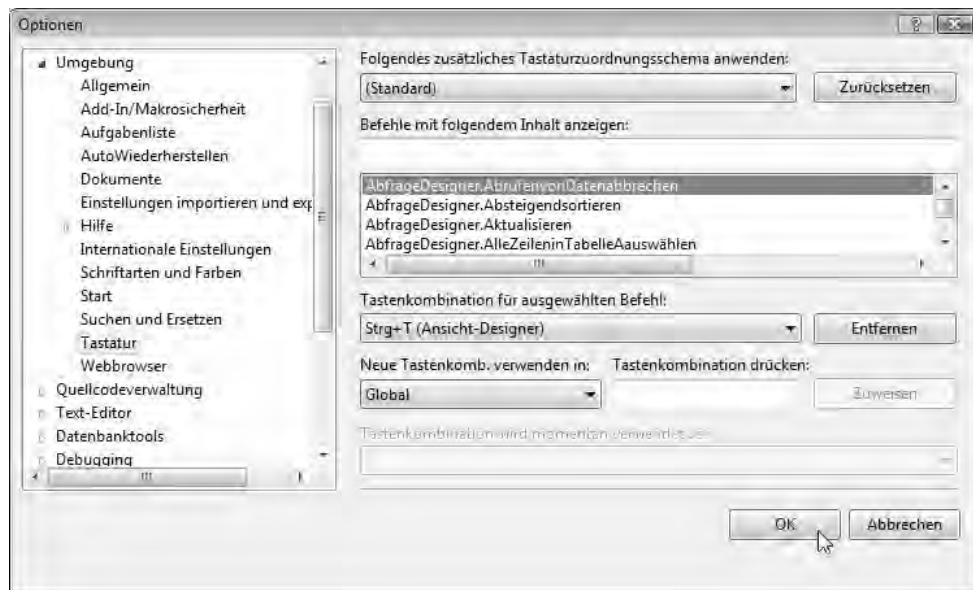


Abbildung 4.18 Mit diesem Dialog können Sie Tastenkürzel für Befehle anpassen

Kurzbeschreibung	Tastenkombination	Beschreibung	Befehlsname
Cursor zum nächsten Element	[F8]	Verschiebt den Cursor zum nächsten Element – beispielsweise einer Aufgabe oder einem Fehler im Aufgabenfenster.	Bearbeiten.GehezunächsterInstanz
Cursor zum vorherigen Element	[Shift] [F8]	Verschiebt den Cursor zum vorherigen Element (Fehler, Aufgabe im Aufgabenfenster).	Bearbeiten._GehezuvorherigerInstanz
Cursor zur Definition	[Shift] [F12]	Verschiebt den Cursor zur Definition des Elementes, das sich derzeit unterhalb des Cursors befindet.	Bearbeiten.GehezuVerweis
Rückgängig	[Strg] [Z] oder [Alt] [←]	Macht die letzte Aktion rückgängig.	Bearbeiten.Rückgängig
Rückgängig rückgängig machen	[Strg] [Y]	Stellt die zuvor rückgängig gemachte Aktion wieder her.	Bearbeiten.Wiederholen
Alles speichern	[Strg] [S]	Speichert alle Dateien, an denen seit dem letzten Speichern Änderungen vorgenommen wurden.	Datei.AllesSpeichern
Zur letzten Änderung springen	[Strg] [–]	Setzt den Cursor auf die Position im Code, an der Sie die letzte Änderung vorgenommen haben. Sie können nicht nur einen Schritt zurücknavigen.	Ansicht.Rückwärtsnavigieren
Zur vorherigen Änderung springen	[Strg] [Shift] [–]	Nachdem Sie rückwärts navigiert haben (siehe vorherigen Eintrag), erreichen Sie damit wieder die ursprüngliche Position.	Ansicht.Vorwärtsnavigieren

Kurzbeschreibung	Tastenkombination	Beschreibung	Befehlsname
Suchdialog öffnen	[Strg] [F]	Öffnet den Suchen-Dialog.	Bearbeiten.Suchen
Inkrementelles Suchen	[Strg] [I]	Startet das inkrementelle Suchen.	Bearbeiten._InkrementelleSuche
Zur Zeile mit Nummer springen	[Strg] [G]	Öffnet den Dialog »Gehe zu Zeilennummer« und erlaubt den Sprung zur Zeile mit angegebener Nummer.	Bearbeiten.GeheZu
Automatischen Zeilenumbruch ein- und ausschalten	[Strg] [R], [Strg] [R]	Schaltet den automatischen Zeilenumbruch ein und aus. Drücken Sie beide Tastenkombinationen dazu nacheinander.	Bearbeiten._Zeilenumbruchumschalten
Lesezeichen einfügen/entfernen	[Strg] [K], [Strg] [K]	Schaltet ein Lesezeichen in einer Zeile ein bzw. wieder aus, wenn bereits eines gesetzt war.	Bearbeiten._Lesezeichenumschalten
Zum nächsten Lesezeichen	[Strg] [K], [Strg] [N]	Setzt den Cursor in die Zeile, in der sich das nächste Lesezeichen befindet.	Bearbeiten._NächstesLesezeichen
Zum vorherigen Lesezeichen	[Strg] [K], [Strg] [P]	Setzt den Cursor in die Zeile, in der sich das vorherige Lesezeichen befindet.	Bearbeiten._VorherigesLesezeichen
Aufgabeneintrag für aktuelle Codezeile hinzufügen/löschen	[Strg] [K], [Strg] [H]	Fügt einen Aufgabeneintrag in die Aufgabenliste mit Verweis auf die aktuelle Zeile hinzu bzw. löscht den Eintrag wieder, wenn für die Zeile bereits einer vorhanden ist.	Bearbeiten._Aufgabenverknüpfung_umschalten

Tabelle 4.4 Die wichtigsten Tastaturkommandos. Bitte beachten Sie, dass die Kommandos zur Tastaturanpassung in VB-Manier mit dem Unterstrich getrennt sind und eigentlich eine Zeile bilden

Verbesserungen an der integrierten Entwicklungs- umgebung (IDE) in Visual Studio 2008

Auch wenn Sie nicht sofort den Eindruck gewinnen: auf den zweiten Blick wird erkennbar, dass sich auch an der IDE eine Menge seit der 2005er-Version von Visual Studio getan hat. Die folgenden Abschnitte beschreiben die Neuerungen.

Einfacheres Positionieren und Andocken von Toolfenstern

Das Verschieben und Andocken von Toolfenstern ist leichter geworden; die Zielposition lässt sich leichter erkennen und bestimmen, wie in der folgenden Abbildung zu sehen.

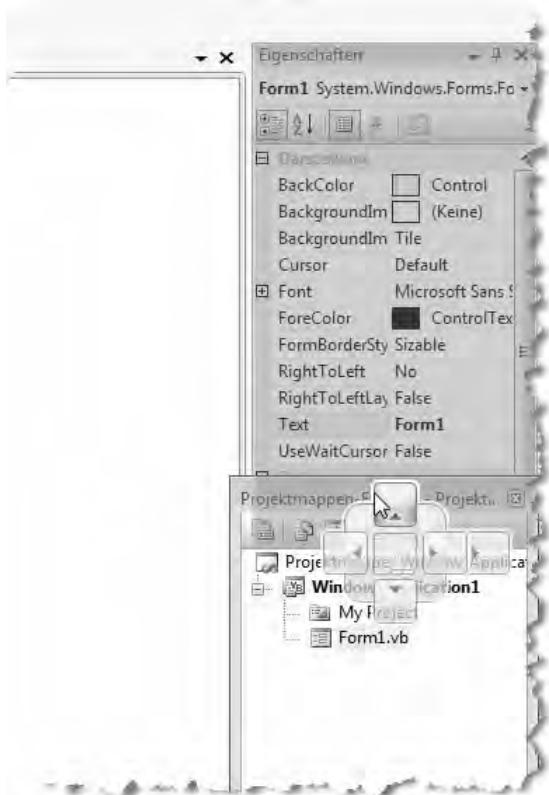


Abbildung 4.19 Das Andocken von Toolfenstern wurde überarbeitet – Fenster lassen sich leichter positionieren, Zielpositionen leichter erkennen

Navigieren durch Dateien mit dem IDE-Navigator

Um im Codeeditor zu einer bestimmten geöffneten Datei zu navigieren, egal wann diese zuletzt verwendet wurde, können Sie den IDE-Navigator verwenden. Die Funktionsweise des IDE-Navigators ist in etwa dieselbe, wie die des Task-Switchers in Windows selbst, den Sie mit der Tastenkombination **Alt** **←** bedienen. Der IDE-Navigator ist nicht über die Menüs verfügbar, sondern wird mithilfe von Tastenkombinationen aufgerufen. Je nachdem, in welcher Reihenfolge Sie durch die Dateien navigieren möchten, können Sie den IDE-Navigator über einen von zwei Befehlen aufrufen. In den allgemeinen Entwicklungseinstellungen sind dafür die Tastenkombination **Strg** **↑** **←** und die Tastenkombination **Strg** **←** zuständig.

Wechseln zu bestimmten Dateien im Editor

Drücken Sie **Strg** **←**, um den IDE-Navigator anzuzeigen (siehe Abbildung 4.20).



Abbildung 4.20 Den IDE-Navigator rufen Sie mit **Strg** ↩ oder **Strg** ⌘ ↩ ins Leben

Drücken Sie bei gehaltener **Strg**-Taste so oft ↩, bis die gewünschte Datei ausgewählt ist.

TIPP Um die Reihenfolge umzukehren, in der Sie durch die Liste der aktiven Dateien navigieren, halten Sie **Strg** ⌘ gedrückt, und drücken Sie dabei ↩. Sie können Sie auch direkt mit der Maus anklicken.

Alternativ klicken Sie in der oberen rechten Ecke des Editors auf die Schaltfläche *Aktive Dateien* neben der Schaltfläche *Schließen* und wählen die gewünschte Datei aus der Liste aus.

Navigieren zwischen Toolfenstern in der IDE

Mit dem IDE-Navigator können Sie auch zwischen den Toolfenstern navigieren, die in der IDE geöffnet sind. Je nachdem, in welcher Reihenfolge Sie durch die Toolfenster navigieren möchten, können Sie den IDE-Navigator über einen von zwei Befehlen aufrufen. Mit ⌘ Alt F7 können Sie zu dem zuletzt verwendeten Toolfenster navigieren, und Alt F7 ermöglicht Ihnen die Navigation in umgekehrter Reihenfolge.

Umgebungsschriftart definieren

Für nicht näher bestimmte IDE-Elemente konnte man bislang keine Schriftart definieren, was insbesondere bei der Entwicklung auf Monitoren größer als 24 Zoll Probleme machte. Wählen Sie aus dem Menü *Extras* den Menüpunkt *Optionen*, und dann *Umgebung* und *Schriftarten und Farben*. In der Klappliste *Einstellungen anzeigen für Umgebungsschriftart* wählen Sie die Option *Umgebungsschriftart*. Anschließend bestimmen Sie eine neue Schriftart und den entsprechenden Schriftgrad.

Betroffen von der Umgebungsschriftart sind alle Schriften, die Sie nicht extra festlegen können, also Pull-down-Menüschriften, Dialogschriften, Projektmappen-Explorer, Toolbox, Registerkartenbeschriftungen, wie beispielsweise in Abbildung 4.21 zu sehen.

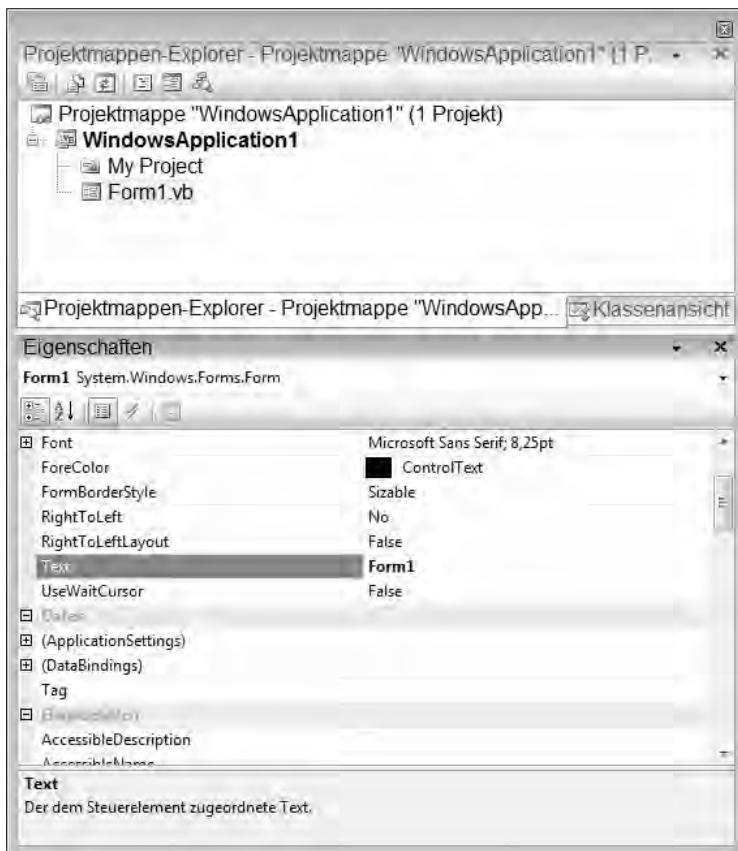


Abbildung 4.21 Mit der Umgebungs-schriftart lässt sich die Basisschriftart der IDE verändern, und das hat Auswir-kungen auf verschiedene Toolfenster, Dialoge und Pulldown-Menüs – wie hier im Bild beispielsweise im Projektmappen-Explorer

HINWEIS Nach der Konvertierung eines Projektes in das Visual Basic 2008-Format konvertierte Projekt auf das .NET Framework 2.0 – Sie haben hier also weiterhin »nur« die Möglichkeit, alte .NET 2.0-Funktionalität in Ihrem Projekt weiterzuverwenden, aber dafür ist das Kompilat Ihres Projektes auch in der Lage, ältere Windows 2000-Clients⁵ zu bedienen, die mit den neueren Framework-Versionen nicht ausgestattet werden können.

Visual Basic 2008 erlaubt es jedoch, auch gegen unterschiedliche .NET Framework-Versionen zu entwickeln – der nächste Absatz hält dazu genauere Informationen bereit. Möchten Sie Ihr konvertiertes Projekt umstellen, sodass Sie auch die Funktio-nalitäten der neueren 3.0 bzw. 3.5 Framework-Versionen nutzen können, ändern Sie die Projekteinstellungen, wie in Kapitel 11 beschrieben.

⁵ Oder, mit eingeschränkter Funktionalität, sogar alte Windows 98-Clients – doch das sei wirklich nur der Vollständigkeit halber erwähnt, denn schon aus Sicherheitsaspekten sollten Sie Ihren Kunden gegenüber fairerweise gar nicht mehr erwähnen, dass Ihre Software theoretisch auch unter Windows 98 lauffähig ist.

Designer für Windows Presentation Foundation-Projekte

Mit dem Windows Presentation Foundation (WPF)-Designer können Sie WPF-Anwendungen und benutzerdefinierte Steuerelemente in der IDE erstellen. Der WPF-Designer kombiniert die Echtzeitbearbeitung der XAML (Extended Application Markup Language) mit einer verbesserten grafischen Entwurfszeiterfahrung. Ein neues WPF-Projekt erstellen Sie, indem Sie

1. aus dem Menü *Datei* den Befehl *Neu/Projekt* auswählen.
2. im Dialog, der jetzt erscheint, unter *Projekttypen* den Eintrag *Windows* wählen und unter *Vorlagen* den Eintrag *WPF-Anwendung*.
3. unter *Name* einen neuen Namen für Ihr Projekt bestimmen, ebenso den Speicherort und anschließend auf *OK* klicken, um das neue WPF-Projekt bearbeiten zu können.

Der WPF-Designer unterscheidet sich vom Windows Forms-Designer grundlegend. Anders als beim Windows Forms-Designer kann der Aufbau eines WPF-Forms in einer so genannten XAML-Datei gespeichert werden – der Aufbau eines WPF-Forms nur durch reinen Code wie beim Windows Forms-Designer ist natürlich ebenfalls möglich; die Trennung zwischen Code und Aufbaubeschreibung macht es jedoch möglich, dass ein beauftragter Designer unabhängig vom Entwickler der Formularlogik das Formular designen kann.

Dem Thema WPF ist ein eigenes Kapitel in diesem Buch gewidmet (siehe Kapitel 6 und Ab Kapitel 41). Die hier beschriebenen Features und Vorgehensweisen sollen nur einen kurzen und groben Überblick über die Neuheiten aus IDE-Sicht bieten.

Die folgenden Features sind im WPF-Designer neu.

- Mithilfe der *SplitView*-Funktionalität können Sie Objekte im grafischen Designer anpassen und die Änderungen des zugrunde liegenden XAML-Codes direkt anzeigen lassen. Entsprechend werden Änderungen am XAML-Code sofort im grafischen Designer umgesetzt. Abbildung 4.22 vermittelt Ihnen einen Eindruck davon.

HINWEIS Da der WPF-Designer lange nicht über den gleichen Funktionsumfang wie der Windows Forms-Designer verfügt, ist dieses erste Feature auch gleichzeitig sein bestes: In vielen Fällen ist es leichter, den für den Formularaufbau erforderlichen XAML-Code zu erstellen, als zu versuchen, beispielsweise Steuerelementkombinationen mit dem Designer interaktiv zusammenzuklicken. Aus diesem Grund werden wir später, ab dem 6. Kapitel, das Augenmerk auch mehr auf XAML an sich als auf den Umgang mit dem Designer legen.

Eine viel bessere Alternative zum eingebauten WPF-Designer ist das Tool *Expression Blend*, mit dem Sie sogar WPF-konform interaktive Animationen erstellen können – ein Feature, das der eingebaute Designer überhaupt nicht unterstützt. Mehr zum Thema Expression Blend erfahren Sie unter <http://www.microsoft.com/expression/>.

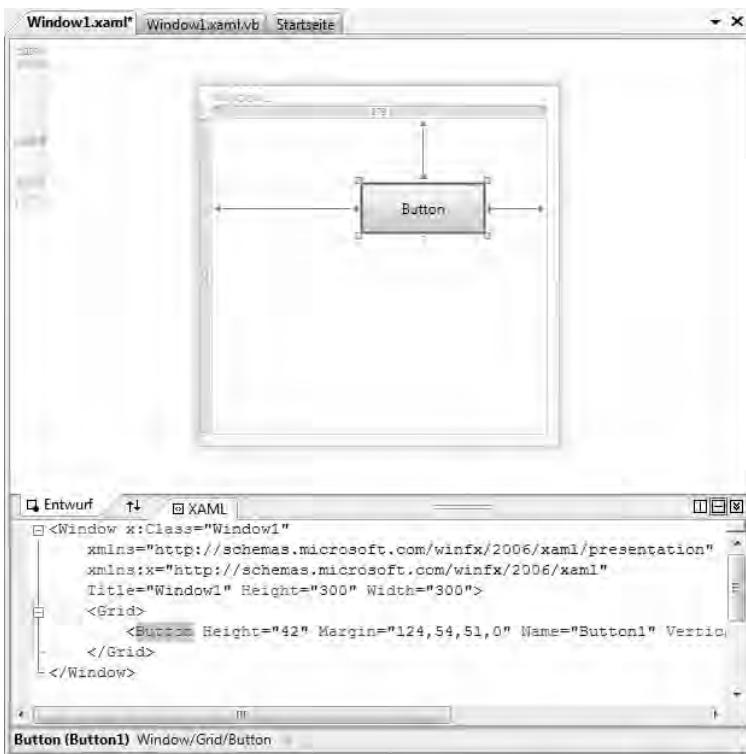


Abbildung 4.22 Der WPF-Designer bietet Split View-Funktionalität:
Grafische Änderungen werden direkt in XAML-Code umgesetzt, Änderungen im XAML-Code spiegeln sich direkt in der grafischen Repräsentation wieder

- Im Fenster *Dokumentgliederung* können Sie den XAML-Code bei vollständiger Auswahl synchronisierung zwischen Designer, Dokumentgliederung, XAML-Editor und Eigenschaftenfenster anzeigen lassen und darin navigieren.
- IntelliSense im XAML-Editor ermöglicht den schnellen Codeeintrag. IntelliSense unterstützt jetzt selbstdefinierte Typen.

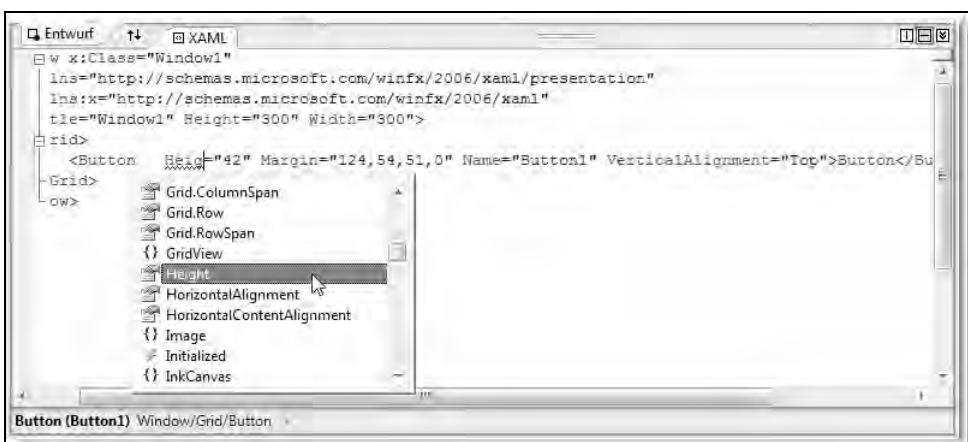


Abbildung 4.23 Der XAML-Editor verfügt über volle IntelliSense-Unterstützung

- Rasterlinien können den Rastern im Designer hinzugefügt werden, um die einfache rasterbasierte Platzierung von Steuerelementen zu ermöglichen.
- Steuerelemente und Text können leicht an Ausrichtungslinien ausgerichtet werden.
- Der Designer unterstützt jetzt das Laden der von Ihnen definierten Typen. Dazu gehören benutzerdefinierte Steuerelemente und Benutzersteuerelemente.
- Sie können das Laden großer XAML-Dateien abbrechen.
- Die Entwurfszeiterweiterung unterstützt Entwurfsmodus und Eigenschaften-Editoren.

Dem Thema WPF ist ein eigener Buchteil gewidmet, der sich auch ein wenig näher mit dem WPF-Designer auseinandersetzt. In Kapitel 6 und dann ab Kapitel 41 finden Sie Näheres zu diesem Thema.

IntelliSense-Verbesserungen

Das Visual Basic-Team hat die Unterstützung des Entwicklers beim Coden durch IntelliSense in Visual Studio 2008 in verschiedenen Punkten verbessert.

Syntax-Tooltipps

Am auffälligsten ist die Einführung so genannter Syntax-Tooltipps, wie Sie sie vielleicht schon kennen, sollten Sie nicht nur in Visual Basic, sondern auch schon in C# 2005 entwickelt haben. In Visual Basic 2008 zeigt die Entwicklungsumgebung das IntelliSense-Fenster bereits an, wenn Sie die ersten Buchstaben gedrückt haben (siehe folgende Abbildungen).

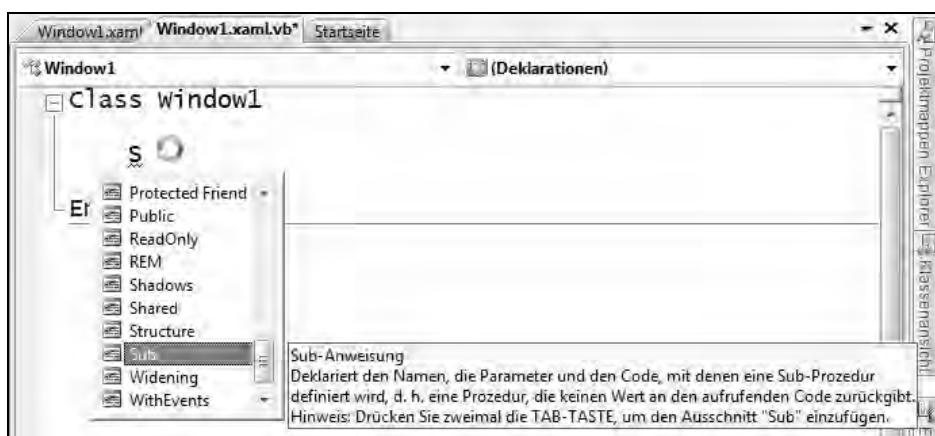


Abbildung 4.24 IntelliSense wird bereits aktiv, wenn Sie den ersten Buchstaben in das Code Fenster eingeben und unterstützt Sie nicht nur mit der Auflistung bestimmter Schlüsselworte ...

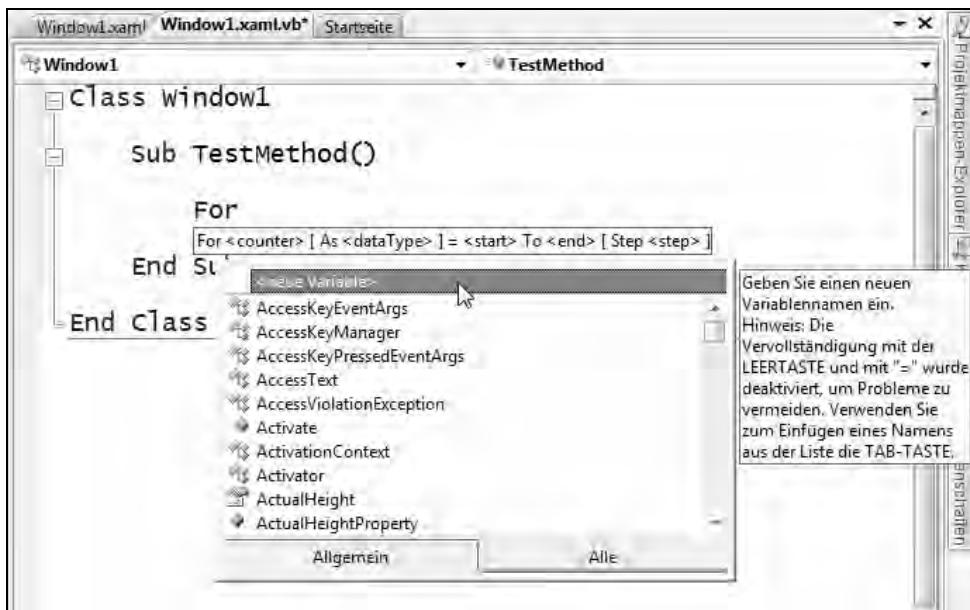


Abbildung 4.25 ... sondern zeigt in ausführlichen Tooltips auch die genaue Syntax und Funktionsweise von Strukturbefehlen an

TIPP In vorherigen Versionen störten die IntelliSense-Listen und Tooltips ab und an, da man nicht durch sie hindurchsehen konnte. Das wird mit Visual Studio 2008 anders: Drücken Sie einfach **Strg**, während die IntelliSense-Elemente angezeigt werden, und sie geben die Sicht auf den darunterliegenden Code frei!

Filtern beim Tippen

In vorherigen Versionen wurde man oftmals durch die langen IntelliSense-Listen erschlagen. In der 2008er Version werden die Inhalte der IntelliSense-Listen beim Tippen gefiltert, wie in den folgenden beiden Abbildungen zu sehen:

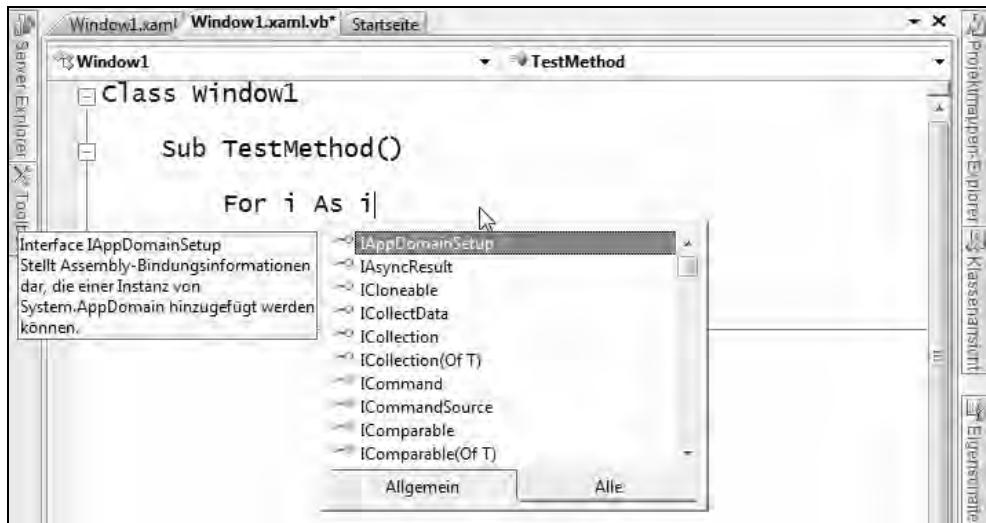


Abbildung 4.26 Die IntelliSense-Auswahlliste wird durch die bislang eingegebenen Zeichen gefiltert. Mit jedem weiteren getippten Buchstaben ...

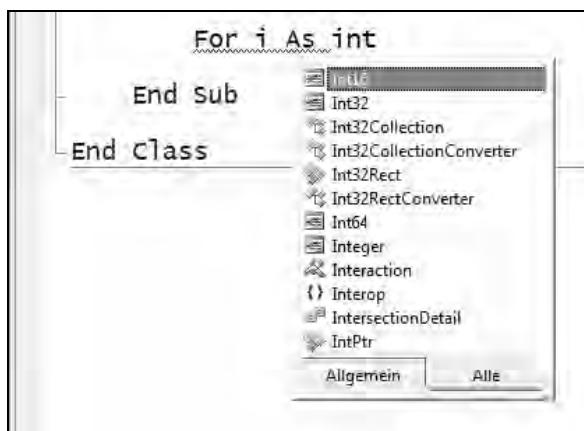


Abbildung 4.27 ...verkleinert sich die Liste auf die Elemente, die dem bislang Getippten entsprechen

Kapitel 5

Einführung in Windows Forms – Designer und Codeeditor am Beispiel

In diesem Kapitel:

Das Fallbeispiel – Der DVD-Hüllen-Generator »Covers«	100
Gestalten von Formularen mit dem Windows Forms-Designer	105
Der Codeeditor	133
Weitere Funktionen des Codeeditors	163

Wenn es darum geht, Windows Forms-Anwendungen, die auch unter dem Modenamen »Smartclients«¹ gehandelt werden, zu entwickeln – und darauf legt dieses Buch erklärerweise seinen Schwerpunkt –, dann sind der Formular-Designer und der Codeeditor die Werkzeuge, die Sie neben den Debugging-Werkzeugen wohl am häufigsten verwenden werden. Darum sei ihnen auch ein eigenes Kapitel gewidmet.

Nun gibt es zwei nahe liegende Ansätze, die wichtigen Werkzeuge einer Entwicklungsumgebung zu erklären: eine theoretische und eine praktische Vorgehensweise. Die theoretische gibt Ihnen einen schnellen Überblick, was für Möglichkeiten es bei dem einen oder anderen Werkzeug gibt, macht Sie aber nicht praxisicher. Die praktische bringt ungleich mehr: Vielleicht unterfordert Sie das Beispiel aus Entwicklersicht ein wenig, das Sie dazu am besten von vorne bis hinten durchexerzieren müssen; aber auf diese Weise lernen Sie am schnellsten die Feinheiten und Möglichkeiten der Werkzeuge kennen, und Sie sind viel besser für die Praxis gerüstet.

Aus diesem Grund habe ich mich dazu entschieden, die Vorstellung der Möglichkeiten von Formular-Designer und Codeeditor an einem konkreten Beispiel vorzuführen. Das Beispiel geht sogar noch ein wenig über die thematischen Kapitelgrenzen hinaus und liefert Ihnen schon jetzt die eine oder andere Info zu völlig anderen Themen, die mir aber schon zu diesem Zeitpunkt im Kontext sinnvoll erscheinen. Natürlich werden fast alle Punkte im Laufe des Buches noch weiter vertieft – und denken Sie daran, dass es in erster Linie darum geht, Ihnen schnellstmöglich zu so vielen Kenntnissen zu verhelfen, dass Sie ohne das Buch zunächst komplett lesen und verstehen zu müssen, schon Ihre ersten kleinen Projekte realisieren können.

Das Fallbeispiel – Der DVD-Hüllen-Generator »Covers«

Es ist schon eindrucksvoll, wie Zeitschriften, Zeitungen, Tankstellen, Buchclubs und andere Institutionen inzwischen um die Gunst ihrer Kunden werben. Auf diese Weise bin ich in den Genuss einer DVD-Sammlung gekommen, die sich sehen lassen kann, und für die ich vergleichsweise wenig Geld investiert habe. So liegen beispielsweise fast allen großen Fernsehzeitschriften immer mal wieder DVDs einfach so als »Goody« bei – klasse Spielfilme, mehrsprachig, untertitelt und in überragender Qualität, quasi zum Nulltarif. Doch was verständlicherweise fehlt, ist jeweils ein ordentliches Cover, mit dessen Hilfe man diese DVD wie ein Buch in den Schrank stellen könnte, und in meterlangen Regalen – so die Idee – DVDs im Bedarfsfall auch wieder finden kann. Das gilt umso mehr für die Cover von Filmen, die man beispielsweise von Pay-TV-Sendern aufgenommen oder sogar selber gedreht hat. Klar – es gibt die unterschiedlichsten Cover-Designer für wenig Geld auf dem Markt; und jeder, der mit seinem Computer einen CD- oder DVD-Brenner erworben hat, wird ohnehin ein solches Programm sein Eigen nennen können, denn die meisten Brennprogramme, die den Brennern beiliegen, verfügen über teilweise recht leistungsfähige Designer.

¹ Die Kapitel ab Teil G beschäftigen sich mit diesem Thema im Detail.

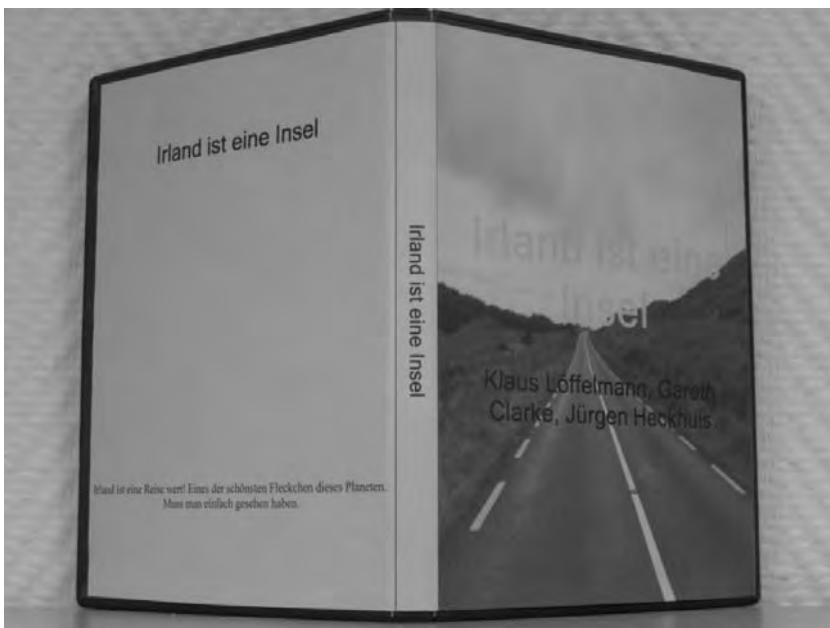


Abbildung 5.1 Ein ausgedrucktes Inlay schneiden Sie einfach entlang der Markierungslinien aus und stecken es in das Plastikcover – hier ein Beispiel

Diese allerdings können in meinen Augen und für meine Belange viel zu viel. Wenn ich Zeit hätte, könnte ich damit bestimmt auch Cover gestalten, die vielleicht sogar fast genau so gut sind wie manches Original. Doch habe ich nie oder selten Zeit, und sollte ich sie doch mal haben, werde ich sicherlich keine Cover in meiner freien Zeit gestalten. Was ich brauche – was viele andere vielleicht auch haben wollen – ist ein Programm, mit dem ich Titel, Schauspieler, Kurzbeschreibung eintippen, vielleicht noch ein Bild auswählen kann, und das mir daraus dann ein Cover zaubert. Das geht schnell und reicht völlig.

Das »Pflichtenheft« von Covers

Ein solches Programm sollte folgendermaßen funktionieren:

- Das Programm sollte einen einfachen, aber dynamisch vergrößerbaren Dialog anbieten, in dem man die Grunddaten des Films in simplen Texteingabefeldern erfassen kann: Filmtitel, Schauspieler, Kurzbeschreibung und ein Bild, das auf der Frontseite des Covers abgedruckt werden soll. Abbildung 5.1 zeigt, wie sowas in etwa im Ergebnis ausschauen kann; die beiden folgenden Abbildungen vermitteln Ihnen einen Eindruck von der Bedienung des Programms.

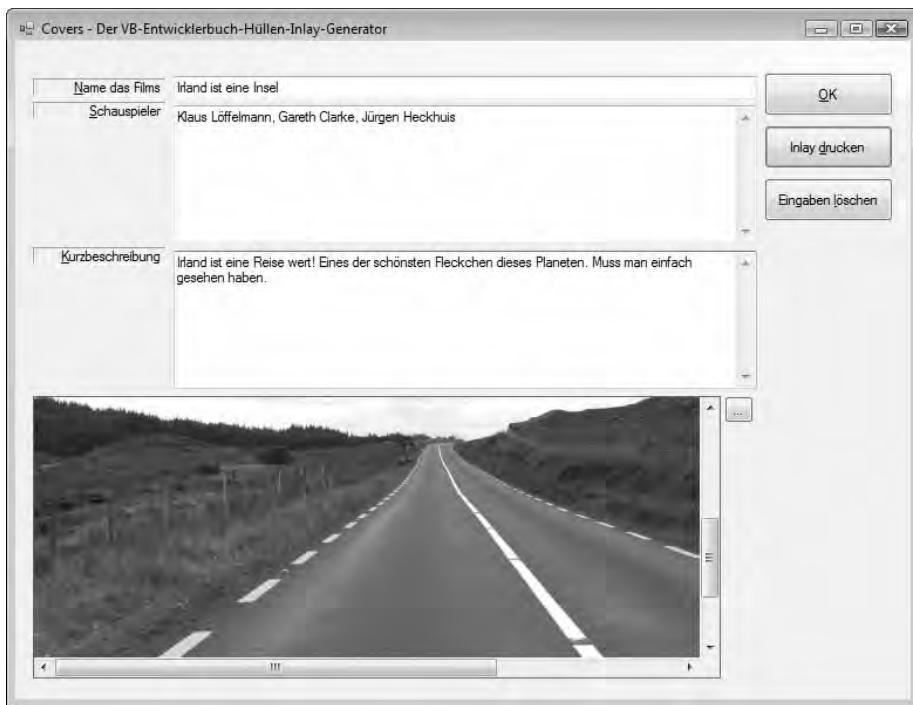


Abbildung 5.2 Dazu soll »Covers« in der Lage sein: Aus einer simplen Erfassung der Filmdaten druckt es buchstäblich auf Knopfdruck ...



Abbildung 5.3 ... das Inlay für das Film-Cover

- Wenn Sie das Programmfenster von »Covers« vergrößern oder verkleinern, sollen sich alle Steuerelemente automatisch an die Größe des Formulars anpassen. Die Steuerelemente wachsen nach unten im Verhältnis zur Vergrößerung des Formulars mit; beim Verkleinern werden auch die Steuerelemente wieder entsprechend niedriger. Vergrößern Sie das Formular nach rechts, wandern die Schaltflächen auf der rechten Seite quasi mit dem rechten Fensterrand mit. Die anderen Steuerelemente, die für die Dateneingaben vorhanden sind, vergrößern sich dementsprechend.
- Sie können ein Bild in das Formular laden, das dann später automatisch auf die kompletten Ausmaße der Covervorderseite skaliert wird. Sollte das Bild nicht in das Bildelement auf dem Formular passen, erscheinen automatisch Rollbalken, mit denen Sie den Ausschnitt des Bildes einstellen können.
- Mit der Schaltfläche *Eingaben löschen* setzen Sie alle Eingaben wieder zurück. Diese Schaltfläche ist praktisch, wenn Sie direkt hintereinander mehrere Cover drucken wollen.
- Das Programm soll sich alle Eingaben »merken«. Wenn Sie das Programm verlassen und neu starten, sollen sämtliche zuletzt getätigte Eingaben oder Bilddefinitionen wieder so vorzufinden sein, wie sie vor dem letzten Beenden des Programms vorhanden waren.
- Wenn Sie Daten in das Formular eingetragen haben, gelangen Sie durch *Inlay drucken* zur Druckvorschau, die Ihnen das Inlay auf dem Bildschirm so darstellt, wie es später auch gedruckt werden soll. Auch dieses Vorschaufenster soll sich dynamisch vergrößern lassen.
- Vor dem Drucken sollen Sie die Möglichkeit haben, einen Drucker, den Sie verwenden wollen, auszuwählen und einzustellen. Der Drucker soll das Inlay aber immer unabhängig von den Einstellungen im Querformat drucken.

Die folgende Schritt-für-Schritt-Anleitung, die sich über mehrere Abschnitte erstreckt, zeigt, wie Sie diese Software von A-Z mit Visual Studio 2008 und Visual Basic 2008 erstellen. Sie werden erstaunt sein, wie wenig Code dazu nötig ist – das eigentliche Drucken macht dabei noch den größten Teil aus.

BEGLEITDATEIEN

Natürlich ist das vollständige Projekt in den Begleitdateien zum Buch enthalten – Sie sollten aber in diesem Fall besser vollständig auf sie verzichten und kleinere Codepassagen wirklich abtippen, um ein Gefühl und besseres Verständnis für den Codeeditor zu erlangen. Die vergleichsweise umfangreiche Druckroutine wäre allerdings doch ein wenig zu viel des Guten, und Sie finden sie deswegen als reine Textdatei im Verzeichnis

...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 05\\Covers\\Druckroutine für Covers.txt

Erstellen eines neuen Projektes

Und los geht's. Wir beginnen so einfach wie möglich: mit dem Erstellen eines neuen Projektes. Dazu starten Sie Visual Studio, falls Sie es noch nicht getan haben.

HINWEIS Um den Speicherort des Projektes, wie im Folgenden beschrieben, schon beim Anlegen festlegen zu können, rufen Sie den Options-Dialog über den Menübefehl *Extras/Optionen* auf. Setzen Sie das Häkchen *Neues Objekt beim Erstellen speichern* im Bereich *Projekte und Projektmappen/Allgemein*.

1. Um ein neues Projekt anzulegen, klicken Sie auf der Startseite im linken oberen Bereich auf *Projekt* hinter *Erstellen*: Abbildung 5.4 hilft Ihnen bei der Orientierung.



Abbildung 5.4 Wenn Sie ein neues Projekt erstellen wollen, rufen Sie die entsprechende Funktion direkt aus der Startseite auf

2. Im anschließend erscheinenden Dialog wählen Sie in der Baumstruktur unter *Projekttypen* den Zweig *Visual Basic/Windows*. Unter *Vorlagen* wählen Sie *Windows-Anwendung*. Geben Sie unter *Namen* den Namen Ihres neuen Projektes ein – für unser Beispiel wählen Sie *Covers*. Unter *Speicherort* bestimmen Sie das Verzeichnis, in dem alle Projektdateien gespeichert werden sollen. *Durchsuchen...* bringt Sie dafür zu einem Dialog, der Ihnen bei der Bestimmung des Verzeichnisses helfen kann. Stellen Sie sicher, dass das Häkchen neben *Projektmappenverzeichnis erstellen* nicht zu sehen ist. Abbildung 5.5 hilft Ihnen bei der Orientierung. Klicken Sie anschließend auf *OK*.

TIPP Wenn Sie mehrere Projekte in einer Projektmappe erstellen möchten – Sie möchten beispielsweise eine Anwendung in mehrere Schichten (Datenschicht, Geschäftslogik, Benutzeroberfläche, Steuerelemente, etc.) aufteilen und diese Schichten in verschiedenen Klassenbibliotheken verteilen –, können Sie Ihr Windows-Anwendungsprojekt schon jetzt in einer Projektmappe einordnen. Die Projektmappe bekommt dann ein eigenes Verzeichnis; ihr Windows-Projekt liegt in einem Verzeichnis unterhalb des Projektmappen-Verzeichnisses. In diesem Fall setzen Sie das Häkchen in *Projektmappenverzeichnis erstellen* und geben Sie im Eingabefeld *Projektmapppenname* den Namen der Projektmappe ein.

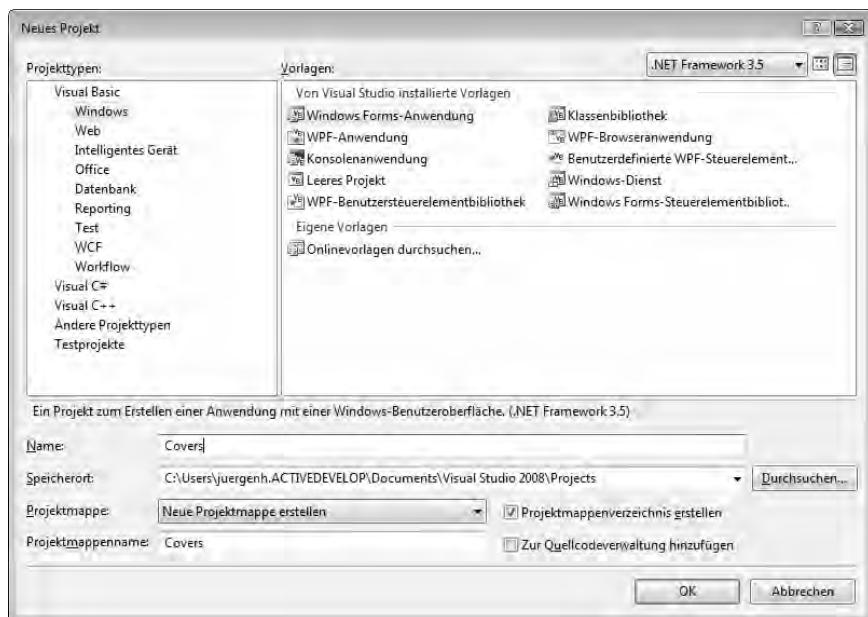


Abbildung 5.5: Mit diesem Dialog richten Sie eine neue Windows-Anwendung ein

Gestalten von Formularen mit dem Windows Forms-Designer

Wir beginnen unser Fallbeispiel aus den letzten Abschnitten mit dem Entwurf des Formulars. Da wir die Toolbox zu diesem Zweck häufig gebrauchen werden, macht es Sinn, diese ständig im Vordergrund zu haben. An der linken Seite der Visual Studio-IDE finden Sie eine »eingefahrene« Registergruppe, bestehend aus dem Server-Explorer und der Toolbox.

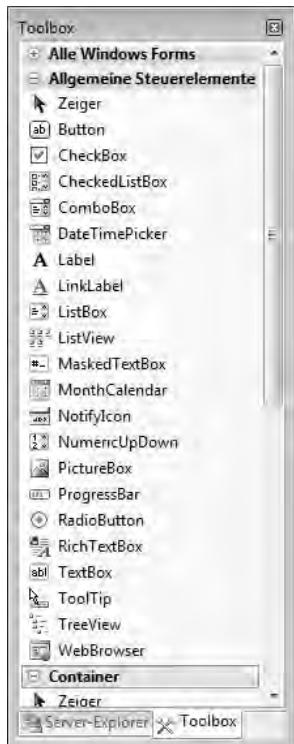


Abbildung 5.6 Die Toolbox zum Einfügen von Steuerelementen in Formularen in der Visual Basic 2008 Entwicklungsumgebung

Fahren Sie mit dem Mauszeiger auf das Toolbox-Symbol, worauf sich die Toolbox in den Vordergrund schiebt, und klicken Sie auf das Pin-Symbol im Fenstertitel (links im Bild zu sehen). Dadurch bleibt die Toolbox, wo sie ist, und fährt nicht wieder automatisch in den Hintergrund, wenn der Mauszeiger sie verlässt.

Damit viel Platz für Experimente bleibt, vergrößern Sie das vorhandene Formular, sodass es einen Großteil der Arbeitsfläche einnimmt.

Positionieren von Steuerelementen

Wie in Abbildung 5.2 zu sehen, verfügt das Formular über drei Schaltflächen, mit denen verschiedene Funktionen ausgelöst werden. Diese Schaltflächen werden wir als erstes auf dem Formular positionieren.

1. Dazu klicken Sie in der Toolbox auf das *Button*-Symbol. Fahren Sie mit dem Mauszeiger in das Formular und ziehen Sie eine Schaltfläche ungefähr in der Größe auf, wie sie den Schaltflächen in Abbildung 5.2 entspricht.
2. Ziehen Sie zwei weitere Schaltflächen irgendwo im Formular auf – Größe und Position der Schaltflächen sind dabei zunächst völlig egal. Wir werden gleich die entsprechenden Funktionen kennen lernen, mit denen wir die Schaltflächen angleichen können.

TIPP Wenn Sie ein Steuerelement, wie z. B. die Schaltfläche in der Toolbox, auswählen und dann auf die Formularoberfläche an eine gewünschte Position einfach klicken, wird das Steuerelement in der Windows-Standardgröße eingefügt.

3. Per Drag & Drop verschieben Sie nun die Schaltfläche mit der Aufschrift *Button1* in die rechte, obere Ecke des Formulars. Wenn Sie in der oberen, rechten Ecke ankommen, werden Sie merken, dass die Schaltfläche an einer bestimmten Ecke »anschnappt«, und Sie können zwei Hilfslinien erkennen (siehe Abbildung 5.7), die die Schaltfläche auf Abstand halten.

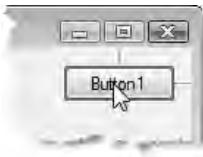


Abbildung 5.7 Ausrichtungslinien helfen Ihnen bei der Positionierung von Steuerelementen am Formularrand oder untereinander

Ausrichtungslinien (Guidelines) und die Margin/Padding-Eigenschaften von Steuerelementen

Jedes Steuerelement, das Sie im Formular positionieren können, basiert auf einem Grundsteuerelement namens *Control*. Dieses Steuerelement stellt nicht nur einen Grundpool an Funktionalität zur Verfügung, auf denen andere Steuerelemente beispielsweise für das Anzeigen ihrer eigentlichen Inhalte (eine Schaltfläche muss andere Inhalte darstellen als ein CheckBox-Steuerelement) aufbauen können – es stellt auch eine grund-sätzliche Designerfunktionalität zur Verfügung (der folgende graue Kasten liefert genauere Infos zu diesem Thema).

Das ist der Grund, wieso jedes Steuerelement, das Sie im Formular platzieren können, automatisch und ausnahmslos über eine Margin-Eigenschaft verfügt. Diese Eigenschaft bestimmt, wie groß der Abstand zwischen einem Steuerelement und einem anderen Steuerelement sein wird, wenn Sie bei deren Platzierung die durch die Ausrichtungslinien vorgegebenen Abstände akzeptieren (oder mit anderen Worten: ab welchem Abstand die Ausrichtungslinien »anspringen« und das Steuerelement bei diesem Abstand »einschnappt«).

Wenn Steuerelemente als »Träger« anderer Steuerelemente fungieren – ein solches Steuerelement nennt man übrigens Container-Steuerelement bzw. *ContainerControl* – kommt eine weitere Eigenschaft ins Spiel: die Padding-Eigenschaft. Diese Eigenschaft bestimmt, wie viel Abstand zusätzlich zum äußeren Rand einge-halten werden soll, wenn ein beinhaltendes Steuerelement am Rand eines beinhaltenden Steuerelements angeordnet wird.

HINWEIS Formulare selbst machen dabei offensichtlich noch eine Ausnahme: Beim Anordnen von Steuerelementen am Rand von Formularen werden zusätzlich ein paar Pixel Abstand eingehalten, wenn die Ausrichtungslinien dargestellt werden, möglicherweise um Microsofts Style-Guide für das Gestalten von Formularen zu entsprechen.²

TIPP Das Einschnappen geschieht übrigens nicht nur an Stellen, die den durch die Margin- bzw. die Padding-Eigenschaften festgelegten Abständen entsprechen. Visual Studio stellt Ihnen das Einschnappen auch zur Verfügung, um Steuerelemente an Basislinien ihrer Textzeilen exakt auszurichten, wenn diese nebeneinander stehen. Das ist häufig der Fall, wenn Sie ein Label-Steuerelement dafür verwenden, ein Textbox-Steuerelement im Formular zu beschriften, wie in Abbildung 5.8 zu sehen.

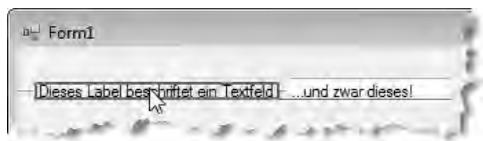


Abbildung 5.8: Gerade beim Beschriften von *Textbox*-Steuerelementen mit *Label*-Steuerelementen helfen Ihnen Ausrichtungslinien, da sich die Steuerelemente auch an den Textbasislinien einschnappen

TIPP Entwickler, die lieber mit der Rasterfunktionalität arbeiten, die sie aus Visual Studio .NET 2003 bzw. Visual Basic 6.0 kennen, müssen darauf auch in Visual Studio 2008 nicht verzichten. Im *Optionen*-Dialog von Visual Studio, den Sie über das *Extras*-Menü erreichen, können Sie auf der Registerkarte *Windows Forms-Designer* die *LayoutMode*-Eigenschaft von *SnapLines* auf *SnapToGrid* ändern. Möchten Sie zwar im neuen Layout-Modus von Visual Studio 2008 arbeiten, aber ohne die Einschnapp-Funktionalität auskommen, setzen Sie im gleichen Dialog die *SnapToGrid*-Eigenschaft auf *False*.

Wem »gehört« eigentlich der Formular-Designer?

Rein rechtlich gesehen, natürlich Ihnen – schließlich haben Sie Visual Studio und damit auch den Formular-Designer erworben. Aber das meine ich gar nicht. Die eigentliche Frage lautet: In welchem Programmnteil des gesamten Visual Studio/Framework-Komplex ist die Designer-Funktionalität eigentlich untergebracht? Die Antwort mag Sie überraschen, denn: Es ist nicht so, wie man vielleicht vermuten würde, dass Visual Studio komplett selbst für die Bearbeitung der Steuerelemente im Designer zuständig ist. Visual Studio fungiert im Grunde nur als Ereignisweiterreicher an die betroffene Komponente. Ein Steuerelement besteht nämlich grundsätzlich aus zwei Teilen: aus dem Steuerelement, das die eigentliche Funktionalität zur Verfügung stellt, die Sie zur Laufzeit Ihres Programms benötigen und aus einer weiteren Komponente, dem Designer. Ja genau. Jede Komponente verfügt streng genommen über ihren eigenen Designer. Aber natürlich haben nicht hunderte von Entwicklern hunderte verschiedene Designer programmiert. Genauso, wie Control alle Grundfunktionalitäten für ein neues, eigentliches Steuerelement zur Verfügung stellt, das diese lediglich weiter ausbaut, gibt es auch eine weitere Komponente namens ControlDesigner, die alle Design-Grundfunktionalitäten beinhaltet. Visual Studio spielt dabei nur die Rolle eines Gastgebers und stellt das Umfeld, damit ein solcher Designer Platz zum Austoben hat. Im Fachter-

² Die Online-Hilfe meint zu diesem Thema sinngemäß: »Die Ausrichtungslinien entsprechen der Addition von Padding- und Margin-Eigenschaften«; für Formulare scheint das nicht zu gelten. Eine entsprechende Diskussion finden Sie im Product Feedback Center unter dem IntelliLink **A0501**.

minus lautet das: »Visual Studio ist der *Designer Host*«. Wird ein neues Steuerelement geschaffen, und es bringt keinen eigenen Designer (komplett neu oder auf ControlDesigner basierend spielt dabei keine Rolle), verwendet Visual Studio eben einfach ControlDesigner selbst für dieses Steuerelement – und dessen Basisfunktionen erlauben zumindest das Verschieben, Kopieren, »Einschnappen« oder andere Funktionen, die Sie beim Aufbau eines Formulars benötigen. Für diejenigen unter Ihnen, die sich ein wenig mit den .NET-Assemblies auskennen und die es interessiert: Die Designer aller Steuerelemente von .NET befinden sich in der Assembly *System.Design.dll*.

Angleichen von Größe und Position von Steuerelementen

Nun befinden sich alle drei Steuerelemente wild platziert im Formular – Ziel ist es aber, dass sie alle drei nicht nur wie an einer Schnur ausgerichtet, bündig untereinander stehen, sondern dass sie auch alle drei die gleiche Größe haben. Es gäbe die Möglichkeit, diese Änderungen manuell vorzunehmen, indem Sie die Steuerelemente solange »zurechtzuppeln« und verschieben, bis sie passen. Dank der Ausrichtungslinienfunktionalität ist das kein großer, aber immerhin ein Akt.

Es geht auch einfacher. Sie können mehrere Steuerelemente markieren und anschließend Operationen die Größe und Position betreffend durchführen, bei denen alle Steuerelemente sich an dem als erstes markierten Steuerelement orientieren. Wenn Sie die Steuerelemente markiert haben (wie gesagt: das zuerst markierte ist immer das Referenzsteuerelement), bietet Ihnen Visual Studio bestimmte Funktionen an, die Sie im Abschnitt »Funktionen zum Layouten von Steuerelementen im Designer« ab Seite 131 beschrieben finden.

Selektieren mehrerer Steuerelemente und Bestimmen des Referenzsteuerelements

Um mehrere Steuerelemente im Formular gleichzeitig zu selektieren, halten Sie entweder die -Taste gedrückt und klicken anschließend alle betroffenen Steuerelemente nacheinander an, oder Sie ziehen mit der Maus einen Rahmen um die Steuerelemente, die selektiert werden sollen. Das Referenzsteuerelement für bestimmte Funktionen, die Sie in Tabelle 5.4 ab Seite 131 beschrieben finden, ist das, welches als erstes selektiert wurde, wenn Sie die Steuerelemente mit gedrückter -Taste selektieren. Möchten Sie das Referenzsteuerelement nach der Selektion der Steuerelemente ändern, klicken Sie es einfach an (dabei halten Sie die -Taste *nicht* mehr gedrückt). Anders als bei vielen anderen Selektionsverfahren unter Windows, wird dabei die Selektion nicht aufgehoben und das neu angeklickte Element selektiert, sondern Sie ändern wirklich nur das Referenzsteuerelement. Möchten Sie die Selektion komplett aufheben, klicken Sie einfach in einen freien Bereich innerhalb des Formulars oder auf ein zuvor nicht selektiertes Steuerelement.

TIPP Das Referenzsteuerelement erkennen Sie übrigens daran, dass es mit weißen Anfasspunkten versehen ist, während alle anderen markierten Steuerelemente über schwarze Anfasspunkte verfügen.

1. Für unser Beispiel selektieren Sie auf diese Weise zunächst alle Schaltflächen im Formular. Wichtig dabei ist, dass die Schaltfläche in der linken, oberen Ecke zum Referenzsteuerelement wird.

HINWEIS Für den folgenden Schritt müssen Sie unter Umständen die benötigte Symbolleiste einschalten. Klicken Sie dazu mit der rechten Maustaste auf einen freien Bereich neben einer schon eingeblendeten Symbolleiste, und wählen Sie aus dem Kontextmenü, das jetzt erscheint, den Eintrag *Layout*.

2. Wählen Sie aus der Symbolleiste die Funktion *Größe angleichen* – die Tabelle im Abschnitt »Funktionen zum Layouten von Steuerelementen im Designer« ab Seite 131 hilft Ihnen, das richtige Symbol dafür zu finden.

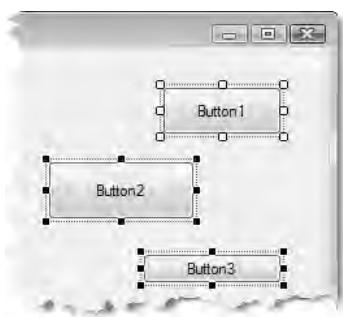


Abbildung 5.9 Hier sehen Sie, dass sich alle Schaltflächen wild platziert im Formular befinden – mit Ausnahme der ersten, die ihre endgültige Position und Größe bereits hat ...

3. Klicken Sie anschließend auf das Symbol *Links ausrichten*, um die Schaltflächen linksbündig untereinander auszurichten. Damit stehen anschließend alle drei Schaltflächen genau ausgerichtet untereinander.
4. Die Abstände zwischen den Schaltflächen passen Sie dann entweder durch Verschieben der Schaltflächen mit der Maus per Drag & Drop und der Unterstützung durch die Ausrichtungslinien an, oder Sie verwenden die Funktion *Vertikalen Abstand angleichen*, und so oft die Funktionen *Vertikalen Abstand vergrößern* bzw. *Vertikalen Abstand verkleinern*, bis Sie mit dem Ergebnis zufrieden sind.

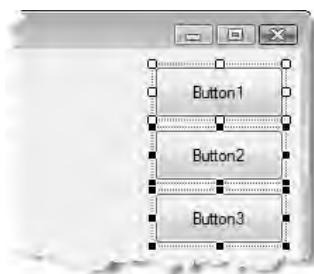


Abbildung 5.10 ... und hier nach Anwenden der Funktionen Größe angleichen, Links ausrichten, Vertikalen Abstand angleichen sowie Vertikalen Abstand verkleinern

Im günstigsten Fall mussten Sie die Schaltflächen nach dem Aufziehen im Formular nicht einmal mehr »anpacken«. Innerhalb von Sekunden konnten Sie sie mithilfe der Funktionen der Layout-Symbolleiste so anpassen, dass sie mit sauberem Layout im Formular standen.

Häufige Arbeiten an Steuerelementen mit Smarttags erledigen

Es gibt komplexe Steuerelemente, wie beispielsweise das `TableLayoutPanel` (das wir für den nächsten Schritt unseres Beispiels benötigen), das für das bequeme Einstellen bestimmter Verhaltensweisen so genannte Smarttags anbietet. Smarttags, wie in Abbildung 5.11 zu sehen, führen zu einer Liste mit Kontextaufgaben, die an das jeweilige Steuerelement gekoppelt ist, und das Ihnen eine Art Abkürzung zu den Eigenschaften bietet und eine gleichzeitig komfortablere Einstellung der Eigenschaften desselben ermöglicht.

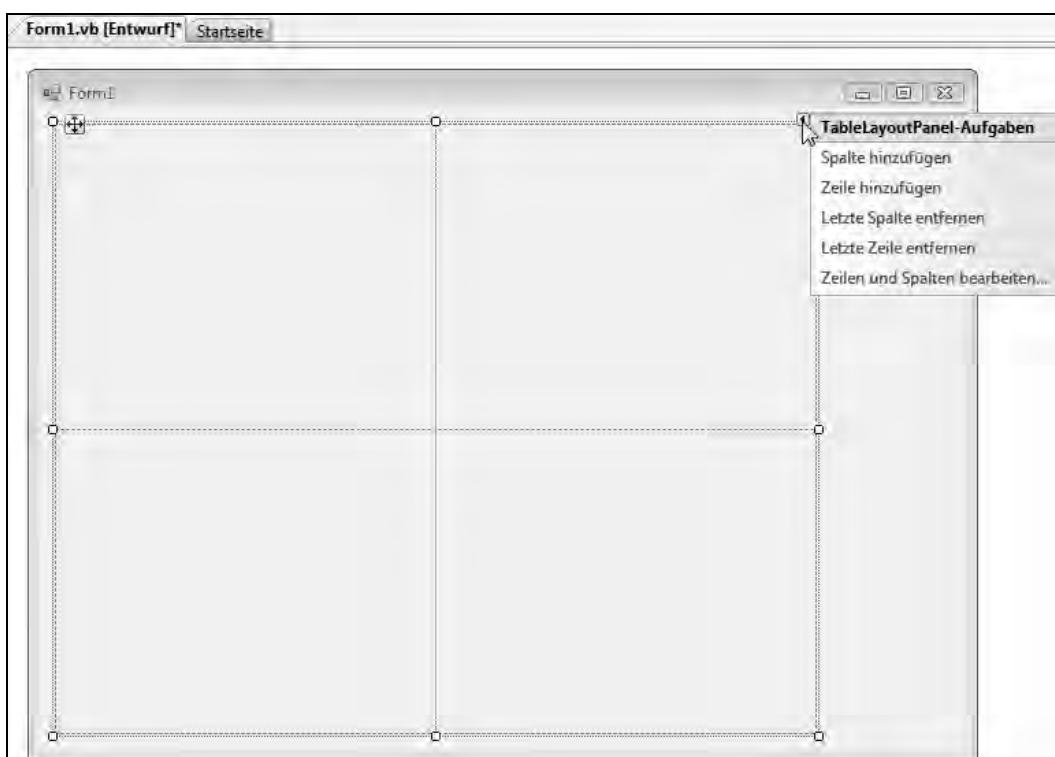


Abbildung 5.11 Ein Mausklick auf den Smarttag eines Steuerelements öffnet das Aufgabenfenster, das Funktionen zum Erledigen der häufigsten Aufgaben anbietet

1. Für unser Beispiel wählen Sie aus der Toolbox per Mausklick das TableLayoutPanel aus. Ziehen Sie dieses Steuerelement im Formular in etwa in der Größe auf, dass es Abbildung 5.11 entspricht.
2. Sofort öffnet sich die hinter dem Smarttag stehende Liste mit Kontextaufgaben. Sie können diese Kontextaufgabenliste jederzeit mit einem Mausklick auf den Smarttag auf- und zuklappen.

HINWEIS Nicht jedes Steuerelement verfügt über einen Smarttag mit einer dahinter stehenden Liste mit Kontextaufgaben, aber wenn ein Steuerelement darüber verfügt, dann steht Ihnen diese Liste jederzeit zur Verfügung.

Dynamische Anordnung von Steuerelementen zur Laufzeit

Denken Sie mal einige Jahre zurück und dabei an den Aufwand, den es noch in Visual Basic 6.0 machte, eine Benutzeroberfläche wie den Windows-Explorer zur implementieren. Sie mussten alleine mehrere Mannstunden dafür opfern, den Code dafür zu entwickeln, die Steuerelemente dynamisch neu anzurichten, falls der Anwender des Programms zur Laufzeit das Fenster vergrößerte oder verkleinerte.

Seit Visual Basic .NET ist das sehr viel einfacher geworden. Geschickt umgesetzt müssen Sie nicht eine einzige Zeile Code schreiben, um Steuerelemente in Abhängigkeit einer Formulargrößenänderung neu zu positionieren und auszurichten. Und mit Visual Basic 2008 ist dies noch ungleich bequemer geworden, denn es stellt neue Container-Steuerelemente zur Verfügung, die ausschließlich diesem Zweck dienen. Folgende Features stehen Ihnen in Visual Basic 2008 zur Verfügung, mit denen Sie diese Problematik – wie gesagt ohne Programmierung – in den Griff bekommen:

- **Anchor-Eigenschaft:** Erlaubt das Verankern jeder Seite eines Steuerelements auf dem Formular. Vergrößern oder verkleinern Sie das Formular, dann »wandern« die Seiten des Steuerelements im gleichen Verhältnis hinter den Seiten des Formulars her, an denen es verankert ist.
- **Dock-Eigenschaft:** Erlaubt das Andocken eines Steuerelements an einen Formularrand oder an ein bereits gedocktes Steuerelement. Dabei wird dafür gesorgt, dass das Steuerelement an den gedockten Seiten immer zum Formularrand oder zum ersten schon gedockten Steuerelement aufschließt.
- **SplitContainer-Steuerelement:** Stellt einen Doppelcontainer für weitere Steuerelemente zur Verfügung. Dessen Besonderheit ist, dass sich die Größe eines Containerteils auf Kosten des anderen Containerteils vergrößern lässt. Denken Sie als Beispiel an den Windows-Explorer. In der Mitte können Sie einen vertikalen Trennbalken mit der Maus packen und nach links oder rechts verschieben – dementsprechend vergrößert bzw. verkleinert sich die linke TreeView mit den Verzeichnissen und anderen Hauptelementen (wie Systemsteuerung, Mobile Geräte, etc.) während sich die Elementeliste im rechten Containerteil, die durch eine ListView realisiert wird, verkleinert bzw. vergrößert.
- **TableLayoutPanel-Steuerelement:** Stellt einen Container für weitere Steuerelemente zur Verfügung. Die Besonderheit ist, dass dieser Container mehrere Steuerelemente in einer Tabelle anordnet, deren Zeilen und Spalten sich in einstellbaren Verhältnissen vergrößern, wenn das TableLayoutPanel-Steuerelement selbst sich ebenfalls vergrößert oder verkleinert. Damit wird es möglich, dass nicht nur bestimmte Steuerelemente sich vergrößern, wenn sich das Formular vergrößert, sondern dass Steuerelemente in bestimmten Verhältnissen anwachsen oder sich verkleinern, im Falle einer Vergrößerung oder Verkleinerung des Formulars.
- **FlowLayoutPanel-Steuerelement:** Ist am besten mit einem Texteditor mit aktiviertem Wortumbruch zu erklären. Wenn ein Wort nicht mehr in eine Zeile passt, wird es automatisch in die nächste Zeile umbrochen. Beim FlowLayoutPanel ist dies genauso, jedoch nicht mit Worten, sondern mit Steuerelementen. Sie ordnen Steuerelemente nicht an festen Positionen, sondern nebeneinander bzw. untereinander an. Wenn sich das Formular vergrößert oder verkleinert, versucht das FlowLayoutPanel möglichst viele Steuerelemente beispielsweise in eine Zeile zu bekommen. Passen sie nicht mehr in eine Zeile, da es zu »eng« wird im Formular, springen sie wie beim Wortumbruch des Editors automatisch in die nächste Zeile.

Die folgenden Abschnitte demonstrieren die wichtigsten dieser Features an unserem Fallbeispiel.

Verankern von Steuerelementen mit der Anchor-Eigenschaft

Die Anchor-Eigenschaft, die jedem Steuerelement anheim ist, löst das wichtigste, da häufigste, aller Probleme auf schnelle und elegante Weise. Sie dient dazu, die Seiten von Steuerelementen mit den Seiten eines Formulars oder des sie beinhaltenden Container-Steuerelements zu verankern. Jede verankerte Seite des Steuerelements

behält den Abstand zum Rand seines Containers, wenn dieser sich vergrößert oder verkleinert, was zur Folge hat, dass das Steuerelement (wenn es an zwei gegenüberliegenden Seiten verankert ist) sich entsprechend vergrößert oder seine Position verändert (wenn es an nur jeweils einer Seite verankert ist).

Für unser Beispiel werden wir im Folgenden die drei Schaltflächen so neu verankern, dass sie hinter dem rechten Formularrand mitwandern, wenn der Anwender das Formular in X-Richtung vergrößert oder verkleinert. Das TableLayoutPanel1, das später die Eingabefelder, Beschriftungen und das Bild enthält, wird an allen vier Seiten verankert, damit es in allen Richtungen dynamisch mit dem Formular mitwächst und seinerseits wieder dafür sorgt, dass sich später alle in ihm enthaltenen Steuerelemente proportional anpassen.

1. Markieren Sie alle drei Schaltflächen.
2. Suchen Sie im Eigenschaftenfenster nach der Anchor-Eigenschaft. Klappen Sie die Aufklappliste auf, und klicken Sie mit der Maus auf die kleinen grauen Zwischenräume zwischen den weißen Flächen, um die Verankerungspositionen festzulegen. Orientieren Sie sich dabei am besten an Abbildung 5.12.

TIPP Sie müssen wie hier im Beispiel nicht Eigenschaften verschiedener Steuerelemente, die Sie mit gleichen Werten belegen möchten, nacheinander setzen. Selektieren Sie stattdessen die Steuerelemente, für die der gleiche Wert einer Eigenschaft gelten soll, und weisen Sie mit dem Eigenschaftenfenster diese Eigenschaft allen selektierten Steuerelementen »in einem Rutsch« zu. Visual Studio findet bei mehreren selektierten Steuerelementen übrigens automatisch den größten gemeinsamen Nenner in Sachen vorhandene Eigenschaften, zeigt also bei mehreren selektierten Steuerelementen nur die Eigenschaften im Eigenschaftenfenster an, über die alle selektierten Steuerelemente verfügen.

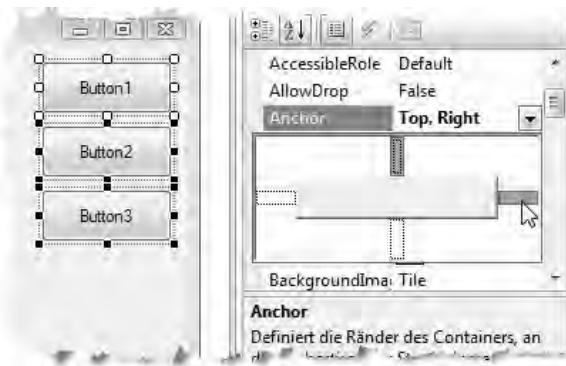


Abbildung 5.12: Setzen Sie die *Anchor*-Eigenschaft von ursprünglich *Top, Left* auf *Top, Right* bewirkt das, dass die Steuerelemente immer den gleichen Abstand zum oberen und rechten Formularrand behalten und beim Vergrößern des Formulars nach rechts quasi hinter dem Rand herlaufen

3. Vergrößern Sie als nächstes das TableLayoutPanel1 so, dass es mit den drei betroffenen Seiten möglichst nah an den Formularrändern liegt.
4. Setzen Sie die Anchor-Eigenschaft des TableLayoutPanel1 so, dass es an alle vier Seiten gebunden ist.

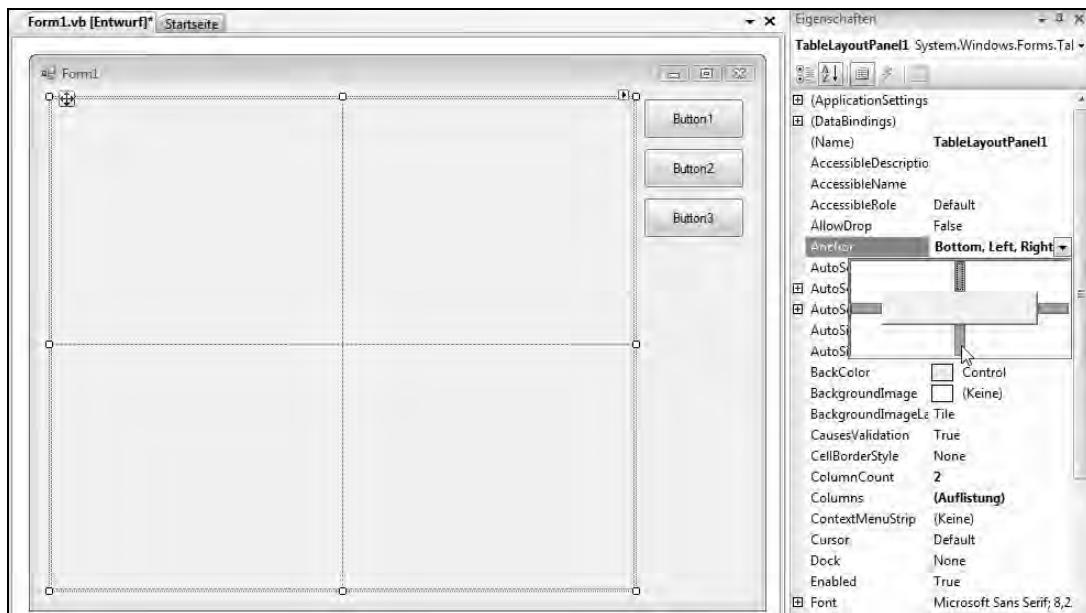


Abbildung 5.13 Setzen Sie die Anchor-Eigenschaft eines Steuerelements für alle vier Seiten, vergrößert sich das Steuerelement in alle Richtungen proportional mit dem Formular

Nachdem Sie alle Einstellungen vorgenommen haben, können Sie schon jetzt im Entwurfsmodus das Formular vergrößern und verkleinern, und Sie werden feststellen, dass sich bereits jetzt alle Steuerelemente an die jeweils neuen Formularausmaße anpassen.

Proportionales Anpassen von Steuerelementen an Formular-Größenänderungen mit dem TableLayoutPanel

Wenn Sie das TableLayoutPanel in Abbildung 5.13 betrachten, sehen Sie, dass es aus insgesamt vier Zellen besteht. Diese Zellen dienen, wie beispielsweise das Panel, als Container für jeweils ein weiteres Steuerelement. Das Besondere: Wenn sich das TableLayoutPanel an sich vergrößert bzw. verkleinert, vergrößern bzw. verkleinern sich die in ihm enthaltenen Zellen im Verhältnis. Bei geschickter Anordnung von Steuerelementen in den Zellen reichen Sie die verhältnismäßige Vergrößerung oder Verkleinerung an diese weiter. Ihre Formulare wachsen damit dynamisch, wenn mehr Platz auf dem Bildschirm zur Verfügung steht, und nutzen diesen damit wesentlich besser aus.

TIPP Übrigens: Wenn Sie verstanden haben, wie das TableLayoutPanel-funktioniert, wird Ihnen das Design von Anwendungen unter der Windows Presentation Foundation viel leichter fallen. Formulare bauen Sie dort standardmäßig auf Basis des Grid-Containers auf, und der verhält sich prinzipiell so, wie Sie es hier beim TableLayoutPanel-Steuerelement kennen lernen.

Einstellen der vorhandenen Spalten und Zeilen des TableLayoutPanel

Standardmäßig, also wenn Sie ein TableLayoutPanel das erste Mal im Formular aufziehen, besteht es aus vier Zellen: nämlich aus zwei Spalten und zwei Zeilen. Um die Anzahl an Spalten oder Zeilen zu erhöhen,

verwenden Sie entweder – wie bei allen Steuerelementen – das Eigenschaftenfenster, oder Sie verwenden die Kontextaufgabenliste, die Sie über den Smarttag des Steuerelements erreichen.

1. Für unser Beispiel klicken Sie auf den Smarttag des sich bereits im Formular befindlichen TableLayoutPanel und klicken anschließend auf *Zeilen und Spalten bearbeiten*. Orientieren Sie sich dafür an Abbildung 5.14.
2. Wählen Sie aus der Aufklappliste *Anzeigen* das Element *Zeilen*.

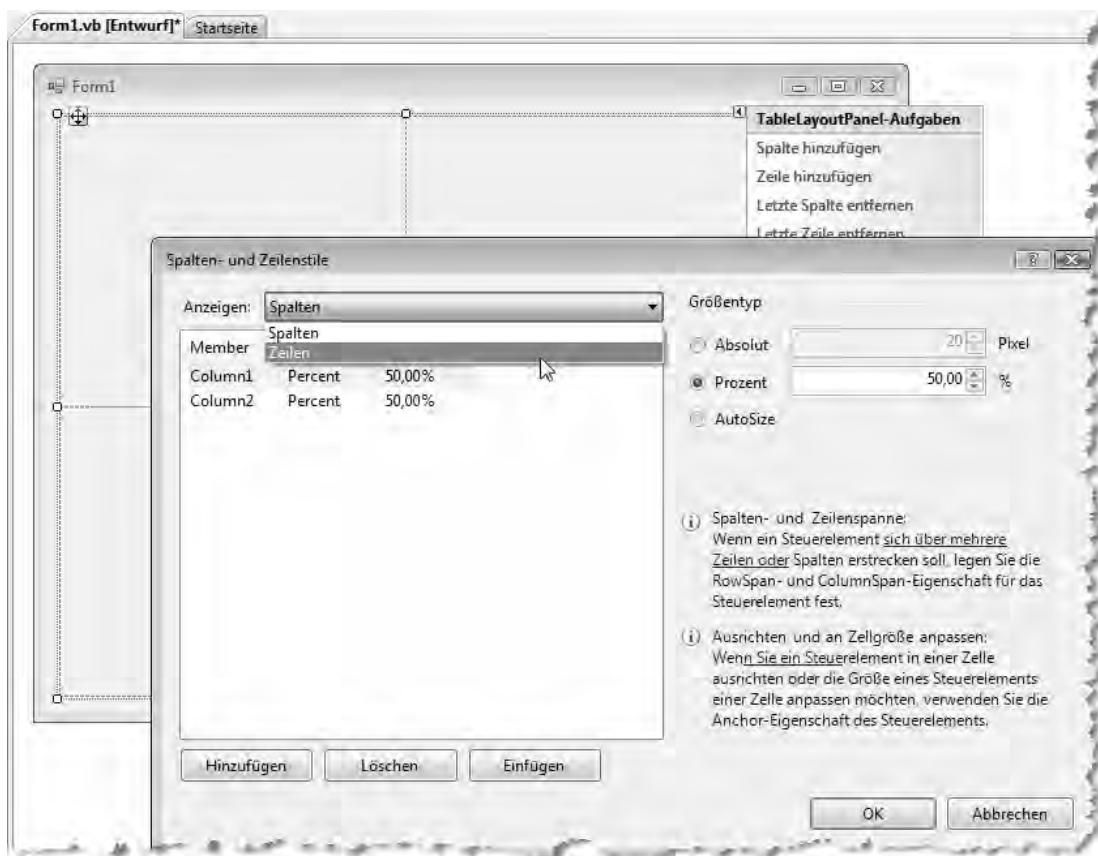


Abbildung 5.14 Wählen Sie aus der Kontextaufgabenliste des *TableLayoutPanel*, die Sie durch Klick auf sein Smarttag öffnen, *Zeilen und Spalten* bearbeiten und unter *Anzeigen* das Element *Zeilen*, um die Anzahl und Eigenschaften der Zeilen einzustellen

3. Klicken Sie zweimal auf *Hinzufügen*, um dem TableLayoutPanel zwei zusätzliche Zeilen hinzuzufügen.

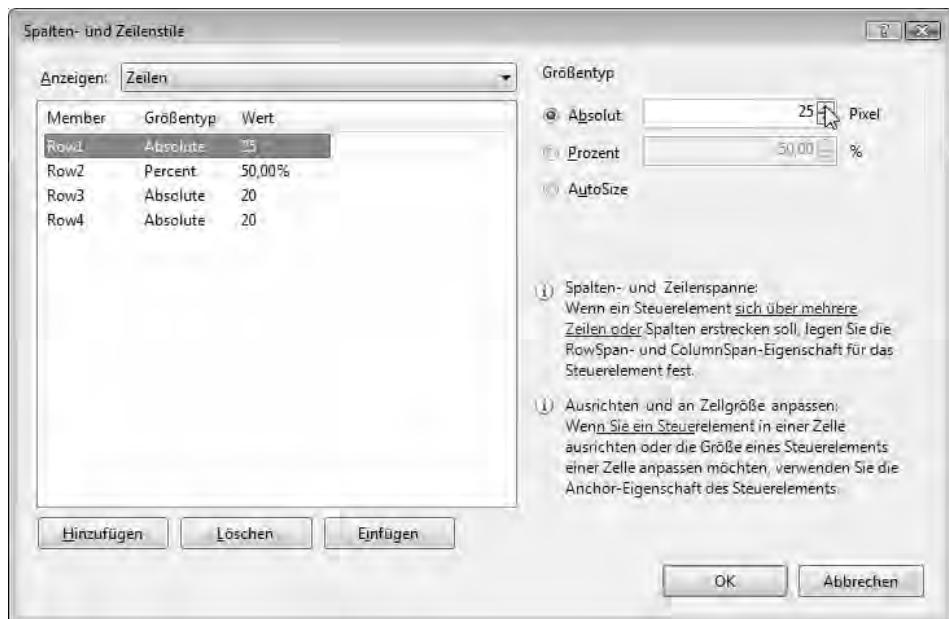


Abbildung 5.15 In diesem Dialog konfigurieren Sie die Größenverhältnisse der einzelnen Zellen

- Klicken Sie auf die erste Zeile der Zeilenliste, und stellen Sie den Größentyp auf *Absolut*. Geben Sie 25 Pixel als Höhe der ersten Zeile ein. Orientieren Sie sich an Abbildung 5.15. Sie bestimmen damit, dass sich diese Zeile nicht dynamisch vergrößert, sondern stets eine Höhe von 25 Pixeln enthält. Da sich in dieser Zeile später das Eingabefeld für den Filmtitel befindet, der nur einzeilig eingegeben werden soll, braucht sich das Eingabefeld nicht in Y-Richtung zu vergrößern, egal, wie groß der Abstand nach unten auch wird. Vielmehr macht es durch diese Einstellung sogar Sinn, den Abstand klein zu halten, damit anderen Zeilen, die Steuerelemente enthalten werden und die sich vergrößern sollen, mehr Platz für ihre Inhalte zur Verfügung steht.

WICHTIG Wenn Sie eine Wertänderung vorgenommen haben, drücken Sie NICHT , sondern klicken Sie, falls Sie weitere Größentypen oder Werte ändern möchten, einfach auf die nächste Zeile in der Liste, deren Einstellungen Sie ändern möchten. bewirkt das Auslösen der OK-Schaltfläche, und damit verschwindet der Dialog vom Bildschirm; Sie müssen den Dialog dann erneut aufrufen.

HINWEIS Wenn Sie einen Mix aus absoluten und prozentualen Größentypen verwenden, dann werden vom maximal zur Verfügung stehenden Platz (egal ob für Zeilen in der Höhe oder Spalten in der Breite) die absoluten Pixelangaben abgezogen. Der Rest des zur Verfügung stehenden Platzes wird zwischen den anderen Zellen so aufgeteilt, wie es den prozentualen Werteangaben entspricht.

5. Für *Row2* (Zeile 2) belassen Sie den Größentyp auf *Prozent*, geben als Wert allerdings **25%** ein.
 6. Für *Row3* und *Row4* ändern Sie den Größentyp auf *Prozent* und geben als Wert **25%** und **50%** ein.
 7. Wählen Sie anschließend aus der Aufklappliste *Anzeigen* das Element *Spalten*. Klicken Sie die Zeile *Column1* (Spalte 1) an, setzen Sie den Größentyp auf *Absolut* und geben Sie **120** als feste Pixelbreite ein.
 8. An *Column2* (Spalte 2) nehmen Sie ebenfalls eine kleine Änderung vor, indem Sie die Prozentangabe für diese Spalte auf **100%** einstellen.
- Mit diesen Einstellungen haben Sie nun erreicht, dass die linke Spalte, die die immer gleichlautenden Beschriftungen enthalten wird, stets eine Breite von 120 Pixel behält, während sich die rechte Spalte dynamisch vergrößern oder verkleinern kann.
9. Sie können den Dialog nun mit bestätigen und damit schließen, da Sie ihn jetzt nicht mehr benötigen.
 10. Klappen Sie die noch geöffnete Kontextaufgabenliste durch Klick auf den Smarttag des *TableLayoutPanel* auch wieder zu.

Das Ergebnis sollte anschließend in etwa wie in Abbildung 5.16 ausschauen.

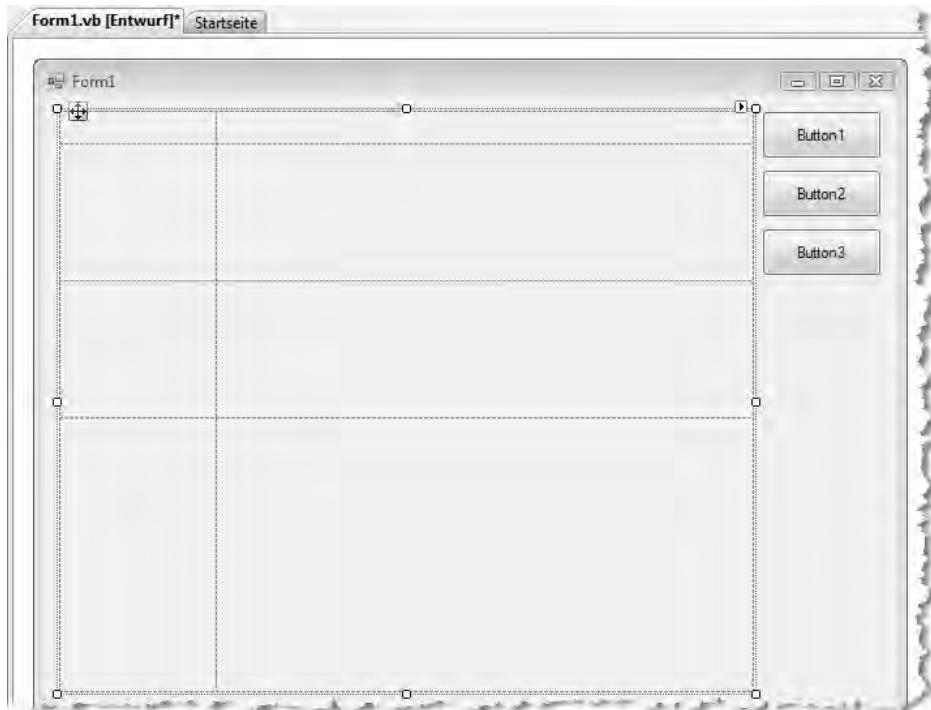


Abbildung 5.16 So sollte das vorläufige Ergebnis ausschauen, nachdem Sie die Zeilen- und Spaltenparameter der Zellenbestimmt haben

Anordnen von Steuerelementen in den Zellen eines TableLayoutPanel

Ein Blick auf das Ergebnis in Abbildung 5.2 offenbart, was als nächstes auf uns zukommt: die Anordnung der TextBox- und Label-Steuerelemente für die Eingabe der Daten sowie deren Beschriftung.

- Zum Platzieren der Steuerelemente fügen Sie sie per Drag & Drop aus der Toolbox in die Zellen des TableLayoutPanel ein. Abbildung 5.17 hilft Ihnen bei diesem Vorgang.

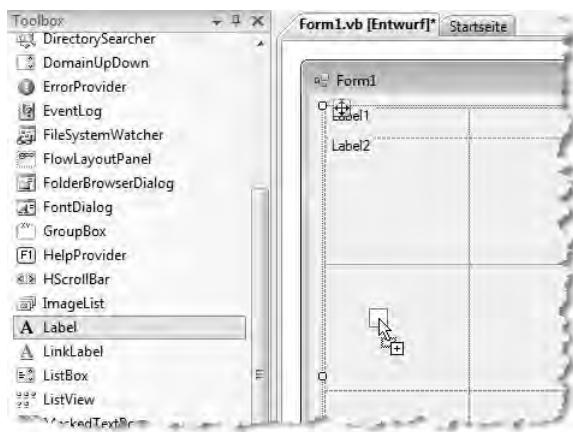


Abbildung 5.17 Ziehen Sie zum Platzieren der Steuer-elemente diese einfach per Drag & Drop in die jeweiligen Zellen des TableLayoutPanel

- Nach diesem Verfahren ziehen Sie drei Label-Elemente in die oberen drei linken Zellen und drei TextBox-Elemente in die oberen drei rechten Zellen.
- In die untere linke Zelle fügen Sie ein Panel-Steuerelement ein, das später als Träger für die Auslassungsschaltfläche (...) zur Wahl der Grafikdatei sowie für ein weiteres Panel und ein darin geschachteltes PictureBox-Steuerelement dient.

Anschließend sollte sich das Ergebnis in etwa wie in Abbildung 5.18 gestalten.

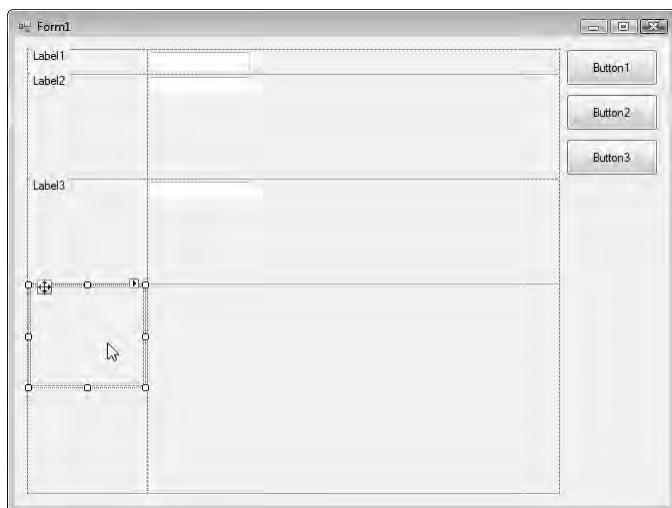


Abbildung 5.18 Nach dem Einfügen aller Steuerelemente sollten Sie in etwa dieses Ergebnis vorfinden

HINWEIS Wozu dient das Panel in der linken, unteren Ecke (wir benötigen dort schließlich ein Bild und eine Auslassungsschaltfläche für die Auswahl des Bildes)? Das verhält sich folgendermaßen: In jeder Zelle eines TableLayoutPanel darf sich nur ein Steuerelement befinden. Das bedeutet jedoch nicht, dass dieses eine Steuerelement nicht als Container für andere Steuerelemente fungieren kann. Auf diese Weise stehen Ihnen Tür und Tor offen für komplexe Gebilde, die jedoch, je komplexer und verschachtelter sie werden, einen entscheidenden Nachteil haben: Das Neuanordnen solcher Gebilde zur Laufzeit kostet Zeit – gerade bei langsamem Grafikkarten, weil natürlich die Inhalte alle Steuerelemente bei der kleinsten Größenänderung aktualisiert werden müssen. Sie sollten sich eigentlich mit maximal einem Container pro Zelle begnügen und die Anzahl der Steuerelemente in Containern pro Zelle wirklich so eng wie möglich begrenzen. Nur in Sonderfällen – die folgenden Abschnitte demonstrieren einen solchen – sollten Sie eine weitere Verschachtelungstiefe (im Sinne von »Container enthält Container enthält die eigentlichen Steuerelemente«) in Erwägung ziehen.

Verankern von Steuerelementen im TableLayoutPanel

Nun ist die Anordnung, wie Sie sie in Abbildung 5.18 sehen können, noch nicht besonders elegant. Die Beschriftungen »hängen« in der linken, oberen Ecke und die Textbox-Elemente erstrecken sich nicht über die volle Breite des Formulars. Das zu ändern hängt wieder buchstäblich an der Anchor-Eigenschaft der einzelnen Steuerelemente.

1. Setzen Sie die Anchor-Eigenschaft des oberen Label-Steuerelements auf *Left, Right*.
2. Setzen Sie die Anchor-Eigenschaften der verbleibenden Label-Steuerelemente auf *Left, Top, Right*.
3. Damit die Label-Elemente einen schöneren Eindruck machen, selektieren Sie alle, und...
4. ...setzen Sie alle BorderStyle-Eigenschaften der Label auf *Fixed3D*.
5. Setzen Sie alle TextAlign-Eigenschaften der Label auf *MiddleRight*.
6. Setzen Sie die Anchor-Eigenschaft des oberen Textbox-Steuerelements auf *Left, Right*.
7. Bevor wir die Anchor-Eigenschaften der zwei unteren Textbox-Steuerelemente ändern können, müssen wir diese für den mehrzeiligen Betrieb einstellen. Nur dann kann eine Textbox den gesamten zur Verfügung stehenden Bereich einer Zelle ausnutzen und an allen vier Seiten ohne Abstand verankert werden. Selektieren Sie dazu beide unteren Textbox-Steuerelemente.
8. Stellen Sie die Multiline-Eigenschaft der Textbox auf *True*. Damit diese Textboxen automatisch Rollbalken erhalten, falls ein Anwender mehr Zeilen eingibt als sichtbarer Platz zur Verfügung steht, setzen Sie die Scrollbars-Eigenschaft auf *Vertical*.
9. Selektieren Sie nun zusätzlich das Panel, indem Sie die Taste `Strg` festhalten und auf das Steuerelement klicken.
10. Stellen Sie jetzt die Anchor-Eigenschaft aller selektierten Steuerelemente auf *Top, Bottom, Left, Right*. Das Ergebnis sollte nun ausschauen wie in Abbildung 5.19.

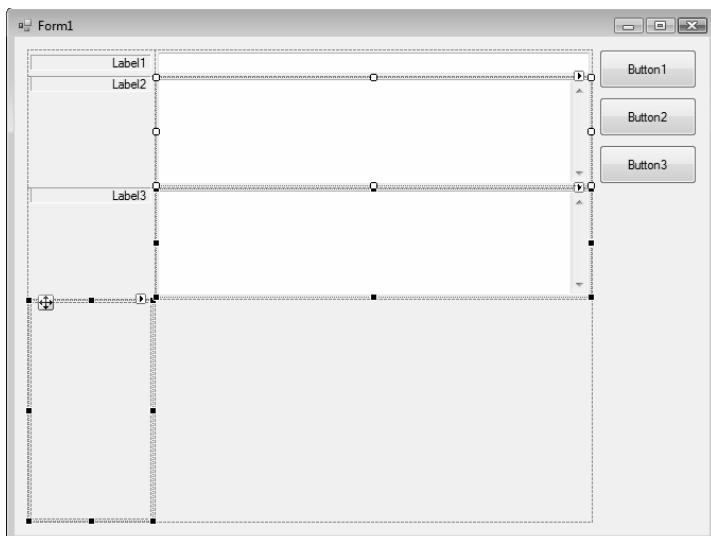


Abbildung 5.19 Nach dem Verankern aller Steuerelemente sollte das Formular wie hier ausschauen

Verbinden von Zeilen oder Spalten des TableLayoutPanel

Das TableLayoutPanel fungiert neben seiner schon bekannten Funktionalität auch als so genannter *Property Extender*.

Was ist ein Property Extender?

Ein Steuerelement oder eine Komponente, die als Property Extender (etwa *Eigenschaftenerweiterer*) ausgetragen ist, ergänzt Steuerelemente, die im gleichen Container wie sie selbst liegen, oder Steuerelemente, die sie beinhaltet (wenn es sich bei dem Property Extender selbst um ein Container-Steuerelement handelt), um weitere Eigenschaften.

Eine solche Komponente schaut sich dabei an, welche Steuerelemente, die in ihrer Reichweite liegen, für die Erweiterung durch neue Eigenschaften, die im Kontext Sinn machen, in Frage kommen und stattet diese Steuerelemente mit neuen Eigenschaften aus. Diese Eigenschaften existieren aber dann nur für diese erreichbaren Steuerelemente, da sich diese neuen Eigenschaften grundsätzlich auf irgendeine Interaktion mit dem Property-Extender-Steuerelement beziehen sollten.

Steuerelemente außerhalb der Reichweite des Property-Extender oder Steuerelemente in anderen Formularen sind davon nicht betroffen.

Aus diesem Grund haben alle Steuerelemente, die sich innerhalb einer Zelle eines TableLayoutPanel befinden, vier weitere Eigenschaften: *Column*, *ColumnSpan*, *Row* und *RowSpan*. Mit *Column* und *Row* wird festgelegt, in welcher Zeile und Spalte sich das Steuerelement befindet. Interessanter sind *ColumnSpan* und *RowSpan*: Diese bestimmen, über wie viele Zeilen und Spalten sich das Steuerelement erstrecken soll.

Das Panel, das später PictureBox und Auslassungsschaltfläche beinhalten soll, erstreckt sich bei genauerer Betrachtung (siehe Abbildung 5.2) über die komplette Spaltenbreite. Aus diesem Grund sollte seine *ColumnSpan*-Eigenschaft auf 2 (die Anzahl an Spalten im TableLayoutPanel) gesetzt werden.

1. Löschen Sie dazu als erstes die aktuelle Selektion der anderen Steuerelemente, damit Sie nicht versehentlich die `ColumnSpan`-Eigenschaften der zuletzt selektierten Steuerelemente mit neu setzen (ist mir gerade passiert). Klicken Sie dazu auf einen freien Bereich im Formular.
2. Selektieren Sie das Panel und setzen Sie die `ColumnSpan`-Eigenschaft auf 2.
3. Für weitere Aufgaben, die in den folgenden Abschnitten besprochen werden, fügen Sie innerhalb des Panel eine Schaltfläche mit vergleichsweise kleinen Ausmaßen sowie ein weiteres Panel ein.
4. Platzieren Sie die Schaltfläche in der rechten oberen Ecke des sie umschließenden Panel. Kontrollieren Sie die `Location`-Eigenschaft, die die linke obere Ecke des Steuerelements widerspiegelt, dahingehend, dass der Y-Wert nicht negativ sondern am besten 0 ist. Damit liegt die Schaltfläche ganz oben an.

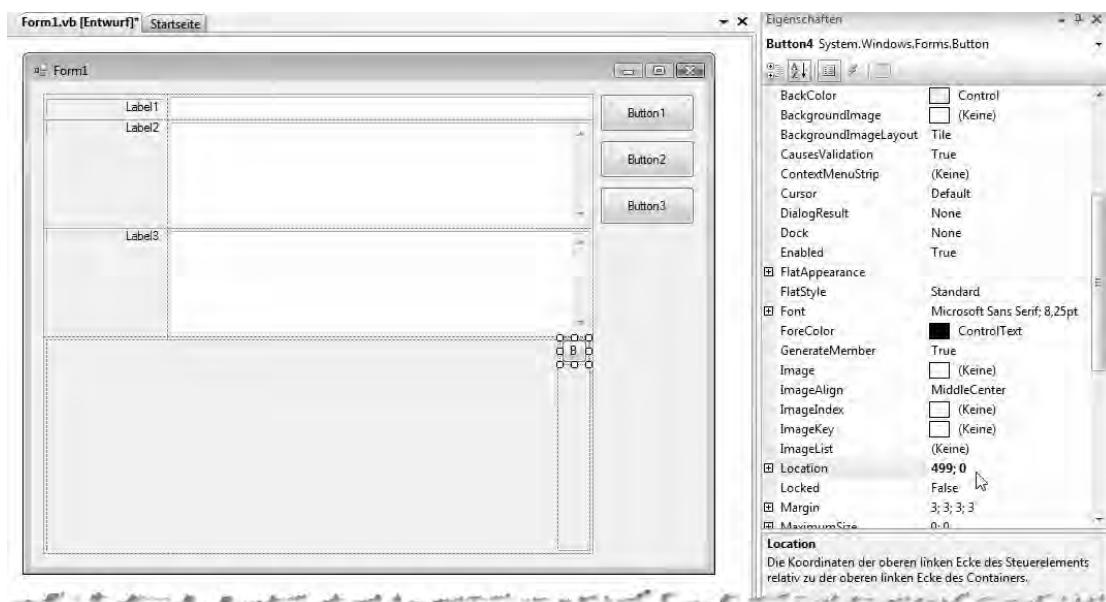


Abbildung 5.20 Die untere Zeile des TableLayoutPanel enthält nun ein Panel, das sich über zwei Spalten erstreckt. Es beinhaltet ein weiteres Panel sowie eine Schaltfläche zur späteren Bilddateiauswahl.

5. Vergrößern Sie das zweite Panel, das Sie gerade eingefügt haben, so, dass es möglichst ohne Abstand an den äußeren Rändern des umgebenden Panel anliegt. Die Eigenschaften `Location` und `Size` können Ihnen beim Feintuning helfen.
6. Setzen Sie die `Anchor`-Eigenschaft der Schaltfläche auf `Top, Right`.
7. Setzen Sie die `Anchor`-Eigenschaft des zweiten Panel auf `Top, Bottom, Left, Right`.

Sie sollten anschließend ein Ergebnis etwa wie das in Abbildung 5.20 sehen.

Automatisches Scrollen von Steuerelementen in Containern

Vielleicht stellen Sie sich die Frage, wieso wir neben die Schaltfläche ein weiteres Panel und dort nicht direkt die PictureBox platziert haben. Die Antwort zeigt sich wie von selbst, wenn Sie nochmals einen Blick auf Abbildung 5.2 werfen: Der Bildausschnitt bei Coverbildern, die größtmäßig nicht in die PictureBox passen, soll gescrollt werden können. Leider stellt die PictureBox eine solche Funktionalität nicht zur Verfügung.

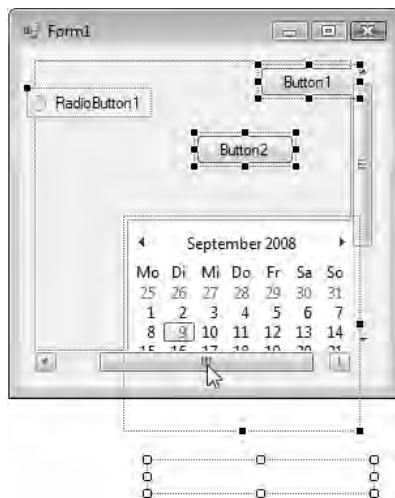


Abbildung 5.21 In diesem Beispiel befinden sich alle Steuerelemente in einem Panel, das als Container fungiert. Da nicht alle Steuerelemente in den sichtbaren Ausschnitt passen und seine AutoScroll-Eigenschaft gesetzt ist, lässt sich der dargestellte Ausschnitt zur Entwurfs- und Laufzeit mit automatisch zur Verfügung gestellten Rollbalken einstellen.

Allerdings stellen Container-Steuerelemente eine Funktionalität zur Verfügung, mit der wir sehr nah an das herankommen, was wir erreichen wollen. Wenn Sie die AutoScroll-Eigenschaft eines Container-Steuerelements auf True setzen, dann lassen sich die in ihm enthaltenen, aber durch einen zu kleinen sichtbaren Ausschnitt ausgeblendeten Steuerelemente mit den automatisch erzeugten Rollbalken in das Sichtfenster des Containers »einrollen«. Abbildung 5.21 demonstriert dieses Verhalten recht anschaulich.

Dieses Verhalten nutzen wir nun aus: Die PictureBox erlaubt eine Einstellung, die das PictureBox-Steuerelement an die Größe des Bildes anpasst – diese Eigenschaft lautetSizeMode, und sie muss zu diesem Zweck auf AutoSize gestellt werden. Befindet sich die PictureBox in einem Panel, dessen AutoSize-Eigenschaft gesetzt ist, und wird ein Bild geladen, das die PictureBox dazu zwingt sich so weit zu vergrößern, dass sie nicht mehr in das Panel passt, bekommt das Panel ohne weiteres Zutun Rollbalken. Wenn nun nur das Panel über einen Rahmen verfügt und die PictureBox keine weiteren sichtbaren Abgrenzungen aufweist, dann schaut es für den Anwender so aus, als ließe sich das Bild in der PictureBox scrollen – in Wirklichkeit scrollt aber das Panel die ganze PictureBox. Ihr Vorteil: Sie haben nicht nur gerade mehrere Tage Entwicklungsarbeit gespart,³ Sie brauchten sogar noch *nicht einmal eine einzige Zeile* Code dazu zu schreiben. Gehen wir's also an:

³ Kein Scherz: Ein Steuerelement mit scrollbarem Inhalt zu entwickeln, ist wirklich keine triviale Sache.

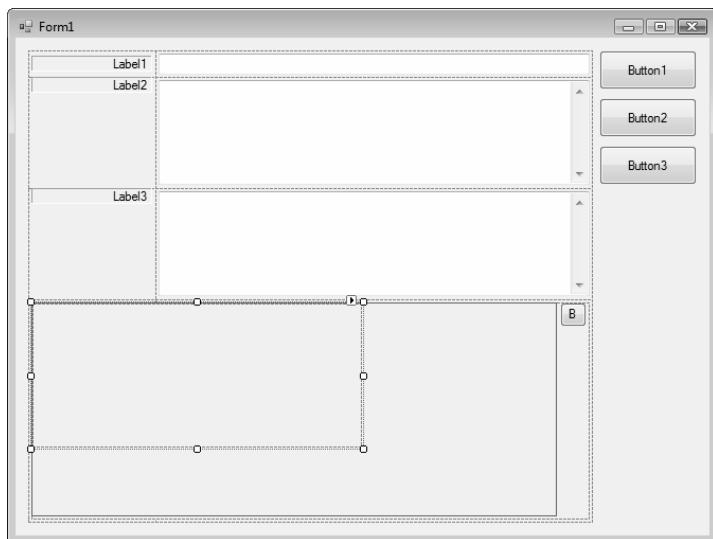


Abbildung 5.22 Die geschachtelte Kombination aus Panel, Panel und PictureBox stellt später die Scroll-Funktionalität zur Verfügung – so sollte das vorläufige Ergebnis ausschauen

1. Selektieren Sie das Panel neben der Auslassungsschaltfläche, sofern dieses noch nicht selektiert ist.
2. Setzen Sie dessen BorderStyle-Eigenschaft auf FixedSingle.
3. Setzen Sie dessen AutoScroll-Eigenschaft auf True.
4. Ziehen Sie eine PictureBox im Panel auf.
5. Damit es mit der Maus nicht zu filigran wird: Setzen Sie die Location-Eigenschaft der PictureBox »zu Fuß« auf 0;0.

HINWEIS Denken Sie daran, dass sich Koordinaten eines Steuerelements immer auf seinen unmittelbaren Container beziehen.

6. Setzen Sie dieSizeMode-Eigenschaft der PictureBox auf AutoSize.

Das Ergebnis sollte nun in etwa dem in Abbildung 5.22 entsprechen.

Selektieren von Steuerelementen, die Sie mit der Maus nicht erreichen

Bei solchen Verschachtelungen, wie Sie sie im vorherigen Abschnitt kennen gelernt haben, wird es ziemlich schwierig, bestimmte Steuerelemente mit der Maus zu selektieren – Sie treffen bisweilen einfach nicht mehr eine Begrenzungslinie des Steuerelements, das Sie eigentlich treffen wollten, und selektieren ein ganz anderes.

In diesem Fall selektieren Sie einfach ein anderes. Drücken Sie dann so oft , bis Sie das Steuerelement selektiert haben, das Sie auch tatsächlich selektieren wollten. Und falls das auch nicht mehr hilft:

Selektieren von Steuerelementen mit dem Eigenschaftenfenster

Das Eigenschaftenfenster gibt Ihnen in der oberen Zeile Information über das derzeit selektierte Steuerelement. Bei dieser Infozeile handelt es sich allerdings um eine Aufklappliste. Wählen Sie aus dieser Liste ein Steuerelement oder eine Komponente aus, wird diese im Formular-Designer selektiert. Hier erkennen Sie, wie wichtig es ist, den Steuerelementen sprechende und selbsterklärende Namen zu geben: Sonst wünschen wir Ihnen viel Spaß z. B. bei Suche nach der gewünschten TextBox in der Liste unter TextBox1, TextBox2, etc.

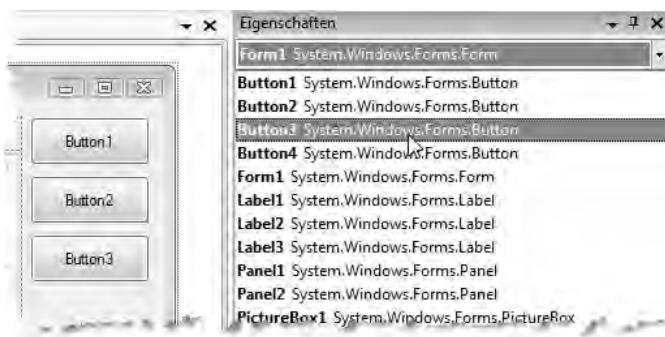


Abbildung 5.23 Mithilfe des Eigenschaften-fensters können Sie jedes Steuerelement im Designer selektieren

Festlegen der Tabulatorreihenfolge (Aktivierreihenfolge) von Steuerelementen

Die Tabulatorreihenfolge – die Microsoftterminologie lautet »Aktivierreihenfolge« – bestimmt, welches Steuerelement jeweils als nächstes aktiviert wird, wenn der Anwender es vorzieht, mit der Tastatur zu arbeiten, und mit **[Tab]** zum jeweils nächsten Element springen will.

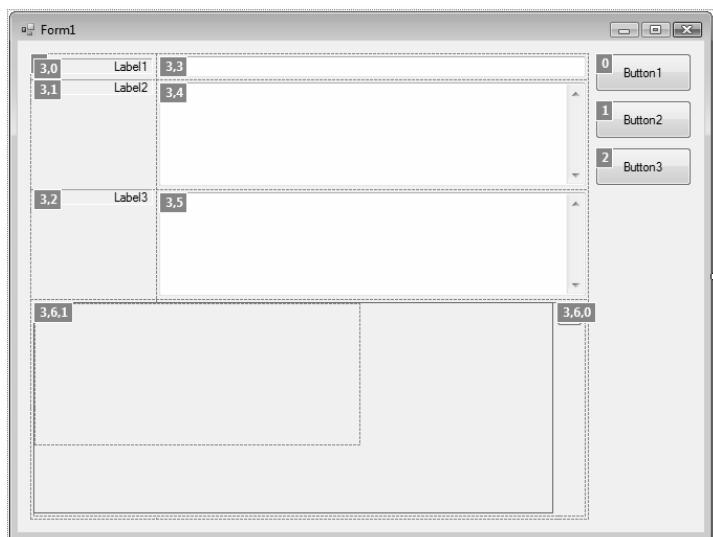


Abbildung 5.24 Nach dem Aufrufen von Ansicht/Aktivierreihenfolge können Sie die Reihenfolge festlegen, mit der Steuerelemente zur Laufzeit per **[Tab]** zu erreichen sind. Das Formular muss dazu selektiert sein.

Mit Visual Studio 2008 ist das ruckzuck erledigt. Um beim Beispiel zu bleiben:

1. Klicken Sie auf die Titelzeile des Formulars, um es zu selektieren.
2. Wählen Sie aus dem Menü *Ansicht* den Befehl *Aktivierreihenfolge*. Das Formular ändert seine Darstellung, etwa wie in Abbildung 5.24 zu sehen.
3. Klicken Sie nacheinander die Steuerelemente in der Reihenfolge an, wie sie durch  zur Laufzeit des Programms aktiviert werden soll. Beziehen Sie dabei auch die Container mit ein; etwas knifflig wird es beim Bestimmen des ersten Steuerelements – dem *TableLayoutPanel* – dessen Aktivierreihenfolgenordinalnummer wie in Abbildung 5.24 zu sehen, etwas in den Hintergrund gerät (sie liegt hinter der Beschriftung 3,0). Klicken Sie am besten auf den äußersten Rand der linken oberen Ecke, um es »zu erwischen«. Orientieren Sie sich dabei am Ergebnis, das Sie in Abbildung 5.25 sehen können.

TIPP Wenn Sie sich im Aktivierreihenfolgenmodus befinden und mit dem Mauszeiger über ein Steuerelement fahren, wird es zur besseren Orientierung mit einem dicken, gerasterten Rahmen gekennzeichnet. In Abbildung 5.25 sehen Sie das bei *Button3*.

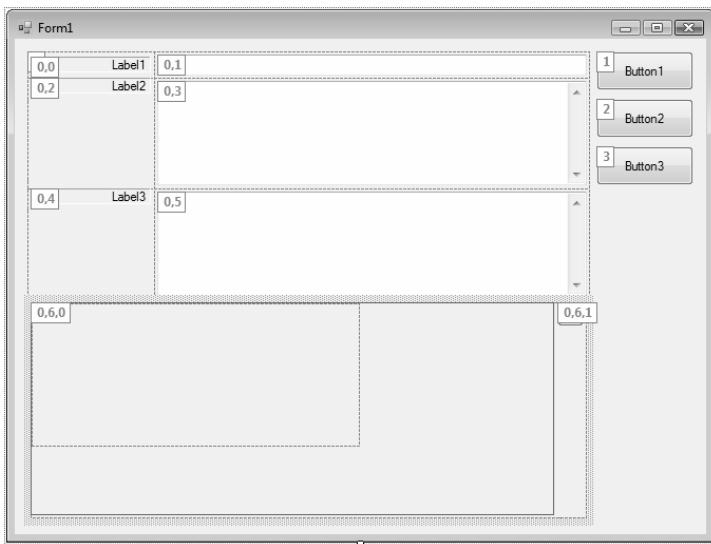


Abbildung 5.25 Die Ordinalnummern aller Steuerelemente, die Sie bereits bestimmt haben, werden mit einem hellen Hintergrund eingefärbt. Drücken Sie , wenn Sie mit der Zuweisung fertig sind.

4. Nachdem Sie die letzte Schaltfläche angeklickt haben, drücken Sie , um den Aktivierreihenfolgenmodus zu verlassen.

TIPP Die Aktivierreihenfolge wird durch die *TabIndex*-Eigenschaft eines Steuerelements festgelegt. Sie können die Aktivierreihenfolge deswegen auch ändern, indem Sie die *TabIndex*-Eigenschaften der Steuerelemente manuell anpassen. Es erfolgt allerdings keine automatische Neunummerierung. Wenn Sie zwischen dem Steuerelement mit dem *TabIndex*3 und dem mit *TabIndex*4 ein weiteres Steuerelement einfügen wollen, müssen Sie ab dem Steuerelement mit dem *TabIndex*4 alle *TabIndex*-Einträge um 1 erhöhen. Hoffentlich hat Ihr Formular nicht 36 Steuerelemente.

Der eigentliche Aufbau des Formulars samt seiner kompletten Größenanpassungs-Funktionslogik ist damit abgeschlossen. Rekapitulieren Sie: Für das, was Sie schätzungsweise in der letzten Stunde erreicht haben, hätten Sie noch in Visual Basic 6.0 sehr, sehr lange programmieren müssen. Mit dem PictureBox-Steuerelement, das eine Scroll-Funktionalität für seinen Inhalt angeboten hätte, vielleicht sogar mehrere Tage.

Über die Eigenschaften Name, Text und Caption

Was für das Beispiel auf Designer-Seite jetzt noch fehlt, sind die Beschriftungen (für die Steuerelemente, die es betrifft) und die Namen der Steuerelemente, unter denen Sie sie beim Programmieren »erreichen« können.

Diese Dinge werden mit den Eigenschaften **Name** (für den Namen eines Steuerelements, mit dem Sie es beim Programmieren ansprechen) und **Text** (für Beschriftungen) festgelegt.

TIPP Ausnahmslos jedes Steuerelement verfügt über eine **Text**-Eigenschaft, da diese auch in **Control** implementiert ist, auf das jedes Steuerelement aufbaut. Zwar gibt es Steuerelemente, die ihre **Text**-Eigenschaft nicht im Eigenschaftenfenster offen legen, dennoch ist die **Text**-Eigenschaft bei diesen vorhanden.

WICHTIG Die **Caption**-Eigenschaft, wie Sie sie von Visual Basic 6 beispielsweise für das Festlegen der Beschriftung von Schaltflächen und **Label**-Steuerelementen kennen, gibt es in .NET NICHT mehr. Auch hier übernimmt, wie es unter VB6 beispielsweise auch schon immer beim **Textbox**-Steuerelement der Fall war, die **Text**-Eigenschaft diese Aufgabe.

Und noch ein ...

TIPP Wenn Sie **Text**- und **Name**-Eigenschaften zuweisen, sollten Sie das nicht wechselweise sondern nacheinander machen. Bestimmen Sie also erst alle **Name**-Eigenschaften und dann alle **Text**-Eigenschaften. Der Grund: Wenn Sie das erste Steuerelement angeklickt und dann beispielsweise die **Text**-Eigenschaft im Eigenschaftenfenster festgelegt haben, brauchen Sie anschließend einfach nur das nächste Steuerelement anzuklicken und einfach drauflos zu tippen. Visual Studio merkt sich, dass Sie beim letzten Gebrauch des Eigenschaftenfenseters die **Text**-Eigenschaft gesetzt hatten, und leitet in dem Moment, in dem Sie nach der Selektion eines neuen Steuerelements zu tippen begonnen haben, die Tastatureingaben automatisch an die zuletzt verwendete Eigenschaft im Eigenschaftenfenster weiter.

Schnellzugriffstasten durch die **Text**-Eigenschaft bestimmen

Die **Text**-Eigenschaft übernimmt bei vielen Steuerelementen übrigens noch eine weitere Funktion: Buchstaben, vor die Sie das &-Zeichen stellen, markieren Sie als Schnellzugriffstasten. Sie erkennen die Schnellzugriffstasten eines Steuerelements dadurch, dass diese unterstrichen dargestellt werden. Zur Laufzeit befähigen Sie den Anwender Ihres Programms dann, dieses Steuerelement anzusteuern oder – bei Schaltflächen beispielsweise – auszulösen, indem er die Schnellzugriffstaste in Verbindung mit **Alt** betätigt. Wenn Sie z.B. eine **TextBox** über diese Schnellzugriffstaste anspringen wollen, sollten Sie ein **Label**-Steuerelement in der Aktivierreihenfolge VOR die **TextBox** stellen und dort die Schnellzugriffstaste durch das &-Zeichen festlegen. Da **Label** (also reine Beschriftungen) unter Windows standardmäßig nicht angesprungen werden, wird die nachfolgende **TextBox** angesteuert. Unten im Beispiel sehen Sie dieses Vorgehen.

Auf diese Weise bestimmen Sie zunächst die Text-Eigenschaften aller Label- und Button- (Schaltflächen-) Steuerelemente. Orientieren Sie sich dabei am besten an Abbildung 5.26 und an folgender Tabelle.

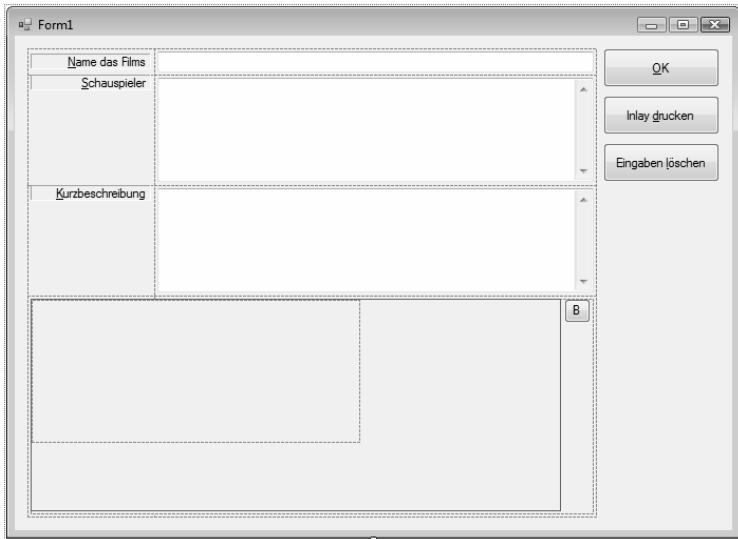


Abbildung 5.26 Nehmen Sie bei der Zuweisung von Beschriftungen und Schnellzugriffstasten an die Steuerelemente diese Grafik zu Hilfe

Steuerelement	Name (Name-Eigenschaft)	Beschriftung (Text-Eigenschaft)
Label (links, oben)	lblNameDesFilms	"Name des &Films"
Label (links, mitte)	lblSchauspieler	"&Schauspieler"
Label (links, unten)	lblKurzbeschreibung	"&Kurzbeschreibung"
Schaltfläche (rechts, oben)	btnOK	"&OK"
Schaltfläche (rechts, mitte)	btnInlayDrucken	"Inlay &drucken..."
Schaltfläche (rechts, unten)	btnEingabenLöschen	"Eingaben &löschen"
Textbox (oben, mitte)	txtNameDesFilms	"" (Leerstring – Löschen Sie die Eingabe)
Textbox (mitte, mitte)	txtSchauspieler	""
Textbox (unten, mitte)	txtKurzbeschreibung	""
Schaltfläche (neben der PictureBox)	btnCoverbildWählen	"..."
PictureBox	picCoverbild	– nicht anwendbar –

Tabelle 5.1: Verwenden Sie diese Name- und Text-Eigenschaften für die Steuerelemente im Formular

Sie erkennen, dass ich bei der Benennung von Steuerelementen nach einem bestimmten Schema vorgehe, indem ich im Namen mit einer Drei-Buchstaben-Abkürzung kenntlich mache, um was für ein Steuerelement es sich handelt. Ob Sie für eigene Namensgebungen von Steuerelementen diese Konvention adaptieren oder nicht, bleibt Ihnen natürlich überlassen. Viele Entwickler machen das, einige nicht – Microsoft sagt,

man soll es *nicht* tun. Sollten Sie sich dafür entscheiden – im Abschnitt »Namensgebungskonventionen für Steuerelemente« ab Seite 130 finden Sie eine Tabelle, die die Konventionen für die wichtigsten Steuerelemente auflistet.

HINWEIS Auch das Formular selbst verfügt über eine Text-Eigenschaft. In diesem Fall bestimmt sie den Text, der in der Titelzeile des Formulars erscheinen soll.

1. Selektieren Sie das Formular.
2. Bestimmen Sie im Eigenschaftenfenster als Text-Eigenschaft einen Text, der Ihnen geeignet scheint – beispielsweise »Der VB-Entwicklerbuch-Hüllen-Inlay-Generator« oder Ähnliches.

Einrichten von Bestätigungs- und Abbrechen-Funktionalitäten für Schaltflächen in Formularen

Wie Sie Schnellzugriffstasten einrichten, haben Sie im letzten Abschnitt bereits erfahren. Es gibt zwei Tasten bei der Formularbedienung, denen eine besondere Funktionalität zukommt, und die Sie mit diesem Verfahren allerdings nicht definieren können: **↵** (in der Regel für die OK-Schaltfläche), um einen Dialog zu bestätigen und **Esc**, um einen Dialog wieder zu verlassen, ohne dass die Eingaben übernommen werden.

VB6-Entwickler werden bei den Schaltflächeneigenschaften vergeblich nach diesen Einstellungsmöglichkeiten suchen – sie wurden in .NET nämlich in das Formular verlagert. Das macht letzten Endes auch mehr Sinn, schließlich kann nur jeweils eine Schaltfläche für das Abbrechen und eine weitere für das Bestätigen eines Dialogs in Frage kommen.

Diese Aufgaben übernehmen also zwei Formulareigenschaften namens `AcceptButton` und `CancelButton`. Beim Setzen dieser Eigenschaften klappen Sie eine Aufklappliste auf, die alle Schaltflächenelemente des Formulars enthält, die für die jeweilige Aufgabe in Frage kommen.

Für unser Beispiel weichen wir ein wenig vom Standard ab. Wir legen die Abbrechen-Funktionalität auf die OK-Schaltfläche und die Bestätigen-Funktionalität auf die *Inlay drucken*-Schaltfläche. Das macht mehr Sinn für den Anwender, denn er verlässt ja schließlich auch mit OK den Dialog. Ein versehentliches Betätigen von **↵** führt damit nicht zum Dialogende (und damit zum Beenden des Programms).

1. Selektieren Sie das Formular im Designer.
2. Setzen Sie die `AcceptButton`-Eigenschaft auf die Schaltfläche `btnInlayDrucken`.
3. Setzen Sie die `CancelButton`-Eigenschaft auf die Schaltfläche `btnOK`.

Hinzufügen neuer Formulare zu einem Projekt

Eine Smartclient-Anwendung besteht in den wenigsten Fällen aus nur einem Formular. In unserem Fall ist das genauso wenig der Fall. Wir benötigen ein weiteres Formular, das später die Druckvorschau enthält, und mit dem man den Druckvorgang auslösen kann. Diesem Formular werden dann später zur Laufzeit die zu druckenden Daten übergeben – die Funktion des Druckens und der Vorschau darstellung soll es dann völlig autonom übernehmen.

HINWEIS Trennen Sie sich schon beim Designen der Formulare Ihrer Anwendung von der prozeduralen Denkweise, falls Sie es noch nicht getan haben. Es zeugt von einem schlechten Programmierstil, wenn Sie quasi von außen, also beispielsweise aus einem anderen Formular, einem anderen Modul oder einer anderen Klasse auf die Elemente eines Formulars zugreifen. Stellen Sie besser nur wenige öffentliche Methoden im Formular bereit, die ausschließlich dazu da sind, Parameter entgegen zu nehmen, und überlassen Sie dem Formular selbst die Auswertung dieser Parameter. Sie sollten – auch in Vorbereitung auf objektorientierte Programmierung – jedes Formular als eine kleine, in sich geschlossene Anwendung betrachten, der Sie lediglich Daten übergeben können und Resultate von diesem wiederbekommen.

WICHTIG Was im Formular passiert (das Setzen oder Abfragen von Eingabefeldern, das Auswerten von Benutzeraktionen) sollte einzig und allein der Formulkarklasse vorbehalten sein! Vermeiden Sie es, Inhalte von Benutzersteuerelementen von anderen Modulen, Klassen oder Formularen aus direkt zu manipulieren!

Um dem Projekt ein weiteres Formular hinzuzufügen, verfahren Sie wie folgt:

1. Öffnen Sie im Projektmappen-Explorer das Kontextmenü des Projektes (nicht der Projektmappe!) mit der rechten Maustaste. Falls Sie keine Projektmappe für Ihr Projekt sehen, öffnen Sie den Dialog *Extras/Optionen*, und wählen Sie im Bereich *Allgemein* die Option *Projektmappe immer anzeigen*.

Wählen Sie *Hinzufügen* und *Windows Form*.

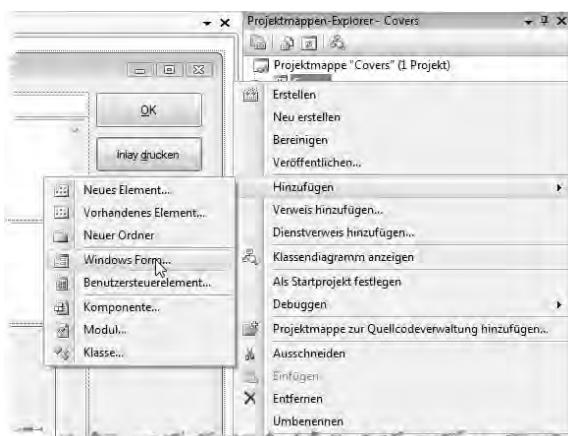


Abbildung 5.27 Mithilfe des Eigenschaftenfensters können Sie jedes Steuerelement im Designer selektieren

2. Im Dialog, der jetzt erscheint, belassen Sie den vorgegebenen Namen zunächst mit *Form2.vb* so, wie er ist.
3. Klicken Sie auf *Hinzufügen*.
4. Vergrößern Sie das neue Formular, das jetzt sofort angezeigt wird, so, dass es genug Platz für weitere Steuerelemente bereitstellt. Orientieren Sie sich am besten dazu an Abbildung 5.28.
5. Suchen Sie in der Toolbox in der Sektion *Drucken* (die Sie im Bedarfsfall aufklappen müssen) nach dem *PrintPreviewControl*. Ziehen Sie es in ausreichender Größe im Formular auf. Stellen Sie dann die einzelnen Seiten des Steuerelements mithilfe der Ausrichtungslinien so ein, dass sie alle den gleichen Abstand zum oberen, linken und rechten Formularrand haben.
6. Setzen Sie die Anchor-Eigenschaft dieses Steuerelements auf *Top, Bottom, Left, Right*.

7. Fügen Sie zwei Schaltflächen in das Formular nebeneinander ein. Setzen Sie die Anchor-Eigenschaften beider Schaltflächen auf *Bottom, Right*.
8. Weisen Sie Name- und Text-Eigenschaften der Steuerelemente mithilfe der folgenden Tabelle zu.

Steuerelement	Name (Name-Eigenschaft)	Beschriftung (Text-Eigenschaft)
PrintPreviewControl	(lassen Sie es so, wie es heißt)	–
Linker Button	btnOK	&OK
Rechter Button	btnDrucken	&Drucken...
Formular	bleibt, wie sie ist	Covervorschau zeigen und Cover drucken

Tabelle 5.2 Name- und Text-Eigenschaften für die Steuerelemente im zweiten (Druck-) Formular

9. Speichern Sie das komplette Projekt ab, indem Sie entweder aus dem Menü *Datei* den Menüpunkt *Alles Speichern* oder das entsprechende Symbol aus der Werkzeugleiste verwenden.

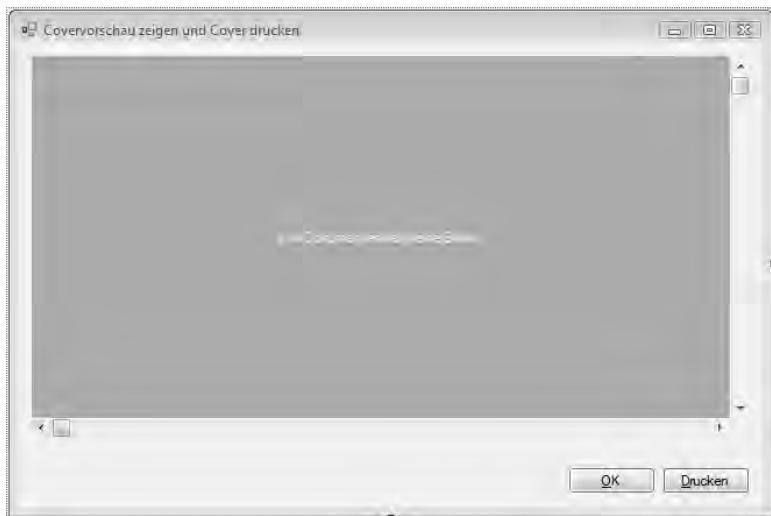


Abbildung 5.28 So soll das Druckformular im Designer endgültig aussehen

Wie geht's weiter?

Der Design-Teil unseres kleinen Softwareprojektes ist abgeschlossen – jetzt geht es darum, die entsprechenden Steuerelemente mit Programmlogik zu füttern. Die beiden folgenden Abschnitte sollen Ihnen als Referenzen zum Nachschlagen dienen – verlieren Sie ruhig einen Blick darauf, und wenn Sie dann bereit sind, mit dem Programmieren zu beginnen und die Software fertig zu stellen, fahren Sie mit dem Abschnitt »Der Codeeditor« ab Seite 133 fort.

Namensgebungskonventionen für Steuerelemente in diesem Buch

Microsoft erklärt ausdrücklich, dass die Namensgebung von Objektvariablen keinerlei Hinweise darauf enthalten soll, welchen Typ sie darstellen. Diverse Stil- und Sicherheitsüberprüfungswerkzeuge für Quellcode (beispielsweise FxCop, das bei den »größeren« Versionen von Visual Studio in den Projekteigenschaften implementiert ist) würden das bei der Quellcodeanalyse sogar anmahnen.

Eine Funktion von IntelliSense, so das Argument, reicht aus, um den Typ einer Objektvariablen eindeutig zu ermitteln – Sie brauchen lediglich den Mauszeiger auf die entsprechende Objektvariable zu positionieren, um den Typ der Variablen als Tooltip anzeigen zu lassen.

Für die Beispiele in diesem Buch habe ich mich dennoch dazu entschlossen, zumindest bei der Benennung von Objektvariablen, die Steuerelemente referenzieren, darauf zu verzichten. Ich finde, dass das das Lesen von Listings auf Papier erleichtert und zum besseren Verständnis beiträgt. IntelliSense steht Ihnen nämlich nur im Codeeditor von Visual Studio zur Verfügung – aber wer weiß: Vielleicht arbeitet Microsoft schon an einer Papierversion von IntelliSense?

Komponente	Präfix-/Namenkombination
Label	lblName
Button	btnName oder cmdName ⁴
TextBox	txtName
CheckBox	chkName
RadioButton	optName ⁵ oder rbtName
GroupBox	grpName
PictureBox	picName
Panel	pnlName
ListBox	lstName
ComboBox	cmbName
ListView	lvwName
TreeView	twvName

Tabelle 5.3 Ein Vorschlag für Namenskonventionen bei der Namensgebung von Steuerelementen

⁴ Von CommandButton – so hieß eine Schaltfläche offiziell unter VB6. Unter .NET ist dies aber nicht mehr üblich; man sieht es nur noch hier und da von VB6-Portierungen.

⁵ Von OptionButton, der Name der Optionsschaltfläche unter VB6.

Funktionen zum Layouten von Steuerelementen im Designer

Die folgende Tabelle zeigt Ihnen die Funktionen, die sich hinter den Symbolen der Werkzeugeiste *Layout* befinden. Die folgende Grafik zeigt die Ausgangspositionierung der drei Steuerelemente im Formular. In der Tabelle finden Sie in der rechten Spalte das Ergebnis der Anordnung, nachdem Sie auf das entsprechende Symbol geklickt haben.

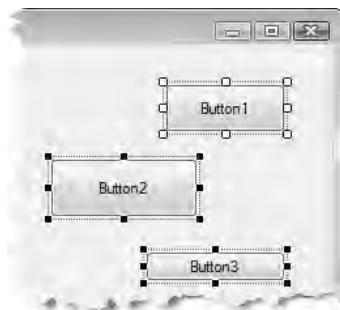


Abbildung 5.29 Die Ausgangsanordnung der Schaltflächen für die folgende Tabelle. Achten Sie darauf, dass sich bestimmte Funktionen an der oberen rechten Schaltfläche orientieren.

Symbol	Funktion	Ergebnis nach Anwendung auf Steuerelemente in Abbildung
	Links ausrichten	
	Zentrieren	
	Rechts ausrichten	
	Oben ausrichten	
	Mittig ausrichten	

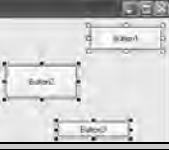
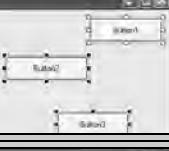
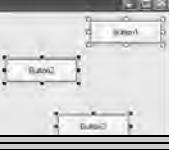
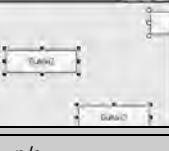
Symbol	Funktion	Ergebnis nach Anwendung auf Steuerelemente in Abbildung
	Unten ausrichten	
	Breite angleichen	
	Höhe angleichen	
	Größe angleichen	
	Horizontalen Abstand angleichen	
	Horizontalen Abstand vergrößern	– n/a –
	Horizontalen Abstand verkleinern	– n/a –
	Horizontalen Abstand entfernen	– n/a –
	Vertikalen Abstand angleichen	
	Vertikalen Abstand vergrößern	– n/a –
	Vertikalen Abstand verkleinern	– n/a –
	Vertikalen Abstand entfernen	– n/a –
	Horizontal im Formular zentrieren	– n/a –
	Vertikal im Formular zentrieren	– n/a –
	In den Vordergrund bringen	– n/a –
	In den Hintergrund bringen	– n/a –

Tabelle 5.4 Symbole, mit denen Sie erweiterte Funktionalitäten des Ausgabefensters steuern

Tastaturkürzel für die Platzierung von Steuerelementen

Taste	Aufgabe
Pfeiltasten	Verschiebt das ausgewählte Steuerelement pixelweise in die den Pfeiltasten entsprechende Richtung.
Strg + Pfeiltasten	Verschiebt das ausgewählte Steuerelement zur ersten, in der den Pfeiltasten entsprechenden Richtung liegenden Ausrichtungslinie.
↑ + Pfeiltasten	Vergrößert oder verkleinert das ausgewählte Steuerelement pixelweise entsprechend der verwendeten Pfeiltaste.
Strg ↑ + Pfeiltasten	Vergrößert oder verkleinert das Steuerelement so, dass es – abhängig von der verwendeten Pfeiltaste – mit der entsprechenden Seite an der jeweils nächsten Ausrichtungslinie anliegt.
Alt	Schaltet die Ausrichtungslinienfunktion so lange aus, wie die Taste beim Verschieben von Steuerelementen mithilfe der Maus gedrückt bleibt.

Tabelle 5.5 Symbole, mit denen Sie erweiterte Funktionalitäten des Ausgabefensters steuern

Der Codeeditor

Der Codeeditor ist mit Sicherheit das Element in Visual Studio, in dem Sie die mit Abstand meiste Zeit verbringen werden. Doch ist er über die Jahre zu einem solch genialen Werkzeug avanciert, das Ihnen soviel Arbeit abnehmen kann, dass das Arbeiten auch nach Jahren noch Spaß macht und flüssig von der Hand geht.

Das ist aber erst dann der Fall, wenn Sie alle seine Feinheiten und auch die eine oder andere Tücke kennen und zu beherrschen wissen. Und dabei geht es schon los mit der Wahl der richtigen Schriftart, die Ihnen ein ermüdfreies Arbeiten ermöglicht:

Die Wahl der richtigen Schriftart für ermüdfreies Arbeiten

Sie werden lachen: Für diesen kleinen Abschnitt habe ich tatsächlich fast zwei Stunden herumtelefoniert. Mich interessierte, ob:

1. Andere befreundete Programmierer die Original-Einstellung der Schriftart des Editors genauso stört wie mich, und
2. welche Schriftart die meisten Entwickler, die mit Visual Studio 2005 oder Visual Studio 2008 arbeiten, bevorzugen.

Das Ergebnis war eine statistische Relevanz für zwei Schriftarten, von der Meinungsforscher nur träumen können:

- Bis zu einer Auflösung von 1.024 x 768 Pixel auf einem Monitor – dies betraf besonders die Entwickler, die oft im Außendienst mit Notebooks arbeiten – ist FixedSys die bevorzugte Schrift. Da diese Schrift eine Bitmap-Schriftart ist, spielt die Fontgröße keine Rolle; sie erscheint immer in der gleichen Größe.
- Ab einer Auflösung von 1.280 x 1.024 Pixel oder bei 1.024 x 768 Pixeln im Mehrmonitorbetrieb über 2 Monitore verteilt, war Lucida Console (aber nur ab 14 Punkt) die bevorzugte Größe.

- Die originale Courier-Schrift war bei keinem der befragten 11 Entwickler im Einsatz.

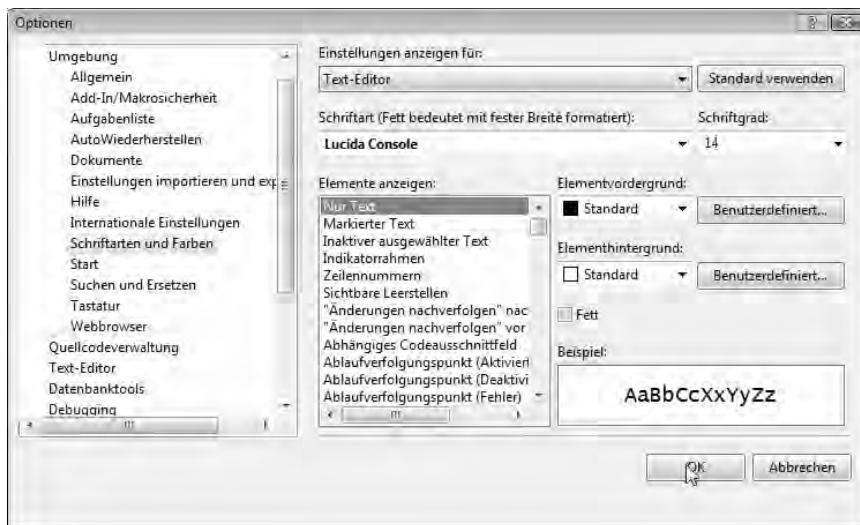


Abbildung 5.30 Auf dieser Registerkarte stellen Sie die Fonts unter anderem auch für den Codeeditor von Visual Studio ein

Um die Schriftart für den Editor umzustellen, wählen Sie aus dem Menü *Extras* den Menüpunkt *Optionen*. Wählen Sie die Registerkarte *Schrifarten und Farben*, die Sie im Zweig *Umgebung* finden. Der Aufruf dieses Dialogs dauert beim ersten Mal ein paar Sekunden – also nicht gleich ungeduldig werden und einen Absturz vermuten!

HINWEIS Die Screenshots in diesem Buch sind übrigens ebenfalls alle mit der Schriftart Lucida Console in 14 Punkt Größe entstanden. Für Präsentationen, Schulungen oder Projektbesprechungen am Beamer empfiehlt sich diese Schriftart im Übrigen auch bei kleineren Auflösungen.

Viele Wege führen zum Codeeditor

Es gibt die verschiedensten Möglichkeiten, den Codeeditor ins Leben zu rufen. Der einfachste Weg, Module oder reine Klassendateien zu bearbeiten, läuft natürlich über den Projektmappen-Explorer: Ein Doppelklick auf eine Klassen- oder eine Moduldatei öffnet die Datei im Editor direkt. Um den Klassencode eines Formulars einzusehen, müssen Sie das Formular im Projektmappen-Explorer selektieren und anschließend auf das Symbol *Code anzeigen* klicken, das Sie in der oberen Zeile des Projektmappen-Explorers finden (der Tooltip hilft Ihnen, das richtige Symbol zu erhaschen). Folgende Möglichkeiten gibt es, den Codeeditor ins Leben zu rufen:

- Doppelklick auf eine reine Klassen- oder Moduldatei im Projektmappen-Explorer.
- Bei ausgewählter Formularklasse, Mausklick auf das Symbol *Code anzeigen* im Projektmappen-Explorer.
- Bei einem Compiler-Fehler: Doppelklick auf die entsprechende Fehlermeldung im Ausgabefenster.

- Bei einem Compiler-Fehler: Doppelklick auf die entsprechende Fehlermeldung in der Fehlerliste.
- Bei Kommentaren in der Aufgabenliste: Doppelklick auf den entsprechenden Kommentar.
- Doppelklick auf ein Element in einem Designer. Ein Doppelklick auf eine Schaltfläche in einem Formular bringt Sie beispielsweise zur bereits vorhandenen Ereignisbehandlung für diese Schaltfläche oder öffnet den Editor für das Formular und fügt den Coderumpf für die Ereignisbehandlungsroutine ein.
- Nach dem Auftreten eines Fehlers während der Ausführung einer Anwendung im Debug-Modus.

IntelliSense – Ihr stärkstes Zugpferd im Coding-Stall

Das Konzept von IntelliSense erspart Ihnen beim Entwickeln die meiste Zeit. Warum? IntelliSense liefert Ihnen alle denkbaren Informationen über Objekte und Sprachelemente, die Sie gerade in Bearbeitung haben. In Visual Basic 2008 wurde IntelliSense noch mal deutlich überarbeitet, und steht jetzt schon zur Verfügung, sobald Sie im Editor den ersten Buchstaben in einer neuen Codezeile eingegeben haben.

Zur Demonstration implementieren wir nun die ersten Codezeilen unseres Beispiels.

1. Öffnen Sie *Form1* im Designer, indem Sie auf die Formulardatei im Projektmappen-Explorer doppelklicken.
2. Doppelklicken Sie anschließend auf die Schaltfläche *OK*. Visual Studio bringt Sie daraufhin zum Codeeditor für den Klassencode von *Form1* und stellt automatisch den Funktionsrumpf für die Ereignisbehandlung der *OK*-Schaltfläche zur Verfügung. Das ist die Methode, die aufgerufen und ausgeführt wird, wenn der Anwender zur Laufzeit auf die *OK*-Schaltfläche klickt.
3. In die Zeile zwischen *Private Sub...* und *End Sub* geben Sie nun **me.** ein. Sobald Sie den Punkt getippt haben, öffnet sich eine Liste mit allen Elementen, die für das Objekt anwendbar sind (in diesem Fall ist **me** das Formular selbst, da wir uns in der Klassendatei *Form1* befinden; die genaue Bedeutung lernen Sie in den Klassenkapiteln noch kennen – wir wollen uns an dieser Stelle auf die Editorfähigkeiten konzentrieren). Diese Liste nennt sich »Vervollständigungsliste«.
4. Um das Formular beim Mausklick auf *OK* zu schließen, wollen wir die *Close*-Methode verwenden. Geben Sie nun die ersten Buchstaben von *Close* weiter ein, springt der Auswahlbalken irgendwann auf die gesuchte Methode (siehe Abbildung 5.31).

HINWEIS Sie werden feststellen, dass IntelliSense Ihnen nicht nur eine vollständige Elementliste für das Objekt liefert, sondern auch eine Kurzbeschreibung des jeweils ausgewählten Elements als Tooltip – ebenfalls in Abbildung 5.31 zu erkennen.

5. Sie brauchen den Methodennamen nun gar nicht weiter einzugeben. Sobald der richtige Methodename markiert ist, drücken Sie einfach – der Codeeditor fügt die restlichen Buchstaben des Methodennamens dann automatisch ein und stellt den Cursor in die nächste Zeile.

TIPP Möchten Sie hinter dem Methodennamen keine neue Zeile beginnen, drücken Sie **Strg** . Damit vervollständigen Sie lediglich den Methodennamen. Oder Leertaste, dann erscheint die Methode gefolgt von einem Leerschritt.

Die erste Funktionalität des Programms haben Sie damit schon implementiert: Starten Sie das Programm testweise mit **F5**, und probieren Sie die *OK*-Schaltfläche aus!

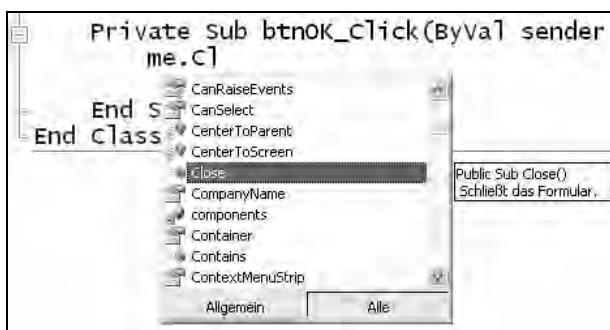


Abbildung 5.31 Der Punkt nach einem Objektnamen öffnet dank IntelliSense die Vervollständigungsliste, die eine Übersicht liefert, welche Elemente für das Objekt zur Anwendung kommen können

Filtern von Elementen in der Vervollständigungsliste

Wie Sie in Abbildung 5.31 erkennen können, verfügt die von IntelliSense gezeigte Elementliste über zwei Registerzüge, mit denen Sie die Elemente nach Wichtigkeit filtern können. Welche Elemente dabei »wichtig« sind, bestimmt Microsoft – schweigt sich aber über das Selektionsverfahren aus.

Zitat der Online-Hilfe: »Auf der standardmäßig aktivierten Registerkarte *Allgemein* werden Elemente angezeigt, die am häufigsten zum Vervollständigen der geschriebenen Anweisung verwendet werden. Auf der Registerkarte *Alle* sind alle für die automatische Vervollständigung verfügbaren Elemente aufgeführt, einschließlich der Elemente auf der Registerkarte *Allgemein*.«

Anzeigen der Parameterinfo von Elementen

1. Für das nächste IntelliSense-Feature werden wir die Funktion »Eingaben löschen« implementieren.
2. Wechseln Sie mit **Strg** + **←** erneut in die Designerdarstellung des Formulars *Form1*.
3. Doppelklicken Sie auf die Schaltfläche *Eingaben löschen*, um den Funktionsrumpf für die Ereignisbehandlungsroutine dieser Schaltfläche einzufügen.
4. Beginnen Sie einzugeben:

```
Dim locDr As DialogResult
```

Sie werden feststellen, dass Ihnen IntelliSense nach dem Schreiben des Schlüsselworts *As* auch wieder die Vervollständigungsliste anbietet. Tippen Sie soviel vom Wort »*DialogResult*«, bis diese Enumeration in der Liste erscheint und markiert ist. Drücken Sie anschließend **←**.

5. Schreiben Sie in die nächste Zeile

```
locDr =
```

Auch hier wird IntelliSense wieder aktiv. Diesmal zeigt es Ihnen alle möglichen Member der Enumeration *DialogResult* an, da es davon ausgeht, dass Sie der Variablen *locDr* eines ihrer Elemente zuweisen wollen. Ignorieren Sie die Liste aber einfach in diesem Fall und schreiben Sie weiter (das Show brauchen Sie dabei dank Vervollständigungsfunktion auch nicht komplett selbst zu schreiben).

```
MessageBox.Show(
```

In dem Moment, in dem Sie die Klammer tippen, zeigt Ihnen IntelliSense eine vollständige Parameterinfo der `MessageBox.Show`-Methode, mit der Sie übrigens ein Meldungsfeld auf dem Bildschirm darstellen können.



Abbildung 5.32 Bei Methoden, die über Parameter verfügen, zeigt IntelliSense alle Parameter mitsamt Erklärungen der Parameter an. Der jeweils aktuelle Parameter, den Sie gerade eingegeben, ist in Fettschrift gekennzeichnet.

Mehrzeilige Befehlszeilen und die Parameterinfo

Viele von Ihnen wissen bereits aus VB6-Zeiten oder von Erfahrungen mit VBScript, dass Sie eine logische Codezeile in Visual Basic der besseren Übersicht wegen mithilfe des Underscore-Zeichens (»_«) auf mehreren physischen Zeilen im Editor verteilen können. Am Ende der (physischen) Zeile fügen Sie dazu ein Leerzeichen, gefolgt vom Underscore-Zeichen, ein. Die eigentliche (logische) Zeile schreiben Sie dann ganz normal weiter, als hätten Sie nie einen Zeilenumbruch eingefügt.

Um beim Beispiel zu bleiben:

- Ergänzen Sie die Zeile (die ja noch nicht vollständig eingegeben wurde), um

```
"Sind Sie sicher?", "Eingaben löschen?", _
```

und drücken anschließend .

Nach dem Zeilenumbruch ist die Parameterinfo verschwunden. Um die Parameterinfo auch in der neuen Zeile wieder darzustellen, drücken Sie einfach .

- Geben sie nun den restlichen Teil der logischen Befehlszeile ein.

```
MessageBoxButtons.YesNo, _  
MessageBoxIcon.Question, MessageBoxButton.Default.Button2)
```

Automatische Vervollständigung von Struktur-Schlüsselworten und Codeeinrückung

- Für die Komplettierung der Methode zum Löschen der Eingabefelder geben Sie bitte die folgende Zeile ein:

```
If locDr = Windows.Forms.DialogResult.Yes Then
```

Sobald Sie nach dem Eingeben dieser Zeile betätigt haben, fügt der Editor automatisch ein entsprechendes `End If` zwei Zeilen darunter ein und platziert die Schreibmarke zwischen den beiden Zeilen.

2. Wenn Sie nun die restlichen Zeilen

```
txtKurzbeschreibung.Text = ""
txtNameDesFilms.Text = ""
txtSchauspieler.Text = ""
picCoverbild.Image = Nothing
myBilddateiname = ""
```

dazwischen eingeben, werden Sie feststellen, dass egal in welcher Spalte Sie zu tippen beginnen, die Zeilen sich immer der von If/End If vorgegebenen Struktur anpassen und entsprechend eingerückt formatiert werden.

HINWEIS Das funktioniert auch bei geschachtelten Strukturen; Sie behalten auf diese Weise immer den Überblick, in welcher Strukturverschachtelungsebene Sie sich gerade befinden.

Fehlererkennung im Codeeditor

Visual Basic verfügt über einen so genannten Hintergrund-Compiler (*Background Compiler*). Dieser Hintergrund-Compiler leistet einiges an Vorarbeit für den eigentlichen Compiler, der ja erst dann aktiv wird, wenn Sie ein Projekt erstellen (und das führt dazu, dass Anwendungen, die Sie in Visual Basic entwickeln, wesentlich kürzere Turn-Around-Zeiten⁶ aufweisen, als die, die Sie in anderen .NET-Sprachen entwickeln). Dieser Hintergrund-Compiler spart Ihnen eine ganze Menge Zeit beim Entwickeln, denn im Gegensatz zu anderen Sprachen wie C++ oder C# (oder auch zum alten VB6) entdeckt der Hintergrund-Compiler syntaktische Fehler bereits nachdem Sie eine Codezeile vollständig eingegeben haben.⁷

Einfache Fehlerkennzeichnung im Editor

Wenn Sie die letzten Änderungen am Code aufmerksam nachvollzogen haben, bemerkten Sie sicherlich einen Fehler in der letzten Zeile, die Sie eingegeben haben. Dieser Fehler war auch gar nicht schwer zu bemerken, denn schließlich hat der Codeeditor diese Zeile gekennzeichnet wie in Abbildung 5.33 zu sehen.

```
If locDr = Windows.Forms.DialogResult.OK Then
    txtKurzbeschreibung.Text = "Das ist ein toller Film"
    txtNameDesFilms.Text = "Der Name des Films"
    txtSchauspieler.Text = "Die Schauspieler"
    picCoverbild.Image = Nothing
    myBilddateiname = "C:\temp\MyImage.jpg"
End Sub
```

Der Name "myBilddateiname" wurde nicht deklariert.

Abbildung 5.33 Fehler, die der Hintergrundcompiler feststellt, werden direkt im Editor noch vor dem eigentlichen Kompilierungsvorgang beim Erstellen des Projektes gekennzeichnet

⁶ Als »Turn-Around-Zeit« (etwa: »Wenden-Zeit«, wie das Wenden beim Autofahren) bezeichnet man die Zeitspanne, die vom Starten des Compilers über das Kompilieren eines Projektes bis zum eigentlichen Anwendungsstart vergeht.

⁷ Hintergrundcompiler gibt es zwar auch in diesen Sprachen, aber die sind lange nicht so konsequent wie in Visual Basic .NET bzw. Visual Basic 2008, und bei ihnen passiert es oft, dass der eigentliche Compiler erst beim Erstellen des Projekts syntaktische Fehler oder nicht deklarierte Variablen findet. In Visual Basic .NET, 2005 bzw. 2008 passiert das äußerst selten – quasi nie.

Diese betroffene Variable, die wir später im gesamten Formular für die Speicherung des Dateinamens benötigen, wurde nicht deklariert. Also machen wir das als nächstes.

Editorunterstützung bei Fehlern zur Laufzeit

Direkt unterhalb der Klassendefinition fügen Sie die im Folgenden in Fettschrift formatierte Zeile ein:

```
Public Class Form1

    Dim myBilddateiname As Integer
```

Sie sehen, dass der Fehler nun nicht mehr durch Unterschlängelung markiert ist.⁸

Nun probieren wir das Programm in seinem derzeitigen Zustand aus.

- Starten Sie das Programm mit **[F5]**.
- Geben Sie ein paar Zeilen in die Eingabefelder an.
- Klicken Sie auf *Eingabe löschen*.
- Bestätigen Sie das Meldungsfeld mit *Ja*.

Statt die Eingabefelder zu löschen, bricht das Programm mit folgender Meldung ab (Abbildung 5.34):

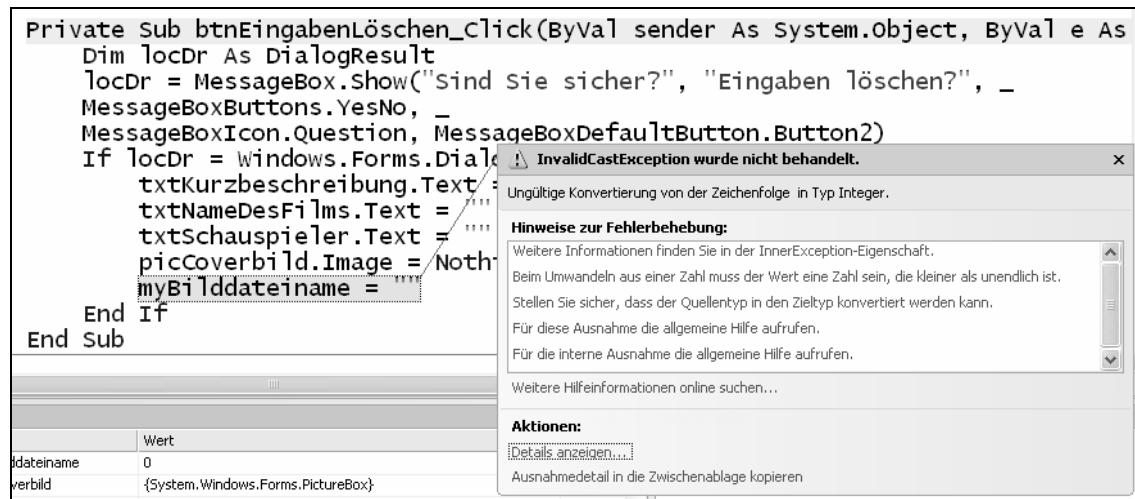


Abbildung 5.34 Tritt während der Anwendungsausführung im Debug-Modus ein Fehler (eine so genannte »Ausnahme« – engl.: Exception) auf, ruft Visual Studio den Editor auf, unterlegt die betroffene Stelle gelb und zeigt einen entsprechenden Hinweis an

⁸ Kleiner Hinweis am Rande: Bevor Sie nun eine E-Mail schreiben, da Sie glauben, einen Fehler gefunden zu haben – warten Sie erst die nächsten Absätze ab! ;-)

Fehler, die zur Laufzeit in einer .NET-Anwendung auftreten und dafür verantwortlich sind, dass ein Programm nicht fortgesetzt werden kann, werden als Ausnahmen (engl: *Exception* – sprich: »Ixäpschen«) bezeichnet. In diesem Fall ist uns ein Fehler beim Deklarieren der Variablen `myBilddateiname` passiert,⁹ der schließlich zu dieser Ausnahme geführt hat. .NET versuchte zur Laufzeit, den Leerstring der Variablen zuzuweisen, die aber dummerweise als Integervariable deklariert wurde – das führte zu einer `InvalidCastException` (etwa: *Ausnahme wegen ungültiger Typkonvertierung*).

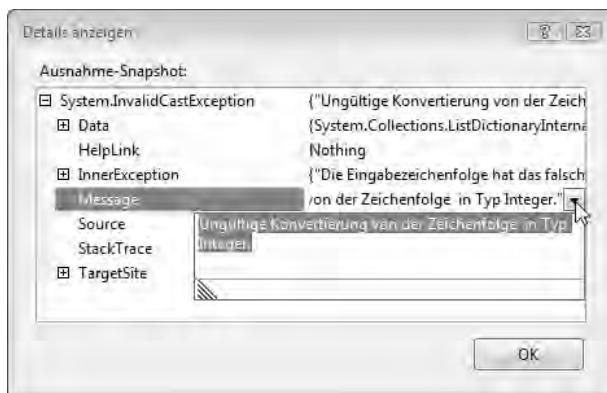


Abbildung 5.35 Mithilfe des Ausnahmedetail-Dialogs erfahren Sie Genaueres über die Umstände, die zur Ausnahme führten

Falls Sie in einem solchen Fall genauere Information zum Ausnahmenu stand benötigen, klicken Sie unten im Dialog unter *Aktionen* auf *Details anzeigen...*. Der Editor zeigt Ihnen anschließend einen weiteren Dialog (siehe Abbildung 5.35), mit dem Sie diese erweiterten Informationen abrufen können.

3. Für das weitere Nachvollziehen des Beispiels klicken Sie im Dialog auf *OK*.
4. Schließen Sie den darunter liegenden Dialog mit einem Mausklick auf die Schließschaltfläche in der rechten oberen Ecke.
5. Beenden Sie das Debuggen: Wählen Sie dazu aus dem Menü *Debuggen* den Befehl *Debuggen beenden* oder klicken Sie in der Symbolleiste auf das entsprechende Symbol zum Beenden des Debuggens.

Fehlerverbesserungsvorschläge des Editors bei Typkonflikten – erzwungene Typsicherheit

Passiert wäre das nicht, wenn wir in unserer Anwendung von vornherein Typsicherheit erzwungen hätten. In diesem Fall hätten wir den Fehler bereits gemerkt, noch bevor wir das Programm gestartet hätten – der Editor hätte uns darauf aufmerksam gemacht.

Seit Visual Basic .NET 2002 kennt Visual Basic die Anweisung `Option Strict [On|Off]`. Schalten Sie `Option Strict` mit `On` ein, sorgt schon der Hintergrund-Compiler dafür, dass Sie nur gleiche Typen zuweisen können – oder optional strikt dafür sorgen, dass eine saubere Typkonvertierung (beispielsweise von `Integer` zu `String`) stattfindet.

⁹ O.K., o.k. – mir, nicht Ihnen.

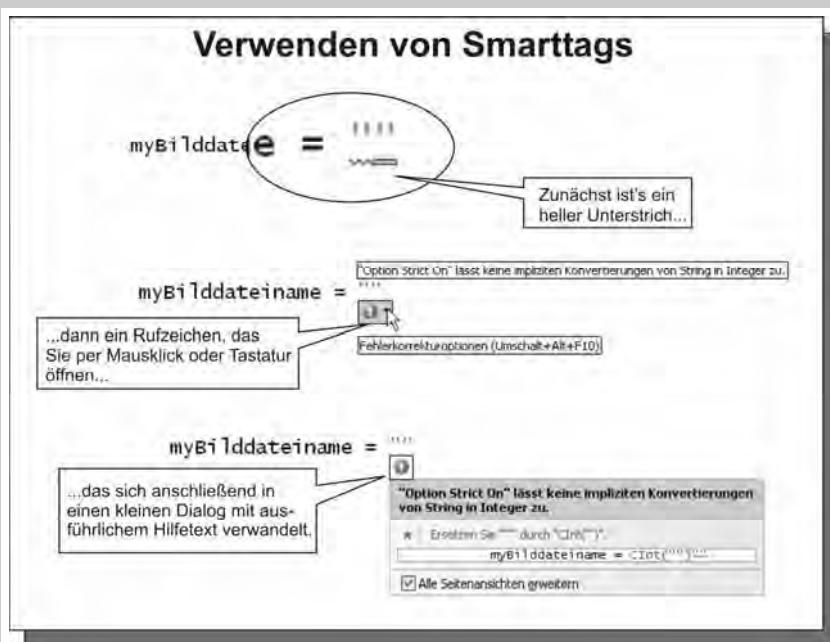
Sobald Sie die Anweisung

Option Strict on

ganz oben in den Code einfügen (noch vor der Klassendefinitionsanweisung Class Form1) und anschließend zurück zur betroffenen Zeile scrollen, sehen Sie, dass nicht nur der vermutlich zur Ausnahme führende Teil unterschlängelt ist, sondern am Ende auch einen gelben Balken aufweist.

Smarttags im Editor von Visual Basic

Einen Smarttag erkennen Sie zunächst als kleinen, hellen Unterstrich in einer Codezeile im Visual Basic-Editor, an der es seiner Meinung nach irgendetwas zu verbessern oder anzumerken gibt.



Fahren Sie mit dem Mauszeiger auf diesen Unterstrich, verwandelt er sich in ein kleines Symbol mit der Form eines Ausrufezeichens, das Ihnen verschiedene Arten von Unterstützung anbietet. Smarttags können dabei ganz unterschiedliche Formen zusätzlicher Unterstützung anbieten – nicht nur Hilfestellungen bei Typkonflikten geben, wie hier im Beispiel zu sehen.

Autokorrektur für intelligentes Kompilieren

In vielen Fällen verbirgt sich hinter dem Smarttag ein Dialog, der Ihnen einen Korrekturvorschlag für den erkannten »Fehler« unterbreitet – etwa, wie in der Abbildung zu sehen. Solche Hilfedialoge, die sich hinter Smarttags verbergen, nennt Microsoft übrigens »Autokorrektur für intelligentes Kompilieren«. Falls die Autokorrektur für intelligentes Kompilieren Recht behält, und es sich tatsächlich um den vermuteten Fehler handelt, brauchen Sie noch nicht einmal die Tastatur zu bemühen: Klicken Sie einfach auf den blauen Korrekturvorschlag, und der Editor nimmt die Korrektur im Programmcode vor.

WICHTIG In unserem Beispiel greift die Autokorrektur für intelligentes Kompilieren an der Stelle des Fehlers leider nicht, weil es sich um einen Folgefehler handelt. Aber Sie sehen, dass wir auf jeden Fall einen kompletten Turn-Around-Lauf gespart hätten, wenn Option Strict von vornherein eingeschaltet gewesen wäre. Aus diesem Grund sollten Sie Option Strict projektweit einschalten und sich die Anweisung vor jeder Codedatei sparen. Außerdem sollten Sie die Visual Studio-Optionen so einstellen, dass Option Strict grundsätzlich beim Anlegen jedes neuen Projektes eingeschaltet ist. Wie das geht, zeigt der folgende graue Kasten.

Um den Fehler zu beheben, ändern Sie die Zeile

```
Private myBilddateiname As Integer
```

einfach in

```
Private myBilddateiname As String
```

Erzwungene Typsicherheit (Option Strict) projektweit einstellen

Sie können dafür sorgen, dass Typsicherheit grundsätzlich in einem Projekt erzwungen wird, ohne dass Sie dazu Option Strict On über jede Codedatei schreiben müssen. Dazu öffnen Sie im Projektmappen-Explorer das Kontextmenü des entsprechenden Projekts (nicht der Projektmappe!) und wählen *Eigenschaften*. Die Projekteigenschaften öffnen sich nun als Dokumentenfester in der Dokumentenregisterkartengruppe, und Sie können auf der Registerkarte *Kompilieren* Option Strict global mit *On* einschalten.

Erzwungene Typsicherheit für alle folgenden neuen Projekte

Möchten Sie, dass diese Einstellung grundsätzlich gilt, wenn Sie ein neues Projekt anlegen, müssen Sie den Optionsdialog von Visual Studio bemühen. Rufen Sie ihn mit *Extras | Optionen* auf, und wählen Sie den Bereich *Projekte und Projektmappen*. Auf der Registerkarte *VB-Standard* nehmen Sie die Einstellungen für Option Strict vor.

XML-Dokumentationskommentare für IntelliSense bei eigenen Objekten und Klassen

Wie Ihnen IntelliSense beim Finden der richtigen Klassen, Objekte, Methoden und anderer Elemente bei der Entwicklung von Anwendungen helfen kann, haben Sie bereits kennen gelernt. Doch damit ist noch lange nicht das Ende der Fahnenstange erreicht.

Zur Demonstration müssen wir ein wenig mehr vorbereitenden Aufwand betreiben. Dazu implementieren wir im Folgenden eine Methode, die eine Bilddatei aus einer Datei in ein Image-Objekt lädt. Auch hier soll die eigentliche Funktionsweise nicht von primärem Interesse sein – es geht schließlich immer noch um die Erarbeitung der Funktionalitäten der Codeeditor-Fähigkeit.

1. Fügen Sie zu diesem Zweck die folgenden Zeilen in den Klassencode von *Form1* ein:

```
Function CoverbildAusDateinamen(ByVal CoverbildDateiname As String) As Image
    Dim locImage As Image
    If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
        locImage = Image.FromFile(CoverbildDateiname)
```

```

        Return locImage
    End If
    Return Nothing
End Function

```

2. Wechseln Sie mit **[F7]** zum Entwurfsmodus (zum Designer-Dokumentenfenster). Übrigens: Mit **[F7]** wechseln Sie zwischen Designer- und Codedarstellung eines Formulars.
3. Doppelklicken Sie auf die Auslassungsschaltfläche neben der PictureBox, um den Codeeditor zu öffnen und die Ereignisbehandlungsroutine für diese Schaltfläche zu bearbeiten.
4. Fügen Sie in die Stub¹⁰ (in den Funktionsrumpf, also zwischen Private Sub btnCoverbildWählen ... und End Sub) folgende Zeilen ein:

```

Dim locOfd As New OpenFileDialog

With locOfd
    locOfd.CheckFileExists = True
    locOfd.DefaultExt = "*.bmp"
    locOfd.Filter = "JPeg-Bilder (*.jpg)|*.jpg|Windows Bitmap (*.bmp)|*.bmp|Alle Dateien
(*.*)|*.*"

    Dim locDr As DialogResult = locOfd.ShowDialog()
    If locDr = Windows.Forms.DialogResult.Cancel Then
        Return
    End If

    myBilddateiname = locOfd.FileName
End With

```

5. Zwischen den letzten beiden Zeilen ergänzen Sie nun eine weitere Zeile, die Sie bitte noch nicht komplett eingeben:

```
picCoverbild.Image = CoverbildAusDateinamen(
```

Sobald Sie die Klammer getippt haben, sehen Sie, dass IntelliSense auch bei selbst geschriebenen Methoden (und anderen Elementen wie Eigenschaften, etc.) greift. Doch schauen Sie sich Abbildung 5.36 an. Fällt Ihnen was auf?

myBilddateiname = locOfd.FileName picCoverbild.Image = CoverbildAusDateinamen() End With End Sub	CoverbildAusDateinamen (CoverbildDateiname As String) As System.Drawing.Image
---	--

Abbildung 5.36 Auch bei selbst geschriebenen Methoden funktioniert IntelliSense – natürlich fehlen dabei zunächst noch die Erklärungen

¹⁰ Sprich: »Stap«.

Genau wie bei eingebauten Methoden wird IntelliSense zwar aktiv – doch die sonst so hilfreichen Erklärungen finden wir hier natürlich nicht. Woher sollen sie auch stammen! Noch in Visual Basic 2003 war ohne Tools von Drittherstellern in Sachen IntelliSense bei selbst entwickelten Methoden an dieser Stelle Schluss. Doch mit Visual Basic 2008 ist das anders, wie Sie gleich sehen werden:

1. Bevor Sie nämlich die Zeile nun komplettieren, positionieren Sie die Schreibmarke oberhalb der Zeile

```
Function CoverbildAusDateinamen(ByVal CoverbildDateiname As String) As Image
```

2. Tippen Sie zwei Hochkomma (**[;** **[#]**).
3. Sobald Sie das dritte Hochkomma getippt haben, wird der Codeeditor aktiv und greift Ihnen unter die Arme.

```
''' <summary>
''' |
''' </summary>
''' <param name="CoverbildDateiname"></param>
''' <returns></returns>
''' <remarks></remarks>
Function CoverbildAusDateinamen(ByVal CoverbildDateiname As String) As Image
    Dim locImage As Image
    If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> ""
        locImage = Image.FromFile(CoverbildDateiname)
        Return locImage
    End If
    Return Nothing
End Function
```

Abbildung 5.37 Der Editor fügt ein XML-Skelett über der Methode ein, in der Sie ihre Dokumentation nur zu ergänzen brauchen

4. Für die folgenden Schritte orientieren Sie sich auch an Abbildung 5.38. Geben Sie zwischen `<summary>` und `</summary>` eine Funktionsbeschreibung ein.

```
''' <summary>
''' Lädt ein Bild, so vorhanden, aus einer Datei und liefert das Bild als Image (oder Nothing) zurück.
''' </summary>
''' <param name="CoverbildDateiname">Name der Coverbild-Datei.</param>
''' <returns>Image-Objekt, das das Bild der angegebenen Datei enthält.</returns>
''' <remarks>Erstellt von Klaus Löffelmann am 26.10.2005</remarks>
Function CoverbildAusDateinamen(ByVal CoverbildDateiname As String) As Image
    Dim locImage As Image
    If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
        locImage = Image.FromFile(CoverbildDateiname)
        Return locImage
    End If
    Return Nothing
End Function
```

Abbildung 5.38 Komplettieren Sie die XML-Tags etwa nach dieser Vorlage. Denken Sie beim mehrzeiligen Verteilen eines Textes an die Leerzeichen (blaues Kästchen in der Abbildung).

HINWEIS Denken Sie daran, dass Sie beim Verteilen von Funktionsbeschreibungen über mehrere Zeilen ein Leerzeichen entweder *hinter* das letzte Wort der vorherigen oder *vor* das erste Wort der nächsten Zeile setzen, damit die Wörter bei der späteren Anzeige als Tooltip nicht zusammenlaufen.

5. Zwischen `<param name="CoverbildDateiname">` und `</param>` geben Sie die Bedeutung des Parameters *CoverbildDateiname* an.

HINWEIS Sollten Sie es zu einem späteren Zeitpunkt mit Methoden oder Eigenschaften zu tun haben, die mehrere Parameter entgegennehmen, werden an dieser Stelle weitere `param`-Tags aufgelistet, zwischen denen Sie die Beschreibungen der weiteren Parameter platzieren können.

6. Zwischen `<returns>` und `</returns>` geben Sie die Bedeutung des Rückgabewertes an.

7. Optional geben Sie zwischen `<remarks>` und `</remarks>` eine Bemerkung ein.

HINWEIS Außer den hier von uns benutzten Tags wie `<returns>` und `<remark>` sind noch weitere möglich. Schauen Sie in die Hilfe unter dem Stichwort »Empfohlene XML-Tags für Dokumentationskommentare« nach.

8. Wenn Sie alle Eingaben abgeschlossen haben, kehren Sie zur noch nicht vollständig eingegebenen Zeile

```
picCoverbild.Image = CoverbildAusDateinamen(
```

zurück. Löschen Sie alle Zeichen, bis nur noch »Cov« von »CoverbildAusDateinamen(« übrig bleibt, und drücken Sie anschließend **Strg** **↓**. Sie sehen nun, dass die Funktion nicht nur in der Vervollständigungsliste zu sehen ist (das war sie zuvor auch schon), sondern nunmehr auch die Funktionsbeschreibung enthält, die Sie mithilfe der XML-Tags angegeben haben (siehe Abbildung 5.39).



Abbildung 5.39 Wenn Sie Ihre Methoden- und Eigenschaftenprozeduren mit entsprechenden XML-Tags versehen haben, zeigt IntelliSense sowohl eine Funktionsbeschreibung ...

9. Drücken Sie **Strg** **↓**, um den Funktionsnamen zu vervollständigen.

10. Tippen Sie die Klammer, wird IntelliSense wieder aktiv (siehe Abbildung 5.40). Sie sehen, dass IntelliSense nun nicht nur den Parameternamen und den Parametertyp, sondern auch die Parameterdokumentation zeigt.

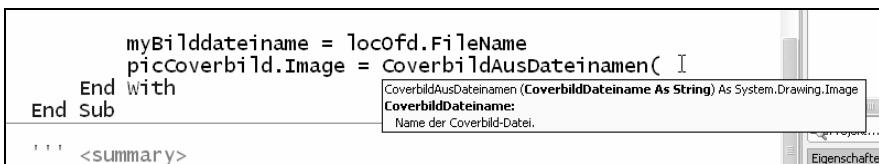


Abbildung 5.40 ... als auch entsprechende Parametererklärungen an

11. Vervollständigen Sie die Zeile, sodass diese komplett wie folgt lautet:

```
picCoverbild.Image = CoverbildAusDateinamen(myBilddateiname)
```

HINWEIS Übrigens: So ganz nebenbei haben Sie mit nur ein paar Zeilen Code die komplette Bilddarstellung im Formular implementiert – und dank der Einstellungen, die Sie zuvor im Designer vorgenommen haben, läuft die Bildausschnittseinstellung mit Rollbalken ebenfalls schon. Den Beweis dazu können Sie selbst antreten: Starten Sie das Programm mit **F5**, klicken Sie auf die Auslassungsschaltfläche und wählen Sie im Dialog, der jetzt gezeigt wird, eine Bilddatei aus.

Hinzufügen neuer Codedateien zum Projekt

Für unser Beispiel benötigen wir eine Datenstruktur, mit der die Eingaben, die der Anwender im Hauptformular getätigt hat, an das Druckformular übergeben werden.

Das Programm verwendet also diese Datenstruktur in diesem Beispiel, um die einzelnen Datenfelder in dieser Datenstruktur – nennen wir Sie *CoverInhalt* – zunächst zwischenzuspeichern, und sie dann in einem Rutsch an das Druckformular zu übergeben.

Auf diese Weise sind die beiden Aufgaben – Datenerfassung und Drucken – sauber voneinander getrennt. Der Hauptdialog sorgt in eigener Regie für die Datenerfassung, der Druckdialog selbstständig für das Drucken der Daten. Der Hauptdialog muss so nicht auf den Druckdialog direkt zugreifen, und dort irgendwelche Variablen manipulieren, sondern sagt dem Druckdialog nur mithilfe einer einzelnen öffentlichen Funktion, wie dieser das Cover in der Vorschau darstellen oder auf einem Drucker ausdrucken soll.

Eleganterweise bringen wir diese Datenstruktur in einer eigenen Codedatei unter: Der Klassendatei, die den gleichen Namen wie die Datenstruktur (die Klasse) selbst bekommen soll: *CoverInhalt*.

HINWEIS Visual Basic 6 Programmierer kennen Datenstrukturen der einfachsten Ausführung in Form von benutzerdefinierten Typen. Ohne den Klassen- oder Umstiegsteil des Buches vorwegzunehmen: Die einfachste vorstellbare Klasse, wie Sie sie gleich kennen lernen werden, entspricht in etwa der Definition eines neuen Typen mit Type.

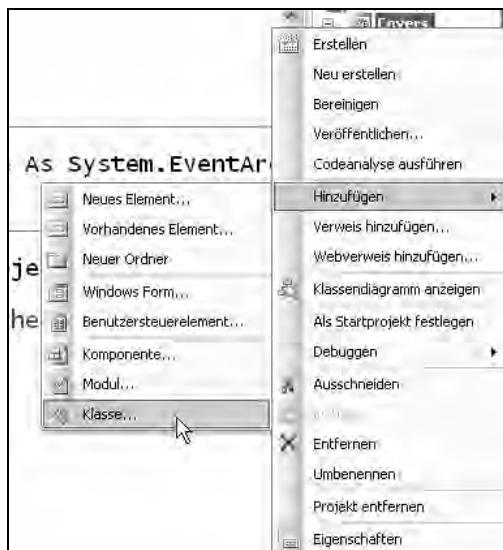


Abbildung 5.41 So fügen Sie eine neue Klassendatei zum Projekt hinzu

- Um eine neue Klassendatei zu einem Projekt hinzuzufügen, verfahren Sie auf fast genau dieselbe Art und Weise, wie Sie es beim Hinzufügen einer Formular-Klassendatei schon getan haben. Rufen Sie das Kontextmenü des Projektes **Covers** (nicht der Projektmappe!) im Projektexplorer auf.
- Wählen Sie **Hinzufügen | Klasse**.
- Im jetzt erscheinenden Dialog geben Sie den Namen der Klassendatei (ohne Dateiendung) ein – für unser Beispiel **CoverInhalt**.
- Klicken Sie auf **OK** oder drücken Sie **Enter**.

Der Codeeditor wird geöffnet; die Klassendefinition ist bereits vorgegeben. Geben Sie nun zwischen **Public Class CoverInhalt** und **End Class** die folgenden Zeilen ein:

```
Public FilmTitel As String
Public Schauspieler As String
Public Beschreibung As String
Public Coverbild As Image
```

Mit der Einführung dieser neuen Klasse haben wir die Möglichkeit, die Daten aus dem Hauptformular auszulesen und dem Druckformular zu übergeben. Genau das werden wir als nächstes implementieren.

TIPP Damit dieser Abschnitt für Sie nicht in ein endloses Getippe ausartet, werden wir es uns mit dem Code für das Drucken einfach machen, und diesen aus einer Textdatei in das Formular hineinkopieren. Sie finden sie deswegen als reine Textdatei im Verzeichnis **.\VB 2008 - Entwicklerbuch\B - IDE\03 - Covers\Druckroutine für Covers.txt**.

- Öffnen Sie die Textdatei **\B - Ein- und Umstieg\Druckroutine für Covers.txt** mit dem Notepad von Windows.
- Drücken Sie **Strg A** (alles markieren), **Strg C** (in die Zwischenablage kopieren).
- Schließen Sie das Notepad.

4. Wechseln Sie zurück zu Visual Studio.
5. Klicken Sie im Projektmappen-Explorer auf *Form2.vb* und anschließend in der Symbolleiste des Projektmappen-Explorers auf das Symbol *Code anzeigen*.
6. Drücken Sie **Strg A** (*alles markieren*), **Strg V** (*Zwischenablageinhalt einfügen*). Damit ist der Code für das Druckformular vollständig. Die genaue Funktionalität soll uns an dieser Stelle noch nicht interessieren; es würde bedeuten, zu viele Themen vorwegnehmen zu müssen, deren ausführliche Erklärung für ein genaueres Verständnis erforderlich wäre.
7. Speichern Sie alle Änderungen und schließen Sie das Dokument *Form2.vb*.

Um den Code zu implementieren, der eine neue Instanz der Klasse *CoverInhalt* bildet, sie mit Daten aus den Eingabefeldern füttert und die Klasse zur Weiterverarbeitung dem Druckformular übergibt, verfahren Sie wie folgt:

1. Wechseln Sie mit **Strg F1** zum Entwurfsmodus von *Form1.vb* (also zu *Form1.vb [Entwurf]*).
2. Doppelklicken Sie auf die Schaltfläche *Inlay drucken*, um dafür zu sorgen, dass die Stub für die Ereignisbehandlungsroutine von *btnInlayDrucken_Click()* im Code eingefügt wird.
3. Geben Sie den folgenden Code ein (die Kommentare dienen nur zur Groberklärung des Codes und müssen natürlich nicht mit eingegeben werden).

```
'CoverInhalt-Klasse in ein Objekt instanzieren
Dim locCoverInhalt As New CoverInhalt

'Dem CoverInhalt-Objekt die Daten zuordnen
locCoverInhalt.FilmTitel = txtNameDesFilms.Text
locCoverInhalt.Schauspieler = txtSchauspieler.Text
locCoverInhalt.Beschreibung = txtKurzbeschreibung.Text
locCoverInhalt.Coverbild = picCoverbild.Image

'Die Druckvorschau aufrufen, und das CoverInhalt-Objekt
'mit den Daten übergeben.
Dim locCoverDruckenForm As New Form2
locCoverDruckenForm.DialogDarstellen(locCoverInhalt)
```

Code umgestalten (Refactoring)

Visual C# 2008 wurde leider in viel größerem Umfang mit Refactoring-Werkzeugen bedacht als Visual Basic 2008.¹¹ Im Grunde genommen gibt es in Visual Basic 2008 nur eine einzige Funktion, die das automatische Umgestalten von Code erlaubt – das Umbenennen von Namen von Methoden, Eigenschaften, Objekten oder Ereignissen. Doch dazu später mehr.

Was bedeutet Refactoring genau?

Stellen Sie sich vor, Sie entwickeln eine relativ umfangreiche Funktion, und Sie stellen nach einer Weile fest, dass nur diese eine Funktion bereits aus 1000 Zeilen Code besteht. Sie müssen (oder sollten zumindest) sich dann eingestehen, dass Sie das Problem, das Sie lösen wollen, besser auf mehrere Unterfunktionen verteilt

¹¹ Es gibt allerdings eine kostenfreie Version des Addins Refactor!, das Sie sich unter dem IntelliLink **A0502** herunterladen können.

hätten. Aber dazu ist es ja nicht zu spät. Wenn Sie aus einem Teil der Funktion nun eine neue Unterfunktion generieren, dann betreiben Sie bereits aktives Refactoring.

Am schlimmsten ist es bei solchen Aktionen, wenn sich Namen, die Sie beispielsweise Methoden gegeben haben, als falsch oder nicht ausreichend aussagekräftig entpuppen. Gerade wenn Sie im Team arbeiten, sind Sie auf dieses Problem sicherlich schon das ein oder andere Mal gestoßen. Und jeder, der eine Funktion aus solchen Gründen umbenennen musste, weiß, was die Umbenennerei für eine Arbeit macht, weil die Funktion doch viel öfter referenziert wird, als man es eigentlich erwartet hätte.

Suchen & Ersetzen kommt für diesen Zweck auch nicht in Frage, denn stellen Sie sich vor: Sie möchten eine Eigenschaft namens »bindControl« in »BoundControl« umbenennen. Es gibt aber auch eine Methode namens »UnbindControl«; an die Sie aber überhaupt nicht mehr denken, und die Sie aus Versehen und ohne es zu merken, ebenfalls umbenennen, nämlich in »UnboundControl«.

Noch ernster wird es bei der Umbenennung von Variablen, die Sie in verschiedenen Gültigkeitsbereichen mehrfach verwenden. Wenn Sie beispielsweise eine Methode in zwei Klassen implementiert haben, dann möchten Sie unter Umständen, dass sich nur der Name der Methode in einer Klasse ändert – und überall dort, wo Sie ihn verwendet haben. Mit Suchen & Ersetzen wäre das fehlerfrei zu tun fast ein Ding der Unmöglichkeit.

Hier kommt das einzige Refactoring-Werkzeug von Visual Basic 2008 ins Spiel – das Umbenennen-Werkzeug. Das Umbenennen von Namen über das Refactoring bezieht sich immer nur auf das Element, das Sie umbenennen, und es benennt nicht nur das Element, sondern auch alle Referenzen um. Um das auszuprobieren, machen Sie Folgendes:

1. Suchen Sie im Code von *Form1.vb* die Funktion *CoverbildAusDateinamen*.
2. Klicken Sie mit der rechten Maustaste auf den Funktionsnamen, und aus dem Kontextmenü, das der Editor nun öffnet, wählen Sie *Umbenennen...*
3. Im Dialog, der jetzt dargestellt wird, geben Sie einen neuen Namen für die Funktion ein – beispielsweise *GetCoverImage*.

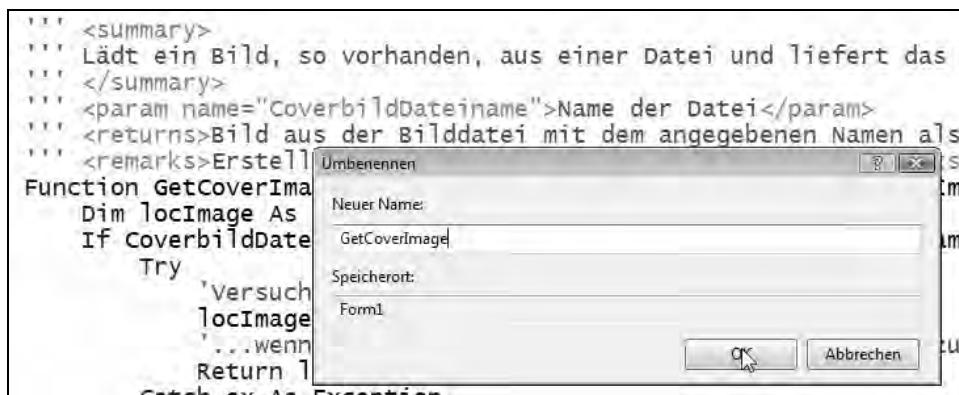


Abbildung 5.42 Mit dem Umbenennen beispielsweise einer Methode (einer Funktion) ändern Sie nicht nur deren Namen, sondern auch alle ihre Referenzen

4. Klicken Sie auf *OK*.
5. Bewegen Sie die Schreibmarke zur Funktion *btnCoverbildWählen_Click*. Sie werden hier erkennen, dass nicht nur der Name der Funktion, sondern auch die (zugegebenermaßen einzige) Referenz auf die Funktion geändert wurde, wie im Listing der Funktion fett markiert zu sehen:

```
Private Sub btnCoverbildWählen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnCoverbildWählen.Click
    Dim locOfd As New OpenFileDialog

    With locOfd
        locOfd.CheckFileExists = True
        locOfd.DefaultExt = "*.*"
        locOfd.Filter = "JPeg-Bilder (*.jpg)|*.jpg|Windows Bitmap (*.bmp)|*.bmp|Alle Dateien (*.*)|*.*"

        Dim locDr As DialogResult = locOfd.ShowDialog()
        If locDr = Windows.Forms.DialogResult.Cancel Then
            Return
        End If

        myBilddateiname = locOfd.FileName
        picCoverbild.Image = GetCoverImage(myBilddateiname)
    End With
End Sub
```

Code umgestalten (Klassennamen-Anpassung) beim Umbenennen von Projektdateien oder Objekteigenschaften

Solange Sie Ihre Klassendateien oder Formulardateien genauso nennen wie die Klassen oder Formularklassen, die diese speichern, funktioniert das Umbenennen hier genauso – mit dem Unterschied, dass Sie auch die physische Datei umbenennen, die den eigentlichen Klassencode enthält.

Probieren Sie es aus:

1. Öffnen Sie das Kontextmenü von *Form2.vb* im Projektmappen-Explorer.
2. Wählen Sie *Umbenennen*.
3. Geben Sie einen neuen Namen für die Formulardatei ein – beispielsweise **frmCoverDrucken.vb**.

HINWEIS Sie benennen dabei eine physische Datei um. Geben Sie deswegen die Dateiendung unbedingt mit ein!

4. Drücken Sie **Eingabe**.
5. Bewegen Sie die Schreibmarke zur Funktion *btnInlayDrucken_Click* im Code von *Form1*. Sie sehen, dass die letzten Zeilen dieser Funktion nunmehr folgendermaßen lauten:

```
'Die Druckvorschau aufrufen, und das CoverInhalt-Objekt
'mit den Daten übergeben.
Dim locCoverDruckenForm As New frmCoverDrucken
locCoverDruckenForm.DialogDarstellen(locCoverInhalt)
```

6. Woran das liegt, sehen Sie, wenn Sie den Code von »ehemals« *Form2.vb* (und nunmehr *frmCoverDrucken.vb*) öffnen. Durch das Umbenennen der Datei wurde auch der Klassenname in *frmCoverDrucken* geändert und dementsprechend die geänderte Referenz, die Sie sich gerade in der Funktion *btnInlayDrucken_Click* anschaut haben.

HINWEIS Bei umfangreichen Projekten mit vielen Formulardateien oder Klassendateien kann sich das Refactoring von Klassen- und Formularklassen durch das Umbenennen ihrer Code datei als störend erweisen. Sie können das »Dateinamen-Umbenennen-Refactoring« im *Optionen*-Dialog von Visual Studio ausschalten. Rufen Sie diesen Dialog dazu aus dem *Extras*-Menü auf, und wählen Sie das Register *Windows Forms-Designer*. Wie in Abbildung 5.43 zu sehen, setzen Sie die *EnableRefactoringOnRename*-Eigenschaft auf *False*.¹²

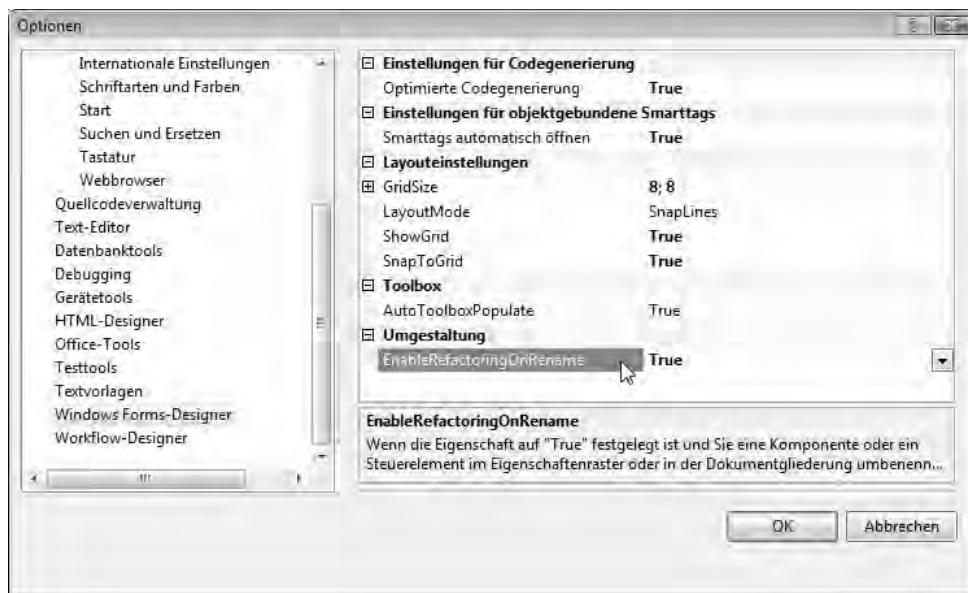


Abbildung 5.43 Hier schalten Sie das Refactoring durch Umbenennen von Codedateien aus

Die Bibliothek der Codeausschnitte (Code Snippets Library)

Unser Beispielprogramm kann sich bis zu diesem Zeitpunkt schon sehen lassen. Allerdings läuft es noch nicht wirklich fehlerfrei und ist nicht gegen »groben Unfug« geschützt. Warum? Nun, Sie könnten beispielsweise versuchen, statt einer Bilddatei eine Textdatei als Coverbild zu laden. Einen solchen Versuch quittiert das Programm direkt mit einer Ausnahme – einer Laufzeitfehlermeldung –, die es aber nicht abfängt:

¹² Die in VS 2008 enthaltenen Tools zum Refactoring gehen auf die Werkzeuge einer Firma namens Developer Express zurück. Wenn Sie das vollständige Tool zum Refactoring benutzen wollen (auch für VB.NET), können Sie dies unter dem IntelliLink **A0503** als Testversion oder kostenpflichtige Vollversion beziehen.

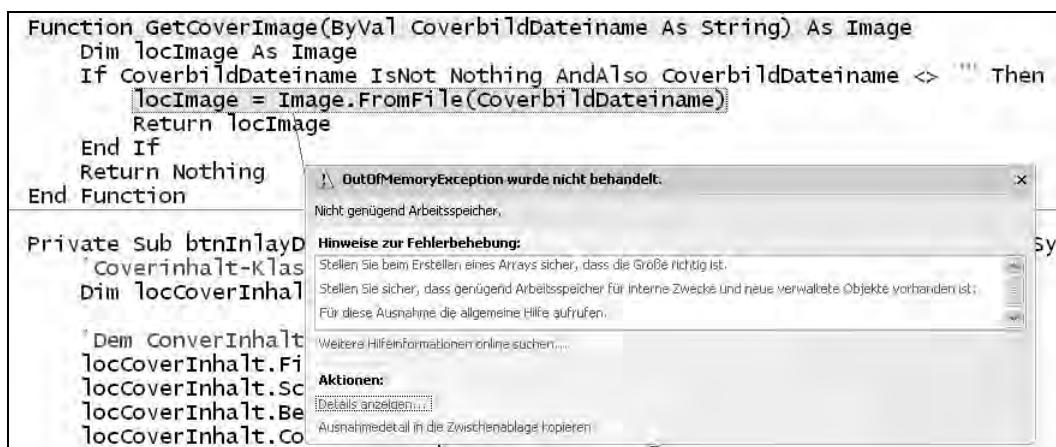


Abbildung 5.44 Das Laden einer kleinen Textdatei führt zu einer »Zu-wenig-Speicher-Ausnahme«? Das lässt Platz für Vermutungen, wird aber wohl an den internen Grafikfiltern liegen, die Textbytes fälschlicherweise als Größenangaben für eine zu ladende Grafik interpretieren. So tritt der Fehler bereits beim Speicherreservieren für die Grafik und nicht erst beim Lesen der »falschen« Bytefolgen auf.

In einem professionellen Programm darf so etwas natürlich nicht passieren – schon gar nicht, wenn die ausgelöste Ausnahme, wie hier im Beispiel, den Anwender auf eine völlig falsche Fährte lockt (siehe Bildunterschrift).

Schön wäre es überdies, wenn dieses Fehlverhalten nicht nur nicht zum Abbruch des Programms führte, sondern den Fehler einer zuständigen Stelle obendrein noch meldete! Zum Beispiel, indem es versucht, eine E-Mail an die E-Mail-Adresse eines Administrators zu schicken.

Ich würde Ihnen gerne erklären, wie das funktioniert. Aber wissen Sie was? Ich kann's nicht. Ich müsste dazu stundenlang recherchieren, und ob ich die Infos zur Programmierung eines solchen Features selbst dann überhaupt finden würde, wäre fraglich... ;-)

Aber wissen Sie noch was: Das muss ich auch gar nicht. Denn Visual Basic 2008 verfügt über eine Codeauschnittsbibliothek, die für jeden Geschmack etwas Passendes bereitstellt. Zum Beispiel, wie man einen Coderumpf zum Auffangen eines Fehlers implementiert. Und das geht so:

1. Schließen Sie zunächst den Dialog mit der Ausnahme, der immer noch auf dem Bildschirm zu sehen sein sollte.
2. Stoppen Sie das Programm mit einem Klick auf das *Debuggen beenden*-Symbol (der Tooltip des Symbols hilft Ihnen, das richtige zu finden).
3. Bewegen Sie die Schreibmarke in die Methode GetCoverImage, und zwar so, dass sie sich genau vor der Zeile

```
locImage = Image.FromFile(CoverbildDateiname)
```

befindet.

4. Rufen Sie mit der rechten Maustaste das Kontextmenü auf und wählen Sie *Ausschnitt einfügen*.
5. Sie sehen nun eine Liste mit Ordnern, die die verschiedenen Codeausschnitt-Oberbegriffe enthalten. Doppelklicken Sie auf *Allgemeine Codemuster*.

6. Doppelklicken Sie auf *Ausnahmebehandlung*.

```

Function GetCoverImage(ByVal CoverbildDateiname As String) As Image
    Dim locImage As Image
    If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname
        Ausschnitt einfügen: Allgemeine Codemuster > Ausnahmebehandlung > Dateiname)
        Return locImage
    End If
    Return Nothing
End Function

Private Sub btnInlayDrucken_Click(ByVal sender As System.Object, By

```

Abbildung 5.45 Beim Einfügen eines Codeausschnittes (Code Snippet) sehen Sie die verschiedenen Kategorien hierarchisch in blau gefärbt nebeneinander stehen. Ein Tooltip gibt Ihnen genauere Infos zum ausgewählten Ausschnitt. Achten Sie dabei auch auf die dort ausgewiesene Verknüpfungszeichenfolge für »das nächste Mal«!

7. In der Liste, die sich daraufhin öffnet, doppelklicken Sie auf *Try...Catch...End Try-Anweisung*.

HINWEIS Merken Sie sich am besten dabei gleich die Verknüpfung, die der Tooltip anzeigt, für das nächste Mal, wenn Sie den Codeausschnitt häufiger benötigen. Sie können diese Verknüpfung dann später verwenden, um mit weniger Aufwand den Codeausschnitt einzufügen.

Der Editor fügt nun den kompletten Rumpf zum Abfangen eines Fehlers ein, den Sie im Prinzip nur ein wenig umgestalten müssen, etwa wie in den folgenden Codezeilen zu sehen:

```

If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
    Try
        'Versuche das hier ohne Fehler, und...
        locImage = Image.FromFile(CoverbildDateiname)
        '...wenn kein Fehler auftrat, liefere das Ergebnis zurück:
        Return locImage
    Catch ex As Exception
        'Beim Auftreten eines Fehlers landet man hier.

    End Try
End If

```

HINWEIS Denken Sie dabei bitte auch daran, *ApplicationException* in *Exception* umzuwandeln, damit nicht nur Ausnahmen vom Typ *ApplicationException*, sondern alle denkbaren Ausnahmen abgefangen werden können. Mehr zu diesem Thema, das insbesondere für VB6-Umsteiger wichtig ist, erfahren Sie in Kapitel 10.

Mit dieser Codeänderung haben wir den Fehler auf alle Fälle schon mal abgefangen. Jetzt müssen wir im Catch-Block nur noch dafür sorgen, auf ihn auch entsprechend zu reagieren – zum Beispiel indem wir eine E-Mail an einen zuständigen Administrator versenden.

Einfügen von Codeausschnitten mithilfe von Verknüpfungen

Auch dafür kommen uns die Codeausschnitte zu Hilfe. In der Kategorienliste *Konnektivität und Netzwerk* findet sich ein Eintrag namens *E-Mail-Nachricht erstellen*, der übrigens die Verknüpfungszeichenfolge

conEmail trägt. Wenn Sie, wie in diesem Fall, die Verknüpfungszeichenfolge kennen, brauchen Sie sie lediglich an die Stelle im Code einzufügen und  zu drücken, an dem der ganze Ausschnitt erscheinen soll. Und so wird ...

```
Function GetCoverImage(ByVal CoverbildDateiname As String) As Image
    Dim locImage As Image
    If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
        Try
            'Versuche das hier ohne Fehler, und...
            locImage = Image.FromFile(CoverbildDateiname)
            '...wenn kein Fehler auftrat, liefere das Ergebnis zurück:
            Return locImage
        Catch ex As Exception
            'Beim auftreten eines Fehlers, landet man hier.
            conEmail
        End Try
    End If
    Return Nothing
End Function
```

... nach dem Drücken von  hinter *conEmail* die Funktion folgendermaßen abgeändert:

```
Function GetCoverImage(ByVal CoverbildDateiname As String) As Image
    Dim locImage As Image
    If CoverbildDateiname IsNot Nothing AndAlso CoverbildDateiname <> "" Then
        Try
            'Versuche das hier ohne Fehler, und...
            locImage = Image.FromFile(CoverbildDateiname)
            '...wenn kein Fehler auftrat, liefere das Ergebnis zurück:
            Return locImage
        Catch ex As Exception
            'Beim Auftreten eines Fehlers, landet man hier.

            Dim message As New MailMessage("absender@adresse", "an@adresse", _
                "Betreff", "Nachtentext")
            Dim emailClient As New SmtpClient("E-Mail-Servername")
            emailClient.Send(message)
        End Try
    End If
    Return Nothing
End Function
```

Abbildung 5.46 Der eingefügte Codeausschnitt ist hier zu sehen. Mit  springen Sie bequem von Parameter zu Parameter und ändern diese in einem Rutsch.

Sie brauchen nun nur mit  von Parameter zu Parameter zu springen und die entsprechend gültigen Werte einzutragen, bis sich im Catch-Block beispielsweise folgendes Bild ergibt:

```
.
.
.

Catch ex As ApplicationException
    'Beim Auftreten eines Fehlers, landet man hier.
```

```
Dim message As New MailMessage("covers@loeffelmann.de", "klaus@loeffelmann.de", _
    "Fehler bei der Programmausführung", ex.Message)
Dim emailClient As New SmtpClient("192.168.0.1")
emailClient.Send(message)
End Try
```

HINWEIS Damit – und das sei nur der Vollständigkeit halber erwähnt – dieses Beispiel auf Ihrem System laufen kann, müssen Sie natürlich einen entsprechend konfigurierten SMTP-(Mail-)Server im Netzwerk zur Verfügung haben. Tragen Sie dann die für Sie gültigen Daten anstelle der hier im Listing abgedruckten E-Mail-Daten ein, auch die hier angegebene TCP/IP Nummer (192.168.0.1) müssen Sie durch die TCP/IP Nummer oder den Hostnamen (z.B. smtp.web.de) Ihres SMTP Servers ersetzen. Falls Sie auf SMTP-Server zugreifen müssen, die keine offene Relay-Funktion unterstützen,¹³ ergänzen Sie im Bedarfsfall noch folgende Zeile (fett im folgenden Listingauszug), mit der Sie die Anmeldeinformationen übergeben.

```
Catch ex As Exception
    'Beim Auftreten eines Fehlers, landet man hier.

    Dim message As New MailMessage("covers@loeffelmann.de", "klaus@loeffelmann.de", _
        "Fehler bei der Programmausführung", ex.Message)
    Dim emailClient As New SmtpClient("192.168.0.1")
    emailClient.Credentials = New Net.NetworkCredential("Username", "Passwort")
    emailClient.Send(message)
End Try
```

Einstellen des Speicherns von Anwendungseinstellungen mit dem Settings-Designer

Viele Anwendungen müssen beim Beenden Einstellungen speichern. Früher war das ein vergleichsweise großer Aufwand, denn Anwendungen mussten sich komplett selbst um die so genannte Serialisierung¹⁴ ihrer Einstellungsdaten kümmern.

¹³ Ein Weiterleiten von und an beliebige E-Mail-Adressen ohne Anmeldung, und das sollte bei den meisten SMTP-Servern (hoffentlich!) nicht gegeben sein, es sei denn, sie erlauben das Relaying aufgrund Verwendung der integrierten Sicherheit in Active Directory-Netzwerken. In diesen Fällen übernimmt die Anmeldung an das Netzwerk beim Starten von Windows auch das implizite Anmelden am SMTP-Server im Bedarfsfall.

¹⁴ Serialisierung nennt man den Vorgang, bei dem ein Objekt, das innerhalb einer Anwendung Daten im Hauptspeicher speichert, diese durch einen Datenstrom in einen Zielspeicher (seriell) ablegt, entweder zu dem Zweck, die Daten für die Weiterverwendung durch ein Objekt gleicher »Bauart« bereitzustellen oder dauerhaft auf einem Datenträger zu speichern. Beim entgegengesetzten Vorgang entsteht aus einem Datenstrom wieder die ursprüngliche Instanz des Objektes – es entspricht also dem Laden der Daten in den Hauptspeicher.

Programmierte Methoden innerhalb der Anwendung hatten dafür zu sorgen, die wichtigen Daten so aufzubereiten, dass sie im richtigen Format abgespeichert werden konnten. Das Konvertieren in das richtige Format (beispielsweise numerische Werte in Zeichenketten beim Serialisieren oder Zeichenketten in numerische Werte beim Deserialisieren) stellte dabei den größten Aufwand dar, weil falsche Typkonvertierungen (Datum liegt als Zeichenkette vor, es wurde aber beispielsweise versucht, die Zeichenkette in eine numerische Variable zu konvertieren) in vergleichsweise großem Aufwand abgefangen und ausgeschlossen werden mussten.

In .NET 3.5 bzw. Visual Studio 2008 geht das ungleich einfacher. Interaktiv können Sie mit einem speziellen Designer Einstellungsvariablen einrichten, auf die Sie dann von Ihrer Anwendung aus zugreifen und diese speziell zum Abspeichern von Anwendungseinstellungen verwenden können. Und das Tolle: Eine Visual Basic-Anwendung kümmert sich automatisch darum, dass die Inhalte dieser Variablen, wenn Sie es wünschen, beim Programmende gesichert und beim nächsten Programmstart automatisch wieder gestartet werden.

Einrichten von Settings-Variablen

Der Designer zum Einrichten der Settings-Variablen verbirgt sich in den Projekteigenschaften.

- Um diesen Designer also aufzurufen, wählen Sie aus dem Kontextmenü des Projekts *Covers* (nicht der Projektmappe!) im Projektmappen-Explorer den Menüpunkt *Eigenschaften*.
- Wählen Sie die Registerkarte *Einstellungen*, indem Sie auf die entsprechende Registerzunge an der linken Seite klicken.
- Für unser Beispielprogramm möchten wir, dass die Eingaben, die der Anwender in den Texteingabefeldern zur Laufzeit vorgenommen hat, zum Programmende gespeichert und, wenn das Programm erneut startet, wieder geladen und in die Eingabefelder geschrieben werden. Unsere erste Settings-Variablen nennen wir daher *LetzterFilmtitel*, und sie soll vom Typ String sein, da sie Zeichenketten speichert.
- Klicken Sie, wie in Abbildung 5.47 zu sehen, in die erste Namens-Zelle der Settings-Tabelle und geben Sie als Variablennamen *LetzterFilname* ein. Drücken Sie .



Abbildung 5.47 Den Settings-Designer (Einstellungs-Designer) finden Sie in den Projekteigenschaften, die Sie aus dem Kontextmenü des Projektes erreichen

5. In der nächsten Spalte würden Sie in der Aufklappliste den gewünschten VariablenTyp auswählen, was Sie in diesem Fall nicht machen müssen, da String für Texteingaben bereits der passende ist. Drücken Sie daher einfach .
6. Im Bereich wählen Sie aus, ob die Settings-Variable schreibgeschützt oder durch das Programm veränderbar sein soll. Benutzer wählen Sie, wenn Sie den Variableninhalt zur Laufzeit verändern wollen und der neue Inhalt nach Programmende auch gesichert werden soll; Anwendung wählen Sie, wenn es sich um eine Konstante handeln soll, die nur Sie zur Entwurfszeit einstellen können, die man aber zur Laufzeit nicht verändern darf. Lassen Sie auch hier die Einstellung Benutzer so, wie sie ist.
7. Das Feld Wert lassen Sie frei. Hier könnten Sie einen Standardwert einfügen, den die Settings-Variable beim ersten Start der Anwendung unter dem Benutzerkonto eines Benutzers haben würde.
8. Drücken Sie , um in die nächste Zeile zu gelangen, die vom Settings-Designer automatisch angelegt wird.
9. Auf diese Weise legen Sie weitere Variablen (alle vom Typ String und dem Bereich Benutzer) namens LetzterSchauspieler, LetzteBeschreibung und LetzterCoverbildDateiname an.
10. Klicken Sie auf das Ausgewählte Element Speichern-Symbol, um die Änderungen zu übernehmen.

Verwenden von Settings-Variablen im Code

1. Wechseln Sie nun mit  zum Entwurfsmodus von Form1.vb.
2. Doppelklicken Sie irgendwo ins Formular – am besten unterhalb der drei Schaltflächen, damit Sie nicht versehentlich doch ein anderes Steuerelement erwischen.
3. Platzieren Sie die Schreibmarke zwischen Private Sub Form1_Load und End Sub.
4. Beginnen Sie zu schreiben:

```
txtKurzbeschreibung.Text = My.Settings.
```

5. In dem Moment, in dem Sie den Punkt hinter My.Settings getippt haben, wird IntelliSense aktiv und Sie bekommen, etwa wie in Abbildung 5.48 zu sehen, alle Settings-Variablen aufgelistet, die Sie gerade eingerichtet haben.

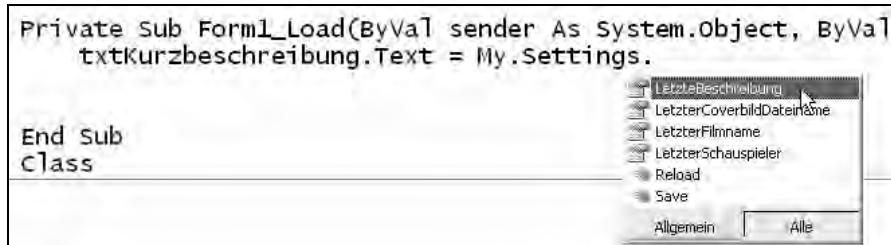


Abbildung 5.48 Den Zugriff auf die Settings-Variablen nehmen Sie über My.Settings – IntelliSense hilft Ihnen im Codeeditor anschließend beim Finden der richtigen Settings-Variablen

6. Wählen Sie für diesen Fall aus der Vervollständigungsliste **LetzteBeschreibung** aus.
7. Ergänzen Sie nach diesem Schema den Code um folgende Zeilen, sodass sich folgender Gesamtcodeblock ergibt:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles MyBase.Load
    txtKurzbeschreibung.Text = My.Settings.LetzteBeschreibung
    txtNameDesFilms.Text = My.Settings.LetzterFilmname
    txtSchauspieler.Text = My.Settings.LetzterSchauspieler

    Dim locImage As Image = GetCoverImage(My.Settings.LetzterCoverbildDateiname)
    picCoverbild.Image = locImage
End Sub
```

HINWEIS Sie werden übrigens feststellen, dass IntelliSense in der Vervollständigungsliste so lange, wie Sie noch keine Buchstaben zur genauen Identifizierung des zu vervollständigenden Wortes eingegeben haben, immer das Element in der Liste markiert, das Sie am häufigsten verwendet haben. Wenn Sie also beispielsweise zum dritten Mal **My** und anschließend den Punkt eingegeben haben, wird **Settings** automatisch selektiert.

Was macht **Form1_Load** nun genau?

Nun, durch den Zusatz **Handles MyBase.Load** wird bestimmt, dass **Form1_Load** dann automatisch aufgerufen wird, wenn das Framework das Formular darstellt – und das ist beim Programmstart der Fall. **Form1_Load** macht dann nichts weiter, als die Programmeinstellungen, die sich bereits in den **Settings**-Variablen befinden, in die Textfelder zu übertragen. Da Bilder übrigens nicht direkt in **Settings**-Variablen gespeichert werden können, bedienen wir uns eines Tricks: Wir speichern einfach den letzten bekannten Dateinamen, und versuchen das Bild beim Programmstart einfach wieder aus der gleichen Quelle zu laden. Da unsere Coverbild-Ladefunktion Fehler dabei abfangen kann, können wir uns die Eventualität leisten, dass das Bild nicht mehr an seiner ursprünglichen Stelle zu finden ist.

Damit das Konzept aufgeht, benötigen wir nun noch den umgekehrten Fall: Wenn das Formular geschlossen wird, müssen die Texte in den Textfeldern in die **Settings**-Variablen übertragen werden, damit dafür gesorgt werden kann, dass die Inhalte der **Settings**-Variablen (und damit der Feldinhalte) beim Programmende gesichert werden.

Der beste Zeitpunkt, genau dafür zu sorgen, ist, wenn das Formular im Begriff ist, sich zu schließen. Zu diesem Zeitpunkt sind die Inhalte der Steuerelemente nämlich noch alle erhalten. Dieses Ereignis trägt den Namen **FormClosing**,¹⁵ und die entsprechende Ereignisbehandlungsroutine werden wir im Folgenden implementieren:

1. Wählen Sie mit **[Strg] [F4]** das Formular *Form1.vb* im Entwurfsmodus aus.
2. Klicken Sie auf die Titelzeile des Formulars, um es zu selektieren.
3. Klicken Sie im Eigenschaftenfenster auf das Symbol *Ereignisse* (das Blitz-Symbol), um die Ereignisse anzeigen zu lassen.

¹⁵ Mehr zum Thema »Schließen von Formularen« erfahren Sie in Kapitel 37.

4. In der Rubrik *Verhalten* finden Sie das `FormClosing`-Ereignis; auf diesen Eintrag doppelklicken Sie nun.
5. Der Editor hat nun die Stub für das `FormClosing`-Ereignis eingefügt, das Sie nur noch um die entsprechenden Codezeilen zu ergänzen brauchen:

```
Private Sub Form1_FormClosing(ByVal sender As System.Object,
    ByVal e As System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
    My.Settings.LetzteBeschreibung = txtKurzbeschreibung.Text
    My.Settings.LetzterFilname = txtNameDesFilms.Text
    My.Settings.LetzterSchauspieler = txtSchauspieler.Text
    My.Settings.LetzterCoverbildDateiname = myBilddateiname
End Sub
```

Verknüpfen von Settings-Werten mit Formular- oder Steuerelementeigenschaften

Settings-Werte können übrigens auch dazu verwendet werden, Eigenschaften von Formularen und/oder deren Steuerelementen zu speichern. Dabei werden die entsprechenden Eigenschaftswerte sogar direkt an die Settings-Werte gebunden. Und was haben Sie davon?

Ein Beispiel: Angenommen Sie möchten, dass zur Laufzeit Ihres Programms beim Öffnen eines Formulars dieses automatisch an der letzten Position dargestellt wird. In diesem Fall könnten Sie Settings-Werte zur Speicherung der `Location`-Eigenschaft verwenden, die die Position des Formulars bestimmt. Sie müssten innerhalb des `Load`-Ereignisbehandlers des Formulars die Werte für die entsprechende Eigenschaft aus den Settings lesen, und sie umgekehrt beim Schließen des Formulars in `FormClosing` wieder in die Settings übernehmen.

Doch diesen Vorgang können Sie auch automatisieren – Sie binden die entsprechende Eigenschaft einfach an einen Settings-Wert, und das funktioniert folgendermaßen:

1. Öffnen Sie `Form1` durch Doppelklick auf die entsprechende Klassendatei im Projektmappen-Explorer.
2. Klicken Sie auf den Titelbalken des Formulars, um das Formular selbst zu selektieren.
3. Suchen Sie im Eigenschaftenfenster (auf das Zurückschalten auf die Eigenschaftenliste achten!) nach dem Eintrag (`ApplicationSettings`), und öffnen Sie den Zweig durch Klick auf das davor stehende +-Symbol.
4. Klicken Sie auf den Eintrag `Location` und öffnen Sie die Aufklappliste, wie in Abbildung 5.49 zu sehen.

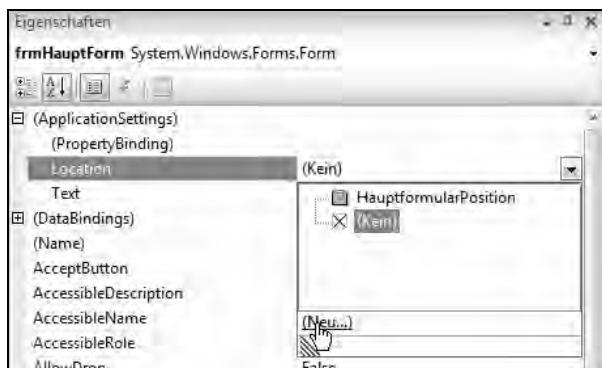


Abbildung 5.49 An dieser Stelle steuern Sie das Binden von Formulareigenschaften an die Anwendungseinstellungen (Application Settings)

5. In der Liste, die sich nun öffnet, klicken Sie auf (*Neu...*).
6. Visual Studio öffnet nun einen weiteren Dialog, in dem Sie ohne in die Anwendungseinstellungstabelle wechseln zu müssen, direkt einen neuen Settings-Wert mit dem für die Eigenschaft automatisch richtigen Typ einrichten können. Geben Sie dazu zunächst eine Standardposition für das Formular unter **DefaultValue** ein (denken Sie daran, die beiden Zahlen mit einem Semikolon und nicht mir einem Komma zu trennen!), und bestimmen Sie ferner den Namen für den Settings-Wert – beispielsweise **HauptformularPosition**.

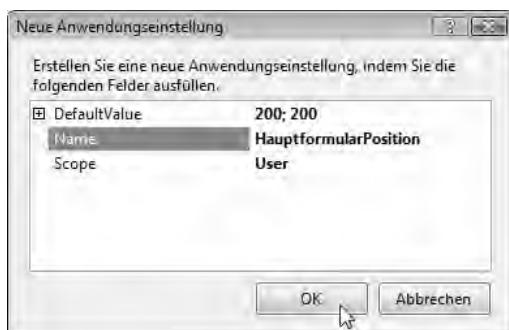


Abbildung 5.50 An dieser Stelle steuern Sie das Binden von Formulareigenschaften an die Anwendungseinstellungen (Application Settings)

7. Beenden Sie den Dialog mit OK.

Wenn Sie das Programm nun starten, öffnet sich das Hauptformular des Programms beim ersten Mal an Position 200; 200. Verschieben Sie anschließend das Formular und schließen es, dann wird es beim nächsten Start automatisch wieder an der Stelle erscheinen, an der es auch beim letzten Beenden positioniert war.

HINWEIS Die Größe des Formulars können Sie leider nicht auf diese Art und Weise an Settings-Werte binden. Die **Size**-Eigenschaft ist aus Framework-internen Gründen nämlich nicht an die Anwendungseinstellungen bindbar. Zwar ließe sich die Formulargröße auch indirekt durch die **ClientSize**-Eigenschaft an einen Settings-Wert binden, doch wenn Sie das machen, werden bei der Wiederherstellung der Größe die Anchor-Einstellungen der anderen Steuerelemente nicht korrekt berücksichtigt. In diesem Fall haben Sie also nur die Möglichkeit, wie in einem der vorherigen Absätze schon beschrieben, die Eigenschaftenzuweisung für die Größe des Formulars durch **Size** in den Ereignisbehandlungsroutinen **Load** und **FormClosing** manuell zu erledigen.

Und wo werden die Settings-Daten abgelegt?

Im persönlichen Anwendungsdatenverzeichnis unter *Lokale Einstellungen* auf dem Windows-Installationslaufwerk in weiteren Unterordnern in Abhängigkeit von Ihrem Benutzernamen sowie dem in den Assembly-Infos hinterlegten Firmennamen und dort in weiteren Unterverzeichnissen, die sich aus Laufzeitumgebung (Debug- oder Nicht-Debug-Modus), dem Anwendungsnamen und der Anwendungsversion ergeben. Alles klar?

Oder besser, da verständlicher: In meinem Fall lautet das Verzeichnis:

```
C:\Dokumente und Einstellungen\loeffel.ACTIVEDEVELOP\Lokale
Einstellungen\Anwendungsdaten\AndereFirma\Covers.vshost.exe_Url_ohvclojvckpztoiykw1hs3ba2acq2v4i\1.0.0.0
```

Warum?

- Mein Anmeldename in unserem Active Directory-Netzwerk lautet *loeffel*. Unsere Domäne *ActiveDevelop* (mein Geburtsdatum ist übrigens der 24.7.69, aber Sie werden sich mit diesem Wissen dennoch nicht bei uns anmelden können ...)
- Als Firmennamen habe ich im Assembly-Infodialog *Andere Firma* eingegeben. Diesen Dialog erreichen Sie, indem Sie aus dem Kontextmenü des Projektes im Projektmappen-Explorer *Eigenschaften* aufrufen. Wählen Sie das Register *Anwendung* (das ist die vorgewählte Eigenschaftenseite), und klicken Sie auf die Schaltfläche *Assemblyinformationen*. Das vorvorletzte Verzeichnis im Pfad entsteht aus dem Namen, den Sie unter *Firma* bestimmt haben.
- Das nächste Verzeichnis wird durch den Programmnamen bestimmt. Es entsteht aus dem Laufzeitprogrammnamen, der variieren kann. Wenn Sie das Programm im Debug-Modus starten, ist es nämlich nicht das Programm selbst, das gestartet wird, sondern ein Host-Prozess, der dafür sorgt, dass Sie Programmcode auch während des Debuggens verändern können¹⁶ und das dafür sorgt, dass die Geschwindigkeit beim Debuggen einigermaßen erträglich bleibt. Dieser Hostprozess hat als Anwendungsnamen eine Kombination aus eigentlichem Anwendungsnamen (*Covers*) und *.vshost.exe*. Diesem Block wird die Zeichenkette *Url_* sowie eine weitere Kennung angehängt – deren genaue Bedeutung beim Entstehen dieser Zeilen noch nicht zu ermitteln war.
- Schließlich folgt die Versionsnummer des Programms als weiteres Unterverzeichnis, in dem sich schließlich die *user.config*-Datei befindet – eine XML-Datei, die die eigentlichen Einstellungen speichert, wie im folgenden Beispiellisting zu sehen:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <userSettings>
        <Covers.My.MySettings>
            <setting name="LetzterFilname" serializeAs="String">
                <value>Terminator III - Rise of the Machines</value>
            </setting>
            <setting name="LetzterSchauspieler" serializeAs="String">
                <value>Arnold Schwarzenegger, Kristanna Loken, Nick Stahl, Claire Danes</value>
            </setting>
            <setting name="LetzteBeschreibung" serializeAs="String">
                <value>Kristanna versucht einen Kranwagen einzuparken, was nicht wirklich klappt.  
Arnie  
hilft nach, legt ihn aber dann aufs Dach.</value>
            </setting>
            <setting name="LetzterCoverbildDateiname" serializeAs="String">
                <value>C:\Dokumente und Einstellungen\All Users\Dokumente\  
Eigene Bilder\Beispielbilder\Arnie.jpg</value>
            </setting>
            <setting name="PrintFormPosition" serializeAs="String">
                <value>303, 123</value>
            </setting>
        </Covers.My.MySettings>
    </userSettings>
</configuration>
```

¹⁶ Dabei handelt es sich übrigens um das viel diskutierte Edit & Continue-Feature, das es seit Visual Basic .NET 2002 nicht mehr und erst mit Visual Basic 2008 wieder gibt.

HINWEIS Benutzerdaten und Anwendungsdaten, die aus Settings-Einstellungen hervorgehen, werden übrigens nicht in den gleichen Konfigurationsdateien gespeichert. Wenn Sie im Settings-Designer unter *Bereich* den Eintrag *Anwendung* gewählt und damit eine Nur-Lesen-Settings-Eigenschaft bestimmt haben, die für alle Benutzer gilt, werden diese Einstellungen in der Datei *appname.exe.config* gespeichert – wobei dabei *appname* dem Namen Ihrer Anwendung Ihres Projektes entspricht. Diese Datei befindet sich wiederum im *bin*-Unterverzeichnis des Verzeichnisses, das dem Namen der Konfiguration für die Erstellung des Projektes entspricht. Standardmäßig gibt es die Konfigurationseinstellungen *Debug* und *Release* – die sich in ihren Parametern zunächst nicht unterscheiden, außer, dass das Kompilat durch die unterschiedlichen Konfigurationsnamen auch in unterschiedlichen Unterverzeichnissen abgespeichert wird.

Haben Sie also beispielsweise Ihr Projekt im Hauptverzeichnis von Laufwerk »D:« unter dem Namen *Covers* erstellt, finden Sie die ausführbaren Dateien (und auch die *appname.exe.config*) im Verzeichnis *d:\covers*. Mehr zu den Konfigurationseinstellungen finden Sie im anschließenden grauen Kasten.

Über die Konfigurationseinstellungen »Debug« und »Release« sowie die Geschwindigkeiten der Codeausführung

Gerade Entwickler, die ihre ersten Gehversuche mit der Visual Studio-IDE absolvieren, neigen anfangs dazu, das Konfigurationsmanagement von Projekt-Compilereinstellungen misszuverstehen. Sie stellen oft fest, dass die Geschwindigkeit der Codeausführung in der *Debug*-Konfigurationseinstellung wohl nicht der echten entsprechen kann, und sie sind dann enttäuscht, wenn auch das Umstellen auf *Release* keine Geschwindigkeitsrekorde erzielt.

Aber das kann es auch gar nicht. Denn sie haben nur die vordefinierten Konfigurationseinstellungen von *Debug* auf *Release* umgestellt, und abgesehen davon, dass sich diese Einstellungen von vorne herein 1. überhaupt nicht unterscheiden (außer durch den Namen) und 2. genauso gut auch »Margarine« und »Plattenspieler« heißen könnten, dient die Konfigurationsumschaltung auch gar nicht dazu, die Ausführungsgeschwindigkeit in irgendeiner Form zu beeinflussen. Die Konfigurationsnamen *Debug* und *Release* implizieren das allerdings auf unglückliche Weise.

Wenn Sie wissen wollen, wie schnell ihr Programm später beim Kunden wirklich zu laufen in der Lage ist, dann starten Sie es über das Menü *Debuggen* und den Befehl *Starten ohne Debuggen* – oder drücken Sie einfach die Tastenkombination **Strg** **F5**.

Mit den Konfigurationseinstellungen, die Sie übrigens tatsächlich um die Einstellungen »Plattenspieler« oder »Margarine« ergänzen könnten, legen Sie hingegen beispielsweise fest, auf welche Plattformen das Kompilat abzielen soll (x86, egal welcher Prozessor – »Any«, etc.), welche Projekte innerhalb einer Projektmappe kompiliert werden sollen oder nicht und Ähnliches. Und im Übrigen auch, in welchen Unterverzeichnissen des Projektes die ausführbaren Dateien generiert werden.

WICHTIG Sobald Sie ein Programm im Debug-Modus (also mit **F5**) starten, wird es immer langsamer laufen, als wenn Sie es ohne zu debuggen starten (mit **Strg** **F5**).

Herzlichen Glückwunsch!

Sie haben soeben Ihr erstes funktionsfähiges (und wie ich finde auch recht brauchbares) Windows Framework 3.5-Programm fertig gestellt. Und nun: Viel Spaß beim Arbeiten mit Covers. Testen Sie es! Produzieren Sie damit die Hüllen Ihrer Urlaubsvideos. Drucken Sie, bis der Drucker qualmt!

Weitere Funktionen des Codeeditors

Nun hat Ihnen das Durchexerzieren des Beispiels schon viele der Funktionen des Editors nahe gebracht – nur leider nicht alle. Einige, von denen ich meine, dass Sie Ihnen beim Entwickeln von Projekten ebenfalls gute Dienste leisten können, finden Sie in den folgenden Abschnitten beschrieben.

Aufbau des Codefensters

Das Codefenster ist in drei vertikale Bereiche aufgeteilt; von links nach rechts gesehen sind das der so genannte *Indikatorrand* (der graue Bereich an der äußerst linken Seite), der *Auswahlrand* (die weiße, recht schmale Spalte links daneben) sowie der eigentliche *Codebereich*, der den Programmtext enthält.

Der Indikatorrand dient dazu, Haltepunkte, Lesezeichen oder Verknüpfungen aufzunehmen. Möchten Sie beispielsweise, dass Ihr Programm zu Testzwecken an einer bestimmten Programmzeile unterbrochen wird, setzen Sie mit **F9** einen Haltepunkt in der Zeile, vor der dann anschließend im Indikatorrand ein roter Haltepunkt zu sehen ist.

Sie können im Indikatorrand ebenfalls erkennen, welche Änderungen Sie seit dem Öffnen einer Codedatei am Code durchgeführt haben, und welche dieser Änderungen wiederum schon gespeichert wurden. Diese Codezustandsanzeige wird durch entsprechende Balken im Indikatorrand hervorgehoben. Abbildung 5.51 verdeutlicht die Funktionsweise des Indikatorrands.

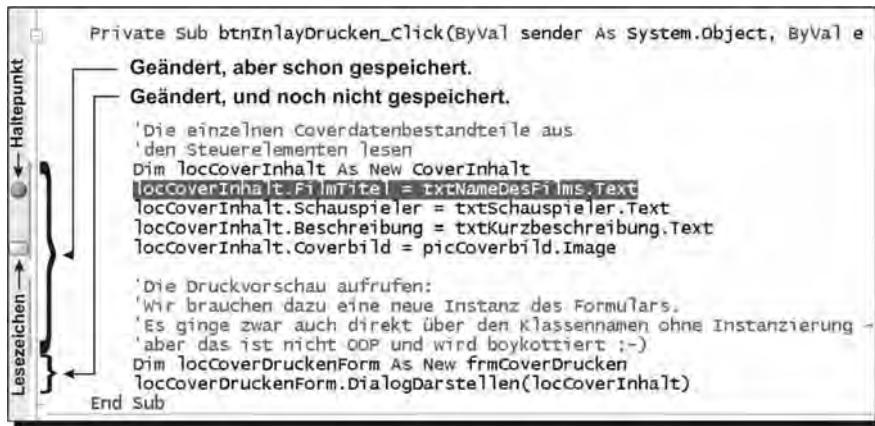


Abbildung 5.51 Der Indikatorrand hält Sie über den Speicher- und Änderungszustand einer Codedatei auf dem Laufenden und zeigt Elemente wie Lesezeichen oder Haltepunkte

Automatischen Zeilenumbruch aktivieren/deaktivieren

Mit der Tastenfolge (Achtung: Sie drücken die beiden Tastenkombinationen *nacheinander*) **Strg** **E**, **Strg** **W** können Sie Zeilen des Codes, die nicht in den sichtbaren Bereich passen, automatisch umbrechen lassen (siehe Abbildung 5.52).

```

Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
    'Das geht auch anders!
    Me.Close()
End Sub

Private Sub btnCoverbildwählen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCoverbildwählen.Click
    'Neue Instanz des OpenFileDialogs
    Dim locOfd As New OpenFileDialog

    'Parameter für den OpenFileDialog setzen
    With locOfd
        .CheckFileExists = True
        .DefaultExt = "*.bmp"
        .Filter = "JPEG-Bilder (*.jpg)|*.jpg|Windows Bitmap (*.bmp)|*.bmp|Alle Dateien (*.*)|*.*"
    End With

    'OpenFileDialog darstellen
    Dim locDr As DialogResult = locOfd.ShowDialog()
    'Falls Abbrechen angeklickt wurde...
    If locDr = Windows.Forms.DialogResult.Cancel Then

```

Abbildung 5.52 Mit dem automatischen Zeilenumbruch werden auch lange Zeilen auf einen Blick erkennbar

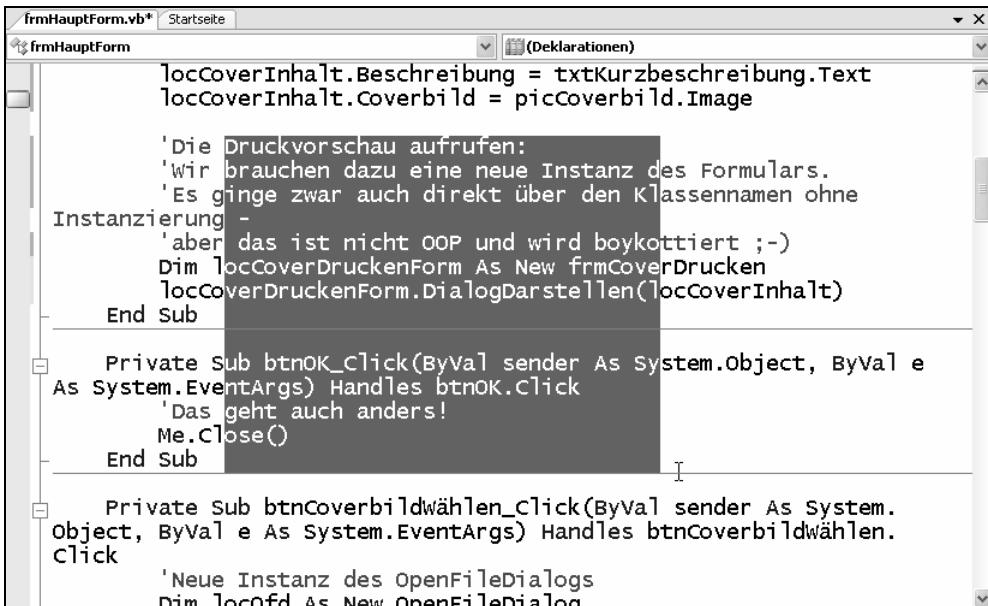
Doch aufgepasst: Eine derart umbrochene Zeile entspricht nicht dem Codezeilenumbruch von Visual Basic mit dem »_«-Zeichen am Zeilenende. Die Gefahr ist groß, eine durch den Editor umbrochene Zeile mit Tabulatoren oder Leerzeichen bündig zu formatieren – Sie würden dadurch aber Leerzeilen in die eigentliche Codezeile einfügen. Sie schalten mit der gleichen Tastenkombination den automatischen Zeilenumbruch auch wieder aus.

Navigieren zu vorherigen Bearbeitungspositionen im Code

Um schnell zu einer vorherigen Bearbeitungsposition zu gelangen, können Sie die Navigationsschaltfläche der Symbolleiste verwenden, oder Sie verwenden alternativ die Tasten **Strg** **[←]**, um rückwärts bzw. **Strg** **[↑]** **[←]**, um vorwärts zu navigieren.

Rechteckige Textmarkierung

Wenn Sie die Taste **Alt** auf der Tastatur gedrückt halten, können Sie mit dem Mauszeiger einen rechteckigen Textausschnitt markieren, etwa wie in Abbildung 5.53 zu sehen.



```

frmHauptForm.vb* Startseite | Deklarationen
frmHauptForm
    locCoverInhalt.Beschreibung = txtKurzbeschreibung.Text
    locCoverInhalt.Coverbild = picCoverbild.Image

    'Die Druckvorschau aufrufen:
    'Wir brauchen dazu eine neue Instanz des Formulars.
    'Es ginge zwar auch direkt über den Klassennamen ohne
    Instanzierung -
    'aber das ist nicht OOP und wird boykottiert ;)
    Dim locCoverDruckenForm As New frmCoverDrucken
    locCoverDruckenForm.DialogDarstellen(locCoverInhalt)
End Sub

    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
        'Das geht auch anders!
        Me.Close()
    End Sub

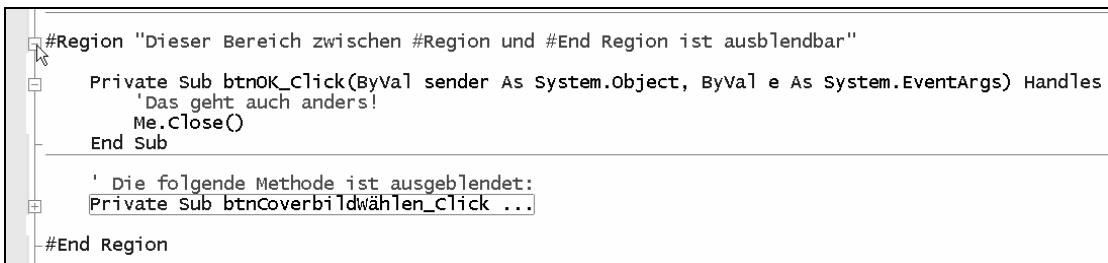
    Private Sub btnCoverbildwählen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCoverbildwählen.Click
        'Neue Instanz des OpenFileDialogs
        Dim locOfd As New OpenFileDialog

```

Abbildung 5.53 Der Editor von Visual Studio .NET erlaubt die rechteckige Ausschnittsmarkierung von Text bei gedrückter **[Alt]**-Taste

Gliederungsansicht

Der Codeeditor erlaubt Ihnen, bestimmte Codeteile ein- und auszublenden. Standardmäßig sind in Visual Basic .NET Prozeduren (Sub, Function), Klassen und Eigenschaften mit einem kleinen Gliederungszeichen versehen – mit einem Mausklick auf dieses Plus-Zeichen (siehe Abbildung 5.54) können Sie den Code ausblenden.



```

#Region "Dieser Bereich zwischen #Region und #End Region ist ausblendbar"
    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
        'Das geht auch anders!
        Me.Close()
    End Sub

    ' Die folgende Methode ist ausgeblendet:
    Private Sub btnCoverbildwählen_Click ...
#End Region

```

Abbildung 5.54 Dieser Codeabschnitt demonstriert den Einsatz der Gliederungsfunktion im Codeeditor von Visual Basic .NET

Mit den Funktionen im Menü *Bearbeiten/Gliederung* können Sie die Gliederungsansicht steuern.

Möchten Sie Teile des Quellcodes ausblenden, die weniger Codezeilen als eines der Standardelemente umfassen, dann markieren Sie den Codeteil, den Sie ausblenden wollen und wählen aus dem Menü *Bearbeiten* den Befehl *Gliedern/Aktuelles Element umschalten*.

Codeteile, die sich über mehrere Objekte erstrecken, können Sie auch mit den Direktiven

```
#Region
```

und

```
#End Region
```

(ebenfalls in Abbildung 5.54 zu sehen) gliedern.

Suchen und Ersetzen, Suche in Dateien

Sie erreichen die Suchen- bzw. die Ersetzenfunktion, indem Sie aus dem Menü *Bearbeiten* den Menüpunkt *Suchen und Ersetzen* abrufen. Hier öffnet sich ein weiteres Menü, das verschiedene Suchoptionen zur Verfügung stellt.

Mit der Schnellsuchfunktion finden Sie in der aktuellen Datei bzw. in allen geöffneten Dateien einen beliebigen Suchtext. Diese Funktion arbeitet bei Bedarf auch mit regulären Ausdrücken; Sie können den zu durchsuchenden Bereich genauer spezifizieren; Sie können nach Sonderzeichen suchen und haben sogar die Möglichkeit, alle Stellen, an denen ihr Suchbegriff vorkommt, durch Lesezeichen zu markieren (siehe auch Abbildung 5.55).

HINWEIS Reguläre Ausdrücke sind leistungsfähige Werkzeuge – allerdings braucht man ein wenig Einarbeitungszeit, um sie nutzen zu können. Im Kapitel 27 über reguläre Ausdrücke finden Sie ausführliche Beschreibungen zu diesem Thema. Auch wenn Sie nicht mit regulären Ausdrücken programmieren werden – allein für die reine Anwendung in Visual Studio (oder auch in Word 2003, das diese Funktion ebenfalls bietet) lohnt sich die Lektüre dieses Kapitels.

Das Schnellersetzen erlaubt es nicht nur, nach einem Begriff zu suchen, sondern diesen auch durch einen anderen Begriff zu ersetzen. Auch hier haben Sie die Möglichkeit, mit regulären Ausdrücken zu arbeiten.

Weitere Funktionen des Codeeditors

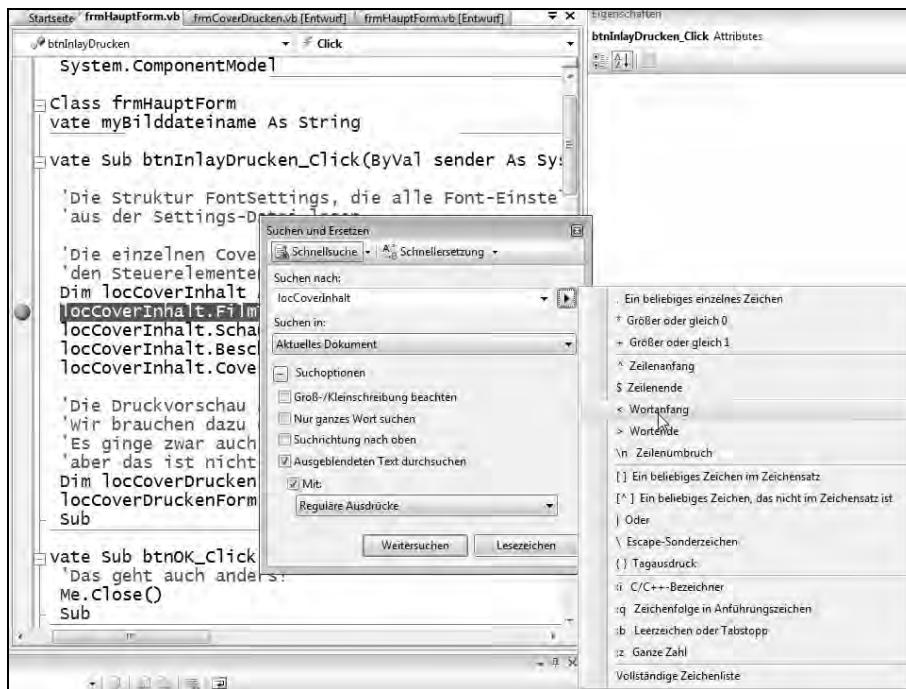


Abbildung 5.55 Mit der Suchfunktion können Sie nach regulären Ausdrücken suchen und alle Stellen, an denen der Suchbegriff vorkam, durch Lesezeichen markieren lassen

Suchen in Dateien



Abbildung 5.56 Mit der Suche in Dateien finden Sie alle Vorkommnisse eines Suchbegriffs in den Codedateien eines Projektes oder einer gesamten Projektmappe ...

Die Schnellsuchenfunktion wird nur auf die aktuelle Datei oder alle geöffneten Dateien angewendet. Wenn Sie eine Referenzliste aller Dateien innerhalb eines Projektes oder einer Projektmappe erhalten wollen, die einen bestimmten Suchbegriff beinhalten, bedienen Sie sich der Suche in Dateien.

```

Suchergebnisse: 1
Alle suchen "as", Unterordner, Suchergebnisse: 1, "Gesamte Projektmappe", "*.vb;*.resx;*.xsd;*.wsdl;*.xaml;*.xml;*.htm;*.html;*.css"
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverInhalt.vb(1): Public Class CoverInhalt
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverInhalt.vb(3):     Public FilmTitel As String
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverInhalt.vb(4):     Public Schauspieler As String
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverInhalt.vb(5):     Public Beschreibung As String
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverInhalt.vb(6):     Public Coverbild As Image
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverInhalt.vb(8): End Class
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(5): Public Class frmCoverDrucken
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(7):     Private WithEvents myPrintDoc
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(8):     Private myInhalt As CoverInhalt
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(10):    Public Function DialogDarstel
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(12):    'Die übergebenen Werte de
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(15):    'Das folgende Objekt brau
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(21):   'drucken, da das Cover sc
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(25):   'Dazu dem Preview-Steuere
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(26):   'Sobald die Zuweisung erf
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(36):  Private Sub btnDruckAufStand
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(38):  'Den Anwender den Drucker
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(39):  Dim locPD As New PrintD
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(40):  Dim locDR As DialogResult
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(53): Private Sub myPrintDocument_I
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(55): Dim g As Graphics = e.Gr
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(59): 'TODO: Man könnte die For
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(60): Dim locFontHaupttitel As
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(61): Dim locFontUntertitel As
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(62): Dim locFontRückensteg As
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(63): Dim locFontFließtext As
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(66): Dim locOffset_X As Single
C:\Users\juergenb.ACTIVEDEVELOP\Documents\Visual Studio 2008\Projects\Covers\Cover\frmCoverDrucken.vb(67): Dim locOffset_Y As Single

```

Abbildung 5.57 ... die dann in einer Dateiliste angezeigt werden. Ein Doppelklick auf eine Zeile der Ergebnisliste öffnet dann die Datei im Editor und positioniert den Cursor auf dem gesuchten Begriff.

Die Abbildungsunterschriften der beiden Abbildungen veranschaulichen die Funktionsweise der Suche in den Dateien.

Inkrementelles Suchen

Mit der inkrementellen Suche brauchen Sie den *Suchen*-Dialog erst gar nicht zu bemühen. Durch die Tastenkombination **Strg** **I** schalten Sie die inkrementelle Suche ein. Beginnen Sie anschließend direkt, die gesuchte Zeichenfolge einzutippen – währenddessen springt die Einfügemarkie automatisch an die erste Stelle, die dem bis dorthin eingetippten Suchtext entspricht. Den bis dahin eingegebenen Suchbegriff zeigt die Visual Studio-IDE in der Statuszeile an.

Mit erneutem Drücken der Tastenkombination **Strg** **I** finden Sie die nächste Stelle, die dem bisher eingegebenen Suchbegriff entspricht. Mit **Esc** beenden Sie die inkrementelle Suche.

Gehe zu Zeilennummern

Möchten Sie direkt zu einer bestimmten Zeile springen, deren Zeilennummer Ihnen bekannt ist, wählen Sie aus dem Menü *Bearbeiten* den Menüpunkt *Gehe zu*, oder Sie drücken einfach die Tastenkombination **Strg** **G**. Im Dialog, der jetzt erscheint, tippen Sie die Nummer der Zeile ein, zu der Sie die Einfügemarkie bewegen wollen.

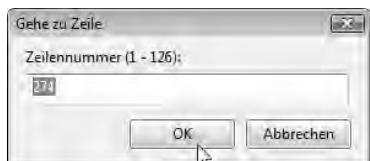


Abbildung 5.58 Mit diesem Dialog gelangen Sie zu jeder Zeile durch Eingabe der Zeilennummer – die aktuelle Zeilennummer wird vorgegeben. Zeilennummern brauchen nicht eingeschaltet zu sein.

Lesezeichen

Lesezeichen ermöglichen Ihnen, sich bestimmte Stellen zu merken, die Sie später noch überprüfen und bearbeiten wollen. Um ein Lesezeichen in einer Zeile zu setzen oder ein vorhandenes wieder zu löschen, verwenden Sie die Tastenreihenfolge [Strg] [K], [Strg] [K]. Ein gesetztes Lesezeichen wird im Indikatorrand des Codefensters durch eine kleine blaue Marke angezeigt. Mit [Strg] [K], [Strg] [N] verschieben Sie die Einfügemarke zum nächsten Lesezeichen, mit [Strg] [K], [Strg] [P] zum vorherigen. Mit [Strg] [K], [Strg] [L] löschen Sie alle Lesezeichen.

WICHTIG Das Löschen geschieht ohne weiteres Nachfragen – seien Sie also vorsichtig mit dieser Tastenkombination!

Alternativ können Sie alle beschriebenen Funktionen auch aus dem Menü *Textmarken* aufrufen, das Sie im Menü *Bearbeiten* finden.

Sie können übrigens, wenn Sie mit der Suchfunktion nach Begriffen in Ihren Dateien suchen, alle Stellen, an denen der Suchbegriff vorkommt, mit Lesezeichen markieren. Dazu wählen Sie im Schnellsuchendialog einfach die Schaltfläche *Lesezeichen*.

TIPP Alternativ zu Lesezeichen können Sie auch eine Zeile für die Anzeige zur Nachbearbeitung in der Aufgabenliste markieren. Zu diesem Zweck fahren Sie mit dem Cursor in die entsprechende Zeile, und wählen aus den Menüs *Bearbeiten/Textmarken* die Funktion *Verknüpfung für Aufgabenliste hinzufügen* aus. Im Indikatorrand wird anschließend ein entsprechendes Symbol aufgerufen. Um eine solche Kennzeichnung wieder zu löschen, rufen Sie die gleiche Funktion abermals auf.

Kapitel 6

Einführung in Windows Presentation Foundation

In diesem Kapitel:

Was ist die Windows Presentation Foundation?	172
Was ist so neu an WPF?	173
Wie WPF Designer und Entwickler zusammenbringt	185
Der WPF-Designer	194
Logischer und visueller Baum	195
Die XAML-Syntax im Überblick	197
Ein eigenes XAMLPad	201
Zusammenfassung	206

Das Auge isst mit. Diesen Spruch habe ich mir von meiner Mutter schon anhören müssen, als ich als kleiner Junge den Tisch decken musste – na ja, sagen wir: entsprechende Versuche veranstaltet habe, diesen ihren ästhetischen Bedürfnissen entsprechend zu decken. Mein eigener Grundsatz »Form follows function«, in leicht abgeänderte Version im Deutschen eher als »Funktion vor Design« bekannt, widersprach diesem Grundsatz meiner Ma und war – ich gebe es offen zu – sicherlich auch Beweggrund, mich seit Visual Basic 4 eher auf Windows Forms-Anwendungen als auf Web-Anwendungen zu spezialisieren. Und mich im Übrigen auch nicht als Innenarchitek zu versuchen. Viele Webanwendungen lassen sich sicherlich grafisch und designtechnisch ansprechend, auf alle Fälle aber sehr viel ansprechender als Windows Forms-Anwendungen designen. Leider aber nur von den richtigen Leuten, zu denen ich zugegeben leider nicht gehöre.

Was ist die Windows Presentation Foundation?

Und dann brachte Microsoft mit dem Framework 3.0 die Windows Presentation Foundation. Erklärtes Ziel war es eigentlich, und daraus leitet sich auch der Name ab, eine Technik zu schaffen, mit der Informationen in grafisch vollendet Form vor allem präsentiert werden können. Aber nicht nur. Die WPF, wie die Windows Presentation Foundation gerne abgekürzt genannt wird, sollte auch dazu dienen, diese Grafik als Benutzerschnittstelle zu nutzen, also um mit dem Benutzer zu interagieren – mit Maus, mit Stift oder, wie es wohl schätzungsweise mit Windows 7, dem Nachfolger von Windows Vista der Fall sein wird, mit fingerberührungssempfindlichen Displays auf Notebooks¹ oder speziellen Tischdisplays.²

Und das bedeutet für uns Windows Forms-Programmierer: Umdenken, und zwar heftigst, wie wir Westfalen sagen. Der deutsche Spruch passt nicht mehr, denn nunmehr gilt: *Funktion ist Design*. Mit dem englischen Idiom sind wir immer noch gut aufgehoben, denn es ändert sich nichts an »Form follows function«.

Die WPF unterliegt allerdings in diesem Moment, in dem diese Zeilen entstehen, einem kleinen, von keinem und dann wieder von allen vorangetriebenen Evolutionsprozess. Denn obwohl Microsoft die WPF niemals zum designierten Nachfolger von Windows Forms erklärt hat, zeichnet es sich zur Zeit ab, dass genau das passiert: Viele der großen VB6-Anwendungen müssen auf eine neue Plattform migriert werden, da zum Ersten der Support für VB6 am 8. April 2008 geendet hat und zum Zweiten schon jetzt viele VB6-Anwendungen unter Vista erhebliche Probleme bereiten und unter dem Nachfolger von Windows Vista vermutlich überhaupt nicht mehr funktionieren werden. Die einzige Lösung dazu lautet: Die VB6-Anwendungen müssen auf bzw. zu Visual Basic.NET migriert werden. Und während dieses Migrationsprozesses, während also die Konzeptionen für die beste Vorgehensweise einer solchen Umstellung von den tausenden verschiedenen Entwicklerteams, die es da draußen gibt, ausgearbeitet wird, stellt sich diesen erfahrungsgemäß immer wieder die gleiche Frage: Wie sollen die neuen Frontends und Benutzeroberflächen gestaltet werden? Und siehe da: In diesem Prozess kommen immer mehr Teams zu dem Schluss, dass sich die WPF *durchaus* auch als UI-Ersatz für Windows Forms eignet. Die langfristigen Benefits, die sich daraus ergeben, scheint man als wichtiger zu erachten, als die Nachteile und den erheblichen Zusatzaufwand, den man vor allen Dingen zurzeit noch bei der Ablösung von Windows Forms durch WPF hinnehmen muss. Und dieser Zeitaufwand liegt vor allem an folgendem:

¹ Demo gefällig? YouTube hilft, und der IntelliLink **A0601** zeigt wo.

² Mehr dazu zeigt der IntelliLink **A0602**.

- Der Einarbeitungsaufwand in die Technik der WPF ist ganz erheblich. Das gilt vor allen Dingen für Teams, die aus der VB6-Welt kommen, und die sich ohnehin erst an das objektorientierte Programmierkonzept gewöhnen müssen. Und aufgepasst: Die Rede ist dabei von *objektorientiert* – viele Entwickler, die selbst wirklich alles aus VB6 herausgeholt haben, und dessen Tücken und Fallen aus dem FF kennen, haben dennoch – konzeptbedingt – bestenfalls *objektbasiert* programmieren können. Das ist ein großer Unterschied, gerade was den bestmöglichen Aufbau einer Anwendung anbelangt, der in Visual Basic .NET – objektorientiert – zu einem komplett anderen Ansatz führen kann, bei dem im Vergleich zur Ausgangsanwendung, kein Stein auf dem anderen verbleibt.
- Der Aufbau von Formularen, oder das, was dem am nächsten kommt, unterliegt einem ganz anderen Konzept als in Windows Forms. Es gibt hier – ähnlich wie bei HTML zur Beschreibung von Webseiten – eine an XML angelehnte Sprache namens XAML (sprich: Gsämmel), die dafür zuständig ist. Hier ist viel Lernaufwand erforderlich, insbesondere dann, wenn einem schon das Konzept von Seitenaufbau beschreibenden Dialekten wie HTML eher fremd ist.
- Windows Forms verwendet einen Designer, der altbewährt ist. Selbst der .NET-Designer für Windows Forms-Anwendungen hat bereits 5 Jahre auf dem Buckel, und schon der allererste Designer dieser .NET-Version basierte konzeptionell weitestgehend auf dem, was zuletzt in Visual Basic 6 zu finden war. Der WPF-Designer hat – wohlwollend ausgedrückt – noch einiges an Verbesserungspotential, und so sollten Sie sich darauf einrichten, dass Sie viele Dinge, die Sie unter Windows Forms natürlich interaktiv mit Designerunterstützung machen, für die WPF direkt in XAML eintippen müssen. Während Sie die Ergebnisse Ihrer Arbeit anfangs oft mit positivem Erstaunen kommentieren werden, wundern Sie sich deswegen andererseits nicht, dass Ihnen auf den Weg dahin sicherlich der ein oder andere Fluch entgleiten wird.

HINWEIS Wenn Sie professionelle WPF-Anwendungen entwickeln wollen, werden Sie um den Einsatz einer speziell auf WPF abgestimmten Designer-Software nicht drum herumkommen. Ihr Name: Express Blend – und es gibt sie inzwischen in der zweiten Version. Der unverständliche Nachteil: Expression Blend ist NICHT in Visual Studio 2008 enthalten – Sie müssen es wohl oder übel dazu kaufen. Spätestens hier bietet sich der Kauf eines so genannten MSDN-Abos an: Schon in der MSDN-Professional Edition dieses Abonnements sind sowohl Visual Studio 2008 Professional als auch Expression Blend 2 enthalten. Und natürlich erhalten Sie dazu alle Betriebssysteme, Server und die gängigsten Anwendungsprogramme aus dem Hause Microsoft für Entwicklungszwecke obendrein. Mehr zum Thema MSDN erfahren Sie unter dem IntelliLink **A0603**.

Was ist so neu an WPF?

Ein guter Freund sagte mir heute, er hasse an vielen Computerbüchern, dass sie bestimmte Sachverhalte nicht einfach Punkt für Punkt an simplen, einfachen Beispielen erklären. Und er bezog sich damit indirekt auch auf die Rohfassung des Textes, den Sie gerade lesen. An dieser Stelle, so seine Meinung, seien schon zwei Seiten ins Land gegangen, und er wisst immer noch nicht, wie man eine einfache Linie mit der WPF auf den Bildschirm malt – das könnte doch nicht so schwer sein. Und da hat er vielleicht irgendwie recht, gleichzeitig

aber auch nicht verstanden, was die WPF ist. Denn seine Frage impliziert schon, dass er meint, dass der Rendering³-Engine von WPF so funktioniert, wie das was er kennt, sodass beispielsweise aus einer Linie ein Rechteck und aus einem Rechteck eine Schaltfläche entsteht. So war es aber nicht gedacht.

Dummerweise ist es nämlich völlig anders. Wenn Sie aber verstehen wollen, wie die WPF funktioniert, sollten Sie etwas über ihre Entstehungsgeschichte wissen. Warum? Ganz einfach, und als Metapher erzählt: Sonst geht es Ihnen wie diesem Freund, der bislang nur das Waschbrett kannte. Und der sich beim erstmaligen Anblick der Waschmaschine fragt, wie man so etwas unergonomisches bauen kann, wie eine Waschmaschine, bei der die Waschreibe auf der Innenseite einer Trommel angeordnet ist, und man zum Waschen der Wäsche ja in diese hineinkriechen müsste.

Sein einfaches Beispiel, das er verlangt, kann er bekommen. Doch nicht ohne den angekündigten Exkurs in Sachen *Geschichte von Windows*, und wie dessen grafische Elemente bislang den Weg auf den Bildschirm fanden.

25 Jahre Windows, 25 Jahre gemalte Schaltflächen

Windows-Anwendungen sind mittlerweile seit mehr als 25 Jahren zugegen. Und seien wir ehrlich: Auch wenn es Windows 1.0 und 2.0⁴ schon gab, wurde erst Windows 2.11 als Bundle-Version mit Adobes Page-maker etwas bekannter. Und erst mit Windows 3.0 kam der Mainstream-Durchbruch dank neuem Look & Feel bei der Bedienung von Anwendungen, dessen grundsätzliches Konzept sich später allerdings selbst mit Windows 95 und Windows NT sowie über Windows 2000 und Windows XP hinweg im Wesentlichen kaum änderte.

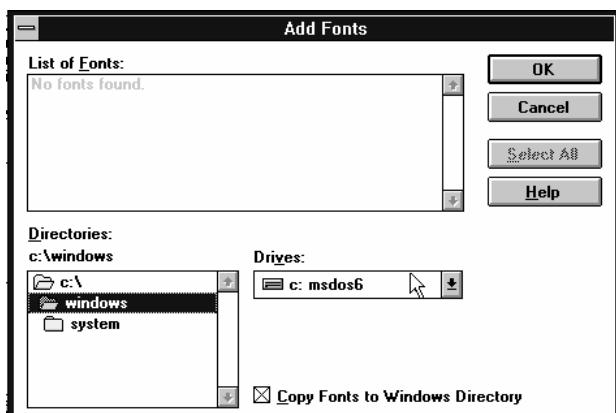


Abbildung 6.1 Der Dialog zum Hinzufügen von Schriftarten in einem englischen Windows 3.0 ...

³ Lustigerweise gibt es keinen wirklich passende Übersetzung, der das Konzept des Renderings des »Entstehenlassens oder des Umsetzens« so ganz passend wieder gibt. Was gemeint ist – der IntelliLink **A0604**.

⁴ Mehr dazu liefern die IntelliLinks **A0605**.

Sicherlich: Windows ist vor allen Dingen, was die auf NT-basierenden Versionen anbelangt, im Lauf der Zeit deutlich stabiler geworden; Windows 95 brachte genau wie Windows NT 4 viele neue grafische Elemente, die mit Windows XP nochmals bunter, moderner und mit einer Menge Farbverläufen gekrönt wurden. Doch eines hat sich im Grunde genommen seit der ersten Version von Windows nicht geändert: Die Art und Weise, wie die Grafik ihren Weg auf die Mattscheibe findet.



Abbildung 6.2 ... und in Windows Vista. Hier ist deutlich zu sehen, dass sich über all die Jahre wirklich nicht viel am Prinzip der Windows Bedienung und des Renderings von WinForms-Dialogen geändert hat.

Das geschieht nämlich im Grunde genommen bei 99% aller Programme mithilfe des so genannten GDIs, des *Graphic Device Interface* von Windows. Diese API⁵ beinhaltet Funktionen, mit der zweidimensionale Zeichenfunktionen (das Malen von Linien, Rechtecken, Kreisen und verschiedenen Pinselstärken, Formen und Mustern) durch Grafikkarten unterstützt möglich sind. *Grafikkarten unterstützt* bedeutet dabei, dass die Treiber der Grafikkarte bestimmte Befehle beispielsweise zum Zeichnen einer Linie nicht selber durch Algorithmen und mithilfe des Prozessors umsetzen, sondern sie vielmehr an die Grafikkarte weiterreichen können, damit die Grafikkarte diese Linie selbst malt – natürlich viel, viel schneller, als es der Prozessor Ihres Computers jemals könnte. Und wie das in der Praxis ausschaute, daran soll uns das folgende kleine Beispiel noch einmal erinnern:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 06\\SimpleSampleGDIPlus

Öffnen Sie dort die Projektmappe (.SLN-Datei) *SimpleSampleGDIPlus*.

⁵ API bedeutet *Application Programming Interface*, in etwa Anwendungsschnittstelle zum Programmieren.

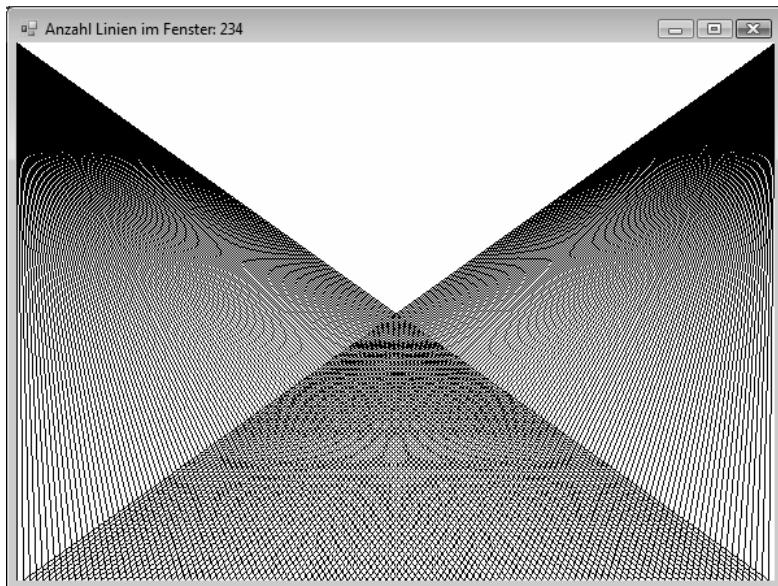


Abbildung 6.3 Das Ergebnis nach einem Doppelklick ins Fenster – eine Reihe von aufgefächerten Linien zeigt eindrucksvoll den bei Fernsehmachern so gefürchteten Moiré⁶-Effekt.

Die Figur, die dort gezeichnet wird (siehe Abbildung 6.3), interessiert uns allerdings nur am Rande. Vielmehr soll Augenmerk darauf gelegt werden, *wie* die Linien auf den Bildschirm kommen, und wie Windows dafür sorgt, dass sie dort verbleiben. Und dazu schauen wir uns erstmal das kleine, zur Abbildung zugehörige Listing an, das die Grafik beim Doppelklick ins Fenster bringt.

```
Public Class Form1

    Private mydoppelGeklickt As Boolean

    'OnLoad überschreiben - damit das Load-Ereignis behandeln, ...
    Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
        MyBase.OnLoad(e)
        '...und hier den Fenstertext setzen.
        Me.Text = "Doppelklicken Sie in das Fenster, um die Grafik darzustellen."
    End Sub

    'Wir hätten auch das Load-Ereignis behandeln können...
    Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles Me.Load
        'aber warum ein Ereignis der Instanz in der Instanz behandeln,
        'die das Ereignis auslöst? Dann lieber an der Stelle eingreifen,
        'die das Ereignis auslösen, und das ist OnLoad.
    End Sub
```

⁶ Sprich: Muareh – Mehr dazu zeigt der IntelliLink **A0606**.

```
'Grafik wird ins Fenster gemalt, sobald der Anwender doppelklickt.
Protected Overrides Sub OnDoubleClick(ByVal e As System.EventArgs)
    MyBase.OnDoubleClick(e)
    Dim g As Graphics = Graphics.FromHwnd(Me.Handle)
    DrawDemo(g)
    mydoppelGeklickt = True
End Sub

'Und hier wird gemalt.
Public Sub DrawDemo(ByVal g As Graphics)

    'Erstmal den Fensterinhalt löschen.
    g.Clear(Color.White)

    'tatsächliche Breite des Clientbereichs ermitteln und merken
    Dim tatsächlicheBreiteGemerkt = Me.ClientSize.Width
    Dim linienZähler = 0

    'Jeden 5. Pixel berücksichtigen
    For x = 0 To tatsächlicheBreiteGemerkt Step 5

        'Erst die Linie von links oben nach rechts unten malen.
        g.DrawLine(Pens.Black, 0, 0, x, Me.ClientSize.Height)

        'Und dann noch eine von rechts oben nach links unten.
        g.DrawLine(Pens.Black, tatsächlicheBreiteGemerkt, _
                   0, tatsächlicheBreiteGemerkt - x, _
                   Me.ClientSize.Height)

        'Linienzähler aktualisieren
        linienZähler += 2
    Next

    'Im Fenstertitel über die Anzahl gemalter Linien informieren.
    Me.Text = "Anzahl Linien im Fenster: " & linienZähler
End Sub
End Class
```

Zum Malen von Inhalten im GDI und GDI+ ist dabei folgendes zu sagen: Sie sind höchst volatil. Wenn Sie nämlich nicht selbst dafür sorgen, dass der Fensterinhalt, nachdem er zerstört wurde, weil etwas anderes über das Fenster geschoben wurde, oder jemand das Fenster vergrößert oder verkleinert hat, dann ist dessen Inhalt eben weg, wie die folgende Grafik anschaulich zeigt, bei der das Fenster zunächst ein paar Pixel horizontal und vertikal vergrößert wurde, und sich anschließend auch der Windows-Taschenrechner über das Fenster geschoben hat.

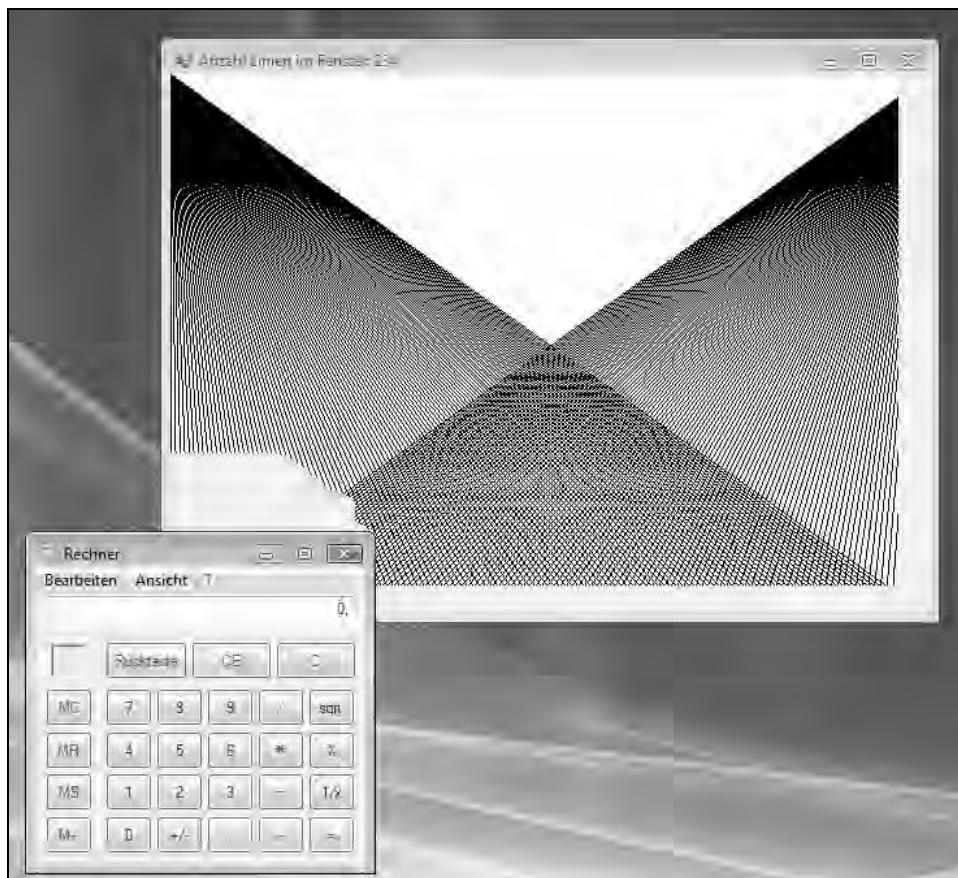


Abbildung 6.4 Um das Aktualisieren des Fensterinhalts muss sich der Entwickler unter GDI oder GDI+ selbst kümmern. Tut er es nicht, gibt's beim Vergrößern oder beim Verdecken des Fensters Effekte, wie hier in der Abbildung zu sehen.

Damit solche Probleme wie in Abbildung 6.4 nicht passieren können, muss der Entwickler selbst Hand anlegen: Im Paint-Ereignis, das dann ausgelöst wird, wenn sich irgendetwas vor das Fenster legt, und das Neuzeichnen des Inhalts beim Wiederhervorholen des Fensters erforderlich macht, sorgt er dafür, dass die Anwendung informiert wird, den Fensterinhalt aktualisieren zu müssen. Ein Ergänzen der folgenden Zeilen würde in unserem Beispiel also bereits für Abhilfe schaffen.

```
'Wird ausgelöst, wenn der Fensterinhalt zerstört wurde
'und neu gezeichnet werden muss
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    MyBase.OnPaint(e)

    'Nur wenn der Doppelklick bereits stattgefunden hat.
    If mydoppelGeklickt Then
        '
        DrawDemo(e.Graphics)
    End If
End Sub
```

Gegebenfalls muss er auch ein entsprechendes Resize-Ereignis behandeln, das ausgelöst wird, wenn sich die Fenstergröße durch Verkleinern oder Vergrößern des Fensters ändert.

Und dieses Prinzip des Darstellens von Dokumenteninhalten gilt für Windows Forms seit der ersten Windows-Version bis heute. Und nicht nur für Grafiken, die in Fenstern dargestellt werden, sondern für alles. Und dann wiederum doch nur für Grafiken, die in Fenstern dargestellt werden, denn vielen Leuten ist es überhaupt nicht bewusst, wieso Windows Windows heißt. Warum? Weil nicht nur Dokumenteninhalte jeglicher Form in Fenstern, die auch als solche zu erkennen sind, dargestellt werden. Jedes Element in Windows – mit wenigen Ausnahmen – ist nämlich ein Fenster. Eine Schaltfläche ist ein Fenster. Eine ListBox ist ein Fenster. Ein aufgeklapptes Kombinationslistenfeld sind zwei Fenster – die TextBox ist eines und die Liste darunter ein weiteres. Streng genommen ist auch ein Tooltip oder eine Sprechblase des SysTrays ein Fenster, und alle diese Windows-Klassen müssen dafür sorgen, dass sie ihren Inhalt neu malen, in dem Moment, wo dieser durch Überlappung mit einem anderen Fenster zerstört wurde. Die Grafikkarte hilft dabei hardwaremäßig nur wenig:

- Sie kopiert Bildblöcke in rasender Geschwindigkeit.
- Sie zeichnet Grundlinienfiguren wie Linien, Rechtecke, Ellipsen und füllt diese mit Inhalten. Das macht sie bis heute aber leider nur, wenn die nativen Zeichenbefehle des Betriebssystems, genau gesagt, des GDIs verwendet werden. Die .NET-Grafikbefehle werden von einer moderneren Grafiklibrary umgesetzt, des so genannten GDI+. Diese lässt leider auf vielen Grafikkarten die entsprechende Hardwareunterstützung vermissen, sodass sie zwar die deutlich komfortableren Grafikoperationen anbietet, diese aber *deutlich* langsamer als ihre GDI-Pendants sind.

HINWEIS Zwei Ausnahmen betreffen einerseits das Rendern von Text unter .NET. Hier gibt es den TextRenderer, der tatsächlich als einziges direktes Grafikobjekt vom GDI Gebrauch macht. Zum Zeichnen bestimmter UI-Elemente gibt es darüber hinaus auch die ControlPaint-Klasse im Namespace System.Windows.Forms, deren statische Methoden teilweise auch vom GDI Gebrauch machen (zum Beispiel DrawReversibleFrame zum invertierten Zeichnen eines Rahmens oder DrawReversibleLine zum invertierten Zeichnen einer Linie).

Spielende Protagonisten

Grafikkarten können allerdings schon seit etlichen Jahren erheblich mehr als nur diese »alten Kamellen« in Sachen Grafikrendering. Das so genannte DirectX gibt es bereits seit Windows 95, und die Älteren unter uns erinnern sich sicherlich noch an den ersten Egosshooter *Wolfenstein*, der zwar einen sehr umstrittenen Spielgegenstand lieferte, dessen Entwickler der Firma id-Software aber nichtsdestotrotz demonstrierten, was eigentlich aus einem Bürocomputer der frühen 90er Jahre in Sachen Spiele und damit natürlich in Sachen Grafik und Sound herauszuholen war. Das grundsätzliche Problem der damaligen Zeit war allerdings, dass für diese enorm hohe Performance eine Anwendung wirklich alleine und ohne störende »Mitbewerber« laufen musste. Und das war, obwohl Windows 3.11 schon als Quasi-Multitasking-Betriebssystem existierte, eben nur unter MS-DOS möglich. DirectX erst lieferte die nötigen Voraussetzungen, damit das – inzwischen war Windows 95 auf dem Markt – auch innerhalb eines Multitasking-Betriebssystems möglich wurde. Und auch hier war es wieder id-Software, die letzten Endes, und auch erst mit der Version 3.0 von DirectX durch die Portierung des Computerspiels Doom auf Windows 95, zu DirectX' Geburtshelfer wurde. Denn erst mit dieser Version gab es genügend Spieleentwickler, die DirectX als echte Multimedia-Alternative zum Stand-Alone-Konzept von DOS sahen, und DirectX gewann die nötige Akzeptanz.

Und gerade die Gamer sahen von diesem Zeitpunkt an atemberaubende Benutzeroberflächen für ihre heißgeliebten Spiele, die nur durch 3-dimensionale, mit virtueller Realität ausgestattete, KI-Systeme in Science-Fictionfilmen übertroffen wurden. Mal ehrlich: Für welchen Windows-Anwender der damaligen Zeit war nicht der Film *Enthüllung* mit Michael Douglas und Demi Moore ein benutzeroberflächentechnischer Hochgenuss – wenn auch mit dem leicht bitteren Beigeschmack, nämlich zu wissen, dass man sich 45 Minuten später am seit Jahren bekannten und deshalb recht eintönige User Interface⁷ von Windows NT SP4a der realen Welt einfinden musste.

Und genau dieses Problem sollte uns Windows-Anwendern dann noch die nächsten ca. 10 Jahre erhalten bleiben, sodass viele von uns Anwendern und sicherlich nicht nur unter uns Software-Entwicklern sich sehnlichst wünschten, wenigstens eine kleine Bewegung in Richtung Spiele-UIs erleben zu dürfen. Doch wir sollten über Windows 2000, dessen zahlreichen Service Packs, und auch Windows XP und dessen leider weniger zahlreichen Service Packs enttäuscht bleiben. Doch dann kamen Windows Vista und das Framework 3.0.

Mit Windows Vista gab es nämlich auch endlich ein neues Grafiksystem, dass die Darstellung von bestimmten Dingen in einer Funktionsbibliothek auf der Basis von DirectX erlaubte – genau das war die Windows Presentation Foundation, kurz WPF. WPF-basierende Anwendungen laufen am flüssigsten und am, wenn ich dieses Wort einmal kreieren darf, *hardwareunterstütztesten* unter Windows Vista. Mit einer besonderen Version kommen aber auch Windows XP-Anwender in den Genuss, WPF basierende Anwendungen einzusetzen, aber eben nicht in dieser hochwertigen Darstellungsvollendung wie Windows Vista.

Zum Zeitpunkt, an dem diese Zeilen entstehen, ist die Multimedia-Entwicklergemeinde, was DirectX anbelangt, inzwischen bei der Version 10.1 angelangt. Version 10 beinhaltet die neusten und aufwendigsten Multimedia-Funktionsbibliotheken, die es jemals unter einer Windows Version gab. Und wer die Entwicklung von Windows Vista mitbekommen hat, der weiß nicht nur, dass DirectX 10 ausschließlich Vista-Benutzern und -Entwicklern vorbehalten ist, sondern auch, was es für ein unsäglicher Akt war, endlich in den Genuss stabiler NVIDIA und ATI-Grafikkartentreiber für die volle DirectX 10-Unterstützung zu gelangen.

Theoretisch könnten wir Entwickler mit WPF, das, der Vollständigkeit halber erwähnt, übrigens schon auf Basis von DirectX 9 funktioniert, seit diesem Framework 3.0 bzw. Windows Vista auch anspruchsvolle Anwendungen entwickeln. Wenn, ja, wenn es da nicht ein kleines Problem gegeben hätte: Es gab zu diesem Zeitpunkt nämlich kein wirklich funktionierendes Tool, mit dem das Entwickeln möglich gewesen wäre. Lediglich ein hastig zusammengefickelter Aufsatz für Visual Studio 2005 musste zum Lernen und für die ersten Gehversuche herhalten; an eine richtige Entwicklung von WPF-Anwendungen war zu diesem Zeitpunkt noch nicht zu denken.

All diese Ereignisse der jüngeren Geschichte sorgten so dafür, dass wir erst mit Windows Vista SP1 und mit Visual Studio 2008 SP1 bzw. dem .NET Framework 3.5SP1 so richtig in WPF loslegen können. Naja, vielleicht nicht sofort. Denn nachdem wir das bislang gültige Konzept uns noch mal in Erinnerung gerufen haben, wie Grafik mit dem GDI bzw. GDI+ seinen Weg auf den Bildschirm findet, und was man machen muss, damit eine Grafik auch persistent bleibt (oder zumindest so wirkt), wollen wir meinen Kollegen jetzt auch noch mal mit der notwendigen Demo versorgen. Diese zeigt, wie man in der WPF einerseits Linien zeichnet, und wieso es um das Zeichnen von Linien mit der WPF, so, wie wir es vom GDI kennen, gar nicht gehen darf.

⁷ Englisch für *Benutzeroberfläche*, kurz: UI.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 06\\SimpleSampleWPF

Öffnen Sie dort die Projektmappe (.SLN-Datei) *SimpleSampleWPF*.

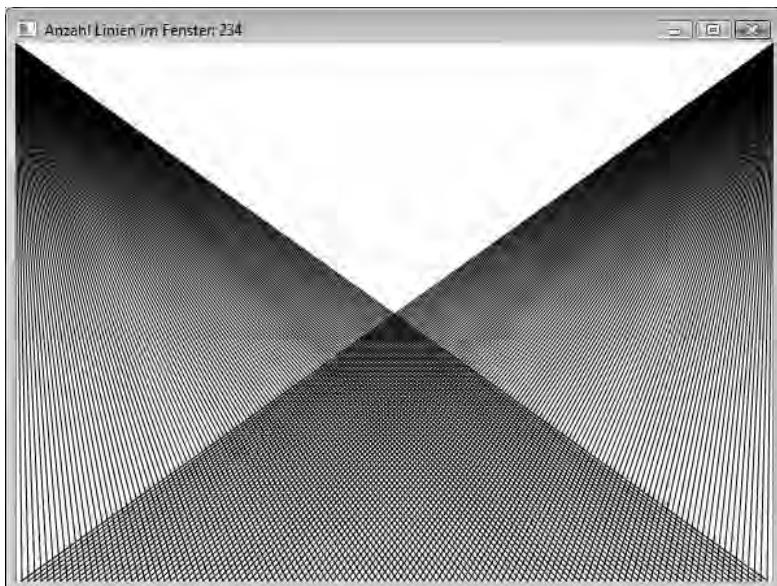


Abbildung 6.5 Die WPF-Version des Linienzaubers sieht erst mal der GDI-Version recht ähnlich ...

Auf den ersten Blick scheint das Ergebnis sich von der GDI+-Version nicht sonderlich zu unterscheiden. Die Linien scheinen irgendwie sauberer gezeichnet zu sein, und im Gegensatz zum GDI+ läuft das Zeichnen jedenfalls unter Vista ordentlicher, mit weniger Flimmern und subjektiv ein wenig schneller ab.

Erste wirkliche Prinzipunterschiede fallen aber bereits auf, wenn Sie das gleiche Spielchen mit diesem Fenster machen, wie wir es versuchsweise wie in Abbildung 6.4 gemacht haben. Hier geht beim Überlappen mit einem anderen Fenster nichts verloren – auch wenn der eigentliche Malvorgang, anders als im ersten Beispiel, im SizeChange-Ereignis des Fensters passiert, und dementsprechend für das Neuzeichnen des Inhaltes beim Vergrößern oder Verkleinern bereits gesorgt ist, wie das folgende Listing dieses Beispiels zeigt:

```
Imports System.Windows.Media.Animation

Class Window1

    Private Sub Window1_SizeChanged(ByVal sender As Object, ByVal e As
System.Windows.SizeChangedEventArgs) Handles Me.SizeChanged
        'meineLeinwand kommt aus der XAML-Definition
        'dort haben wir das vordefinierte Grid rausgeschmissen,
        'und durch die Leinwand (Canvas) ersetzt.
        meineLeinwand.Children.Clear()
```

```

'tatsächliche Breite der Leinwand ermitteln und merken
Dim tatsächlicheBreiteGemerkt = meineLeinwand.ActualWidth
Dim linienZähler = 0

'Jeden 5. Pixel berücksichtigen
For x = 0 To tatsächlicheBreiteGemerkt Step 5

    'Neues Linienobjekt anlegen, und Parameter setzen
    Dim eineLinie As New Line()
    'Parameter entsprechend setzen
    eineLinie.Stroke = Brushes.Black
    eineLinie.X1 = 0
    eineLinie.Y1 = 0
    eineLinie.X2 = x
    eineLinie.Y2 = meineLeinwand.ActualHeight
    eineLinie.HorizontalAlignment = HorizontalAlignment.Left
    eineLinie.VerticalAlignment = VerticalAlignment.Center
    eineLinie.StrokeThickness = 1
    'Der Leinwand hinzufügen
    meineLeinwand.Children.Add(eineLinie)

    'Die entgegenlaufende Linie definieren
    eineLinie = New Line()
    'Wieder Parameter entsprechend setzen
    eineLinie.Stroke = Brushes.Black
    eineLinie.X1 = tatsächlicheBreiteGemerkt
    eineLinie.Y1 = 0
    eineLinie.X2 = tatsächlicheBreiteGemerkt - x
    eineLinie.Y2 = meineLeinwand.ActualHeight
    eineLinie.HorizontalAlignment = HorizontalAlignment.Left
    eineLinie.VerticalAlignment = VerticalAlignment.Center
    eineLinie.StrokeThickness = 1
    'Und der Leinwand hinzufügen
    meineLeinwand.Children.Add(eineLinie)

    'Linienzähler aktualisieren
    linienZähler += 2
Next

Me.Title = "Anzahl Linien im Fenster: " & linienZähler
End Sub
.
.
.
End Class

```

Tatsache ist: Anders als beim GDI oder GDI+ ist das Gezeichnete hier persistent. Und das wird umso deutlicher, wenn Sie abermals einen Doppelklick in die Grafik machen, denn dann sehen Sie, dass jede einzelne Linie beginnt, zu animieren, und damit einen Effekt generiert, der sich wirklich sehen lassen kann:

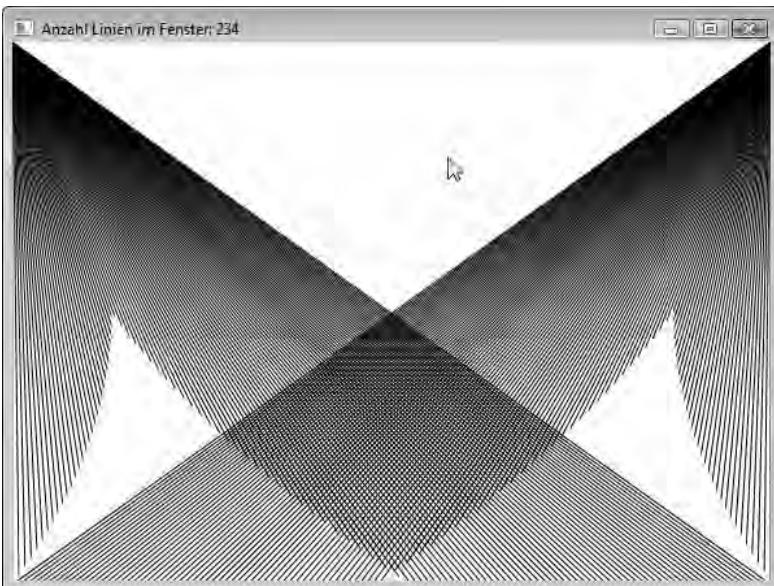


Abbildung 6.6 ...hinterlässt aber nach einem Doppelklick einen völlig anderen Eindruck, denn damit startet dann nämlich ein zwar einfacher Animationsvorgang, der aber jede einzelne der im Fenster 234 Linien betrifft!

Sie können nur ahnen, welchen Aufwand man im GDI+ bzw. GDI betreiben müsste, um diesen Animationseffekt so flimmerfrei wie hier im Beispiel hinzubekommen. Mit Doppelpufferung und Zwischenspeicherung des Anzeigebildes müsste man arbeiten, und man müsste die aktuellen Längen jeder einzelnen Linie zwischenspeichern und selbst kontrollieren. Nicht so in der WPF.

```
Private Sub Window1_MouseDoubleClick(ByVal sender As Object,_
    ByVal e As System.Windows.Input.MouseEventArgs) Handles Me.MouseDoubleClick
    Dim count = 0

    'Funktioniert natürlich nur, da sich nur
    'Linien auf dem Canvas-Objekt befinden.
    For Each lineItem As Line In meineLeinwand.Children
        'Y2-Eigenschaft wird animiert, daher der Name.
        'Die Klasse DoubleAnimation animiert Eigenschaften
        'vom Typ Double.

        'Animiert wird von der aktuellen Höhe der Linie
        'bis zur halben Höhe der Linie.
        Dim Y2Animation As New DoubleAnimation(lineItem.ActualHeight,
            lineItem.ActualHeight / 2, TimeSpan.FromSeconds(3))

        'Hier wird festgelegt, dass die Animation automatisch
        'wenn sie einmal komplett durchlaufen wurde.
        Y2Animation.RepeatBehavior = RepeatBehavior.Forever

        'Bestimmt, dass die DoubleAnimation hoch- und durch
        'AutoReverse=True anschließend wieder runterzählt.
        Y2Animation.AutoReverse = True
```

```

'Die Animation für jede weitere Linie wird um
'500 Millisekunden verzögert zur vorherigen begonnen,
Y2Animation.BeginTime = New TimeSpan(500000 * count)

'Animation starten ...
lineItem.BeginAnimation(Line.HeightProperty, Y2Animation)
count += 1
Next
End Sub

```

Jede einzelne Linie ist hier in der WPF nicht nur eine simple Methode, die bewirkt, dass Veränderungen am Grafikspeicher vorgenommen werden, dessen Repräsentation dann wieder den Eindruck einer gemalten Linie bewirkt. Vielmehr ist eine Linie, wie wir sie hier verwenden, abgeleitet von einer Klasse, die schon unfassbar viel an Infrastruktur mitbringt. Die nicht nur dafür sorgt, dass eine Instanz dieser Klasse sich zu gegebener Zeit rendert und aktualisiert, sondern die sich auch in eine Animationsinfrastruktur einfügt und sich damit, wie viele der meisten anderen Objekte, die die WPF als »sichtbares Ding«, also so genanntes Visual, zu verhalten und steuern vermag.

Alleine 234 filigrane Objekte auf diese Weise zu animieren erfordert schon einiges an Rechenleistung. Wenn dann 3D-Objekte die Bühne betreten, und die WPF ist ebenfalls in der Lage, auch diese zu verwalten, zu rendern, darzustellen und zu animieren, würde einem Prozessor ziemlich schnell die Puste ausgehen. Er ist hier auf das Mitwirken der Grafikkarte mehr als angewiesen, und zwar in Form von mehr als nur dem Malen von zweidimensionalen geometrischen Figuren oder einfachem *Bit-Blitting*, also Bitmap-Verschiebungen innerhalb des Grafikspeichers.

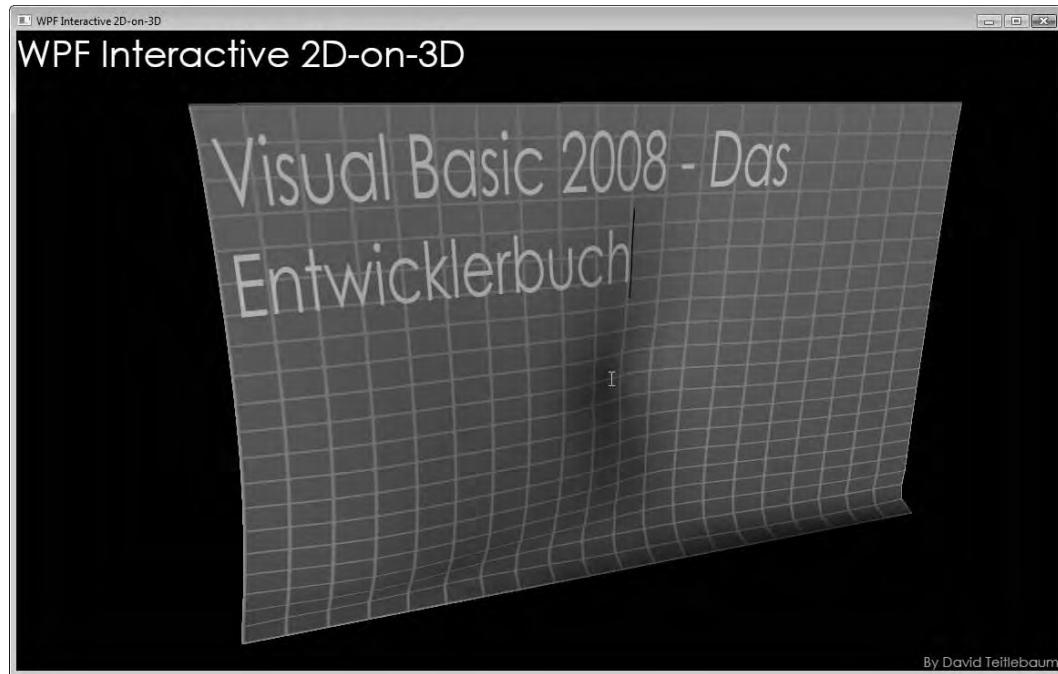


Abbildung 6.7 Eine TextBox, die auf eine animierte Stoffsimulation projiziert wird, und dabei voll funktionsfähig bleibt – eine der eindrucksvollsten WPF-Demos derzeit, da mit klugem Kopf aber minimalem Aufwand in die Tat umgesetzt.

Möglich wird das, weil die Grafikkarte hier aktiv in den Render-Prozess eingreift, und glauben Sie mir – Sie haben hier wirklich nur die Spitze des Eisbergs gesehen. Zu was die WPF seit dem Service Pack 1 von .NET 3.5 in der Lage ist, zeigt eine Demo von David Teitlebaum, die Sie unter dem IntelliLink **A0607** einsehen können, auf Channel 9 übrigens sehr eindrucksvoll. Die Demodateien können Sie übrigens über David Teitlebaums Blog abrufen – sie sind allerdings leider nur in C# verfügbar.

Wie WPF Designer und Entwickler zusammenbringt

Wenn Sie bisher eine Benutzerschnittstelle entworfen und programmiert haben, dann wurden normalerweise Design und Logik schnell vermischt. In einer Windows Forms-Anwendung wird in einer Klasse, die von `System.Windows.Forms.Form` abgeleitet wird, in der Methode `InitializeComponent` das Aussehen eines Fensters in Form von Visual Basic .NET-Code festgelegt. Nun hat ein Grafik-Designer mit Programmcode sehr wenig im Sinn. Der Grafiker benutzt diverse Werkzeuge, um schöne, bunte Grafiken zu erstellen. Er wird aber wohl kaum VB-Code schreiben wollen, um das Aussehen eines Fensters oder Steuerelements zu verändern. Andererseits ist es unmöglich, den logischen Teil der Applikation, also z.B. die Datenzugriffe oder das Berechnen von Zahlen mit einem Grafikwerkzeug durchzuführen.

Auf unserer Welt gibt es leider nur wenige Programmierer, die gleichzeitig auch hervorragende Designer sind. Das kann man natürlich auch anders herum betrachten: Es gibt kaum Top-Designer, die auch noch perfekt programmieren können.

Hier stoßen also zwei Welten aneinander, für die es gilt, eine für beide Seiten akzeptable Brücke zu errichten. Auf der einen Seite befindet sich der Grafiker mit seinen Werkzeugen, die irgend etwas abliefern, das der Softwareentwickler auf der anderen Seite mit der Applikationslogik »unterfüttern« kann. Dabei sollte sich der Grafiker so wenig wie möglich mit Programmierung auseinandersetzen müssen, und der Entwickler benötigt kaum eine Ahnung über die eingesetzten Designerwerkzeuge.

Ziel ist letztendlich, dass ein Top-Designer zusammen mit einem Top-Softwareentwickler eine Top-Applikation erstellt!

1. Beginnen wir im alten Stil und schreiben eine WPF-Applikation nur mit Visual Basic .NET-Code. Wir führen zunächst einmal keine Trennung von Design und Code ein. Es handelt sich natürlich um eine weitere Version des bekannten *HalloWelt*-Programms. Die Vorgehensweise ist folgende:
2. Starten Sie Visual Studio 2008 und wählen Sie *Datei > Neu > Projekt* aus.
3. Im Dialogfeld *Neues Projekt* wählen Sie als Programmiersprache Visual Basic aus. Darunter wählen Sie ein leeres »Windows«-Projekt aus.
4. Als Referenzen fügen Sie Verweise auf folgende Bibliotheken ein: `System`, `WindowBase`, `PresentationFramework` und `PresentationCore`. Benutzen Sie hierzu im Projektmappen-Explorer durch Anklicken mit der rechten Maustaste den Befehl *Verweis hinzufügen*. Wählen Sie die angegebenen Referenzen im Dialogfeld aus.
5. Fügen Sie nun im Projektmappen-Explorer eine neue VB-Code-Datei mit dem Namen »*Hallo.vb*« hinzu. Benutzen Sie wieder die rechte Maustaste mit dem Befehl *Hinzufügen > Neues Element*.
6. Tippen Sie nun den Code des nächsten Listings ein.
7. In den Projekt-Eigenschaften stellen Sie auf der Seite *Anwendung* den Ausgabetyp *Windows-Application* ein.

- Führen Sie das Programm aus, indem Sie aus dem Menü *Debuggen* den Befehl *Starten ohne Debugging* aufrufen.

```

Imports System
Imports System.Windows

Namespace Hallo

    Public Class Hallo
        Inherits System.Windows.Window

        <STAThread()>
        Shared Sub main()
            Dim w As New Window With {.Title = "Hallo, Welt!", .Width = "200", .Height = "200"}
            w.Show()

            Dim app As New Application
            app.Run()
        End Sub

    End Class
End Namespace

```

Wenn wir das Programm laufen lassen, erscheint ein ziemlich eintöniges Fenster mit dem Titel »Hallo, Welt!«.



Abbildung 6.8 Das erste WPF-Fenster

Schauen wir uns den Code des Beispiels nun etwas genauer an. Nach dem Einfügen der benötigten Namensräume aus den Referenzen definieren wir einen eigenen Namensraum mit der Klasse »Hallo«. In WPF muss vor der statischen Main-Methode das Attribut `[STAThread]` angewendet werden, um das Threading-Modell für die Applikation auf »Single-Threaded Appartement« zu setzen. Dies ist ein Relikt aus alten COM-Zeiten, aber es sorgt ggf. für eine Kompatibilität in die Welt des *Component Object Model*.

In der Main-Methode erzeugen wir zunächst ein `Window`-Objekt, welches über die Eigenschaft `Title` den Text für die Titelzeile erhält. Die Methode `Show` aus der `Window`-Klasse sorgt für die Darstellung des Fensters auf dem Bildschirm. Wenn wir die beiden letzten Code-Zeilen der Main-Methode nicht eingeben und unser Programm starten, dann werden wir das Fenster nur sehr, sehr kurz sehen. Die Applikation wird nämlich

sofort wieder beendet. Hier kommen nun diese beiden letzten Zeilen ins Spiel. Im erzeugten Application-Objekt wird die Methode Run aufgerufen, um für dieses Hauptfenster eine Meldungsschleife (Message Loop) zu erzeugen. Nun kann unser kleines Programm Meldungen empfangen. Das Fenster bleibt solange sichtbar, bis diese Meldungsschleife beendet wird, z.B. durch Anklicken der Schließen-Schaltfläche oben rechts in der Titelzeile des Fensters.

Nun haben wir in diesem Beispiel allerdings den Designer arbeitslos gemacht, denn wir haben als Softwareentwickler das Design im VB-Code implementiert. Eigentlich besteht das Design dieser Applikation nur aus einer Zeile:

```
Dim w As New Window With {.Title = "Hallo, Welt!", .Width = "200", .Height = "200"}
```

Alles andere möchte ich in diesem einfachen Beispiel einmal als Applikationslogik bezeichnen. Da ein Designer jedoch nicht mit VB-Code oder anderen Programmiersprachen arbeiten will, muss eine andere »Sprache« her, mit der man Benutzeroberflächen und Grafiken »deklarieren« kann.

In Windows Presentation Foundation wird hierzu die Extensible Application Markup Language, sprich: »gsämmel«) benutzt.

XAML: Extensible Application Markup Language

XAML ist, wie XML, eine hierarchisch orientierte Beschreibungssprache. Wir können mit XAML Benutzeroberflächen oder Grafiken deklarieren. Kommen wir wieder zu unserem ersten Beispiel (Abbildung 6.8) zurück. Als nächstes deklarieren wir nun den Designer-Teil mit XAML. Vorab jedoch noch ein kleiner Hinweis: Ein Designer wird natürlich den XAML-Code nicht »von Hand« eintippen, so wie wir es nun im nächsten Beispiel machen. Er wird stattdessen Werkzeuge verwenden, welche das erstellte Design schließlich als XAML-Code zur Verfügung stellen. In folgenden Listing können Sie nun den erforderlichen XAML-Code für unser erstes Beispielprogramm sehen.

```
<Window x:Class="Hallo2.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo, Welt!"
    Height="250"
    Width="250"
    >
</Window>
```

Was wir hier sehen, ist nicht gerade überwältigend! Diese wenigen Zeilen XAML-Code machen im Grunde genommen nichts anderes, als ein Window-Objekt zu deklarieren, den Titel des Fensters auf »Hallo, Welt!« zu setzen und die Größe des Fensters auf 250 mal 250 Einheiten zu definieren. Hierzu werden die Eigenschaften Title, Width und Height auf die gewünschten Werte gesetzt.

Alle Eigenschaften in XAML werden mit Texten gefüllt. Darum müssen die Werte in Anführungszeichen gesetzt werden. Um XAML-Code zu nutzen, sollten Sie zwei Namensräume deklarieren:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Visual Studio 2008 fügt diese Namensräume automatisch in Ihre .NET Framework 3.5-Projekte in die XAML-Dateien ein.

Dieses Projekt wurde mit Visual Studio angelegt, und darum übernimmt Visual Studio auch den Rest der Arbeit. Im Hintergrund wird nämlich eine weitere Datei erzeugt, die VB-Code enthält und unter anderem ein Application-Objekt erstellt und die dazugehörige Run-Methode aufruft. Dieser, von Visual Studio erzeugte Code, soll uns aber gar nicht weiter interessieren. Wenn wir das Programm starten, werden wir wieder das gleiche Fenster wie im ersten Beispiel sehen.

Für das zweite Beispiel ist die Vorgehensweise mit Visual Studio 2008 folgende:

1. Starten Sie Visual Studio 2008 und wählen Sie *Datei > Neu > Projekt* im Datei-Menü aus.
2. Im Dialogfeld *Neues Projekt* wählen Sie als Programmiersprache Visual Basic aus. Darunter wählen Sie ein *WPF-Anwendung*-Projekt aus.
3. Öffnen Sie nun im Projektmappen-Explorer die XAML-Datei mit dem Namen *Window1.xaml*.
4. Tippen Sie nun den Code des letzten Listings ein. Ändern Sie den Code, welchen Visual Studio erzeugt hat, einfach ab.
5. Führen Sie das Programm aus, indem Sie aus dem Menü *Debuggen* den Befehl *Starten ohne Debugging* aufrufen.

Nun haben wir die Deklaration der Benutzerschnittstelle in XAML hinterlegt. Dieser XAML-Code kann durch Grafikwerkzeuge erzeugt werden.

Jetzt aber wieder zurück zum Programmieren. Wir kommen nun auf die Applikationslogik zurück. Im nächsten Beispiel wollen wir eine minimale Logik implementieren. Für die Benutzerschnittstelle verwenden wir wiederum XAML, für die Logik wird Visual Basic eingesetzt.

Das Beispiel soll zeigen, wie eine Benutzerschnittstelle und die Applikationslogik in einer WPF-Applikation zusammenarbeiten. Dazu deklarieren wir mitten im Fenster eine Schaltfläche in einer bestimmten Größe (siehe folgendes Listing). Das ist unsere Benutzerschnittstelle, die von einem Designer erstellt wurde.

```
<Window x:Class="Hallo3.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo 3" Height="200" Width="200">
    <Grid>
        <Button Click="OnClick" Width="120" Height="40" Margin="28,58,21,64"
            >Bitte anklicken!</Button>
    </Grid>
</Window>>
```

Innerhalb der Deklaration für das Hauptfenster wird ein weiteres WPF-Element definiert: die Schaltfläche. In XAML heißt dieses Element *Button*. Eine Schaltfläche hat natürlich Eigenschaften, die Sie setzen können. Hier wird die Breite (*Width*) der Schaltfläche mit 120 Einheiten angegeben und die Höhe (*Height*) mit 40 Einheiten. Der Text auf der Schaltfläche lautet: »Bitte anklicken!«. Wenn wir den XAML-Code eingeben und das Programm starten, passiert natürlich beim Klicken auf die Schaltfläche zunächst einmal gar nichts.

Wenn der Anwender des Programms auf die Schaltfläche klickt, dann soll jedoch ein Text ausgegeben werden. Sie ahnen, welcher Text? Natürlich »Hallo, Welt!«. Das ist die Applikationslogik, welche ein Softwareentwickler erstellt (siehe nächstes Listing). Die Verbindung zwischen den *Button*-Element und dem VB-

Code wird über ein Ereignis hergestellt. In XAML (vorheriges Listing) wird im Button-Element das Click-Ereignis benutzt, welches auf die Ereignismethode OnClick zeigt. Diese Methode wird nun in Visual Basic implementiert.

Dieses Projekt wurde ebenfalls mit Visual Studio 2008 erzeugt. Zunächst einmal werden nur die Namensräume System und System.Windows benötigt. Visual Studio erzeugt fast den gesamten Code aus dem folgenden Listing, wir müssen nur die Methode OnClick (im unteren Bereich) eingeben.

```
Imports System
Imports System.Windows

Namespace Hallo3
    Partial Public Class Window1
        Inherits System.Windows.Window

        ''' <summary>
        ''' Logik für dieses Beispiel
        ''' </summary>

        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnClick(ByVal sender As System.Object, ByVal e As _
                           System.Windows.RoutedEventArgs)
            MessageBox.Show("Hallo, Welt!")
        End Sub
    End Class
End Namespace
```

Im Konstruktor der Fensterklasse wird die Methode InitializeComponent aufgerufen. Auch der Code dieser Methode wird in einer separaten VB-Codedatei von Visual Studio generiert. Der VB-Befehl, der hier zum Einsatz kommt, um die verschiedenen Dateien für den Compiler korrekt zusammenzuführen, heißt partial. Der gesamte Code der Klasse kann auf mehrere VB-Dateien verteilt werden: Ein Teil der Klasse ist der Code aus dem Listing und der andere, unsichtbare Teil ist der von Visual Studio erzeugte Zusatzcode, der die Methode InitializeComponent und eine Art Abbildung des XAML-Codes in Visual Basic enthält. Beim Übersetzen werden beide Dateien zusammengefügt. Aus diesem Grund erhalten Sie auch eine Fehlermeldung vom Compiler, wenn Sie das Beispiel übersetzen, ohne vorher die Methode OnClick einzugeben.

Die Benutzerschnittstelle, die in XAML definiert wurde, wird übrigens nicht von einem XML-Interpreter abgearbeitet, also interpretiert. Das wäre sicherlich zu langsam. Auch XAML wird kompiliert. Letztendlich entsteht daraus ganz normaler MSIL-Code, wie wir ihn aus jedem .NET-Programm gewohnt sind. Dieser »Zwischen-Code«, der in der EXE-Datei steht, wird dann vom JIT-Compiler (Just-In-Time) zur Laufzeit des Programms in die Maschinensprache der jeweiligen Zielplattform übersetzt.

In die Ereignismethode werden zwei Parameter übergeben: Von Typ object bekommen wir die Variable sender, die uns angibt, welches Objekt das Ereignis ausgelöst hat. Der Parameter e vom Typ EventArgs gibt uns zusätzliche Daten an, die zu dem jeweiligen Ereignis gehören. Diese Art der Ereignisprogrammierung kennen wir schon aus Windows Forms. In der Methode selbst wird dann einfach die Methode MessageBox.Show mit dem gewünschten Text aufgerufen.

Das Ergebnis unserer Bemühungen zeigt Abbildung 6.9.



Abbildung 6.9 XAML und Logik werden ausgeführt

Wir sind aber noch nicht ganz am Ziel unserer Wünsche! Das ganze Programm nützt uns nichts, wenn wir die Objekte, die wir in XAML deklariert haben, nicht aus unserem Applikationscode manipulieren können. Die Schaltfläche *Bitte anklicken!* ist ein Element vom Typ Button und enthält diverse Eigenschaften und Methoden. Wie kann man diese nun aus dem VB-Code aufrufen?

Hierzu wollen wir das letzte Beispiel wieder ein bisschen ändern. Zunächst muss die Schaltfläche einen Namen bekommen. Dies erledigen wir durch Setzen der Name-Eigenschaft. Es handelt sich hierbei um die einzige Änderung am XAML-Code.

```
<Window x:Class="Hallo4.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo 4" Height="200" Width="200">
    <Button Name="btn" Click="OnClick" Width="120" Height="40"
        Margin="34,61,24,61">Bitte anklicken!</Button>
</Window>
```

Da die Schaltfläche nun einen Namen (btn) hat, können wir sie ganz normal aus dem VB-Code heraus ansprechen und die Eigenschaften neu setzen oder Methoden des Objektes btn aufrufen. Im Beispiel (nächstes Listing) ändern wir mit der Eigenschaft Content zunächst den Text, der auf der Schaltfläche ausgegeben wird. Schließlich werden die Fontgröße (FontSize) und die Vordergrundfarbe (Foreground) neu gesetzt. In allen Fällen wird vor der jeweiligen Eigenschaft der Name der Schaltfläche, der in XAML definiert wurde, angegeben.

```
Imports System
Imports System.Windows

Namespace Hallo4
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub
```

```
Private Sub OnClick(ByVal sender As System.Object,
                    ByVal e As System.Windows.RoutedEventArgs)
    MessageBox.Show("Hallo Welt!")

    ' Nach der Messagebox: Schaltfläche ändern
    With btn
        .Content = "Guten Tag!"
        .FontSize = 20
        .Foreground = Brushes.Red
    End With

End Sub
End Class
End Namespace
```

Abbildung 6.10 zeigt das Fenster nach dem Ausführen der Ereignismethode. Die Eigenschaften der Schaltfläche wurden entsprechend geändert, nachdem im MessageBox-Element die *OK*-Schaltfläche angeklickt wurde.



Abbildung 6.10 Darstellung des Fensters nach dem MessageBox.Show-Aufruf

Nun können wir »beide Richtungen« zwischen XAML und Code benutzen. Um aus XAML die Applikationslogik »aufzurufen«, verwenden wir Ereignisse. Diese stehen uns in den vielen WPF-Elementen in großer Anzahl zur Verfügung. Wir implementieren Ereignismethoden, welche die Applikationslogik enthalten. Um umgekehrt die in XAML deklarierten WPF-Elemente zur Laufzeit zu verändern, benutzen wir das ganz normale Objektmodell dieser Elemente und rufen die Eigenschaften und Methoden aus dem Programmcode auf. Die einzelnen Objekte werden durch ihre Namen identifiziert. Die in XAML deklarierten Eigenschaftswerte sind also die Initialeinstellungen bei der Darstellung der WPF-Elemente.

Wie eben bereits erwähnt wurde, stellt die XAML-Deklaration der Benutzerschnittstelle den Startzustand der Anwendung dar. Sie können allerdings, wenn erforderlich, die Startwerte sofort aus dem Programmcode ändern, indem Sie eine Methode für das Ereignis *Loaded* implementieren. Dies wird in einem Beispiel in den folgenden beiden Codelistings gezeigt.

HINWEIS Aus diesem Beispiel wurde der nicht benötigte VB-Code, der von den Visual Studio-Erweiterungen erzeugt wurde, entfernt, um das Listing etwas kürzer zu halten.

Für das Hauptfenster mit dem Klassennamen *Window1* wird das Ereignis *Loaded* an die Methode *OnLoaded* gebunden. Der Code, der in der entsprechenden VB-Methode implementiert ist, wird nun direkt nach dem Konstruktoraufzug von *Window1* ausgeführt. Sobald das Fenster auf dem Bildschirm erscheint, enthält es bereits die vergrößerte Schaltfläche, da die beiden Eigenschaften *Width* und *Height* in der Methode *OnLoaded*

entsprechend gesetzt wurden. Wird die Schaltfläche danach angeklickt, so wird die Ereignismethode `OnClick` aufgerufen und ausgeführt.

Zunächst der XAML-Code:

```
<Window x:Class="Hallo5.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo 5" Height="300" Width="300"
    Loaded="OnLoaded">
    <Button Name="btn" Click="OnClick" Width="120" Height="40"
        Margin="34,61,24,61">Bitte anklicken!</Button>
</Window>
```

Und der Visual Basic-Code:

```
Imports System
Imports System.Windows

Namespace Hallo5
    Partial Public Class Window1
        Inherits System.Windows.Window

        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnClick(ByVal sender As System.Object,
                            ByVal e As System.Windows.RoutedEventArgs)
            MessageBox.Show("Hallo Welt!")

            ' Nach der Messagebox: Schaltfläche ändern
            With btn
                .Content = "Guten Tag!"
                .FontSize = 20
                .Foreground = Brushes.Red
            End With
        End Sub

        Private Sub OnLoaded(ByVal sender As System.Object,
                            ByVal e As System.Windows.RoutedEventArgs)
            btn.Height = 80
            btn.Width = 150
        End Sub
    End Class
End Namespace
```



Abbildung 6.11 Nach der Ausführung der OnLoaded-Methode

WICHTIG Grundsätzlich können wir sagen, dass alles, was in XAML angegeben und deklariert werden kann, auch in VB.NET mithilfe von Programmcode erzeugt werden kann. Umgekehrt gilt das allerdings nicht!

Wir können nun das Design der letzten Beispielanwendung ändern, ohne den Applikationscode zu modifizieren. Dazu wollen wir etwas ziemlich Verrücktes machen: Die Schaltfläche soll schräg im Fenster stehen und wir benötigen dazu eine Transformation, die wir in Kapitel 43 noch ausführlich behandeln werden. Benutzen Sie den entsprechenden XAML-Code mit der Transformation im Moment einfach so, wie in folgendem Listing aufgeführt.

```
<Window x:Class="Hallo5a.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo 5a" Height="300" Width="300"
    Loaded="Window_Loaded">
    <Button Name="btn" Click="OnClick" Width="120" Height="40" Margin="34,61,24,61">
        Bitte anklicken!
        <Button.RenderTransform>
            <SkewTransform AngleX="10" />
        </Button.RenderTransform>
    </Button>
</Window>
```

Die Logik, die in Visual Basic implementiert wurde (Ereignismethoden `OnLoaded`, `OnClick`), bleibt unverändert und wird für dieses Beispiel gar nicht mehr aufgelistet. Wir benutzen weiterhin den Code aus dem letzten Listing. Wenn Sie das Programm starten, sehen Sie zunächst die schräge Schaltfläche (Abbildung 6.12 links); die Funktionalität hinter dieser Schaltfläche ist jedoch unverändert geblieben. Ein Anklicken löst die `MessageBox.Show`-Methode aus und danach werden die Eigenschaften der nun schräg liegenden Schaltfläche wie bisher geändert (Abbildung 6.12 rechts).

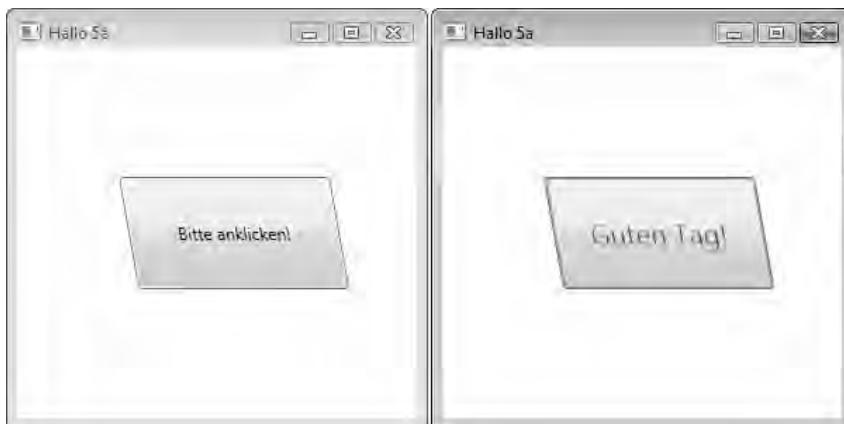


Abbildung 6.12
Eine Applikation mit
geändertem Design

Wir haben nun also eine Trennung zwischen dem Design der Applikation und der Logik erreicht. Der Softwareentwickler kann die Logik der Applikation ändern, ohne das Design (versehentlich) zu modifizieren. Umgekehrt kann ein Designer das Aussehen der Anwendung ändern, ohne dass der Programmcode ihm dabei ständig »in die Quere kommt«.

WICHTIG Wir können also sagen: XAML + Code = Anwendung.

Der WPF-Designer

Natürlich müssten Sie eine Anwendung nicht unbedingt komplett codieren – bislang haben wir ja sowohl Programmlogik als auch den XAML-Designpart in guter alter Entwicklermanier mit wenn auch verschiedenen text-basierten Editoren erstellt.

Zumindest beim Design-Part war das in Windows Forms anders. Wann immer es galt, Windows Forms-Anwendungen zu gestalten, war der WinForms-Designer von der Partie – sollte das nicht in WPF genau so sein? Sollte man meinen. Tatsache ist, dass der WPF-Designer einen WPF-Entwickler sicherlich in die Lage versetzt, rudimentärste WPF-Formulare zu erstellen. Wenn es aber darum geht, komplexere Objektverschachtelungen aus designtechnischer Sicht zu bauen, sind die Grenzen schnell erreicht. Den Komfort eines WinForms-Designers – beispielsweise beim interaktiven Erstellen von Werkzeugelementen oder Pulldown-Menüs – werden Sie beim derzeitigen Stand des WPF-Designers vergeblich suchen. Und von einer Unterstützung der viel weiter reichenden Fähigkeiten von WPF (Animationen, 3D) kann aus designtechnischer Sicht schon gar keine Rede sein.

Aus diesem Grund und der Tatsache, dass es aus didaktischer Sicht sicherlich besser ist, trittsicher in XAML zu werden, lassen wir den WPF-Designer in den folgenden Abschnitten und Kapiteln weitestgehend außen vor und bedienen uns nur seiner Split-View-Fähigkeit.

HINWEIS In einigen Fällen ist es notwendig, dass der Designer aufgrund von Codeänderungen seinen Inhalt neu laden muss – eine entsprechende Warnmeldung finden Sie dann am oberen Rand des Design-Fensters. Ein einfacher Mausklick genügt dann, um den Inhalt des Designfensters an die aktualisierten Gegebenheiten anzupassen.

Ereignisbehandlungsroutinen in WPF und Visual Basic

Im Zusammenhang mit dem Designer möchte ich auf das »Verdrahten« von Ereignisbehandlungs Routinen in WPF-Anwendungen ein kleines bisschen näher eingehen.

In Windows-Forms-Anwendungen sind Visual Basic-Entwickler es gewohnt, Ereignisprozeduren per Doppelklick auf das entsprechende Steuerelement zu verdrahten – das vorherige Kapitel hat gezeigt, wie es geht. Das geht in WPF-Anwendungen mit dem Designer prinzipiell ebenfalls. In beiden Fällen fügt Visual Basic dabei den Rumpf der Ereignisroutine mit einem geeigneten Namen ein, die automatisch die korrekte Signatur für das Behandeln des Ereignisses erhält. Das `Handles`-Schlüsselwort zeigt dem Visual Basic-Compiler an, dass hier die Infrastruktur für das Hinzufügen der Ereignisdelegatenliste in die Form- (bzw. – für WPF – Window-) Klasse eingefügt werden muss.

Bei WPF-Anwendungen gibt es in Ergänzung dazu auch eine Möglichkeit, die Sie auch bei der Entwicklung in jedem Fall vorziehen sollten, nämlich im XAML-Code selbst zu bestimmen, welche Ereignisprozedur für ein Ereignis eines bestimmten Objekts aufgerufen werden soll. Das funktioniert ähnlich einfach, wie das Doppelklicken auf ein Steuerelement, wie die nachstehende Abbildung zeigt:

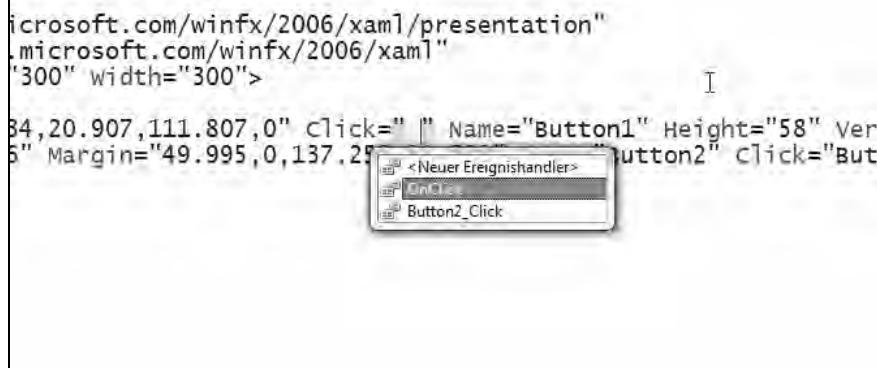


Abbildung 6.13 Ereignisprozeduren lassen sich in WPF auch in Visual Basic-Programmen, anders als Sie es von Windows-Forms-Anwendungen gewohnt sind, auch per XAML-Code »verdrahten«

Sie können den Namen des Ereignisses direkt in die XAML-Elementdefinition hineinschreiben, und IntelliSense hilft Ihnen anschließend, das Ereignis entweder mit einer bereits vorhandenen Methode (die natürlich über die entsprechende Signatur des Ereignisses verfügen muss) zu verknüpfen, oder den Rumpf einer neuen Ereignisbehandlungsroutine einzufügen. In letztem Fall würden Sie in der IntelliSense-Objektliste einfach den ersten Eintrag `<Neuer Ereignishandler>` auswählen.

Logischer und visueller Baum

In Windows Presentation Foundation gibt es zwei Hierarchien für die WPF-Elemente, die von großer Wichtigkeit sind. Beginnen wir mit dem logischen Baum (*Logical Tree*). Der logische Baum bildet den Zusammenhang zwischen den Objekten ab. Diese Hierarchie ist entscheidend für die Vererbung von Eigenschaften zwischen den Elementen. Innerhalb dieser Hierarchie gibt es verschiedene Methoden, um durch den Baum zu navigieren:

- GetParent
- GetChildren
- FindLogicalNode

Die drei genannten statischen Methoden finden Sie in der Klasse LogicalTreeHelper.

```
<Window x:Class="LogicalTree.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="LogicalTree" Height="300" Width="300"
    >
    <Grid>
        <Button Name="btn" Width="200" Height="30" Click="OnClick">Hallo</Button>
    </Grid>
</Window>
```

Die logische Hierarchie für den XAML-Code ist einfach:

- Window1-Element
- Grid-Element
- Button-Element

Diese Hierarchie können wir mit den Methoden GetParent und GetChildren durchlaufen. Die Methoden geben Objekte vom Typ DependencyObject zurück, wie das folgende Listing zeigt:

```
Imports System
Imports System.Windows
Imports System.Windows.Controls

Namespace LogicalTree
    Partial Public Class Window1
        Inherits System.Windows.Window

        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub onClick(ByVal sender As System.Object,
                           ByVal e As System.Windows.RoutedEventArgs)

            ' Richtung Wurzel-Element das Eltern-Element
            Dim Grid As DependencyObject
            Grid = LogicalTreeHelper.GetParent(CType(sender, FrameworkElement))
            MessageBox.Show(Grid.GetType().ToString())

            ' Richtung Wurzel-Element das nächste Eltern-Element
            Dim Window As DependencyObject
            Window = LogicalTreeHelper.GetParent(Grid)
            MessageBox.Show(Window.GetType().ToString())

            ' Alle Kindelemente des Grids
            Dim obj As DependencyObject
            For Each obj In LogicalTreeHelper.GetChildren(Grid)
```

```
    MessageBox.Show(obj.GetType().ToString())
Next

' Ein bestimmtes Element suchen
Dim depobj As DependencyObject
depobj = LogicalTreeHelper.FindLogicalNode(Window, "btn")

Dim btn As New Button
btn = CType(depobj, Button)
btn.Content = "WPF"
End Sub
End Class
End Namespace
```

Hier können Sie sehen, wie die Hierarchie abgearbeitet werden kann. In diesem einfachen Fall geben wir die Typen aus der Hierarchie nur in Meldungsfenstern aus. Im zweiten Teil des VB-Codes wird ermittelt, welche Kindelemente es im Grid-Element gibt. Hier wird nur ein Element, nämlich die Schaltfläche, gefunden. Im dritten Teil des Beispiels wird ein bestimmtes Element mit der Methode `LogicalTreeHelper.FindLogicalNode` aufgesucht. Das gefundene Objekt wird in ein `Button`-Element konvertiert. Dann können wir das Objekt benutzen und verschiedene Eigenschaften ändern.

Die zweite wichtige Hierarchie, die visuelle Hierarchie (Visual Tree) entscheidet darüber, wie die einzelnen Elemente »gerendert«, also dargestellt werden. Eine Schaltfläche ist nicht einfach nur eine Bitmap, die in der Grafikkarte dargestellt wird, sondern besteht aus einem Rechteck mit abgerundeten Ecken, einem Hintergrund und einem Inhalt (der wiederum beliebig kompliziert gestaltet sein kann). Die visuelle Hierarchie entscheidet also über die Darstellung der einzelnen Teile eines gesamten Elements. Hier wird auch die Reihenfolge in der Z-Richtung berücksichtigt, d.h., wie die Elemente übereinander liegen. Weiterhin spielt die visuelle Hierarchie für das Hit-Testing und die Transformationen der WPF-Elemente eine große Rolle.

Die XAML-Syntax im Überblick

Dieser Abschnitt gibt Ihnen einen kurzen Überblick über die XAML-Syntax. Der XAML-Code soll auch mit normalem VB-Code verglichen werden. Viele Leser werden sicherlich schon Erfahrung mit XML gesammelt haben. Trotzdem soll hier mit einigen Beispielen die Syntax von XAML erläutert werden.

Eine XAML-Hierarchie beginnt immer mit einem Wurzelement, einem `Window`- oder `Page`-Element. Dort werden die benötigten XML-Namensräume definiert.

Nun wird die Hierarchie der Benutzerschnittstelle deklariert. Alle Elemente, die in XAML benutzt werden können, existieren als normale Klassen im .NET Framework 3.0. Als Programmierer wissen wir, dass Klassen unter anderem Eigenschaften, Methoden und Ereignisse enthalten. Wir wollen nun betrachten, wie sich das in XAML verhält.

```
<Button Name="btnClear" Width="80" Height="25" Click="OnClear">Löschen</Button>
```

In der obigen XAML-Zeile wird ein Element vom Typ `Button` mit dem Namen `btnClear` deklariert. Zur Laufzeit wird also ein Objekt vom Typ `Button` erzeugt. Die Eigenschaften `Width` und `Height` werden gesetzt und außerdem wird das `Click`-Ereignis mit der Ereignismethode `OnClear` verbunden. Der Text auf der Schaltfläche wird über die `Default`-Eigenschaft des `Button`-Elements gesetzt. Das Gleiche könnten Sie mit folgendem VB-Code erreichen:

```
Dim btnClear As New Button
btnClear.Width = 80
btnClear.Height = 25
btnClear.Content = "Löschen"
AddHandler btnClear.Click, AddressOf OnClear
Me.Content = btnClear
```

Auf Grund des Inhaltsmodells von WPF müssen Sie in der letzten VB-Zeile die Schaltfläche als Inhalt in das Elternelement einbringen. In XAML wird das einfach durch die Deklarationshierarchie erledigt:

```
<Window ...
  Width="300" Height="300">
  <!-- Die folgende Schaltfläche ist der Inhalt des Fensters -->
  <Button Click="OnClear">Test</Button>
</Window>
```

Die Ereignismethode `OnClear` wird im VB-Code implementiert und hat normalerweise folgende Kopfzeile:

```
Public Sub OnClear(ByVal sender As System.Object,
                  ByVal e As System.Windows.RoutedEventArgs)
```

Angehängte Eigenschaften (*attached properties*) werden Sie dann benutzen, wenn Sie auf Eigenschaften des Elternelements zugreifen müssen:

```
<Grid>
  ...
  <Button Grid.Row="0" Grid.Column="0">Button 1</Button>
  <Button Grid.Row="1" Grid.Column="0">Button 2</Button>
</Grid>
```

In den obigen Zeilen wird von den `Button`-Deklarationen aus auf die Eigenschaften `Row` und `Column` des äußeren `Grid`-Elements zugegriffen.

Oftmals werden Sie in XAML Elemente deklarieren, ohne deren Standorteigenschaft zu benutzen. In diesem Fall können Sie eine gekürzte Schreibweise benutzen. Statt

```
<TextBox Name="text" Width="100"></TextBox>
```

können Sie auch schreiben

```
<TextBox Name="text" Width="100" />
```

Häufig müssen Sie einer Eigenschaft nicht einfach nur eine Zahl oder einen Text, sondern ein komplexes Element, welches wiederum eigene Eigenschaften besitzt, zuweisen. Als Beispiel wollen wir der Eigenschaft `RenderTransform` einer Schaltfläche (Button) ein Element vom Typ `RotateTransform` zuweisen, für welches die Eigenschaften `Angle`, `CenterX` und `CenterY` gesetzt werden sollen:

```
<Button Name="btnClear" Width="80" Height="25" Click="OnClear">
    Löschen
    <Button.RenderTransform>
        <RotateTransform Angle="25" CenterX="40" CenterY="12.5" />
    </Button.RenderTransform>
</Button>
```

Zunächst werden für die Schaltfläche selbst einige Eigenschaften »direkt« gesetzt (`Name`, `Width`, ...). In der `Button`-Klasse gibt es die Eigenschaft `RenderTransform`, der nun ein `RotateTransform`-Objekt zugewiesen werden soll. Darum wird innerhalb der Hierarchie das `RotateTransform`-Element deklariert und die Eigenschaften `Angle`, `CenterX` und `CenterY` werden gesetzt. Der entsprechende VB-Code sieht folgendermaßen aus:

```
' Button-Element erzeugen
Dim btnClear As New Button
btnClear.Width = 80
btnClear.Height = 25
btnClear.Content = "Löschen"
AddHandler btnClear.Click, AddressOf OnClear

' Rotation erzeugen
Dim rot As New RotateTransform
rot.Angle = 25
rot.CenterX = 40
rot.CenterY = 12.5

' Rotation der Schaltfläche zuweisen
btnClear.RenderTransform = rot
Me.Content = btnClear
```

Diese Hierarchien können beliebig tief geschachtelt werden. Wie Sie am letzten Beispiel sehen können, ist XAML bei der Definition von Hierarchien meistens wesentlich einfacher und übersichtlicher als eine normale Programmiersprache.

Oft gibt es in den WPF-Objekten Eigenschaften, denen Sie mehrere Elemente eines Typs zuweisen können. In einer `ListBox` können Sie mehrere `ListBoxItem`-Elemente einfügen:

```
<ListBox>
    <ListBox.Items>
        <ListBoxItem>Test1</ListBoxItem>
        <ListBoxItem>Test2</ListBoxItem>
        <ListBoxItem>Test3</ListBoxItem>
    </ListBox.Items>
</ListBox>
```

Die drei `ListBoxItem`-Elemente werden in einem Collection-Objekt angelegt und der `ListBox`-Eigenschaft `Items` zugewiesen. In dem gezeigten Fall gibt es auch noch eine einfachere Schreibweise:

```
<ListBox>
    <ListBoxItem>Test1</ListBoxItem>
    <ListBoxItem>Test2</ListBoxItem>
    <ListBoxItem>Test3</ListBoxItem>
</ListBox>
```

Zum Vergleich auch hier der passende VB-Code zur Erzeugung der `ListBox`:

```
Dim lb As New ListBox
Dim item As New ListBoxItem
item.Content = "Test1"
lb.Items.Add(item)
item = New ListBoxItem
item.Content = "Test2"
lb.Items.Add(item)
item = New ListBoxItem
item.Content = "Test3"
lb.Items.Add(item)
Me.Content = lb
```

Mit XAML können Sie oft sehr einfach ganze Listen von Objekten deklarieren und zuweisen. Im folgenden Beispiel wird ein `MeshGeometry3D`-Element mit Daten initialisiert:

```
<MeshGeometry3D Positions="0,0,0 5,0,0 0,0,5"
                  TriangleIndices="0 2 1"
                  Normals="0,1,0 0,1,0 0,1,0" />
```

In diesem Beispiel wird die Eigenschaft `Positions` mit drei Elementen vom Typ `Point3D` initialisiert. Jedes `Point3D`-Element wird wiederum mit drei double-Zahlen initialisiert, die jeweils durch Kommata getrennt in der Liste angegeben werden. Ganz ähnlich wird die Eigenschaft `Normals` gesetzt. Für die Eigenschaft `TriangleIndices` umfasst die Liste nur drei Zahlen, die einfach hinzugefügt werden. Der VB-Code zu diesem XAML-Beispiel sieht folgendermaßen aus:

```
Dim mesh As New MeshGeometry3D
mesh.Positions.Add(New Point3D(0.0, 0.0, 0.0))
mesh.Positions.Add(New Point3D(5.0, 0.0, 0.0))
mesh.Positions.Add(New Point3D(0.0, 0.0, 5.0))
mesh.TriangleIndices.Add(0)
mesh.TriangleIndices.Add(2)
mesh.TriangleIndices.Add(1)
mesh.Normals.Add(New Vector3D(0.0, 1.0, 0.0))
mesh.Normals.Add(New Vector3D(0.0, 1.0, 0.0))
mesh.Normals.Add(New Vector3D(0.0, 1.0, 0.0))
```

Wie Sie in diesem Beispiel leicht erkennen können, gibt es für die Eigenschaften `Positions`, `TriangleIndices` und `Normals` der `MeshGeometry3D`-Klasse immer eine `Add`-Methode, welche die Daten aus den XAML-Listenangaben korrekt verarbeitet und hinzufügt.

Mit diesen wenigen Beispielen haben wir die wichtigsten Syntaxelemente von XAML kennen gelernt und sollten in der Lage sein, die XAML-Hierarchien in diesem Buch zu lesen und zu verstehen. Mit etwas Übung werden Sie dann auch eigenen XAML-Code erstellen können.

HINWEIS Denken Sie daran, dass in Zukunft die XAML-Hierarchien nicht »von Hand« eingegeben, sondern von grafisch orientierten Werkzeugen erzeugt werden (wenn auch nicht gerade vom eingebauten WPF-Designer – Microsoft stellt aber weitere Werkzeuge zur Verfügung, über die Sie sich unter <http://www.microsoft.com/expression/> informieren können).

Ein eigenes XAMLPad

Im Windows-Software Development Kit (SDK) wird ein kleines Werkzeug mitgeliefert, welches eine XAML-Hierarchie in einem Fenster darstellen kann. Das Werkzeug heißt XAMLPad (Abbildung 6.14). Wir wollen zum Schluss dieses Kapitels versuchen, unser eigenes *MeinXAMLPad* zu entwickeln.

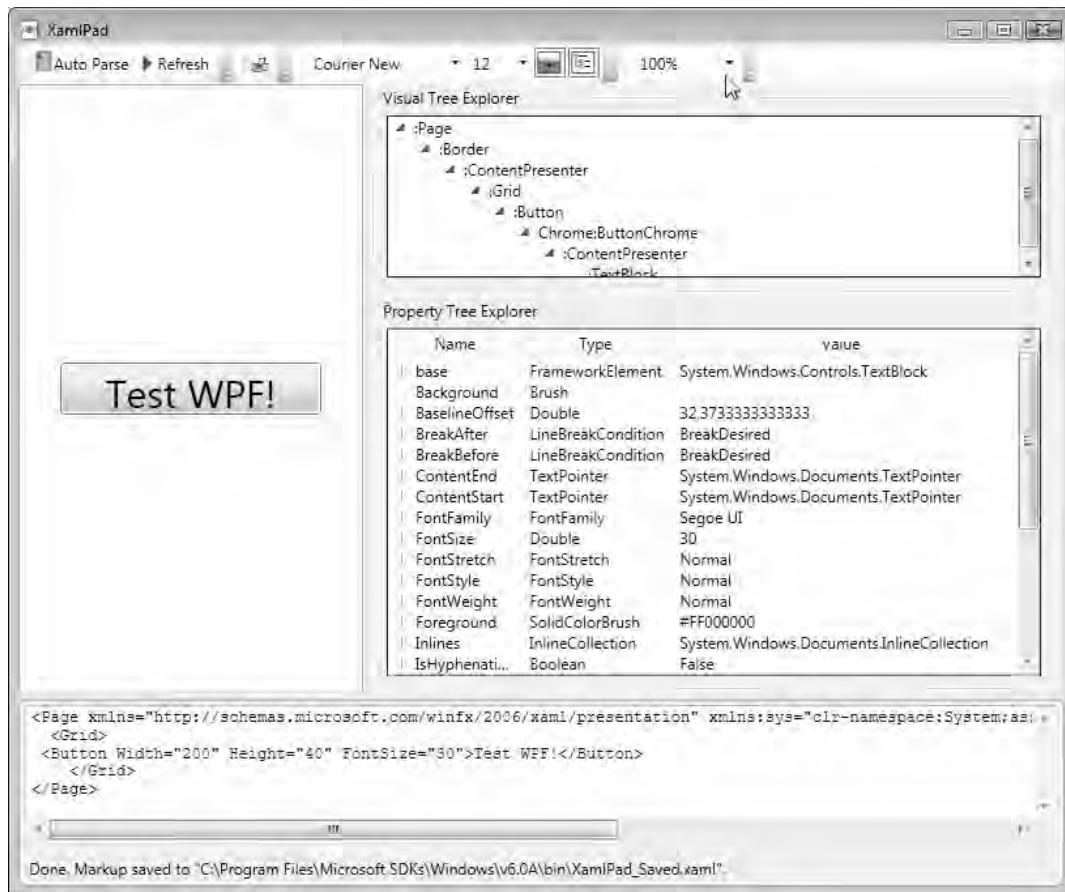


Abbildung 6.14 Das XAMLPad aus dem Windows Software Development Kit

In Abbildung 6.14 wird im unteren Bereich des Fensters eine XAML-Hierarchie eingegeben, die im oberen Bereich visualisiert wird. Das Programm eignet sich gut, um ein bisschen mit XAML zu experimentieren und die Hierarchien für ein WPF-Element zu erforschen. Es eignet sich allerdings nicht dazu, WPF-Applikationen zu entwickeln und zu testen. Sie können mit diesem Werkzeug keinen VB-Code definieren, der z.B. als Ereignismethode für das Click-Ereignis der Schaltfläche aufgerufen werden kann.

Zunächst wollen wir eine einfache Benutzerschnittstelle wie in folgendem Listing definieren.

```
<Window x:Class="MeinXAMLPad.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MeinXAMLPad" Height="500" Width="600"
    >
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition Height="40"/>
    </Grid.RowDefinitions>

    <Grid Name="gridCode" Grid.Column="0" Grid.Row="0" />

    <TextBox Grid.Column="0" Grid.Row="1" Name="textCode"
        VerticalScrollBarVisibility="Visible" HorizontalScrollBarVisibility="Visible"
        FontFamily="Lucida Console" FontSize="14" AcceptsReturn="True" />

    <StackPanel Grid.Column="0" Grid.Row="2" Orientation="Horizontal">
        <Button Name="btnClear" Width="80" Margin="5" Click="OnClear">Löschen</Button>
        <Button Name="btnCompile" Width="80" Margin="5" Click="OnCompile">Darstellen</Button>
        <Button Name="btnExit" Width="80" Margin="5" Click="OnExit">Beenden</Button>
    </StackPanel>
</Grid>
</Window>
```

Die hier verwendeten WPF-Steuerelemente (Grid, StackPanel, Button und TextBox) werden wir an späterer Stelle noch genauer kennen lernen. Das verwendete Grid-Element enthält in unserem Beispiel drei Zellen untereinander. Jede Zelle kann weitere WPF-Elemente enthalten. Die oberste Zelle enthält wiederum ein Grid-Element mit dem Namen gridCode. Dieses Grid wird später die WPF-Elemente darstellen, welche wir im darunter liegenden Textfeld eingeben.

Die Texteingabe wird über ein mehrzeiliges TextBox-Element realisiert. Um aus dem VB-Code auf das Element zugreifen zu können, vergeben wir auch hier einen Namen: textCode. Außerdem setzen wir einige weitere Eigenschaften, um die Eingabe etwas zu vereinfachen. Wichtig ist, die Eigenschaft AcceptsReturn auf True zu setzen, damit wir problemlos mehrzeilige Texte eingeben können. Weiterhin können Sie die Benutzbarkeit der Bildlaufleiste (Scrollbar) mit den Eigenschaften VerticalScrollBarVisibility und HorizontalScrollBarVisibility einschalten. Eine andere Schriftart (FontFamily) mit einer etwas größeren FontSize rundet das Bild ab.

Im unteren Teil des Fensters legen wir nun ein StackPanel-Element mit horizontaler Anordnung an. Hier befinden sich mehrere Schaltflächen, die *MeinXAMLPad* steuern. Alle Schaltflächen sind mit Ereignismethoden verbunden, deren Funktion sich aus dem Namen ergibt. Die interessanteste Schaltfläche mit dem Namen btnCompile sorgt für die Übersetzung und Darstellung des eingegebenen XAML-Codes. Der dazugehörige VB-Code ist in folgendem Listing zu sehen.

```
Imports System
Imports System.Windows
Imports System.Windows.Markup
Imports System.Xml
Imports System.IO

Namespace MeinXAMLPad
    Partial Public Class Window1
        Inherits Windows.Window

        'Das folgende Element enthält den Code für unser WPF-Element
        Dim uie As UIElement
        Dim code As UIElement
        Public Sub New()
            uie = Nothing
            InitializeComponent()
            InitTextBox()
        End Sub

        Private Sub InitTextBox()
            'Starttext in die Textbox schreiben
            textCode.Text =
                "<Grid xmlns=""http://schemas.microsoft.com/winfx/2006/xaml/presentation"" _"
                & vbCrLf
            textCode.AppendText(
                (" " xmlns:x=""http://schemas.microsoft.com/winfx/2006/xaml"" _"
                & vbCrLf)
            textCode.AppendText(" >" & vbCrLf & vbCrLf & "</Grid>" & vbCrLf)
            textCode.Focus()
        End Sub

        Private Sub OnClear()
            If Not uie Is Nothing Then
                'Existierende WPF-Elemente entfernen
                gridCode.Children.Remove(uie)
                uie = Nothing
            End If
            InitTextBox()
        End Sub

        Private Sub OnCompile()
            'WPF-Element in Grid darstellen
            If Not uie Is Nothing Then
                gridCode.Children.Remove(uie)
            End If

            'WPF-Code aus TextBox übersetzen
            uie = GetTheCode(textCode.Text)

            If Not uie Is Nothing Then
                'WPF im Grid darstellen
                gridCode.Children.Add(uie)
            End If
        End Sub
    End Class
End Namespace
```

```

Private Function GetTheCode(ByVal strText As String) As UIElement
    uie = Nothing
    Try
        ' Stream aus Eingabetext erzeugen
        Dim sr As StringReader = New StringReader(strText)
        ' Text aus der Box als XmlReader darstellen
        Dim xr As XmlReader = XmlReader.Create(sr)
        ' Laden und übersetzen des XAML-Codes, Zuordnen zum UIElement
        code = CType(XmlReader.Load(xr), UIElement)
    Catch ex As Exception
        'Einfachste Fehlerbehandlung
        MessageBox.Show("Fehler im XAML-Code:" _
            & vbCrLf & vbCrLf & ex.Message)
    End Try
    Return code
End Function

Private Sub OnExit()
    Me.Close()
End Sub
End Class
End Namespace

```

Bei der Initialisierung der Applikation wird aus dem Konstruktor der Fensterklasse die Methode `InitTextBox` aufgerufen. Hier wird nur ein XAML-Grundgerüst mit den erforderlichen Namensräumen in das `TextBox`-Element geschrieben, damit wir beim Testen nicht so viel tippen müssen. Der Code für die Schaltfläche *Beenden* ist eigentlich klar. Hier schließen wir einfach das Hauptfenster der Applikation.

Kommen wir nun zur Ereignismethode `OnCompile`, die natürlich die Hauptarbeit macht. Der übersetzte XAML-Code wird in einem `UIElement` gespeichert, welches in der Fensterklasse deklariert wird. Das Objekt `UIElement` liegt in der WPF-Klassenhierarchie ziemlich weit oben:

- System.Object
- System.Windows.Threading.DispatcherObject
- System.Windows.DependencyObject
- System.Windows.Media.Visual
- **System.Windows.UIElement**
 - System.Windows.FrameworkElement
 - System.Windows.Controls.Control
 - System.Windows.Controls.ContentControl
 - System.Windows.Controls.Primitives.ButtonBase
 - System.Windows.Controls.Button

Sie sehen in der obigen Liste, dass ein normales WPF-Steuerellement (hier: `Button`-Element) von `UIElement` abgeleitet wird, sodass wir in unserem Beispiel ein Objekt von diesem Typ für die Speicherung des eingegebenen XAML-Segments gut benutzen können. Wenn bereits ein `UIElement` dargestellt wurde, dann wird dieses Element in `OnCompile` zunächst gelöscht, indem für das Kindelement in `gridCode` die `Remove`-Methode aufgerufen wird. Nun kann der eingegebene XAML-Code über die Methode `GetTheCode` verarbeitet werden. In dieser

Methode wird der Text aus dem Eingabefeld `textCode` als `StringReader`-Element geöffnet. Der erzeugte Datenstrom wird nun als `XmlReader` mit der `Create`-Methode geöffnet. Diese Schritte sind erforderlich, da das nun verwendete `XamlReader`-Element XML als Eingabe für die `Load`-Methode benötigt. Dieser Methodenaufruf im `XamlReader` erzeugt nun das `UIElement`, welches schließlich als Kindelement im Grid `gridCode` dargestellt wird.

Wenn Sie die Schaltfläche *Löschen* anklicken, passieren in der Ereignismethode `OnClear` zwei Dinge. Sollten bereits WPF-Elemente im Grid-Element `gridCode` dargestellt werden, dann müssen diese mit der `Remove`-Methode gelöscht werden. Nun sehen Sie wieder ein leeres Ausgabefenster; das Element `gridCode` enthält kein `UIElement` mehr. Außerdem wird das `TextBox`-Element wieder initialisiert.

MeinXAMLPad ist nun fertig. Der Aufruf der Applikation zeigt uns Abbildung 6.15. Im Eingabefeld unserer Applikation haben wir ein `StackPanel`-Element mit einer Schaltfläche, einem Eingabefeld und einem Kontrollkästchen eingegeben. Natürlich ist die Deklaration von Benutzerschnittstellen mit unserem Werkzeug nicht so angenehm. Zum einen vermissen wir natürlich die IntelliSense-Möglichkeit aus Visual Studio 2008, außerdem können wir keine Ereignismethoden definieren und der XAML-Code wird in nacktem Schwarz dargestellt. Wenn Sie Spaß daran haben, können Sie dieses Beispiel natürlich weiter entwickeln.



Abbildung 6.15 MeinXAML-Pad in der Anwendung

Zusammenfassung

In diesem ersten WPF-Kapitel haben Sie die Grundlagen von Windows Presentation Foundation kennen gelernt. Sie wissen nun, dass Code und Design getrennt voneinander vorliegen. Der Code wird in Visual Basic implementiert, während das Design der Benutzeroberfläche in XAML deklariert wird.

Alle WPF-Elemente, egal ob Steuerelemente oder grafische Figuren, werden als Vektorgrafik ausgegeben. Dadurch wird immer eine hervorragende Darstellungsqualität erzielt, auch wenn Elemente der Benutzerschnittstelle extrem vergrößert werden.

Die Verbindung von Design und Code wird über den Aufruf von Ereignismethoden aus XAML bzw. über die mit Namen versehenen Elemente aus dem Programmcode vollzogen.

Wenn Sie mit WPF arbeiten, wird niemals XAML-Code zur Laufzeit »interpretiert«. Letztendlich liegt auch die mit XAML deklarierte Benutzeroberfläche als MSIL-Code in der auszuführenden Datei vor. Dieser wird dann zur Laufzeit vom JIT-Compiler der Common Language Runtime in Maschinencode übersetzt. Somit sollten WPF-Benutzerschnittstellen ähnlich schnell wie bereits bekannte Frameworks arbeiten.

Die Trennung von Code und Logik mit WPF bringt uns folgenden großen Vorteil: Sie bringen einen Top-Designer mit einem Top-Softwareentwickler zusammen und die beiden entwickeln für Sie eine Top-Software!

Soviel zu einer ersten Einführung in die WPF. Detaillierte Informationen zu Vorgehensweise bei der Programmierung in der Windows Presentation Foundation liefern Ihnen dann Kapitel 41 bis 43 im WPF-Teil dieses Buchs.

Kapitel 7

Die essentiellen .NET-Datentypen

In diesem Kapitel:

Numerische Datentypen	208
Der Datentyp Char	230
Der Datentyp String	231
Der Datentyp Boolean	254
Der Datentyp Date	255
.NET-Äquivalente primitiver Datentypen	263

Sie wissen nun spätestens seit dem 2. Kapitel, was Variablen sind und dass es unterschiedliche Typen gibt. Ein paar der .NET-Datentypen haben Sie in diesem Beispiel auch schon kennen gelernt. Allerdings fehlen bislang immer noch genaue Informationen über das Konzept einiger spezieller Datentypen, die in .NET fest integriert sind, und die Sie nahezu ständig beim Entwickeln eigener Applikationen verwenden (müssen).

Das sind vor allem die schon erwähnten primitiven Datentypen, wie Integer, Double, Date, String, etc. Sie sind in den .NET-Sprachen von C# oder Visual Basic fest verankert, was Sie im Übrigen auch daran feststellen können, dass der Editor sie blau markiert, wenn Sie die Schlüsselwörter dieser Typen ausgeschrieben haben.

Unter primitive Datentypen fallen alle Typen, die in einer Programmiersprache unter .NET fest im Sprachschatz verankert sind. Das heißt genauer:

- **Konstanter Wert:** Ein konstanter Wert jedes primitiven Datentyps kann in Schriftform angegeben werden. Die Angabe 123.324D bezeichnet so beispielsweise einen Wert bestimmter Größe vom Typ Decimal.
- **Primitiver Datentyp als Konstante:** Es ist möglich, einen primitiven Datentyp als Konstante zu deklarieren. Wenn ein bestimmter Ausdruck ausschließlich als Konstante definiert wird (also beispielsweise beim Ausdruck 123.32D*2+100.23D), kann er schon beim Kompilieren ausgewertet werden.
- **Funktionsdelegierung an den Prozessor:** Viele Operationen und Funktionen bestimmter primitiver Datentypen können vom Framework zur Ausführung direkt an den Prozessor delegiert werden. Dann ist keine Programmlogik erforderlich, um einen arithmetischen Ausdruck zu berechnen (beispielsweise eine Fließkommazahlendivision), der Prozessor kann das vielmehr selbstständig und damit natürlich sehr schnell. Dazu gehören die meisten der Operationen der Datentypen Byte, Short, Integer, Long, Single, Double und Boolean.

Numerische Datentypen

Für die Verarbeitung von Zahlen bietet Ihnen Visual Basic die primitiven Datentypen Byte, Short, Integer, Long, Single, Double und Decimal an. Neu seit Visual Basic 2005 waren darüber hinaus die Datentypen SByte, UShort, UInteger und ULong. Sie unterscheiden sich durch den Wertebereich, den sie abdecken können, die Präzision, mit der sie rechnen (Anzahl Nachkommastellen – auch Skalierung genannt) und den Speicherbedarf, den sie benötigen.

HINWEIS Die sogenannten Nullable-Datentypen sind ebenfalls seit Visual Basic 2005 bekannt. Nullable-Datentypen verhalten sich wie Ihre primitiven Datentypenpendants, haben aber auch die Möglichkeit, einen nicht definierten Zustand, nämlich *Null*¹, *Nichts* oder in Visual Basic *Nothing* widerzuspiegeln). Neu mit Visual Basic 2008 ist allerdings, dass diese Datentypen sich nun auch in die Sprachsyntax von Visual Basic eingliedern, und diese mit dem Fragezeichen als Typliteral definiert werden. Um einen Integer-Datentyp also als Nullable zu definieren, würde man wie folgt vorgehen:

```
Dim t As Integer?
```

Mehr zum Thema Nullable-Datentypen finden Sie im Kapitel 11.

¹ Nicht mit dem Wert 0 verwechseln! Null (englisch ausgesprochen *Nall*) kommt damit eher dem »Zustand« *Nichts* nahe. Im Englischen würde man die Zahl 0 auch nicht engl. *Null* (*Nall*) sondern eher *Zero* oder *Ought* aussprechen (*Oht* oder oft auch einfach *Oh* ausgesprochen).

Numerische Datentypen deklarieren und definieren

Alle numerischen Datentypen werden wie alle primitiven Datentypen ohne das Schlüsselwort `New` deklariert; Zuweisungen von konstanten Werten können direkt im Programm erfolgen, wobei es bestimmte Markierungszeichen gibt, von welchem Typ ein konstanter Wert ist, der durch eine Ziffernfolge angegeben wird. Eine Variable vom Typ `Double` kann beispielsweise mit der Anweisung

```
Dim locDouble As Double
```

deklariert und sofort verwendet werden.

Numerische Datentypen werden mit im Programmcode verankerten Konstanten definiert, indem Sie ihnen eine Ziffernfolge zuweisen, der im Bedarfsfall das Typliteral folgt, wie im folgenden Beispiel das `D` hinter der eigentlichen Konstanten:

```
locDouble = 123.3D
```

Genau wie andere primitive Datentypen können Deklaration und Zuweisung in einer Anweisung erfolgen. So könnten Sie natürlich die beiden oben stehenden einzelnen Anweisungen durch die folgende ersetzen:

```
Dim locDouble As Double = 123.3D
```

Das hier gezeigte Beispiel gilt für alle anderen numerischen Datentypen äquivalent – wobei sich die Typlitale natürlich von Typ zu Typ unterscheiden können.

Typename	Typzeichen	Typliteral	Beispiel
Byte	-	-	<code>Dim var As Byte = 128</code>
SByte	-	-	<code>Dim var As SByte = -5</code>
Short	-	S	<code>Dim var As Short = -32700S</code>
UShort	-	US	<code>Dim var As UShort = 65000US</code>
Integer	%	I	<code>Dim var% = -123I oder</code> <code>Dim var As Integer = -123I</code>
UInteger	-	UI	<code>Dim var As UInteger = 123UI</code>
Long	&	L	<code>Dim var& = -123123L oder</code> <code>Dim var As Long = -123123L</code>
ULong	-	UL	<code>Dim var As ULong = 123123UL</code>
Single	!	F	<code>Dim var! = 123.4F oder</code> <code>Dim var As Single = 123.4F</code>
Double	#	R	<code>Dim var# = 123.456789R oder</code> <code>Dim var As Double = 123.456789R</code>
Decimal	@	D	<code>Dim var@ = 123.456789123D oder</code> <code>Dim var As Decimal = 123.456789123D</code>

Typename	Typzeichen	Typliteral	Beispiel
Boolean	–	–	Dim var As Boolean = True
Char	–	C	Dim var As Char = "A" C
Date	–	#dd/MM/yyyy HH:mm:ss# oder #dd/mm/yyyy hh:mm:ss am/pm#	Dim var As Date = #12/24/2008 04:30:15 PM#
Object	–	–	In einer Variable vom Typ Object kann jeder beliebige Typ gekapselt („boxed“) oder mit dieser referenziert werden.
String	\$	"Zeichenkette"	Dim var\$ = "Zeichenkette" oder Dim var As String = "Zeichenkette"

Tabelle 7.1 Typliterale und Variablenotypzeichen der primitiven Datentypen in Visual Basic 2008

Delegation numerischer Berechnungen an den Prozessor

Die Eigenschaft, einige mathematische Operationen dem Prozessor direkt zu überlassen, zeigt das folgende Beispiel eindrucksvoll. Dazu müssen Sie wissen: Der `Decimal`-Typ wird, anders als `Double` oder `Single`, auf Grund seiner Rechengenauigkeit nicht alleine durch die Fließkommaeinheit des Prozessors, sondern durch entsprechenden Programmcode der Base Class Library berechnet.

HINWEIS Das folgende Beispiel steigt ein wenig tiefer ins System ein – Sie lernen dabei auch, wie Sie die Assembler- bzw. Maschinensprachenrepräsentation, also das konkrete Kompilat Ihres Programms, so, wie es der Prozessor sieht, betrachten können. Interessant ist das allemal, und es hilft auch beim Verständnis, wie Daten vom Prozessor verarbeitet werden – notwendiges Wissen zum Entwickeln eigener Programme ist es nicht wirklich.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

... \VB 2008 Entwicklerbuch\A – Einführung\Kapitel 07\Primitives01

Öffnen Sie dort die Projektmappe (.SLN-Datei) *Primitives01*.

Bevor Sie dann den folgenden Beispielcode ausführen, setzen Sie mit der Taste `F9` einen Haltepunkt in der folgenden, fett markierten Codezeile:

```
Public Class Primitives
    Public Shared Sub main()
        Dim locDouble1, locDouble2 As Double
        Dim locDec1, locDec2 As Decimal

locDouble1 = 123.434D
        locDouble2 = 321.121D
        locDouble2 += 1
        locDouble1 += locDouble2
        Console.WriteLine("Ergebnis der Double-Berechnung: {0}", locDouble1)
```

```

    locDec1 = 123.434D
    locDec2 = 321.121D
    locDec2 += 1
    locDec1 += locDec2
    Console.WriteLine("Ergebnis der Double-Berechnung: {0}", locDec1)

End Sub
End Class

```

Wenn Sie dieses Programm anschließend starten, unterbricht das Programm in der Zeile, in der Sie zuvor den Haltepunkt gesetzt haben. Wählen Sie aus dem Menü *Debuggen/Fenster* den Punkt *Disassembly*. Mithilfe dieses Fensters können Sie sehen, was der JITter² aus dem zunächst in die IML kompilierten Programm gemacht hat:

```

Dim locDouble1, locDouble2 As Double
Dim locDec1, locDec2 As Decimal

locDouble1 = 123.434D
00000055 movsd      xmm0,mmword ptr [000002E8h]
0000005d movsd      mmword ptr [rsp+50h],xmm0
    locDouble2 = 321.121D
00000063 movsd      xmm0,mmword ptr [000002F0h]
0000006b movsd      mmword ptr [rsp+58h],xmm0
    locDouble2 += 1
00000071 movsd      xmm0,mmword ptr [000002F8h]
00000079 addsd      xmm0,mmword ptr [rsp+58h]
0000007f movsd      mmword ptr [rsp+58h],xmm0

```

Hier werden, entgegen einer möglichen Erwartung, keine speziellen Methoden der Double-Struktur aufgerufen; die Addition wird vielmehr durch die Fließkommafunktionalität des – in diesem Fall – 64-Prozessors (**addsd**,³ im Listing fett markiert) selbst erledigt. Ganz anders ist das weiter unten in der Disassembly, wenn die gleichen Operationen mit dem Decimal-Datentyp ausgeführt werden:

```

locDec2 += 1
0000017e mov        rcx,129F1180h
00000188 mov        rcx,qword ptr [rcx]
0000018b add        rcx,8
0000018f mov        rax,qword ptr [rcx]
00000192 mov        qword ptr [rsp+000000A8h],rax
0000019a mov        rax,qword ptr [rcx+8]
0000019e mov        qword ptr [rsp+000000B0h],rax
000001a6 lea        rcx,[rsp+000000A8h]

```

² Das Disassembly-Fenster kann übrigens nur Assembler-Code anzeigen, der nicht sehr gut optimiert ist. Daran ändert auch keine andere Einstellung etwas – sehen Sie immer nur den Debug- und nicht den optimierten Code. Im späteren optimierten Code werden, wann immer es möglich ist, die Prozessorregister als Träger lokaler Variablen genutzt, was die Laufgeschwindigkeit Ihrer Anwendungen natürlich drastisch erhöht!

³ *Scalar Double-Precision Floating-Point Add*, in etwa: *Skalare Fließkommaaddition mit doppelter Genauigkeit*.

```

000001ae mov rax,qword ptr [rcx]
000001b1 mov qword ptr [rsp+000000E0h],rax
000001b9 mov rax,qword ptr [rcx+8]
000001bd mov qword ptr [rsp+000000E8h],rax
000001c5 lea rcx,[rsp+40h]
000001ca mov rax,qword ptr [rcx]
000001cd mov qword ptr [rsp+000000D0h],rax
000001d5 mov rax,qword ptr [rcx+8]
000001d9 mov qword ptr [rsp+000000D8h],rax
000001e1 lea r8,[rsp+000000E0h]
000001e9 lea rdx,[rsp+000000D0h]
000001f1 lea rcx,[rsp+000000B8h]
000001f9 call FFFFFFFF381460 // Hier wird die Additionsroutine von ...
000001fe mov qword ptr [rsp+00000128h],rax
00000206 lea rcx,[rsp+000000B8h]
0000020e mov rax,qword ptr [rcx]
00000211 mov qword ptr [rsp+40h],rax
00000216 mov rax,qword ptr [rcx+8]
0000021a mov qword ptr [rsp+48h],rax
    locDec1 += locDec2
0000021f lea rcx,[rsp+40h]
00000224 mov rax,qword ptr [rcx]
00000227 mov qword ptr [rsp+00000110h],rax
0000022f mov rax,qword ptr [rcx+8]
00000233 mov qword ptr [rsp+00000118h],rax
0000023b lea rcx,[rsp+30h]
00000240 mov rax,qword ptr [rcx]
00000243 mov qword ptr [rsp+00000100h],rax
0000024b mov rax,qword ptr [rcx+8]
0000024f mov qword ptr [rsp+00000108h],rax
00000257 lea r8,[rsp+00000110h]
0000025f lea rdx,[rsp+00000100h]
00000267 lea rcx,[rsp+000000F0h]
0000026f call FFFFFFFF381460 // ... Decimal aufgerufen. Hier auch.
.
.
.

```

Ungleich mehr Vorbereitungen zur Addition sind hier nötig, da die notwendigen Operanden zunächst auf den Stack kopiert werden müssen. Und hier wird die eigentliche Addition auch nicht durch den Prozessor selbst erledigt, sondern durch entsprechende Routinen der BCL, die, im Disassembly zu sehen, mit Call aufgerufen werden (im Listing fett markiert).

TIPP Das ist übrigens auch der Grund, weshalb die Performance des Decimal-Datentyps auch nur etwa einem Zehntel der Performance des Double-Datentyps entspricht. Decimal sollten Sie nur dann einsetzen, wenn Sie absolut genaue Berechnungen durchführen müssen und sich keine Rundungsfehler erlauben können (lesen Sie dazu bitte auch die Ausführungen im Abschnitt »Tabellarische Zusammenfassung der numerischen Datentypen« auf Seite 220).

Hinweis zur CLS-Konformität

Einige der in Visual Basic 2005 eingeführten Datentypen entsprechen nicht der so genannten CLS-Compliance⁴. Dazu gehören sowohl generische Datentypen als auch einige primitive Datentypen, die vorzeichenlose Integerwerte speichern (sowie der primitive Datentyp SByte). Methoden, die Typen benötigen oder zurückliefern, die als nicht *CLS-Compliant* gelten, sollten nicht offen gelegt werden, also nicht in Komponenten zur Verfügung gestellt werden, die auch andere .NET-Programmiersprachen verwenden können. Sie dürfen nämlich nicht davon ausgehen, dass diese Typen auch in allen .NET-Programmiersprachen »erreichbar« sind. Visual Basic .NET prüft übrigens nicht automatisch auf CLS-Konformität. Falls Sie Komponenten vom VB-Compiler auf Konformität überprüfen lassen wollen, können Sie das so genannte *CLSCopliant*-Attribut auf Klassen-Ebene einsetzen, also etwa so:

```
<CLSCopliant(True)>
Public Class EineKlasse
    Private myMember As UShort
    Public Property NichtCLSEntsprechend() As UShort
        Get
            Return myMember
        End Get
        Set(ByVal value As UShort)
            myMember = value
        End Set
    End Property
End Class
```

Falls Sie eine einzelne Methode auf CLS-Compliance überprüfen wollen, funktioniert das äquivalent:

```
<CLSCopliant(True)>
Public Shared Function EineNichtCLSCopliantMethode() As UShort
    Dim locTest As ClassLibrary1.EineKlasse
    locTest = New ClassLibrary1.EineKlasse
End Function
```

HINWEIS Entgegen vieler landläufiger Meinungen ist es nicht richtig, dass eine Methode, eine Assembly oder gar Ihre gesamte Anwendung nicht CLS-konform ist, wenn Sie nur einen nicht-konformen Typen verwenden. Nicht konform wird Ihre Assembly nur dann, wenn Sie nicht-CLS-konforme Typen exportieren, also einen solchen entweder von einer öffentlichen Methode mit Rückgabewert zurückgeben oder ihn als Parameter erwarten.

Die numerischen Datentypen auf einen Blick

Die Verwendung numerischer Datentypen und ihre darstellbaren Wertebereiche finden Sie in den folgenden kurzen Abschnitten beschrieben.

⁴ Etwa: »Entsprechung der Common Language Specification«.

HINWEIS Der Punkt *Arithmetik durch den Prozessor* in der nachstehenden Aufstellung sagt aus, dass bestimmte arithmetische oder boolesche Operationen nicht durch Prozeduren des Frameworks, sondern durch den Prozessor ausgeführt werden; das äußert sich in einer äußerst schnellen Verarbeitung.⁵

Byte

.NET-Datentyp: System.Byte

Stellt dar: Integer-Werte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich

Wertebereich: 0 bis 255

Typliteral: nicht vorhanden

Speicherbedarf: 1 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einByte As Byte  
einByte = 123
```

Anmerkung: Dieser Datentyp speichert nur vorzeichenlose, positive Zahlen im angegebenen Zahlenbereich.

CLS-Compliant: ja

Konvertierung von anderen Zahlentypen: CByte(objVar) oder Convert.ToByte(objVar)

```
einByte = CByte(123.45D)  
einByte = Convert.ToByte(123.45D)
```

SByte

.NET-Datentyp: System.SByte

Stellt dar: Integer-Werte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich

Wertebereich: -128 bis 127

Typliteral: nicht vorhanden

Speicherbedarf: 1 Byte

Arithmetik durch den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einByte As SByte  
einByte = 123
```

Anmerkung: Dieser Datentyp speichert negative und positive Zahlen im angegebenen Zahlenbereich.

⁵ Diese Aussage gilt unter Umständen *nicht* für eine andere als die Intel Pentium Plattform. Auf Pocket-PCs beispielsweise, auf denen Applikationen unter .NET ebenfalls entwickelt werden können, kann sich das unter Umständen anders verhalten.

CLS-Compliant: nein

Konvertierung von anderen Zahlentypen: CSByte(objVar) oder Convert.ToSByte(objVar)

```
einByte = CByte(123.45D)
einByte = Convert.ToByte(123.45D)
```

Short

.NET-Datentyp: System.Int16

Stellt dar: Integer-Werte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich

Wertebereich: -32.768 bis 32.767

Typliteral: S

Speicherbedarf: 2 Byte

Delegation an den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einShort As Short
einShort = 123S
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlenbereich. Bei der Konvertierung in den Datentyp Byte kann durch den größeren Wertebereich von Short eine OutOfRangeException erzeugt werden.

CLS-Compliant: ja

Konvertierung von anderen Zahlentypen: CShort(objVar) oder Convert.ToInt16(objVar)

```
'Nachkommastellen werden abgeschnitten
einShort = CShort(123.45D)
einShort = Convert.ToInt16(123.45D)
```

UShort

.NET-Datentyp: System.UInt16

Stellt dar: positive Integer-Werte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich

Wertebereich: 0 bis 65.535

Typliteral: US

Speicherbedarf: 2 Byte

Delegation an den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einUShort As UShort
einUShort = 123US
```

Anmerkung: Dieser Datentyp speichert vorzeichenlose, also nur positive Zahlen im angegebenen Zahlbereich. Bei der Konvertierung in den Datentyp Byte oder Short kann durch den (teilweise) größeren Wertebereich von UInt16 eine OutOfRangeException erzeugt werden.

CLS-Compliant: nein

Konvertierung von anderen Zahlentypen: CUInt16(objVar) oder Convert.ToInt16(objVar)

```
'Nachkommastellen werden abgeschnitten
einUShort = CUInt16(123.45D)
einUShort = Convert.ToInt16(123.45D)
```

Integer

.NET-Datentyp: System.Int32

Stellt dar: Integer-Werte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich

Wertebereich: -2.147.483.648 bis 2.147.483.647

Typliteral: I

Speicherbedarf: 4 Byte

Delegation an den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einInteger As Integer
Dim einAndererInteger%      ' auch als Integer deklariert
einInteger = 123I
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlbereich. Bei der Konvertierung in die Datentypen Byte, Short und UInt16 kann durch den größeren Wertebereich von Integer eine OutOfRangeException erzeugt werden. Durch Anhängen des Zeichens »%« an eine Variable kann der Integer-Typ für die Variable erzwungen werden (darauf sollten Sie allerdings zugunsten eines besseren Programmierstils lieber verzichten).

CLS-Compliant: ja

Konvertierung von anderen Zahlentypen: CInt(objVar) oder Convert.ToInt32(objVar)

```
einInteger = CInt(123.45D)
einInteger = Convert.ToInt32(123.45D)
```

UInteger

.NET-Datentyp: System.UInt32

Stellt dar: positive Integer-Werte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich

Wertebereich: 0 bis 4.294.967.295

Typliteral: UI

Speicherbedarf: 4 Byte

Delegation an den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einUInteger As UInteger  
einUInteger = 123UI
```

Anmerkung: Dieser Datentyp speichert vorzeichenlose, also nur positive Zahlen im angegebenen Zahlensbereich. Bei der Konvertierung in die Datentypen Byte, Short, UShort und Integer kann durch den (teilweise) größeren Wertebereich von UInteger eine OutOfRangeException erzeugt werden.

CLS-Compliant: nein

Konvertierung von anderen Zahlentypen: CUInt(objVar) oder Convert.ToInt32(objVar)

```
einUInteger = CUInt(123.45D)  
einUInteger = Convert.ToInt32(123.45D)
```

Long

.NET-Datentyp: System.Int64

Stellt dar: Integer-Werte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich

Wertebereich: -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807

Typliteral: L

Speicherbedarf: 8 Byte

Delegation an den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einLong As Long  
Dim einAndererLong& ' auch als Long definiert  
einLong = 123L
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlensbereich. Bei der Konvertierung in alle anderen Integer-Datentypen kann durch den größeren Wertebereich von Long eine OutOfRangeException erzeugt werden. Durch Anhängen des Zeichens »&« an eine Variable kann der Long-Typ für die Variable erzwungen werden (darauf sollten Sie allerdings zugunsten eines besseren Programmierstils lieber verzichten).

CLS-Compliant: ja

Konvertierung von anderen Zahlentypen: CLng(objVar) oder Convert.ToInt64(objVar)

```
einLong = CLng(123.45D)  
einLong = Convert.ToInt64(123.45D)
```

ULong

.NET-Datentyp: System.UInt64

Stellt dar: positive Integer-Werte (Zahlen ohne Nachkommastellen) im angegebenen Wertebereich

Wertebereich: 0 bis 18.446.744.073.709.551.615

Typliteral: UL

Speicherbedarf: 8 Byte

Delegation an den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einULong As ULong
einULong = 123L
```

Anmerkung: Dieser Datentyp speichert vorzeichenlose, also nur positive Zahlen im angegebenen Zahlbereich. Bei der Konvertierung in alle anderen Integer-Datentypen kann durch den größeren Wertebereich von Long eine OutOfRangeException erzeugt werden.

CLS-Compliant: nein

Konvertierung von anderen Zahlentypen: CULng(objVar) oder Convert.ToInt64(objVar)

```
einULong = CULng(123.45D)
einULong = Convert.ToInt64(123.45D)
```

Single

.NET-Datentyp: System.Single

Stellt dar: Fließkommawerte (Zahlen mit Nachkommastellen, deren Skalierung⁶ mit Anwachsen des Wertes kleiner wird) im angegebenen Wertebereich

Wertebereich: Die Werte reichen von $-3,4028235 \cdot 10^{38}$ bis $-1,401298 \cdot 10^{-45}$ für negative Werte und von $1,401298 \cdot 10^{-45}$ bis $3,4028235 \cdot 10^{38}$ für positive Werte.

Typliteral: F

Speicherbedarf: 4 Byte

Delegation an den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einSingle As Single
Dim einAndererSingle! ' auch als Single definiert
einSingle = 123.0F
```

⁶ Skalierung in diesem Zusammenhang bezeichnet die Anzahl der Nachkommastellen einer Fließkommazahl.

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlenbereich. Durch Anhängen des Zeichens »!« an eine Variable kann der Single-Typ für die Variable erzwungen werden (darauf sollten Sie allerdings zugunsten eines besseren Programmierstils lieber verzichten).

CLS-Compliant: ja

Konvertierung von anderen Zahlentypen: CSng(objVar) oder Convert.ToSingle(objVar)

```
einSingle = CSng(123.45D)
einSingle = Convert.ToSingle(123.45D)
```

Double

.NET-Datentyp: System.Double

Stellt dar: Fließkomma-Werte (Zahlen mit Nachkommastellen, deren Skalierung mit Anwachsen des Wertes kleiner wird) im angegebenen Wertebereich

Wertebereich: Die Werte reichen von $-1,79769313486231570 \cdot 10^{308}$ bis

$-4,94065645841246544 \cdot 10^{-324}$ für negative Werte und von $4,94065645841246544 \cdot 10^{-324}$ bis
 $1,79769313486231570^{308}$ für positive Werte

Typliteral: R

Speicherbedarf: 8 Byte

Delegation an den Prozessor: ja

Deklaration und Beispielzuweisung:

```
Dim einDouble As Double
Dim einAndererDouble# ' auch als Double definiert
einDouble = 123.0R
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlenbereich. Durch Anhängen des Zeichens »#« an eine Variable kann der Double-Typ für die Variable erzwungen werden (darauf sollten Sie allerdings zugunsten eines besseren Programmierstils lieber verzichten).

CLS-Compliant: ja

Konvertierung von anderen Zahlentypen: CDb1(objVar) oder Convert.ToDouble(objVar)

```
einDouble = CDb1(123.45D)
einDouble = Convert.ToDouble(123.45D)
```

Decimal

.NET-Datentyp: System.Decimal

Stellt dar: Fließkomma-Werte (Zahlen mit Nachkommastellen, deren Skalierung mit Anwachsen des Wertes kleiner wird) im angegebenen Wertebereich

Wertebereich: Der Wertebereich hängt von der Anzahl der verwendeten Dezimalstellen ab. Werden keine Dezimalstellen verwendet – man spricht dabei von einer Skalierung von 0 –, liegen die maximalen/minimalen Werte zwischen +79.228.162.514.264.337.593.543.950.335. Bei der Verwendung einer maximalen Skalierung (28 Stellen hinter dem Komma – es können dann nur noch Werte zwischen >-1 und <+1 dargestellt werden) liegen die maximalen/minimalen Werte zwischen +0,99999999999999999999999999999999.

Typliteral: D

Speicherbedarf: 16 Byte

Delegation an den Prozessor: nein

Deklaration und Beispielzuweisung:

```
Dim einDecimal As Decimal
Dim einAndererDouble@ ' auch als Decimal definiert
einDecimal = 123.23D
```

Anmerkung: Dieser Datentyp speichert vorzeichenbehaftete, also negative und positive Zahlen im angegebenen Zahlenbereich. Durch Anhängen des Zeichens »@« an eine Variable kann der Decimal-Typ für die Variable erzwungen werden (darauf sollten Sie allerdings zugunsten eines besseren Programmierstils lieber verzichten). Wichtig: Bei sehr hohen Werten müssen Sie das Typliteral an eine Literalkonstante anhängen, um eine Overflow-Fehlermeldung zu vermeiden.

HINWEIS Bitte beachten Sie ebenfalls, dass beim Decimal-Datentyp keine Delegation arithmetischer Funktionen an den Prozessor erfolgt, dieser Datentyp also im Vergleich zu den Fließkommadatentypen Single und Double sehr viel langsamer verarbeitet wird. Gleichzeitig treten aber keine Rundungsfehler durch die interne Darstellung von Werten im Binärsystem auf. Darüber erfahren Sie im folgenden Abschnitt Näheres.

CLS-Compliant: ja

Konvertierung von anderen Zahlentypen: CDec(objVar) oder Convert.ToDecimal(objVar)

```
einDecimal = CDec(123.45F)
einDecimal = Convert.ToDecimal(123.45F)
```

Tabellarische Zusammenfassung der numerischen Datentypen

Typename	.NET-Typname	Aufgabe	Wertebereich
Byte	System.Byte	Speichert vorzeichenlose Integer-Werte mit 8 Bit Breite.	0 bis 255
SByte	System.SByte	Speichert vorzeichenbehaftete Integer-Werte mit 8 Bit Breite.	-127 bis 128
Short	System.Int16	Speichert vorzeichenbehaftete Integer-Werte mit 16 Bit (2 Byte) Breite.	-32.768 bis 32.767
UShort	System.UInt16	Speichert vorzeichenlose Integer-Werte mit 16 Bit (2 Byte) Breite.	0 bis 65.535

Typename	.NET-Typname	Aufgabe	Wertebereich
Integer	System.Int32	Speichert vorzeichenbehaftete Integer-Werte mit 32 Bit (4 Byte) Breite. HINWEIS: Auf 32-Bit-Systemen ist dies der am schnellsten verarbeitete Integer-Datentyp.	-2.147.483.648 bis 2.147.483.647
UInteger	System.UInt32	Speichert vorzeichenlose Integer-Werte mit 32 Bit (4 Byte) Breite.	0 bis 4.294.967.295
Long	System.Int64	Speichert vorzeichenbehaftete Integer-Werte mit 64 Bit (8 Byte) Breite.	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
ULong	System.UInt64	Speichert vorzeichenlose Integer-Werte mit 64 Bit (8 Byte) Breite.	0 bis 18.446.744.073.709.551.615
Single	System.Single	Speichert Fließkommazahlen mit einfacher Genauigkeit. Benötigt 4 Bytes zur Darstellung.	-3,4028235E+38 bis -1,401298E-45 für negative Werte; 1,401298E-45 bis 3,4028235E+38 für positive Werte
Double	System.Double	Speichert Fließkommazahlen mit doppelter Genauigkeit. Benötigt 8 Bytes zur Darstellung. HINWEIS: Dies ist der schnellste Datentyp zur Fließkommazahlberechnung, da er direkt an die Mathe-Einheit des Prozessors zur Berechnung delegiert wird.	1,79769313486231570E+308 bis -4,94065645841246544E-324 für negative Werte; 4,94065645841246544E-324 bis 1,79769313486231570E+308 für positive Werte
Decimal	System.Decimal	Speichert Fließkommazahlen im binärcodierten Dezimalformat. HINWEIS: Dies ist der langsamste Datentyp zur Fließkommazahlberechnung, seine besondere Speicherform schließt aber typische Computerrundungsfehler aus.	0 bis +/-79.228.162.514.264.337.593.543.950.335 (+/-7,9...E+28) ohne Dezimalzeichen; 0 bis +/-7,9228162514264337593543950335 mit 28 Stellen rechts vom Dezimalzeichen; kleinste Zahl ungleich 0 (null) ist +/-0,00000000000000000000000000000001 (+/-1E-28)

Tabelle 7.2 Die numerischen, primitiven Datentypen in .NET 2.0/3.5 bzw. VB2008

Rundungsfehler bei der Verwendung von Single und Double

Es ist eigentlich eine ganz normale Sache, dass ein bestimmtes Zahlensystem einige Brüche nicht genau darstellen kann. Dennoch gibt es immer wieder Programmierer, die glauben, einen Fehler in einer Programmiersprache gefunden zu haben, oder behaupten, der Computer könne nicht richtig rechnen. Dabei kennen Sie Rundungs- bzw. Konvertierungsfehler von einem Zahlensystem in das andere aus dem täglichen Leben auch beim 10er-System: Wenn Sie die Zahl 1 durch 3 teilen, erhalten Sie eine Zahl mit unendlichen Nachkommastellen, nämlich 0,333333333333. Im Dreiersystem ein Drittel darzustellen, benötigt wesentlich weniger Ziffern. Es ist schlicht 0,1.

Nun ist es ganz gleich, wie viele Ziffern Sie für die Darstellung eines für ein Zahlensystem problematischen Bruches verwenden; solange Sie eine endliche Anzahl von Ziffern in einem Zahlensystem verwenden, das einen Bruch nur periodisch darstellen kann, erhalten Sie beim Addieren dieser Zahlen Rundungsfehler.

Ein Beispiel: $3^*1/3$ im Dreiersystem führt zur Berechnung von:

0.1
+0.1
+0,1
=====

Und das entspricht im Dezimalsystem ebenfalls 1,0. Das Ausrechnen dieser Addition im Dezimalsystem ist ungenau, denn selbst wenn Sie über 60 Nachkommastellen für die Darstellung der Zahlen verwenden, so erreichen Sie in der Addition dennoch niemals den Wert 1:

Dieser Wert ist zwar ziemlich nah dran an 1, aber eben nicht ganz 1. Und wenn Sie mehrere Ergebnisse im Laufe einer Berechnung haben, können sich diese Darstellungsfehler schnell zu größeren Fehlern summieren, die auch irgendwann relevant werden.

Das gleiche Problem hat der Computer bei bestimmten Zahlen, wenn er im Binärsystem rechnet. Während wir beispielsweise die Zahl 69,82 im Dezimalsystem ganz genau mit einer endlichen Anzahl von Ziffern darstellen können, bekommt der Computer mit dem Binärsystem Probleme:

Die Umwandlung von 69 funktioniert noch einwandfrei, aber bei der 0,82 wird es schwierig:

Wenn Sie wissen, dass Nachkommastellen durch negative Potenzen der Basiszahl dargestellt werden, dann ergibt sich folgende Rechnung:

0.5	$1*2^{-1}$	Zwischenergebnis: 0.5
0.25	$1*2^{-2}$	Zwischenergebnis: 0.75
0.125	$1*2^{-3}$	Zwischenergebnis: 0.8125
0.0625	$0*2^{-4}$	Zwischenergebnis: 0.8125
0.03125	$0*2^{-5}$	Zwischenergebnis: 0.8125
0,015625	$0*2^{-6}$	Zwischenergebnis: 0.8125
0,0078125	$0*2^{-7}$	Zwischenergebnis: 0.8125
0,00390625	$1*2^{-8}$	Zwischenergebnis: 0.81640625
0,001953125	$1*2^{-9}$	Zwischenergebnis: 0.818359375
0,0009765625	$1*2^{-10}$	Zwischenergebnis: 0.8193359375
0,00048828125	$1*2^{-11}$	Zwischenergebnis: 0.81982421875

Wir sind inzwischen bei der Zahl 0,11100001111 angelangt und haben das gewünschte Ziel immer noch nicht erreicht. Die Wahrheit ist: Sie können dieses Spielchen bis in alle Ewigkeit weiterspielen. Sie werden die Zahl 0,82 des Dezimalsystems mit einer endlichen Anzahl an Ziffern im Binärsystem niemals darstellen können.

BEGLEITDATEIEN Was hat das für Auswirkungen auf die Programmierung unter Visual Basic? Nun, schauen Sie sich dazu einmal das folgende kleine Beispielprogramm an, das Sie in folgendem Verzeichnis finden.

...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 07\\Primitives02

```
Public Class Primitives

    Public Shared Sub main()

        Dim locDouble1, locDouble2 As Double
        Dim locDec1, locDec2 As Decimal

        locDouble1 = 69.82
        locDouble2 = 69.2
        locDouble2 += 0.62

        Console.WriteLine("Die Aussage locDouble1=locDouble2 ist {0}", locDouble1 = locDouble2)
        Console.WriteLine("aber locDouble1 lautet {0} und locDouble2 lautet {1}", _
                          locDouble1, locDouble2)

        locDec1 = 69.82D
        locDec2 = 69.2D
        locDec2 += 0.62D
        Console.WriteLine("Die Aussage locDec1=locDec2 ist {0}", locDec1 = locDec2)

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()

    End Sub

End Class
```

Auf den ersten Blick sollte man meinen, dass beide `WriteLine`-Methoden den gleichen Text ausgeben. Sie brauchen keinen Taschenrechner zu bemühen, um zu sehen, dass der erste Wert und damit die erste Variable innerhalb des Programms die Addition des zweiten und dritten Wertes darstellt und die beiden Variablenwerte aus diesem Grund gleich sein sollten. Leider ist dem nicht so. Während Sie mit dem `Decimal`-Datentyp im zweiten Teil des Programms den richtigen Wert herausbekommen, versagt der `Double`-Typ im ersten Part des Programms.

Noch verwirrender wird es, wenn Sie die zweite, auskommentierte `WriteLine`-Methode wieder ins Programm nehmen: Beide Variablen enthalten nämlich augenscheinlich den gleichen Wert, bis auf die letzte Nachkommastelle genau:

```
Die Aussage locDouble1=locDouble2 ist False
aber locDouble1 lautet 69,82 und locDouble2 lautet 69,82
Die Aussage locDec1=locDec2 ist True

Taste drücken zum Beenden!
```

Das Geheimnis darum ist aber schnell gelüftet: Bei der Umwandlung in eine Zeichenkette findet eine Rundung statt, die über das wahre Ergebnis hinwegtäuscht.

Aus dieser Tatsache leiten sich folgende Grundsätze ab:

- Vermeiden Sie es nach Möglichkeit, innerhalb von Schleifen gebrochene Double- oder Single-Werte zu verwenden. Sie laufen sonst Gefahr, dass sich Ihr Programm auf Grund der beschriebenen Ungenauigkeiten in Endlosschleifen verrennt.
- Verwenden Sie Single- und Double-Datentypen nur dort, wo es nicht auf die x-te Stelle hinter dem Komma ankommt. Bei der Berechnung von Grafiken beispielsweise, wo Rundungsfehler durch eine geringere Bildschirmauflösung als die Rechenpräzision ohnehin keine Rolle spielen, sollten Sie immer die schnelleren, prozessorberechneten Datentypen Single und Double dem manuell berechneten Decimal-Datentyp vorziehen.
- Bei finanztechnischen Anwendungen sollten Sie *in jedem Fall* den Decimal-Datentyp einsetzen. Nur mit ihm ist gewährleistet, dass Additionen und andere Berechnungen von nicht exakt darstellbaren Zahlen nicht in größere Fehler münden.
- Verwenden Sie andererseits den Decimal-Datentyp, wenn es eben geht, nie in Schleifen und setzen Sie ihn schon gar nicht als Zählvariable ein. Er erfährt nämlich keine Unterstützung durch den Prozessor und bremst ihr Programm extrem aus! Verwenden Sie dort nach Möglichkeit nur eine der zahlreichen Integer-VariablenTypen.
- Wenn Sie – aus Geschwindigkeitsgründen – dennoch Double- oder Single-Typen auf Gleichheit testen müssen, arbeiten Sie besser mit einer Abfrage des Deltas, etwa:

```
If Math.Abs(locDouble1 - locDouble2) < 0.0001 then
    'Werte sind annähernd dieselben, also quasi gleich.
End If
```

Besondere Funktionen, die für alle numerischen Datentypen gelten

Alle numerischen Datentypen verfügen über Methoden, die bei allen Typen nach der gleichen Vorgehensweise verwendet werden. Sie dienen zur Umwandlung einer Zeichenkette (einer Ziffernfolge) in den entsprechenden Wert sowie zur Umwandlung des Wertes in eine Zeichenkette. Mit anderen Funktionen können Sie den größten oder kleinsten Wert ermitteln, den ein Datentyp darstellen kann.

Zeichenketten in Werte wandeln und Vermeiden von kulturabhängigen Fehlern

Zum Umwandeln einer Zeichenkette in einen Wert dienen die statischen Funktionen Parse oder TryParse, die jedem numerischen Datentyp zur Verfügung stehen. Um beispielsweise die Ziffernfolge »123« in den Integerwert 123 umzuwandeln, genügen die folgenden Anweisungen:

```
Dim locInteger As Integer
locInteger = Integer.Parse("123")
```

WICHTIG Allerdings können Sie seit Visual Basic 2005 nicht mehr

```
locInteger = locInteger.Parse("123") ' Sollte so nicht mehr gemacht werden!
```

schreiben, denn Parse ist eine statische Funktion und sollte nicht mehr über eine Objektvariable sondern nur über den entsprechenden Klassennamen angesprochen werden. Zwar ließe sich das Programm beim Ansprechen der statischen Funktion über eine Objektvariable noch kompilieren, bereits im Codeeditor würden Sie aber eine Warnung in der Fehlerliste sehen.

Es gibt auch die Möglichkeit des Umwandlungsversuchs einer Zeichenkette in einen numerischen Wert, wie es das folgende Beispiel zeigt:

```
Dim locInteger As Integer
If Integer.TryParse("123", locInteger) Then
    'Umwandlung war erfolgreich
Else
    'Umwandlung war nicht erfolgreich
End If
```

Bei erfolgreicher Umwandlung steht die umgewandelte Zahl anschließend in der TryParse übergebenden Variablen – im Beispiel also in locInteger.

Auch das Framework-Äquivalent von Integer ermöglicht die Umwandlung durch

```
locInteger = System.Int32.Parse("123") ' Und auch das ginge.
```

und um die Liste komplett zu machen, geht es natürlich auch über die Convert-Klasse im Framework-Stil mit

```
locInteger = Convert.ToInt32("123") ' Und das ginge.
```

und in alter Visual Basic-Manier tätigt es die Anweisung

```
locInteger = CInt("123") ' Letzte Möglichkeit, mehr fallen mir nicht ein.
```

ebenfalls.

Aber aufgepasst: Wenn Sie das folgende Programm auf einem deutschen System starten, passiert möglicherweise nicht das, was Sie erwarten:

```
Dim locString As String = "123.23"
Dim locdouble As Double = Double.Parse(locString)
Console.WriteLine(locdouble.ToString())
```

Sie rechnen vielleicht damit, dass die Ziffernfolge korrekt in den Wert 123,23 umgewandelt wird. Anstelle dessen gibt das Programm

```
12323
```

aus – definitiv nicht das Ergebnis, das Sie erwartet haben. Starten Sie das Programm auf einem englischen System, ist das Ergebnis korrekt und wie erwartet?⁷

123.23

Na ja, vielleicht nicht ganz. Wir Deutschen haben uns angewöhnt, die Nachkommastellen von den Vorkommastellen mit einem Komma zu trennen (vermutlich heißen sie auch deshalb *Nachkommastellen*). Englischsprachige Länder machen das allerdings mit einem Punkt, und die Ausgabe, die Sie über diesem Absatz sehen, ist eine korrekte englische Formatierung. Welche Auswirkungen hat dieses Verhalten auf Ihre Programme? Nun, zunächst einmal sollten Sie es unbedingt vermeiden, im Programmcode selbst numerische Konstanten als String zu speichern, wenn Sie sie später in einen numerischen Typ wandeln wollen (wie im letzten Beispiel gezeigt). Wenn Sie numerische Datentypen innerhalb Ihres Programms definieren, dann machen Sie es bitte grundsätzlich nur im Code direkt, nicht mit Zeichenketten (in Anführungszeichen) und deren Umwandlungsfunktionen. Sie haben sicherlich schon festgestellt, dass im Code abgelegte Ziffernfolgen (ohne Anführungszeichen) zur Zuweisung eines Wertes grundsätzlich nur im englischen Format abgelegt werden.

Solange Sie keine Dateien mit als Text gespeicherten Informationen, aus denen Ihr Programm Werte generieren muss, über Kulturgrenzen hinweg austauschen müssen, haben Sie nichts zu befürchten: Läuft Ihre Anwendung auf einem englischen System, werden Zahlen mit Punkt als Trennzeichen in die Datei geschrieben, hier im deutschsprachigen Raum eben als Komma. Da die Kultureinstellungen beim Einlesen äquivalent berücksichtigt werden, kann Ihre Anwendung auch die richtigen Werte aus der Textdatei zurückgenerieren.

Problematisch wird es dann, wenn auch die Dateien mit den Texten über Kulturgrenzen hinweg ausgetauscht werden sollen. Dies geschieht z.B. schnell, wenn Sie mit einer .NET Windows Forms Anwendung von Ihrem deutschen Windows XP auf einen Datenbankserver im Unternehmen zugreifen. Viele IT-Abteilungen betreiben Ihre Server aus den verschiedensten Gründen ausschließlich in den englischen Versionen. Dann würde eine USA-Plattform die Datei mit Punkt als Trennzeichen exportieren und hier in Deutschland würde die Parse-Funktion den Punkt als Tausenderpunkt ansehen, damit unter den Tisch fallen lassen und so fälschlicherweise den Wert verhundertfachen. In diesem Fall müssen Sie dafür sorgen, dass der Export in eine Textdatei kulturreutral erfolgt, und das können Sie auf folgende Weise erreichen:

Sowohl die Parse-Funktion als auch die ToString-Funktion aller numerischen Typen können bei der Umwandlung durch einen so genannten *Format Provider* (etwa: Formatanbieter) spezifisch gesteuert werden. Es gibt die unterschiedlichsten Format Provider in .NET für numerische Typen, nämlich auf der einen Seite solche, mit denen Sie Formate in Abhängigkeit von der Anwendung (finanztechnisch, wissenschaftlich, etc.) oder in Abhängigkeit von der Kultur steuern können. Dazu dienen die Klassen `NumberFormatInfo` und `CultureInfo`. Beiden lassen sich, zuvor entsprechend instanziert und aufbereitet, als Parameter sowohl der ToString- als auch der Parse-Funktion übergeben. Genaueres über den Umgang mit diesen Klassen erfahren Sie ab Kapitel 26; für den Moment soll das folgende Beispiel genügen, das demonstriert, wie Sie erzwingen, dass Umwandlungen nicht von der aktuellen Kultur abhängig gemacht werden, sondern kulturreutral erfolgen.

⁷ Wobei dieses Verhalten streng genommen nicht dadurch bedingt wird, dass es sich um ein englisches System handelt, sondern dass ein englisches Betriebssystem voreingestellt andere Ländereinstellungen aufweist als ein deutsches. Natürlich könnten Sie auch ein deutsches Betriebssystem so konfigurieren, dass es das gleiche Ergebnis liefert.

WICHTIG Diese Vorgehensweise, wie sie im Folgenden zu sehen ist, sollten Sie immer bei Anwendungen anwenden, die mit internationalem Anspruch entwickelt werden, um Fehler bei Typkonvertierungen von vorneherein auszuschließen!

```
Dim locString As String = "123.23"
Dim locdouble As Double

locdouble = Double.Parse(locString, CultureInfo.InvariantCulture)
Console.WriteLine(locdouble.ToString(CultureInfo.InvariantCulture))
Console.ReadLine()
```

HINWEIS Damit Sie auf kulturbezogene Klassen und Funktionen zugreifen können, müssen Sie am Anfang des Programms den entsprechenden Namensbereich System.Globalization mit Imports eingebunden haben, etwa so:

```
Imports System.Globalization
```

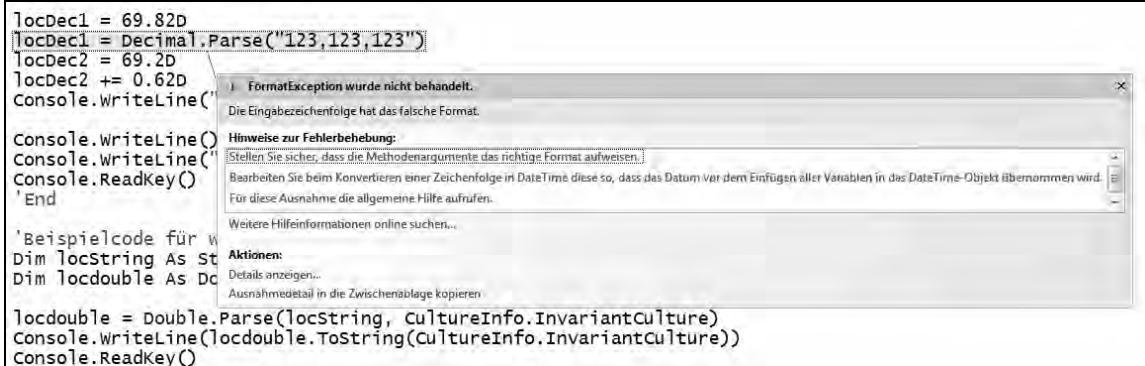


Abbildung 7.1 Falls die Zeichenkette zur Umwandlung auf Grund ihres Formates nicht umgewandelt werden kann, generiert das Framework eine Ausnahme

Die statische Eigenschaft InvariantCulture liefert direkt eine Instanz einer CultureInfo-Klasse zurück, die mit den entsprechenden Eigenschaften bestückt wurde.

HINWEIS Wenn der Umwandlungsversuch fehlschlägt, da die Zeichenkette schlicht und ergreifend kein konvertierbares Format enthält und sich deswegen nicht in einen Wert umwandeln lässt, generiert das Framework eine Ausnahme (siehe Abbildung 7.1). Sie können die Ausnahme entweder mit Try/Catch abfangen oder alternativ die statische Funktion TryParse (siehe unten) verwenden, die grundsätzlich keine Ausnahme beim Konvertierungsversuch erzeugt.

Ermitteln von minimal und maximal darstellbarem Wert eines numerischen Typs

Die numerischen Datentypen kennen zwei spezielle statische Eigenschaften, mit denen Sie jeweils den größten und kleinsten darstellbaren Wert ermitteln lassen können. Die Eigenschaften lauten MinValue und MaxValue, und Sie können sie wie jede statische Funktion entweder durch den Typnamen selbst oder durch eine Objektvariable des Typs aufrufen. Beispiel:

```
Dim locInteger As Integer  
Dim locdouble As Double  
Dim locDecimal As System.Decimal  
  
Console.WriteLine(locInteger.MaxValue)  
Console.WriteLine(Double.MinValue)  
Console.WriteLine(locDecimal.MaxValue)
```

Spezielle Funktionen der Fließkommatypen

Die Fließkommatypen verfügen über einige besondere Eigenschaften zur Darstellung von speziellen Werten, auf deren Zustand Sie eine Objektvariable mit entsprechenden Funktionen überprüfen können.

Unendlich (Infinity)

Wenn Sie einen Fließkommawertetyp durch 0 teilen, dann erzeugen Sie damit keine Ausnahme (keinen Fehler). Vielmehr ist das Ergebnis *unendlich* (*infinity*), und sowohl `Single` als auch `Double` können dieses Ergebnis darstellen, wie das folgende Beispiel demonstriert:

```
Dim locdouble As Double  
locdouble = 20  
locdouble /= 0  
Console.WriteLine(locdouble)  
Console.WriteLine("Die Aussage locDouble ist +unendlich ist {0}.", locdouble = Double.PositiveInfinity)
```

Wenn Sie dieses Beispiel ausführen, erzeugt es keine Fehlermeldung in Form einer Ausnahme, sondern das Programm schreibt vielmehr ein Ergebnis auf den Bildschirm:

```
+unendlich  
Die Aussage locDouble ist +unendlich ist True.
```

Anstatt den Vergleich auf Unendlichkeit durch den Vergleichsoperator durchzuführen, können Sie auch die statische Funktion `IsInfinity` verwenden:

```
Console.WriteLine("Die Aussage locDouble ist +unendlich ist {0}.", locdouble.IsInfinity(locdouble))
```

Das hat den Vorteil, dass Sie im Vorfeld nicht wissen müssen, ob ein Ergebnis positiv oder negativ unendlich ist. Mit den Funktionen `IsPositiveInfinity` und `IsNegativeInfinity` können Sie dennoch die Differenzierung innerhalb einer Abfrage vornehmen.

Um einer Variable gezielt den Wert unendlich zuzuweisen, verwenden Sie die statischen Funktionen `PositiveInfinity` und `NegativeInfinity`, die entsprechende Konstanten zurückliefern.

Keine Zahl (NaN, Not a Number)

Und noch einen Sonderfall decken die primitiven Fließkommatypen ab: Die Division von 0 und 0, die mathematisch nicht definiert ist und *keine gültige Zahl* ergibt:

```
'Sonderfall: 0/0 ist mathematisch nicht definiert und ergibt "Not a Number"  
einDouble = 0  
einDouble = einDouble / 0  
If Double.IsNaN(einDouble) Then  
    Debug.Print("einDouble ist keine Zahl!")  
End If
```

Ließen Sie diesen Code laufen, würde der Text in der If-Abfrage ausgegeben werden.

WICHTIG Überprüfungen auf diese Sonderwerte lassen sich nur durch die Eigenschaften testen, die statische Funktionen direkt an den Typen »hängen«. Zwar können Sie beispielsweise mit der Konstante der Fließkommatypen `NaN` den Wert »keine gültige Zahl« einer Variablen zuweisen; diese Konstante eignet sich allerdings nicht, auf diesen Zustand (»Wert«) zu testen, wie das folgende Beispiel zeigt:

```
Dim einDouble As Double  
  
'Sonderfall: 0/0 ist mathematisch nicht definiert und ergibt "Not a Number"  
einDouble = 0  
einDouble = einDouble / 0  
  
'Der Text sollte erwartungsgemäß ausgegeben werden,  
'wird er aber nicht!  
If einDouble = Double.NaN Then  
    Debug.Print("Test 1:einDouble ist keine Zahl!")  
End If  
  
'Nur so kann der Test erfolgen!  
If Double.IsNaN(einDouble) Then  
    Debug.Print("Test 2:einDouble ist keine Zahl!")  
End If
```

In diesem Beispiel würde nur der zweite Text ausgegeben.

Versuchte Umwandlungen mit TryParse

Alle numerischen Datentypen kennen die statische Funktion `TryParse`, die versucht, eine Zeichenkette in einen Wert umzuwandeln. Im Gegensatz zu `Parse` erzeugt sie allerdings keine Ausnahme, wenn die Umwandlung nicht gelingt. Vielmehr übergeben Sie ihr eine Variable als Referenz zur Speicherung des Rückgabewertes, und das Funktionsergebnis informiert Sie darüber, ob die Konvertierung gelang (`True`) oder nicht (`False`):

```
Dim locdouble As Double  
Dim locString As String = "Einhundertdreiundzwanzig"  
  
'locdouble = Double.Parse(locString) ' Ausnahme  
'Keine Ausnahme:  
Console.WriteLine("Konvertierung erfolgreich? {0}", _  
    Double.TryParse(locString, NumberStyles.Any, New CultureInfo("de-DE"), locdouble))
```

Spezielle Funktionen des Wertetyps Decimal

Der Wertetyp `Decimal` verfügt ebenfalls über spezielle Funktionen, von denen viele allerdings in Visual Basic nicht zur Anwendung kommen (nichtsdestotrotz können Sie sie verwenden, aber es ergibt wenig Sinn – tatsächlich wurden sie für andere Sprachen, die eine Operatorenüberladung [noch] nicht kennen, aufgenommen). Nehmen wir als Beispiel die statische `Add`-Funktion, die zwei Zahlen vom Typ `Decimal` addiert. Sie liefert als Ergebnis ein `Decimal` zurück. Stattdessen können Sie aber auch auf den `+`-Operator von Visual Basic zurückgreifen, der ebenfalls zwei Zahlen vom Typ `Decimal` addieren kann – und das natürlich für die spätere Nacharbeitung in einem viel leichter lesbaren Code. Lediglich die Funktionen der folgenden Tabelle sind sinnvoll im Gebrauch:

Funktionsname	Aufgabe
<code>Remainder(Dec1, Dec2)</code>	Ermittelt den Rest der Division der beiden <code>Decimal</code> -Werte.
<code>Round(Dec, Integer)</code>	Rundet einen <code>Decimal</code> -Wert auf die angegebene Anzahl der Nachkommastellen.
<code>Truncate(Dec)</code>	Gibt den Vorkommateil des angegebenen <code>Decimal</code> -Wertes zurück.
<code>Floor(Dec)</code>	Rundet den <code>Decimal</code> -Wert auf die nächste kleinere ganze Zahl.
<code>Negate(Decimal)</code>	Multipliziert den <code>Decimal</code> -Wert mit <code>-1</code> .

Tabelle 7.3 Die wichtigsten Funktionen des `Decimal`-Typs

Der Datentyp Char

Der `Char`-Datentyp speichert ein Zeichen im Unicode-Format (mehr zu diesem Thema finden Sie im Abschnitt »Speicherbedarf von Strings« auf Seite 235) und belegt damit 16 Bit, bzw. 2 Byte. Anders als `String` ist der `Char`-Datentyp ein Wertetyp. Die folgende Kurzübersicht klärt nähere Details:

.NET-Datentyp: `System.Char`

Stellt dar: ein einzelnes Zeichen

Wertebereich: Die Wertebereich beträgt 0–65535, damit Unicode-Zeichen dargestellt werden können.

Typliteral: `c`

Speicherbedarf: 2 Byte

Delegation an den Prozessor: ja

CLS-Compliant: ja

Anmerkungen: `Chars` werden häufig in Arrays verwendet, da ihre Verarbeitung in vielen Fällen praktischer ist als die Verarbeitung von `Strings`. Sie können `Char`-Arrays wie jeden anderen Datentyp mit Konstanten definieren; ein Beispiel dafür finden Sie unter dem nächsten Punkt.

Weitere Beispiele, wie Sie `Char`-Arrays anstelle von `Strings` beispielsweise zum Verarbeiten Buchstabe für Buchstabe verwenden, finden Sie im Abschnitt über `Strings`.

Auch wenn Char intern als vorzeichenloser 16-Bit-Wert und damit wie ein Short gespeichert wird, können Sie keine impliziten Konvertierungen in einen numerischen Typ vornehmen. Sie können aber, in Ergänzung zur in der Online-Hilfe beschriebenen Möglichkeit nicht nur die Funktionen AscW und ChrW zur Umwandlung von Char in einen numerischen Datentyp und umgekehrt, sondern auch die Convert-Klasse verwenden. Beispiel:

```
'Normale Deklaration und Definition
Dim locChar As Char
locChar = "1"c
Dim locInteger As Integer = Convert.ToInt32(locChar)
Console.WriteLine("Der Wert von '{0}' lautet {1}", locChar, locInteger)
```

Wenn Sie dieses Beispiel laufen lassen, sehen Sie folgende Ausgabe auf dem Bildschirm:

```
Der Wert von '1' lautet 49
```

Die Funktionen Chr und Asc können Sie ebenfalls verwenden; sie funktionieren, aber nur für Nicht-Unicode-Zeichen (ASCII 0-255). Außerdem haben sie durch etliche interne Bereichsüberprüfungen einen enormen Overhead und sind deswegen lange nicht so schnell wie AscW, ChrW (die am schnellsten sind, weil eine direkte interne Typumwandlung von Char in Integer und umgekehrt stattfindet) oder die Convert-Klasse (die den Vorteil hat, auch von Nicht-Visual Basic-Entwicklern verstanden zu werden).

Deklaration und Beispielzuweisung (auch als Array):

```
'Normale Deklaration und Definition
Dim locChar As Char
locChar = "K"c

'Ein Char-Array mit Konstanten deklarieren und definieren.
Dim locCharArray() As Char = {"A"c, "B"c, "C"c}

'Ein Char-Array in einen String umwandeln.
Dim locStringAusCharArray As String = New String(locCharArray)

'Einen String in ein Char-Array umwandeln.
'Das geht natürlich auch mit einer Stringvariablen.
locCharArray = "Dies ist ein String".ToCharArray
```

Der Datentyp String

Mit Strings speichern Sie Zeichenketten. Anders als bei Visual Basic 6 gibt es für den Umgang mit Strings einen objektorientierten Ansatz, mit dem die Programmierung der Verarbeitung von Strings viel einfacher wird. Man wird Ihre Programme viel leichter lesen können, wenn Sie dieses OOP-Konzept verwenden.

Im Laufe der vergangenen Kapitel sind Ihnen Strings schon an vielen Stellen begegnet. Dennoch lohnt es sich, einen weiteren Blick hinter die Kulissen zu werfen, und nicht zuletzt durch die Klasse Regex (Abkürzung von »Regular Expressions« – etwa: »reguläre Ausdrücke«) gibt Ihnen das Framework eine Unterstützung in Sachen Zeichenketten an die Hand, wie sie besser eigentlich nicht mehr sein kann.

Anders als andere primitive Typen handelt es sich bei Strings um Referenztypen. Dennoch ist es nicht notwendig, eine neue String-Instanz mit dem Schlüsselwort `New` zu definieren.⁸

Erreicht wird das durch das Eingreifen des Compilers, der ohnehin anderen Code generieren muss als bei anderen Objekten.

Die folgenden Abschnitte geben Ihnen eine Übersicht über die besonderen Eigenarten der Strings der Base Class Library. Am Ende dieses Abschnittes finden Sie im Referenzstil die Anwendung der wichtigsten String-Funktionen beschrieben.

Strings – gestern und heute

Durch die neue Implementierung des Datentyps `String`, der ebenso wie andere Objekte durch Instanziierung seiner Klasse entsteht, gibt es seit Visual Studio 2002 und dem Framework 1.0 eine völlig neue Herangehensweise bei der Arbeit mit Zeichenketten.

Fast alle Befehle und Funktionen, die es in Visual Basic noch »allein stehend« gab, gibt es auch noch in den Framework-Versionen von Visual Basic. Allerdings sind sie nicht nur überflüssig, da es sich mit den vorhandenen Methoden und Eigenschaften des `String`-Objektes viel eleganter zum Ziel kommen lässt, sondern sie bremsen Programme auch unnötig aus, da sie letzten Endes selbst die `String`-Objektfunktionen aufrufen.

Für (fast) jede der alten String-Funktionen gibt es eine entsprechende Klassenfunktion, die Sie statt ihrer verwenden sollten. Die folgenden Abschnitte demonstrieren Ihnen den Umgang mit Strings anhand kurzer Beispiele.

Strings deklarieren und definieren

Strings werden, wie alle primitiven Datentypen, ohne das Schlüsselwort `New` deklariert; Zuweisungen können direkt im Programm erfolgen. Eine Zeichenkette kann also beispielsweise mit der Anweisung

```
Dim locString As String
```

deklariert und sofort verwendet werden. Die Instanzbildung des `String`-Objektes geschieht auf IML-Ebene.

Strings werden definiert, indem Sie ihnen eine Zeichenkette zuweisen, die Sie in Anführungszeichen setzen, wie im folgenden Beispiel:

```
locString = "Miriam Sonntag"
```

Genau wie andere primitive Datentypen können Deklaration und Zuweisung in einer Anweisung erfolgen. So könnten Sie natürlich die beiden oben stehenden einzelnen Anweisungen durch die folgende ersetzen:

```
Dim locString As String = "Miriam Sonntag"
```

⁸ Ähnlich wie bei Nullables beim Boxing (siehe Kapitel 11) greift die CLR hier in das Standardverhalten für Referenztypen ein und ändert es – da Strings Referenztypen sind, müssten sie natürlich strenggenommen mit `New` instanziert werden. Der Gleichheitsoperator müsste allerdings ebenfalls lediglich eine Referenz zuweisen – bei Strings wird beim Zuweisen einer Instanz an eine Objektvariable allerdings ein Cloning des Inhalts vorgenommen, ebenfalls abweichend vom Standard. Mehr zum Thema Referenz und Verweistypen, die diesen Zusammenhang genauer erklären, finden Sie in Kapitel 16.

Der String-Konstruktor als Ersatz von String\$

Normalerweise dient ein Konstruktor einer Klasse zum Erstellen einer Instanz und einer Struktur, um bestimmte Parameter mit Werten vorzubelegen – der OOP-Teil dieses Buchs verrät Ihnen mehr zu diesem Thema.

Und obwohl Sie Strings wie primitive Datentypen ohne Konstruktor erstellen, haben Sie dennoch die Möglichkeit, einen Konstruktor zu verwenden. Jedoch verwenden Sie den Konstruktor nicht ausschließlich zur Neuinstanziierung eines leeren String-Objektes (der parameterlose Konstruktor ist dafür auch gar nicht erlaubt), sondern emulieren (unter anderem) eigentlich die alte String\$-Funktion aus Visual Basic 6.0.

Mit ihrer Hilfe war es möglich, eine Zeichenfolge programmgesteuert zu wiederholen und in einem String abzuspeichern. Übrigens: Während viele der alten Visual Basic-6.0-Befehle auch noch in den Framework Versionen 1.0, 1.1 und 2.0 vorhanden sind, gibt es die String-Funktion selbst – wohl wegen der Verwendung ihres Schlüsselwortes als Typezeichner – in Visual Basic .NET nicht mehr.

Um den String-Konstruktor als String\$-Funktionsersatz zu verwenden, verfahren Sie folgendermaßen:

```
Dim locString As String = New String("A"c, 40)
```

Sie sehen am Typliteral »c«, dass Sie im Konstruktor einen Wert vom Typ Char übergeben müssen. Damit beschränkt sich die Wiederholungsfunktion dummerweise auf ein Zeichen, was bei String\$ nicht der Fall war. Eine eigene Repeat-Funktion zu implementieren, die diese Aufgabe löst, ist aber kein wirkliches Problem:

```
Public Function Repeat(ByVal s As String, ByVal repeatitions As Integer) As String  
  
    Dim locString As String  
  
    For count As Integer = 1 To repeatitions  
        locString &= s  
    Next  
  
    Return locString  
End Function
```

HINWEIS Dieses Konstrukt dient in erster Linie als Beispiel, und Sie sollten Zeichenketten auf diese Weise nur »zusammenbauen«, wenn es sich um weniger Zeichen handelt. Für größere Mengen verwenden Sie aus Performance-Gründen besser die StringBuilder-Klasse, die Sie im Abschnitt »StringBuilder vs. String – wenn es auf Geschwindigkeit ankommt« ab Seite 247 beschrieben finden. Warum das so ist, klärt der Abschnitt »Strings sind unveränderlich« ab Seite 235.

Neben der Fähigkeit des Konstruktors, Strings aus einem sich wiederholenden Zeichen zu generieren, können Sie ihn ebenfalls dazu verwenden, einen String aus einem Char-Array oder einem Teil eines Char-Arrays zu erstellen, wie das folgende Beispiel zeigt:

```
Dim locCharArray() As Char = {"K"c, ".c, " "c, "L"c, "ö"c, "f"c, "f"c, "e"c, "l"c, "m"c, "a"c, "n"c,  
"n"c}  
Dim locString As String = New String(locCharArray)  
Console.WriteLine(locString)  
locString = New String(locCharArray, 3, 6)  
Console.WriteLine(locString)
```

Wenn Sie dieses Programm laufen lassen, erscheint im Konsolenfenster folgende Ausgabe:

K. Löffelmann
Löffel

Einem String Zeichenketten mit Sonderzeichen zuweisen

Wenn Sie Anführungszeichen im String selbst verwenden wollen, dann bedienen Sie sich doppelter Anführungszeichen. Um die Zeichenkette

Miri sagte, "es ist erst 13.00 Uhr, ich schlafe noch ein wenig".

im Programm zu definieren, würde die Zuweisungsanweisung lauten:

```
locString = "Miriam sagte, ""ich schlafe noch ein wenig!"".
```

Möchten Sie andere Sonderzeichen in Strings einbauen, bedienen Sie sich im Basic-Sprachschatz vorhandener Konstanten. Um beispielsweise einen Absatz in einen String einzubauen, müssen Sie die ASCIIIs für *Linefeed* (Zeilenvorschub) und *Carriage Return* (Wagenrücklauf) in den String einfügen. Sie erreichen das durch die Verwendung der Konstanten, wie im folgenden Beispiel zu sehen:

```
locAndererString = "Miriam sagte ""ich schlafe noch ein wenig!"" + vbCr + vbLf +_
    "Sie schlief direkt wieder ein."
```

Weniger Schreibarbeit haben Sie mit der folgenden Version, die exakt das gleiche Resultat liefert:

```
locAndererString = "Miriam sagte ""ich schlafe noch ein wenig!"" + vbNewLine +_
    "Sie schlief direkt wieder ein."
```

Ihnen stehen zum Einfügen von Sonderzeichen die folgenden Konstanten in Visual Basic zur Verfügung:

Konstante	ASCII	Beschreibung
vbCrLf oder vbNewLine	13; 10	Wagenrücklaufzeichen/Zeilenvorschubzeichen
vbCr	13	Wagenrücklaufzeichen
vbLf	10	Zeilenvorschubzeichen
vbNullChar	0	Zeichen mit dem Wert 0
vbNullString	Zeichenfolge ""	Zeichenfolge mit dem Wert 0. Entspricht nicht einer Zeichenfolge mit 0-Länge (""); diese Konstante ist für den Aufruf externer Prozeduren gedacht (COM-Interop).
vbTab	9	Tabulatorzeichen
vbBack	8	Rückschrittzeichen
vbFormFeed	12	Wird in Microsoft Windows nicht verwendet.
vbVerticalTab	11	Steuerzeichen für den vertikalen Tabulator, der in Microsoft Windows aber nicht verwendet wird.

Tabelle 7.4 Die einfachsten Möglichkeiten, Sonderzeichen in Strings einzubauen

Speicherbedarf von Strings

Jedes Zeichen, das in einem String gespeichert wird, belegt zwei Bytes an Arbeitsspeicher. Auch wenn Strings in Buchstabenform ausgegeben werden, so hat im Speicher selbst natürlich jedes Zeichen einen bestimmten Wert. Die Codewerte von Strings entsprechen bis 255 dem *American Standard Code for Information Interchange* – kurz ASCII –, wobei nur Werte bis 127 bei jedem verwendeten Ausgabezeichensatz einheitlich sind. Sonderzeichen spezieller Länder sind, abhängig vom verwendeten Zeichensatz, in den Bereichen 128–255 definiert, wobei auch hier in der Regel die Codes für die in europäischen Ländern verwendeten Sonderzeichen wie »öäüÖÄÜéé« in jedem Font dieselben Codes haben (Ausnahmen bestätigen wie immer die Regel). Werte über 255 stellen Sonderzeichen dar, die beispielsweise für kyrillische, arabische oder asiatische Zeichen verwendet werden. Die Codierungskonvention, nach der ein Zeichen Werte über 255 annehmen darf, und die die Codierung einer größeren Anzahl von Zeichen erlaubt, nennt man übrigens *Unicode*. .NET Framework-Strings speichern Zeichenketten generell im *Unicode*-Format.

Strings sind unveränderlich

Strings sind generell Referenztypen,⁹ aber dafür grundsätzlich konstant, also unveränderlich. Das bedeutet für Sie in der Praxis keine Einschränkung mit dem gewohnten Umgang von Zeichenketten, denn: Wenn es eigentlich so aussieht, als hätten Sie einen String verändert, dann haben Sie in Wahrheit einen neuen erzeugt, der die Veränderungen widerspiegelt. Wissen müssen Sie das nur bei Anwendungen, die sehr viele Stringoperationen beanspruchen. In diesem Fall sollten Sie die so genannte *StringBuilder*-Klasse verwenden, da diese zwar nicht die Flexibilität von Strings und auch nicht den Vorteil von primitiven Datentypen mit sich bringt, aber für diese Fälle deutlich leistungsfähiger ist (der Abschnitt »Stringbuilder vs. String – wenn es auf Geschwindigkeit ankommt« ab Seite 247 verrät mehr darüber).

Viel wichtiger ist die Auswirkung dieser Unveränderlichkeit von Strings beim Einsatz in Ihren Programmen: Obwohl Strings als Referenztypen gelten, haben sie letzten Endes das Verhalten von Wertetypen, eben dadurch, dass sie unveränderlich sind. Wenn zwei Stringvariablen auf denselben Speicherbereich verweisen und Sie den Inhalt eines Strings verändern, dann sieht es nämlich nur so aus, als würden Sie ihn verändern. In Wirklichkeit legen Sie ja ein komplett neues String-Objekt im Speicher ab und lassen die vorhandene Variable darauf verweisen. Damit kommen Sie nie in die Situation, die Sie von Referenztypen kennen: Das Verändern eines Objektinhaltes durch die eine Objektvariable führt bei Strings nie dazu, dass eine andere Objektvariable, die auf den gleichen Speicherbereich zeigte, den Stringinhalt verändert wiedergibt, denn Strings werden ja nicht verändert. Diese Tatsache erklärt, dass Strings zwar Referenztypen sind, sich aber wie Wertetypen »anfühlen«.

Mehr Informationen dazu erhalten Sie auch im nächsten Abschnitt und den programmtechnischen Beweis dafür im Abschnitt »Trimmen von Strings« auf Seite 241.

⁹ Mehr zu Referenz und zu Wertetypen erfahren Sie im Kapitel 16.

Speicheroptimierung von Strings durch das Framework

Für die Speicherung von Strings gibt es einen so genannten *internen Pool*, der dazu verwendet wird, Redundanzen bei der Zeichenkettenspeicherung zu vermeiden. Wenn Sie innerhalb Ihres Programms zwei Strings mit der gleichen Konstante definieren, erkennt der Visual Basic-Compiler das und legt den String im Speicher nur ein einziges Mal ab, lässt beide Objektvariablen aber auf denselben Speicherbereich zeigen, wie das folgende Beispiel beweist:

```
Dim locString As String
Dim locAndererString As String

locString = "Miriam" & " Sonntag"
locAndererString = "Miriam Sonntag"
Console.WriteLine(locString Is locAndererString)
```

Wenn Sie dieses Programm starten, gibt es True aus – beide Strings verweisen also auf den gleichen Speicherbereich.

Diese Tatsache gilt aber nur so lange, wie der Compiler die Gleichheit der Strings erkennen kann, und dafür müssen die Konstanten in der gleichen Zeile zusammengesetzt werden. Schon bei der Veränderung in die folgende Version kann der Compiler die Gleichheit der Strings nicht mehr erkennen, und das Ergebnis wird False:

```
Dim locString As String
Dim locAndererString As String

locString = "Miriam"
locString &= " Sonntag"
locAndererString = "Miriam Sonntag"
Console.WriteLine(locString Is locAndererString)
```

Es liegt auf der Hand, dass dieses Verhalten zur Laufzeit zu viel Zeit kosten würde, um es sinnvoll anzuwenden. Bei sehr hohem String-Aufkommen würde die BCL zu viel Zeit nach der Suche bereits vorhandener Strings verschwenden. Allerdings haben Sie die Möglichkeit, einen String, den Sie zur Laufzeit erstellen, gezielt dem Pool hinzuzufügen. Gleichen sich mehrere Strings, die Sie dem internen Pool hinzufügen, werden die Speicherbereiche wieder nicht-redundant zugewiesen – mehrere, gleich lautende Strings teilen sich dann den Speicher. Sinnvoll ist das natürlich nur dann, wenn vorauszusehen ist, dass es viele gleich lautende Strings innerhalb eines Programms geben wird. Ein Beispiel zeigt, wie das Hinzufügen eines Strings mit der statischen Funktion Intern zum internen Pool explizit vorgenommen wird:

```
Dim locString As String = New String(locCharArray)
Dim locAndererString As String

locString = "Miriam"
locString &= " Sonntag"
locString = String.Intern(locString)
locAndererString = String.Intern("Miriam Sonntag")
Console.WriteLine(locString Is locAndererString)
```

Wenn Sie dieses Programm starten, lautet die Ausgabe wieder True.

Ermitteln der String-Länge

Visual Basic 6 kompatibler Befehl: Len

Visual Basic .NET: strVar.Length

Anmerkung: Mit diesem Befehl ermitteln Sie die Länge eines Strings in Zeichen (nicht in Bytes!).

Beispiel: Das folgende Beispiel liest eine Zeichenkette von der Tastatur ein und gibt die Zeichen in umgekehrter Reihenfolge aus:

```
Dim locString As String
Console.WriteLine("Geben Sie einen Text ein: ")
locString = Console.ReadLine()
For count As Integer = locString.Length - 1 To 0 Step -1
    Console.Write(locString.Substring(count, 1))
Next
```

Das gleiche Beispiel mit den VB6-Kompatibilitätsbefehlen finden Sie im nächsten Abschnitt.

Ermitteln von Teilen eines Strings oder eines einzelnen Zeichens

Visual Basic 6 kompatible(r) Befehl(e): Left, Right, Mid

Visual Basic .NET: strVar.SubString

Anmerkung: Mit diesem Befehl können Sie einen bestimmten Teil eines Strings als String zurückgeben lassen.¹⁰

Beispiel: Das folgende Beispiel liest eine Zeichenkette von der Tastatur ein und gibt die Zeichen in umgekehrter Reihenfolge aus. Das gleiche Beispiel finden Sie im vorherigen Abschnitt mit den Funktionen des String-Objektes.

```
Dim locString As String
Console.WriteLine("Geben Sie einen Text ein: ")
locString = Console.ReadLine()
For count As Integer = Len(locString) To 1 Step -1
    Console.Write(Mid(locString, count, 1))
Next
```

HINWEIS Bitte achten Sie darauf, dass die VB6-Kompatibilitätsfunktionen die Zeichenzählung eines Strings durch Left, Right oder Mid bei 1 beginnen lassen, die Funktion SubString jedoch bei 0 beginnen lässt.

¹⁰ Warum das gute alte Left bzw. Right als jeweilige Methoden der Klasse String weggelassen wurden, weiß allein der Programmierer. Vielleicht weiß es der auch nicht, sondern hat es einfach vergessen oder kennt nur C und kann sich nicht vorstellen, dass die Welt so einfach wie in Basic sein kann.

Angleichen von String-Längen

Visual Basic 6 kompatible(r) Befehl(e): RSet, LSet

Visual Basic .NET: strVar.PadLeft; strVar.PadRight

Anmerkung: Mit diesen Befehlen können Sie die Länge eines Strings auf eine bestimmte Zeichenanzahl erweitern; der String wird dabei entweder vorne oder am Ende mit Leerzeichen aufgefüllt.

Beispiel: Das folgende Beispiel demonstriert den Umgang mit der PadLeft- und der PadRight-Methode.

```
Dim locString As String = "Dieser String ist so lang"
Dim locString2 As String = "Dieser nicht"
Dim locString3 As String = "Dieser"
Len(locString)
locString2 = locString2.PadLeft(locString.Length)
locString3 = locString3.PadRight(locString.Length)

Console.WriteLine(locString + ":")
Console.WriteLine(locString2 + ":")
Console.WriteLine(locString3 + ":")


```

Wenn Sie dieses Programm laufen lassen, generiert es die folgende Ausgabe:

```
Dieser String ist so lang:
      Dieser nicht:
Dieser          :
```

Suchen und Ersetzen

Visual Basic 6 kompatible(r) Befehl(e): Instr; InstrRev; Replace

Visual Basic .NET: strVar.IndexOf; strVar.IndexOfAny; strVar.Replace; strVar.Remove

Anmerkung: Mit dem VB6-kompatiblen Befehl Instr können Sie nach dem Vorkommen eines Zeichens oder einer Zeichenfolge in einem String suchen. InstrRev macht das Gleiche, beginnt die Suche aber von hinten. Replace erlaubt Ihnen, eine Zeichenfolge im String durch eine andere zu ersetzen.

Mit der String-Funktion IndexOf suchen Sie nach dem Vorkommen eines Zeichens oder einer Zeichenfolge im aktuellen String. IndexOfAny erlaubt Ihnen darüber hinaus, das Vorhandensein verschiedener Zeichen, die in einem Char-Array übergeben werden, im String zu finden. Replace ersetzt einzelne Zeichen oder Zeichenfolgen durch andere im aktuellen String und mit Remove haben Sie die Möglichkeit, eine bestimmte Zeichenfolge ganz aus dem String zu entfernen.

Beispiel: Das folgende Beispiel demonstriert den Umgang mit den Suchen- und Ersetzen-Methoden des String-Objektes:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 07\\Strings - Suchen und Ersetzen
```

Öffnen Sie dort die Projektmappe (.SLN-Datei).

```
Imports System.Globalization

Module Strings
    Sub Main()
        Dim locString As String =
            "Weisheiten:" + vbCrLf +
            "* Wenn man 8 Jahre, 7 Monate und 6 Tage schreien würde," + vbCrLf +
            " hätte man genug Energie produziert, um eine Tasse Kaffee heiß zu machen." + vbCrLf +
            "* Wenn man seinen Kopf gegen die Wand schlägt, verbraucht man 150 Kalorien." + vbCrLf +
            -
            "** Elefanten sind die einzigen Tiere, die nicht springen können." + vbCrLf +
            "** Eine Kakerlake kann 9 Tage ohne Kopf überleben, bevor sie verhungert." + vbCrLf +
            "** Gold und andere Metalle entstehen ausschließlich in" + vbCrLf +
            " Supernovae (Sternenexplosionen)." + vbCrLf +
            "** Der Mond besteht aus den Trümmern der Kollision eines Mars großen" + vbCrLf +
            " Planeten mit der Erde." + vbCrLf +
            "** New York wird ""Big Apple"" genannt, weil ""Big Apple"" in der Sprache" + vbCrLf +
            " der Jazz-Musiker ""das große Los ziehen"" bedeutete. In New York Karriere" + vbCrLf +
            -
            " zu machen war ihr großes Los." + vbCrLf +
            "* Der Ausdruck ""08/15"" für etwas Unorganisches war ursprünglich " + vbCrLf +
            " die Typenbezeichnung für das Maschinengewehr LMG 08/15;" + vbCrLf +
            " er wurde Metapher für geistlosen, militärischen Drill." + vbCrLf +
            "* ""Durch die Lappen gehen"" ist ein Begriff aus der Jagd:" + vbCrLf +
            " Hirsche ließen nicht durch eine aus Lappen bestehende," + vbCrLf +
            " flatternde Umzäunung - aus Angst. Außer manchmal."
        'Zahlenkombi durch Buchstaben ersetzen
        locString = locString.Replace("08/15", "Null-Acht-Fünfzehn")

        'Satzzeichen zählen
        Dim locPosition, locCount As Integer

        Do
            locPosition = locString.IndexOfAny(New Char() {"c", "c", ":", "?", "c"}, locPosition)
            If locPosition = -1 Then
                Exit Do
            Else
                locCount += 1
            End If
            locPosition += 1
        Loop

        Console.WriteLine("Der folgende Text...")
        Console.WriteLine(New String("c", 79))
        Console.WriteLine(locString)
        Console.WriteLine(New String("c", 79))
        Console.WriteLine("...verfügt über {0} Satzzeichen.", locCount)
        Console.WriteLine()
        Console.WriteLine("Und sieht nach dem Ersetzen von 'Big Apple' durch 'Großer Apfel' so aus:")
        Console.WriteLine(New String("c", 79))
```

```
'Noch eine Ersetzung
locString = locString.Replace("Big Apple", "Großer Apfel")
Console.WriteLine(locString)
Console.ReadLine()

End Sub

End Module
```

Dieses Beispiel gibt folgendes auf dem Bildschirm aus:

```
Der folgende Text...
=====
Weisheiten:
* Wenn man 8 Jahre, 7 Monate und 6 Tage schreien würde,
  hätte man genug Energie produziert, um eine Tasse Kaffee heiß zu machen.
* Wenn man seinen Kopf gegen die Wand schlägt, verbraucht man 150 Kalorien.
* Elefanten sind die einzigen Tiere, die nicht springen können.
* Eine Kakerlake kann 9 Tage ohne Kopf überleben, bevor sie verhungert.
* Gold und andere Metalle entstehen ausschließlich in
  Supernovae (Sternenexplosionen).
* Der Mond besteht aus den Trümmern der Kollision eines Mars großen
  Planeten mit der Erde.
* New York wird "Big Apple" genannt, weil "Big Apple" in der Sprache
  der Jazz-Musiker "das große Los ziehen" bedeutete. In New York Karriere
  zu machen war ihr großes Los.
* Der Ausdruck "Null-Acht-Fünfzehn" für etwas Unorigines war ursprünglich
  die Typenbezeichnung für das Maschinengewehr LMG Null-Acht-Fünfzehn;
  er wurde Metapher für geistlosen, militärischen Drill.
* "Durch die Lappen gehen" ist ein Begriff aus der Jagd:
  Hirsche liefen nicht durch eine aus Lappen bestehende,
  flatternde Umzäunung - aus Angst. Außer manchmal.
=====
...verfügt über 23 Satzzeichen.
```

Und sieht nach dem Ersetzen von 'Big Apple' durch 'Großer Apfel' so aus:

```
=====
Weisheiten:
* Wenn man 8 Jahre, 7 Monate und 6 Tage schreien würde,
  hätte man genug Energie produziert, um eine Tasse Kaffee heiß zu machen.
* Wenn man seinen Kopf gegen die Wand schlägt, verbraucht man 150 Kalorien.
* Elefanten sind die einzigen Tiere, die nicht springen können.
* Eine Kakerlake kann 9 Tage ohne Kopf überleben, bevor sie verhungert.
* Gold und andere Metalle entstehen ausschließlich in
  Supernovae (Sternenexplosionen).
* Der Mond besteht aus den Trümmern der Kollision eines Mars großen
  Planeten mit der Erde.
* New York wird "Großer Apfel" genannt, weil "Großer Apfel" in der Sprache
  der Jazz-Musiker "das große Los ziehen" bedeutete. In New York Karriere
  zu machen war ihr großes Los.
* Der Ausdruck "Null-Acht-Fünfzehn" für etwas Unorigines war ursprünglich
  die Typenbezeichnung für das Maschinengewehr LMG Null-Acht-Fünfzehn;
  er wurde Metapher für geistlosen, militärischen Drill.
* "Durch die Lappen gehen" ist ein Begriff aus der Jagd:
  Hirsche liefen nicht durch eine aus Lappen bestehende,
  flatternde Umzäunung - aus Angst. Außer manchmal.
```

TIPP Das Beispiel zu »Algorithmisches Auflösen eines Strings in Teile« auf Seite 242 enthält eine weitere, selbst geschriebene Funktion, die ich ReplaceEx genannt habe, und mit der Sie nach mehreren Zeichen suchen und, falls eines von ihnen dem durchsuchten Zeichen entsprach, es durch ein angebares Zeichen ersetzen können.

Trimmen von Strings

Visual Basic 6 kompatible(r) Befehl(e): Trim; RTrim; LTrim

Visual Basic .NET: strVar.Trim; strVar.TrimEnd; strVarTrimStart

Anmerkung: Mit diesen Befehlen können Sie überflüssige Zeichen am Anfang, am Ende oder an beiden Seiten eines Strings entfernen. Die Objektmethoden des Strings sind dabei den Kompatibilitätsfunktionen vorzuziehen, da Sie bei ersteren auch bestimmen können, welche Zeichen getrimmt werden sollen, wie das unten stehende Beispiel zeigt. Die VB6-Kompatibilitätsfunktionen beschränken ihre Trimmfähigkeit auf Leerzeichen.

Beispiel: Das folgende Beispiel generiert ein String-Array, dessen einzelne Elemente am Anfang und am Ende unerwünschte Zeichen (nicht nur Leerzeichen) haben, die durch die Trim-Funktion entfernt werden.

HINWEIS Dieses Beispiel zeigt dabei ebenfalls, dass Strings, obwohl sie als Referenztypen gelten, sich durch die Tatsache, dass sie an sich unveränderlich sind, anders verhalten als herkömmliche Objekte. Wenn Sie ein Objekt zwei Objektvariablen zuweisen und den Inhalt eines Objektes über eine Variable verändern, dann spiegelt die zweite Objektvariable ebenfalls den geänderten Inhalt des Objektes wider. Obwohl Strings Referenzvariablen sind, zeigen sie dennoch dieses Verhalten nicht, da Sie den String-Inhalt nicht verändern können. Strings werden immer nur neu erstellt, nie verändert (lesen Sie Weiteres zu diesem Thema im Abschnitt »Strings sind unveränderlich« auf Seite 235).

```
Dim locStringArray() As String = {
    " - Hier geht der eigentliche Text los!", _
    "Dieser Text endet mit komischen Zeichen! .-", _
    " - Hier sind beide Seiten problematisch - "}
    
For Each locString As String In locStringArray
    locString = locString.Trim(New Char() {" ", ".", "-", "c"})
    Console.WriteLine("Sauber und ordentlich: " + locString)
Next

'Wichtig: String ist zwar ein Referenztyp, am Array hat sich dennoch nichts verändert.
'Das liegt daran, dass Strings nicht direkt verändert, sondern immer neu – dabei verändert – angelegt
'reden.
For Each locString As String In locStringArray
    Console.WriteLine("Immer noch unordentlich: " + locString)
Next
```

Wenn Sie dieses Programm laufen lassen, generiert es die folgende Ausgabe:

```
Sauber und ordentlich: Hier geht der eigentliche Text los!
Sauber und ordentlich: Dieser Text endet mit komischen Zeichen!
Sauber und ordentlich: Hier sind beide Seiten problematisch
Immer noch unordentlich:  - Hier geht der eigentliche Text los!
Immer noch unordentlich: Dieser Text endet mit komischen Zeichen! .-
Immer noch unordentlich:  - Hier sind beide Seiten problematisch -
```

Algorithmisches Aulösen eines Strings in Teile

Visual Basic 6 kompatible(r) Befehl(e): Split

Visual Basic .NET: strVar.Split

Anmerkung: Die .NET-Split-Methode des String-Objektes ist der Kompatibilitätsfunktion insofern überlegen, als sie erlaubt, mehrere Separatorzeichen in einem Char-Array anzugeben. Damit werden Ihre Programme ungleich flexibler bei der Analyse und dem Neuaufbau von Texten.

Beispiel: Das folgende Beispiel zerlegt die einzelnen, durch verschiedene Separatorzeichen getrennten Begriffe oder Abschnitte eines Strings in Teilstrings, die anschließend als Elemente eines String-Arrays vorliegen und durch weitere Funktionen noch besser aufbereitet werden.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 07\\String - Split

Öffnen Sie dort die Projektmappe (.SLN-Datei).

```
Module Strings
Sub Main()
    Dim locString As String =
        "Einzelne, Elemente; durch, die , verschiedenen - Zeichen , getrennt."
    Console.WriteLine("Aus der Zeile:")
    Console.WriteLine(locString)
    Console.WriteLine()
    Console.WriteLine("Wird ein String-Array mit folgenden Elementen:")
    Dim locStringArray As String()
    locStringArray = locString.Split(New Char() {"c", ";", "-", ".})
    For Each locStr As String In locStringArray
        Console.WriteLine(ReplaceEx(locStr, New Char() {"c", ";", "-", ".}, _
            Convert.ToChar(vbNullChar)).Trim)
    Next
    Console.ReadLine()
End Sub

Public Function ReplaceEx(ByVal str As String, ByVal SearchChars As Char(), _
    ByVal ReplaceChar As Char) As String
    Dim locPos As Integer
    Do
        locPos = str.IndexOfAny(SearchChars)
        If locPos = -1 Then Exit Do
        If AscW(ReplaceChar) = 0 Then
            str = str.Remove(locPos, 1)
        Else
            str = str.Remove(locPos, 1).Insert(locPos, ReplaceChar.ToString)
        End If
    Loop
    Return str
End Function
End Module
```

Wenn Sie dieses Programm laufen lassen, generiert es die folgende Ausgabe:

Aus der Zeile:
Einzelne, Elemente; durch, die , verschiedensten - Zeichen , getrennt.

Wird ein String-Array mit folgenden Elementen:
Einzelne
Elemente
durch
die
verschiedensten
Zeichen
getrennt

Ein String-Schmankerl zum Schluss

Ich muss gestehen, ich habe ein Faible für bestimmte Fernsehserien. Besonders angetan haben es mir *Emergency Room* und *Star Trek Enterprise*.¹¹ Mein Faible geht so weit, dass ich – natürlich nur für den eigenen, privaten Bedarf – nicht nur eine Folge verpassen will, sondern sie mir auch gerne noch mal anschau. Aus dem Grund habe ich einen DVD-Rekorder, mit dem ich die Folgen aufzeichne und anschließend auf meinem Computer ins DivX-Format umrechne – natürlich nur für mich privat – damit ich sie mir auch auf meinem Notebook anschauen kann, wenn ich auf Reisen bin. Manchmal jedoch vergesse ich, eine Folge aufzuzeichnen, und dann kommt mein guter Freund Christian ins Spiel. Auch er hat ein Faible für Enterprise und ER und springt beim Aufnehmen manchmal für mich ein, wenn ich einmal vergessen habe, den Rekorder zu programmieren. Nur leider benennt er die für ausschließlich seinen eigenen Bedarf ins DivX-Format umgewandelten Videodateien nach einem anderen System. Er verwendet anstelle von Leerzeichen häufig den Unterstrich. Außerdem kennzeichnet er Staffelnummer und Episode nicht – so wie ich – im Format »sxee«, sondern setzt noch jeweils ein »S« und ein »E« davor. Aber er macht das auch nicht immer. So passiert es immer wieder, dass ich eine Staffel komplett auf der Platte gespeichert habe, aber die Dateinamen irgendwie unschön unregelmäßig benannt sind, etwa so:

```
F:\Video\ER\ER - 9x01 - chaos theory.mpg
F:\Video\ER\ER - 9x02 - dead again.mpg
F:\Video\ER\ER - 9x03 - insurrection.mpg
F:\Video\ER\ER - 9x04 - walk like a man.mpg
F:\Video\ER\ER - 9x05 - a hopeless wound.mpg
F:\Video\ER\ER - S09E11 - A Little Help from My Friends.von_Christian.mpg
F:\Video\ER\ER - S09E14 - No strings Attached.von_Christian.mpg
F:\Video\ER\ER - S09E15 - A Boy Falling out of the Sky.von_Christian.mpg
F:\Video\ER\ER - S09E19 - Things Change.von_Christian.mpg
F:\Video\ER\ER - S09E20 - Foreign.Affairs.von_Christian.mpg
```

Sie sehen an der Liste, dass das manuelle Umbenennen dieser Dateien eine Ewigkeit benötigen würde, um die Dateinamen in ein einheitliches Format zu bringen. Aus diesem Grund habe ich ein kleines Programm geschrieben, das das Umbenennen enorm erleichtert. Mit ihm können Sie nicht nur die Nummerierung von Episodendateien in mein bevorzugtes¹² Format durchführen lassen – Sie können sogar in den Dateinamen ein Suchen und Ersetzen ausführen, um – in meinem Fall – das nervige »von_Christian«¹³ zu entfernen.

¹¹ Und ich verstehе nicht, wieso Star Trek Enterprise nach nur 4 Seasons eingestellt werden musste...

¹² Mit ein paar Handgriffen können Sie natürlich das Programm so ändern, dass Ihr bevorzugtes Format dargestellt wird.

¹³ No offense, Kricke!

Sie können dieses Tool natürlich auch für andere Aufgaben einsetzen. Es hat mir bei diesem Buch beispielsweise mehrfach geholfen, die Bilder, die Kapitelnummerpräfixe trugen, auf neue Kapitelnummern anzupassen – Anwendungen für das Tool gibt es viele.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 07\\Dateinamenangleicher

Öffnen Sie dort die Projektmappe (.SLN-Datei).

Wenn Sie das Programm starten, sehen Sie einen Dialog, der in seiner Größe beliebig veränderbar ist, etwa wie in Abbildung 7.2 zu sehen.

In der linken Spalte sehen Sie die Originaldateinamen. In der rechten Spalte sehen Sie die durch Ihre Eingriffe veränderten. Wichtig: Das Programm benennt die Dateinamen erst dann physisch auf der Festplatte um, wenn Sie die Schaltfläche *Markierte Dateien umbenennen* anklicken – Sie können also beruhigt mit den Dateinamen experimentieren, ohne Angst haben zu müssen, dass Sie sich Dateinamen »zerschießen«.

Sie finden in den Unterverzeichnissen *Testfiles* und *Backup Testfiles* Dateien (natürlich leere Testdateien), mit denen Sie experimentieren können. Die Liste dieser Dateien ist übrigens auch in der Abbildung zu sehen. Um das angezeigte Verzeichnis zu wechseln, klicken Sie auf die Schaltfläche *Verzeichnis*. Die Dateinamenliste wird dann eingelesen und dargestellt.

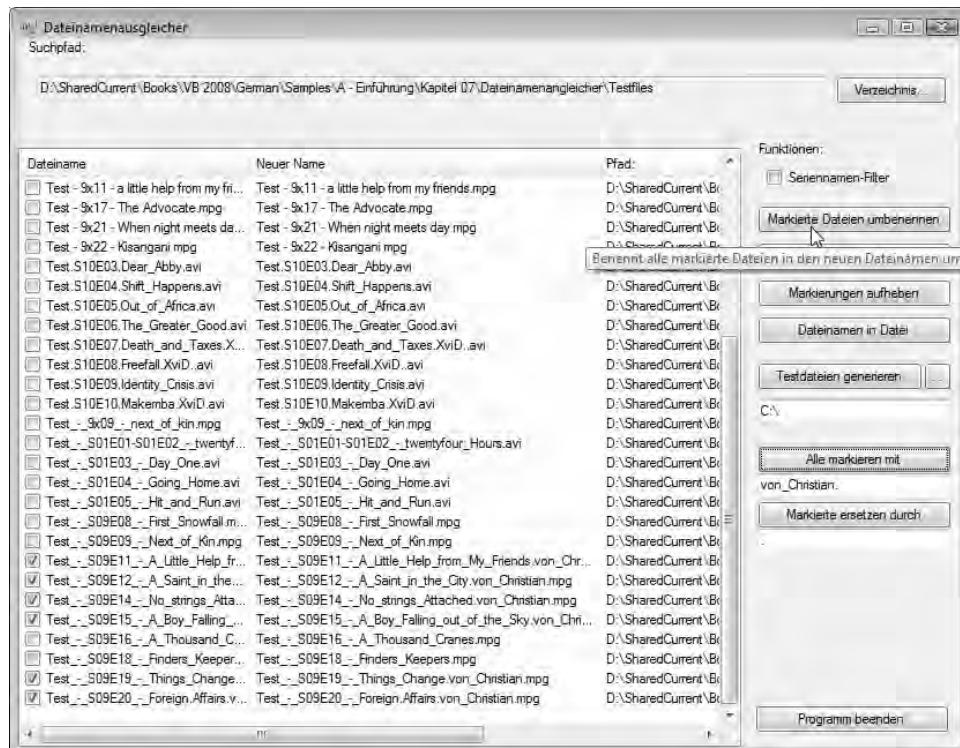


Abbildung 7.2 Mit dem Dateinamenausgleicher können Sie Dateinamen algorithmisch umbenennen; auch das Suchen und Ersetzen in Dateinamen ist damit möglich

Möchten Sie den Algorithmus zum Ausgleichen des Dateinamens für Seriennamen nicht anwenden, entfernen Sie einfach das Häkchen vor *Seriennamen-Filter*.

Die anderen Funktionen lassen Sie sich am besten durch das Programm erklären: Hinter jeder Schaltfläche verbirgt sich ein Tooltip, der Sie über die jeweilige Funktion aufklärt.

Erst, wenn Sie alle Veränderungen vorgenommen haben, klicken Sie auf die Schaltfläche *Markierte Dateien umbenennen*, um Ihre Änderungen zu übernehmen.

Und jetzt, nachdem Sie wissen, wie Sie das Programm anwenden, werden Sie eine ungefähre Vorstellung davon haben, dass String-Funktionen bei seiner Programmierung nicht zu kurz kamen. Allerdings werde ich mich in diesem Zusammenhang – aus Platzgründen – bei der Erklärung des Programms auf die String-relevanten Funktionen beschränken.

Nur soviel zur generellen Funktion des Programms: Es verwendet keine eigenen Klassen zur Speicherung der Daten, sondern erweitert die vorhandenen Klassen `ListView` für die Darstellung der Dateien, und `ListViewItem` für einen einzelnen darzustellenden Eintrag innerhalb der `ListView`. Durch diese Vorgehensweise wird es extrem kompakt.

Die für die String-Verarbeitung interessanten Punkte befinden sich im Seriennamen-Algorithmus, der über die Eigenschaft `NewFilename` der aus `ListViewItem` abgeleiteten Klasse `FilenameEnumeratorItem` realisiert wird:

```
Public Overridable ReadOnly Property NewFilename() As FileInfo
    Get
        Dim locFilename As String = myFilename.Name
        Dim locParts As New ArrayList
        Dim blnCharTypesChanged As Boolean
        Dim locChar, locPrevChar As Char
        Dim locCurrentPart As String
        Dim locNumberStartPart, locNumberEndPart As Integer
        Dim locProhibitFurtherCharTypeChanges As Boolean
        Dim locPrefix, locNumPart, locPostfix As String
        If Not myEpisodeNameFilter Then
            Return myFilename
        End If

        'Finden des Nummern-Parts und Ersetzen aller Unterstriche
        'durch Leerzeichen.
        For count As Integer = 0 To locFilename.Length - 1
            locChar = locFilename.Chars(count)
            If locChar = "_"c Then locChar = " "c

            'Den Nummernpart des Dateinamens suchen.
            If Not locProhibitFurtherCharTypeChanges Then
                If locChar.IsDigit(locChar) And Not blnCharTypesChanged Then
                    blnCharTypesChanged = True
                    'Falls "S" oder "s" davor stand, Buchstaben mit einbeziehen.
                    If locPrevChar = "S"c Or locPrevChar = "s"c Then
                        locNumberStartPart = count - 1
                    Else
                        locNumberStartPart = count
                    End If
                End If
            End If
        End If
    End Get

```

```

If Not locProhibitFurtherCharTypeChanges Then
    'Wenn Nummernpart schon vorbei, und wieder ein Buchstabe...
    If Char.IsLetter(locChar) And bInCharTypesChanged Then
        If count < locFilename.Length - 2 Then
            Dim locNextChar As Char = locFilename.Chars(count + 1)
            '...aber nur wenn der Buchstabe kein Episodenkennzeichner ist...
            If Not ((Char.IsLetter(locChar) And Char.IsDigit(locNextChar)) And
                (locChar = "E"c Or locChar = "e"c Or locChar = "x" Or locChar = "X"))
Then
    '...ist der Nummernpart vorbei, und es folgt wieder Text.
    locNumberEndPart = count
    locProhibitFurtherCharTypeChanges = True
    End If
    End If
    End If
    locCurrentPart += locChar.ToString
    locPrevChar = locChar
Next

'Sonderfall: Dateiname endet mit Nummer.
If locNumberEndPart = 0 Then
    locNumberEndPart = locFilenameLength
End If

'Dateinamen auseinander bauen.
locPräfix = locCurrentPart.Substring(0, locNumberStartPart)
locNumPart = locCurrentPart.Substring(locNumberStartPart, _
    locNumberEndPart - locNumberStartPart)
locPostfix = locCurrentPart.Substring(locNumberEndPart)
'Alle denkbaren "Umbauten" im Dateinamen durchführen.
locPräfix = locPräfix.Replace(".c, " "c")
locPräfix = locPräfix.Trim(New Char() {" "c, "-c"})
locNumPart = locNumPart.Replace(".c, """)
locNumPart = locNumPart.Replace("S"c, """
locNumPart = locNumPart.Replace("s"c, """
locNumPart = locNumPart.Replace("E"c, "x"c)
locNumPart = locNumPart.Replace("e"c, "x"c)
locNumPart = locNumPart.Replace("X"c, "x"c)
locNumPart = locNumPart.Trim(New Char() {" "c, "-c"})
locPostfix = locPostfix.Trim(New Char() {" "c, "-c"})

'Und neuen Dateinamen zurückliefern.
Return New FileInfo(Filename.DirectoryName + "\" + locPräfix + " - " + locNumPart + _
    " - " + locPostfix)
End Get
End Property

```

Iterieren durch einen String

Sie sehen, dass diese Routine für das Iterieren durch die einzelnen Buchstaben eine weitere Variation verwendet: Mit der `Chars`-Eigenschaft eines `String`-Objekts greifen Sie auf ein Char-Array zu, das die einzelnen Zeichen des Strings repräsentiert. Da das `String`-Objekt auch die Funktion `GetEnumerator` anbietet, gäbe es auch die folgende Möglichkeit, durch einen String zu iterieren:

```
For Each locChar As Char In "Dies ist ein String"  
    'Tu irgendwas.  
Next
```

Durch die Mächtigkeit des `String`-Objektes sind die Funktionen Suchen und Ersetzen des Programms erschreckend einfach zu realisieren, wie der folgende Codeausschnitt eindrucksvoll zeigt:

```
Private Sub btnCheckFound_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles btnCheckFound.Click  
    For Each locFEI As FilenameEnumeratorItem In fneFiles.Items  
        Dim locFilename As String = locFEI.Filename.Name  
        If locFilename.IndexOf(txtSearch.Text) > -1 Then  
            locFEI.Checked = True  
        Else  
            locFEI.Checked = False  
        End If  
    Next  
End Sub  
  
Private Sub btnReplaceChecked_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles btnReplaceChecked.Click  
    For Each locFEI As FilenameEnumeratorItem In fneFiles.Items  
        Dim locFilename As String = locFEI.SubItems(1).Text  
        If locFEI.Checked Then  
            If locFilename.IndexOf(txtSearch.Text) > -1 Then  
                locFilename = locFilename.Replace(txtSearch.Text, txtReplace.Text)  
                locFEI.SubItems(1).Text = locFilename  
            End If  
        End If  
    Next  
End Sub
```

StringBuilder vs. String – wenn es auf Geschwindigkeit ankommt

Sie haben im Laufe des String-Abschnittes gesehen, dass Ihnen .NET mit dem Datentyp `String` ein mächtiges Werkzeug für die Bearbeitung von Zeichenketten in die Hände legt. Wenn Sie den Abschnitt über die Speicherverwaltung gelesen haben, dann werden Sie aber auch bemerkt haben, dass es um die Geschwindigkeit bei der Verarbeitung von Strings in bestimmten Szenarien nicht so gut bestellt ist. Der Grund dafür ist einfach: Strings sind unveränderlich. Wenn Sie mit Algorithmen hantieren, die Strings im Laufe ihrer Entstehungsgeschichte zeichenweise zusammensetzen, dann wird für jedes Zeichen, das zum String hinzukommt, ein komplett neuer String erstellt. Und das kostet Zeit.

Eine Alternative dazu bildet die `StringBuilder`-Klasse. Sie hat bei weitem nicht die Funktionsvielfalt eines `Strings`, aber sie hat einen entscheidenden Vorteil: Sie wird dynamisch verwaltet und ist damit ungleich schneller. Das heißt für Sie: Wann immer es darum geht, Strings zusammenzusetzen (indem Sie Zeichen anhängen, einfügen oder löschen), sollten Sie ein `StringBuilder`-Objekt einsetzen – gerade wenn große Datenmengen im Spiel sind.

Der Umgang mit einem `StringBuilder`-Objekt ist denkbar einfach. Um auf das Objekt zurückgreifen zu können, benötigen Sie Zugriff auf den Namensbereich `Text`, den Sie mit der Anweisung

```
Imports System.Text
```

in Ihre Klassen- oder Moduldatei einbinden.

Sie deklarieren eine Variable einfach vom Typ `StringBuilder` und definieren sie mit einer der folgenden Anweisungen:

```
'Deklaration ohne Parameter:  
Dim locSB As New StringBuilder  
'Deklaration mit Kapazitätsreservierung  
locSB = New StringBuilder(1000)  
'Deklaration aus einem vorhandenen String  
locSB = New StringBuilder("Aus einem neuen String entstanden")  
'Deklaration aus String mit der Angabe einer zu reservierenden Kapazität  
locSB = New StringBuilder("Aus String entstanden mit Kapazität für weitere", 1000)
```

Sie können, falls Sie das möchten, eine Ausgangskapazität bei der Definition eines `StringBuilder`-Objektes angeben. Damit wird der Platz, den Ihr `StringBuilder`-Objekt voraussichtlich benötigen wird, direkt reserviert – zusätzlicher Speicher muss zur Laufzeit nicht angefordert werden, und das spart zusätzlich Zeit.

Um den String um Zeichen zu erweitern, verwenden Sie die `Append`-Methode. Mit `Insert` können Sie weitere Zeichen in ein `StringBuilder`-Objekt einfügen. `Replace` erlaubt Ihnen das Ersetzen einer Zeichenkette durch eine andere. Und mit `Remove` haben Sie die Möglichkeit, eine bestimmbare Anzahl von Zeichen ab einer bestimmten Zeichenposition zu entfernen.

Beispiel:

```
locSB.Append(" - und das wird an den String angefügt")  
locSB.Insert(20, ">>das kommt irgendwo in die Mitte<<")  
locSB.Replace("String", "StringBuilder")  
locSB.Remove(0, 4)
```

Wenn der String komplett zusammengesetzt wurde, können Sie ihn mit der `Tostring`-Funktion in einen »echten« String umwandeln:

```
'StringBuilder hat den String fertig zusammengesetzt,  
'in String umwandeln.  
Dim locString As String = locSB.ToString  
Console.WriteLine(locString)
```

Führten Sie dieses Beispiel aus, würde es folgenden Text im Konsolenfenster anzeigen:

StringBuilder entstanden>> das kommt irgendwo in die Mitte<<n mit Kapazität für weitere - und das wird an den StringBuilder angefügt

Performance-Vergleich: String gegen StringBuilder

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 07\\StringVsStringBuilder

Öffnen Sie dort die Projektmappe (.SLN-Datei).

Das Programm kreiert eine bestimmmbare Anzahl von String-Elementen, die jeweils aus einer ebenfalls bestimmmbaren Menge aus zufälligen Zeichen bestehen. Wenn Sie das Programm starten, bestimmen Sie diese Parameter:

```
Geben Sie die String-Länge eines Elementes ein: 100
Geben die Anzahl der zu erzeugenden Elemente ein: 100000

Erzeugen von 100000 Stringelementen mit der String-Klasse...
Dauer: 2294 Millisekunden

Erzeugen von 100000 Stringelementen mit der StringBuilder-Klasse...
Dauer: 1111 Millisekunden
```

Sie sehen, dass bei einer Elementlänge von *100* Zeichen die Verwendung der *StringBuilder*-Klasse bereits eine Verdopplung der Geschwindigkeit mit sich bringt.

Starten Sie anschließend das Programm erneut. Geben Sie für die Elementlänge *1000* ein und bestimmen Sie für die Anzahl der zu erzeugenden Elemente den Wert *10000*. Die Geschwindigkeitsausbeute ist jetzt noch beeindruckender:

```
Geben Sie die String-Länge eines Elementes ein: 1000
Geben die Anzahl der zu erzeugenden Elemente ein: 10000

Erzeugen von 10000 Stringelementen mit der String-Klasse...
Dauer: 6983 Millisekunden

Erzeugen von 10000 Stringelementen mit der StringBuilder-Klasse...
Dauer: 1091 Millisekunden
```

Mit diesen Parametern ist der *StringBuilder* ca. um den Faktor 6 schneller – im Vergleich zum normalen *String*-Objekt!

Je mehr Zeichen für einen String zur Laufzeit generiert werden müssen, desto mehr lohnt sich der Einsatz eines *StringBuilder*-Objektes.

Das Programm selbst greift für die Messungen übrigens auf die Klasse *HighSpeedTimeGauge* zurück, zu dem der anschließende graue Kasten mehr zu sagen weiß. Das folgende Listing zeigt seine Verwendungsweise:

```
Imports System.Text

Module StringsVsStringBuilder

    Sub Main()
        Dim locTimeGauge As New HighSpeedTimeGauge
        Dim locAmountElements As Integer
        Dim locAmountCharsPerElement As Integer
        Dim locVBStringElements As VBStringElements
        Dim locVBStringBuilderElements As VBStringBuilderElements

        'StringBuilderBeispiele()
        'Return

        Console.WriteLine("Geben Sie die String-Länge eines Elementes ein: ")
        locAmountCharsPerElement = Integer.Parse(Console.ReadLine)
        Console.WriteLine("Geben die Anzahl der zu erzeugenden Elemente ein: ")
        locAmountElements = Integer.Parse(Console.ReadLine)
        Console.WriteLine()
        Console.WriteLine("Erzeugen von " & locAmountElements &
            " Stringelementen mit der String-Klasse...")
        locTimeGauge.Start()
        locVBStringElements = New VBStringElements(locAmountElements, locAmountCharsPerElement)
        locTimeGauge.Stop()
        Console.WriteLine("Dauer: " & locTimeGauge.ToString())
        locTimeGauge.Reset()
        Console.WriteLine()
        locTimeGauge.Reset()
        Console.WriteLine("Erzeugen von " & locAmountElements &
            " Stringelementen mit der StringBuilder-Klasse...")
        locTimeGauge.Start()
        locVBStringBuilderElements = New VBStringBuilderElements(locAmountElements, _
            locAmountCharsPerElement)
        locTimeGauge.Stop()
        Console.WriteLine("Dauer: " & locTimeGauge.ToString())
        locTimeGauge.Reset()
        Console.WriteLine()
        Console.ReadLine()
    End Sub

    Sub StringBuilderBeispiele()

        'Deklaration ohne Parameter:
        Dim locSB As New StringBuilder
        'Deklaration mit Kapazitätsreservierung
        locSB = New StringBuilder(1000)
        'Deklaration aus einem vorhandenen String
        locSB = New StringBuilder("Aus einem neuen String entstanden")
        'Deklaration aus String mit der Angabe einer zu reservierenden Kapazität
        locSB = New StringBuilder("Aus String entstanden mit Kapazität für weitere", 1000)

        locSB.Append(" - und das wird an den String angefügt")
        locSB.Insert(20, ">>das kommt irgendwo in die Mitte<<")
        locSB.Replace("String", "StringBuilder")
        locSB.Remove(0, 4)
    End Sub

```

```
'StringBuilder hat den String fertig zusammengesetzt,  
'in String umwandeln  
Dim locString As String = locSB.ToString  
Console.WriteLine(locString)  
Console.ReadLine()  
End Sub  
End Module  
  
Public Class VBStringElements  
    Private myStrElements() As String  
  
    Sub New(ByVal AmountOfElements As Integer, ByVal AmountChars As Integer)  
        ReDim myStrElements(AmountOfElements - 1)  
        Dim locRandom As New Random(DateTime.Now.Millisecond)  
        Dim locString As String  
  
        For locOutCount As Integer = 0 To AmountOfElements - 1  
            locString = ""  
            For locInCount As Integer = 0 To AmountChars - 1  
                Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)  
                If locIntTemp > 26 Then  
                    locIntTemp += 97 - 26  
                Else  
                    locIntTemp += 65  
                End If  
                locString += Convert.ToChar(locIntTemp).ToString  
            Next  
            myStrElements(locOutCount) = locString  
        Next  
    End Sub  
End Class  
Public Class VBStringBuilderElements  
  
    Private myStrElements() As String  
  
    Sub New(ByVal AmountOfElements As Integer, ByVal AmountChars As Integer)  
        ReDim myStrElements(AmountOfElements - 1)  
        Dim locRandom As New Random(DateTime.Now.Millisecond)  
        Dim locStringBuilder As StringBuilder  
  
        For locOutCount As Integer = 0 To AmountOfElements - 1  
            locStringBuilder = New StringBuilder(AmountChars)  
            For locInCount As Integer = 0 To AmountChars - 1  
                Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)  
                If locIntTemp > 26 Then  
                    locIntTemp += 97 - 26  
                Else  
                    locIntTemp += 65  
                End If  
                locStringBuilder.Append(Convert.ToChar(locIntTemp))  
            Next  
            myStrElements(locOutCount) = locStringBuilder.ToString  
        Next  
    End Sub  
End Class
```

Wrapper-Klassen für Betriebssystemaufrufe

Es ist eine allgemein übliche Vorgehensweise, Aufrufe an das Windows-Betriebssystem durch so genannte Wrapper-Klassen zu realisieren. Eine Wrapper-Klasse kapselt¹⁴ Betriebssystemaufrufe, sodass sie auf gewohnte »Framework«-Weise aufrufbar sein. Wrapper-Klassen stellen dazu ganze Funktionsbibliotheken zur Verfügung, die nur als Schnittstelle zu den dafür benötigten Betriebssystemen fungieren, die aber die eigentliche Aufgabe übernehmen. Viele Steuerelemente von Windows sind auf diese Weise im Framework integriert.

Schon ein simples TextBox-Steuerelement ist im Framework nicht von Grund auf neu entwickelt worden. Vielmehr bildet die TextBox-Klasse des Frameworks lediglich einen Wrapper um die Windows-TextBox und stellt entsprechende Bearbeitungsfunktionen FCL-konform zur Verfügung.

Die im Beispiel verwendeten Betriebssystemaufrufe steuern den Performance-Counter (etwa: Leistungsmesser), der durch den Windows-Kernel zur Verfügung gestellt wird. Er erlaubt eine viel genauere Zeitmessung, insbesondere von extrem kurz andauernden Operationen und liefert daher aussagekräftigere Zahlen, als Sie diese über die normalen DateTime-Funktionen ermitteln könnten. Das folgende Listing zeigt die Funktionsweise:

```
Option Explicit On
Option Strict On

Public Class HighSpeedTimeGauge

    'Die Routinen brauchen wir zum "Hochgeschwindigkeitsmessen" aus dem Kernel
    Declare Auto Function QueryPerformanceFrequency Lib "Kernel32" (ByRef lpFrequency As Long) _
        As Boolean
    Declare Auto Function QueryPerformanceCounter Lib "Kernel32" (ByRef lpPerformanceCount As Long) _
        As Boolean

    'So ginge es übrigens auch:
    '<System.Runtime.InteropServices.DllImport("KERNEL32")>
    'Private Shared Function QueryPerformanceCounter(ByRef lpPerformanceCount As Long) As Boolean
    'End Function

    '<System.Runtime.InteropServices.DllImport("KERNEL32")>
    'Private Shared Function QueryPerformanceFrequency(ByRef lpFrequency As Long) As Boolean
    'End Function

    Private myStartTime As Long = 0
    Private myEndTime As Long = 0
    Private myDuration As Long = 0
    Private myFrequency As Long = 0

    Public Sub New()
        QueryPerformanceFrequency(myFrequency)
    End Sub
```

¹⁴ von engl. »to wrap«: einpacken (etwa in Geschenkpapier)

```
Public Sub Start()

    myStartValue = 0
    QueryPerformanceCounter(myStartValue)

End Sub

Public Sub [Stop]()

    myEndValue = 0
    QueryPerformanceCounter(myEndValue)

    myDuration = myEndValue - myStartValue

End Sub

Public Sub Reset()

    myStartValue = 0
    myEndValue = 0
    myDuration = 0
End Sub
Public ReadOnly Property DurationInSeconds() As Double
    Get
        Return CDbl(myDuration) / CDbl(myFrequency)
    End Get
End Property

Public ReadOnly Property DurationInMilliseconds() As Long
    Get
        Return CLng(1000 * DurationInSeconds)
    End Get
End Property

Public ReadOnly Property Frequency() As Long
    Get
        Return myFrequency
    End Get
End Property

Public Overrides Function ToString() As String
    Return DurationInMilliseconds & " Millisekunden"
End Function
End Class
```

Der Einsatz dieser Klasse ist denkbar einfach. Mit einer Klasseninstanz haben Sie Zugriff auf die Funktionen Start, Stop und Reset, mit denen Sie den Performance-Counter starten, anhalten und zurücksetzen können. Die Funktion ToString liefert Ihnen eine gemessene Zeitperiode in Millisekunden zurück.

Der Datentyp Boolean

Der Boolean-Datentyp speichert binäre Zustände, also eigentlich nicht viel: Sein Wert kann entweder *falsch* oder *wahr* sein – etwas anderes kann er nicht speichern. Dieser Datentyp wird am häufigsten bei der Ausführung von bedingtem Programmcode verwendet; zu diesem Thema haben Sie bereits in Kapitel 2 alles wesentliche erfahren.

.NET-Datentyp: System.Boolean

Stellt dar: einen von zwei Zuständen – True oder False

Typliteral: keins vorhanden

Speicherbedarf: 2 Byte

Delegation an den Prozessor: ja, als Integer

Anmerkungen:

Wenn Sie eine boolesche Variable definieren wollen, verwenden Sie dazu die Schlüsselwörter True und False direkt und ohne Anführungszeichen im Programmtext, etwa wie im folgenden Beispiel:

```
Dim locBoolean As Boolean
locBoolean = True ' Ausdruck ist 'wahr'.
locBoolean = False ' Ausdruck ist 'falsch'.
```

Konvertieren von und in numerische Datentypen

Sie können einen booleschen Typ in einen numerischen Datentyp umwandeln.

WICHTIG Beachten Sie, dass Visual Basic .NET bei der internen Darstellung des primitiven Datentyps Boolean vom Framework abweicht. Wenn Sie mit Visual Basic-Befehlen einen Boolean- in beispielsweise einen Integer-Datentyp umwandeln, wird der Wert True in -1 umgewandelt. Wenn Sie mit Framework-Konvertierungen – also beispielsweise der Convert-Klasse – arbeiten, wird True zu +1 umgewandelt.

Das folgende Beispiel verdeutlicht, was gemeint ist:

```
Dim locInt As Integer = CInt(locBoolean)      ' locInt ist -1
locInt = Convert.ToInt32(locBoolean)           ' locInt ist jetzt +1!!!
Dim locLong As Long = CLng(locBoolean)         ' locLong ist -1
locLong = Convert.ToInt64(locBoolean)           ' locLong ist +1
```

Bei der umgekehrten Konvertierung ist das Verhalten der Convert-Klasse des Frameworks und den Konvertierungsanweisungen von Visual Basic .NET einerlei. Nur der Zahlenwert 0 ergibt das boolesche Ergebnis False, alle anderen Werte ergeben True, wie das folgende Beispiel zeigt.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 07\\Primitives03
```

Öffnen Sie dort die Projektmappe (.SLN-Datei).

```
locBoolean = CBool(-1)           ' locBoolean ist True.  
locBoolean = CBool(0)            ' locBoolean ist False.  
locBoolean = CBool(1)            ' locBoolean ist True.  
locBoolean = Convert.ToBoolean(-1) ' locBoolean ist True.  
locBoolean = Convert.ToBoolean(+1) ' locBoolean ist True.  
locBoolean = CBool(100)           ' locBoolean ist True.  
locBoolean = Convert.ToBoolean(100) ' locBoolean ist True.
```

Konvertierung von und in Strings

Wenn Sie einen booleschen Datentyp in eine Zeichenkette konvertieren – beispielsweise um ihren Status in einer Datei abzuspeichern –, wird der jeweilige Wert in eine Zeichenkette umgewandelt, die durch die statischen Nur-Lese-Eigenschaften `TrueString` und `FalseString` der Boolean-Struktur festgehalten sind. Sie ergeben in der aktuellen Framework-Version (2.0) »True« und »False«, egal, welche Kultureinstellungen für das jeweilige Framework gelten.

Bei der Konvertierung von `String` in `Boolean` ergibt jede Zeichenkette außer »True« den Wert `False`.

Der Datentyp Date

Mit dem `Date`-Datentyp speichern Sie Datumswerte. Er hilft Ihnen, Zeitdifferenzen zu berechnen, Datums-werte aus Zeichenketten einzulesen (zu parsen) und wieder formatiert in Zeichenketten zurückzuverwandeln.

.NET-Datentyp: `System.DateTime`

Stellt dar: Zeiten und Uhrzeiten im Bereich vom 1.1.0001 (0 Uhr) bis 31.12.9999 (23:59:59 Uhr) mit einer Auflösung von 100 Nanosekunden (diese Einheit wird als *Tick* bezeichnet)

Typliteral: keins vorhanden

Speicherbedarf: 8 Byte

Delegation an den Prozessor: ja, als `Long`

Anmerkungen: Da `Date`-Datentypen ebenfalls zu den primitiven Datentypen zählen, sind sie direkt durch Literale im Programmcode definierbar. Allerdings gilt hier das Gleiche wie bei numerischen Werten: Das englische bzw. amerikanische Darstellungsformat zählt. Falls Sie mit dieser Kultur – was die Zeitmessung und Schreibweise anbelangt – nicht so sehr vertraut sind, lassen Sie mich kurz die Besonderheiten erklären.

Datums-Werte werden in der Reihenfolge Monat/Tag/Jahr vorgenommen und durch Schrägstrich und nicht durch Punkt voneinander getrennt. Diese Schreibweise kann leicht für Verwirrung sorgen (und zu totalen Fehlbuchungen führen), wenn Sie nicht darüber Bescheid wissen. Beim Datum

12/17/2004

ahnen Sie noch, dass etwas nicht stimmt, da es keinen 17. Monat im Jahr gibt.

Aber es macht schon einen Unterschied, ob Sie das Datum

12/06/2004

als 12. Juni oder 6. Dezember interpretieren.

Ähnliches gilt für die Uhrzeit. Die 24-Stunden-Anzeige finden Sie in den USA vielleicht auf dem einen oder anderen Busfahrplan oder beim Militär. Ansonsten wird durch die Postfixe »AM« (für »ante meridiem«, etwa »am Vormittag«) und »PM« (»post meridiem«, etwa »am Nachmittag«) definiert, welches 3:00 Uhr beispielsweise gemeint ist. Kritisch wird es bei 12:00 (0:00 gibt es nicht!) – vielleicht haben Sie selbst schon mal die Erfahrung gemacht, ein amerikanisches Videorekordermodell zu programmieren, das nicht die gewünschte Sendung, sondern eine andere, 12 Stunden später ausgestrahlte, aufgenommen hat.

12:00 AM entspricht unseren 0:00 Uhr, also Mitternacht; 12:00 PM entspricht Mittag.

Wertzuweisungen an einen Date-Datentyp im Programmcode werden durch Einklammern in das Nummernzeichen (»#«) realisiert. Die folgenden Beispiele zeigen, wie es geht:

```
Dim Mitternacht As Date = #12:00:00 AM#
Dim Mittag As Date = #12:00:00 PM#
Dim Silvester As System.DateTime = #12/31/2004#
Dim ZeitFürSekt As System.DateTime = #12/31/2004 11:58:00 PM#
Dim ZeitFürAsperin As System.DateTime = #1/1/2008 11:58:00 AM#
```

Der halbwegs intelligente Editor hilft Ihnen dabei ein wenig, das korrekte Format zu finden. Den Ausdruck

```
#0:00#
```

wandelt er beispielsweise selbstständig in

```
#12:00:00 AM#
```

um. Er ergänzt ebenfalls fehlende Sekundeneingaben oder Nullen, wenn die Eingabe eines Bereichs nur einstellig erfolgte. Sie können Uhrzeiten auch im 24-Stunden-Format eingeben; der Editor wandelt diese Zeit dann automatisch ins 12-Stunden-Format um.

Rechnen mit Zeiten und Datumswerten – TimeSpan

Das Besondere am Date-Datentyp ist, dass er das Errechnen von Zeitdifferenzen erlaubt. Um jedoch sinnvolle Ergebnisse, beispielsweise die Differenz zweier Datumswerte oder Zeitwerte darzustellen, ist der Date-Datentyp selbst nicht sonderlich geeignet. Aus diesem Grund gibt es einen weiteren Datentyp – TimeSpan – der Ergebnisse von Zeitberechnungen aufnimmt. Er selbst zählt allerdings nicht zu den primitiven Datentypen von .NET.

Der Umgang mit diesem Datentyp ist sehr einfach: Sie können entweder einen Datumswert von einem anderen subtrahieren, um die dazwischenliegende Zeitspanne zu ermitteln. Oder Sie erstellen einen TimeSpan-Datentyp und verwenden ihn, um einen bestimmten Zeitabschnitt auf ein Datum zu addieren oder es von ihm abzuziehen. Ein paar Beispielcodezeilen sollen das verdeutlichen.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\\VB 2008 Entwicklerbuch\\A - Einführung\\Kapitel 07\\DateTime
```

Öffnen Sie dort die Projektmappe (.SLN-Datei).

```
Dim locDate1 As Date = #3:15:00 PM#
Dim locDate2 As Date = #4:23:32 PM#
Dim locTimeSpan As TimeSpan = locDate2.Subtract(locDate1)
Console.WriteLine("Der Zeitunterschied zwischen {0} und {1} beträgt", _
    locDate1.ToString("HH:mm:ss"), -
    locDate2.ToString("HH:mm:ss"))
Console.WriteLine("{0} Sekunde(n) oder", locTimeSpan.TotalSeconds)
Console.WriteLine("{0} Minute(n) und {1} Sekunde(n) oder", -
    Math.Floor(locTimeSpan.TotalMinutes), -
    locTimeSpan.Seconds)
Console.WriteLine("{0} Stunde(n), {1} Minute(n) und {2} Sekunde(n) oder", -
    Math.Floor(locTimeSpan.TotalHours), -
    locTimeSpan.Minutes, locTimeSpan.Seconds)
Console.WriteLine("{0} Ticks", -
    locTimeSpan.Ticks)
```

Bibliothek mit brauchbaren Datumsrechenfunktionen

Sie finden im gleichen Beispiel eine Klassendatei namens *DateCalcHelper.vb*, die eine statische Klasse gleichen Namens enthält. Diese Klasse stellt einige, wie ich meine, ganz brauchbare Funktionen zur Verfügung, die einerseits die Berechnung bestimmter relativer Zeitpunkte vereinfacht und andererseits das Rechnen mit Datumswerten verdeutlicht.

Die Klasse erklärt sich durch die XML-Kommentare selber – das komplette Listing dieser Klasse finden Sie im Folgenden. Wenn Sie eigene Programme entwickeln, die intensiv von Berechnungen relativier Zeitpunkte Gebrauch machen, fügen Sie diese Codedatei einfach Ihrem Projekt (oder der Assembly Ihres Projekts) hinzu.

Zur besseren Orientierung sind die Funktionserklärungen und Methodennamen der einzelnen Funktionen im folgenden Listing in Fettschrift gesetzt,

```
Public NotInheritable Class DateCalcHelper

    ''' <summary>
    ''' Errechnet das Datum, das dem 1. des Monats entspricht,
    ''' der sich aus dem angegebenen Datum ergibt.
    ''' </summary>
    ''' <param name="CurrentDate">Datum, dessen Monat für die Berechnung zugrunde gelegt wird.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function FirstDayOfMonth(ByVal CurrentDate As Date) As Date
        Return New Date(CurrentDate.Year, CurrentDate.Month, 1)
    End Function

    ''' <summary>
    ''' Errechnet das Datum, das dem Letzen des Monats entspricht,
    ''' der sich aus dem angegebenen Datum ergibt.
    ''' </summary>
    ''' <param name="CurrentDate">Datum, dessen Monat für die Berechnung zugrunde gelegt wird.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function LastDayOfMonth(ByVal CurrentDate As Date) As Date
        Return New Date(CurrentDate.Year, CurrentDate.Month, 1).AddMonths(1).AddDays(-1)
    End Function
```

```
    /// <summary>
    /// Errechnet das Datum, das dem 1. des Jahres entspricht,
    /// das sich aus dem angegebenen Datum ergibt.
    /// </summary>
    /// <param name="CurrentDate">Datum, dessen Jahr für die Berechnung zugrunde gelegt wird.</param>
    /// <returns></returns>
    /// <remarks></remarks>
Public Shared Function FirstOfYear(ByVal CurrentDate As Date) As Date
    Return New Date(CurrentDate.Year, 1, 1)
End Function

    /// <summary>
    /// Errechnet das Datum, das dem ersten Montag der ersten Woche des Monats entspricht,
    /// der sich aus dem angegebenen Datum ergibt.
    /// </summary>
    /// <param name="CurrentDate">Datum, dessen Woche für die Berechnung zugrunde gelegt wird.</param>
    /// <returns></returns>
    /// <remarks></remarks>
Public Shared Function MondayOfFirstWeekOfMonth(ByVal CurrentDate As Date) As Date
    Dim locDate As Date = FirstDayOfMonth(CurrentDate)
    If Weekday(locDate) = DayOfWeek.Monday Then
        Return locDate
    End If
    Return locDate.AddDays(6 - Weekday(CurrentDate))
End Function

    /// <summary>
    /// Errechnet das Datum, das dem Montag der Woche entspricht,
    /// die sich aus dem angegebenen Datum ergibt.
    /// </summary>
    /// <param name="CurrentDate">Datum, dessen Woche für die Berechnung zugrunde gelegt wird.</param>
    /// <returns></returns>
    /// <remarks></remarks>
Public Shared Function MondayOfWeek(ByVal CurrentDate As Date) As Date
    If Weekday(CurrentDate) = DayOfWeek.Monday Then
        Return CurrentDate
    Else
        Return CurrentDate.AddDays(-Weekday(CurrentDate) + 1)
    End If
End Function

    /// <summary>
    /// Errechnet das Datum, das dem ersten Montag der zweiten Woche des Monats entspricht,
    /// der sich aus dem angegebenen Datum ergibt.
    /// </summary>
    /// <param name="CurrentDate">Datum, dessen Woche für die Berechnung zugrunde gelegt wird.</param>
    /// <returns></returns>
    /// <remarks></remarks>
Public Shared Function MondayOfSecondWeekOfMonth(ByVal currentDate As Date) As Date
    Return MondayOfFirstWeekOfMonth(currentDate).AddDays(7)
End Function

    /// <summary>
    /// Errechnet das Datum, das dem ersten Montag der letzten Woche des Monats entspricht,
    /// der sich aus dem angegebenen Datum ergibt.
    /// </summary>
```

```
''' </summary>
''' <param name="CurrentDate">Datum, dessen Woche für die Berechnung zugrunde gelegt wird.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function MondayOfLastWeekOfMonth(ByVal CurrentDate As Date) As Date
    Dim locDate As Date = FirstDayOfMonth(CurrentDate).AddDays(-1)
    If Weekday(locDate) = DayOfWeek.Monday Then
        Return locDate
    End If
    Return locDate.AddDays(-Weekday(CurrentDate) + 1)
End Function

''' <summary>
''' Ergibt das Datum des nächsten Arbeitstages.
''' </summary>
''' <param name="CurrentDate">Datum der Berechnungsgrundlage</param>
''' <param name="WorkOnSaturdays">True, wenn Samstag Arbeitstag ist.</param>
''' <param name="WorkOnSundays">True, wenn Sonntag Arbeitstag ist.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function NextWorkday(ByVal CurrentDate As Date, ByVal WorkOnSaturdays As Boolean, _
    ByVal WorkOnSundays As Boolean) As Date
    CurrentDate = CurrentDate.AddDays(1)
    If Weekday(CurrentDate) = DayOfWeek.Saturday And Not WorkOnSaturdays Then
        CurrentDate = CurrentDate.AddDays(1)
    End If
    If Weekday(CurrentDate) = DayOfWeek.Sunday And Not WorkOnSundays Then
        CurrentDate = CurrentDate.AddDays(1)
    End If
    Return CurrentDate
End Function

''' <summary>
''' Ergibt das Datum des vorherigen Arbeitstages.
''' </summary>
''' <param name="CurrentDate">Datum der Berechnungsgrundlage</param>
''' <param name="WorkOnSaturdays">True, wenn Samstag Arbeitstag ist.</param>
''' <param name="WorkOnSundays">True, wenn Sonntag Arbeitstag ist.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function PreviousWorkday(ByVal CurrentDate As Date, ByVal WorkOnSaturdays As Boolean, _
    ByVal WorkOnSundays As Boolean) As Date
    CurrentDate = CurrentDate.AddDays(-1)
    If Weekday(CurrentDate) = DayOfWeek.Sunday And Not WorkOnSundays Then
        CurrentDate = CurrentDate.AddDays(-1)
    End If
    If Weekday(CurrentDate) = DayOfWeek.Saturday And Not WorkOnSaturdays Then
        CurrentDate = CurrentDate.AddDays(-1)
    End If
    Return CurrentDate
End Function
End Class
```

Zeichenketten in Datumswerte wandeln

Genau wie die numerischen primitiven Datentypen können Sie Zeichenketten auch in einen Date-Datentyp umwandeln lassen. Allerdings stellt Ihnen der Date-Datentyp zwei Funktionen zur Verfügung, die eine Zeichenkette analysieren und den eigentlichen Datumswert daraus bilden: Parse und ParseExact.

Umwandlungen mit Parse

Wenn Sie Parse verwenden, versucht der Parser jeden Trick, den er kennt, um ein Datum, eine Zeit oder eine Kombination aus beiden in einen Zeitwert umzuwandeln, wie das folgende Beispiel zeigt:

```
Dim locToParse As Date
locToParse = Date.Parse("13.12.04") ' OK, deutsche Grundeinstellung wird verarbeitet.
locToParse = Date.Parse("6/7/04") ' OK, aber deutsch trotz "/" wird angewendet.
locToParse = Date.Parse("13/12/04") ' OK, wie oben.
locToParse = Date.Parse("06.07") ' OK, wird um Jahreszahl erweitert.
locToParse = Date.Parse("06,07,04") ' OK, Komma ist akzeptabel.
locToParse = Date.Parse("06,07") ' OK, Komma ist akzeptabel; Jahreszahl wird ergänzt.
'locToParse = Date.Parse("06072004") ' --> Exception: wurde nicht als gültiges Datum erkannt!
'locToParse = Date.Parse("060705") ' --> Exception: wurde nicht als gültiges Datum erkannt!
locToParse = Date.Parse("6,7,4") ' OK, Komma wird akzeptiert; führende Nullen werden ergänzt.

locToParse = Date.Parse("14:00") ' OK, 24-Stunden-Darstellung wird akzeptiert.
locToParse = Date.Parse("PM 11:00") ' OK, PM darf vorne...
locToParse = Date.Parse("11:00 PM") ' ...und auch hinter der Zeitangabe stehen.
'locToParse = Date.Parse("12,00 PM") ' --> Exception: wurde nicht als gültiges Datum erkannt!

'Beide Datum- Zeitkombinationen funktionieren:
locToParse = Date.Parse("6.7.04 13:12")
locToParse = Date.Parse("6,7,04 11:13 PM")
```

Sie sehen hier aber auch, dass ein in Deutschland sehr übliches EingabefORMAT nicht erkannt wird – wenn Sie nämlich die einzelnen Wertegruppen des Datums ohne irgendein Trennzeichen hintereinander schreiben. Doch auch dafür gibt es eine Lösung.

Umwandlungen mit ParseExact

Wenn Parse – so flexibel es auch sein mag – versagt, haben Sie die Möglichkeit, ein Erkennungsmuster für die Eingabe vorzugeben, indem Sie die Methode ParseExact benutzen, um die Zeichenkettenumwandlung vorzunehmen. ParseExact sollten Sie auch dann verwenden, wenn Sie die Eingabe eben nicht so flexibel gestalten möchten, wie Parse es vorsieht.

Insbesondere wenn es Zeitwerte und Datumswerte zu unterscheiden gilt, ist ParseExact der richtige Kandidat. In einem Feld, in dem vom Anwender Ihres Programms eine Zeiteingabe verlangt wird, weiß Ihr Programm genau, dass die Eingabe

23,12

die Uhrzeit 23:12:00 meint, und nicht den 23.12.2004. Parse kann den Kontext natürlich nicht erkennen und wird in diesem Fall nicht funktionieren.

`ParseExact` benötigt, neben dem zu analysierenden String, mindestens zwei weitere Parameter. Zum einen ist dies eine Zeichenkette, die das spezifische Erkennungsmuster enthält, und einen so genannten *Format Provider*, der weitere Formatierungsvorschriften vorgibt.

Es gibt verschiedene Format Provider, die Sie dafür verwenden können. Wichtig ist, dass Sie auf diese nur zugreifen können, wenn Sie zuvor folgende Zeile zur Einbindung des erforderlichen Namensbereichs am Anfang Ihres Moduls oder Ihrer Klassendatei eingefügt haben:

```
Imports System.Globalization
```

Die einfachste Version, um eine Uhrzeit als Uhrzeit zu erkennen, wenn sie im oben stehenden Format eingegeben würde, sähe beispielsweise folgendermaßen aus:

```
locToParse = Date.ParseExact("12,00", "HH,mm", CultureInfo.CurrentCulture)
```

Die Zeichenfolge »HH« besagt, dass Uhrzeiten (Hour = Stunde) im 24-Stunden-Format akzeptiert werden. Gäben Sie hier zwei kleine »h« ein, akzeptierte der Parser nur das englische 12-Stunden-Format. Mit dem anschließenden Komma geben Sie das Trennzeichen vor und anschließend die Minuten mit der Zeichenfolge »mm«.

Nun passiert es in der Praxis recht selten, dass sich Anwender an bestimmte Vorgaben halten, und darum sollte ihr Programm in der Lage sein, mehrere Versionen von Zeiteingabeformaten zu erkennen. Aus diesem Grund können Sie mit der `ParseExact`-Funktion gleich eine ganze Reihe von möglichen Formaten vorgeben, anhand derer der Parser die Umwandlung vornehmen kann. In diesem Fall definieren Sie einfach ein String-Array mit den zugelassenen Formaten und übergeben es anschließend der `ParseExact`-Methode zusammen mit der zu analysierenden Zeichenfolge. Wenn Sie sich zu dieser Methode des Parsens entschließen, müssen Sie jedoch noch einen weiteren Parameter bestimmen, der die Flexibilität des Parsens regelt (beispielsweise ob Leerzeichen im zu analysierenden String enthalten sein dürfen, die aber ignoriert werden sollen). Diese Angabe wird durch einen Parameter vom Typ `DateTimeStyles` geregelt, der folgende Einstellungen zulässt:

Member-Name	Beschreibung	Wert
AdjustToUniversal	Bestimmt, dass das Datum und die Zeit in koordinierte Weltzeit bzw. Greenwich Mean Time ¹⁵ (GMT) (Deutschland –1 Stunde) konvertiert werden müssen.	16
AllowInnerWhite	Bestimmt, dass zusätzliche Leerzeichen innerhalb der Zeichenfolge beim Analysieren ignoriert werden, es sei denn, die <code>DateTimeFormatInfo</code> -Formatmuster enthalten Leerzeichen.	4
AllowLeadingWhite	Bestimmt, dass vorangestellte Leerzeichen während der Analyse ignoriert werden, es sei denn, die <code>DateTimeFormatInfo</code> -Formatmuster enthalten Leerzeichen.	1
AllowTrailingWhite	Bestimmt, dass nachgestellte Leerzeichen während der Analyse ignoriert werden, es sei denn, die <code>DateTimeFormatInfo</code> -Formatmuster enthalten Leerzeichen.	2 ►

¹⁵ Zum besseren Mitreden: Die korrekte Aussprache lautet »Grännitsch Miehn Teim«. Und auch wenn es sich so ausgesprochen wie ein Irischer Dialekt anhört – auch Engländer aus Oxford sprechen es so aus.

Member-Name	Beschreibung	Wert
AllowWhiteSpaces	Bestimmt, dass zusätzliche Leerzeichen, die sich an einer beliebigen Stelle in der Zeichenfolge befinden, während der Analyse ignoriert werden müssen, es sei denn, die <i>DateTimeFormatInfo</i> -Formatmuster enthalten Leerzeichen. Dieser Wert stellt eine Kombination aus dem <i>AllowLeadingWhite</i> -Wert, dem <i>AllowTrailingWhite</i> -Wert und dem <i>AllowInnnerWhite</i> -Wert dar.	7
NoCurrentDateDefault	Datum und Zeit sind untrennbar im <i>Date</i> -Datentyp vereint. Auch wenn Sie nur eine Zeit zuweisen, spiegelt ein <i>Date</i> -Wert immer auch ein gültiges Datum wider. Diese Einstellung bestimmt, dass die <i>DateTime.Parse</i> -Methode und die <i>DateTime.ParseExact</i> -Methode das Datum nach dem gregorianische Kalender mit Jahr = 1, Monat = 1 und Tag = 1 zugrunde legen, wenn die analysierte Zeichenfolge nur die Uhrzeit und nicht das Datum enthält. Wenn dieser Wert nicht verwendet wird, wird vom gerade aktuellen Datum ausgegangen.	8
None	Bestimmt, dass die Standardformatierungsoptionen verwendet werden müssen. Dies ist das Standardformat für <i>DateTime.Parse</i> und <i>DateTime.ParseExact</i> .	0

Tabelle 7.5 Diese erweiterten Einstellungen können Sie mit ParseExact anwenden

Die folgenden Codezeilen zeigen, wie Sie ParseExact für die Umwandlung von Zeichenketten in Datumswerte mit geregelten Vorgaben für zugelassene Datums-Formate einsetzen können.

```
Imports System.Globalization

Module Module1

    Sub Main()

        Dim locToParseExact As Date
        Dim locZeitenMuster As String() = {"H,m", "H.m", "ddMMyy", "MM\dd\yy"}

        'Funktioniert, ist im Uhrzeitenmuster.
        locToParseExact = Date.ParseExact("12,00", _
                                         locZeitenMuster, _
                                         CultureInfo.CurrentCulture, _
                                         DateTimeStyles.AllowWhiteSpaces)

        'Funktioniert, ist im Uhrzeitenmuster, und "Whitespaces" sind erlaubt.
        locToParseExact = Date.ParseExact(" 12 , 00 ", _
                                         locZeitenMuster, _
                                         CultureInfo.CurrentCulture, _
                                         DateTimeStyles.AllowWhiteSpaces)

        'Funktioniert nicht, ist zwar im Uhrzeitenmuster, es sind aber keine "Whitespaces" erlaubt.
        'locToParseExact = Date.ParseExact(" 12 , 00 ", _
        '                                 locZeitenMuster, _
        '                                 CultureInfo.CurrentCulture, _
        '                                 DateTimeStyles.None)

        'Funktioniert, ist im Uhrzeitenmuster.
        locToParseExact = Date.ParseExact("1,2", _
                                         locZeitenMuster, _
                                         CultureInfo.CurrentCulture, _
                                         DateTimeStyles.None)

        'Funktioniert, ist im Uhrzeitenmuster.
    End Sub
End Module
```

```
'Das Datum entspricht aber dem 1.1.0001 und wird
'als Tooltip deswegen nicht mit angezeigt, im Gegensatz
'zu allen anderen hier gezeigten Beispielen.
locToParseExact = Date.ParseExact("12.2", _
    locZeitenMuster, _
    CultureInfo.CurrentCulture, _
    DateTimeStyles.NoCurrentDateDefault)

'Funktioniert, ist nicht im Uhrzeitenmuster, da hier Sekunden mit im Spiel sind
'locToParseExact = Date.ParseExact("12,2,00", _
'    locZeitenMuster, _
'    CultureInfo.CurrentCulture, _
'    DateTimeStyles.NoCurrentDateDefault)

'Funktioniert nicht, ist mit Doppelpunkt nicht im Uhrzeitenmuster.
'locToParseExact = Date.ParseExact("1:20", _
'    locZeitenMuster, _
'    CultureInfo.CurrentCulture, _
'    DateTimeStyles.None)

'Funktioniert jetzt, da im Zeitenmuster als Datum hinterlegt.
'(drittes Element im String-Array)
locToParseExact = Date.ParseExact("241205", _
    locZeitenMuster, _
    CultureInfo.CurrentCulture, _
    DateTimeStyles.AllowWhiteSpaces)

'Funktioniert mit Übernahme im englisch-amerikanischen Format,
'da durch die Schrägstriche und die vorgegebene Reihenfolge der Gruppen definiert.
'(viertes Element im String-Array).
locToParseExact = Date.ParseExact("12/24/05", _
    locZeitenMuster, _
    CultureInfo.CurrentCulture, _
    DateTimeStyles.AllowWhiteSpaces)

End Sub

End Module
```

Beachten Sie bei der Definition von Schrägstrichen als Gruppentrennzeichen, dass Sie ihnen jeweils ein Backslash voransetzen, damit der einfache Schrägstrich nicht als Steuerzeichen behandelt wird.

Eine Liste mit den entsprechenden Steuerzeichen und Datenformatierungen finden Sie in Kapitel 26. Dieses Kapitel gibt Ihnen ebenfalls einen tieferen Einblick in den Umgang mit Format Providern.

.NET-Äquivalente primitiver Datentypen

Trotz ihrer festen Verankerung in der Sprache gibt es für jeden der primitiven Datentypen ein .NET-Äquivalent. Die folgende Tabelle gibt Ihnen Auskunft darüber:

Primitiver Datentyp in VB	.NET-Datentyp Äquivalent
Byte	System.Byte
SByte	System.SByte
Short	System.Int16
UShort	System.UInt16
Integer	System.Int32
UInteger	System.UInt32
Long	System.Int64
ULong	System.UInt64
Single	System.Single
Double	System.Double
Decimal	System.Decimal
Boolean	System.Boolean
Date	System.DateTime
Char	System.Char
String	System.String

Tabelle 7.6 Primitive Visual Basic-Datentypen und ihre .NET-Äquivalente

Das bedeutet: Es ist vollkommen egal, ob Sie beispielsweise einen 32-Bit-Integer mit

```
Dim loc32BitInteger as Integer
```

oder mit

```
Dim loc32BitInteger as System.Int32
```

deklarieren. Die Objektvariable loc32BitInteger ist in beiden Fällen nicht nur vom gleichen, sondern vom selben¹⁶ Typ. Wenn Sie sich den erzeugten IML-Code aus dem kleinen Programm

```
Public Shared Sub main()
    Dim locDate As Date = #12/14/2003#
    Dim locDate2 As DateTime = #12/14/2003 12:13:22 PM#
    If locDate > locDate2 Then
        Console.WriteLine("locDate ist größer als locDate2")
    Else
        Console.WriteLine("locDate2 ist größer als locDate")
    End If
End Sub
```

¹⁶ Und zwar in der ursprünglichen Bedeutung des Unterschiedes von »selber« und »gleicher« – Sie und ich können zu einer bestimmten Zeit nie dasselbe, nur das gleiche Auto fahren.

anschauen, werden Sie anhand der generierten Codezeilen

```
.method public static void main() cil managed
{
    // Code size      75 (0x4b)
    .maxstack 2
    .locals init ([0] valuetype [mscorlib]System.DateTime locDate,
                 [1] valuetype [mscorlib]System.DateTime locDate2,
                 [2] bool VB$CG$t_bool$S0)
    IL_0000: nop
    IL_0001: ldc.i8    0x8c58fec59f98000
    IL_000a: newobj     instance void [mscorlib]System.DateTime::ctor(int64)
    IL_000f: nop
    IL_0010: stloc.0
    IL_0011: ldc.i8    0x8c59052cd35dd00
    IL_001a: newobj     instance void [mscorlib]System.DateTime::ctor(int64)
    IL_001f: nop
    IL_0020: stloc.1
    IL_0021: ldloc.0
    IL_0022: ldloc.1
    .
    .
    .
```

feststellen, dass diese Tatsache zutrifft. Beide lokalen Variablen sind, wie in den fett markierten Zeilen zu sehen, als System.DateTime-Typ deklariert worden.

Kapitel 8

Tipps & Tricks für das angenehme Entwickeln zuhause und unterwegs

In diesem Kapitel:

Der Einsatz mehrerer Monitore	268
Sichern, Wiederherstellen oder Zurücksetzen aller Visual Studio-Einstellungen	270
Zurücksetzen der IDE-Einstellungen in den Ausgangszustand	274
Wieviel Arbeitsspeicher darf's denn sein?	276
Testen Ihrer Software unterwegs und zuhause mit Virtualisierungssystemen	277
Hilfe zur Selbsthilfe	283
Erweitern Sie die Codeausschnittsbibliothek um eigene Codeausschnitte	288

So Sie die letzten Kapitel durchgearbeitet haben, wissen Sie, wie sehr Ihnen Visual Studio 2008 die tägliche Entwicklungsarbeit erleichtern kann. Doch es kann nicht alles. Sie können mit ein wenig finanziellem Aufwand die Ergonomie Ihres Arbeitsplatzes enorm verbessern und Ihr eigenes Wohlbefinden und damit nicht zuletzt auch Ihren täglichen Output steigern. Auch einiges zusätzlich Wissenswertes zur IDE, das in Kapitel 3 noch nicht berücksichtigt wurde, trägt dazu bei.

Der Einsatz mehrerer Monitore

Die wohl beste Errungenschaft von Windows 98 – die Älteren unter uns werden sich an dieses Betriebssystem noch erinnern – war seine Möglichkeit, den Windows-Desktop auf mehrere Monitore zu erweitern – entsprechende Hardware vorausgesetzt. Für mich war das »damals« zu meinen Visual Basic 6.0-Zeiten der einzige Grund, nicht auf Windows NT4 (damals mit dem neu erschienenen Internet-Explorer 4.0 und der entsprechenden Desktoperweiterung, die das generelle Feeling von Windows 98 brachte) zu wechseln – denn NT4 sah dieses Feature wegen des anderen Treibermodells zu diesem Zeitpunkt noch nicht vor.

Unter Visual Basic 6.0 war das auch nicht unbedingt notwendig. Sah man sich schon damals in der finanziellen Lage, einen Monitor mit 1280×1024 Punkten Auflösungsfähigkeit zu erwerben, und hatte man auch die entsprechende Grafikkarte zur Verfügung, deren RAMDAC diese Auflösung mit mehr als flimmernden 60 Hertz darstellen konnte, gab's unter VB6 keine Probleme.

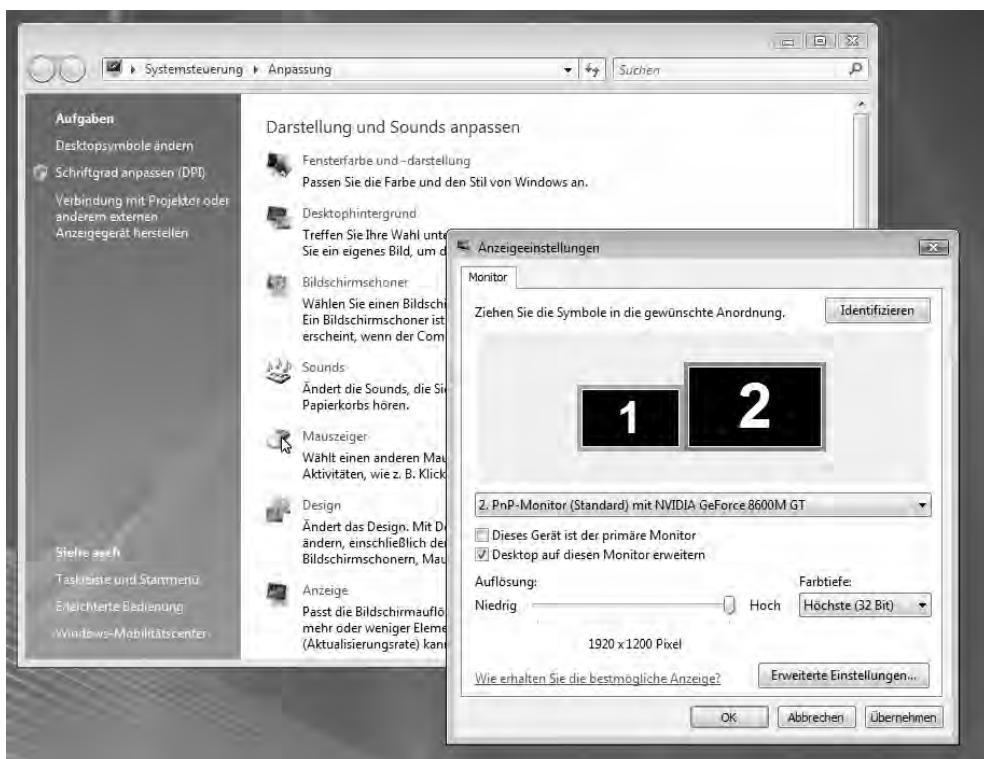


Abbildung 8.1 Mit dem Dialog Eigenschaften von Anzeige erweitern Sie auf der entsprechenden Registerkarte den Windows-Desktop auf mehrere Monitore

Heute ist das anders. 1024×768 für ein wirklich produktives Arbeiten mit Visual Studio 2008 ohnehin zu wenig – ich denke, darüber brauchen wir uns gar nicht zu unterhalten. 1280×1024 ist in meinen Augen buchstäblich die Untergrenze, 1600×1200 macht erst bei deutlich mehr als 20 Zoll TFT Sinn, will man nicht nur 10 Zentimeter entfernt vor dem Monitor kleben.

Aber es geht auch besser, flexibler und billiger – mit mehreren Monitoren an einem Rechner. Wenn Ihr Arbeitsplatzrechner nicht deutlich älter als 6 Jahre ist, können Sie durchaus das Glück haben, dass Ihre Grafikkarte bereits über zwei Anschlüsse für Monitore verfügt, und falls Ihr Rechner oder Ihr Notebook¹ nicht älter als 3 Jahre sind (Stand 2008), werden sie zu 99% aller Fälle Multi-Monitor-fähig sein. Und dann beschaffen Sie sich für wenig Geld einfach einen zweiten, z.B. 17"-Monitor, der nur für die Aufnahme der Toolfenster dient. Auf dem linken Hauptmonitor platzieren Sie nur Toolbox und die Dokumentenregisterkartengruppe, und Sie ersparen sich damit ein lästiges ständiges Umpositionieren bzw. Auf- und Zuklappen der Toolfenster.

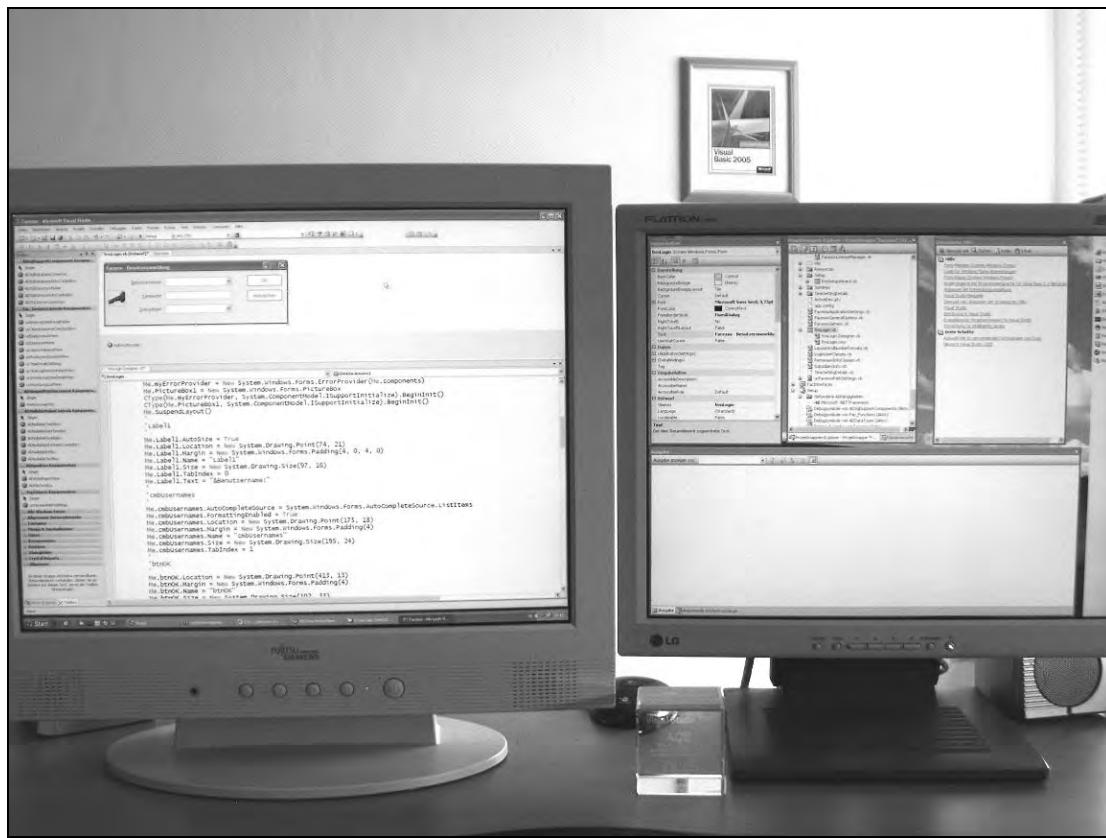


Abbildung 8.2 Die Visual Studio-Oberfläche verteilt auf zwei Monitoren – das lässt richtig Platz für kreative Ergüsse und spart viel Zeit!

¹ Bei Notebooks können Sie ein externes Display in der Regel parallel zum Notebook-Display betreiben.

Und die Bildschirmschirmdarstellung auf Notebooks?

Auch hier gilt: Je mehr Auflösung, desto besser. Allerdings sollte das Display des Notebooks selbst entsprechend groß sein. An ein längeres konzentriertes Arbeiten ist mit einer 1650×1050 -er Auflösung auf einem 15,4"-Display kaum zu denken, zumindest aber grenzwertig – auch wenn Sie auf beiden Augen volle Sehkraft haben. Günstige 17-Zöller befinden sich inzwischen im Handel, und als ideal ausgeglichen zwischen »hoher Auflösung« und »noch zu erkennen« würde ich bei einem 17"-Notebook eine Wide-Screen-Auflösung mit 1650×1050 Pixel empfehlen.

TIPP Falls Sie die Entscheidung treffen müssen, ein neues Notebook auch für Vor-Ort-Entwicklungseinsätze zu erwerben, sollten Sie auf eine schnelle Grafikkarte Wert legen, die vor allen Dingen über einen eigenen Grafikspeicher verfügt. Das gilt insbesondere dann, wenn Sie für Vista WPF-Anwendungen entwickeln. Ein Vista-Grafik-Performanceindex von zwischen 3 und 4 sollte es mindestens sein – sie können das ja leicht bei einem Markt für Multimedia-Geräte im Showroom testen –, und Sie sollten darauf achten, dass die Grafikkarte auch wirklich DirectX-10-fähig ist.

Abgesehen davon, dass »Shared Graphics«-Systeme sich aus dem immer knappen Arbeitsspeicher Ihres Rechners bedienen (aus 1 GByte werden so ruckzuck 768 MByte Hauptspeicher, die Ihnen nur noch zur Verfügung stehen), sind diese auch spürbar langsamer. Und da es sich bei Visual Studio schon um eine recht grafikintensive Anwendung handelt, wäre ein Sparen an der Grafikausstattung ein Sparen an der falschen Stelle.

Sichern, Wiederherstellen oder Zurücksetzen aller Visual Studio-Einstellungen

Nachdem Sie sich zum ersten Mal Ihre persönliche IDE-Umgebung so eingerichtet haben, wie Sie sie wirklich haben wollten, wissen Sie, wie viel Aufwand das ist. Visual Studio bietet Ihnen eine einfache Möglichkeit, alle Einstellungen von Visual Studio, die Sie einmal vorgenommen haben, in einer Datei zu speichern, sodass Sie sie beispielsweise auch auf andere Computer übertragen können. Wie Sie alle Visual Studio-Einstellungen wieder in den Ausgangszustand zurückversetzen, konnten Sie bereits in Kapitel 4 lesen.

Sichern der Visual Studio-Einstellungen

Um die Visual Studio-Einstellungen zu sichern, damit Sie sie auf andere Rechner übertragen oder für die Wiederherstellung desselben Rechners verwenden können, verfahren Sie folgendermaßen:

- Wählen Sie aus dem Menü *Extras* den Menüpunkt *Einstellungen importieren und exportieren*.
- Visual Studio zeigt Ihnen nun einen Assistenten, etwa wie in Abbildung 8.3 zu sehen.

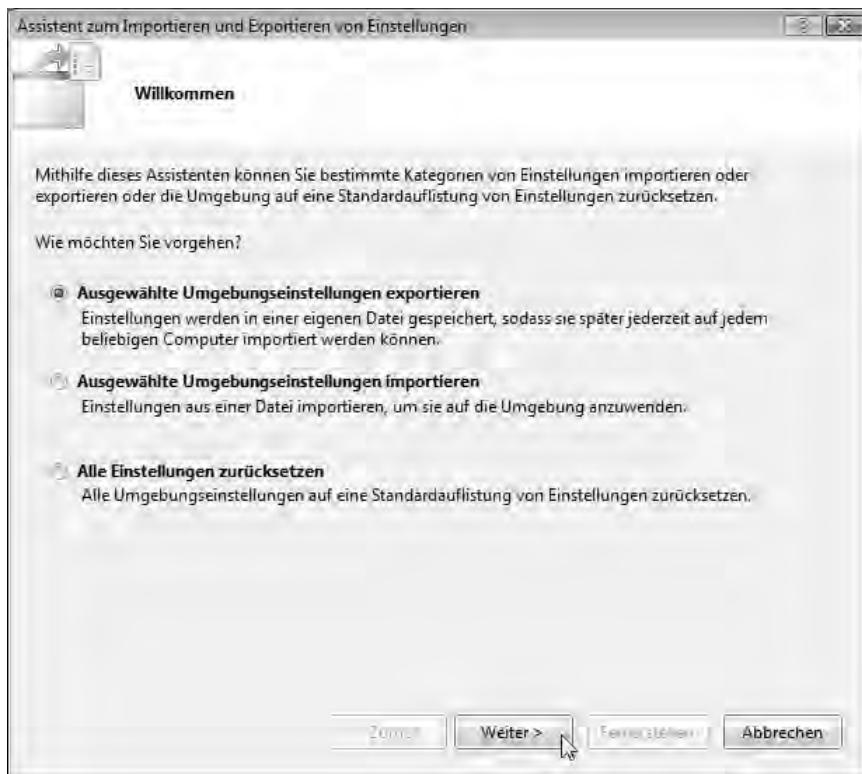


Abbildung 8.3 Mit diesem Assistenten können Sie einige oder alle Einstellungen von Visual Studio in einer Datei speichern, um sie beispielsweise auf andere Rechner zu übertragen oder als Backup zu archivieren

- Wählen Sie den ersten Punkt der Liste *Ausgewählte Umgebungseinstellungen exportieren*, und klicken Sie auf *Weiter*.

Wählen Sie im Dialog, den Sie auch in Abbildung 8.4 erkennen können, die Einstellungen, die Sie exportieren möchten, und klicken Sie anschließend auf *Weiter*.

HINWEIS Die mit dem Ausufezeichen versehenen Einstellungstypen könnten u.U. persönliche Informationen enthalten. Seien Sie sich dessen bewusst, wenn Sie Visual Studio-Einstellungen anderen Entwicklern Ihres Teams auf diese Weise zur Verfügung stellen.

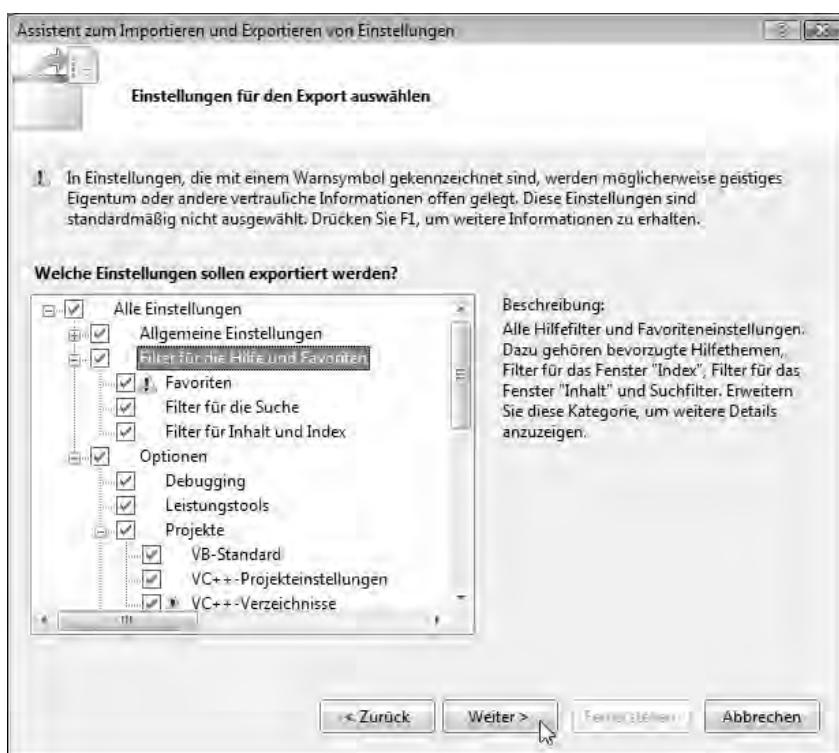


Abbildung 8.4 Wählen Sie, welche Einstellungen Sie exportieren möchten. Die mit dem Ausrufezeichen versehenen könnten übrigens persönliche Informationen enthalten.

- Bestimmen Sie im folgenden Assistentenschritt Dateinamen und Zielordner.
- Klicken Sie auf *Fertigstellen*, um das Speichern der Einstellungen zu starten. Dieser Vorgang kann einige Zeit in Anspruch nehmen.

Wiederherstellen von Visual Studio-Einstellungen

Um wie im vorherigen Abschnitt beschriebene Einstellungen in eine Visual Studio-Installation zu importieren, verfahren Sie folgendermaßen:

- Wählen Sie aus dem Menü *Extras* den Menüpunkt *Einstellungen importieren und exportieren*.
- Visual Studio zeigt Ihnen nun einen Assistenten, etwa wie in Abbildung 8.3 zu sehen, und dort wählen Sie *Ausgewählte Umgebungseinstellungen importieren*.

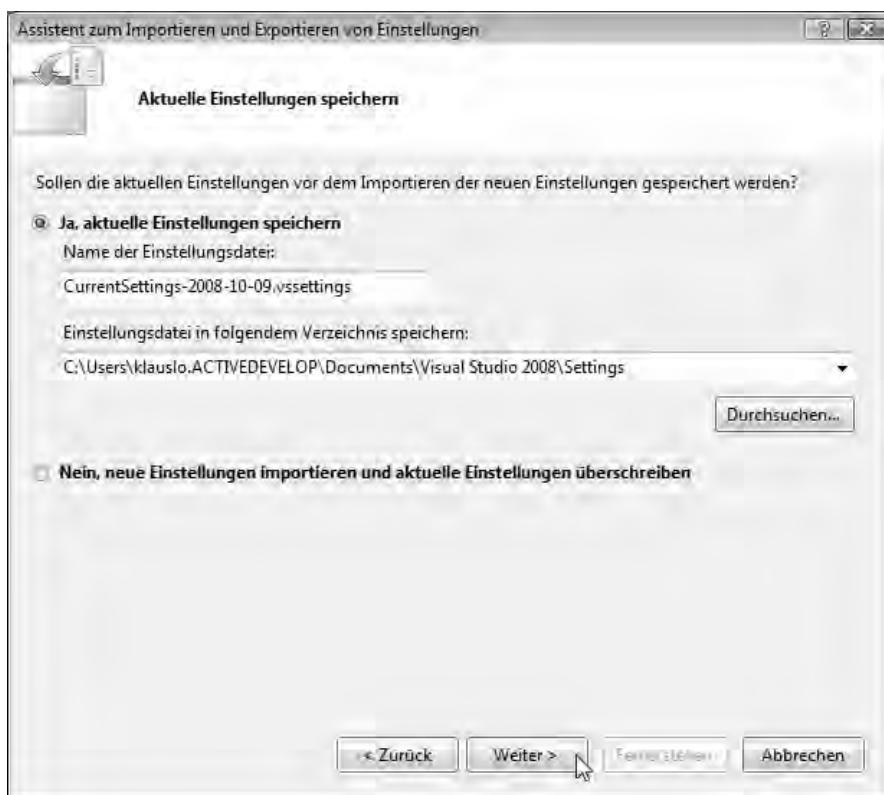


Abbildung 8.5 Sie können vor dem Importieren gespeicherter Einstellungen die aktuellen Einstellungen sichern

- Sie können vor dem Importieren gespeicherter Einstellungen die aktuellen Einstellungen sichern – dazu wählen Sie im Dialog, den Sie auch in Abbildung 8.5 sehen, den ersten Punkt und geben den Dateinamen und den Speicherort ein. Wählen Sie anderenfalls die zweite Option.
- Klicken Sie auf *Weiter*.
- Im nächsten Schritt haben Sie zwei grundsätzliche Möglichkeiten: Sie können aus einem ganzen Pool von Standardeinstellungen für die unterschiedlichsten Zwecke wählen oder eine zuvor ausgewählte Einstellungsdatei auswählen und diese damit für das Importieren festlegen. Um eine zuvor gespeicherte Einstellungsdatei für das Importieren auszuwählen, klicken Sie auf die Schaltfläche *Durchsuchen*, die Sie auch in Abbildung 8.6 sehen können.

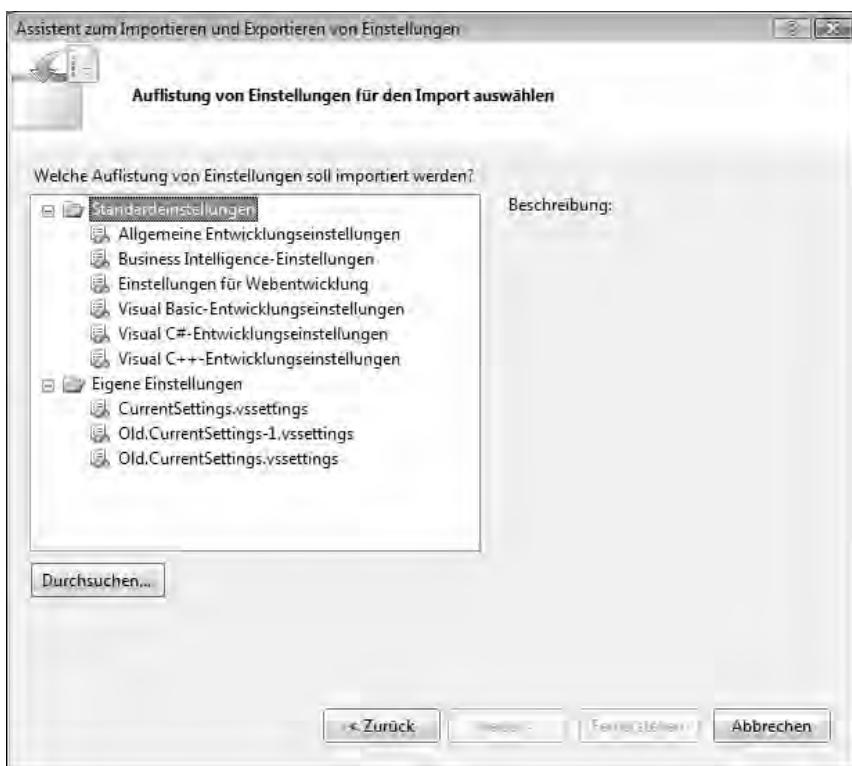


Abbildung 8.6 Entscheiden Sie sich an dieser Stelle entweder für eine Reihe standardmäßig vorhandener Einstellungsvorgaben oder für eine Einstellungsdatei, die zuvor exportierte Einstellungsdaten enthält

- Klicken Sie anschließend auf *Fertigstellen*, um den Importvorgang der Einstellungen zu starten. Dieser Vorgang kann eine Weile dauern.

Zurücksetzen der IDE-Einstellungen in den Ausgangszustand

Wenn Sie die gesamte IDE in ihren Ausgangszustand zurückversetzen möchten, wählen Sie aus dem Menü *Extras* den Menüpunkt *Einstellungen importieren oder exportieren*. Visual Studio zeigt Ihnen anschließend den Dialog, den Sie auch in Abbildung 8.7 sehen können.

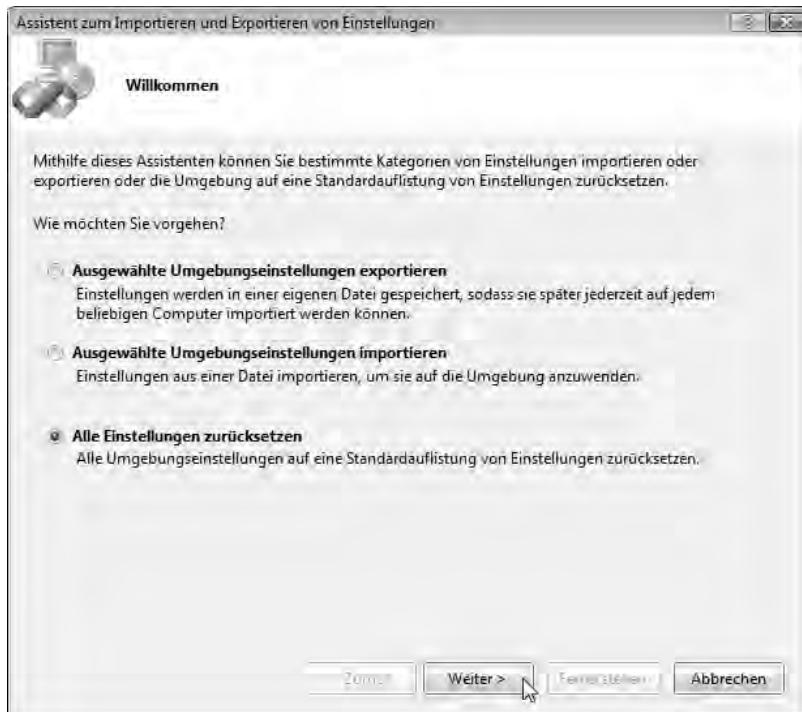


Abbildung 8.7 Um die IDE-Einstellungen in den Ursprungszustand zurückzuversetzen, wählen Sie aus dem Menü Extras den Menüpunkt Importieren und Exportieren von Einstellungen

HINWEIS Beachten Sie, dass der Vorgang, den Sie hier durchführen, die gesamte Entwicklungsumgebung in den Ausgangszustand zurücksetzt. Das bedeutet auch, dass bestimmte Schriftartenzuordnungen oder Tastaturkürzel, die Sie umdefiniert haben, dabei verloren gehen. Falls Sie nur das Fensterlayout von Visual Studio in den Originalzustand zurücksetzen wollen, wählen Sie aus dem Menü Fenster den Menüpunkt *Fensterlayout zurücksetzen*.

Klicken Sie auf *Weiter*, und im Dialog, der jetzt erscheint, wählen Sie *Nein, nur Einstellungen zurücksetzen und die aktuellen Einstellungen überschreiben*. Wählen Sie Ihre bevorzugten Einstellungen und bestätigen Sie den Dialog mit *Fertig stellen*.

Wieviel Arbeitsspeicher darf's denn sein?

Um es ganz klar zu sagen: Die Speichervoraussetzungen, die Microsoft empfiehlt, reichen meiner Meinung nach mit 512 MByte auch auf Windows XP-Systemen bei weitem nicht aus.² Es sind im wahren Wort Sinn tatsächlich Mindestanforderungen.

1 GByte sind, wie ich finde, nötig aber auch ausreichend, wenn Sie ausschließlich mit Visual Studio 2008 auf einem XP-System arbeiten; bei Vista tun Sie sich selber einen Gefallen, und fangen unter 2 GByte Hauptspeicher gar nicht erst an. Falls Sie Datenbankanwendungen entwickeln möchten und dazu auf SQL Express zurückgreifen, das auf dem gleichen Computer installiert sein soll, sollten Sie am besten gleich grundsätzlich 2 GByte Hauptspeicher für ein zügiges Arbeiten in Betracht ziehen.

Ein 32-Bit-Betriebssystem kann keine 4 GByte Arbeitsspeicher nutzen. Wirklich!

RAM ist inzwischen so preiswert, dass auch 4 GByte Hauptspeicher zur Drucklegung des Buchs keine Seltenheit in neueren Computermodellen und auch in Notebooks mehr sind. Allerdings, und ich weiß, dass ich meinem guten Freund und Fachlektor dieses Buchs Ruprecht Dröge da sehr aus der Seele spreche, sind 4 GByte in einem 32-Bit-Betriebssystem, ganz gleich ob es Windows XP, Windows Vista, Windows Server, Linux, MacOS oder sonst wie heißt, einfach nicht zu adressieren.

Warum? Ganz einfach: Wir sind ja Programmierer, und das Kennen des Binärsystems gehört deswegen ja Gott sei dank zu unserem Handwerkszeug. Also: 4 GByte sind rund 4 Milliarden Bytes. Und wenn Sie sich schon ein wenig mit den Datentypen aus dem letzten Kapitel auseinander gesetzt haben, dann werden Sie wissen, dass sich mit 32 Bit (entspricht 4 Bytes) durch den Datentyp UInteger vorzeichenlos maximal der Zahlenbereich 0 bis 4.294.967.295 abdecken lässt. Und wissen Sie was? Das sind dann auch nur 4.294.967.296 Bytes, die sich innerhalb eines Computers mit einem 32-Bit-Betriebssystem adressieren lassen (das 0. Byte müssen Sie nämlich mitzählen). Und 4.294.967.296 Bytes sind (geteilt durch 1024) 4.194.304 Kilobytes und die entsprechen, wer hätte es gedacht (geteilt durch 1024) 4.096 MBytes und – tah dah – das sind eben 4 GByte. Und da ist Schicht. Soweit wäre es also kein Problem, auf einem 32-Bit-System auch 4 GByte Arbeitsspeicher nutzen zu können, doch dummerweise gibt es ja noch andere Komponenten in Ihrem Computer, von denen die Grafikkarte in Sachen Speicher in 32-Bit-Systemen das größte Sorgenkind darstellt. Die bringt nämlich mal ganz locker in der heutigen Zeit noch mal 512 MByte mit; in vielen Fällen sogar schon ein eigenes Gigabyte. Doch auch der Grafikkartenspeicher fällt unter die 32-Bitgrenze, was bedeutet: Die Systemkomponenten werden durch das BIOS Ihres Computers einfach über den Hauptspeicher geblendet. Bei 2 GByte Hauptspeicher lässt das eine Lücke von 1 GByte; bei 4 GByte überlappen sich die Speicherbereiche allerdings, und die Systemkomponenten gewinnen dabei immer! In vielen Fällen sehen Sie deshalb in einem 32-Bit-Betriebssystem eben nur 3,25 GByte; je nach verwendetem BIOS-Hersteller manchmal auch nur 3 GByte. Und Sie können es drehen und wenden, wie Sie es wollen, daran ändern Sie nichts – solange wie Sie ein 32-Bit-Betriebssystem verwenden. ▶

² Im Übrigen finde ich, dass selbst 512 MByte auf Windows-XP-Systemen oder 1 GByte auf Windows Vista nur dann ausreichen, wenn Sie nichts anderes außer dem Betriebssystem installieren. Aber dann müssten Sie Ihre Berichte mit WordPad und Ihre Kalkulationen mit dem Taschenrechner durchführen. Tipp dazu: Stellen Sie ihn auf wissenschaftlich, dann berücksichtigt er immerhin die Hierarchieregeln... ;-)

Und das Fazit: Nur die Installation von Windows XP bzw. Windows Vista in der 64-Bit-Version schafft hier Abhilfe. Mit 32 Bit können Sie nun einmal maximal 4 GByte adressieren, und die Systemkomponenten belegen immer mindestens den Speicherbereich von 768 MByte, egal, ob sie ihn wirklich benötigen oder nicht.

Und, wenn ich noch einmal meinem Fachlektor aus der Seele sprechen darf:³ Dies gilt auch für Serveranwendungen wie z.B. SQL Server auf einem 32-Bit Windows Server – ganz gleich, was Sie machen; es sei denn, Sie bedienen sich einer Technik namens AWE⁴ (ein kleiner böser Trick), eine Technik, die der SQL Server zu nutzen weiß, und die per Mapping unterschiedliche Speicherbänke in bestimmte Speicherbereiche ein- bzw. umblenden kann – aber das sei hier nur am Rande erwähnt. Falls Sie das Thema interessiert, recherchieren Sie doch einmal im Internet die Stichpunkte „/3GB Schalter der BOOT.INI“ und für den SQL Server „AWE“. Aber denken Sie daran: AWE ist eine Technik, die per Mapping zusätzlichen Speicher im Bedarfsfall einblendet; mehrere Speicherbereiche überlappen sich dabei, und um an den richtigen Speicher zu kommen, muss jedes Mal Aufwand (nämlich das Bank-Swapping) betrieben werden, und das kostet wertvolle Zeit!

Testen Ihrer Software unterwegs und zuhause mit Virtualisierungssystemen

Dass Ihre entwickelte Software in Ihrer Entwicklungsumgebung läuft, bedeutet nicht notwendigerweise, dass sie das auch auf einem frisch installierten PC beim Kunden macht. Das Testen Ihrer Software gehört deswegen neben der Entwicklung zu Ihren wichtigsten Aufgaben – sollte es zumindest.

Doch dieses Testen ist unter Umständen nicht minder aufwändig als die Softwareentwicklung selbst, denn:

- Läuft Ihre Software auch auf »frischen« Rechnern ohne Visual Studio-Entwicklungsumgebung, die schließlich dafür sorgt, dass ohnehin alle von Ihrer Software benötigten Komponenten vorhanden sind?
- Wenn Sie Ihre Software unter Vista entwickelt haben, läuft sie auch unter Windows XP, oder andersherum: Haben Sie an die Benutzerkontensteuerung von Vista gedacht? Wie schaut sie dort aus? Wie sieht es für .NET Framework 2.0-basierende Anwendungen aus mit alten Windows 2000SP4 oder Windows Server-Betriebssystemen (2003/2008)?
- Als Entwickler haben Sie sicherlich andere Rechte in einer Domäne als ein »einfacher« Benutzer? Vielleicht entwickeln Sie gar nicht in einer Active-Directory-Umgebung, doch Ihre Software soll vielleicht genau dort eingesetzt werden!

³ Der diesen Absatz ohnehin selbst geschrieben hat – ich bin mir also sicher, dass ich ihm aus der Seele spreche! ;-)

⁴ Adress Windowing Extension – Mehr dazu gibt's über den IntelliLink **A0801**.

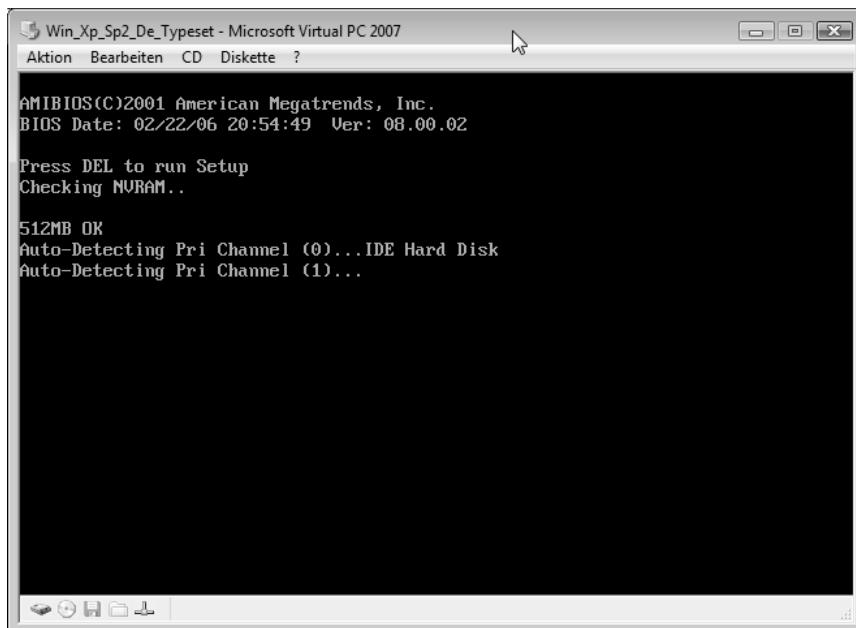


Abbildung 8.8 Ein virtueller PC beginnt mit seiner Arbeit ...

Sie ahnen es: All diese Konstellationen nachzustellen, würde einen ganzen Rechnerpark erfordern. Doch das ist nicht notwendig, denn exakt für diese Testkonfigurationen gibt es Software, die einen kompletten Computer mit Bios, virtuellen Festplatten, die in Dateien gemapt werden, emulierten Grafik- und Soundkarten und umgeleiteter Tastatur- und Maussteuerung mit nahezu 100%iger Kompatibilität in ein Windows-Fenster packen – entsprechend viel Rechenpower und ausreichend Speicher vorausgesetzt. Microsoft selbst stellt dazu zwei Tools zur Verfügung – Microsoft Virtual PC und Microsoft Virtual Server.

Microsoft Virtual PC

Virtual PC installieren Sie am besten genau aus den zuvor genannten Gründen, nämlich eben um Ihre Testaufgaben als Entwickler am einfachsten und elegantesten erledigen zu können.

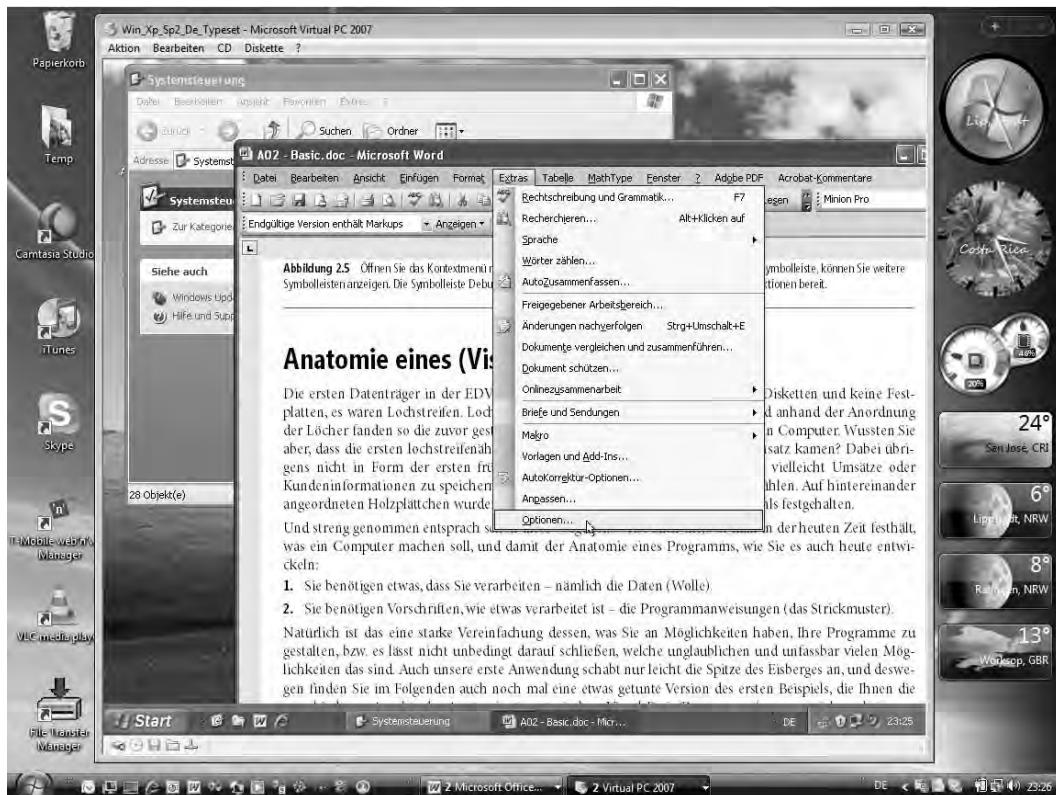


Abbildung 8.9 ... um Sekunden später ein voll funktionsfähiges und durchaus schnell laufendes Windows XP zu präsentieren, mit Windows Vista als Gastgeber, und mit allen Rechten und Pflichten

Sie können mehrere virtuelle Maschinen nebeneinander installieren und diese untereinander sogar virtuell vernetzen. Sie können eine virtuelle Maschine genau wie einen richtigen PC an einer Domäne anmelden – der Domänencontroller wird den Unterschied nicht einmal bemerken. Abbildung 8.8 und Abbildung 8.9 demonstrieren dieses Konzept der virtuellen Computer, wie ich finde, auf eindrucksvolle Weise. Dabei gehen die Möglichkeiten in einem Virtual PC teilweise über die des Gastsystems hinaus. Besonders einfach lässt sich beispielsweise ein Rückgängig-Datenträger konfigurieren. Alle Änderungen während einer Sitzung werden dann zunächst nur auf diesem gespeichert. Beim Ausschalten bzw. Herunterfahren des Virtual PCs werden Sie dann gefragt, ob Sie die gemachten Änderungen endgültig übernehmen wollen: ideal für Tests oder Trainings, wo es verlässlich einen immer gleichen Ausgangszustand herzustellen gilt. Wie oft habe ich mir das für Windows selbst gewünscht!

HINWEIS Bedenken Sie aber auch, dass Sie für jede virtuelle Maschine genauso die entsprechenden Betriebssystemlizenzen benötigen, als handelte es sich bei diesen um »echte« PCs – wobei dort auch wieder gilt: Keine Regel ohne Ausnahme. Sie dürfen Windows Vista, wie alle anderen vorherigen Betriebssysteme auch, nur mit *einer* Lizenz wahlweise als Wirtssystem *oder* als Gast in einer virtuellen Maschine installieren. Für jede weitere Installation (egal wo) ist eine weitere Lizenz fällig. Ausnahmen bilden Vista Enterprise und Vista Ultimate, die man jeweils als Host und zusätzlich als Gastbetriebssystem installieren darf. Ultimate nur ein einziges Mal, Enterprise aber bis zu viermal. Das immer wieder kolportierte Verbot einer Installation der Home-Edition von Vista in einer VM gilt in Deutschland nicht.

Microsoft greift Ihnen aber gerade als Softwareentwickler mit dem MSDN-Abo⁵ unter die Arme. Oder als einfacher Microsoft-Partner mit dem sehr zu empfehlenden Action Pack. Das Gleiche gilt auch für den Arbeitsspeicher, der vom Gastgeberrechner abgezwackt wird. Möchten Sie einen virtuellen PC mit Windows XP und 512 MByte Arbeitsspeicher installieren, sollte Ihr System, das den VPC hostet, über 512 MByte zzgl. der Menge an Arbeitsspeicher verfügen, um selbst zügig zu laufen. Bei mehreren VPCs, die gleichzeitig laufen sollen, weil Sie Ihre Software vielleicht in einem simulierten Active-Directory-Netzwerk testen wollen, sind Sie dabei schnell im Gigabyte-Bereich, was den Arbeitsspeicherbedarf anbelangt.

Lassen Sie sich von der Arbeitsspeicheranzeige in der Sidebar von Windows Vista in Abbildung 8.9 nicht täuschen – hier blieben offensichtlich 50% Arbeitsspeicher übrig. 50% von einem Hauptspeicher mit »echten« 4 GByte auf einem 64-Bit-System im übrigen, und das bei nur 512 MByte, die fürs Gastsystem reserviert wurden. Sie sehen: Mit 2 GByte Hauptspeicher kommen Sie schnell an die Grenzen des Erträglichen, denn bei mehr Bedarf würde nämlich gerade das speicherhungrige Vista sehr schnell beginnen, Arbeitsspeicher auf die Festplatte auszulagern, und das kostet Ihre wertvolle Zeit!

Wie in Abbildung 8.9 oben links zu sehen, steuern Sie alle VPC-Installationen mit einem zentralen Kommandocenter, über das Sie auch deren Einstellungen wie Speicherzuordnungen, virtuelle Festplatten oder Weiteres vornehmen. Virtual PC von Microsoft ist aber in der Version 2007 SP1 im Übrigen zwischenzeitlich kostenlos zu erhalten, und Sie finden es unter dem IntelliLink **A0802**. Ein etwas anderes Konzept verfolgt Virtual Server von Microsoft.

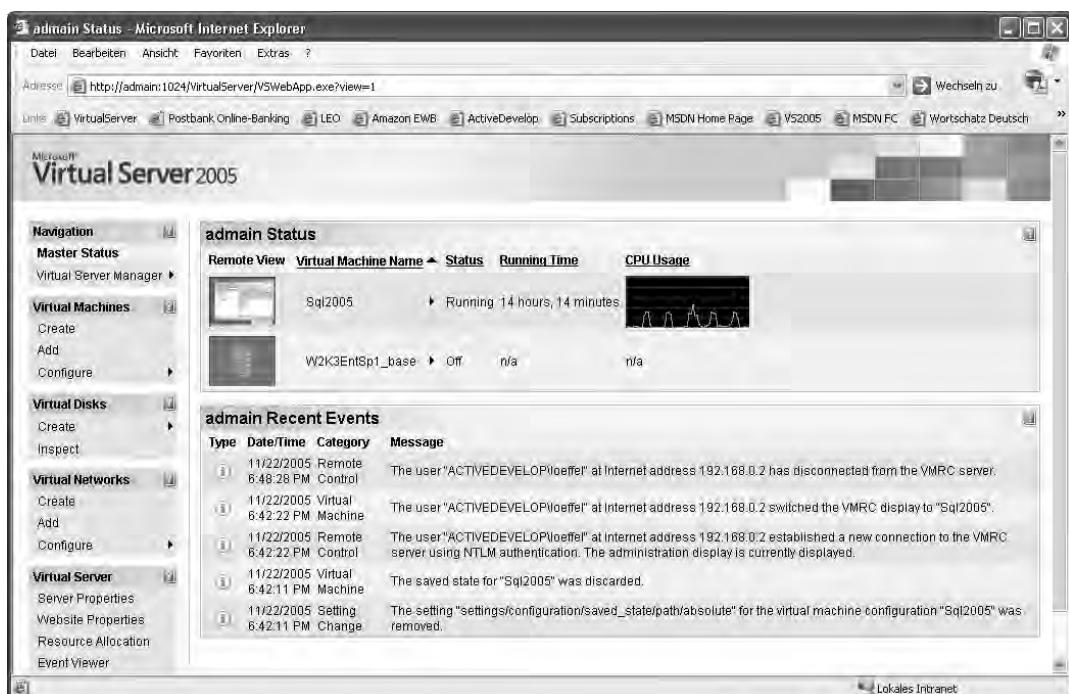


Abbildung 8.10 Alle auf einem Server installierten virtuellen Server werden von jedem beliebigen Client des Netzwerkes aus auf denkbar einfachste Weise administriert: über den Internet-Explorer ...

⁵ Hinweise zu den verschiedenen MSDN-Abos gibt es unter dem IntelliLink **A0501**.

Virtual Server

Wie Virtual PC dient auch Virtual Server in erster Linie dazu, ganze Computersystemplattformen auf einem entsprechend ausgestatteten Windows Server (oder auch Windows XP) zu simulieren. Doch das dient vor allem dazu, mehrere physisch vorhandene Server in einem Server in Form von virtuellen Servern zu konsolidieren. Im Gegensatz zu Virtual PC ist Virtual Server jedoch als Dienst und nicht als Anwendung implementiert. Bedenken Sie, dass ein Anwender an einer Maschine angemeldet sein muss, um eine Anwendung zu starten, ein Dienst jedoch vom Betriebssystem gestartet werden kann, ohne dass ein User sich anmeldet. Stellen Sie sich also vor, sie möchten Ihrem SQL- oder Exchange Server in einer virtuellen Maschine betreiben: Bei Virtual PC müssten Sie den Gastrechner starten, sich am System anmelden und dann die virtuelle Maschine, in der Ihr Server läuft, starten, und angemeldet bleiben - wohl kaum das richtige Szenario für einen reibungslosen Serverbetrieb. Bei Virtual Server jedoch würden Sie einmal dem System mitteilen, dass mit dem Betriebssystem-Start auch Virtual Server und die darin konfigurierten virtuellen Maschinen gestartet werden. Der SQL- oder Exchange-Server würde dann mit dem Gastserver, der beispielsweise als Druck- oder Dateiserver dienen kann, starten – ohne und unabhängig von der Anmeldung eines Benutzers an dieser Maschine.

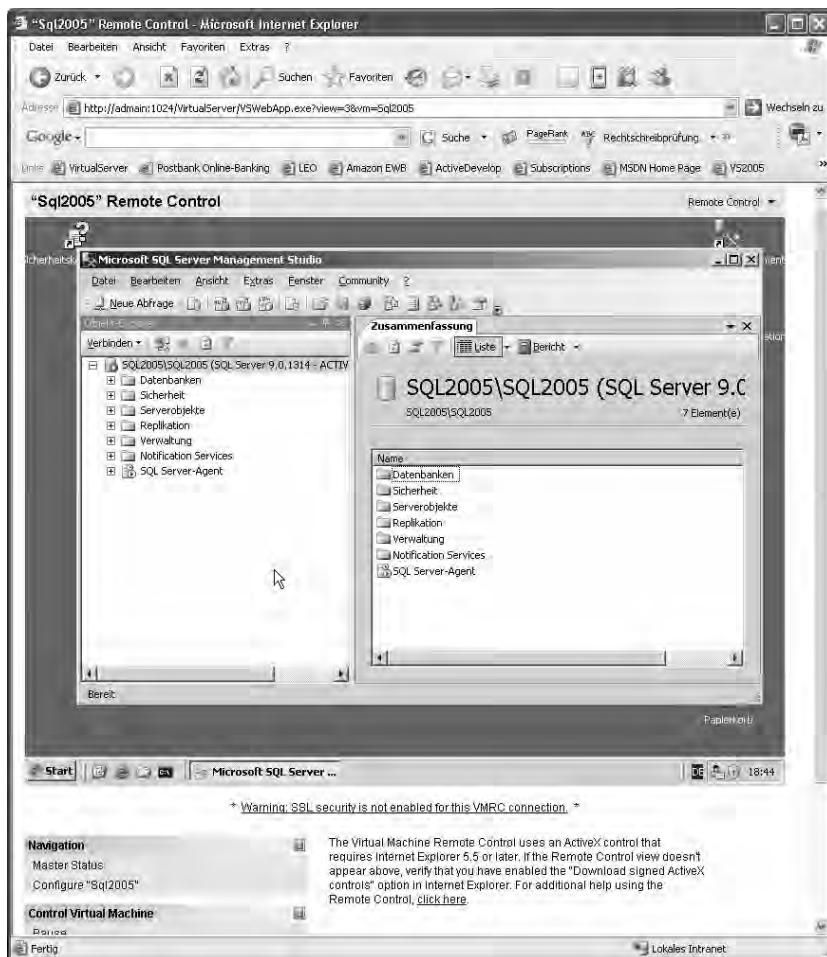


Abbildung 8.11 ... und dieser dient schließlich sogar dazu, einen virtuellen PC zu bedienen

Im Klartext heißt das: Statt einen Server beim Hardwareanbieter Ihres Vertrauens zu bestellen, auf dem beispielsweise ein SQL Server seine Dienste verrichtet und dann einen weiteren zu ordern, der als Internet-Anwendungsserver läuft, und noch einen weiteren, der alle Exchange-Server-Aufgaben übernimmt, setzen Sie nur einen ausreichend groß dimensionierten ein, und lassen ihn als Gastgeber für weitere virtuelle Server werkeln. Diese zusätzlichen Server laufen dann, anders als bei Virtual PC, nahezu unbemerkt als Dienst im Hintergrund – wenn Sie vor dem Desktop des eigentlichen Servers sitzen, werden Sie vergeblich nach Kommandozentralen oder Fenstern mit den virtuellen Servern suchen. Stattdessen – und dieses Konzept finde ich persönlich sehr genial – administrieren Sie alle virtuellen Server über das hauseigene Intranet! Wenn Sie mit einem virtuellen Server arbeiten möchten, klicken Sie ihn auf der durch Virtual Server zur Verfügung gestellten Intranetseite einfach an, – und anschließend haben Sie, wiederum im Internet-Explorer gekapselt, eine ähnliche Oberfläche, wie Sie sie vom Virtual PC kennen. Diese Administrationsoberfläche ist aber als so genanntes ActiveX Control implementiert, daher ist ein Zugriff über Internet meist durch Firewalls oder Sicherheitskonzepte eingeschränkt. Dazu kann der Virtual Server auch eine SCSI Schleife simulieren, wie es etwa für das Testen von Cluster unter Windows hilfreich sein kann, und den einzelnen laufenden Servern individuell RAM und Prozessorleistung zuteilen.

Hyper-V in Windows Server 2008

Und schließlich gibt es Hyper-V als festen Bestandteil von Windows Server 2008, vereinfacht ausgedrückt, der Virtual PC als fester Bestandteil des Betriebssystems. Doch dies sei an dieser Stelle nur der Vollständigkeit halber erwähnt, denn hier geht es wirklich in erster Linie darum, mehrere notwendige Serversysteme auf einer sehr leistungsstarken Maschine zu konsolidieren.

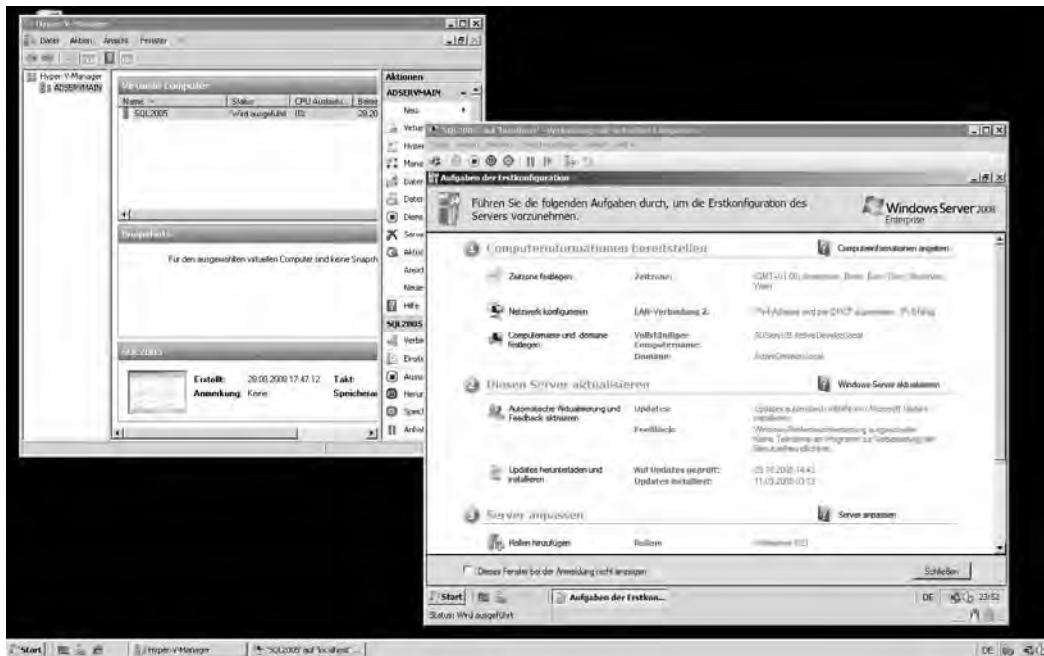


Abbildung 8.12 Hyper-V läuft auf Windows 2008 Servern und dient in erster Linie zur Konsolidierung von weiteren Servern auf einer sehr leistungsfähigen Maschine

Hilfe zur Selbsthilfe

Wenn Sie sich schnell in eine neue Thematik der Frameworks einarbeiten müssen – und das gilt umso mehr, wenn man Einsätze beim Kunden fährt –, lautet die oberste Prämisse: Intelligenz ist, zu wissen wo es steht!

Bei rund über 8.000 verschiedenen Klassen alleine im .NET Framework 2.0 können Sie unmöglich, auch mit jahrelanger Erfahrung, die genaue Funktionsweise jeder dieser Klassen gelernt haben und direkt aus dem Stehgriep anwenden. Was Sie allerdings lernen und perfektionieren können, ist, so schnell wie möglich an die gewünschten Informationen zu kommen.

Die Kombination der Hilfsmittel *Codeausschnittsbibliothek*, *Vervollständigungsliste von IntelliSense*, *dynamische Hilfe* und *Autokorrektur für automatisches Komplizieren* ist dabei eine Ressource für Recherchen, die Sie auch dann nutzen können, wenn Sie beim Kunden vor Ort am Projekt arbeiten, wo Ihnen nicht unbedingt immer ein Internetzugang zur Verfügung steht.

Bei bestimmten Problemstellungen kann Sie die Codeausschnittsbibliothek, die Sie in Kapitel 5 schon kennen gelernt haben, nicht nur mit Code versorgen; sie kann auch Ausgangspunkt für weitere Recherchen nach den richtigen Klassen und Funktionsweisen sein.

Ein Beispiel: Angenommen, ein Kunde kommt zu Ihnen, und bittet Sie, ähnlich wie in Kapitel 5 zu sehen, im Falle eines Fehlers eine E-Mail an eine bestimmte Adresse zu schicken. Leider kennen Sie sich in diesem Bereich noch gar nicht aus. Bis jetzt. Es gilt nun also, seine Professionalität unter Beweis zu stellen, und sich schnellstmöglich die Informationen zu beschaffen, die man für die Realisierung dieser Problemlösung benötigt.

Hier hilft das Suchen in der Codeausschnittsbibliothek erst einmal, um eine Verbindung zwischen dem Thema (E-Mail-Versand) und den zur Lösung benötigten Objekten herzustellen. Mit ein wenig Glück findet sich in der Codeausschnittsbibliothek etwas Entsprechendes:

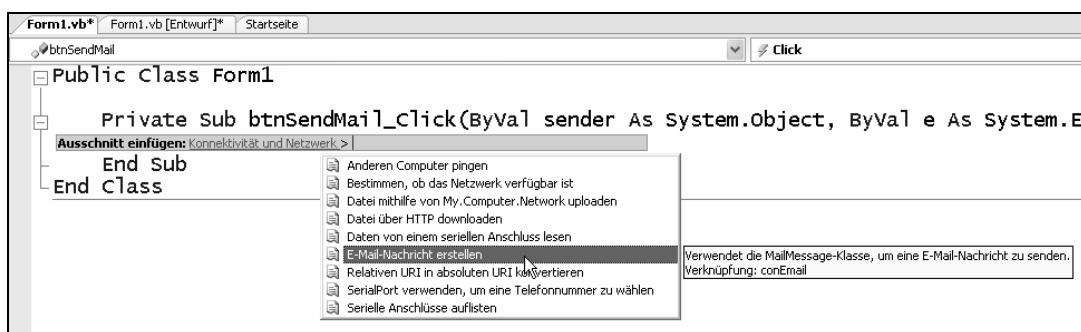


Abbildung 8.13 Über das Editor-Kontextmenü erreichen Sie zunächst die Codeausschnittsbibliothek, in der Sie erst einmal nach dem richtigen Codeschnipsel suchen, um ein Objekt für weitere Recherchen zu erhalten ...

Recht schnell haben Sie auf diese Weise einen ersten kleinen Codeblock erstellt, und die dort vorhandenen Objekte können Sie nun als Ausgangspunkt für weitere Recherchen verwenden.

Der eingefügt Code reicht Ihnen leider momentan noch nicht aus, um das Problem in den Griff zu bekommen, denn der nachfolgende Code

```

Imports System.Net.Mail

Public Class Form1

    Private Sub btnSendMail_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnSendMail.Click

        Dim message As New MailMessage("sender@address", "from@address", "Subject", "Message Text")
        Dim emailClient As New SmtpClient("Email Server Name")
        emailClient.Send(message)

    End Sub
End Class

```

ist zwar in der Lage, eine E-Mail zu versenden, aber leider nicht in der Lage, sich auch beim Mail-Server zuvor anzumelden. Doch alleine das Einfügen des Codeausschnittes hat Ihnen schon einige zusätzliche Hinweise gebracht, wo Sie mit weiteren Recherchen fortfahren können:

- Das Codeausschnittseinfügen hat unter anderem bewirkt, dass in der obersten Zeile eine Imports-Anweisung eingefügt wurde. Weitere Klassen, die sich mit der Thematik befassen, werden also mit großer Wahrscheinlichkeit auch im System.Net.Mail-Namespace zu finden sein.
- Eine Nachricht wird offensichtlich durch die MailMessage-Klasse gekapselt.
- Ein Smtp-Client zum Weiterleiten einer Mail wird offensichtlich durch die SmtpClient-Klasse angesteuert.

Mit diesen Informationen lässt sich doch jetzt schon einiges anfangen. Rekapitulieren wir: Wir können zum jetzigen Zeitpunkt zwar eine Mail schicken, uns aber nicht am Mail-Server anmelden. Also müssen wir herausfinden, welche zusätzlichen Möglichkeiten die SmtpClient-Klasse bietet, und ob es dort nicht irgendwelche »Dinge« gibt, die uns bei der Problemlösung helfen können.



Abbildung 8.14 Die dynamische Hilfe hält kontextabhängige Querverweise in die eigentliche Hilfe parat

An diesem Punkt kommt – es geht ja um die schnelle Suche nach Infos – die dynamische Hilfe ins Spiel. Wenn Sie mit dem Cursor auf die Instanziierungsanweisung für das SmtpClient-Objekt emailClient fahren, dann stellt die dynamische Hilfe im gleichen Moment einige Querverweisvorschläge zur Verfügung, wie in Abbildung 8.14 zu sehen. Und dort sehen wir als Hilfetopic beispielsweise den Dokumentationsverweis zur SmtpClient-Klasse.

Hilfe zur Selbsthilfe



Abbildung 8.15 Ein genaues Lesen der Hilfetexte ist oft hilfreich, um schnell an erforderliche Infos zu kommen

Ein Klick auf den Verweis öffnet anschließend die eigentliche Hilfe. Und hier hilft ein genaues Lesen der Hilfetexte oft schon enorm weiter, zumindest um weitere Hinweise für die Lösung des Problems zu erhalten. In Abbildung 8.15 sieht man es exemplarisch. Es findet sich tatsächlich ein Hinweis auf »Anmeldeinformationen für die Authentifizierung«, und ein weiterer Klick auf den in diesem Zusammenhang stehenden Link bringt uns noch einen Schritt in der Recherche-Kette weiter: Der Link zur Dokumentation der `Credentials`-Eigenschaft.

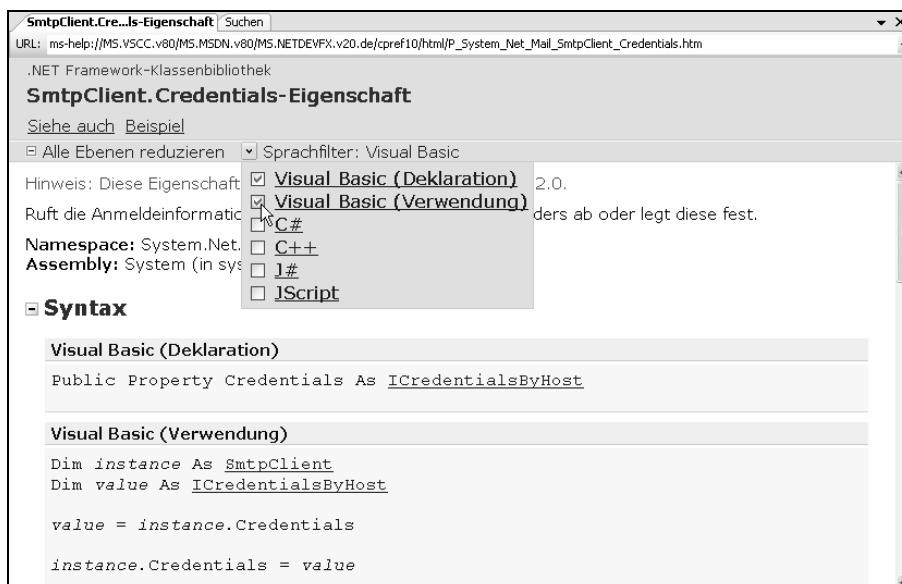


Abbildung 8.16 Sorgen Sie bei der Recherche für den »Blick aufs Wesentliche«

An diesem Punkt angelangt sind zwei Dinge zu bemerken: Wenn Sie zum ersten Mal mit der Hilfe von Visual Studio gearbeitet haben, werden Sie spätestens jetzt feststellen, dass Hilfethemen zu Klassen, Methoden, Eigenschaften oder Ereignissen immer nach dem gleichen Schema aufgebaut sind. So gibt es für Beispiele und Prototypenbeschreibungen die Möglichkeit, wie in Abbildung 8.16 zu sehen, über den Sprachfilter zu steuern, für welche der .NET-Sprachen die Beispiele dargestellt werden sollen. Beschränken Sie an dieser Stelle die Beispiele auf Visual Basic, um sich nicht durch zu viel »anderen« Beispielcode ablenken zu lassen.

Gleichzeitig ist es auch hilfreich, sich mit möglichst allen Techniken im objektorientiert arbeitenden Visual Basic auszukennen. So werden Sie später wissen, nachdem Sie den OOP-Teil dieses Buchs durchgearbeitet haben, dass die so genannten Interfaces (Schnittstellen) Vorschriften für erst eigentlich verwendbare Klassen sind, bestimmte Funktionalitäten einzubinden. Sie werden dann auch wissen, dass Sie Interfaces bereits an ihrer Namenskennung identifizieren können – diese beginnen nämlich grundsätzlich mit einem »I«. Sie werden dann Ihr Wissen in einen Kontext setzen können, und schlussfolgern, dass, wenn die Credentials-Eigenschaft vom Typ `ICredentialsByHost` ist, es irgendwelche konkreten Klassen geben muss, die dieses Interface einbinden. Denn genau diese Klassen können diese Eigenschaft nämlich verarbeiten. Es gilt nun also abzuchecken, welche Klassen diese Schnittstelle einbinden, und ob sie für unsere Problemlösung in Frage kommen.

TIPP Vielleicht sind Sie für den Moment noch durch die vielen, vielleicht für Sie neuen OOP-Elemente wie Schnittstellen und Klassen überfordert. In diesem Fall empfehle ich Ihnen, dieses Kapitel abermals zu lesen, wenn Sie sich mit der OOP-Programmierung im entsprechenden Teil dieses Buchs vertraut gemacht haben.

Leider bietet die Hilfe bis heute keine Möglichkeit, eine Liste aller Klassen abzurufen, die eine bestimmte Schnittstelle einbinden. Aber es gibt ja immer noch die Suchen-Funktion der Hilfe, und die ist in Visual Studio 2008 gar nicht schlecht. Da für jede Klasse, die eine bestimmte Schnittstelle einbindet, diese Schnittstelle auch in der Hilfe beschrieben wird, ist die Wahrscheinlichkeit denkbar groß, auf diese Weise eine Klasse zu finden, mit der die `Credentials` – die Anmeldeinformationen – in der `Credentials`-Eigenschaft verpackt werden können. Und siehe da, die Eingabe von `ICredentialsByHost` als Suchbegriff in der Visual Studio-Hilfe liefert im Handumdrehen die folgenden Suchergebnisse:

Hilfe zur Selbsthilfe

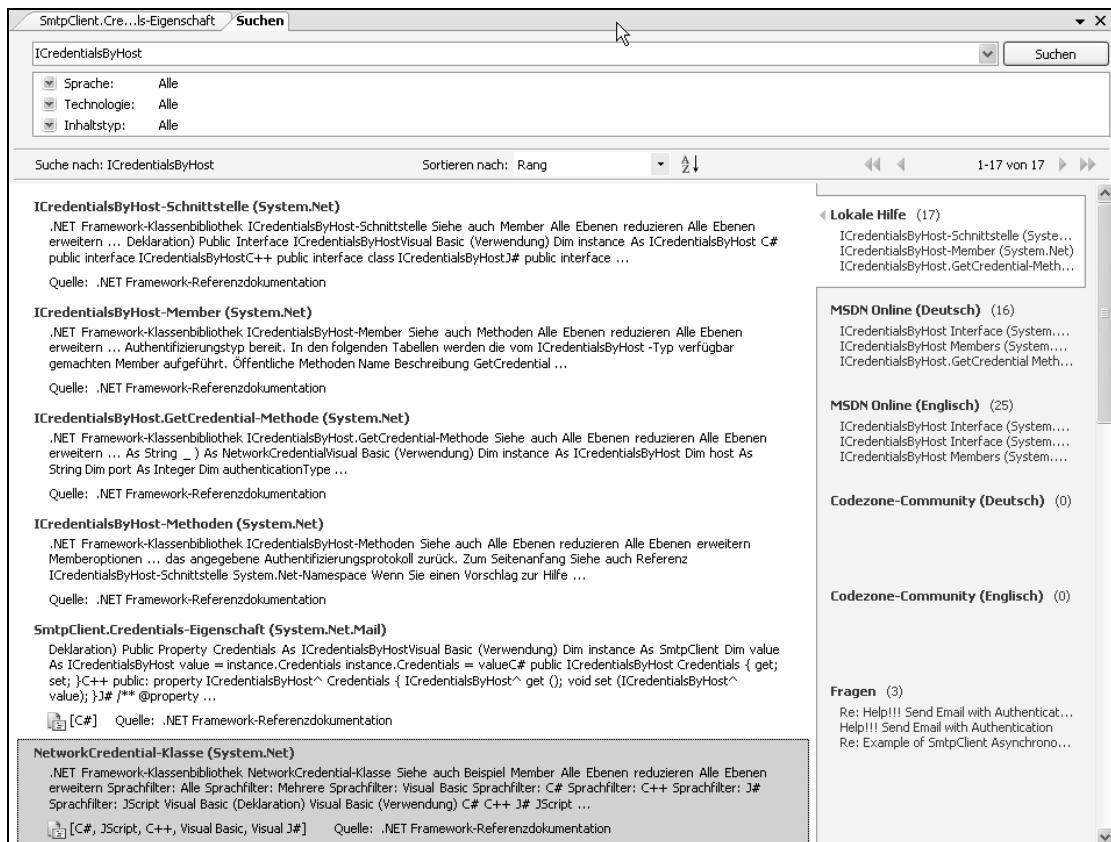


Abbildung 8.17 Erster Versuch, Volltreffer. Die *NetworkCredential*-Klasse sieht doch recht viel versprechend aus.

Ein Klick auf die *NetworkCredential*-Klasse in der Suchergebnisliste offenbart dann auch, wonach wir suchten – nach einer Klasse, die die Anmeldeinformationen aufnimmt und die entsprechenden Schnittstellen einbindet, sodass eine Instanz dieser Klasse an die *SmtpClient*-Instanz – oder besser: deren *Credentials*-Eigenschaft übergeben werden kann.

Mit diesem Wissen kann das Problem nun vergleichsweise schnell gelöst werden und könnte dann beispielsweise folgendermaßen aussehen:

```
Public Class Form1
```

```
Private Sub btnSendMail_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnSendMail.Click
    'Gesucht und gefunden! - Aber Achtung. Die Anmeldeinformationen werden im Klartext übertragen!
    'Dieses Verfahren also nur bei Nicht-AD-Clients im Intranet anwenden!
    Dim myCred As New NetworkCredential("k_loeffel", "password", "ActiveDevelop.local")
```

```
'Das wurde durch die Codeausschnittsbibliothek eingefügt
Dim message As New MailMessage("VomTestprogramm@loeffelmann.de", _
    "klaus@loeffelmann.de", _
    "Testmail",_
    "Das Programm hat eine Testnachricht versendet!")

Dim emailClient As New SmtpClient("192.168.0.1")
'Die Credentials übergeben wir nun!
emailClient.UseDefaultCredentials = False
emailClient.Credentials = myCred
'Und ab dafür!
emailClient.Send(message)
End Sub
End Class
```

Erweitern Sie die Codeausschnittsbibliothek um eigene Codeausschnitte

Es gibt bei der Entwicklung von Projekten immer wiederkehrende Dinge, die oftmals in pure Fließbandarbeit ausarten. Was mich persönlich beispielsweise am meisten nervt, ist das Programmieren von Dateiauswahl-Dialogen. Der Benutzer muss auf eine Schaltfläche klicken, daraufhin öffnet sich ein Datei-Öffnen-Dialog, der durch die OpenFileDialog-Klasse gesteuert wird, der Benutzer wählt seine Datei aus, oder er bricht den Dialog ab.

BEGLEITDATEIEN Ein kleines Projekt, das die generelle Vorgehensweise in VB2008 demonstriert, finden Sie unter:

...\\VB 2008\\A - Einführung\\Kapitel 08\\Codeausschnitte

Öffnen Sie dort die Projektmappe (.SLN-Datei).

Starten Sie dieses Programm, können Sie mit der Auslassungsschaltfläche (...) den Datei-Öffnen-Dialog ins Leben rufen, eine Textdatei auswählen und den Dialog mit OK bestätigen. Der Dateiname steht anschließend im Textfeld neben der Auslassungsschaltfläche:

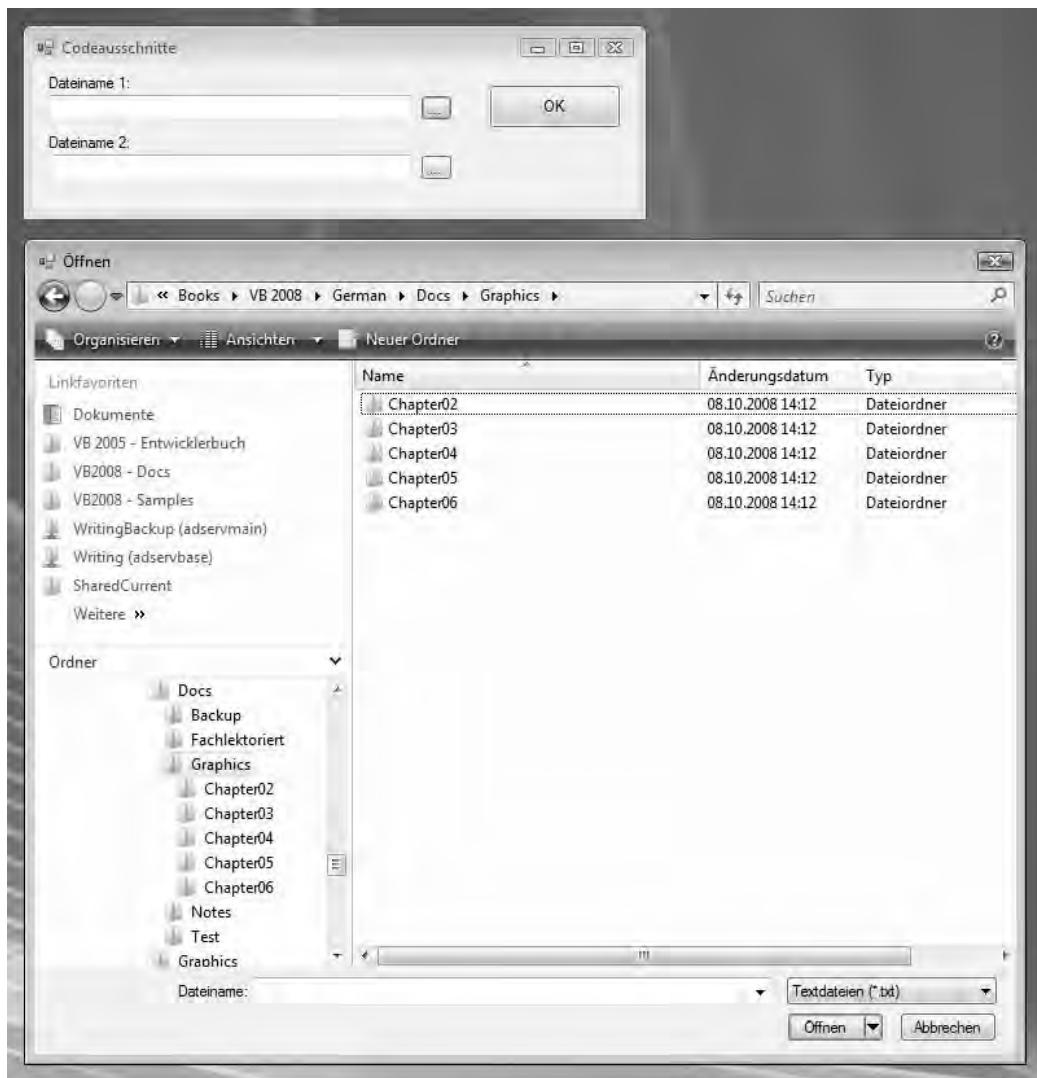


Abbildung 8.18 Ein Vorgang, den Sie in Ihrem Leben sicherlich schon zigmals programmieren mussten: das Zur-Verfügung-Stellen von Dateiauswahl-Diallogen

In Visual Basic 2008 bzw. dem .NET Framework funktioniert das Aufrufen eines solchen Dialogs in einer Windows Forms-Anwendung auf folgende Weise:

```

Public Class Form1

    Private Sub btnDateiAuswählen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnDatei1Auswählen.Click

        'Eine neue OpenFileDialog-Klasse instanziieren
        Dim dateiÖffnenDialog As New OpenFileDialog

        'Alles Weitere bezieht sich nun darauf, bis 'End With'
        With dateiÖffnenDialog
            .CheckFileExists = True ' Datei muss existieren
            .CheckPathExists = True ' der Pfad ebenfalls
            .DefaultExt = "*.txt"   ' Standardendung ist *.TXT

            'Alle angezeigten Dateifilter werden folgendermaßen angegeben
            .Filter = "Textdateien (*.txt)|*.txt|Alle Dateien (*.*)|*.*"

            'Diese Enum-Variable nimmt das Dialogergebnis (OK, Abbrechen) entgegen
            Dim dialogErgebnis As DialogResult = .ShowDialog
            'Falls das Dialogergebnis 'Abbrechen' war,
            If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
                Exit Sub
            End If
            txtDateiname1.Text = .FileName
        End With
    End Sub
End Class

```

Auch wenn Sie einen solchen Dialog in VB2008 noch nicht programmiert haben – ich denke, der Code ist nicht sonderlich schwer zu verstehen. Was man auch gut nachvollziehen kann: Dieser Code kann, wenn man ihn öfters braucht, auch recht lästig zu tippen werden – sowas immer wieder zu implementieren kann dann wirklich zur Fließbandarbeit ausarten.

Das Ziel sollte es deswegen sein, einen entsprechenden Coderumpf in der Codeausschnittsbibliothek zu hinterlegen. In diesem Fall können Sie ihn, wann immer Sie ihn benötigen, dort herausholen, an der entsprechenden Stelle einfügen und sich so einen Haufen Arbeit sparen. Also, auf geht's:

Erstellen einer Code Snippets-XML-Vorlage

Codeausschnitte nennen sich auf Englisch *Code Snippets* (eigentlich im Deutschen mit dem viel bekannteren Begriff »Codeschnipsel« zu übersetzen – aber eine Grundsatzdiskussion über den Gebrauch der Sprache in Microsoft-Technologien möchte ich Ihnen und mir an dieser Stelle lieber ersparen...). Und jedes der schon in Visual Studio vorhandenen Codeschnipsel befindet sich in einer im XML-Format abgelegten Datei, die die Endung *.snippet* trägt. Doch keine Angst: Auch wenn Sie sich noch nicht mit dem Thema XML beschäftigt haben – Sie werden keine Probleme haben, die entsprechenden XML-Rümpfe zu bauen und sie auf Ihre Bedürfnisse anzupassen.

Für die ersten Experimente zur Erstellung der Snippet-XML-Vorlage finden Sie im gleichen Verzeichnis, in dem sich auch das Beispielprojekt befindet, eine Datei namens *Vorlage.snippet*. Um diese XML-Datei zunächst zu öffnen, verfahren Sie wie folgt:

- Wählen Sie aus dem Menü *Datei* den Menüpunkt *Öffnen* und *Datei*.
- Suchen Sie im Datei-Öffnen-Dialog im entsprechenden Projektverzeichnis nach der Datei *Vorlage.snippet*, und klicken Sie auf *OK*, um die Datei in den Editor zu laden.

Die Datei, die Sie anschließend sehen sollten, schaut folgendermaßen aus:

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets
    xmlns="http://schemas.microsoft.com/VisualStudio/2008/CodeSnippet">
<CodeSnippet Format="1.0.0">
    <Header>
        <Title>
            <!-- An dieser Stelle den Namen des Codeausschnitts einfügen: -->
            MeinCodeausschnitt
        </Title>
    </Header>
    <Snippet>
        <References>
            <Reference>

                <!-- An dieser Stelle den Namen einer benötigten
                    Assembly-Referenz einfügen: -->
                <Assembly>System.Windows.Forms.dll</Assembly>

            </Reference>
        </References>

        <!-- Hier folgt die eigentliche Codedefinition: -->
        <Code Language="VB">
            <![CDATA[
                ' Hier steht der einzufügende
                ' Visual Basic 2008 - Programmcode
            ]]>

        </Code>
    </Snippet>
</CodeSnippet>
</CodeSnippets>
```

Sie erkennen im Listing drei in Fettschrift gesetzte Blöcke, die folgende Funktion haben:

- Der erste Block definiert den Namen des Code Snippets. Zwischen den XML-Tags *<Title>* und *</Title>* platzieren Sie den Namen, den das Code Snippet erhalten soll.

- Nun kann es sein, dass Sie im Codeblock, den Sie später in Ihr Projekt einfügen wollen, Referenzen auf erforderliche Assemblies benötigen. Da sich die `OpenFileDialog`-Klasse in der `Assembly System.Windows.Forms.dll` befindet, ergibt ein Verweis auf diese Assembly an dieser Stelle Sinn. Sollte es nämlich später in Ihrem Projekt noch keinen Verweis auf diese Assembly geben, wird beim Einfügen des Snippets der Verweis automatisch im Projekt eingerichtet, und Sie garantieren damit, dass alle verwendeten Objekte Ihres Snippets auch vom Basic-Compiler direkt gefunden werden können.
- Im dritten Block schließlich wird der eigentliche Code platziert, der beim Aufruf des Snippets eingefügt werden soll. Dieser Code muss sich in den eckigen Klammern der `CDATA`-Direktive befinden, so wie in der Vorlage der beiden Visual Basic-Kommentar-Zeilen zu sehen.

Im Grunde genommen ist also nicht viel zu tun. In der Vorlage brauchen wir nur die entsprechenden Anpassungen vorzunehmen, den Code einzufügen und die Snippet-Vorlage unter einem neuen Namen zu speichern.

- Ändern Sie also den ersten Block folgendermaßen ab:

```
<Header>
<Title>
    <!-- An dieser Stelle den Namen des Codeausschnitts einfügen: -->
    Datei-Öffnen-Dialog
</Title>
</Header>
```

- Den zweiten Block belassen Sie, wie er ist – dort steht nämlich der für unsere Zwecke benötigte richtige Assembly-Verweis auf die `System.Windows.Forms.dll` bereits drin.
- Im dritten Block fügen wir nun den eigentlichen Code ein. Dazu klauen wir uns den Code einfach aus dem Projekt, kopieren ihn und fügen ihn an der richtigen Stelle in der XML-Snippet-Datei wieder ein, sodass sich folgendes Ergebnis ergibt:

```
<!-- Hier folgt die eigentliche Codedefinition: -->
<Code Language="VB">
    <![CDATA[
        Dim dateiÖffnenDialog As New OpenFileDialog
        With dateiÖffnenDialog
            .CheckFileExists = True
            .CheckPathExists = True
            .DefaultExt = "*.*"
            .Filter = "Textdateien (*.txt)|*.txt|Alle Dateien (*.*)|*.*"
            Dim dialogErgebnis As DialogResult = .ShowDialog
            If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
                Exit Sub
            End If
        End With
    ]]>

</Code>
```

Und damit ist die erste einfache Code Snippet-Vorlage bereits fertig gestellt. Speichern Sie diese jetzt noch unter einem anderen Namen ab (*Datei | Vorlage.Snippet speichern unter...*) – beispielsweise unter *DateiÖffnenDialogSnippet*.

Als nächstes müssen wir Visual Studio noch mitteilen, wo sich das neue Snippet befindet, damit es eingebunden werden kann. Wie das geschieht, zeigt der folgende Abschnitt.

Hinzufügen einer neuen Snippet-Vorlage zur Snippet-Bibliothek (Codeausschnittsbibliothek)

Für diese Zwecke kennt Visual Studio den so genannten Codeausschnitt-Manager, den Sie über das *Extras*-Menü öffnen können.

TIPP Unter Umständen müssen Sie diesen Menüpunkt in Visual Studio 2008 erst einfügen (wenn er, wie bei manchen Konfigurationen, aus bislang ungeklärten Gründen nicht mehr in der Standardkonfiguration enthalten ist). Öffnen Sie dazu das Menü *Extras*, dann *Anpassen*. Wechseln Sie zu der Registerkarte *Befehle*, wählen Sie dort die Kategorie *Extras*, und ziehen Sie den Befehl *Code-Auschnittmanager* unter einen Menüpunkt Ihrer Wahl.

- Klicken Sie auf den entsprechenden Menüpunkt, erscheint ein Dialog, wie Sie ihn auch in Abbildung 8.19 sehen können.

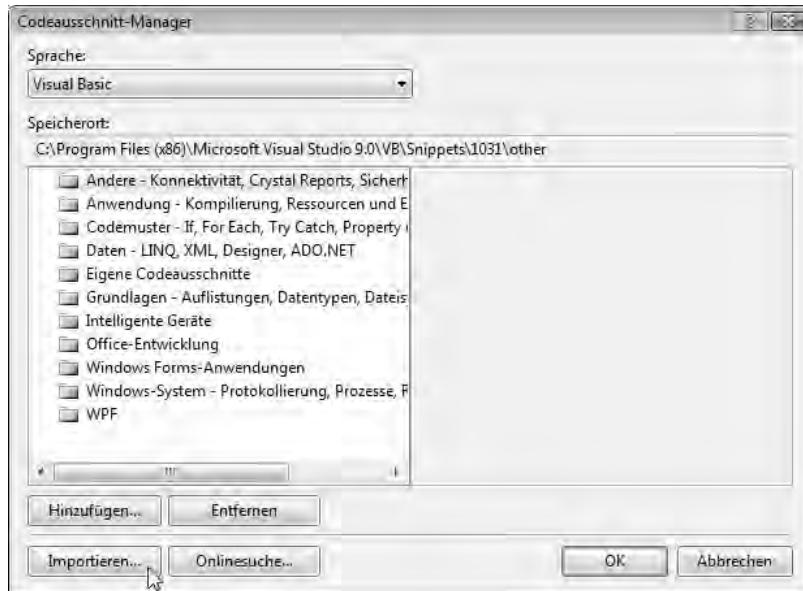


Abbildung 8.19 Mit dem Codeausschnitt-Manager können Sie vorhandene Codeausschnitte verwalten und neue Codeausschnittvorlagen hinzufügen

- Klicken Sie auf *Importieren*, um die Snippet-Vorlage auswählen zu können. Geben Sie dabei die gerade gespeicherte Snippet-Vorlage *DateiÖffnenDialog.Snippet* an.



Abbildung 8.20 Bestimmen Sie, in welcher Rubrik der neue Codeausschnitt eingesortiert werden soll

- Wählen Sie im Dialog, der anschließend erscheint, in der Liste *Speicherort*, in welcher Rubrik der neue Codeausschnitt eingesortiert werden soll.
- Klicken Sie anschließend auf *Fertigstellen*.

Verwenden des neuen Codeausschnittes

Nachdem der neue Codeausschnitt in der Bibliothek eingesortiert worden ist, können Sie als nächstes ausprobieren, ob er sich tatsächlich von dort aus abrufen und verwenden lässt. Und dazu verfahren Sie wie folgt:

- Doppelklicken Sie im Projektmappen-Explorer auf *Form1*, um diese im Designer darzustellen.
- Doppelklicken Sie im Formular auf die zweite Auslassungsschaltfläche (die zum Abrufen der zweiten Datei dienen soll), um den Codeeditor zu öffnen und den Rumpf für die Ereignisbehandlungsroutine einzufügen zu lassen.
- Wenn der Editor den Code dargestellt und den Cursor im Coderumpf platziert hat, klicken Sie auf die rechte Maustaste, um das Kontextmenü zu öffnen, und wählen anschließend den Menüpunkt *Ausschnitt einfügen*.

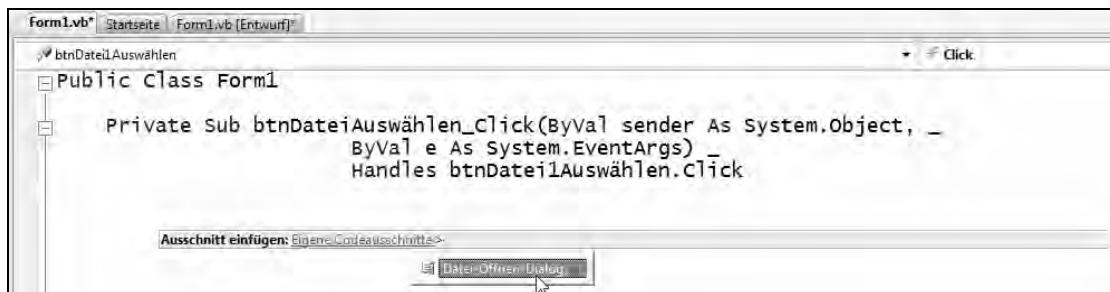


Abbildung 8.21 Der neue Codeausschnitt steht jetzt zum Einfügen in der Codeausschnittsbibliothek zur Verfügung

- Doppelklicken Sie in der Liste, die sich nun öffnet, nacheinander auf *My Code Snippets* und auf *Datei-Öffnen-Dialog*. Der gewünschte Codeblock wird nach dem zweiten Doppelklick im Editor eingefügt.

Parametrisieren von Codeausschnitten

Codeausschnitte in der Form, wie Sie sie gerade erstellt und der Bibliothek hinzugefügt haben, erleichtern die Arbeit schon ungemein. Aber das Konzept von Codeausschnitten in Visual Studio geht, was den Komfort anbelangt, noch weit über das hinaus, was wir bislang kennen gelernt haben. Betrachten wir noch mal zur Rekapitulation den Codeausschnitt, den wir gerade eingefügt haben:

```
Dim dateiÖffnenDialog As New OpenFileDialog
With dateiÖffnenDialog
    .CheckFileExists = True
    .CheckPathExists = True
    .DefaultExt = "*.txt"
    .Filter = "Textdateien (*.txt)|*.txt|Alle Dateien (*.*)|*.*"
    Dim dialogErgebnis As DialogResult = .ShowDialog
    If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
        Exit Sub
    End If
End With
```

Sie sehen, dass es hier einige Änderungsanforderungen gibt, die Sie zusätzlich erledigen müssen, um diesen Codeausschnitt auf einen jeweils neuen Kontext anzupassen – denn Sie werden schließlich nicht immer nur Textdateien, sondern auch mal Bilddateien öffnen wollen. Oder Sie möchten vielleicht, dass eben nicht auf das Vorhandensein von Pfad und Datei geprüft werden soll, wenn ein Anwender eine Datei auswählt.

Bequemer wäre es, ein vorhandenes Verfahren beim Bearbeiten eingefügter Codeausschnitte zu verwenden, das Sie gezielt innerhalb des eingefügten Ausschnittes zu den Stellen springen lässt, die geändert werden müssen. Es müsste also beispielsweise reichen, nachdem Sie den Codeausschnitt eingefügt haben, mit **→** direkt auf das erste True hinter `.CheckFileExists` zu gelangen, um es etwa in False zu ändern. Und in diesem Moment sollte sich der zweite hinter `.CheckPathExist` stehende Wert auch gleichzeitig in False ändern. Mit weiteren **→**-Tastendrücken gelangen Sie dann auf das erste hinter `.DefaultExt` stehende `*.txt`. Wenn Sie

dieses ändern, beispielsweise in `*.bmp`, sollten sich auch gleichzeitig alle anderen `*.txt` in `*.bmp` ändern. Genau das ist möglich – aber dafür bedarf es ein wenig an Umgestaltung unserer ursprünglichen Codeausschnitt-Vorlagendatei.

Um eine solche Funktionalität zu implementieren, bedarf es zweier neuer XML-Elemente in der Vorlagendatei – literale Ersetzungen und Objektersetzungen.

Literele Ersetzungen in Codeausschnittvorlagen

Literele Ersetzungen sind definierte Platzhalter reinen Texts, die an verschiedenen Stellen innerhalb des Codeausschnittes die gleichen Inhalte darstellen sollen. Ist ein literaler Platzhalter definiert, und wird er innerhalb der Codeausschnittvorlage verwendet, dann wird dieser beim Einfügen des eigentlichen Codeausschnitts in den Code im Editor entsprechend markiert. Ein Druck auf springt direkt in das entsprechende Platzhalter-Feld, und Sie können den Text wunschgemäß ändern. Wird die exakt gleich lautende literale Ersetzung an anderer Stelle in der Codevorlage abermals verwendet, bewirkt die erste Änderung eine Anpassung des Textes an allen entsprechenden anderen Stellen. Die folgende Abbildung soll dieses verdeutlichen:

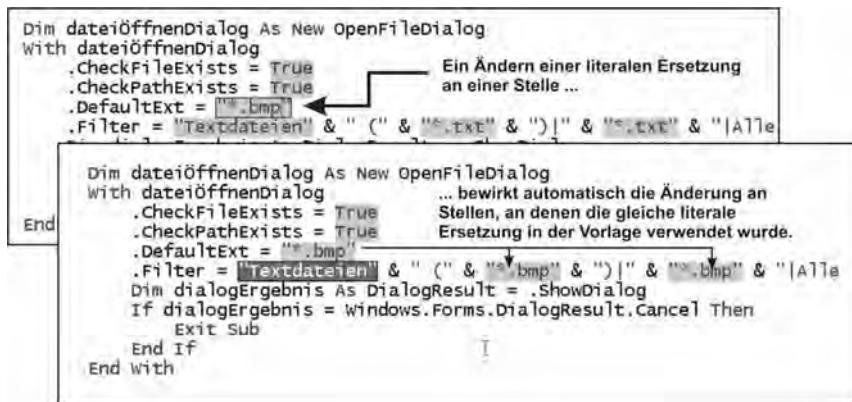


Abbildung 8.22 Das Prinzip von mehrfach eingesetzten literalen Ersetzungen

Für unsere Codeausschnitt-Vorlage bedeutet das, dass eine literale Ersetzung definiert werden muss, und diese an Stelle der eigentlichen Zeichenketten `*.txt` im Codetext verwendet wird. Die erste Stufe der Umgestaltung unserer Originalvorlage sieht damit aus, wie im anschließend gezeigten Listing:

TIPP Falls Sie die Änderungen nicht alle eingeben möchten, ist das O.K. Die komplette XML-Datei befindet sich natürlich auch in den Begleitdateien, und Sie können sie später direkt öffnen und der Ausschnittsbibliothek hinzufügen. Achten Sie aber darauf, die einzelnen Änderungsstufen an der XML-Datei wirklich gesehen und verstanden zu haben.

```

<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets
    xmlns="http://schemas.microsoft.com/VisualStudio/2008/CodeSnippet">
    <CodeSnippet Format="1.0.0">
        <Header>

```

```
<!-- An dieser Stelle den Namen des Codeausschnitts einfügen: -->
<Title>
    Datei-Öffnen-Dialog (erweitert)
</Title>
</Header>
<Snippet>

<Declarations>
    <!-- An dieser Stelle folgen die Deklarationen für literale Ersetzungen -->
    <Literal>
        <ID>Standardfilter</ID>
        <ToolTip>Definiert die zu verwendende Standardfilterabkürzung (z.B. *.txt).</ToolTip>
        <Default>"*.txt"</Default>
    </Literal>
</Declarations>

<!-- An dieser Stelle den Namen einer benötigten
     Assembly-Referenz einfügen: -->
<References>
    <Reference>
        <Assembly>System.Windows.Forms.dll</Assembly>
    </Reference>
</References>

<!-- Hier folgt die eigentliche Codedefinition: -->
<Code Language="VB">
    <![CDATA[
        Dim dateiÖffnenDialog As New OpenFileDialog
        With dateiÖffnenDialog
            .CheckFileExists = True
            .CheckPathExists = True
            .DefaultExt = $Standardfilter$
            .Filter = "Textdateien (" & $$Standardfilter$ & ")|" & _
$Standardfilter$ & "|Alle Dateien (*.*)|*.*"
            Dim dialogErgebnis As DialogResult = .ShowDialog
            If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
                Exit Sub
            End If
        End With
    ]]>
</Code>
</Snippet>
</CodeSnippet>
</CodeSnippets>
```

An den in Fetschrift gehaltenen Stellen können Sie erkennen, wie das Zusammenspiel zwischen literalen Ersetzungen und deren Einbau in den eigentlichen Code funktioniert.

Zunächst einmal bedarf es eines Declaration-Blocks im XML-Code der Vorlage, in denen jede literale Ersetzung definiert wird. Jedes neue Literal, das Sie definieren möchten, leiten Sie anschließend, wie im Beispieldialog zu sehen, mit dem `<Literal>`-Tag ein. Anschließend folgt die Definition der literalen Ersetzung durch das `<ID>`-Tag, das dem Kind einen Namen gibt – in diesem Beispiel *Standardfilter*. Anstelle also später, im Code, direkt vordefinierten Text zu verwenden, setzen Sie dann an den entsprechenden Stellen diesen Namen ein – und zwar in \$-Zeichen eingeklammert (also etwa `$Standardfilter$`).

Damit später beim ersten Einfügen des Ausschnittes in den Code an den entsprechenden Stellen überhaupt etwas erscheint, definieren Sie im gleichen Abschnitt auch einen Standardwert, der zum Einfügen kommt. Und dieser Standardwert ergibt sich aus den Angaben, die durch das `<Default>`-Tag definiert werden. Für das i-Tüpfelchen an Komfort sorgt schließlich noch eine einblendbare Tooltip-Beschreibung, die mit dem `<ToolTip>`-Tag festgelegt wird.

Ist die Definition dieser literalen Ersetzung vervollständigt, können Sie sie anschließend in die eigentliche Codevorlage wie beschrieben einbauen. In unserem Beispiel ist zunächst nur eine einzige Zeile davon betroffen:

```
.Filter = "Textdateien (" & $Standardfilter$ & ")|" & _  
$Standardfilter$ & "|Alle Dateien (*.*)|*.*"
```

Sie sehen, dass hier nun nicht mehr die festen Texte (`*.txt`) direkt stehen, sondern diese durch die literalen Ersetzungen ausgetauscht wurden.

Auf diese Weise können Sie auch weitere literale Ersetzungen in die Codeausschnittsvorlage einbauen – Sinn in unserem Beispiel macht es auch, das Wort *Textdateien* durch eine literale Ersetzung auszutauschen. Zwar kommt es im Ausschnitt nur ein einziges Mal vor; aber wie Sie gesehen haben, trägt nicht nur das Austauschen von Ersetzungen an mehreren Stellen gleichzeitig zum Komfort bei und spart Zeit. Auch das bloße Vorhandensein einer literalen Ersetzung an nur einer Stelle sorgt für Zeitersparnis und zusätzlichen Komfort, da sich eine entsprechende Stelle direkt mit der Tabulatortaste erreichen lässt. Unsere XML-Datei erfährt also einen weiteren »Umbau«:

```
<?xml version="1.0" encoding="utf-8"?>  
<CodeSnippets  
  xmlns="http://schemas.microsoft.com/VisualStudio/2008/CodeSnippet">  
  <CodeSnippet Format="1.0.0">  
    <Header>  
  
      <!-- An dieser Stelle den Namen des Codeausschnitts einfügen: -->  
      <Title>  
        Datei-Öffnen-Dialog (erweitert)  
      </Title>  
  
    </Header>  
    <Snippet>  
  
      <Declarations>  
        <!-- An dieser Stelle folgen die Deklarationen für literale Ersetzungen -->  
        <Literal>  
          <ID>Standardfilter</ID>  
          <ToolTip>Definiert die zu verwendende Standardfilterabkürzung (z.B. *.txt).</ToolTip>  
          <Default>"*.txt"</Default>  
        </Literal>  
  
        <Literal>  
          <ID>Standardfiltername</ID>  
          <ToolTip>Definiert den zu verwendenden Standardfilternamen  
            (z.B. "Textdateien" für *.txt).</ToolTip>  
          <Default>"Textdateien"</Default>  
        </Literal>  
      </Declarations>
```

```
<!-- An dieser Stelle den Namen einer benötigten
     Assembly-Referenz einfügen: -->
<References>
    <Reference>
        <Assembly>System.Windows.Forms.dll</Assembly>
    </Reference>
</References>

<!-- Hier folgt die eigentliche Codedefinition: -->
<Code Language="VB">
    <![CDATA[
        Dim dateiÖffnenDialog As New OpenFileDialog
        With dateiÖffnenDialog
            .CheckFileExists = True
            .CheckPathExists = True
            .DefaultExt = $Standardfilter$
            .Filter = $$Standardfiltername$ & " (" & $Standardfilter$ & ")|" & _
$Standardfilter$ & "|Alle Dateien (*.*)|*.*"
            Dim dialogErgebnis As DialogResult = .ShowDialog
            If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
                Exit Sub
            End If
        End With
    ]]>

    </Code>
</Snippet>
</CodeSnippet>
</CodeSnippets>
```

Für den letzten Feinschliff in Sachen Ersetzungen fehlt nun noch die angekündigte Funktionalität, die dafür sorgt, dass auch die Überprüfung auf vorhandene Dateien bzw. Ordner in einem Rutsch ein- bzw. ausgeschaltet werden kann. Doch für diesen Zweck sind literale Ersetzungen nicht geeignet, denn: Hier müssen wir eine Objektvariable manipulieren – im konkreten Fall bedeutet das, eine boolesche Variable mit den entsprechenden Werten zu versehen. Und zu diesem Zweck kommen die so genannten *Objektersetzungen* bei Codeausschnittvorlagen zum Einsatz.

Objektersetzungen in Codeausschnittvorlagen

Das Prinzip von Objektersetzungen ist dem von literalen Ersetzungen sehr ähnlich. Objektersetzungen werden ebenfalls im Declarations-Teil der Vorlagen-XML-Datei definiert, nur mit einer leicht veränderten Syntax, wie der folgende Listingausschnitt unserer XML-Datei demonstriert:

```
.
.
.

</Header>
<Snippet>

<Declarations>
    <!-- An dieser Stelle folgen die Deklarationen für literale Ersetzungen -->
    <Literal>
        <ID>Standardfilter</ID>
```

```

<ToolTip>Definiert die zu verwendende Standardfilterabkürzung (z.B. *.txt).</ToolTip>
<Default>*.txt</Default>
</Literal>
<Literal>
<ID>Standardfiltername</ID>
<ToolTip>Definiert den zu verwendenden Standardfilternamen
(z.B. "Textdateien" für *.txt).</ToolTip>
<Default>"Textdateien"</Default>
</Literal>

<!-- An dieser Stelle folgen die Deklarationen für Objektersetzungen -->
<Object>
<ID>ÜberprüfeAufVorhandensein</ID>
<Type>System.Boolean</Type>
<ToolTip>Legt fest, ob auf Vorhandensein von Pfad und Datei geprüft werden soll.</ToolTip>
<Default>True</Default>
</Object>

</Declarations>

<!-- An dieser Stelle den Namen einer benötigten
Assembly-Referenz einfügen: -->
.
.
.
```

Übrigens: XML-Puristen mögen mir in diesen Beispielen die Verwendung von deutschen Umlauten verzeihen – ich habe sie nur der Einfachheit halber eingesetzt.

Objektersetzungen werden, anders als literale Ersetzungen, mit dem `<Object>`-Tag eingeleitet. Aber auch Objektersetzungen bedürfen einer ID, die als Platzhalter im eigentlichen VB-Code eingesetzt werden kann. Zusätzlich zu literalen Ersetzungen müssen Sie aber ebenfalls bestimmen, um was für einen Objekttyp es sich bei der Objektersetzung handelt, und das geschieht mithilfe des `<Type>`-Tags, wie im Beispieldressing zu sehen. Wiederum genauso wie bei literalen Ersetzungen können Sie einen Standardwert mit `<Default>` und einen erklärenden Tooltip mit `<ToolTip>` definieren.

Eine auf diese Weise definierte Objektersetzung kommt dann im eigentlichen einzufügenden Code der Codeausschnittvorlage folgendermaßen zum Einsatz:

```

.
.
.

<!-- Hier folgt die eigentliche Codedefinition: -->
<Code Language="VB">
<![CDATA[
Dim dateiÖffnenDialog As New OpenFileDialog
With dateiÖffnenDialog
    .CheckFileExists = $ÜberprüfeAufVorhandensein$
    .CheckPathExists = $ÜberprüfeAufVorhandensein$
    .DefaultExt = $Standardfilter$
    .Filter = $Standardfiltername$ & " (" & $Standardfilter$ & ")|" & $Standardfilter$ & _
    "|Alle Dateien (*.*)|*.*"

```

```
Dim dialogErgebnis As DialogResult = .ShowDialog
If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
    Exit Sub
End If
End With
]]>
.
.
```

Sie sehen: Die Anwendung von Objektersetzungen ist zu diesem Zeitpunkt dieselbe wie bei literalen Ersetzungen. Und im Übrigen: Unsere Codeausschnittvorlage ist nun fertig!

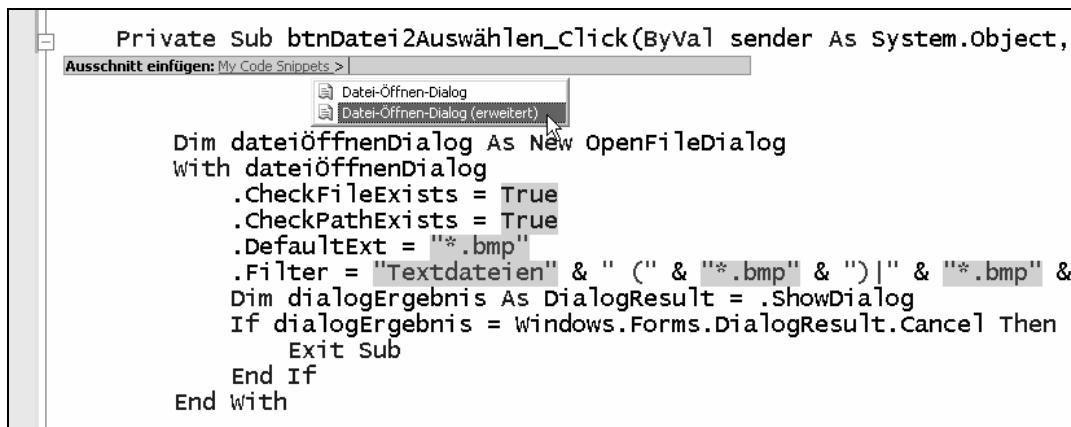


Abbildung 8.23 Die fertige Codeausschnittvorlage in Aktion!

Falls Sie die Änderungen bis zu diesem Zeitpunkt selbst durchgeführt haben, speichern Sie die XML-Vorlage am besten jetzt erst einmal ab – beispielsweise unter dem Namen *DateiÖffnenDialogEx.snippet*. Falls nicht, finden Sie die Vorlage unter dem gleichen Namen im Projektverzeichnis des Beispielprojektes.

Und dann wiederholen Sie das Spielchen aus Abschnitt »Hinzufügen einer neuen Snippet-Vorlage zur Snippet-Bibliothek (Codeausschnittsbibliothek)« ab Seite 293.

Das Einfügen des neuen Codeausschnittes und der eingefügte Ausschnitt selbst sollten anschließend so ausschauen, wie es Abbildung 8.23 zeigt.

Teil B

Umsteigen auf Visual Basic 2008

In diesem Teil:

Migrieren zu Visual Basic 2008 – Vorüberlegungen	305
Migration von Visual Basic 6-Anwendungen	313
Neues im Visual Basic 2008-Compiler	361

Kapitel 9

Migrieren zu Visual Basic 2008 – Vorüberlegungen

In diesem Kapitel:

Grundsätzliches zur Umstellung von Visual Basic 6-Anwendungen auf .NET	308
Aufwandsabschätzung einer Migration	310
Interop Forms Toolkit – »Weiche Migration« für kleinere und modulare Projekte	311
10-teilige Webcast-Reihe zum Thema Migration	311

Wikipedia definiert den Begriff *Migration* (*Informationstechnik*) folgendermaßen: »Unter Migration versteht man im Rahmen der Informationstechnik den Umstieg eines wesentlichen Teils der eingesetzten Software beziehungsweise den Transfer von Daten aus einer Umgebung in eine andere, sowie die Umstellung von Hardware einer alten Technologie in neue Technologien unter weitgehender Nutzung vorhandener Infrastrukturen. Die häufig eng miteinander zusammenhängenden Prozesse lassen sich in ›Software-Migration‹ und ›Daten-Migration‹ aufteilen. Diese Umstellung kann Probleme bereiten. Voraussehbare Folgen werden als ›missbilligt‹ (engl.: ›deprecated‹) markiert.«¹

Wenn Sie Ihre Software bereits in einer der seit Januar 2002 vorhandenen .NET Entwicklungsumgebungen entwickelt haben, dann gibt es kein nennenswerten Problem mit einer Migration auf eine neuere .NET- oder Visual Studio-Version. Sie könnten sich noch überlegen, ob Sie eine »veraltete« .NET-Technologie durch eine neue ersetzen; ob – gerade wenn Ihre Software noch unter dem Framework 1.1 entstanden ist – Sie sie für das .NET Framework 2.0 oder 3.5 anpassen, alte 1.1 Features durch die neuen des 2.0, 3.0 oder 3.5-Networks ersetzen, Ihre Anwendung neu durchtesten und innerhalb kürzester Zeit wieder aktualisiert einsetzbar machen. Sie können jede .NET-Projektmappe, die in einer älteren Version von Visual Studio entwickelt wurde, mit einem entsprechenden Assistenten in das neue Format überführen. In den wenigstens Fällen werden Ihnen so genannte *Breaking Changes*² zwischen den unterschiedlichen Visual Basic- bzw. .NET Framework-Versionen (1.0 auf 2.0 bzw. 1.1 auf 2.0 ist, wenn überhaupt, kritisch) so sehr ins Gehege kommen, dass Ihre Software den Dienst verweigert – dass überhaupt Probleme dabei auftauchen, ist eher selten der Fall. Dank Framework-Targeting (siehe Kapitel 11) können Sie auch mit Visual Studio 2008 noch Anwendungen für das .NET Framework 2.0 entwickeln. Dann erwarten Sie überhaupt keine Umstellungsprobleme.

Sie können in diesem Fall dieses und das folgende Kapitel bedenkenlos überblättern.

Anders sieht es aus, wenn Sie noch eine der immer noch zahlreich sich im Einsatz befindlichen Visual Basic 6.0-Anwendungen pflegen, weiterentwickeln bzw. nunmehr auf einen ersten .NET-Stand heben müssen.

Das Problem der Visual Basic 6.0-Migration

Wenn Sie Ihre Software allerdings immer noch unter Visual Basic 6.0 laufen haben, dann sollten Sie sich das Wort *deprecated* gut merken. Denn leider ist Ihre Software, auch wenn es Ihnen vielleicht noch nicht so bewusst geworden ist, bereits als ganzes mit diesem Wort abgestempelt, und das liegt an Folgendem:

Einige der in Visual Basic 6.0-Software typischerweise verwendeten Komponenten macht bereits unter Windows Vista ganz erhebliche Schwierigkeiten, teilweise so erheblich, dass die Software nicht mehr fehlerfrei durchlaufen kann. Das betrifft insbesondere:

- Das Ausführen von VB6-Programmen im Allgemeinen, wenn diese nicht mit erweiterten Benutzerrechten (*elevated UAC*) gestartet werden.

¹ Quelle: [http://de.wikipedia.org/wiki/Migration_\(Informationstechnik\)](http://de.wikipedia.org/wiki/Migration_(Informationstechnik)), Stand 12.10.2008

² Als *Breaking Change* bezeichnet man in diesem Kontext eine Änderung der Verhaltensweise einer bestimmten Funktion einer Bibliothek, die beispielsweise aufgrund eines Fehlers notwendig geworden ist, von der allerdings andere Funktionen einer Anwendungssoftware abhängig sind, und man Gefahr läuft, dass sich die abhängige Anwendung durch Korrigieren des Fehlers anders oder fehlerhaft verhalten könnte.

- Den hart-codierten Zugriff auf Systemverzeichnisse (der im Grunde genommen sowieso ein Unding ist, aber aus Unwissenheit vieler VB6-Entwickler leider in vielen Entwicklungen an die nächste Generation vererbt wurde ...).
- Das Benutzen bestimmter Scripting-Host-Objekte, wie beispielsweise das `FileSystemObject`.
- Selbstgeschriebene Setups auf Basis des Setup-Quellcodes, der mit Visual Basic 6.0 zur Verfügung gestellt wird, sowie dessen Bootstrapper.
- Zugriff auf die Registry bzw. bestimmter Teile der Registry unabhängig von ein- oder ausgeschalteter UAC.
- Unterschiedliche/Abweichende Funktionsweise bei der Verwendung der Socket-Komponenten unter Windows Vista.

Und das sind nur einige wenige Punkte.

In Windows Vista und Windows Server 2008 gibt es, damit VB6-Anwendungen *überhaupt* noch laufen können, einige speziell angepasste DLLs zu genau diesem Zwecke als Bestandteil des Betriebssystems.³ Natürlich unterliegen diese angepassten DLLs genau den gleichen Supportgewährleistungsgarantien, wie alle anderen Betriebssystemteile, oder anders aus gedrückt: Sie haben Support auf diese Kompatibilitätsbibliotheken, sodass es auf alle Fälle momentan auf jedenfalls noch grundsätzlich möglich ist, VB6-Anwendungen auf modernen Betriebssystemen zu installieren. Einige weitere DLLs und OCX-Steuerlemente werden darüber hinaus vom Betriebssystem »toleriert« und sollten – ohne Garantie von Microsoft – keine Probleme zur Laufzeit machen.

Es gibt allerdings leider auch eine Aussage zu Komponenten, für die es nicht nur keinen Support gibt, sondern die auch auf den Betriebssystemen Vista und Windows Server 2008 nicht unterstützt werden, die aber typischerweise zu einer VB6-Anwendung gehören können. Und dabei handelt es sich um die folgenden:

- | | | | |
|----------------|----------------|----------------|----------------|
| ■ anibtn32.ocx | ■ graph32.ocx | ■ keysta32.ocx | ■ autmgr32.exe |
| ■ autprx32.dll | ■ racmgr32.exe | ■ racreg32.dll | ■ grid32.ocx |
| ■ msoutl32.ocx | ■ spin32.ocx | ■ gauge32.ocx | ■ gswdll32.dll |
| ■ ciscnfg.exe | ■ olecnv32.dll | ■ rpcltc1.dll | ■ rpcltc5.dll |
| ■ rpcltccm.dll | ■ rpclts5.dll | ■ rpcltscm.dll | ■ rpcmqcl.dll |
| ■ rpcmqsvr.dll | ■ rpcss.exe | ■ dbmsshrn.dll | ■ dbmssocn.dll |
| ■ windbver.exe | ■ msderun.dll | ■ odkob32.dll | ■ rdocurs.dll |
| ■ vbar332.dll | ■ visdata.exe | ■ vsdbflex.srg | ■ threed32.ocx |
| ■ MSWLess.ocx | ■ tlbinf32.dll | ■ triedit.dll | |

Weitaus schlimmer wird es, was mit Windows 7 die Nachfolge von Windows Vista bzw. Windows Server 2008 anbelangt, denn nach derzeitigem Stand der Dinge wird es für dieses Betriebssystem und seine Derivate dann gar keine Unterstützung dieser Programme mehr geben, oder, wenn man diese Aussage ganz streng auslegt: **Visual Basic 6-Programme werden unter Windows 7 und Ablegern nach den derzeitigen Aussagen von Microsoft nicht mehr laufen.**

³ Mehr dazu verrät der IntelliLink **B0901**.

Nicht nur Microsoft ist davon überrascht gewesen, wie viele Visual Basic 6.0-Anwendungen zum Zeitpunkt des Auslaufens des Supports für Visual Basic 6.0 am 8.4.2008 noch nicht einmal im Anfangsstadium der Umstellung waren – und mittlerweile wird es richtig eng.

Sollten Sie also noch eine unternehmenskritische VB6-Anwendung in Betrieb haben, und möchten Sie, dass diese auch in Zukunft noch auf moderneren Betriebssystemen läuft, kann ich Ihnen nur empfehlen, sich eher gestern als heute mit der Migrationsproblematik von VB6-Anwendungen auseinander zu setzen.

Kurzfristige Rettung mit virtuellen PCs

Was können Sie machen, wenn Sie feststellen, dass Ihre Kunden bereits auf Vista umgestellt haben? In vielen Fällen können Sie heutzutage gar keine Rechner mehr beziehen, und sich noch für ein älteres Betriebssystem entscheiden – lediglich die meisten Netbooks werden zum Zeitpunkt, zu dem diese Zeilen entstehen, noch standardmäßig mit Windows XP SP3 vorinstalliert ausgeliefert. Alle anderen Desktop- und Notebook-Systeme bieten nur noch in seltensten Fällen eine so genannte Downgrade-Option auf Windows XP, geschweige, dass sie noch mit einem vorinstallierten XP an den Start gehen.

In diesem Fall können virtuelle PCs die kurzfristige Rettung sein, indem Sie nämlich versuchen, eine gebrauchte Betriebssystemlizenz für Ihre Kunden zu erwerben, und Ihre Software nicht in einem echten, sondern in einem virtuellen PC, vielleicht unter Windows 2000 oder Windows XP für eine kurze Zeit noch lauffähig zu halten.

Mehr zu diesem Thema können Sie im vorherigen Kapitel 8 finden.

Grundsätzliches zur Umstellung von Visual Basic 6-Anwendungen auf .NET

Es gibt eine einfache, aber leider in vielen Fällen nicht funktionierende Methode, eine VB6-Anwendung auf Visual Basic .NET umzustellen – den Migrationsassistenten von Visual Studio 2008. Sie können in Visual Basic .NET seit der ersten Version eine VB6-Projektmappe öffnen, um diesen Migrationsassistenten ins Leben zu rufen.

Die vorherige Auflage dieses Buchs begann das Kapitel zu diesem Thema mit den Worten: »[...] Visual Basic 2005 verfügt wie Visual Basic 2003 über einen Upgrade-Assistenten, der aus Ihrem VB6-Projekt ein VB2005-Projekt machen und die dafür erforderlichen Umbauten vornehmen soll. Für kleinere Projekte oder Algorithmen, die projektunabhängig formuliert wurden, ist dieser Upgrade-Assistent sicherlich sinnvoll. [...] Wenn Sie planen [...], eine größere Entwicklung auf .NET 2.0 zu portieren, tun Sie sich selber den Gefallen und lassen Sie den Upgrade-Wizard einfach außen vor.«

Diese Aussage war für die 2005er-Version des Migrationsassistenten sicherlich richtig. Die Qualität des Migrationsassistenten in der Visual Basic 2008-Version ist ganz erheblich besser geworden, dennoch zeigen Erfahrungen des Autors, dessen Mitarbeiter, seiner Kollegen und seines Fachlektors, dass Sie sich bei der Migration größerer Projekte über Folgendes bewusst sein sollten.

VB2008-Migrationsassistent: Go's ...

- Der Migrationsassistent ist vergleichsweise gut, was das Übersetzen von reinem Visual Basic-6-Code anbelangt. Er berücksichtigt viele der Unterschiede zwischen den BASIC-Versionen, und zu diesem Zweck wird auch das folgende Kapitel für Sie interessant sein, das die Neuheiten in der 2008er-Version vorstellt. Für sehr algorithmusorientierte Anwendungen ist er sicherlich eine gute Alternative zum kompletten neu Programmieren der Anwendung.
- Der Migrationsassistent hilft Ihnen in seiner jetzigen Version ebenfalls, Projekte mit vielen Visual Basic 6-Formularen auf den aktuellen Stand zu bringen. In Details ist dabei sicherlich noch Nacharbeit erforderlich, jedoch ist es einfacher, Formulare auf diese Weise zu migrieren, als viele Formulare von Grund auf neu zu erstellen.

ACHTUNG Aber: Dieser gerade genannte Punkt gilt uneingeschränkt nur für Formulare, die in der ursprünglichen Version keine ActiveX-Steuerelemente verwenden. Probleme gibt es schon dann, wenn beispielsweise Komponenten wie das MsFlexGrid zu .NET portiert werden sollen. In diesem Fall sollte ein anderer Weg eingeschlagen werden, der aber eine Fall-zu-Fall-Entscheidung notwendig macht. Das gilt insbesondere dann, wenn Sie ActiveX-Steuerelemente oder solche von Drittherstellern wie beispielsweise Component One-Komponenten verwenden. Der Migrationsassistent wird nämlich nicht auf die entsprechenden .NET-Pendents umstellen, sondern lediglich einen »nur« funktionierenden so genannten .NET-Wrapper implementieren, um das das alte COM-orientierte Steuerelement unter .NET weiterzuverwenden, was natürlich nicht der Sinn einer Migration sein kann.

Es gibt aber entsprechende Tools und Vorgehensweise, mit denen sich auch diese Problematik im Rahmen einer automatisierten Code-Migrierung lösen lässt, die sich bei größeren Projekten finanziell sicherlich lohnt. Mehr zum Thema Migration komplexer VB6-Anwendungen finden Sie auch unter dem IntelliLink **B0902** auf den Seiten von ActiveDevelop.

... und No-Go's

- Wenn es um das Migrieren komplexerer Anwendungen geht (Richtwert: ab 20.000 Zeilen Code Projektgröße), bei denen insbesondere auch Datenbank-Provider-Anpassungen (DAO, RDO, ADO auf ADO.NET oder LINQ) notwendig sind, sollten Sie ein professionelles Konvertierungstool verwenden, das Ihnen bei der automatischen Umformulierung des Codes weiterhelfen kann. Auch zu diesem Thema finden Sie unter dem IntelliLink **B0902** entsprechende Informationen.
- Natürlich kann der Migrationsassistent keine Datenbankenplattformen migrieren. Wenn Sie sich schon die Mühe machen, bestehende Anwendungen zu migrieren, sollten auch in Erwägung gezogen werden, andere Komponenten der Anwendung wie die benutze Datenbank durch entsprechend moderne Systeme abzulösen. Sie sollten beispielsweise bei der Ablösung Ihrer Access-Anwendung in Erwägung ziehen, die Access-Datenbank auf SQL Server 2008 umzustellen.

HINWEIS SQL Server 2008 gibt es in einer sogenannten Express Edition, die Sie zusammen mit Ihrer Anwendung kostenlos verteilen dürfen, und die eine kaum eingeschränkte Version des großen Microsoft SQL Servers, was die relationale Datenbank angeht, darstellt. Einschränkungen gibt es in Hinsicht auf Datenbankgröße (maximal 4 GByte), maximal verwendeter Hauptspeicher (max. 1 GByte) und verwendete Prozessoren (maximal einer, aber Achtung: mit allen Kernen, sodass Sie auch mit einem Quad-Core-Prozessor durchaus extrem performante hochkomplexe Datenbankabfragen ausführen können).

Darüber hinaus bietet Microsoft auch für die Vollversionen seiner SQL Server spezielle und sehr günstige Lizenzvereinbarungen an, wenn Sie sie ausschließlich zur Nutzung mit Ihrer Anwendung bei Ihren Kunden einsetzen, sodass Ihre Anwendung, was die Datenbank betrifft, fast schon beliebig skalierbar wird, wenn Sie sie auf SQL Server 2008 migrieren.

TIPP Wir sehen es in unseren Schulungen, bei Coachings und Beratungen immer wieder, dass IT-Teams versuchen, nicht ausreichend performante SQL Server, durch eine regelrechte Hardwareschlacht mit Gigabytes an Speicher und noch mehr Prozessoren wegzuskalieren. Dabei vergessen sie oft, dass Datenbankdesign und SQL Server-Optimierung in Verbindung mit der Wahl der richtigen Plattform (16 GByte in einem 32-Bit Windows Server 2003 Standard ergeben leider keinen Sinn!) mindestens genau so viel Optimierungsbedarf ergeben. Hier kann eine Beratung in diese Richtung gehend sicherlich eine überdenkenswerte Alternative sein. Informieren Sie sich bei Microsoft oder unter dem IntelliLink **B0903**.

- VB6 kannte keine Vererbung im klassischen Sinne, und damit keine echte Polymorphie, die sich ein Klassenmodell zu nutzen machen konnte. Dank Visual Basics Fähigkeit zum Late Binding war Polymorphie allerdings trotz nicht vorhandener Technik für die Entwicklung regelrechter Klassenmodelle dennoch in Maßen möglich. Auch hier wird der Migrationsassistent scheitern, da er verwendete Klassen nach Möglichkeit konkretisiert.

Der letzte Punkt ist ohne ein Showstopper für den Migrationsassistenten bzw. für überhaupt eine Art von »rüberschaufeln« des Codes von VB6 zu .NET. In dem Moment, in dem Ihr .NET-Programm besser mit einem solchen Klassenmodell neu aufgestellt werden soll, ergibt das Neuprogrammieren mehr Sinn, und Sie – wenn überhaupt – holen nur noch einzelne Blöcke von Code aus VB6 zu .NET herüber.

Aufwandsabschätzung einer Migration

Erfahrende Consultants des Costa Ricanischen Migrationsspezialisten ArtinSoft kalkulieren die Migration einer großen VB6-Anwendung mit ca. 1 bis 1,5\$ pro Codezeile – und die Erfahrung zeigt, dass eine Neuentwicklung einer größeren Anwendung auch genau diese Kosten verursacht. Dabei schlägt die Konzeptierung des neuen Projektes mit ca. 10–20%, die eigentliche Code-Konvertierung mit ca. 40–50% und das Testing mit weiteren 40% zu Buche. Diese Richtwerte gelten für Projekte mit einer Größe ab ca. 25.000 Zeilen Code – kleinere Projekte sind sicherlich einfacher zu migrieren, da sie in der Regel nicht so komplex sind und auch nicht so viele anzupassende Komponenten verwenden.

Die Aufwandsabschätzung einer Migration gestaltet sich in vielen Fällen aber auch schon deswegen extrem schwierig, weil eine professionelle VB6-Anwendung in der Regel über mehrere Entwicklergenerationen gewachsen ist.

ArtinSoft hat deswegen zusammen mit Microsoft ein so genanntes Assessment-Tool zur Verfügung gestellt, dass Ihnen die Aufwandsabschätzung einer Migration deutlich erleichtert. Sie können dieses Tool unter dem IntelliLink **B0902** herunterladen.

Interop Forms Toolkit – »Weiche Migration« für kleinere und modulare Projekte

Mit dem Interop Forms Toolkit von Microsoft haben Sie die Möglichkeit, eine Migration Ihrer Software Zug um Zug vorzunehmen. Der Vorteil dabei: Ihre Software bleibt über die verschiedenen Zeiträume immer voll einsatzfähig – jedenfalls solange Sie noch nicht unmittelbar die Notwendigkeit haben, die Software auf einem VB6-inkompatiblen Betriebssystem verwenden zu müssen.

Das heißt im Klartext: Solange Sie mit Vista oder XP noch eine Weile vorlieb nehmen können, nutzen Sie die Zeit, um mithilfe des Interop Forms Toolkit Ihre Software nach und nach zu .NET »herüberzuholen«.

Das funktioniert folgendermaßen:

- Das Interop Forms Toolkit erlaubt es Ihnen, sowohl Formulare als auch Steuerelemente in .NET zu formulieren und diese auf eine sehr einfache Weise in Ihre vorhandene VB6-Anwendung einzubauen.
- Es erlaubt Ihnen ebenfalls, in beiden Systemen – wenn auch mit ein wenig mehr Aufwand – quasi gleichzeitig zu debuggen. Sie können also im Einzelschrittmodus sowohl in ein .NET-Projekt, dessen Assemblies Sie in VB6 eingebunden haben, hineinsteppen, als auch das Debuggen anschließend in dem hostenden VB6-Projekt fortsetzen.

Auf diese Weise können Sie verschiedene Komponenten identifizieren, die Sie in Ihrem VB6-Projekt verwenden, und diese »weich«, also eine nach der anderen migrieren.

Zu erklären, wie das im Detail funktioniert, würde den Rahmen an dieser Stelle sprengen. Eine Demonstration des Interop Forms Toolkit ist aber ebenfalls Bestandteil der 10-teiligen Webcast-Reihe, von der im folgenden Abschnitt die Rede ist.

TIPP Genießen Sie die Vorführung des Internet Forms Toolkit doch einfach durch den Autor dieses Buchs am heimischen Fernseher! ;-)

10-teilige Webcast-Reihe zum Thema Migration

Aus Platzgründen können wir das Thema Migration natürlich nicht in aller Ausführlichkeit behandeln – auch wenn es mindestens so wichtig ist, wie das Erlernen des .NET-Handwerkszeugs, denn ohne dieses wird Ihnen eine Migration natürlich Probleme machen.

Microsoft stellt auf dem Microsoft Developer Network (MSDN) eine 10-teilige Webcast-Reihe zum Thema bereit, die Sie über den IntelliLink **B0904** erreichen können. Die Videodateien dieser Serie finden Sie ebenfalls auf der beiliegenden Buch-DVD direkt zum Anschauen.

Sollten Sie übrigens direkt persönliche Beratungen zum Thema Migration zu Visual Basic .NET oder auch zu C#.NET benötigen, können Sie sich jederzeit mit ActiveDevelop unter Info@ActiveDevelop.de oder unter den Info-Nummern, die Sie unter www.activedevelop.de finden, in Verbindung setzen.

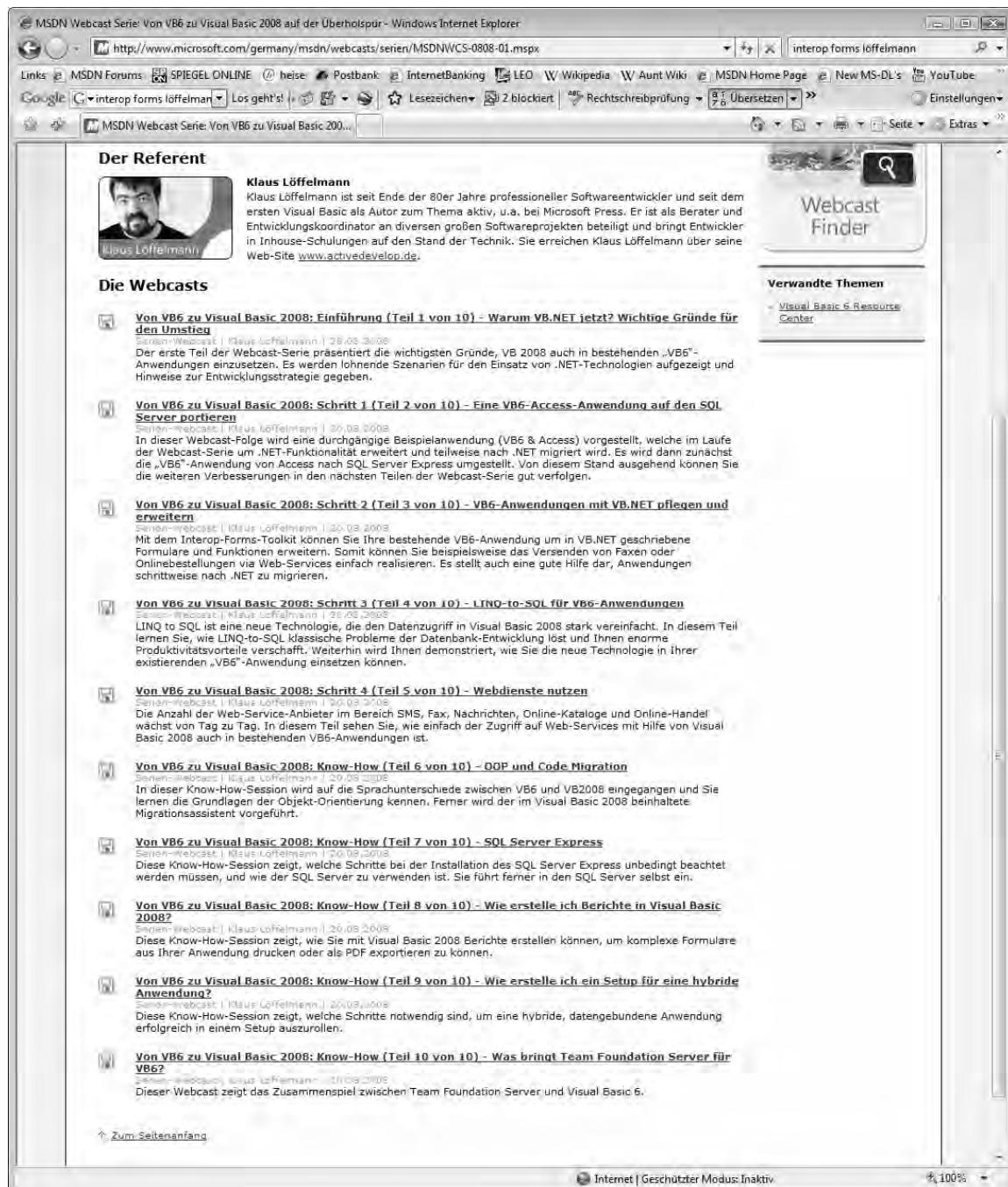


Abbildung 9.1 Eine 10-teilige Webcast-Reihe zum Thema Migration zeigt Ihnen alles, was Sie wissen müssen, anschaulich an Praxisbeispielen

Kapitel 10

Migration von Visual Basic 6-Anwendungen

In diesem Kapitel:

Unterschiede in der Variablenbehandlung	315
Alles ist ein Objekt oder »let Set be«	329
Gezieltes Freigeben von Objekten mit Using	330
Direktdeklaration von Variablen in For-Schleifen	333
Continue in Schleifen	334
Gültigkeitsbereiche von Variablen	335
Die Operatoren += und -= und ihre Verwandten	340
Fehlerbehandlung	342
Kurzschlussauswertungen mit OrElse und AndAlso	348
Variablen und Argumente auch an Subs in Klammern!	350
Namespaces und Assemblies	350

Zu wissen, welche Strategien Sie bei der Migration Ihrer VB6-Anwendung grundsätzlich anwenden müssten, ist das Eine. Welche grundsätzlichen Unterschiede es aber zwischen VB6-Anwendungen und .NET-Visual Basic gibt, ist mindestens genau so wichtig. Einige der im folgenden beschriebenen Fakten finden Sie sicherlich in der einen oder anderen Form auch noch an anderer Stelle im Buch besprochen – eine, wie ich finde, durchaus akzeptable Gefahr, möchte ich an dieser Stelle dennoch eingehen, redundante Fakten zu liefern, damit Sie alle relevanten, für eine Migration wichtigen Infos dazu an einer Stelle gebündelt nachlesen können.

Über etwas Wichtiges sollten Sie sich im Klaren sein: Visual Basic .NET hat natürlich eine gewisse Ähnlichkeit zu Visual Basic 6.0. Eine gar nicht so kleine sogar. Doch selbst bei den eigentlichen Sprachelementen gibt es Unterschiede, und gemessen an der Tatsache, dass Ihnen schon alleine mit dem .NET Framework 2.0 rund 8.000 Klassen zur Lösung der unterschiedlichsten Aufgaben zur Verfügung stehen, möchte ich Visual Basic .NET nicht nur als einfache Weiterentwicklung von VB6 bezeichnen. Wenn Sie mit der Vorstellung an das Erlernen von VB.NET herangehen, dass Sie im Grunde genommen eine komplett neue Sprache lernen, kommt das 1. nicht nur der Realität recht nahe, sondern Sie werden 2. auch schnellere Erfolgsergebnisse haben, da Sie mit Ihrem VB6-Können zumindest einen – na ja, sagen wir – mittelgroßen Vorsprung genießen.

Die wichtigsten Unterschiede in diesem Bereich, also dort, wo es scheinbar eine Schnittmenge zwischen Visual Basic 6 und Visual Basic .NET gibt, die aber dann doch gar keine ist, soll dieses Kapitel klären. Nicht mehr und nicht weniger. Die eigentlichen Neuerungen in Visual Basic 2008 werden Sie im nächsten Kapitel und natürlich auch in den weiteren Buchteilen dann im Detail erfahren.

BEGLEITDATEIEN

Die abgedruckten Codeausschnitte dieses Buchs vereinigt ein Projekt, das Sie im Pfad

...\\VB 2008 Entwicklerbuch\\B - Migration\\Kapitel 10\\VonVb6ZuVbNet

unter dem Projektnamen *VonVb6ZuVbNet.sln* finden. Es besteht aus einem Formular und mehreren Schaltflächen, die die jeweiligen Beispiele laufen lassen. Ausgaben werden dabei im Ausgabefenster angezeigt. Sollte dieses Fenster während des Programmablaufs nicht sichtbar sein, lassen Sie es einfach mit **[Strg] [Alt] 0** anzeigen. Kleinere Codeschnipsel, VB6-Code oder Code, der zu Demonstrationszwecken mit Absicht Fehler enthält, sind dort nicht berücksichtigt.

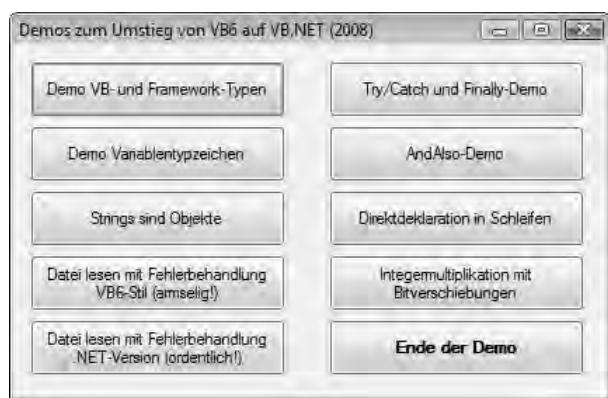


Abbildung 10.1 Mit diesem Programm können Sie die Demos in diesem Kapitel ausprobieren und nachvollziehen

Unterschiede in der Variablenbehandlung

Los geht es mit den Elementen des Entwickelns, die Sie am häufigsten benötigen, den primitiven Variablen und Arrays. »Primitive Variablen« meint dabei übrigens die »eingebauten und fest verdrahteten« Variablen-typen in Visual Basic, wie Integer, Double, String, Boolean etc. Und schon hier gibt es eine nicht unerhebliche Anzahl an Unterschieden.

Veränderungen bei primitiven Integer-Variablen – die Größen von Integer und Long und der neue Typ Short

Mal davon abgesehen, dass es einige Variablentypen in VB.NET nicht mehr gibt und viele neue hinzugekommen sind, haben sich einige Variablentypen auch verändert.

Integer- und Short-Variablen

Integer-Variablen verfügen seit der ersten VB.NET-Version über einen 32-Bit-breiten vorzeichenbehafteten Darstellungsbereich, anders als in VB6, wo sie mit 16-Bit auskommen mussten.

Damit stellen sie in VB.NET den Bereich von -2.147.483.648 bis 2.147.483.647 dar, im Gegensatz zu VB6, bei dem sich der Bereich nur von -32.768 bis 32.767 erstreckte. 16-Bit vorzeichenbehaftete Datentypen werden in VB.NET durch den Datentyp Short dargestellt – Short in VB.NET entspricht also Integer in VB6.

Long-Variablen

Das was Long-Variablen in VB6 waren sind sie jetzt in Form von Integer in VB.NET. Long gibt es aber auch in VB.NET, nur basiert sie hier auf 64 und nicht auf 32 Bit. Ihr Darstellungsbereich erweitert sich also auf -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807. Oder im Klartext gesprochen, von rund minus 9,2 Trillionen auf plus 9,2 Trillionen.

TIPP: Schon zu VB 6.0-Zeiten gab es Hinweise, dass man für Zählvariablen etc. doch Long als Datentyp verwenden soll. Die Intel Prozessoren sind nämlich für den Zugriff auf gerade Speicheradressen optimiert und ein Long in VB 6.0 ist ein DWORD (also 4 Bytes lang). In VB.NET gilt dieser Ratschlag daher für Integer: Auch diese können immer wieder an geraden Adressen (da auch als DWORD implementiert) ausgerichtet werden.

Anekdoten aus der Praxis

Ein Schulungsteilnehmer fragte mich in diesem Zusammenhang, wo denn bitteschön in der Praxis mit so hohen Werten hantiert werde. Ich musste diese Frage erst gar nicht beantworten, denn ein Kollege reagierte prompt und fragte ihn: »Wie viele Bytes hat denn deine 300 GByte-Festplatte, die du dir gestern neu bestellt hast?« Er antwortete triumphierend: »300 Millionen, und da komme ich doch locker mit 32-Bit hin.« Sein Kollege entgegnete ihm: »300 Millionen, wow, da kannst du ja dann eine halbe CD drauf speichern!«

Der gute Mann hatte kurzerhand drei Zehnerpotenzen unterschlagen und hätte die Kapazität seiner Festplatte in Bytes natürlich nicht mit einem 32-Bit-Wert darstellen können, darüber hinaus auch das Gelächter auf seiner Seite.

Typen, die es nicht mehr gibt ...

Von einigen Variabtentypen hat sich VB.NET trennen müssen – in erster Linie übrigens, um mit dem so genannten *CTS* – dem Common Type System – das unter anderem den Standard aller primitiven Datentypen aller auf dem Framework basierenden Programmiersprachen regelt, kompatibel zu bleiben.

Good bye Currency, welcome Decimal!

Den Datentypen *Currency*, den es in VB6 zur Verarbeitung von Währungsdaten, sprich: Geldbeträgen gab, gibt es nicht mehr. Stattdessen verwenden Sie *Decimal*. Im Gegensatz zu *Double* oder *Single* gibt es bei *Decimal* keine zahlenkonvertierungsbedingten Rundungsfehler. Für das Berechnen von Geldbeträgen ist das natürlich eine sinnvolle Voraussetzung. *Decimal* wird andererseits komplett »manuell« durch den Prozessor berechnet, und nicht durch dessen Mathe-Logik. Damit ist *Decimal* gleichzeitig auch der langsamste numerische Datentyp.

And good bye Variant – hello Object!

Für *Variant* gilt das Gleiche – es gibt diesen Typ im .NET Framework nicht mehr, na ja, sagen wir, nicht mehr an der Oberfläche.¹ Aber auch das ist keine Schikane der Framework-Entwickler, damit Sie Ihre Programme, die Sie bisher in V6.0 entwickelt hatten, auch wirklich gründlich umschreiben müssen – dieser Datentyp entspricht in seiner damaligen Funktionsweise einfach nicht dem Standard in Sachen Typsicherheit. In vielen Fällen können Sie aber *Object* verwenden, um den eigentlichen Datentyp zu kapseln. Doch zu diesem Thema erfahren Sie im Teil »OOP« mehr.

HINWEIS Falls Sie aus Gewohnheit versucht sein sollten, einen *Variant*-Datentyp im Codeeditor zu deklarieren, wandelt der Editor *Variant* automatisch in *Object* um.

... und die primitiven Typen, die es jetzt gibt

Und das sind eine ganze Menge. Die folgende Tabelle zeigt Ihnen eine kurze Übersicht – neu hinzugekommene sind in Fettschrift dargestellt.

Detailliertes zu diesem Thema gibt's in Teil D dieses Buchs.

Typename	.NET-Typename	Aufgabe	Wertebereich
Byte	System.Byte	Speichert vorzeichenlose Integer-Werte mit 8 Bit Breite.	0 bis 255
SByte	System.SByte	Speichert vorzeichenbehaftete Integer-Werte mit 8 Bit Breite.	-127 bis 128
Short	System.Int16	Speichert vorzeichenbehaftete Integer-Werte mit 16 Bit (2 Byte) Breite.	-32.768 bis 32.767

¹ Wobei das nicht ganz richtig ist, denn es gibt ihn Framework-intern noch (für diejenigen, die es genau wissen wollen: *System.Variant* befindet sich in der *mscorlib*, also der Common Language Runtime-Bibliothek), Sie können ihn nur nicht mehr verwenden. *Variant* wird intern aus Kompatibilitätsgründen zur Zusammenarbeit mit COM-Objekten verwendet. Kleine Randnotiz: In einer der ersten Betas von Visual Basic 2002 konnten Sie ihn sogar noch verwenden; da *Object* seine Funktionalität jedoch weitestgehend übernehmen kann, beschlossen die Entwickler des Frameworks, ihn für die Außenwelt unzugänglich zu machen, weil sie eine zu große Konfusion bei den Entwicklern zwischen *Variant* und *Object* befürchteten.

Typename	.NET-Typename	Aufgabe	Wertebereich
UShort	System.UInt16	Speichert vorzeichenlose Integer-Werte mit 16 Bit (2 Byte) Breite.	0 bis 65.535
Integer	System.Int32	Speichert vorzeichenbehaftete Integer-Werte mit 32 Bit (4 Byte) Breite. HINWEIS: Auf 32-Bit-Systemen ist dies der am schnellsten verarbeitete Integer-Datentyp.	-2.147.483.648 bis 2.147.483.647
UInteger	System.UInt32	Speichert vorzeichenlose Integer-Werte mit 32 Bit (4 Byte) Breite.	0 bis 4.294.967.295
Long	System.Int64	Speichert vorzeichenbehaftete Integer-Werte mit 64 Bit (8 Byte) Breite.	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
ULong	System.UInt64	Speichert vorzeichenlose Integer-Werte mit 64 Bit (8 Byte) Breite.	0 bis 18.446.744.073.709.551.615
Single	System.Single	Speichert Fließkommazahlen mit einfacher Genauigkeit. Benötigt 4 Bytes zur Darstellung.	-3,4028235E+38 bis -1,401298E-45 für negative Werte; 1,401298E-45 bis 3,4028235E+38 für positive Werte
Double	System.Double	Speichert Fließkommazahlen mit doppelter Genauigkeit. Benötigt 8 Bytes zur Darstellung. HINWEIS: Dies ist der schnellste Datentyp zur Fließkommazahlberechnung, da er direkt an die Mathe-Einheit des Prozessors zur Berechnung delegiert wird.	1,79769313486231570E+308 bis -4,94065645841246544E-324 für negative Werte; 4,94065645841246544E-324 bis 1,79769313486231570E+308 für positive Werte
Decimal	System.Decimal	Speichert Fließkommazahlen im binärcodierten Dezimalformat. HINWEIS: Dies ist der langsamste Datentyp zur Fließkommazahlberechnung, seine besondere Speicherform schließt aber typische Computerrundungsfehler aus.	0 bis +/- 79.228.162.514.264.337.593.543.950.335 (+/-7,9...E+28) ohne Dezimalzeichen; 0 bis +/- 7,9228162514264337593543950335 mit 28 Stellen rechts vom Dezimalzeichen; kleinste Zahl ungleich 0 (null) ist +/-0,00000000000000000000000000000001 (+/-1E-28)
Boolean	System.Boolean	Speichert boolesche Zustände.	True oder False
Char	System.Char	Speichert ein Unicode-Zeichen mit einem Speicherbedarf von 2 Byte.	Ein Unicodezeichen im Bereich von 0-65535
Date	System.DateTime	Speichert einen Datumswert, bestehend aus Datum und Zeitanteil.	0:00 Uhr am 01.01.0001 bis 23:59:59 Uhr am 31.12.9999
Object	System.Object	Speichert die Referenz auf Daten bestimmten Typs im Managed Heap.	Belegt entweder 32-Bit (4 Bytes) auf 32-Bit-Betriebssystemen oder 64 Bit auf 64-Bit-Betriebssystemen und dient – wie jede andere Objektvariable (Referenztyp) – als Zeiger auf die eigentlichen Objektdaten im Managed Heap
String	System.String	Speichert Unicode-Zeichenfolgen.	Variable Länge. 0 bis ca. 2 Milliarden Unicode-Zeichen

Tabelle 10.1 Die primitiven Datentypen in .NET 2.0 bzw. VB.NET

HINWEIS Übrigens, eigentlich kann man die Regel aufstellen, dass jeder Typ, den der Codeeditor blau einfärbt, ein primitiver Datentyp ist. Das funktioniert natürlich nur so lange, wie Sie die Visual Basic-Äquivalente der primitiven .NET-Datentypen verwenden. Sie müssen das aber nicht, wie die in Fettschrift gesetzten Zeilen des folgenden Listings zeigen. Der folgende Code würde nämlich durchaus funktionieren:

```
Public Class Form1
    Private Sub btnVbUndFrameworkTypen_Click(ByVal sender As System.Object,
                                                ByVal e As System.EventArgs) Handles btnVbUndFrameworkTypen.Click
        'Ein im VB-deklarierte Datentyp unterscheidet...
        Dim locDatum1 As Date
        '...sich nicht von seiner Framework-Version!
        Dim locDatum2 As System.DateTime

        'Das ging in VB6 übrigens auch nicht!
        locDatum1 = #12/24/2008 6:30:00 PM#

        'Hier ist der Beweis: Keiner meckert.
        locDatum2 = locDatum1

        'Übrigens: {0} und {1} werden durch die darauffolgenden
        'Variableninhalte ersetzt. Hinter dem Doppelpunkt folgt
        'jeweils die Formatierungszeichenfolge.
        Debug.Print("Dieses Jahr am {0:dd.MM.yy}, also Heiligabend essen wir um {1:HH:mm}", _
                   locDatum2.Date,
                   locDatum2.TimeOfDay)
    End Sub
```

Deklaration von Variablen und Variablentypzeichen

Anders als in VB6 können Variablen gleichen Typs in einem Rutsch deklariert werden.

Darüber hinaus gibt es in VB.NET, wie in VB6, Variablentypzeichen. Die kennen Sie, wenn Sie, wie beispielsweise ich, seit Jahrzehnten Basic programmieren, von der ersten Stunde an. Solche Zeichen definieren, welchen Typs eine Variable sein soll, wenn Sie die Typanweisung (beispielsweise As Integer) nicht mit ausformulieren.

Betrachten Sie das folgende Codebeispiel, das diese Zusammenhänge genauer verdeutlichen soll:

```
Public Class Form1
    Private Sub btnVariablentypzeichen_Click(ByVal sender As System.Object,
                                              ByVal EArgs As System.EventArgs) Handles btnVariablentypzeichen.Click

        'Beide sind Integer - anders als in VB6!
        Dim locInt1, locInt2 As Integer

        'Auch das geht...
        Dim locDate1, locDate2 As Date, locLong1, locLong2 As Long

        'Und das hier auch:
        Dim a%, b&, c@, d!, e#, f$
        f$ = "Muss was drin sein!"
```

```
'Alle Variablentypen in Klartext ausgeben:  
Debug.Print("locInt1 ist: " & locInt1.GetType.ToString)  
Debug.Print("locInt2 ist: " & locInt2.GetType.ToString)  
Debug.Print("locDate1 ist: " & locDate1.GetType.ToString)  
Debug.Print("locDate2 ist: " & locDate2.GetType.ToString)  
Debug.Print("locLong1 ist: " & locLong1.GetType.ToString)  
Debug.Print("locLong2 ist: " & locLong2.GetType.ToString)  
  
Debug.Print("a ist: " & a.GetType.ToString)  
Debug.Print("b ist: " & b.GetType.ToString)  
Debug.Print("c ist: " & c.GetType.ToString)  
Debug.Print("d ist: " & d.GetType.ToString)  
Debug.Print("e ist: " & e.GetType.ToString)  
Debug.Print("f ist: " & f.GetType.ToString)  
End Sub  
End Class
```

Wenn Sie dieses Beispiel laufen lassen, wird folgende Ausgabe im Ausgabefenster angezeigt:

```
locInt1 ist: System.Int32  
locInt2 ist: System.Int32  
locDate1 ist: System.DateTime  
locDate2 ist: System.DateTime  
locLong1 ist: System.Int64  
locLong2 ist: System.Int64  
a ist: System.Int32  
b ist: System.Int64  
c ist: System.Decimal  
d ist: System.Single  
e ist: System.Double  
f ist: System.String
```

Im Grunde genommen verdeutlicht dieses Beispiel dreierlei:

- Zum Ersten sehen Sie abermals, dass die .NET-Datentypen und die Visual Basic-Datentypen absolut dasselbe sind.
- Zweitens sehen Sie, dass – anders als in VB6 – das gleichzeitige Deklarieren von Variablen vom selben Typ möglich ist. In VB wäre locInt1 automatisch vom vorgegebenen Standarddatentyp oder – falls dieser mit DefXXX nicht festgelegt worden wäre – vom Typ Variant gewesen.

HINWEIS Standarddatentypdefinitionen mit DefXXX, die beim Weglassen des Typqualifizierers bei einer Variablendeclaration automatisch den zu verwendenden Datentyp festgelegt haben (also beispielsweise DefInt A-Z, um alle Variable, die nicht explizit als bestimmter Typ ausgewiesen wurden, automatisch als Integer zu deklarieren), gibt es in keinem der Visual Basic.NET-Derivate mehr.

- Und drittens: Variablentypzeichen sind optional. Sie können sie zwar verwenden, aber Sie sollten sie nicht verwenden. Dabei spielt es im Übrigen keine Rolle, auf welche Weise eine Variablen zu »ihrem Typ geworden ist« – nur durch das Typzeichen oder durch die ausformulierte Angabe des Typs mit as VarType.

TIPP Eines ist aber klar: Ihr Code ist in jedem Fall einfacher lesbar, wenn Sie die Variablenarten explizit angeben. Typzeichen sind meiner Meinung nach Relikte aus alter Zeit und eigentlich überflüssig. Im Übrigen gibt es sie bis auf eine kleine Ausnahme², in keiner anderen »Erwachsenenprogrammiersprache« (wie C, C++, C# oder Java), und wir wollen ja schließlich alle, dass Basic endlich gemeinhin bescheinigt werden kann, bei den Großen mitspielen zu dürfen. Und wenn Sie sich in einer umfangreichen Prozedur einmal nicht daran erinnern können, von welchem Typ eine bestimmte Variable war, fahren Sie einfach mit dem Mauszeiger im Editor darauf. Der gibt Ihnen mit einem Tooltip sofort darüber Auskunft.

```
'Und das hier auch:
Dim a%, b&, c$, d!, e#, f$
f$ = "Muss wDim c As Decimal sein!"'
```

Abbildung 10.2 Ein simples Darauffahren mit dem Mauszeiger zur Entwurfszeit genügt, um den Typ einer Variablen zu erfahren

Typsicherheit und Typliterale zur Typdefinition von Konstanten

Typliterale – meiner Meinung nach sollten diese »Typerzwingungsliterale für Konstanten« heißen – dienen dazu, eine Konstante dazu zu zwingen, ein bestimmter Typ zu sein.

Sie kennen das auch von Visual Basic 6.0. Wenn Sie eine numerische Konstante dazu zwingen wollen, eine Zeichenkette zu sein, damit Sie diese ohne sie umwandeln zu müssen an eine String-Variablen zuweisen können, dann hüllen Sie sie in Anführungszeichen ein. Das Anführungszeichen (oder *die* Anführungszeichen in diesem Fall) dienen also dazu, aus einer Zahl eine Zeichenkette zu machen.

In Visual Basic .NET (also in allen Versionen seit VB2002) beschränken sich Typliterale nicht nur auf Strings – es gibt sie auch für andere Datentypen. Eines haben Sie bereits im ersten Listing dieses Kapitels kennen gelernt – das Typliteral für Datumswerte »#«. Neben dem Anführungszeichen bei Strings, ist dies das einzige weitere, das eine Konstante quasi einklammert.

Andere Typliterale werden der Konstante einfach nachgestellt. In einigen Fällen bestehen diese übrigens nicht nur aus einem Buchstaben sondern aus zweien.

Die folgende Tabelle zeigt, wie Sie Konstanten mit Typliteralen definieren und gibt Beispiele. Sollte es Variablenarten für einen bestimmten Typ der Tabelle geben, finden Sie diese ebenfalls in der Tabelle angegeben.

Typename	Typzeichen	Typliteral	Beispiel
Byte	–	–	Dim var As Byte = 128
SByte	–	–	Dim var As SByte = -5
Short	–	S	Dim var As Short = -32700S
UShort	–	US	Dim var As UShort = 65000US
Integer	%	I	Dim var% = -123I oder Dim var As Integer = -123I ►

² In C# 2005 gibt es die Möglichkeit, einen primitiven Datentyp auf sein nullbares Pendant durch ein Typzeichen festzulegen.

Typename	Typzeichen	Typliteral	Beispiel
UInteger	-	UI	Dim var As UInteger = 123UI
Long	&	L	Dim var& = -123123L oder Dim var As Long = -123123L
ULong	-	UL	Dim var As ULong = 123123UL
Single	!	F	Dim var! = 123.4F oder Dim var As Single = 123.4F
Double	#	R	Dim var# = 123.456789R oder Dim var As Double = 123.456789R
Decimal	@	D	Dim var@ = 123.456789123D oder Dim var As Decimal = 123.456789123D
Boolean	-	-	Dim var As Boolean = True
Char	-	C	Dim var As Char = "A"C
Date	-	#dd/MM/yyyy HH:mm:ss# oder #dd/mm/yyyy hh:mm:ss am/pm#	Dim var As Date = #12/24/2008 04:30:15 PM#
Object	-	-	In einer Variable vom Typ Object kann jeder beliebige Typ gekapselt (»boxed«) werden oder mit dieser referenziert werden.
String	\$	"Zeichenkette"	Dim var\$ = "Zeichenkette" oder Dim var As String = "Zeichenkette"

Tabelle 10.2 Typliterale und Variablentypzeichen der primitiven Datentypen ab Visual Basic 2005

HINWEIS Mit der 2008er Version kennt Visual Basic auch Nullables, die auf Basis von Wertetypen definiert werden können, und die zusätzlich zum eigentlichen Datenwert auch den Zustand »nichts« in Form von Nothing (Null in C#) speichern können. Nullables werden mit dem Fragezeichen als Typzeichen gekennzeichnet und bilden den einzigen Datentyp, beim dem Typzeichen noch »moralisch statthaft« sind; selbst in C# sind Nullables die einzigen Variablentypen, die von einem Typzeichen Gebrauch machen. Mehr zu Nullables finden Sie im nächsten Kapitel.

Vertiefen wir das in der Tabelle Gezeigte ein wenig, damit deutlicher wird, worum es geht.

Betrachten Sie sich den folgenden Visual Basic 6.0-Code:

Achtung! VB6-Code!

```
Dim einString As String
einString = "1.23"

Dim einAndererString As String
einAndererString = 1.23

Debug.Print (einString & ", " & einAndererString)
```

Wenn Sie diesen kleinen Codeausschnitt laufen ließen, würde er folgendes Ergebnis produzieren:

1.23, 1,23

Warum, ist jedem erfahrenen VB6-Programmierer klar:

einString hat seinen Wert durch eine direkte Zuweisung einer Konstanten bekommen, die als sein ureigener Typ zu erkennen war: Die Anführungszeichen sorgen dafür, dass die Konstante als Zeichenkette behandelt wird, und da es sich eben um Typengleichheit handelt, enthält die Stringvariable exakt den angegebenen Typ.

Die Konstante, die an einAndererString zugewiesen wird, ist nicht vom Typ String, denn sie ist nicht in Anführungszeichen eingefasst. Bei ihr handelt es sich um eine Fließkommakonstante. Also muss zunächst eine implizite Typkonvertierung bei der Zuweisung stattfinden – die Fließkommazahl wird in eine Zeichenkette umgewandelt. Und da die meisten von uns wohl auf einem deutschen Betriebssystem arbeiten, wird die deutsche Kultur bei der Umwandlung berücksichtigt: Anders als die Amerikaner trennen wir Nachkoma von Vorkommastellen mit einem Komma und nicht mit einem Punkt. So wird aus der Konstante 1.23 die Zeichenkette 1,23.

Sie sehen also, dass die Bestimmung von Typen bei Konstanten schon im alten Visual Basic durchaus wichtig sein konnte.

.NET ist Typsicher

In .NET ist das noch viel, viel wichtiger, denn .NET ist grundsätzlich typsicher. Typsicher bedeutet, dass Sie unterschiedliche Typen bei Zuweisungen nicht »einfach so« mischen und damit die folgende Zeile eigentlich gar nicht kompiliert, sondern mit einer Fehlermeldung quittiert bekämen (sofern Sie sich meine Ratschläge zur Konfiguration von Option Strict zu Eigen gemacht haben).

```
Dim einAndererString As String
einAndererString = 1.23
```

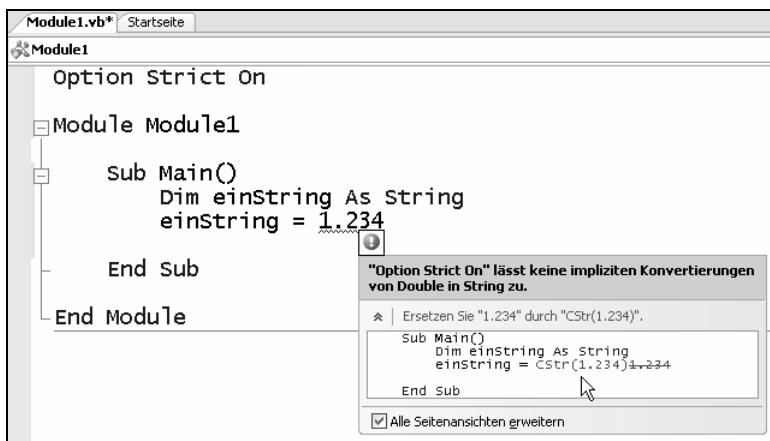


Abbildung 10.3 Typsicherheit unter .NET verlangt, dass implizit nur gleiche oder sichere Typen einander zugewiesen werden können

Die Autokorrektur für intelligentes Komplizieren zeigt Ihnen, was das Problem ist: Implizit, also ohne weiteres Zutun (oder anders gesagt »von selbst«) können Sie unter .NET im Grunde genommen nur gleiche Typen zuweisen, oder Typen die zwar unterschiedlich sind, bei denen eine implizite Konvertierung also auf jeden Fall sicher ist.

Und was heißt »auf jeden Fall sicher«?

Bei den Strings haben wir es bereits schon im oben stehenden VB6-Beispiel erlebt. Diese implizite Konvertierung war nicht sicher. Auf einem amerikanischen System hätte man ein anderes Resultat als auf einem deutschen System gehabt. Die implizite Konvertierung ist also nicht sicher, weil es offensichtlich Einflussgrößen gibt (hier: die Kultureinstellungen), die das Ergebnis beeinflussen können.

Anders ist es zum Beispiel, wenn Sie einen Integer-Typ einem Long-Typ zuweisen:

```
Dim einLong As Long
Dim einInt As Integer = 10000
einLong = einInt
```

Kein .NET-Compiler würde hier was zu meckern haben, denn bei dieser Typkonvertierung kann nichts schief gehen. Integer deckt in jedem Fall einen viel kleineren Zahlenbereich als Long ab, und deswegen ist hier eine implizite Konvertierung gefahrlos möglich.

Umgekehrt sieht es schon wieder anders aus, wie die folgende Abbildung zeigt:

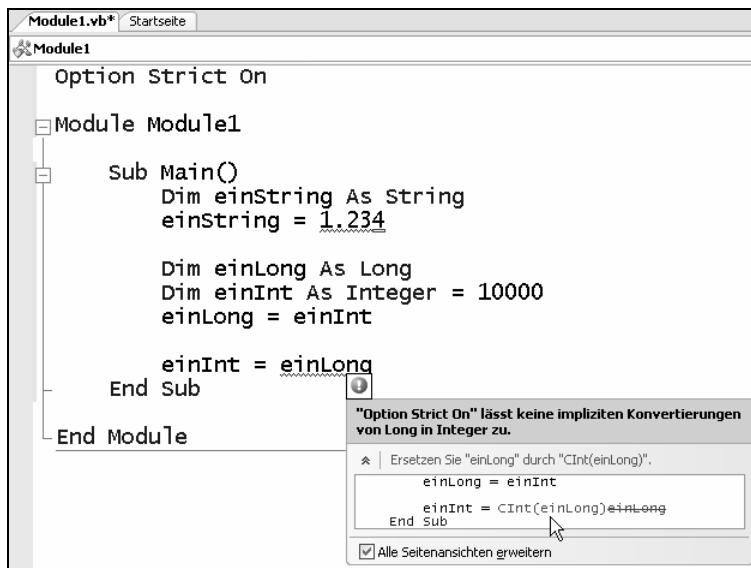


Abbildung 10.4 Während »kleinere« Typen sicher in »größere« konvertiert werden können, ist das umgekehrt nicht typsicher und deswegen auch nicht implizit gestattet

Bei einer Konvertierung von Long zu Integer können Informationen verloren gehen, deswegen stuft .NET diese Art der Konvertierung als nicht typsicher ein. Natürlich können Sie eine derartige Konvertierung vornehmen, nur eben nicht implizit. Sie müssen .NET explizit mitteilen, dass Sie sich sozusagen »der Gefahr bewusst sind«, und entsprechende Aktionen vornehmen, um die Konvertierung vornehmen zu können. Die Autokorrektur für intelligentes Kompilieren macht Ihnen auch direkt einen entsprechenden Vorschlag: »Setzen Sie CInt (etwa: Convert to Integer – in Integer konvertieren) ein«, schlägt sie vor, »und zeigen Sie dem Compiler damit, dass Sie sich der Konvertierung von Long in Integer bewusst sind.«

Und vielleicht ahnen Sie jetzt auch schon, wozu die Typliterale in VB.NET dienen. Typsicherheit muss auch für Konstanten gelten. Und damit muss es auch einen Weg geben, eine Konstante dazu zu bringen, ein bestimmter Typ zu sein. Und eben genau das geschieht mit Typliteralen. Funktioniert das Folgende implizit?

```
Dim einChar As Char
Dim einString As String = "Hallo"
einChar = einString
```

Nein. Denn dabei würde »allo« auf der Strecke bleiben. Ein Char-Typ kann nur ein einzelnes Unicode-Zeichen und keinen ganzen String aufnehmen. Auch wenn dieser Codeausschnitt folgendermaßen lauten würde:

```
Dim einChar As Char = "H"
```

String ist String, und Char ist Char. Nur in Anführungszeichen definieren Sie eben einen String, egal wie viele Zeichen der hat. Und selbst wenn es passen würde (so wie in dieser Zeile), wäre die Typsicherheit dennoch verletzt. Sie müssen aus dem »H« einen Char-Typ machen, und das erreichen Sie durch das Hintenanstellen eines Typliterals, wie die folgende Abbildung zeigt:

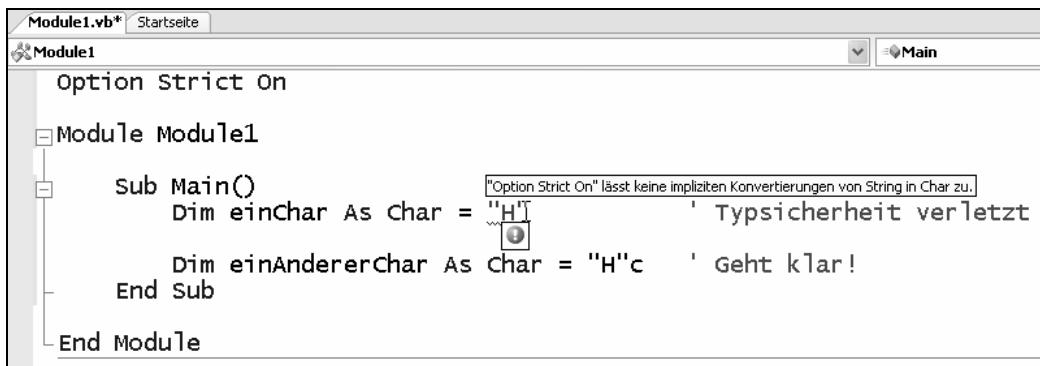


Abbildung 10.5 Mit Typliteralen zwingen Sie Konstanten in einen bestimmten Typ

Das gilt natürlich nicht nur für String und Char, sondern auch für die unterschiedlichen numerischen Datentypen.

Der folgende Codeausschnitt zeigt ein paar Beispiele, die demonstrieren, wann der Einsatz von Typliteralen Sinn macht:

```
'Fehler: Von Double nach Decimal geht nicht implizit.
Dim decimal1 As Decimal = 1.2312312312312
'Hier ist es ein Decimal durch das D am Ende
Dim decimal2 As Decimal = 1.2312312312312D

Dim decimal3 As Decimal = 9223372036854775807    ' OK.
' Überlauf - Ohne Typliteral ist es implizit ein
' Long-Wert, und in diesem Fall liegt er außerhalb seines Wertebereichs.
Dim decimal4 As Decimal = 9223372036854775808
' Mit dem Typliteral "D" wird Decimal als Konstante erzwungen, und es passt.
Dim decimal5 As Decimal = 9223372036854775808D  ' Kein Überlauf.

'Fehler: Ohne Typliteral ist's wieder implizit ein Long,
'aber außerhalb des Long-Wertebereichs.
Dim ushort1 As ULong = 9223372036854775808
```

```
' Mit Typliteral wird Decimal als Konstante erzwungen, und es passt.  
Dim ushort2 As Decimal = 9223372036854775808UL ' Kein Überlauf mehr.
```

Deklaration und Definition von Variablen »in einem Rutsch«

Vielleicht haben Sie es beim Betrachten der vorherigen Beispiele schon gemerkt: Anders als in VB6 können Variablen in VB.NET (und auch in den Vorgängerversionen) in einer Zeile nicht nur deklariert, sondern auch definiert werden.

Ob Sie also schreiben

```
Dim einInteger As Integer  
einInteger = 10
```

oder

```
Dim einInteger As Integer = 10
```

bleibt sich absolut gleich.

Das gilt in VB.NET auch für Instanziierungen³ von Objekten, wie das folgende Beispiel zeigt.

```
Dim einObject As Object  
Dim einObject = New Object
```

Bleibt sich gleich mit

```
Dim einObject As New Object
```

WICHTIG In VB6 gab es einen großen Unterschied zwischen den beiden Versionen der Objektinstanzierung, auf den der übriges Abschnitt eingeht.

Lokaler Typrückschluss

Ab Visual Basic 2008 reicht es, bei lokalen Variablen eine Zuweisung an eine Variable während ihrer Deklaration zu machen, um ihren Typ zu definieren. Diese Vorgehensweise wird *lokaler Typrückschluss* genannt. Ein Beispiel:

```
Dim bInValue = True
```

³ Falls Ihnen der Begriff »Instanzierung« im Moment nichts sagt: Sie kennen diesen Vorgang beispielsweise aus VB6, wenn Sie eine Collection für die Aufnahme von anderen Variablen oder Objekten benötigten.

Wenn Sie einer Variable, wie hier im Beispiel, den Wert True zuweisen und dazu noch Typsicherheit definiert ist, so muss es sich bei der Variablen einfach um den booleschen Datentyp handeln, und deswegen können Sie die Formulierung As Boolean an dieser Stelle auch auslassen.

Lokaler Typrückschluss wurde in Visual Basic 2008 notwendig, damit LINQ-Abfragen (ab Kapitel 31) auch anonyme Typen deklarieren können – bequemer beim Entwickeln ist das natürlich auch für Typen, die es gibt (die also nicht anonym sind), alle Male!

Lokaler Typrückschluss funktioniert aus diesem Grund auch nur maximal auf Prozedurenebene – Sie können nämlich keine anonymen Typen auf Klassenebene verwenden; daraus ergibt sich auch der Name lokaler Typrückschluss – im englischen übrigens *local type inference*, weswegen Sie dieses Verhalten am Anfang einer Codedatei auch mit

Option Infer On|Off

ein- und ausschalten können.

HINWEIS Lokaler Typrückschluss ist bei Projekten, die Sie von einer früheren Visual Basic-Version zu Visual Basic 2008 migriert haben, nicht aktiviert, Sie müssen es über die Projekteigenschaft explizit aktivieren. Weitere Beispiele zu diesem Thema, und wie Sie den lokalen Typrückschluss innerhalb Ihrer Projekte global steuern können, finden Sie im entsprechenden Abschnitt des nächsten Kapitels.

Vorsicht: New und New können zweierlei in VB6 und VB.NET sein!

Grundsätzlich gibt es zwei Möglichkeiten, ein neues Objekt in Visual Basic (sowohl in 6.0 als auch in VB.NET) zu erstellen. Typischer VB6-Code, der beispielsweise eine Auflistung vom Typ Collection instanziert, könnte folgendermaßen ausschauen, und das haben viele von der Riege der alten VB6-Programmierer auch so gemacht:

Achtung! VB6-Code!

```
Dim locObj As Collection  
Set locObj = New Collection  
Debug.Print (locObj Is Nothing)  
  
Set locObj = Nothing  
Debug.Print (locObj Is Nothing)
```

Wenn Sie diesen Code unter VB6 laufen lassen, zeigt Ihnen das Direktfenster erwartungsgemäß Folgendes an:

Falsch
Wahr

Denn: Eine neue Collection wird in den ersten beiden Zeilen definiert; die Abfrage auf Nothing ist damit in der ersten Ausgabe Falsch. Anschließend wird das Objekt auf Nothing zurückgesetzt, und die folgende Ausgabe gibt wahrheitsgemäß Wahr aus.

Doch schauen Sie, was passiert, wenn Sie dieses kleine Programm folgendermaßen abändern:

Achtung! VB6-Code!

```
Dim locObj As New Collection  
'Set locObj = New Collection  
Debug.Print (locObj Is Nothing)  
Set locObj = Nothing  
Debug.Print (locObj Is Nothing)
```

Nun erscheint folgendes Ergebnis im Direktfenster:

```
Falsch  
Falsch
```

Warum? Weil es anders als unter .NET in VB6 *einen Unterschied* macht, wie Sie eine neue Instanz einer Klasse erstellen. Die Syntax sieht zwar aus wie das Deklarieren und Instanziieren in einer Zeile, das Sie nun aus VB.NET kennen; in VB 6.0 können Sie dies aber gar nicht. Die VB 6.0 Syntax

```
Dim locObj As New IrgendEinObjekt
```

deklariert eine *sich automatisch instanzierende Variable* locObj vom Typ IrgendEinObjekt. Der Compiler von VB 6.0 setzt dann vor jeden Methoden- oder Eigenschaftszugriff in Ihrem Code eine Überprüfung, ob das Objekt vielleicht Nothing ist und wenn ja, instanziert er es *dann* mit New für Sie. Dies ist auch der Grund, warum von dieser Syntax in VB 6.0 abzuraten war: Erstens geht die Überprüfung auf Nothing dann per Definition immer false aus (vor der Überprüfung wird das Objekt eben automatisch instanziert) und zweitens ist die vor jedem Zugriff nötige Überprüfung nicht gerade performant.

WICHTIG In VB.NET ist die Syntax Dim locObj As New IrgendEinObject dagegen eben *nichts* anderes als das Deklarieren (Dim) und das sofortige Instanziieren (New) in einer Zeile.

Überläufe bei Fließkommazahlen und nicht definierte Zahlenwerte

Überläufe bei Fließkommazahlen führten bei VB6 zu Laufzeitfehlern. Zu den Überläufen zählt – auch wenn das mathematisch nicht ganz korrekt ist – in diesem Zusammenhang eine Division durch 0. Der folgende Code unter VB6 hätte also zur Laufzeit gleich zwei Fehler ausgelöst:

Achtung! VB6-Code!

```
Dim einDouble As Double  
einDouble = 9.7E+307  
'Fehler: Überlauf!  
einDouble = einDouble * 2  
  
einDouble = 1  
'Fehler: Division durch 0!  
einDouble = einDouble / 0
```

VB.NET verhält sich hier anders. Wenn eine Fließkommazahl einen bestimmten Wert nicht mehr annehmen kann, dann wird ihr ein besonderer Wert zugewiesen, wie das folgende Beispiel zeigt:

```
Dim einDouble As Double
einDouble = 9.7E+307
einDouble = einDouble * 2
Debug.Print("einDouble hat den Wert:" & einDouble)

einDouble = 1
einDouble = einDouble / 0
Debug.Print("einDouble hat den Wert:" & einDouble)
```

Ließen Sie diesen Codeausschnitt laufen, würde er überhaupt keinen Fehler produzieren. Stattdessen zeigt er folgendes Ergebnis im Ausgabefenster an:

```
einDouble hat den Wert:+unendlich
einDouble hat den Wert:+unendlich
```

Dieser »Wert« tritt nicht nur unter bestimmten Umständen (Überlauf, Division durch 0) während des Programmablaufs auf. Sie können ihn auch manuell zuweisen oder ihn abfragen, wie das folgende Beispiel zeigt:

```
'Auf "unendlich" (Überlauf) prüfen
If Double.IsInfinity(einDouble) Then
    Debug.Print("einDouble ist unendlich!")
End If

'Auf "minus unendlich" (negativen Überlauf) prüfen
If Double.IsNegativeInfinity(einDouble) Then
    Debug.Print("einDouble ist minus unendlich!")
End If

'Gezielt auf "plus unendlich" (positiven Überlauf) prüfen
If Double.IsPositiveInfinity(einDouble) Then
    Debug.Print("einDouble ist plus unendlich!")
End If

'"plus unendlich" zuweisen
einDouble = Double.PositiveInfinity

'"minus unendlich" zuweisen
einDouble = Double.NegativeInfinity
```

Und noch einen Sonderfall decken die primitiven Fließkommatypen Single, Double und Decimal ab: Die Division von 0 und 0, die mathematisch nicht definiert ist und *keine gültige Zahl* ergibt:

```
'Sonderfall: 0/0 ist mathematisch nicht definiert und ergibt "Not a Number"
einDouble = 0
einDouble = einDouble / 0
If Double.IsNaN(einDouble) Then
    Debug.Print("einDouble ist keine Zahl!")
End If
```

Ließen Sie diesen Code laufen, würde der Text in der If-Abfrage ausgegeben werden.

WICHTIG Überprüfungen auf diese Sonderwerte lassen sich nur durch die Eigenschaften testen, die (als sogenannte statische⁴ Funktionen bzw. statische Eigenschaften) direkt an den Typen »hängen«. Zwar können Sie beispielsweise mit der Konstante der Fließkommatypen NaN den Wert »keine gültige Zahl« einer Variablen zuweisen; diese Konstante eignet sich allerdings nicht, auf diesen Zustand (»Wert«) zu testen, wie das folgende Beispiel zeigt:

```
Dim einDouble As Double

'Sonderfall: 0/0 ist mathematisch nicht definiert und ergibt "Not a Number"
einDouble = 0
einDouble = einDouble / 0

'Der Text sollte erwartungsgemäß ausgegeben werden,
'wird er aber nicht!
If einDouble = Double.NaN Then
    Debug.Print("Test 1:einDouble ist keine Zahl!")
End If

'Nur so kann der Test erfolgen!
If Double.IsNaN(einDouble) Then
    Debug.Print("Test 2:einDouble ist keine Zahl!")
End If
```

In diesem Beispiel würde nur der zweite Text ausgegeben.

Alles ist ein Objekt oder »let Set be«

Wenn Sie den folgenden VB.NET-Code betrachten,

```
Dim einString As String
einString = "Ruprecht Dröge"

'Position des Leerzeichens ermitteln
Dim spacePos As Integer = einString.IndexOf(" ")

'Nur den Vornamen ermittelt - das ging "früher" mit Mid$
einString = einString.Substring(0, spacePos)
Debug.Print(einString)
```

so können Sie anhand der Anweisung einString.Substring(0, spacePos) schon vermuten, dass es sich bei einString nicht um eine »bloße Grunddatentypenvariable« handelt, wie man das von VB6 kennt, sondern dass das, was dort verarbeitet wird, eigentlich eher wie ein Objekt aussieht – denn wie bei einem richtigen Objekt verfügt eine String-Variablen offensichtlich über Methoden und Eigenschaften – etwa, wie hier zu sehen, über eine Substring-Methode.

⁴ Statisch deswegen, weil Sie keine definierte Variable brauchen, um die Funktion bzw. Eigenschaft verwenden zu können, sondern Ihnen diese direkt über den Typennamen immer (statisch) zur Verfügung steht.

Die Wahrheit ist: In .NET »ist alles ein Objekt« oder zumindest von diesem abgeleitet. Buchstäblich, denn Object, von dem schon die Rede war, ist der grundlegendste aller Datentypen.

Reine Grunddatentypen wie in Visual Basic 6.0 gibt es im Framework.NET nur noch per Definition. In .NET heißen diese dann auch »primitive Datentypen«, und wie alles in .NET handelt es sich im Grunde genommen auch bei ihnen um Objekte. Und da man, hätte man das alte Konzept zur Verarbeitung von Objekten weiterverfolgt, bei der Zuweisung von Inhalten *immer* das Schlüsselwort »Set« hätte verwenden müssen, haben die VB.NET-Entwickler es schlicht hinausgeworfen.

Diese Tatsache hat weitere (und wie ich finde sehr angenehme) Konsequenzen: Auch die primitiven Datentypen (Integer, Double, String und wie sie alle heißen) haben Eigenschaften und Methoden. Das String-Objekt beispielsweise lässt sich zwar nach wie vor noch mit Left, Mid und Right buchstäblich auseinander nehmen, aber das sollten Sie damit nicht mehr machen. Das String-Objekt in .NET bietet viel elegantere Möglichkeiten für seine Verarbeitung (die Methode SubString ist beispielsweise das Mid-Pendant), und wenn Softwareentwickler, die in anderen .NET-Sprachen entwickeln, Ihre Programme lesen, dann helfen Sie ihnen, indem Sie die sprachübergreifenden Methoden der Objekte verwenden und nicht die proprietären von Visual Basic.

Kapitel 7 zeigt im entsprechenden Abschnitt anhand vieler Beispiele, wie Sie die »neuen« String-Methoden idealerweise einsetzen. Der Migrationsassistent von Visual Basic 2008 ist leider nicht in der Lage, die veraltete VB6-Formulierung in die neue .NET-orientierte umzuwandeln.

TIPP Wenn Sie schnell herausfinden wollen, welche Methoden und Eigenschaften Ihnen die primitiven Datentypen anbieten, deklarieren Sie einfach irgendwo in einem einfachen Projekt einen primitiven Datentyp Ihrer Wahl, schreiben Sie den Variablen eine Zeile darunter, und recherchieren Sie mithilfe der Vervollständigungsliste, welche Methoden und Eigenschaften er Ihnen bietet.

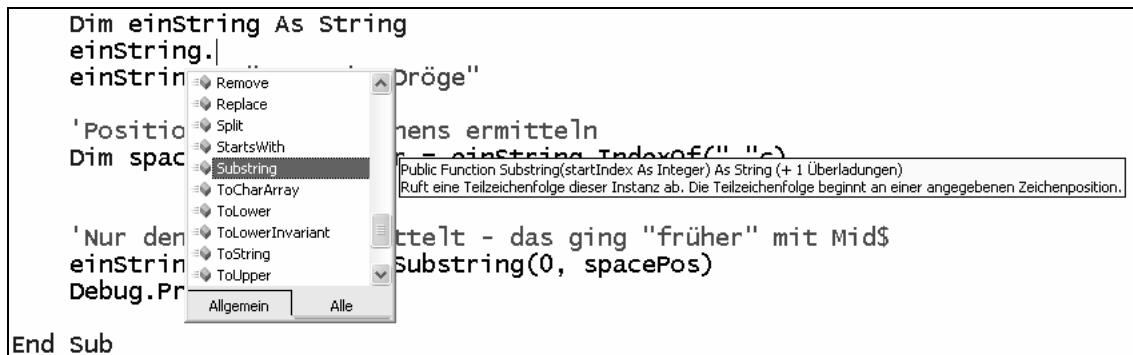


Abbildung 10.6 Wenn Sie wissen wollen, welche Methoden und Eigenschaften ein primitiver Variabtentyp zur Verfügung stellt – deklarieren Sie ihn, und benutzen Sie IntelliSense für eine Funktionsübersicht

Gezieltes Freigeben von Objekten mit Using

Normalerweise sorgt die in .NET eingebaute »Müllabfuhr« (der Garbage Collector) dafür, dass Objekte, die Sie verwendet haben, die aber nicht mehr benötigt werden, speichertechnisch entsorgt werden. Sie müssen sich also nicht selber darum kümmern, Speicher freizugeben, wenn Sie ein Objekt nicht mehr benötigen.

Wenn Sie ein Objekt aus einer Klasse mit `New` instanzieren, reservieren Sie Speicher in einem bestimmten Teil des Hauptspeichers – im so genannten Managed Heap – in dem es seine Daten ablegt. Wird das Objekt nicht mehr benötigt, und das ist beispielsweise dann der Fall, wenn es innerhalb einer Prozedur deklariert wird, die Prozedur dann aber verlassen wird, sorgt der Garbage Collector irgendwann dafür, dass der Speicher wieder freigegeben wird.

In einigen Fällen kann es aber erforderlich sein, ein Objekt gezielt zur Freigabe zu markieren, dem Framework also mitzuteilen: »Dieses Objekt benötige ich nicht mehr, du kannst es beim nächsten Mal, wenn du zur Mülltonne gehst, bitte mitnehmen.« Dieser Fall tritt dann ein, wenn nicht nur Speicher für das Objekt auf dem Managed Heap (also im Arbeitsspeicher) für das Speichern seiner Daten reserviert ist, sondern wenn auch noch andere Ressourcen des Betriebssystems reserviert werden müssen – zum Beispiel Dateikanäle beim Lesen aus oder Schreiben in Dateien. In diesem Fall ergibt es Sinn, dem Objekt mitzuteilen: »Ich brauch dich jetzt nicht mehr, bitte gib alle Systemressourcen, die du vielleicht belegt hast, wieder frei. Und wo wir schon mal dabei sind: Teile dem Müllmann mit, wenn er das nächste Mal vorbeikommt, dass er dich gleich mitnehmen soll.« Das klingt zwar herzlos, ist aber pragmatisch.

Objekte, die es gestatten, sich selbst auf die Entsorgung hinzuweisen, implementieren eine Methode namens `Dispose` (etwa: *entsorgen*). Rufen Sie `Dispose` eines Objektes auf, gibt es alle belegten Systemressourcen frei und signalisiert dem Garbage Collector gleichzeitig, dass er es »mitnehmen« kann. Eine typische Vorgehensweise zur Anwendung sieht dann so aus, wie es das folgende Beispillisting demonstriert, das eine Textdatei schreibt:

```
Dim locSw As StreamWriter
Try
    locSw = New StreamWriter("C:\Textdatei1.txt")
    Try
        locSw.WriteLine("Erste Textzeile")
        locSw.WriteLine("Zweite Textzeile")
        'Schreibpuffer leeren
        locSw.Flush()
    Catch ex As Exception
        Debug.Print("Fehler beim Schreiben der Datei!")
    Finally
        'Alle Systemressourcen wieder freigeben
        locSw.Dispose()
    End Try
Catch ex As Exception
    Debug.Print("Fehler beim Öffnen der Datei!")
End Try
```

Weil das Freigeben der belegten Systemressourcen in diesem Fall so wichtig ist, befindet sich der Aufruf der `Dispose`-Methode in diesem Fall im `Finally`-Teil des `Try/Catch`-Blocks. Dadurch wird sichergestellt, dass die Systemressourcen des Objektes in jedem Fall freigegeben werden, egal ob es innerhalb des `Try`-Teils zu einer Ausnahme kam oder nicht.

HINWEIS Zwei geschachtelte `Try`-Blöcke sind hier übrigens sinnvoll, weil beim Öffnen der Datei andere Ausnahmen auftreten können, als beim Schreiben. Die Systemressourcen selbst werden aber nur nach *erfolgreichem* Öffnen des `StreamWriter`-Objektes von diesem belegt, und müssen deswegen auch nur in diesem Fall (innerer `Try/Catch`-Block) mit `Dispose` wieder freigegeben werden.

`Using` erlaubt nun, die Lebensdauer eines Objektes, das die `Dispose`-Methodik implementiert, gezielt zu steuern. Das folgende Beispiellisting entspricht dem obigen Beispiel im Detail – `Using` vereinfacht aber den Umgang mit dem Objekt, und der Code wird leichter lesbar:

```
'Schreiben einer Datei - mit Using wird die Lebensdauer von locSw2 gesteuert
Try
    'Alternativ ginge auch die Weiterverwendung von locSw:
    'locSw = New StreamWriter("C:\Textdatei2.txt")
    'Using locSw
    Using locSw2 As New StreamWriter("C:\Textdatei2.txt")
        Try
            locSw2.WriteLine("Erste Textzeile")
            locSw2.WriteLine("Zweite Textzeile")
            'Schreibpuffer leeren
            locSw2.Flush()
        Catch ex As Exception
            Debug.Print("Fehler beim Schreiben der Datei!")
        End Try
        'Hier wird automatisch locSw.Dispose durchgeführt
        End Using
    Catch ex As Exception
        Debug.Print("Fehler beim Öffnen der Datei!")
    End Try
End Try
```

Sie sehen hier, dass der `Finally`-Teil des inneren `Try/Catch`-Blocks überflüssig geworden ist – denn am `End Using` kommt das Programm nicht vorbei. Selbst, wenn es versuchte, beispielsweise mit `Return` aus dem `Using`-Block herauszuspringen, würde die `Dispose`-Methode des Objektes dennoch vor dem eigentlichen Verlassen der Prozedur noch ausgeführt.

Mehr zum Thema `Dispose` und dem Garbage Collector erfahren Sie übrigens in Kapitel 18.

TIPP `Using` wird besonders häufig bei Verbindungen zum SQL Server eingesetzt. Sobald Sie eine Verbindung zum SQL Server benötigen, verwenden Sie das dazu benötigte `Connection`-Objekt in einer `Using`-Anweisung. Führen Sie alle Operationen durch, für die Sie eine offene Verbindung benötigen. Am Ende aller Operationen schließen Sie die SQL Server-Verbindung mit `End Using`. Auf diese Weise können Sie auch gefahrlos einen solchen Codeblock verlassen, ohne explizit dafür Sorge getragen haben zu müssen, die Verbindung zum SQL Server wieder zu schließen. Zusätzlich zum Schließen der Verbindung, was auch die Methode `Close()` leisten könnte, wird im `Dispose()` auch der *SharedLock*⁵ auf der benutzten Datenbank entfernt. Ihre Anwendung wird dadurch robuster, denn Sie vermeiden doppelt oder mehrfach offene Verbindungen, weil Sie es etwa schlicht vergessen haben, eine offene Verbindung wieder zu schließen, wenn Sie einen Codeblock, in dem SQL-Operationen verarbeitet wurden, vorzeitig verlassen haben.

⁵ Damit die Maxime ACID beim SQL Server (*Atomic, Consistent, Isolated, Durable* – in etwa: *Ganzheitlich, Konsequent, Isoliert* und *Verlässlich*) nicht verletzt wird, dürfen sich natürlich keine Daten ändern, während er Daten in einer Verbindung zum Beispiel in einer `SELECT`-Abfrage zur Verfügung stellt. Ein *Shared Lock* erlaubt also, dass mehrere Verbindungen gleichzeitig lesend zugreifen, dass aber keine Daten geändert werden können.

Direktdeklaration von Variablen in For-Schleifen

Um in Visual Basic 6.0 Schleifenvariablen zu verwenden, mussten Sie diese umständlich zuerst deklarieren und konnten diese erst danach verwenden:

Achtung! VB6-Code!

```
Dim zähler As Integer
For zähler = 1 To 100
    Debug.Print zähler
Next
```

In allen .NET-Basic-Derivaten geht das eleganter:

```
Private Sub btnSchleifendemo_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnSchleifendemo.Click
    For zähler As Integer = 0 To 100
        Debug.Print("Wert:" & zähler)
    Next
    .
    .
    .
```

Das gilt im Übrigen auch für For Each-Schleifen, wie man Sie in VB6 überwiegend beim Iterieren durch Collection-Objekte verwendet hat:

HINWEIS Die Variable ist dann zudem noch eine »Blockvariable«, d.h. sie ist nur in einem Block, hier der Next-Schleife gültig. Somit kennt .NET auch neue Gültigkeitsbereiche von Variablen – doch dazu mehr im Abschnitt »Gültigkeitsbereiche von Variablen« ab Seite 335.

Achtung! VB6-Code!

```
Dim auflistung As Collection
Set auflistung = New Collection
auflistung.Add 5
auflistung.Add 10
auflistung.Add 15
auflistung.Add 20

Dim element As Variant
For Each element In auflistung
    Debug.Print element
Next
```

Das VB.NET-Pendant sieht typischerweise folgendermaßen aus:

```
.
.
.

Dim auflistung As New ArrayList
auflistung.Add(5)
```

```

auflistung.Add(10)
auflistung.Add(15)
auflistung.Add(20)

For Each element As Integer In auflistung
    Debug.Print("Wert:" & element)
Next
End Sub

```

HINWEIS

Lokaler Typrückschluss gilt natürlich auch bei der Deklaration von Schleifenvariablen.

Unterschiede bei verwendbaren Variablentypen für For/Each in VB6 und VB.NET

In diesem Zusammenhang eine kleine Anmerkung: In VB6 konnten innerhalb von For Each-Schleifen nur Variant- oder aus Klassen entstandene Objektvariablen verwendet werden. Direkt einen Grunddatentyp zu verwenden war, wie im obigen .NET-Listing zu sehen, nicht möglich.

Diese Einschränkung muss natürlich in .NET wegfallen, weil, wie Sie bereits lesen konnten, jede Variable auf der Basisklasse aller Basisklassen Object basiert. Aus diesem Grund können Sie in For Each-Schleifen in jedem der .NET-Basic-Derivate auch primitive Datentypen wie Integer, Long, Single, Double, Date, Decimal, String, etc. verwenden.

Continue in Schleifen

Continue erlaubt ein vorzeitiges Wiederholen (nicht Beenden) einer Schleife. Für ein solches Konstrukt mussten Sie sich bislang entweder einer Hilfsvariablen oder dem (verpönten) GoTo-Befehl bedienen. Der folgende Code zeigt zwei Schleifenbeispiele, die exakt dasselbe machen und die auch dieselbe Performance aufweisen – einmal jedoch mit Goto und ein anderes Mal mit Continue arbeiten.

```

Public Class Form1

    Private Sub btnContinue_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnContinue.Click
        'For/Next-Schleife - Variante 1 - Mit Goto
        'Zahlen ausgeben, aber bis 10 nur gerade, ab 10 nur ungerade
        For c As Integer = 0 To 20
            If c < 10 Then
                If (c \ 2) * 2 <> c Then
                    GoTo SkipLoop
                End If
                Debug.Print("Gerade: " & c)
            End If

            If c > 10 Then
                If (c \ 2) * 2 = c Then
                    GoTo SkipLoop
                End If
                Debug.Print("Ungerade: " & c)
            End If
        Next
    End Sub

```

```
    End If
SkipLoop:
    Next

    'For/Next-Schleife - Variante 2 - Mit Continue For
    'Zahlen ausgeben, aber bis 10 nur gerade, ab 10 nur ungerade
    For c As Integer = 0 To 20
        If c < 10 Then
            If (c \ 2) * 2 <> c Then
                Continue For
            End If
            Debug.Print("Gerade: " & c)
        End If

        If c > 10 Then
            If (c \ 2) * 2 = c Then
                Continue For
            End If
            Debug.Print("Ungerade: " & c)
        End If
    Next
End Sub
End Class
```

HINWEIS Continue können Sie auch in Do- bzw. While-Schleifen einsetzen. Auch in diesem Fall bewirkt Continue einen Sprung zurück zum Schleifenanfang. Bei geschachtelten Schleifen desselben Typs, z.B. einer Do-Schleife in einer weiteren Do-Schleife, springt eine Continue Do-Anweisung zum Beginn der inneren Do-Schleife. Sie können hier hingegen nicht mit Continue zur äußeren Schleife desselben Typs springen. Bei geschachtelten Schleifen von unterschiedlichem Typ, z.B. einer Do-Schleife in einer For-Schleife, können Sie mit Continue Do bzw. Continue For gezielt zur nächsten Iteration einer der beiden Schleifenkonstrukte springen.

Gültigkeitsbereiche von Variablen

VB6 kannte nur drei Gültigkeitsbereiche innerhalb von Modulen, Klassen oder Formularen.

- Globale Variablen, die mit `Global` innerhalb von Modulen definiert wurden, und auf die Sie vom ganzen Projekt aus zugreifen konnten.
- Member-Variablen eines Moduls, eines Formulars oder einer Klasse, die innerhalb des ganzen Code-files, aber nur da, gültig waren.
- Lokale Variablen, die innerhalb einer Sub, einer Function oder einer Property-Prozedur definiert wurden, und dann ab dem Zeitpunkt ihrer Definition für den kompletten Rest der Prozedur ihre Gültigkeit behielten.

Globale bzw. öffentliche (public) Variablen

Globale Variablen nach dem Vorbild von VB6 gibt es nicht mehr; jedenfalls nicht mit dem `Global`-Modifizierer von VB6. Möchten Sie, dass auf eine Variable eines Moduls von außen zugegriffen werden

kann, definieren Sie sie als öffentlich, und das geschieht mit dem Modifizierer `Public`. VB6 konnte das im Übrigen auch schon – viele Umsteiger von DOS-Basic-Derivaten waren aber an die Global-Syntax gewöhnt und verwendeten sie daher auch weiter.

Anstatt also wie in VB6 vielfach gemacht auf Modulebene eine Variable mit

```
Global einInteger As Integer
```

zu deklarieren, verwendeten Sie in VB.NET

```
Public einInteger As Integer
```

HINWEIS Auch in einer objektorientierten Programmiersprache gibt es die Möglichkeit, bestimmte Instanzen innerhalb seines Projektes zu haben, an denen allgemein zugängliche Variablen liegen. Bestimmte Programmeinstellungen, die von überall im Projekt abgerufen werden können, machen das auch bei der OOP erforderlich. Doch Sie sollten es auf jeden Fall vermeiden, eine komplette Kommunikation zwischen verschiedenen Programmeinheiten über diese Art und Weise abzuwickeln, denn das entspricht nicht der OOP.

Es ist aber typisch für die prozedurale Programmierung bzw. schlechte prozedurale Programmierung, da man den Austausch von Informationen auch hier über dezidierte Parameter abwickeln sollte.

Vermeiden Sie also das Offenlegen von Variablen auf die hier gezeigte Methode, wann immer es geht. Welche Alternativen sich Ihnen stattdessen bieten, werden Sie fast schon automatisch wissen, wenn Sie den OOP-Teil dieses Buchs durchgearbeitet haben.

TIPP Gerade globale Variablen müssen zum Ende einer Anwendung oft »festgehalten« werden; ihre aktuellen Werte also speichern, damit sie beim nächsten Start der Anwendung wieder geladen und in ihrer ursprünglichen Konfiguration zur Verfügung stehen. Das .NET Framework bietet für Windows Forms-Anwendungen einen automatisierten Prozess, genau dieses zu erreichen, und in Visual Basic .NET ist das durch die Visual Basic-Vereinfachungen, die mit dem `My`-Namespace zur Verfügung gestellt werden, noch einmal einfacher zu realisieren. Kapitel 29 zeigt Ihnen anhand von Beispielen, wie Sie die entsprechenden Techniken nutzen können, um globale Variablenzustände zum einen in Ihrer Anwendung global zur Verfügung zu stellen und diese zum anderen beim Start Ihrer Anwendung automatisch geladen bzw. am Ende der Anwendung automatisch gesichert zu haben.

Variablen mit Gültigkeit auf Modul, Klassen oder Formularebene

Hier hat sich im Gegensatz zu VB6 nichts Konzeptionelles getan. Eine Variable, die Sie beispielsweise als `Private` auf Modul, Formular oder Klassenebene definiert haben, war von außen zwar nicht sichtbar (eben »privat«), aber Sie konnten auf sie überall im Modul, Formular oder in Ihrer Klasse zugreifen:

Achtung! VB6-Code!

```
Private einIntergerMember As Integer

Private Sub Command1_Click()
    'Von hier aus ist dranzukommen
    einIntergerMember = 10
End Sub

Private Sub Command2_Click()
```

```
'Von hier aus auch
einIntegerMember = 10
End Sub
```

In den VB.NET-Derivaten hat sich an diesem Gültigkeitsbereich nichts geändert. In der OOP spricht man übrigens bei derartig deklarierten Variablen von »Membervariablen« (etwa: Mitgliedervariablen), und sie sind sogar ein ganz wichtiges Merkmal und elementarer Bestandteil von Klassen. Denn ein Formular ist in .NET (und war es auch schon in VB6) eine Klasse.

Gültigkeitsbereiche von lokalen Variablen

Bei Variablen, die auf Prozedurebene deklariert wurden (also innerhalb von Subs, Functions oder Property-Prozeduren), gibt es eine ganz entscheidende, sehr wichtige Neuerung. Während in VB6 Variablen innerhalb von Prozeduren überall ab dem Zeitpunkt ihrer Deklaration gültig waren, sind sie das in VB.NET ab dem Zeitpunkt ihrer Deklaration nur innerhalb der Struktur, in der sie definiert sind. »Struktur« in diesem Zusammenhang bedeutet im Prinzip irgendetwas, was Code in irgendeiner Form einklammern kann – das können beispielsweise If/Then/Else-Abfragen, For/Next- oder Do/Loop-Schleifen aber auch With-Blöcke sein. Es gilt dabei die Regel: Jede Strukturanweisung, die dazu führt, dass der Editor den dazwischen platzierten Code einrückt, begrenzt automatisch den Gültigkeitsbereich von Variablen, die in diesem Block definiert sind. Ein Beispiel zeigt Abbildung 10.7.

```
Option Strict On
Module Module1
    Sub Main()
        For zähler As Integer = 0 To 10
            Dim fünfGefunden As Boolean
            If zähler = 5 Then
                fünfGefunden = True
            End If
        Next
        If fünfGefunden Then
            Debug.Print("Die Zahl 5 kam in der Zahlenreihe vor!")
        End If
    End Sub
End Module
```

Abbildung 10.7 Variablen sind nur in dem Strukturblock gültig, in dem sie deklariert wurden

Wie in der Abbildung zu sehen, versucht das Programm beim zweiten Mal auf fünfGefunden zuzugreifen, nachdem der Strukturbereich und damit auch der Gültigkeitsbereich der Variable verlassen wurde – und das führt zu einem Fehler.

Wollten Sie fünfGefunden in jedem Strukturblock zugreifbar machen, müssten Sie folgende Änderungen vornehmen:

```
Sub Main()
    Dim fünfGefunden As Boolean
    For zähler As Integer = 0 To 10
```

```

'Dim fünfGefunden As Boolean
If zähler = 5 Then
    fünfGefunden = True
End If
Next

If fünfGefunden Then
    Debug.Print("Die Zahl 5 kam in der Zahlenreihe vor!")
End If
End Sub

```

Die Änderungen sind hier im Listing durch Fetschrift gekennzeichnet.

Diese Regel für Gültigkeitsbereiche führt dazu, dass Variablen innerhalb mehrerer Strukturböcke mehrfach unter gleichem Namen deklariert werden können, wie das folgende Beispiel zeigt:

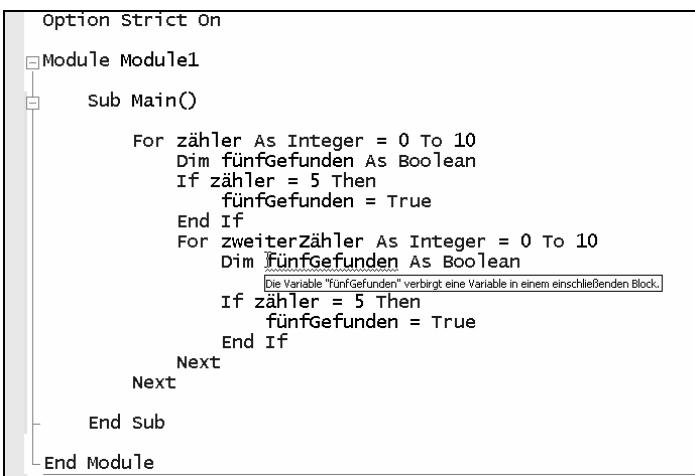
```

Sub Main()
    For zähler As Integer = 0 To 10
        Dim fünfGefunden As Boolean
        If zähler = 5 Then
            fünfGefunden = True
        End If
    Next
    For zähler As Integer = 0 To 10
        Dim fünfGefunden As Boolean
        If zähler = 5 Then
            fünfGefunden = True
        End If
    Next
End Sub

```

Hier wird die Variable `fünfGefunden` innerhalb einer Prozedur zweimal deklariert – dennoch meldet Visual Basic keinen Fehler, weil sich beide Deklarationen in unterschiedlichen Gültigkeitsbereichen befinden.

Allerdings:



The screenshot shows a code editor window with the following code:

```

Option Strict On

Module Module1
    Sub Main()
        For zähler As Integer = 0 To 10
            Dim fünfGefunden As Boolean
            If zähler = 5 Then
                fünfGefunden = True
            End If
        For zweiterZähler As Integer = 0 To 10
            Dim FünfGefunden As Boolean
            Die Variable "FünfGefunden" verbirgt eine Variable in einem einschließenden Block.
            If zähler = 5 Then
                fünfGefunden = True
            End If
        Next
    Next
End sub
End Module

```

A tooltip is visible near the second declaration of `FünfGefunden`, stating: "Die Variable "FünfGefunden" verbirgt eine Variable in einem einschließenden Block."

Abbildung 10.8 Variablen eines übergeordneten Gültigkeitsbereichs dürfen solche eines untergeordneten nicht verbergen

Das Deklarieren von Variablen gleichen Namens in einem Gültigkeitsbereich, der einen anderen kapselt, funktioniert natürlich nicht, denn Variablen in einem übergeordneten Gültigkeitsbereich sind ohnehin immer von einem untergeordneten aus zugreifbar. Eine entsprechende Fehlermeldung würde, falls Ihnen das passiert, dann so ausschauen, wie in Abbildung 10.8 zu sehen.

Arrays

In Visual Basic 6.0 konnten Array-Grenzen frei definiert werden. Das folgende Codekonstrukt hatte dort also durchaus Gültigkeit:

Achtung! Visual Basic 6.0-Code!

```
Dim einArray(-5 To 10) As Integer
For z% = -5 To 10
    einArray(z%) = 10
Next z%
```

Dieses Konstrukt funktioniert in den .NET-Basic-Derivaten so nicht mehr. Arrays in VB.NET sind mit VB-Boardmitteln ausschließlich 0-basierend anlegbar, und können grundsätzlich nicht mehr mithilfe von `To` mit beliebigen Ober- und Untergrenzen definiert werden.

Dazu kommt: Anders als in anderen .NET-Programmiersprachen geben Sie in Visual Basic .NET nicht die *Anzahl* der zu dimensionierenden Elemente sondern die *höchste obere Array-Grenze* an. Die Anweisung:

```
Dim einArray(10) As Integer
```

definiert also nicht 10 Elemente, sondern ausreichend Platz für die Elemente 0–10 – und das sind 11 Elemente.

Der besseren Lesbarkeit halber erlaubte deswegen schon Visual Basic 2005 wieder die Angabe von `To` bei der Dimensionierung – der untere Wert muss dabei aber grundsätzlich 0 sein. Die erneute Verwendungsmöglichkeit von `To` bei der Dimensionierung ist also reine Kosmetik:

```
'Ab VB2005 möglich:
Dim einArray(0 To 10) As Integer ' Definiert 11 Elemente
```

HINWEIS Wie schon erwähnt gilt diese Art der Dimensionierung von Elementen eines Arrays nur in Visual Basic .NET. In C# beispielsweise wird, wie in allen anderen .NET-Programmiersprachen, die standardmäßig von Microsoft mit Visual Studio ausgeliefert werden, die Anzahl der Elemente angegeben, und das sieht dann beispielsweise so aus:

```
// 10 Elemente definieren
int[] einArray=new int[10];
// Alle 10 Elemente (0-9) durchlaufen...
for (int zähler=0; zähler<10; zähler++)
    // ...und jedem den Wert 10 zuweisen.
    einArray[zähler]=10;
```

Das ist beispielsweise dann wichtig zu wissen, wenn Sie ein Beispiel für die Lösung eines Problems im Internet nur als C#-Version gefunden haben. Hier werden häufig Fehler beim »Übersetzen« von C# nach VB gemacht.⁶

Die Operatoren += und -= und ihre Verwandten

Mit += und -= ist Visual Basic um Operatoren für numerische Berechnungen und bei &= auch für String-verkettungen reicher geworden – das folgende Beispiel verdeutlicht ihre Verwendung.

```
If Not IsRangeOkProper(txtValue.Text) Then
    MsgBox("Wertebereich wurde überschritten!" & vbCrLf & "(Nur im Integerbereich von 0 bis 32768)" & vbCrLf
        & "Info: Das war die " & Str(locErrorCount + 1) & ". Fehleingabe", _
        MsgBoxStyle.OKOnly Or MsgBoxStyle.Exclamation, "Falsche Eingabe")
    locErrorCount += 1
    txtValue.Focus()
    Exit Sub
End If
```

Es gibt andere Operatoren in VB.NET, die das ebenfalls können – die folgende Tabelle zeigt, welche das sind:

Operation	Kurzform	Beschreibung
var = var + 1	var += 1	Den Variableninhalt um eins erhöhen.
var = var - 1	var -= 1	Den Variableninhalt um eins verringern.
var = var * 2	var *= 2	Den Variableninhalt verdoppeln (mal zwei nehmen).
var = var / 2	var /= 2	Den Variableninhalt halbieren (durch zwei teilen – Fließkomma-division).
var = var \ 2	var \= 2	Den Variableninhalt ohne Rest halbieren (durch zwei teilen – Integerdivision).
var = var ^ 3	var ^= 3	Den Variableninhalt mit drei potenzieren.
varString = VarString & "Klaus"	varString &= "Klaus"	An den Inhalt des Strings varString die Zeichenkette »Klaus« anhängen.

Tabelle 10.3 Kurzformen von Operatoren in Visual Basic

⁶ Es gibt übrigens Hilfsprogramme im Internet, mit der sich Codeausschnitte von C# nach Visual Basic .NET und umgekehrt übersetzen lassen – in einer inzwischen recht erstaunlichen Qualität sogar. Unter dem IntelliLink **B1001** finden Sie diese Tools.

HINWEIS Auch wenn die Verwendung der Kurzformen weniger Tipparbeit macht – eine schnellere Codeausführung bewirkt sie nicht.

Also ganz egal ob Sie

```
intvar = intvar + 1
```

oder

```
intvar += 1
```

schreiben – der Compiler wird aus beiden Angaben den gleichen Code erzeugen.

Die Bitverschiebeoperatoren << und >>

Zusätzlich zu den genannten Operatoren gibt es ab VB.NET 2003 Bitverschiebeoperatoren, mit denen die einzelnen Bits von Integerwerten schrittweise nach links oder nach rechts verschoben werden können. Ohne im Detail auf die Funktionsweise des Binärsystems eingehen zu wollen: Eine Verschiebung der Bits eines Integerwertes nach links verdoppelt seinen Wert, eine Verschiebung nach rechts teilt seinen Wert ohne Rest durch 2.

Aus binär 101 (dezimal 5) wird durch Rechtsverschiebung also binär 10 (dezimal 2) und durch Linksverschiebung binär 1010 (dezimal 10).

Der folgende Code demonstriert einen Multiplikationsalgorithmus auf Bitebene. Die Älteren unter uns erinnern sich sicherlich noch an Ihre Commodore 64-Zeiten. Dort galt das Anwenden solcher Algorithmen in Assembler zur täglichen Praxis!

```
Private Sub btnMultiplikationMitBitverschiebung_Click(ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles btnMultiplikationMitBitverschiebung.Click  
    Dim wert1, wert2, ergebniswert, hilfswert As Integer  
    wert1 = 10  
    wert2 = 5  
    ergebniswert = 0  
    hilfswert = wert2  
  
    'Dieser Algorithmus funktioniert so, wie Sie  
    'auch im Dezimalsystem "zu Fuß" multiplizieren:  
    '  
    '(10)  (5)  
    '1010 * 101 =  
    '-----  
    '      1010 +  
    '      0000 +  
    '      1010  
    '-----  
    '      101010 = 50  
  
    'Die "5" wird dazu bitweise nach rechts verschoben,  
    'um ihr rechtes äußeres Bit zu testen. Ist es gesetzt,
```

```

'wird der Wert von 10 zunächst addiert, und dann sein
'kompletter "Bitinhalt" um eine Stelle nach links verschoben;
'ist es hingegen nicht gesetzt, passiert gar nichts.
'Dieser Vorgang wiederholt sich solange, bis alle
'Bits von "5" verbraucht sind - die Variable hilfswert,
'die diesen Wert verarbeitet, also 0 geworden ist.
'Für eine Multiplikation sind also gerade so viele
'Additionen nötig, wie Bits im zweiten Wert gesetzt sind.

Do
    If (hilfswert And 1) = 1 Then
        ergebniswert += wert1
    End If
    wert1 = wert1 << 1
    hilfswert = hilfswert >> 1
Loop Until hilfswert = 0
Debug.Print("Das Ergebnis lautet:" & ergebniswert)
End Sub

```

Fehlerbehandlung

Mir persönlich war die Fehlerbehandlung im alten Visual Basic immer ein Gräuel. Trat in einer sehr langen Routine im fertigen Programm ein Fehler auf, war es vergleichsweise schwierig oder mit großem Aufwand verbunden, die genaue Stelle des Fehlers zu lokalisieren. Zwar konnten Sie mit der Systemvariablen `Erl` die Zeile in der Fehlerbehandlungsroutine anzeigen lassen, in der der Fehler aufgetreten war, aber dazu mussten Sie manuell vor jede Zeile eine Zeilennummer setzen – selbst für »damalige« Verhältnisse war das doch eine eher vorsintflutliche Vorgehensweise.

Das geht heute viel, viel einfacher – auch wenn die VB.NET-Entwickler die ursprüngliche Verfahrensweise auch noch zulassen. Wahrscheinlich wollten sie nicht zu viele Inkompatibilitäten schaffen. Doch schauen wir uns abermals das Vorher und das Nachher an – hier am Beispiel einer kleinen VB6-Routine, die eine Datei in eine String-Variable liest, oder dies zumindest versucht. Achten Sie im Listing auf die Kommentare, die mit den Ziffern den Programmverlauf im Falle eines Fehlers kennzeichnen.

Achtung! Visual Basic 6.0-Code!

```

Private Sub Command1_Click()

Dim DateiNichtGefundenFlag As Boolean
On Local Error GoTo 1000

Dim ff As Integer
Dim meinDateiInhalt As String
Dim zeilenspeicher As String
ff = FreeFile
'1: Hier tritt der Fehler auf
Open "C:\EineTextdatei.TXT" For Input As #ff
'3: dann wieder hier hin
If DateiNichtGefunden Then
    '4: um dann diese Meldung auszugeben.
    MsgBox ("Die Datei existiert nicht")
Else

```

```

'Dieser Block wird nur ausgeführt,
'wenn alles OK war.
Do While Not EOF(ff)
    Line Input #ff, zeilenspeicher
    meinDateiInhalt = meinDateiInhalt & zeilenspeicher
Loop
Close ff
Debug.Print meinDateiInhalt
End If
'Und das ist auch nötig, damit das
'Programm nicht in die Fehlerroutine rennt.
Exit Sub

'2: Hier springt dann das Programm hin
1000 If Err.Number = 53 Then
    DateiNichtGefunden = True
    Resume Next
End If

End Sub

```

Unfassbar, oder? Um einen simplen Fehler abzufangen, muss sich das Programm kreuz und quer durch den Programmcode quälen, und – einem Steinbock im Himalaja gleich – hin und her springen, was das Zeug hält. Und wir fangen hier gerade mal einen einzigen Fehler ab!

Das noch Unangenehmere ist: Prinzipiell ist dieser On Error GoTo-Quatsch auch noch in allen VB.NET-Derivaten möglich, wenn auch alles andere als nötig – wie Sie gleich noch sehen werden. Das Einlesen einer Textdatei funktioniert hier zwar ein wenig anders, da es Open und Close in dieser VB6-Form nicht mehr gibt. Aber darum geht es an dieser Stelle auch gar nicht.

WICHTIG Der einzige Unterschied, jedenfalls was die Fehlerbehandlung anbelangt: Zeilennummern müssen, wie andere Sprungmarken, mit Doppelpunkten versehen werden. Wollte man also das obige Programm in VB.NET übersetzen, käme vielleicht dieses (leider) mögliche Ergebnis dabei heraus:

```

Private Sub btnDateiLesenDotNet_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnDateiLesenDotNet.Click
    'WICHTIG:
    'Um auf die IO-Objekte zuzugreifen, muss "Imports System.IO"
    'am Anfang der Codedatei stehen!

    Dim DateiNichtGefundenFlag As Boolean
    'Selbe Blödsinn hier!
    On Error GoTo 1000

    Dim meinDateiInhalt As String
    '1: Wenn hier ein Fehler auftritt
    Dim dateiStromLeser As New StreamReader("C:\EineTextdatei.txt")
    '3: um das durch Resume Next hier wieder zu landen
    If DateiNichtGefundenFlag Then
        '4: und den Fehler abzufangen
        MessageBox.Show("Datei wurde nicht gefunden!")
    Else

```

```

'Trat kein Fehler auf, wird dieser Block ausgeführt
meinDateiInhalt = dateiStromLeser.ReadToEnd()
dateiStromLeser.Close()
'Und der Dateiinhalt ausgegeben.
Debug.Print(meinDateiInhalt)
End If
Exit Sub
'2: Fährt der Programmablauf hier fort
1000: If Err.Number = 53 Then
    DateiNichtGefundenFlag = True
    Resume Next
End If
End Sub

```

Elegantes Fehlerabfangen mit Try/Catch/Finally

Es gibt aber eine viel, viel elegantere Möglichkeit in VB.NET, Fehlerbehandlungen zu implementieren. Im Gegensatz zu On Error GoTo erlaubt die Struktur Try/Catch/Finally Fehler an den Stellen zu behandeln, an denen sie auch tatsächlich auftreten können. Schauen Sie sich zur Verdeutlichung das folgende Beispiel an, das die auf Try/Catch adaptierte Version des obigen Beispiels enthält:

```

Private Sub btnDateiLesenDotNet_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnDateiLesenDotNet.Click
    'WICHTIG:
    'Um auf die IO-Objekte zuzugreifen, muss "Imports System.IO"
    'am Anfang der Codedatei stehen!

    Dim meinDateiInhalt As String
    Dim dateiStromLeser As StreamReader

    Try
        'Diese folgenden Befehle probieren (try=versuchen, ausprobieren).
        dateiStromLeser = New StreamReader("C:\meineTextdatei.txt")
        meinDateiInhalt = dateiStromLeser.ReadToEnd()
        dateiStromLeser.Close()
        Debug.Print(meinDateiInhalt)
    Catch ex As FileNotFoundException
        'Hier werden nur FileNotFoundExceptions abgefangen
        MessageBox.Show("Datei wurde nicht gefunden!" & vbCrLf &
                        vbCrLf & "Der Ausnahmetext lautete:" & vbCrLf & ex.Message, _
                        "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
    Catch ex As Exception
        'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
        'behandelten Ausnahmen abgefangen
        'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
        'behandelten Ausnahmen abgefangen
        MessageBox.Show("Beim Verarbeiten der Datei trat eine Ausnahme auf!" & vbCrLf &
                        "Die Ausnahme war vom Typ:" & ex.GetType.ToString & vbCrLf &
                        vbCrLf & "Der Ausnahmetext lautete:" & vbCrLf & ex.Message, _
                        "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
    End Try
End Sub

```

Was passiert hier? Alle Anweisungen, die zwischen der Try- und der ersten Catch-Anweisung platziert sind, befinden sich in einer Art »Versuchsmodus«. Tritt bei der Ausführung einer dieser Anweisungen ein Fehler auf, springt das Programm automatisch in den »zutreffenden« Catch-Block. Und was bedeutet dabei »zutreffend«?

Das Ausnahmenabfangen ist nicht nur auf einen Ausnahmetyp beschränkt

Wenn Sie in den Editor einfach nur die Zeichenfolge »Try« eingeben und anschließend drücken, fügt dieser automatisch den folgenden Block ein:

```
Try  
Catch ex As Exception  
End Try
```

Exception lautet die in der Vererbungshierarchie zuoberst stehende Klasse, deren Instanzen (hier durch ex referenziert) ausnahmslos alle Ausnahmen abfangen können. Doch es könnte, wie im Beispielcode zu sehen, sinnvoll sein, zwischen den vielen verschiedenen Ausnahmentypen zu differenzieren. Ihr Programm soll nämlich vielleicht auf eine Ausnahme, die durch eine nicht vorhandene Datei ausgelöst wird, anders reagieren, als auf eine Datei, die zwar vorhanden, aber gerade mit einem anderen Programm verarbeitet wird.

Sie können das am Beispielcode nachvollziehen. Wenn Sie diesen starten, wird eine Ausnahme ausgelöst, die sich FileNotFoundException nennt. Ein Catch mit diesem Klassennamen als Argument fängt nur Ausnahmen dieses Typs ab. Dementsprechend wird der dort darunter stehende Code ausgeführt, und der Code sorgt dafür, dass ein entsprechender Dialog angezeigt wird.

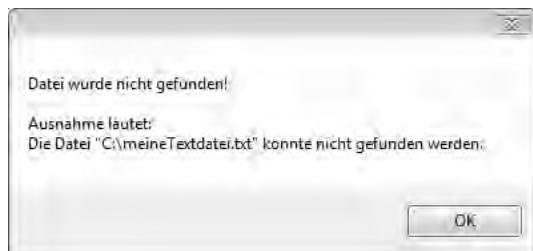


Abbildung 10.9 Im Beispielcode wird diese Meldung ausgegeben, wenn die Datei nicht vorhanden war, also eine FileNotFoundException vorlag

Wenn Sie nun aber auf Laufwerk C: eine Textdatei mit diesem Namen anlegen, und diese Datei anschließend beispielsweise in Microsoft Word öffnen, dann produziert die Zeile

```
dateiStromLeser = New StreamReader("C:\meineTextdatei.txt")
```

abermals eine Ausnahme – doch dieses Mal keine vom Typ FileNotFoundException sondern vom Typ IOException:

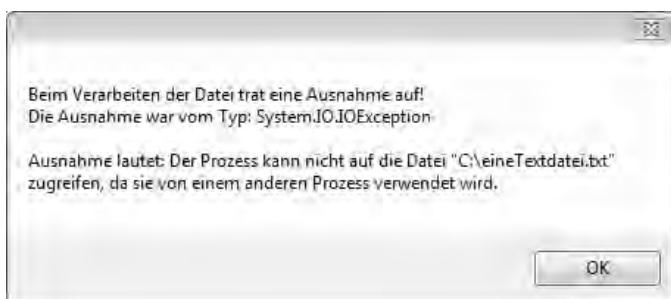


Abbildung 10.10 Jetzt ist die Datei zwar vorhanden, aber durch Word bereits geöffnet

Und wieso wird diese Ausnahme durch Catch as Exception abgefangen, obwohl Sie vom Typ IOException ist? Das liegt daran, dass IOException auf Exception basiert. In der objektorientierten Programmierung können durch Vererbung aus Klassen erweiterte Klassen entstehen, aus diesen wiederum nochmals spezialisierte, und so fort. Genau so ist das bei Ausnahmeklassen. Exception selbst ist die Basisklasse. Davon abgeleitet ist die Ausnahmeklasse SystemException. Und davon wiederum ist IOException abgeleitet. Da wir im Beispiel aber IOException nicht gesondert behandeln, wird der Catch-Block angesteuert (so vorhanden), der zumindest eine der Basisklassen der aufgetretenen Ausnahmeklassen repräsentiert. Bei der Verwendung von Exception sind Sie dabei immer auf der sicheren Seite, denn auf dieser basieren *alle* anderen Ausnahmeklassen. Und so wird zwar die Ausnahme FileNotFoundException gesondert behandelt, aber bei allen anderen Ausnahmen der Catch-Block ausgeführt, der mit Exception selbst arbeitet.

WICHTIG Sie können in einem Try-Catch-Block so viele Catch-Zweige implementieren, wie Sie möchten, und damit alle denkbaren Ausnahmetypen gezielt abfangen. Achten Sie dabei aber darauf, dass Sie die spezialisierteren Ausnahmetypen nach oben stellen, und die, auf denen diese basieren, erst weiter unten kommen. Andernfalls handeln Sie sich Ärger mit dem Compiler ein, der das schlichtweg nicht zulässt. Denn eine Basisausnahmeklasse würde in diesem Fall eine spezialisierte Ausnahme schon längst verarbeitet haben, bevor der Catch-Block mit der spezialisierteren Ausnahme zum Zuge käme. Und diesen »Un-Sinn« unterbindet der Compiler von vornherein:

```

Try
    'Diese folgenden Befehle probieren (try=versuchen, ausprobieren).
    dateiStromLeser = New StreamReader("C:\eineTextdatei.txt")
    meinDateiInhalt = dateiStromLeser.ReadToEnd()
    dateiStromLeser.Close()
    Debug.Print(meinDateiInhalt)
Catch ex As Exception
    'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
    'behandelten Ausnahmen abgefangen
    MessageBox.Show("Beim Verarbeiten der Datei trat eine Ausnahme auf!" & vbCrLf &
                   "Die Ausnahme war vom Typ:" & ex.GetType.ToString() & vbCrLf &
                   vbCrLf & "Der Ausnahmetext lautete:" & vbCrLf & ex.Message, _
                   "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
Catch ex As FileNotFoundException
    'Der Catch-Block wird niemals erreicht, da "System.IO.FileNotFoundException" von "System.Exception" erbte
    'MessageShow("Datei wurde nicht gefunden!" & vbCrLf &
    '           vbCrLf & "Der Ausnahmetext lautete:" & vbCrLf & ex.Message, _
    '           "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try

```

Abbildung 10.11 Catch-Blöcke mit spezialisierteren Ausnahmeklassen müssen vor den Basisausnahmeklassen platziert werden, sonst meckert der Compiler

Und wozu dient Finally?

Programmcode, den man in einem Finally-Block platziert, wird in jedem Fall ausgeführt. Auch dann, wenn ein Fehler auftritt, mit Catch abgefangen wird und die Anweisung ergeht, die gesamte Prozedur beispielsweise mit Return eigentlich noch im Catch-Block zu verlassen.

Denn normalerweise ist ja Return der letzte Befehl in einer Prozedur, ganz gleich, wo Return platziert wurde. Nach Return ist Schicht. Doch in einigen Fällen ist das nicht sinnig, gerade wenn es um Fehlerbehandlungen geht, und deswegen bildet Finally in diesem Fall die goldene Ausnahme.

Angenommen, Sie lesen aus einer Datei, und diese Datei haben Sie dazu vorher geöffnet. Beim Öffnen der Datei ist zwar kein Fehler aufgetreten, aber beim Lesen – vielleicht ist der Fall eingetreten, dass Sie über das Dateiende hinaus gelesen haben, und nun sind Zeilen, die Sie verarbeiten wollen, leer (also Nothing). Diesen Fall (das Abfangen einer Null-Referenz) haben Sie mit einem entsprechenden Catch-Block behandelt, und da es nichts Weiteres zu tun gibt, möchten Sie nach der Anzeige einer Fehlermeldung Ihre Leseroutine mit Return direkt aus dem Catch-Block heraus verlassen. Das Problem: die Datei ist noch geöffnet, und Sie sollten sie schließen. Mit Finally lässt sich ein solcher Vorgang elegant implementieren.

Das folgende Beispiel simuliert einen solchen Prozess. Es liest »c:\eineTextdatei.txt« zeilenweise aus und wandelt die einzelnen Zeilen in Großbuchstaben um, bevor es sie zu einem Gesamttextblock zusammenführt. Doch da es viel zu viele Zeilen liest, funktioniert die ToUpper-Methode des Strings, die die Zeile in Großbuchstaben umwandeln soll, irgendwann nicht mehr, denn die ReadLine-Funktion greift ins Leere und liefert Nothing zurück:

```
Private Sub btnTryCatchFinally_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnTryCatchFinally.Click
    'WICHTIG:
    'Um auf die IO-Objekte zuzugreifen, muss "Imports System.IO"
    'am Anfang der Codedatei stehen!

    Dim meinDateiInhalt As String
    Dim dateiStromLeser As StreamReader

    Try
        'Diese folgenden Befehle probieren (try=versuchen, ausprobieren).
        dateiStromLeser = New StreamReader("C:\eineTextdatei.txt")
        Dim locZeile As String
        meinDateiInhalt = ""
        'Jetzt lesen wir zeilenweise aber viel zu viele Zeilen,
        'und schließen daher irgendwann über das Ende der Datei hinaus:
        Try
            ' Wenn Ihre Datei "C:\eineTextdatei" nicht gerade
            ' 1001 Zeilen enthält, knallt es hier, denn:
            For zeilenzähler As Integer = 0 To 1000
                locZeile = dateiStromLeser.ReadLine()
                'locZeile ist jetzt Nothing, und dann kann
                'die Konvertierung in Großbuchstaben nicht mehr
                'funktionieren.
                locZeile = locZeile.ToUpper
                meinDateiInhalt &= locZeile
            Next
            Catch ex As NullReferenceException
                MessageBox.Show("Die Zeile konnte nicht umgewandelt werden, weil sie leer war!")
            End Try
        End Try
    End Try
End Sub
```

```

    ' Return? Aber die Datei ist doch noch geöffnet!!!
    Return
Finally
    ' Egal! Auch bei Return im Catch-Block wird Finally in jedem Fall noch ausgeführt!
    dateiStromLeser.Close()
End Try
'Aber diese Zeile wird nur im Erfolgsfall abgearbeitet:
Debug.Print(meinDateiInhalt)
Catch ex As FileNotFoundException
    'Hier werden nur FileNotFoundExceptions abgefangen
    MessageBox.Show("Datei wurde nicht gefunden!" & vbCrLf &
                    vbCrLf & "Der Ausnahmetext lautete:" & vbCrLf & ex.Message,
                    "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
Catch ex As Exception
    'Hier werden alle anderen bis zu diesem Zeitpunkt nicht
    'behandelten Ausnahmen abgefangen
    MessageBox.Show("Beim Verarbeiten der Datei trat eine Ausnahme auf!" & vbCrLf &
                    "Die Ausnahme war vom Typ:" & ex.GetType().ToString() & vbCrLf &
                    vbCrLf & "Der Ausnahmetext lautete:" & vbCrLf & ex.Message,
                    "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try
End Sub

```

TIPP Sie können diesen Vorgang übrigens gut nachvollziehen, indem Sie einen Haltepunkt mit **F9** in der Zeile setzen, die **Return** beinhaltet, und das Programm starten. Wenn Sie anschließend mit **F11** schrittweise durch das Programm steppen, werden Sie feststellen, dass nach dem **Return** der Code im **Finally**-Block noch abgehandelt wird.

Kurzschlussauswertungen mit OrElse und AndAlso

Betrachten Sie den folgenden Codeblock:

```

'Kurzschlussauswertung beschleunigt den Vorgang.
If locChar < "0" OrElse locChar > "9" Then
    locIllegalChar = True
    Exit For
End If

```

Auffällig ist hier das Schlüsselwort **OrElse**. Es gibt ein weiteres, das nach dem gleichen Prinzip funktioniert: **AndAlso**. Beide entsprechen den Befehlen **Or** bzw. **And**, und auch sie dienen dazu, boolesche Ausdrücke logisch miteinander zu verknüpfen und auszuwerten – nur viel schneller. Ein Beispiel aus dem täglichen Leben soll das verdeutlichen:

Wenn Sie sich überlegen, ob Sie einen Regenschirm zu einem Spaziergang mitnehmen, da es vielleicht regnet *oder auch* (*or else*) zumindest sehr verhangen ausschaut, dann machen Sie sich – berechtigterweise – schon keine Gedanken mehr, wie der Himmel aussieht, wenn Sie bereits festgestellt haben, *dass* es regnet. Sie brauchen das zweite Kriterium also gar nicht zu prüfen – der Schirm muss mit, sonst wird's nass! Genau das macht **OrElse** (bzw. **AndAlso**), und man nennt diese Vorgehensweise »Kurzschlussauswertung«.

Gerade bei Objekten bzw. dem Aufruf von Methoden können Ihnen Kurzschlussauswertungen helfen, Ihre Programme sicherer zu machen, wie das folgende Beispiel zeigt:

```

Private Sub btnAndAlsoDemo_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnAndAlsoDemo.Click
    Dim einString As String = "Klaus ist das Wort, mit dem diese Zeile beginnt"
    If einString IsNot Nothing AndAlso einString.Substring(0, 5).ToUpper = "KLAUS" Then
        MessageBox.Show("Die Zeichenfolge beginnt mit Klaus!")
    End If

    If einString IsNot Nothing And einString.Substring(0, 5).ToUpper = "KLAUS" Then
        MessageBox.Show("Die Zeichenfolge beginnt mit Klaus!")
    End If
End Sub

```

Wie erwartet, zeigt dieser Programmcode zweimal einen Meldungstext an. Denn einString hat in beiden Fällen einen Inhalt, und beide Male beginnt die Zeichenfolge (es ist ja auch dieselbe) mit »Klaus«. Doch ersetzen Sie nun die Zeile

```
Dim einString As String = "Klaus ist das Wort, mit dem diese Zeile beginnt"
```

durch

```
Dim einString As String = Nothing
```

und lassen Sie das Programm ein weiteres Mal laufen. Das Ergebnis zeigt die folgende Abbildung:

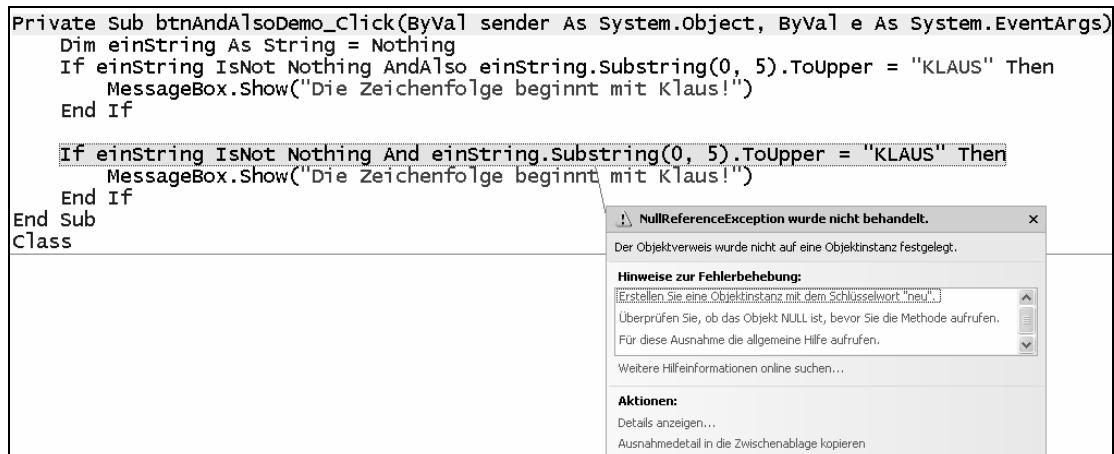


Abbildung 10.12 AndAlso dient Ihnen bei kombinierten Abfragen auf Nothing und Verwendungen von Instanzmethoden

Hier wird die Arbeitsweise von AndAlso deutlich. Die erste Abfrage funktioniert, weil der zweite, durch AndAlso verknüpfte Teil einString.Substring(0, 5).ToUpper = "KLAUS" schon gar nicht mehr abgehandelt wird. Das Objekt einString war nämlich Nothing, und bei der Verwendung von AndAlso interessieren alle folgenden Prüfteile nicht mehr.

In der zweiten Variante nur mit And wird, obwohl es in diesem Zusammenhang keinen Sinn macht, der zweite Part sehr wohl abgehandelt. Aber eben weil einString den Wert Nothing aufweist, können Sie seine Instanzfunktionen (SubString, ToUpper) nicht verwenden; das Programm bricht mit einer NullReferenceException ab.

Variablen und Argumente auch an Subs in Klammern!

Klammern bei der Übergabe von Parametern an Funktionen waren schon in VB6 Usus. Das gilt nunmehr in allen VB.NET-Derivaten auch für die Übergabe von Variablen an Subs. Es mag Ihnen am Anfang lästig erscheinen, aber der Codeeditor nimmt Ihnen die Klammererei sogar ab, falls Sie sie nicht selber durchführen wollen. Man hätte dieses Verhalten sicherlich beim Alten lassen können. Aber auf diese Weise nähert sich Visual Basic dem Standard anderer Programmiersprachen an. Sowohl in C++ als auch in C# als auch in J# werden Argumente an »Subs« nicht ohne Klammern übergeben. Es gibt dort nämlich gar keine Subs, sondern nur Funktionen vom Typ »kein Rückgabewert«, der dort paradoxeise obendrein noch einen speziellen Namen hat, nämlich **Void** (engl. etwa für »Hohlraum«, »Leere«, »Lücke«).

Streng genommen ist

```
Sub Methode(Übergabe as Irgendwas)
```

im Grunde also

```
'Das funktioniert natürlich nicht:  
Function Methode(Übergabe as Irgendwas) as Void
```

damit eine Funktion, und ihre Parameter werden ergo auch in Klammern übergeben.

Namespaces und Assemblies

Umsteigern von Visual Basic 6.0 macht das Konzept von Namespaces und Assemblies oftmals zu Anfang Schwierigkeiten, obwohl die dahinter stehenden Konzepte im Grunde genommen recht einfach zu verstehen sind. Die folgenden Abschnitte demonstrieren, was Assemblies und Namespaces sind, und wie sie bei .NET Framework-Entwicklungen zur Anwendung kommen.

Assemblies

Genau genommen ist eine Assembly eine Zusammenfassung von so genannten Modulen. Innerhalb eines Moduls können sich ausführbare Dateien oder Klassenbibliotheken befinden, die dann in einer Assembly zusammengefasst werden.

Doch in der Regel befindet sich – wenn Sie nicht explizit anderes sagen – entweder *ein* ausführbares Programm oder *eine* Klassenbibliothek in einer Assembly. Mit anderen, vereinfachenden Worten:

- Eine .EXE-Datei, die aus Ihrem Programmprojekt hervorgeht, ist eine Assembly.
- Eine .DLL-Datei, die aus einem Klassenbibliotheksprojekt hervorgeht, ist ebenfalls eine Assembly.
- Das Framework selbst besteht aus ganz vielen verschiedenen Klassenbibliotheken; und bei ihnen handelt es sich ebenfalls um Assemblies.

Die Nutzung einer Assembly, bei der es sich um eine ausführbare .EXE-Datei handelt, ist am einfachsten. Wenn eine .EXE-Datei aus Ihrem Projekt hervorgegangen ist, können Sie sie – vorausgesetzt, das Framework ist auf dem entsprechenden Computer installiert – direkt starten.

Bei der Nutzung von Assembly-DLLs ist das anders. Assemblies, die in DLLs residieren, können nicht direkt gestartet werden – sie stellen nur eine gewisse Grundfunktionalität für verschiedene Themenbereiche zur Verfügung, derer sich andere Assemblies bedienen können. Zu diesem Zweck müssen Assemblies, die sich anderer Assemblies bedienen wollen, diese referenzieren.

Wie sieht das in der Praxis aus?

Schauen wir uns das an einem einfachen Beispiel an:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\B - Migration\\Kapitel 10\\AssemblyDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Projekt ist eine Windows Forms-Anwendung, und sie besteht aus nichts weiter als einem Formular und einer Schaltfläche. Beim genauen Hinsehen werden Sie jedoch feststellen, dass direkt nach dem Laden des Projektes in der Fehlerliste eine Reihe von Fehlern auftaucht. Und wenn Sie versuchen, das Formular zu öffnen, bekommen Sie statt des Formulars im Designer eine Fehlermeldung zu sehen.



Abbildung 10.13 Irgendetwas hindert das Formular daran, dass es korrekt angezeigt werden kann

Und woran liegt das? Ganz einfach: Das Projekt muss auf eine Klassenbibliothek, auf eine Assembly aus dem Framework zurückgreifen. Doch das kann sie nicht, denn ich habe beim Erstellen des Projektes die Assembly-Verweisliste sabotiert. Das Projekt benötigt zur korrekten Darstellung des Formulars bestimmte Funktionalitäten aus der Assembly *System.Drawing.dll*, die Teil des Frameworks ist. Doch diese Funktionalitäten stehen dem Projekt momentan nicht zur Verfügung, weil sich die Assembly eben nicht in der Verweisliste befindet, und das Projekt sich der Funktionalitäten der Assembly auch nicht bedienen kann.

Sie können den Fehler folgendermaßen beheben:

- Schließen Sie zunächst den Designer, der die Fehlermeldung zeigt.
- Klicken Sie im Projektmappen-Explorer auf das Symbol *Alle Dateien anzeigen* – der entsprechende Tooltip, der erscheint, wenn Sie den Mauszeiger auf ein Symbol bewegen, hilft Ihnen, das richtige Symbol zu identifizieren.
- Der Projektmappen-Explorer blendet nun weitere Zweige mit entsprechenden Dateien und anderen Elementen ein – unter anderem einen namens *Verweise*.
- Öffnen Sie diesen Zweig. Dieser Zweig enthält alle Assemblies, auf die eine Windows Forms-Anwendung standardmäßig verweist; in diesem Fall aber eben nicht auf die fehlende *System.Drawing.dll*-Assembly.
- Klicken Sie mit der rechten Maustaste auf den Zweig *Verweise*, und wählen Sie aus dem Kontextmenü, das nun erscheint, *Verweis hinzufügen*.
- Es kann nun einen Augenblick dauern, bis der Dialog erscheint, den Sie auch in Abbildung 10.14 sehen. In diesem Dialog suchen Sie anschließend die .NET Framework-Assembly *System.Drawing.dll*, klicken Sie an und klicken anschließend auf *OK*.

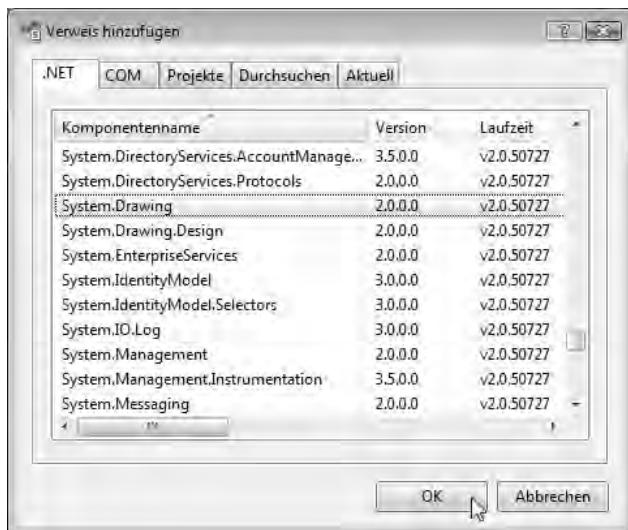


Abbildung 10.14 Mithilfe dieses Dialogs fügen Sie einen Verweis auf eine externe Assembly hinzu

Sie sehen, dass nun alle Fehler aus der Fehlerliste verschwunden sind. Ein Doppelklick auf das Formular öffnet dieses jetzt ordnungsgemäß, da der Designer nun auf die Objekte aus der gerade eingebundenen Assembly zurückgreifen kann, die für die korrekte Darstellung des Formulars und seiner Elemente erforderlich sind.

Namespaces

Nun gibt es ja, wie schon mehrfach erwähnt, nach einer Grobzählung nicht weniger als ca. 8.000 verschiedene Klassen im .NET Framework 2.0.⁷ Und alle Klassen bieten verschiedene Methoden, Eigenschaften und Ereignisse – die Anzahl an Elementen, mit denen Sie es bei größeren Projekten zu tun haben, ist also schier gewaltig. Aus diesem Grund macht es Sinn, die Klassen des Frameworks, die ja alle in verschiedenen Assemblies zu Hause sind,⁸ in irgendeiner Form zu kategorisieren. Hier kommen die Namespaces ins Spiel.

Ein Namespace ist nichts weiter als eine »Sortier- und Wiederfindhilfe« für Klassen, die sich in irgendwelchen Assemblies befinden. Namespaces lassen auch überhaupt keinen Aufschluss auf den Namen einer Assembly zu. Eine Klasse, die sich in der Assembly *xyz.dll* befindet, kann sich im gleichen Namespace befinden, wie eine ganz andere Klasse der Assembly *zyx.dll*. Genauso gut kann eine Assembly Klassen für gleich mehrere Namespaces hervorbringen.

Um zu schauen, wie es sich mit den Namespaces beim Programmieren verhält, öffnen Sie das Formular aus dem vorangegangenen Beispiel (was sich ja nun öffnen lassen sollte), und doppelklicken Sie im Designer auf die Schaltfläche.

Wir möchten nun Code entwickeln, der zur Laufzeit eine zweite Schaltfläche im Formular platziert.

Dazu deklarieren wir eine Objektvariable vom Typ Button und fügen diese der Controls-Auflistung des Formulars hinzu. Im Ergebnis sollte nach dem Programmstart ein Klick auf die schon vorhandene Schaltfläche bewirken, dass ein zweiter Button (Schaltfläche) im Formular erscheint.

Doch wenn Sie die folgende Abbildung betrachten, stellen Sie fest, dass schon die Deklaration eines Button-Objektes Probleme macht:

```

Public Class Form1
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
        Dim neueSchaltfläche As New Button
    End Sub
End Class

```

The screenshot shows a Microsoft Visual Studio code editor. A tooltip is displayed over the word 'Button' in the line 'Dim neueSchaltfläche As New Button'. The tooltip content is as follows:

- Der Typ "Button" ist nicht definiert.
- Ändern Sie "Button" in "System.Windows.Forms.Button".
- Ändern Sie "Button" in "System.Windows.Forms.VisualStyles.VisualStyleElement.Button".
- Ändern Sie "Button" in "System.Windows.Forms.VisualStyles.VisualStyleElement.ToolBar.Button".

Abbildung 10.15 Prinzipiell ist diese Zeile nicht falsch, doch findet der Compiler die Button-Klasse aus irgendwelchen Gründen nicht

⁷ Ein bekannter MS-Press-Autor hat bis zur Version 2.0 immer mitgezählt. Seine Aussage zu 3.5: »Keine Ahnung, keine Zeit mehr zum Zählen – es sind jedenfalls unfassbar viele!«. Für das Thema ASP.NET sei Ihnen sein Buch empfohlen, das im Dezember 2008 erscheint: Microsoft ASP.NET 2.0/3.5 mit Visual Basic 2008 – Das Entwicklerbuch. Mehr dazu zeigt der IntelliLink **B1002**.

⁸ Abbildung 10.14 gibt Ihnen auch einen Eindruck, wie viele verschiedene Assemblies es gibt, denn die Liste ist ja schon eindrucksvoll lang.

Die *Autokorrektur für intelligentes Kompilieren* gibt Ihnen, wie in der Abbildung zu sehen, einen Hinweis darauf, was schief läuft: Sie meint, es müsse System.Windows.Button und nicht einfach nur Button heißen. Der Hintergrund: Die Button-Klasse befindet sich in einer Assembly namens *System.Windows.Forms.dll*. Und diese Assembly definiert gleichzeitig, dass sich die Button-Klasse in einem Namespace (etwa: Namensbereich) namens *System.Windows.Forms* befindet.

Nun können Sie gleich drei Dinge tun, um den Fehler zu korrigieren:

- Sie geben den vollqualifizierten Namen der Klasse ein, also nicht nur seinen Klassennamen sondern auch den Namen des Namespaces. Das entspräche dem Korrekturvorschlag der *Autokorrektur für intelligentes Kompilieren*. Die Zeile müsste dann:

```
Dim neueSchaltfläche As New System.Windows.Forms.Button
```

lauten.

- Oder: Sie setzen eine Imports-Anweisung an den Anfang der Codedatei, die den Namensbereich für diese Codedatei einbindet. Dann können Sie innerhalb der Codedatei auf alle Klassen des importierten Namensbereiches zugreifen, ohne ständig den vollqualifizierten Namen eingeben zu müssen. Beispiel:

```
Imports System.Windows.Forms

Public Class Form1
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles Button1.Click
        Dim neueSchaltfläche As New Button ' geht jetzt ohne Fehler, wegen 'Imports'
    End Sub
End Class
```

- Oder: Sie importieren den erforderlichen Namespace global für das ganze Projekt. Das funktioniert aber nicht mit einer Anweisung im Code, sondern lässt sich über die Projekteigenschaften steuern. Rufen Sie dazu die Eigenschaften des Projektes ab (Menüpunkt *Projekt/AssemblyDemo-Eigenschaften*). Auf der Registerkarte *Verweise* finden Sie eine Liste aller eingebundenen Assemblies und darunter die für das Projekt global importierten Namespaces.

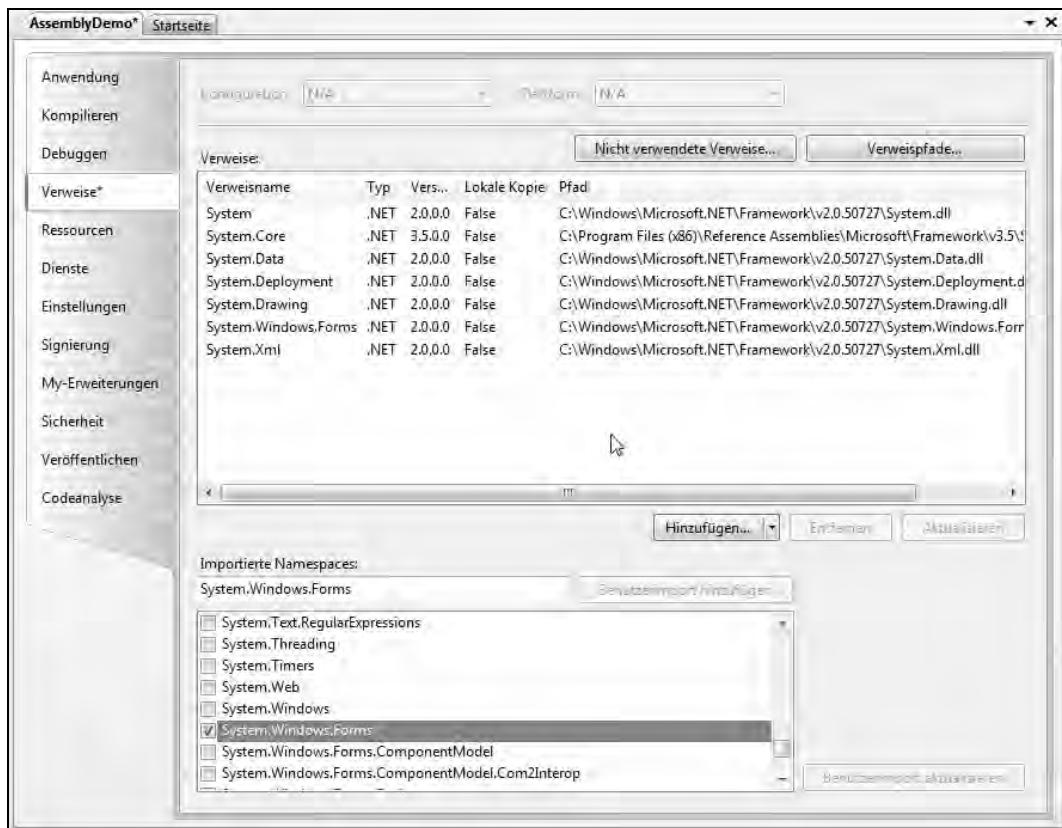


Abbildung 10.16 In dieser Liste bestimmen Sie, welche Namespaces global für das ganze Projekt eingebunden werden sollen

- Suchen Sie in dieser Liste den Namespace `System.Windows.Forms`. Anders als in der (zugegebenermaßen leicht frisierten) Abbildung wird dieser nicht oben in der Liste sondern viel weiter unten zu finden sein.
- Klicken Sie den Eintrag zweimal an (erst beim zweiten Mal erscheint das Häkchen).
- Klicken Sie auf das *Speichern*-Symbol in der Symbolleiste und schließen Sie das Eigenschaftenfenster.

Nun können Sie die Imports-Anweisung wieder aus der obersten Zeile der Codedatei löschen; der Compiler wird die Deklaration dennoch akzeptieren, weil es nun durch die projektglobalen Imports-Einstellungen über den zu verwendenden `System.Windowws.Forms`-Namespace Bescheid weiß.

Übrigens: Abbildung 10.15 gibt Ihnen einen weiteren Hinweis, wieso die Sortierung von Objekten in Namespaces so wichtig ist. Bei über 8.000 zu verwaltenden Objekten kann es nämlich schon einmal vorkommen, dass es Klassen mit gleichen Namen gibt. So gibt es ja nicht nur einen »normalen« `Button`, sondern auch einen `Toolbar.Button` – also eine Schaltfläche, die in einer Symbolleiste ihr Zuhause hat. Deswegen kann Ihnen die Autokorrektur für intelligentes Komplizieren auch an dieser Stelle nicht exakt sagen, wie Sie den Fehler korrigieren sollen. Sie weiß nämlich selber nicht, welche der drei möglichen `Button`-Klassen aus den verschiedenen Namespaces gemeint ist.

Der Vollständigkeit halber sei hier übrigens noch der Rest des Codes nachgereicht, der die neue Schaltfläche tatsächlich zur Laufzeit in das Formular zaubert.

```

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
Button1.Click

    'Neue Schaltfläche instanziiieren
    Dim neueSchaltfläche As New Button

    'Ein paar Eigenschaften setzen
    With neueSchaltfläche
        'Position:
        .Location = New Point(20, 20)
        'Größe
        .Size = New Size(150, 40)
        'Beschriftung:
        .Text = "Neue Schaltfläche"
        'Beschriftungsausrichtung:
        .TextAlign = ContentAlignment.MiddleCenter
    End With
    ' "Me" ist das Formular. Und sobald
    ' dessen Controls-Auflistung eine
    ' gültige Control-Instanz hinzugefügt
    ' wird, spiegelt sich das durch das Erscheinen des
    ' dahinter steckenden Steuerelements im Formular wider.
    Me.Controls.Add(neueSchaltfläche)
End Sub

```

HINWEIS Wie die fehlende Assembly, war auch der fehlende Imports-Verweis eine gewollte Sabotage des Projektes meinerseits. Wenn Sie ein neues Windows Forms-Projekt erstellt hätten, wäre natürlich die Assembly-Referenz auf *System.Windows.Forms.dll* vorhanden gewesen. Auch der entsprechende Namespace *System.Windows.Forms* wäre importiert gewesen. Bei größeren Projekten, in denen Sie in weit größerem Umfang von der Framework-Klassenbibliothek Gebrauch machen werden, sind aber möglicherweise nicht alle erforderlichen Assembly-Referenzen oder Namespace-Imports von vornherein vorhanden. Aber jetzt wissen Sie ja, wie Sie sich in solchen Fällen helfen können.

Um es übrigens nochmals zu wiederholen: Dass die *System.Windows.Forms.dll*-Assembly die *Button*-Klasse in einem Namespace definiert, der genau so heißt, wie die Assembly selbst, ist in diesem Fall zwar so, muss aber nicht so sein. So definiert dieselbe Assembly etwa auch eine *ResXResourceSet*-Klasse (es soll gar nicht interessieren, wozu die da ist), die sich aber im Namespace *System.Ressources* befindet.

So bestimmen Sie Assembly-Namen und Namespace für Ihre eigenen Projekte

Genauso, wie bestimmte Assemblies des Frameworks bestimmte Namen tragen und ihre Klassen bestimmten Namespaces zuordnen, können Sie das auch mit Ihren eigenen Projekten machen.

Grundsätzlich trägt die Assembly, die aus Ihrem Projekt hervorgeht, den gleichen Namen wie das Projekt selbst. Und das gilt gleichermaßen für den Namespace. Das muss aber nicht so sein. Verfahren Sie folgendermaßen, um Assembly-Namen oder Namespace-Namen für ein Projekt zu ändern:

- Rufen Sie dazu die Eigenschaften des Projektes ab (Menüpunkt *Projekt/AssemblyDemo-Eigenschaften*).

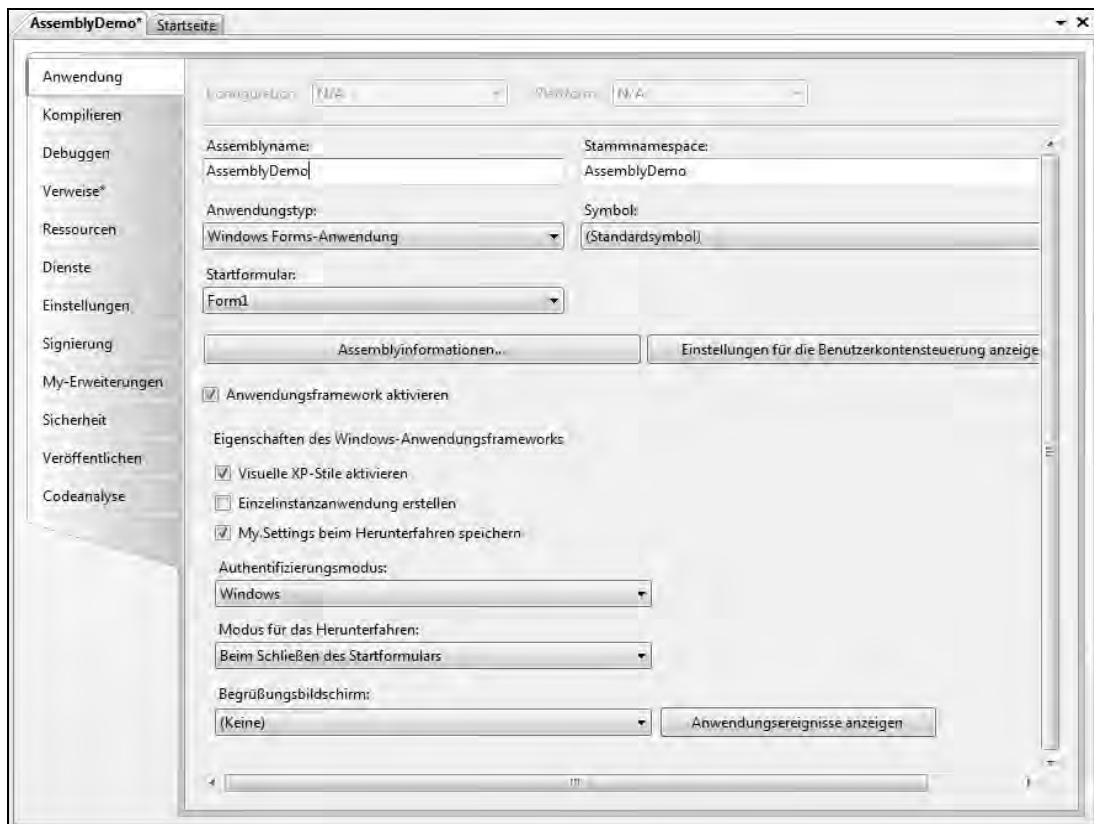


Abbildung 10.17 Unter Assemblyname definieren Sie den Namen der Assembly, der aus dem Projekt hervorgeht. Alle Klassen, die Ihre Assembly definiert, landen standardmäßig im unter Stammnamespace definierten Namespace

- Auf der Registerkarte *Anwendung* finden Sie die Felder *Assemblyname* und *Stammnamespace*, mit deren Hilfe Sie die entsprechenden Namen für Assembly und Namespace erfassen können.

Verschiedene Namespaces in einer Assembly

Und zu guter Letzt: Wie das Beispiel der *System.Windows.Forms.dll* zeigt, können Assemblies unterschiedlichen Klassen verschiedene Namespaces zuweisen. Das funktioniert auch in Ihren eigenen Projekten. Wenn Sie das Beispielprojekt der vergangenen Abschnitte noch parat haben, nehmen Sie unterhalb des Codes von *Form1* folgende Änderungen vor:

```
.
.
'
 dahinter steckenden Steuerelement im Formular wider.
 Me.Controls.Add(neueSchaltfläche)
End Sub
End Class
```

```

Namespace ZwarAssemblyDemo.AberAnderer.Namspace

    Public Class EineSimpleKlasse
        Public Function EineFunktion() As String
            Return "Rückgabertext"
        End Function
    End Class

End Namespace

```

Hier nun sehen Sie, wie Sie eine neue Klasse erstellen können, die innerhalb Ihrer Assembly aber in einem anderen Namespace liegt. Um diese Klasse, die zugegebenermaßen nicht das meiste kann, beispielsweise im Formularcode zu verwenden, gelten die gleichen Konventionen wie für externe Assemblies, die andere Namespace-Bereiche definieren: Sie müssen also entweder den vollqualifizierten Namen der Klasse eingeben, wollen Sie ein Objekt aus diesem Namespace verwenden, oder die Imports-Anweisung verwenden, um den Namespace in der Codedatei (oder einer anderen Codedatei des Projektes) zu verwenden.

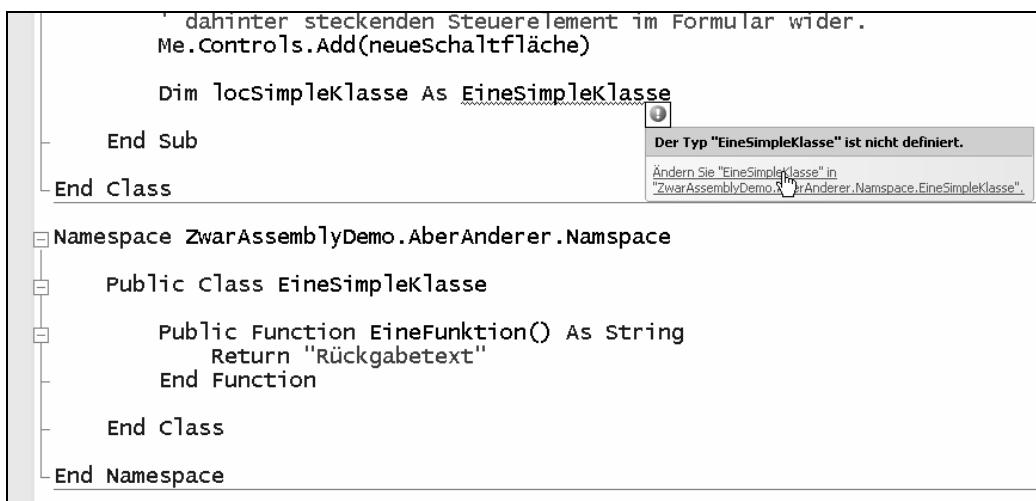


Abbildung 10.18 Um Zugriff auf eine Klasse zu nehmen, die zwar in der gleichen Assembly, aber in einem anderen Namespace liegt, müssen Sie den vollqualifizierten Klassennamen oder Imports verwenden

Zugriff auf den Framework-System-Namespace mit Global

Das Schlüsselwort `Global` gestattet Ihnen den Zugriff auf ein .NET Framework-Programmierelement auch dann, wenn Sie es mit einer Namespace-Struktur blockiert haben.

Wenn Sie eine geschachtelte Hierarchie aus Namespaces definiert haben, kann der Code innerhalb dieser Hierarchie möglicherweise nicht auf den System-Namespace von .NET Framework zugreifen. Im folgenden Beispiel wird eine Hierarchie gezeigt, in der der `SpecialSpace.System`-Namespace den Zugriff auf `System` blockiert, da er durch `System` im voll qualifizierten Namespace-Namen einen Namenskonflikt verursacht:

```
Namespace SpecialSpace
    Namespace System
        Class abc
            Function getValue() As System.Int32
                Dim n As System.Int32
                Return n
            End Function
        End Class
    End Namespace
End Namespace
```

Das führt dazu, dass der Visual Basic-Compiler den Verweis auf `System.Int32` nicht auflösen kann, da `Int32` durch `SpecialSpace.System` nicht definiert wird. Sie können das `Global`-Schlüsselwort verwenden, um die Qualifikationskette an der obersten Ebene der .NET Framework-Klassenbibliothek zu beginnen. Dies ermöglicht es Ihnen, den `System`-Namespace oder irgendeinen anderen Namespace in der Klassenbibliothek anzugeben. Dies wird anhand des folgenden Beispiels veranschaulicht:

```
Namespace SpecialSpace
    Namespace System
        Class abc
            Function getValue() As Global.System.Int32
                Dim n As Global.System.Int32
                Return n
            End Function
        End Class
    End Namespace
End Namespace
```

Sie können `Global` verwenden, um auf andere Namespaces auf Stammebene zuzugreifen, z.B. `Microsoft.VisualBasic`, sowie auf jeden anderen Namespace, der dem Projekt zugeordnet ist.

Das `Global`-Schlüsselwort kann in den folgenden Kontexten verwendet werden:

- Class-Anweisung
- Const-Anweisung
- Declare-Anweisung
- Delegate-Anweisung
- Dim-Anweisung
- Enum-Anweisung
- Event-Anweisung
- For...Next-Anweisung
- For Each...Next-Anweisung
- Function-Anweisung
- Interface-Anweisung
- Operator-Anweisung
- Property-Anweisung
- Structure-Anweisung
- Sub-Anweisung
- Try...Catch...Finally-Anweisung
- Using-Anweisung

Kapitel 11

Neues im Visual Basic 2008-Compiler

In diesem Kapitel:

First and definitely not least – die Performance des Visual Basic 2008-Compilers	362
Framework Version-Targeting (Framework-Versionszielwahl) bei Projekten	364
Lokale Typrückschlüsse (Local Type Inference)	366
If-Operator vs. IIf-Funktion	368
Festlegen der Projekteinstellungen für die Benutzerkontensteuerung (Windows Vista)	368
Nullable-Typen	369
Anonyme Typen	374
Lambda-Ausdrücke	375
Abfrageausdrücke mit LINQ	375
Erweiterungsmethoden	375
Inhalte von System- oder externen Typen in Vorschaufenstern fürs Debuggen konfigurieren	377
Zugriff auf den .NET Framework-Quellcode beim Debuggen	378

An der Sprache von Visual Basic 2008 hat sich alleine schon wegen der Fähigkeit, LINQ-Datenabfragen in der eigentlichen BASIC-Sprache genau wie For/Next-Schleifen oder If/Then-Konstrukte zu implementieren, eine ganze Menge getan. Doch das ist nicht das Einzige. So ist der Compiler an sich gründlich überarbeitet worden, und insgesamt – was Sie bei umfangreichen Projekten auch deutlich merken werden – sehr viel schneller beim Erstellen Ihrer Projekte geworden.

Microsoft hat darüber hinaus auch die Pforten zum Allerheiligsten, nämlich zum Quellcode-Verließ geöffnet, und mit kleinen Einstellungsticks entlocken Sie Visual Basic 2008 die Fähigkeit, in den originalen Quellcode zu debuggen, in dem Sie natürlich, anders als es bei Tools wie beispielsweise Reflector der Fall ist, auch die originalen Kommentare der .NET-Entwickler finden.

Dieses Kapitel stellt die Neuigkeiten rund um den Visual Basic 2008-Compiler und die neuen BASIC-Sprachelemente vor.

First and definitely not least – die Performance des Visual Basic 2008-Compilers

Es gibt eine ganze Reihe an Kleinigkeiten, die das Visual Basic 2008-Team in das neue Visual Basic 2008 integriert hat. Es gibt auch Dinge, die man nicht sieht, die aber dennoch von unglaublicher Wichtigkeit sind – und das ist die Performance des Compilers, mal ganz davon zu schweigen, dass er reibungslos funktionieren und nicht hier und da »mal« abstürzen sollte.

Letzteres war nämlich vor dem Service Pack 1 beim Visual Basic 2005-Compiler gerade auf Mehrprozessor- oder Multi-Core-Prozessor-Systemen bei größeren und großen Projekten sehr häufig der Fall. Mitunter mussten Projekte sogar liegen bleiben, bis die ersten Patches für diese »Race-Condition«, die sich zwischen Editor und Background-Compiler wohl entwickelte, vom Visual Basic-Team »geklärt« war.

Doch das ist glücklicherweise Schnee von gestern. Heute geht es nicht darum, dass der Compiler richtig funktioniert (denn das tut er!), sondern wie schnell. Hier ist in Sachen Produktivität einiges an Steigerungspotential herauszuholen, und die Jungs und Mädels aus Redmond haben sich das richtig zu Herzen genommen; im Falle von Visual Basic 2008 übrigens eher sogar die Mädels.

Im Folgenden finden Sie eine Tabelle,¹ die die Performance-Unterschiede zum Visual Basic 2005-Compiler dokumentiert – diese Tabelle stammt übrigens ursprünglich aus einem Blog von Lisa Feigenbaum, einer der Entwicklerinnen im Visual Basic 2008-Team.

Verstehen Sie die Tabelle bitte nicht falsch: Es geht dabei nicht um die Geschwindigkeit, mit der die späteren Programme laufen, sondern wie schnell Background-Compiler und Main-Compiler arbeiten und damit, wie groß (oder: klein) die Turn-Around-Zeiten beim Kompilierungsprozess sind.

¹ Quelle: Lisa Feigenbaum, IntelliLink **B1101**, Stand 15.10.2008, gemessen auf einer P4-Dual-Core-Maschine mit 3 GHZ, 1GBYTE Hauptspeicher und einer Festplatte mit 10K U/min.

Szenario	VB2005 (ms)	VB2008 (ms)	VB2008 ist x-mal schneller als VB2005	VB2008 braucht x% der Zeit von VB2005
Build eines umfangreichen Projektes (bei Verwendung von Hintergrund-Kompilierung mit dem Background-Compiler).	222.206,25	1.352,88	164,25	0,61%
Build einer großen Projektmappe mit mehreren Projekten (explizite Build-Erstellung).	1.618.604,75	57.542,75	28,13	3,56%
Build einer großen Projektmappe mit mehreren Projekten (bei Verwendung von Hintergrund-Kompilierung mit dem Background-Compiler).	222.925,50	19.861,88	11,22	8,91%
Reaktionszeit nach dem Hinzufügen eines Members zu einer Klasse.	327,00	36,50	8,96	11,16%
Reaktionszeit nach dem Öffnen eines Projekts.	255.551,25	38.769,38	6,59	15,17%
IntelliSense aufrufen, um die Typenliste anzeigen zu lassen (erster Aufruf).	1.192,50	530,5	2,25	44,49%
Edit-and-Continue in einer Projektmappe, die XML-Kommentare enthält (erstes Mal).	441,25	210,5	2,10	47,71%
Reaktionszeit nach dem Ändern eines Methoden-Statements.	390,25	236,38	1,65	60,57%
10 Schritte im Debugger (kummulierte Zeiten).	1.850,75	1.167,13	1,59	63,06%
IntelliSense aufrufen, um eine Typenliste zu bearbeiten (Folgezeiten).	79,25	51,5	1,54	64,98%
Ausführen nach F5 wenn die Projektmappe bereits vollständig erstellt wurde.	385,20	278,7	1,38	72,35%
Zeit, nachdem ein Fehler der Fehlerliste hinzugefügt wird.	531,25	394,5	1,35	74,26%
10 Schritte im Debugger (zum ersten Mal).	1.336,50	1.150	1,16	86,05%
Reaktionsverhalten, während der Hintergrundcompiler eine offene Projektmappe verarbeitet.	4.803,00	4.284,75	1,12	89,21%
Laden einer umfangreichen Projektmappe.	13.667,5	12.407,25	1,10	90,78%
Laden einer umfangreichen Projektmappe (erstmalig).	19.946,25	18.222	1,09	91,36%
HINWEIS: Entspricht der Verbesserung auf Windows XP. Unter Vista ist der Geschwindigkeitsvorteil noch mal doppelt so schnell.				

Framework Version-Targeting (Framework-Versionszielwahl) bei Projekten

Visual Studio 2008 bzw. Visual Basic 2008 erlaubt, dass Sie das Framework, mit dem Sie Ihre Anwendung, die Sie entwickeln, bestimmen können. Was bedeutet das genau?

Wenn Sie mit Visual Studio 2002 gearbeitet haben, haben Sie, weil es nichts anderes gab, mit dem Framework 1.0 entwickelt. Als dann Visual Studio 2003 auf den Markt kam, gab es gleichzeitig das Framework 1.1. Mit dieser Visual Studio-Version haben Sie automatisch gegen dieses Framework entwickelt. Mit Visual Studio 2005 war es automatisch das Framework 2.0.

Schon zu Visual Studio 2005-Zeiten wäre es wünschenswert gewesen, zwar die damals neue Entwicklungsumgebung zu verwenden, aber dennoch, aus Abwärtskompatibilitätsgründen, gegen eine ältere Framework-Version zu entwickeln.

Mit Visual Studio 2008 ist das möglich. Sie können zwar nicht mehr gegen das 1.0er oder das 1.1er-Framework entwickeln, aber für alle neueren Versionen – 2.0, 3.0 und 3.5 – ist die Auswahl zunächst einmal beim Erstellen des Projektes möglich. Bedenken Sie bei der Auswahl der Framework-Version Folgendes:

- Wenn Sie gegen das Framework 2.0 entwickeln, können Sie auch Computer mit älteren Betriebssystemen bedienen, wie beispielsweise Windows 2000 (nur mit SP4!), und eingeschränkt sogar noch Windows 98. Sie müssen dann allerdings auf LINQ-Unterstützung (dafür ist 3.5 erforderlich) und auch auf die Windows Presentation Foundation (WPF), die Windows Communication Foundation (WCF) und WF (Windows Workflow) verzichten (dafür ist 3.0 erforderlich).
- Wenn Sie gegen das Framework 3.0 entwickeln, können Sie auf die .NET Framework-Bibliothek für die WPF, WCF und WF zurückgreifen; auf älteren Windows-Versionen läuft Ihre Anwendung dann allerdings nicht mehr: Sie benötigen mindestens Windows Server 2003 bzw. Windows XP mit Service Pack 2. Für Windows Vista ist keine Framework-Installation erforderlich, da das Framework bereits Bestandteil dieses Betriebssystems ist.
- Entwickeln Sie gegen das Framework 3.5, sind Sie ebenfalls mit Windows XP SP2 und Windows Server 2003 dabei – dieses Framework muss allerdings gezielt auf dem System, auf dem Sie Ihre Anwendung laufen lassen wollen, mit installiert werden – auch auf Windows Vista und Windows Server 2008. Mit dem Framework 3.5 steht Ihnen zusätzlich die komplette LINQ-Funktionalität auf dem Zielsystem zur Verfügung.

Framework Version-Targeting (Framework-Versionszielwahl) bei Projekten



Abbildung 11.1 Beim Erstellen eines neuen Projekts wählen Sie aus der Aufklappliste am rechten, oberen Dialogrand, gegen welche Framework-Version Ihr neues Projekt abzielen soll

Beim Erstellen eines neuen Projektes wählen Sie aus dem Menü *Datei* den Menüpunkt *Neu* und weiter *Projekt*. Visual Studio zeigt Ihnen anschließend einen Dialog (siehe Abbildung 11.1), mit dem Sie nicht nur wie gewohnt ein neues Projekt erstellen können, sondern ebenfalls aus der rechts oben zu findenden Aufklappliste die Framework-Version auswählen können, gegen die Ihr Projekt abzielen soll.

Sollten Sie hingegen bereits ein Projekt angelegt haben, und sich anschließend noch für eine andere Framework-Version entscheiden, dann verfahren Sie wie folgt:

1. Klicken Sie das entsprechende Projekt mit der *rechten Maustaste* im Projektmappen-Explorer an.
2. Im Kontextmenü, das jetzt aufklappt, wählen Sie *Eigenschaften*.
3. Wählen Sie die Registerkarte *Kompilieren* im Eigenschaftendialog.
4. Klicken Sie auf *Erweiterte Kompilierungsoptionen*. Bei einer kleineren Bildschirmauflösung müssen Sie ggf. das Fenster nach unten scrollen, um die Schaltfläche sehen zu können.
5. Aus der Aufklappliste *Zielframework (alle Konfigurationen)* wählen Sie die Framework-Version, gegen die Sie entwickeln wollen (siehe auch Abbildung 11.2).
6. Bestätigen Sie das anschließend erscheinende Meldungsfeld mit *OK*. Damit wird Ihr Projekt bzw. die Projektmappe kurz entladen und anschließend wieder geladen – Sie entwickeln ab jetzt gegen die ausgewählte Framework-Version.

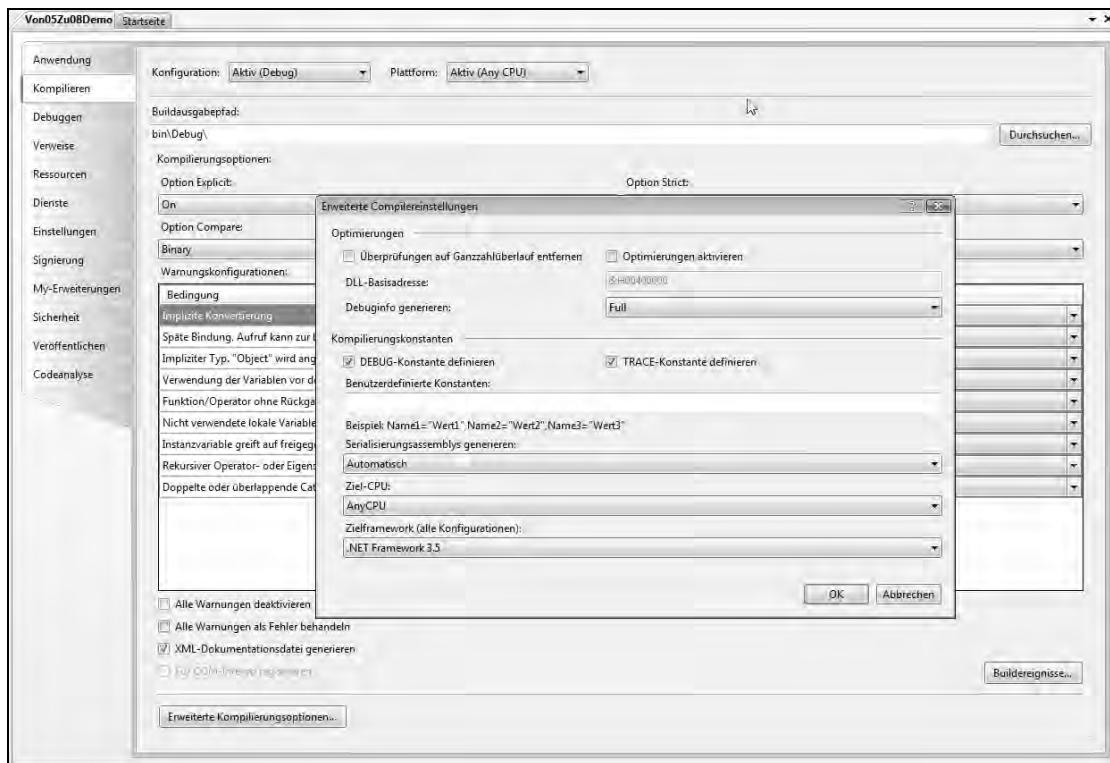


Abbildung 11.2 Falls Sie die Version des Zielframeworks Ihres Projektes ändern möchten, wählen Sie in den Projekteigenschaften das Register Kompilieren, klicken Erweiterte Kompilierungsoptionen und ändern die Framework-Version in der Klappliste Zielframework

Lokale Typrückschlüsse (Local Type Inference)

Visual Basic 2008 erlaubt, dass Typen auch aufgrund ihrer initialen Zuweisungen festgelegt werden. Ganz eindeutig wird das beispielsweise an der Zuweisung

```
Dim blnValue = True
```

Wenn Sie einer primitiven Variable den Wert True zuweisen und dazu noch Typsicherheit definiert ist, dann muss es sich bei der Variablen einfach um den booleschen Datentyp handeln. Genau so ist das bei

```
Dim strText = "Eine Zeichenkette."
```

strText *muss* eine Zeichenkette sein – das bestimmt die Zuweisung. Anders ist es bei numerischen Variablen. Hier muss man wissen, dass durch Zuweisung einer ganzen Zahl an eine bislang noch nicht typbestimmte Variable, der Integer-Typ definiert wird, durch Zuweisung einer Fließkommazahl der Double-Typ. Doch diese Standardtypen von Konstanten gab es vorher schon – letzten Endes bestimmen die Konstanten mit ihren Typliteralen, welchen Typ sie darstellen.

```
Dim einInteger = 100 ' Integer, einfache Fließkommalose Zahl definiert Integerkonstante
Dim einShort = 101S ' Short, weil das Typliteral S eine Short-Konstante bestimmt
Dim einSingle = 101.5F ' Single, weil das Typliteral F eine Single-Konstante bestimmt
```

WICHTIG Lokaler Typrückschluss funktioniert übrigens nur auf Prozedurebene, nicht auf Klassenebene (deswegen auch die Bezeichnung »lokaler« Typrückschluss).

Gesteuert wird der lokale Rückschluss übrigens durch Option Infer, die als Parameter Off oder On übernimmt (von engl. *Inference*, etwa: *der Rückschluss*). Standardmäßig ist der lokale Typrückschluss eingeschaltet.

Sie können sie also durch die entsprechende Anweisung

```
Option Infer Off
```

direkt am Anfang der Codedatei eben nur für die Klassen und Module dieser Codedatei oder aber global für das ganze Projekt ausschalten (oder eben anschalten).

Generelles Einstellen von Option Infer, Strict, Explicit und Compare

Um Einstellungen von Option Infer, aber auch von Option Strict (Typsicherheit), Option Explicit (Deklarationszwang von Variablen) und Option Compare (Vergleichsverhalten) für ein ganzes Projekt global durchzuführen, öffnen Sie das Kontextmenü des Projektes (nicht der Projektmappe!) im Projektmappen-Explorer und wählen den Menüpunkt *Eigenschaften* aus.

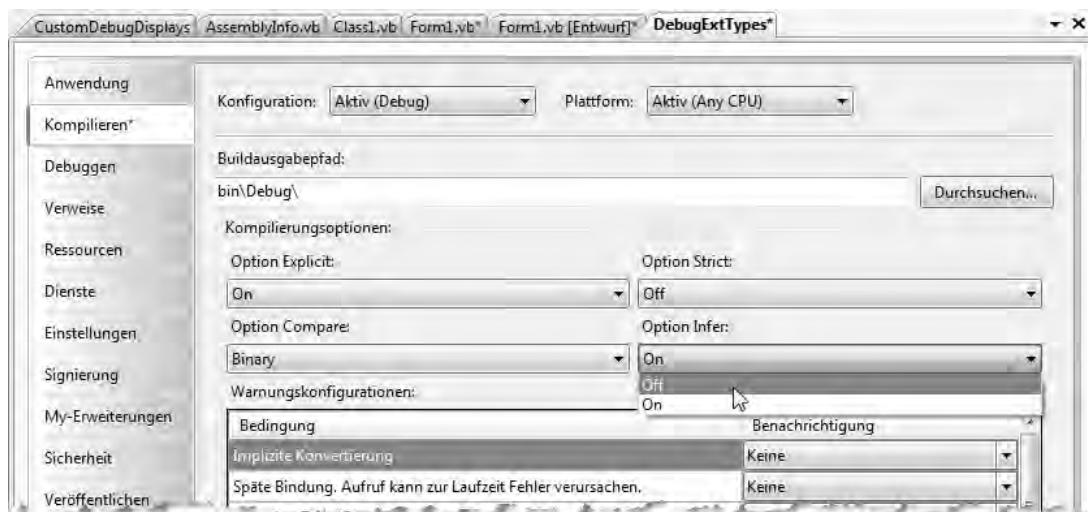


Abbildung 11.3 Im Eigenschaftendialog des Projektes stellen Sie das Option XXX-Verhalten auf der Registerkarte Kompilieren projektglobal ein

If-Operator vs. IIf-Funktion

Einfacher ist auch der Umgang mit IIf geworden. Bislang mussten Sie, wenn Sie die IIf-Funktion (mit zwei »i«) verwendet haben, das Ergebnis in den Typ casten, der der Zuweisung entsprach, da die IIf-Funktion nur Object zurücklieferte, also beispielsweise:

```
Dim c As Integer
'Liefert 10 zurück
c = CInt(IIf(True, 10, 20))
```

Das geht jetzt einfacher, denn das Schlüsselwort If (mit einem »i«) ist für den Gebrauch als Operator erweitert worden:

```
Dim c As Integer
'Liefert 20 zurück
c = If(False, 10, 20)
```

Noch weniger Schreibaufwand verursacht das, wenn Sie den If-Operator mit lokalem Typrückschluss kombinieren:

```
'Liefert 20 zurück
Dim c = If(False, 10, 20)
```

Das Mischen von verschiedenen Typen bei der Typrückgabe bringt den Compiler allerdings ins Straucheln, wie in der folgenden Abbildung zu sehen:

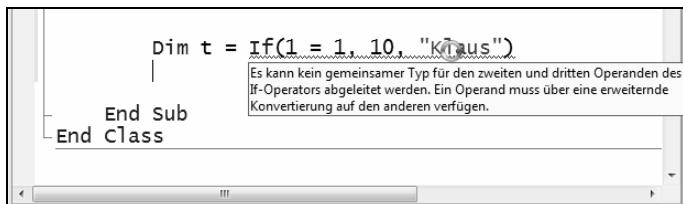


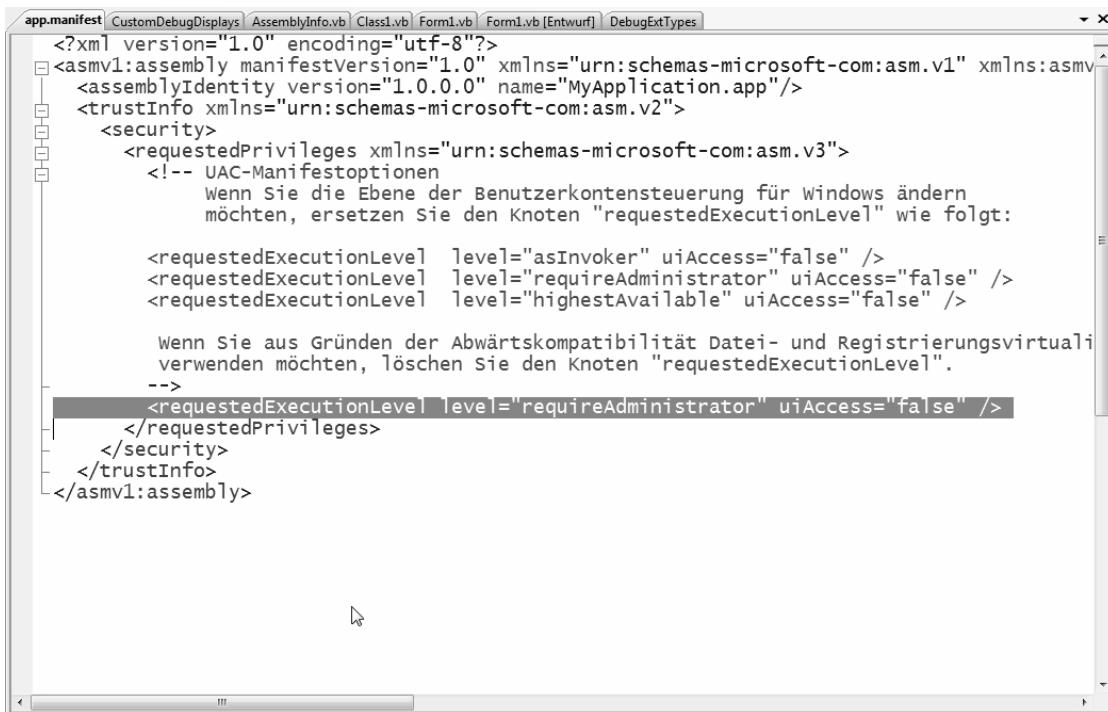
Abbildung 11.4 If als Ersatz für IIf funktioniert nur dann, wenn Sie dem Compiler die Chance geben, die entsprechenden Typen eindeutig zu ermitteln

Festlegen der Projekteinstellungen für die Benutzerkontensteuerung (Windows Vista)

Wenn Sie es wünschen, können Sie für ausführbare Programme die Regeln für die Benutzerkontensteuerung für Betriebssysteme ab Windows Vista oder Windows Server 2008 festlegen. Wenn nichts anderes gesagt wird, starten Ihre Anwendungen (sofern es die Rechte des jeweiligen Anwenders überhaupt erlauben, die Anwendung zu starten) im Kontext des angemeldeten Benutzers. Sie können allerdings bestimmen, dass höhere Rechte als die vorhandenen des Benutzers erforderlich sein sollen.

- Dazu klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf das Projekt (nicht auf die Projektmappe!), und wählen aus dem Kontextmenü, das jetzt erscheint, *Eigenschaften*.
- Im Dialog, den Sie jetzt sehen, klicken Sie auf *Einstellungen für die Benutzerkontensteuerung anzeigen*.

- Visual Studio zeigt Ihnen jetzt die Manifest-Datei Ihrer Anwendung an.
- Ändern Sie den Eintrag *level* für *requestedExecutionLevel* auf die gewünschte Einstellung. Möglich sind dabei *asInvoker* (als Aufrufer – dies ist die Standardeinstellung), *requireAdministrator* (Administrator erforderlich) sowie *highestAvailable* (höchst möglich).



```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1" xmlns:asmv1="urn:schemas-microsoft-com:asm.v1" xmlns:asmv2="urn:schemas-microsoft-com:asm.v2" xmlns:security="urn:schemas-microsoft-com:asm.v3">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <!-- UAC-Manifestoptionen
        Wenn Sie die Ebene der Benutzerkontensteuerung für Windows ändern möchten, ersetzen Sie den Knoten "requestedExecutionLevel" wie folgt:
        <requestedExecutionLevel level="asInvoker" uiAccess="false" />
        <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
        <requestedExecutionLevel level="highestAvailable" uiAccess="false" />
        Wenn Sie aus Gründen der Abwärtskompatibilität Datei- und Registrierungsvirtualisierung verwenden möchten, löschen Sie den Knoten "requestedExecutionLevel".
        -->
        <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</asmv1:assembly>
```

Abbildung 11.5 In der Manifest-Datei Ihrer Anwendung bestimmen Sie die erforderlichen Einstellungen für die Benutzerkontensteuerung unter Windows Vista bzw. Windows Server 2008 (oder höher)

Nullable-Typen

Nullable-Typen gab es zwar schon in Visual Basic 2005, aber sie waren nur – sagen wir einmal – halbherzig in Visual Basic implementiert. Zwar erfuhren sie schon eine (notwendige!) Sonderbehandlung durch die CLR (siehe Abschnitt »Besonderheiten bei Nullable beim Boxen« ab Seite 372), aber anders als in C# waren sie noch nicht mit einem eigenen Typliteral implementiert. Das hat sich geändert.

Nullable ist ein generischer Datentyp mit einer Einschränkung auf Wertetypen. Er ermöglicht, dass ein beliebiger Wertetyp neben seiner eigentlichen Werteart einen weiteren Zustand »speichern« kann – nämlich *Nothing*.

HINWEIS Generics gibt es zwar schon seit der Version 2005, jedoch sind sie vielen Entwicklern noch nicht geläufig. Generics und der Anwendung von Generics ist deswegen ein eigenes Kapitel, nämlich das Kapitel 23 gewidmet.

Ist das wichtig? Oh ja! Beispielsweise in der Datenbankprogrammierung. Wenn Sie bereits Erfahrungen in der Datenbankprogrammierung haben, wissen Sie auch sicherlich, dass Ihre Datenbanktabellen über Datenfelder verfügen können, die den »Wert« Null »speichern« können – als Zeichen dafür, dass eben *nichts* (auch nicht die Zahl 0) in diesem Feld gespeichert wurde.

Ein anderes Beispiel sind CheckBox-Steuerelemente in Windows Forms-Anwendungen: Sie verfügen über einen Zwischenzustand, der den Zustand »nicht definiert« anzeigen soll. Eine einfache boolesche Variable könnte alle möglichen Zustände nicht aufnehmen – True und False sind dafür einfach zu wenig. Anders ist es, wenn Sie eine Variable vom Typ Boolean definieren würden und könnten, die auch den »Nichts-ist-gespeichert«-Wert widerspiegeln könnte.

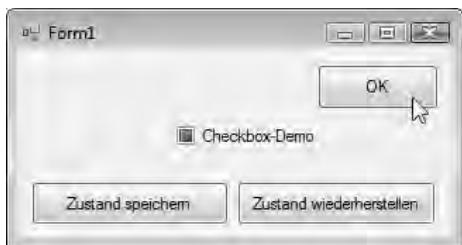


Abbildung 11.6 Ein Boolean eignet sich beispielsweise dazu, auch Zwischenzustände eines CheckBox-Steuerelements – wie hier im Bild zu sehen – zu speichern

Und das geht: In Visual Basic 2008 definiert man eine primitive Variable als Nullable-Datentyp, indem man ihrem Variablennamen ein Fragezeichen als Typliteral anhängt. Das sieht entweder so ...

```
Private myCheckBoxZustand As Boolean?
```

... oder so aus:

```
Private myCheckBoxZustand? As Boolean
```

In Visual Basic 2005 war es notwendig, Nullables auf folgende Weise zu definieren:

```
Private myCheckBoxZustand As Nullable(Of Boolean)
```

Aber keine Angst: Sie müssen sich jetzt nicht durch hunderte, vielleicht tausende Zeilen Code hangeln, und für Visual Basic 2008 die entsprechenden Änderungen mit dem Fragezeichen vornehmen. Bei allen drei Varianten kommt nämlich exakt die gleiche Variante heraus.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\\VB 2008 Entwicklerbuch\\B - Migration\\Kapitel 11\\NullableUndCheckbox
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Beispiel demonstriert, wie alle Zustände eines CheckBox-Steuerelements, dessen ThreeState-Eigenschaft zur Anzeige aller *drei* Zustände auf True gesetzt wurde, in einer Member-Variablen vom Typ Boolean? gespeichert werden können. Klicken Sie auf *Zustand speichern*, um den Zustand des CheckBox-Steuerelements

in der Member-Variablen zu sichern, verändern Sie anschließend den Zustand, und stellen Sie den ursprünglichen Zustand des CheckBox-Steuerelements mit der entsprechenden Schaltfläche wieder her.

Der entsprechende Code dazu lautet folgendermaßen:

```
Public Class Form1

    Private myCheckBoxZustand As Boolean?

    'Das ginge auch:
    'Private myCheckBoxZustand? As Boolean

    'Und das auch:
    'Private myCheckBoxZustand As Nullable(Of Boolean)

    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
        btnOK.Click
        Me.Close()
    End Sub

    Private Sub btnSpeichern_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
        btnSpeichern.Click

        If chkDemo.CheckState = CheckState.Indeterminate Then
            myCheckBoxZustand = Nothing
        Else
            myCheckBoxZustand = chkDemo.Checked
        End If
    End Sub

    Private Sub btnWiederherstellen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
        btnWiederherstellen.Click
        If Not myCheckBoxZustand.HasValue Then
            chkDemo.CheckState = CheckState.Indeterminate
        Else
            If myCheckBoxZustand.Value Then
                chkDemo.CheckState = CheckState.Checked
            Else
                chkDemo.CheckState = CheckState.Unchecked
            End If
        End If
    End Sub
End Class
```

Die Zeilen, in denen die Member-Variable Boolean? zum Einsatz kommt, sind im Listing fett markiert. Dabei fällt Folgendes auf:

- **Wertzuweisung:** Wenn Sie einen Wert des zugrunde liegenden Typs an type? zuweisen wollen, können Sie die implizite Konvertierung verwenden, den entsprechenden Wert also direkt zuweisen, etwa wie in der Zeile

```
myCheckBoxZustand = chkDemo.Checked
```

zu sehen.

- **Auf Nothing zurücksetzen:** Möchten Sie eine Nullable-Instanz auf Nothing zurücksetzen, weisen Sie ihr einfach den »Wert« Nothing zu – wie im Listing an dieser Stelle zu sehen:

```
myCheckBoxZustand = Nothing
```

- **Auf Wert prüfen:** Möchten Sie wissen, ob eine Nullable-Instanz einen Wert oder Nothing enthält, verwenden Sie deren Eigenschaft HasValue. Auch dafür gibt es ein Beispiel im Listing:

```
If Not myCheckBoxZustand.HasValue Then
    chkDemo.CheckState = CheckState.Indeterminate
Else
    .
    .
    .
```

- **Wert abrufen:** Und schließlich müssen Sie natürlich auch den Wert, den eine Nullable-Instanz trägt, wenn sie nicht Nothing ist, ermitteln können. Dazu dient die Eigenschaft Value. Ein Beispiel dafür:

```
If myCheckBoxZustand.Value Then
    chkDemo.CheckState = CheckState.Checked
Else
    chkDemo.CheckState = CheckState.Unchecked
End If
.
.
.
```

HINWEIS Erst in diesem Beispiel fiel auf, dass man offensichtlich ein CheckBox-Steuerelement, dessen ThreeState-Eigenschaft gesetzt ist und das den *Intermediate*-Zustand momentan trägt, nicht mit seiner Checked-Eigenschaft in einen anderen Zustand versetzen kann (Checked oder Unchecked). Sie können in diesem Fall nur die CheckState-Eigenschaft verwenden, um das CheckBox-Steuerelement programmgesteuert wieder aus dem *Intermediate*-Zustand herauszuholen!

Besonderheiten bei Nullable beim Boxen

Der Datentyp `Nullable` (`type?`) ist das, was man in Visual Basic als Struktur programmieren würde, also ein Wertetyp. Doch Sie könnten diesen Wertetyp nicht 1:1 nachprogrammieren, denn er erfährt durch die Common Language Runtime eine besondere Behandlung – und das ist auch gut so.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2008 Entwicklerbuch\B - Migration\Kapitel 11\NullableDemo
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie eine Instanz einer beliebigen Struktur – also eines beliebigen Wertetyps – verarbeiten, kommt irgendwann der Zeitpunkt, an dem Sie diesen Wertetyp in einer Objektvariablen boxen müssen – beispielsweise wenn Sie ihn als Bestandteil eines Arrays oder einer Auflistung (Collection) speichern.

Wann immer Sie einen definierten Wertetyp in einem Objekt boxen, kann dieses Objekt logischerweise nicht `Nothing` sein, ganz egal, welchen »Wert« diese Struktur hat. Im Falle des `Nullable`-Typs ist das anders, wie das folgende Beispiel zeigt:

```
Module NullableDemo

Sub Main()
    Dim locObj As Object
    Dim locNull0fInt As Integer? = Nothing

    'Es gibt natürlich eine verwendbare Instanz, denn
    'Integer? ist ein Wertetyp!
    Console.WriteLine("Hat locNull0fInt einen Wert: " & locNull0fInt.HasValue)

    'Und dennoch ergibt das folgende Konstrukt True,
    'als würde locObj keine Referenz haben!
    locObj = locNull0fInt
    Console.WriteLine("Ist locObj Nothing? " & (locObj Is Nothing).ToString)
    Console.WriteLine()

    'Und auch das "Entboxen" geht!
    'Es gibt keine Null-Exception!
    locNull0fInt = DirectCast(locObj, Integer?)

    'Und geht das dann auch? - Natürlich!
    locNull0fInt = DirectCast(Nothing, Integer?)

    'Und noch weiter. Wir boxen einen Integer?
    locNull0fInt = 10
    '

    locObj = locNull0fInt
    Dim locInt As Integer = DirectCast(locObj, Integer)

    Console.WriteLine("Taste drücken zum Beenden!")
    Console.ReadKey()

    'Das geht übrigens nicht, obwohl Nullable die
    'Constraints-Einschränkung im Grunde genommen erfüllt!
    'Dim locNull0fInt As Nullable(Of Integer?)
End Sub
End Module
```

Wenn Sie dieses Beispiel ausführen, gibt es die folgenden Zeilen aus:

```
Hat locNull0fInt einen Wert: False
Ist locObj Nothing? True

Taste drücken zum Beenden!
```

Das, was hier passiert, ist beileibe keine Selbstverständlichkeit – aber dennoch sauberes Design der CLR, denn: Zwar wird `locNull0fInt` nicht initialisiert (oder, um es in diesem Beispiel deutlich zu machen, mit `Nothing` – aber das kommt auf dasselbe raus), aber natürlich existiert dennoch eine Instanz der Struktur. Sie spiegelt eben nur den Wert `Nothing` wider. Gemäß den bekannten Regeln müsste das anschließende Boxen in der Variablen `locObj` auch ergeben, dass `locObj` einen Zeiger auf eine Instanz der `Nothing` widerspiegelnden `locNull0fInt` enthält und keinen Null-Zeiger. Doch das ist nicht der Fall, denn die anschließende Ausgabe von

```
Console.WriteLine("Ist locObj Nothing?" & (locObj Is Nothing).ToString)
```

zeigt

```
Ist locObj Nothing? True
```

auf dem Bildschirm an.

Das »zurückcasten« von `Nothing` in `Nullable` ist damit natürlich genauso gestattet, wie ebenfalls im Listing zu sehen.

Und noch eine Unregelmäßigkeit erfahren `Nullable`-Datentypen, nämlich wenn es darum geht, einen geboxeden Typ (vorausgesetzt, er ist eben nicht `Nothing`) in seinen Grundtyp zurückzucasten, wie der folgende Codeausschnitt zeigt:

```
'Und noch weiter. Wir boxen einen Integer?  
locNullOfInt = 10  
'  
locObj = locNullOfInt  
Dim locInt As Integer = DirectCast(locObj, Integer)
```

Hier wird ein `Nullable`-Datentyp in einem Objekt geboxt, aber später zurück in seinen *Grunddatentypen* gewandelt. Ein, wie ich finde, logisches Design, was allerdings dem »normalen« Vorgehen beim Boxen von Wertetypen in Objekten völlig widerspricht.

HINWEIS Kleine Anekdote am Rande: Dieses Verhalten ist erst zu einem sehr, sehr späten Zeitpunkt beim Entwickeln von Visual Studio 2005 und dem Framework in die CLR eingebaut worden und hat für erhebliche Mehrarbeit bei allen Entwicklungsteams und viel zusätzlichen Testaufwand gesorgt. Dass sich Microsoft dennoch für das nunmehr implementierte Verhalten entschieden hat, geht nicht zuletzt auf das Drängen von Kunden und Betatestern zurück, die das Design mit der ursprünglichen, »normalen« CLR-Behandlung von Nullables nicht akzeptieren konnten und als falsch erachteten.

Anonyme Typen

Mit anonymen Typen können eine Reihe schreibgeschützter Eigenschaften in einem einzelnen Objekt gekapselt werden, ohne zuerst einen konkreten Typ – also eine Klasse oder eine Struktur – dafür explizit definieren zu müssen. Natürlich entsteht durch den Compiler ein .NET-konformer Typ bei diesem Vorgang, doch der Typename wird vom Compiler generiert, und er ist nicht auf Quellcodeebene verfügbar. Der Typ der Eigenschaften wird vom Compiler abgeleitet.

Der Sinn von anonymen Typen wird nur im Rahmen von LINQ deutlich: Sie werden normalerweise in der `Select`-Klausel eines Abfrageausdrucks verwendet, um einen Typen zu erstellen, der ein untergeordnetes Set der Eigenschaften enthält, die aus jedem Objekt, das in der Abfrage berücksichtigt wird, quasi »vorkommen«.

Wie Sie anonymous Typen genau verwenden, erfahren Sie sinnvollerweise im richtigen Kontext und deswegen im LINQ-Teil dieses Buchs – an dieser Stelle seien sie nur der Vollständigkeit halber erwähnt.

Kapitel 31 zeigt Ihnen im Einführungsteil von LINQ, wie Sie anonymous Typen im Zusammenhang mit den LINQ-Erweiterungsmethoden anwenden können.

Lambda-Ausdrücke

Bei Lambda-Ausdrücken handelt es sich um Funktionen, die nur im Kontext definiert werden, die Ausdrücke und Anweisungen enthalten und die für die Erstellung von Delegaten oder Ausdrucksbaumstrukturen verwendet werden können. Man nennt sie auch anonyme Funktionen, weil sie Funktionen bilden, die aber selbst keinen Namen haben. Gerade wieder mit Schwerpunkt auf LINQ ist es oft notwendig, an bestimmten Stellen Delegaten – also Funktionszeiger – einzusetzen, die aber ausschließlich bei einem konkreten Aufruf und mit einer Minimalausstattung an Code zureckkommen.

Der Visual Basic-Compiler kennt Lambda-Ausdrücke seit der Version 2008 – sie seien an dieser Stelle aber nur der Vollständigkeit halber erwähnt. In Kapitel 31 finden Sie die ausführliche Erklärung von Lambda-Ausdrücken im richtigen Kontext von Delegaten und Klassen und mit anschaulichen Praxisbeispielen erklärt.

Abfrageausdrücke mit LINQ

LINQ (*Language integrated Query*, etwa: *sprachintegrierte Abfrage*) ist eine mächtige Erweiterung des VB-Compilers, und LINQ ermöglicht es, beispielsweise Datenabfragen, Datensortierungen und Datenselektionen durchzuführen, beispielsweise von Business-Objekten, die in Auflistungen gespeichert sind, aber auch von Daten, die in SQL Server-Datenbanken gespeichert sind.

Mit dem Service Pack 1 von Visual Basic 2008 ist es darüber hinaus möglich, LINQ-Abfragen für weitaus mehr Daten-Provider durchzuführen, für die eine ADO.NET-Implementierung vorhanden ist (*LINQ to Entities*); auch die Implementierung eigener LINQ-Provider ist damit möglich sein.

Dem Thema LINQ ist ein eigener Buchteil gewidmet; da LINQ jedoch den größten Aufwand bei der Entwicklung des Visual Basic 2008-Compilers darstellte, sei das Thema im Rahmen dieses Kapitels zumindest der Vollständigkeit halber erwähnt.

Erweiterungsmethoden

Prinzipiell gab es schon immer Möglichkeiten, Klassen um neue Methoden oder Eigenschaften zu erweitern. Sie vererbten sie, und fügten ihnen anschließend im vererbten Klassencode neue Methoden oder Eigenschaften hinzu. Das funktionierte solange, wie es sich um Typen handelte, die ...

- ... nicht mit dem `NotInheritable`-Modifizierer (etwa: *nicht vererbbar*; der Vollständigkeit halber: `Sealed` in C#, etwa: *versiegelt*) gekennzeichnet waren, oder
- ... keine Wertetypen waren – denn mit `Structure` erstellte Typen sind automatisch Wertetypen und die sind implizit nicht vererbbar.

Für eine bessere Strukturierung ihres Codes wurden daher in Visual Basic 2008 sogenannte Erweiterungsmethoden eingeführt, mit deren Hilfe Entwickler bereits definierten Datentypen benutzerdefinierte Funktionen hinzufügen können, ohne eben einen neuen, vererbten Typ zu erstellen. Erweiterungsmethoden sind eine besondere Art von statischen Methoden, die Sie jedoch wie Instanzmethoden für den erweiterten Typ aufrufen können. Für in Visual Basic (und natürlich auch C#) geschriebenen Clientcode gibt es keinen sichtbaren Unterschied zwischen dem Aufrufen einer Erweiterungsmethode und den Methoden, die in einem Typ tatsächlich definiert sind.

Allerdings gibt es dabei Einschränkungen bzw. Konventionen, was das Erweitern von Klassen auf diese Weise anbelangt:

- Erweiterungsmethoden müssen mit einem Attribut besonders gekennzeichnet werden. Dazu dient das ExtensionAttribut, das Sie im Namespace System.Runtime.CompilerServices finden.
- Bei einer Erweiterungsmethode kann es sich ausschließlich um eine Sub oder eine Function handeln. Sie können eine Klasse um Erweiterungsmethoden, *nicht* um Eigenschaften, neue Member-Variablen (Felder) oder Ereignisse erweitern.
- Eine Erweiterungsmethode muss sich in einem gesonderten Modul befinden.
- Im Bedarfsfall muss das Modul, sollte es sich in einem anderen Namespace befinden, als die Instanz, die es verwendet, wie jede andere Klasse auch, entsprechend importiert werden.
- Eine Erweiterungsmethode weist mindestens einen Parameter auf, der den Typ (die Klasse, die Struktur) bestimmt, die sie erweitert. Möchten Sie beispielsweise eine Erweiterungsmethode für den String-Datentyp erstellen, muss der erste Parameter, den die Erweiterungsmethode entgegennimmt, vom Typ String sein. Aus diesem Grund kann ein Optional-Parameter oder ein ParamArray-Parameter nicht der erste Parameter in der Parameterliste sein, da diese zur Laufzeit variiert, dem Compiler aber bereits zur Entwurfszeit bekannt sein muss.

Der eigentliche Zweck von Erweiterungsmethoden

Sie werden Erweiterungsmethoden in Ihren eigenen Klassen und Assemblies (Klassenbibliotheken) so gut wie nie benötigen, denn Sie haben den Quellcode, und können ihn – natürlich immer versionskompatibel(!) – so erweitern, wie Sie es wünschen.

Der eigentliche Zweck von Erweiterungsmethoden ist es, eine Infrastruktur für LINQ geschaffen haben zu können, mit denen sich die Verwendung allgemeingültiger, generischer und statischer Funktionen typgerecht »anfühlt«. Für das tiefere Verständnis brauchen Sie allerdings gute Kenntnisse von Schnittstellen; den Abschnitt über Lambda-Ausdrücke sollten Sie ebenfalls bereits durchgearbeitet haben.

Die Entwickler des .NET Frameworks 3.5 standen vor der Aufgabe, in vorhandene Typen quasi eine zusätzliche Funktionalität einzubauen, ohne aber den vorhandenen Code dieser Typen – insbesondere waren dabei alle generischen Auflistungsklassen betroffen – in irgendeiner Form auch nur um ein einzelnes Byte zu verändern. Das hätte nämlich zwangsläufig zu sogenannten *Breaking Changes* (Änderungen mit unberechenbaren Auswirkungen auf vorhandene Entwicklungen) geführt. So war es prinzipiell nur möglich, zusätzliche Funktionalität in statischem, generischem Code unterzubringen, der sich allerdings sehr »uncool« angefühlt hätte.

Erweiterungsmethoden sind im Grunde genommen nur eine Mogelpackung, denn sie bleiben das, was sie sind: Simple statische Methoden, die sich, wie gesagt, lediglich wie Member-Methoden der entsprechenden Typen anfühlen. Aber, jetzt kommt der geniale Trick: Da Erweiterungsmethoden alle Typen als ersten Parameter aufweisen können, können Ihnen auch Schnittstellen als Typ übergeben werden. Das führt aber zwangsläufig dazu, dass alle Typen, die diese Schnittstelle implementieren, in Folge auch mit den entsprechenden scheinbar zusätzlichen Methoden ausgestattet sind. In Kombination mit Generics und Lambda-Ausdrücken ist das natürlich eine geniale Sache, da die eigentliche Funktionalität durch – je nach Ausbaustufe der Überladungsversionen der statischen Erweiterungsmethoden – einen oder mehrere Lambda-Ausdrücke gesteuert wird und der eigentlich zu verarbeitende Typ eben durch die Verwendung von Generics erst beim späteren Typisieren (Typ einsetzen) ans Tageslicht kommt. Mit diesem Kunstgriff schafft man im

Handumdrehen eine komplette, typsichere Infrastruktur beispielsweise für das Gruppieren, Sortieren, Filtern, Ermitteln oder sich gegenseitige Ausschließen von Elementen aller Auflistungen, die eine bestimmte, um Erweiterungsmethoden ergänzte Schnittstelle implementieren – beispielsweise bei `IEnumerable(T)` um die statischen Methoden der Klasse `Enumerable`. Oder kurz zusammengefasst: Alle Auflistungen, die `IEnumerable(T)` einbinden, werden scheinbar um Member-Methoden ergänzt, die in `Enumerable` zu finden sind, und das sind genau die, die für LINQ benötigt werden, um Selektionen, Sortierungen, Gruppierungen und weitere Funktionalitäten von Daten durchzuführen. Es ist aber nur scheinbar so – die ganze Funktionalität von LINQ wird letzten Endes in eine rein prozedural aufgebaute Klasse mit einer ganzen Menge statischer Funktionen delegiert.

Ein Beispiel für Erweiterungsmethoden finden Sie in der LINQ-Einführung in Kapitel 31.

Inhalte von System- oder externen Typen in Vorschaufenstern fürs Debuggen konfigurieren

Beim Debuggen von Anwendungen war die vollständige Konfiguration der Wertentsprechungen von Systemtypen oder von Typen, zu denen Sie den Quellcode nicht besitzen, in Variablen-Vorschaufenstern (anders als beispielsweise in C#) nicht implementiert. In Visual Basic 2008 wurde dieses Manko ausgemerzt und die Unterstützung für `DebuggerDisplayAttribute` komplettiert.

Worum geht's genau? Ganz vereinfacht ausgedrückt um die Darstellung des eigentlichen Wertes eines Typs beim Debuggen in einem Vorschaufenster, etwa wie in der folgenden Abbildung zu sehen:

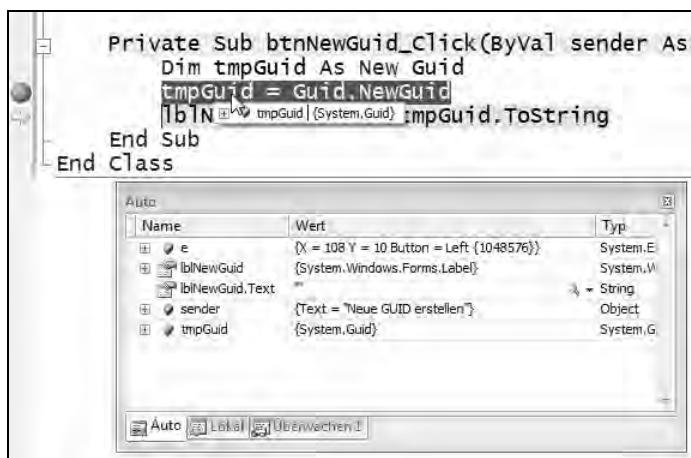


Abbildung 11.7 Bei bestimmten Typen ist keine Entsprechung bei der Ausgabe seines Inhalts als Text definiert – stattdessen wird, wie hier bei `Guid`, der nicht sehr informative Typename ausgegeben

- Um dem Abhilfe zu leisten, definieren Sie eine neue Assembly in Form einer Klassenbibliothek. Wählen Sie dazu aus dem Kontextmenü des Projektmappen-Explorers über die Projektmappe (nicht das Projekt!) den Menüpunkt *Hinzufügen/Neues Projekt*.
- Wählen Sie unter Projekttypen *Visual Basic/Windows* und unter Vorlagen *Klassenbibliothek* aus, bestimmen Sie Projektnamen (beispielsweise *CustomDebugDisplays*) und Speicherort, und klicken Sie auf *OK*. Falls nur das Projekt, nicht aber die Projektmappe im Projektmappen-Explorer zu sehen ist,

wählen Sie aus dem Menü *Datei* den Befehl *Neu/Projekt* aus, um an den entsprechenden Dialog zu gelangen.

- Klicken Sie im Projektmappen-Explorer das neu hinzugefügte Projekt an, und wählen Sie das Projektmappen-Explorer-Symbol mit der Tooltipp-Beschreibung *Alle Dateien anzeigen* aus. Öffnen Sie anschließend den Zweig *My Projekt*.
- Doppelklicken Sie auf *AssemblyInfo.vb*, um die Assembly-Infos im Editor anzeigen zu lassen.
- Fügen Sie über das *Assembly*-Attribut die entsprechenden *DebuggerDisplay*-Attribute hinzu, etwa:

```
<Assembly: DebuggerDisplay("ToString", Target:=GetType(Guid))>
```

- Mit dieser Anweisung bestimmen Sie die *ToString*-Methode (erstes Argument) zur Anzeige des Guid-Typs (zweites Argument).
- Erstellen Sie die Projektmappe neu und kopieren Sie die neue Assembly in das Verzeichnis *Visual Studio 2008/Visualizers*, das Sie im *Eigene Dokumente*-Verzeichnis Ihres Windows-Laufwerkes finden. Sie finden die Assembly im Unterverzeichnis *.\bin\debug* des Projektverzeichnisses.

Beim erneuten Debuggen finden Sie die korrekte Darstellung des Guid-Typs in allen Vorschaufenstern:

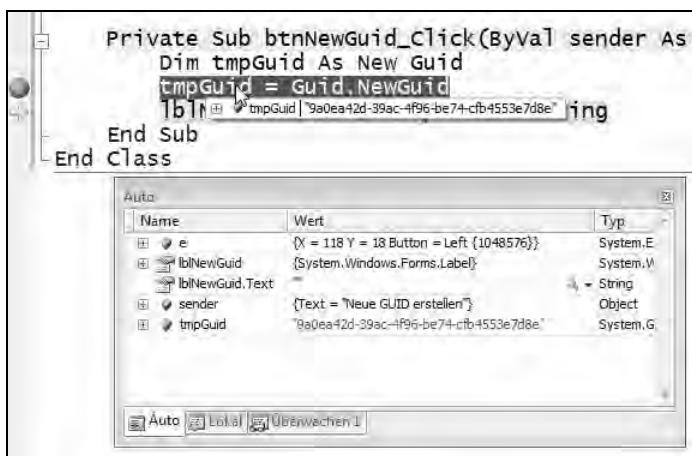


Abbildung 11.8 Nachdem die neue Assembly in *Visual Studio 2008/Visualizers* unterhalb des *Eigene Dokumente*-Ordners kopiert wurde, wird der Inhalt einer Guid-Instanz in den Vorschaufenstern korrekt dargestellt

Zugriff auf den .NET Framework-Quellcode beim Debuggen

Dieser Abschnitt beschäftigt sich mit einem Thema, dessen Ankündigung in der Presse schon für eine kleine Sensation sorgte. Die simple und dazu umgekehrt proportional Aufmerksamkeit erregende Meldung lautete: »Microsoft legt .NET-Quelltexte offen²«. Das gelesen, stellten sich viele enthusiastische C#-Entwickler dann

² So der Heise-Ticker unter www.heise.de am 4.10.2007 um 11:15. Unter <http://www.heise.de/newsticker/meldung/96909> können Sie diese Meldung direkt abrufen (Stand: 18.1.2008).

vor, »Oh wie cool – dann lade ich die Quelltexte herunter und bastele mir mein eigenes Framework³«. Doch ganz so einfach hat es uns Microsoft nicht gemacht, an die begehrten Quellen des Frameworks zu kommen.

Zunächst einmal benötigen Sie mindestens Visual Studio 2008 in der Standardversion. Ältere Visual Studio-Versionen und auch sämtliche Express-Versionen bleiben außen vor. Und dann können Sie die Quelltexte der jeweiligen Komponenten, die Sie interessieren, nur dann bekommen, wenn Sie innerhalb von Visual Studio in diese hinein debuggen – und zu nichts anderem sind die Quellcodes bzw. die Freigabe derselben auch gedacht: Nämlich um Sie beim Debuggen innerhalb Ihrer eigenen Projekte zu unterstützen.

Und für uns Visual Basic-Entwickler gibt es eventuell noch eine kleine Einschränkung, und ich möchte mich hier bewusst nicht weiter auf das Vorwort dieses Buchs beziehen: Der verwaltete Teil des .NET Frameworks ist natürlich zu ganz erheblichen Teilen in C# geschrieben worden. Zu welchen genau entzieht sich meiner Kenntnis, aber müsste ich schätzen, dann würde ich sagen: 95% sind in C#, der Rest (beispielsweise die *Microsoft.VisualBasic.dll*-Assembly) in Visual Basic. Das wiederum bedeutet: Sie sehen den Quelltext natürlich auch nur in der Sprache, in der er entwickelt wurde, und das ist eben in den meisten Fällen C#.

Doch auch bevor das geschehen kann, müssen Sie – Stand 18.01.2008 – noch ein paar Vorbereitungen treffen, um in den Genuss des Framework-Debuggings zu gelangen. Wie es genau geht, zeigt die folgende Schritt-für-Schritt-Anleitung.

- Zunächst benötigen Sie ein Projekt, das Sie debuggen möchten. In guter alter Fernsehköchemanier haben wir da mal was vorbereitet – und was könnte es anderes sein, als die 132 gazillionste Version von »Hello World«!

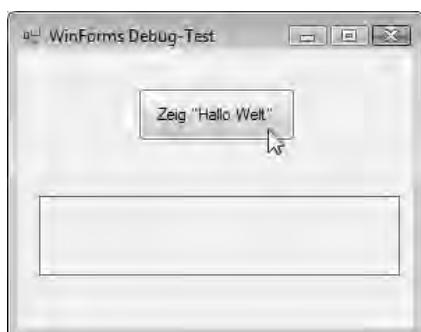


Abbildung 11.9 Diese spektakuläre Windows-Anwendung wird das Debuggen von .NET Framework-Code demonstrieren: Entdecken Sie, was wirklich passiert, wenn Sie die Schaltfläche drücken!

- Und bevor wir uns daran machen können, herauszufinden, was wirklich passiert, wenn wir einem Label-Steuerelement eine neue Zeichenkette zuweisen ...

```
'Mehr iss nich!
Public Class Form1

    Private Sub btnShowHelloWorld_Click(ByVal sender As System.Object, _
                                         ByVal e As System.EventArgs) _
                                         Handles btnShowHelloWorld.Click
```

³ Kleine Anekdote am Rande: Hier bei ActiveDevelop sorgte die Meldung auch für lustiges Tohuwabohu, nachdem die *schon* ein wenig sarkastische Diskussion über den Nutzen des Frameworks »entglitt« und unser Chefentwickler sich der Realisierung seiner Weltübernahmepläne mithilfe einer eigenen Framework-Version einen Schritt näher wöhnte... ;-)

```
'Text der Schaltfläche setzen
lblHelloWorld.Text = "Hallo Welt!"

End Sub
End Class
```

- ... müssen wir noch einige Einstellungen an der Benutzeroberfläche von Visual Studio vornehmen. Dazu wählen Sie in der IDE von Visual Studio aus dem Menü *Extras* den Menüpunkt *Optionen* aus.
- Im Dialog, der anschließend erscheint, wählen Sie in der linken Liste den Eintrag *Debugging*.

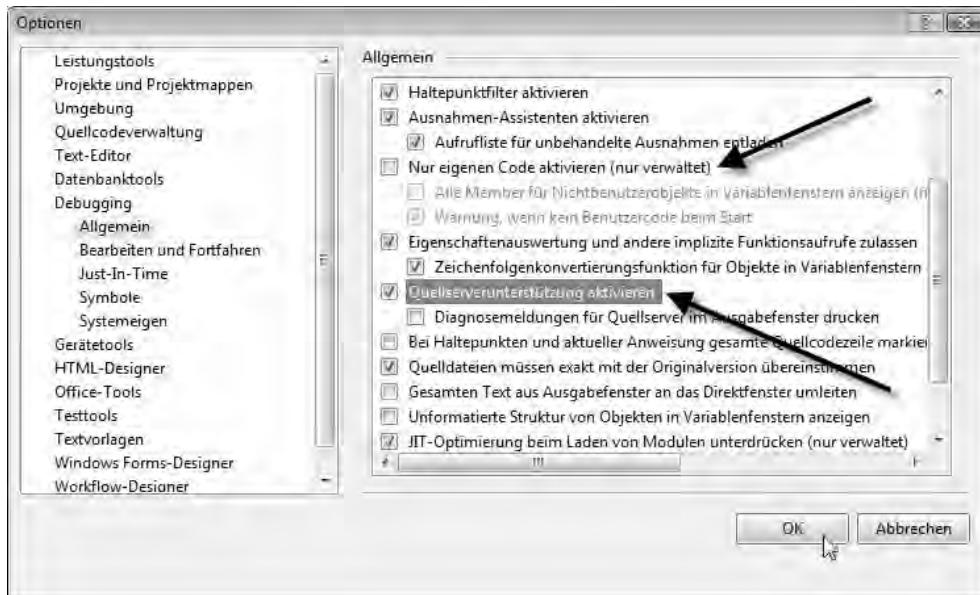


Abbildung 11.10 Deselektieren Sie die Option *Nur eigenen Code aktivieren (nur verwaltet)* und aktivieren Sie die Option *Quellsupport aktivieren*

- Auf der rechten Seite des Dialogs werden nun die allgemeinen Debugging-Einstellungen angezeigt. Hier deseletieren Sie die Option *Nur eigenen Code aktivieren (nur verwaltet)* und aktivieren die Option *Quellsupport aktivieren*.
- Wechseln Sie in der linken Spalte desselben Dialogs auf den *Debugging*-Untereintrag *Symbole*. Dort klicken Sie auf die *Neu*-Schaltfläche (das Symbol, auf das die Maus in Abbildung 11.11 zeigt), und geben Sie den Microsoft-Symboldateienserver an: <http://referencesource.microsoft.com/symbols>.⁴
- Unter dem Punkt *Symbole vom Symbolserver in diesem Verzeichnis zwischenspeichern* geben Sie anschließend einen Speicherort auf Ihrer Festplatte an, an dem die Quellcodedateien zwischengespeichert werden. Das Zwischenspeichern hat den Vorteil, dass die Dateien nur beim ersten Mal vom Microsoft-

⁴ Stand: 18.01.2008 – dieser Eintrag könnte sich, wenn auch mit nicht großer Wahrscheinlichkeit, ändern – das sollte dann aber ausreichend früh von Microsoft kommuniziert werden.

Server aus dem Internet geladen werden müssen – schon beim zweiten Quellcode-Debuggen fällt dieser Schritt weg, und die Quellcode-Dateien werden viel schneller aus diesem Cache-Speicher entnommen.

- Wenn Sie den Dialog nun mit OK bestätigen, sehen Sie einen EULA, den Sie ebenfalls nach natürlich aufmerksamem Studium mit OK bestätigen müssen – und dann dauert es ein kleines Weilchen, bis das Nächste passiert, da Visual Studio sofort anfängt, benötigte Symboldateien für das Beispielprojekt herunterzuladen. In der Statuszeile von Visual Studio können Sie beobachten, was passiert.

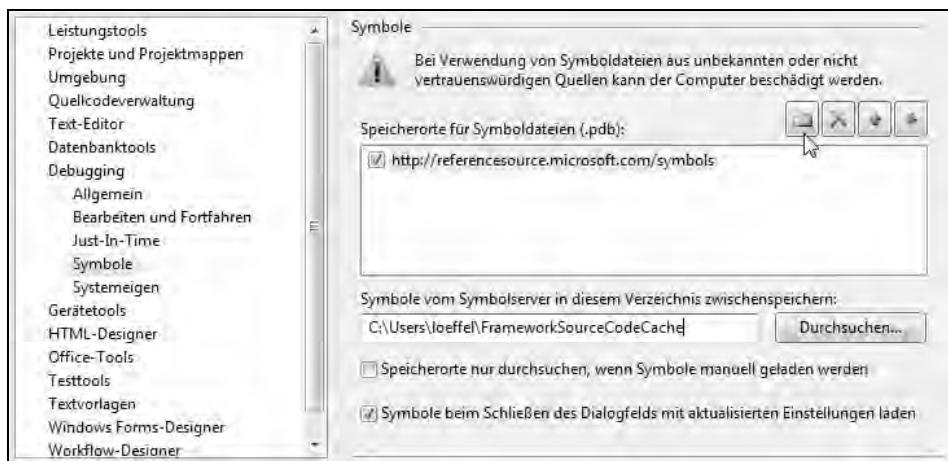


Abbildung 11.11 Im Symbole-Dialog geben Sie die Quelle der Symboldateien an, die auch den eigentlichen Quellcode des .NET Frameworks enthalten sowie ein Verzeichnis, in dem die Quellcode-Dateien zwischengespeichert werden können

- Den anschließenden Sicherheitshinweis bestätigen Sie ebenfalls mit Ja (natürlich auch nicht, bevor Sie ihn aufmerksam gelesen haben!).
- Und damit haben Sie quasi die Voraussetzungen für das Debuggen gelegt. Um das Debuggen nun zu starten, setzen Sie mit F9 einen Haltepunkt – am besten an der Stelle, an der im Programmcode die Textzuweisung von »Hallo Welt!« an das Label erfolgt:

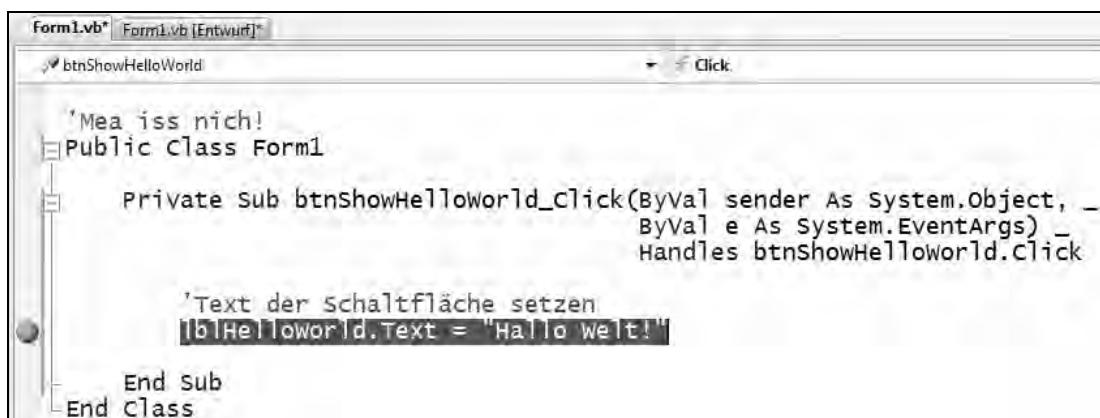


Abbildung 11.12 Setzen Sie in der Zeile den Haltepunkt, die die Ausgangsbasis für den Sprung ins Framework sein soll

- Starten Sie anschließend Debuggen mit **F5**.
- Wenn das Programm gestartet ist, klicken Sie auf die Schaltfläche *Zeig "Hallo Welt"*.
- Das Programm trifft auf die entsprechende Haltepunktzeile, und die Programmausführung wird an dieser Stelle unterbrochen.
- Wählen Sie jetzt aus dem Menü *Debuggen* den Menüpunkt *Fenster* und weiter *Aufrufliste*. Alternativ drücken Sie **Strg Alt C**.
- Sollten die Einträge für die Assembly *System.Windows.Forms*, anders als in der folgenden Abbildung zu sehen, ausgegraut sein, öffnen Sie das Kontext-Menü mit der rechten Maustaste und wählen den Menüpunkt *Symbole laden*.

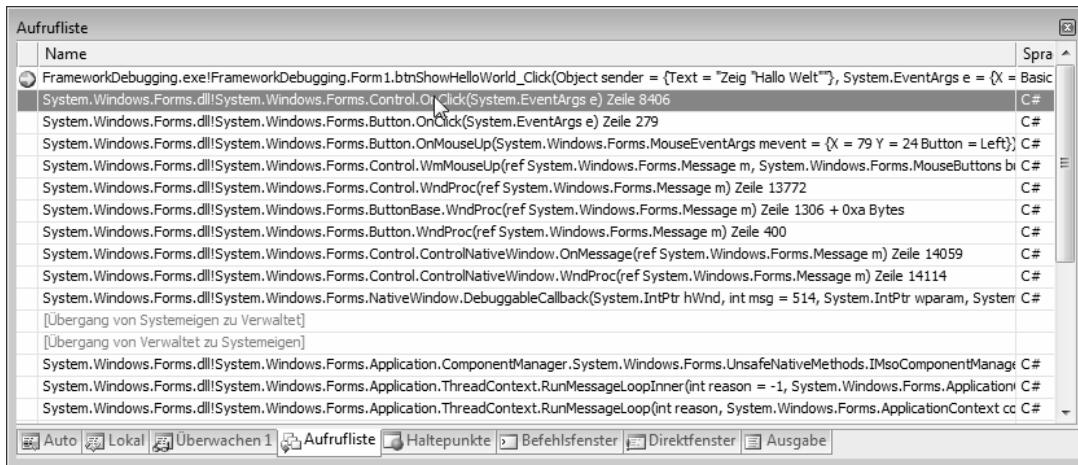


Abbildung 11.13 Im Unterschied zu »sonst«, sollten die Framework-Assembly-Methoden und Eigenschaften in der Aufrufliste nicht ausgegraut sein, als Zeichen dafür, dass Sie in sie hinein-debuggen können. Falls doch: Symbole über das Kontextmenü laden!

- Wenn Sie anschließend mit **F11** in die Routine hineinstappen, landen Sie tatsächlich im Framework-Quell-Code, wie Abbildung 11.14 zeigt.



The screenshot shows the Microsoft Visual Studio IDE with the 'Label.cs' file open. The tab bar at the top has 'Label.cs' selected, followed by 'Disassembly', 'Form1.vb', and 'Form1.vb [Entwurf]'. The main window displays the C# source code for the 'Label' class. The code includes several XML comments (///) and developer documentation (/// <devdoc>). One notable comment describes the 'Text' property: 'Gets or sets the text in the Label. Since we can have multiline support this property just overrides the base to pluck in the Multiline editor.' The code also shows the implementation of the 'Text' property, which gets the value from the base class and sets it back if modified. It includes event handlers for the 'TextAlignChanged' event, which add and remove event handlers for the 'EVENT_TEXTALIGNCHANGED' event.

```
    /// Gets or sets the text in the Label. Since we can have multiline support
    /// this property just overrides the base to pluck in the Multiline editor.
    /// </para>
    /// </devdoc>
    [
        Editor("System.ComponentModel.Design.MultilineStringEditor, " + AssemblyRef.SystemDesign),
        SettingsBindable(true)
    ]
    public override string Text {
        get {
            return base.Text;
        }
        set {
            base.Text = value;
        }
    }

    /// <include file='doc\Label.uex' path='docs/doc[@for="Label.TextAlignChanged"]'/>
    /// <devdoc>
    /// <para>[To be supplied.]</para>
    /// </devdoc>
    [SRCategory(SR.CatPropertyChanged), SRDescription(SR.LabelOnTextAlignChangedDescr)]
    public event EventHandler TextAlignChanged {
        add {
            Events.AddHandler(EVENT_TEXTALIGNCHANGED, value);
        }
        remove {
            Events.RemoveHandler(EVENT_TEXTALIGNCHANGED, value);
        }
    }

    /// <devdoc>
    /// Determines whether to use compatible text rendering engine (GDI) or not (GDI+).
    /// </devdoc>
```

Abbildung 11.14 Voila! – Der Source-Code des Frameworks, natürlich nur ein ganz, ganz kleiner Ausschnitt (hier: *Label.cs* – der Sourcecode des Label-Controls)

Teil C

Objektorientiertes Programmieren

In diesem Teil:

Einführung in die objektorientierte Programmierung	387
Klassentreffen	393
Klassencode entwickeln	407
Vererben von Klassen und Polymorphie	449
Entwickeln von Wertetypen	519
Typenumwandlung (Type Casting) und Boxing von Wertetypen	535
Beerdigen von Objekten – Dispose, Finalize und der Garbage Collector	549
Eigene Operatoren für benutzerdefinierte Typen	571
Ereignisse und Ereignishandler	589

Kapitel 12

Einführung in die objektorientierte Programmierung

In diesem Kapitel:

Was spricht für Klassen und Objekte?

389

Ohne Klassen und Objekte läuft in .NET gar nichts mehr – und das gilt für Visual Basic .NET nicht weniger als für alle die anderen .NET-Programmiersprachen. Doch gerade für Visual Basic-Programmierer ist das objektorientierte Programmieren ein Gebiet, was sich bis zuletzt (heißt: Visual Basic 6.0) vermeiden ließ. Zwar gab es Klassen und Objekte auch dort, aber die Techniken der »klassischen« objektorientierten Programmierung waren schon sehr eingeschränkt und viele VB 6.0-Programmierer haben die vorhandenen Möglichkeiten einfach nicht genutzt. So gibt es Umsteiger, die schon einige Erfahrungen mit Visual Basic .NET gesammelt haben, die aber immer noch der Meinung sind, dass sie die objektorientierte Programmierung nicht benötigen und sich deswegen nicht mit ihr beschäftigen.

Natürlich können diese Entwickler selbst nach 7 Jahren vorhandener Möglichkeit, auch in Visual Basic .NET objektorientiert zu programmieren, immer noch in ihrem alten prozeduralem Stil weiterprogrammieren. Doch ganz ehrlich: Das ist wie AMG-Mercedes¹ fahren mit angezogener Handbremse. Visual Basic .NET wurde gerade durch seine damals neuen OOP-Fähigkeiten endlich erwachsen, und gerade wenn Sie auf moderne Technologien wie LINQ für das Entwickeln zukunftssicherer Anwendungen vertrauen wollen, kommen Sie an der objektorientierten Programmierung nicht mehr vorbei.

Das gilt umso mehr, als dass LINQ nicht nur beim Abfragen von Datenbanken eine immer zunehmendere Rolle spielen wird, sondern auch in der Parallelisierung von Prozessen. Denken Sie daran, dass Prozessoren sich seit ca. 2002 nicht mehr oder nur gering über Taktfrequenzen leistungsmäßig skalieren lassen. Bis dahin galt: Benötige ich ein schnelleres System, bediene ich mich einfach eines Prozessors mit höherer Taktfrequenz,² denn überwiegend darauf waren die jeweils leistungsfähigeren Versionen der nächsten Generationen beschränkt. Das funktioniert nicht mehr ganz so, denn Taktfrequenzen von Prozessoren lassen sich aufgrund des Verlustleistungsproblems nur noch in Maßen erhöhen. Also skalierte man nicht mehr auf der vertikalen Geschwindigkeitsachse, sondern baute die Prozessoren in die Breite, was bedeutet: Ein Prozessor bekommt mehrere sogenannte Cores, und ein moderner Prozessor besteht heutzutage mindestens mal aus zwei Cores, bei Desktopsystemen sind 4 inzwischen üblich. Doch was nutzt das? Wenn Software so konzeptioniert ist, dass sie nicht mehrere Dinge zur gleichen Zeit ausführt – dafür müssen Sie als Entwickler nämlich jedenfalls zur Zeit noch explizit sorgen – läuft Ihre Anwendung selbst bei Volllast aus Anwendungssicht auf einem Dual Core nur mit 50%, auf einem Quad Core nur mit 25% der möglichen Leistung. Und daran ändert keine Einstellung oder auch kein Betriebssystem dieser Welt etwas. LINQ könnte nun nicht nur beim Abfragen von Daten, sondern überhaupt beim iterierten Berechnen von Daten der Schlüssel zu einer halbwegs automatischen Parallelisierung von Algorithmen sein. Der Plan ist: LINQ untersucht die Abfrage/Iteration und erstellt einen Ausführungsplan, der dann automatisch auf mehrere, zur Verfügung gestellte Prozessorcores verteilt wird. Diese Technik gibt es zurzeit, zu der diese Zeilen entstehen, lediglich experimentell, als so genannte P-LINQ-(Parallel LINQ) Technologie. Sie benötigen aber in jedem Fall umfassende Kenntnisse von Klassen, Instanzen, Auflistungen, Generics und Delegaten, um diese Technik jetzt und in Zukunft einzusetzen. Die reine prozedurale Programmierung ist hier eine komplette Sack-

¹ Porsche stand hier die letzten Jahre – wir wollen aber natürlich keinen der großen, deutschen Autobauer benachteiligen. BMW baut sicherlich auch tolle Autos, Audi, VW und andere selbstverständlich ebenfalls. In diesem Zusammenhang: Autor und Fachlektor streiten sich seit Jahren darüber, welches Auto schöner ist: Das 2008er Audi A4 Cabrio (Fachlektor) oder das 2008er Mercedes CLK Cabrio (Autor). Wir bitten herzlichst um Ihre Meinungen an klaus@loeffelmann.de oder droege@beconstructed.de.

² Natürlich immer gemessen an der Architektur des vorherigen Prozessormodells, um nicht Äpfel mit Birnen zu vergleichen. Schon in den 80er Jahren war ein 1 MHZ getakteter 6502 Prozessor von MOS-Technologies in etwa so schnell, wie ein Z80 Prozessor mit 4 MHz von Zilog. Ein einziger Core eines heutzutage verbauten Intel Dual Core Prozessor T9300 mit 2,5 GHZ ist sicherlich deutlich schneller als ein 3 GHZ schneller Pentium 4 aus dem Jahre 2003.

gasse, sodass fundiertes Wissen über die OOP der Schlüssel zum Entwicklerglück wird. Zumal natürlich die vielen immer wieder erwähnten Vorteile des objektorientierten Programmierens auch für Visual Basic .NET gelten. Machen Sie von den neuen Möglichkeiten Gebrauch, dann wird Ihr Code lesbarer, einfacher zu verwalten und zu erweitern und er ist besser in späteren Projekten wieder zu verwenden.

Und vielleicht ein Tipp an dieser Stelle: Lesen oder wenigstens »browsen« Sie doch auch dann durch die folgenden Kapitel, wenn Sie meinen, Klassenprogrammierung grundsätzlich zu beherrschen und in Ihren eigenen Projekten schon längst verwenden: Riskieren Sie es doch dennoch, sich die folgenden Kapitel zu Gemüte zu führen, um Altes aufzufrischen und auch, um den neuen Technologien von Visual Basic 2008 im richtigen Kontext immer mal wieder hier und da zu begegnen. Denken Sie daran: Zum Thema Klassen und OOP gehört weit mehr, als nur das Wissen um ihre bloße Instanziierung und das Erstellen von Eigenschaften und Methoden, denn: Wie sieht es mit Klassenvererbung, abstrakten Klassen und Schnittstellen aus? Kennen Sie die Unterschiede zwischen Verweis- und Wertetypen? Können Sie erklären, wieso ein Objekt ein Referenztyp ist, alle primitiven Typen zwar rein .NET-infrastrukturgemäß von Object abgeleitet sind, diese aber dennoch zu Wertetypen werden? Wie steht's mit dem »Boxing« von Wertetypen – wissen Sie genau, was dabei wirklich passiert, und welche Tücken Ihnen hier und da begegnen? Kennen Sie auch alle Feinheiten der Polymorphie, und wissen Sie, wann Sie die Referenzierungen Me, MyClass und MyBase einsetzen müssen? Wie schaut es aus mit Generics, den an C++-Templates angelehnten Objekttypen von .NET, mit denen Sie – richtige Anwendung vorausgesetzt – viele Dinge des täglichen Entwicklerlebens enorm vereinfachen und extrem schnell wieder verwendbare Klassen schaffen können?

Sie sehen: Es gibt eine ganze Menge, was Sie über objektorientierte Programmierung und den Einsatz von Klassen wissen können (und sollten). Je trittfester Sie beim Einsatz von Klassen werden, desto robuster, pflegeleichter und weniger fehleranfällig werden später Ihre Programme sein.

TIPP

Die Kapitel und einzelnen Abschnitte dieses Teils, so umfangreich sie auch sind, bauen schrittweise aufeinander auf. Deswegen sei Ihnen empfohlen, dass Sie sich ein wenig Zeit nehmen und die folgenden Abschnitte am besten hintereinander durcharbeiten. Das Verstehen des ganzen Zusammenhangs von Klassen, Vererbung, Schnittstellen und allem was sonst noch dazu gehört, wird Ihnen dann sicherlich viel leichter fallen – und Sie werden sich im Handumdrehen zum »Klassenprimus« mausern!

Was spricht für Klassen und Objekte?

Klassen sind der Hauptstützpfeiler der objektorientierten Programmierung. Spricht man von Klassen und Klassenkonzepten, dann sind Technologien wie Schnittstellen, Klassenmodelle, das Ableiten bzw. Vererben von Klassen sowie Polymorphie mit dem gezielten Überschreiben von Klassenmethoden und -eigenschaften nicht weit. Doch darum soll es vordergründig in diesem Kapitel gar nicht gehen.

Klassen sind in erster Linie mal Schablonen, die ihre Daten reglementieren und von der Außenwelt schützen – und auch nur unter diesem Gesichtspunkt sollen sie in diesem einführenden Kapitel erklärt werden.

»Eine Klasse schafft die Strukturen für das Speichern von Daten und beinhaltet gleichzeitig Programmcode, der diese Daten reglementiert.« Diese Erklärung finden Sie nicht nur oftmals als die Erklärung für das Konzept von Klassen, sie ist auch kurz, knackig, absolut zutreffend und so abstrakt, dass jemand, der sich zum ersten Mal mit dieser Materie beschäftigt, damit überhaupt nichts anfangen kann.

Deswegen wollen wir im Folgenden das Pferd bewusst von der anderen Seite aufzäumen und ein Beispiel bemühen, das zeigt, wie ein Programm Daten speichert und organisiert, das *ohne* Klassen zureckkommen muss.

Miniadresso – die prozedurale Variante

Also lassen Sie uns mal einen Blick auf die »So-besser-nicht«-Variante werfen – hochtrabend *Miniadresso-Prozedu* genannt.

HINWEIS Bei den Beispielprogrammen der folgenden Abschnitte handelt es sich nicht um Windows-, sondern um so genannte Konsolenanwendungen – um Anwendungen also, die sich ausschließlich unter der Windows-Eingabeaufforderung verwenden lassen, und wie Sie sie schon in Kapitel 2 dieses Buchs kennengelernt haben.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 12\\MiniAdressoProzedu

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie das Programm starten, sehen Sie ein Konsolenfenster, das die Programmausgabe beinhaltet, wie Sie sie in etwa auch in Abbildung 12.1 betrachten können.

TIPP Bei den vergleichsweise großen Auflösungen der 22 und 24 Zoll-Monitore, die heutzutage schon für kleines Geld zu haben sind, stellen Sie den Font der Konsolenfenster am besten auf die größte Schriftart (10 × 18), die Ihnen das Systemmenü über den Menüpunkt *Standardwerte* für alle zukünftigen Instanzen des Eingabeaufforderungsfensters, mit *Eigenschaften* für die aktuelle Instanz des Eingabeaufforderungsfensters anbietet.

```

file:///C:/Data/FolderShare/SharedCurrent/Books/VB 2008/German/Samples/C - OOP/Kapitel 12/MiniAdressoProzedu/bin/Debug/MiniAdr...
003: Langenbach, Christian, 41729, Lippstadt
004: Löffelmann, Michaela, 69493, Dortmund
005: Meier, Melanie, 47824, Bad Waldliesborn
006: Albrecht, Michaela, 91508, Bad Waldliesborn
007: Jungemann, Hans, 61733, Hildesheim
008: Tiemann, Klaus, 99072, Wuppertal
009: Weichelt, Christian, 95897, Wiesbaden
010: Neumann, Uta, 98014, Lippstadt

Adressen werden sortiert ... fertig!

000: Ademmer, Margarete, 93068, Dortmund
001: Ademmer, Axel, 83968, Berlin
002: Ademmer, Momo, 71591, Stirpe
003: Ademmer, Theo, 71591, Dortmund
004: Ademmer, Uta, 71591, Braunschweig
005: Albrecht, Michaela, 71591, Stirpe
006: Albrecht, Rainer, 83968, Soest
007: Albrecht, Franz, 71591, Soest
008: Albrecht, Gabriele, 71591, Bielefeld
009: Albrecht, Anne, 71591, Stirpe
010: Braun, Theo, 71591, Soest

Taste zum Beenden drücken...

```

Abbildung 12.1 Die erste Version des Programms macht scheinbar Dienst nach Vorschrift – doch durch das prozedurale Konzept hat sich leider ein schwerer Fehler eingeschlichen, der nicht gleich ersichtlich ist

OK – hier haben wir also die erste Version unseres kleinen Beispielprogramms, das nichts weiter macht, als sich ein paar Kontaktadressen auszudenken, deren Einzeldaten in dafür vorgesehene Arrays abzulegen, die Kontakte nach Nachnamen zu sortieren und sie anschließend auszugeben. Schauen wir uns an, wie das Programm generell mit seinen Daten umgeht:

```
Module Hauptmodul

    Dim Nachname(0 To 100) As String
    Dim Vorname(0 To 100) As String
    Dim PLZ(0 To 100) As String
    Dim Ort(0 To 100) As String

    Sub Main()
```

Es benötigt für jede Kontaktadresse jeweils vier Einzelemente, die es in insgesamt vier Arrays ablegt, und genau das macht es so schwierig für den Entwickler, die Daten einer einzigen Kontaktadresse auch wirklich zusammen zu halten. Ohne eine *zusammenhängende* Entität zu schaffen, die die Daten eines Kontakts auch programmtechnisch mehr oder weniger automatisch bzw. durch ihr Konzept bedingt zusammenhält, sind Fehler buchstäblich vorprogrammiert – und genau ein solcher Fehler hat sich dann auch in dieser Programmversion eingeschlichen (OK, natürlich habe ich ihn zu Demozwecken einschleichen lassen, aber dieses prozedurale Programmbeispiel ist typisch für eine interne Datenverwaltung; genau so typisch eben wie die Fehler, die dabei auftreten können).

```
Sub AdressenSortieren()

    'Beispiel für lokalen Typrückschluss
    Dim anzahlElemente = 101
    Dim delt = 1

    Dim aeussererZaehler As Integer
    Dim innererZaehler As Integer
```

Bei einem Blick auf die Sortierroutine fällt zunächst etwas Besonderes auf, was es neu in Visual Basic 2008 gibt und unter dem Namen *lokaler Typrückschluss* bezeichnet wird. Dabei werden die Typen von Variablen nur durch reine Zuweisung von Konstanten festgelegt. Kapitel 11 erklärt, wie der lokale Typrückschluss genau funktioniert.

Doch zurück zum eigentlichen Ausgangsproblem: Die Sortierroutine des Programms ist in diesem Beispiel der Übeltäter: Gerade sie müsste durch sorgfältige Ausführung bei der Programmierung dafür sorgen, dass ein Adressdatensatz logisch zusammengehörig bleibt, damit die Integrität des gesamten Datenaufkommens nicht verletzt wird. Schauen wir uns diese Sortierroutine jedoch genauer an ...

```
Dim tempVorname, tempNachname, tempPLZ, tempOrt As String

'Größten Wert der Distanzfolge ermitteln
Do
    delta = 3 * delta + 1
Loop Until delta > anzahlElemente
'
Do
```

```
'Späteres Abbruchkriterium - entsprechend kleiner werden lassen
delta \= 3

'Shellsort's Kernalgorithmus
For aeussererZaehler = delta To anzahlElemente - 1
    tempVorname = Vorname(aeussererZaehler)
    tempNachname = Nachname(aeussererZaehler)
    tempPLZ = PLZ(aeussererZaehler)
    tempOrt = Ort(aeussererZaehler)

    innererZaehler = aeussererZaehler
    Do
        If tempNachname >= Nachname(innererZaehler - delta) Then Exit Do
        Vorname(innererZaehler) = Vorname(innererZaehler - delta)
        Nachname(innererZaehler) = Nachname(innererZaehler - delta)
        PLZ(innererZaehler) = PLZ(innererZaehler - delta)

        innererZaehler = innererZaehler - delta
        If (innererZaehler <= delta) Then Exit Do
    Loop
    Vorname(innererZaehler) = tempVorname
    Nachname(innererZaehler) = tempNachname
    Ort(innererZaehler) = tempOrt
    Next
Loop Until delta = 0
End Sub
```

... so stellen wir bei näherem Hinsehen fest, dass sich, was die Datenintegrität anbelangt, ein dicker Fehler in den Algorithmus eingeschlichen hat: Beim Dreieckstausch eines Kontaktelements nämlich, werden nicht wirklich alle Felder getauscht. Beim »Hintauschen« bleibt der Ort, beim »Zurücktauschen« die Postleitzahl auf der Strecke; für ein kommerzielles Programm wäre das natürlich eine Katastrophe, zumal selbst bei einer Liste von nur 100 Kontaktadressen dieser Fehler nicht sofort ins Auge sticht.

Kapitel 13

Klassentreffen

In diesem Kapitel:

Was ist eine Klasse?	394
Klassen mit New instanziieren	394
New oder nicht New – wieso es sich bei Objekten um Verweistypen handelt	396
Nothing	399
Klassen anwenden	400
Wertetypen	401
Konstanten vs. Felder (Klassen-Membervariablen)	405

Was ist eine Klasse?

Versetzen Sie sich in Ihre Kindheit zurück. Und zwar so weit, dass Sie sich vorstellen können, im Sandkasten zu sitzen und mit Förmchen und nassem Sand zu spielen. Sie werden es nicht glauben, aber genau zu diesem Zeitpunkt haben Sie bereits das Klassenkonzept angewendet. Ein Förmchen ist im Grunde genommen nämlich nichts anderes als eine Klasse. Das Förmchen gibt vor, wie Objekte ausschauen sollen, die aus ihm entstehen werden, aber selbst ist es noch kein Objekt, sondern nur eine Vorlage. Wenn Sie aus einer Klasse (einem Förmchen) einen Sandkuchen (ein Objekt) machen wollen, dann müssen Sie ein Objekt dieser Klasse instanzieren. Um bei der Analogie zu bleiben: Sie instanziieren einen Sandkuchen aus einem Förmchen, indem Sie nassen Sand in die Form hineingeben, das ganze Ding umdrehen und die Form abziehen.

Klassen im .NET Framework sind natürlich etwas abstrakter, aber Sie machen sich daran auch nicht die Hände schmutzig. Sie bestehen erst einmal aus einer Reihe von sogenannten Member-Variablen – auch Felder oder Feldvariablen genannt – und der Klassenrumpf-Definition, die diese Variablen einschließt und der Klasse ihren Namen gibt.

Damit existiert dann bereits die einfachste Version einer Klasse, die, wie wir gleich sehen werden, ihre Daten logisch kapselt. Immerhin erlaubt sie uns schon, das Problem unseres ersten Beispielprogramms im Ansatz zu ersticken – die zweite Version dieses Demoprogramms stellt das unter Beweis. Hier gibt es nun eine Klasse namens Kontakt, die aus vier Feldern besteht – die vier Datenfelder, die eine einzige Adresse ausmachen. In der Folge benötigen wir nun nicht mehr vier, eigentlich völlig zusammenhanglose Arrays, sondern nur noch ein einziges Array¹, das aus Elementen besteht, die jeweils eine komplette Adresse speichern.

Um das zu tun, fügen wir dem bestehenden Programm einfach eine neue Codedatei hinzu – eine Klassen-Datei – namens *Kontakt.vb*. In diese Klassen-Codedatei schreiben wir dann folgende Zeilen:

```
Public Class Kontakt
    Public Nachname As String
    Public Vorname As String
    Public PLZ As String
    Public Ort As String
End Class
```

Diese Klasse namens Kontakt stellt die einfachste Form einer Klasse dar. Sie enthält nur vier öffentlich zugängliche Felder, aber die Klasse fasst eben diese vier Felder logisch zu einem Datensatz zusammen.

Klassen mit New instanziieren

Um eine so genannte Instanz dieser Klasse zu schaffen, bedient man sich des Schlüsselwortes `New` – intern wird damit der entsprechende Speicherplatz geschaffen, den man benötigt, um die Daten der Member-Variablen der Klasse im Arbeitsspeicher² abzulegen. Eine Objektvariable, die man zuvor als Typ dieser

¹ Mehr zu Arrays und Auflistungen erfahren Sie in Kapitel 22.

² Genau genommen in einem bestimmten Teil des Arbeitsspeichers, der durch das .NET Framework verwaltet, d.h. ständig aufgeräumt wird. Sein Name: *Managed Heap*.

Klasse definiert, dient dann dazu, auf die Elemente der Klasseninstanz zuzugreifen. Die Anwendung dieser einfachen Beispielklasse würde also wie folgt aussehen:

```
Dim adr As Kontakt  
adr = New Kontakt
```

Die erste Codezeile legt eine Objektvariable vom Typ Kontakt an und die zweite Codezeile weist ihr eine neue Instanz zu. Sie können diese beiden Zeilen auch in einer Zeile zusammenfassen:

```
Dim adr As New Kontakt
```

Anschließend verwenden Sie die Instanz dieser Klasse mithilfe der definierten Objektvariablen, um auf die einzelnen Felder zuzugreifen:

```
adr.Nachname = "Dröge"  
adr.Vorname = "Ute"  
adr.PLZ = "59555"  
adr.Ort = "Lippstadt"
```

Um nochmals auf unsere Analogie mit dem Sandkuchen zurückzukommen: Die Objektinstanz entspricht hier einem Sandkuchen, der aus einem Förmchen hervorgegangen ist. Die Klasse hingegen ist das Förmchen. Die Objektvariable benennt wiederum den Sandkuchen, der aus dem Förmchen hervorgegangen ist, und natürlich können Sie, wie aus dem Förmchen, beliebig viele Sandkucheninstanzen erstellen. Beliebig viele? Nicht ganz: Natürlich nur so viele, bis der »verwaltete Sandberg« zuneige geht, aus dem Sie die Sandkuchen machen, der in der Analogie dem Managed Heap entspricht, aus dem Sie den Speicherplatz für Ihre Objektinstanzen beziehen.

Ein gern gemachter Fehler am Anfang ist es übrigens, die Klasse nicht zu instanziieren, sondern direkt verwenden zu wollen:

```
'So geht's natürlich nicht!  
Kontakt.Nachname = "Dröge"  
Kontakt.Vorname = "Ute"  
Kontakt.PLZ = "59555"  
Kontakt.Ort = "Lippstadt"
```

Warum? – Denken Sie mal darüber nach. In Analogie zu einer primitiven Variablen würden Sie praktisch statt

```
Dim i as Integer  
i = 5
```

einfach

```
Integer = 5
```

schreiben, quasi den Bezeichner für den Datentyp als Variablenamen verwenden. Das geht natürlich nicht! Später werden wir sehen, dass es tatsächlich Eigenschaften und Methoden gibt, die Sie direkt am Typenamen aufrufen können wie etwa bei `if File.Exists("C:\windows\calc.exe") then` – aber dabei handelt es sich um sogenannte statische Methoden, die wir später als Ausnahme von der Regel betrachten werden.

TIPP Wie Sie bestimmen können, dass bestimmte Aufgaben beim Instanziieren einer Klasse durch Ihren eigenen Code erledigt werden, erfahren Sie im Abschnitt »Der Konstruktor einer Klasse – bestimmen, was bei New passiert« im nächsten Kapitel.

Öffentliche Felder oder Eigenschaften beim Instanziieren initialisieren

Visual Basic erlaubt seit der 2008er Version auch eine kürzere Variante beim Vorbelegen der Felder bzw. Eigenschaften mit Werten: Sie können öffentliche Felder oder Eigenschaften direkt beim Instanziieren mithilfe des `With`-Schlüsselwortes definieren, wie im folgenden Beispiel zu sehen:

```
Dim adr As New Kontakt With {.Nachname = "Dröge",
    .Vorname = "Ute", .PLZ = "59555", .Ort = "Lippstadt"}
```

New oder nicht New – wieso es sich bei Objekten um Verweistypen handelt

Zum besseren Verständnis für alle späteren Kapitel ist es überaus sinnvoll, ein paar Worte zur Speicherung von Objekten, also Instanzen von Klassen zu verlieren. Objektvariablen und Objekte sind nämlich nicht *so* miteinander verbunden, wie man es sich zunächst vielleicht vorstellt. Im Gegenteil: Der Verbund einer Objektvariablen mit den eigentlichen Daten des Objektes hält bestenfalls so gut, wie eine amerikanische Prominentenehe: Was auf den ersten Blick so innig und für immer geschaffen zu sein scheint, ist in der nächsten Sekunde auch schon wieder sauber getrennter Schnee von gestern.

Die Wahrheit ist nämlich: Eine Objektvariable speichert im Grunde genommen nur *einen Speicheradressen-zeiger* (oder einfach nur kurz Zeiger genannt) auf die eigentlichen Daten im Managed Heap, in den die Daten der Objekte gelangen, die mithilfe von `New` aus Klassen instanziiert werden.

Und wie wir (die Älteren unter uns jedenfalls) aus unseren Anfängertagen mit 64ern, Atari STs und MaschinenSprache noch alle wissen, untergliedert sich der Arbeitsspeicher eines Computers in bestimmte Speicherstellen, die alle bestimmte »Hausnummern« (die Speicheradressen) besitzen.

Wenn Sie nun ein Objekt aus einer Klasse erstellen – zum Beispiel indem Sie einen Kontakt-Datentyp³ mit `New` in die Objektvariable `objVarKontakt` instanziiieren – dann legt das Framework die Daten für diese Objektinstanz beispielsweise an Speicheradresse 460.386 auf dem Managed Heap ab, und die Objektvariable wird intern sozusagen zu einer Integer-Variablen (oder auf 64-Bit-Systemen zu einer Long-Variablen), die diese Adresse trägt. Bildlich sieht das folgendermaßen aus:

³ Für diese Erklärung wurde übrigens auch der Datentypname Kontakt dem Namen Adresse vorgezogen, um nicht zu viel Konfusion mit dem Wort Speicheradresse entstehen zu lassen.

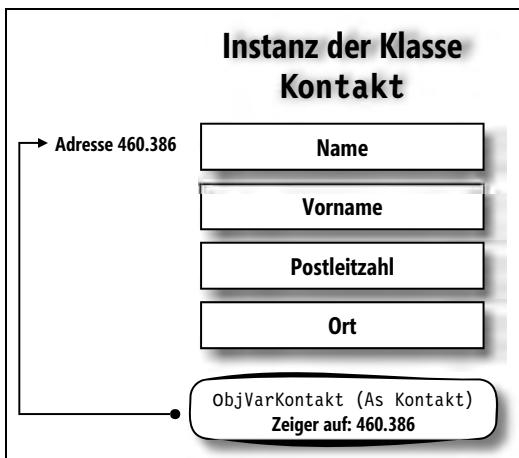


Abbildung 13.1 Objektvariablen speichern im Grunde genommen nur die Speicheradressen auf die eigentlichen Daten, die das Framework im Managed Heap ablegt

Diese Tatsache hat aber entscheidende Folgen, wie das folgende Beispiel gleich zeigen wird.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 13\\KlassenObjekteSpeicher

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module Module1

Sub Main()

    'Instanziieren mit New und dadurch
    'Speicher für das Kontakt-Objekt
    'auf dem Managed Heap anlegen.
    'Instanz gleichzeitig mit Daten initialisieren.
    Dim objVarKontakt As New Kontakt With {.Nachname = "Pfeiffer", _
        .Vorname = "Ute", .PLZ = "59555", .Ort = "Lippstadt"}

    'Nur Objektvariable anlegen,
    'es wird aber kein Speicher reserviert!
    Dim objVarKontakt2 As Kontakt

    'objVarKontakt2 "zeigt" ab jetzt auf
    'dieselbe Instanz wie objVarKontakt
    objVarKontakt2 = objVarKontakt

    'Und das kann man auch beweisen:
    'Das Ändern der Instanz geschieht...
    objVarKontakt2.Nachname = "Dröge"

    'durch beide Objektvariablen, die natürlich
    'auch dasselbe widerspiegeln.
    Console.WriteLine(objVarKontakt.Nachname)
```

```

Console.WriteLine()
Console.WriteLine("Zum Beenden Taste drücken")
Console.ReadKey()
End Sub

End Module

Public Class Kontakt
    Public Vorname As String
    Public Nachname As String
    Public PLZ As String
    Public Ort As String
End Class

```

Wenn Sie dieses Beispiel laufen lassen, erhalten Sie die Ausgabe

```

Dröge
Zum Beenden Taste drücken

```

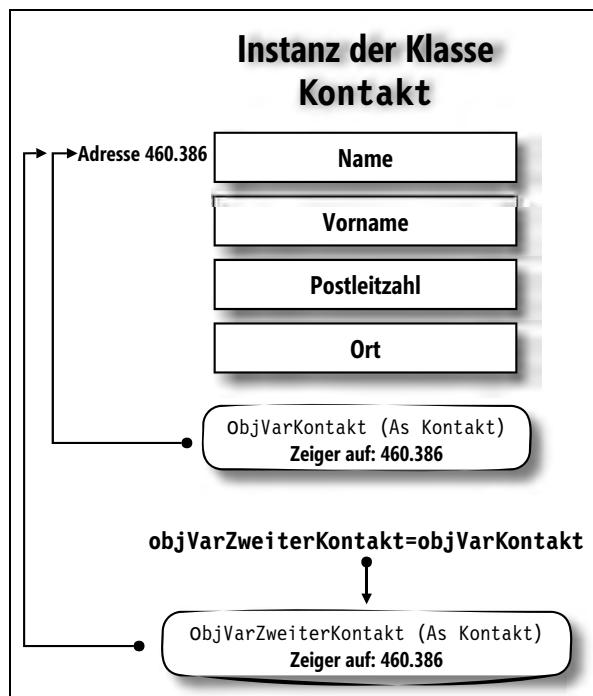


Abbildung 13.2 Das Kopieren einer Objektvariablen in eine andere Objektvariable kopiert nur den Zeiger auf die Instanz – die dann durch beide Variablen manipulierbar und abrufbar wird

Beachten Sie, dass es in diesem Beispiel zwar nur eine einzige Dateninstanz der Klasse Kontakt gibt, die jedoch durch zwei Objektvariablen angesprochen und damit auch widergespiegelt wird: Beim Instanziieren wird die Adresse des Speichers, an dem die Daten der Instanz abgelegt werden, in der Objektvariablen objVarKontakt gespeichert. Diese Adresse wird in die Variable objVarKontakt2 übertragen, und wichtig, hierbei wird nicht etwa eine Kopie der gesamten Instanz im Managed Heap angelegt!

Eine Objektvariable »zeigt« also quasi auf die Daten, die sie verwaltet. In anderen Programmiersprachen wie C++ gibt es zu diesem Zweck besondere Variablen, die, wie schon erwähnt, auch als (*Speicheradress-*) Zeiger bezeichnet werden. In Visual Basic wird eine Objektvariable, die die Instanz einer Klasse verwaltet, automatisch zu dem, was man in anderen Programmiersprachen als Zeiger bezeichnet. In Form einer Grafik sieht das Ganze dann aus wie in Abbildung 13.2.

Dieses Verhältnis zwischen Objektvariable und eigentlichem Objekt (eigentlicher Instanz) sollten Sie sich gut einprägen, da sie einerseits Quelle schwer zu findender Fehler ist – schließlich kann es auch versehentlich passieren, dass, wenn Sie nicht aufpassen, eine Objektvariable die Instanz eines Objektes verändert, die eigentlich und ausschließlich durch eine ganz andere Objektvariable angesprochen werden sollte. Andererseits kann Ihnen dieses Verhältnis auch zum Vorteil gereichen, nämlich wenn es darum geht, nicht Objekte zu kopieren, sondern nur die Zeiger auf diese – beispielsweise wenn es beim Sortieren großer Objektmen gen auf Geschwindigkeit ankommt und deswegen keine ganzen Speicherblöcke mit den eigentlichen Daten sondern nur die Zeiger auf die Objektinstanzen kopiert werden sollen.

Nothing

Mit dem Wissen des vorherigen Abschnittes können Sie sich auch erklären, wieso eine Objektvariable den Wert Nothing (null in C# – nur der Vollständigkeit halber erwähnt) aufweisen kann. Wenn Sie mit New eine Instanz einer Klasse erstellen, weisen Sie diese Instanz (genauer: die Adresse dieser Instanz) der Objektvari ablen zu. Erst dann können Sie mithilfe der Objektvariablen auf die öffentlichen Felder, Eigenschaften und Methoden der Klasseninstanz (des Objektes) zugreifen.

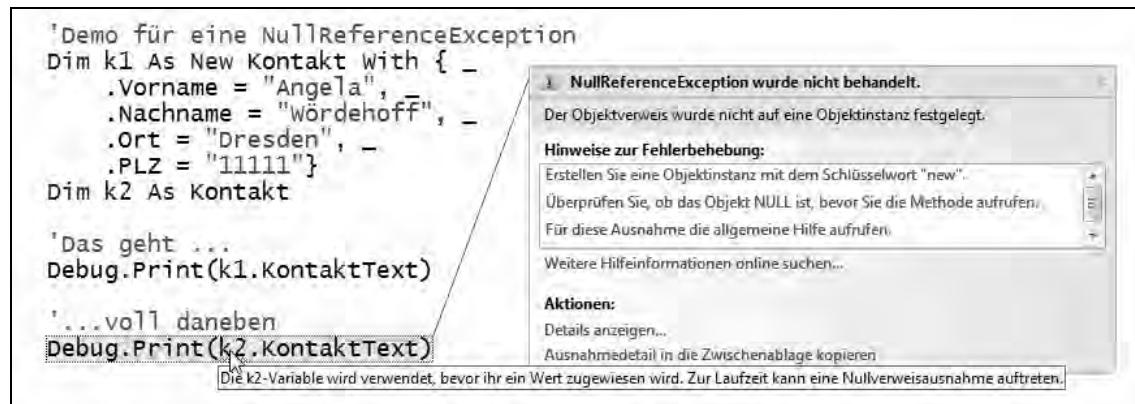


Abbildung 13.3 Wenn Sie mit einer Objektvariablen, der keine Klasseninstanz zugeordnet ist, auf Klassenelemente (Felder, Eigenschaften, Methoden) zugreifen, wird eine NullReferenceException ausgeöst

Wenn Sie versuchen mit einer nicht initialisierten Objektvariablen auf Klassen-Member zuzugreifen, wird zur Laufzeit eine entsprechende Ausnahme ausgelöst. In vielen Fällen kann der Hintergrundcompiler von Visual Basic diesen Zustand voraussehen und zeigt eine entsprechende Warnmeldung an (siehe Abbildung 13.3). Ihre Objektvariable zeigt dann nämlich auf Nichts (auf jeden Fall auf kein gültiges Objekt), also englisch Nothing.

Wenn Sie überprüfen wollen, ob eine Variable auf ein gültiges Objekt oder auf Nothing verweist, müssen Sie übrigens (anders als in C# beispielsweise) in VB.NET leider einen besonderen Vergleichsoperator benutzen. Die Abfrage If locObjEinObjekt = Nothing then... funktioniert leider nicht (warum weiß nur Microsoft), die richtige Syntax lautet: If locObjEinObjekt Is Nothing then...

Klassen anwenden

Natürlich handelt es sich bei Klassen zwar um ganz neue Datentypen, die Sie aber genauso handhaben können, wie alle anderen Datentypen des .NET Frameworks. Und damit können einzelne Instanzen einer Klasse genauso in Arrays abgelegt werden, wie das bei ganz normalen primitiven Datentypen der Fall ist – wir sind damit beim ersten richtigen Anwendungsbeispiel für Klassen, nämlich der zweiten Version unseres Adressenbeispiele.

Mit all den Infos aus den vorherigen Abschnitten können wir das anfängliche Beispielprogramm nun so umschreiben, dass es »nur« noch mit einem Array auskommt, in dem Elemente vom gerade neu geschaffenen Typ Kontakt vorhanden sind, und dazu sind nicht einmal viele Änderungen notwendig:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 13\\MiniAdressoClass

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Die erste betrifft das Einrichten der Datenstruktur am Anfang des Programms:

Module Hauptmodul

 Dim Kontakte(0 To 100) As Kontakt

Im Gegensatz zur ursprünglichen Version kommen wir jetzt mit nur noch einem Array aus, das Elemente enthält, die die Felder eines kompletten Kontaktes kapseln. Viel aufgeräumter sieht nun auch der Sortieralgorithmus aus (Änderungen an ihm sind fett hervorgehoben).

```
Sub AdressenSortieren()
    Dim anzahlElemente As Integer = 101
    Dim aeussererZaehler As Integer
    Dim innererZaehler As Integer
    Dim delta As Integer
    Dim tempKontakt As Kontakt
    delta = 1
    'Größten Wert der Distanzfolge ermitteln
    Do
        delta = 3 * delta + 1
    Loop Until delta > anzahlElemente
```

```
Do
    'Späteres Abbruchkriterium - wieder kleiner werden lassen
    delta = delta \ 3

    'Shellsort's Kernalgorithmus
    For aeussererZeahler = delta To anzahLElemente - 1
        tempKontakt = Kontakte(aeussererZeahler)

        innererZeahler = aeussererZeahler
        Do
            If tempKontakt.Nachname >= Kontakte(innererZeahler - delta).Nachname Then Exit Do
            Kontakte(innererZeahler) = Kontakte(innererZeahler - delta)

            innererZeahler = innererZeahler - delta
            If (innererZeahler <= delta) Then Exit Do
        Loop
        Kontakte(innererZeahler) = tempKontakt
    Next
    Loop Until delta = 0
End Sub
```

Da der neue Datentyp Kontakt die Daten einer kompletten Kontaktadresse kapselt, kann der Fehler aus der letzten Programmversion gar nicht mehr passieren. Der Dreieckstausch vollzieht sich jetzt zwangsläufig an einem vollständigen Kontakt. Somit ist die Gefahr ausgeschlossen, dass einzelne Felder auf der Strecke bleiben. Die fettgedruckten Zeilen im vorherigen Listing dokumentieren das »Aufräumen«.

Wertetypen

Im Gegensatz zu den Referenztypen, die aus Klassen entstehen, gibt es im .NET Framework ein weiteres Grundelement zum Speichern von Daten, und zwar den Wertetyp. Und warum? Ganz einfach: Sie können sich vorstellen, dass es ein vergleichsweise aufwändiger Vorgang ist, schon bei der kleinsten Integervariable Speicherplatz reservieren zu müssen, den Wert, der durch die Integervariable repräsentiert werden soll, dann dort hineinzuschreiben, die Referenz auf diesen Speicherplatz in einer Objektvariablen zu hinterlegen, und wann immer man auf diese Variable zugreifen will, sie wieder mit ganz, ganz langen Armen aus dem Speicher zu fischen. So geht's nicht.

Also führte man im .NET Framework die so genannten Wertetypen ein, und die landen nicht auf dem Managed Heap, also auf dem Objektespeicherhaufen, sondern im Prozessorstack, dem Speicherbereich, auf den der Prozessor am schnellsten zugreifen kann. Bei einigen primitiven Datentypen erzeugt der CIL-Compiler so optimierten Code, dass bei der Verwendung einer Integer-Variable sogar direkt ein Prozessorregister für die Speicherung des Wertes verwendet werden kann. Das ist doppelt performant, denn bei dieser Vorgehensweise fällt auch das Dereferenzieren durch eine Adressvariable unter den Tisch. Der Wert wird direkt auf dem Stack abgelegt oder sogar in einem Prozessorregister gehalten, daher auch der Ausdruck Wertetyp.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

... \VB 2008 Entwicklerbuch\ C - OOP\ Kapitel 13\ ReferenzVsWertetypen

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Das hat allerdings auch Auswirkungen auf die Handhabung, denn wenn der Speicher für einen Wertetyp implizit zugewiesen wird, brauchen Sie ihn auch nicht explizit mit `New` anzufordern.

Und genau das ist der Grund, weswegen Sie das folgende Konstrukt zwar schreiben können, es aber überhaupt keinen Sinn ergibt:

```
Dim intVar As New System.Int32()
intVar = 5
Dim intVar As Integer
```

Abbildung 13.4 Sie können jeden primitiven Typ über sein Framework-Äquivalent deklarieren – im Ergebnis macht das keinen Unterschied

Sie sehen in der Abbildung anhand des Tooltips, dass diese Konstruktion auf den ersten Blick auch keinen Unterschied bei der Definition des VariablenTyps macht: Obwohl Sie `System.Int32` – das ist der `Integer`-Typ aus Sicht der Frameworks – angegeben haben, sagt der Tooltip, dass `intVar` vom Typ `Integer` ist.

Was hier passiert, ist, dass Sie einen Typ, der eigentlich ein Wertetyp ist, als Objekt anlegen – denn Sie haben das Schlüsselwort `New` angegeben. Doch mit dieser so erstellten Instanz des `Int32`-Objektes passiert anschließend gar nichts. Es wird auf dem Managed Heap angelegt und verfällt. Der Garbage Collector – die Müllabfuhr in .NET, auf die wir im Detail noch in Kapitel 18 zu sprechen kommen – entsorgt es bei nächster Gelegenheit.

In der Tat wird hier ein zusätzlicher Speicherbereich verwendet – entweder auf dem Prozessorstack oder, wenn später der optimierte Code zur Ausführung kommt, eben ein Prozessorregister. Denn `System.Int32` ist nun einmal ein Wertetyp, und, wie der Tooltip richtig informiert, als solcher wird er auch verwendet. Die betroffene Codezeile hat also nur eine einzige Daseinsberechtigung, und die ist, die Variable selbst vom Typ `Int32` zu definieren, was normalerweise durch den Compiler auch mit dem Schlüsselwort `Integer` geschieht.⁴

Wertetypen müssen also nicht instanziert werden, sie können direkt verwendet werden. Die CLR sorgt dafür, dass Wertetypen einfach anders gehandhabt werden, und zwar, wie gesagt, rein aus Performance-Gründen.

Und exakt dieses Verhalten ergibt nicht nur für primitive Datentypen Sinn, sondern auch für Datentypen, von denen Sie der Meinung sind, dass sie sehr performant verarbeitet werden sollen.

⁴ Das Konstrukt `New Integer()` hätte der Compiler im Übrigen direkt mit einem Fehler quittiert, da er »weiß«, dass das Konstrukt lediglich *eine Instanz* von `Int32` als *Objekt* anlegen würde, auf die es keine Referenz gäbe: Die einzige Variable hält nämlich die Verbindung zum Speicherplatz für den `Integer` auf dem Stack.

Wertetypen mit Structure erstellen

Prinzipiell erstellen Sie Wertetypen genau wie Referenztypen. Doch während Referenztypen aus Instanzen von Klassen entstehen, schaffen Sie die Vorlagen für Wertetypen mit so genannten Strukturen.

So wir also unsere Beispielklasse Kontakt als Wertetyp erstellen möchten – wir nennen sie im Folgenden zur besseren Unterscheidung KontaktStructure – verwenden wir dazu das Schlüsselwort Structure im Code, der dann auf das vorhandene Klassenbeispiel übertragen, folgendermaßen auszusehen hat:

```
Public Structure KontaktStructure
    Public Vorname As String
    Public Nachname As String
    Public PLZ As String
    Public Ort As String
End Structure
```

Für diese Struktur gilt jetzt das Gleiche wie für primitive Variablen – Sie brauchen sie nicht mehr mit New zu instanziieren, sodass der folgende Code funktioniert:

```
'Ein Kontakt wird als Struktur definiert und kann
' auch ohne New sofort verwendet werden - der Speicher
' wird implizit auf dem Stack zur Verfügung gestellt.
Dim einKontakt As KontaktStructure
einKontakt.Vorname = "Angela"
einKontakt.Nachname = "Wördehoff"
einKontakt.PLZ = "59555"
einKontakt.Ort = "Lippstadt"
```

Wertetypen durch Zuweisung klonen

Und noch ein anderes Verhalten ist bemerkenswert – in Anlehnung an die primitiven Datentypen aber nur logisch, und jetzt, wo Sie wissen, wie Wertetypen gehandhabt werden, ist diese Verhaltensweise auch einleuchtend:

Genau wie primitive Datentypen werden die Inhalte eines Wertetyps durch Zuweisung an eine andere Variable gleichen Typs kopiert – man nennt dieses Verhalten auch klonen. Schauen Sie sich den folgenden Code an:

```
'Und hier zeigt sich das andere Verhalten von
'Wertetypen zu Referenztypen:
'Beim Zuweisen Wertetyp zu Wertetyp werden keine Referenzen,
'sondern die Inhalte kopiert.
Dim einAndererKontakt As KontaktStructure
einAndererKontakt = einKontakt

'Ein Ändern der einen "Instanz" hat damit
'keine Auswirkung auf das Ändern der zweiten Instanz.
einAndererKontakt.Nachname = "Löffelmann"
Console.WriteLine(einKontakt.Nachname & ", " &
                  einKontakt.Vorname)
Console.ReadLine()
```

Dieser Code gibt die Zeile

Löffelmann, Angela

aus. Anders als bei Referenztypen zeigt die zweite Variable des Beispiels nämlich nicht auf die erste Instanz von Kontakt; es gibt nämlich keine Instanz im Objektsinne. Vielmehr liegen die relevanten Daten auf dem Stack, und wenn Sie diese Daten durch eine Zuweisung einer anderen Wertevervariablen zuweisen, dann werden diese Daten dupliziert. Wertetypvariablen können also bestenfalls die gleichen aber niemals dieselben Daten preisgeben.

HINWEIS Fürs Erste soll diese Beschreibung des Unterschieds zwischen Werte- und Referenztypen ausreichen. Es gibt bei der Programmierung, gerade wenn Sie Daten in Auflistungen und Arrays speichern, noch viel mehr Wissenswertes, mit dem sich dann Kapitel 16 und 17 auseinander setzen werden.

Wann Werte-, wann Referenztypen verwenden?

Wir haben ja gehört, dass Wertetypen in erster Linie aus Performance-Gründen geschaffen worden sind. Und um bestimmte Algorithmen schnell zu verarbeiten, kommt man am Besten auch mit wenig Daten aus, die zum Steuern dieser Daten erforderlich sind. Genau diese Daten sollten als Wertetypen angelegt werden. Beispiele:

- Eine Grafik ist, wenn Sie von der Festplatte in den Arbeitsspeicher geladen wird, ein riesiger Speicherbereich, der natürlich als Referenztyp angelegt werden sollte. Koordinaten hingegen, die Positionen auf der Grafik beschreiben, und häufig in Algorithmen zur Berechnung dieser Grafik einfließen, ergeben als Wertetypen natürlich mehr Sinn, denn mit ihnen muss schnell hantiert werden.
- Datensätze, die Sie zum Beispiel im Rahmen einer Kontaktmanagement-Anwendung im Speicher halten müssen, benötigen viel Platz; sie sollten daher als Referenztypen konzeptioniert werden. Eine Datenstruktur, die lediglich eine Beziehung zwischen einer Datensatz-ID und einem Suchschlüssel herstellt, könnte wiederum als Wertetyp wieder besser platziert sein.
- Sollten Sie den Ehrgeiz entwickeln, einen eigenen Webbrowser zu programmieren, würden die Daten, die eine HTML-Seite ausmachen, natürlich als Referenztyp angelegt werden. Ein Element einer DNS-Liste, die Aufschluss zwischen IP-Adresse und Hostheader gibt, machte als Wertetyp vielleicht wieder mehr Sinn.
- Wertetypen sind auch dann sinnvoll, wenn Sie die vorhandenen primitiven Datentypen um weitere ergänzen wollen. Der SQL Server kennt ab der 2008er Version beispielsweise geographische Daten; diese sind nicht nur komplett in Managed .NET Code realisiert, sondern bei ihnen handelt es sich auch um Wertetypen auf Basis von Strukturen.
- Denkbar als Wertetypen sind auch neue numerische Typen, die andere Zahlensysteme berücksichtigen – dazu finden Sie ebenfalls im Kapitel 16 noch ein ausführliches Beispiel – oder mit größeren Zahlbereichen hantieren müssen, als es die eingebauten numerischen primitiven Datentypen könnten.

Sie sehen: Wann immer kleine Datenmengen schnell verarbeitet werden sollen, bietet es sich an, Wertetypen auf Basis einer Struktur zu entwerfen. Wenn Datentypen komplexer werden, oder selbst auf Referenztypen basieren, sollten Sie sich für Ihren Entwurf als Referenztyp auf Basis einer Klasse entscheiden.

Infofern handelte es sich bei unserer Kontakt-Struktur sicherlich um ein schlechtes Beispiel, aber um ein anschauliches.

Konstanten vs. Felder (Klassen-Membervariablen)

Neben den Feldern, die Sie innerhalb von Klassen und Strukturen als Member für die eigentlichen Daten implementieren können, kennt Visual Basic .NET bzw. die CLR auch Konstanten. Konstanten fühlen sich an wie Variablen, werden allerdings intern komplett anders gespeichert – um nicht zu sagen: Sie werden eigentlich gar nicht wie Variablen gespeichert.

Konstanten verwenden Sie immer dann, wenn Sie einen definierten Wert an mehreren Stellen innerhalb Ihres Projektes benötigen, Sie ihn aber zentral »verwalten« wollen. So könnte der Name Ihrer Anwendung, der an mehreren Stellen innerhalb der Anwendung verwendet werden soll, als Konstante definiert werden. Sie definieren eine Konstante syntaxmäßig genau so wie eine Variable, nur, dass Sie zusätzlich das Schlüsselwort `Const` verwenden. Zum Beispiel:

```
Public Const anwendungsName As String = "Typdemo"
```

Auf diese Weise können Sie natürlich auch andere Konstanten definieren – Sie brauchen lediglich den entsprechenden Typ hinter der `As`-Klausel anzugeben.

Doch aufgepasst:

WICHTIG Damit könnte man meinen, dass man Konstanten auch als Feldvariablen »missbrauchen« könnte, die nur lesbar sind. Das kann allerdings fürchterlich nach hinten losgehen, wenn auf öffentliche Konstanten in anderen Assemblies zugegriffen wird.

Wenn Sie eine Konstante mit einem Wert versehen, wird nämlich niemals ein konkreter Wert erzeugt und zur Laufzeit des Programms gespeichert. Konstanten benötigen so gesehen also überhaupt keinen Speicherplatz zur Laufzeit – weder auf dem Managed Heap noch auf dem Stack noch innerhalb irgendwelcher Prozessorregister. Der eigentliche Wert versteckt sich vielmehr in den so genannten Meta-Daten der Assembly, in der die Konstante definiert wird. Und das kann tragischerweise zu Versionsproblemen führen.

Konstanten können nämlich, eben weil sie nur in den Metadaten Ihrer Assembly gespeichert werden, nur zur Compile-Zeit Ihrer Anwendung ausgewertet werden. Wenn Sie also beispielsweise in Assembly A eine öffentliche Konstante definieren, und auf diese von einer Assembly B aus zugreifen, benötigen Sie Assembly A im Grunde genommen nur während der Erstellung von Assembly B. Der Compiler schaut nämlich vereinfacht ausgedrückt lediglich in Assembly A nach, während er Assembly B erstellt, und überträgt den Werte der Konstanten in Assembly B an den Stellen, an denen es notwendig ist. Bräuchten Sie die Assembly A ausschließlich zum Abrufen der Konstanten, könnten Sie sie im Anschluss genauso gut entsorgen. Und dabei liegt das Problem. Da nicht wirklich zur Laufzeit auf die Assembly A zugegriffen wird, sondern nur zur Compile-Zeit, ist es nicht ausreichend, Assembly A auszutauschen, falls sich der dort definierte Wert der Konstante ändert. Da Assembly B nämlich nur zur Compile-Zeit auf Assembly A zugegriffen hat, bekommt sie davon nichts mit.

Aus diesem Grund müssen Sie mit öffentlichen Konstanten vorsichtig sein, und eine Anwendung grundsätzlich inklusive aller beteiligten Assemblies neu erstellen, wenn sich eine wie hier beschriebene Konstellation ergibt. Möchten Sie dieses Verhalten bei diesem Szenario von vorneherein vermeiden, sollten Sie alternativ auf so genannte Nur-Lesen-Eigenschaften zurückgreifen. Den Themenkomplex *Eigenschaften*, der erklärt, wie das funktioniert, finden Sie im nächsten Kapitel beschrieben.

Kapitel 14

Klassencode entwickeln

In diesem Kapitel:

Eigenschaften	408
Der Konstruktor einer Klasse – bestimmen, was bei New passiert	418
Klassenmethoden mit Sub und Function	426
Überladen von Methoden, Konstruktoren und Eigenschaften	427
Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften – Gültigkeitsbereiche definieren	434
Statische Komponenten	437
Delegaten und Lambda-Ausdrücke	441
Über mehrere Codedateien aufgeteilter Klassencode – Das Partial-Schlüsselwort	447

Betrachten wir nochmals die anfängliche Definition von Klassen: »Eine Klasse schafft die Strukturen für das Speichern von Daten und beinhaltet gleichzeitig Programmcode, der diese Daten reglementiert.« Bislang haben wir erst die eine Komponente einer Klasse kennen gelernt: sogenannte Member-Variablen, die die Möglichkeit schaffen, bestimmte Dinge in einer Klasseninstanz zu speichern.

Der zweite Teil, der den Zugriff auf die Daten reglementieren soll, fehlt bislang. Das Klassenkonzept sieht es vor, dass datenreglementierender Code in Form von zwei Prozedurtypen in einer Klasse vorhanden sein kann: In Methoden (Sub, Function) und in Eigenschaftenprozeduren (Property).

Darüber hinaus haben Sie die Möglichkeit, auch durch den sogenannten Konstruktor zu bestimmen, was beim Instanziieren einer Klasse passieren soll – also außer den Maßnahmen, welche die CLR sowieso für sie durchführt, wie beispielsweise das Reservieren des Speichers für die eigentlichen Daten Ihrer Klasseninstanz.

Eigenschaften

Die bislang einzige Beispielklasse dieses Kapitels bestand aus nur vier Member-Variablen, auf die man, da sie mit dem Schlüsselwort `public` (also als öffentlich zugänglich) definiert wurden – in jeder Instanz zugreifen konnte. Das hat mit der Kapselung von Daten natürlich denkbar wenig zu tun; dieses Beispiel diente bislang lediglich dazu, verschiedene Daten zu einer logischen Dateneinheit zusammenzufassen – von wirklicher Kapselung der Daten, bei der das Rein und Raus der Daten durch irgendeine Klassenpolizei geregelt wurde, konnte man dabei allerdings noch nicht sprechen.

Jetzt stellen Sie sich vor, Sie möchten, dass die letzten drei Buchstaben des Ortes unserer Kontakte-Klasse, falls er mehr als 30 Buchstaben hat, durch drei Auslassungspunkte »...« ersetzt werden.

In der prozeduralen Welt würden Sie dazu eine Funktion entwickeln, die überprüft, ob eine Zeichenfolge mehr als dreißig Zeichen aufweist. Wäre das der Fall, würden Sie sie jedes Zeichen nach dem 30. Zeichen abschneiden sowie die letzten drei Zeichen der Zeichenkette durch die drei Auslassungspunkte ersetzen lassen.

Sie würden dann nicht direkt auf die Feldvariable zugreifen, sondern, wie im folgenden Beispiel, jeden Zugriff auf das Feld `Ort` in einen Funktionsaufruf einbauen. Das könnte dann im Ergebnis folgendermaßen ausschauen:

Wir haben also zunächst unsere »AuslassungszeichenAnText«-Routine ...

```
Function EllipseString(ByVal text As String, ByVal MaxLength As Integer) As String
    Dim tmpText As String

    If text.Length > MaxLength Then
        tmpText = text.Substring(0, MaxLength - 3) + "..."
    Else
        tmpText = text
    End If
    Return tmpText
End Function
```

... und diese Routine rufen Sie immer dann auf, wenn Sie auf das kritisch zu behandelnde Feld `Ort` einer Instanz der Klasse `Kontakt` zugreifen:

```
Sub KlasseVerarbeiten()
    .
    .
    .
    Dim tmpKontakt As New Kontakt
    .
    .
    .
    tmpKontakt.Ort = EllipseString(Console.ReadLine, 30)
End Sub
```

Es gibt aber noch eine viel elegantere und vor allen Dingen *sichere* Möglichkeit, dafür zu sorgen, dass die Member-Variable Ort der Kontakt-Klasse niemals eine zu lange Zeichenkette zugewiesen bekommt: Und das sind die sogenannten *Eigenschaften-Prozeduren*.

Wenn Sie schon längere Zeit mit Visual Basic (egal, ob mit .NET oder 6.0) programmieren, haben Sie Eigenschaften vermutlich längst kennen gelernt. Mithilfe von Eigenschaften können Sie in der Regel bestimmte Zustände von Objekten abfragen *und* verändern, und von außen »fühlt« sich das dann so an, als würden Sie direkt ein öffentliches Feld einer Klasse manipulieren oder abfragen. Möchten Sie beispielsweise wissen, ob die Schaltfläche eines Formulars anwählbar ist, verwenden Sie die Eigenschaft in Abfrageform, etwa wie hier:

```
If Schaltfläche.Enabled Then TuWas
```

Oder Sie legen die Eigenschaft eines Objektes fest, etwa wie mit der folgenden Zeile:

```
Schaltfläche.Enabled = false      ' Abschließen, an den Button kommt keiner mehr 'ran
```

Sie wissen nun aber aus Erfahrung, dass Sie beim Setzen einer Schaltfläche nicht nur den Zustand einer Variablen dieser Button-Instanz verändern, sondern dass das Setzen dieser Eigenschaft auch noch etwas Weiteres bewirkt – im Fallbeispiel nämlich, dass die Schaltfläche auch sichtbar auf dem Formular ausgegraut wird (oder eben wieder – beim Zuweisen von True – den visuellen Einschaltzustand bekommt). Der Code, der dafür sorgt, muss innerhalb der Klasse natürlich irgendwo platziert werden, und so lautet die viel interessantere Frage: Wie statten Sie Ihre eigenen Klassen mit Eigenschaften aus?

Visual Basic stellt Ihnen zu diesem Zweck, wie schon erwähnt, Eigenschaftenprozeduren zur Verfügung. Eine Eigenschaft wird in Visual Basic innerhalb einer Klasse folgendermaßen definiert:

```
Property EineEigenschaft() As Datentyp
    Get
        Return DatentypObjektvariable
    End Get
    Set(ByVal Value As Datentyp)
        DatentypObjektvariable = Value
    End Set
End Property
```

Wenn Sie diese Eigenschaft in einer Klasse implementieren, können Sie sie bei instanzierten Objekten dieser Klasse auf folgende Weise verwenden:

Zuweisen von Eigenschaften

Mit der Anweisung

```
Object.EineEigenschaft = Irgendetwas
```

weisen Sie der Eigenschaft `EineEigenschaft` des Objektes einen Wert zu. Sie können im *Set-Accessor*¹ (`Set(ByVal Value as Datentyp)`) der Eigenschaftenprozedur mit `Value` auf das Objekt zugreifen, das sich in `Irgendetwas` befindet. Nur der *Set*-Teil der Eigenschaftenprozedur wird in diesem Fall ausgeführt.

Ermitteln von Eigenschaften

Umgekehrt können Sie mit der Anweisung

```
Irgendetwas = Object.EineEigenschaft
```

den Inhalt der Eigenschaft wieder auslesen. In diesem Fall wird nur der *Get-Accessor* der Eigenschaftenprozedur ausgeführt, die das Ergebnis mit `Return` zurückliefert und dieses der Objektvariablen `Irgendetwas` zuweist, die natürlich vom gleichen Typ wie die Eigenschaft sein muss.

Mit diesem Wissen können wir unsere Klasse jetzt folgendermaßen umschreiben:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 14\\MiniAdressoClass V2
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Public Class Kontakt
    Private myVorname As String
    Private myNachname As String
    Private myPLZ As String
    Private myOrt As String

    Public Property Vorname() As String
        Get
            Return myVorname
        End Get
        Set(ByVal value As String)
            myVorname = value
        End Set
    End Property
```

¹ Etwa: »Zugreifer«.

```
Public Property Nachname() As String
    Get
        Return myNachname
    End Get
    Set(ByVal value As String)
        myNachname = value
    End Set
End Property

Public Property PLZ() As String
    Get
        Return myPLZ
    End Get
    Set(ByVal value As String)
        myPLZ = value
    End Set
End Property

Public Property Ort() As String
    Get
        Return myOrt
    End Get
    Set(ByVal value As String)
        myOrt = EllipseString(value, 30)2
    End Set
End Property

Private Function EllipseString(ByVal text As String, ByVal MaxLength As Integer) As String
    Dim tmpText As String

    If text.Length > MaxLength Then
        tmpText = text.Substring(0, MaxLength - 3) + "..."
    Else
        tmpText = text
    End If
    Return tmpText
End Function

End Class
```

Sie sehen an diesem einfachen Beispiel, wie der Schutz der eigentlichen Datenstruktur durch Eigenschaftenprozeduren vonstatten geht: Die Daten, die in einer Klasseinstanz gespeichert werden, weisen private Zugriffsmodifizierer auf (mehr zu Zugriffsmodifizierern finden Sie im Abschnitt »Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften« ab Seite 434), und sie können damit nicht mehr direkt von außen, sondern nur noch durch den Klassencode selbst verändert werden. Die Member-Variablen heißen auch

² Fußnote im Listing? Wieso nicht: Der Fachlektor dieses Buchs regt an dieser Stelle zurecht eine Alternative an: Anstelle myOrt hier wirklich »kaputt« zu machen, könnte man die Ursprungszuweisung auch in der vollen Länge speichern, und beim Abrufen der Eigenschaft EllipseString aufrufen. So könnte man im Bedarfsfall immer noch auf die ursprüngliche Eigenschaftendefinition zugreifen.

nicht mehr wie zuvor, sondern tragen den Zusatz bzw. das Präfix »my« als rein textlichen Hinweis darauf, dass es sich dabei um »meine« Variablen aus Klassen- bzw. Klasseninstanzsicht handelt. Dies ist eine durchaus übliche, aber nicht vorgeschriebene Konvention – Sie können die Benennung der Member-Variablen nach eigenem Belieben vornehmen; Sie sollten allerdings darauf achten, dass Sie die Member-Variablen anders als Ihre von außen zugreifbaren Eigenschaften nennen, denn das würde zu einem Kompilierungsfehler führen.

Die Eigenschaftenprozeduren schließlich heißen so wie die Namen der ursprünglich öffentlichen Member-Variablen. Aus Entwicklersicht hat sich damit nichts an der Handhabung geändert – im Ergebnis allerdings schon, denn durch die Eigenschaftenroutinen wird nun der Zugriff auf die Member-Variablen exklusiv geregelt. In diesem kleinen Beispiel sehen Sie, dass der Ort durch dieses Verfahren niemals mehr als 30 Zeichen aufweisen wird. Bei mehr als 30 Zeichen wird durch das Einbinden der `EllipseString`-Methode die Anpassung der Zeichenkette mit den Auslassungszeichen vorgenommen und auf den eigentlichen Datenspeicher des Orts – die Member-Variable `myOrt` – kann von außen niemand mehr zugreifen, da ihr Zugriff durch Verwendung des Zugriffsmodifizierers `private` für diese (wie auch für alle anderen Member-Variablen) geschützt ist. Genau das ist es, was eines der Hauptanliegen einer Klasse ist, nämlich *Daten zu kapseln und den Zugriff auf sie durch Klassencode zu reglementieren*.

Eigenschaften mit Parametern

In Visual Basic können Eigenschaften, wie Funktionen, beliebig viele Parameter übernehmen. Die Parameter werden in der Eigenschaftenprozedur genauso wie bei Funktionen eingebunden, und auf die Parameter lässt sich dann ebenfalls genauso Zugriff darauf nehmen.

Ein Beispiel: Angenommen, Sie haben eine Eigenschaftenprozedur etwa wie die folgende definiert,

```
Property EigenschaftMitParametern(ByVal Par1 As Integer, ByVal Par2 As String) As Integer
    Get
        If Par1 = 0 Then
            Return 10
        Else
            Return 20
        End If
    End Get

    Set(ByVal Value As Integer)
        If Par2 = "Klaus" Then
            'MachIrgendEtwas
        ElseIf Par1 = 20 Then
            'MachIrgendetwasAnderes
        End If
    End Set

End Property
```

dann können Sie sie mit beispielsweise folgenden Anweisungen zuweisen bzw. abfragen:

```
'Eigenschaft mit Parametern setzen.
locObjektInstanz.EigenschaftMitParametern(10, "Klaus") = 5
```

```
'Eigenschaft mit Parametern abfragen.  
If locObjektInstanz.EigenschaftMitParametern(20, "Test") = 20 Then  
    'Mach Irgendetwas  
End If
```

HINWEIS Im Gegensatz zu Visual Basic 6.0 werden Wertetypen-Parameter als Wert und nicht als Referenz übergeben. Änderten Sie einen Wert innerhalb einer Eigenschaftenprozedur in Visual Basic 6.0, so veränderte sich auch der Wert der Variablen im aufrufenden Code (es sei denn, er war explizit durch Einklammern vor dem Überschreiben geschützt oder Sie haben, wie es in VB.NET automatisch geschieht, die Übergabe ByVal und ByRef explizit angegeben). In Visual Basic .NET übergeben Sie standardmäßig den Wert an eine Eigenschaft – Sie arbeiten also in jedem Fall mit einer Kopie der übergebenden Variablen.

Eine Differenzierung mit Set und Let wie in Visual Basic 6.0 gibt es übrigens in Visual Basic .NET nicht mehr. Da alles von Object abgeleitet ist und dadurch im Grunde genommen ausschließlich Objekte manipuliert werden, ist auch das Let überflüssig geworden, und alles müsste mit Set manipuliert werden – und dann kann man Set auch direkt weglassen.

WICHTIG Eigenschaften mit Parametern sollten Sie nur in Ausnahmefällen verwenden, da sie eine besondere Eigenart von Visual Basic sind. Im Gegensatz zu Visual Basic kennt beispielsweise C# zwar auch Eigenschaften, kann aber nur auf parameterlose Eigenschaften oder Default-Eigenschaften zugreifen (mehr dazu im Abschnitt »Default-Eigenschaften« im folgenden Abschnitt). Wenn Sie also Klassen im Team oder für die breite Öffentlichkeit entwickeln, und erwarten, dass Ihre Assemblies auch von Entwicklern genutzt werden, die in einer anderen .NET-Sprache als Visual Basic .NET entwickeln, sollten Sie auf Eigenschaften mit Parametern nach Möglichkeit ganz verzichten.

Default-Eigenschaften (Standardeigenschaften)

Default-Eigenschaften (Standardeigenschaften) haben in Visual Basic 6.0 eine erleichternde Funktion für schreibfaule Entwickler gehabt. Mithilfe einer Default-Eigenschaft konnten sie bestimmen, welche Eigenschaft verwendet wird, wenn Sie beim Zugriff auf ein Objekt gar keine Eigenschaft verwendet haben. Das CTS erlaubt derartige Typunsicherheiten nicht – denn bei der Zuweisung beispielsweise von

```
Dim EineTextBox as TextBox  
Dim einObjekt as Object  
. . .  
einObject = EineTextBox
```

ist natürlich nicht sichergestellt, ob Sie die TextBox selbst oder das Ergebnis der Default-Eigenschaft der TextBox an einObjekt zuweisen wollen. Ausnahmen bilden dabei parametrisierte Eigenschaften, da durch die Signatur der Eigenschaft deutlich wird, dass Sie nicht das Objekt selbst, sondern das Resultat einer parametrisierten Eigenschaft zurückliefern wollen.³ In diesem Fall ergibt das auch Sinn, denn:

³ Ähnlich wie bei Überladungen, bei denen auch nur durch die Signaturen unterschieden wird, welche der mehreren vorhandenen Funktionen gemeint ist.

Stellen Sie sich vor, Sie entwickeln eine `NumericList`-Klasse, die verschiedenste numerische Werte als Liste speichert, verwaltet, und die es erlaubt, auf diese mit einem Index zuzugreifen, um diese Elemente abzurufen. Gäbe es keine Eigenschaften mit Parametern, müssten Sie ein Element der Liste auf folgende Weise abfragen:

```
'Index setzen.  
numListe.Index = 5  
'Element abfragen.  
MachEtwasMit = numListe.Item
```

Das wäre natürlich äußerst umständlich. Einfacher wird es so, wie es auch tatsächlich funktioniert, nämlich durch die Spezifizierung des Indexes und die Abfrage in einer Zeile. In Visual Basic ist das kein Problem durch eine Eigenschaft mit einem Parameter, etwa wie folgt:

```
MachEtwasMit = numListe.Item(5)
```

Noch einfacher wird es, wenn die Eigenschaft `Item` in diesem Beispiel zur Default-Eigenschaft erklärt wird. Dann brauchen Sie den Eigenschaftenamen nämlich gar nicht mehr anzugeben, und die folgende Zeile wäre ausreichend:

```
MachEtwasMit = numListe(5)
```

Die entsprechende Definition für die `Item`-Eigenschaft sähe in diesem Fall folgendermaßen aus:

```
Default Public Property Item(ByVal Index As Integer) As Integer  
  
    Get  
        Return myInterneListe(Index)  
    End Get  
  
    Set(ByVal Value As Integer)  
        myInterneListe(Index) = Value  
    End Set  
  
End Property
```

WICHTIG Im Gegensatz zu Visual Basic 6 gibt es in Visual Basic .NET keine parameterlosen Default-Eigenschaften. Ebenfalls gut zu wissen: Default-Eigenschaften können nicht als statisch deklariert werden. Das hindert Sie aber natürlich nicht daran, eine Eigenschaft zu implementieren, die sich statisch verhält. Die folgende Beispielimplementierung macht das deutlich.

Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage?

Jetzt haben Sie schon so viel über Eigenschaften erfahren – vielleicht fragen Sie sich, wieso man sie anstelle von einfachen öffentlichen Member-Variablen einsetzen sollte.

Solange, wie Sie Eigenschaften in einer Klasse nur benötigen, um irgendwelche Werte zu speichern, aber beim Abfragen oder Setzen dieser Werte nichts Weiteres passieren muss, wären öffentliche Variablen eigentlich ausreichend.

Die Klasse

```
Class MitEinerEigenschaft
    Public DieEigenschaft As Integer
End Class
```

erfüllt zunächst nämlich den gleichen Zweck wie die folgende Klasse,

```
Class AuchMitEinerEigenschaft

    Private myDieEigenschaft As Integer

    Public Property DieEigenschaft() As Integer
        Get
            Return myDieEigenschaft
        End Get
        Set(ByVal Value As Integer)
            myDieEigenschaft = Value
        End Set
    End Property
End Class
```

die natürlich sehr viel mehr Schreibarbeit erfordert. Prinzipiell ist das richtig. Und dennoch ist die zweite Methode der ersten Methode vorzuziehen, aus folgenden Gründen, die Ihnen teilweise schon bekannt sind:

- **Datenkapselung:** Eigenschaften kapseln und reglementieren den Zugriff auf Daten, wie Sie im vorangegangenen Beispiel bereits gesehen haben. Auch wenn Sie eine Eigenschaft nur komplett an eine Member-Variable durchreichen – für mögliche Erweiterungen, was Reglementierungsalgorithmen anbelangt, sind Sie auf jeden Fall schon mal vorbereitet.
- **Datenbindung:** Felder lassen sich grundsätzlich nicht für die Datenbindung⁴ verwenden. Vielfach werden nicht nur Quellen wie Datenbanken, sondern auch Objekte, wie beispielsweise Auflistungen (Collections) als Datenquellen verwendet. Öffentliche Felder (also öffentliche Member-Variablen) können dabei nicht als Datenquelle dienen.
- **Polymorphie:** Ein weiteres wichtigeres Argument ist das Ersetzen von Eigenschaften durch die so genannte Polymorphie beim Vererben von Klassen, das wir im nächsten Kapitel noch kennen lernen werden. Vorab schon mal soviel: Eine einmal als öffentlich deklarierte Variable bleibt für alle Zeiten öffentlich. Sie können in vererbten Klassen keine zusätzliche Steuerung hinzufügen, die ihren Zugriff reglementieren würde. Haben Sie hingegen Ihre Daten nur durch Eigenschaftenprozeduren nach außen offen gelegt, können Sie zu einem späteren Zeitpunkt noch zusätzliche Regeln (Bereichsabfragen, Fehler abfangen) hinzufügen oder vorhandene Verhaltensweisen modifizieren. Sie brauchen dazu die ursprüngliche Klasse kein bisschen zu verändern.

⁴ Ein Verfahren, bei dem man Eigenschaften eines Objektes an die eines anderen bindet. Damit kann man Daten zwischen Objekten automatisch synchronisieren, und häufig wird dieses Verfahren bei Steuerelementen angewandt, die den Inhalt einer Datenklasse darstellen sollen, und dessen Inhalt sich automatisch ändert, wenn sich der Datenklasseninhalt ändert.

Eigenschaften sind also öffentlichen Feldern in jedem Fall vorzuziehen. Damit das Erstellen von Eigenschaften nicht zu viel Tipparbeit verschlingt, gibt es in Form der Visual Basic-Codeausschnittsbibliothek eine Eingabehilfe, die der folgende Kasten beschreibt.

Zeit sparen beim Erstellen von Eigenschaftenprozeduren mit Code-Ausschnitten

An der Beispielklasse der vorherigen Abschnitte lässt sich eindrucksvoll aufzeigen, dass das Verwalten, Regeln und auch Schützen der Member-Variablen einer Klasse durch entsprechende Eigenschaftenprozeduren durchaus sinnvoll ist, Ihnen jedoch auch eine Menge Tipparbeit abverlangt.

Mithilfe von Codeausschnitten können Sie sich gerade beim »Properties kloppen«, wie es umgangssprachlich in Entwicklerkreisen genannt wird, eine Menge an Zeit sparen. Möchten Sie eine neue Eigenschaft in Ihrer Klasse einführen, die auf einer Member-Variablen basiert, verfahren Sie am besten wie folgt:

- Schreiben Sie die Zeichenfolge **Pro** an die Stelle, an der Sie die neue Eigenschaftenprozedur erstellen wollen. Sie sehen anschließend einen Dialog, etwa wie auch in Abbildung 14.1 zu sehen.
- Drücken Sie zweimal **Tab**, um den Property-Codeausschnitt einzufügen. Sie sehen anschließend ein Szenario, etwa wie in Abbildung 14.1 zu sehen.

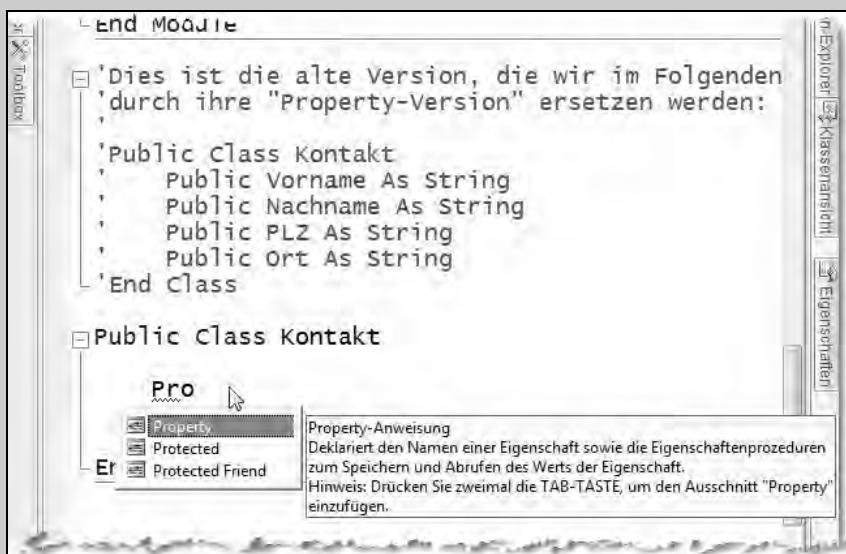


Abbildung 14.1 Um eine Eigenschaftenprozedur durch Codeausschnitte einzufügen, schreiben Sie die Zeichenfolge »Pro«. Drücken Sie, wie im Hinweis der IntelliSense-Liste zu sehen, zweimal **Tab**, um den Property-Codeausschnitt einzufügen, ...

- Editieren Sie anschließend die Member-Variable, die zunächst mit dem Vorgabenamen `newPropertyValue` eingefügt wurde, in den Namen, den Sie für die Member-Variable vorsehen. ►

```
Public Class Kontakt

    Private newPropertyValue As String
    Public Property NewProperty() As String
        Get
            Return newPropertyValue
        End Get
        Set(ByVal value As String)
            newPropertyValue = value
        End Set
    End Property

End Class
```

Abbildung 14.2 ... um dann anschließend die selektierte Member-Variable zu editieren und mit den passenden Datentyp hinzuzufügen, ...

- Sobald Sie gedrückt haben, ändern sich alle Referenzen der Member-Variablen im Codeblock in den neu benannten Namen. Gleichzeitig gelangen Sie zum nächsten editierbaren Feld des Codeausschnitts, dem Typen der Eigenschaftenprozedur. Ändern Sie diesen, und drücken Sie abermals die Taste .

```
Public Class Kontakt

    Private myVorname As String
    Public Property NewProperty() As String
        Get
            Return myVorname
        End Get
        Set(ByVal value As String)
            myVorname = value
        End Set
    End Property

End Class
```

Abbildung 14.3 ... den Sie anschließend im Bedarfsfall ebenfalls editieren, um schließlich mit einem weiteren das Erstellen der Eigenschaftenprozedur mit dem Editieren des eigentlichen Eigenschaftennamens abzuschließen.

- Sobald Sie gedrückt haben, ändern sich alle Referenzen der Member-Variablen im Codeblock in den neuen Namen. Gleichzeitig gelangen Sie zum nächsten editierbaren Feld des Codeausschnitts, dem Typen der Eigenschaftenprozedur.

- Ändern Sie diesen, und drücken Sie abermals die Taste . Damit ändern Sie die Typen sowohl für die Eigenschaftenprozedur (und deren value-Variablen, mit dem im Set-Accessor der Zuweisungswert an die Eigenschaft übergeben wird) als auch für die korrelierende Member-Variable, die in der Eigenschaftenprozedur gesetzt wird.
- Schließlich ändern Sie noch den Namen der Eigenschaft. Damit sind Sie mit der Definition der Eigenschaftsroutine fertig.

Im Ergebnis stehen alle Eigenschaften und deren entsprechende Member-Variablen jeweils als Pärchen zusammen. Viele Entwickler mögen das nicht in dieser Form; sie bevorzugen, dass Member-Variablen am Anfang der Klassencodedatei definiert werden – Sie können die Member-Variablen natürlich anordnen, wie Sie möchten; das tut der Funktionalität der Klasse keinen Abbruch.

Der Konstruktor einer Klasse – bestimmen, was bei New passiert

Wenn Sie eine Klasse in ein Objekt instanziieren, dann schaffen Sie gesteuert durch die Common Language Runtime zunächst mal nur den entsprechenden Speicherplatz für die Member-Variablen auf dem Managed Heap. Intern passieren noch ein paar weitere Dinge, die für die Verwaltung des Objektes auf dem Managed Heap zuständig sind, doch das ist es erstmal gewesen.

In vielen Fällen ist es aber erforderlich, dass gewisse Pflegearbeiten sozusagen zum Vorbereiten der Klasse auf ein korrektes Funktionieren ebenfalls durchgeführt werden können. Denken Sie zum Beispiel daran, was passiert, wenn Sie eine neue Schaltfläche für Ihre Windows Forms-Anwendung instanziieren: Hier muss man auch die notwendige Windows-Infrastruktur berücksichtigen, damit nicht nur Platz für die Daten, die die Steuerinformationen für die Schaltfläche bilden, auf dem Managed Heap geschaffen wird, sondern notwendigerweise unter anderem auch die `CreateWindow`-Funktion der `Win32.Dll` des Windows-Betriebssystems aufzurufen, damit der Button auch aus Betriebssystemssicht entstehen kann.

Oder, um es an einem einfachen Beispiel festzumachen: Wie würden Sie in unsere bisherige Beispielklasse eine Funktionalität implementieren, die automatisch das Datum der Klassenerstellung festhält, sobald eine Instanz der Kontakt-Klasse erstellt wurde? Prinzipiell gibt es dazu zwei Möglichkeiten:

- Sie implementieren eine weitere Eigenschaft, sagen wir namens `ErstellungsZeitpunkt`. Und wann immer Sie eine Instanz der Kontakt-Klasse erstellen, setzen Sie die Eigenschaft auf `Date.Now` – eine Funktion, die die aktuelle Datum/Uhrzeit-Kombination ermittelt. Die Gefahr dabei: Sie könnten diesen Schritt vergessen, und genau das widerspricht eigentlich guter objektorientierter Programmierung, bei der Sie idealerweise Objekte schaffen, die so selbstverwaltet sind, dass Sie solche Dinge gar nicht erst vergessen oder falsch machen *können* – ganz ähnlich, wie Sie mit Eigenschaftenprozeduren den Zugriff auf Klassen-Member so regeln, dass Sie auch diese nicht versehentlich kompromittieren können.
- Die bessere Alternative: Sie implementieren eine Funktionalität, bei der Sie auf die Eigenschaft `ErstellungsZeitpunkt` nur noch lesend zugreifen können – wie das im Detail funktioniert, erfahren Sie ausführlicher im Abschnitt »Unterschiedliche Zugriffsmodifizierer für Eigenschaften-Accessors« ab Seite 436; das ist aber der eher unwichtige Teil. Viel wichtiger in diesem Zusammenhang: Sie implementieren

eine spezielle Methode, die die Member-Variable, auf der die Eigenschaft basiert, automatisch mit dem Erstellungsdatum ausstattet, sobald ein Objekt der Klasse durch ihre Instanziierung ins Leben gerufen wird. Damit garantieren Sie, dass keine Instanz der Klasse erstellt werden kann, ohne dass das Erstellungsdatum *nicht* gesetzt wird.

Genau diesen letzten Punkt wollen wir im Folgenden umsetzen, und dazu bedienen wir uns der Möglichkeit, einen Konstruktor in unserer Klasse zu implementieren. Der Programmcode des Konstruktors wird nämlich genau dann ausgeführt, nachdem die CLR das Schaffen der Infrastruktur hinter sich gebracht hat. Das Erstellen eines Konstruktors funktioniert in Visual Basic mit der Sub New:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 14\\MiniAdressoClass V3

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Public Class Kontakt

    Private myVorname As String
    Private myNachname As String
    Private myPLZ As String
    Private myOrt As String
Private myErstellungsdatum As Date

    Sub New()
        myErstellungsdatum = Date.Now
    End Sub

    Public ReadOnly Property ErstellungsDatum() As Date
        Get
            Return myErstellungsdatum
        End Get
    End Property
    .
    .
    .
End Class
```

Die relevanten Änderungen sind hier im Listing fett hervorgehoben, und erleichtern so das Verdeutlichen der Funktionsweise: Sobald Sie eine neue Instanz der Klasse ins Leben rufen, legt die CLR das Objekt intern an und ruft im Anschluss daran sofort den Konstruktor der Klasse auf. Der Konstruktor – definiert durch Sub New – setzt nun die Member-Variable myErstellungsdatum auf die aktuelle Uhrzeit/das aktuelle Datum. Möchten Sie nun im Folgenden herausfinden, wann die Klasse erstellt worden ist, greifen Sie einfach auf das Erstellungsdatum über die entsprechende Nur-Lesen-Eigenschaft zu. Durch diese Vorgehensweise sind die drei Wunschkriterien unseres Pflichtenheftes erfüllt:

- Die Klasse kann nicht instanziert werden, ohne dass das Erstellungsdatum festgehalten wird – das erledigt nämlich der Konstruktor mit Sub New.

- Das Erstellungsdatum ist durch die Eigenschaftenprozedur abrufbar.
- Das Erstellungsdatum wird aber durch die Verwendung einer *Nur-Lesen-Eigenschaft* eben *nur abrufbar* gemacht, kann also nachträglich nicht verändert werden.

Und damit ist dieser Funktionsblock unserer Klasse bombensicher!

Parametrisierte Konstruktoren

Sicherlich haben Sie bemerkt, dass die Klasse aus dem ersten Beispiel ein wenig unhandlich war. Sie musste erst instanziert werden, daran führt sowieso nie ein Weg vorbei, und anschließend konnten Sie dem aus ihr hervorgehenden Objekt durch die Benutzung von Eigenschaften entsprechende Inhalte zuweisen. Von Klassen, die Sie aus dem Framework möglicherweise bereits verwendet haben, wissen Sie aber sicherlich schon, dass die Instanziierung einer Klasse mit `New` und der Angabe eines Parameters die Initialisierung der Klasseninstanz einfach und unkompliziert macht. Die bisher verwendete Beispielklasse ist hinsichtlich dessen ein wenig armselig, doch das soll sich jetzt ändern.

Wenn Sie möchten, dass Code im Konstruktor einer Ihrer Klassen ausgeführt wird, in dem Sie dann beispielsweise Member-Variablen initialisieren können, dann implementieren Sie ebenfalls `Sub New`. Der einzige Unterschied zu normalen Methoden besteht darin, dass der Prozedurenname aus einem Schlüsselwort besteht (nämlich `New`), welches Visual Studio-Editor und -Compiler als ein solches erkennen und blau markieren. Konstruktoren können übrigens nur aus Subs und nicht aus Functions bestehen – was auch keinen Sinn ergäbe, da das Instanziieren einer Klasse mit `New` ja eigentlich schon das instanzierte Objekt sozusagen als »Funktionsergebnis« von `New` zurückliefert.⁵

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 14\\MiniAdressoClass V4

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Zurück zum Beispielprogramm: Um die Klasseninstanz zu deklarieren *und* ihr Initialwerte zuzuweisen, benötigen Sie mit dieser Version des Beispielprogramms nur noch eine kleine Modifikation. `Sub Main` des Beispielprogramms dieser Version sieht dann folgendermaßen aus:

```
Public Class Kontakt

    Private myVorname As String
    Private myNachname As String
    Private myPLZ As String
    Private myOrt As String
    Private myErstellungsdatum As Date
```

⁵ Wobei diese Erklärung rein technisch natürlich falsch ist und lediglich als Eselsbrücke dienen soll.

```

'Parameterloser Konstruktor
Sub New()
    myErstellungsdatum = Date.Now
End Sub

'Konstruktor übernimmt Initialisierungsparameter
Sub New(ByVal Vorname As String, ByVal Nachname As String, _
        ByVal PLZ As String, ByVal Ort As String)

    'Und ruft - und das geht NUR als erstes oder gar nicht -
    'die parameterlose Konstruktorprozedur auf.
    Me.New()

    'Member-Variablen mit den übergebenen Werten initialisieren.
    myVorname = Vorname
    myNachname = Nachname
    myPLZ = PLZ
    myOrt = Ort
End Sub

Public ReadOnly Property ErstellungsDatum() As Date
    Get
        Return myErstellungsdatum
    End Get
End Property
.
.
.
End Class

```

Und mit dieser Version unserer Klasse können wir einen Kontakt nun auf die folgende Weise instanziieren, und auch Standardparameter direkt zuweisen.

```

Module Hauptmodul

Sub Main()

    'Hier 'mal farbtechnisch was Anderes im Konsolenfenster
    Console.BackgroundColor = ConsoleColor.White
    Console.ForegroundColor = ConsoleColor.Black
    Console.Clear()

    'Eine Adresse mit Parametrisierten Konstruktor angeben:
    Dim AngisKontakt As New Kontakt("Angela", "Wördehoff, ", _
                                    "99999", "Bahamas")

    'So geht's natürlich ab 2008 auch - das *ersetzt* aber nicht
    'den parametrisierten Konstruktor. Außerdem ist's mehr Tipparbeit.
    Dim LöffelsKontakt As New Kontakt With {.Vorname = "Klaus", _
                                             .Nachname = "Löffelmann", _
                                             .PLZ = "59555", _
                                             .Ort = "Lippstadt"}
End Sub
End Module

```

HINWEIS Vielleicht könnte man argumentieren, dass parametisierte Konstruktoren seit Visual Basic 2008 ein wenig obsolet geworden sind, weil Visual Basic 2008 ja nunmehr auch über Eigenschafteninitialisierer beim Konstruktor mit dem Schlüsselwort `With` verfügt – wie hier im zweiten Beispiel zu sehen. Aber das ist nur begrenzt richtig. Denn zum einen können Sie diese Eigenschaftenzuweisungen nur durch die Eigenschaftenprozeduren selbst steuern – ein parametrisierter Konstruktor will hier vielleicht noch andere Wertüberprüfungen übernehmen. Zum anderen ist die Standardwerteübergabe für bestimmte Member mit einem parametrisierten Konstruktor immer noch weniger Tipparbeit – und vielleicht möchten Sie ohnehin Parameter an die Klasse übergeben, die vielleicht durch gar keine Eigenschaft abgedeckt oder offengelegt ist. Denkbar wäre es beispielsweise, dem Konstruktor auch eine Kontakt-Instanz zu übergeben, damit er einen vorhandenen Kontakt klonnt.

Darüber hinaus kennen andere .NET-Sprachen vielleicht keine Eigenschafteninitialisierer – diese werden nämlich ausschließlich durch den Compiler umgesetzt, der entsprechenden Code dafür generiert. Falls Sie also Assemblies entwickeln, auf die von anderen .NET-Sprachen ebenfalls zugegriffen werden soll, sind parametisierte Konstruktoren sicherlich auch ein erweiterter Komfort für den Entwickler, der Ihre Klassen verwendet.

Konstruktoren lassen sich übrigens genau wie Methoden überladen – nur beim gegenseitigen Aufrufen müssen bestimmte Dinge beachtet werden. Mehr dazu hält der Abschnitt »Überladen von Methoden, Konstruktoren und Eigenschaften« ab Seite 427 dieses Kapitels für Sie bereit.

Hat jede Klasse einen Konstruktor?

Oh ja. Zwar gab es ganz am Anfang dieses Kapitels eine Version, die weder über Eigenschaften noch über einen Standardkonstruktor im Quellcode verfügte, aber für einen solchen Fall verlangt es das CTS,⁶ dass der Compiler entsprechende Maßnamen ergreift, damit dieser Zustand eben nicht eintritt. Und das heißt was im Klartext?

Dazu ein Beispiel: Lassen Sie uns noch einmal zu der vorvorherigen Version des Beispielprogramms zurückkehren, in der es erst einen Konstruktor gab – die V3-Version unseres Beispiels. Diesen Konstruktor werden wir dann anschließend auskommentieren, und eine der Member-Variablen mit einem Standardwert vorinitialisieren, also in etwa wie folgt:

```
Public Class Kontakt

    'Sub New()
    '    myErstellungsdatum = Date.Now
    'End Sub

    Private myVorname As String
    Private myNachname As String
    Private myPLZ As String
    Private myOrt As String

    Private myErstellungsdatum As Date = Date.Now

```

⁶ Wir erinnern uns an Kapitel 3: Das Common Type System ist ein System, das strikte Typbindung und Codekonsistenz erfordert und damit die Common Language Runtime (CLR) zur Coderobustheit zwingt.

```
Public ReadOnly Property ErstellungsDatum() As Date
    Get
        Return myErstellungsdatum
    End Get
End Property
```

Fassen wir zusammen: In dieser Version ist nun der Standardkonstruktor auskommentiert und damit nicht mehr vorhanden; gleichzeitig gibt es eine Zuweisung der aktuellen Zeit/Datum-Kombination an die entscheidende Member-Variable `myErstellungsdatum` – was im Übrigen auch statthaft ist, denn der Compiler meckert ja nicht. Die Frage: Wann und durch was wird diese Initialisierung eigentlich ausgeführt?

Wie in Kapitel 3 schon kurz angerissen, übersetzt der Compiler Ihr Programm nicht direkt in nativen Maschinencode, sondern in eine »Zwischensprache« namens *Intermediate Language* oder kurz *IL*. Die Rahmendaten Ihres Programms, beispielsweise Konstanten, definierte Attribute, aber auch Informationen über die Version werden in einem speziellen Datenbereich in der gleichen Datei abgelegt. Diese Daten nennt man in *.NET Metadaten*. Wenn Sie nun ein Programm starten, dann gibt es zu diesem Zeitpunkt natürlich noch nichts, was der Prozessor ausführen kann, denn er versteht ja – was Intel-Plattformen anbelangt – nur Pentium- bzw. x86-Code. Es muss also einen Mechanismus geben, der zwischen den Zeitpunkten von Programmstart und Programmausführung Ihr Programm in Maschinencode übersetzt – und dieses Werkzeug ist Bestandteil der CLR und nennt sich *JITter*.⁷ Den »vorläufigen« IL-Code können Sie sich mit einem speziellen Werkzeug aus der .NET-Werkzeugsammlung anschauen – er offenbart alle Wahrheiten, auch solche, die der Compiler ohne Ihr Zutun hinzugefügt hat.

Mit diesem Wissen bewaffnet, können Sie sich jedes kompilierte Visual Basic-Programm in seinem »IL-Zustand« anschauen und herausfinden, was der VB-Compiler daraus gemacht hat. Dabei sind gar nicht so sehr die einzelnen Befehle entscheidend, sondern die Metadaten, die auch Auskunft darüber geben, aus welchen Komponenten, Typen, Signaturen, etc. sich Ihr Programm zusammensetzt.

Zusätzliche Werkzeuge für .NET

Es gibt einige zusätzliche und nützliche Tools zu Visual Studio 2008, bei denen es Microsoft schafft, sie trotz standardmäßiger Installation im Rahmen von Visual Studio 2008 einigermaßen sicher vor Ihnen zu verstecken. Diese befinden sich nämlich in einem besonderen Programmgruppen-Abschnitt des Start-Menüs, der impliziert, dass sie Bestandteil eines SDKs der Versionsnummer 6.0A sind. Hallo Microsoft-Versionsdurcheinander. SDK ist die Abkürzung von Software Development Kit, und bislang war es so, dass es alle Tools beinhaltete, mit denen Sie auch ohne Visual Studio Programme kompilieren, austesten, Resourcen anpassen und vieles mehr können.

Bei dem standardmäßig mitinstallierten »SDK« scheint es sich aber lediglich um ein Excerpt zu handeln, denn das aktuelle SDK hat einen Download-Umfang als ISO-Image von nicht weniger als 1,3 GByte – und Sie finden es im Übrigen unter dem IntelliLink **C1401**, mal ganz abgesehen davon, dass es dort Windows SDK 2008 und nicht 6.0 heißt, um das Versionschaos perfekt zu machen. Wir fassen also zusammen: Zum Framework 3.5 installiert Visual Studio 2008 ein SDK in der Programmgruppe SDK 6.0a, und Microsoft stellt ein SDK-Download zur Verfügung, der das Herunterladen des Windows SDK 2008 ermöglicht. Nun gut. ►

⁷ Zur Auffrischung: JIT ist die Abkürzung von »Just in time«, auf deutsch etwa »genau rechtzeitig«.

Diese zusätzlichen Tools finden Sie auf jeden Fall in der genannten 6.0a-Programmgruppe, und im Folgenden werden wir uns für ein bestimmtes namens IL-Disassembler interessieren, mit dem wir uns den CIL bzw. MSIL-Code sowie die Metadaten anschauen können, aus denen Ihre Assemblies bestehen, bevor der JITter ausführbaren Code daraus macht.

Nach diesem kurzen Exkurs in die Versionshölle des SDKs lassen Sie uns zurück zu unserem Vorhaben kommen:

- Wählen Sie *Projektmappe neu erstellen* aus dem Menü *Erstellen*, damit der Visual Basic Compiler den IL-Code innerhalb der Exe-Datei (also unserer Ausgabe-Assembly) neu erstellt.
- Starten Sie anschließend den Intermediate-Language-Disassembler (siehe vorheriger grauer Kasten) aus der Programmgruppe Windows SDK 6.0A.
- Wählen Sie *Öffnen* aus dem Menü *Datei* und im Dialog, der jetzt erscheint, die .EXE-Datei des Beispielprogramms. Sie finden die Programmdatei *MiniAdressoClass.exe* im Projektverzeichnis und dort im Unterverzeichnis *\bin\Debug*.

HINWEIS Sie werden im Debug-Verzeichnis übrigens zwei .EXE-Dateien finden – eine davon endet mit *vshost.exe*. Diese verwenden Sie bitte *nicht*. Bei ihr handelt es sich um ein kleines Hilfsprogramm, das von der Visual Studio IDE benötigt wird, um einerseits die Erstellzeiten Ihrer Projekte zu beschleunigen, und um das vielfach gewünschte *Edit & Continue* zu ermöglichen, das es Ihnen gestattet, Änderungen an Ihrem Programm vorzunehmen, während Sie es debuggen. *Edit & Continue* steht Ihnen in dieser Version von Visual Studio übrigens nicht zur Verfügung, wenn dieses unter einem 64-Bit-Windows-Betriebssystem läuft.

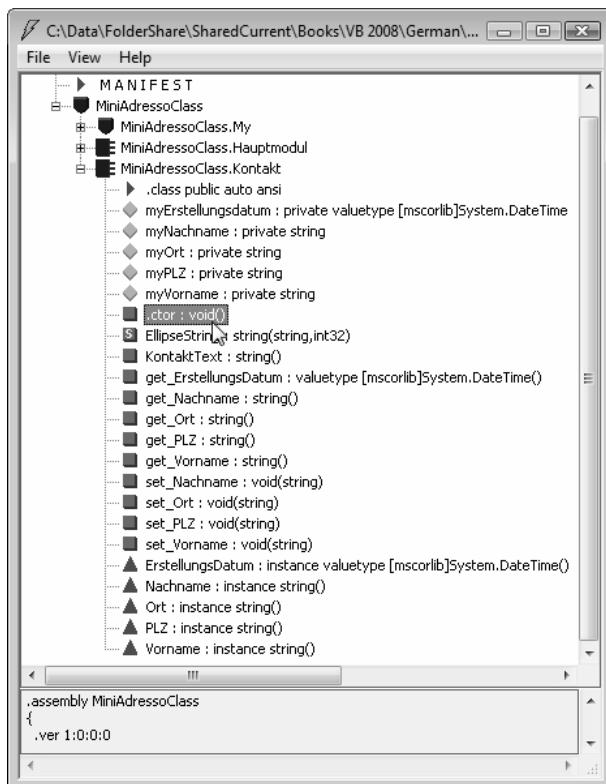


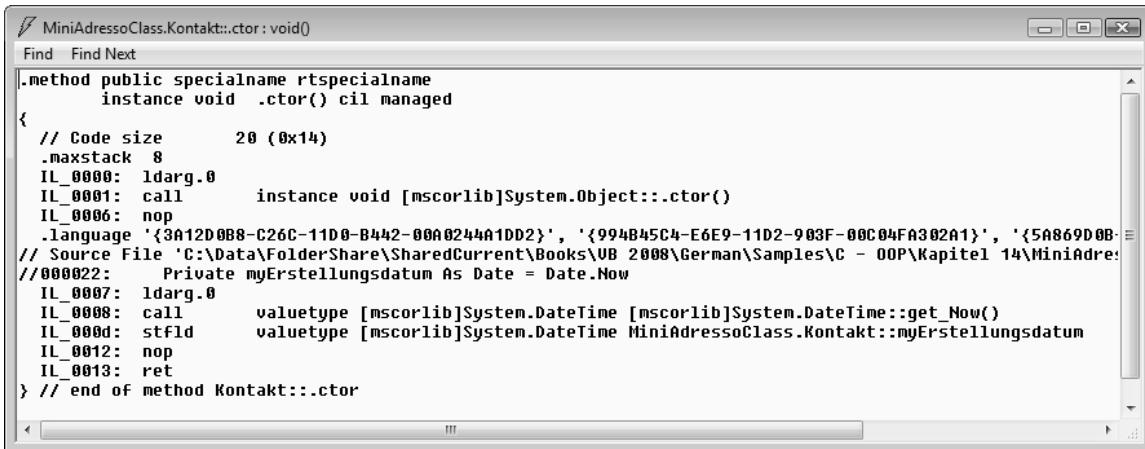
Abbildung 14.4 Die Metadaten der EXE-Datei erlauben die Ansicht der Programmstruktur im IL-Disassembler

- Im Fenster, das anschließend erscheint, öffnen Sie den Zweig *MiniAdressoClass* und den sich darunter befindlichen Zweig *MiniAdressoClass.Kontakt* ebenfalls – denn darin befinden sich alle Elemente, aus denen unsere Kontakt-Klasse besteht.
- Aus dem Menü *Ansicht* wählen Sie die Option *Quellcodezeilen anzeigen*. Wenn wir dann gleich den CIL-Code betrachten, werden die entsprechenden Visual Basic-Quellcodezeilen mit dort eingeblendet. Sie sollten anschließend ein Bild vor Augen haben, das etwa dem in Abbildung 14.4 entspricht.

In dieser Abbildung sehen Sie die Struktur des Programms. Sie finden Elemente wieder, die Sie selber bestimmt haben – beispielsweise die Funktionsnamen. Und Sie erkennen ebenfalls eine Methode, für die Sie nicht der unmittelbare Urheber waren – die Methode `.ctor`.

Und das liegt daran, dass diese Methode keine Methode im herkömmlichen Sinne ist, sondern vielmehr handelt es sich bei ihr um einen Konstruktor, um den Part der Klasse also, der entsteht, wenn Sie in Visual Basic eine Sub `New` implementieren würden. Genau die haben Sie aber im Quellcode auskommentiert, bevor Sie die Anwendung neu erstellt und IL-Disassembler gestartet haben.

Der Compiler hat also selbstständig dafür gesorgt, dass ein Konstruktor in der Klasse vorhanden ist, und genau das schreibt das CTS auch vor: Jede instanzierbare Klasse muss einen Konstruktor haben. Ob der dann aufrufbar ist, steht noch mal auf einem anderen Papier geschrieben; Sie können beispielsweise den Konstruktor der Klasse mit einem `Private`-Zugriffsmodifizierer ausstatten, sodass die Klasse nur noch durch sich selbst instanziert werden könnte, aber eben nicht mehr von außen.⁸



```

// MiniAdressoClass.Kontakt::ctor : void()
Find Find Next
.method public specialname rtspecialname
    instance void  .ctor() cil managed
{
    // Code size     20 (0x14)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call      instance void [mscorlib]System.Object::ctor()
    IL_0006:  nop
    .language '{3A12D0B8-C26C-11D0-B442-00A0244A1DD2}', '{994B45C4-E6E9-11D2-903F-00C04FA302A1}', '{5A869D0B-...
// Source File 'C:\Data\FolderShare\SharedCurrent\Books\VB 2008\German\Samples\C - OOP\Kapitel 14\MiniAdre...
//000022:  Private myErstellungsdatum As Date = Date.Now
    IL_0007:  ldarg.0
    IL_0008:  call      valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
    IL_000d:  stfld    valuetype [mscorlib]System.DateTime MiniAdressoClass.Kontakt::myErstellungsdatum
    IL_0012:  nop
    IL_0013:  ret
} // end of method Kontakt::ctor

```

Abbildung 14.5 Der disassemblierte IML-Code des Standardkonstruktors

Auch ohne genau zu wissen, welche IL-Anweisung für welche Aufgabe zuständig ist, wird eines doch sofort deutlich: Im ersten Teil der Methode wird offensichtlich ein weiterer Konstruktor aufgerufen, und das ist richtig: Da jede Klasse, die Sie neu erstellen, implizit vom Grundtypen der obersten Hierarchieebene `Object`

⁸ Ergibt keinen Sinn, meinen Sie? Au contraire! Es gibt ein sogenanntes Entwicklungs-Pattern, eine bestimmte Art, eine Klasse zu konzipieren, die sich Singleton nennt. Dieses Pattern hat lediglich einen privaten Konstruktor, aber dafür eine öffentliche statische Methode, über die Sie eine Instanz anfordern können. Ein Beispiel für eine Singleton-Klasse finden Sie am Ende des nächsten Kapitels.

abgeleitet wird, wenn Sie es nicht anders bestimmen, wird hier der Konstruktor dieser Basisklasse aufgerufen.⁹ Und anhand der eingebetteten Quellzeile können wir hier auch sicher sein, dass die ursprünglich nur unterhalb des Klassenrumpfs stehende Variablen-deklaration/-definition des Members `myErstellungsdatum` ihren Weg in den Konstruktor gefunden hat – denn dort wird sie nämlich mit der aktuellen Uhrzeit/dem aktuellen Datum ausgestattet.

Dieses Beispiel zeigt also, dass Ihnen der Visual Basic-Compiler eine ganze Menge an Arbeit abgenommen hat. Er hat dafür gesorgt,

- dass es einen Standardkonstruktor mit der Methode `.ctor` überhaupt gibt, auch wenn Sie nicht explizit mit `Sub New()` einen implementiert haben.
- dass innerhalb des Standardkonstruktors der Standardkonstruktor Basisklasse `Object` (`call instance void [mscorlib]System.Object::ctor()`) aufgerufen wird und
- dass alle Member-Variablen, falls erforderlich, innerhalb dieses Standardkonstruktors definiert – also mit den Werten »gefüllt« werden, so wie dies bei deren Deklarierung unterhalb des Klassenrumpfs angegeben haben.

Klassenmethoden mit Sub und Function

Klassencode wird nicht nur in Eigenschaftenprozeduren ausgeführt. Auch Methoden, die in Visual Basic durch `Sub` oder `Function` definiert werden, gehören dazu. Methoden nicht statischer Natur (solche also, die nicht, wie in »Statische Methoden« ab Seite 438 beschrieben über die Klasseninstanz definiert sind) sind dabei Methoden, die mit den Member-Variablen einer Klasse arbeiten sollten.

Ein Beispiel dafür ist das folgende: Eine Funktion wie beispielsweise `KontaktText` könnte für das Zurückliefern der textrepräsentativen Ausgabe der Klasse sorgen – und die entsprechenden Routinen dafür würden dann folgendermaßen aussehen:

```
Public Function KontaktText() As String

    Return Me.Nachname & ", " & Me.Vorname & ", " & _
           Me.PLZ & ", " & Me.Ort

End Function
```

Diese Member-Methode der Klasse bedient sich direkt der Instanzvariablen, und gibt sie verkettet als einen zusammenhängenden String aus. Das bedeutet, dass wir entsprechende Änderungen auch bei der Ausgabekoutine im Modul machen können, die die einzelnen Instanzen verwendet:

```
Sub AdressenAusgeben(ByVal von As Integer, ByVal bis As Integer)

    For c As Integer = von To bis
        Console.WriteLine(c.ToString("000") & ":" & Kontakte(c).KontaktText())
```

⁹ Was *Ableiten von Klassen* genau bedeutet, und wieso genau hier der Konstruktor der Basisklasse aufgerufen wird, erfahren Sie im nächsten Kapitel.

Next

End Sub

Die Methode KontaktText wird hier für jede auszugebende Kontakt-Instanz aufgerufen und liefert natürlich auch für jede Instanz ein unterschiedliches Ergebnis – hieran sieht man deutlich, dass es sich um eine Member-Methode handelt.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 14\\MiniAdressoClass V5

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Übrigens: Eine Anwendung startet immer mit einer Sub Main – das gilt auch für Windows-Anwendungen, die Sie als Windows Forms-Projekte unter Visual Basic .NET erstellen. Warum das so ist, lesen Sie in Kapitel 2.

Überladen von Methoden, Konstruktoren und Eigenschaften

Falls Sie zu den alten VB6-Hasen gehören, dann kennen Sie sicherlich den Vorteil von optionalen Parametern. Das Überladen von Funktionen in .NET ist ein nur auf den ersten Blick ähnliches aber letzten Endes dennoch völlig anderes Konzept. Gemeinsam haben beide Konzepte, dass sie eine Liberalisierung von Parameterübergaben an Funktionen ermöglichen. Das war es aber dann auch schon mit den Gemeinsamkeiten.

Mit dem Überladen von Funktionen geben Sie Ihren Klassen eine enorme Flexibilität und Anpassungsgabe. Überladen von Funktionen bedeutet: Sie erstellen verschiedene Funktionen mit gleichen Namen, die sich nur durch den Typ, die Typeihenfolge oder die Anzahl der übergebenden Parameter unterscheiden. Als Beispiel schauen Sie sich bitte den folgenden Codeausschnitt an:

```
Sub EineProzedur()
    'Tu was.
End Sub

Sub EineProzedur(ByVal ein_Parameter As Integer)
    'Tu was anderes.
End Sub

Sub EineProzedur(ByVal ein_anderer_Parameter As String)
    'Tu was anderes.
End Sub

Sub EineProzedur(ByVal ein_Parameter As Integer, ByVal ein_anderer_Parameter As String)
    'Tu was anderes.
End Sub
```

```
Sub EineProzedur(ByVal ein_ganz_anerer_Parameter As Integer)
    'Fehler: ein Integer als Parameter gab's schon mal.
    ''Die Methode 'EineProzedur' wurde mehrfach mit identischen Signaturen definiert.
End Sub
```

Es ist, als würde die Signatur – so nennt man die Mischung aus Parametertypen und Parameterreihenfolge beim Aufrufen einer Funktion – Bestandteil des Namens werden, und daran wird dann erkennbar, welche der vorhandenen EineProzedur aufgerufen werden soll. Der Variablenname hat damit übrigens überhaupt nichts zu tun – nur der übergebene Typ ist für die Identifizierung der Signatur entscheidend.

Aus diesem Grund bereitet die letzte Sub des Beispiels auch Probleme. Ihr wird genau wie der ersten eine Variable vom Typ Integer übergeben. Zwar ist der Variablenname ein anderer, aber darauf kommt es überhaupt nicht an – Namen sind hier tatsächlich nicht mehr als Schall und Rauch.

Wozu eignet sich die Funktionsüberladung in der Praxis? Nun, die Anwendung von Klassen wird dadurch ungleich flexibler. Schon bei unserem ständigen Klassenbeispielbegleiter kommen Sie spätestens beim Anwenden der Klasse Kontakt in den Genuss des Komforts von überladenen Methoden bzw. Konstruktoren, für die das Gesagte gleichermaßen gilt. Sie müssen nicht wissen, welche Methode beispielsweise für welche Teilaufgabe zuständig ist; sie können einfach die (eine) Funktion verwenden, und IntelliSense¹⁰ unterstützt Sie bei der Auswahl der richtigen Signatur sogar. Wenn Sie die Zeile in Ihr Programm eingeben, die die Klasse in ein Objekt instanziert, dann zeigt Ihnen IntelliSense, nachdem Sie die geöffnete Klammer hinter New eingegeben haben, Ihre Optionen an, etwa wie in der folgenden Grafik zu sehen:

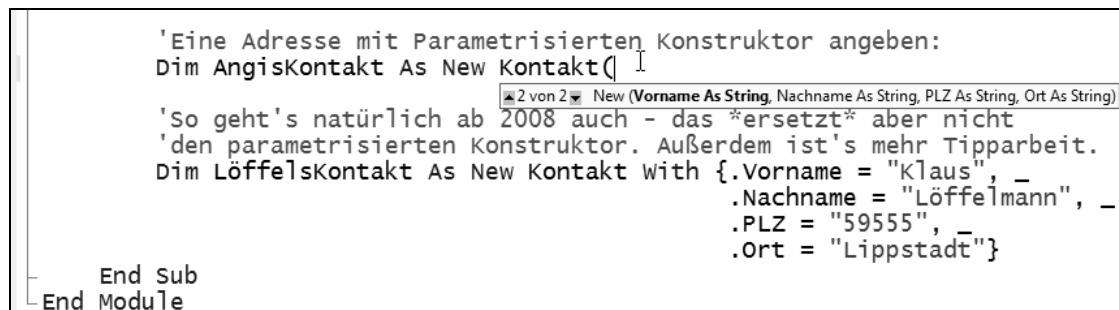


Abbildung 14.6 IntelliSense hilft Ihnen bei der Auswahl der richtigen Signatur bei überladenen Methoden wie Konstruktoren

Im Gegensatz zu optionalen Parametern können Sie Methoden bzw. Konstruktoren implementieren, deren Versionen völlig unterschiedliche Dinge tun. Natürlich haben beide Konstruktorversionen des Beispiels thematisch miteinander zu tun (sollten sie auch, anderenfalls sollten Sie ihnen komplett andere Namen geben). Wichtig ist: Die Funktionsweise der beiden überladenen Prozeduren kann im Gegensatz zu *einer* Prozedur mit optionalen Parametern nicht nur komplett anders implementiert werden (die erste initialisiert die Klasse mit einem Integer, die andere durch die Angabe eines römischen Numerales), die beiden Methoden sind auch visuell sauber voneinander getrennt.

¹⁰ Zur Wiederholung: So nennt sich die Eingabehilfe des Codeeditors, mit deren Hilfe sich Elementnamen vervollständigen, alle Elemente eines Objektes in Listen anzeigen oder Funktions- und Eigenschaftenüberladungen sich schon bei der Codeeingabe sichtbar machen lassen. Abbildung 14.6 zeigt IntelliSense in Aktion.

HINWEIS Das Prinzip der Funktionsüberladung funktioniert für Subs genauso wie für Functions. Allerdings ist Folgendes bei der Verwendung von Funktionen wichtig zu wissen: Der Rückgabetyp kann nicht als Differenzierungskriterium von Signaturen verwendet werden (wobei auch bei einer Funktion der Rückgabetype nicht das einzige Differenzierungskriterium sein kann). Das heißt im Klartext:

```
Function EineFunktion() As Integer
    'Tu was.
End Function

Function EineFunktion(ByVal AndereSignatur As Integer) As Integer
    'Das funktioniert.
End Function

Function EineFunktion() As String
    '"Public Function EineFunktion() As Integer" und "Public Function EineFunktion() As String"
    'können sich nicht gegenseitig überladen, da sie sich nur durch Rückgabetypen unterscheiden.
End Function
```

Die ersten beiden Funktionen sind o.k., da sich die Signaturen voneinander unterscheiden. Die letzte Funktion unterscheidet sich von der ersten allerdings nur durch den Rückgabetyp, und aus diesem Grund meldet der Visual Basic-Compiler schon zur Entwurfszeit einen Fehler.

TIPP Sie können – zur besseren Lesbarkeit – das Overloads-Schlüsselwort verwenden, um die Überladung einer Methode deutlich zu machen:

```
Overloads Function EineFunktion() As Integer
    'Wenn Overloads verwenden, ...
End Function

Overloads Function EineFunktion(ByVal AndereSignatur As Integer) As Integer
    '...dann bei den Funktionen
End Function
```

Dabei sollten Sie berücksichtigen: Wenn Sie sich für das Overloads-Schlüsselwort entscheiden, müssen Sie es bei *allen* Methodenvariationen mit Überladungen verwenden.

Methodenüberladung und optionale Parameter

Auch in der .NET-Version bietet Ihnen Visual Basic noch das Hilfsmittel der optionalen Parameter an. Optionale Parameter haben gegenüber überladenen Methoden entscheidende Nachteile:¹¹

¹¹ Bei der ganzen Negativliste wollen wir aber einen entscheidenden Vorteil auch nicht vergessen: Die Entwicklung von Office-Anwendungen mit VSTO ist in Visual Basic .NET sicherlich angenehmer, eben *weil* die Office-Libraries optionale Parameter verwenden, die in VB.NET dann anwendbar sind. In C# müssen Sie eine Methode aufrufen, und für jeden nicht angegebenen Parameter ein entsprechendes Auslassungsschlüsselwort übergeben (`System.Reflection.Missing.Value`).

- Sie werden von vielen anderen .NET-Programmiersprachen nicht unterstützt. Wenn Sie in Ihren Klassen optionale Parameter verwenden, haben ausschließlich Visual Basic-Entwickler etwas davon. Weder C# noch J# noch Managed C++¹² unterstützen optionale Parameter.
- Die Verwendung von optionalen Parametern macht Ihren Code schwerer lesbar und kaum wiederverwendbar. Wenn Sie optionale Parameter verwenden, müssen Sie Fallunterscheidungen durchführen, indem Sie durch die Abfrage von Standardwerten herausfinden, welche Parameter vom Aufrufer übergeben wurden und welche nicht. Eine Methode, die nur aufgrund ihrer Parameter vielleicht zwei völlig verschiedene Dinge macht, trägt dann quasi den gequetschten »Doppelcode« in einem Funktionsrumpf. Mit überladenen Funktionen hingegen haben Sie zwei Problemlösungen auch optisch sauber voneinander getrennt.
- Sollten Sie ohne strikte Typbindung arbeiten (Option Strict Off) und gleichzeitig überladene Funktionen und optionale Parameter für die gleichen Methoden verwenden, bedeutet das einen unglaublichen Leistungsverzicht, da die Laufzeitbibliothek von Visual Basic unter Umständen erst herausfinden muss, welche der Routinen am ehesten zum angegebenen Parameter passt. Bei sehr flexiblen Parameterübergaben können das obendrein potenzielle Fehlerquellen werden, die ich – ganz ehrlich gesagt – bei Fehlverhalten niemals debuggen möchte.

Aus diesen Gründen gehe ich auf die Eigenschaft, Parameter optional an Funktionen zu übergeben, auch nicht näher ein. Meine Empfehlung: Falls Sie optionale Parameter in früheren Visual Basic-Versionen kennen gelernt haben, versuchen Sie sie am besten nicht mehr zu verwenden. Falls Sie das Konzept der optionalen Parameterübergabe gar nicht kennen: umso besser!

Gegenseitiges Aufrufen von überladenen Methoden

Das Überladen von Funktionen begegnet Ihnen im Framework an jeder Ecke. In der Regel wird die Funktionsüberladung vom Framework verwendet, um dem Anwender die Handhabung so angenehm wie möglich zu machen. So werden ihm Signaturen für bestimmte Funktionen mit nur sehr wenigen Parametern angeboten, um Tipparbeit sparen, und gleichzeitig andere Versionen derselben Funktion mit sehr viel mehr Parametern für die größtmögliche Flexibilität.

Die `WriteLine`-Methode, die Sie ja schon mehrfach im Einsatz bewundert haben, ist hier ein sehr anschauliches Beispiel, da sie mit nicht weniger als 18 Überladungen daherkommt. Wenn Sie im Codeeditor von Visual Basic die Anweisung `Console.WriteLine` eingeben und anschließend die geöffnete Klammer eintippen, zeigt Ihnen IntelliSense die Signaturen der 18 verschiedenen Überladungen an.

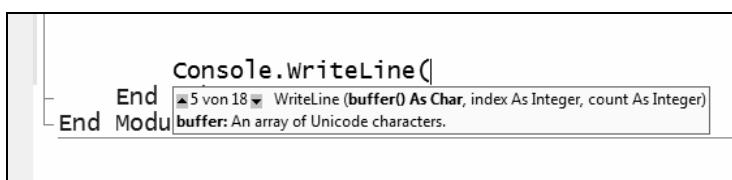


Abbildung 14.7 Die `WriteLine`-Methode hat nicht weniger als 18 Überladungen vorzuweisen!

¹² Managed C++ nennt sich die .NET-Version von C++, die wie alle anderen .NET-Programmiersprachen verwalteten (Managed) Code erzeugt. C++ ist übrigens die einzige Sprache im Visual Studio Paket, mit der Sie auch noch Anwendungen entwickeln können, die nicht auf Managed Code basieren.

Natürlich wäre das Überladen von Methoden keine wirkliche Arbeitserleichterung, wenn Entwickler die eigentliche Funktionalität für alle überladenen Methoden immer wieder implementieren müssten. Überladene Methoden können sich deswegen gegenseitig – vom Sonderfall Sub New einmal abgesehen – ohne Einschränkungen aufrufen.

Der übliche Weg, den Anwendern Ihrer Klassen (und damit meistens sich selbst) große Flexibilität in die Hand zu geben ist, eine universale Methode zu entwickeln, die alles kann und sie anschließend durch Überladungen »nach unten abzuspecken«.

Ein Beispiel: Angenommen, Sie haben eine Klasse entwickelt, die eine Methode zur Verfügung stellt, mit der man Kreise auf den Bildschirm malen kann. Sie nennen diese Methode *Circle*, und diese stellt in ihrer Universalversion folgende Fähigkeiten zur Verfügung:

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
                  ByVal YRadius As Integer, ByVal StartAngle As Integer, ByVal EndAngle As Integer)
    'Hier steht der Code für Circle.
End Sub
```

Nun benötigen Sie die komplette Flexibilität dieser Methode aber nur in den seltensten Fällen. Sie müssen nicht jedes Mal X- *und* Y-Radius der Figur und noch seltener den Start- und Endwinkel des Kreises mit angeben. Der Aufruf einer solchen vereinfachten Version ist einfach praktischer, daher nennt man solche Methoden auch Convenience-Methoden. Eine abgespeckte Version könnte daher wie folgt aussehen:

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal Radius As Integer)
    Circle(Xpos, YPos, Radius, Radius, 0, 359)
End Sub
```

Auf den ersten Blick sieht es so aus, als würde sich die Funktion selber aufrufen. Macht sie aber nicht. Da die Signatur der verwendeten *Circle*-Methode (2. Zeile) nicht der eigenen Signatur (1. Zeile) entspricht, schaut der VB-Compiler, welche *Circle*-Methode in Frage kommt und verwendet in diesem Beispiel die zuerst verwendete.

TIPP Aus Geschwindigkeitsgründen sollten Sie davon absehen, dass überladene Methoden quasi treppchenweise die jeweils nächst flexiblere Methode aufrufen, wenn Sie im Vorfeld wissen, dass solche Methoden beispielsweise in Schleifen hunderte von Malen im Laufe eines Programmlebens aufgerufen werden. Implementieren Sie eine universale Methode, die alles kann, und rufen Sie sie von jeder weiteren Version der Methode direkt auf – das spart Ausführungszeit in solchen Fällen.

Schlechtes Beispiel:

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal Radius As Integer)
    'Nicht so gut, wir "stolpern" quasi zum Ziel.
    Circle(Xpos, YPos, Radius, Radius)
End Sub

Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
                  ByVal YRadius As Integer)
    Circle(Xpos, YPos, XRadius, YRadius, 0, 359)
End Sub
```

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
ByVal YRadius As Integer, ByVal StartAngle As Integer, ByVal EndAngle As Integer)
    'Hier steht der Code für Circle.
End Sub
```

Gutes Beispiel:

```
Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal Radius As Integer)
    'Besser: direkter Sprung zur eigentlichen Methode.
    Circle(Xpos, YPos, Radius, Radius, 0, 359)
End Sub

Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
ByVal YRadius As Integer)
    Circle(Xpos, YPos, XRadius, YRadius, 0, 359)
End Sub

Public Sub Circle(ByVal Xpos As Integer, ByVal YPos As Integer, ByVal XRadius As Integer, _
ByVal YRadius As Integer, ByVal StartAngle As Integer, ByVal EndAngle As Integer)
    'Hier steht der Code für Circle.
End Sub
```

Gegenseitiges Aufrufen von überladenen Konstruktoren

Ein wenig problematischer wird es, wenn sich überladene Konstruktoren gegenseitig aufrufen sollen – betriebsbedingt gibt es dabei nämlich Einschränkungen. Mit dem Wissen des vorherigen Abschnittes starten Sie vielleicht rein aus dem Gefühl heraus den Versuch, das Beispiel auch bei den Konstruktoren folgendermaßen umzugestalten:

```
'Parameterloser Konstruktor
Sub New()
    myErstellungsdatum = Date.Now
End Sub

'Konstruktor übernimmt Initialisierungsparameter
Sub New(ByVal Vorname As String, ByVal Nachname As String, _
        ByVal PLZ As String, ByVal Ort As String)

    'Member-Variablen mit den übergebenen Werten initialisieren.
    myVorname = Vorname
    myNachname = Nachname
    myPLZ = PLZ
    myOrt = Ort

    'Und ruft - und das geht NUR als erstes oder gar nicht -
    'Ein Aufruf an einen Konstruktor ist nur als erste Anweisung in einem Instanzenkonstruktor gültig.
    Me.New()
End Sub
```

Abbildung 14.8 Wenn Sie versuchen, eine andere Konstruktor-Methode nicht an erster Stelle eines Konstruktors aufzurufen, quittiert Ihnen Visual Basic das mit einem Fehler

Visual Basic quittiert diesen Aufruf mit einem simplen Syntaxfehler¹³ und lässt den Aufruf ganz einfach nicht zu.

Übrigens ist es an dieser Stelle schon mal wichtig zu wissen, dass Sie genau spezifizieren, welches New Sie aufrufen. Prinzipiell gibt es bei jeder Klasse nämlich zwei Versionen von New, die aber nicht durch Überladen entstanden sind: Das New, also der Konstruktor in Ihrer Klasse und das New der Klasse, von der Sie Ihre Klasse abgeleitet haben. Da Sie hier im Beispiel nicht explizit bestimmt haben, aus welcher Klasse Sie ableiten, hat Ihre Klasse automatisch von Object geerbt. Natürlich hat auch Object einen Konstruktor, und Sie können sowohl Methoden der Basisklasse als auch Methoden Ihrer Klasse aufrufen. Wenn Sie Me angeben, spezifizieren Sie Ihre Klasse; wenn Sie MyBase angeben, würden Sie damit die Basisklasse spezifizieren – in diesem Fall die Klasse Object. Mehr über das Vererben und über Aufrufe von Funktionen von Basisklassen erfahren Sie im nächsten Kapitel.

Überladen von Eigenschaften-Prozeduren mit Parametern

Eigenschaften, die Parameter entgegen nehmen, lassen sich wie Funktionen überladen. Zum Einsatz kommt dieses Prinzip fast ausschließlich bei Default-Eigenschaften (siehe »Default-Eigenschaften (Standardeigenschaften)« auf Seite 413). Wie bei »normalen« Funktionen kann nur die Eigenschaftensignatur (die Reihenfolge, die Typen und die Anzahl der übergebenden Parameter) nicht aber der Rückgabetyp zur Unterscheidung herhalten. Daher ist die Überladung von Eigenschaften auch nur dann möglich, wenn mindestens eine Eigenschaftenvariation Parameter entgegennimmt.

Ein Beispiel für das Überladen von Eigenschaften:

```
Property Überladung() As Integer
    Get
        'Hier der Code für die Ermittlung der Eigenschaft.
    End Get
    Set(ByVal Value As Integer)
        'Hier der Code für die Zuweisung.
    End Set
End Property

Property Überladung(ByVal Par1 As Integer) As Integer
    Get
        'Hier der Code für die Ermittlung der Eigenschaft.
    End Get
    Set(ByVal Value As Integer)
        'Hier der Code für die Zuweisung.
    End Set
End Property

Property Überladung() As String
    Get
        'Geht nicht, da sich diese Eigenschaft...
    End Get
```

¹³ Kleine nostalgische Randbemerkung: Das ist eine Fehlermeldung, die es schon beim Commodore 64 gab – der verfügte auch über ein Microsoft-Basic – und die sich bis heute gehalten hat!

```

Set(ByVal Value As String)
    'nur durch den Rückgabetyp von der ersten unterscheidet.
End Set
End Property

```

HINWEIS Für das Überladen von Eigenschaften benötigen Sie mindestens eine Eigenschaftenprozedur, die Parameter entgegen nimmt. Denken Sie an das bereits gesagte, dass das – von Standardeigenschaften mal abgesehen – eine Visual Basic-spezifische Vorgehensweise ist, und andere Sprachen wie C# möglicherweise daran hindern würden, von Eigenschaften Gebrauch zu machen, die Parameter entgegen nehmen. C# beispielsweise kennt nämlich außer bei Standardeigenschaften keine Eigenschaftenprozeduren, die Parameter entgegen nehmen.

Zugriffsmodifizierer für Klassen, Methoden und Eigenschaften – Gültigkeitsbereiche definieren

Zugriffsmodifizierer sind Schlüsselworte, mit denen Sie in Visual Basic bestimmen können, von wo aus der Zugriff auf ein Element gestattet ist und von wo aus nicht. Die Zugriffsmodifizierer `Private` und `Public` haben Sie schon kennen gelernt. Sie bestimmen, ob auf ein Element nur innerhalb eines bestimmten Gültigkeitsbereiches zugegriffen werden kann (`Private`) oder von überall aus (`Public`). Welche weiteren Zugriffsmodifizierer es für Objekte, Klassen und Funktionen/Eigenschaften gibt, zeigen die folgenden Tabellen:

Zugriffsmodifizierer bei Klassen

HINWEIS Wenn nichts anderes gesagt wird, werden Klassen standardmäßig als `Friend` deklariert.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Private	Private	<p>Als <code>Private</code> können Klassen nur dann definiert werden, wenn sie geschachtelt in einer anderen Klasse definiert sind. Beispiel:</p> <pre> Public Class A Private Class B End Class End Class Public Class C 'Zugriff verweigert, Class B ist Private! Dim b as A.B End Class </pre>
Public	Public	Sie können auf die Klasse uneingeschränkt von außen zugreifen, auch aus anderen Assemblys heraus.
Friend	Assembly	Sie können innerhalb der Assembly auf die Klasse zugreifen, aber nicht aus einer anderen Assembly heraus. ►

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Protected	Family	Es gilt das für Private Gesagte. Zusätzlich gilt: Auch aus der Klasse abgeleitete Klassen können auf die mit Protected gekennzeichneten und geschachtelten »inneren« Klassen zugreifen.
Protected Friend	FamilyOrAssembly	Der Zugriff auf die geschachtelte Klasse ist in abgeleiteten Klassen und von Klassen der gleichen Assembly aus möglich.

Tabelle 14.1 Mögliche Zugriffsmodifizierer für Klassen in Visual Basic .NET

Zugriffsmodifizierer bei Prozeduren (Subs, Functions, Properties)

HINWEIS Wenn nichts anderes gesagt wird, werden Subs, Functions und Properties standardmäßig als Public deklariert. Sie sollten gerade bei diesen Elementen aber auf jeden Fall einen Zugriffsmodifizierer verwenden, damit beim Blättern durch den Quellcode schnell deutlich wird, welchen Zugriffsmodus ein Element innehat.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Private	Private	Nur innerhalb einer Klasse kann auf die Prozedur zugegriffen werden.
Public	Public	Sie können auf die Prozedur uneingeschränkt von außen zugreifen, auch aus anderen Assemblies heraus.
Friend	Assembly	Sie können innerhalb der Assembly auf die Prozedur zugreifen, aber nicht aus einer anderen Assembly heraus.
Protected	Family	Nur innerhalb der Klasse oder einer abgeleiteten Klasse kann auf die Prozedur zugegriffen werden.
Protected Friend	FamilyOrAssembly	Nur innerhalb der Klasse, einer abgeleiteten Klasse oder innerhalb der Assembly kann auf die Prozedur zugegriffen werden.

Tabelle 14.2 Mögliche Zugriffsmodifizierer für Prozeduren in Visual Basic .NET

Zugriffsmodifizierer bei Variablen

Variablen, die auf Klassenebene nur mit Dim deklariert werden, gelten als Private, also nur von der Klasse aus zugreifbar. Variablen, die innerhalb eines Codeblocks oder auf Procedurebene deklariert werden, gelten nur für den entsprechenden Codeblock. Innerhalb eines Codeblocks können Sie nur die Dim-Anweisung und keine anderen Zugriffsmodifizierer verwenden. Auf Procedurebene können Sie eine Variable zusätzlich als static deklarieren. Mehr über den Static-Modifizierer erfahren Sie im Abschnitt »Statische Komponenten«.

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Private	Private	Nur innerhalb einer Klasse kann auf die Variable zugegriffen werden. Variablen innerhalb von Prozeduren oder noch kleineren Gültigkeitsbereichen können nicht explizit als Private definiert werden, sind es aber standardmäßig.
Public	Public	Von außen kann auf die Klassenvariable uneingeschränkt zugegriffen werden. Sie sollten Variablen aber bestenfalls als Protected deklarieren und sie nur durch Eigenschaften nach außen offen legen. Mehr zu diesem Thema erfahren Sie im Abschnitt »Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage?« auf Seite 412. ►

Zugriffsmodifizierer	CTS-Bezeichnung	Beschreibung
Friend	Assembly	Sie können innerhalb der Assembly auf die Klassenvariable zugreifen, aber nicht aus einer anderen Assembly heraus. Es gilt das für Public Gesagte.
Protected	Family	Nur innerhalb derselben oder einer abgeleiteten Klasse kann auf die Variable zugegriffen werden. Variablen sollten in Klassen, bei denen Sie davon ausgehen, dass sie später des Öfteren vererbt werden, als Protected definiert werden, damit abgeleitete Klassen ebenfalls darauf zugreifen können.
Protected Friend	FamilyOrAssembly	Nur innerhalb der Klasse, einer abgeleiteten Klasse oder innerhalb der Assembly kann auf die Klassenvariable zugegriffen werden. Von dieser Kombination sollten Sie absehen.
Static	- - -	Sonderfall in Visual Basic. Lesen Sie dazu bitte die Ausführungen im Abschnitt »Statische Komponenten«.

Tabelle 14.3 Mögliche Zugriffsmodifizierer für Variablen in Visual Basic .NET

Diese Tabellen sollen Ihnen kompakt und auf einen Blick die Zugriffsmodifizierer verdeutlichen. Die CTS-Bezeichnungen der Zugriffsmodifizierer benötigen Sie, wenn Sie sich den IML-Code einer Klasse anschauen, um zu erkennen, welchen Zugriffsmodus beispielsweise eine Methode hat.

Unterschiedliche Zugriffsmodifizierer für Eigenschaften-Accessors

Seit Visual Studio 2005 ist es möglich, dass die Get- und Set-Accessors unterschiedliche Zugriffsmodifizierer aufweisen. So haben Sie beispielsweise die Möglichkeit, zu bestimmen, dass zwar eine bestimmte Eigenschaft von jedem Ort aus gelesen werden kann (`Public`), aber Eigenschaften nur von derselben Klasse aus geschrieben werden dürfen (`Private`). Im Code würde eine solche Eigenschaft folgendermaßen ausschauen:

```
Module Module1

Sub Main()
    Dim locEigenschaftenTest As New EigenschaftenTest("Text für Eigenschaft")

    'Das Auslesen der Eigenschaft ist problemlos möglich
    Console.WriteLine("Eigenschaft enthält: " & locEigenschaftenTest.EineEigenschaft)

    'FEHLER: Der Set-Zugriffsmodifizierer verbietet aber das Schreiben,
    'weil er 'private' ist.
    locEigenschaftenTest.EineEigenschaft = "Neuer Text"
End Sub

End Module

Public Class EigenschaftenTest

    Private myEineEigenschaft As String

    Sub New(ByVal textFürEigenschaft As String)
        'Ist erlaubt - die Klasse darf die
        'Eigenschaft beschreiben!
        EineEigenschaft = textFürEigenschaft
    End Sub
End Class
```

```
Public Property EineEigenschaft() As String
    Get
        Return myEineEigenschaft
    End Get
    Private Set(ByVal value As String)
        myEineEigenschaft = value
    End Set
End Property

End Class
```

Dieses kleine Beispiel besteht aus zwei Einheiten – einem Modul und einer Klasse. Die Klasse `Eigen-schaftenTest` enthält einen parametrisierten Konstruktor sowie eine Eigenschaft, die aber Accessoren mit unterschiedlichen Zugriffsmodifizierern hat.

Innerhalb des Konstruktors (`Sub New`) ist der Schreibzugriff auf die Eigenschaft problemlos möglich, da aus der Klasse selbst heraus auch auf Elemente zugegriffen werden kann, deren Zugriff mit `Private` eingeschränkt wurde. Innerhalb des Moduls funktioniert der Zugriff allerdings nicht mehr, da sich das Programm zum Zeitpunkt des Zugriffs außerhalb der Klasse befindet, und wegen `Private` ist von diesem Punkt aus kein Herankommen an die Eigenschaft möglich.

Doch wozu brauchen Sie unterschiedliche Zugriffsmodifizierer in Eigenschaften während der Entwicklung Ihrer Software? Denken Sie beispielsweise an das zur Verfügung Stellen von Assemblies (Klassenbibliotheken) für andere Entwickler: Sie möchten beispielsweise in der Lage sein, bestimmte Eigenschaften Ihrer Klassen von jedem Punkt Ihrer Assembly aus zu manipulieren; Sie möchten aber gleichzeitig verhindern, dass ein Entwickler, der Ihre Assembly verwendet, dazu auch in der Lage ist. In diesem Fall definieren Sie den Get-Accessor als `Public` – das Lesen der Eigenschaft stellt schließlich kein Risiko dar und kann von überall aus erfolgen – aber den Set-Accessor als `Friend`. Vom gesamten Programmcode, der sich in Ihrer Klassenbibliothek befindet, können Sie dann die Eigenschaft der betreffenden Klasse manipulieren; Entwickler, die die Assembly einbinden, also von außerhalb Ihrer Assembly zugreifen, können Sie aber nicht mehr direkt manipulieren. Solcherlei Eigenschaften sind dann wichtig, wenn es sich bei ihnen um Quasi-Konstanten handeln soll. Ihre Assembly definiert die Eigenschaft wann und wie sie will auf Grund bestimmter Zustände. Die Assemblies, die sie konsumieren, müssen aber mit dieser Einstellung leben. Klassen, die beispielsweise Einstellungen aus der Windows-Registry widerspiegeln, können davon Gebrauch machen. Denkbar wäre auch, eine Verbindungszeichenfolge zu einem SQL Server auf diese Weise freizulegen – Sie hätten zwar aus Ihrer Assembly heraus Manipulationsspielraum für die Verbindungszeichenfolge; eine Assembly könnte aber immer nur die Verbindungszeichenfolge mit der Eigenschaft auslesen, die Sie innerhalb Ihrer Assembly vorgeben.

Statische Komponenten

Bislang haben Sie im Rahmen von Klassen nur Member-Variablen (Felder), Member-Eigenschaften und Member-Methoden kennengelernt, also solche Methoden, die zum sauberen Funktionieren auf die Instanzvariablen zugreifen mussten, wofür sie in jedem Fall eine Instanz der Klasse benötigten – anderenfalls hätten Sie eine `NullReferenceException` ausgelöst, wie in Kapitel 13 im Abschnitt »Nothing« zu lesen.

Sie haben aber auch die Möglichkeit, statische Komponenten zu erstellen, die eben nicht auf Member-Variablen einer Klasse zurückgreifen, und damit auch ohne Instanziierung aufrufbar sind. Die folgenden Abschnitte zeigen, wie's geht:

Nicht durch die Schlüsselwörter Static und Shared in Visual Basic verwirren lassen!

Static im Gegensatz zu Shared bewirkt, dass eine Variable, die nur innerhalb einer Prozedur (Sub oder Function) verwendet wird, ihren Inhalt auch nach Verlassen der Unterroutine nicht verliert. Dennoch können Sie auf diese Variable nur innerhalb der Unterroutine zugreifen, in der sie definiert wird. Im Prinzip ist eine mit Static definierte Variable eine, die als Shared-Member für die Klasse definiert und mit einem internen Attribut versehen wurde, das die Verwendung auf den Gültigkeitsbereich reglementiert, in dem sie deklariert wurde.

Vorhanden sind Static-Variablen übrigens nur aus Gründen der Kompatibilität zum alten Visual Basic 6.0 (und vorherigen Versionen). Und um die Verwirrung komplett zu machen: In C# ist static das, was in VB.NET Shared ist!

Statische Methoden

Statische Methoden sind Methoden (Sub, Function) die mit dem Schlüsselwort Shared (nicht Static! – Siehe vorheriger Kasten!) gekennzeichnet und damit direkt über die Klasse und nicht über die Klasseninstanz definiert sind. Das bedeutet: Anders als bei Instanzmethoden können Sie eine statische Methode direkt über den Klassennamen aufrufen und brauchen eben keine Instanz dafür.

Ein gutes Beispiel für eine statische Methode ist die Parse-Methode aller numerischen Datentypen. Möchten Sie beispielsweise eine Zeichenfolge in einen Integer-Wert konvertieren, bedienen Sie sich genau dieser Funktion:

```
Dim intVar as Integer = Integer.Parse("1234")
```

Hier sehen Sie, dass Sie sich für den Funktionsaufruf keiner Instanzmethode bedienen, da auch keine Member-Variablen der Instanz für den Konvertierungsprozess benötigt werden. Im umgekehrten Fall ist das anders. Möchten Sie einen Integer-Wert in eine Zeichenkette umwandeln, bedienen Sie sich der Instanzmethode ToString, die natürlich auf die interne Member-Variable des Integer-Datentyps zurückgreift:

```
Dim intVar as Integer = 5
Debug.Print(intVar.ToString())
```

HINWEIS Im Grunde genommen müsste die EllipseString-Methode unseres Klassenbeispiels ebenfalls statisch und damit mit Shared definiert werden, da sie ebenfalls nur einen Algorithmus kapselt, der auf *keine* Member-Variable der Klasse direkt zugreift. Konsequenterweise sollte diese Methode dann auch als öffentlich definiert werden, sodass ihre korrekte Signatur folgendermaßen aussehen sollte:

```
Public Shared Function EllipseString(ByVal text As String, ByVal MaxLength As Integer) As String
    .
    .
    .
End Function
```

Die richtige Funktionsweise dieser Routine wäre dann:

```
Dim s As String = Kontakt.EllipseString("Dies ist die abzuschneidende Zeichenkette", 20)
```

Sollten Sie eine Objektvariable auf eine instanzierte Kontakt-Instanz haben, könnten Sie rein theoretisch auch über diese Objektvariable auf die statische Methode zugreifen. Doch Sie sollten das nicht machen, da es für Verwirrung sorgt. Die Verwendung von Objektvariablen impliziert immer, dass Sie etwas mit der Klasseninstanz (also dem Dateninhalt der Klasse) anstellen wollen, was Sie in diesem Fall gar nicht machen, da, wie gerade gesagt, statische Methoden keine Member-Variablen nutzen (und auch gar nicht nutzen können!). Dieser Meinung ist auch der Background-Compiler von Visual Basic, denn einen Versuch, eine statische Methode über eine Objektvariable aufzurufen, quittiert er, wie in Abbildung 14.9 zu sehen, mit einer Warnung.

Wie in Abbildung 14.9 zu sehen, gibt es – vielleicht kennen Sie das schon von Office 2003 – im Visual Studio-Editor nach einer Bearbeitung eines Objektes bzw. Objektnamens so genannte Smarttags; so Sie sich Kapitel 5 zu Gemüte geführt haben, wissen Sie darüber längst Bescheid. Der folgende graue Kasten fasst den Umgang mit Smarttags nochmals kurz zusammen.

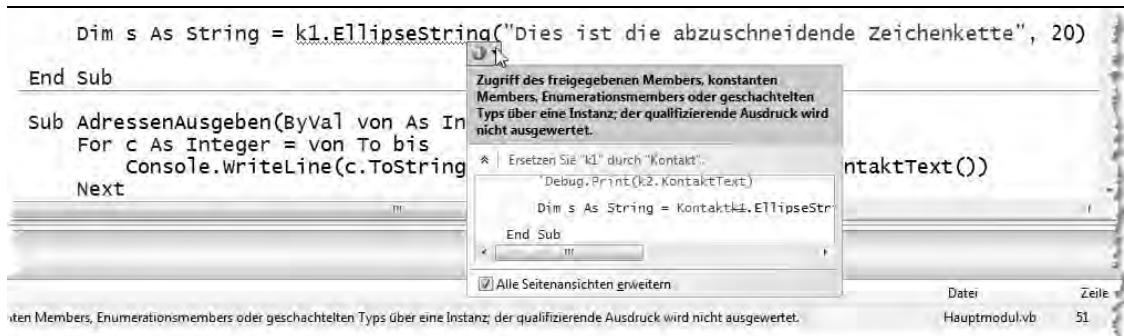


Abbildung 14.9 Einen Versuch, eine statische Methode über eine Objektvariable aufzurufen, quittiert der Background-Compiler von Visual Basic mit einer Warnung. Die Fehlerkorrekturoptionen helfen Ihnen übrigens beim Berichtigen.

Smarttags im Editor von Visual Basic

Einen Smarttag erkennen Sie zunächst als kleinen, hellen Unterstrich in einer Codezeile im Visual Basic-Editor, an der es seiner Meinung nach irgendetwas zu verbessern oder anzumerken gibt.

Fahren Sie mit dem Mauszeiger auf diesen Unterstrich, verwandelt er sich in ein kleines Symbol mit der Form eines Ausrufungszeichen, das Ihnen verschiedene Arten von Unterstützung anbietet (welche, kommt ganz auf den Zusammenhang an). In vielen Fällen verbirgt sich hinter dem Smarttag ein Dialog, der Ihnen einen Korrekturvorschlag für den erkannten »Fehler« unterbreitet.

Die folgende Abbildung verdeutlicht den Zusammenhang.



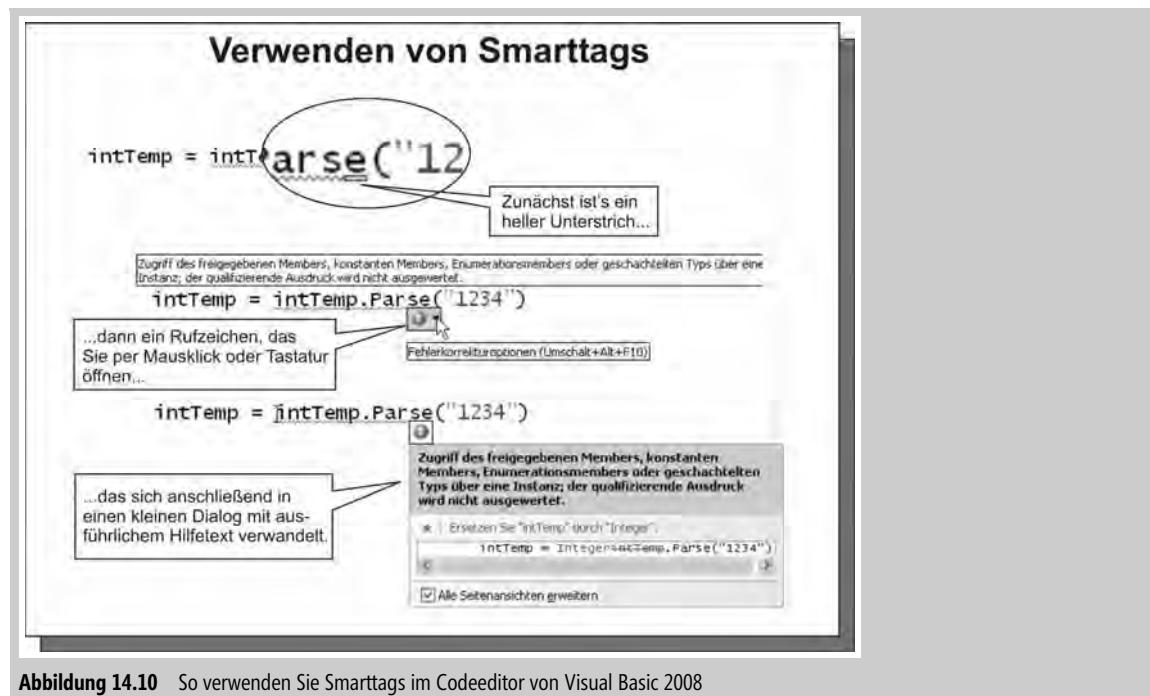


Abbildung 14.10 So verwenden Sie Smarttags im Codeeditor von Visual Basic 2008

Statische Eigenschaften

Visual Basic sieht auch statische Eigenschaften vor. Genau wie bei statischen Funktionen sind statische Eigenschaften direkt über die Klasse und nicht über die Klasseninstanz definiert. Das bedeutet, dass statische Eigenschaften Funktionalitäten bereitstellen sollten, die für alle Objekte dieser Klasse gelten und nicht für eine bestimmte Objektinstanz.

Das beste Beispiel für eine statische Eigenschaft ist die `Now`-Eigenschaft von `DateTime`. Sie liefert die aktuelle Uhrzeit zurück und ist natürlich von den Eigenschaften einzelner `DateTime`-Instanzen völlig unabhängig. Statische Eigenschaften werden, ebenfalls wie statische Funktionen, mit dem `Shared`-Schlüsselwort deklariert. Ein Beispiel für eine statische Eigenschaft finden Sie im folgenden Abschnitt.

Module in Visual Basic – automatisch statische Klassen erstellen

Module gibt es bei allen bislang existierenden .NET-Programmiersprachen nur in Visual Basic. Und auch hier ist ein Modul nichts weiter als eine Mogelpackung, denn das, was als Modul bezeichnet wird, ist im Grunde genommen nichts anderes als eine abstrakte Klasse (eine, die also nicht instanziierbar ist) mit ausschließlich statischen Methoden, statischen Eigenschaften und statischen öffentlichen Feldern.

Halten wir fest:

- Ein Modul ist nicht instanziierbar. Eine abstrakte Klasse auch nicht.
- Ein Modul kann keine überschreibbaren Prozeduren zur Verfügung stellen. Die statischen Prozeduren einer Klasse können das auch nicht.

- Ein Modul kann nur Prozeduren zur Verfügung stellen, auf die aber nur ohne Instanzobjekt direkt zugegriffen werden kann. Das Gleiche gilt für die statischen Prozeduren einer abstrakten Basisklasse.

Es gibt aber auch feine Unterschiede: So können Module beispielsweise keine Schnittstellen implementieren; das können zwar abstrakte Klassen, aber Sie können keine statischen Schnittstellenelemente definieren. Insofern ist dieser scheinbare Unterschied in Wirklichkeit gar keiner. Ein Modul kann auch nur auf oberster Ebene definiert und nicht in einem anderen geschachtelt werden.

Module setzen Sie bei der OOP vorschlagsweise so wenig wie möglich ein, denn sie widersprechen dem Anspruch von .NET, möglichst wieder verwendbaren Code zu schaffen.

Hier im Buch finden Sie aus diesem Grund Module nur, wenn es um »Quick-And-Dirty«-Projekte geht, bei denen beispielsweise eine Konsolen-Anwendung Tests durchzuführen hat oder »mal eben« etwas demonstrieren soll, genauso, wie Sie es in den vergangenen Kapiteln bereits erlebt haben.

Delegaten und Lambda-Ausdrücke

Lambda-Ausdrücke sind quasi anonyme Funktionen, die Ausdrücke und Anweisungen enthalten und für die Erstellung von Delegaten oder Ausdrucksbaumstrukturen verwendet werden können.

Gerade wieder mit Schwerpunkt auf LINQ ist es oft notwendig, an bestimmten Stellen Delegaten – also Funktionszeiger – einzusetzen, die aber ausschließlich bei einem konkreten Aufruf und mit einer Minimalausstattung an Code zureckkommen.

Delegaten sind Typen, die Adressen speichern – ähnlich wie Objektvariablen Zeiger auf Instanzen von Klassen. Allerdings speichern Delegaten keine Speicheradressen auf Objektdaten im Managed Heap, sondern die Speicheradressen von Methoden, die dann, wie eine Sub oder Function selbst, über die Delegaten aufgerufen werden können. Das praktische daran: Eine DelegatenvARIABLE kann mal auf die eine oder mal auf die andere Methode zeigen – je nachdem, welche Methoden eben im entsprechenden Kontext Sinn machen. Wichtig nur: Die Methoden müssen dabei die gleiche *Signatur* besitzen, also die gleichen Typen von Parametern in der gleichen Reihenfolge übernehmen.

Umgang mit Delegaten am Beispiel

Schauen wir uns eine weitere Version unseres bisherigen Beispiels an, zunächst um ein Gefühl für die Verwendung von Delegaten zu bekommen.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 14\\MiniAdressoClass V6

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wir bauen die Sortierroutine des ursprünglichen Beispiels dort so um, dass der Sortierkriteriumsausdruck, der bislang fest im Programm mit dem Nachnamen des Kontakts verdrahtet war, durch einen Delegaten ersetzt wird. Der Delegat wiederum erwartet zwei Parameter, nämlich die Kontakte, die miteinander verglichen werden sollen. Den Rückgabewert liefert er vom Typ Integer zurück, mit folgendem Hintergrund:

Ergibt das Funktionsergebnis 0, sind beide Kontakte gleich, bei 1 ist der erste größer, bei -1 ist der erste kleiner. Was letzten Endes verglichen wird, regelt dann eine Prozedur, auf die die Delegatenvariable zeigen soll. Die Änderungen dazu sehen zunächst einmal folgendermaßen aus:

```

Public Delegate Function ComparerDelegate(ByVal k1 As Kontakt, ByVal k2 As Kontakt) As Integer

Sub AdressenSortieren(ByVal cDel As ComparerDelegate)

    Dim anzahlElemente As Integer = 101

    Dim aeussererZaehler As Integer
    Dim innererZaehler As Integer
    Dim delta As Integer

    Dim tempKontakt As Kontakt

    delta = 1

    'Größten Wert der Distanzfolge ermitteln
    Do
        delta = 3 * delta + 1
    Loop Until delta > anzahlElemente

    Do
        'Ist Abbruchkriterium - deswegen herunterzählen
        delta \= 3

        'Shellsort's Kernalgorithmus
        For aeussererZaehler = delta To anzahlElemente - 1
            tempKontakt = Kontakte(aeussererZaehler)

            innererZaehler = aeussererZaehler
            Do
                If cDel.Invoke(tempKontakt, Kontakte(innererZaehler - delta)) = 1 OrElse _
                    cDel.Invoke(tempKontakt, Kontakte(innererZaehler - delta)) = 0 Then Exit Do
                Kontakte(innererZaehler) = Kontakte(innererZaehler - delta)

                innererZaehler = innererZaehler - delta
                If (innererZaehler <= delta) Then Exit Do
            Loop
            Kontakte(innererZaehler) = tempKontakt
        Next
    Loop Until delta = 0
End Sub

```

An den zuletzt in Fettschrift formatierten Zeilen dieses Listings können Sie erkennen, wie der eigentliche Aufruf einer Methode erfolgt, die durch eine Delegatenvariable referenziert wird – nämlich mit der `Invoke`-Methode, der genau die Parameter übergeben werden, die sie normalerweise der Methode übergeben würde.

Und jetzt schauen wir uns an, wie wir uns diesen Umbau für eine flexiblere Nutzung der Sortierroutine zu Nutze machen können:

```

Sub Main()
.

.

'Zufallsadressen generieren
Console.WriteLine("Zufallsadressen werden generiert ... ")
ZufallsAdressenGenerieren()
Console.WriteLine("fertig!")

'Die ersten 10 Zufallsadressen ausgeben
AdressenAusgeben(0, 10)

'Die Adressen nach Nachnamen sortieren
Console.WriteLine()
Console.WriteLine("Adressen werden nach Nachnamen sortiert ... ")
Dim compDelegate As ComparerDelegate = AddressOf KontaktNachnamenVergleich
AdressenSortieren(compDelegate)
Console.WriteLine("fertig!")
Console.WriteLine()

'Die ersten 10 Zufallsadressen ausgeben
AdressenAusgeben(0, 10)

'Die Adressen nach Ortsnamen sortieren
Console.WriteLine()
Console.WriteLine("Adressen werden nach Ortsnamen sortiert ... ")
compDelegate = AddressOf KontaktOrtVergleich
AdressenSortieren(compDelegate)
Console.WriteLine("fertig!")
Console.WriteLine()

.

.

End Sub

```

Hier werden jetzt an zwei Stellen der Sortierroutine die Vergleichsdelegaten übergeben, mit denen dann die eigentliche Sortierung durchgeführt wird. Diese beiden Methoden schauen folgendermaßen aus:

```

Function KontaktNachnamenVergleich(ByVal k1 As Kontakt, ByVal k2 As Kontakt) As Integer
    Return String.Compare(k1.Nachname, k2.Nachname)
End Function

Function KontaktOrtVergleich(ByVal k1 As Kontakt, ByVal k2 As Kontakt) As Integer
    Return String.Compare(k1.Ort, k2.Ort)
End Function

```

Soweit, so gut. Und was passiert jetzt genau, wenn das Programm gestartet wird? Die folgende Schritt-für-Schritt-Erklärung macht es deutlich:

1. Nachdem die Zufallsadressen erstellt wurden, definiert die Anwendung die Delegatenvariable `compDelegate` und weist dieser gleichzeitig die Adresse der `KontaktNachnamenVergleich`-Methode zu.
2. Sie ruft die Methode `AdressenSortieren` auf und übergibt ihr gleichzeitig die Delegatenvariable.

3. AdresseSortieren wiederum führt Invoke auf die Delegatenvariable aus und ruft dabei für den Elementvergleich die Methode KontaktNachnamenVergleich auf. Die Kontaktliste wird damit nach Nachnamen sortiert.
4. Wieder zurück im Hauptmodul wird der Delegatenvariable compDelegate nach der Sortierung die Adresse der KontaktOrtVergleich-Methode zugewiesen.
5. Die Anwendung ruft abermals die Methode AdressenSortieren auf und übergibt ihr gleichzeitig die Delegatenvariable, die jetzt allerdings auf eine andere Methode als zuvor zeigt.
6. AdresseSortieren führt Invoke auf die Delegatenvariable aus und ruft aber dieses Mal dabei für den Elementvergleich die Methode KontaktOrtVergleich auf. Die Kontaktliste wird damit beim zweiten Durchlauf nach dem Ortsnamen sortiert.

Lambda-Ausdrücke

Bei einem Lambda-Ausdruck handelt es sich um eine quasi namenlose Funktion, die ein einzelnes Ergebnis zurückliefert. Lambda-Ausdrücke können an allen Stellen verwendet werden, an denen ein Delegatyp gültig ist.

Das folgende Beispiel stellt einen Lambda-Ausdruck dar, der das ihm übergebene Argument um eins erhöht und den Wert als Funktionsergebnis zurückliefert.

```
Function (num As Integer) num + 1
```

So für sich steht dieser Lambda-Ausdruck recht »lose im Raum«. Zu seiner wiederholten Verwendung kann in vereinfachter Form durch lokalen Typenrückschluss (siehe Kapitel 11) eine Delegatenvariable definiert werden, die die Adresse dieses Lambda-Ausdrucks aufnimmt:

```
Dim add1 = Function(num As Integer) num + 1
```

Im Weiteren können Sie dann für den Aufruf diese Variable verwenden, wobei Sie den Wert in gewohnter Weise als Parameter übergeben.

```
'Die folgende Codezeile bewirkt, dass 6 ausgegeben wird:  
Console.WriteLine(add1(5))
```

Alternativ kann die Funktion gleichzeitig deklariert und ausgeführt werden.

```
Console.WriteLine((Function(num As Integer) num + 1)(5))
```

Ein Lambda-Ausdruck kann als Wert eines Funktionsaufrufs zurückgegeben oder als Argument an einen Delegatenparameter übergeben werden. Im folgenden Beispiel werden boolesche Lambda-Ausdrücke als Argument an die testResult-Methode übergeben. Mit dieser Methode wird der boolesche Test auf ein Ganzzahlenargument, value, angewendet. Wenn der Lambda-Ausdruck True zurückgibt, wird »Erfolgreich« angezeigt, falls er False zurückgibt, wird »Fehlgeschlagen« angezeigt.

```
Module Module2  
  
Sub Main()  
    ' Die folgende Zeile gibt 'Erfolgreich' aus, da 4 gerade ist.  
    testResult(4, Function(num) num Mod 2 = 0)  
    ' Die folgende Zeile gibt 'Fehlgeschlagen' aus, da 5 nicht > 10 entspricht.  
End Sub
```

```
    testResult(5, Function(num) num > 10)
End Sub

Sub testResult(ByVal value As Integer, ByVal fun As Func(Of Integer, Boolean))
    If fun(value) Then
        Console.WriteLine("Erfolgreich")
    Else
        Console.WriteLine("Fehlgeschlagen")
    End If
End Sub

End Module
```

Lambda-Ausdrücke am Beispiel

Unser uns permanent begleitendes Beispiel können wir mithilfe von Lambda-Ausdrücken stark vereinfachen, da wir den Zwischenschritt der Delegatenvariablendefinition nicht mehr benötigen, sondern der Sortierroutine direkt den Sortierkriteriumsausdruck als Lambda-Ausdruck übergeben können.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 14\\MiniAdressoClass V7

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Sub Main()
.

.

.

'Die Adressen nach Nachnamen sortieren
Console.WriteLine()
Console.Write("Adressen werden nach Nachnamen sortiert ... ")
AdressenSortieren((Function(k1 As Kontakt, k2 As Kontakt) -
String.Compare(k1.Nachname, k2.Nachname)))
Console.WriteLine("fertig!")
Console.WriteLine()

'Die ersten 10 Zufallsadressen ausgeben
AdressenAusgeben(0, 10)

'Die Adressen nach Ortsnamen sortieren
Console.WriteLine()
Console.Write("Adressen werden nach Ortsnamen sortiert ... ")
AdressenSortieren((Function(k1 As Kontakt, k2 As Kontakt) -
String.Compare(k1.Ort, k2.Ort)))
Console.WriteLine("fertig!")
Console.WriteLine()

.

.

.

End Sub
```

Die Sortierroutine kann diesen Lambda-Ausdruck ebenfalls direkt als solchen entgegennehmen und braucht ebenfalls nicht mehr den »Umweg« über einen definierten Delegatentyp zu gehen:

```
'Lambda-Funktion nimmt zwei Kontakt-Objekte entgegen und liefert Integer zurück
Sub AdressenSortieren(ByVal cDel As Func(Of Kontakt, Kontakt, Integer))

    Dim anzahlElemente As Integer = 101

    Dim aeussererZaehler As Integer
    Dim innererZaehler As Integer
    Dim delta As Integer

    Dim tempKontakt As Kontakt

    delta = 1

    'Größten Wert der Distanzfolge ermitteln
    Do
        delta = 3 * delta + 1
    Loop Until delta > anzahlElemente

    Do
        'Ist Abbruchkriterium - deswegen herunterzählen
        delta \= 3

        'Shellsort's Kernelalgorithmus
        For aeussererZaehler = delta To anzahlElemente - 1
            tempKontakt = Kontakte(aeussererZaehler)

            innererZaehler = aeussererZaehler
            Do
                'Invoke ist optional! - Es geht auch ohne:
                If cDel(tempKontakt, Kontakte(innererZaehler - delta)) = 1 OrElse
                    cDel(tempKontakt, Kontakte(innererZaehler - delta)) = 0 Then Exit Do
                Kontakte(innererZaehler) = Kontakte(innererZaehler - delta)

                innererZaehler = innererZaehler - delta
                If (innererZaehler <= delta) Then Exit Do
            Loop
            Kontakte(innererZaehler) = tempKontakt
        Next
    Loop Until delta = 0
End Sub
```

HINWEIS Der Einsatz von Lambda-Ausdrücken macht insbesondere dort Sinn, wo Sie es mit so genannten Predicate-, Comparison- oder Action-Delegaten zu tun haben, und das ist sowohl bei der nichtgenerischen ArrayList als auch bei der generischen List(Of)-Auflistung der Fall. Kapitel 23 hält mehr darüber bereit. Dort finden Sie auch im Rahmen eines etwas ausführlicheren List(Of)-Beispiels entsprechenden Beispielcode dazu.

Über mehrere Codedateien aufgeteilter Klassencode – Das Partial-Schlüsselwort

Sie können die Definition einer Klasse oder Struktur mithilfe des `Partial`-Schlüsselworts auf mehrere Definitionsabschnitte aufteilen. Das bedeutet: Sie können beliebig viele Teildefinitionen einer Klasse in beliebig vielen unterschiedlichen Quelldateien unterbringen. Alle Definitionen müssen sich jedoch in der gleichen Assembly und dem gleichen Namespace befinden.

HINWEIS Visual Basic verwendet partielle Klassendefinitionen, um in jeweils eigenen Quelldateien generierten Code von dem Code zu trennen, der vom Benutzer erstellt wurde. Zum Beispiel definiert der Windows Form-Designer partielle Klassen für Steuerelemente, z.B. Form. Sie sollten den generierten Code in diesen Codedateien nicht ändern.

Normalerweise wird die Entwicklung einer einzelnen Klasse oder Struktur nicht auf zwei oder mehr Codeklassen verteilt. In der Regel benötigen Sie das `Partial`-Schlüsselwort daher nicht. Sinn ergibt die Anwendung von `Partial` aber dann, wenn der Code einer Klasse dermaßen anwächst, dass die Übersicht sehr darunter leiden würde. Eine gute Vorgehensweise besteht dann darin, innerhalb der Projektmappe für die mit `Partial` aufgeteilte Klasse einen zusätzlichen Ordner anzulegen, und die einzelnen Codedateien der Klasse dort abzulegen.

Beim Erstellen einer partiellen Klasse oder Struktur gelten übrigens alle Regeln für die Erstellung von Klassen und Strukturen, beispielsweise diejenigen für die Verwendung und Vererbung von Modifizierern.

Übrigens: Nur die zusätzlichen Codedateien benötigen das `Partial`-Schlüsselwort, um den Klassencode zu erweitern – beim Ausgangsklassencode *kann* man es angeben, muss dies aber nicht tun.

Ein Beispiel für die Anwendung von `Partial` finden Sie in Kapitel 27 beim Beispiel des Formel Parsers.

Partieller Klassencode bei Methoden und Eigenschaften

Was für die Klasse selbst gilt, hat auch für Methoden- und Eigenschaften-Definitionen Gültigkeit, nur in leicht abgeänderter Weise. Partieller Klassencode dient insbesondere dazu, automatisch generierten Code mit Code, der diesen modifiziert, besser pflegen zu können.

Angenommen, Sie haben aus einem Entitäts-Modell des Entity-Framework für den Zugriff auf eine Datenbank über ein Objektmodell Code generieren lassen, und müssen diesen aus irgendwelchen Gründen erweitern. Ändern Sie anschließend das Entitäts-Modell und lassen diesen Code dann neu erstellen, wären Ihre Änderungen verloren.

'Auszug aus einer Codegenerator erstellten Datei für ein Objektmodell eines Entity Data Modells:

```
'''<summary>
'''Initialisiert ein neues AWEntities-Objekt.
'''</summary>
Public Sub New(ByVal connection As Global.System.Data.EntityClient.EntityConnection)
    MyBase.New(connection, "AWEntities")

    'Hier wird eine Methode generierten Klasse aufgerufen ...
    Me.OnContextCreated
End Sub
```

```
'Die aber hier nur durch Partial ihren Methodenrumpf definiert.  
'Der eigentliche Code kann in einer separaten Codedatei liegen,  
'der beim Neuerstellen dieses Codes nicht mehr überschrieben werden kann.  
Partial Private Sub OnContextCreated()  
End Sub
```

Aus diesem Grund gibt es auch auf Methoden-Ebene die Möglichkeit, mit **Partial** »leere« Methodenrümpfe zu definieren, den eigentlichen Code, der diese Methoden »ausfüllt«, aber in dann in einer anderen Code-datei unterzubringen. Das Ergebnis: Sie können den generierten Code nun immer wieder neu erstellen, Ihr manuell editierter Code bleibt stets in einer separaten Codedatei erhalten.

Kapitel 15

Vererben von Klassen und Polymorphie

In diesem Kapitel:

Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance)	450
Überschreiben von Methoden und Eigenschaften	459
Das Speichern von Objekten im Arbeitsspeicher	463
Polymorphie	466
Abstrakte Klassen und virtuelle Prozeduren	485
Schnittstellen (Interfaces)	488
Die Methoden und Eigenschaften von Object	503
Shadowing (Überschatten) von Klassenprozeduren	510
Sonderform »Modul« in Visual Basic	516
Singleton-Klassen und Klassen, die sich selbst instanzieren	516

Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance)

Vererbung und die Wiederverwendbarkeit in Form von Erweiterung von Klassen sind zentrale Programmietechniken, von dem Framework sehr regen Gebrauch macht. Man könnte sogar soweit gehen zu sagen, dass ohne die Möglichkeit, Klassen zu vererben, .NET keinen Sinn machen würde.

Visual Basic 6.0 beherrschte die Polymorphie¹ – die Möglichkeit, über gleiche Methodennamen verschiedene Klassen anzusprechen – nur sehr unzulänglich. In Visual Basic 6.0 konnten Sie Klassen nur durch die so genannte Delegation vererben: Dabei wird eine Klasse in eine andere Klasse als Member-Variable eingebunden und deren Eigenschaften durch neue Funktionen und Eigenschaftenprozeduren nach außen offen gelegt. In VB6 war die richtige Polymorphie auf Schnittstellen beschränkt – mehr zu diesem Thema gibt's im Abschnitt »Schnittstellen (Interfaces)« ab Seite 488.

Bevor wir uns die Technik der Vererbung für das Beispiel zunutze machen, möchte ich Ihnen anhand einfacher Codebeispiele den Vorgang des Vererbens erklären. Unser Beispiel möchte ich dazu nicht als erstes verwenden, da es bereits einer Sonderbehandlung bei der Vererbung bedarf, die am Anfang verwirren würde und dem Verständnis für einen Moment im Wege wäre.

BEGLEITDATEIEN

Sie finden das folgende Beispielprojekt im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Vererbung01

Sie sollten aber vorschlagsweise die folgende Übung Schritt für Schritt nachvollziehen, die dieses Projekt aufbaut, um ein besseres Verständnis für das Drumherum zu bekommen!

- Legen Sie ein neues Projekt an, indem Sie *Neu* und *Projekt* aus dem Menü *Datei* wählen.
- Klicken Sie auf *Visual Basic-Projekte* in der Liste *Projekttypen*, wählen Sie *Konsolenanwendung* unter *Vorlagen* und bestimmen Sie »Vererbung« als Projektname.
- Bestimmen Sie ein Speicherverzeichnis Ihrer Wahl.
- Klicken Sie auf *OK*, um das neue Projekt zu erstellen.
- Doppelklicken Sie auf *Module1.vb* im Projektmappen-Explorer, um den Code für das Projekt anzuzeigen. Lassen Sie sich zunächst nicht durch das Schlüsselwort *Module* irritieren. Auf das Thema Module (das aus Framework-Sicht betrachtet in Visual Basic wieder eine Sonderrolle spielt) werde ich im Laufe dieses Kapitels noch eingehen.
- Ändern Sie den Namen von *Module1.vb* in *mdlMain.vb*, indem Sie aus dem Kontextmenü den Befehl *Umbenennen* wählen und einen neuen Dateinamen eingeben. Denken Sie daran, die Dateinamenerweiterung *.vb* mit anzugeben!
- Doppelklicken Sie im Projektexplorer auf *mdlMain.vb*, um das Codefenster zu öffnen.

¹ Etwa: »Vielgestaltigkeit«.

Im Codeeditor unterhalb der Moduldefinition (also hinter *End Module*) geben Sie nun

```
Class ErsteKlasse
```

ein. Visual Basic erstellt automatisch die Zeile

```
End Class
```

darunter, um den Klassen-Codeblock abzuschließen. Diese erste Klasse soll eine Eigenschaft für die Wertezuweisung und eine Methode bekommen, mit der der Inhalt der Klasse ausgedruckt werden kann. Fügen Sie folgenden Code in die Klasse ein:

```
Class ErsteKlasse
    Protected myEinWert As Integer
    'Eigenschaft, um den Wert verändern zu können.
    Property EinWert() As Integer
        Get
            Return myEinWert
        End Get
        Set(ByVal Value As Integer)
            myEinWert = Value
        End Set
    End Property
    'Funktion, um den Inhalt als Zeichenkette (String) zurückzuliefern.
    Function AlsZeichenkette() As String
        Return CStr(myEinWert)
    End Function
End Class
```

Im Module *mdlMain* erstellen Sie nun ein kleines Rahmenprogramm, das diese neue Klasse zu Demonstrationszwecken verwendet:

```
Module mdlMain
    Sub Main()
        Dim klasse1 As New ErsteKlasse
        klasse1.EinWert = 5
        Console.WriteLine(klasse1.AlsZeichenkette())
        Console.WriteLine()
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadKey()
    End Sub
End Module
```

Sie können das Programm nun starten, und das Ergebnis wird dem entsprechen, was Sie sicherlich erwartet haben.

Für die nächsten Schritte stellen Sie sich einfach vor, dass Ihnen der Quelltext von ErsteKlasse nicht zur Verfügung steht. Unterhalb der Klassendefinition (die Sie natürlich im Geiste gar nicht sehen ...) fügen Sie nun eine weitere Klassendefinition ein:

```
Class ZweiteKlasse
    Inherits ErsteKlasse

End Class
```

Sie haben damit eine neue Klasse geschaffen, namens ZweiteKlasse. Diese Klasse hat keine Eigenschaften und keine Methoden, und sie hat sie doch. Wenn Sie den Cursor in die erste Prozedur der Codedatei platzieren, und unterhalb der Zeile

```
Console.WriteLine(klasse1.AlsZeichenkette())
```

die Zeile

```
Dim klasse2 As New ZweiteKlasse
```

einfügen, darunter den Instanznamen einfügen und den Punkt eintippen, sehen Sie, wie IntelliSense Ihnen die Elemente der neuen Klasse anbietet:

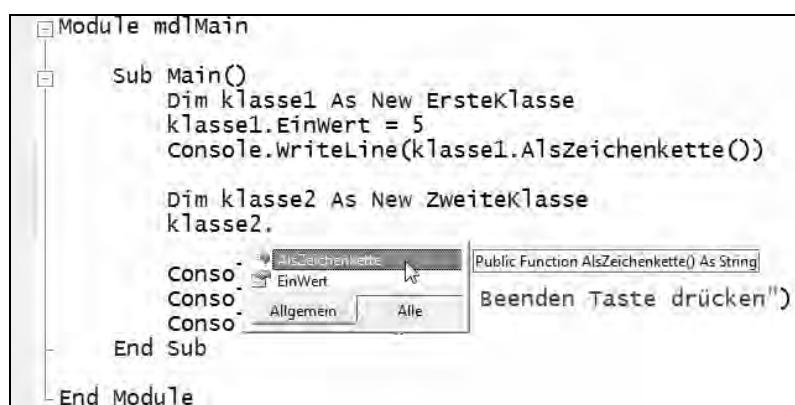


Abbildung 15.1 Obwohl es bislang keine Elementdefinitionen für ZweiteKlasse gibt, zeigt Ihnen IntelliSense dennoch eine Eigenschaft und eine Methode an

Was ist passiert?

Durch die Anweisung

```
Inherits ErsteKlasse
```

haben Sie bestimmt, dass ZweiteKlasse alle Elemente von ErsteKlasse erbt. Alles, was ErsteKlasse kann, können Sie auch mit ZweiteKlasse machen.

WICHTIG Genau darum geht es bei der objektorientierten Programmierung: Durch die Vererbung schreiben Sie wieder verwendbaren Code. Wenn Sie vorhandene Klassen verändern wollen, um sie an ein bestimmtes Problem anzupassen, dann können Sie die Originalklasse in dem Zustand belassen, den sie hat. Sie vererben sie einfach in eine neue Klasse (man sagt auch: Sie »leiten sie ab«), und verändern dann die vererbte Klasse, um sie für Ihre neue Problemlösung anzupassen und zu erweitern.

Angenommen, Sie brauchen für ein bestimmtes Problem eine Klasse, die genau das kann, was ErsteKlasse kann, nur benötigen Sie zusätzlich eine weitere Eigenschaft, die den aktuellen Instanzwert um den Wert 10 erhöht ermittelt. In diesem Fall vererben Sie Klasse ErsteKlasse, so Sie es im Beispiel schon kennengelernt haben, in ZweiteKlasse und fügen dann dort die neue Eigenschaftenprozedur hinzu:

```
Class ZweiteKlasse
    Inherits ErsteKlasse

    ReadOnly Property Um10Mehr() as Integer

        Get
            Return myEinWert + 10
        End Get
    End Property
End Class
```

Und das war es schon. Die gesamte Funktionalität, die ErsteKlasse hatte, hat ZweiteKlasse auch, und sie hat obendrein noch eine Eigenschaft mehr.

Gegenprobe: Unterhalb der Zeile

```
Dim klasse2 As New ZweiteKlasse
```

fügen Sie den Text **Console.WriteLine(klasse2.** ein; sobald Sie den Punkt tippen, zeigt Ihnen IntelliSense wieder die Elemente der Klasse an, und siehe da: Alle alten Elemente und die neue Eigenschaft sind in der Liste vorhanden!



Komplettieren Sie die Zeile,

```
Console.WriteLine(klasse2.Um10Mehr)
```

starten Sie das Programm anschließend, und schauen Sie, was passiert:

```
5  
10
```

Zum Beenden Taste drücken

Haben Sie erwartet, dass 15 als zweites Ergebnis ausgegeben wird? Natürlich nicht, denn die aus ErsteKlasse entstandene Instanz klasse1 ist völlig unabhängig von der Instanz klasse2, die aus ZweiteKlasse entstanden ist. Sie haben sich von ErsteKlasse nur den Code als Vorlage »geborgt« – die Objekte, die aus beiden Klassen entstehen können, haben natürlich beide einen eigenen und damit unterschiedlichen Datenbereich.

HINWEIS Wichtig für das Verständnis von Klassen ist, wie sie intern verwaltet werden. Der Code einer Klasse ist immer nur ein einziges Mal vorhanden – auch wenn Sie mehrere Instanzen einer Klasse anlegen. Deswegen ist es natürlich auch Unsinn zu glauben, dass eine Klasse die zehnmal soviel Programmcode hat wie eine Klasse »X«, bei zehnfacher Instanziierung in zehn verschiedene Objekte hundertmal so viel Speicher benötigt wie Klasse X. Sie braucht für den Programmcode immer gleich viel Speicher, egal wie viele Instanzen aus ihr entstehen. Der Programmcode ist natürlich auch nicht doppelt vorhanden, wenn Sie eine Klasse aus einer anderen ableiten. Nur der zusätzliche Programmcode durch das Hinzufügen oder Verändern vorhandener Elemente belegt zusätzlichen Speicher. Ausschließlich die Member-Variablen einer Klasse sind dafür ausschlaggebend, wie viel zusätzlichen Speicher eine Klasse beim Instanziieren in Objekte benötigt.

Damit das erwartete Ergebnis eintritt, müssen Sie das Programm wie folgt abändern:

```
Module mdlMain

Sub Main()
    Dim klasse1 As New ErsteKlasse
    klasse1.EinWert = 5
    Console.WriteLine(klasse1.AlsZeichenkette())

    Dim klasse2 As New ZweiteKlasse
    klasse2.EinWert = 5
    Console.WriteLine(klasse2.Um10Mehr)
    Console.WriteLine()
    Console.WriteLine("Zum Beenden Taste drücken")
    Console.ReadKey()
End Sub
End Module
```

Jetzt betrachten wir die Klassen mit dem IL-Disassembler, den Sie ja im letzten Kapitel schon kennengelernt haben, und schauen, ob uns der Visual Basic-Compiler wieder irgendetwas an Arbeit abgenommen hat.

- Starten Sie IL-Disassembler aus dem Start-Menu (der Programmgruppe Microsoft Windows SDK 6.0A), und öffnen Sie die Datei *Vererbung.exe*, die Sie im Verzeichnis *\bin\Debug* des Verzeichnisses finden, in dem Sie das Projekt angelegt hatten.
- Öffnen Sie per Mausklick auf das davor stehende Pluszeichen den Zweig *ErsteKlasse* und den Zweig *ZweiteKlasse*. Anschließend sollten Sie ein Bild vor sich sehen, etwa wie in Abbildung 15.3 zu sehen.

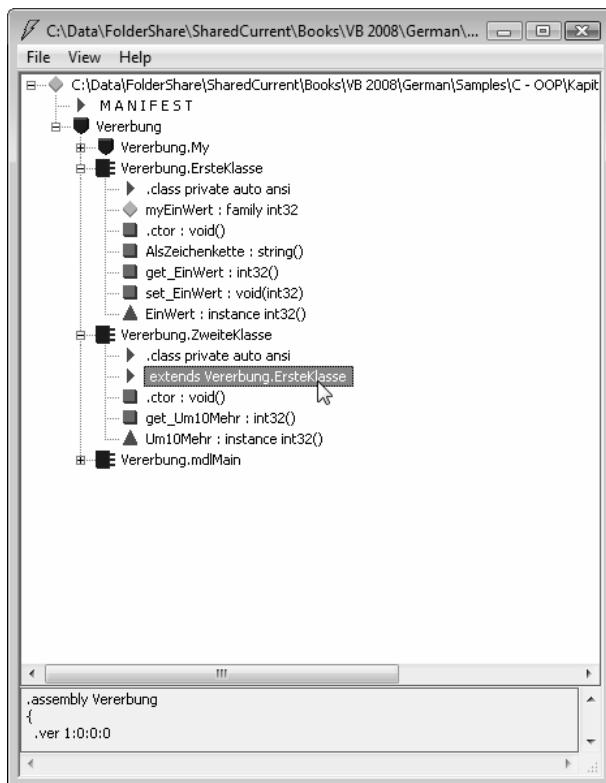


Abbildung 15.3 Die beiden Testklassen im IML-Disassembler

Genau wie in einem der vorherigen Beispiele, hat Ihnen Visual Basic wieder Arbeit abgenommen und sowohl in der Basisklasse ErsteKlasse als auch in der abgeleiteten Klasse ZweiteKlasse entsprechende Konstruktorprozeduren (.ctor) eingefügt. In der Abbildung ebenfalls auf den ersten Blick erkennbar: Eigenschaften werden intern in Funktionen bzw. Methoden umgewandelt. Aus der EinWert-Eigenschaft der Basisklasse, die jeweils einen Get- und einen Set-Accessor hat, macht der Visual Basic-Compiler die beiden Funktionen get_EinWert und set_EinWert. Die Eigenschaft selbst wird darunter definiert, und der IML-Code in ihr bestimmt lediglich, welche der in der Klasse vorhandenen Funktionen die Eigenschaft auflösen (per Doppelklick auf den Eigennamennamen können Sie sich den folgenden Code anzeigen lassen).

```

.property instance int32 EinWert()
{
  .set instance void Vererbung.ErsteKlasse::set_EinWert(int32)
  .get instance int32 Vererbung.ErsteKlasse::get_EinWert()
} // end of property ErsteKlasse::EinWert

```

Was noch auffällt: Genau wie vermutet, sind in ZweiteKlasse nur die zusätzlich vorhandenen Elemente als Code vorhanden. Dass die Klasse von der Basisklasse erbt und damit auch deren Elemente »mitnutzen« kann, wird – wie in Abbildung 15.3 gezeigt – durch extends Vererbung.ErsteKlasse geregelt. Da die einzige Eigenschaft nur einen Get-Accessor hat, gibt es übrigens auch nur eine Eigenschaftsfunktion in Form von get_Um10Mehr.

Ebenfalls erwähnenswert: Die Member-Variable `myEinWert` wurde vorausschauend als `Protected` deklariert. Laut Tabelle des letzten Abschnitts ist eine als `Protected` deklarierte Variable eine, »... [auf die nur] innerhalb derselben Klasse oder einer abgeleiteten Klasse zugegriffen werden kann«. Wäre die Variable nur als `Private` deklariert, könnte die abgeleitete Klasse `ZweiteKlasse` sie nicht manipulieren; die zusätzliche Eigenschaft, die sie implementiert, würde ergo nicht funktionieren.

TIPP Member-Variablen, die Sie in Klassen verwenden, welche später durch das Vererben wieder verwendet werden sollen, deklarieren Sie deshalb nach Möglichkeit als `Protected`, es sei denn, Sie wünschen ausdrücklich, dass nur die Basisklasse die Variable manipulieren darf.

Nachdem nun bekannt ist, dass abgeleitete Klassen automatisch den Standardkonstruktor der Basisklasse aufrufen, wäre es interessant zu erfahren, was passiert, wenn die Basisklasse keinen Standardkonstruktor hat. Einen solchen Fall haben Sie mit der letzten Version der Klasse aus dem Kontakt-Beispiel nämlich kennengelernt. Eine Klasse hat dann keinen automatisch generierten Standardkonstruktor, wenn sie einen parametrisierten Konstruktor hat.

Was passiert also, wenn wir der Klasse `ErsteKlasse` einen Konstruktor hinzufügen, der einen Parameter (zum Beispiel den Initialisierungswert für `myEinWert`) übernimmt? Probieren Sie es aus: Fügen Sie die Zeile

```
Sub New(ByVal NeuerWert As Integer)
```

in den Klassencode von `ErsteKlasse` ein. Sobald Sie die Änderungen eingefügt haben, sehen Sie diverse Fehlermarkierungen im Quelltext und die entsprechenden Beschreibungen dazu in der Fehlerliste, etwa wie in Abbildung 15.4 zu sehen.

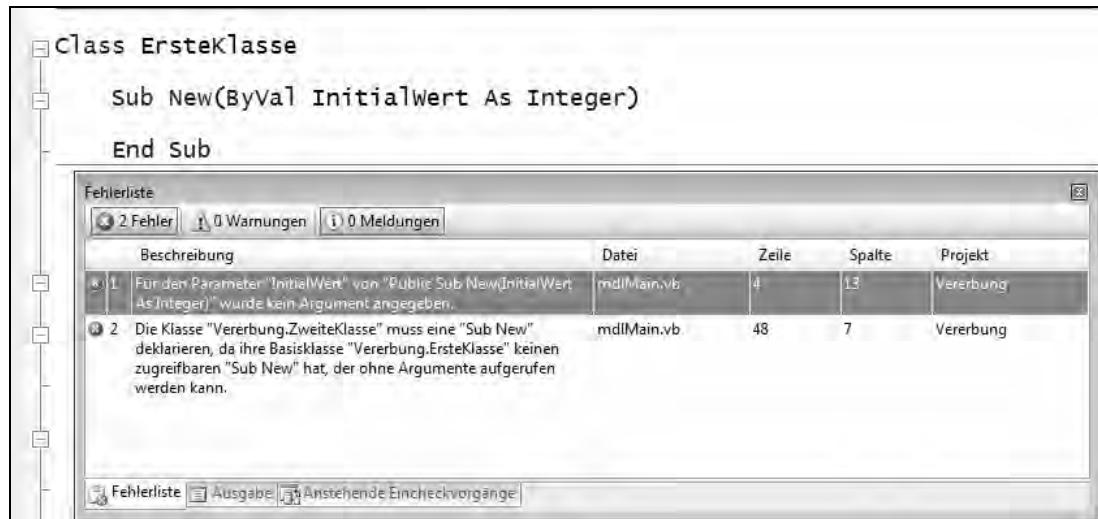


Abbildung 15.4 Nach dem Einfügen eines parametrisierten Konstruktors sehen Sie gleich zwei Fehler in der Fehlerliste

Der Grund: Im Modul innerhalb von Sub Main kann Klasse1 nicht mehr instanziert werden, denn hinter

```
Dim Klasse1 As New ErsteKlasse
```

steht kein Parameter. Da wir einen parametrisierten Konstruktor in die Klassendefinition eingefügt haben, gibt es keinen Standardkonstruktor mehr. Sobald Sie selbst irgendeinen Konstruktor (parametrisiert oder nicht) in eine Klasse einfügen, hört Visual Basic auf, Ihnen mit dem »Hineinkompilieren« eines Standardkonstruktors unter die Arme zu greifen. Kein Standardkonstruktor heißt: nehmen, was da ist.

Daher können wir noch einmal festhalten: Wenn Sie keinen Konstruktor (Sub New...) explizit implementieren, schreibt der Compiler einen solchen für Sie implizit (automatisch). Wenn Sie jedoch einmal anfangen, Kontruktoren explizit zu implementieren, müssen Sie alle Kontruktoren die Sie wünschen (auch den Standardkonstruktor ohne Parameter) explizit implementieren.

Und da ist nur einer, dem Parameter übergeben werden – der erste Fehler ist also erst erledigt, wenn Sie den Initialisierungswert für die Klasse mit angeben, etwa so:

```
Dim Klasse1 As New ErsteKlasse(5)
Console.WriteLine(Klasse1.AlsZeichenkette())
.
.
.
```

Kein Standardkonstruktor bedeutet aber auch: Die abgeleitete Klasse weiß nicht, wie sie die Basisklasse instanziieren soll (was auf jeden Fall passieren muss). In diesem Fall müssen Sie also selbst dafür Sorge tragen, dass die Basisklasse aufgerufen wird. Sie erreichen das, indem Sie den Konstruktor der Basisklasse mit dem MyBase-Schlüsselwort aufrufen.

Das folgende Listing stellt die funktionierende Version dar (Änderungen sind fett hervorgehoben; einige unwichtige Teile sind durch »...« abgekürzt):

```
Module mdlMain

    Sub Main()
        Dim Klasse1 As New ErsteKlasse(5)
        Console.WriteLine(Klasse1.AlsZeichenkette())

        Dim Klasse2 As New ZweiteKlasse(5)
        Console.WriteLine(Klasse2.Uml10Mehr)

        Console.WriteLine()
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadLine()
    End Sub

End Module

Class ErsteKlasse

    Protected myEinWert As Integer

    Sub New(ByVal NeuerWert As Integer)
```

```

    myEinWert = NeuerWert
End Sub

'Eigenschaft, um den Wert verändern zu können.
Property EinWert() As Integer
...
End Property
'Funktion, um den Inhalt als Zeichenkette (String) zurückzuliefern.
Function AlsZeichenkette() As String

    Return CStr(myEinWert)

End Function
End Class

Class ZweiteKlasse
    Inherits ErsteKlasse

    Sub New(ByVal NeuerWert As Integer)
        MyBase.New(NeuerWert)
    End Sub

    ReadOnly Property Um10Mehr() As Integer
    ...
End Property

End Class

```

Initialisierung von Member-Variablen bei Klassen ohne Standardkonstruktoren

Interessant ist zu sehen, was passiert, wenn Sie Member-Variablen schon bei ihrer Deklarierung definieren, etwa wie im folgenden Beispiel:

```

Class ErsteKlasse

    Protected myEinWert As Integer = 9

    Sub New()
        'Hat keinen Sinn, füllt aber den Konstruktor mit Code.
        Console.WriteLine("Testausgabe: Parameterloser Konstruktor")
    End Sub

    Sub New(ByVal AuszugebenderText As String)
        'Hat keinen Sinn, füllt aber den Konstruktor mit Code.
        Console.WriteLine("Testausgabe: " & AuszugebenderText)
    End Sub.

    .

```

In diesem Beispiel gibt es einen Standardkonstruktor und zusätzlich zwei parametrisierte. Da kein Code »außerhalb« einer Klasse ausgeführt werden kann, stellt sich die Frage: Wo findet die Zuweisung

```
Protected myEinWert As Integer = 9
```

denn eigentlich statt? Die Antwort offenbart wieder ein Blick in den IML-Code der Klasse. Der Compiler treibt hier den größten Aufwand, denn er muss den Code für das Initialisieren der Member-Variablen in jedem Konstruktor einbauen (siehe Abbildung 15.4). Nur so ist gewährleistet, dass alle erforderlichen Variableninitialisierungen *in jedem Fall* durchgeführt werden.

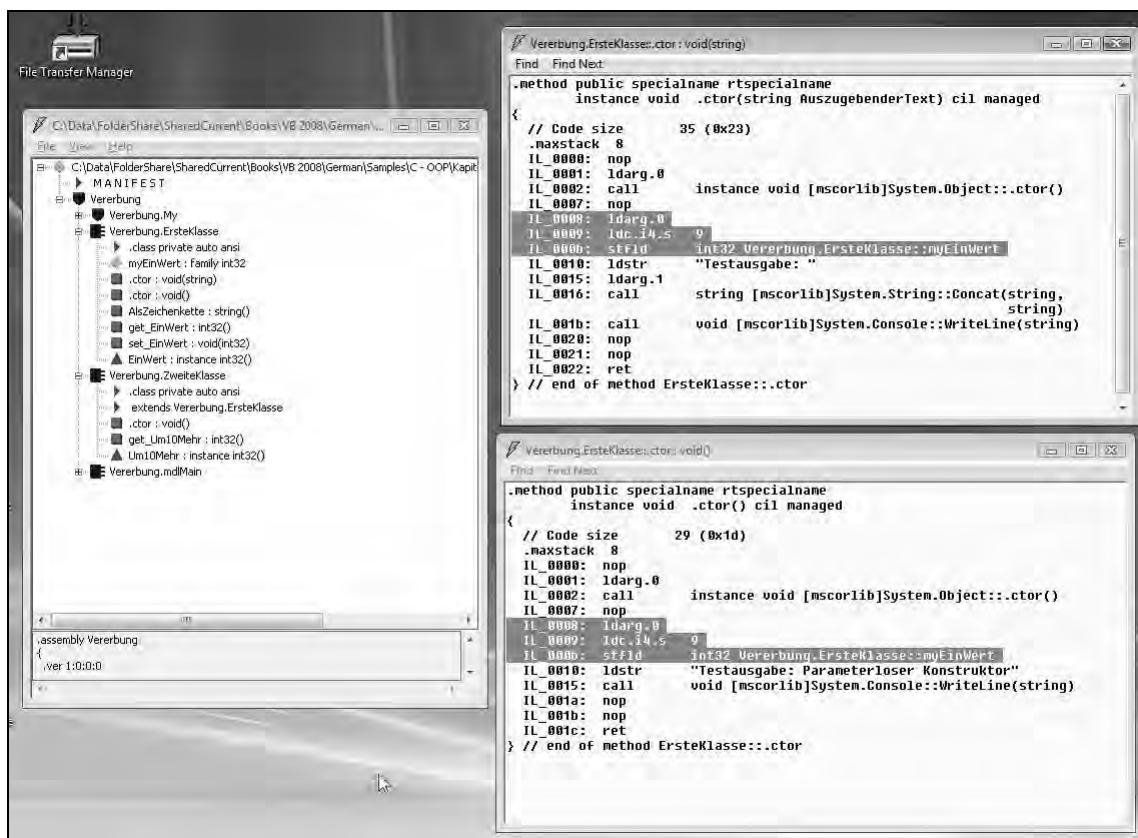


Abbildung 15.5 Initialisierungen von Member-Variablen werden im Bedarfsfall in jede Konstruktorprozedur eingebaut

Überschreiben von Methoden und Eigenschaften

Sie denken sicherlich auch, dass das Ableiten von Klassen eine ziemlich geniale Sache ist. Es strukturiert Ihre Programme, macht sie leichter lesbar und hilft Ihnen insbesondere beim sauberen Aufbau größerer und komplexer Projekte. Doch Klassen wären nicht evolutionär, und zwar im wahrsten Sinne des Wortes, wenn es nicht die Möglichkeit gäbe, bestimmte Funktionen einer Basisklasse durch eine andere der abgeleiteten auszutauschen. Klassen, die andere Klassen einbinden, werden dadurch unglaublich flexibel.

Das Ersetzen von Methoden oder Eigenschaften durch andere in einer abgeleiteten Basisklasse erfordert allerdings, dass die Basisklasse der Klasse, die sie ableitet, auch gestattet, bestimmte Funktionen zu überschreiben. Funktionen bzw. Methoden oder Eigenschaften müssen von Ihnen also explizit gekennzeichnet werden, damit sie später überschrieben werden dürfen. In Visual Basic geschieht die Kennzeichnung einer Prozedur einer Basisklasse zum Überschreiben mit dem Schlüsselwort `Overridable` (überschreibbar). Alle Funktionen und Eigenschaften, die mindestens als `Protected` gekennzeichnet sind, können den `Overridable`-Modifizierer tragen. Die Eigenschaften bzw. Methoden, die dann anschließend eine Funktion in der Basisklasse überschreiben, müssen wiederum mit dem Schlüsselwort `Overrides` (überschreibt) gekennzeichnet sein.

HINWEIS Es ist übrigens wichtig, dass Sie Überschreiben und Überladen nicht in einen Topf werfen, kräftig drin herrumröhren, und schauen, was anschließend dabei herauskommt: Visual Basic erlaubt es nämlich, dass eine abgeleitete Klasse die Prozedur einer Basisklasse überlädt. In diesem Fall ersetzen Sie die Basisfunktion nicht, Sie ergänzen diese nur um eine weitere Überladung. Das heißt im Klartext: Wenn Sie Prozeduren einer Basisklasse wirklich überschreiben (sie also ersetzen) wollen, dann grundsätzlich nur mit der gleichen Signatur wie die der Basisklasse.

WICHTIG In diesem Zusammenhang eine vielleicht nicht unwichtige Ergänzung: Während Sie beim Überladen von Funktionen innerhalb einer Klasse, die Sie komplett neu implementieren, das `Overloads`-Schlüsselwort auch weglassen und nur mit doppelten Methodennamen (aber unterschiedlichen Signaturen!) auskommen können, müssen Sie `Overloads` verwenden, wenn Sie eine Methode einer Basisklasse um eine weitere »Überladungsversion« in einer abgeleiteten Klasse ergänzen!

Ein kleines Beispiel zu überschriebenen Methoden: Angenommen, in der abgeleiteten Klasse `ZweiteKlasse` passt Ihnen die Art und Weise nicht, wie der String durch die Funktion `AlsZeichenkette` den Wert der Variablen als Zeichenkette zurückliefert. Sie möchten beispielsweise, dass die `AlsZeichenkette`-Funktion den Wert im Stil »der Wert lautet: xxx« ausgibt. In diesem Fall würden Sie die Klassen wie folgt verändern:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Vererbung02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module mdlMain
    Sub Main()
        Dim klasse1 As New ErsteKlasse(5)
        Console.WriteLine("Klasse1 ergibt durch AlsZeichenkette: " & klasse1.AlsZeichenkette())

        Dim klasse2 As New ZweiteKlasse(5)
        Console.WriteLine("Klasse2 ergibt durch AlsZeichenkette: " & klasse2.AlsZeichenkette())

        Console.WriteLine()
        Console.WriteLine("Return drücken, zum Beenden")
        Console.ReadLine()
    End Sub
End Module

Class ErsteKlasse

```

```
Protected myEinWert As Integer = 9

Sub New(ByVal NeuerWert As Integer)
    myEinWert = NeuerWert
End Sub

Sub New(ByVal NeuerWert As Integer, ByVal NurZumTesten As Integer)
    myEinWert = NeuerWert
End Sub

'Eigenschaft, um den Wert verändern zu können.
Property EinWert() As Integer
    .
    .
    .
End Property

'Um die Funktion als String auszudrucken
Public Overrides Function AlsZeichenkette() As String

    Return CStr(myNeuerWert)

End Function

End Class

Class ZweiteKlasse
    Inherits ErsteKlasse

    Sub New(ByVal NeuerWert As Integer)
        MyBase.New(NeuerWert)
    End Sub

    ReadOnly Property Um10Mehr() As Integer
        .
        .
        .
    End Property

    Public Overrides Function AlsZeichenkette() As String
        Return "der Wert lautet: " & MyBase.AlsZeichenkette()
    End Function

End Class
```

Wenn Sie diese veränderte Version des Programms laufen lassen, sehen Sie das folgende Ergebnis auf dem Bildschirm:

```
Klasse1 ergibt durch AlsZeichenkette: 5
Klasse2 ergibt durch AlsZeichenkette: der Wert lautet: 5

Zum Beenden Taste drücken
```

Überschreiben vorhandener Methoden und Eigenschaften von Framework-Klassen

Nun ist der Name unserer Beispielfunktion nicht sonderlich glücklich gewählt – `AlsZeichenkette` ist bestens ein deutsches Ausdrucksfragment, und um die Umwandlung in eine Zeichenkette beispielsweise in korrektem Englisch auszudrücken, um sich der Sprache des Frameworks selbst anzunähern, müsste es `ToString` heißen.

Wenn Sie allerdings die vorhandene Funktion `AlsZeichenkette` in der ersten Klasse in `ToString` ändern, passiert etwas, was nicht unbedingt vorhersagbar ist (siehe Abbildung 15.6):



Abbildung 15.6 Versuchen Sie `ToString` zu implementieren, sehen Sie diese Fehlermeldung

Nur, welche Methode ist hier gemeint? IntelliSense listet die vorhandenen Member eines Objektes doch auf, wie in Abbildung 15.2 zu sehen, war eine Methode namens `ToString` nicht dabei. Doch wir haben uns in der verwendeten Einstellung bisher auch nicht alle Member anzeigen lassen. IntelliSense erlaubt es nämlich, die weniger wichtigen Methoden in der Vervollständigungsliste auszublenden. Erst wenn Sie in der Liste auf *Alle* umschalten, werden auch wirklich alle Member angezeigt, so auch `ToString`, wie in Abbildung 15.7 zu sehen.

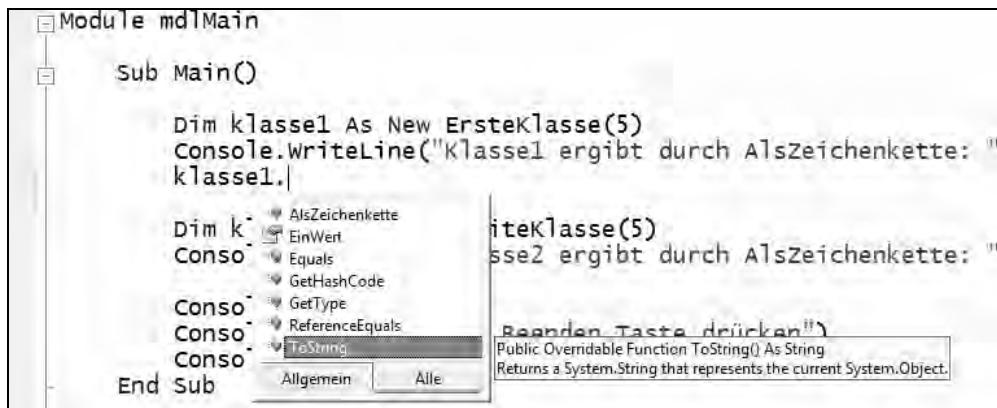


Abbildung 15.7 Jetzt sehen Sie alle Member der ErsteKlasse-Instanz

IntelliSense ist Ihnen an dieser Stelle von besonderem Nutzen, weil Sie die Modifizierer der `ToString`-Funktion in der Tooltip-Beschreibung auch gleich sehen können. Diese verraten Ihnen, dass die Funktion als `Overridable` deklariert wurde – Sie haben also selbst die Möglichkeit, die Ableitung dieser Klasse in `ErsteKlasse` mit `Overrides` zu überschreiben.

BEGLEITDATEIEN

Die auf diese Weise modifizierte Version des Beispiels finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Vererbung03

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Das Speichern von Objekten im Arbeitsspeicher

Bevor wir uns dem Allerwichtigsten dieses Kapitels nähern, wäre es zum besseren Verständnis überaus ratsam, ein paar Worte zur Speicherung von Objekten (und nahezu allen von ihnen abgeleiteten Klassen) zu verlieren. Objektvariablen und Objekte sind nämlich nicht so miteinander verbunden, wie man es sich zu Anfang vielleicht vorstellt. Im Gegenteil: Der Verbund einer Objektvariablen mit den eigentlichen Daten des Objektes hält bestenfalls so gut, wie eine amerikanische Prominentenehe: Was auf den ersten Blick so innig und für immer geschaffen zu sein scheint, ist in der nächsten Sekunde auch schon wieder sauber getrennter Schnee von gestern.

Die Wahrheit ist nämlich: Wir wissen ja schon, dass eine Objektvariable im Grunde genommen nur einen Zeiger auf die eigentlichen Daten im Managed Heap speichert. Und wie wir (Älteren jedenfalls) aus unseren Anfängertagen, in denen es noch 64er und Atari STs in Maschinensprache zu programmieren galt, noch alle wissen, untergliedert sich der Arbeitsspeicher eines Computers in bestimmte Speicherstellen, die alle bestimmte »Hausnummern« (die Speicheradressen) besitzen.

Rekapitulieren wir also das Gelernte aus dem 13. Kapitel: Wenn Sie nun ein Objekt instanziiieren – dabei spielt es gar keine Rolle, ob tatsächlich Object oder eine aus Object Klasse dazu herhält – dann legt das Framework beispielsweise die Daten für diese Objektinstanz an Speicheradresse 460.386 auf dem Managed Heap ab, und die Objektvariable wird zu einer Integervariablen oder (auf 64-Bit-Systemen) zu einer Long-Variablen, die diese Adresse trägt. Hier noch einmal die Grafik, die das verdeutlicht:

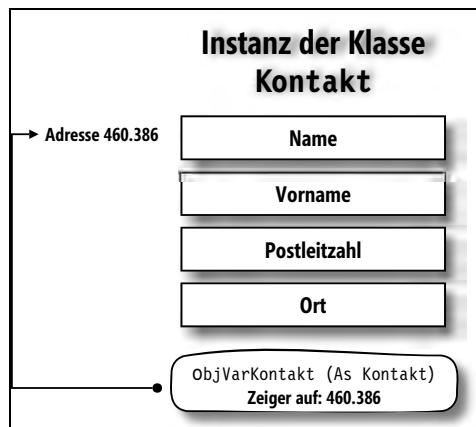


Abbildung 15.8 Objektvariablen speichern im Grunde genommen nur die Speicheradressen auf die eigentlichen Daten, die das Framework im Managed Heap ablegt

Diese Tatsache hat aber entscheidende Folgen, wie das folgende Beispiel gleich zeigen wird.

BEGLEITDATEIEN

Die Begleitdateien zum Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Vererbung04

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Module mdlMain

```
Sub Main()
    'Instanziieren mit New und dadurch
    'Speicher für das Kontakt-Objekt
    'auf dem Managed Heap anlegen
    Dim objVarKontakt As New Kontakt

    'Daten zuordnen
    With objVarKontakt
        .Nachname = "Halek"
        .Vorname = "Sarah"
        .Plz = "99999"
        .Ort = "Musterhausen"
    End With

```

```
'Nur Objektvariable anlegen,  
'es wird aber kein Speicher reserviert!  
Dim objVarKontakt2 As Kontakt  
'objVarKontakt2 "zeigt" ab jetzt auf  
'die selbe Instanz wie objVarKontakt  
objVarKontakt2 = objVarKontakt  
  
'Und das kann man auch beweisen:  
'Das Ändern der Instanz geschieht...  
objVarKontakt2.Nachname = "Löffelmann"  
  
'durch beide Objektvariablen, die natürlich  
'auch dasselbe widerspiegeln.  
Console.WriteLine(objVarKontakt.Nachname)  
  
Console.WriteLine()  
Console.WriteLine("Zum Beenden Taste drücken")  
Console.ReadKey()  
End Sub  
  
End Module  
  
Class Kontakt  
  
    Public Nachname As String  
    Public Vorname As String  
    Public Plz As String  
    Public Ort As String  
  
End Class
```

Wenn Sie dieses Beispiel laufen lassen, erhalten Sie

Löffelmann

Zum Beenden Taste drücken

als Ergebnis. Soweit ist das noch nichts Besonderes, doch bemerkenswert wird es dann, wenn Sie erkennen, dass es nur eine einzige Dateninstanz der Klasse Kontakt in diesem Beispiel gibt, die offensichtlich durch zwei Objektvariablen angesprochen und damit auch wiedergespiegelt wird. Beim Instanziieren wird die Adresse des Speichers, an dem die Daten der Instanz abgelegt werden, in der Objektvariablen objVarKontakt gespeichert. Diese Adresse wird in die Variable objVarKontakt2 übertragen, und wichtig, hierbei wird nicht etwa eine Kopie der gesamten Instanz im Managed Heap angelegt! Eine Objektvariable »zeigt« also quasi auf die Daten, die sie verwaltet. In anderen Programmiersprachen wie C++ gibt es zu diesem Zweck besondere Variablen, die auch als »Zeiger« bezeichnet werden. In Visual Basic wird eine Objektvariable, die die Instanz einer Klasse verwaltet, automatisch zu dem, was man in anderen Programmiersprachen als Zeiger bezeichnet.

In Form einer Grafik sieht das Ganze folgendermaßen aus:

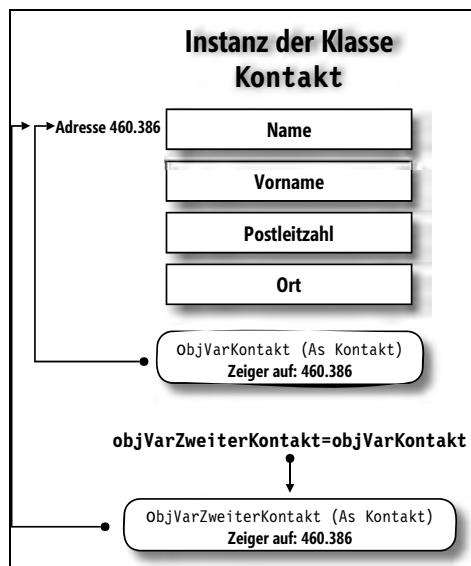


Abbildung 15.9 Das Kopieren einer Objektvariablen in eine andere Objektvariable kopiert nur den Zeiger auf die Instanz – die dann durch beide Variablen manipulierbar und abrufbar wird

Dieses Verhältnis zwischen Objektvariable und eigentlichem Objekt (eigentlicher Instanz) sollten Sie sich gut einprägen, da sie einerseits Quelle schwer zu findender Fehler ist – schließlich kann es auch versehentlich passieren, dass, wenn Sie nicht aufpassen, eine Objektvariable die Instanz eines Objektes verändert, die eigentlich und ausschließlich durch eine ganz andere Objektvariable angesprochen werden sollte. Andererseits kann Ihnen dieses Verhältnis auch zum Vorteil gereichen, nämlich wenn es darum geht, nicht Objekte zu kopieren, sondern nur die Zeiger auf diese – beispielsweise wenn es beim Sortieren großer Objektmen gen auf Geschwindigkeit ankommt, und keine ganzen Speicherblöcke (mit den Daten der eigentlichen Instanzen) sondern nur die Zeiger auf die Objektinstanzen selbst kopiert werden müssten.

Polymorphie

Sie haben nun viel über Vererbung und über das Überschreiben von Klassen-Membern erfahren – jetzt lautet die große Frage, wie Ihnen diese Techniken von Nutzen sein können. Damit Vererbung und Funktionsüberschreibung richtig Sinn ergeben, braucht es eine Technik, die sich »Polymorphie²« nennt. Und würden Sie mich bitten, Polymorphie in einem Satz zu erklären, dann würde ich Sie zunächst bitten, sich festzuhalten und Ihnen anschließend Folgendes zur Antwort geben: »Polymorphie in der OOP beschreibt die Möglichkeit, Methoden oder Eigenschaften in den Klassen einer Klassenerbfolge auszutauschen, und die gleichnamigen aber dennoch funktionell unterschiedlichen Methoden und Eigenschaften der veränderten Klassen der Erbfolge über dieselbe Objektvariable vom Typ der Basisklasse in Abhängigkeit von der tatsächlich instanzierten abgeleiteten Klasse erreichen zu können.« Das klingt sehr abstrakt, und das ist es auch.

² Etwa »vielgestaltig«, »in verschiedenen Formen auftretend«.

Ich habe lange darüber nachgedacht, ein halbwegs plausibles Beispiel für Polymorphie im täglichen Leben zu finden, um zunächst einmal nur das abstrakte Konzept ein wenig zu vereinfachen. Ich gebe zu, dass ich zunächst auch nach stundenlangem Grübeln entnervt aufgab. Doch dann kam mir mein Kollege Jürgen mit einem Zufall zu Hilfe, den ich in meinem Leben so bislang noch nicht erlebt hatte.

Jürgen stand an der Kasse unseres örtlichen Lebensmittelhändlers und war gerade dabei, seine Einkäufe zu bezahlen, als sein Handy schellte, was die freundliche Kassiererin allerdings nicht mitbekam. Da er sein Handy zwischen Schulter und Ohr einklemmte, sah die Kassiererin auch nicht, dass er telefonierte. Und wie es der Zufall wollte, wählte Jürgen, der im Grunde genommen nun mit seiner Freundin sprach, und – eben weil er an der Kasse stand – vergleichsweise kurz angebunden war, seine Worte so, dass sie zufällig aber perfekt auf zwei Gespräche passten, die er dann (eines freiwillig mit seiner Freundin, eines unfreiwillig mit der Kassiererin) auch führte. Die Worte passten sogar so perfekt zu dem, was er zur Kassiererin hätte sagen können, dass sich zwischen ihm und der Kassiererin ein regelrechtes Kurzgespräch entwickelte. Diese Schlagabtausche wiederholten sich einige Male, bis beiden das Missverständnis auffiel, da er die Aussage der Kassiererin »das macht dann also 21,45 Euro« mit der schönen Floskel »ich dich auch« parierte.

Jürgen führte also zwei komplett unabhängige Gespräche, mit einem jeweils absolut anderen Inhalt; Jürgen verwendete aber die exakt gleiche Methode zur »Erlidigung seines Anliegens«. Seine Methode, also die Wahl seiner Worte, war an dieser Stelle also absolut vielgestaltig – eben polymorph –, obwohl die Worte in beiden Gesprächen dieselben waren.

Doch zurück zur Programmierung: Was würden Sie erwarten, wenn Sie im Beispiel von *Vererbung03* die Zeile des Moduls *mdlMain* von

```
Dim klasse2 As New ZweiteKlasse(5)
```

in die Zeilen

```
Dim klasse2 As ErsteKlasse  
klasse2 = New ZweiteKlasse(5)
```

abändern? Glauben Sie, dass Visual Basic einen Fehler auslöst, da Sie *klasse2* als *ErsteKlasse* deklariert, dieser Objektvariablen anschließend aber eine Instanz von zweiter Klasse zugewiesen haben? Mitnichten! Diese Vorgehensweise ist nicht nur kein Fehler, Sie haben gerade sogar das wohl mächtigste Werkzeug der objekt-orientierten Programmierung kennengelernt.

Vererbung und Polymorphie – »Wie, ok!« aber: »Warum?«

In den letzten Abschnitten haben Sie schon einiges über Vererbungstechniken und Polymorphie kennengelernt. Diese Abschnitte haben bereits einige Techniken vorgestellt, wie Sie ein Klasse dazu bekommen, sich polymorph zu verhalten. Und diese Technik anhand von Beispielen zu vermitteln, ist ja so schwierig nicht; auch nicht, die Sachverhalte zu verstehen. Aber was nützt Ihnen das? Was nützt es Ihnen, die Beispiele zu verstehen, und nachvollziehen zu können, wenn aus den Erklärungen nicht wirklich hervorgeht, wie Sie Probleme erkennen können, die danach verlangen, Ihre eigenen Klassenmodelle zu entwickeln. Die größte didaktische Schwierigkeit ist also nicht, zu vermitteln, *wie* Klassenvererbung und Polymorphie funktionieren, sondern vielmehr, *warum* Klassenvererbung und Polymorphie in vielen Szenarien so wichtig ist und Sinn ergeben, da diese Techniken großes Potential haben, Ihre Programme stabiler zu machen und viel, viel einfacher strukturieren können.

Im Prinzip nähern wir uns dem wirklichen Leben beim Beschreiben von Objekten mithilfe der objekt-orientierten Programmierung viel mehr, als es die prozedurale Programmierung jemals könnte. Dummerweise sind aber gerade langjährige Visual Basic-Programmierer geschichtsbedingt viel prozeduraler in ihrer Denke, sodass das in vielen Fällen ihr »Normalzustand« ist; dass es »Klick« macht (und das muss es!) dauert bei Entwicklern mit langem Visual Basic 6-Hintergrund länger und ist schwieriger, als bei Entwicklern, die entweder direkt mit Visual Basic .NET begonnen haben oder von einer wirklich objektorientiert arbeitenden Programmiersprache wie beispielsweise C++ kommen.

Im täglichen Leben haben wir es mit Objekten zu tun. Sie wissen, wenn Sie einen Hund sehen, mit sehr großer Wahrscheinlichkeit, dass es sich um einen Hund handelt, selbst wenn Sie diese Hunderasse noch nie in Ihrem Leben zuvor gesehen haben. Ähnlich ist es beispielsweise mit Behältern. Ihr Geist ist in der Lage, Objekte wie Eimer, Fass, Tetrapak, Glasflasche, Becher, Badewanne, Tank und was es sonst noch an Behältern gibt, zu kategorisieren – übrigens etwas was schon Kant im Rahmen von *transzendentalen Schemata* beschreibt.

Bei der objektorientierten Programmierung es gilt nun, solche Objektzusammenhänge auch bei der Programmierung zu erkennen, übrigens auch oder sogar zum großen Teil mit dem Zweck, viel leichter im Team entwickeln zu können.

Nehmen wir zum Beispiel eine Anwendung wie Microsoft Publisher, Adobe Illustrator oder Corel Draw. Sie alle stellen – vereinfacht ausgedrückt – sehr ähnliche Objekte zum Zeichnen bereit, die sich alle nur in ihrer Ausgestaltung unterscheiden. Sowohl eine Linie, ein Kreis, ein Rechteck, eine Ellipse, ein Pentagon oder eine Grafik lassen sich auf einen gemeinsamen funktionellen Nenner bringen:

- Sie müssen an einer bestimmten Stelle auf dem Papier bzw. in einem Dokumentenfenster platziert werden können, haben also Positionen und Ausmaße, die durch zwei Koordinaten (x_1, y_1, x_2, y_2) definiert werden.
- Sie müssen sich selber rendern können, also die Möglichkeit haben, sich in einen bestimmten grafischen Kontext zeichnen zu können (Drucker, Bildschirm, Plotter, etc.).
- Sie müssen bestimmte Eigenschaften haben, mit denen näher beschrieben wird, *wie* sie sich zeichnen sollen, also beispielsweise mit welcher Strichstärke, mit welcher Farbe, etc.

Eine solche Anwendung objektorientiert zu verfassen, drängt sich gerade zu auf. Man implementiert eine Basisklasse, die all diese Grundfähigkeiten zur Verfügung stellt. Diese Basisklasse könnte man nun beispielsweise `GraphicsObject` nennen. All die Methoden, wie beispielsweise das Zeichnen selbst, können natürlich nicht in dieser Basisklasse implementiert werden – denn `GraphicsObject` selbst ist ja nur der größte gemeinsame Nenner, zu dem das konkrete Zeichnen des Objektes aber nicht gehört.

Was soll `GraphicsObject` zum Zeitpunkt seiner Definition auch Zeichnen? Es kann aber schon zu diesem Zeitpunkt beispielsweise Eigenschaften-Code wie das Verwalten von Farben oder Ausmaßen konkret implementieren – denn diese Eigenschaften sollen ja auch *alle* Objekte aufweisen, die sich später aus diesem `GraphicsObject` ableiten.

Erst wenn es dann darum geht, die konkrete Implementierung der wirklichen Objekte zu programmieren, kann auch das eigentliche Rendern in den abgeleiteten Klassen implementiert werden. Ein `Circle`-Objekt malt dann beispielsweise einen Kreis, ein `Line`-Objekt eine Linie und ein `Rectangle`-Objekt eben ein Rechteck.

Und wieso ist die objektorientierte Implementierung dieses Problems nun so teamfreundlich? Ganz einfach: Zwei Grundanforderungen der Anwendung können gleichzeitig von Teams implementiert werden, ohne dass sie zeitlich von einander abhängig sind. Nachdem die Basisklassenimplementierung von `GraphicsObject` abgeschlossen wurde, kann sich Team A daran machen, die Objektverwaltung zu implementieren. Team A ist es dabei völlig egal, was eine Objektvariable vom Typ `GraphicsObject` tatsächlich malt. Team A verwaltet nur eine Liste dieses Typs und sorgt dafür, dass an den schon durch `GraphicsObject` definierten Positionen und in den entsprechenden Farben und Linienstärke die Objekte gemalt werden. Zum Testen benötigt Team A bestenfalls noch ein einziges konkret implementiertes Objekt. Doch was Team A beim Durchlaufen der Liste dann letzten Endes zeichnet, kann Team A völlig egal sein. Zwar ruft Team A die `Render`-Methode beim Zeichnen des Objektes von `GraphicsObject` auf, aber die Liste von `GraphicsObject` sind ja in Wirklichkeit keine `GraphicsObject`-Objekte, sondern andere, die Team B nacheinander konkretisiert. Nur eine Objektvariable vom Typ der Basisklasse wird also verwendet, um das Objekt zu zeichnen, was sich in der Liste wirklich hinter der Basisklassenobjektvariable verbirgt. Team B konkretisiert also von Team A unabhängig, *was* alles gezeichnet werden kann. Während Team A zwar die `Render`-Methode aufruft, hat Team B ja in den abgeleiteten konkreten Implementierungen der Klassen genau diese überschrieben, und dann passiert angewandte Polymorphie: Trotz gleichem Aufrufs passiert etwas anderes – im Beispiel wird halt je nach wirklichem Objekttyp in der Liste etwas anderes gezeichnet (Linie, Kreis, Rechteck, etc.). So programmiert Team A mithilfe des entworfenen Objektmodells eine Funktionalität, dessen wirkliches Ergebnis Team A weder kennt noch – und das ist das Entscheidende – auch gar nicht kennen muss!

Im Vergleich zur prozeduralen Programmierung gibt es hier darüber hinaus noch einen weiteren Vorteil: Bei der prozeduralen Programmierung ist das »Vererben« von »Objekten« in Form von Code-Copy/Paste an der Tagesordnung gewesen: Wenn die Routine zum Zeichnen des Kreises mit allen Attributen stand, dann hat man (leider) vielfach den Code kopiert und anschließend daraus die Prozedur für das Linienzeichnen gebastelt. Wurde dann ein Fehler in der ersten `Circle`-Routine festgestellt, musste dieser Fehler natürlich auch in allen »abgeleiteten« (da kopierten) Codezeilen behoben werden. Natürlich führte das oft zu Fehlern, die ihren Weg bis in die Endversion der Anwendung nahmen, denn bei vielleicht 20 oder 30 Prozeduren, die so zu korrigieren waren, wurde vielleicht schon mal die eine oder andere vergessen.

Bei der objektorientierten Vorgehensweise kann das nicht passieren. Ein Fehler in der Basisklasse `GraphicsObject` wird dort behoben; er war aber auch nur ein einziges Mal implementiert. Das bedeutet: Alle abgeleiteten Klasse erben nicht nur die Grundfunktionsweise der Basisklasse, sondern natürlich auch alle Fehlerkorrekturen an ihr. Stabilere Programme lassen sich deswegen mithilfe der objektorientierten Entwicklungsweise viel eher garantieren, als es mit der prozeduralen Entwicklung möglich wäre.

Eine Gefahr bei der Entwicklung von Objektmodellen besteht allerdings, und umso mehr, wenn vormalige und langjährige Entwickler der prozeduralen Entwicklung sich ihrer erstmalig bedienen: Sie sind oftmals so fasziniert von den neuen Möglichkeiten der Vererbung und Polymorphie, dass sie beginnen, die unglaublichesten Objektmodelle zu entwerfen und dabei häufig den Fokus auf das verlieren, was eigentlich benötigt wird: Sie legen das Fundament für einen 100-stöckigen Wolkenkratzer, obwohl sie dann nur ein Einfamilienhaus darauf setzen sollen. Eine Kontrollinstanz in Form eines Projektleiters, die dafür sorgt, dass sich ein Objektmodell nicht in Selbstverliebtheit seine OOP-Features verliert, ist deswegen oft von großer Bedeutung!

Polymorphie in der Praxis

Ein neues Corel-Draw im Rahmen dieses Buches nur als Paradebeispiel für Polymorphie zu entwerfen, wäre vielleicht zu ambitioniert. Dennoch soll ein etwas größeres Praxisbeispiel zur anschaulichen Demonstration der Polymorphie dienen. Stellen Sie sich vor, Sie arbeiten in der EDV-Abteilung eines größeren Versandhauses. Ihre Aufgabe besteht darin, eine Software zu entwickeln, die Textlisten mit Artikeln verarbeitet, sie formatiert und anschließend den Summenwert ausgibt. Dabei können die verarbeiteten Artikel im Programm Datentypen mit ganz unterschiedlichen Eigenschaften sein: Video- und DVD-Artikeldatentypen speichern neben dem Titel auch die Laufzeit und den Hauptdarsteller; Bücher-Artikeldatentypen speichern stattdessen den Autor und haben obendrein einen anderen Mehrwertsteuersatz.

Ohne Polymorphie wäre die Erstellung ein vergleichsweise schwieriges Unterfangen. Sie müssten eine Art »Superset« schaffen, das alle Eigenschaften beherrscht und für die jeweils geforderten Sonderfälle programmieren. Regelrechter Spaghetti-Code wäre dabei buchstäblich vorprogrammiert.

Die Listen, die in simplen Textdateien vorliegen, sollen in diesem Beispiel ein bestimmtes Format ausweisen – Ihr Hauptprogramm muss dieses Format berücksichtigen und intern die Daten entsprechend speichern. Eine Liste, wie Sie sie von anderen Mitarbeitern Ihrer Abteilung als Textdatei bekommen, sieht typischerweise wie folgt aus:

```
;Inventarliste, die durch das Programm Inventory ausgewertet werden kann
;Format:
;Typ (1=Buch, 2=Cd oder Video), Bestellnummer, Titel, Zusatz, Brutto in Cent, Zusatz2
1;0001;Die Nachwächter;Terry Pratchett und Andreas Brandhorst;1990;
1;0002;Kristall der Träume;Barbara Wood;2490;
1;0003;Volle Deckung - Mr. Bush: Dude where is my country;Michael Moore;1290;
1;0004;Du bist nie allein;Nicholas Sparks und Ulrike Thiesmeyer;1900;
2;0005;X-Men 2 - Special Edition;128;2299;Patrik Steward
2;0006;Sex and the City: Season 5;220;2999;Sarah Jessica Parker
2;0007;Indiana Jones (Box Set 4 DVDs);359;4499;Harrison Ford
2;0008;Die Akte;135;1499;Julia Roberts
```

Sie sehen: Die Liste enthält verschiedene Artikeltypen, die berücksichtigt werden müssen. Ihr Programm muss die einzelnen Artikel der Liste verarbeiten und sollte anschließend eine neue Liste mit folgendem Format ausspucken:

0001	Die Nachwächter		
18,60 Euro	1,30 Euro	19,90 Euro	
0002	Kristall der Träume		
23,27 Euro	1,63 Euro	24,90 Euro	
0003	Volle Deckung - Mr. Bush: Dude where is my country		
12,06 Euro	0,84 Euro	12,90 Euro	
0004	Du bist nie allein		
17,76 Euro	1,24 Euro	19,00 Euro	
0005	X-Men 2 - Special Edition		
19,82 Euro	3,67 Euro	22,99 Euro	

0006 Sex and the City: Season 5
25,85 Euro 4,79 Euro 29,99 Euro

0007 Indiana Jones (Box Set 4 DVDs)
38,78 Euro 7,18 Euro 44,99 Euro

0008 Die Akte
12,92 Euro 2,39 Euro 14,99 Euro

Gesamtsumme: 189,66 Euro

Die Artikeltypen unterscheiden sich dabei in zweierlei Hinsicht: Zum einen gibt es bei Büchern einen anderen Mehrwertsteuersatz. Zum anderen sind es bei Büchern die Autoren, die in die Artikelinfo einlaufen, bei Filmen ist es die Lauflänge. Das Programm muss obendrein die Möglichkeit bieten, kurze und ausführliche Listen zu erstellen. In der ausführlichen Liste sollen dann die erweiterten Eigenschaften der einzelnen Artikeltypen zu sehen sein. Eine ausführliche Liste sieht folgendermaßen aus:

0001 Die Nachwächter
Autor: Terry Pratchett und Andreas Brandhorst
18,60 Euro 1,30 Euro 19,90 Euro

0002 Kristall der Träume
Autor: Barbara Wood
23,27 Euro 1,63 Euro 24,90 Euro

0003 Volle Deckung - Mr. Bush: Dude where is my country
Autor: Michael Moore
12,06 Euro 0,84 Euro 12,90 Euro

0004 Du bist nie allein
Autor: Nicholas Sparks und Ulrike Thiesmeyer
17,76 Euro 1,24 Euro 19,00 Euro

0005 X-Men 2 - Special Edition
Laufzeit: 128 Min.
Hauptdarsteller: Patrik Steward
19,82 Euro 3,67 Euro 22,99 Euro

0006 Sex and the City: Season 5
Laufzeit: 220 Min.
Hauptdarsteller: Sarah Jessica Parker
25,85 Euro 4,79 Euro 29,99 Euro

0007 Indiana Jones (Box Set 4 DVDs)
Laufzeit: 359 Min.
Hauptdarsteller: Harrison Ford
38,78 Euro 7,18 Euro 44,99 Euro

0008 Die Akte
 Laufzeit: 135 Min.
 Hauptdarsteller: Julia Roberts
 12,92 Euro 2,39 Euro 14,99 Euro

 Gesamtsumme: 189,66 Euro

Zur Programmplanung: Da Sie vorher (also während der Entwurfszeit Ihres Programms) nicht wissen, wie viele Artikel Ihnen eine Datei zur Verfügung stellt, müssen Sie den Artikelspeicher dynamisch gestalten. Dazu gibt es zwei Möglichkeiten:

- Sie lesen die Datei komplett von vorne bis hinten durch und finden, noch bevor Sie einen Artikel verarbeitet haben, heraus, wie viele Artikel Sie verarbeiten müssen. Dann dimensionieren Sie ein Array in der Größe, die der Anzahl der Artikel entspricht, die Sie ja jetzt kennen. Anschließend lesen Sie in einem zweiten Durchgang die Artikel in das Array ein.
- Oder: Sie schaffen eine Klasse zum Speichern der Artikel, die sich dynamisch vergrößert. Die Klasse selbst könnte beispielsweise ein Array vordefinieren, mit einer Initialgröße von beispielsweise 4 Elementen. Wenn diese 4 Elemente nicht mehr ausreichen, legt sie ein neues temporäres Array an, dann beispielsweise mit 8 Elementen, kopiert die vorhandenen 4 Elemente des »alten« Arrays in das neue und tauscht anschließend die beiden Arrays aus, sodass das alte Member-Array der Klasse nun Platz für 8 Elemente hat. Das Kopieren der Arrayelemente von einem zum anderen Array scheint nur auf den ersten Blick zeitintensiv – dieser Vorgang kopiert, wie wir im vorherigen Abschnitt kennen gelernt haben, aber nur die Zeiger auf die eigentlichen Artikeldaten, und das absolvieren moderne 32- und 64-Bit-Prozessoren in Lichtgeschwindigkeit.

Würden Sie sich für die erste Option entscheiden, müsste die zu importierende Datei ein zweites Mal verarbeitet werden, was gerade bei größeren Dateien natürlich viel zeitintensiver ist.

Die Artikel selbst werden ebenfalls in Klassen gespeichert. Da beide Artikel viele Gemeinsamkeiten aufweisen und sich nur marginal voneinander unterscheiden, liegt es nahe, eine so genannte Basisklasse zu entwerfen, die die Gemeinsamkeiten abdeckt und die Sonderfälle der jeweiligen Artikeltypen in zwei davon abgeleiteten Klassen zu implementieren. Wichtig für die Typsicherheit: Die »Listenklasse«, die die einzelnen Artikeldatentypen speichert, sollte nur Artikelklassen und von ihr abgeleitete Klassen aufnehmen.

Polymorphie ist in unserem Beispiel von unschätzbarem Wert. Die Artikelklassen, die von der Artikelbasisklasse abgeleitet sind, lassen sich nämlich über eine Objektvariable der Artikelbasisklasse steuern. Für die Entwicklung bedeutet das: Sie können ein Array verwalten, das nur den Typ »Artikelbasisklasse« aufnimmt, obwohl ganz andere (na ja, vielleicht nicht *ganz* andere, sondern nur erweiterte) Klasseninstanzen darin gespeichert werden. Der entscheidende Clou dabei ist, dass Sie im Code zwar beispielsweise eine Methode der Artikelbasisklasse für das Ausführen einer bestimmten Funktion angeben, dann aber doch die (andere) Methode einer abgeleiteten Klasse aufgerufen wird, wenn diese in der Basisklasse überschrieben wurde.

BEGLEITDATEIEN Das mag zunächst ein wenig verwirrend klingen, wird aber klar, wenn Sie das Beispielprogramm zur Hilfe nehmen, das Sie unter dem Projektmappen-Namen *Inventory01* im folgenden Verzeichnis finden:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Inventory01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Doch bevor wir uns mit dem Polymorphieaspekt dieses Beispiels beschäftigen, lassen Sie uns zunächst die Techniken klären, mit denen die Voraussetzungen zum Speichern der Daten geschaffen werden. Betrachten Sie dazu als erstes die Klasse DynamicList zur Speicherung der Artikel:

```
Class DynamicList

    Protected myStep As Integer = 4           ' Schrittweite, die das Array erhöht wird.
    Protected myCurrentArraySize As Integer   ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer     ' Zeiger auf aktuelles Element
    Protected myArray() As ShopItem           ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub

    Sub Add(ByVal Item As ShopItem)

        'Element im Array speichern
        myArray(myCurrentCounter) = Item

        'Zeiger auf nächstes Element erhöhen
        myCurrentCounter += 1

        'Prüfen, ob aktuelle Arraygrenze erreicht wurde
        If myCurrentCounter = myCurrentArraySize - 1 Then
            'Neues Array mit mehr Speicher anlegen,
            'und Elemente hinüberkopieren. Dazu:

            'Neues Array wird größer:
            myCurrentArraySize += myStep

            'temporäres Array erstellen
            Dim locTempArray(myCurrentArraySize - 1) As ShopItem

            'Elemente kopieren; das geht mit dieser
            'statischen Methode extrem schnell, da zum Einen nur die
            'Zeiger kopiert werden, zum anderen diese Routine
            'intern nicht in Managed Code sondern nativem Assembler ausgeführt wird.
            Array.Copy(myArray, locTempArray, myArray.Length)

            'Auch hier werden nur die Zeiger auf die Elemente "verbogen".
            'Die vorherige Liste der Zeiger in myArray, die nun verwaist ist,
            'fällt dem Garbage Collector zum Opfer.
            myArray = locTempArray
        End If
    End Sub

    'Liefert die Anzahl der vorhandenen Elemente zurück
    Public Overridable ReadOnly Property Count() As Integer
        Get
            Return myCurrentCounter
        End Get
    End Property
```

```
'Erlaubt das Zuweisen
Default Public Overridable Property Item(ByVal Index As Integer) As ShopItem
    Get
        Return myArray(Index)
    End Get

    Set(ByVal Value As ShopItem)
        myArray(Index) = Value
    End Set
End Property
End Class
```

Die Add-Methode in dieser Klasse ist das Entscheidende. Sie überprüft, ob das Array noch ausreichend groß ist. Falls das nicht der Fall ist, führt sie den Tauschvorgang mit einer lokalen Arrayvariablen durch, die entsprechend größer definiert wurde, und in die die Arrayelemente zuvor kopiert wurden. Damit »zeigt« der Klassen-Member myArray jetzt auf die neuen Arrayelemente – die Zeiger auf die alten Array-Elemente werden buchstäblich »vergessen«. Übrigens: Den Arbeitsspeicher, der auf diese Weise unnötigerweise belegt wird, holt sich das Framework eigenständig mithilfe der so genannten Garbage Collection (»Müllabfuhr«) wieder. Der Garbage Collector (kurz »GC«) schaut sich – vereinfacht ausgedrückt – an, ob Objekte, die Speicher belegen, noch irgendeinen Bezug zu einer verwendeten Objektvariablen haben. Falls nicht, werden sie entsorgt. In diesem Beispiel haben die Zeiger auf die ursprünglichen Elemente, die im Member-Array myArray gespeichert waren, nach dem Kopieren keinen Bezug mehr zu irgendeinem Objekt: Der Objektname wurde mit der Zeile

```
'temporäres Array dem Memberarray zuweisen
myArray = locTempArray
```

auf die neuen Elemente »umgebogen«. Die ursprünglichen Elemente stehen anschließend bezugslos im Speicher und werden beim nächsten GC-Durchlauf entsorgt.

WICHTIG Behalten Sie im Hinterkopf, dass die Daten der eigentlichen Objektinstanzen bei diesem Vorgang überhaupt nicht angetastet werden. Und das ist auch der Grund, weswegen das Kopieren eines Arrays mit `Array.Copy` so unglaublich schnell vorstatten geht (was ebenfalls der Fall wäre, würden Sie den Vorgang selbst in die Hand nehmen, das ganze Array in einer For-Schleife durchlaufen, und die einzelnen Elemente dem neuen Array zuweisen).

Die Add-Methode nimmt ausschließlich Objekte eines bestimmten Typs entgegen. In diesem Beispiel habe ich sie `ShopItem` (»Ladenartikel«) genannt. Diese Klasse stellt die Basis für die Artikelspeicherung dar und sieht folgendermaßen aus:

```
Class ShopItem

    Protected myTitle As String          ' Titel
    Protected myNetPrice As Double       ' Nettopreis
    Protected myOrderNumber As String    ' Artikelnummer
    Protected myPrintTypeSetting As PrintType ' Ausgabeform

    Public Sub New()
        myPrintTypeSetting = PrintType.Detailed
    End Sub
```

```
Public Sub New(ByVal StringArray() As String)
    Title = StringArray(FieldOrder.Title)
    'FieldOrder ist übrigens eine Enum und wird weiter unten definiert:
    GrossPrice = Double.Parse(StringArray(FieldOrder.GrossPrice)) / 100
    OrderNumber = StringArray(FieldOrder.OrderNumber)
    PrintTypeSetting = PrintType.Detailed
End Sub

Public Property Title() As String
    Get
        Return myTitle
    End Get
    Set(ByVal Value As String)
        myTitle = Value
    End Set
End Property

Public Property OrderNumber() As String
    Get
        Return myOrderNumber
    End Get
    Set(ByVal Value As String)
        myOrderNumber = Value
    End Set
End Property

Public Property NetPrice() As Double
    Get
        Return myNetPrice
    End Get
    Set(ByVal Value As Double)
        myNetPrice = Value
    End Set
End Property

Public ReadOnly Property NetPriceFormatted() As String
    Get
        Return NetPrice.ToString("#,##0.00") + " Euro"
    End Get
End Property

Public Overridable Property GrossPrice() As Double
    Get
        Return myNetPrice * 1.19
    End Get
    Set(ByVal Value As Double)
        myNetPrice = Value / 1.19
    End Set
End Property
```

```

Public ReadOnly Property GrossPriceFormatted() As String
    Get
        Return GrossPrice.ToString("#,##0.00") + " Euro"
    End Get
End Property

Public ReadOnly Property VATAmountFormatted() As String
    Get
        Return (GrossPrice - myNetPrice).ToString("#,##0.00") + " Euro"
    End Get
End Property

Public Overridable ReadOnly Property Description() As String
    Get
        Return OrderNumber & vbTab & Title
    End Get
End Property

Public Property PrintTypeSetting() As PrintType
    Get
        Return myPrintTypeSetting
    End Get

    Set(ByVal Value As PrintType)
        myPrintTypeSetting = Value
    End Set
End Property

Public Overrides Function ToString() As String

    If PrintTypeSetting = PrintType.Brief Then
        'Kurzform: Es wird in jedem Fall
        'die Description-Eigenschaft des Objektes
        'verwendet
        Return MyClass.Description & vbCrLf &
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCrLf & vbCrLf
    Else
        'Langform: Die Description Eigenschaft des Objektes
        'selber wird verwendet
        Return Me.Description & vbCrLf & vbCrLf &
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCrLf & vbCrLf
    End If
End Function

End Class

```

Es ist nicht sonderlich schwer, diese Klasse zu begreifen, denn sie dient nur zwei Aufgaben: Dem Speichern von Daten, die durch Eigenschaften zugänglich gemacht werden und der formatierten Ausgabe einiger dieser Daten. Einige der Eigenschaften sind obendrein nur als `ReadOnly` definiert, da es keinen Sinn ergäbe, sie zu beschreiben.

Zahlen mit ToString formatiert in Zeichenketten umwandeln

Erwähnenswert an dieser Stelle ist die Eigenschaft der `ToString`-Funktion primitiver Datentypen (`Integer`, `Double`, etc.), Zahlen formatiert auszugeben. In diesem Fall verwenden Sie eine Überladung von `ToString`, die einen String als Parameter akzeptiert. In diesem Beispiel lautet der String »#,##0.00«, der bewirkt, dass Nachkommastellen unabhängig vom Wert grundsätzlich zweistellig, Vorkommastellen im Bedarfsfall mit Tausender trennzeichen formatiert werden. Bitte beachten Sie, dass Sie hierbei die amerikanische/englische Schreibweise verwenden, bei der das Tausender trennzeichen ein Komma und das Nachstellentrennzeichen ein Punkt ist. Mehr zu diesem Thema finden Sie übrigens in Kapitel 26.

Interessant ist es, als nächstes die aus der Basisklasse abgeleiteten Klassen zu erkunden, die Sie im Folgenden abgedruckt finden:

```
Class BookItem
    Inherits ShopItem

    Protected myAuthor As String

    Public Sub New(ByVal StringArray() As String)
        MyBase.New(StringArray)
        Author = StringArray(FieldOrder.AdditionalRemarks1)
    End Sub

    Public Overridable Property Author() As String
        Get
            Return myAuthor
        End Get
        Set(ByVal Value As String)
            myAuthor = Value
        End Set
    End Property

    Public Overrides Property GrossPrice() As Double
        Get
            Return myNetPrice * 1.07
        End Get

        Set(ByVal Value As Double)
            myNetPrice = Value / 1.07
        End Set
    End Property

    Public Overrides ReadOnly Property Description() As String
        Get
            Return OrderNumber & vbTab & Title & vbCr & vbLf & "Autor: " & Author
        End Get
    End Property

End Class
```

Durch das `Inherits`-Schlüsselwort direkt nach dem `Class`-Schlüsselwort bestimmen Sie, dass die neue Klasse, die die Bücherartikel speichert, von der Basisklasse `ShopItem` abgeleitet wird. Das befähigt Objektvariablen, die als `ShopItem` definiert wurden, auch Instanzen von `BookItem` zu referenzieren, denn es gilt: Jede abgeleitete Klasse kann durch eine Objektvariable der Basisklasse »angesprochen« werden.

Dazu ein Beispiel: Wenn Sie eine Objektvariable namens `EinArtikel` auf folgende Weise deklariert

```
Dim EinArtikel as ShopItem
```

und entsprechend, etwa durch

```
EinArtikel=New ShopItem()
```

definiert haben, kann ihr Inhalt später im Programm durch die Anweisung

```
Console.WriteLine(EinArtikel.GrossPriceFormatted)
```

ausgegeben werden, und es wird, wie zu erwarten, `GrossPriceFormatted` (etwa: »Bruttopreis formatiert«) der Basisklasse `ShopItem` verwendet.

Wird hingegen – und jetzt wird es interessant – eine Instanz der abgeleiteten Klasse etwa durch die folgende Zeile

```
EinArtikel=New BookItem(...)
```

durch **dieselbe** Objektvariable `EinArtikel` angesprochen, und wird später die gleiche Funktion aufgerufen, etwa durch

```
Console.WriteLine(EinArtikel.GrossPriceFormatted)
```

so wird dieses Mal nicht die `GrossPriceFormatted`-Funktion der Basisklasse, sondern die der abgeleiteten Klasse `BookItem` verwendet. Und genau das sind die unschlagbaren Vorteile der Polymorphie: Sie haben ein Steuerprogramm, das in einer Basisklassenobjektvariablen die augenscheinlich gleiche Funktion aufruft, und doch können Sie durch das Überschreiben genau dieser Funktion in einer abgeleiteten Klasse ein anderes Ergebnis erzielen.

Die Basisklasse unseres Beispiels stellt einen parametrisierten Konstruktor bereit, die ein String-Array übernimmt. Aus diesem String-Array werden die Daten für den Inhalt einer Klasseninstanz entnommen:

```
Public Sub New(ByVal StringArray() As String)
    Title = StringArray(FieldOrder.Titel)
    GrossPrice = Double.Parse(StringArray(FieldOrder.GrossPrice)) / 100
    OrderNumber = StringArray(FieldOrder.OrderNumber)
    PrintTypeSetting = PrintType.Detailed
End Sub
```

Das übergebende Array enthält die einzelnen Daten immer in Elementen mit dem gleichen Index, die zum einfacheren Verständnis des Quellcodes übrigens in einer `Enum`-Aufzählung festgehalten sind (mehr zum Thema *Enums* erfahren Sie in Kapitel 21):

```
Enum FieldOrder ' Zuständig für die Reihenfolge der Felder
    Type
    OrderNumber
    Titel
    AdditionalRemarks1
    GrossPrice
    AdditionalRemarks2
End Enum
```

Die abgeleitete Klasse `BookItem` hat nun viel weniger Arbeit, eine Instanz zu definieren. Sie ruft einfach den Konstruktor der Basisklasse auf und ergänzt ihren eigenen Konstruktor nur noch um die Zuweisung eines bestimmten Array-Elementes des ihr übergebenen Parameters:

```
Public Sub New(ByVal StringArray() As String)
    MyBase.New(StringArray)
    Author = StringArray(FieldOrder.AdditionalRemarks1)
End Sub
```

Da Bücher im Gegensatz zu vielen anderen Artikeln mit einer anderen Mehrwertsteuer belegt sind (jedenfalls noch), muss die Funktion, die den Bruttopreis berechnet, überschrieben werden:

```
Public Overrides Property GrossPrice() As Double
    Get
        Return myNetPrice * 1.07
    End Get

    Set(ByVal Value As Double)
        myNetPrice = Value / 1.07
    End Set
End Property
```

Und schon wieder sehen Sie Polymorphie in Aktion: Die abgeleitete Klasse muss jetzt die Methode, die den formatierten Bruttopreis als String zurückgibt, nicht noch zusätzlich überschreiben, denn wenn Sie sich die Funktion der Basisklasse betrachten

```
Public ReadOnly Property GrossPriceFormatted() As String
    Get
        Return GrossPrice.ToString("#,##0.00") + " Euro"
    End Get
End Property
```

stellen Sie fest, dass sie nicht direkt auf eine Member-Variable zurückgreift, sondern ihrerseits eine in der Klasse implementierte Eigenschaft bemüht.

Aus der Sicht der abgeleiteten Klasse BookItem wird hier nicht GrossPrice der Basisklasse, sondern der (über-schriebenen) abgeleiteten Klasse aufgerufen – der formatierte Bruttoreis eines Buchs berücksichtigt, also korrekt 7% Mehrwertsteuer und nicht die 19% der Basisklasse. GrossPriceFormatted ruft also die überschrie-bene Funktion der abgeleiteten Klasse auf, obwohl diese Funktion ausschließlich in der Basisklasse zu finden ist!

Eine ähnliche Vorgehensweise legen die Funktionen Description (für die Zusammensetzung des Beschrei-bungstextes) und VATAmountFormatted (für den Betrag der Mehrwertsteuer) an den Tag.

Und auch die zweite Klasse zur Speicherung von Artikeln – sie nennt sich DVDItem – macht sich die Poly-morphie zunutze:

```
Class DVDItem
    Inherits ShopItem

    Protected myRunningTime As Integer
    Protected myActor As String

    Public Sub New(ByVal StringArray() As String)
        MyBase.New(StringArray)
        RunningTime = Integer.Parse(StringArray(FieldOrder.AdditionalRemarks1))
        Actor = StringArray(FieldOrder.AdditionalRemarks2)
    End Sub

    Public Overridable Property RunningTime() As Integer
        Get
            Return myRunningTime
        End Get
        Set(ByVal Value As Integer)
            myRunningTime = Value
        End Set
    End Property

    Public Overridable Property Actor() As String
        Get
            Return myActor
        End Get
        Set(ByVal Value As String)
            myActor = Value
        End Set
    End Property

    Public Overrides ReadOnly Property Description() As String
        Get
            Return OrderNumber & vbTab & Title & vbCrLf & "Laufzeit: " & myRunningTime & " Min." &
                vbCrLf & vbCrLf & "Hauptdarsteller: " & Actor
        End Get
    End Property

End Class
```

Sie sehen an diesem Beispiel, wie sehr Sie die Polymorphie bei der Wiederverwendbarkeit von Code unter-stützt und enorm Arbeit spart. Und das gleich in doppeltem Sinne: Wenn die Basisklasse funktioniert,

funktionieren die abgeleiteten Klassen bis auf ihre zusätzliche Funktionalität genau so reibungslos. Neben dem weniger vorhandenen Tippaufwand sparen Sie obendrein viel Zeit beim Suchen von Fehlern.

Die Klassen für unser Beispiel sind damit komplett fertig gestellt. Was jetzt noch zu tun bleibt, ist die Implementierung des Hauptprogramms, das die Ursprungstextdatei einliest, die Elemente aus jeder Textzeile erstellt, sie der Liste hinzufügt und die Ergebnisse schließlich ausgibt:

```
Module mdlMain

    'Die Inventardatei muss im Programmverzeichnis stehen.
    Private Filename As String = My.Application.Info.DirectoryPath & "\Inventar.txt"

    Sub Main()

        'StreamReader zum Einlesen der Textdateien
        Dim locSr As StreamReader
        Dim locList As New DynamicList ' Die Dynamische Liste
        Dim locElements() As String      ' Die einzelnen ShopItem-Elemente
        Dim locShopItem As ShopItem     ' Ein einzelnes Shop-Element
        Dim locDisplayMode As PrintType ' Der Darstellungsmodus

        'Schauen, ob die Textdatei vorhanden ist:
        'Der Einfachheit halber einfach hier mögliche Fehler abfangen ... sorry, Herr Fachlektor! :-)
        Try
            locSr = New StreamReader(Filename, System.Text.Encoding.Default)
        Catch ex As Exception
            Console.WriteLine("Fehler beim Lesen der Inventardatei!" & _
                vbCrLf & ex.Message)
            Console.WriteLine()
            Console.WriteLine("Taste drücken zum Beenden")
            Console.ReadKey()
            Exit Sub
        End Try

        Console.WriteLine("Wählen Sie (1) für kurze und (2) für ausführliche Darstellung")
        Dim locKey As Char = Console.ReadKey.KeyChar
        If locKey = "1"c Then
            locDisplayMode = PrintType.Brief
        Else
            locDisplayMode = PrintType.Detailed
        End If

        Do
            Try
                'Zeile einlesen
                Dim locLine As String = locSr.ReadLine()

                'Nichts eingegeben, dann war's das!
                If String.IsNullOrEmpty(locLine) Then
                    locSr.Close()
                    Exit Do
                End If

                'Semicolon überlesen
                If Not locLine.StartsWith ";" Then
```

```

'So braucht man kein explizites Char-Array zu deklarieren
'um die Zeile in die durch Komma getrennten Elemente zu zerlegen
locElements = locLine.Split(New Char() {";"c})

If locElements(FieldOrder.Type) = "1" Then
    locShopItem = New BookItem(locElements)
Else
    locShopItem = New DVDItem(locElements)
End If
locList.Add(locShopItem)
End If

Catch ex As Exception
    Console.WriteLine("Fehler beim Auswerten der Inventardatei!" & _
                      vbCrLf & ex.Message)
    Console.WriteLine()
    Console.WriteLine("Taste drücken zum Beenden")
    Console.ReadKey()
    locSr.Close()
    Exit Sub
End Try

Loop

Dim locGrossAmount As Double = 0

'Alle Elemente ausgeben
For count As Integer = 0 To locList.Count - 1
    locList(count).PrintTypeSetting = locDisplayMode
    Console.WriteLine(locList(count).ToString())
    locGrossAmount += locList(count).GrossPrice
Next

Console.WriteLine()
Console.WriteLine("-----")
Console.WriteLine("Gesamtsumme: " & locGrossAmount.ToString("#,##0.00") & " Euro")
Console.WriteLine("-----")
Console.WriteLine("-----")
Console.WriteLine()
Console.WriteLine("Return drücken, zum Beenden")
Console.ReadLine()

End Sub

End Module

```

Sie sehen: Den meisten Platz beanspruchen die Print-Zeilen, die sich um die Ausgaben kümmern und die Artikel auf dem Bildschirm darstellen. Da die überwiegende Lösung des Problems bereits von den Artikelklassen bewältigt wird, beschränkt sich die eigentliche Aufgabe des Hauptprogramms darauf, die Datei zu öffnen, ein Element aus der Klasse zu lesen und zu speichern.

Auch das Hauptprogramm bedient sich der Polymorphie, nämlich dann, wenn es die Liste auf dem Bildschirm ausgibt:

```
'Alle Elemente ausgeben
For count As Integer = 0 To locList.Count - 1
    locList(count).PrintTypeSetting = locDisplayMode
    Console.WriteLine(locList(count).ToString())
    locGrossAmount += locList(count).GrossPrice
Next
```

Es iteriert in einer Zählschleife durch die einzelnen Elemente der Klasse. Da die `Item`-Eigenschaft der `DynamicList`-Klasse die Default-Eigenschaft ist, muss der Eigenschaftenname `Item` an dieser Stelle noch nicht einmal angegeben werden – es reicht aus, das gewünschte Element durch eine direkt an den Objektnamen angefügte Klammer zu indizieren.

Mit `ToString` des indizierten Objektes erfolgt anschließend der Ausdruck auf dem Bildschirm. Welches `ToString` das Programm dabei verwendet, ist wieder abhängig von der Klasse, deren Instanz im Arrayelement gespeichert wird. Ist es ein `BookItem`-Objekt, wird `ToString` von `BookItem` aufgerufen; bei einem `DVDItem`-Objekt wird die `ToString`-Funktion eben dieser Klasse aufgerufen. Gleiches gilt anschließend für die Berechnung der Mehrwertsteuer und die Funktion `GrossPrice`.

Polymorphie und der Gebrauch von Me, MyClass und MyBase

Die Option, dem Benutzer entweder eine ausführliche Liste oder eine sehr kurz gehaltene Liste auszudrucken, ist eine der Anforderungen an das Programm. Nun gäbe es zwei theoretische Ansätze, die Lösung dieses Problems zu realisieren. Die erste Möglichkeit: Der Ausdruck wird komplett durch das steuernde Hauptprogramm durchgeführt. In diesem Fall müsste das Hauptprogramm dafür sorgen, dass die Sonderfälle unterschieden würden. Im Prinzip gäbe es dabei zwei verschiedene Druckroutinen, die jeweils einmal für den ausführlichen und einmal für den kompakten Ausdruck zuständig wären. Möglichkeit Nr. 2: Die Klassen selbst sorgen für die Aufbereitung der entsprechenden Texte.

Zufälligerweise gibt es eine Eigenschaft in der Basisklasse, die einen recht kompakten, beschreibenden Text für den Inhalt einer Objektinstanz zurückliefert. Diese Eigenschaft hat den Namen `Description`. Diese Eigenschaft wird von den abgeleiteten Klassen überschrieben; sie erweitern die Eigenschaft dahingehend, dass auch die zusätzlichen gespeicherten Informationen bei der Ausgabe berücksichtigt werden.

Viel leichter wäre es jetzt also, wenn das Programm in Abhängigkeit von der Eingabe des Anwenders entweder die Eigenschaft der Basisklasse oder die der jeweiligen abgeleiteten Klasse für die Ausgabe auf den Bildschirm verwenden würde. Die Basisklasse müsste zu diesem Zweck eine weitere Eigenschaft bereitstellen, mit der sich steuern ließe, ob die kompakte oder die ausführliche Form bei der Ausgabe der Artikelbeschreibung zu berücksichtigen ist.

Aus diesem Grund gibt es bereits in der Basisklasse eine Eigenschaft namens `PrintType`, die nur zwei Zustände aufweisen kann, welche in einer `Enum` definiert sind:

```
Enum PrintType  ' Reportform
    Brief      ' kurz
    Detailed   ' ausführlich
End Enum
```

Bevor nun die eigentliche Ausgabe auf dem Bildschirm des Inhalts eines Artikelobjektes erfolgt, muss das die Ausgabe steuernde Programm dafür sorgen, dass die PrintType-Eigenschaft auf den korrespondierenden Wert gesetzt wird, den der Anwender beim Start des Programms definiert hat. Wenn diese Voraussetzung erfüllt ist, kann ein Artikelobjekt selbst entscheiden, ob die Beschreibung der Basisklasse oder der eigenen Klasse zurückgeliefert werden soll. Da dieser »Entscheidungsalgorithmus« sowohl in der Basisklasse als auch in allen abgeleiteten Klassen derselbe ist, reicht es aus, ihn ausschließlich in der Basisklasse zu implementieren. Durch die Polymorphie ist es anschließend möglich, die »richtige« Description-Eigenschaft der jeweiligen Klasse abzufragen und daraus den Rückgabetext zusammenzubasteln.

Das letzte Problem ist die Realisierung der kompakten Form des Artikeldrucks. Hier müsste der ToString-Funktion die Möglichkeit gegeben sein, die Polymorphie außer Kraft zu setzen und gezielt die Description-Eigenschaft der Basisklasse aufzurufen, egal ob es sich bei dem gespeicherten Objekt um ein abgeleitetes oder um ein Original handelt. Genau das erreichen Sie mit dem Schlüsselwort `MyClass`. `MyClass` erlaubt, gezielt auf ein Element der Klasse zuzugreifen, in der `MyClass` »steht«. Sie heben damit quasi die Überschreibung einer Methode für die Dauer des Aufrufs auf und verwenden das Original der Klasse, in der `MyClass` eingesetzt wird. Die `ToString`-Funktion der Basisklasse macht sich genau diese Eigenschaft zunutze:

```
Public Overrides Function ToString() As String
    If PrintTypeSetting = PrintType.Brief Then
        'Kurzform: Es wird in jedem Fall
        'die Description-Eigenschaft des Objektes
        'verwendet.
        Return MyClass.Description & vbCrLf & vbLf &
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCrLf & vbLf
    Else
        'Langform: Die Description Eigenschaft des Objektes
        'selber wird verwendet.
        Return Me.Description & vbCrLf & vbLf &
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCrLf & vbLf
    End If
End Function
```

Die nachstehende Tabelle fasst die verschiedenen Schlüsselworte zusammen, die den Klassenebenenzugriff spezifizieren:

Schlüsselwort	Beschreibung
<code>Me</code>	Das Element der eigenen Klasse wird verwendet.
<code>MyClass</code>	Das Element der Klassenableitung, die das <code>MyClass</code> -Schlüsselwort enthält, wird verwendet.
<code> MyBase</code>	Das Element der Basisklasse wird verwendet.

Tabelle 15.1 Schlüsselworte zur Bestimmung von überschriebenen Elementen unterschiedlichen Rangs in der Vererbungshierarchie einer Klasse

Abstrakte Klassen und virtuelle Prozeduren

Schreiben Sie Ihre Texte auch mit Microsoft Word? Kennen Sie dann auch Dokumentenvorlagen? Ja? Gut. Dann wissen Sie quasi auch, was abstrakte Klassen sind. Dokumentenvorlagen in Word dienen dazu, eine Formatierungsrichtlinie für Dokumente festzuschreiben.

Wenn ich, wie in diesem Buch, einen »Standardabsatz« schreibe, dann bestimmt die Dokumentenvorlage, dass die Schrift dafür »Minion« und die Schriftgröße auf 10 Punkt gesetzt wird. Wenn ein Dokument auf Basis einer Dokumentenvorlage erstellt ist, übernimmt es alle ihre Eigenschaften. Nur so ist gewährleistet, dass die Kapitel eines Buchs später auch alle im gleichen Stil formatiert sind.

Abstrakte Klassen funktionieren nach ähnlichem Konzept. Sie stellen Prototypen von Prozeduren bereit, aber sie dienen ausschließlich als Vorlage. Um Entwickler dabei – rabiat ausgedrückt – dazu zu zwingen, bestimmte Elemente neu zu implementieren, stellen sie zusätzlich virtuelle Prozeduren bereit. Eine virtuelle Prozedur hat zwei Aufgaben:

- Sie gibt dem Entwickler die Signatur einer Prozedur (oder die Signaturen für überladene Prozeduren) vor.
- Sie »zwingt« den Entwickler, in der Ableitung der Klasse die vorhandene Prozedur zu überschreiben und sie vor allen Dingen damit zu implementieren.

Ein Praxisbeispiel (und dazu muss ich gar nicht weit ausholen): Stellen Sie sich vor, das Beispielprogramm aus dem letzten Abschnitt verrichtet seinen Dienst in einem großen Versandhaus wie Otto, Quelle oder Amazon. Hier sind natürlich wesentlich mehr Artikelgruppen zu berücksichtigen – bei Versendern wie Amazon, die weltweit operieren, sind auch noch mehr Mehrwertsteuersätze zu berücksichtigen.

Es würde ziemlich viel Sinn ergeben, die vorhandene Artikelklasse `ShopItem` als abstrakte Klasse zu formulieren. Damit Teammitglieder gezwungen sind, die mehrwertsteuerbezogenen Funktionen neu zu implementieren (sodass sie nicht vergessen werden), läge es nahe, die betroffenen Funktionen als virtuelle Funktionen auszulegen. Sie stellen damit sicher, dass jeder Entwickler *bewusst* programmiert, denn implementiert er die mehrwertsteuerbezogenen Funktionen nicht, wird seine Klasse nicht kompilierbar sein.

HINWEIS Das gilt übrigens auch für unser hypothetisches Beispiel, das Sie im grauen Kasten ab Seite 467 beschrieben finden. Es macht hier viel Sinn, die Basisklasse `GraphicsObject` abstrakt zu implementieren, da sie selber eine `Render`-Methode noch gar nicht implementieren könnte! Was sollte die auch zeichnen?

An dem Beispielprogramm müssen Sie dabei kaum Änderungen vornehmen. Sie können nach dem Umbau `ShopItem` auch weiterhin als Objektvariable verwenden. Sie können es nur nicht mehr direkt instanziiieren – aber das haben wir im Beispielprogramm ohnehin nicht gemacht (als hätte ich es kommen sehen ...).

Eine Klasse mit `MustInherit` als abstrakt deklarieren

Um eine abstrakte Klasse zu erstellen, verwenden Sie das Schlüsselwort `MustInherit`. Wenn Sie dieses Schlüsselwort vor das Schlüsselwort `Class` der Klasse `ShopItem` in unserem Beispiel stellen, passiert erst einmal gar nichts. Sie haben damit zunächst einmal erreicht, dass die Klasse nur noch in einer Ableitung instanziert werden kann. Da wir `ShopItem` selbst nicht instanziiieren, bleibt alles beim Alten, und es gibt auch keine Fehlermeldungen.

Was Sie allerdings nun nicht mehr können (aber vorher hätten können), ist, die Klasse irgendwo im Programm zu instanziieren. Versuchen Sie es: Fügen Sie in der Sub Main folgende Zeile ein,

```
Dim locShopItemVersuchsInstanz As New ShopItem
```

so erhalten Sie eine Fehlermeldung etwa wie in Abbildung 15.10 zu sehen:



Abbildung 15.10 Eine mit MustInherit als abstrakt definierte Klasse kann nicht instanziert werden

Eine Methode oder Eigenschaft einer abstrakten Klasse mit MustOverride als virtuell deklarieren

Fehler wünschen Sie sich als Entwickler normalerweise eher weniger. Ist die Fehlerquelle jedoch eine virtuell definierte Methode oder Eigenschaft, sieht das allerdings anders aus: Eine Fehlermeldung tritt in der Regel deswegen auf, weil sie in einer abgeleiteten Klasse nicht überschrieben – ergo: neu implementiert – wurde. Aber genau das wollen Sie mit virtuellen Methoden bzw. Eigenschaften erreichen.

In unserem Beispiel ist die Eigenschaft GrossPrice, die den Bruttopreis eines Artikels aus dem Nettopreis mit dem gültigen Mehrwertsteuersatz errechnet, eine willkommene Demonstration für eine virtuelle Eigenschaft, denn: Es ist wichtig, dass sich jeder Entwickler darüber im Klaren ist, dass er in einer Ableitung der Klasse ShopItem selbst für das korrekte Errechnen des Bruttopreises verantwortlich ist und eine Variante dieser Eigenschaft auf jeden Fall in seiner abgeleiteten Klasse implementiert.

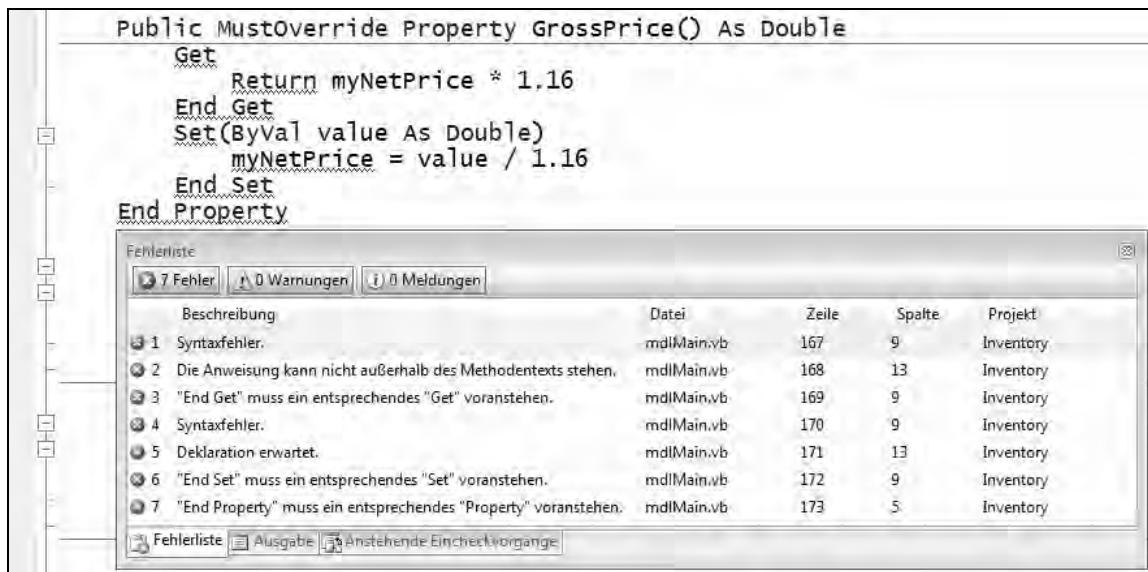


Abbildung 15.11 Mit MustOverride als virtuell gekennzeichnete Prozeduren dürfen keinen Code enthalten und müssen in abgeleiteten Klassen überschrieben werden

Sie deklarieren eine Prozedur mit dem Schlüsselwort **MustOverride** als virtuell. Wenn Sie die vorhandene Eigenschaft **GrossPrice** der Klasse **ShopItem** folgendermaßen abändern,

```
Public MustOverride Property GrossPrice() As Double
```

bekommen Sie ein paar – in diesem Fall – »gewünschte« Fehlermeldungen, etwa wie in Abbildung 3.19 zu sehen:

Ein Fehler liegt direkt in der Eigenschaftenprozedur, denn: Virtuelle Prozeduren selbst dürfen keinen Code enthalten. Sie dienen nur dazu, zu sagen: »Vergiss nicht, lieber Entwickler, du musst diese Prozedur in deiner abgeleiteten Klasse implementieren.« Und dass das gut funktioniert, zeigt die folgende Fehlermeldung:

Die **DVDItem**-Klasse muss als **MustInherit** deklariert werden oder folgende geerbte **MustOverride**-Member überschreiben:

```
Inventory.ShopItem : Public MustOverride Property GrossPrice() As Double.
```

Denn es ist in der Tat richtig, dass die Klasse **DVDItem** diese Eigenschaft nicht implementiert. Stellen Sie sich vor, Videos und DVDs würden mit 19% MwSt. besteuert – wie leicht hätte dem Entwickler »durchrutschen« können, die Eigenschaft neu zu implementieren! Die betroffenen Artikel wären in diesem Fall fälschlicherweise mit 16% MwSt. besteuert worden.

Die Änderungen, die Sie am Programm durchführen müssen, halten sich in Grenzen:

- Sie entfernen den Code der Eigenschaftenprozedur **GrossPrice** in der Klasse **ShopItem**, sodass nur der Prototyp stehen bleibt (der Prozedurenkopf).
- Sie fügen eine Eigenschaftenprozedur **GrossPrice** in die Klasse **DVDItem** ein – eigentlich genau den Code, der sich zuvor in der Klasse **ShopItem** befand.

Danach sind alle Fehler eliminiert, und Ihr Programm ist bereit für die kommende Mehrwertsteuererhöhung bzw. »Subventionsstreichung« ;-).

BEGLEITDATEIEN

Die Begleitdateien zum Beispiel mit abstrakten Klassen finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Inventory02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Schnittstellen (Interfaces)

Das Konzept von Schnittstellen bietet Ihnen eine weitere Möglichkeit, Klassen zu standardisieren. Schnittstellen dienen dazu – ähnlich wie abstrakte Klassen – Vorschriften zu erstellen, dass innerhalb der Klassen, die von einer Schnittstelle ableiten, bestimmte Elemente zwingend erforderlich sind. Im Gegensatz zu abstrakten Klassen enthalten Schnittstellen jedoch nur diese Vorschriften; sie enthalten überhaupt keinen Code. Und das ist besonders interessant: Klassen können in .NET (anders als in C++, wo auch noch eine oftmals verwirrende Mehrfachvererbung möglich ist) immer nur von einer Klasse erben, aber beliebig viele Schnittstellen implementieren. In der Tat kann man sich Schnittstellen unter .NET genauso vorstellen, wie wir den Begriff im Alltag verwenden: Eine Klasse – und damit ein Objekt – z. B. ein Notebook (das von der Basisklasse Rechner abgeleitet wurde) kann durchaus mehrere Schnittstellen implementieren. In unserem Beispiel kann die Notebook Klasse die Bluetooth-, die VGA-, die USB- und weitere Schnittstellen implementieren.

Da Schnittstellen keinerlei Code enthalten, können sie, genau wie abstrakte Klassen, nicht direkt instanziiert werden. Allerdings können Objektvariablen vom Typ einer Schnittstelle dazu verwendet werden, ableitende Klassen zu referenzieren. Dieses Verhalten erlaubt es, Klassenbibliotheken zu erstellen, die eine enorme Flexibilität aufweisen.

Durch die Eigenarten, die Schnittstellen aufweisen, werden sie gerade innerhalb des Frameworks zu zweierlei Zwecken verwendet:

- Sie dienen auf der einen Seite lediglich dazu, den Programmierer, der sie einbindet, an bestimmte Konventionen zu binden, ohne dass seine Klassen und die Objekte, die daraus entstehen, später durch Polymorphie vom Framework selber über diese Schnittstellen gesteuert würden. Man spricht in diesem Fall von einem so genannten »Interface Pattern« (Schnittstellenmuster). In Kapitel 17 finden Sie ein typisches Beispiel dafür.
- Sie dienen auf der anderen Seite dazu, allgemein gültige Verwalterklassen zu entwickeln, in der später alle die Objekte verwendet werden können, die die vorgegebenen Schnittstellen dafür implementieren. Um noch mal auf unser hypothetisches Beispiel zu sprechen zu kommen, dass Sie im grauen Kasten ab Seite 467 beschrieben finden: Auch hier böte es sich an, die Objekte, die gezeichnet werden sollen, in Form von Schnittstellen zu beschreiben, sodass das Team, das die konkreten Objekte zum Zeichnen implementieren muss, eine *größtmögliche* Flexibilität bei der Implementierung hätte. Sie könnten dann sowohl code-enthaltene Basisklassen für die Objekte mit dem größten gemeinsamen Nenner entwerfen, könnten aber auch Sonderobjekte entwerfen, die lediglich die entsprechenden Schnittstellen einbinden, aber komplett von Grund auf neu entwickelt wären. Damit könnte man auch sehr individuelle Implementierungen erreichen, die sich dennoch in das durch die Schnittstellen vorgegebene Objektmodell einfügen.

Für den letzten Punkt gibt es dazu häufig eine Art Drei-Stufen-Konzept für Komponenten, die dem Entwickler zur Verfügung gestellt werden: Auf oberster Ebene gibt es eine Klasse, die die eigentliche Aufgabe übernimmt. Sie kann Objekte einbinden, die eine bestimmte Schnittstelle implementieren. Auf unterster Ebene stellt die Komponente dem Entwickler eben diese Schnittstelle(n) zur Verfügung. Damit der Entwickler, der die Komponente verwenden will, nicht die komplette Implementierung selbst durchführen muss, sollte ihm die Komponente basierend auf den angebotenen Schnittstellen abstrakte Klassen zur Verfügung stellen, die eine gewisse Grundfunktionalität enthalten.

Ein weiteres Beispiel soll diese Zusammenhänge einmal mehr klären: Angenommen, Sie werden mit der Aufgabe betraut, eine Komponente zu entwickeln, die Daten tabellarisch auf dem Bildschirm darstellt. Diese Komponente – nennen wir Sie *Tabellensteuerelement* – ist prinzipiell aus zwei Unterkomponenten aufgebaut: Zum einen ist das die Komponente, die die Tabelle zeichnet, sie mit Gitternetzlinien versieht, dafür sorgt, dass der spätere Anwender einen Cursor in der Tabelle bewegen kann usw. Diese Komponente wird aber idealerweise nicht selbst dafür zuständig sein, den Inhalt einer Tabellenzelle zu malen, sondern diese Aufgabe einer weiteren Klasse überlassen und bindet sie lediglich ein. Die Aufgabe der weiteren Komponente auf der anderen Seite ist es, den Inhalt einer einzigen Tabellenzelle zu zeichnen. Diese zweite Komponente speichert also nicht nur die Daten für die einzelnen Tabellenzellen, sie sorgt auch dafür, dass diese Daten zu gegebener Zeit in Form einer Tabellenzelle auf den Bildschirm gemalt werden. Damit der spätere Entwickler, der das Tabellensteuerelement verwendet, die größtmögliche Flexibilität in seiner Verwendung hat, sollte die zweite Klasse – die Tabellenzelle – nach Möglichkeit nicht als festgeschriebene Klasse, sondern auf unterster Ebene auch als Schnittstelle zur Verfügung stellen. Auf der zweiten Ebene sollte das Tabellensteuerelement dem Entwickler auch mindestens eine auf der Schnittstelle basierende abstrakte Klasse zur Verfügung stellen, die die wichtigste Grundfunktionalität bereits enthält. Und schließlich stellt die Komponente auf oberster Ebene fix und fertige Tabellenzellenkomponenten zur Verfügung, mit denen der Entwickler direkt loslegen und Tabellen mit Daten füllen kann.

Stellt der Entwickler dann nach geraumer Zeit fest, dass die vorhandenen Zellenkomponenten nicht die Möglichkeiten bieten, die er benötigt, kann er sich entscheiden:

- Er kann entweder die abstrakte Klasse ableiten und daraus eine Tabellenzellenkomponente entwickeln, die seinen Anforderungen genügt. Der Aufwand dafür hält sich in Grenzen, weil die abstrakte Klasse einen Großteil des notwendigen Verwaltungscodes bereits zur Verfügung stellt, den er lediglich ergänzen muss.
- Falls ihn diese Vorgehensweise immer noch zu sehr einschränkt, kann er die *vollständige* Implementierung der Zellenklasse übernehmen. Er muss in diesem Fall die von der Tabellenkomponente zur Verfügung gestellte Schnittstelle einbinden, um sicherzustellen, dass alles an Funktionalität innerhalb seiner Zellenklasse vorhanden ist, was die Tabellenkomponente vorschreibt. Der Nachteil: Der Aufwand, dieses Vorhaben zu realisieren, ist logischerweise erheblich größer.

Das bislang verwendete Beispiel eignet sich ebenfalls, die Verwendung von Schnittstellen zu demonstrieren. Dazu wird eine Schnittstelle implementiert, die die Grundfunktionen vorschreibt, auf die das Hauptprogramm bislang über die Klasse `ShopItem` Zugriff hatte. Das Interface für das Beispielprogramm soll den Namen `IShopItem` bekommen (das Voranstellen des Buchstabens »I« für Interface ist eine gängige Konvention für die Benennung einer Schnittstellenklasse), und die Implementierung geschieht im Beispielprojekt noch vor dem Hauptmodul:

```
Interface IShopItem
    Property PrintTypeSetting() As PrintType
    ReadOnly Property ItemDescription() As String
    Function ToString() As String
    Property GrossPrice() As Double
End Interface
```

Als nächstes kümmern wir uns um das Hauptprogramm, das zur Steuerung jetzt nicht mehr die abstrakte Artikelbasisklasse, sondern die neue Schnittstelle verwenden soll. Dazu ist eine einzige Änderung notwendig, die Deklaration der abstrakten Artikelbasisklasse:

```
Sub Main()
    'Streamreader zum Einlesen der Textdateien
    Dim locSr As StreamReader
    Dim locList As New DynamicList ' Die Dynamische Liste
    Dim locElements() As String ' Die einzelnen ShopItem-Elemente
    Dim locShopItem As IShopItem ' Ein einzelnes Shop-Element
    Dim locDisplayMode As PrintType ' Der Darstellungsmodus
```

Da die Schnittstelle IShopItem alle Eigenschaften der ursprünglichen abstrakten Basisklasse ShopItem enthält, sind keine weiteren Änderungen erforderlich. Dennoch hat das Verändern der einen Zeile enorme Auswirkungen, und es hagelt geradezu Fehler (siehe Abbildung 15.12):

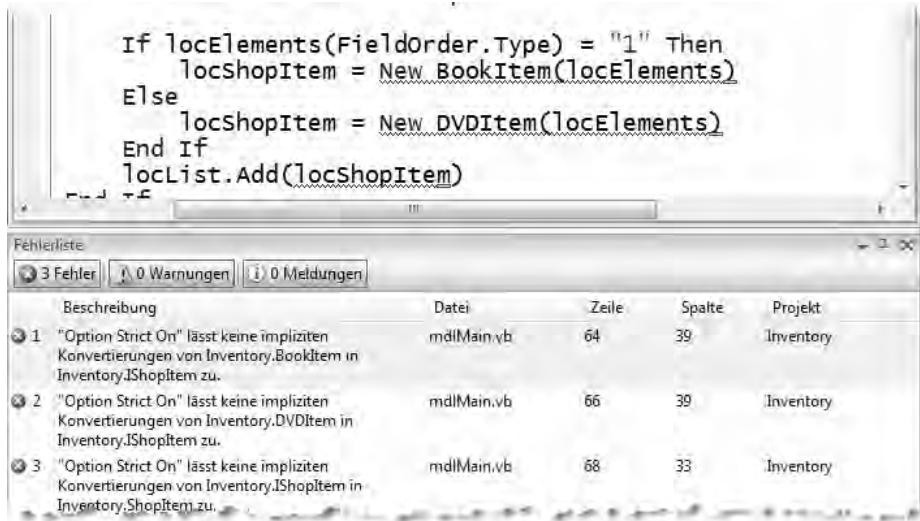


Abbildung 15.12 Da zum jetzigen Zeitpunkt noch keine der abgeleiteten Klassen die Schnittstelle implementiert, ist ein Referenzieren der verwendeten Klassen noch nicht möglich

Das Hauptprogramm kann zwar nun alle Objekte verarbeiten, die die neue Schnittstelle implementiert haben. Da aber bislang keine der Klassen die Schnittstellen implementiert, ist das Programm zu diesem Zeitpunkt nicht lauffähig. Visual Basic beschwert sich zu diesem Zeitpunkt mit einem nicht so klar verständlichen Fehler: Da es keinen Zusammenhang zwischen locShopItem (das als IShopItem deklariert ist) und den Klassen BookItem und DVDItem herstellen kann, macht es das Nächstliegende:

Es versucht, die Typen implizit zu konvertieren. Da ein implizites Konvertieren von einem Typ zum anderen durch Option Strict On (als global gültig in den Projekteigenschaften eingestellt) unterbunden ist, liefert es eine entsprechende Fehlermeldung, die zunächst ein wenig in die Irre führen kann.

Nun leiten sich aber alle Klassen, die wir im Programm tatsächlich verwenden, von der bisherigen Klasse ShopItem ab. Die Implementierung der Schnittstelle IShopItem in der Klasse ShopItem bewirkt, dass anschließend auch alle von ShopItem abgeleiteten Klassen automatisch IShopItem implementieren. Es reicht also aus, wenn Sie alle geforderten Prozeduren in der abstrakten Basisklasse einbauen, damit das Programm anschließend wieder lauffähig ist.

Bei der Implementierung einer Schnittstelle in eine Klasse müssen Sie in Visual Basic auf zwei Sachen achten:

- Sie geben bei der Definierung der Klasse mit Implements an, welche Schnittstelle implementiert werden soll.
- Sie bestimmen für jede betroffene Prozedur der Klasse individuell, welche Schnittstellenprozedur sie implementieren soll. Auch das geschieht mit dem Schlüsselwort Implements.

Um die Implements-Anweisung hinzuzufügen, geben Sie bitte unterhalb der Zeile `MustInherit Class ShopItem` **Implements IShopItem** ein, drücken aber zunächst NICHT , sondern verlassen die Zeile mit einer der Cursor-Tasten.

Sobald Sie das Implements-Schlüsselwort der Klassendefinition auf diese Weise hinzugefügt haben, verschwinden zwar die bisherigen Fehlermeldungen. Allerdings werden diese in der nächsten Sekunde durch andere ersetzt, wie in Abbildung 15.13 zu sehen:

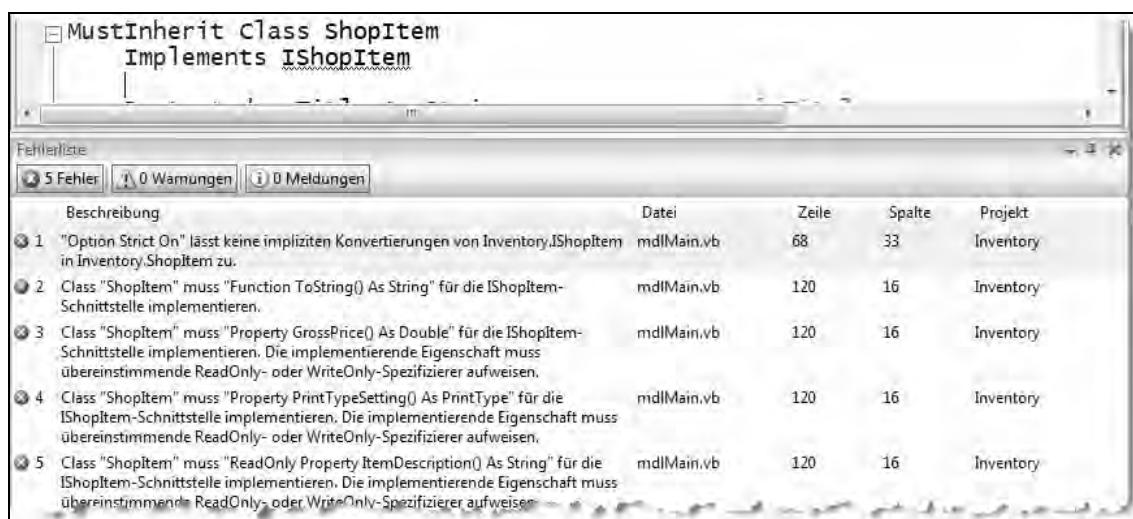


Abbildung 15.13 Wenn Sie eine Schnittstelle implementieren, müssen Sie auch für die Implementierung der einzelnen Prozeduren der Schnittstelle sorgen

Jetzt fragen Sie sich möglicherweise, wieso es nicht ausreicht, der Prozedur in der Klasse denselben Namen wie den der Schnittstellenprozeduren zu geben – beim Ableiten von Klassen reicht es schließlich aus.

In C# beispielsweise funktioniert das übrigens ohne Probleme, die Frage ist also berechtigt. Sie haben in C# auf der anderen Seite auch nicht wie in Visual Basic die Möglichkeit, einen ganz anderen Namen in Ihrer Klassenprozedur zu definieren, der eben nicht dem Namen der Interface-Prozedur entspricht.³ Insofern bringt Ihnen dieses Feature von Visual Basic schon eine größere Flexibilität. Dazu kommt, dass Microsoft Intermediate Language, in die jede .NET-Anwendung zunächst kompiliert wird, ebenfalls so konzipiert ist, dass sie die explizite Angabe der zu implementierenden Schnittstellenprozedur erfordert – in C# erledigt diese Umsetzung der Compiler zwar implizit, aber auch eben nur ausschließlich implizit.

Außerdem genießen Sie in Visual Basic dank IntelliSense einen Vorteil in Form einer Eingabehilfe: Sobald Sie am Ende einer Prozedur, der Sie ein Schnittstellenelement zuweisen wollen, das **Implements**-Schlüsselwort eingegeben haben, bietet Ihnen IntelliSense die möglichen Implementierungen zur Auswahl in einer Liste an (siehe Abbildung 15.14):

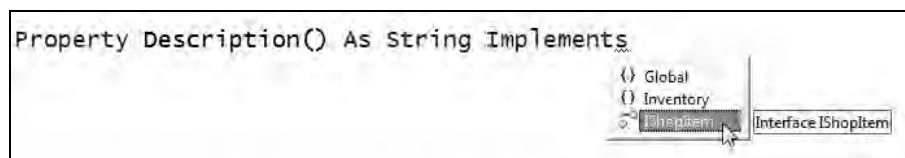


Abbildung 15.14 IntelliSense unterstützt Sie bei der Auswahl der richtigen Schnittstellenimplementierung

Um die Beispielanwendung wieder zum Laufen zu bringen, müssen Sie die folgenden Prozeduren bearbeiten und ihnen die richtigen Schnittstellenprozeduren zuweisen:

```

Public MustOverride Property GrossPrice() As Double Implements IShopItem.GrossPrice
Public Property PrintTypeSetting() As PrintType Implements IShopItem.PrintTypeSetting
    Get
        Return myPrintTypeSetting
    End Get
    Set(ByVal Value As PrintType)
        myPrintTypeSetting = Value
    End Set
End Property

Public Overrides Function ToString() As String Implements IShopItem.ToString
    If PrintTypeSetting = PrintType.Brief Then
        'Kurzform: Es wird in jedem Fall
        'die Description-Eigenschaft des Objektes
        'verwendet.
        Return MyClass.Description & vbCrLf & _
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
            Me.GrossPriceFormatted & vbCrLf & vbCrLf
    Else
        'Langform: Die Description Eigenschaft des Objektes
        'selber wird verwendet.
        Return Me.Description & vbCrLf & vbCrLf & _
    End If
End Function
  
```

³ Von einer Ausnahme abgesehen, bei der es allerdings dennoch nicht so flexibel wie in VB.NET zugeht – doch dazu später in Kapitel 22 noch mehr.

```

        Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
        Me.GrossPriceFormatted & vbCr & vbLf
    End If
End Function

Public Overridable ReadOnly Property Description() As String Implements IShopItem.ItemDescription
    Get
        Return OrderNumber & vbTab & Title
    End Get
End Property

```

HINWEIS Die letzte Eigenschaft verdient hier besonderes Interesse, denn sie bindet eine Schnittstellenprozedur ein, die einen anderen Namen hat als die Prozedur, die sie einbindet. Sie sehen also, dass Namen an dieser Stelle nur Schall und Rauch sind. Erst mit `Implements` wird festgelegt, welches Schnittstellenelement welchem Klassenelement zugeordnet werden soll. Dabei haben Sie natürlich nicht komplett freie Hand: Sie können – wie anzunehmen ist – nicht eine in einer Schnittstelle definierten Eigenschaft in einer Klassmethode, sondern eben auch nur als Eigenschaft implementieren. Sie können auch keine Methode in einer Schnittstellendefinition, die als Parameter einen `String` entgegennimmt, einer Methode einer Klasse zuordnen, die einen `Integer` als Parameter erwartet. Und auch die Rückgabetypen müssen sich in der einbindenden Klasse und der Schnittstelle, die eingebunden wird, entsprechen. Fehlzuweisungen in dieser Form würden das Common Type System von .NET grob verletzen, und deswegen spielt auch schon der Background-Compiler nicht mit, wenn Sie einen solchen Versuch unternehmen, wie Abbildung 15.15 zeigt.

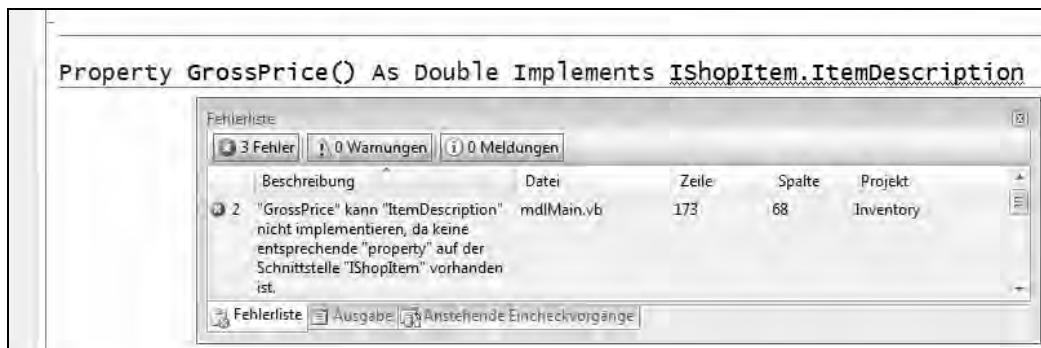


Abbildung 15.15 Signaturen, Elementtyp (Eigenschaft, Methode, Ereignis) und Rückgabetypen in einer Schnittstellendefinition müssen denen in der einbindenden Klasse entsprechen

Nachdem Sie diese Änderungen durchgeführt haben, werden Sie nur noch einige »Fehler« in der Klasse `DynamicList` finden. Diese Klasse akzeptierte bisher lediglich `ShopItem`-Objekte in den Methoden `Add` und `Item`. Wenn Sie die verwendeten `ShopItem`-Objekte durch solche vom Typ `IShopItem` ersetzen, ist das Programm wieder lauffähig (die geänderten Stellen sind wieder fett hervorgehoben).

```
Class DynamicList

    Protected myStepIncreaser As Integer = 4
    Protected myCurrentArraySize As Integer
    Protected myCurrentCounter As Integer
    Protected myArray() As IShopItem

    Sub New()
        myCurrentArraySize = myStepIncreaser
        ReDim myArray(myCurrentArraySize)
    End Sub

    Sub Add(ByVal Item As IShopItem)

        'Prüfen, ob aktuelle Arraygrenze erreicht wurde.
        If myCurrentCounter = myCurrentArraySize - 1 Then
            'Neues Array mit mehr Speicher anlegen,
            'und Elemente hinüberkopieren. Dazu:

            'Neues Array wird größer:
            myCurrentArraySize += myStepIncreaser

            'Temporäres Array erstellen.
            Dim locTempArray(myCurrentArraySize - 1) As IShopItem

            'Elemente kopieren.
            'Wichtig: Um das Kopieren müssen Sie sich,
            'anders als bei VB6, selber kümmern!
            For locCount As Integer = 0 To myCurrentCounter
                locTempArray(locCount) = myArray(locCount)
            Next

            'Temporäres Array dem Memberarray zuweisen.
            myArray = locTempArray
        End If

        'Element im Array speichern
        myArray(myCurrentCounter) = Item

        'Zeiger auf nächstes Element erhöhen.
        myCurrentCounter += 1
    End Sub

    'Liefert die Anzahl der vorhandenen Elemente zurück.
    Public Overridable ReadOnly Property Count() As Integer
        Get
            Return myCurrentCounter
        End Get
    End Property
```

```
'Erlaubt das Zuweisen.  
Default Public Overridable Property Item(ByVal Index As Integer) As IShopItem  
    Get  
        Return myArray(Index)  
    End Get  
    Set(ByVal Value As IShopItem)  
        myArray(Index) = Value  
    End Set  
End Property  
End Class
```

BEGLEITDATEIEN

Das Programmbeispiel mit den Änderungen für Schnittstellen finden Sie übrigens im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Inventory03

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Unterstützung bei abstrakten Klassen und Schnittstellen durch den Editor

Ihnen ist sicherlich aufgefallen, dass Sie die Änderung für die Schnittstellen-Implementierung durch Implements in der Klasse ShopItem nicht mit bestätigen, sondern die Zeile mit einer der Cursor-Tasten verlassen.

Hätten Sie nämlich die Zeile mit verlassen, dann wäre die Editor-Unterstützung für das Implementieren der Funktionsrümpfe angesprungen. In unserem Fall, in dem die Routinen aber schon komplett vorhanden waren, wäre das allerdings eher hinderlich gewesen, wie die beiden folgenden Beispiele zeigen.

Zu diesem Zweck erstellen Sie neues Visual Basic-Konsolenprojekt unter einem beliebigen Namen. Implementieren Sie anschließend eine Schnittstelle und eine Klasse nach folgender Vorlage, die Sie oberhalb der Zeile `Module1` in der Codedatei `Module1.vb` erfassen.

```
Interface ITest  
    Property EineEigenschaft() As String  
    Function EineMethode() As Integer  
End Interface  
  
Public Class BindetITestEin  
  
End Class
```

Anschließend bewegen Sie die Einfügemarkie unter die Zeile `Public Class BindetITestEin`, geben **Implements ITest** ein und drücken anschließend .

```

Test.vb* Startseite
BindetITestEin
Interface ITest
    Property EineEigenschaft() As String
    Function EineMethode() As Integer
End Interface

Public Class BindetITestEin
    Implements ITest

        Public Property EineEigenschaft() As String Implements ITest.EineEigenschaft
            Get
            End Get
            Set(ByVal value As String)
            End Set
        End Property

        Public Function EineMethode() As Integer Implements ITest.EineMethode
        End Function
    End Class

```

Abbildung 15.16 Nachdem Sie mit `Implements` eine Schnittstelle eingebunden haben, fügt der Editor die kompletten Coderümpfe der benötigten Elemente ein

Diese Vorgehensweise funktioniert auch ergänzend, und das heißt: Sie können eine Methode, eine Eigenschaft oder ein Ereignis (zu Ereignissen später mehr) in der Schnittstellendefinition hinzufügen, bekommen dann natürlich eine Fehlermeldung, weil dieses neue Element nicht in der implementierenden Klasse vorhanden ist. Bewegen Sie die Einfügemarke allerdings anschließend hinter den Schnittstellennamen der `Implements`-Anweisung unterhalb der Klassendefinition und drücken mit , wird der Elementcoderumpf wieder an das Ende der Klasse angehängt. Probieren Sie es aus:

- Fügen Sie in der Schnittstelle `ITest` eine weitere Methode vom Typ `String` namens `ZweiteMethode` ein, sodass sich folgender Code ergibt:

```

Interface ITest
    Property EineEigenschaft() As String
    Function EineMethode() As Integer
    Function ZweiteMethode() As String
End Interface

```

Anschließend sehen Sie in der Fehlerliste eine Fehlermeldung mit dem Hinweis darauf, dass diese neue Methode nicht in der Klasse implementiert wurde, die die Schnittstelle einbindet.

- Um diesen Fehler zu beheben, reicht es aus, die Einfügemarke unter die Zeile `Public Class BindetITestEin` hinter `Implements ITest` zu bewegen und mit zu betätigen.

In diesem Moment verschwindet der Fehler, da der Editor den Coderumpf für die Methode `ZweiteMethode` einfügt, sie mit der Schnittstelle per `Implements` verbindet und damit den Implementierungsvorschriften der Schnittstelle genüge tut.

Die Tücken der automatischen Codeergänzung bei Schnittstellen oder abstrakten Klassen

Allerdings funktioniert die Unterstützung durch den Editor bisweilen nicht immer so reibungslos und kann zu – na ja, nennen wir es – »Orientierungsproblemen« führen, wenn es bereits Methoden oder Eigenschaften der Basisklasse gibt, die genau so heißen wie die zu implementierenden Methoden oder Eigenschaften, aber noch nicht mit diesen verknüpft sind.

Um das nachzustellen entfernen Sie bitte hinter der Funktion

```
Public Function ZweiteMethode() As String Implements ITest.ZweiteMethode
End Function
```

die Schnittstellenimplementierung `Implements ITest.ZweiteMethode`.

Daraufhin unterschlängelt der Editor die Zeile

```
Implements ITest
```

blau, da, wie er richtig erkennt, die Methode `ZweiteMethode` der Schnittstelle `ITest` nicht mehr korrekt implementiert ist. Doch die Editorunterstützung versagt anschließend ein wenig. Denn wenn sie nun abermals die Einfügemarke unter die Zeile `Public Class BindetITestEin` hinter `Implements ITest` bewegen und betätigen, erscheint am Ende der Klasse ein Funktionsrumpf, wie Sie ihn auch in Abbildung 15.17 erkennen können.

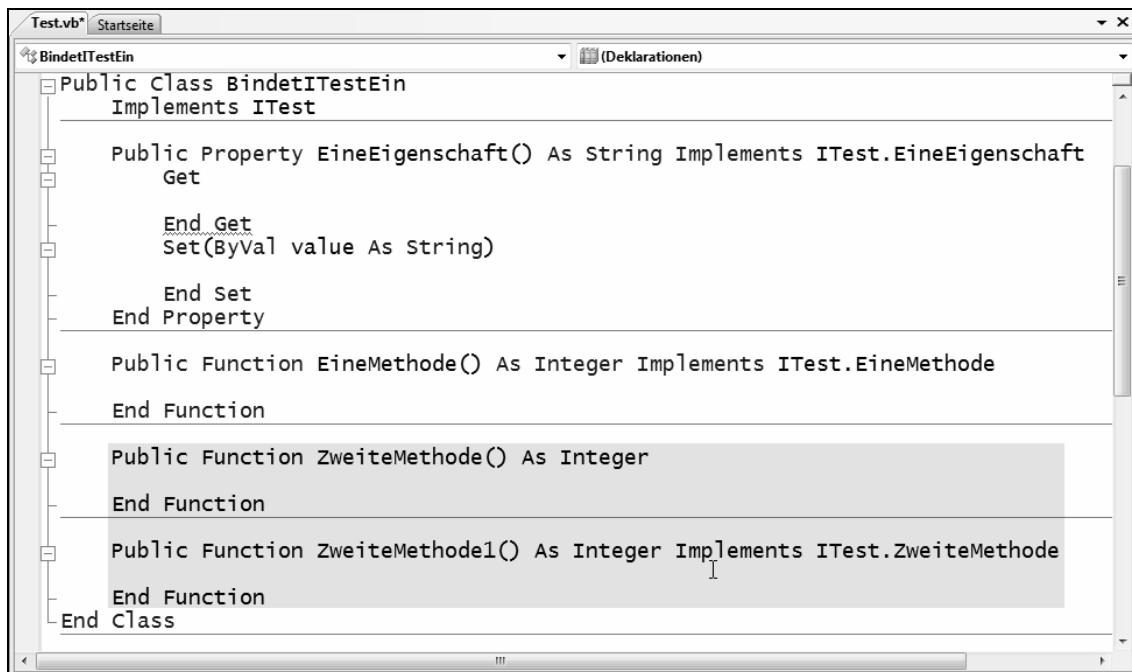


Abbildung 15.17 Beim automatischen Komplettieren von Coderümpfen von Elementen, die mit gleichem Namen schon vorhanden sind, wird ein »ähnlicher« Elementname erfunden

Die Methode `ZweiteMethode` gibt es nun (mehr oder weniger) zweimal – einmal mit nachgestellter »1«. Anstelle also hinter `ZweiteMethode` korrekterweise die `Implements`-Anweisung mit dem entsprechenden Schnittstellenelementnamen zu ergänzen, kreiert der Editor eine ganz neue Methode, und implementiert das Schnittstellenelement in dieser.

Falls Ihnen solche Dinge beim Entwickeln Ihrer eigenen Schnittstellenkreationen passieren, müssten Sie die `Implements`-Anweisung zur eigentlich gemeinten (und schon vorhandenen) Methode übertragen, und die vom Editor »erfundene« Methode löschen.

Noch diffuser wird es, wenn es um die Implementierung einer Methode oder Eigenschaft geht, die nur aus der Basisklasse hervorgeht.

`ToString` ist standardmäßig solch ein Kandidat. `ToString` ist in jeder neu erstellten Klasse enthalten, da er eine Methode von `Object` darstellt, und jede neue Klasse wird, wie wir ja schon wissen, implizit von `Object` abgeleitet, wenn nichts anderes gesagt wird. Damit erbt natürlich auch jede neue Klasse die `ToString`-Methode (warum die `ToString`-Methode an `Object` »hängt«, klärt übrigens der Abschnitt »Die Methoden und Eigenschaften von `Object`« ab Seite 503).

Nun achten Sie darauf was passiert, wenn wir der Schnittstelle exakt diese Methode hinzufügen. Erwartungsgemäß sehen wir zunächst einen Fehler im Programm, da die neue Schnittstellendefinition erwartet, dass wir eine `ToString`-Funktion implementieren. Das machen wir durch das standardmäßige Vorhandensein von `ToString` ja auch ohne weiteres Zutun; was allerdings fehlt, ist, unsere (geerbte) `ToString`-Funktion mit der Schnittstellenmethodendefinition `ToString` per `Implements` (á la Abbildung 15.14) zu verheiraten.

Mit `Implements` würde das auch nicht so ohne weiteres funktionieren, denn es gibt ja schließlich überhaupt keinen Methodencode, hinter den wir `Implements` für die Zuweisung des Schnittstellen-`ToString` hängen könnten.

Ergründen wir es also zunächst, wie uns der Editor »hilft«, und wie wir anschließend wirklich zum Ziel kommen:

- Fügen Sie in der Schnittstelle `ITest` eine weitere Methode vom Typ `String` namens `ToString` ein, sodass sich folgender Code ergibt:

```
Interface ITest
    Property EineEigenschaft() As String
    Function EineMethode() As Integer
    Function ZweiteMethode() As String
    Function ToString() As String
End Interface
```

Anschließend sehen Sie wieder in der Fehlerliste eine Fehlermeldung mit dem Hinweis darauf, dass diese neue Methode nicht in der Klasse implementiert wurde, die die Schnittstelle einbindet.

- Versuchen wir den Fehler auf die gleiche Weise mit der Editorunterstützung zu beheben: Platzieren Sie die Einfügemarke wieder unterhalb der Zeile `Public Class BindetITestEin` hinter `Implements ITest`, und betätigen Sie .

Das Ergebnis ist wieder dasselbe wie im vorherigen Beispiel, nur leider nicht so einfach zu durchschauen, da es in der Klasse ja keinen Code für `ToString` gibt (denn dieser wurde ja aus der Basisklasse `Object` übernommen).

```

Test.vb* Startseite
BindetITestEin (Deklarationen)

Public Class BindetITestEin
    Implements ITest

    Public Property EineEigenschaft() As String Implements ITest.EineEigenschaft
        Get
            ...
        End Get
        Set(ByVal value As String)
            ...
        End Set
    End Property

    Public Function EineMethode() As Integer Implements ITest.EineMethode
        ...
    End Function

    Public Function ZweiteMethode() As Integer Implements ITest.ZweiteMethode
        ...
    End Function

    Public Function ToString1() As String Implements ITest.ToString
        ...
    End Function
End Class

```

Abbildung 15.18 Verwirrender ist das Vorgehen des Editors bei der Codeergänzung, wenn es eine geerbte Methode oder Eigenschaft mit gleichem Namen wie in der Schnittstellendefinition gibt

Prinzipiell hat der Editor das gleiche Problem. Es gibt bereits eine Methode (`ToString`), nur ist sie dieses Mal nicht sichtbar, denn sie wurde aus der Basisklasse (`Object` in diesem Beispiel) übernommen. Also kreiert der Editor wieder eine Abwandlung des Funktionsnamens, und nennt diesen genau so wie den eigentlichen nur mit einer zusätzlich hinten angestellten »1«.

Diesen Fehler können Sie nur folgendermaßen beheben:

- Sie überschreiben die Methode bzw. Eigenschaft, die es zu implementieren gilt.
- Sie rufen innerhalb der Methode oder Eigenschaft nichts weiter auf als die Basismethode (oder Eigenschaft).
- Sie implementieren die Schnittstelle an der überschriebenen Methode bzw. Eigenschaft. Der Code dafür würde in unserem Fall folgendermaßen aussehen:

```

Public Overrides Function ToString() As String Implements ITest.ToString
    Return MyBase.ToString
End Function

```

Editorunterstützung bei abstrakten Klassen

Übrigens: Die Unterstützung, die Sie bei der Implementierung von Schnittstellen erfahren, gibt's auch bei abstrakten Klassen. Ergänzen Sie zur Demonstration das Beispiel um folgende abstrakte Klasse:

```
MustInherit Class AbstractTest
    Public MustOverride Property EineEigenschaft() As String
    Public MustOverride Function EineMethode(ByVal EinParameter As String) As String
End Class
```

Erstellen Sie nun eine Klasse, die auf AbstractTest basiert:

```
Public Class BasiertAufAbstractTest
    Inherits AbstractTest
End Class
```

In dem Moment, in dem Sie hinter `Inherits AbstractTest`  betätigen, komplettiert der Editor für Sie automatisch alle Methoden und Eigenschaften in Form von Coderümpfen, die mit `MustOverride` als virtuelle Methoden bzw. Eigenschaften gekennzeichnet wurden:

```
Public Class BindetITestEin
    Implements ITest

    Public Property EineEigenschaft() As String Implements ITest.EineEigenschaft
        Get
        End Get
        Set(ByVal value As String)
        End Set
    End Property

    Public Function EineMethode() As Integer Implements ITest.EineMethode
    End Function

    Public Function ZweiteMethode() As String Implements ITest.ZweiteMethode
    End Function

    Public Overrides Function ToString() As String Implements ITest.ToString
        Return MyBase.ToString
    End Function

End Class
```

Schnittstellen, die Schnittstellen implementieren

Auch Schnittstellen können vom Prinzip her Schnittstellen implementieren. Allerdings tun sie das nicht auf die gleiche Weise, wie Klasse Schnittstellen implementieren. Stattdessen können Schnittstellen von anderen Schnittstellen erben – genau wie Klassen von anderen Klassen erben können und dabei deren komplette Funktionalität übernehmen.

Dass es aber bei Schnittstellen keine Funktionalitäten, sondern nur virtuelle Methoden, Eigenschaften oder Ereignisse gibt, erben Schnittstellen von anderen Schnittstellen nur deren Implementierungsvorschriften. Oder anders und einfacher ausgedrückt: Eine Schnittstelle, die von einer anderen Schnittstelle erbt, enthält sämtliche ihrer Vorschriften und die der Schnittstelle, von der sie erbt. Und darüber hinaus können Schnittstellen auch nur von Schnittstellen erben, nicht von Klassen (was aber wahrscheinlich auch ohne explizite Erwähnung klar ist, denn Schnittstellen dürfen ja anders als Klassen kein Code enthalten).

Sie können sich auch diese Zusammenhänge an einem Beispiel verdeutlichen. Ergänzen Sie das Modul aus dem vorherigen Beispiel um folgenden Code:

```
Interface IBaseInterface
    Property EineEigenschaft() As String
End Interface

Interface IMoreComplexInterface
    Inherits IBaseInterface

    Property ZweiteEigenschaft() As String
End Interface
```

Die Schnittstelle IMoreComplexInterface beinhaltet in diesem Fall die Vorschriften beider Schnittstellendefinitionen (der eigenen und der, von der sie erbt).

Wenn Sie diese »vereinigte« Schnittstelle in einer Klasse implementieren, ergibt sich – dank Codeergänzung durch den Editor ist das ja schnell getan – nach dem Betätigen von  hinter Implements IMoreComplexInterface im folgenden Code folgendes Ergebnis:

```
Public Class ComplexClass
    Implements IMoreComplexInterface

    Public Property EineEigenschaft() As String Implements IBaseInterface.EineEigenschaft
        Get
            End Get
            Set(ByVal value As String)
                End Set
            End Property

        Public Property ZweiteEigenschaft() As String Implements IMoreComplexInterface.ZweiteEigenschaft
            Get
                End Get
                Set(ByVal value As String)
                    End Set
                End Property
        End Class
```

Einbinden mehrerer Schnittstellen in eine Klasse

Mehrfachvererbung ist in der .NET-Infrastruktur nicht vorgesehen. Bei der Mehrfachvererbung entsteht eine neue Klasse aus mehr als einer Basisklasse und dabei übernimmt die erbende Klasse die Funktionalität von beiden Basisklassen.

Allerdings können Sie auch in Visual Basic .NET mehr als eine Schnittstelle in einer Klasse implementieren, und auch wenn das nicht so »toll«⁴ wie Mehrfachvererbung ist, so können Sie sich mit der Einbindung mehrerer Schnittstellen wenigstens ein wenig behelfen. Man könnte das Einbinden mehrerer Schnittstellen in einer Klasse sozusagen als »Mehrfachvererbung Light« bezeichnen.

Das Einbinden mehrerer Schnittstellen ist ein Kinderspiel. Das vorherige Beispiel der Schnittstellendefinitionen ändern wir dazu folgendermaßen ab:

```
Interface IBaseInterface
    Property EineEigenschaft() As String
End Interface

Interface IMoreComplexInterface
    'Inherits wurde entfernt, beide Schnittstellen liegen somit auf "gleicher Ebene".
    Property ZweiteEigenschaft() As String
End Interface

Public Class ComplexClass
    Implements IBaseInterface, IMoreComplexInterface

    Public Property EineEigenschaft() As String Implements IBaseInterface.EineEigenschaft
        Get
        End Get
        Set(ByVal value As String)
            End Set
        End Property

        Public Property ZweiteEigenschaft() As String Implements IMoreComplexInterface.ZweiteEigenschaft
            Get
            End Get
            Set(ByVal value As String)
                End Set
            End Property
        End Class
```

⁴ Mehrfachvererbung ist nur auf den ersten Blick etwas Feines, und es wird sie in Visual Basic sehr wahrscheinlich nicht geben, da selbst einer der Erfinder von C++, der Herr Stroustrup, das inzwischen sehr skeptisch sieht. (In Java ist es ja auch *absichtlich* weggelassen worden.) Mehrfachvererbung sorgt sogar für unentscheidbare Konflikte, die der Compiler willkürlich auflösen muss. (Zwei Methoden gleichen Namens in beiden Klassen mit Implementierung, usw.) Stroustrup dazu: »[...] Mehrfachvererbung bleibt verständlich, wenn nur eine Basisklasse wirklich Member-Funktionen implementiert und die anderen nur pure virtuelle Funktionen deklarieren [...]«.

Am Code der einbindenden Klasse muss in diesem Fall nur die in Fettschrift gesetzte Zeile geändert werden. An dem restlichen Code, der die Schnittstellelemente den Klassenelementen zuordnet, müssen keinerlei Änderungen vorgenommen werden.

Die Methoden und Eigenschaften von Object

Object selber stellt einige Grundmethoden und -eigenschaften zur Verfügung, die damit jede neue Klasse automatisch erbt. Der Hintergrund ist, dass die Entwickler des .NET Frameworks damit sichergestellt haben, dass jedes Objekt über eine gewisse Grundfunktionalität verfügt, auf deren Vorhandensein sich andere Klassen hundertprozentig verlassen können.

Polymorphie am Beispiel von `ToString` und der `ListBox`

Ein Beispiel dazu entführt uns für einen Moment in die Windows Forms-Entwicklung – genauer gesagt zu einer simplen Anwendung, die eine `ListBox` verwendet.

BEGLEITDATEIEN

Das Grundgerüst für die folgenden Experimente finden Sie unter:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\ToStringDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Projekt besteht aus zwei »Elementen«, dem Formular mit der Steuerung der einzigen Schaltfläche sowie einer Klasse, die die Aufgabe hat, Kontaktdaten in Form von Namen und Vornamen zu speichern:

```
'Der Formularcode
Public Class frmMain

    Private Sub btnKontaktHinzufügen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnKontaktHinzufügen.Click

        'Neues Kontaktobjekt instanzieren und es mit den Inhalten
        'der Textbox füllen
        Dim locKontakt As New Kontakt(txtVorname.Text, txtNachname.Text)

        'Der Listbox hinzufügen
        lstkontakte.Items.Add(locKontakt)
    End Sub
End Class

'Die Kontakt-Klasse
Public Class Kontakt

    Private myVorname As String
    Private myNachname As String
```

```
Sub New(ByVal Vorname As String, ByVal Nachname As String)
    myVorname = Vorname
    myNachname = Nachname
End Sub

Public Property Vorname() As String
    Get
        Return myVorname
    End Get
    Set(ByVal value As String)
        myVorname = value
    End Set
End Property

Public Property Nachname() As String
    Get
        Return myNachname
    End Get
    Set(ByVal value As String)
        myNachname = value
    End Set
End Property
End Class
```

Interessant an dieser Stelle ist die fett geschriebene Codezeile im Listing. Sobald der Benutzer Vor- und Nachnamen in die TextBox-Steuerelemente eingegeben und die Schaltfläche betätigt hat, legt das Programm eine neue Instanz der Kontakt-Klasse an und fügt diesen Kontakt dann mithilfe der Add-Methode der Items-Auflistung der ListBox den Listbox-Elementen hinzu. Diese Vorgehensweise ist möglich, da anders, als noch in Visual Basic 6.0, die Add-Methode der ListBox nicht nur String-Elemente sondern beliebige Objekte aufnimmt.

Das Problem: Wenn die ListBox in ihrer Items-Auflistung Objekte vom Typ Object aufnimmt, kann man ihr sämtliche Typen übergeben (so auch unser Kontakt-Objekt). Das ist möglich, da, wie wir ja schon wissen, eine Objektvariable nicht auf den »eigenen Typ« sondern auch auf jeden abgeleiteten Typ verweisen kann (siehe Abschnitt »Polymorphie« und folgende ab Seite 466). Und da alle Klassen implizit von Object abgeleitet werden, basiert letzten Endes jede neu erfundene Klasse auch auf Object. Wann immer Ihnen also ein Parameter vom Typ Object begegnet, können Sie jeden beliebigen Typ übergeben. So weit, so gut.

Doch wer oder was sorgt nun dafür, dass ein Sinn ergebender Text in der ListBox angezeigt wird? In der jetzigen Ausbaustufe des Programms zeigt die ListBox jedenfalls noch nicht die gewünschten Ergebnisse an, was Sie schnell herausfinden können, wenn Sie das Programm laufen lassen:

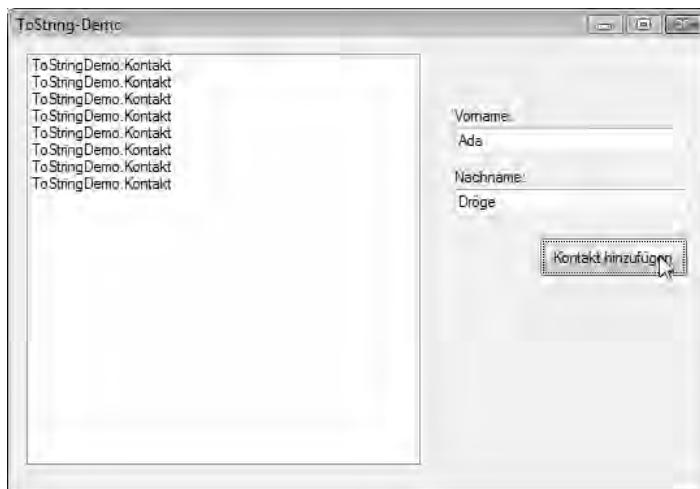


Abbildung 15.19 Das Hinzufügen eines Objektes zur ListBox führt zunächst nur zur Anzeige des voll qualifizierten Objektnamens

Das Ergebnis entspricht sicherlich nicht dem erwarteten. Aber das kann es auch gar nicht, und Sie werden sehen, warum das so ist, wenn Sie erfahren, wie die `ListBox` arbeitet, um den Text eines Objektes zu ermitteln.

Vielleicht ahnen Sie es schon: Wenn die `ListBox` ihren Anzeigebereich mit Texten füllt, ruft sie die `ToString`-Funktion eines jeden Elementes auf,⁵ das ihrer `Items`-Auflistung hinzugefügt wurde. Das kann sie machen, ohne befürchten zu müssen, dass ein Objekt, das sie beinhaltet, kein `ToString` hat, denn `ToString` ist Bestandteil von `Object`. `Object` ist wiederum die Basisklasse aller Basisklassen und damit verfügt jedes andere Objekt auch über eine `ToString`-Funktion.

Die Standardimplementierung von `ToString` in `Object` sorgt allerdings lediglich dafür, dass der Typname des Objektes ausgegeben wird. Die Standardimplementierung der `ToString`-Funktion von `Object` schaut nämlich folgendermaßen aus:

```
Public Overridable Function ToString() As String
    Return Me.GetType.ToString()
End Function
```

`GetType` liefert den Typ eines Objektes zurück, der wiederum durch `ToString` in eine lesbare Zeichenfolge übersetzt wird. Und auf diese Weise zeigt `ToString` für jedes Objekt dessen Typnamen an – der Polymorphie sei Dank.

⁵ Um genau zu sein: Sie ruft `GetItemText` ihrer Basisklasse `ListControl` auf, und *die* ermittelt den Text durch `ToString`, falls es sich beim Objekt nicht um ein Steuerelement handelt, das an eine andere Eigenschaft gebunden werden kann. In diesem Fall würde `GetItemText` den Text des Objektes per `ToString` ermitteln, der an das Steuerelement gebunden ist (oder einen Leer-String zurückliefern, falls es keine Bindung gibt). Aus diesem Grund sehen Sie auch einen Leer-String in der Liste, falls Sie der `Items`-Auflistung die Instanz irgendeines Steuerelements übergäben (wie beispielsweise eine Schaltfläche oder das Formular selbst).

Damit `ToString` einer Klasse beispielsweise bei der Darstellung in einer `ListBox` einen Sinn ergebenden Text ausgibt, müssen Sie die `ToString`-Funktion in der entsprechenden Klasse überschreibend implementieren. Würden Sie also die Klasse `Kontakt` um folgenden Code ergänzen,

```
Public Overrides Function ToString() As String
    Return Nachname & ", " & Vorname
End Function
```

wäre das Ergebnis wie erwartet, was auch die folgende Abbildung zeigt:

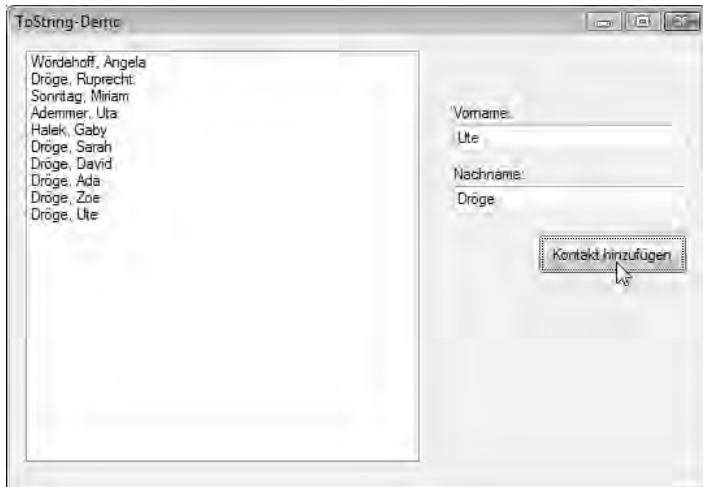


Abbildung 15.20 Durch das Überschreiben von `ToString()` in der Klasse, deren Objektinstanz der Liste hinzugefügt wird, lässt sich der dargestellte Listentext steuern

Prüfen auf Gleichheit von Objekten mit `Object.Equals` oder dem `Is/ IsNot`-Operator

Die deutsche Sprache wird immer mehr vereinfacht. Man sieht das an Wörtern wie beispielsweise »das Gleiche« und »dasselbe«. Inzwischen sind diese laut Duden dasselbe (oder das Gleiche?) – aber das war nicht immer so. Dasselbe bezeichnete einst buchstäblich ein und dasselbe. Zwei verschiedene Leute konnten nicht gleichzeitig mit demselben Auto fahren – es sei denn, sie hätten es in der Mitte durchgeschnitten (und irgendwie fahrbereit gemacht) oder es wäre ein Fahrlehrer mit seinem Schüler, was einfacher wäre. Wohl aber mit dem gleichen: Dann wären es unterschiedliche Autos gewesen, die sich einfach nur sehr ähnlich waren.

Diesen Unterschied sollten Sie für das bessere Verständnis der folgenden Erklärung kennen.

Es gibt für jedes Objekt eine Methode namens `Equals`, die überprüft, ob es sich bei einer Objektinstanz, die durch eine Objektvariable referenziert wird, um dieselbe handelt, die durch eine andere Objektvariable referenziert wird.

WICHTIG Es wird dabei NICHT überprüft, ob der Inhalt zweier Objekte der gleiche ist, falls die Objektvariablen auf zwei verschiedene Instanzen verweisen würden.

Ein Beispiel soll das verdeutlichen, und zum besseren Verständnis sollten Sie zuvor nochmals einen Blick auf Abbildung 15.9 werfen.

HINWEIS Statt die Equals-Methode zu verwenden, der Sie ein anderes Objekt zum Vergleichen übergeben können, haben Sie auch die Möglichkeit den Is-Operator (bzw. den IsNot-Operator) einzusetzen. Das folgende Beispiel demonstriert den Einsatz beider Möglichkeiten.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\EqualsDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module mdlMain

Sub Main()

    'Instanziieren mit New und dadurch
    'Speicher für das Kontakt-Objekt
    'auf dem Managed Heap anlegen
    Dim objVarKontakt As New Kontakt

    'Daten zuordnen
    With objVarKontakt
        .Nachname = "Halek"
        .Vorname = "Sarah"
        .Plz = "99999"
        .Ort = "Musterhausen"
    End With

    'objVarKontakt2 zeigt auf dasselbe Objekt;
    'die referenzierte Instanz ist dieselbe!
    Dim objVarKontakt2 As Kontakt
    objVarKontakt2 = objVarKontakt

    'objVarKontakt3 zeigt auf ein gleiches Objekt;
    'die referenzierte Instanz ist nicht dieselbe, nur die gleiche!
    Dim objVarKontakt3 As New Kontakt
    'Daten zuordnen
    With objVarKontakt
        .Nachname = "Halek"
        .Vorname = "Sarah"
        .Plz = "99999"
        .Ort = "Musterhausen"
    End With

    'Der Beweis dafür:
    Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt2 ist "" " & _
        objVarKontakt.Equals(objVarKontakt2))
```

```

Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt3 ist "" " & _
    objVarKontakt.Equals(objVarKontakt3))

Console.WriteLine()

'Alternativ durch das IS-Schlüsselwort:
Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt2 ist "" " & _
    (objVarKontakt Is objVarKontakt2))

Console.WriteLine("Die Aussage ""objVarKontakt entspricht objVarKontakt3 ist "" " & _
    (objVarKontakt Is objVarKontakt3))

Console.WriteLine()
Console.WriteLine("Zum Beenden Taste drücken")
Console.ReadKey()
End Sub

End Module

Class Kontakt

    Public Nachname As String
    Public Vorname As String
    Public Plz As String
    Public Ort As String

End Class

```

Diese kleine Demo ist weitestgehend selbsterklärend. Sie definiert drei Objektvariablen auf Basis der Klasse Kontakt. Sie instanziert daraus zwei Objekte und füttert sie mit den gleichen Daten. Die dritte Objektvariable verweist allerdings auf dieselbe Instanz wie die erste Objektvariable. Deswegen ist ein Vergleich mit Equals oder Is wahr.

TIPP Die Regel lautet: Verweisen zwei Objektvariablen auf die gleiche Instanz, und speichern sie deswegen die gleichen Zeiger auf die eigentlichen Daten im Managed Heap, ergeben Is oder Equals als Ergebnis True, ansonsten False. Die Daten selbst (die Instanzen der Objekte) werden bei diesem Vergleich nicht verwendet.

```

Das Ausgabergebnis dieses Programm sieht demzufolge folgendermaßen aus:
Die Aussage "objVarKontakt entspricht objVarKontakt2 ist " True
Die Aussage "objVarKontakt entspricht objVarKontakt3 ist " False

Die Aussage "objVarKontakt entspricht objVarKontakt2 ist " True
Die Aussage "objVarKontakt entspricht objVarKontakt3 ist " False

Zum Beenden Taste drücken

```

Equals, Is und IsNot im praktischen Entwicklungseinsatz

Die Frage, die sich stellt, lautet wie bei allem, was man neu lernt: Wozu brauche ich das? Was das Testen auf identische Objekte anbelangt, ist die Antwort einfach: an fast jeder Stelle.

Das geht spätestens dann los, wenn Sie ein bestimmtes Objekt in einer Auflistung wie beispielsweise der Items-Auflistung der `ListBox` suchen. Sie möchten beispielsweise einen bestimmten Eintrag aus der Liste einer `ListBox` löschen. Zu diesem Zweck würden Sie die `Remove`-Methode verwenden, die an der `Items`-Eigenschaft der `ListBox` »hängt«. Die `Remove`-Methode nimmt jedes beliebige Objekt entgegen und ist natürlich nicht in der Lage, die Inhalte der Objektinstanzen zu vergleichen, die durch ein `ListBoxItem` repräsentiert werden und dem Objekt entsprechen, das Sie entfernen möchten. Sie ist aber in der Lage herauszufinden, ob zwei Objektvariablen auf die gleiche Instanz verweisen, und daran erkennt `Remove`, welches Objekt es zu entfernen hat.

Die folgende Demo zeigt dies im praktischen Einsatz. Mit dem schon bekannten Beispiel können Sie eine `ListBox` mit Kontaktdaten füllen. Doch ferner verfügen Sie über eine weitere Schaltfläche, mit der Sie einen selektierten Eintrag wieder aus der Liste entfernen können.

Den Verweis auf ein selektiertes Objekt einer `ListBox` können Sie mit `SelectedItem` der `Items`-Auflistung in Erfahrung bringen. Liefert `SelectedItem` den Wert `Nothing` zurück, handelt es sich um einen so genannten Null-Verweis; eine Objektvariable zeigt in diesem Fall auf keine Instanz mit Daten.

Liefert `SelectedItem` jedoch einen »Wert« zurück, der eben nicht `Nothing` ist, dann war ein Objekt der Liste selektiert. Dieses Objekt können Sie dann in einer Objektvariablen speichern und es der `Remove`-Methode der `ListBoxItems`-Auflistung übergeben:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\EqualsPraxis

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Der folgende Auszug zeigt die Ereignisbehandlungsroutine der Schaltfläche, die ein Element aus der Liste löscht, sofern es selektiert war:

```
Private Sub btnKontaktLöschen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnKontaktLöschen.Click
    Dim locKontakt As Object
    locKontakt = lstKontakte.SelectedItem
    If locKontakt IsNot Nothing Then
        lstKontakte.Items.Remove(locKontakt)
    End If
End Sub
```

Intern macht die `Remove`-Methode genau das gerade Gesagte: Sie iteriert (durchläuft) durch die Elemente und stellt fest, ob das zu löschen Element dem gerade bearbeiteten Element der Liste entspricht. Dabei führt sie die Prüfung auf Entsprechung ebenfalls mit `Is` bzw. der `Equals`-Methode durch.

Übersicht über die Eigenschaften und Methoden von Object

Die folgende Tabelle fasst die Methoden der Klasse Object zusammen:

Member	Beschreibung
Equals	Stellt fest, ob das aktuelle Objekt, dessen <i>Equals</i> -Member verwendet wird, mit dem angegebenen <i>Object</i> identisch ist. Zwei Objekte sind dann identisch, wenn ihre Instanzen (das heißt der Datenbereich, auf den sie zeigen) übereinstimmen. <i>ValueType</i> überschreibt diese Methode, und vergleicht die einzelnen Member. Da Strukturen automatisch von <i>ValueType</i> ableiten, gilt dieses Verhalten für alle Strukturen.
GetHashCode	Produziert einen Hashcode (eine Art Identifizierungsschlüssel) auf Grund des Objektinhaltes. Dieser Hashcode wird beispielsweise dann verwendet, wenn ein Objekt in einer Tabelle (einem Array) gesucht werden muss. Daher sollte <i>GetHashCode</i> nach Möglichkeit eindeutige Werte liefern, aber gleichzeitig auch vom Inhalt abhängige Werte als Grundlage der Hashcode-Berechnung miteinbeziehen. Der in <i>Object</i> implementierte Algorithmus garantiert weder Eindeutigkeit noch Konsistenz – Sie sind also angehalten, nach Möglichkeit eigene Hashcode-Algorithmen für abgeleitete Objekte zu entwickeln und diese Methode zu überschreiben, wenn Sie Ihr Objekt des Öfteren in Hash-Tabellen speichern wollen.
GetType	Ruft den aktuellen Typ der Instanz als Type-Objekt ab. Mehr dazu erfahren Sie im Kapitel 25.
(Shared) ReferenceEquals	Diese statische Methode entspricht der <i>Equals</i> -Methode, übernimmt die beiden zu vergleichenden Objekte aber als Parameter. Da diese Methode statischer Natur ist, können Sie sie nur über den Typnamen aufrufen (<i>Object.ReferenceEquals</i>).
ToString()	Liefert eine Zeichenkette zurück, die das aktuelle Objekt beschreibt. In der ursprünglichen Version ist das wörtlich zu nehmen; <i>ToString</i> liefert nämlich den Klassennamen zurück, wenn Sie diese Funktion nicht überschreiben. <i>ToString</i> sollte nach Möglichkeit den Inhalt des Objektes – wenigstens teilweise – als Zeichenkette zurückliefern.
Finalize	Wenn die <i>Finalize</i> -Methode eines Objektes aufgerufen wird, ist die Quelle des Aufrufs der Garbage Collector. Er signalisiert dem Objekt durch diesen Aufruf, dass es im Rahmen der »Müllabfuhr« im Begriff ist, entsorgt zu werden, und das Objekt hat in diesem Rahmen die Möglichkeit, Ressourcen freizugeben, die es nicht mehr benötigt. Mehr dazu finden Sie in Kapitel 18.
MemberwiseClone	Erstellt eine so genannte »flache Kopie« von dem Objekt, das die Methode beherbergt. Wenn Sie <i>MemberwiseClone</i> aufrufen, dann legt diese Methode eine Kopie aller Wertetyp-Member (dazu später mehr) an und stellt diese in einer weiteren Objektinstanz zur Verfügung. Für Referenztypen werden nur Adresskopien erstellt – sie zeigen anschließend also auf dieselben Objekte im Managed Heap, auf die auch die Referenztypen des Originals zeigen.

Tabelle 15.2 Die Beschreibung der Object-Member

Shadowing (Überschatten) von Klassenprozeduren

Visual Basic kennt eine weitere Version des Ersetzens von Prozeduren in einer Basisklasse durch andere gleichen Namens einer abgeleiteten Klasse. Diesen Vorgang nennt man das so genannte »Shadowing« oder »Überschatten« von Elementen.

Wenn Sie in einer Basisklasse eine Funktion definiert haben, dann kann eine Funktion gleichen Namens in einer abgeleiteten Klasse das Original schlichtweg ausblenden, wie im folgenden Beispiel.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Shadowing01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module mdlMain

    Sub Main()
        Dim locBasisinstanz As New Basisklasse
        Console.WriteLine(locBasisinstanz.EineFunktion().ToString())

        locBasisinstanz = New AbgeleiteteKlasse
        Console.WriteLine(locBasisinstanz.EineFunktion().ToString())

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

End Module

Public Class Basisklasse

    Protected test As Integer

    Sub New()
        test = 10
    End Sub

    Public Function EineFunktion() As Integer
        Return test * 2
    End Function

End Class

Public Class AbgeleiteteKlasse
    Inherits Basisklasse

    Public Shadows Function EineFunktion() As Integer
        Return test * 3
    End Function

End Class
```

Ohne zunächst ganz genau auf die beiden Klassenimplementierungen zu achten, würden Sie wahrscheinlich annehmen, dass das Modul zunächst den Wert 20 und in der folgenden Zeile den Wert 30 ausgibt.

Doch bei diesem Beispiel handelt es sich nur augenscheinlich um die Anwendung von Polymorphie, denn die einzige Funktion der Basisklasse ist weder durch das Schlüsselwort `Overridable` als überschreibbar definiert, noch versucht die gleiche Funktion der abgeleiteten Klasse, diese Funktion mit `Overrides` zu überschreiben.

Bei genauerer Betrachtung zeigt Visual Basic für EineFunktion der abgeleiteten Klasse eine Warnmeldung an, etwa wie in Abbildung 15.21 zu sehen:



Abbildung 15.21 In diesem Beispiel blendet eine Funktion der abgeleiteten Klasse die der Basisklasse aus

Visual Basic gibt Ihnen hier eine Warnmeldung aus, kompiliert das Programm aber dennoch. Die vermeintliche Fehlermeldung ist in diesem Fall nämlich nur eine Warnmeldung und macht Sie darauf aufmerksam, dass die Funktion von einer anderen überschattet wird.

Das hat Auswirkungen auf das Verhalten der Klasse. Denn obwohl die Objektvariable eine Instanz der abgeleiteten Instanz enthält, wird dennoch die Funktion der Basisklasse aufgerufen. Shadows unterbricht die Erbfolge der Klasse an dieser Stelle und stellt sicher, dass eine Funktion, die nicht zum Überschreiben markiert ist, auch tatsächlich nicht überschrieben werden kann.

Sie werden diese Warnmeldung übrigens los, indem Sie das Schlüsselwort Shadows vor die Funktion in der abgeleiteten Klasse setzen.

Shadows als Unterbrecher der Klassenhierarchie

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Shadowing02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```

Option Explicit On
Option Strict On

Module mdlMain

```

```
Sub Main()
    Dim locVierteKlasse As New VierteKlasse
    Dim locErsteKlasse As ErsteKlasse = locVierteKlasse
    Dim locZweiteKlasse As ZweiteKlasse = locVierteKlasse
    Dim locDritteKlasse As DritteKlasse = locVierteKlasse

    Console.WriteLine(locErsteKlasse.EineFunktion)
    Console.WriteLine(locZweiteKlasse.EineFunktion)
    Console.WriteLine(locDritteKlasse.EineFunktion)
    Console.WriteLine(locVierteKlasse.EineFunktion)

    Console.WriteLine()
    Console.WriteLine("Taste drücken zum Beenden!")
    Console.ReadKey()

End Sub
End Module

Public Class ErsteKlasse

    Public Overridable Function EineFunktion() As String
        Return "Erste Klasse"
    End Function

End Class

Public Class ZweiteKlasse
    Inherits ErsteKlasse

    Public Overrides Function EineFunktion() As String
        Return "Zweite Klasse"
    End Function
End Class

Public Class DritteKlasse
    Inherits ZweiteKlasse

    Public Overrides Function EineFunktion() As String
        Return "Dritte Klasse"
    End Function

End Class

Public Class VierteKlasse
    Inherits DritteKlasse
    Public Overrides Function EineFunktion() As String
        Return "Vierte Klasse"
    End Function

End Class
```

Was glauben Sie, kommt heraus, wenn Sie das Beispiel laufen lassen? Die verschiedenen Objektnamen und Klassennamen mögen anfangs ein wenig verwirren, aber das Ergebnis ist klar: Das Programm gibt viermal den Text »vierte Klasse« aus. Klar, denn vierte Klasse wird ein einziges Mal instanziert und durch jede andere Variable referenziert.

Jede der anderen Objektvariablen ist über eine Klasse der Klassenerfolge definiert und kann damit auch – wie Sie es an vielen Beispielen auf den vergangenen Seiten dieses Kapitels schon kennengelernt haben – auf jede Instanz einer abgeleiteten Klasse verweisen.

Jetzt nehmen Sie eine kleine Veränderung an der zweiten Klasse vor,

```
Public Class DritteKlasse
    Inherits ZweiteKlasse

    '''Overrides'' wurde gegen "Overridable Shadows" ausgetauscht:
    Public Overridable Shadows Function EineFunktion() As String
        Return "Dritte Klasse"
    End Function

End Class
```

und raten Sie, was beim erneuten Start des Programms passiert. Die Ausgabe lautet:

```
Zweite Klasse
Zweite Klasse
Vierte Klasse
Vierte Klasse

Taste drücken zum Beenden!
```

Hätten Sie es gewusst? Dabei ist das Ergebnis gar nicht so schwer zu verstehen:

Im Prinzip hat `EineFunktion` der dritten und vierten Klasse überhaupt nichts mehr mit den ersten beiden Klassen zu tun. Durch `Shadows` in der dritten Klasse wird die Funktion komplett neu implementiert. Aus diesem Grund ist wie bei einer komplett anderen Funktion, die in `DritteKlasse` »dazukommt«, auch ein erneutes `Overridable` notwendig, denn anderenfalls könnte `VierteKlasse` die Funktion gar nicht mit `Overrides` überschreiben.

Verwirrend mag ein wenig sein, dass die erste Klasse den Text »`Zweite Klasse`« ausgibt. Doch welchen anderen Text soll sie sonst ausgeben? »`Erste Klasse`«, mag man auf den ersten Blick denken, doch das ist falsch, denn das würde gegen das Prinzip der Polymorphie verstößen. »`Vierte Klasse`« kann nicht ausgegeben werden, denn diese Funktion ist aus Sicht von `ErsteKlasse` gesehen nicht erreichbar. In `DritteKlasse` wird diese Funktion schlicht und ergreifend durch eine komplett neue Version ersetzt. Das Framework rettet also, was zu retten ist, und versucht in der Erbfolge soweit wie möglich nach vorne zu gehen – und damit ist `ZweiteKlasse` die letzte Funktion, die durch Polymorphie erreichbar ist, bevor die Erbfolge durch `Shadows` in `DritteKlasse` unterbrochen wird.

Dieses Verhalten ist durchaus erwünscht, denn wenn Sie nicht wollen, dass ein Element einer Klasse überschrieben wird, dann lässt es sich auch nicht überschreiben. Die CLR garantiert immer, dass eine nicht überschreibbare Funktion ihre ursprünglichen Fähigkeiten behält, selbst wenn sie andere Funktionen in abgeleiteten Klassen mit gleichem Namen überschatten.

Intern gibt es zwei verschiedene Versionen von `EineFunktion`, und wenn Sie das Beispielprogramm wie folgt abändern, wird klar, was eigentlich passiert.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Shadowing03

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Option Explicit On
Option Strict On

Module mdlMain
    Sub Main()
        Dim locVierteKlasse As New VierteKlasse
        Dim locErsteKlasse As ErsteKlasse = locVierteKlasse
        Dim locZweiteKlasse As ZweiteKlasse = locVierteKlasse
        Dim locDritteKlasse As DritteKlasse = locVierteKlasse

        Console.WriteLine(locErsteKlasse.EineFunktion_a)
        Console.WriteLine(locZweiteKlasse.EineFunktion_a)
        Console.WriteLine(locDritteKlasse.EineFunktion_b)
        Console.WriteLine(locVierteKlasse.EineFunktion_b)

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()

    End Sub
End Module

Public Class ErsteKlasse

    Public Overridable Function EineFunktion_a() As String
        Return "Erste Klasse"
    End Function

End Class

Public Class ZweiteKlasse
    Inherits ErsteKlasse

    Public Overrides Function EineFunktion_a() As String
        Return "Zweite Klasse"
    End Function

End Class

Public Class DritteKlasse
    Inherits ZweiteKlasse

    Public Overridable Function EineFunktion_b() As String
        Return "Dritte Klasse"
    End Function
End Class
```

```

Public Class VierteKlasse
    Inherits DritteKlasse

    Public Overrides Function EineFunktion_b() As String
        Return "Vierte Klasse"
    End Function

End Class

```

Sonderform »Modul« in Visual Basic

Module gibt es bei allen bislang existierenden .NET-Programmiersprachen nur in Visual Basic. Und auch hier ist ein Modul nichts weiter als eine Mogelpackung, denn das, was als Modul bezeichnet wird, ist im Grunde genommen nichts anderes als eine abstrakte Klasse mit statischen Methoden, Eigenschaften und Membern.

Halten wir fest:

- Ein Modul ist nicht instanziierbar. Eine abstrakte Klasse auch nicht.
- Ein Modul kann keine überschreibbaren Prozeduren zur Verfügung stellen. Die statischen Prozeduren einer Klasse können das auch nicht.
- Ein Modul kann nur Prozeduren zur Verfügung stellen, auf die aber nur ohne Instanzobjekt direkt zugegriffen werden kann. Das gleiche gilt für die statischen Prozeduren einer abstrakten Basisklasse.

Es gibt aber auch feine Unterschiede: So können Module beispielsweise keine Schnittstellen implementieren; das können zwar abstrakte Klassen, aber sie können keine statischen Schnittstellenelemente definieren. Insofern ist dieser scheinbare Unterschied in Wirklichkeit gar keiner. Ein Modul kann auch nur auf oberster Ebene definiert und nicht in einem anderen geschachtelt werden.

Module setzen Sie bei der OOP vorschlagsweise so wenig wie möglich ein, denn sie widersprechen dem Anspruch von .NET, möglichst wiederverwendbaren Code zu schaffen.

Hier im Buch finden Sie aus diesem Grund Module nur, wenn es um »Quick-And-Dirty«-Projekte geht, bei denen beispielsweise eine Konsolen-Anwendung Tests durchzuführen hat oder »mal eben« etwas demonstrieren soll, genauso, wie Sie es in vergangenen Kapiteln bereits erlebt haben.

Singleton-Klassen und Klassen, die sich selbst instanziieren

Stellen Sie sich vor, Sie möchten eine Klasse entwerfen, die beispielsweise die Funktionen eines Bildschirms oder eines Druckers steuert. Das Besondere daran ist, dass Sie eine Kontrolle über die Instanziierung dieser Klasse benötigen. Es reicht nicht aus, der Klasse selbst zu überlassen, wie oft sie sich instanziert – einen bestimmten Drucker gibt es nur ein einziges Mal, und nach einer Instanz sollte Schluss sein.

Eine abstrakte Klasse mit statischen Prozeduren (meinetwegen auch ein Modul) könnte vielleicht eine Alternative dazu sein – doch das Problem dabei ist: Weder die Funktionen eines Moduls noch die statischen Funktionen einer abstrakten Klasse können Sie in anderen Klassen überschreiben.

Die Lösung dazu sind so genannte Singleton-Klassen. Singleton-Klassen sind, anders als die Klassenvarianten, die Sie bislang kennengelernt haben, keine »eingebauten« Klassentypen der FCL. Sie müssen sie selber entwickeln – doch das ist einfacher, als Sie denken.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 15\\Singleton

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Imports System.Threading

Module mdlMain

    Dim varModule As Integer = 5

    Sub New()
    End Sub

    Sub Main()
        Dim locSingleton As Singleton = Singleton.GetInstance()
        Dim locSingleton2 As Singleton = Singleton.GetInstance()

        Console.WriteLine(locSingleton Is locSingleton2)
        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

    Property test() As Integer
        Get
        End Get
        Set(ByVal Value As Integer)
        End Set
    End Property

End Module

Class Singleton

    Private Shared locSingleton As Singleton
    Private Shared locMutex As New Mutex

    'Konstruktor ist privat,
    'damit kann die Klasse nur von "sich selbst" instanziiert werden
    Private Sub New()
    End Sub
```

```

'GetInstance gibt eine Singleton-Instanz zurück
'Nur durch diese Funktion kann die Klasse instanziert werden
Public Shared Function GetInstance() As Singleton
    'Vorgang Thread-sicher machen
    'Wartet, bis ein anderer Thread, der diese Klasse
    'instanziert, damit fertig ist
    locMutex.WaitOne()
    Try
        If locSingleton Is Nothing Then
            locSingleton = New Singleton
        End If
    Finally
        'Instanziieren abgeschlossen,
        '...kann weiter gehen...
        locMutex.ReleaseMutex()
    End Try

    'Instanz zurückliefern
    Return locSingleton
End Function

End Class

```

Ihnen wird als Erstes auffallen, dass diese Klasse nicht direkt instanziert werden kann, denn ihr Konstruktor ist privat. Wie also kommen Sie dann an überhaupt an eine Instanz der Klasse?

Die Antwort auf diese Frage liegt in der statischen Funktion GetInstance. Sie sorgt dafür, dass die Klasse sich selbst instanziert, wenn es erforderlich ist. Gleichzeitigachtet sie darauf, dass der Instanziierungsvorgang der Klasse thread-sicher ist: Das Framework erlaubt es nämlich, wahrhaftiges Multitasking in eigenen Programmen zu implementieren. Dieser Vorgang bezeichnet Zustände, bei denen mehrere Aufgaben innerhalb eines Programms (oder mehrerer Programme) gleichzeitig ausgeführt werden können. Damit ein weiterer Instanziierungsversuch einer Singleton-Klasse nicht ausgerechnet dann passiert, wenn ein anderer Thread fast (aber eben noch nicht ganz) mit der Instanziierung fertig ist – beide Threads sich sozusagen »mittendrin« treffen würden und es damit doch die zu verhindernden zwei Instanzen gäbe –, nutzt die Klasse eine Instanz von Mutex⁶, um dieses Auftreffen zu vermeiden.

Solange Sie keine Multithreading-Programmierung anwenden, brauchen Sie jedoch keinen Gedanken daran zu verschwenden. Mehr zu diesem Thema erfahren Sie im Threading-Kapitel später in diesem Buch (ab Kapitel 44).

Das Hauptprogramm prüft die Funktionalität der Singleton-Klasse. Es holt sich eine Instanz mit GetInstance und speichert sie in einer Objektvariablen. Den gleichen Vorgang wiederholt es anschließend mit einer weiteren Objektvariablen und vergleicht die beiden »Instanzen« anschließend mit Is.

Und in der Tat: Das Programm gibt True auf dem Bildschirm aus, denn die Singleton-Klasse hat nur *eine* Instanz ihrer selbst kreiert – beide Objektvariablen zeigen also auf die gleiche Instanz.

⁶ Mutex, abgeleitet von »mutual exclusion«, etwa » gegenseitiger Ausschluss«.

Kapitel 16

Entwickeln von Wertetypen

In diesem Kapitel:

Erstellen von Wertetypen mit Structure am praktischen Beispiel	520
Unterschiedliche Verhaltensweisen von Werte- und Referenztypen	526
Performance-Unterschiede zwischen Werte- und Referenztypen	533

Kapitel 13 ist ja bereits schon grundsätzlich auf die Möglichkeit eingegangen, neben den aus Klassen entstehenden Referenztypen auch Wertetypen mit Structure zu entwickeln. Dieses Kapitel möchte sich diesem Thema noch mal etwas ausführlicher widmen, und vor allen Dingen ein wenig tiefer in die Praxis eintauchen: Es zeigt nicht nur die Entwicklung einer wiederverwendbaren Komponente, die Sie auch in Ihren eigenen Anwendungen einsetzen können, sondern geht in weiteren Abschnitten auch auf spezielle Details von Wertetypen ein – wie Sie beispielsweise die Speicherabfolge von Membervariablen innerhalb einer Struktur genau steuern können oder wie es sich mit Konstruktoren bei Wertetypen verhält.

Erstellen von Wertetypen mit Structure am praktischen Beispiel

Erinnern wir uns. Grundsätzlich gilt: Eine Klasse produziert einen Referenztyp; eine Struktur produziert einen Wertetyp. Prinzipiell erstellen Sie eine Struktur genau wie eine Klasse, und beide haben auch ähnliche Fähigkeiten, wie es durch das folgende Beispiel deutlich werden soll:

Zum Beispielszenario: Gerade beim Programmieren kommt es immer wieder vor, dass Sie Zahlen von einem Zahlensystem ins andere konvertieren müssen. Einige Konvertierungen werden vom Framework unterstützt – so können Sie beispielsweise hexadezimale Werte mit der statischen Parse-Funktion etwa so

```
Dim EinInteger As Integer = Integer.Parse("FFFF", Globalization.NumberStyles.HexNumber)
```

in eine Dezimalzahl umwandeln; eine andere Möglichkeit besteht durch die Benutzung derToInt-Funktion bzw. der ToString-Funktion der Convert-Klasse. Mit

```
Dim EinInteger as Integer = Convert.ToInt32(AnderesZahlensystemString, ZahlenSystemInteger)
```

können Sie ebenfalls einen String, der eine Zahl eines anderen Systems beinhaltet, in einen Integer zurückwandeln. Allerdings können Sie mit ZahlenSystemInteger nur eine Zahl des Dual-(Basis 2), des Oktal- (Basis 8), natürlich des Dezimal- (Basis 10) und des Hexadezimalsystems (Basis 16) umwandeln. Die gleichen Beschränkungen gelten für die Umwandlungen eines Integer-Wertes in ein anderes Zahlensystem, der dann in Form eines Strings abgebildet wird. Diese Aufgabe können Sie mit dem Gegenstück durchführen, etwa durch

```
Dim AnderesZahlensystemString as String = Convert.ToString(IntegerUmzuwandeln, ZahlenSystemInteger)
```

um die Integervariable IntegerUmzuwandeln in ein anderes Zahlensystem umzuwandeln, das durch ZahlenSystemInteger (nur 2, 8, 10 oder 16 sind auch hier wieder gültige Werte) definiert wird.

Zum Glück – denn einen Wertetyp zu schaffen, der beliebige Konvertierungen (nicht nur vom und ins Hexadezimalsystem) beherrscht, ist ein willkommenes, da brauchbares Beispiel.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 16\\Structure01
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

So könnte Ihnen beispielsweise sogar das 32er-Zahlensystem¹ von Nutzen sein: Sie meinen, Sie hätten das nie benutzt? Vielleicht ja doch: Denn wenn Sie Visual Studio selbst installiert haben, dann mussten Sie zur Installation einen Key eingeben, der natürlich mithilfe eines Algorithmus berechnet wurde. Der Algorithmus basiert wahrscheinlich weniger auf dem Hantieren mit Buchstaben, sondern wird – viel wahrscheinlicher – berechnet. Ein `ULong`-Wert (64 Bit, ohne Vorzeichen, neu ab Visual Basic 2005 übrigens), der dieses Seriennumericalgorithmusergebnis speichert, könnte dann beispielsweise auf Basis des Duotrigesimalsystems (32er-System) in eine Zeichenkette umgewandelt werden. Die Zahl

```
9.223.372.036.854.775.807
```

würde in diesem Zahlensystem wie folgt ausschauen

```
7VVVVVVVVVVVVV
```

und die Zahl

```
9.153.672.076.852.735.401
```

um ein anderes Beispiel zu zeigen, lautete wie folgt:

```
7U23085PJGBD9
```

Und hier das erklärte Listing der Struktur `NumberSystems`:

```
Public Structure NumberSystems

    Private myUnderlyingValue As ULong
    Private myNumberSystem As Integer
    Private Shared myDigits As Char()

    Shared Sub New()
        myDigits = New Char() {"0"c, "1"c, "2"c, "3"c, "4"c, "5"c, "6"c, "7"c, "8"c, "9"c, "A"c, _
                             "B"c, "C"c, "D"c, "E"c, "F"c, "G"c, "H"c, "I"c, "J"c, "K"c, "L"c, _
                             "M"c, "N"c, "O"c, "P"c, "Q"c, "R"c, "S"c, "T"c, "U"c, "V"c, "W"c, _
                             "X"c, "Y"c, "Z"c}
    End Sub

```

Die Tabelle für die Umwandlung wird in einem statischen Array gespeichert, das im statischen Konstruktor der Struktur initialisiert wird. Der Konstruktor ist überladen: Ihm wird der Wert übergeben, den die Struktur speichert und im Bedarfsfall als String mit entsprechender Numerale des gewünschten Zahlensystems

¹ Bester Name dafür wäre das »Duotrigesimal-System«, für den es jedoch keine historische Absicherung gibt; ohne Gewähr, dass dieser Link zum Zeitpunkt der Drucklegung noch funktioniert, können Sie sich unter dem IntelliLink [C1601](#) über die Benennung von Zahlensystemen informieren. Kleines Kuriosum am Rande: Das Hexadezimalsystem ist eigentlich recht inkonsistent benannt, denn es wurde aus einem griechischen und einem lateinischen Wortstamm gebildet (»Hexa« griechisch; »decem« lateinisch). Streng genommen müsste es »sedezimal« oder »sexadesimal« heißen. Da die Kurzform für Hexadezimal kurz »Hex« lautet, können wir über den griechisch-lateinischen Mischmasch aber eher dankbar sein.

zurückliefern kann. Sie können dem Konstruktor entweder nur einen Integer- oder einen ULong-Wert übergeben oder einen ULong-Wert und einen weiteren Parameter, der bestimmt, in welchem Zahlensystem Sie arbeiten möchten (bis maximal zum 33er-System).

```

Sub New(ByVal Value As Integer)
    Me.New(CULng(Value), 16)
End Sub

Sub New(ByVal Value As ULong)
    Me.New(Value, 16)
End Sub

Sub New(ByVal Value As ULong, ByVal NumberSystem As Integer)

    myUnderlyingValue = Value
    If NumberSystem < 2 OrElse NumberSystem > 33 Then
        Dim Up As Exception = New OverflowException_
            ("Kennziffer des Zahlensystems außerhalb des gültigen Bereichs!")
        'Kleiner Scherz für die Englisch Sprechenden:
        Throw Up
    End If
    myNumberSystem = NumberSystem

End Sub

Public Property Value() As ULong

    Get
        Return myUnderlyingValue
    End Get
    Set(ByVal Value As ULong)
        myUnderlyingValue = Value
    End Set
End Property

```

Die Value-Eigenschaft dient lediglich dazu, den für die Konvertierung in das jeweilige Zahlensystem gespeicherten Wert neu zu bestimmen oder abzufragen.

```

Public Property NumberSystem() As Integer
    Get
        Return myNumberSystem
    End Get
    Set(ByVal Value As Integer)
        If Value < 2 OrElse Value > 33 Then
            Dim Up As Exception = New OverflowException_
                ("Kennziffer des Zahlensystems außerhalb des gültigen Bereichs!")
            Throw Up
        End If
        myNumberSystem = Value
    End Set
End Property

```

Die NumberSystem-Eigenschaft verwenden Sie, um das Zahlensystem, in das Sie später die Umwandlungen vornehmen wollen, neu zu bestimmen oder abzufragen.

```
Public Overrides Function ToString() As String

    Dim locResult As String = ""
    Dim locValue As ULong = myUnderlyingValue

    Do
        Dim digit As Integer = CInt(locValue Mod NumberSystem)
        locResult = CStr(myDigits(digit)) & locResult
        locValue \= CULng(NumberSystem)
    Loop Until locValue = 0

    Return locResult

End Function
```

Die `ToString`-Methode ist der erste »Dienstleister« für die eigentliche Aufgabe. Sie verfährt nach folgendem Algorithmus, um die Zeichen (die Numeralia) für das eingestellte Zahlensystem zu ermitteln:

Zunächst kopiert sie den Ursprungswert in eine temporäre Variable, um die `Value`-Eigenschaft nicht zu »zerstören« – diese Variable wird im Folgenden nämlich verändert. Nun führt `ToString` eine Restwertdivision mit der `Mod`-Funktion durch. Diese liefert nicht das Ergebnis der Division, sondern den Restwert. Ein Beispiel im 10er System soll verdeutlichen, was gemeint ist:

Wenn Sie den Wert 129 durch 10 teilen, kommt 12 dabei heraus, und es bleibt ein Rest von 9. Genau diese 9 ist aber in diesem Fall wichtig, denn sie entspricht der ersten gesuchten Ziffer der Ergebniszahlenkette (der äußerst rechts stehenden, um genau zu sein). Anschließend wird der Wert durch die Basiszahl des Zahlensystems geteilt – um bei diesem Beispiel zu bleiben also durch 10 – und als Ergebnis kommt 12 dabei raus. Da das Divisionsergebnis (dieses Mal das Ergebnis, nicht der Restwert) größer ist als 0, wiederholt sich der Vorgang. Wieder wird der Divisionsrestwert ermittelt, und der beträgt dieses Mal 2. Die 2 entspricht der zweiten ermittelten Ziffer. Die Schleife wird nun so lange wiederholt, bis alle Ziffern (bzw. Numeralia) bekannt sind. Die Numeralia selbst werden übrigens aus dem `myDigits`-Array gelesen, dem statischen Array, das durch den statischen Konstruktor der Struktur bei ihrer ersten Verwendung angelegt wird.

HINWEIS Sie finden in diesem Beispiel einige Konvertierungen von Wertetypen in andere mithilfe der Cxxx-Operatoren, zu denen das nächste Kapitel mehr Infos gibt. Nur soviel fürs Erste: Eine Konvertierung vom Typ `Integer` in den `ULong`-Datentyp mithilfe von `CULng` ist in der fett geschriebenen Zeile des oben stehenden Codeausschnittes notwendig, damit die Division über die vollen 64-Bit Breite stattfindet. Ohne diese Konvertierung würde Visual Basic eine (standardmäßige) 32-Bit-Integer-Division initialisieren, die mit der 64-Bit-breiten `ULong`-Variable `locValue` nicht funktionieren kann.

```
Public Shared Function Parse(ByVal Value As String, ByVal NumberSystem As Integer) As NumberSystems

    'Hier wird der Value zusammengebaut
    Dim locValue As ULong

    For count As Integer = 0 To Value.Length - 1
        Try
```

```

'Aktuelles Zeichen im String, das verarbeitet wird
Dim locTmpChar As String = Value.Substring(count, 1)

'Binäresuche anwenden, um das Zeichen im Array zu finden und damit die Ziffernnummer
Dim locDigitValue As Integer = CInt(Array.BinarySearch(myDigits, CChar(locTmpChar)))

'Prüfen, ob sich das Zeichen im Gültigkeitsbereich befindet
If locDigitValue >= NumberSystem OrElse locDigitValue < 0 Then
    Dim Up As Exception = New FormatException _
        ("Ziffer '" & locTmpChar & "' ist nicht Bestandteil des Zahlensystems!")
    Throw Up
End If

'Aus der gefundenen Ziffernnummer die Potenz bilden, und zum Gesamtwert addieren
locValue += CULng(Math.Pow(NumberSystem, Value.Length - count - 1) * locDigitValue)

Catch ex As Exception
    'Für den Fall, dass zwischendurch 'was schiefgeht
    Dim Up As Exception = New InvalidCastException
        ("Ziffer des Zahlensystems außerhalb des gültigen Bereichs!")
    Throw Up
End Try
'nächstes Zeichen verarbeiten
Next

Return New NumberSystems(locValue, NumberSystem)
End Function
End Structure

```

Die Parse-Funktion ist eine statische Funktion (genau wie die Parse-Funktionen vieler Datentypen in der CLR), und sie dient dazu, eine Zeichenkette, die dem Wert eines bestimmten Zahlensystems entspricht, in einen NumberSystem-Wert umzuwandeln. Prinzipiell arbeitet Sie nach der Formel

$$\text{Numeraler Wert} = \text{Zifferwert} \times \text{Zahlensystembasiswert}^{\text{Zifferposition}-1}$$

Die Werte der Ziffern sind dabei durchnummeriert von 0–33 (Ziffern von 0–9, gefolgt vom großbuchstabigen Alphabet von A–Z). Mit einer Schleife iteriert die Funktion dabei von vorne nach hinten durch die Zeichen des umzuwendenden Strings. Mit der statischen BinarySearch-Funktion der Array-Klasse ermittelt sie dabei den Ziffernwert. Dieser stellt anschließend den Wert eines Multiplikators des Produkts dar. Der andere Multiplikator ergibt sich durch das Potenzieren des Basiswertes des Zahlensystems mit der Zifferposition. Auch hier soll ein Beispiel helfen, den Algorithmus besser zu verstehen, dieses Mal jedoch mit einer Konvertierung aus dem Hexadezimalsystem:

Gegeben sei die Zahl »F3E«. Um diese umzurechnen, ermittelt die Funktion den Ziffernwert für »F«, der dem Wert 15 entspricht. Da es das dritte Zeichen im String ist (von hinten gesehen), wird 16 mit 2 (es ist Ziffernposition 1) potenziert (ergibt 256) und mit 15 multipliziert – das Ergebnis lautet: 3840. Nun ist das mittlere Zeichen an der Reihe. Der Ziffernwert beträgt 3, der Exponent 1, denn es ist das 2. Zeichen der Zeichenkette. Zwischenergebnis: 48, addiert zum vorherigen Wert ergibt 3888. Was fehlt, ist die letzte Ziffer.

Der Exponent² ist 0, damit entspricht das Produkt dem Ziffernwert (Multiplikation mit eins verändert nichts), und dieser entspricht 14 für das »E«. Die letzte »14« wird zum Zwischenergebnis addiert, und wir haben das Ergebnis 3902.

Zum Projekt: Hauptprogramm und Struktur sind in dem Programmbeispiel in zwei verschiedenen Dateien untergebracht. Ein Doppelklick auf *NumberSystems.vb* im Projektreviewer öffnet den Quellcode der Struktur; *mdlMain.vb* enthält das Modul für das Hauptprogramm.

```

Module mdlMain

Sub Main()
    Dim locLong As ULong
    locLong = &HFFFFFFFFFFFFFFFUL
    Dim locNS As New NumberSystems(locLong)
    Console.WriteLine("{0:#,#0} entspricht:", locLong)

    locNS.NumberSystem = 2 : Console.WriteLine("Binär: " & locNS.ToString)
    locNS.NumberSystem = 8 : Console.WriteLine("Oktal: " & locNS.ToString)
    locNS.NumberSystem = 10 : Console.WriteLine("Dezimal: " & locNS.ToString)
    locNS.NumberSystem = 16 : Console.WriteLine("Hexadezimal: " & locNS.ToString)
    locNS.NumberSystem = 32 : Console.WriteLine("Duotrigesimal: " & locNS.ToString)

    'Der umgekehrte Weg:
    Console.WriteLine()
    Console.WriteLine("Gegenbeispiel:")
    Console.WriteLine("7U23085PJGBD9' duotrigesimal entspricht dezimal: ")
    locNS = NumberSystems.Parse("7U23085PJGBD9", 32)
    Console.WriteLine(locNS.Value)

    Console.WriteLine()
    Console.WriteLine("Return drücken zum Beenden")
    Console.ReadLine()

End Sub
End Module

```

Wenn Sie das Programm starten, erhalten Sie folgende Ausgabe:

² Sie erinnern sich an die Schulzeit? – Jeder Wert potenziert mit 0 ergibt 1. Die Zahl wird dabei durch sich selbst geteilt.

Unterschiedliche Verhaltensweisen von Werte- und Referenztypen

Jetzt haben Sie Ihre erste Struktur entwickelt, aber inwiefern unterscheidet die sich nun von einer Klasse? Nun, wie in der Einführung des letzten Kapitels schon erklärt, speichern Wertetypen – wie beispielsweise die gerade entwickelte NumberSystems-Klasse – im Gegensatz zu Referenztypen ihren Inhalt direkt auf dem Prozessorstack und nicht auf dem Managed Heap. Bei der Zuweisung von einem Wertetyp an einen anderen werden die Inhalte (die Daten) also wirklich kopiert und beide Wertetypen sind anschließend unabhängig voneinander, wie das folgende Beispiel zeigt:

```
'Neuen Wertetyp deklarieren und definieren.  
Dim Wertetyp1 As New NumberSystems(10)  
  
'Zweiter Wertetyp wird deklariert durch den ersten definiert.  
Dim Wertetyp2 As NumberSystems = Wertetyp1  
  
'Zweiter Wertetyp bekommt anderen Wert.  
Wertetyp2.Value = 20  
  
'Erster Wertetyp behält alten Wert.  
Console.WriteLine(Wertetyp1.Value)
```

Die Ausgabe dieses Beispiels lautet »10«.

Wertetyp1 wird Wertetyp2 zugewiesen. Da es sich bei beiden Variablen um Wertetypen handelt (instanziert aus der Struktur des vorherigen Beispielprogramms), kopiert die CLR den Inhalt von Wertetyp1 in Wertetyp2. Eine anschließende Änderung von Wertetyp2 hat keine Auswirkungen auf Wertetyp1, da beide ihren unabhängigen Speicherbereich auf dem Stack beanspruchen.

Anders sieht das mit Referenztypen bei Klasseninstanzen aus. Im Beispielprogramm befindet sich ebenfalls eine Klasse, die nur Demonstrationszwecken dient und ebenfalls einen `ULong`-Wert speichern kann. Wenn für das gleiche Beispiel ein Referenztyp verwendet wird, bekommen Sie ein völlig anderes Ergebnis, und zwar eines, das Sie vielleicht nicht unbedingt erwarten:

```
'Neuen Referenztyp deklarieren und definieren  
Dim Referenztyp1 As New ReferenzTyp(10)  
'Zweiter Referenztyp wird deklariert durch den ersten definiert  
Dim Referenztyp2 As ReferenzTyp = Referenztyp1  
  
'Zweiter Referenztyp bekommt anderen Wert  
Referenztyp2.Value = 20  
  
'Erster damit ebenfalls!!!  
Console.WriteLine(Referenztyp1.Value)
```

Die Ausgabe dieses Beispiels lautet »20«.

Dieses Beispiel verwendet ausschließlich Referenztypen. Sie sehen, dass in diesem Beispiel nur eine einzige Instanz der Klasse erstellt und deswegen auch nur einmal der Speicherbereich für die Instanz auf dem Managed Heap reserviert wird. Die zweite Objektvariable wird zwar deklariert, aber die Zuweisung

```
Dim Referenztyp2 As ReferenzTyp = Referenztyp1
```

erstellt keine neue Instanz, sondern weist der Variablen lediglich einen Zeiger auf die schon vorhandene Instanz zu. Aus diesem Grund ändern Sie oberflächlich betrachtet mit dem ersten Objekt auch das zweite Objekt. Die Wahrheit ist aber, es gibt gar kein zweites Objekt. Mit der zweiten Variablen ändern Sie lediglich das einzig vorhandene Objekt, das jetzt durch die Objektvariablen Referenztyp1 *und* Referenztyp2 repräsentiert wird.

Das gleiche Verhalten lässt sich beim Übergeben von Wertetypen bzw. Referenztypen an Prozeduren beobachten. Wir fügen dem Modul zwei Subs hinzu, die folgende Form haben:

```
Sub NimmtWertetyp(ByVal Wertetyp As NumberSystems)
    Wertetyp.Value = 99
End Sub

Sub NimmtReferenzTyp(ByVal Referenztyp As ReferenzTyp)
    Referenztyp.Value = 99
End Sub
```

Beide Subs machen im Prinzip das gleiche – die eine arbeitet jedoch mit Verweis-, die andere mit Wertetypen. Betrachten Sie jetzt den folgenden Codeauszug, der mit Wertetypen arbeitet:

```
Dim Wertetyp1 as New NumberSystems(10)
NimmtWertetyp(Wertetyp1)
Console.WriteLine(Wertetyp1.Value)

Referenztyp1 as New ReferenzTyp(10)
NimmtReferenzTyp(Referenztyp1)
Console.WriteLine(Referenztyp1.Value)
```

Die erste Ausgabe lautet hier »10«, die zweite »99«. Wenn Sie einen Wertetyp einer Prozedur übergeben, legt die CLR eine Kopie der eigentlichen Daten der Struktur auf den Stack und übergibt sie der aufgerufenen Prozedur. Die Prozedur arbeitet mit der Datenkopie auf dem Stack, und das Verändern der Variablen hat keine Auswirkungen auf die Variable, mit der die Übergabe initiiert wurde.

Anders wiederum ist das bei Referenztypen. Hier übergibt das aufrufende Programm nur einen Verweis auf den Datenbereich im Managed Heap. Es gibt keine Kopie der Klasseninstanz; die Änderung der Daten erfolgt von beiden Objektvariablen und kann auch durch beide reflektiert werden.

Verhalten der Parameterübergabe mit **ByVal** und **ByRef** steuern

Es kann wünschenswert sein, einen Wertetypen durch das Schlüsselwort **ByRef** als Referenz einer Prozedur zu übergeben. In diesem Fall werden beim Aufruf nicht wie sonst die Daten selbst auf den Stack kopiert, sondern ein Zeiger auf die entsprechende Speicherstelle im Stack. Änderungen, die an den Daten innerhalb

der Prozedur erfolgen, spiegeln sich damit direkt in der ursprünglichen Variable wider. Eine Änderung der Variablen innerhalb der *aufgerufenen* Prozedur wirkt sich also auch in einer Änderung des Variableninhaltes des *aufrufenden* Programmteils aus. Die Abänderung des Codes macht diesen Sachverhalt deutlich:

```
Dim Wertetyp1 as New NumberSystems(10)
NimmtWertetyp(Wertetyp1)
Console.WriteLine(Wertetyp1.Value)
Dim Wertetyp2 as NumberSystems = Wertetyp1
Wertetyp2.Value = 50
Console.WriteLine(Wertetyp1.Value)
.
.
.
Sub NimmtWertetyp(ByRef Wertetyp As NumberSystems)
    Wertetyp.Value = 99
End Sub
```

Eine solche Übergabe *ByRef* kann zum Beispiel sinnvoll sein, wenn Sie mehrere Informationen an das übergeordnete Programm zurückgeben wollen. Eine Funktion hat nun einmal nur einen Rückgabewert. Wie aber sollten Sie in einer Funktion mit dem Hauptprogramm kommunizieren, wenn Sie beispielsweise eine Zeichenkette an eine Funktion übergeben wollen, die diese Zeichenkette in eine Zahl konvertiert und einerseits die Information benötigen, ob dies überhaupt möglich war (Rückgabe der Funktion dann entweder *True* oder *False*), Sie aber auf der anderen Seite in einer Zahlenvariablen den konvertierten Wert wissen müssen? So löst das .NET Framework das Problem. Der Typ *Int32* stellt beispielsweise die statische Methode *TryParse* bereit, die als ersten Parameter die zu konvertierende Zeichenkette erwartet, als zweiten Parameter eine *Integer*-Variable *ByRef*, die innerhalb der Funktion auf den konvertierten Wert gesetzt wird und die *True* oder *False* (ob dies überhaupt möglich war) als Funktionsergebnis zurückliefert.

Konstruktoren und Standardinstanziierungen von Wertetypen

Da es keine Strukturinstanzen ohne Daten gibt, sorgt die CLR übrigens automatisch dafür, dass bei der Definition einer Struktur immer auch eine entsprechende Dateninstanz erstellt wird – ganz gleich, ob Sie *New* zur Instanziierung verwendet haben oder nicht. Auch hier zeigt ein Beispiel, was gemeint ist:

```
Dim EinWert As NumberSystems
Dim EineReferenz As ReferenzTyp

EinWert.Value = 10
EineReferenz.Value = 10
```

Diese Zeilen werden bis auf die letzte anstandslos verarbeitet. Bei der letzten tritt jedoch eine Ausnahme auf, weil Sie versuchen, die Eigenschaft eines Objektes zu verändern, das auf dem Managed Heap gar nicht existiert (siehe Abbildung 16.1).

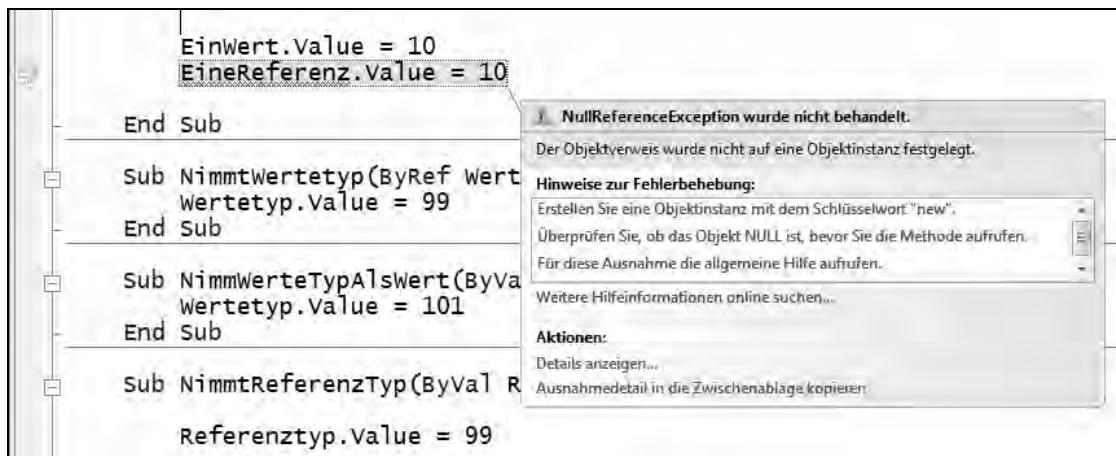


Abbildung 16.1 Versuchen Sie die Eigenschaft eines Objektes zu ändern, das nicht instanziert wurde, erhalten Sie eine Ausnahme

HINWEIS ByVal und ByRef haben für die Übergabe von Parametern, bei denen es sich um Referenztypen handelt, übrigens auch Auswirkungen, allerdings nicht beim Verändern des Objektinhalts, sondern beim Neuzuweisen von Objektvariablen an andere Objektinstanzen. Wenn Sie eine Objektvariable mit ByVal übergeben, und innerhalb dieser Prozedur der Objektvariablen eine neue Instanz zuweisen, dann ändert sich auch die Instanz, auf die die Objektvariable in der aufrufenden Prozedur zeigt. Mit ByRef ist das nicht der Fall.

Keine Standardkonstruktoren bei Wertetypen

Strukturen in VB.NET dürfen also nicht über Standardkonstruktoren verfügen. Ein Standardkonstruktor ist im Falle von VB.NET eine Sub New, die keine Parameter entgegennimmt.

Das folgende Konstrukt ist also beispielsweise fehlerhaft:

```
Public Structure TestValueType
    Private myTestEigenschaft As String

    'Fehler: Ein parameterloser Konstruktor, der nicht als "Shared" deklariert ist,
    'kann nicht in einer Struktur deklariert werden.
    Sub New()
        myTestEigenschaft = "Vorinitialisiert"
    End Sub

    'Das ist OK:
    Sub New(ByVal Vorgabe As String)
        myTestEigenschaft = Vorgabe
    End Sub

    Property TestEigenschaft() As String
        Get
            Return myTestEigenschaft
        End Get
        Set(ByVal Value As String)
    End Set
    End Property
End Structure
```

```

    myTestEigenschaft = Value
End Set
End Property

End Structure

```

Strukturen bilden Wertetypen; und Wertetypen arbeiten, was Konstruktoren und Initialisierung der Instanz anbelangt, sehr abweichend von dem, was wir bislang über Referenztypen, die aus Klassen entstehen, kennen gelernt haben. Die CLR erlaubt es immer, eine Instanz eines Wertetyps zu erstellen; es gibt keine Möglichkeit zu verhindern, dass das geschieht (im Falle von Klassen reichte ja abweichend davon schon ein privater Konstruktor). Und, wir erinnern uns: Wertetypen sind deswegen im Framework implementiert, weil wir als Entwickler Datenstrukturen benötigen, die extrem schnell arbeiten. Wir fassen also zusammen:

- Wertetypen können immer instanziert werden.
- Wertetypen sollten sehr schnell instanziert werden.

Und aus diesen beiden Punkten folgt: Strukturen dürfen in VB.NET (auch nicht in C#) keinen Standardkonstruktor haben, weil das den Entwickler denken lassen könnte, er hätte Einfluss darauf, wann dieser Standardkonstruktor aufgerufen würde. Weil das Framework aus Performancegründen genau das aber nicht garantieren kann, nämlich dass die Konstruktoren auch zum Zeitpunkt der Instanziierung angewendet werden, darf es eben keine selbst erstellbaren Standardkonstruktoren geben. Die Instanziierung von Wertetypen, wie sie durch Strukturen definiert werden, geschieht anders als bei »normalen« Klassen, die mit Class definiert wurden. Klassen können ausschließlich durch das Schlüsselwort New instanziert und anschließend verwendet werden. Strukturen müssen nicht mit New vor ihrer ersten Verwendung instanziert werden – vielmehr kann ihre Instanziierung, wie wir schon kennen gelernt haben, implizit erfolgen, also ist beispielsweise das folgende Konstrukt durchaus erlaubt:

```

'Keine Instanzierung mit New...
Dim tvt As TestValueType
'...aber dennoch vorhanden:
tvt.TestEigenschaft = "Test"

```

Sobald ein Wertetyp, wie oben gezeigt, implizit definiert wird, sorgt das Framework dafür, dass alle seine Member-Variablen mit ihren Standardwerten vorbelegt werden (also beispielsweise 0 bei numerischen Typen, False für Boolean, etc.). Das geschieht aber nicht notwendigerweise dann, wenn der Datentypen instanziert wird, sondern vielleicht erst bei seiner ersten Verwendung.

Das bedeutet aber wiederum, dass der Standardkonstruktor bei einer impliziten Verwendung u.U. nicht oder nicht zeitnah verwendet wird, und um Fehler von vornherein auszuschließen (nämlich, dass der Entwickler von den Standardwerten abweichende Initialisierungen im Standardkonstruktor vornimmt, die den Member-Variablen aber später gar nicht zugewiesen wurden), verbietet Visual Basic das Vorhandensein von Standardkonstruktoren bei Wertetypen von vornherein.

Gezieltes Zuweisen von Speicherbereichen für Struktur-Member mit den StructLayout- und FieldOffset-Attributen

Strukturen erlauben das gezielte Platzieren von Speicherbereichen für Member-Variablen. Wichtig dafür ist, dass Sie die Struktur mit einem speziellen Attribut markieren und dem Compiler damit »Bescheid geben«, wie er die Speicherbereiche der verschiedenen Member-Variablen überlappen soll.

Mit dieser Möglichkeit wird es zum Kinderspiel, mit den verschiedenen Wertigkeiten der Bytes von Integerwerten zu hantieren. Integer-Zahlen werden, wie Sie sicherlich wissen, aus Bytes »zusammengebaut«. Eine Zahl vom Datentyp Short besteht aus 2 Bytes, ein Integer aus 4 Bytes und ein Long aus 8 Bytes. Bei den vorzeichenbehafteten Integer-Werten wird das höchste Bit für das Vorzeichen verwendet (ist es gesetzt, ist der Wert negativ, ansonsten positiv). Relativ aufwändig ist es, Zahlentypen aus anderen größeren Zahlentypen zu »extrahieren«. Ein Long setzt sich beispielsweise aus zwei 32-Bit-Integer-Werten zusammen – dem so genannten höherwertigen DWord (ein Word entspricht einem Byte, ein DWord – Double Word – zwei Words und damit vier Bytes) und dem niederwertigen DWord. Wenn Sie nun wissen möchten, wie der Zahlenwert des höherwertigen DWords lautet, können Sie ihn entweder berechnen oder nur geschickt aus dem Speicher lesen, was natürlich sehr viel schneller geht.

Bleibt die Frage: Wozu brauchen Sie das? 32-Bit-Grafiken beispielsweise speichern pro Pixel drei Farben und einen Alpha-Wert in insgesamt vier Bytes oder als vorzeichenloser 32-Bit-Integer-Wert (UInteger in Visual Basic). Auf die im Folgenden gezeigte Weise könnten Sie beispielsweise eine Struktur schaffen, die auf einem Integer-Wert basiert – und entsprechende Eigenschaften innerhalb der Struktur könnten ohne eine einzige Zeile Rechenaufwand die einzelnen Farbwerte eines Pixels als Byte zurückliefern.

Einige Betriebssystemfunktionen von Windows, die Sie auch durchaus aus Ihren .NET-Programmen heraus aufrufen können (siehe nächster grauer Kasten), verpacken verschiedene Parameter ebenfalls als Kombination in einem einzigen »großbittigen« Datentyp. Der eine Parameter liegt in den höherwertigen Bytes, der andere in den niederwertigen.

Ein Beispiel. Der Long-Wert (vorzeichenlos) \$FFFFAABB00FF besteht aus zwei DWords: \$0000FFFF sowie \$AABB00FF. Um herauszufinden, wie der Wert des oberen DWords lautet, müssten Sie es bei einer Berechnung zunächst mit \$FFFFFFF00000000 ausmaskieren (eine logische AND-Operation durchführen) und dann den kompletten verbleibenden Wert um 32 Bits nach rechts verschieben. Oder aber Sie lesen die oberen 32 Bits direkt aus dem Speicher, weil Sie diesen Teil der Long-Variable auch noch gezielt einer Integer-Variable zuordnen könnten.

Dazu sind zwei Voraussetzungen notwendig: Sie müssen dafür sorgen, dass die Membervariablen einer Struktur ihren benötigten Speicher exakt in der Reihenfolge definieren, in der sie im Quellcode angegeben werden. Dazu dient das StructLayout-Attribut. Und: Sie bestimmen mit dem FieldOffset-Attribut, an welcher Stelle der Speicher einer Member-Variable auf dem Prozessorstack beginnen soll (immer von 0 an gerechnet). Auf diese Weise können Sie dafür sorgen, dass sich Member-Variablen auch verschiedener Typen überlappen, und dann gezielt auf den Speicher der Member-Variablen eines bestimmten Typs, sozusagen unter »vorgegaukeltem Typdeckmantel«, also als anderer Typ, zugreifen.

Die prinzipielle Vorgehensweise beim Erstellen einer solchen Struktur zeigt das folgende Beispielprogramm. Mit ihr als Vorlage können Sie sie problemlos für eigene Bedürfnisse erweitern.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 16\\StructLayout

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Die Struktur, die dort definiert wird, nennt sich LongEx, und sie erlaubt es, einen 64-Bit-Wert zu definieren, und diesen wahlweise als vorzeichenbehafteten, vorzeichenlosen oder als Teilwert in Form von höher- und niedrigerwertigen 32-Bit-Integer-Werten (wahlweise vorzeichenbehaftet oder vorzeichenlos) abzurufen:

```
'Mitteilen, dass die Reihenfolge der
'Bytedefinitionen strikt einzuhalten ist!
<StructLayout(LayoutKind.Explicit)> _
Public Structure LongEx
    <FieldOffset(0)> Private myUnsignedLong As ULong
    <FieldOffset(0)> Private mySignedLong As Long
    <FieldOffset(0)> Private myLowUnsignedInt As UInteger
    <FieldOffset(4)> Private myHighUnsignedInt As UInteger
    <FieldOffset(0)> Private myLowSignedInt As Integer
    <FieldOffset(4)> Private myHighSignedInt As Integer
```

Mit dem StructLayout-Attribut am Anfang der Struktur bestimmen Sie, dass die Reihenfolge der Member-Definition (oder besser die Reihenfolge der Festlegung derer Speicherplätze) strikt eingehalten werden soll. Mit dem FieldOffset-Attribut erreichen sie anschließend das »Überlappen« der Speicherbereiche für die entsprechenden Member-Variablen. FieldOffset bestimmt durch den angebbaren Parameter, an welcher Speicherstelle im Stack relativ zur Startposition der Struktur selbst die Daten eines Datentyps abgelegt werden.

WICHTIG Strukturen können auch Referenztypen aufnehmen. Da in diesem Fall allerdings Zeiger auf die eigentlichen Daten gespeichert werden, wäre die Gefahr groß, dass Sie durch FieldOffset einen Zeiger auf einen Referenztyp »verbiegen« – und das verbietet das Framework rigoros, da ein solches Vorgehen nicht nur typunsicheren Code erzeugen, sondern das gesamte verwaltete Speicherkonzept über den Haufen werfen würde. Wenn Sie also mit FieldOffset arbeiten, dürfen Sie als Member-Variablen auch nur ausschließlich Wertetypen einsetzen – was im Übrigen auch den Einsatz von Arrays ausschließt, da eine Array-Variable auch nur den Zeiger (die Referenz) auf die eigentlichen Array-Daten darstellt.

Die restlichen Eigenschaftenprozeduren dienen dann nur noch dem Auslesen der Member-Variablen auf die gewohnte Weise:

```
Sub New(ByVal Value As ULong)
    myUnsignedLong = Value
End Sub

Sub New(ByVal Value As Long)
    mySignedLong = Value
End Sub

Sub New(ByVal value As UInteger)
    myLowUnsignedInt = value
End Sub
```

```
Sub New(ByVal value As Integer)
    myLowSignedInt = value
End Sub

Public Property Value() As ULong
    Get
        Return myUnsignedLong
    End Get
    Set(ByVal value As ULong)
        myUnsignedLong = value
    End Set
End Property

Public ReadOnly Property SignedLong() As Long
    Get
        Return mySignedLong
    End Get
End Property

Public ReadOnly Property HighUnsignedInt() As UInteger
    Get
        Return myHighUnsignedInt
    End Get
End Property

Public ReadOnly Property LowUnsignedInt() As UInteger
    Get
        Return myLowUnsignedInt
    End Get
End Property

Public ReadOnly Property HighSignedInt() As Integer
    Get
        Return myHighSignedInt
    End Get
End Property

Public ReadOnly Property LowSignedInt() As Integer
    Get
        Return myLowSignedInt
    End Get
End Property
End Structure
```

Performance-Unterschiede zwischen Werte- und Referenztypen

Ein weiterer wichtiger Unterschied ist die Performance zwischen Werte- und Referenztypen – das war ja überhaupt die Grundüberlegung, Wertetypen abweichend von Referenztypen im .NET Framework zu haben. An die Daten der Wertetypen kommen Sie in der Regel schneller heran, weil ein zusätzlicher Dereferenzierungsschritt nicht erforderlich ist. Rekapitulieren wir:

- Wenn Sie mit den Daten eines Wertetyps arbeiten müssen, befinden sich die Daten auf dem Prozessorstack oder sogar schon in einem Prozessorregister, und der Prozessor kann direkt mit ihnen arbeiten.
- Wenn Sie mit den Daten eines Referenztyps arbeiten wollen, holt sich der Prozessor zunächst die Adressdaten des Speicherbereichs für die eigentlichen Daten im Managed Heap vom Stack. Jetzt erst lädt er die Daten aus dem Managed Heap, indem er die Adresse dereferenziert; dieser Vorgang dauert natürlich entsprechend länger: Zum Einen muss der Prozessor einen zusätzlichen Schritt durchführen – nämlich das Dereferenzieren der Adresse. Zum Anderen teilt sich das Objekt seinen Datenbereich mit anderen Daten. Wenn Sie sehr große Mengen innerhalb Ihrer Anwendung speichern, ist die Wahrscheinlichkeit natürlich groß, dass der Zugriff auf die Daten physisch im Arbeitsspeicher erfolgt. Liegen die Daten auf dem Stack, ist die Wahrscheinlichkeit größer, dass sie sich im Second- oder sogar First-Levelcache Ihres Prozessors befinden; der Datenzugriff hier erfolgt wesentlich schneller als auf den Arbeitsspeicher.

HINWEIS Sie sollten diese Geschwindigkeitsvorteile allerdings nicht als zu groß bewerten, denn: Wenn Sie beispielsweise eine Struktur erstellen, und die Daten, die diese Struktur speichert, in einem Array verwalten, dann verhält sich die Struktur genau wie ein Referenztyp. Array selbst ist nämlich ein Referenztyp, und wenn Referenztypen Daten speichern, die aus Strukturen hervorgehen, verwalten sich diese Strukturinstanzen nahezu wie Referenztypen (der Stack ist nämlich dann nicht mehr erreichbar). Das nächste Kapitel weiß Genaueres zu diesem Thema.

Wieso kann durch Vererben aus Object (Referenztyp) ValueType (Wertetyp) werden?

Berechtigte Frage. Sie haben seit Beginn des Buchs diese Tatsache schon fast als eine Art Dogma kennen gelernt, nämlich dass jedes im Framework verwendete Objekt, sei es ein primitiver Datentyp, eine Formular-Komponente, ein Datenbankobjekt, ein Thread etc. von `Object` abgeleitet ist. Das gilt auch für `ValueType` – diese »Klasse« ist zunächst einmal direkt von `Object` abgeleitet, und man würde erwarten, dass sich damit ein `ValueType` genau wie ein `Object` verhält. Dass sich Wertetypen anders verhalten, liegt an der Implementierung der Klasse `ValueType` in der CLR von .NET. Diese wird zwar von `Object` abgeleitet – man könnte aber fast schon sagen, »nur pro forma«. Die CLR sorgt nämlich dafür, das ursprüngliche Objektverhalten völlig umzukrempeln. Allerdings geschieht dies nur zum Teil durch Methoden, auf die Sie oder ich ebenfalls zurückgreifen könnten, denn vieles davon geschieht »tief im Inneren« der CLR. Das heißt im Klartext: Wenn Sie eine Struktur in Visual Basic erstellen, dann heißt das für Ihr Objekt, dass es implizit von `ValueType` abgeleitet ist und all seine Fähigkeiten erbt. Neben einer neuen Vorgehensweise, das Objekt anhand seines Inhaltes möglichst eindeutig zu erkennen – die so genannte `GetHashCode`-Funktion wird dabei durch einen neuen Algorithmus ersetzt – ändert sich auch die `Equals`-Methode, die zwei Objekte miteinander vergleicht. Hier werden dann nämlich die tatsächlichen Inhalte verglichen. Da `ValueType` eine abstrakte Klasse ist, können Sie sie nicht instanziiieren. Sie dient also quasi nur als »Vorlage« für Wertetypen, die Sie aber nicht in eine andere Klasse ableiten, sondern eben mit `Structure` kreieren. Dass sie innerhalb der CLR einer anderen Speicherverwaltung unterliegt, dafür sorgt dann die CLR intern. Wir »Anwender« haben darauf keinen Einfluss.

HINWEIS Es gibt viele Fälle, bei denen aus einem Wertetyp ein Referenztyp wird – beispielsweise, wenn Sie Wertetypen in Arrays speichern. Was bei diesem Vorgang des so genannten »Boxing« passiert, dazu gibt das folgende Kapitel nähere Auskunft.

Kapitel 17

Type Casting und Boxing von Wertetypen

In diesem Kapitel:

Konvertieren von primitiven Typen	536
Konvertieren von und in Zeichenketten (Strings)	538
Casten von Referenztypen mit DirectCast	541
Boxing von Wertetypen und primitiven Typen	542
Wertänderung von durch Schnittstellen geboxten Wertetypen	545

Typen können in vielen Fällen in andere Typen umgewandelt werden; diesen Vorgang, einen Typen in einen anderen umzuwandeln, nennt man neudeutsch auch »Type Casting«¹:

- Das physische Umwandeln eines konkreten Werts in einen anderen Typ – dabei werden Daten verarbeitet, analysiert und an anderer Stelle neu gespeichert.
- Das Zuweisen des Zeigers auf eine Klasseninstanz an eine andere Objektvariable anderen Typs, die aber ebenfalls Teil der Klassenerbfolge ist. Eine Objektvariable der Basisklasse ErsteKlasse referenziert dabei beispielsweise eine Instanz von ZweiterKlasse; eine Objektvariable der abgeleiteten Klasse ZweiteKlasse soll die Instanz nach der Umwandlung referenzieren.
- Den Vorgang des »Boxen« oder »Boxing« (etwa: *Schachtelns, in eine Schachtel packen*) oder »Unboxing« (»Auspakens«). Dabei wird ein Wertetyp in einen Referenztyp umgewandelt, sodass dieser anschließend auch durch eine Objektvariable einer Klasse der Klassenfolge referenziert werden kann.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 17\\Casting

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Konvertieren von primitiven Typen

In Visual Basic gibt es mehrere Möglichkeiten, einen primitiven Datentyp in einen anderen umzuwandeln. Die einfachste Vorgehensweise ist die einer direkten Zuweisung, die, Option Strict On vorausgesetzt, nicht mit allen Datentypen funktionieren kann. Ein Beispiel:

```
Dim EinInt as Integer=1000
Dim EinLong as Long
'Int kann verlustfrei konvertiert werden; implizite Konvertierung ist möglich!
EinLong=EinInt
```

Bei diesem Vorgang wird eine implizite Konvertierung des Wertes einer Integer-Variablen in eine Long-Variable vorgenommen. Das ist implizit (also ohne weiteres Zutun) möglich, da bei diesem Vorgang niemals ein Verlust auftreten kann. Long speichert nämlich einen weit größeren Zahlenbereich als Integer; alle denkbaren Integer-Werte sind locker von Long speicherbar. Andersherum sieht es hingegen schon schlechter aus:

```
Dim EinInt As Integer
Dim EinLong As Long = Integer.MaxValue + 1L
'Long kann nicht verlustfrei konvertiert werden; implizite Konvertierung ist nicht möglich!
EinInt = EinLong
```

¹ Von engl. »to cast«, »auswerfen«, »abgießen« (aus einer Form). Aber auch »eine Rolle besetzen«.

Wenn Sie diesen Code eingeben, meckert Visual Basic nach der Eingabe der letzten Zeile. Der Grund: Visual Basic kann eine verlustfreie Konvertierung nicht gewährleisten und nimmt Sie in die Verantwortung. Sie müssen jetzt selbst Hand anlegen, und Ihnen stehen dazu jetzt mehrere Optionen zur Verfügung, um dennoch zum gewünschten Ziel zu gelangen:

- Sie verwenden `CInt`, um den `Long`-Wert in einen `Integer`-Typ umzuwandeln. Das sähe dann folgendermaßen aus.

```
'Int kann nicht verlustfrei konvertiert werden, explizite Konvertierung ist nötig!
EinInt = CInt(EinLong)
```

- Sie verwenden `CType` für den gleichen Vorgang. `CType` arbeitet prinzipiell wie `CInt`, ist allerdings nicht auf `Integer` als Zieltyp beschränkt. Deswegen bestimmen Sie als zweiten Parameter, in welchen Typ Sie das angegebene Objekt umwandeln möchten:

```
'Int kann nicht verlustfrei konvertiert werden; explizite Konvertierung ist nötig!
EinInt = CType(EinLong, Integer)
```

- Als letzte Option können Sie die `Convert`-Klasse des Frameworks verwenden – allerdings ist dies auch die langsamste Methode. Seit Visual Basic 2005 sorgt der Visual Basic-Compiler nämlich dafür, dass mit `Cxxx` oder `CType` immer die schnellste Konvertierungsmöglichkeit zur Anwendung kommt. Visual Basic 2002 und 2003 waren an dieser Stelle längst nicht so hoch optimiert – in einigen Fällen war hier der Einsatz der `Convert`-Klasse den eingebauten Möglichkeiten mit `Cxxx` bzw. `CType` sogar vorzuziehen. Der Vollständigkeit halber – der Aufruf mit der `Convert`-Klasse gestaltet sich folgendermaßen:

```
'Int kann nicht verlustfrei konvertiert werden; explizite Konvertierung ist nötig!
EinInt = Convert.ToInt32(EinLong)
```

Rein theoretisch gäbe es noch folgende Möglichkeit, die Konvertierung durchzuführen: Sie entscheiden sich für `Option Strict Off`. In diesem Fall kümmert sich Visual Basic zur Laufzeit um die Konvertierung in den richtigen Datentyp. Da Ihnen der Codeeditor in diesem Fall aber nicht einmal eine Warnung im Falle einer verlustmöglichen Konvertierung meldet, rate ich (schon wieder) dringend davon ab, von dieser Möglichkeit Gebrauch zu machen!

HINWEIS Wenn Sie die Definition des Ausgangswerts wie unten zu sehen ändern, erhalten Sie eine Ausnahme (siehe Abbildung 17.1). Das liegt daran, dass Sie versuchen, einen Wert zu konvertieren, der sich schlüssig und ergreifend nicht konvertieren lässt. Der zu konvertierende Wert wird mit

```
Dim EinLong As Long = Integer.MaxValue + 1L
```

den größtmöglichen mit dem `Integer`-Datentyp speicherbaren Wert plus eins festgelegt. Damit überschreitet er die zulässige »Wertekapazität« von `Integer`, und das Framework präsentiert Ihnen die Ausnahme.

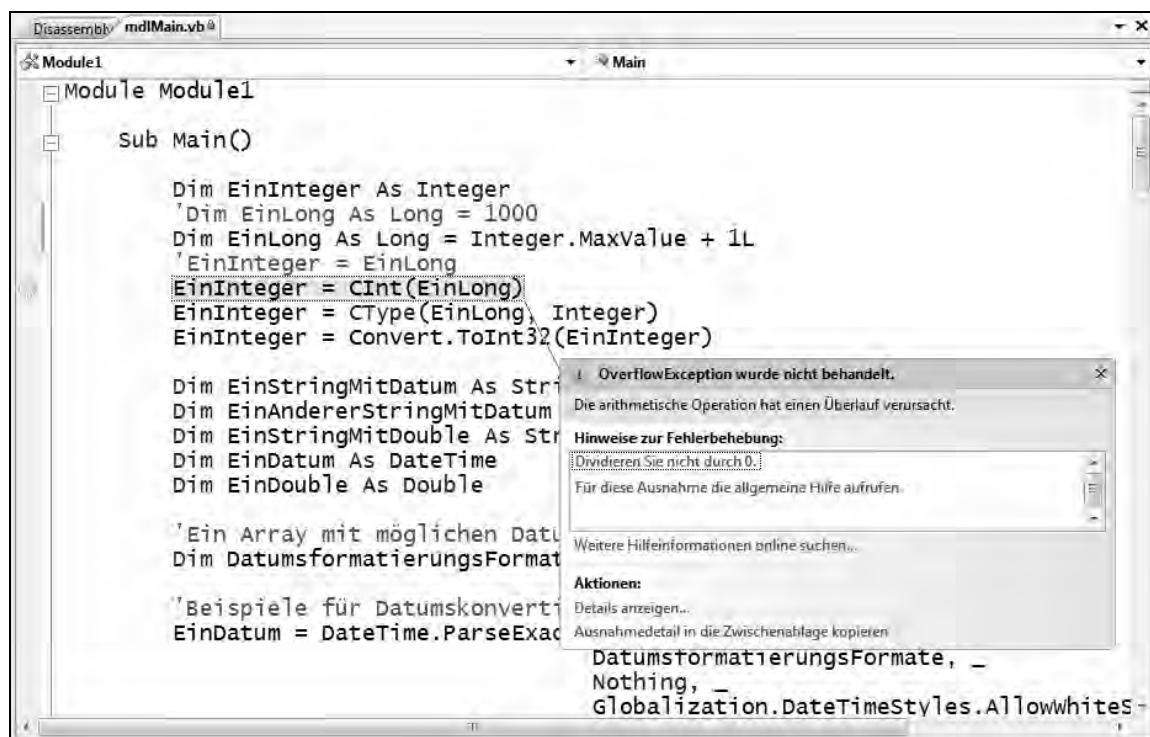


Abbildung 17.1 Beim Type Casting müssen Sie natürlich darauf achten, dass sich ein Typ, was seine Größe oder Eigenschaften anbelangt, auch in einen anderen Typ casten lässt!

Konvertieren von und in Zeichenketten (Strings)

Eine Konvertierung von primitiven Datentypen ist natürlich nicht nur auf numerische Typen beschränkt. Viel interessanter ist die Konvertierung von einer Zeichenkette in einen numerischen Wert oder in einen Datumswert oder umgekehrt.

Grundsätzlich besteht auch hierbei die Möglichkeit, mit den bislang vorgestellten Verfahren eine Konvertierung durchzuführen. So haben Sie die Möglichkeit, beispielsweise einen String, der ein Datum als Zeichenkette speichert, in einen echten Datumswert umzuwandeln. Die Möglichkeiten dafür sind:

```

Dim EinStringMitDatum As String = "24.12.2003"
Dim EinDatum As DateTime

EinDatum = CDate(EinStringMitDatum)
EinDatum = CType(EinStringMitDatum, DateTime)
EinDatum = Convert.ToDateTime(EinStringMitDatum)

```

Konvertieren von Strings mit den Parse- und ParseExact-Methoden

Allerdings gibt es eine weitere Möglichkeit, die Ihnen eine größere Flexibilität zur Verfügung stellt. Die `DateTime`-Struktur² verfügt über die statische Methode `Parse`³, die ebenfalls eine Zeichenkette, die ein Datum enthält, in einen `DateTime`-Wert umwandeln kann; Sie bietet Ihnen aber eine wesentlich größere Flexibilität: Die `Parse`-Funktion enthält mehrere Überladungen. Von der »einfachen« Version angefangen, können Sie einen so genannten »Format-Provider« angeben, mit dem Sie bestimmen können, wie die Datumsvorgabe auszusehen hat, damit die Umwandlung gelingen kann. Zusätzlich können Sie mit einem optionalen dritten Parameter ein gewisses Toleranzverhalten bei der Zeichenkettenanalyse bestimmen.

Noch mehr Kontrolle erhalten Sie, wenn Sie die `ParseExact`-Methode verwenden. Diese erlaubt Ihnen auch zusätzlich noch, ein String-Array mit Eingabemustern als Richtlinie für die String-Analyse zu geben. Möchten Sie beispielsweise ein Eingabefeld in einem Programm schaffen, in dem das Datum nicht starr im Format »dd.MM.yy« eingegeben werden muss, sondern – ergonomisch für den Anwender – auch Eingabeformate wie »ddMMyy« oder »ddMM« möglich sind, verwenden Sie die `ParseExact`-Methode, um den Datumswert umzuwandeln. Das Framework nimmt Ihnen dabei alles an Analysearbeit ab und wandelt den String nach Ihren Vorgaben in einen Datums Wert um oder generiert eine abfangbare Ausnahme, wenn der Anwender eben nicht die Daten im entsprechenden Format eingegeben hat.

Das über Zeichenketten Gesagte gilt in gleichem Maße auch für die numerischen Datentypen. Auch sie verfügen über eine `Parse`-Methode zur Zahlenaumwandlung, die wesentlich flexibler als die bisher vorgestellten Alternativen sind.

HINWEIS

Eine `ParseExact`-Methode steht für numerische Datentypen nicht zur Verfügung.

Ein Beispiel für die Anwendung dieser Methoden finden Sie im nächsten Abschnitt.

Konvertieren in Strings mit der `ToString`-Methode

Das Gegenstück zu `Parse` bildet die `ToString`-Methode. Grundsätzlich können Sie – was primitive Datentypen wie `Date`, `Integer`, `Double`, `Decimal` etc. anbelangt – jeden Variableninhalt mit der `ToString`-Methode in eine Zeichenkette umwandeln. Gerade die Formatierung von Zahlen und Datums Werten wird aber durch das Framework besonders unterstützt – ein Blick in die Hilfe eines jeweiligen Objektes für die aktuelle Funktionsweise ist auf jeden Fall hilfreich und angebracht. Kapitel 26 liefert Ihnen zu diesem Thema ebenfalls noch viele zusätzliche Informationen.

Die folgenden kleinen Programmauszüge zeigen Ihnen im Schnellüberblick, wie der Einsatz mit den Methoden `Parse`-, `ParseExact`- und `ToString` aussehen kann. Sie werden sie aber nicht nur hier, sondern auch an vielen anderen Stellen in diesem Buch noch zu sehen bekommen!

² Diese entspricht dem `Date`-Datentyp von Visual Basic übrigens exakt – es ist also völlig egal ob Sie `Date.Parse` oder `DateTime.Parse` schreiben.

³ Vom Englischen »to parse«, etwa: »analysieren«.

```

Dim EinStringMitDatum As String = "24122003"
Dim EinAndererStringMitDatum As String = "2412"
Dim EinStringMitDouble As String = "1.123,23"
Dim EinDatum As DateTime
Dim EinDouble As Double

'Ein Array mit möglichen Datumsformaten für Date.ParseExact
Dim DatumsformatierungsFormate() As String = New String() {"ddMMyyyy", "ddMM" }

'Beispiele für Datumskonvertierung
EinDatum = DateTime.ParseExact(EinStringMitDatum,
                                DatumsformatierungsFormate, _
                                Nothing, _
                                Globalization.DateTimeStyles.AllowWhiteSpaces)
Console.WriteLine("Datum " & EinDatum.ToString("ddd, dd.MMM.yyyy"))

EinDatum = DateTime.ParseExact(EinAndererStringMitDatum,
                                DatumsformatierungsFormate, _
                                Nothing, _
                                Globalization.DateTimeStyles.AllowWhiteSpaces)
Console.WriteLine("Datum " & EinDatum.ToString("ddd, dd.MMM.yyyy"))

'Zahlenbeispiele
EinDouble = Double.Parse(EinStringMitDouble, Globalization.NumberStyles.Currency)
Console.WriteLine("Wert " & EinDouble.ToString("#,###.## Euro"))

```

Abfangen von fehlschlagenden Typkonvertierungen mit TryParse oder Ausnahmebehandlern

Wenn Ihre Anwendungen später beim Kunden laufen, müssen Sie natürlich mit dem »Fehlverhalten« der Anwender rechnen, oder anders gesagt: Dass diese in Ihren Anwendungen alles Mögliche versuchen einzugeben, nur nicht das, was Sie sich beim Programmieren der Anwendungen vorgestellt haben. Bei Eingaben von Datumswerten oder Zahlen ist es also angebracht, Fehler abzufangen und den Anwender im Bedarfsfall darauf aufmerksam zu machen.

Dazu stehen Ihnen zwei Möglichkeiten zur Verfügung:

- Sie arbeiten mit Parse oder ParseExact (bei Datumswerten) und schließen diese Methode in einen Try/Catch-Block ein. Lösen Parse oder ParseExact eine Ausnahme aus, was sie bei falschem Format machen, dann fangen Sie diese Ausnahme im Catch-Block ab, geben eine entsprechende Fehlermeldung aus und erlauben dem Anwender, einen weiteren Eingabevorschuss durchzuführen.
- Sie arbeiten mit der ebenfalls statischen TryParse-Methode, die es seit Visual Basic 2005 auch für alle numerischen Datentypen gibt. TryParse liefert nicht den eigentlich geparssten Wert als Funktionsergebnis zurück, sondern True bzw. False. Lautete das Ergebnis True, war das Parsen der Zeichenfolge erfolgreich. TryParse übergeben Sie gleichzeitig eine entsprechende Variable, die bei erfolgreichem Parsen das Ergebnis aufnimmt (die Variable wird also explizit ByRef übergeben – um ein wenig an das letzte Kapitel zu erinnern).

Das folgende Beispiel demonstriert diese beiden Vorgehensweisen:

```
'So können fehlerhafte Eingabeformate abgefangen werden:

'Version 1: Mit TryParse (gibt's für Datum- und numerische Typen)
If Double.TryParse("1.234,56", EinDouble) Then
    Console.WriteLine("Zeichenkette konnte gelesen werden. Ergebnis: " & EinDouble.ToString)
Else
    Console.WriteLine("Zeichenkette konnte nicht gelesen werden!")
End If

'Version 2: Mit einer Ausnahmebehandlung
Try
    'Das kann nicht klappen!
    EinDatum = DateTime.ParseExact("2005_12_24", _
        DatumsformatierungsFormate, _
        Nothing, _
        Globalization.DateTimeStyles.AllowWhiteSpaces)

    Catch ex As Exception
        Console.WriteLine("Das Parsen generierte eine Ausnahme:" & vbCrLf & _
            ex.Message)
    End Try
```

Casten von Referenztypen mit DirectCast

Wenn Sie mit Polymorphie arbeiten, dann müssen Sie vergleichsweise häufig Referenztypen konvertieren, die in einer Erbfolge stehen. Ein Beispiel: Sie haben eine Klasse `AbgeleiteteKlasse`, die von `EineKlasse` erbt. Sie definieren eine Objektvariable vom Typ `EineKlasse`, die Sie aber mit der Instanz von `AbgeleiteteKlasse` belegen. Die Gründe, das zu tun, haben mit der Nutzung von Polymorphie zu tun – Beispiele dafür haben Sie schon kennen gelernt.

Nun brauchen Sie im Laufe des Programms aber eine Funktion, die nur von `AbgeleiteteKlasse` zur Verfügung gestellt wird. Da es sich um eine Instanz dieser Klasse handelt, steht diese Funktion, die Sie brauchen, zwar prinzipiell zur Verfügung, doch Sie kommen über die verwendete Objektvariable nicht an die Funktion heran. `DirectCast` bietet Ihnen hier die Möglichkeit, den Verweis auf die Objektinstanz auf eine Objektvariable vom »richtigen« Typ einzurichten; die Funktion lässt sich anschließend aufrufen.

Ein weiteres Beispiel soll diesen Sachverhalt verdeutlichen:

```
'Klassen-Casting
Dim locEineKlasse As EineKlasse
Dim locAbgeleiteteKlasse As AbgeleiteteKlasse = New AbgeleiteteKlasse

'Implizites Casting möglich, denn es geht in der Erbhierarchie Richtung Basisklasse
locEineKlasse = locAbgeleiteteKlasse

'Geht nicht, Funktion nicht vorhanden.
'locEineKlasse.AddValues()

'Geht auch nicht; es geht in der Erbhierarchie nach unten, und dann
'kann nicht implizit konvertiert werden:
'locAbgeleiteteKlasse = locEineKlasse
```

```
'So gehts:  
locAbgeleiteteKlasse = DirectCast(locEineKlasse, AbgeleiteteKlasse)  
locAbgeleiteteKlasse.AddValues()  
  
Console.WriteLine("KAbgeleitet: " & locAbgeleiteteKlasse.ToString())  
Console.WriteLine("KWertepaar: " & locEineKlasse.ToString())
```

Sie könnten übrigens in diesem Beispiel ebenfalls wieder CType einsetzen, indem Sie die Zeile

```
locAbgeleiteteKlasse = DirectCast(locEineKlasse, AbgeleiteteKlasse)
```

durch diese

```
locAbgeleiteteKlasse = CType(locEineKlasse, AbgeleiteteKlasse)
```

ersetzen. Allerdings empfiehlt es sich der besseren Übersichtlichkeit wegen, bei Referenztypen grundsätzlich DirectCast zu verwenden; der Compiler wandelt intern CType in DirectCast um; da können Sie direkt DirectCast verwenden, und Sie sehen so auf den ersten Blick, dass es sich um keine Datenkonvertierung im Sinne von primitiven Datentypen, sondern nur um eine Art Hinweis an den Compiler handelt, den Typen, um den es sich dreht, als den Typen anzusehen, den Sie mit DirectCast angeben.

Boxing von Wertetypen und primitiven Typen

Wenn Sie mit Wertetypen arbeiten, ganz gleich ob mit primitiven Datentypen wie beispielsweise Integer, Long oder Double oder mit selbst gestrickten Strukturen, werden Sie niemals in Verlegenheit kommen, Probleme wie im vorherigen Beispiel lösen zu müssen, denn Wertetypen können Sie nicht vererben.

Allerdings gibt es eine Ausnahme. Dass alle Wertetypen von Object abgeleitet sind, gilt für Wertetypen gleichermaßen. Das hat aber zur Folge, dass Sie zwar keine eigene Erbfolge auf einem Wertetyp basierend erstellen können, aber Object und ValueType an sich bereits in der Erbfolge vorhanden sind, heißt: Eine Object-Objektvariable müsste in der Lage sein, auf einen Wertetyp zu verweisen – doch das klingt schon wie ein Gegensatz in sich.

Das Framework löst dieses Problem durch eine Sonderregel des Common Type Systems, die mit »Boxing«⁴ oder – eingedeutscht – »Boxen« bezeichnet wird.

Dazu ein kleines Beispiel: Nehmen wir an, Sie haben eine Struktur entwickelt und ihr den Namen Matrjoschka⁵ gegeben. Dieser Wertetyp dient als Träger einer bestimmten Datenstruktur, von der Sie mehrere Elemente erstellen wollen und diese in einem Array speichern. Nun soll dieses Array nicht nur Matrjoschka-Werte aufnehmen, sondern soll für zukünftige Erweiterungen vorbereitet sein und deswegen auch andere

⁴ Von engl. »to box« etwa »einpacken«, »verpacken«; »the box«: »der Behälter«, »die Box«. Kann aber auch (hat nichts mit dem Thema zu tun, ist aber dennoch interessant) »Anhieb« bedeuten. Das ist die Einkerbung in einem zu fällenden Baum, um dessen Fallrichtung zu bestimmen und den Stamm vor dem Splittern zu bewahren...

⁵ Sie kennen die kleinen, russischen Figuren, von denen die Äußere in sich geschachtelt mehrere Kleine aufnehmen kann? Gut. Das ist der Name dieser Figur, und das was sie macht, kann gut mit Boxing verglichen werden.

Typen aufnehmen können. Also definieren Sie das Array nicht vom Typ Matrjoschka, sondern vom Typ Object und können die verschiedensten Elementtypen darin speichern. Das folgende Beispiel demonstriert diesen Vorgang und verwendet dazu den Wertetyp Matrjoschka, der – um das Beispiel simpel zu halten – nichts weiter macht, als eine Generationsnummer zu speichern:

```
Structure Matrjoschka
    Private myGeneration As Integer

    Sub New(ByVal Generation As Integer)
        myGeneration = Generation
    End Sub

    Property Generation() As Integer
        Get
            Return myGeneration
        End Get
        Set(ByVal Value As Integer)
            myGeneration = Value
        End Set
    End Property

End Structure
```

Zufallszahlen mit der Random-Klasse

Da wir's gerade brauchen, hier ein kleiner Ausritt in Sachen Zufallszahlen: In Visual Basic 6.0 war es üblich, Zufallszahlen mit der RND-Funktion zu erzeugen. Die Framework Class Library bietet zu diesem Zweck eine spezielle Klasse an, die die alte Funktion ersetzt. Wenn Sie diese Klasse instanziiieren, geben Sie in ihrem Konstruktor einen Ausgangswert an, der die Basis für eine Zufallszahlenfolge darstellt, die im Folgenden generiert werden soll. Damit schon dieser Wert zufällig ist, ergibt es Sinn, hier wirklich »zufällige« Werte, wie beispielsweise die aktuelle Millisekunde (Sie werden mit großer Wahrscheinlichkeit immer eine andere »treffen«) oder eine Mauszeigerposition zu verwenden.

Wenn Sie die Klasse instanziiert haben, können Sie Ihre Instanz verwenden. Sie haben mehrere Methoden, mit denen Sie Zufallszahlen ermitteln können, nämlich die Next-, die NextBytes- und die NextDouble-Methode.

Die Next-Methode liefert den jeweils nächsten zufälligen Integer-Wert zurück. Im Bedarfsfall können Sie bei dieser Methode auch noch die Grenzen angeben, innerhalb derer sich die Zufallszahlen bewegen dürfen.

Mit der NextDouble-Methode erhalten Sie eine Zufallszahl zwischen 0 und 1, also einen Bruch. Diese Methode kommt der ursprünglichen RND-Funktion am nächsten.

Die NextBytes-Methode liefert Ihnen ein definierbar großes Byte-Array mit Zufallszahlen zurück.

Die wirklich einfache Verwendung dieser Klasse demonstriert ebenfalls das folgende Beispiel.

Im Hauptprogramm des Beispiels findet anschließend erst das eigentlich interessante Geschehen statt. Das Programm erstellt 10 Elemente vom Typ Matrjoschka und weist ihnen als Generationsnummer mithilfe der Random-Klasse zufällige Werte im Integer-Wertebereich zu. Anschließend findet es heraus, welches Matrjoschka-Element des Arrays die größte Generationsnummer hatte.

```

Module Module1
    Sub Main()
        'Boxen von Wertetypen
        Dim locObjectArray(9) As Object
        Dim locRandom As New Random(Now.Millisecond)
        Dim loc.MaxValue As Integer

        For locCount As Integer = 0 To 9
            'Implizites Casting ist möglich, es geht in der Erbhierarchie nach oben
            'aber Deckung! - Hier wird geboxt!
            locObjectArray(locCount) = New Matrjoschka(locRandom.Next)
        Next

        'Rausfinden, welches das Objekt mit der höchsten Generationsnummer war
        For locCount As Integer = 0 To 9
            'Zwar sind nur Matrjoschka-Werte im Array drin, doch das Array
            '"kann" nur Objects. Die Generation-Eigenschaft steht nicht zur
            'Verfügung, deswegen funktioniert diese Zeile nicht:
            'loc.MaxValue = locObjectArray(locCount).Generation
            Dim locMatrjoschka As Matrjoschka

            'Entboxen - aus dem referenzierten Wertetyp wird wieder ein "echter" Wertetyp
            locMatrjoschka = DirectCast(locObjectArray(locCount), Matrjoschka)

            'Jetzt kommen wir an die Generation-Eigenschaft heran:
            If loc.MaxValue < locMatrjoschka.Generation Then
                loc.MaxValue = locMatrjoschka.Generation
            End If
        Next

        Console.WriteLine("Die höchste Generationsnummer war: " & loc.MaxValue.ToString())

        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden!")
        Console.ReadLine()
    End Sub
End Module

```

Was passiert hier genau? Mit einer einfachen »Umreferenzierung« von Verweisen in einen Speicherbereich wie bei Klassen ist es bei Wertetypen nicht getan, denn: Es gibt nichts, das auf etwas zeigen könnte. Sie erinnern sich: Die Daten von Wertetypen befinden sich für die direkte Verwendung auf dem Stack. Beim Boxing von Wertetypen werden diese deshalb kopiert und dabei wie bei einem Referenztyp auf den Managed Heap geschrieben. Die Objektvariable kann jetzt auf diesen Speicherbereich zeigen und obwohl es sich um die Daten eines Wertetyps handelt, diese doch referenzieren.

Beim »Entboxen« passiert genau das Gegenteil: Die Wertetypvariable nimmt jetzt die Daten entgegen, die sich auf dem Managed Heap befinden, und auf die die Objektvariable zeigt, die zum Boxing verwendet wurde. Die Daten werden also aus dem Managed Heap wieder zurück auf den Stack kopiert.

Übrigens geschieht der Vorgang des Boxing grundsätzlich, wenn Wertetypen in einem Array gespeichert werden, da ein Array selbst immer auch einen Referenztyp darstellt.

Übrigens: Was DirectCast nicht kann

DirectCast kann Referenztypen, die einen Wertetyp boxen, zurück in den Wertetyp casten (»Entboxen«), es kann allerdings keine Wertetypen casten. Das kann auch nicht funktionieren, denn schließlich können Sie in der Vererbungshierarchie nur in Richtung »weitere Ableitung« casten. Da Wertetypen an sich aber nicht vererbt werden können, bleibt dieser Weg verschlossen.

DirectCast kann natürlich auch keine primitiven Datentypen in andere primitive Datentypen konvertieren; hier verwenden Sie, wie schon gesagt, die Cxxx, CType (je nach Datentyp) Parse, ParseExact, TryParse oder (für alle) die Convert-Klasse.

Wann wird und wann wird nicht geboxt?

Wir haben gesehen, dass Wertetypen dann geboxt werden, wenn sie – beispielsweise wenn Sie eine ganze Menge von ihnen in Arrays oder Auflistungen speichern müssen – in ein Objekt umgewandelt werden müssen. Es gibt aber auch noch einige andere Fälle, wo das Boxing nicht direkt offensichtlich ist. Beispielsweise:

- Geboxt wird, wenn Sie einen Wertetyp als Parameter an eine Methode übergeben, die ein Objekt erwartet. In diesem Fall wird natürlich ihr Wertetyp zuvor in ein Objekt geboxt, und anschließend erst wird die Referenz des geboxeden Wertetyps als Parameter weitergereicht.
- Geboxt wird auch, wenn Sie eine Schnittstelle in einem Wertetyp implementiert haben und jetzt irgendeinen öffentlichen Member über diese Schnittstelle aufrufen. In diesem Fall muss die CLR ebenfalls den Wertetyp boxen, da es sich bei Schnittstellen per Definition um Referenztypen handelt.
- Geboxt wird ebenfalls, wenn Sie GetType für einen Wertetypen aufrufen, denn GetType ist eine von Object geerbte Methode, die nur mit Referenztypen funktioniert.
- Geboxt wird jedoch nicht, wenn Sie eine normale Methode eines Wertetyps aufrufen. Die CLR ist in der Lage, Methoden, die an einem Wertetyp »hängen«, direkt anzuspringen.
- Geboxt wird auch nicht, wenn Sie die ToString-Funktion aufrufen. Man könnte das meinen, denn die ToString-Funktion ist ja prinzipiell überschreibbar, und wenn diese Methode überschrieben würde, dann müsste natürlich eine unter Umständen überschreibende Methode und nicht die Ausgangsroutine aufgerufen werden. Dazu wäre es allerdings notwendig, in der Methodentabelle des überschreibenden Typen nachzuschlagen, wozu wiederum dessen Typ ermittelt werden müsste, was im Umkehrschluss nur dann ginge, läge dieser als Referenztyp vor, und das bedeutete: *Boxing*. Dann wiederum: Wertetypen lassen sich nicht vererben, und nur aus diesem Grund ist ToString eines Wertetyps direkt aufrufbar, ohne dass die CLR den Wertetyp zuvor in einen Referenztyp umwandeln müsste.

Wertänderung von durch Schnittstellen geboxeden Wertetypen

Beim Boxen und dem anschließenden Entboxen von Wertetypen kann es mitunter zu Verhaltensweisen kommen, die auf den ersten Blick nicht wirklich nachzuvollziehen sind. Sie haben eben in der oben stehenden Auflistung gelesen, dass ein Zugreifen auf Member eines Wertetyps automatisch zur Folge hat, dass der

betroffene Wertetyp sofort in ein Objekt geboxt wird. Das folgende Beispiel demonstriert, was dieses Verhalten für Auswirkungen haben kann, und dass sich bestimmte Verhaltensweisen, obwohl es sich um dieselbe Wertetypinstanz handelt, stark unterscheiden.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 17\\Boxing

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Interface IMussValueHaben
    Property Value() As Integer
End Interface

Module mdlMain

    Sub Main()
        Dim EinWertetyp As New Wertetyp(10)
        Dim EinVerweistyp As New Verweistyp(10)

        EinWertetyp.Value = 20
        Console.WriteLine(EinWertetyp.Value) ' 20 -> direkt geändert!
        WertetypÄndern(EinWertetyp)
        Console.WriteLine(EinWertetyp.Value) ' 20 -> in der Stackkopie geändert
        VerweistypÄndern(EinVerweistyp)
        Console.WriteLine(EinVerweistyp.Value) ' 30 -> auf dem Managed Heap geändert

        Dim EinInterface As IMussValueHaben = EinWertetyp
        ÜberInterfaceÄndern(EinInterface)
        Console.WriteLine(EinInterface.Value) ' 40, wird auf dem Managed Heap geändert
        Console.WriteLine(EinWertetyp.Value) ' 20, haben nichts miteinander zu tun

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadLine()

    End Sub

    'Ändert die Eigenschaft des Wertetyps.
    Sub WertetypÄndern(ByVal EinWertetyp As Wertetyp)
        EinWertetyp.Value = 30
    End Sub

    'Ändert die Eigenschaft des Verweistyps.
    Sub VerweistypÄndern(ByVal EinVerweistyp As Verweistyp)
        EinVerweistyp.Value = 30
    End Sub

    'Ändert die Eigenschaft über das Interface.
    Sub ÜberInterfaceÄndern(ByVal EinInterface As IMussValueHaben)
        EinInterface.Value = 40
    End Sub

```

```
End Module

'Testklasse des Wertetyps
Structure Wertetyp
    Implements IMussValueHaben

    Dim myValue As Integer

    Sub New(ByVal Value As Integer)
        myValue = Value
    End Sub

    Property Value() As Integer Implements IMussValueHaben.Value
        Get
            Return myValue
        End Get
        Set(ByVal Value As Integer)
            myValue = Value
        End Set
    End Property
End Structure

'Testklasse des Verweistyps
Class Verweistyp
    Implements IMussValueHaben

    Dim myValue As Integer

    Sub New(ByVal Value As Integer)
        myValue = Value
    End Sub

    Property Value() As Integer Implements IMussValueHaben.Value
        Get
            Return myValue
        End Get
        Set(ByVal Value As Integer)
            myValue = Value
        End Set
    End Property
End Class
```

Dieses Beispiel definiert eine Schnittstelle, einen Referenztyp und einen Wertetyp. Beide Typen binden die ganz am Anfang definierte Schnittstelle ein. Insgesamt drei Prozeduren dienen dazu, die Inhalte der übergebenen Objekte auf sehr einfache Weise zu ändern.

Die erste Wertänderung ist klar: Die Eigenschaft des Wertetyps wird hier direkt geändert, folglich spiegelt sich der geänderte Wert der Eigenschaft auch beim Ausgeben wider.

Die zweite Wertänderung ist hingegen schon nicht mehr so offensichtlich – aber ein Fall, den wir bereits besprochen haben. Hier wird eine Kopie des Wertes auf dem Stack abgelegt; Änderungen auf dem Stack sind nur temporär. Es gibt keine Verbindung zum Objekt, also wird der ursprünglich zugewiesene Wert beibehalten.

Anders ist das bei der Werteänderung des Referenztyps durch die Methode `ReferenztypÄndern`. Es gibt keine Kopie des Objektes, sondern nur verschiedene Zeiger auf die Daten im Managed Heap. Ergo: Eine Änderung in der Prozedur spiegelt die Änderung der Objektvariablen auch im die Prozedur aufrufenden Programmteil wider.

Interessant wird es, wenn die Verwendung einer Schnittstellenvariablen ins Spiel kommt, die einen Wertetyp referenziert. Hier wird der Wertetyp nämlich von vornherein in einen Referenztyp umgewandelt; seine Daten landen auf dem Managed Heap. Die Änderungen erfolgen durch die Unterroutine genau dort, und spiegeln sich deshalb auch durch den Ursprungsverweis auf das Objekt des Managed Heap wieder – durch die Objekt-(Interface-)Variable `EinInterface`. Aber: Nur durch diese Variable greifen Sie auf die »Version« auf dem Managed Heap zu. Die Ursprungsvariable, aus der die Kopie auf dem Managed Heap entstanden ist, steht in keiner Verbindung zur Objektvariablen. Die Ursprungsvariable `EinWertetyp` behält deswegen auch ihren ursprünglichen Wert.

HINWEIS Sie sehen: Boxing ist deswegen eine oft mit Vorsicht zu genießende Geschichte, weil die Boxing-Vorgänge oftmals auf den ersten Blick nicht transparent sind und eine Menge Fehlerpotential bilden können. Gerade beim Zugreifen auf Wertetypen mit Schnittstellenvariablen müssen Sie besonders vorsichtig sein, wie das letzte Beispiel, wie ich finde, eindrucksvoll demonstriert hat.

Oberste Prämisse bei der Verwendung von Wertetypen ist: Nutzen Sie sie immer dann, wenn es wirklich auf Geschwindigkeit ankommt, und nur dann, wenn die Typen, die Sie entwickeln, nicht viel Speicherplatz benötigen. Und darüber hinaus: Nutzen Sie sie am besten direkt; wenn Sie viele Wertetypen in Auflistungen speichern müssen, bearbeiten Sie einen Wertetyp der Liste nicht durch irgendwelche Tricks, sondern casten Sie es in eine Wertetypvariable, bearbeiten Sie die Instanz dort, und boxen Sie sie anschließend wieder zurück in das entsprechende Array-Element.

Kapitel 18

Beerdigen von Objekten – Dispose, Finalize und der Garbage Collector

In diesem Kapitel:

Der Garbage Collector – die Müllabfuhr in .NET	552
Die Geschwindigkeit der Objektbereitstellung	554
Finalize	556
Dispose	560

Das .NET Framework stellt Entwicklern eine außerordentlich solide Plattform für das Erstellen und das Verwenden der unterschiedlichsten Typen zur Verfügung. Wer in seinem Entwicklerleben aber schon mit Speicherlöchern und Pufferüberläufen zu kämpfen hatte, der weiß eine Komponente von .NET besonders zu schätzen: dessen Müllabfuhr nämlich. Müllabfuhr heißt auf Englisch Garbage Collection, und genau das ist der technisch korrekte Ausdruck für den Teil der Common Language Runtime, die Objekte, die nicht mehr benötigt werden, identifiziert und bei Gelegenheit entsorgt.

Um diese Vorgehensweise zu erörtern und deutlich zu machen, möchte ich noch einmal auf ein Beispiel zu sprechen kommen, das ursprünglich der Demonstration eines ganz anderen Themas diente. Sie erinnern sich noch an die Klasse `DynamicList` im Abschnitt zur Polymorphie, die die Beispielanwendung nutzte, um die Artikeldatensätze (`ShopItem`) zu speichern? Neben der eigentlichen Fähigkeit, eine Methode zu implementieren, die eine dynamische Vergrößerung des benötigten Speichers demonstriert, zeigt dieses Beispiel noch etwas anderes – was allerdings mehr eine Fähigkeit des .NET Framework selbst ist: nämlich den nicht benötigten Speicher wieder freizugeben.

Rufen Sie sich die `Add`-Methode dieser Klasse noch mal in Erinnerung:

```
Sub Add(ByVal Item As ShopItem)

    'Prüfen, ob aktuelle Arraygrenze erreicht wurde
    If myCurrentCounter = myCurrentArraySize - 1 Then
        'Neues Array mit mehr Speicher anlegen,
        'und Elemente hinüberkopieren. Dazu:

        'Neues Array wird größer:
        myCurrentArraySize += myStepIncreaser

        'Temporäres Array erstellen.
        Dim locTempArray(myCurrentArraySize - 1) As ShopItem

        'Elemente kopieren
        'Wichtig: Um das Kopieren müssen Sie sich,
        'anders als bei VB6, selber kümmern!
        For locCount As Integer = 0 To myCurrentCounter
            locTempArray(locCount) = myArray(locCount)
        Next

        'Temporäres Array dem Memberarray zuweisen.
        myArray = locTempArray
    End If

    'Element im Array speichern.
    myArray(myCurrentCounter) = Item

    'Zeiger auf nächstes Element erhöhen.
    myCurrentCounter += 1

End Sub
```

Schauen Sie sich diesen Codeblock noch einmal an, aber dieses Mal unter einem anderen Aspekt. Dieses Mal steht nicht der Speicherplatz im Vordergrund, der benötigt wird, und die Art und Weise, wie Arrays wachsen können, sondern der Speicher, der durch denselben Vorgang überflüssig wird.

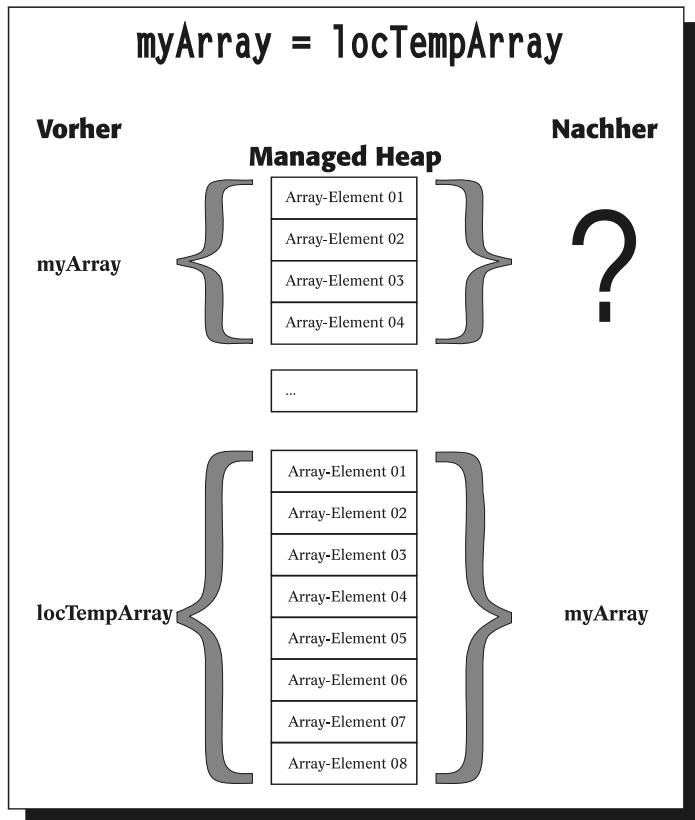


Abbildung 18.1 Was passiert mit den nach der Zuweisung im leeren Raum stehenden Array-Elementen?

Arrays in .NET sind keine Werte, sondern Verweistypen. Benötigter Speicher für alles, was von `System.Array` abgeleitet ist – und dazu zählen auch Arrays, die Sie durch `Dim` deklarieren – wird also auf dem Managed Heap reserviert. Im Codeauszug des Beispielprogramms gibt es zwei entscheidende Zeilen, die eigentlich ein »Speicherleck«, besser bekannt unter dem neudeutschen Begriff »Memory Leak«, verursachen würden, programmierten wir nicht im .NET Framework. Abbildung 18.1 macht das Problem deutlich. Zunächst gibt es das Array und den auf dem Managed Heap dafür reservierten Speicherbereich, aber das Array ist nunmehr zu klein geworden. Also definiert die Prozedur ein neues Array und nimmt dafür die Variable `locTempArray` zu Hilfe. Nun passiert das Entscheidende: `locTempArray` wird `myArray` zugewiesen, der Zeiger auf den Speicherbereich der entsprechenden Elemente wird dabei quasi »verbogen«. `myArray` zeigt anschließend auf die Arrayelemente, auf die kurz zuvor noch `locTempArray` zeigte. Die Adressen auf die Elemente¹, auf die von `myArray` verwiesen wurde, liegen nun unbrauchbar, da nicht mehr referenziert, irgendwo im Speicher – was passiert jetzt mit ihnen?

¹ Wichtig: Es sind natürlich nur die Adressen, die hier brach liegen, nicht die Elemente selbst. Beim Kopieren der Elemente haben wir ja nicht wirklich die Inhalte der Elemente kopiert, sondern nur die Adressen auf die Elemente. Beim Umbiegen der Arrayzeiger bleiben natürlich nur die 5 Zeiger auf die Elemente übrig, die wir aber jetzt weiterverwenden – nämlich als Bestandteil des Arrays mit den nunmehr 10 Elementen.

Erinnern wir uns, wie das bei COM (dem »Vorläufer« von .NET, eine Technologie, die Sie immer noch bewußt oder unbewußt benutzen, wenn Sie – auch in .NET – beispielsweise die Möglichkeit nutzen, Word oder Excel »fernzusteuern«) geregelt war. Bei COM gab es für jedes Objekt einen Referenzzähler. Bei der ersten Zuweisung an eine Objektvariable wurde der Zähler auf Eins gesetzt. Mit jeder weiteren Zuweisung an eine Variable – also mit jeder weiteren Referenzierung – wurde dieser Zähler um eins erhöht. Nun trat der umgekehrte Fall ein: Einer Variablen, die zuvor das Objekt referenzierte, wurde ein anderes Objekt oder Nothing zugewiesen, oder das Programm verließ den Gültigkeitsbereich der Variablen, sodass sie aus diesem Grund das Objekt nicht mehr referenzieren konnte. In diesem Fall wurde der Referenzzähler um eins vermindert. Wurde er 0, dann konnte das Objekt entsorgt werden. Es wurde zu diesem Zeitpunkt nicht länger benötigt, da von keiner Stelle des Programms aus mehr referenziert.

Dieses Verfahren hatte allerdings zwei Nachteile: Zum Einen kostete des Prinzip des Referenzzählers Rechenzeit. Der andere Nachteil war das Problem der so genannten Zirkelverweise: Ein Objekt, das auf ein Objekt zeigte, was seinerseits wieder auf das Ausgangsobjekt zeigte, führte dazu, dass der Referenzzähler niemals null werden konnte. Selbst wenn es in diesem Fall keine Referenzierung mehr durch das eigentliche Programm gab, so referenzierten sich die Objekte dennoch selbst. Ein Speicherleck war oft genug die Folge.

Das .NET Framework oder um genau zu sein, die Common Language Runtime, löst dieses Problem, indem sie ein komplett anderes Verfahren anwendet, Objekte zu entsorgen.

Der Garbage Collector – die Müllabfuhr in .NET

Den vorhandenen Speicher des Managed Heap teilen sich alle Assemblies, die in einer so genannten *Application Domain* (»Anwendungsdomäne«, kurz »AppDomain«) laufen. Normale Windows-Anwendungen, die man parallel startet, werden durch verschiedene, streng voneinander abgeschottete Prozesse isoliert. Ein Prozess kann unter normalen Umständen nicht auf einen anderen Prozess zugreifen, auch die Daten verschiedener Prozesse sind integer und können bestenfalls durch so genannte Proxys (Stellvertreter) untereinander ausgetauscht werden.

AppDomains in .NET, die durch die Common Language Runtime verwaltet werden, erlauben das gleichzeitige Ausführen mehrere Anwendungen in *einem* Prozess. Die CLR garantiert dabei, dass die Anwendungen ebenso isoliert und ungestört laufen können, wie das bei einem Windows-Prozess der Fall wäre. Jede AppDomain verwaltet einen Speicherbereich, in dem die notwendigen Daten der Assemblies und Programme,² die innerhalb der AppDomain laufen, abgelegt werden. Dieser Speicherbereich wird Managed Heap genannt, und mit ihm haben wir uns bislang schon einige Male beschäftigt.

Wenn der Speicher im Managed Heap (und auch sonst schon einmal aus anderen Gründen) knapp wird, dann startet ein Prozess, der im wahrsten Sinne des Wortes der Müllabfuhr im echten Leben entspricht: Die Garbage Collection findet statt.

² Streng genommen ist auch ein Programm eine Assembly.

Der Garbage Collector läuft in einem eigenen Thread,³ der die Objekte des Managed Heap dahingehend untersucht, ob sie noch in irgendeiner Form referenziert werden. Währenddessen hält der Garbage Collector im Übrigen alle anderen Threads an, da diese ihm natürlich durch das neue Reservieren von Objektspeicher ins Gehege kommen könnten. Objekte, denen eine noch gültige Referenzquelle zugeordnet werden kann, markiert der Garbage Collector. Im zweiten Teil sammelt der Garbage Collector alle markierten Objekte ein und ordnet sie im oberen Teil des Managed Heap an. Objekte, die nicht markiert waren, gibt der GC frei.

Der Algorithmus, mit dem der GC die verwendeten Objekte markiert, ist sehr hoch entwickelt. So erkennt der GC auch Objekte, die nur indirekt durch andere Objekte referenziert sind, und markiert sie. Bei dieser Vorgehensweise löst sich das COM-Problem der Zirkelverweise wie von selbst: Objekte, die von einer Anwendung der AppDomain aus nicht erreichbar sind, werden auch nicht markiert – selbst wenn sie sich untereinander referenzieren. Sie werden im zweiten Durchlauf des GC ebenfalls entsorgt.

Die Geschwindigkeit, mit der der GC seine Arbeit erledigt, ist erstaunlich hoch.⁴

Generationen

Die hohe Geschwindigkeit, mit der der GC arbeitet, bedingt sich insbesondere auch dadurch, dass der GC die zu testenden Objekte in Generationen klassifiziert. Bei der Entwicklung des GC-Algorithmus nahm man an, dass Objekte, die beim Start einer Applikation erzeugt werden, länger im Speicher verbleiben als solche, die irgendwann zwischendurch oder lokal in Prozeduren generiert werden. Diese Annahme führt zu dem Schluss, dass es Sinn ergibt, bei der Objektentsorgung eine Klassifizierung der Objekte in eben diese Generationen zur Optimierung des GC-Algorithmus vorzunehmen. Der Garbage Collector markiert Objekte nicht nur für die weitere Instandhaltung, er stattet sie auch mit einem Zähler aus, der aussagt, wie oft ein Objekt für die Entsorgung durch den Garbage Collection getestet wurde. Je öfter der Garbage Collector das Objekt bereits »besucht« und nicht entsorgt hat, desto älter ist logischerweise das Objekt (und umso höher ist demzufolge auch seine Generationsnummer), aber desto unwahrscheinlicher ist es auch, dass das Objekt in einem erneuten GC-Lauf entsorgt werden wird.

Wenn der Speicherplatz knapp wird, reicht es deshalb in der Regel aus, Objekte älterer Generationen zunächst außen vor zu lassen, denn die Wahrscheinlichkeit, dass sie entsorgt werden können, ist wie gesagt eher gering. Der GC kümmert sich in der Regel also nur um Generation-0-Objekte und versucht diese zu entsorgen. Erst wenn diese Vorgehensweise nicht geholfen hat, für genügend neuen freien Speicher zu sorgen, schaut der GC-Prozess, ob nicht auch bei Objekten älterer Generationen etwas zu holen ist.

³ Grob erklärt: Ein Thread ist ein Programmteil, der die Eigenschaft hat, neben anderen Programmteilen quasi gleichzeitig laufen zu können. Ein Anwendungsprogramm besteht mindestens aus einem Thread. Bei einem Windows-Programm wird der Thread, der die Benutzereingaben überwacht und als Ereignisse weiterleitet, übrigens *UI-Thread* genannt (»UI« als Abkürzung von User Interface = Benutzeroberfläche). Mehr zu Threads finden Sie ab Kapitel 44.

⁴ Übrigens: Das Grundprinzip des Garbage Collectors ist gar nichts Neues. Schon das alte Commodore-Basic (C64, VC20 – meines Wissens auch das Apple-II-Basic) kannte den Garbage Collector für die Entsorgung nicht mehr benötigter Variablen. Und es ist keine Blasphemie wenn man erwähnt, dass das durchaus ältere Java mit der Möglichkeit auffiel, nicht benötigten Speicher wieder freizugeben. Dies geschah damals durch eine sehr durchdachte und neuartige Technologie, die sich Garbage Collector nannte.

Leider wirft diese Vorgehensweise wieder ein Problem ganz anderer Art auf. Objekte können nicht wissen, wann sie entsorgt werden – denn nur die CLR entscheidet, wann ein GC-Durchlauf stattfindet (mit einer Ausnahme):

- Die Common Language Runtime fährt herunter. Das passiert in der Regel dann, wenn eine .NET-Applikation beendet wird.
- Der Speicher wird knapp, weil es zu viele Objekte gibt. Der Garbage Collector startet, um zu sehen, ob Generation-0-Objekte entsorgt werden können und entsorgt sie im Bedarfsfall.
- Der Garbage Collector ist in der AppDomain gezwungenermaßen durch die Anweisung `GC.Collect()` gestartet worden. – Aber ganz wichtig: Nicht zu Hause nachmachen, liebe Kinder, vor allem wenn der Klaus nicht da ist.

Hier hatte COM einen eindeutigen Vorteil. Wurde die letzte Referenz aufgelöst und der Referenzzähler stand auf 0, dann trat das Terminate-Ereignis ein, und das Objekt konnte zur »richtigen« Zeit die notwendigen Schritte einleiten, um sich zu entsorgen.

Normalerweise ist es gar kein Problem, dass ein Objekt nicht weiß, dass es entsorgt wird. Wenn es weg ist, dann ist es eben weg. Wichtig, den Zeitpunkt seiner Entsorgung zu kennen, wird es für ein Objekt erst dann, wenn es Aufräumarbeiten erledigen muss, und zwar nicht hinsichtlich der eigenen Speicherverwaltung (denn wenn es andere Objekte referenziert, sorgt der Garbage Collector ja ebenfalls für deren Entsorgung), sondern hinsichtlich der Freigabe von Ressourcen, auf die der Garbage Collector keinen Zugriff hat.

Dies wurde übrigens früher oder wird heute noch bei anderen OOP-Sprachen mit einem Destruktor erlebt. Genau wie wir den Konstruktor kennen gelernt haben, als »Ereignisprozedur« die ausgeführt wird, wenn ein Objekt erstellt wird, so wird der Destruktor ausgeführt, wenn das Objekt zerstört wird.

Das sind beispielsweise Fälle, in denen das Objekt ein Handle⁵ auf eine bestimmte Geräte- oder Betriebssystemressource erhalten hat. Damit dieses Handle wieder freigegeben werden kann – eine geöffnete Datei beispielsweise sollte geschlossen werden – muss das Objekt die dafür erforderlichen Aktionen spätestens kurz bevor es vom Garbage Collector zerstört wird, durchführen.

Genau das geht nicht mehr in .NET. Objekte können nicht voraussehen oder den genauen Zeitpunkt erfahren, wann sie entsorgt werden. Es gibt allerdings die Möglichkeit, dass Objekte erfahren, *dass* sie entsorgt werden, und dann die notwendigen Schritte einleiten, um Ressourcen, die sie belegen, freizugeben.

Daher spricht man in .NET übrigens auch von »nicht-deterministischen Destruktoren«, es ist also nicht voraussagbar (determiniert), wann der Destruktor läuft: Nämlich dann, wenn es dem GC passt und wir wissen nicht, wann es ihm passt.

Die Geschwindigkeit der Objektbereitstellung

.NET bzw. die .NET-Infrastruktur hatte seit ihrer ersten Veröffentlichung mit dem Vorurteil der langsamem Geschwindigkeit zu kämpfen. Nun liegt es in der Natur von Vorurteilen, dass an Einigen von ihnen immer auch ein Quäntchen Wahres dran ist. Doch man sollte hier und da gerade in Sachen Geschwindigkeit auch

⁵ Eine vom Betriebssystem erteilte Kennung zur Nutzung einer bestimmten Ressource (Datei, Bildschirm, Schnittstellen, spezielle Windows-Betriebssystemobjekte, etc.).

einmal eine Lanze für die .NET-Infrastruktur brechen. Und wenn schon das On-Demand-Kompilieren durch den JITter der CLR sicherlich eine Technik mit vielen Vorteilen bildet, so sicherlich nicht in Sachen Geschwindigkeit.

Doch dafür gibt es einige Vorteile der .NET-Infrastruktur in Sachen Geschwindigkeit, und die liegt beim Schaffen von Speicherplatz für ein neues Objekt. Und seien wir ehrlich: Es ist in fast jedem Szenario besser, Objekte so schnell wie möglich zu erstellen, als sie so schnell wie möglich zu entsorgen. Und warum ist das Erstellen der Objekte so viel schneller als sagen wir beim herkömmlichen Runtime Heap eines typischen C-basierten Systems?

Bei einem C-Runtime-Heap liegen alle speicherreservierten Objekte (man sagt auch *allokierte* Objekte) als verkettete Liste vor. Wenn nun Speicherplatz für ein neues Objekt benötigt wird, muss die C-Runtime durch diese Liste iterieren, und falls Sie zwischen zwei Objekten ausreichenden Speicherplatz findet, bricht sie die Kette an dieser Stelle auf und platziert das neue Objekt an dieser Stelle.

Im .NET Framework funktioniert das Reservieren des Speichers anders. Es gibt einen Adresszeiger der auf den Speicherplatz zeigt, an dem das jeweils nächste Objekt abgelegt werden soll; neue Objekte werden also nicht irgendwo mittendrin platziert, sondern immer am Ende der Liste der Objekte im Managed Heap. Dieser Adresszeiger – im .NET Framework wird er übrigens *NextObjPtr* genannt – stellt also den jeweils nächsten Speicherstart für ein Objekt jederzeit zur Verfügung. Das Reservieren des Speichers kostet also, anders als bei den gängigen C-Runtime-Systemen – im .NET Framework keine erwähnenswerte Zeit, und das ist ein klarer Vorteil, den die CLR geschwindigkeitstechnisch für sich verbuchen kann.

In C-Runtime-Systemen kann es aus diesem Grund auch passieren, dass Objekte, die nacheinander erstellt werden, spechertechnisch weit voneinander entfernt sind. Auch das passiert beim .NET Framework nicht. Objekte, die nacheinander erstellt werden, liegen auch hintereinander im Speicher.

Das ist natürlich hinsichtlich der Wahrscheinlichkeit sinnvoll, wie und wann für Objekte Speicher erstellt wird. Wenn Sie Speicher für eine Bitmap reservieren, und lassen wir dabei jetzt mal die Tatsache außen vor, dass dafür systembedingt auch Speicher anderen Typs reserviert werden muss, dann ist es auch wahrscheinlich, dass im gleichen Kontext, auch zeitlich, Objekte für Fonts, Pinsel und Brushes erstellt werden müssen. Und vor allen Dingen, dass der Speicher dieser Objekte physisch in einem der schnellen Caches des Prozessors liegt. Bislang scheint also die .NET Framework-Speicherverwaltung nur Positives zu bringen, wenn es da nicht ein kleines Problem geben würde: Uns steht zwar inzwischen sehr, sehr viel Speicher zur Verfügung, aber dennoch ist dieser Speicher nicht unerschöpflich. Irgendwann muss eben ein Aufräumprozess stattfinden – die Garbage Collection – und die kostet Zeit.

Aber immerhin: Der Algorithmus arbeitet eben mit den schon angesprochenen Generationen, und der Generation 0-Algorithmus ist so ausgelegt, dass er in der Regel erst dann »anspringt«, solange sich der benötigte Speicher noch im mindestens in einem Speicherbereich befindet, der in der Regel in den schnellen Cache des Prozessors passt.

Der Kompromiss, den wir Entwickler eingehen müssen, um in den Genuss einerseits der extrem schnellen Speicheranforderung zu kommen, gepaart mit dem Vorteil, dass die Objekte, die wir gleichzeitig benutzen, möglichst eng beieinander und damit mit großer Wahrscheinlichkeit wieder im Prozessorcache liegen, ist damit durchaus zu machen, wie ich finde.

Finalize

Wenn ein Objekt vom Garbage Collector zur Entsorgung markiert wurde, dann ruft der Garbage Collector in der Regel die `Finalize`-Methode des Objektes auf, bevor er den Speicher des Objektes endgültig freigibt. Schon die `Object`-Klasse hat `Finalize` implementiert, und da alle Klassen von `Object` abgeleitet sind, hat jedes Objekt in .NET eine `Finalize`-Methode.⁶

Die `Finalize`-Methode in ihrer Grundimplementierung von `Object` macht überhaupt nichts. Sie ist in erster Linie einfach nur vorhanden, und das bedeutet, dass ein Objekt die `Finalize`-Methode überschreiben muss, wenn es eine eigene Funktionalität für seine »Entsorgungsvorbereitung« implementieren will.

Der Finalizer ist also im Prinzip der (leider eben nicht deterministische) Destruktor von .NET Klassen.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C – OOP\\Kapitel 18\\Finalize01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module mdlMain

Sub Main()
    Dim locTest As New Testklasse("Erste Testklasse")
    Dim locTest2 As New Testklasse("Zweite Testklasse")
    locTest = Nothing
    locTest2 = Nothing
    'GC.Collect()
    Console.WriteLine("Beide Objekte sind nun nicht mehr in Verwendung!")
End Sub

End Module

Class Testklasse

    Private myName As String

    Sub New(ByVal Name As String)
        myName = Name
    End Sub

    Protected Overrides Sub Finalize()
        MyBase.Finalize()
        Console.WriteLine(Me.myName & " wurde entsorgt")
    End Sub
End Class
```

⁶ Wobei `Finalize` von `Object` streng genommen gar nicht im Rahmen des GCs aufgerufen wird, da es ohnedies nichts macht; der GC-Algorithmus findet heraus, ob ein »neues« `Finalize` implementiert wurde, und `Finalize` wird nur dann aufgerufen, wenn die `Finalize`-Methode überschrieben wurde.

TIPP Starten Sie dieses Programm mit der Tastenkombination **Strg F5**, also ohne Debuggen, damit das Konsolenfenster nach dem Beenden des Programms nicht einfach wieder verschwindet. Sie würden das Ergebnis sonst nicht sehen können.

Wenn Sie dieses Programm starten, dann stellen Sie fest, dass die Meldung »Beide Objekte sind nun nicht mehr in Verwendung!« zuerst ausgegeben wird. Erst anschließend erscheinen die Texte, die anzeigen, dass die beiden verwendeten Objekte finalisiert worden sind:

```
Beide Objekte sind nun nicht mehr in Verwendung!
Zweite Testklasse wurde entsorgt.
Erste Testklasse wurde entsorgt
```

Die Ursache dafür liegt schon fast auf der Hand: Der Garbage Collector arbeitet in diesem Programm nicht, während es läuft, und bei einem Speicheraufkommen von nur ein paar Bytes hat er dafür auch gar keinen Grund. Das Finalisieren der Objekte findet dennoch statt, und zwar beim Beenden der Anwendung. Zu diesem Zeitpunkt ist die letzte Zeile des eigentlichen Programms aber längst verarbeitet worden; in diesem Fall war es die Codezeile, die den Meldungstext auf den Bildschirm ausgegeben hat.

Eine andere Ausgabe erscheint, wenn Sie das Kommentarzeichen vor der Zeile

```
'GC.Collect()
```

weglassen. Starten Sie das Programm anschließend, ändert sich die Ausgabe in:

```
Zweite Testklasse wurde entsorgt
Erste Testklasse wurde entsorgt
Beide Objekte sind nun nicht mehr in Verwendung!
```

WICHTIG Im Beispielprogramm habe ich die `Console`-Klasse in der `Finalize`-Methode wie selbstverständlich verwendet. Machen Sie das nicht. Mal ganz davon abgesehen, dass Sie in der `Finalize`-Methode keine wie auch immer gearteten Bildschirmausgaben mehr machen sollten, um sie so schnell wie möglich hinter sich zu bringen, können Sie sich auch nicht sicher sein, ob Objekte, die Sie verwenden, zu diesem Zeitpunkt noch bestehen.

Wann Finalize nicht stattfindet

Unter Umständen verursachen Sie beim Verwenden bestimmter Objekte in `Finalize`, dass neue Objekte während des Vorgangs entstehen, die ihrerseits wiederum neue Objekte anlegen, usw. Diese Objekte müssen natürlich anschließend ebenfalls finalisiert werden. Im schlimmsten Fall lösen Sie damit eine solch enorme Kaskade von neuen Objekten aus, dass diese alle niemals finalisiert werden könnten. Doch ihre Anwendung hängt sich deshalb nicht auf, denn das Framework hat zu diesem Zweck ein paar Sicherheitsmaßnahmen vorgesehen.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

... \VB 2008 Entwicklerbuch\ C - OOP\ Kapitel 18\ FinalizeNoNo01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei). Denken Sie daran, dass dieses Beispiel zeigt, was Sie in Ihren Anwendungen niemals machen sollten!

```
Module mdlMain

    Sub Main()
        Dim locTest As New Testklasse("Testklasse")
    End Sub

End Module

Class Testklasse

    Private myName As String

    Sub New(ByVal Name As String)
        myName = Name
    End Sub

    Sub WriteText()
        Console.WriteLine(myName)
    End Sub

    Protected Overrides Sub Finalize()
        MyBase.Finalize()
        Dim locTemp As New Testklasse("locTemp")
        WriteText()
        Console.WriteLine(" wurde entsorgt")
    End Sub
End Class
```

Wenn Sie dieses Programm starten, wird eine Reihe von Meldungen ausgegeben. Finalize selbst legt dabei dummerweise eine neue Instanz von Testklasse an, um eine Meldung auszugeben. Diese Instanz muss natürlich ebenfalls finalisiert werden, und sie legt ihrerseits wieder eine neue Testklasse-Instanz an usw. Der GC erkennt nach einer Weile, dass der Finalisierungsprozess genau das Gegenteil von dem bewirkt, was er eigentlich bewirken sollte, es entstehen nämlich immer mehr Objekte, und der Speicherbedarf wächst und wächst. Er bricht das Finalisieren nach ein paar Sekunden schlicht und ergreifend ab.

Ein weiteres schlechtes Beispiel ist das Folgende (wenn Sie es unbedingt selbst probieren wollen: Sie finden es im Begleitdateienverzeichnis unter \FinalizeNoNo02). Es legt zwar keine Unmenge von neuem Speicher an, verbraucht für den Finalisierungsprozess aber einfach zu viel Zeit. Der GC wird nach vergleichsweise kurzer Zeit ungeduldig und bricht den gesamten Finalisierungsprozess ebenfalls wieder ab. Das im Beispielprogramm zuerst deklarierte Objekt bekommt keine Chance mehr, finalisiert zu werden.

```
Module mdlMain

Sub Main()
    '"Normales" Objekt, könnte problemlos finalisiert werden.
    Dim locTest1 As New Testklasse(False, "Erstes Testobjekt")
    'Der Störenfried, da Warteschleifenflag gesetzt.
    Dim locTest2 As New Testklasse(True, "Zweites Testobjekt")
End Sub

End Module

Class Testklasse

    'Dieses Flag steuert den Einstieg in die Warteschleife.
    Private myWaitInFinalize As Boolean
    'Eine Eigenschaft zum Unterscheiden von Klasseninstanzen
    Private myName As String

    'Flag fürs Warten und den Namen definieren.
    Sub New(ByVal WaitInFinalize As Boolean, ByVal Name As String)
        myWaitInFinalize = WaitInFinalize
        myName = Name
    End Sub

    Protected Overrides Sub Finalize()
        MyBase.Finalize()

        'Nur wenn das Flag bei New gesetzt
        'wurde, in die Warteschleife springen.
        If myWaitInFinalize Then

            Dim locSecs As Integer
            Dim lastSec As Integer
            lastSec = Now.Second
            Do
                'Jede Sekunde eine Meldung ausgeben.
                If lastSec <> Now.Second Then
                    lastSec = Now.Second
                    locSecs += 1
                    Console.WriteLine("Warte bereits {0} Sekunden", locSecs)
                    'Nach 60 Sekunden wäre Schluss.
                    If locSecs = 60 Then Exit Do
                End If
            Loop

            End If
            'Erfolgreich finalisiert --> Meldung ausgeben
            Console.WriteLine("Objekt {0} wurde finalisiert!", myName)
        End Sub
    End Class
```

Folgende Punkte sind also wichtig, wenn Sie eine eigene Finalisierungslogik in Ihren Klassen implementieren müssen:

- Achten Sie darauf, dass der Finalisierungs-Prozess nicht nur so schnell wie möglich erledigt ist, sondern wirklich kaum Zeit beansprucht.
- Stellen Sie sicher, dass Sie *auf keinen Fall* neue Instanzen von irgendwelchen Objekten innerhalb des Finalisierungsprozesses erstellen.

Wenn Sie diese Punkte beherzigen, tragen Sie erheblich zum einwandfreien Funktionieren Ihrer Klassen bei. Es gibt übrigens Objekte im Framework, deren Vorhandensein die CLR auch noch zum Finalisierungszeitpunkt garantiert. Da Sie Ausgaben bei der Finalisierung wahrscheinlich nur zu Testzwecken machen werden, verwenden Sie dafür besser die Debug-Klasse. Diese Klasse stellt ebenfalls eine Write- bzw. WriteLine-Methode zur Verfügung, hat aber gegenüber Console entscheidende Vorteile: Das Vorhandensein der Klasse zum Finalisierungszeitpunkt ist garantiert, und Ausgaben erfolgen darüber hinaus nur in einen so genannten Trace-Listener⁷.

Soweit zur Finalisierung von Objekten durch das Framework. Finalize ist allerdings eine Methode, die ausschließlich durch das Framework aufgerufen werden darf. Was ist aber, wenn Sie Objekte erstellen wollen, die der Anwender selber – quasi – schließen oder freigeben will?

Das Framework bietet dazu ein Schnittstellenmuster über die so genannte IDisposable- Schnittstelle an. Der nächste Abschnitt verrät mehr über dieses Thema.

Dispose

Mit der IDisposable-Schnittstelle stellt das Framework eine Implementierungsvorschrift bereit, mit deren Hilfe Sie eine Methode implementieren können, die anders als Finalize, auch aus Ihrem Code heraus aufgerufen werden darf, um das Objekt zu entsorgen. Wenn Sie die Schnittstelle per Implements in Ihrer Klasse implementieren, müssen Sie die Dispose-Methode einfügen, die dann für das notwendige Aufräumen die Verantwortung trägt. Damit karriert man natürlich die Funktion eines echten Destruktors in anderen OOP Sprachen, der eben automatisch (man braucht sich halt nicht darum zu kümmern) aufgerufen wird. Der Aufruf von Dispose ist daher ein Kompromiss, den man für die vielen Vorteile des Garbage Collectors eingeht.

HINWEIS Verwenden Sie Using als Strukturblock, um ein implizites Dispose für das verwendete Objekt zu erreichen, wenn das Programm den Gültigkeitsbereich des Strukturblocks verlässt. Ein Beispiel zu Using finden Sie in Kapitel 10 im Abschnitt »Gezieltes Freigeben von Objekten mit Using«.

Diese Aufräumarbeiten sind prinzipiell die gleichen Arbeiten, die auch eine Finalize-Methode durchführt. Doch Dispose hat ein wenig mehr Arbeit. Denn wenn Sie Aufräumarbeiten mit Dispose durchgeführt haben, dann müssen Sie dafür sorgen, dass der GC die Aufräumarbeiten innerhalb Ihres Objektes nicht noch einmal durch den Aufruf dieser Methode erledigt. Zu diesem Zweck gibt es die SuppressFinalize-Methode des Garbage Collectors. Rufen Sie diese Methode auf und übergeben Sie ihr Ihre Klasse als Argument, schließt der Garbage Collector sie für alle folgenden GC-Durchläufe von Aufräumarbeiten dieser Art aus.

⁷ Ein speicherresistentes Programm, das spezielle Debug-Ausgaben »abhört« und in eigenen Fenstern ausgibt oder sonst wie protokolliert. Falls Sie Programme in der Entwicklungsumgebung von Visual Studio laufen lassen, dann ist das Ausgabefenster der vorinstallierte Trace-Listener. Alle Ausgaben, die Sie mit Debug.WriteLine() durchführen, gelangen dann ins Ausgabefenster.

Nun besteht die eigentliche Aufgabe der `Dispose`-Methode darin, zu unterscheiden, ob ein Aufruf durch das eigene Programm eben über `Dispose` (üblich ist bei bestimmten Klassen auch `Close`, das nichts anderes macht als `Dispose`, nur dass es einen anderen Namen trägt⁸) oder durch den Garbage-Collector über `Finalize` erfolgt ist. Ihr `Dispose` muss ebenfalls dafür sorgen, dass erkannt wird, ob ein Objekt aus Ihrer Sicht schon entsorgt wurde, und im Bedarfsfall eine Ausnahme auslösen.

Ein Beispiel für die Implementierung einer vollständigen `Finalize/Dispose`-Lösung finden Sie in der folgenden Anwendung. Es stellt der Hauptanwendung eine Klasse namens `SoapSerializer` zur Verfügung. Mithilfe dieser Klasse können Sie beliebige Objekte im SOAP-Format in einer Datei speichern (mehr zum Serialisieren von Objekten erfahren Sie im Kapitel 24). Umgekehrt kann die Klasse aus einer Datei die ursprünglichen Objekte wieder automatisch herstellen.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 18\\Dispose

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Das Hauptprogramm, das Sie im Folgenden finden, ist dabei sehr einfach gehalten, selbst von eher untergeordnetem Interesse und auch wieder eine Konsolenanwendung. Es fragt den Anwender nach dem Programmstart, ob er eine SOAP-Datei laden und deren Daten anzeigen oder Daten erfassen und abspeichern möchte. Es bedient sich dabei zur Speicherung der Daten einer Klasse, die lediglich Namen und Vornamen einer Person aufnimmt:

```
Möchten Sie Daten erfassen und speichern (1)
oder Daten laden und anzeigen (2)?
Ihre Auswahl :1
Wieviele Daten möchten Sie eingeben? :4
```

```
-----  
Eingabe der 1. Person  
Nachname: Heckhuis  
Vorname: Jürgen
```

```
-----  
Eingabe der 2. Person  
Nachname: Thiemann  
Vorname: Uwe
```

```
-----  
Eingabe der 3. Person  
Nachname: Ademmer
```

⁸ Aber wie heißt es so schön: Ausnahmen bestätigen die Regel. Die `System.Windows.Forms`-Klasse gehört hierzu: Wenn Sie ein Formular mit `Close` schließen, haben Sie die Möglichkeit, das Schließen des Formulars im `FormClosing`-Ereignis zu verhindern (indem Sie `e.Cancel` auf `True` setzen). »Schließen« Sie das Formular jedoch mit `Dispose`, was auch geht, dann »schießen« Sie es sozusagen ab. Es ist sofort zu, weg, verschwunden, entsorgt, und es löst auch keine Ereignisse mehr aus. Bei anderen Objekten wie denen, die Dateien lesen oder schreiben, wird ebenfalls `Close` benutzt. Entweder um aus traditionellen Gründen eine Datei eben zu »schließen« – oder weil die Programmierer nicht zu den Meetings erschienen sind, in denen man sich auf `Dispose` geeinigt hatte.

```
Vorname: Ute
```

```
-----  
Eingabe der 4. Person  
Nachname: Löffelmann  
Vorname: Klaus
```

Wenn Sie den letzten Namen eingegeben haben, wird das Programm auch schon beendet. Die Daten befinden sich anschließend in der Datei »Test.xml« auf Laufwerk »C:«.

Lassen Sie sich diese Datei ruhig einmal im Texteditor anzeigen! Man möchte sagen: Zwar ein ziemlicher SOAP-Overhead für diese paar Daten, aber was soll's: Es ist nur ein Demo und (Festplatten-)Speicher ist billig! Viel wichtiger ist, dass das Programm auch funktioniert, und das finden Sie heraus, indem Sie das Programm abermals starten und anschließend die Funktion zum Anzeigen der Daten auswählen:

```
Möchten Sie Daten erfassen und speichern (1)  
oder Daten laden und anzeigen (2)?  
Ihre Auswahl :2  
Heckhuis, Jürgen  
Thiemann, Uwe  
Ademmer, Ute  
Löffelmann, Klaus
```

Wie arbeitet das Programm nun? Bevor ich zur Erklärung schreite, eine kleine Warnung vorweg: Die Funktionsweise des Programms ist recht wichtig für das spätere Verständnis von `Dispose` und `Finalize`. Wundern Sie sich also bitte nicht, wenn ich zunächst auf den folgenden Seiten ein paar andere Themen aufgreife, bevor wir uns dann dem eigentlichen Gegenstand der Erklärung widmen.

Zurück zum Programm: Zunächst gibt es eine Klasse, die die Daten speichert. Wichtig dabei: Wenn Sie eine Klasse serialisieren, dann müssen folgende Voraussetzungen gegeben sein:

- Alle Datentypen, die die Klasse verwendet, müssen serialisierbar sein.
- Die Klasse selbst muss serialisierbar sein und dazu mit einem besonderen Attribut gekennzeichnet werden:

```
<Serializable()> _  
Class Dataset  
  
    Private myFirstName As String  
    Private myLastName As String  
  
    Sub New(ByVal FirstName As String, ByVal LastName As String)  
        myFirstName = FirstName  
        myLastName = LastName  
    End Sub  
  
    Overrides Function ToString() As String  
        Return myLastName & ", " & myFirstName  
    End Function  
  
End Class
```

Das Hauptmodul ist dafür zuständig, die Daten zu erfassen und abzuspeichern bzw. zu laden und auf dem Bildschirm anzuzeigen. Sie werden überrascht sein, wie wenig Aufwand für das Sichern bzw. das Wiederherstellen erforderlich ist:

```
Module mdlMain
Sub Main()
    '"Menü" auf den Bildschirm zaubern:
    Console.WriteLine("Möchten Sie Daten erfassen und speichern (1)?")
    Console.WriteLine("oder Daten laden und anzeigen (2)? ")
    Console.Write("Ihr Auswahl :")

    'Auswahl einlesen
    Dim locKey As String = Console.ReadLine()

    'Daten sollen erfasst werden.
    If locKey = "1" Then

        Dim locAnzPersonen As Integer
        Dim locName, locVorname As String
        Dim locSoapWriter As SoapSerializer

        Console.Write("Wieviele Daten möchten Sie eingeben? :")
        locAnzPersonen = Integer.Parse(Console.ReadLine())
        If locAnzPersonen = 0 Then
            Exit Sub
        End If

        'Serializer vorbereiten.
        locSoapWriter = New SoapSerializer

        'Zum Schreiben öffnen.
        locSoapWriter.OpenForWriting("C:\Test.XML", True)

        'Anzahl der Datensätze abspeichern.
        locSoapWriter.SaveObject(locAnzPersonen)

        'Soviele Personen einlesen, wie zuvor eingegeben.
        For locCount As Integer = 1 To locAnzPersonen
            Console.WriteLine()
            Console.WriteLine("-----")
            Console.WriteLine("Eingabe der {0}. Person", locCount)
            Console.Write("Nachname: ")
            locName = Console.ReadLine
            Console.Write("Vorname: ")
            locVorname = Console.ReadLine

            'In das Objekt übertragen und abspeichern.
            Dim locData As New Dataset(locVorname, locName)
            locSoapWriter.SaveObject(locData)
        Next

        'Das kann man schon mal vergessen!!!
        locSoapWriter.Close()
```

```

'Der umgekehrte Weg: Die Daten werden geladen.
Else

    Dim locAnzPersonen As Integer
    Dim locData As Dataset
    Dim locSoapReader As SoapSerializer

    'Deserializer vorbereiten.
    locSoapReader = New SoapSerializer

    'Zum Lesen öffnen.
    locSoapReader.OpenForReading("C:\Test.XML")

    'Anzahl der Datensätze lesen und
    'zurückboxen von Object zu Wertetyp Integer.
    locAnzPersonen = Convert.ToInt32(locSoapReader.LoadObject())

    'Soviele Personen-Datensätze lesen, wie ursprünglich erfasst.
    For locCount As Integer = 1 To locAnzPersonen
        'Deserialisieren und in den alten Objekttyp zurückwandeln.
        locData = DirectCast(locSoapReader.LoadObject(), Dataset)
        'Daten ausgeben.
        Console.WriteLine(locData.ToString)
    Next

    'So geht es auch:
    locSoapReader.Dispose()

    End If
End Sub

End Module

```

Sie sehen: Womit das Programm am meisten zu tun hat, ist das Ausgeben der »Menü-Texte« auf dem Bildschirm. Die eigentliche Arbeit erledigt die *SoapSerializer*-Klasse, der wir uns als Nächstes und erklärungstechnisch ein wenig intensiver widmen wollen.

Die Klasse befindet sich in einer eigenen Datei im Projekt (namens *SoapSerializer.vb*). Doppelklicken Sie auf den Dateinamen im Projektmappen-Explorer, um sie im Codeeditor betrachten zu können.

Der Programmcode beginnt mit einer Reihe von Imports-Anweisungen, die verschiedene Namespaces einbinden.

```

Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Soap

```

Da das Programm sowohl die *MemoryStream*-Klasse als auch die Serialisierung auf *Soap*-Basis verwendet, steht die Imports-Anweisung für die Namespaces, denen diese Klassen zugeordnet sind, an erster Stelle in der Codedatei. Zusätzlich gibt es eine Referenz auf die .NET Framework-Assembly *System.Runtime.Serialization.Formatters.Soap*, die zuvor über den Projektmappen-Explorer mit *Verweis hinzufügen* eingebunden wurde.

```
Public Enum SoapSerializerMode
    Close
    OpenForWriting
    OpenForReading
End Enum
```

Die *Enum* benötigen wir lediglich, um das Programm leichter lesbar zu machen (mehr zum Thema *Enum* finden Sie in Kapitel 21).

```
Public Class SoapSerializer
    Implements IDisposable
```

Mit der *Implements*-Anweisung bindet die Klasse die *IDisposable*-Schnittstelle ein. Zur Wiederholung: Bei einem Schnittstellen-*Pattern* wird eine Schnittstelle in erster Linie zur Standardisierung verwendet. Erst in zweiter Linie dient sie, wenn überhaupt, zur Realisierung von polymorphen Aufrufen von Methoden in abgeleiteten Klassen. Die *IDisposable*-Schnittstelle zwingt den Entwickler einer Klasse, die *Dispose*-Methode in einer bestimmten Form zu implementieren. Das Framework selbst ruft, wie ebenfalls bereits erwähnt, *Dispose* nie direkt auf.

```
Protected myFilename As String
Protected myMemoryStream As MemoryStream
Protected mySoapFormatter As SoapFormatter
Protected mySerializerMode As SoapSerializerMode
Protected myDisposed As Boolean

Sub New()
    mySoapFormatter = New SoapFormatter(Nothing,
                                         New StreamingContext(StreamingContextStates.File))
    mySerializerMode = SoapSerializerMode.Close
End Sub
```

Der *SoapFormatter* wird für die Serialisierung der Objekte verwendet. Er steuert quasi »das Aussehen« der Daten, wenn Objekte serialisiert werden. In diesem Zusammenhang möchte ich nicht näher darauf eingehen.

```
Function OpenForWriting(ByVal Filename As String) As Boolean
    Return OpenForWriting(Filename, False)
End Function

Function OpenForWriting(ByVal Filename As String, ByVal OverwriteIfExist As Boolean) As Boolean
    Dim locFile As New FileInfo(Filename)

    If (Not OverwriteIfExist) And locFile.Exists Then
        Return True
    End If

    myFilename = Filename

    Try
        mySerializerMode = SoapSerializerMode.OpenForWriting
        myMemoryStream = New MemoryStream()
    Catch ex As Exception
```

```

mySerializerMode = SoapSerializerMode.Close
End Try

End Function

Function OpenForReading(ByVal Filename As String) As Boolean

    Dim locFile As New FileInfo(Filename)

    If Not locFile.Exists Then
        Return True
    End If

    Try
        mySerializerMode = SoapSerializerMode.OpenForReading
        myMemoryStream = New MemoryStream(My.Computer.FileSystem.ReadAllBytes(Filename))
    Catch ex As Exception
        mySerializerMode = SoapSerializerMode.Close
    End Try

End Function

```

Diese Funktionen erstellen, je nach Anforderung, einen so genannten `MemoryStream`, in den eine Objektserialisierung oder Deserialisierung im Speicher vorgenommen werden kann. Die Vorgehensweise beim Serialisieren in eine Datei ist einfach:

- `MemoryStream` erstellen,
- Objekt in diesen Speicher »hineinserialisieren«,
- `MemoryStream` nach Serialisierung aller Objekte in eine Datei schreiben.

Analog funktioniert das Deserialisieren, bei dem Daten einer Datei, die in einem bestimmten Format vorliegen, zunächst in einen `MemoryStream` geladen werden, aus dem dann wiederum die verschiedenen Objekte »gewonnen« werden können.

Zum nächsten Punkt:

```

Sub SaveObject(ByVal Data As Object)

    'Serialisierung geht nur, wenn SoapSerializer
    'zum Schreiben geöffnet wurde.
    If mySerializerMode <> SoapSerializerMode.OpenForWriting Then
        Dim Up As New IOException("SoapSerializer nicht zum Schreiben geöffnet!")
        Throw Up
    End If

    mySoapFormatter.Serialize(myFileStream, Data)

End Sub

Function LoadObject() As Object

    'Deserialisierung geht nur, wenn SoapSerializer
    'zum Lesen geöffnet wurde.

```

```
If mySerializerMode <> SoapSerializerMode.OpenForReading Then
    Dim Up As New IOException("SoapSerializer nicht zum Lesen geöffnet!")
    Throw Up
End If
Return mySoapFormatter.Deserialize(myFileStream)

End Function
```

Sie sehen, wie einfach das Serialisieren und Deserialisieren von Objekten im Grunde genommen ist. Mit jeweils einem einzigen Befehl können Sie aus einem Objekt einen Datenstrom oder aus einem Datenstrom wieder ein Objekt machen. Beim Deserialisieren lässt die CLR das Objekt in seiner ursprünglichen Gestalt »wieder auferstehen«. Am einfachsten zu vergleichen ist das mit dem Vorgang des Beamens in StarTrek. Wenn Sie eine Person an einen anderen Platz beamen, wird sie zunächst serialisiert, dann in einem Strom (Strahl, o.k.) ans Ziel geschickt und dort wieder deserialisiert. Der Unterschied: Im Framework funktioniert das »zum Leben erwecken« von Objekten schon in unserem Jahrhundert tatsächlich ...

```
'Nur eine andere Form von Disposed
Sub Close()
    Debug.WriteLine("Close() wurde aufgerufen!")
    Dispose()
End Sub

'Hier wird das Entsorgen delegiert
Public Sub Dispose() Implements IDisposable.Dispose
    'Wir kümmern uns um die Entsorgung,
    'der Garbage Collector wird informiert,
    'dass er nichts mehr damit zu tun hat
    Debug.WriteLine("Dispose() wurde aufgerufen!")
    GC.SuppressFinalize(Me)
    Dispose(True)
End Sub

Public Sub Dispose(ByVal Disposing As Boolean)

    Debug.WriteLine("Dispose(Disposing) wurde aufgerufen...")
    'Falls der Aufruf nicht durch die GC kam
    If Disposing Then
        Debug.WriteLine("...kam von der Applikation")
        'prüfen, ob nicht schon entsorgt
        If myDisposed Then
            'Übergründlich geht nicht, dann --> Exception
            Dim up As New ObjectDisposedException("SoapSerializer")
            Throw up
        End If
    Else
        Debug.WriteLine("...kam vom Garbage Collector")
    End If

    'An dieser Stelle werden die Daten in die Datei geschrieben
    'So kann verhindert werden, dass die Datei die ganze Zeit
    'geöffnet bleibt.
    If mySerializerMode = SoapSerializerMode.OpenForWriting Then
        My.Computer.FileSystem.WriteAllBytes(myFilename, myMemoryStream.ToArray, False)
    End If
End Sub
```

```

        Debug.WriteLine("Daten aus MemoryStream geschrieben - Datei ist sicher!")
        myMemoryStream.Dispose()
    End If
    Debug.WriteLine("MemoryStream disposed")
    mySerializerMode = SoapSerializerMode.Close
End Sub

Protected Overrides Sub Finalize()
    'Falls myMemoryStream schon durch den GC entsorgt wurde
    'könnte ein Fehler auftreten, den es abzufangen gilt
    Try
        Debug.WriteLine("Me.Finalize ruft Me.Dispose auf!")
        Dispose(False)
    Finally
        Debug.WriteLine("Base.Finalize aufrufen!")
        MyBase.Finalize()
    End Try
End Sub

End Class

```

Diese letzten Zeilen der Klasse haben es in sich, und nach dem ganzen Vorgeplänkel sind wir leider erst jetzt beim eigentlichen Thema.

Klären wir zunächst die Frage: Wozu braucht diese Klasse überhaupt ein Finalize und ein Dispose? Die Klasse verwendet ein MemoryStream-Objekt. Und: Die Klasse schreibt zunächst in diesen »Dateistrom«, aber erst bei der Ausführung von Close den erstellten Datenstrom in eine Datei. Wenn Sie Daten in einen Datenstrom hineinschreiben, dann müssen Sie sicherstellen, dass Daten, die sich dort befinden, sich später auch bis aufs letzte Byte in der angegebenen Datei befinden. Das gewährleistet während des Close- bzw. Dispose-Vorgangs die Zeile, die Sie im oben stehenden Listing in fetter Schrift sehen.

Nun öffnet unsere Klasse ein MemoryStream-Objekt automatisch, wenn der Entwickler, der die Klasse verwendet, eine der beiden Methoden OpenForWriting oder OpenForReading verwendet. Er kann nun mit SaveObject bzw. LoadObject den Datenstrom verwenden. Er muss anschließend aber auch – und jetzt kommt der IDisposable-Pattern ins Spiel – dafür sorgen, dass alles wieder geschlossen wird, nur dann wird – und das ist wichtig im Falle des Schreibens – der zunächst im Speicher angelegte Datenstrom mit den Objektdaten tatsächlich in die Datei geschrieben. Macht er es nicht, dann sollte unsere Klasse intelligent genug sein, um zu retten, was zu retten ist. Die Klasse sorgt also für das Speichern des Speicherdatenstroms, wenn ...

- ... der Entwickler die Dispose-Methode (oder die Close-Methode – das ist in diesem Fall dasselbe), so wie es sein sollte, selbst aufruft, oder
- ... der Entwickler es vergessen hat, aber der Garbage Collector uns durch den Aufruf von Finalize anzeigt, dass die Klasse zur Entsorgung ansteht, und spätestens jetzt alle verwendeten Ressourcen möglichst schnell aufgeräumt und freigegeben werden sollten.

Nun könnte man meinen, es reiche aus, Finalize einfach Dispose oder umgekehrt aufrufen zu lassen und die Implementierung zum korrekten Freigeben der Ressourcen einfach in einer der beiden Routinen zu verstauen. Das geht aber leider nicht, denn es gibt die drei folgenden Einschränkungen:

- Finalize darf nur vom Garbage Collector aufgerufen werden; Finalize der Basisklasse muss dabei obendrein grundsätzlich, immer und um jeden Preis aufgerufen werden.
- Dispose darf maximal einmal aufgerufen werden, denn ein einmal entsorgtes Objekt kann nicht noch einmal entsorgt werden. Wird Dispose ein zweites Mal aufgerufen, sollte die Klasse eine ObjectDisposedException ausgeben.
- Der Garbage Collector darf Finalize nicht aufrufen, wenn das Objekt bereits quasi »durch sich selbst« (also durch Dispose) entsorgt wurde.

Genau diese drei Fälle werden im Code berücksichtigt:

Sobald der Anwender die Klasse schließen will, ruft er entweder die Methode Close oder Dispose auf. Ruft er Close auf, wird er an Dispose (ohne Parameter) weitergeleitet. Diese Version von Dispose sorgt als erstes mit GC.SuppressFinalize(Me) dafür, dass der Garbage Collector, falls eine Garbage Collection ansteht, Finalize für dieses (Me) Objekt nicht mehr aufrufen wird – Dispose ist ja schließlich gerade dabei, die Finalisierung durchzuführen, und einmal reicht!

HINWEIS Aufgepasst dabei: GC.SuppressFinalize(Me) bedeutet nicht, dass der Garbage Collector das Objekt nicht mehr entsorgen wird – er wird während der Entsorgung lediglich nicht die Finalize-Methode des Objektes aufrufen; die Entsorgung findet immer statt; ein Objekt kann sich nicht dagegen wehren und die Entsorgung auch nicht verzögern oder die Reihenfolge in irgendeiner Form beeinflussen. Wenn es seine Lebensberechtigung verloren hat, ist es fällig, so oder so.

Dispose ruft nun seinerseits die Dispose-Überladung (mit Parameter) auf und übergibt ihr True als Argument und zum Zeichen, dass der Aufruf nicht durch den Finalize-Prozess bedingt war. Die überladene Dispose-Methode kann nun anhand der booleschen Variable feststellen, ob sie von Finalize oder manuell durch das Programm ins Leben gerufen wurde.

Falls das Programm der Grund für den Aufruf war, stellt Dispose sicher, dass das Objekt nicht schon entsorgt wurde – zu diesem Zweck gibt es die Member-Variable myDisposed, die quasi Buch darüber führt. Sollte dies jedoch der Fall gewesen sein, bringt Dispose die geforderte ObjectDisposedException-Ausnahme.

Anschließend erfolgen die eigentlichen Aufräumarbeiten. Der Speicherdatenstrom wird – sofern es sich um einen Schreibvorgang handelt – in eine Datei mit dem zuvor bestimmten Dateinamen geschrieben. Zu guter Letzt wird noch das Flag gesetzt, das das Objekt nunmehr als entsorgt kennzeichnet, damit ein zweites, versehentliches Dispose nicht mehr stattfinden kann. Und das ist auch wichtig, denn das MemoryStream-Objekt gibt es bei einem möglichen zweiten Dispose-Aufruf nicht mehr. Eine Ausnahme wäre die unvermeidliche Folge.

Sollte der Entwickler vergessen, Dispose oder Close aufzurufen, dann erfolgt kein GC.SuppressFinalize(Me) und der Garbage Collector ruft beim Entsorgen des Objektes die Finalize-Methode auf. Damit beim Finalisierungsendspurt keine Fehler zu einer Ausnahme führen, wird mit der Konstruktion Try/Finally dafür gesorgt, dass der Aufräumprozess soweit wie möglich gelingt, die Basismethode aber – ganz gleich ob eine Ausnahme auftrat oder nicht – noch in jedem Fall aufgerufen wird. Finalize des Objektes selbst ruft Dispose auf, jetzt aber mit Disposing = False. Für dieses Beispiel macht das keinen großen Unterschied; der Test auf ein bereits stattgefundenes Dispose findet lediglich nicht statt. Andere Objekte müssen aber möglicherweise diesen Unterschied kennen, um entsprechend auf die verschiedenen Auslöser (manuelles Dispose oder Finalize) reagieren zu können.

Sie können das Verhalten dieser Klasse übrigens sehr einfach nachvollziehen, indem Sie mit dem Programm experimentieren und das Ausgabefenster (nicht das Konsolenfenster) dabei beobachten. Es informiert Sie stets über die gerade durchgeführte Aufgabe, und durch das Verändern einiger Eigenschaften bzw. Aufrufe der SoapSerializer-Klasse im Hauptprogramm können Sie die unterschiedlichen Reaktionen des Finalisierungsprozesses testen.

Unterstützung durch den Visual Basic-Editor beim Einfügen eines Disposable-Patterns

Das Muster, auf das Sie beim Implementieren einer »disposebaren« Klasse achten müssen, ist nicht nur etwas komplexer als bei der Implementierung anderer Funktionen, die Sie durch die Einbindung einer oder mehrerer Schnittstellen implementieren müssen.

Sie müssen auch darauf achten, dass ein internes Muster, so wie Sie es im vorherigen Beispiel kennen gelernt haben, strikt einzuhalten ist – und das geht natürlich weit über das bloße Vorhandensein einer entsprechenden Dispose-Methode hinaus.

Aus diesem Grund gibt es eine besondere Editor-Unterstützung für die IDisposable-Schnittstelle. Sobald sie diese am Klassenkopf einfügen und nach Implements IDisposable  betätigen, fügt der Editor nicht nur den Funktionsrumpf, sondern ein etwas spezifischeres Funktionsgerüst als Code in Ihre Klasse ein, wie im Folgenden zu sehen:

```
Private disposedValue As Boolean = False      ' So ermitteln Sie überflüssige Aufrufe

' IDisposable
Protected Overrides Sub Dispose(ByVal disposing As Boolean)
    If Not Me.disposedValue Then
        If disposing Then
            ' TODO: Nicht verwaltete Ressourcen freigeben, wenn sie explizit aufgerufen werden
        End If
        ' TODO: Gemeinsam genutzte, nicht verwaltete Ressourcen freigeben
    End If
    Me.disposedValue = True
End Sub

#Region " IDisposable Support "
    ' Dieser Code wird von Visual Basic hinzugefügt, um das Dispose-Muster richtig zu implementieren.
    Public Sub Dispose() Implements IDisposable.Dispose
    ' Ändern Sie diesen Code nicht. Fügen Sie oben in Dispose(ByVal disposing As Boolean) Bereinigungscode ein.
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
#End Region
```

Diese Aktion bildet eine, wie ich finde, perfekte Unterstützung für die Implementierung von IDisposable. Sie müssen Modifizierungen nur unterhalb der mit 'TODO gekennzeichneten Codeteile vornehmen – ansonsten können Sie die Implementierung so belassen, wie sie ist.

Kapitel 19

Eigene Operatoren für benutzerdefinierte Typen

In diesem Kapitel:

Einführung in Operatorenprozeduren	572
Vorbereitung einer Struktur oder Klasse für Operatorenprozeduren	574
Implementierung von Operatoren	578
Implementierung von Vergleichsoperatoren	580
Implementierung von Typkonvertierungsoperatoren mit Operator CType	581
Implementieren von Wahr- und Falsch-Auswertungsoperatoren	583
Problembehandlungen bei Operatorenprozeduren	584
Übersicht der implementierbaren Operatoren	586

Es gibt ein vergleichsweise neues Wort, das es nicht einmal für nötig befunden hat, sich wenigstens als Anglizismus, sondern ganz unkaschiert als ursprünglicher englischer Wortstamm in die deutsche Sprache einzuschleichen. Die Rede ist von »Convenience«, zu Deutsch etwa: »Bequemlichkeit«. Einige wirkliche Kenner der Hotelbranche haben wegen des berühmt-berüchtigten Convenience Food inzwischen sogar eine psychosomatische Eierallergie entwickelt, und das in so heftigem Ausmaß, dass sie bei Auswärtsübernachtungen längst auf den morgendlichen gelben Glibber am Buffet verzichten müssen. Und das aus vielleicht gutem Grund, kommt doch die sich in den metallenen Warmhalteschalen befindliche Eierspeise nicht mehr aus verschiedenen Schalen und einer Pfanne, sondern einer luftdicht verschlossenen Tüte mit Trockenrührpulver (wer hätte gedacht, dass es so was überhaupt gibt?) – jedenfalls in vielen Fällen. Beim Convenience Food geht es also in erster Linie darum, dem *Zubereiter* das Leben so angenehm wie möglich zu machen, und weniger dem Gast, der das Wasser-/Rühreipulvergemänsche anschließend verdrückt.

Operatorenprozeduren, die mit Visual Basic 2005 eingeführt wurden, könnte man unter dem Begriff Convenience Tools (»Bequemlichkeitswerkzeuge«)¹ laufen lassen, obschon sie dem Entwickler das Leben zwar erleichtern, sich aber nicht negativ auf den »Consumer« auswirken. Sie stellen nichts bereit, was die Lösung eines bestimmten Entwicklerproblems an sich in irgendeiner Form vereinfachen würde, sie tragen lediglich dazu bei, dass sich Klassen oder Strukturen später einfacher anwenden lassen und Code besser lesbar wird.

Einführung in Operatorenprozeduren

Um was geht's genau?

Operatorenprozeduren dienen dazu, es eigenen Klassen zu gestatten, zusammen mit Operatoren verwendet werden zu können. Wenn Sie einen eigenen Typ geschaffen haben, ganz egal ob auf Basis einer Klasse oder einer Struktur, dann stellt dieser bestimmte Funktionalitäten über Methoden bereit. Und mithilfe von Operatorenprozeduren können Sie diese Methoden an Operatoren binden.

Ein Beispiel dafür könnte eine »Superstring«-Klasse sein. Eine Klasse, die zwar wie der primitive Datentyp String funktioniert, aber den Umgang mit verschiedenen Funktionen stark vereinfacht.

Einen »Rechen«-Operator können Sie bei Strings heute schon verwenden – das Pluszeichen, um zwei Strings aneinander zu hängen – damit ist aber natürlich kein Additions- sondern der in fast allen Programmiersprachen vorhandene String-Verkettungs-Operator gemeint. Wenn Sie also folgende Codezeilen haben

```
'Deklaration und Definition eines normalen Strings
Dim locNormaloString As String
locNormaloString = "Wenn man seinen Kopf gegen eine"
locNormaloString = locNormaloString + "eine Wand schlägt, verbraucht man 150 Kalorien."
Console.WriteLine("Ausgangszeichenfolge:")
Console.WriteLine(locNormaloString)
```

dann hat die Ausführung dieses Codes das nachstehende Ergebnis zur Folge:

```
Ausgangszeichenfolge:
Wenn man seinen Kopf gegen eine eine Wand schlägt, verbraucht man 150 Kalorien.
```

¹ Und es gibt tatsächlich den Begriff der Convenience-Patterns in der IT, gerade beim Überladen von Methoden.

Diese Idee kann man weiterspinnen. So wäre doch beispielsweise auch eine Multiplikation von Strings möglich. Aus

```
"Klaus" * 5
```

würde die Zeichenfolge

```
KlausKlausKlausKlausKlaus
```

entstehen. Und aus

```
"Klaus Löffelmanns Internetauftritt finden Sie unter http://loeffelmann.de" – "Löffelmanns"
```

würde

```
"Klaus Internetauftritt finden Sie unter http://loeffelmann.de"
```

Das Teilen eines Strings mit einem Trennzeichen könnte ein String-Array mit den verschiedenen Teilzeichenketten entstehen lassen. So würde der Ausdruck

```
"Verschiedene|Worte|sind|so|getrennt" : "|"c
```

ein String-Array mit den Elementen

```
Verschiedene  
Worte  
sind  
so  
getrennt
```

ergeben.

Und zu guter Letzt müssten auch implizite (direkte Zuweisungen) bzw. explizite (mit CType) Typumwandlungen in andere Typen möglich sein, sodass auch folgender Code möglich würde:

```
Dim locSuperString as SuperString = "Das hier ist eine String- und keine SuperString-Konstante"
```

bzw.

```
Dim locSuperString as SuperString = CType("das Ganze mit expliziter Zuweisung", SuperString)
```

Vorbereitung einer Struktur oder Klasse für Operatorenprozeduren

Das Wichtigste, was Sie über Operatorenprozeduren wissen müssen: Sie werden ausschließlich als statische Funktionen implementiert. Das liegt einfach daran, dass grundsätzlich zwei Argumente an eine Operatorenprozedur übergeben werden müssen. Die Codezeile

```
TypInstanz3 = TypInstanz1 + TypInstanz2
```

könnte man auch ohne Operatoren ermöglichen. Dann würde die Zeile etwa

```
TypInstanz3 = TypInstanz.Add(TypInstanz1, TypInstanz2)
```

lauten.

Daher müssen Sie sich bei den folgenden Ausführungen immer vor Augen halten, dass es bei der Implementierung von Operatorenprozeduren nur um *eine* (aber eben vielleicht eine besonders praktische) Art und Weise geht, eine bestimmte Aufgabe zu erfüllen. So kann man:

```
Dim testString = "Klaus"  
testString = testString + " zeigt, was man mit Operatoren machen kann."
```

Auch genauso schreiben

```
Dim testString="Klaus"  
testString = String.Concat(testString, " zeigt, was man mit Operatoren machen kann.")
```

Und müßte ergänzen:

```
testString=String.Concat(testString, " Und was auch ohne geht.")
```

Gehen wir davon aus, dass die Klasse oder Struktur, aus der sich TypInstanz1, TypInstanz2 und TypInstanz3 ableiten, TypInstanz lautet, wird klar, dass nur eine statische Implementierung Sinn ergibt. Denn Sie brauchen keine Instanz dieser Klasse oder Struktur, um zwei unabhängige Instanzen der Klasse bzw. Struktur zu addieren – lediglich den Code, der die Addition vornimmt und ein Funktionsergebnis vom Typ TypInstanz zurückliefert.

Das verhält sich ähnlich wie beispielsweise beim Parsen einer Zeichenfolge zur Umwandlung in eine numerische Variable. Auch hier verwenden Sie eine statische Funktion, nämlich Parse, für die Sie keine Strukturinstanz benötigen. Sie können Parse direkt mit der Zeile

```
Dim EinDouble as Double = Double.Parse("123,45")
```

verwenden. Sie müssen aber keine Variable vom Typ Double definieren, um auf die Parse-Funktion zuzugreifen. Da ein zusätzliches Set an statischen Funktionen notwendig wird, ergibt es Sinn, sich zunächst nur um die reine Funktionalitätsimplementierung in der entsprechenden Klasse oder Struktur zu kümmern – und die brauchen fürs Erste natürlich nicht statisch zu sein.

Eine Klasse SuperString könnte also mit komplett implementierter Funktionalität folgendermaßen aussehen:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 19\\Operatorenüberladung

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Public Structure SuperString

    Private myValue As String

    Public Sub New(ByVal Value As String)
        myValue = Value
    End Sub

    Public Overrides Function ToString() As String
        Return myValue
    End Function

    Public Function Addieren(ByVal andererString As SuperString) As SuperString
        Return New SuperString(myValue & andererString.ToString())
    End Function

    Public Function Subtrahieren(ByVal andererString As SuperString) As SuperString
        Return New SuperString(myValue.Replace(anererString.ToString, ""))
    End Function

    Public Function Subtrahieren(ByVal letztenZeichen As Integer) As SuperString
        Try
            Return New SuperString(myValue.Substring(0, myValue.Length - (letztenZeichen + 1)))
        Catch ex As Exception
            Return New SuperString(myValue)
        End Try
    End Function

    Public Function Vervielfachen(ByVal anzahl As Integer) As SuperString

        'Wir sind ordentlich und vermeiden Fehler! ;-)
        If myValue Is Nothing Then
            Return Nothing
        End If

        If myValue = "" Or anzahl < 1 Then
            Return New SuperString("")
        End If

        'Mit dem StringBuilder geht das am schnellsten!
        Dim locSB As New System.Text.StringBuilder

        'Einfach den Ausgangsstring sooft anhängen,
        'wie es 'anzahl' vorgibt.
```

```

For c As Integer = 1 To anzahl
    locSB.Append(myValue)
Next

'und zurück damit!
Return New SuperString(locSB.ToString)
End Function

Public Function Teilen(ByVal trennzeichen As Char) As SuperString()
    Dim locStringArray As String()
    locStringArray = myValue.Split(New Char() {trennzeichen})

    Dim locSuperStringArray(locStringArray.Length - 1) As SuperString
    For z As Integer = 0 To locStringArray.Length - 1
        locSuperStringArray(z) = New SuperString(locStringArray(z))
    Next
    Return locSuperStringArray

End Function
End Structure

```

Sie sehen, dass sich die Funktionen, die wir zur späteren Realisierung der Operatoren zunächst als normale Methoden implementieren, vergleichsweise einfach erstellen lassen – dieser Code bedarf nicht wirklich zusätzlicher Erklärungen.

Mit herkömmlicher Programmierung, also ohne Operatoren, sieht die Verwendung dieser Klasse dann so aus, wie es die Sub Main des Moduls des Projektes demonstriert:

```

Module Main

Sub Main()
    'Deklaration und Definition eines normalen Strings
    Dim locNormaloString As String
    locNormaloString = "Wenn man seinen Kopf gegen eine "
    locNormaloString = locNormaloString + "eine Wand schlägt, verbraucht man 150 Kalorien."
    Console.WriteLine("Ausgangszeichenfolge:")
    Console.WriteLine(locNormaloString)

    'Deklaration und Definition eines Super-Strings
    Dim locSuperString As New SuperString(locNormaloString)

    'SuperString-Funktionsdemo: Addieren (anhängen) anderer Strings
    Console.WriteLine()
    Console.WriteLine("'Addieren' von Strings - 'Toll, was?' anhängen:")
    locSuperString = locSuperString.Addieren(
        New SuperString(vbNewLine + " - Toll, was?"))
    Console.WriteLine(locSuperString.ToString())

    'Subtrahieren (rauslöschen) anderer Strings
    Console.WriteLine()
    Console.WriteLine("'Subtrahieren' von Strings - ', was?' abziehen:")
    locSuperString = locSuperString.Subtrahieren(
        New SuperString(", was"))
    Console.WriteLine(locSuperString.ToString())
    Console.WriteLine()

```

```
'Subtrahieren ist überladen - geht auch mit der Anzahl  
'der letzten Zeichen, die entfernt werden sollen.  
Console.WriteLine("Subtrahieren' von Strings - die letzten 9 Zeichen abziehen:")  
locSuperString = locSuperString.Subtrahieren(9)  
Console.WriteLine(locSuperString.ToString())  
Console.WriteLine()  
  
'Vervielfachen von Strings  
Console.WriteLine("Vervielfachen' von Strings:")  
locSuperString = locSuperString.Vervielfachen(4)  
Console.WriteLine(locSuperString.ToString())  
Console.WriteLine()  
  
'(Auf)teilen von Strings - schon etwas anspruchsvoller  
locSuperString = New SuperString("Der Glückskeks wurde 1916 von George Jung, " & _  
    "einem amerikanischen Nudelmacher erfunden.")  
Console.WriteLine("(Auf)teilen' von Strings:")  
Console.WriteLine(locSuperString.ToString())  
Dim locSuperStrings() As SuperString  
locSuperStrings = locSuperString.Teilen(" "c)  
For Each locString As SuperString In locSuperStrings  
    Console.WriteLine(locString.ToString())  
Next  
Console.WriteLine()  
Console.WriteLine("Taste drücken zum Beenden!")  
Console.ReadKey()  
End Sub
```

Wenn Sie dieses Beispiel laufen lassen, produziert es folgende Ausgabe im Konsolenfenster:

```
Ausgangszeichenfolge:  
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.  
  
'Addieren' von Strings - 'Toll, was?' anhängen:  
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.  
- Toll, was?  
  
'Subtrahieren' von Strings - ', was?' abziehen:  
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.  
- Toll?  
  
'Subtrahieren' von Strings - die letzten 9 Zeichen abziehen:  
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.  
  
'Vervielfachen' von Strings:  
Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.W  
enn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.We  
nn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.Wen  
n man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.  
  
'(Auf)teilen' von Strings:  
Der Glückskeks wurde 1916 von George Jung, einem amerikanischen Nudelmacher erfu  
nden.  
Der  
Glückskeks
```

```
wurde
1916
von
George
Jung,
einem
amerikanischen
Nudelmacher
erfunden.
```

Taste drücken zum Beenden!

Implementierung von Operatoren

Nachdem diese Vorbereitungen abgeschlossen sind, schreiten wir zur nächsten Tat: Dem eigentlichen Implementieren der statischen Operatorenprozeduren. Dazu können wir dann die aus der Mathematik bekannten Rechenoperatoren verwenden.

Hierbei unterscheiden wir zwischen zwei Typen: Den eigentlichen Operatoren, wie +, -, *, / etc. und den Operatoren, die zur Konvertierung von Typen dienen. Wir beginnen mit der ersten Gruppe – der Implementierung der von uns benutzten Rechenoperatoren.

Die generelle Ausführung der Operatorenimplementierung lautet folgendermaßen:

```
Public [Class|Structure] OpTyp
    Public Shared Operator OpChar(ByVal objVar1 As [OpTyp|Typ1], ByVal objVar2 As [OpTyp|Typ2]) As Typ3
        ' Hier steht der Code, der die eigentliche Operation durchführt
    End Operator
End [Class|Structure]
```

Dieser Rumpf soll verdeutlichen, auf was es ankommt:

- Operatoren lassen sich auf Klassen *und* Strukturen anwenden.
- Welcher Operator (+, -, *, / etc.) zur Anwendung kommt, wird durch OpChar bestimmt. Tabelle 19.1 listet auf, welche Rechenoperatoren sich implementieren lassen (und wofür sie eigentlich gedacht sind).
- Mindestens einer der Parameter, den Sie einer Operatorenprozedur übergeben, muss vom Typ sein wie die Klasse bzw. Struktur, die die Operatorenprozedur definiert.
- Die Operatorenprozedur muss, wie schon erwähnt, statischer Natur und deswegen mit dem Modifizierer Shared definiert sein.
- Der Rückgabetypr kann beliebig sein.

Auf unser Beispiel angewendet, würde die Additionsroutine sich dann folgendermaßen gestalten:

```
Public Shared Operator +(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As SuperString
    Return sstring1.Addieren(sstring2)
End Operator
```

Durch den Plus-Operator sollen die beiden Parameter, die jeweils links und rechts vom Plus-Operator stehen, »addiert«, also in unserem Fall miteinander verkettet werden. Der links vom Operator stehende Parameter entspricht dabei dem ersten der Operatorenprozedur übergebenen Parameter, der rechts vom Operator stehende dem zweiten Parameter.

HINWEIS Diese Art der Implementierung funktioniert problemlos, da wir unseren SuperString-Typ auf Basis einer Struktur, also eines Wertetyps, konzipiert haben. Hätten wir ihn als Referenztyp konzipiert, wären bei der Implementierung der Operatorenprozeduren auf diese Weise Probleme buchstäblich vorprogrammiert. Lesen Sie vor der Implementierung von Operatorenprozeduren in eigene Klassen deswegen auch unbedingt den Abschnitt »Implementieren von Wahr- und Falsch-Auswertungsoperatoren« ab Seite 583.

Sobald diese Operatorenprozedur Bestandteil der Klasse geworden ist, können wir den Code im Modul für die Addition der beiden Strings umstellen. Aus

```
locSuperString = locSuperString.Addieren(  
    New SuperString(vbNewLine + " - Toll, was?"))
```

wird dann

```
locSuperString = locSuperString + New SuperString(vbNewLine + " - Toll, was?")
```

Und das ist schon wesentlich bequemer zu handhaben und besser lesbar, finden Sie nicht?

In diesem Stil haben wir dann die Möglichkeit, auch die anderen Operatoren zu implementieren:

```
Public Shared Operator -(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As  
    SuperString  
    Return sstring1.Subtrahieren(sstring2)  
End Operator  
  
Public Shared Operator *(ByVal sstring1 As SuperString, ByVal anzahl As Integer) As SuperString  
    Return sstring1.Vervielfachen(anzahl)  
End Operator  
  
Public Shared Operator /(ByVal sstring1 As SuperString, ByVal trennzeichen As Char) As SuperString()  
    Return sstring1.Teilen(trennzeichen)  
End Operator
```

Überladen von Operatorenprozeduren

Nun gibt es noch einen Punkt, der bei Operatorenprozeduren noch angepasst werden sollte: Unsere ursprüngliche Subtraktionsroutine gibt es in zwei Überladungsversionen. Die erste übernimmt einen SuperString als Parameter; dieser wird dann in der Ausgangszeichenkette gesucht und aus ihr entfernt, wenn es eine Suchübereinstimmung gab. Die zweite Möglichkeit: Sie können einen Integer-Wert als Parameter angeben, der die Anzahl der Zeichen bestimmt, die vom hinteren Teil der Zeichenkette entfernt werden sollen. Diese Funktion ist im Moment noch nicht durch einen Operator aufrufbar.

Doch wie »normale« Methoden können Sie auch für Operatorenprozeduren Überladungen anwenden. Es gilt dabei das in Kapitel 14 im Abschnitt »Überladen von Methoden, Konstruktoren und Eigenschaften« Gesagte. Wenn wir die Sammlung der Operatorenprozeduren um die folgende überladene Methode ergänzen,

```
Public Shared Operator -(ByVal sstring1 As SuperString, ByVal anzahl As Integer) As SuperString
    Return sstring1.Subtrahieren(anzahl)
End Operator
```

wird es möglich, beide Versionen der Subtraktion im Modul auf Operatoren umzustellen:

```
.
.
.

'Subtrahieren (rauslöschen) anderer Strings
Console.WriteLine()
Console.WriteLine("Subtrahieren' von Strings - ', was?' abziehen:")
'locSuperString = locSuperString.Subtrahieren(
    '    New SuperString(", was"))
locSuperString = locSuperString - New SuperString(", was")
Console.WriteLine(locSuperString.ToString())
Console.WriteLine()

'Subtrahieren ist überladen - geht auch mit der Anzahl
'der letzten Zeichen, die entfernt werden sollen.
Console.WriteLine("Subtrahieren' von Strings - die letzten 9 Zeichen abziehen:")
locSuperString = locSuperString - 9
Console.WriteLine(locSuperString.ToString())
Console.WriteLine()

.
```

Implementierung von Vergleichsoperatoren

Prinzipiell lassen sich Vergleichsoperatoren für benutzerdefinierte Typen ähnlich implementieren wie Rechenoperatoren. Es gibt nur zwei zusätzliche Bedingungen:

- Sie müssen als Funktionsergebnis grundsätzlich einen booleschen Datentyp zurückliefern, der bestimmt, ob der Vergleich erfolgreich war oder nicht.
- Sie müssen Vergleichsoperatoren paarweise implementieren. Wenn Sie den Operator implementieren, der auf Gleichheit prüft, müssen Sie auch den implementieren, der auf Ungleichheit prüft. Implementieren Sie den Vergleich auf größer, müssen Sie den auf kleiner ebenfalls einbauen. Das Gleiche gilt für größer gleich und kleiner gleich.

Die Implementierung von Vergleichsoperatoren für unsere SuperString-Klasse sieht folgendermaßen aus:

```
Public Shared Operator <>(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString <> sString2.ToString)
End Operator

Public Shared Operator =(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString = sString2.ToString)
End Operator
```

```

Public Shared Operator <(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString < sString2.ToString)
End Operator

Public Shared Operator >(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString > sString2.ToString)
End Operator

```

Implementierung von Typkonvertierungsoperatoren mit Operator CType

Typkonvertierungsoperatoren sind ein zweischneidiges Schwert. Sie erhöhen den »Degree of Convenience« auf der einen Seite um ein weiteres Maß, können aber unter Umständen auch zu erheblichen Problemen führen – aber dazu später mehr.

Erinnern wir uns. Eine implizite Konvertierung, also eine, bei der nichts Zusätzliches gemacht werden muss, können Sie anwenden, wenn Sie einen kleineren Datentyp in einen größeren Datentyp überführen – beispielsweise einen Integer-Wert in einen Long-Wert.

Beispiel:

```

'Hier geht's mit impliziter (da typerweiternder) Konvertierung
Dim einLong As Long, einInteger As Integer
einInteger = 10
einLong = einInteger

```

Vor den umgekehrten Weg einer typverkleinernden Typkonvertierung schiebt der Basic Compiler jedoch erstmal einen Riegel – jedenfalls so lange, bis Sie sich mit CType (oder einem Cxxx-Operator) da »rauskauen«. Sie sollen sich bewusst sein, dass Daten bei einer typverkleinernden (oder den Typ völlig verändernden) Konvertierung verloren gehen können, und deswegen macht Sie Visual Basic mit dem Einsatzzwang von CType darauf aufmerksam. Dieserart *explizite* Konvertierungen sehen dann so

```

'Das hier erfordert eine explizite, da typverkleinernde Konvertierung
einLong = 1000
einInteger = CType(einLong, Integer)

```

oder so aus:

```

'Aber auch diese Konvertierung muss explizit durchgeführt werden
Dim einDouble As Double, einString As String
einString = "1.828.488.382,45"
einDouble = CType(einString, Long)

```

Nun werden Typen natürlich nicht »einfach so« konvertiert – gerade bei einer komplexeren Typkonvertierung wie von einer Zeichenkette in einen numerischen Wert läuft ein gar nicht so anspruchloses Programm ab, das diese Konvertierung vornimmt.

Und ein solches Programm – oder besser: eine solche Unterroutine – können Sie mit den so genannten Operator CType-Prozeduren für Ihre eigenen Datentypen implementieren. Dabei haben Sie es in der Hand, ob eine implizite oder eine explizite Konvertierung erfolgen soll.

Für unser Beispiel wäre es doch schön, wenn die folgende Zeile funktionieren würde.

```
Dim einSuperString As SuperString = "Dies ist eine Zeichenkette"
```

Das können Sie haben. Bei der Konstanten, die dem SuperString zugewiesen wird, handelt es sich um eine vom Typ String. Da wir an dieser Stelle ohne CType arbeiten wollen (und können, denn es droht bei der Konvertierung kein Verlust), müssen wir eine Konvertierungsoperatorenprozedur implementieren, die den Datentyp erweitert. Und das geht so (und jetzt halten Sie sich fest, was die Modifizierer der folgenden Prozedur betrifft):

```
Public Shared Widening Operator CType(ByVal normaloString As String) As SuperString
    Return New SuperString(normaloString)
End Operator
```

Operator CType zeigt hier an, dass die Routine für eine Typkonvertierung zuständig ist. Der Modifizierer Widening bestimmt, dass es sich um eine *implizite* Konvertierung handelt, bei der die Ausformulierung von CType nicht notwendig ist. Und Shared schließlich, aber das wissen Sie ja, bestimmt, dass die Routine statischer Natur ist. Der eigentliche Konvertierungscode ist simpel: Die Prozedur legt auf Basis des übergebenen String eine neue SuperString-Instanz an und liefert diese als Funktionsergebnis zurück.

Würden Sie wollen, dass der Entwickler, der Ihre Klasse verwendet, eine explizite Typkonvertierung mit CType einleiten muss, dann würden Sie den Modifizierer Widening durch Narrowing (verkleinern) ersetzen.

Natürlich ist diese »Erweitern-/Verkleinern-Geschichte« bei Datentypen nur eine Richtlinie. Ihnen steht es natürlich frei, jeden Datentyp implizit oder explizit konvertierbar zu machen, ganz gleich, ob dabei Daten auf der Strecke bleiben können (wie bei Long zu Integer) oder nicht. Mir persönlich stoßen die Modifizierernamen ein wenig sauer auf, da sie mehr verwirren als nützen. *Implicit* oder *Explicit* als Modifizierernamen wären mir lieber gewesen – aber wahrscheinlich hätte das wieder zu Problemen geführt, weil *Explicit* im Visual Basic-Dialekt schon seit Jahren im Rahmen von Option Explicit eingesetzt wird. Doch das ist eine bloße Vermutung.

Übrigens: Die Konvertierung, die Sie nun implementiert haben, funktioniert derzeit nur in eine Richtung. Würden Sie versuchen, den umgekehrten Weg mit

```
Dim einString as String = einSuperString
```

zu gehen, sähen Sie in der Fehlerliste die Meldung:

```
Der Wert vom Typ "SuperStringVorstellung.SuperString" kann nicht zu "String" konvertiert werden.
```

Damit beide Richtungen funktionierten, müssten Sie eine weitere CType Operator-Prozedur zum Projekt hinzufügen, nämlich:

```
Public Shared Widening Operator CType(ByVal SuperString As SuperString) As String
    Return SuperString.ToString()
End Operator
```

Implementieren von Wahr- und Falsch-Auswertungsoperatoren

Eine Möglichkeit, Wahr- und Falsch-Auswertungsmechanismen zu implementieren, wäre, implizite oder explizite Konvertierungen in den Datentyp Boolean für Ihre Klasse anzubieten.

Doch Visual Basic sieht für diesen Zweck eine weitere Möglichkeit vor – die Operatoren `IsTrue` und `IsFalse`. Diese Operatoren stellen keine Anwendungsmöglichkeit im herkömmlichen Sinne bereit – Sie können die Operatoren `IsTrue` und `IsFalse` also nicht als Namen wie `CType` in ihren eigenen Programmen einsetzen.

Sie dienen vielmehr nur als Hilfen bei der Definition von Operatorenprozeduren, um eine bestimmte Prozedur für eine Operation festzulegen, die, ähnlich der impliziten Datentypkonvertierung, eigentlich gar keine Operatoren benötigt.

Unsere `SuperString`-Klasse könnte beispielsweise einen Mechanismus bereitstellen, der definiert, dass bestimmte Zeichenketteninhalte bei Auswertungen `True` ergeben, alle anderen `False` zum Ergebnis haben. In diesem Fall ließe sich folgende Vorgehensweise implementieren:

```
Sub ExperimenteFürBoolescheAusdrücke()
    'Hier geht's mit impliziter (da typerweiternder) Konvertierung
    Console.WriteLine("Möchten Sie weitere Daten eingeben (Ja, Nein):")
    Dim locSupStr As SuperString = Console.ReadLine
    If locSupStr Then
        Console.WriteLine("OK, dann geben Sie mal ein!")
    Else
        Console.WriteLine("dann halt nicht...")
    End If
    Console.WriteLine()
    Console.WriteLine("Taste drücken zum Beenden!")
End Sub
```

Die entsprechenden Operatorenprozeduren, die diese Vorgehensweise ermöglichen, könnte man beispielsweise folgendermaßen aufbauen:

```
Public Shared Operator IsTrue(ByVal sString As SuperString) As Boolean
    Dim locString As String = sString
    locString = locString.ToUpper
    Select Case locString
        Case "JA"
            Return True
        Case "J"
            Return True
        Case "RICHTIG"
            Return True
        Case "WAHR"
            Return True
        Case "AUSGEWÄHLT"
            Return True
        Case "GEDRÜCKT"
            Return True
        Case "BESTÄTIGT"
            Return True
    End Select
End Operator
```

```

Case "Y"
    Return True
Case "YES"
    Return True
Case "TRUE"
    Return True
Case "CORRECT"
    Return True
Case "SELECTED"
    Return True
Case "PRESSED"
    Return True
Case "ACCEPTED"
    Return True
Case "CONFIRMED"
    Return True
End Select
Return False
End Operator

Public Shared Operator IsFalse(ByVal sString As SuperString) As Boolean
    If sString Then
        Return False
    Else
        Return True
    End If
End Operator

```

HINWEIS Wie bei Vergleichsoperatoren müssen Sie die Operatoren `IsTrue` und `IsFalse` paarweise einbinden. Auch wichtig: Sie sollten es bei der Einbindung von `IsTrue` und `IsFalse` in Erwägung ziehen, auch den `Not`-Operator zu verdrahten. Andernfalls kann der Entwickler, der Ihre Klasse anwendet, einen Ausdruck wie folgenden nicht anwenden:

```

If Not locSupStr Then
    .
    .
    .
End If

```

Problembehandlungen bei Operatorenprozeduren

Operatorenprozeduren können, richtig eingesetzt, eine enorme Erleichterung für den Entwickler darstellen. Gleichzeitig sollten Sie aber auch einige Dinge beherzigen, bei denen Operatorenprozeduren dafür verantwortlich sein können, dass sich durch ihre Implementierung Fehler einschleichen.

Aufgepasst bei der Verwendung von Referenztypen

In unserem Beispiel haben wir die Operatorenprozeduren in eine Struktur eingebaut. Damit handelt es sich automatisch um einen Wertetyp, den auch die Operatorenprozeduren verarbeiten. Nun schauen Sie sich eine der Rechenoperatorenprozeduren noch einmal genauer an:

```
Public Shared Operator +(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As SuperString
    Return sstring1.Addieren(sstring2)
End Operator
```

Die Addieren-Funktion wird hier auf `sstring1` angewendet und `Addieren` liefert – wenn Sie sich den entsprechenden Code anschauen – ohnehin eine neue Instanz von `SuperString` zurück. Aber das muss nicht so sein. Manche Implementierungen »neigen dazu«, das Objekt selbst zu verändern. Würde `Addieren` das machen – wäre `Addieren` also eine Methode, die kein Funktionsergebnis lieferte – würde der entsprechende Code der Operatorenprozedur vielleicht so aussehen:

```
Public Shared Operator +(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As SuperString
    sstring1.Addieren(sstring2)
    Return sstring1
End Operator
```

Solange wie es sich bei den Werten, die `sstring1.Addieren` manipuliert, um Wertetypen handelt und solange `sstring1` selbst ein Wertetyp ist, hat eine solche Prozedur immer noch nichts Gefährliches an sich.

Schlimm kann es nur dann werden, wenn `Addieren` einen Referenztypen manipuliert und es sich bei `sstring1` selbst ebenfalls um einen Referenztypen handeln würde.

In diesem Fall würden Sie nämlich als Parameter in `sstring1` im Grunde genommen einen Zeiger auf die eigentlichen Objektdaten entgegen nehmen. Das anschließende `Addieren` würde also keine Kopie von `sstring1` verändern sondern die einzige existierende Instanz – die ursprünglich im aufrufenden Code vor dem Operator gestanden hätte. Sie würden damit in der Objektvariablen, die Sie als Operanden übergeben, und in der Objektvariablen, der Sie den Ausdruck zuweisen, das gleiche Ergebnis vorfinden. Zur Verdeutlichung (und bei dem folgenden Code nehmen wir an, dass es sich beim Typ, für den Operatoren definiert sind, um einen Referenztyp handelt):

```
Dim objVar1 As New RefType(10)
Dim objVar2 As New RefType(20)
Dim objVar3 As RefType
objVar3 = objVar1 + objVar2
```

Vorausgesetzt, `RefType` würde in der Lage sein, Zahlen zu addieren. Dann würde das Ergebnis 30 – das eben vorgestellte Szenario angenommen – nach Abschluss nicht nur in `objVar3` sondern auch in `objVar1` »stehen«. Und im Grunde genommen ist es sogar noch schlimmer: Da `objVar3` und `objVar1` auf die gleichen Daten zeigen, würde eine Veränderung von `objVar1` immer auch eine Veränderung von `objVar3` nach sich ziehen.

Gerade bei Operatorenprozeduren, bei denen Sie ein solches Verhalten am wenigsten erwarten, sollten Sie darauf achten, dass diese als Rückgabewert immer eine neue Instanz ihres Typs zurückgeben, wenn sie Referenztypen verarbeiten.

Aufgepasst bei Mehrdeutigkeiten bei der Auflösung von Signaturen

In unserer Beispielklasse ist, nachdem wir die Typkonvertierungsfunktionen vollständig implementiert haben, Folgendes erlaubt:

```
einSuperString = einSuperString + "Klaus"
```

Auf den ersten Blick mag das merkwürdig erscheinen, denn für die Addition haben wir keine Überladungsversion implementiert, die einen »einfachen« String akzeptieren würde. Dennoch meldet der Visual Basic Compiler keinen Fehler. Und das zu Recht, denn:

Die einzige Operatorenprozedur, die das Addieren mit dem +-Operator erlaubt, nimmt als zweiten Parameter eine Variable vom Typ SuperString entgegen. Der wiederum verfügt aber über einen impliziten Typkonvertierungsmechanismus, der einen normalen String in einen SuperString umwandeln kann. Der Compiler macht also das einzig Richtige: Er sorgt dafür, dass »Klaus« implizit zunächst in einen SuperString konvertiert wird, und dieser SuperString wird anschließend der Additions-Operatorenprozedur übergeben.

Was passiert aber, wenn es sowohl eine implizite Konvertierung von String in SuperString als auch eine überladene Version des Additionsoperators gibt, die Strings akzeptiert? In diesem Fall wird diese Additionsoperationsprozedur verwendet.

Fehler können sich aber – Sie ahnen es schon – dann einschleichen, wenn beide Routinen auf unterschiedliche Weise vorgehen, um den String in einen SuperString zu konvertieren. Die anschließende Fehlersuche bei Fehlern in komplexen Typen kann sich dann unter Umständen als sehr mühselig entpuppen.

TIPP Es empfiehlt sich also in jedem Fall, die möglichen Parameterkombinationen durchzuprobieren, herauszufinden, wo welche Konvertierung wirklich stattfindet, und alle Operatorenprozeduren dazu, am besten Schritt für Schritt, mit dem Visual Studio Debugger zu durchlaufen.

Übersicht der implementierbaren Operatoren

Sie werden staunen, welche Operatoren sich mit Operatorenprozeduren implementieren lassen. Die folgende Tabelle gibt Ihnen die Übersicht.

WICHTIG Achten Sie darauf, dass Sie einige Funktionen nur paarweise implementieren können.

Die Spalte *Vorgesehene Funktion* soll Ihnen übrigens nur eine grobe Themenrichtung für eine Implementierung vorgeben. Aber letzten Endes könnte Sie natürlich keiner daran hindern, dass Ihre Datentypen mit – addiert und mit + subtrahiert werden, wenn Sie es so wollen.

Operator	Vorgesehene math. Funktion	Bemerkung
+	Addition	
-	Subtraktion	
\	Division ohne Rest	
/	Division	
^	Potenzieren	
&	Verknüpfung	
<<	Nach links verschieben	Normalerweise bitweise bei Integerzahlen
>>	Nach rechts verschieben	Normalerweise bitweise bei Integerzahlen
=	Test auf Gleichheit	Hiermit steuern Sie nicht die Zuweisung an eine Objektvariable – dies ist nur mit dem CType-Operator möglich. Wenn Sie diesen Operator implementieren, müssen Sie < > (ungleich) ebenfalls implementieren.
< >	Test auf Ungleichheit	Wenn Sie diesen Operator implementieren, müssen Sie = (gleich) ebenfalls implementieren.
<	Test auf kleiner	Wenn Sie diesen Operator implementieren, müssen Sie > (größer) ebenfalls implementieren.
>	Test auf größer	Wenn Sie diesen Operator implementieren, müssen Sie < (kleiner) ebenfalls implementieren.
<=	Test auf kleiner oder gleich	Wenn Sie diesen Operator implementieren, müssen Sie > = (größer gleich) ebenfalls implementieren.
>=	Test auf größer oder gleich	Wenn Sie diesen Operator implementieren, müssen Sie < = (kleiner gleich) ebenfalls implementieren.
Like	Test auf Ähnlichkeit	
Mod	Restwertermittlung	
And	Logische Und-Verknüpfung	
Or	Logische Oder-Verknüpfung	
Xor	Logische Exklusiv-Oder-Verknüpfung	
Not	Logische Negation	
CType	Steuerung der impliziten oder expliziten Typkonvertierung	Verwenden Sie Narrowing für die explizite und Widening für die implizite Typkonvertierung.
IsTrue	Auswerten von booleschen Ausdrücken	Sie müssen IsTrue und IsFalse paarweise implementieren. Sie sollten in diesem Fall auch einen Not-Operator zur Verfügung stellen. WICHTIG: Diese Operatoren können nur in Auswertungskonstruktionen wie If/Then/Else, Do While, etc. verwendet werden – sie stellen <i>keine</i> implizite Konvertierung in den Datentyp Boolean bereit!
IsFalse	Auswerten von booleschen Ausdrücken	Es gilt das zu IsTrue gesagte. Mehr Informationen zu diesem Thema finden Sie im Abschnitt »Implementieren von Wahr- und Falsch-Auswertungsoperatoren« ab Seite 583.

Tabelle 19.1 Operatoren in Visual Basic 2008, die für die Implementierung in eigenen Typen zur Verfügung stehen

Kapitel 20

Ereignisse und Ereignishandler

In diesem Kapitel:

Konsumieren von Ereignissen mit WithEvents und Handles	591
Auslösen von Ereignissen	594
Zur-Verfügung-Stellen von Ereignisparametern	596
Dynamisches Anbinden von Ereignissen mit AddHandler	600

Ereignisse kennen Sie wahrscheinlich in erster Linie aus der Windows-Programmierung, und zwar wenn es darum geht, auf bestimmte Benutzerereignisse wie das Klicken von Schaltflächen oder das Auswählen von Elementen in einer Liste zu reagieren. Doch das reine Konsumieren dieser vordefinierten Ereignisse stellt dabei nur die Spitze des Eisberges dar.

Klassen können auch eigene Ereignisse auslösen, die dann wiederum von anderen Klassen benutzt werden. Dabei sollten sich Ereignis auslösende Klassen an bestimmte Regeln halten, von denen später noch die Rede sein wird. Und: Ereignisse können auch nachträglich, also zur Laufzeit, mit entsprechenden Behandlungs-routinen verknüpft und damit nur im Bedarfsfall konsumiert werden. Damit werden auch Szenarien mög-lich, die noch in Visual Basic 6.0 ohne die Hilfe von externen DLLs nicht möglich gewesen wären.

Dieses Kapitel beschäftigt sich mit Ereignissen und stellt diese Technologien nicht nur vor, sondern es zeigt auch, wie Sie für Ihre eigenen Komponenten Ereignisse erstellen, die diese zur Laufzeit auslösen und dann von anderen Komponenten konsumiert werden können.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

... \VB 2008 Entwicklerbuch\Ch - OOP\Kapitel 20\Alarm01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Beispielprogramm, das in der ersten Version einen einfachen Wecker imitiert, werden wir im Folgen-den für die Ereigniserkundigungen verwenden. Wenn Sie dieses Beispielprojekt starten, sehen Sie ein Formular, wie Sie es auch in Abbildung 20.1 erkennen können.

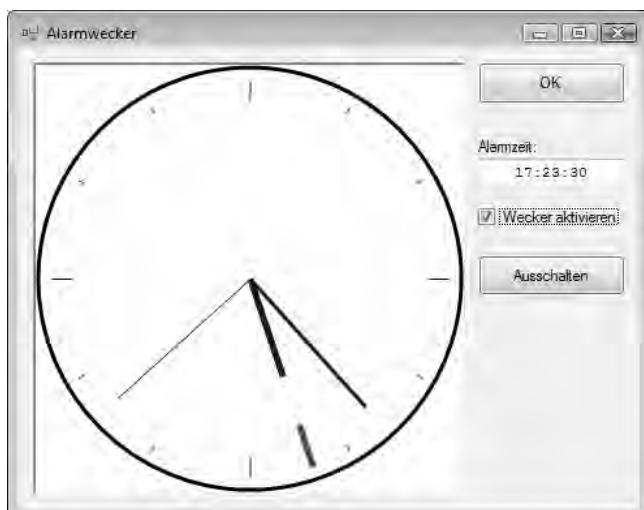


Abbildung 20.1 Die Alarmwecker-Anwendung soll Ihnen anschauliche Beispiele für den Umgang mit Ereignissen geben

Ein paar Worte zur groben Funktionsweise des Programms:

Sie haben die Möglichkeit, im Feld *Alarmzeit* eine Uhrzeit im Format *HH:mm:ss*¹ anzugeben. Klicken Sie anschließend auf *Wecker aktivieren*, zeichnet das Programm eine kleine rote Markierung in das Ziffernblatt der Uhr ein, die die Alarmzeit markiert.

Ist diese Zeit erreicht, geht der Wecker los – in diesem Beispiel durch ein rotes Blinken des gesamten Zeichenbereichs der Uhr.

Wie die Uhr tatsächlich in das PictureBox-Steuerelement gezeichnet wird, ist in diesem Kapitel von untergeordnetem Interesse. Die Grundlagen über den Umgang mit Zeichenfunktionen des GDI+ lesen Sie in Kapitel 39.

Wichtig in diesem Zusammenhang ist eine Klasse, die das »Wecken« übernimmt. Diese Klasse funktioniert im Prinzip wie ein handelsüblicher Alarmwecker im wirklichen Leben. Mit einer bestimmten Eigenschaft stellen Sie die Weckzeit ein, und wenn diese Uhrzeit erreicht ist, löst die Klasse ein Ereignis aus.

Im Grunde genommen gibt es zwei Möglichkeiten, Ereignisse zu konsumieren:

- Durch das Zuweisen einer beliebigen Prozedur mit einer Member-Variable einer Klasse, die mit *WithEvents* deklariert wurde.
- Durch das manuelle Hinzufügen einer Ereignisbehandlungs-Prozedur zur Laufzeit mit *AddHandler*.

Eine schnellere Möglichkeit als das Konsumieren von Ereignissen gibt es, wenn in der Ableitung einer Klasse Ereignisauslöser mit *Onxxx*-Methoden implementiert werden. In diesem Fall ergibt es mehr Sinn, die Ereignis auslösenden Routinen zu überschreiben und die Ereignisbehandlung auf diese Weise zu implementieren. Bei Windows Forms-Anwendungen sollte das der bevorzugte Weg sein, um Formularereignisse zu verdrahten.

HINWEIS Wie Sie mithilfe von Editor bzw. Designer Rümpfe von Ereignisbehandlungsroutinen in Ihren Code einfügen, hat Kapitel 5 (Abschnitt »Das Eigenschaftenfenster«) bereits beschrieben. Dieses Kapitel geht den nächsten Schritt und zeigt, wie die Ereignisbehandlung im Detail funktioniert.

Konsumieren von Ereignissen mit WithEvents und Handles

Wenn Sie in ein leeres Formular eine Schaltfläche einfügen und auf diese Schaltfläche doppelklicken, sorgt die Visual Studio-IDE dafür, dass der Editor geöffnet, der Formularcode angezeigt und ein Funktionsrumpf in diesem eingefügt wird, der später dann aufgerufen wird, wenn der Anwender zur Laufzeit die Schaltfläche betätigt.

¹ Kleiner Tipp am Rande: Die großgeschriebenen »H«s signalisieren, dass es sich um eine Darstellung im 24-Stunden-Format handelt – kleine »h«s würden das 12-Stunden-Format signalisieren. »H« ist dabei die Abkürzung des englischen »Hours« (Stunden), »m« für »Minutes« (Minuten) und »s« für »Seconds« (Sekunden). Für ein Datum wählt man die Formatabkürzungen »y« für »Years« (Jahre), »M« für »Months« (Monate) – großgeschrieben übrigens, um sie von den Minuten zu unterscheiden – sowie »d« für »Days« (Tage). Ein deutsches Datumsformat entspräche demnach *dd.MM.yyyy*, ein typisch amerikanisches *MM/dd/yyyy* (der Monat wird hier zuerst genannt!).

Damit die Behandlung solcher Ereignisse möglich wird, bedarf es dreier Komponenten: Einerseits muss die Objektvariable, die das Ereignis anbietet, mit WithEvents deklariert worden sein. Andererseits muss die Signatur² der Prozedur, die das Ereignis verarbeiten soll, mit der Signatur des Ereignisses übereinstimmen, die das Objekt anbietet. Und zu guter Letzt muss die Prozedur mit dem Schlüsselwort Handles mit dem Objektereignis (oder auch mit anderen Ereignissen dieses Objektes bzw. mit den Ereignissen anderer Objekte, falls deren Signaturen dieselben sind) verdrahtet werden.

Die Routinen, die im Code des Beispiels diese Art von Ereignisbehandlung durchführen, finden Sie im Folgenden:

```
'Ereignisbehandlungsroutine, die ausgelöst wird,
'wenn der Alarmgeber Alarm signalisiert, weil eine
'bestimmte Uhrzeit erreicht wurde.
Private Sub myAlarmgeber_Alarm(ByVal Sender As Object, ByVal e As AlarmEventArgs) _
    Handles myAlarmgeber.Alarm
    'Wecker schellt!
    myAlarmStatus = True
    'Und zwar so lange.
    myAlarmDownCounter = myAlarmDauer
    'Abschalten sollten wir das Schellen können.
    btnAusschalten.Enabled = True
    'Und morgen um die gleiche Zeit soll er wieder schellen.
    e.Neustellen = True
End Sub

'Ereignisbehandlungsroutine, die ausgelöst wird,
'wenn der Inhalt der Picturebox neu gezeichnet werden soll.
Private Sub picWecker_Paint(ByVal sender As Object, ByVal e As System.Windows.Forms.PaintEventArgs)

    Handles picWecker.Paint
    If myAlarmgeber IsNot Nothing AndAlso myAlarmgeber.AlarmAktiviert Then
        DrawClock(e.Graphics, Date.Now, myAlarmgeber.Alarmzeit, myAktuelleFarbe)
    Else
        DrawClock(e.Graphics, Date.Now, myAktuelleFarbe)
    End If
End Sub

.

.

.

'Ereignisbehandlungsroutine, die ausgelöst wird,
'wenn der Timer abgelaufen ist. Dies passiert
'alle 500 Millisekunden, und wir zeichnen dann
'die komplette Uhr neu - berücksichtigen dabei
' auch das Hintergrundblinken, falls der "Wecker
' gerade schellt".
Private Sub myTimer_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles myTimer.Tick
    'Läuft der Alarm gerade?
    If myAlarmStatus Then
        'Ja, Farben alle 500 ms wechseln
```

² Zur Erinnerung: Die Signatur einer Prozedur ergibt sich aus der Abfolge der Typen, die eine Prozedur als Parameter entgegennimmt.

```

If myAktuelleFarbe = Color.White Then
    myAktuelleFarbe = Color.Red
Else
    myAktuelleFarbe = Color.White
End If
'Alarmsdauerzähler vermindern
myAlarmDownCounter -= 1
If myAlarmDownCounter = 0 Then
    'Alarm ausschalten, wenn dieser abgelaufen ist.
    AlarmAusschalten(True)
End If
End If

'Ganze Uhr in jedem Fall neuzeichnen.
picWecker.Invalidate()
End Sub

```

In diesem Fall sind es drei Objekte, die im Gültigkeitsbereich der Klasse deklariert wurden und damit Member-Variablen sind, deren Ereignisse von diesen Prozeduren behandelt werden:

- myTimer,
- picWecker sowie
- myAlarmgeber

Und mit der Ausnahme der Objektvariablen picWecker, bei der es sich um eine Objektvariable handelt, die ein Steuerelement repräsentiert und die mithilfe des Designers ihren Weg ins Formular gefunden hat, sind dann auch alle Ereignis anbietenden Variablen mit dem WithEvents-Schlüsselwort deklariert worden.

```

Public Class frmMain

Private WithEvents myTimer As Timer
Private WithEvents myAlarmgeber As EinfacherAlarmgeber

```

Erst dann bietet IntelliSense entsprechende Ereignisse zum Objekt an, wenn die Verdrahtung einer Prozedur mit Handles erfolgen soll, wie Abbildung 20.2 zeigt.

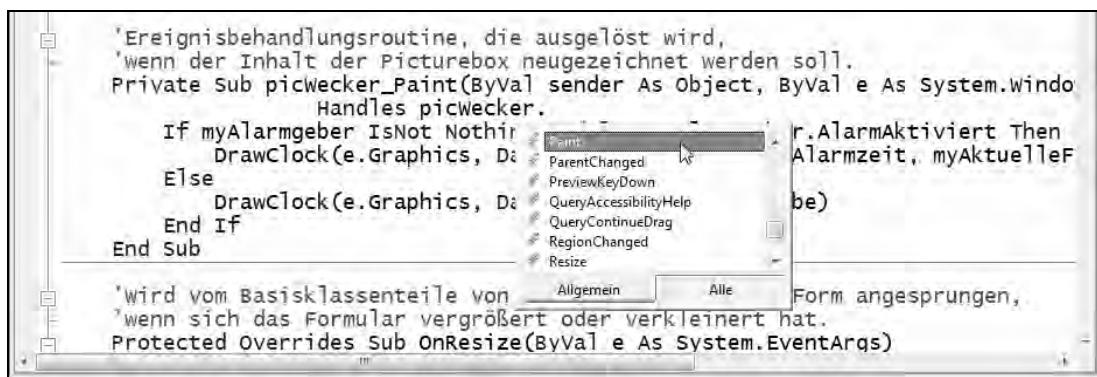


Abbildung 20.2 IntelliSense hilft Ihnen auch beim Verdrahten von Ereignissen mit einer Prozedur – sowohl bei der Auswahl möglicher Ereignis anbietender Objekte als auch bei der Auswahl der Ereignisse

Übrigens: Auch Objektvariablen, die Steuerelemente referenzieren, werden auf die gleiche Weise in den Formularcode eingebunden – Sie können sie jedoch nur auf Anhieb sehen. Der Designer, der ja auch den Formularcode generiert, nutzt nämlich die Möglichkeit von Visual Basic 2008, den Quellcode einer Klasse auf mehrere physische Dateien zu verteilen. Dadurch wird die eigentliche Codedatei eines Formulars aufgeräumter – einige Dinge, die hinter den Kulissen passieren, fehlen aber natürlich in dieser.

Um den Rest des Formularcodes sichtbar zu machen, klicken Sie im Projektmappen-Explorer auf das Symbol *Alle Dateien anzeigen* am oberen Rand dieses Toolfensters (der Tooltip hilft Ihnen beim Finden des richtigen Symbols). Sie werden sehen, dass sich anschließend vor jeder Formulardatei ein kleines Pluszeichen befindet, dessen dahinter stehenden Zweig Sie per Mausklick öffnen können. Sie werden des Weiteren feststellen, dass jedes Formular über eine Datei namens *form.designer.vb* verfügt, die diesen »Hinter-den-Kulissen-Code« beinhaltet. Und hier finden wir auch die Deklaration der verwendeten Steuerelemente – mit dem für die Ereignisverdrahtung notwendigen WithEvents:

```
.
. ' Am Ende der Datei frmMain.Designer.vb:
.

Friend WithEvents picWecker As System.Windows.Forms.PictureBox
Friend WithEvents Label1 As System.Windows.Forms.Label
Friend WithEvents mtbAlarmzeit As System.Windows.Forms.MaskedTextBox
Friend WithEvents chkAlarmAktivieren As System.Windows.Forms.CheckBox
Friend WithEvents btnOK As System.Windows.Forms.Button
Friend WithEvents btnAusschalten As System.Windows.Forms.Button

End Class
```

Auslösen von Ereignissen

Sie sehen im Codeauszug der Ereignisbehandlungs Routinen, der ab Seite 592 beginnt, wie Ereignisse generell mit bestimmten Prozeduren verknüpft werden. Der fett hervorgehobene Teil dieses Listingausschnittes konsumiert ein Ereignis der Instanz einer Klasse, die nicht Bestandteil des Frameworks ist – diese Klasse ist in Heimarbeit entstanden, und sie soll demonstrieren, wie Klassen Ereignisse auslösen können.

Schauen wir uns dazu den relevanten Klassencode der Datei *Alarmgeber.vb* an:

```
Public Class EinfacherAlarmgeber

Private WithEvents myTrigger As Timer
Private myAlarmzeit As Date
Private myAlarmAktiviert As Boolean
Private mySchwellwert As Integer = 2

''' <summary>
''' Wird ausgelöst, wenn eine bestimmte Zeit erreicht wurde.
''' </summary>
''' <param name="Sender">Das Objekt, das dieses Ereignis ausgelöst hat.</param>
''' <param name="e">AlarmEventArgs, die Näheres zum Objekt aussagen.</param>
''' <remarks></remarks>
Public Event Alarm(ByVal Sender As Object, ByVal e As AlarmEventArgs)
```

```
Sub New(ByVal Alarmzeit As Date)
    Me.Alarmzeit = Alarmzeit
End Sub

Sub New(ByVal Alarmzeit As Date, ByVal Aktiviert As Boolean)
    Me.Alarmzeit = Alarmzeit
    Me.AlarmAktiviert = Aktiviert
End Sub

.

.

.

Private Sub myTrigger_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles myTrigger.Tick
    If myAlarmzeit < DateTime.Now Then
        Dim locWeckzeitEventArgs As New AlarmEventArgs(Alarmzeit)
        OnWecken(locWeckzeitEventArgs)
        If locWeckzeitEventArgs.Neustellen Then
            Alarmzeit = locWeckzeitEventArgs.Alarmzeit
        Else
            AlarmAktiviert = False
        End If
    End If
End Sub

''' <summary>
''' Löst das Wecken aus.
''' </summary>
''' <param name="e"></param>
''' <remarks></remarks>
Protected Overrides Sub OnWecken(ByVal e As AlarmEventArgs)
    RaiseEvent Alarm(Me, e)
End Sub
End Class
```

Zunächst einmal benötigt die Klasse selbst eine Hilfe, die für das Auslösen des Ereignisses zur rechten Zeit sorgt, denn sie muss in regelmäßigen Abständen getriggert werden, um herauszufinden, ob die gestellte Alarmzeit bereits erreicht wurde. Dazu verwendet sie ein Timer-Objekt namens `myTrigger`, bindet es mit `WithEvents` als Member-Variable ein und sorgt durch die Verdrahtung dessen `Tick`-Ereignisses mit der Prozedur `myTrigger_Tick`, dass diese beim Ablauen des Timers aufgerufen wird.

Hier findet dann der Vergleich der aktuellen Uhrzeit mit der Weckzeit statt. Und wenn die Weckzeit erreicht wurde, dann wird es interessant:

`OnWecken` wird aufgerufen und diese Routine sorgt nun dafür, dass das eigentliche Ereignis mit `RaiseEvent` ausgelöst wird. `RaiseEvent` kann alle Ereignisse auslösen, die zuvor auf Klassenebene durch das Event-Schlüsselwort definiert wurden – in unserem Beispiel ist das lediglich das Ereignis `Alarm`.

Der Umweg über `Onxxx`

Falls Sie sich nun fragen, wieso `myTrigger_Tick` den Umweg über `OnWecken` nimmt: Denken Sie an OOP und die Polymorphie! Wenn Sie diese Klasse ableiten und in der abgeleiteten Version dieser Klasse das Wecker-Ereignis mitbekommen wollen, dann überschreiben Sie einfach die Methode `OnWecken`. Wenn die Ereignisbe-

handlungsroutine des Tick-Ereignisses `myTrigger_Tick` die Routine `OnWecken` aufruft, ruft es dann nämlich nicht mehr die »Basisversion« sondern Ihre neue Implementierung auf – und Sie bekommen in Ihrer abgeleiteten Klasse das Ereignis auf direktem Wege mit.

Genau das ist das Prinzip bei allen Ereignis auslösenden Komponenten, die Sie im Framework durch Vererbung verwenden – und Formulare gehören übrigens auch dazu. Aus diesem Grund erklärt sich auch die Funktionsweise folgender Routine aus dem Ereignisbeispiel ganz wie von selbst, die die Formularereignisse nicht als Ereignis und mit Handles an bestimmte Prozeduren hängt, sondern einfach den Basiscode des Formulars überschreibt: `OnResize` löst nämlich seinerseits das `Resize`-Ereignis aus, das eintritt, wenn das Formular vergrößert oder verkleinert wird.

```
'Wird vom Basisklassenteil von System.Windows.Forms.Form angesprungen,
'wenn sich das Formular vergrößert oder verkleinert hat.
Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
    'Wichtig: Basisfunktion aufrufen, sonst wird das
    'Resize-Ereignis nicht mehr ausgelöst!
    MyBase.OnResize(e)
    'Inhalt der PictureBox neuzeichnen,
    'wenn sich die Größe des Formulars geändert hat.
    picWecker.Invalidate()
End Sub
```

HINWEIS Dass überschreibbare Routinen, die Ereignisse auslösen, mit »On« beginnen, liegt einfach an der englischen Sprache: »Beim Feststellen der Notwendigkeit zum Auslösen des Größenändern-Ereignisses mache Folgendes« würde auf englisch in etwa »On the call of the resize event do the following« heißen, oder kurz: »BeimGrößenändern« bzw. »OnResize«.

WICHTIG Denken Sie beim Überschreiben von Ereignis auslösenden `Onxxx`-Methoden in abgeleiteten Klassen unbedingt daran, die Basismethode mit `myBase.Onxxx` aufzurufen. Andernfalls verhindern Sie, dass für Konsumenten von Instanzen Ihrer Klasse Ereignisse ausgelöst werden, da `RaiseEvent` in diesem Fall nicht mehr stattfindet.

Zur-Verfügung-Stellen von Ereignisparametern

Wenn Sie sich die verschiedenen Ereignisbehandlungsroutinen anschauen, werden Sie ein immer wiederkehrendes Schema in den Ereignissignaturen erkennen.

```
Private Sub myTrigger_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles myTrigger.Tick
    .
    .
    .
End Sub
```

- Es gibt einen Parameter – `sender` vom Typ `Object` – der über den Auslöser des Ereignisses Auskunft gibt. Die Variable `sender` ist deswegen als `Object` deklariert, da es ganz unterschiedliche Objekte (Textboxen, Listenfelder etc.) sein können, die das Ereignis auslösen.

- Es gibt einen Parameter – e vom Typ EventArgs (oder einer Ableitung von EventArgs) –, der entweder nähere Daten zum Ereignis liefert, oder mit dessen Hilfe sich die weitere Ereigniskette steuern lässt. Bei der Basisklasse System.EventArgs finden sich solche zusätzlichen Informationen nicht. Wenn Sie aber beispielsweise das KeyDown-Ereignis eines Formulars implementieren, finden Sie e definiert vom Typ System.Windows.Forms.KeyPressEventArgs, einem Typ, der unter anderem e.KeyChar (also das Zeichen der gedrückten Taste) bereitstellt.

Die Quelle des Ereignisses: Sender

Sender beinhaltet grundsätzlich die Quelle des Ereignisses als Objekt. Dies ist insbesondere dann wichtig, wenn eine Ereignisbehandlungsroutine gleich mehrere Ereignisse verschiedener Objekte behandeln soll, was in VB 6.0 noch völlig undenkbar war – denn Sie müssen ja wissen, wer für das Ereignis verantwortlich war und dementsprechend reagieren.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 20\\Ereignistest

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie dieses Projekt starten, sehen Sie wie in Abbildung 20.3 ein Formular mit mehreren Schaltflächen.

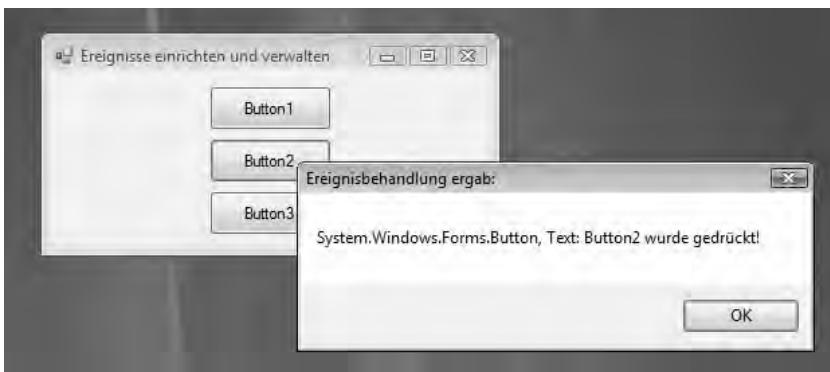


Abbildung 20.3 In diesem Beispiel gibt es eine Ereignisbehandlungsroutine für alle drei Schaltflächen

Soweit scheint es an diesem Beispiel auf den ersten Blick nichts Außergewöhnliches zu geben. Das Besondere ist aber: Alle drei Schaltflächen werden für den Fall des Anklickens vom Code berücksichtigt; es gibt aber lediglich eine einzige Ereignisbehandlungsroutine, die diese Behandlung übernimmt.

```
Public Class frmEreignisse  
  
    Private Sub Button1Und2Ereignisse(ByVal sender As System.Object, ByVal e As System.EventArgs) _  
        Handles Button1.Click, Button2.Click, Button3.Click
```

```
'Eine MessageBox wird dargestellt, wenn der Anwender Button2 oder Button3 auslöst.  
MessageBox.Show(sender.ToString & "wurde gedrückt!")  
  
End Sub  
End Class
```

Hier wird deutlich, wozu sender gut ist: Um überhaupt zu wissen, welche der Schaltflächen das Ereignis ausgelöst hat, müssen Sie sender unter die Lupe nehmen und entsprechende Fallunterscheidungen berücksichtigen, die dann das jeweils Richtige machen.

Übrigens: Da es sich bei sender um keine String-Variable mit beschreibendem Text, sondern um eine wirkliche Referenz auf das Ereignis auslösende Objekt handelt, können Sie sender auch wieder zurück z.B. mit Dim btn As Button= CType(Sender, Button) in seinen Ausgangstyp casten.

So ist es im Beispiel ganz sicher, dass ein Typ-Casting nicht schief gehen kann, da ja ausschließlich Schaltflächen zum Einsatz kommen. Die folgenden Zeilen sind also durchaus denkbar, um von sender »zurück« zum Schaltflächenobjekt (Button-Objekt) zu gelangen und mit diesem Objekt anschließend Manipulationen durchzuführen, die über sender selbst nicht möglich gewesen wären:

```
Public Class frmEreignisse  
  
Private Sub Button1Und2Ereignisse(ByVal sender As System.Object, ByVal e As System.EventArgs) _  
Handles Button1.Click, Button2.Click, Button3.Click  
  
'Eine MessageBox wird dargestellt, wenn der Anwender Button2 oder Button3 auslöst.  
MessageBox.Show(sender.ToString & " wurde gedrückt!", "Ereignisbehandlung ergab:")  
  
'Schaltfläche, die gedrückt wurde, rot einfärben.  
  
'FEHLER! So geht es nicht, da Sender vom Typ Object ist:  
sender.BackColor = Color.Red  
  
'So geht es, denn es gibt nur Schaltflächen, also  
'ist es sicher in Button zu casten:  
Dim locGedrückterButton As Button  
locGedrückterButton = DirectCast(sender, Button)  
  
'Jetzt klappt es mit dem Roteinfärben  
locGedrückterButton.BackColor = Color.Red  
End Sub  
End Class
```

Um zu erfahren, ob es sich bei der Variablen sender um einen bestimmten Typ handelt, muss man in Visual Basic die wenig schöne Syntax If TypeOf sender Is Button Then... benutzen; statt Button können Sie dann natürlich auch beliebig andere Typen überprüfen.

Nähere Informationen zum Ereignis: EventArgs

EventArgs ist eine Klasse, die ausschließlich dafür gedacht ist, Parameter an Ereignis empfangende Instanzen zu senden. Auch wenn ein Ereignis keine Parameter erforderlich macht, werden Ereignisparameter grundätzlich mit einer EventArgs-Instanz übergeben – zu diesem Zweck gibt es übrigens eine statische Funktion namens EventArgs.Empty, die ein inhaltsloses aber dennoch instanziertes EventArgs-Objekt generiert.

Wenn ein Ereignis bestimmte Parameter erforderlich macht, dann vererbt man EventArgs einfach in eine neue Klasse (deren Namen im Übrigen ebenfalls mit »EventArgs« enden sollte) und erweitert diese um die Eigenschaften und Konstruktoren, die erforderlich sind, um die Parameter für das Ereignis zu transportieren.

Eine Instanz von EventArgs sorgt in vielen Fällen aber nicht nur dafür, dass Parameter an die Ereignis einbindenden Instanzen übergeben werden. Eine Ereignis einbindende Instanz kann Parameter einer EventArgs-Instanz auch verändern, um dann ihrerseits zu signalisieren, dass die weitere Ereigniskette in irgendeiner Form gesteuert werden soll. Wenn Sie auf dem beschriebenen Weg eine Ereignisprozedur für das Form_Closing-Ereignis schreiben, können Sie beispielsweise mit e.Cancel=true das Schließen des Formulars auch verhindern – z.B. dann, wenn sich in Ihrer Anwendung noch ungesicherte Daten befinden, das Schließen des Formulars aber im konkreten Fall durch das Herunterfahren von Windows ausgelöst wurde.

HINWEIS Wenn dies der Fall ist

```
if e.ClosingReason=CloseReason.WindowsShutdown and UnsavedData=true then e.Cancel=true
```

wird sogar das Herunterfahren selbst abgebrochen – eine Besonderheit von Windows, da die Sicherung der Daten vorgeht, wie Sie es auch an anderen Windows-Programmen probieren können, wenn Sie z.B. in Word einen ungesicherten Text geöffnet haben und dann versuchen, den Rechner herunterzufahren.

Zurück zu unserem Alarm-Beispiel, dass auch diese Komponente der Ereignisprogrammierung demonstriert. Wenn Sie sich die Ereignis auslösende Prozedur nochmals vor Augen führen,

```
Private Sub myTrigger_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles _
myTrigger.Tick
If myAlarmzeit < DateTime.Now Then
    Dim locWeckzeitEventArgs As New AlarmEventArgs(Alarmzeit)
    OnWecken(locWeckzeitEventArgs)
    If locWeckzeitEventArgs.Neustellen Then
        Alarmzeit = locWeckzeitEventArgs.Alarmzeit
    Else
        AlarmAktiviert = False
    End If
End If
End Sub
```

stellen Sie fest, dass auch sie von einer benutzerdefinierten EventArgs-Klasse abgeleitet wurde, um für das Ereignis die Parameter Alarmzeit und NeuStellen zur Verfügung zu stellen.

Diese AlarmEventArgs-Klasse hat im Übrigen keine andere Funktion, außer diese beiden Parameter zur Verfügung zu stellen, und Sie finden Sie im Folgenden:

```
Public Class AlarmEventArgs
Inherits EventArgs

Private myAlarmzeit As Date
Private myNeuStellen As Boolean

Sub New(ByVal Alarmzeit As Date)
    myAlarmzeit = Alarmzeit
    myNeuStellen = True
End Sub
```

```

Sub New(ByVal Alarmzeit As Date, ByVal Neustellen As Boolean)
    myAlarmzeit = Alarmzeit
    myNeuStellen = Neustellen
End Sub

Public Property Alarmzeit() As Date
    Get
        Return myAlarmzeit
    End Get
    Set(ByVal value As Date)
        myAlarmzeit = value
    End Set
End Property

Public Property Neustellen() As Boolean
    Get
        Return myNeuStellen
    End Get
    Set(ByVal value As Boolean)
        myNeuStellen = value
    End Set
End Property
End Class

```

Und beide Codeausschnitte im Kontakt betrachtet zeigen nun, dass Ereignisparameter quasi in beide Richtungen fließen. Eine Prozedur, die das Ereignis verarbeitet, kann durch Setzen der Neustellen-Eigenschaft bestimmen, ob »der Wecker am nächsten Tag zur gleichen Zeit wieder schellen soll«. Wenn das Ereignis ausgelöst und eine Instanz der `AlarmEventArgs`-Klasse nach »oben hochgereicht« wird, dient ein Parameter (`Alarmzeit`) also der Ereignis einbindenden Klasse, der andere (`Neustellen`) der Ereignis auslösenden Klasse, den sie von der Ereignis einbindenden Klasse zurückhält. Das Setzen dieses Parameters sehen Sie am Beispiel im Listingausschnitt, der auf Seite 594 beginnt (`Private Sub myTrigger_Tick`).

Dynamisches Anbinden von Ereignissen mit AddHandler

Das Einbinden von Ereignissen versagt bei `WithEvents`, wenn Objekte, die Ereignisse zur Verfügung stellen, nur auf Prozedurebene (also lokal) deklariert werden oder Bestandteil eines Arrays oder einer Auflistung sind.

Bei solchen Konstellationen besteht die Möglichkeit, Ereignisse dynamisch und zur Laufzeit mithilfe von `AddHandler` zu verdrahten.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\C - OOP\\Kapitel 20\\Alarm02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Er erweitert die einfache »Alarmwecker«-Anwendung, indem es nicht nur eine einzige Weckzeit sondern gleich eine ganze Reihe von Terminen zu hinterlegen erlaubt.

Wenn Sie dieses Beispiel starten, sehen Sie einen Dialog, wie ihn auch Abbildung 20.4 darstellt.

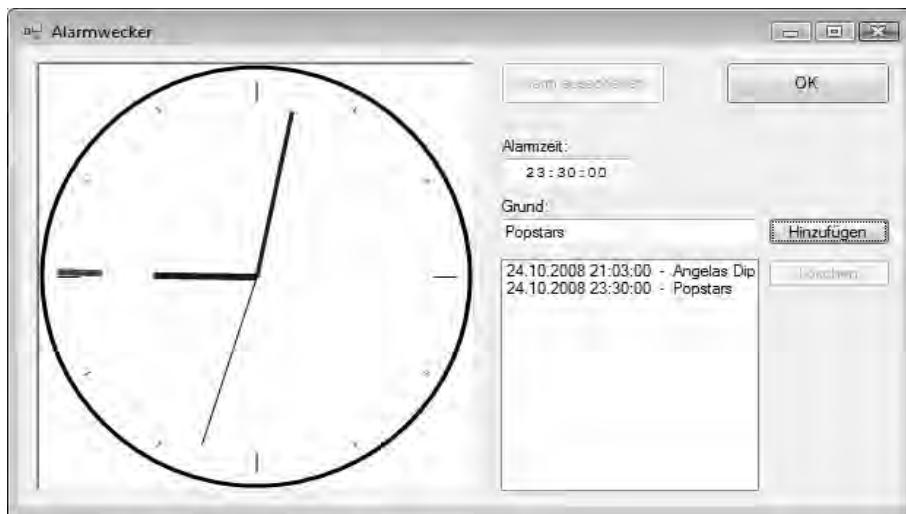


Abbildung 20.4 Mit der modifizierten »Wecker«-Applikation können Sie gleich mehrere Weckzeiten erfassen ...

Mit dieser Anwendung erfassen Sie nicht nur eine Weckzeit – Sie können gleich mehrere angeben. Außerdem erlaubt diese Version des Beispiels auch einen Alarmtext einzugeben, der im Falle des Eintretens des Ereignisses im Ziffernblatt der Uhr eingeblendet wird – wie Abbildung 20.5 zeigt.

Intern führt diese Version dabei eine Klasse zur Verwaltung einer Liste mit Objektinstanzen ähnlich der `DynamicList`, die Sie schon aus anderen Beispielen vorheriger Kapitel kennen. Doch dieses Mal verwenden wir eine, die bereits fest im Framework eingebaut ist – eine generische Auflistung namens `Collection`.

Die Schwierigkeit hierbei ist, dass wir nun nicht mehr eine globale Variable innerhalb der abgeleiteten `Collection`-Klasse verwenden können, die dann mit `WithEvents` zu deklarieren wäre, denn es geht ja nunmehr um eine *Liste* mit Alarmgeber-Instanzen.

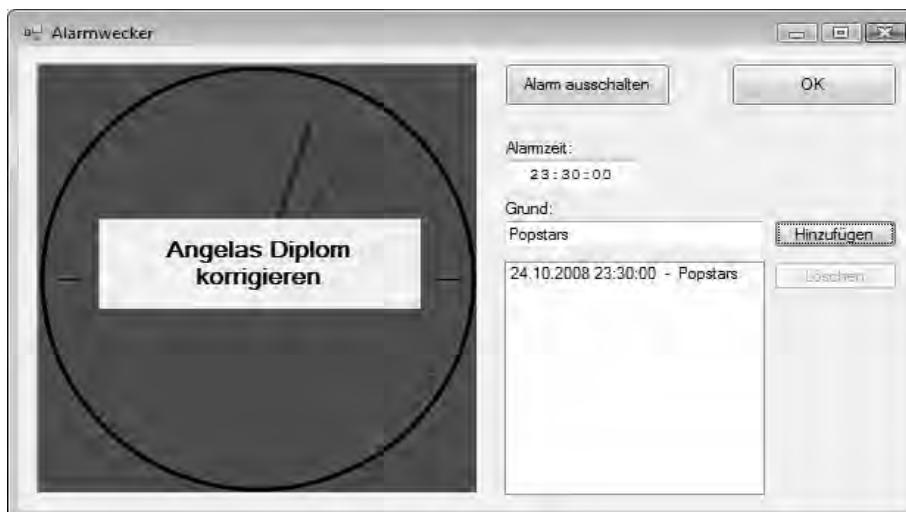


Abbildung 20.5 ... und ein zusätzlicher Meldungstext wird im Alarmfall zusätzlich im Ziffernblatt eingeblendet

Wir müssen also einen anderen Weg gehen, und der sieht folgendermaßen aus: Wann immer der Anwender einen neuen Wecktermin eingibt und somit eine neue Alarmgeber-Instanz der Liste mit Add hinzugefügt wird, müssen wir das in der Collection-Ableitung abfangen und einen Ereignisauslöser, den wir in dieser Klasse neu definieren, zur Laufzeit mit dem Alarm-Ereignis der Alarmgeber-Instanz verdrahten.

Umgekehrt müssen wir dafür sorgen, dass beim Löschen eines Alarmgebers aus der Liste zuvor diese Ereignisverknüpfung wieder aufgehoben wird. Und alle Ereignisverknüpfungen der vorhandenen Alarmgeber-Instanzen in der Liste müssen wir dann löschen, wenn sie mit Clear komplett gelöscht wird.

Und das sieht dann codemäßig wie folgt aus:

```
Imports System.Collections.ObjectModel

''' <summary>
''' Verwaltet eine Liste mit Alarmgeber-Objekten, und löst ein Ereignis aus,
''' wenn eines der Alarmgeber-Objekte das für erforderlich hält.
''' </summary>
''' <remarks></remarks>
Public Class Terminliste
    Inherits Collection(Of Alarmgeber)

    ''' <summary>
    ''' Wird ausgelöst, wenn eines der dieser Instanz hinzugefügten
    ''' Alarmgeber-Objekte seinerseits ein Alarm-Ereignis auslöst.
    ''' </summary>
    ''' <param name="sender">Referenz auf das Alarmgeber-Objekt, das das Ereignis ausgelöst hat.</param>
    ''' <param name="e">AlarmEventArgs-Instanz, die Parameter zum Ereignis enthalten.</param>
    ''' <remarks></remarks>
    Public Event Alarm(ByVal sender As Object, ByVal e As AlarmEventArgs)

        'Enthält Nothing oder das Datum des als nächstes anstehenden Termins.
        Private myNächsterTermin As Nullable(Of Date)

        'Wird beispielsweise durch Add oder Insert der Collection-Klasse ausgelöst.
        'Überschrieben, da das Alarm-Ereignis des Objektes mit der
        'AlarmHandler-Prozedur verknüpft werden muss.
        Protected Overrides Sub InsertItem(ByVal index As Integer, ByVal item As Alarmgeber)

            'Das Alarm-Ereignis des Objektes dynamisch mit der Prozedur AlarmHandler verbinden.
            AddHandler item.Alarm, AddressOf AlarmHandler

            'Die Basisprozedur machen lassen, was sie machen muss
            '(nämlich das Element an die richtige Stelle setzen).
            MyBase.InsertItem(index, item)

            'Liste hat sich geändert - der nächste Termin könnte ein anderer werden!
            AktualisiereNächsteTerminEigenschaft()
        End Sub

        'Wird durch Zuweisung eines Elementes über die Item-Eigenschaft aufgerufen.
        'Überschrieben, da das Alarm-Ereignis des Objektes mit der
        'AlarmHandler-Prozedur verknüpft werden muss.
        Protected Overrides Sub SetItem(ByVal index As Integer, ByVal item As Alarmgeber)
```

```
'Das alte Element an dieser Stelle lösen:  
RemoveHandler Me(index).Alarm, AddressOf AlarmHandler  
  
'Das neue Element verknüpfen  
AddHandler item.Alarm, AddressOf AlarmHandler  
  
'Die Basisprozedur machen lassen, was sie machen muss  
'(nämlich das Element an die richtige Stelle setzen).  
MyBase.SetItem(index, item)  
End Sub  
  
'Wird beispielsweise durch Remove oder RemoveAt der Collection-Klasse aufgerufen.  
'Überschrieben, um das verknüpfte Ereignis wieder mit AlarmHandler zu lösen.  
Protected Overrides Sub RemoveItem(ByVal index As Integer)  
  
'Das Alarm-Ereignis des Objektes dynamisch von der Prozedur AlarmHandler lösen.  
RemoveHandler Me(index).Alarm, AddressOf AlarmHandler  
  
'Die Basisprozedur machen lassen, was sie machen muss  
'(nämlich das Element aus der Liste löschen).  
MyBase.RemoveItem(index)  
  
'Liste hat sich geändert - der nächste Termin könnte ein anderer werden!  
AktualisiereNächsteTerminEigenschaft()  
End Sub  
  
'Beim Löschen aller Elemente werden die Ereignisse aller Objekte gelöst.  
Protected Overrides Sub ClearItems()  
  
'Alle Ereignisse lösen.  
For Each locItem As Alarmgeber In Me  
    RemoveHandler locItem.Alarm, AddressOf AlarmHandler  
Next  
  
'Basisroutine aufrufen.  
MyBase.ClearItems()  
  
'Gibt keinen "nächsten Termin" mehr.  
myNächsterTermin = Nothing  
End Sub  
  
''' <summary>  
''' Löst ein Ereignis aus, sobald diese Routine ihrerseits als  
''' Ereignisbehandlungsroutine eines der Elemente in dieser Collection in Aktion tritt.  
''' </summary>  
''' <param name="sender"></param>  
''' <param name="e"></param>  
''' <remarks></remarks>  
Private Sub AlarmHandler(ByVal sender As Object, ByVal e As AlarmEventArgs)  
    RaiseEvent Alarm(sender, e)  
End Sub  
  
''' <summary>  
''' Sucht den als nächstes anliegenden Termin in der Elementeliste.  
''' </summary>  
''' <remarks></remarks>  
Private Sub AktualisiereNächsteTerminEigenschaft()
```

```

'Keine Elemente vorhanden...
If Me.Count = 0 Then
    '...dann gibt es keinen nächsten Termin
    myNächsterTermin = Nothing
Else
    'Alle Elemente durchsuchen, welches Element "jünger"
    'als alle anderen ist.
    myNächsterTermin = Me(0).Alarmzeit
    For Each locItem As Alarmgeber In Me
        If locItem.Alarmzeit < myNächsterTermin.Value Then
            myNächsterTermin = locItem.Alarmzeit
        End If
    Next
End If
End Sub

''' <summary>
''' Liefert den nächsten anstehenden Termin oder Nothing zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property NächsterTermin() As Nullable(Of Date)
    Get
        'Nur Wert zurückliefern. Die Suche nach dem jüngsten
        'Datum wurde schon bei jeder Listenänderung durchgeführt.
        Return myNächsterTermin.Value
    End Get
End Property
End Class

```

Lassen Sie mich zum besseren Verständnis zunächst ein paar Worte zur generischen Collection verlieren, die übrigens über den Namespace System.Collections.ObjectModel und nicht – wie man vielleicht annehmen könnte – über System.Collections.Generics erreichbar ist.³ Sie tritt an die Stelle unserer bislang verwendeten DynamicList. Durch das Vererben dieser Klasse in eine neue Klasse Terminliste und das anschließende Überschreiben der Methoden InsertItem, RemoveItem und ClearItems können wir auf das Hinzufügen, das Löschen eines Elementes oder das Löschen aller Elemente dennoch genau so Einfluss nehmen, als hätte wir sie selber entwickelt. Dabei spielt es keine Rolle, ob beispielsweise das Einfügen neuer Elemente etwa durch Add oder Insert erfolgt – Collection sorgt dafür, dass in diesen Fällen InsertItem aufgerufen wird und wir so über die Vorgänge der Listenveränderung informiert werden. Mehr zum Thema Auflistungen (*Collections*) erfahren Sie übrigens in Kapitel 22.

³ Wieso das so ist, können Sie übrigens im Blog von Krzysztof Cwalina nachlesen, den Sie unter dem IntelliLink **D2001** finden. Sein Vorschlag, anstelle von Collection(Of Type) lieber List(Of Type) zu verwenden, können wir übrigens nicht in die Tat umsetzen, da List(Of Type) uns keinen Einfluss auf das Einfügen oder Entfernen von Elementen nehmen lassen könnte – List(Of Type) stellt keine überschreibbaren Methoden zur Verfügung, in die wir uns einklinken könnten.

Wenn nun ein neues Alarmgeber-Objekt der Liste hinzugefügt wurde, ist es wichtig, dass wir dessen Ereignis mitbekommen, damit wir, wenn der Ereignisfall dieses Alarmgeber-Objektes eintritt, wiederum ein Ereignis auslösen können. `WithEvents` scheidet aber, wie schon anfangs erwähnt, in diesem Fall aus, da wir es mit einer lokalen Objektvariable zu tun haben, die wir erst dann »kennen lernen«, wenn sie uns durch `InsertItem` zur Verfügung gestellt wird.

Aus diesem Grund verknüpfen wir das Ereignis dynamisch mit `AddHandler` zur Laufzeit – der erste in Fett-schrift gesetzte Block des vorherigen Listingauszugs zeigt, wie das funktioniert. `AddHandler` nimmt zwei Parameter: zum Einen das Objekt und dessen Ereignis (in diesem Fall `item.Alarm`), zum Anderen einen so genannten Delegaten (dazu später mehr), bei dem es sich in diesem Falle vereinfacht ausgedrückt um die Adresse einer Prozedur handelt, die der Signatur des Ereignisses entspricht, das es zu verknüpfen gilt. Wird später das `Alarm`-Ereignis für das entsprechende `Alarmgeber`-Objekt ausgelöst, das wir gerade im Begriff sind zur Liste hinzuzufügen, dann behandelt die Prozedur dieses Ereignis, das im zweiten Parameter von `AddHandler` mit `AddressOf` angegeben wurde.

Im umgekehrten Fall müssen wir dafür sorgen, das Ereignis wieder von dieser Prozedur zu lösen, wenn das `Alarmgeber`-Element aus der Liste entfernt wird. Dazu verwenden wir das Schlüsselwort `RemoveHandler`, das äquivalent zu `AddHandler` funktioniert. Und die gleiche »Umpolerei« passiert, wenn ein Element der Terminliste durch Zuweisung etwa wie

```
Terminlisteninstanz.Item(x)=NeuesAlarmgeberElement
```

geändert wird. In diesem Fall trennen wir das Ereignis des bisherigen Elements mit `RemoveHandler` und verknüpfen das neu zugewiesene mit `AddHandler`.

Im Ergebnis erreichen wir damit, dass egal welches `Alarmgeber`-Objekt der Terminliste ein `Alarm`-Ereignis auslöst, immer die Prozedur `AlarmHandler` aufgerufen wird, die ihrerseits dann ein Ereignis auslöst, das anschließend durch das Formular `frmMain` verarbeitet werden kann.

Damit das Hauptprogramm, das ja eine Instanz vom Typ `Terminliste` zur Speicherung der Termine einbindet, es mit dem Einzeichnen des jeweils nächsten Termins möglichst einfach hat, existiert eine Prozedur, die den jeweils nächsten Termin findet (`AktualisiereNächsteTerminEigenschaft`), in einer Member-Variablen (`myNächsterTermin`) ablegt und über eine Eigenschaft (`NächsterTermin`) dem Hauptprogramm zur Verfügung stellt. Das Aktualisieren dieser Variablen findet immer dann statt, wenn sich irgendetwas an der Terminliste geändert hat.

Für diese Member-Variable kommt dabei übrigens der generische Wertetyp `Nullable(Of)` zum Einsatz. Wie schon zu erfahren war, erlaubt es dieser, aus jedem Wertetyp einen »null-baren« Typ zu machen, der nicht nur seinen eigentlichen Wert, sondern auch `Nothing` speichern kann (was Wertetypen, wie Sie wissen, standardmäßig nicht können). Man kann dann mit der Eigenschaft `HasValue` prüfen, ob die `Nullable`-Instanz einen Wert hat (oder eben `Nothing` ist) und diesen mit dessen `Value`-Eigenschaft ermitteln. Da unsere Terminliste natürlich auch leer sein kann, kommen uns die Eigenschaften der generischen `Nullable`-Klasse sehr gelegen, da wir für den jeweils nächsten Termin lediglich ein Datum oder eben `Nothing` speichern müssen, falls die Liste leer ist. Und genau das wäre mit einer Objektvariablen nur vom Typ `Date` eben nicht möglich.

Die Verarbeitung der Terminliste innerhalb des Hauptprogramms, also in frmMain, sieht dann folgendermaßen aus:

```
Public Class frmMain

    Private WithEvents myTimer As Timer
    Private WithEvents myTerminliste As Terminliste

    'Die Hintergrundfarbe der Uhr, die bei
    'anhaltendem Alarm wechselt.
    Private myAktuelleFarbe As Color

    'Alarmsdauer in 500ms-Schritten (=25 Sekunden).
    Private myAlarmDauer As Integer = 50

    'Zähler für die Dauer des Restalarms.
    Private myAlarmDownCounter As Integer

    'True: Alarm ist gerade aktiv --> die Uhr blinkt.
    Private myAlarmStatus As Boolean

    ' Ist dieser String nicht Nothing, wird eine Textmeldung in der Uhr
    ' angezeigt, ansonsten nicht.
    Private myLetzteAlarmmeldung As String

    Sub New()
        ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
        InitializeComponent()

        'Diesen Timer benötigen wir zum Darstellen der Uhr
        'und zum Blinken der Uhr bei anhaltendem Alarm
        myTimer = New Timer()
        myTimer.Interval = 500
        myTimer.Start()

        'Standardhintergrundfarbe der Uhr ist weiß.
        myAktuelleFarbe = Color.White

        'Terminliste ist zunächst noch leer.
        myTerminliste = New Terminliste
    End Sub
```

Bis zu diesem Abschnitt findet das Vorgeplänkel statt. Benötigte Member-Variablen werden deklariert, und Sub New sorgt für die korrekte Initialisierung aller Member-Variablen (sowie mit dem Aufruf von InitializeComponent() auch für die Ausstattung des Formulars mit allen notwendigen Steuerelementen). Besonders wichtig ist hier natürlich das Timer-Objekt myTimer, das sich um die regelmäßige Aktualisierung der Uhldarstellung kümmert. Dieser Timer läuft alle 500 ms ab und leitet dann in der myTimer_Tick-Ereignisbehandlung mit Invalidate ein Neuzeichnen des PictureBox-Inhaltes ein.

Im Unterschied zur vorherigen Version des Programms verwenden wir an dieser Stelle nicht nur ein einfaches Alarmgeber-Objekt sondern die neu implementierte Liste myTerminliste, die mit WithEvents deklariert wurde, damit uns das Alarm-Ereignis erreicht, sobald eines der in ihr enthaltenen Alarm-Objekte seinerseits ein Alarm-Ereignis auslöst. Dieses Ereignis wird im folgenden Codeteil behandelt.

```
'Ereignisbehandlungsroutine, die ausgelöst wird,  
'wenn der Alarmgeber Alarm signalisiert, weil eine  
'bestimmte Uhrzeit erreicht wurde.  
Private Sub myAlarmgeber_Alarm(ByVal Sender As Object, ByVal e As AlarmEventArgs) _  
    Handles myTerminliste.Alarm  
    'Wecker schellt!  
    myAlarmStatus = True  
    'Und zwar so lange.  
    myAlarmDownCounter = myAlarmDauer  
    'Abschalten sollten wir das Schellen können.  
    btnAlarmAusschalten.Enabled = True  
  
    'Hochgereichten Meldungstext setzen  
    myLetzteAlarmmeldung = e.AlarmText  
  
    'Aus Liste löschen  
    lstTermine.Items.Remove(Sender)  
  
    'Aus Terminliste löschen  
    myTerminliste.Remove(DirectCast(Sender, Alarmgeber))  
End Sub
```

Die AlarmEventArgs – Sie sehen es an dieser Stelle – haben ebenfalls eine kleine Überarbeitung erfahren: Sie liefern den Meldungstext direkt mit, sodass dieser dann anschließend in der Mitte des Ziffernblattes angezeigt werden kann.

Wenn nun eines der Alarmgeber-Elemente ein Alarm-Ereignis ausgelöst hat, erfahren wir es über den Umweg Terminliste an dieser Stelle. Wir sorgen dann dafür, dass die »Weckphase« in der Darstellung eingeleitet wird und im Übrigen auch dafür, dass man den zum Termin gehörigen Meldungstext im Ziffernblatt der Uhr sieht (durch Setzen von myLetzteAlarmmeldung = e.AlarmText – dieser gesetzte Meldungstext wird dann beim nächsten Neuzeichnen der Uhr berücksichtigt).

```
.  
. .  
End Sub  
  
'Wird aufgerufen, wenn der Anwender die Hinzufügen-Schaltfläche betätigt hat  
Private Sub btnHinzufügen_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles btnHinzufügen.Click  
  
    Dim locAlarmzeit As Date  
  
    'Zeit und Termingrund aus den TextBox-Steuerelementen holen.  
    If Date.TryParse(mtbAlarmzeit.Text, locAlarmzeit) And _  
        Not String.IsNullOrEmpty(txtGrund.Text) Then
```

```
'Neues Alarmgeber-Objekt instanziieren, das wir anschließend  
'direkt zur Listbox...  
Dim locAlarm As New Alarmgeber(locAlarmzeit, txtGrund.Text, True)  
lstTermine.Items.Add(locAlarm)  
  
'...sowie zur Terminliste hinzufügen.  
myTerminliste.Add(locAlarm)  
  
'Uhr Neuzeichnen, damit die Weckzeit des nächsten Termins als  
'roter Strich ins Ziffernblatt kommt!  
picWecker.Invalidate()  
Else  
  
'Ups! Solch eine Uhrzeit gibt es nicht - TryParse  
'ist fehlgeschlagen.  
MessageBox.Show("Bitte überprüfen Sie die Eingabe auf Fehler!")  
End If  
End Sub
```

Neu und damit erwähnenswert ist die oben stehende Routine, die dafür sorgt, dass ein neues Alarmgeber-Objekt erstellt und der Liste hinzugefügt wird, wenn der Anwender die Daten für einen neuen Termin erfasst und anschließend auf *Hinzufügen* klickt.

Hier wird übrigens deutlich, dass das Programm mit einem kleinen Manko zu kämpfen hat, denn die Referenzen auf die eigentlichen Termine – die Alarmgeber-Objekte – werden im Grunde genommen in zwei Listen gespeichert: in der Collection(of Alarmgeber)-Auflistung und in der internen Liste des ListBox-Steuerelements lstTermine. Die ListBox-Liste benötigen wir aber einerseits für die Listendarstellung der Termine; die Collection(of Alarmgeber)-Auflistung dafür, dass wir »mitbekommen«, wenn eines der in ihr enthaltenen Alarmgeber-Objekte ein Ereignis ausgelöst hat. Dementsprechend müssen wir beide Listen pflegen, wenn der Anwender einen neuen Termin erfasst oder einen bereits erfassten Termin wieder aus der Liste löscht.

Ein Ansatz, dieses Manko zu umgehen, wäre es, die komplette AddHandler-Logik in das Hauptprogramm zu verlegen. Das würde allerdings dem OOP-Anspruch, wieder verwendbare Komponenten zu schaffen, nicht gerade entsprechen.

Teil D

Programmieren von und mit Datenstrukturen des .NET Frameworks

In diesem Teil:

Enums	611
Arrays und Collections	619
Generics (Generika) und generische Auflistungen	667
Serialisierung von Typen	697
Attribute und Reflection – Wenn Klassen ein Bewusstsein entwickeln	727
Kultursensible Klassen entwickeln	743
Reguläre Ausdrücke (Regular Expressions)	779

Kapitel 21

Enums

In diesem Kapitel:

Bestimmung der Werte der Aufzählungselemente	613
Bestimmung der Typen der Aufzählungselemente	614
Konvertieren von Enums	614
Flags-Enum (Flags-Aufzählungen)	615

Enums (etwa: *Aufzählungen*, nicht zu verwechseln mit *Auflistungen*, den *Collections*) dienen in erster Linie dazu, Programmierern das Leben zu erleichtern. Programmierer müssen sich gerade in den Zeiten von .NET schon eine ganze Menge merken und sind für jede Unterstützung in dieser Richtung dankbar.

Mit Enums rufen Sie konstante Werte, die thematisch zu einer Gruppe gehören, per Namen ab. Gleichzeitig haben Sie die Möglichkeit, die Werte auf diese Namen zu beschränken. Angenommen, Sie haben eine kleine Kontaktverwaltung auf die Beine gestellt, und diese Datenbankanwendung erlaubt es Ihnen, Ihre Kontakte zu kategorisieren. Sie können also einstellen, ob ein Kontakt ein Kunde, ein Bekannter, ein Geschäftskollege ist oder sonst einer Gruppierung angehört, zur Vereinfachung des Beispiels nehmen wir ferner an, dass ein Kontakt jeweils nur zu einer dieser Gruppen gehören kann (es soll sozial Hochbegabte geben, die mit ihren Geschäftskollegen auch noch bekannt sind). Für jede dieser Kategorien haben Sie eine bestimmte Nummer vergeben. Und in Abhängigkeit bestimmter Kategorien (Nummern) können Sie nun spezielle Funktionen in Ihrer Datenbankanwendung aufrufen oder auch nicht (Ansprechpartner-Kontakte lassen sich beispielsweise Firmenkontakten zuordnen, Lieferanten aber nicht, da diese Kontakte je selbst eine Firma darstellen – nur als Beispiel). Ohne Enums könnte man den Code so schreiben, dass man ihre Funktionen auch mit der Nummer (z.B. mit einem Integer) aufruft, aber dies ist zum Einen schwer zu merken und zum Anderen könnten Sie kaum verhindern, dass man die Funktionen mit 42 aufruft; was als Integer vollkommen in Ordnung wäre – nur haben Sie eben keine 42. Kategorie, sondern gerade mal 10 o. Ä. So leisten Enums sowohl die bessere Merkbarkeit als auch die Einschränkung möglicher Werte.

Wie gesagt: Sie können sich eine eigene Liste erstellen und die Zuordnung Kontaktkategorie/Nummer sozusagen im Kopf durchführen. Oder Sie legen praktischerweise eine solche Enum an, die Ihnen die Arbeit erleichtert.

BEGLEITDATEIEN

Die in einem Projekt zusammengefassten Beispiele für dieses Kapitel finden Sie im Verzeichnis

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 21\\Enums

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Public Enum KontaktKategorie
    Familie
    Freund
    Bekannter
    Kollege
    Geschäftspartner
    Kunde
    Lieferant
    ZuMeiden
    Firma
    AnsprechpartnerBeiFirmenKontakt
End Enum
```

Wie setzen Sie diese Enum-Elemente nun ein? Ganz einfache Geschichte: Wenn nichts anderes gesagt wird, vergibt .NET den Enums Werte, und zwar von oben nach unten bei 0 angefangen in aufsteigender Reihenfolge – was Sie nun aber nicht mehr wissen müssen. Sie können Variablen vom Typ der Aufzählung definieren und anstatt eine Nummer zu verwenden, einfach den Enum-Typ angeben:

```
Sub main()
    Dim lckkontakte As KontaktKategorie
    lckkontakte = KontaktKategorie.Geschäftspartner
    Console.WriteLine(lckkontakte)
    Console.ReadLine()

End Sub
```

Wenn Sie diesen Code ausführen, gibt er Ihnen im Konsolenfenster den Wert 4 aus.

Bestimmung der Werte der Aufzählungselemente

Falls Sie mit der vorgegebenen Durchnummerierung nicht einverstanden sind, legen Sie bei der Aufzählungsdefinition eben selbst Hand an, wie im folgenden Beispiel:

```
Public Enum KontaktKategorie
    Familie = 10
    Freund
    Bekannter
    Kollege
    Geschäftspartner
    Kunde = 20
    Lieferant
    ZuMeiden = 30
    Firma
    AnsprechpartnerBeiFirmenKontakt
End Enum
```

Das gleiche Programm, abermals ausgeführt, liefert anschließend den Wert 14.

Dubletten sind erlaubt!

Dubletten sind bei den Aufzählungselementen übrigens erlaubt. So haben in der Aufzählungsdefinition

```
Public Enum Kontaktkategorie
    Familie = 10
    Freund
    Bekannter
    Kollege
    Geschäftspartner = 20
    Kunde
    Lieferant = 19
    ZuMeiden
    Firma
    AnsprechpartnerBeiFirmenKontakt
End Enum
```

sowohl Geschäftspartner als auch ZuMeiden den Wert 20.

Bestimmung der Typen der Aufzählungselemente

Solange nichts anderes gesagt wird, werden die Elemente einer Aufzählung intern als Integer angelegt. Sie können allerdings bestimmen, ob die Aufzählungselemente darüber hinaus als Byte, Short oder Long deklariert werden sollen. Wir erinnern uns jedoch daran, dass Integer (in .NET Double WORDs) aufgrund der Ausrichtung an geraden Speicheradressen bei Intel Prozessoren aus Performance-Gründen meist vorzuziehen sind. Eine entsprechende Typen-Angabe bei der Enum-Definition reicht aus:

```
Public Enum KontaktKategorie As Short
    Familie
    Freund
    Bekannter
    ...
End Enum
```

Ermitteln des Elementtyps zur Laufzeit

Wenn Sie zur Laufzeit ermitteln müssen, welcher primitive Datentyp sich hinter einem Aufzählungselement verbirgt, machen Sie das einfach mit der GetUnderlyingType-Eigenschaft:

```
'Ermittelt den Namen des zu Grunde liegenden primitiven Datentyps einer Aufzählung.
Console.WriteLine([Enum].GetUnderlyingType(GetType(KontaktKategorie)).Name)

'Ermittelt den Typnamen anhand einer Aufzählungsvariablen.
Console.WriteLine([Enum].GetUnderlyingType(locKontakte.GetType).Name)
```

Auf unser bisheriges Beispiel angewendet, würden diese beiden Zeilen folgende Ausgabe im Konsolenfenster erscheinen lassen:

```
Int16
Int16
```

WICHTIG Wenn Sie auf den Enum-Typbezeichner im Quelltext zugreifen, müssen Sie, wie hier im Beispiel, das Schlüsselwort in eckige Klammern setzen, damit es vom Visual Basic-Editor korrekt als Ausdruck verarbeitet werden kann.

Konvertieren von Enums

In vielen Fällen kann es sinnvoll sein, ein Aufzählungselement in seinen zu Grunde liegenden Typ umzuwandeln – beispielsweise um den Wert einer Enum-Variablen in eine Datenbank zu schreiben. Einige Fälle machen es auch nötig, einen Aufzählungswert auf Grund des als Zeichenkette vorliegenden Namens eines Aufzählungselementes entstehen zu lassen oder ein Aufzählungselement in seinen Elementnamen (also in einen String) umzuwandeln.

In Zahlenwerte umwandeln und aus Werten definieren

Um ein Aufzählungselement in seinen Wert umzuwandeln (und im Bedarfsfall auch zurück), verfahren Sie folgendermaßen (das folgende Beispiel geht immer noch von einer Aufzählungstyp-Definition als Short aus):

```
Dim locKontakte As KontaktKategorie  
Dim locShort As Short  
  
    locKontakte = KontaktKategorie.Geschäftspartner  
  
    'Typumwandlung von Aufzählung zu Grunde liegenden Datentyp...  
    locShort = locKontakte  
    locShort = KontaktKategorie.Firma  
  
    '...und umgekehrt, auch nicht schwieriger:  
    locKontakte = CType(locShort, KontaktKategorie)
```

In Strings umwandeln und aus Strings definieren

Falls Sie wissen wollen, welcher Elementname sich hinter dem Wert einer Aufzählungsvariablen verbirgt, verfahren Sie folgendermaßen:

```
Dim locKontakte As KontaktKategorie = KontaktKategorie.Firma  
Console.WriteLine(locKontakte.ToString())
```

Die Ausgabe lautet:

```
Firma
```

Beim umgekehrten Verfahren wird es wieder ein wenig aufwändiger:

```
'Umwandlung zurück in ein Enum-Element aus einem String.  
Dim locString As String = "Geschäftspartner"  
locKontakte = DirectCast([Enum].Parse(GetType(KontaktKategorie), locString), KontaktKategorie)
```

Hier gilt: Die statische Funktion Parse der Enum-Klasse erlaubt das Generieren eines Aufzählungselementes zur Laufzeit. Parse erwartet dabei den Typ der Aufzählung, den Sie zunächst mit GetType ermitteln müssen. Da Parse ein Objekt erzeugt, das ein geboxedes Aufzählungselement enthält, müssen Sie dieses zu guter Letzt mit DirectCast aus dem Object wieder »entboxen«.

Flags-Enum (Flags-Aufzählungen)

Flags-Aufzählungen sind eine ideale Einrichtung, wenn Sie Aufzählungen mit Elementen verwenden müssen, die untereinander kombinierbar sind. So kann es durchaus sein, dass – um bei unserem Beispiel zu bleiben – ein Kontakt in Ihrer Datenbank sowohl ein *Freund* als auch ein *Geschäftskollege* ist. Das Framework unterstützt solche Szenarien auf ideale Weise.

Bei der Definition einer *Flags*-Aufzählung müssen Sie drei Dinge beachten: Sie sollten eine Aufzählungsbezeichnung für keine der Kombinationen definieren (beispielsweise in Form von Keine oder None). Dieser Eintrag hat den Wert 0. Zweitens: Sie müssen Werte vergeben, die sich bitweise kombinieren lassen – und dazu zählen Sie die einzelnen Werte als Zweierpotenzen hoch. Und drittens: Sie stattet die Aufzählung mit dem Flags-Attribut aus.

Das klassische Beispiel sind die Schaltflächenanordnungen und Hinweis-Symbole der in Windows »eingebauten« Message Box. Diese schon sehr alte Windows API aus *User32.dll* konnte man früher in C/C++ direkt aufrufen. Die Definition lautete:

```
int MessageBox(HWND hWnd,LPCTSTR lpText,LPCTSTR lpCaption,UINT uType);
```

Die Syntax aus C soll uns hier nicht interessieren – wie man hier mit welchem Parameter welches Ziel erreichte, ist schon eher interessant: Wollen Sie beispielsweise ein Fragezeichen anzeigen, müssen Sie als uType 32 angeben, für die Ja und Nein-Schaltflächen 4 als uType; 4+32=36 also für beide zusammen. .NET stellt Ihnen diese API in einem Objekt gekapselt ebenfalls zur Verfügung:

```
MessageBox.Show("Hallo world","VB for ever",MessageBoxButtons.YesNo,MessageBoxIcon.Question)
```

Um es allerdings für die Anwender etwas einfacher zu machen, hat man jedoch die Parameter für Schaltflächen und Hinweis-Symbole dort getrennt. Und jetzt raten Sie mal, welche Werte *MessageBoxButtons.YesNo* und *MessageBoxIcon.Question* zugewiesen wurden – doch nicht etwa 4 und 32?

Das folgende Beispiel zeigt, wie es geht:

```
<Flags()>
Public Enum KontaktKategorien
    Keine = 0
    Familie = 1
    Freund = 2
    Bekannter = 4
    Kollege = 8
    Geschäftspartner = 16
    Kunde = 32
    Lieferant = 64
    ZuMeiden = 128
    Firma = 256
    AnsprechpartnerBeiFirmenKontakt = 512
End Enum
```

Wenn Sie anschließend kombinierte Zuweisungen an eine Variable vom Typ *KontaktKategorien* vornehmen wollen, nehmen Sie den logischen Or-Operator zu Hilfe, wie das folgende Beispiel zeigt:

```
Sub EnumFlags()
    Dim lockkontakte As KontaktKategorien
    lockkontakte = KontaktKategorien.Freund Or KontaktKategorien.Geschäftspartner
    Console.WriteLine(lockkontakte)
    Console.WriteLine(lockkontakte.ToString())
```

```
Console.ReadLine()  
End Sub
```

Dieses Beispiel erzeugt die folgende Ausgabe:

```
18  
Freund, Geschäftspartner
```

Abfrage von Flags-Aufzählungen

Bei der Abfrage von *Flags*-Aufzählungen müssen Sie ein wenig aufpassen, da Kombinationen Werte ergeben, die keinem bestimmten Wert eines Elementes entsprechen. Erst ein so genanntes Ausmaskieren (Ermitteln eines einzelnen Bitwertes) mit dem *And*-Operator ergibt einen richtigen Wert. Auch hier demonstriert ein Beispiel, wie es geht:

```
Dim locKontakte As KontaktKategorien  
locKontakte = KontaktKategorien.Freund Or KontaktKategorien.Geschäftspartner  
  
'Achtung bei Flags! Bits müssen ausmaskiert werden!  
'Diese Abfrage liefert das falsche Ergebnis!  
If locKontakte = KontaktKategorien.Geschäftspartner Then  
    Console.WriteLine("Du bist ein Geschäftspartner")  
Else  
    Console.WriteLine("Du bist kein Geschäftspartner")  
End If  
  
'So ist's richtig; diese Abfrage liefert das richtige Ergebnis:  
If (locKontakte And KontaktKategorien.Freund) = KontaktKategorien.Freund Then  
    Console.WriteLine("Du bist ein Freund")  
Else  
    Console.WriteLine("Du bist kein Freund")  
End If  
  
'Und so funktionieren Kombiabfragen:  
If locKontakte = (KontaktKategorien.Freund Or KontaktKategorien.Geschäftspartner) Then  
    Console.WriteLine("Du bist ein Freund und ein Geschäftspartner")  
End If
```

Wenn Sie dieses Beispiel laufen lassen, erhalten Sie das folgende Ergebnis:

```
Du bist kein Geschäftspartner  
Du bist ein Freund  
Du bist ein Freund und ein Geschäftspartner
```

Die erste Zeile wird falsch ausgegeben, da der Wert der *Enum*-Variablen *locKontakte* nicht ausmaskiert und dann verglichen wird.

Kapitel 22

Arrays und Collections

In diesem Kapitel:

Grundsätzliches zu Arrays	620
Enumeratoren	635
Grundsätzliches zu Auflistungen (Collections)	638
Wichtige Auflistungen der Base Class Library	642

Arrays kennt fast jedes Basic-Derivat seit Jahrzehnten – und natürlich können Sie auch in Visual Basic .NET auf diese Datenfelder (so der deutsche Ausdruck) zurückgreifen. Doch das .NET Framework wäre nicht das .NET Framework, wenn nicht auch Arrays viel mehr Möglichkeiten bieten würden, als nur auf Daten über einen numerischen Index zuzugreifen.

Das bedeutet, dass die Leistung von Arrays weit über das reine Zur-Verfügung-Stellen von Containern für die Speicherung verschiedener Elemente eines Datentyps hinausreicht. Da Arrays auf Object basieren und damit eine eigene Klasse darstellen (System.Array nämlich), bietet das Framework über Array-Objekte weitreichende Funktionen zur Verwaltung ihrer Elemente an.

So können Arrays beispielsweise quasi auf Knopfdruck sortiert werden. Liegen sie in sortierter Form vor, können Sie auch binär nach ihren Elementen suchen und viele weitere Dinge mit ihnen anstellen, ohne selbst den entsprechenden Code dafür entwickeln zu müssen.

Zu guter Letzt bildet der Typ System.Array aber auch die Basis für weitere Datentypen – zum Beispiel ArrayList –, die Datenelemente in einer bestimmten Form verwalten können, aber auch die Grundlage für viele der generischen Auflistungstypen, die wiederum die Basis für *LINQ to Objects* bilden.

Dieses Kapitel zeigt Ihnen, was Sie mit Arrays und den von ihnen abgeleiteten Klassen alles anstellen können, und bereitet Sie nicht zuletzt damit auf den Umgang mit *LINQ to Objects* vor.

Grundsätzliches zu Arrays

Arrays im ursprünglichen Basic-Sinne dienen dazu, *mehrere* Elemente desselben Datentyps unter einem bestimmten Namen verfügbar zu machen. Um die einzelnen Elemente zu unterscheiden, bedient man sich eines Indexes – der im übrigen nichts mit einem Index für die Datensätze einer Datenbanktabelle zu tun hat – (ganz einfach ausgedrückt: einer Nummerierung der Elemente), damit man auf die verschiedenen Array-Elemente zugreifen kann.

BEGLEITDATEIEN Viele der größeren nun folgenden Beispiele sind im Projekt *Arrays* in verschiedenen Methoden zusammengefasst. Das Projekt befindet sich im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 22\\Arrays

Sie können dieses Projekt verwenden, um die Beispiele an Ihrem eigenen Rechner nachzuvollziehen oder um eigene Experimente mit Arrays durchzuführen.

Ein Beispiel:

```
Sub Beispiel1()
    'Array mit 10 Elementen fest deklarieren.
    'Wichtig: Anders als in C# oder C++ wird der Index
    'des letzten Elementes, nicht die Anzahl der Elemente
    'festgelegt! Elementzählung beginnt bei 0.
    'Die folgende Anweisung definiert also 10 Elemente:
    Dim locIntArray(9) As Integer

    'Zufallsgenerator initialisieren,
    Dim locRandom As New Random(Now.Millisecond)
```

```
For count As Integer = 0 To 9
    locIntArray(count) = locRandom.Next
Next

For count As Integer = 0 To 9
    Console.WriteLine("Element Nr. {0} hat den Wert {1}", count, locIntArray(count))
Next

Console.ReadLine()

End Sub
```

Wenn Sie dieses Beispiel ausführen, erscheint im Konsolenfenster eine Liste mit Werten, etwa wie im nachstehenden Bildschirmauszug zu sehen (die Werte sollten sich natürlich von Ihren unterscheiden, da es zufällige sind).¹

```
Element Nr. 0 hat den Wert 1074554181
Element Nr. 1 hat den Wert 632329388
Element Nr. 2 hat den Wert 1312197477
Element Nr. 3 hat den Wert 458430355
Element Nr. 4 hat den Wert 1970029554
Element Nr. 5 hat den Wert 503465071
Element Nr. 6 hat den Wert 112607304
Element Nr. 7 hat den Wert 1507772275
Element Nr. 8 hat den Wert 1111627006
Element Nr. 9 hat den Wert 213729371
```

Dieses Beispiel demonstriert die grundsätzliche Anwendung von Arrays. Natürlich sind Sie bei der Definition des Elementtyps nicht auf Integer festgelegt. Es gilt der Grundsatz: Jedes Objekt in .NET kann Element eines Arrays sein.

Änderung der Array-Dimensionen zur Laufzeit

Das Definieren der Array-Größe mit variablen Werten macht Arrays – wie im Beispiel des letzten Abschnitts gezeigt – zu einem sehr flexiblen Werkzeug bei der Verarbeitung von großen Datenmengen. Doch Arrays sind noch flexibler: Mit der ReDim-Anweisung gibt Ihnen Visual Basic die Möglichkeit, ein Array noch nach seiner ersten Deklaration neu zu dimensionieren. Damit werden selbst einfache Arrays zu dynamischen Datencontainern. Auch hier soll ein Beispiel den Umgang verdeutlichen:

```
Sub Beispiel3()
    Dim locAnzahlStrings As Integer = 15
    Dim locStringArray As String()
```

¹ Kleine Anmerkung am Rande: Ganz so zufällig sind die Werte nicht – Random stellt nur sicher, dass generierte Zahlenfolgen zufällig verteilt sind. Bei gleicher Ausgangsbasis (definiert durch den Parameter Seed, den Sie der Random-Klasse beim Instanziieren übergeben) produziert Random auch gleiche Zahlenfolgen. Da wir hier die Millisekunde als Basis übergeben, und eine Sekunde aus 1000 Millisekunden besteht, gibt es eine Wahrscheinlichkeit von 1:1000, dass Sie dieselbe wie die hier abgedruckte Zahlenfolge generieren lassen. Es ist immer schwer, einer deterministischen Maschine so etwas indeterministisches wie Zufall beizubringen...

```

locStringArray = GeneriereStrings(locAnzahlStrings, 30)
Console.WriteLine("Ausgangsgröße: {0} Elemente. Es folgt der Inhalt:", locStringArray.Length)
Console.WriteLine(New String("c", 40))
DruckeStrings(locStringArray)
'Wir brauchen 10 weitere, die alten sollen aber erhalten bleiben!
ReDim Preserve locStringArray(locStringArray.Length + 9)

'Bleiben die alten wirklich erhalten?
Console.WriteLine()
Console.WriteLine("Inhaltsüberprüfung:", locStringArray.Length)
Console.WriteLine(New String("c", 40))
DruckeStrings(locStringArray)

'10 weitere Elemente generieren.
Dim locTempStrings(9) As String

'10 Zeichen mehr pro Element, sodass wir die neuen leicht erkennen können.
locTempStrings = GeneriereStrings(10, 40)

'In das "alte" Array kopieren, aber ab Index 15,
locTempStrings.CopyTo(locStringArray, 15)

'und nachschauen, was nun wirklich drinstehrt!
Console.WriteLine()
Console.WriteLine("Inhaltsüberprüfung:", locStringArray.Length)
Console.WriteLine(New String("c", 40))
DruckeStrings(locStringArray)

Console.ReadLine()
End Sub

```

Dieses Beispiel macht sich die Möglichkeit zunutze (direkt in der ersten Codezeile), die Dimensionierung und Deklaration eines Arrays zeitlich voneinander trennen zu können. Das Array `locStringArray` wird zunächst nur als Array deklariert – wie groß es sein soll, wird zu diesem Zeitpunkt noch nicht bestimmt. Dabei spielt es in Visual Basic übrigens keine Rolle, ob Sie eine Variable in diesem

```
Dim locStringArray As String()
```

oder diesem

```
Dim locStringArray() As String
```

Stil als Array deklarieren.

Die Größe des Arrays wird das erste Mal von der Prozedur `GeneriereString` festgelegt. Hier erfolgt zwar die Dimensionierung eines zunächst völlig anderen Arrays (`locStrings`); da dieses Array aber als Rückgabewert der aufrufenden Instanz überlassen wird, lebt der hier erstellte Array-Inhalt unter anderem Namen (`locStringArray`) weiter. Das durch beide Objektvariablen referenzierte Array ist dasselbe (im ursprünglichen Sinne des Wortes).

Übrigens: Diese Vorgehensweise entspricht eigentlich schon einem typsichereren Neudimensionieren eines Arrays. Wie Sie beim ersten Array-Beispiel gesehen haben, spielt es natürlich keine Rolle, ob Sie eine Array-Variable zur Aufnahme eines Array-Rückgabeparameters verwenden, die zuvor mit einer festen Anzahl an

Elementen oder ohne die Angabe der Array-Größe dimensioniert wurde. Allerdings: Sie verlieren bei dieser Vorgehensweise den Inhalt des ursprünglichen Arrays, denn die Unterroutine erstellt ein neues Array, und mit der Zuweisung an die Objektvariable `locStringArray` wird intern nur ein Zeiger umgebogen. Der Speicherbereich der alten Elemente ist nicht mehr referenzierbar.

WICHTIG Diese Tatsache hat Konsequenzen, denn: Anders, als es bei Visual Basic 6.0 zum Beispiel noch der Fall war, werden Arrays bei einer Zuweisung an eine andere Objektvariable *nicht* automatisch kopiert. Array-Variablen verhalten sich so wie jeder andere Referenztyp auch unter .NET: Ein Zuweisen einer Array-Variablen an eine andere biegt nur ihren Zeiger auf den Speicherbereich der eigentlichen Daten im Managed Heap um. Die Elemente bleiben an ihrem Platz im Managed Heap, und es wird kein neuer Speicherbereich mit einer Kopie der Array-Elemente für die neue Objektvariable erzeugt!

Das Redimensionieren kann nicht nur über Zuweisungen, sondern – wie im Beispielcode zu sehen – auch mit der `ReDim`-Anweisung erfolgen. Mit dem zusätzlichen Schlüsselwort `Preserve` haben Sie darüber hinaus die Möglichkeit zu bestimmen, dass die alten Elemente dabei erhalten bleiben. Man möchte meinen, dass diese Verhaltensweise die Regel sein sollte, doch mit dem Wissen im Hinterkopf, was beim Neudimensionieren mit `ReDim` genau passiert, sieht die Sache schon anders aus:

- Wird `ReDim` aufgerufen, wird ein komplett neuer Speicherbereich dafür reserviert.
- Der Zeiger für die Objektvariable auf den Bereich für die zuvor zugeordneten Array-Elemente wird auf den neuen Speicherbereich umgebogen.
- Der Speicherbereich, der die alten Array-Elementenzeiger enthielt, fällt dem nächsten Garbage-Collector-Durchlauf zum Opfer.
- Wenn `Preserve` hinter der `ReDim`-Anweisung platziert wird, bleiben die alten Array-Elemente erhalten. Doch das entspricht nicht der vollständigen Erklärung des Vorgangs. In Wirklichkeit wird auch hier ein neuer Speicherbereich erstellt, der den Platz für die neu angegebene Anzahl an Array-Elementen bereithält. Auch bei `Preserve` wird der Zeiger auf den Speicherbereich mit den alten Elementen für die betroffene Objektvariable auf den neuen Speicherbereich umgebogen. Doch bevor der alte Speicherbereich freigegeben wird und sich der Garbage Collector über die alten Elemente hermachen kann, werden die vorhandenen Elemente in den neuen Bereich kopiert.

Aus diesem Grund können Sie mit `Preserve` nur Elemente retten, die in eindimensionalen Arrays gespeichert sind oder die durch die letzte Dimension eines mehrdimensionalen Arrays angesprochen werden.

Werteverteilung von Array-Elementen im Code

Alle Arrays, die in den vorangegangenen Beispielen verwendet wurden, sind zur Laufzeit mit Daten gefüllt worden. In vielen Fällen möchten Sie aber Arrays erstellen, die Sie automatisch mit Daten vorbelegen, die das Programm fest vorgibt:

```
Sub Beispiel5()
    'Deklaration und Definition von Elementen mit Double-Werten
    Dim locDoubleArray As Double() = {123.45F, 5435.45F, 3.14159274F}

    'Deklaration und spätere Definition von Elementen mit Integer-Werten
    Dim locIntegerArray As Integer()
```

```

locIntegerArray = New Integer() {1I, 2I, 3I, 3I, 4I}
'Deklaration und spätere Definition von Elementen mit Date-Werten
Dim locDateArray As Date()
locDateArray = New Date() {#12/24/2005#, #12/31/2005#, #3/31/2006#}

'Deklaration und Definition von Elementen im Char-Array:
Dim locCharArray As Char() = {"V"c, "B"c, ".c, "N"c, "E"c, "T"c, " "c, _
                               "r"c, "u"c, "l"c, "e"c, "s"c, !"c}

'Zweidimensionales Array
Dim locZweiDimensional As Integer(,)
locZweiDimensional = New Integer(,) {{10, 10}, {20, 20}, {30, 30}}

'Oder verschachtelt (das ist nicht Zwei-Dimensional)!
Dim locVerschachtelt As Date(){}
locVerschachtelt = New Date()() {New Date() {#12/24/2004#, #12/31/2004#}, -
                                New Date() {#12/24/2005#, #12/31/2005#}}
End Sub

```

Häufigste Fehlerquelle bei dieser Vorgehensweise: Der Zuweisungsoperator wird falsch gesetzt. Bei der kombinierten Deklaration/Definition wird er benötigt; *definieren* Sie nur neu, lassen Sie ihn weg. Beachten Sie auch den Unterschied zwischen mehrdimensionalen und verschachtelten Arrays, auf den ich im nächsten Abschnitt genauer eingehen möchte.

Mehrdimensionale und verschachtelte Arrays

Bei der Definition von Arrays sind Sie nicht auf eine Dimension beschränkt – das ist ein Feature, das schon bei jahrzehntealten Basic-Dialekten zu finden ist. Sie können ein mehrdimensionales Array erstellen, indem Sie bei der Deklaration die Anzahl der Elemente für jede Dimension durch Komma getrennt angeben:

```
Dim DreiDimensional(5, 10, 3) As Integer
```

Möchten Sie die Anzahl der zu verwaltenden Elemente bei der Deklaration des Arrays nicht festlegen, verwenden Sie die folgende Deklarationsanweisung:

```
Dim AuchDreiDimensional As Integer(,,)
```

Mit ReDim oder dem Aufruf von Funktionen, die ein entsprechend dimensioniertes Array zurückliefern, können Sie anschließend das Array definieren.

Verschachtelte Arrays

Verschachtelte Arrays sind etwas anders konzipiert als mehrdimensionale Arrays. Bei verschachtelten Arrays ist ein Array-Element selbst ein Array (welches auch wieder Arrays beinhalten kann usw.). Anders als bei mehrdimensionalen Arrays können die einzelnen Elemente unterschiedlich dimensionierte Arrays enthalten und diese Zuordnung lässt sich auch im Nachhinein noch ändern.

Verschachtelte Arrays definieren Sie, indem Sie die Klammerpaare mit der entsprechenden Array-Dimension hintereinander schreiben – anders als bei mehrdimensionalen Arrays, bei denen die Dimensionen in einem Klammerpaar mit Komma getrennt angegeben werden.

Die folgenden Beispiel-Codezeilen (aus der Sub `Beispiel6`) zeigen, wie Sie verschachtelte Arrays definieren, deklarieren und ihre einzelnen Elemente abrufen können:

```
'Einfach verschachtelt; Tiefe wird nicht definiert.  
Dim EinfachVerschachtelt(10)() As Integer  
  
'Erstes Element hält ein Integer-Array mit drei Elementen.  
EinfachVerschachtelt(0) = New Integer() {10, 20, 30}  
  
'Zweites Element hält ein Integer-Array mit acht Elementen.  
EinfachVerschachtelt(1) = New Integer() {10, 20, 30, 40, 50, 60, 70, 80}  
  
'Drückt das dritte Element des zweiten Elementes (30) des Arrays.  
Console.WriteLine(EinfachVerschachtelt(1)(2))  
  
'In einem Rutsch alles neu zuweisen.  
EinfachVerschachtelt = New Integer()() {New Integer() {30, 20, 10},  
                                         New Integer() {80, 70, 60, 50, 40, 30, 20, 10}}  
  
'Drückt das dritte Element des zweiten Elementes (jetzt 60) des Arrays.  
Console.WriteLine(EinfachVerschachtelt(1)(2))  
Console.ReadLine()
```

Die wichtigsten Eigenschaften und Methoden von Arrays

In den vorangegangenen Beispielprogrammen ließ es sich nicht vermeiden, die eine oder andere Eigenschaft oder Methode des Array-Objektes bereits anzuwenden. Dieser Abschnitt soll sich ein wenig genauer mit den zusätzlichen Möglichkeiten dieses Objektes beschäftigen – denn sie sind mächtig und können Ihnen, richtig angewendet, eine Menge Entwicklungszeit ersparen.

Anzahl der Elemente eines Arrays ermitteln mit Length

Wenn Sie wissen wollen, wie viele Elemente ein Array beherbergt, bedienen Sie sich seiner `Length`-Eigenschaft. Bei zwei- und mehrdimensionalen Arrays ermittelt `Length` ebenfalls die Anzahl aller Elemente.

Aufgepasst bei verschachtelten Arrays: Hier ermittelt `Length` nämlich nur die Elementanzahl des umgebenden Arrays. Sie können die Array-Länge eines Elementes des umgebenden Arrays etwa so ermitteln:

```
'Verschachtelte Arrays  
Dim locVerschachtelt As Date()  
locVerschachtelt = New Date()() {New Date() {"#12/24/2004#", "#12/31/2004#"}, -  
                               New Date() {"#12/24/2005#", "#12/31/2005#"}}  
  
Console.WriteLine("Äußeres Array hat {0} Elemente.", locVerschachtelt.Length)  
Console.WriteLine("Array des 1. Elements hat {0} Elemente.", locVerschachtelt(0).Length)
```

Sortieren von Arrays mit `Array.Sort`

Arrays lassen sich durch eine ganz simple Methode sortieren. Das folgende Beispiel soll demonstrieren, wie es geht:

```
Sub Beispiel7()  
  
'Array-Erstellen:
```

```

Dim locNamen As String() = {"Jürgen", "Martina", "Hanna", "Gaby", "Michaela", "Miriam", "Ute", _
                            "Leonie-Gundula", "Melanie", "Uwe", "Andrea", "Klaus", "Anja", _
                            "Myriam", "Daja", "Thomas", "José", "Kricke", "Flori", "Katrín", "Momo", _
                            "Gareth", "Anne"}
System.Array.Sort(locNamen)
DruckeStrings(locNamen)
Console.ReadLine()
End Sub

```

Wenn Sie dieses Beispiel starten, produziert es folgendes Ergebnis im Konsolenfenster:

```

Andrea
Anja
Anne
Daja
Flori
Gaby
Gareth
Hanna
José
Jürgen
Katrín
Klaus
Kricke
Leonie-Gundula
Martina
Melanie
Michaela
Miriam
Momo
Myriam
Thomas
Ute
Uwe

```

Die Sort-Methode ist eine statische Methode von `System.Array` und sie kann noch eine ganze Menge mehr. So haben Sie beispielsweise die Möglichkeit, einen korrelierenden Index mitsortieren zu lassen, oder Sie können bestimmen, zwischen welchen Indizes eines Arrays die Sortierung stattfinden soll. Die Online-Hilfe zum Framework verrät Ihnen, welche Überladungen die Sort-Methode genau anbietet.

Umdrehen der Array-Anordnung mit `Array.Reverse`

Wichtig in diesem Zusammenhang ist eine weitere statische Methode von `System.Array` namens `Reverse`, die die Reihenfolge der einzelnen Array-Elemente umkehrt. Im Zusammenhang mit der Sort-Methode erreichen Sie durch den anschließenden Einsatz von `Reverse` die Sortierung eines Arrays in absteigender Reihenfolge. Wenn Sie das vorherige Beispiel um diese Zeilen

```

Console.WriteLine()
Console.WriteLine("Absteigend sortiert:")
Array.Reverse(locNamen)
DruckeStrings(locNamen)
Console.ReadLine()

```

ergänzen, sehen Sie schließlich die Namen in umgekehrter Reihenfolge im Konsolenfenster, etwa:

```
Absteigend sortiert:  
Uwe  
Ute  
Thomas  
. . .  
Anja  
Andrea
```

Durchsuchen eines sortierten Arrays mit Array.BinarySearch

Auch bei der Suche nach bestimmten Elementen eines Arrays ist Ihnen das Framework behilflich. Dazu stellt die Array-Klasse die statische Funktion `BinarySearch` zur Verfügung.

WICHTIG Damit eine binäre Suche in einem Array durchgeführt werden kann, muss das Array in sortierter Form vorliegen – ansonsten wird die Funktion höchstwahrscheinlich falsche Ergebnisse zurückliefern. Wirklich brauchbar ist die Funktion überdies nur dann, wenn Sie sicherstellen, dass es keine Dubletten in den Elementen gibt. Da eine binäre Suche erfolgt, ist nicht gewährleistet, ob die Funktion das erste zutreffende Objekt findet oder ein beliebiges, das dem Gesuchten entsprach.

Beispiel:

```
Sub Beispiel18()  
  
'Array-Erstellen:  
Dim locNamen As String() = {"Jürgen", "Martina", "Hanna", "Gaby", "Michaela", "Miriam", "Ute", _  
    "Leonie-Gundula", "Melanie", "Uwe", "Andrea", "Klaus", "Anja", _  
    "Myriam", "Daja", "Thomas", "José", "Kricke", "Flori", "Katrín", "Momo", _  
    "Gareth", "Anne", "Jürgen", "Gaby"}  
System.Array.Sort(locNamen)  
Console.WriteLine("Jürgen wurde gefunden an Position {0}", _  
    System.Array.BinarySearch(locNamen, "Jürgen"))  
Console.ReadLine()  
End Sub
```

Wie `Sort` ist auch `BinarySearch` eine überladene Funktion und bietet weitere Optionen, die die Suche beispielsweise auf bestimmte Indexbereiche beschränkt. IntelliSense und die Online-Hilfe geben hier genauere Hinweise für die Verwendung.

Implementierung von Sort und BinarySearch für eigene Klassen

Das Framework erlaubt es, Arrays von beliebigen Typen zu erstellen, auch von solchen, die Sie selbst erstellt haben. Bei der Erstellung einer Klasse, die als Element eines Arrays fungieren soll, brauchen Sie dabei nichts Besonderes zu beachten.

Anders wird das allerdings, wenn Sie eine Funktion auf das Array anwenden wollen, die einen Elementvergleich erfordert, oder wenn Sie sogar steuern wollen, nach welchen Kriterien Ihr Array beispielsweise sortiert werden soll oder auch nach welchem Kriterium Sie es mit `BinarySearch` durchsuchen lassen möchten. Dazu ein Beispiel:

Sie haben eine Klasse entwickelt, die die Adressen einer Kontaktdatenbank speichert. Der nachfolgend gezeigte Code demonstriert, wie sie funktioniert und wie man sie anwendet:

BEGLEITDATEIEN Im Folgenden sehen Sie zunächst den Klassencode, der einen Adresseneintrag verwaltet – dieses Projekt finden Sie im Verzeichnis

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 22\\IComparer01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Public Class Adresse

    Protected myName As String
    Protected myVorname As String
    Protected myPLZ As String
    Protected myOrt As String

    Sub New(ByVal Name As String, ByVal Vorname As String, ByVal Plz As String, ByVal Ort As String)
        myName = Name
        myVorname = Vorname
        myPLZ = Plz
        myOrt = Ort
    End Sub

    Public Property Name() As String
        Get
            Return myName
        End Get
        Set(ByVal Value As String)
            myName = Value
        End Set
    End Property

    Public Property Vorname() As String
        Get
            Return myVorname
        End Get
        Set(ByVal Value As String)
            myVorname = Value
        End Set
    End Property

    Public Property PLZ() As String
        Get
            Return myPLZ
        End Get
        Set(ByVal Value As String)
            myPLZ = Value
        End Set
    End Property

    Public Property Ort() As String
        Get
            Return myOrt
        End Get
        Set(ByVal Value As String)
```

```
    myOrt = Value
End Set
End Property

Public Overrides Function ToString() As String
    Return Name + ", " + Vorname + ", " + PLZ + " " + Ort
End Function
End Class
```

Sie sehen selbst: einfachstes Visual Basic. Die Klasse stellt ein paar Eigenschaften für die von ihr verwalteten Daten zur Verfügung, und sie überschreibt die `ToString`-Methode der Basis-Klasse, damit sie eine Adresse als kompletten String ausgeben kann.²

Das folgende kleine Beispielprogramm definiert ein Array aus dieser Klasse, richtet ein paar Adressen zum Experimentieren ein und druckt diese anschließend in einer eigenen Unterroutine aus:

```
Module ComparerBeispiel

Sub Main()
    Dim locAdressen(5) As Adresse

    locAdressen(0) = New Adresse("Löffelmann", "Klaus", "11111", "Soest")
    locAdressen(1) = New Adresse("Heckhuis", "Jürgen", "99999", "Gut Uhlenbusch")
    locAdressen(2) = New Adresse("Sonntag", "Miriam", "22222", "Dortmund")
    locAdressen(3) = New Adresse("Sonntag", "Christian", "33333", "Wuppertal")
    locAdressen(4) = New Adresse("Ademmer", "Uta", "55555", "Bad Waldholz")
    locAdressen(5) = New Adresse("Kaiser", "Wilhelm", "12121", "Ostenwesten")

    Console.WriteLine("Adressenliste:")
    Console.WriteLine(New String("=", 40))
    DruckeAdressen(locAdressen)
    'Array.Sort(locAdressen)
    Console.ReadLine()
End Sub

Sub DruckeAdressen(ByVal Adressen As Adresse())
    For Each locString As Adresse In Adressen
        Console.WriteLine(locString)
    Next
End Sub

End Module
```

Auch hier werden Sie erkennen: Es passiert nichts wirklich Aufregendes. Wenn Sie das Programm starten, produziert es, wie zu erwarten, folgende Ausgabe im Konsolenfenster:

```
Adressenliste:
=====
Löffelmann, Klaus, 11111 Soest
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Sonntag, Miriam, 22222 Dortmund
```

² Den Vorgang des Überschreibens konnten Sie bereits im Kapitel zu abstrakten Klassen und Polymorphie kennen lernen. Im Bedarfsfall führen Sie sich zunächst dieses Kapitel zu Gemüte.

```
Sonntag, Christian, 33333 Wuppertal
Ademmer, Uta, 55555 Bad Waldholz
Kaiser, Wilhelm, 12121 Ostenwesten
```

Die Liste, wie wir Sie anzeigen lassen, ist allerdings ein ziemliches Durcheinander. Beobachten Sie, was passiert, wenn wir das Programm um eine Sort-Anweisung ergänzen und wie die Liste anschließend ausschaut. Dazu nehmen Sie die Auskommentierung der Sort-Methode im Listing einfach zurück.

Nach dem Start des Programms warten Sie vergeblich auf die zweite, sortierte Ausgabe der Liste. Stattdessen löst das Framework eine Ausnahme aus, etwa wie in Abbildung 22.1 zu sehen.

Der Grund dafür: Die Sort-Methode der Array-Klasse versucht, die einzelnen Elemente des Arrays miteinander zu vergleichen. Dazu benötigt es einen so genannten *Comparer* (etwa: *Vergleicher*). Da wir nicht explizit angeben, dass wir einen speziellen Comparer verwenden wollen (Welchen auch? – Wir haben noch keinen!), erzeugt es einen Standard-Comparer, der aber wiederum die Einbindung einer bestimmten Schnittstelle in der Klasse verlangt, deren Elemente er miteinander vergleichen soll. Diese Schnittstelle nennt sich *IComparable*. Leider haben wir auch diese Schnittstelle nicht implementiert und die Ausnahme ist die Folge.

Wir haben nun drei Möglichkeiten. Wir binden die *IComparable*-Schnittstelle ein, dann können Elemente unserer Klasse auch ohne die Nennung eines expliziten Comparers verglichen und im Endeffekt sortiert werden.

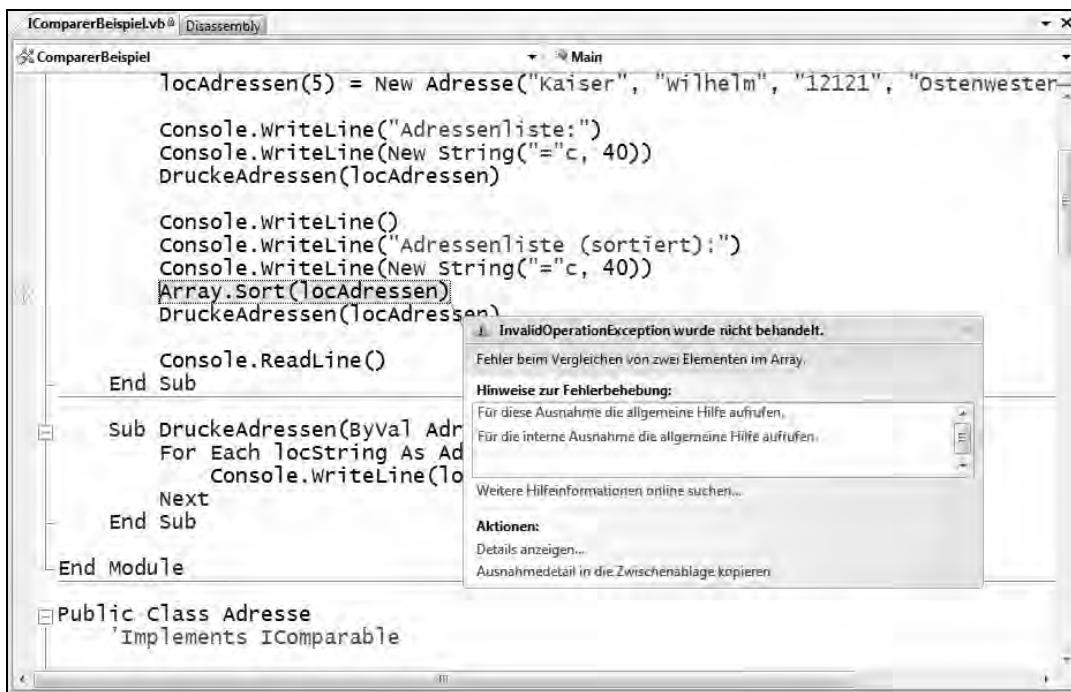


Abbildung 22.1 Wenn Sie das Programm starten, löst es eine Ausnahme aus, sobald die Sort-Methode der Array-Klasse erreicht ist – in der Detailbeschreibung zur Ausnahme sehen Sie, was falsch läuft!

Oder wir stellen der Klasse einen *expliziten* Comparer zur Verfügung; in diesem Fall müssen wir ihn beim Einsatz von Sort (oder auch BinarySearch) benennen. Dieser Comparer hätte den Vorteil, dass sich durch ihn steuern ließe, nach welchem Kriterium die Klasse durchsucht bzw. sortiert werden soll.

Die dritte Möglichkeit: Wir machen beides. Damit wird unsere Klasse universell nutzbar und läuft nicht Gefahr, eine Ausnahme auslösen zu können. Und genau das werden wir in den nächsten beiden Abschnitten in die Tat umsetzen.

Implementieren der Vergleichsfähigkeit einer Klasse durch IComparable

Die Implementierung der IComparable-Schnittstelle, damit Instanzen unserer Klasse miteinander verglichen werden können, ist vergleichsweise simpel.

Erinnern wir uns: Damit bestimmter Code allgemeingültig verwendbar sein kann, kann man sich des Konzeptes von Schnittstellen (engl. *Interfaces*) bedienen. Das Einbinden einer Schnittstelle in eine Klasse oder eine Struktur erzwingt, dass in der Klasse die Elemente (Eigenschaften, Methoden, Ereignisse) vorhanden sein müssen, die die Schnittstelle vorschreibt. Gleichzeitig reicht es aus, beispielsweise eine Methode, die eine Schnittstelle für eine Klasse vorschreibt, über eine Schnittstellenvariable aufzurufen. Damit spielt es für die aufrufende Instanz keine Rolle mehr, mit was sie es genau zu tun hat – ob es beispielsweise ein Integer oder ein Decimal-Datentyp ist. Die aufrufende Instanz arbeitet lediglich mit einer Schnittstellenvariablen, und ruft über diese die gewünschte Methode auf. Bei einem Integer-Datentyp, der diese Schnittstelle einbindet, werden so beispielsweise Integer-Werte verglichen, bei einem String-Datentyp, der die gleiche Schnittstelle einbindet, eben Zeichenketten. Doch das merkt und interessiert die aufrufende Instanz gar nicht (in diesem Fall die Array-Klasse des Frameworks). Sie ist nur am Ergebnis interessiert.

Das Implementieren der Schnittstelle in unserer Beispielklasse erfordert übrigens zusätzlich lediglich das Vorhandensein einer CompareTo-Methode, die die aktuelle Instanz der Klasse mit einer weiteren vergleicht, und das Anzeigen, dass es sich bei der Methode um die von der Schnittstelle vorgeschriebene Methode handelt.

Da durch den Einsatz von IComparable keine Möglichkeit besteht festzulegen, welches Datenfeld das Vergleichskriterium sein soll, wird unser Kriterium eine Zeichenkette sein, die aus Namen, Vornamen, Postleitzahl und Ort besteht – genau die Zeichenkette also, die ToString in der jetzigen Version bereits zurückliefernt. Deswegen brauchen wir den eigentlichen Vergleich noch nicht einmal selbst durchzuführen, sondern können ihn an die CompareTo-Funktion des String-Objektes, das wir von ToString zurück erhalten, weiterreichen. Die Modifizierungen an der Klasse sind also denkbar gering (aus Platzgründen nur gekürzter Code, der die Änderungen widerspiegelt):

```
Public Class Adresse
    Implements IComparable
    .
    .
    .
    Public Function CompareTo(ByVal obj As Object) As Integer Implements System.IComparable.CompareTo
        Dim locAdresse As Adresse
        Try
            locAdresse = DirectCast(obj, Adresse)
        Catch ex As InvalidCastException
            Dim up As New InvalidCastException("CompareTo' der Klasse 'Adresse' kann keine Vergleiche " +
                "mit Objekten anderen Typs durchführen!")
        End Try
    End Function
End Class
```

```

Throw up
End Try
Return ToString.CompareTo(locAdresse.ToString)
End Function
End Class

```

An dieser Stelle vielleicht erwähnenswert ist der fett gekennzeichnete mittlere Bereich im Programm. Auf den ersten Blick mag es unsinnig erscheinen, einen möglichen Fehler abzufangen und ihn anschließend mehr oder weniger unverändert wieder auszulösen. Aber: Der Fehlertext ist entscheidend, und Sie tun sich selbst einen Gefallen, wenn Sie den Fehlertext einer Ausnahme, die Sie generieren, so formulieren, dass Sie eindeutig wissen, wer oder was sie ausgelöst hat. Wenn Sie das Programm anschließend starten, liefert es das gewünschte Ergebnis, wie im Folgenden zu sehen:

```

Adressenliste:
=====
Löffelmann, Klaus, 11111 Soest
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Sonntag, Miriam, 22222 Dortmund
Sonntag, Christian, 33333 Wuppertal
Ademmer, Uta, 55555 Bad Waldholz
Kaiser, Wilhelm, 12121 Ostenwesten

Adressenliste (sortiert):
=====
Ademmer, Uta, 55555 Bad Waldholz
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Kaiser, Wilhelm, 12121 Ostenwesten
Löffelmann, Klaus, 11111 Soest
Sonntag, Christian, 33333 Wuppertal
Sonntag, Miriam, 22222 Dortmund

```

Implementieren einer gesteuerten Vergleichsfähigkeit durch IComparer

So weit, so gut – unser Beispielprogramm läuft immerhin schon, und die Elemente des Arrays lassen sich sortieren. Aber: Das Programm sortiert stumpf nach dem Nachnamen – und diese Verhaltensweise können wir ihm ohne weitere Maßnahmen auch nicht abgewöhnen.

Was die Adresse-Klasse braucht, ist eine Art Steuerungseinheit, die durch Setzen bestimmter Eigenschaften festlegt, wie Vergleiche stattfinden sollen. Wenn Sie sich zum Beispiel die Sort-Funktion von System.Array ein wenig genauer anschauen, werden Sie feststellen, dass sie als optionalen Parameter eine Variable vom Typ IComparer entgegennimmt.

Eine Klasse, die IComparer einbindet, macht genau das Verlangte. Sie kann den Vergleichsvorgang beeinflussen. Wenn Sie IComparer in einer Klasse implementieren, müssen Sie auch die Funktion Compare in dieser Klasse zur Verfügung stellen. Dieser Funktion werden zwei Objekte übergeben, die die Funktion miteinander vergleichen soll. Ist eines der Objekte größer (was auch immer das heißt, denn es ist bis dahin ein abstraktes Attribut), liefert sie den Wert 1, ist es kleiner, den Wert -1, und ist es gleich, liefert sie den Wert 0 zurück.

Ein Comparer steht in keinem direkten Verhältnis zu den Klassen, die er vergleichen soll; er wird also nicht durch weitere Schnittstellen reglementiert. Aus diesem Grund muss der Comparer selber dafür Sorge tragen, dass ihm nur die Objekte zum Vergleichen angeliefert werden, die er verarbeiten will. Bekommt er andere, muss er eine Ausnahme auslösen.

In unserem Fall muss der Comparer noch ein bisschen mehr können. Er muss die Funktionalität zur Verfügung stellen, durch die der Entwickler steuern kann, nach *welchem* Kriterium der *Adresse*-Klasse er vergleichen will. Aus diesen Gründen ergibt sich folgender Code für einen Comparer unseres Beispiels:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 22\\IComparer02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
'Nur für den einfachen Umgang mit der Klasse.
Public Enum ZuVergleichen
    Name
    PLZ
    Ort
End Enum

Public Class AdressenVergleicher
    Implements IComparer

    'Speichert die Einstellung, nach welchem Kriterium verglichen wird.
    Protected myZuVergleichenMit As ZuVergleichen

    Sub New(ByVal ZuVergleichenMit As ZuVergleichen)
        myZuVergleichenMit = ZuVergleichenMit
    End Sub

    Public Function Compare(ByVal x As Object, ByVal y As Object) As Integer _
        Implements System.Collections.IComparer.Compare
        'Nur erlaubte Typen durchlassen:
        If (Not (TypeOf x Is Adresse)) Or (Not (TypeOf y Is Adresse)) Then
            Dim up As New InvalidCastException("Compare' der Klasse 'Adressenvergleicher' kann nur " +
                "Vergleiche vom Typ 'Adresse' durchführen!")
            Throw up
        End If

        'Beide Objekte in den richtigen Typ casten, damit das Handling einfacher wird:
        Dim locAdr1 As Adresse = DirectCast(x, Adresse)
        Dim locAdr2 As Adresse = DirectCast(y, Adresse)

        'Hier passiert die eigentliche Steuerung,
        'nach welchem Kriterium verglichen wird:
        If myZuVergleichenMit = ZuVergleichen.Name Then
            Return locAdr1.Name.CompareTo(locAdr2.Name)
        ElseIf myZuVergleichenMit = ZuVergleichen.Ort Then
            Return locAdr1.Ort.CompareTo(locAdr2.Ort)
        Else
            Return locAdr1.PLZ.CompareTo(locAdr2.PLZ)
        End If
    End Function

    'Legt die Vergleichseinstellung offen.
    Public Property ZuVergleichenMit() As ZuVergleichen
        Get
```

```

        Return myZuVergleichenMit
    End Get
    Set(ByVal Value As ZuVergleichen)
        myZuVergleichenMit = Value
    End Set
End Property

End Class

```

Zum Beweis, dass alles wie gewünscht läuft, ergänzen Sie das Hauptprogramm um folgende Zeilen (fettgedruckt im folgenden Listing):

```

Module ComparerBeispiel

Sub Main()
    Dim locAdressen(5) As Adresse

    locAdressen(0) = New Adresse("Löffelmann", "Klaus", "11111", "Soest")
    locAdressen(1) = New Adresse("Heckhuis", "Jürgen", "99999", "Gut Uhlenbusch")
    locAdressen(2) = New Adresse("Sonntag", "Miriam", "22222", "Dortmund")
    locAdressen(3) = New Adresse("Sonntag", "Christian", "33333", "Wuppertal")
    locAdressen(4) = New Adresse("Ademmer", "Uta", "55555", "Bad Waldholz")
    locAdressen(5) = New Adresse("Kaiser", "Wilhelm", "12121", "Ostenwesten")

    Console.WriteLine("Adressenliste:")
    Console.WriteLine(New String("=", 40))
    DruckeAdressen(locAdressen)

    Console.WriteLine()
Console.WriteLine("Adressenliste (sortiert nach Postleitzahl):")
    Console.WriteLine(New String("=", 40))
    Array.Sort(locAdressen, New AdressenVergleicher(ZuVergleichen.PLZ))
    DruckeAdressen(locAdressen)
    Console.ReadLine()
End Sub
.
.
.
End Module

```

Wenn Sie das Programm nun starten, erhalten Sie folgende Ausgabe auf dem Bildschirm:

```

Adressenliste:
=====
Löffelmann, Klaus, 11111 Soest
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Sonntag, Miriam, 22222 Dortmund
Sonntag, Christian, 33333 Wuppertal
Ademmer, Uta, 55555 Bad Waldholz
Kaiser, Wilhelm, 12121 Ostenwesten

Adressenliste (sortiert nach Postleitzahl):
=====
Löffelmann, Klaus, 11111 Soest
Kaiser, Wilhelm, 12121 Ostenwesten
Sonntag, Miriam, 22222 Dortmund

```

```
Sonntag, Christian, 33333 Wuppertal  
Ademmer, Uta, 55555 Bad Waldholz  
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
```

Enumeratoren

Wenn Sie Arrays oder – wie Sie später sehen werden – Auflistungen für die Speicherung von Daten verwenden, dann müssen Sie in der Lage sein, diese Daten abzurufen. Mit *Indexern* gibt Ihnen das .NET Framework eine einfache – die einfachste – Möglichkeit: Sie verwenden eine Objektvariable und versehen sie mit einem Index, der auch durch eine andere Variable repräsentiert werden kann. Ändern Sie diese Variable, die als Index dient, können Sie dadurch programmtechnisch bestimmen, welches Element eines Arrays Sie gerade verarbeiten wollen. Typische Zählschleifen, mit denen Sie durch die Elemente eines Arrays iterieren, sind die Folge – etwa im folgenden Stil:

```
For count As Integer = 0 To Array.Length - 1  
    TuWasMit(Array(count))  
Next
```

HINWEIS Enumeratoren haben nichts mit Enums zu tun. Lediglich die Namen sind sich etwas ähnlich. Enums sind aufgezählte Benennungen von bestimmten Werten im Programmlisting, während Enumeratoren die Unterstützung von For/Each zur Verfügung stellen!

Wenn ein Objekt allerdings die Schnittstelle `IEnumerable` implementiert, gibt es eine elegantere Methode, die verschiedenen Elemente, die das Objekt zur Verfügung stellt, zu durchlaufen. Glücklicherweise implementiert `System.Array` die Schnittstelle `IEnumerable`, sodass Sie auf Array-Elemente mit dieser eleganten Methode – namentlich mit For/Each – zugreifen können. Ein Beispiel:

```
'Deklaration und Definition von Elementen im Char-Array  
Dim locCharArray As Char() = {"V"c, "B"c, ".c, "N"c, "E"c, "T"c, " "c, _  
    "r"c, "u"c, "l"c, "e"c, "s"c, !"c}  
For Each c As Char In locCharArray  
    Console.Write(c)  
Next  
Console.WriteLine()  
Console.ReadLine()
```

Wenn Sie dieses Beispiel laufen lassen, sehen Sie im Konsolenfenster den folgenden Text:

VB.NET rules!

Enumeratoren werden von allen typdefinierten Arrays unterstützt und ebenso von den meisten Auflistungen. Enumeratoren können Sie aber auch in Ihren eigenen Klassen einsetzen, wenn Sie erlauben möchten, dass der Entwickler, der mit Ihrer Klasse arbeitet, durch Elemente mit For/Each iterieren kann.

WICHTIG Enumeratoren sind lebenswichtig für LINQ (siehe ab Kapitel 31). Nur eine Auflistungsklasse, die `IEnumerable(Of T)` implementiert, kann als Abfrageausdruck in LINQ verwendet werden. Denken Sie daran, wenn Sie eigene Auflistungsklassen entwerfen.

Benutzerdefinierte Enumeratoren durch Implementieren von `IEnumerable`

Enumeratoren können allerdings nicht nur für die Aufzählung von gespeicherten Elementen in Arrays oder Auflistungen eingesetzt werden. Sie können Enumeratoren auch dann einsetzen, wenn eine Klasse ihre Enumerations-Elemente durch Algorithmen zurückliefert.

Als Beispiel dafür möchte ich Ihnen zunächst einen Codeausschnitt zeigen, der nicht funktioniert – bei dem es aber in einigen Fällen wünschenswert wäre, wenn er funktionierte:

```
'Das würde nicht funktionieren:
for d as Date=#24/12/2004# to #31/12/2004#
    Console.WriteLine("Datum in Aufzählung: {0}", d)
Next d
```

Dennoch könnte es für bestimmte Anwendungen sinnvoll sein, tageweise einen bestimmten Datumsbereich zu durchlaufen. Etwa wenn Ihre Anwendung herausfinden muss, wie viele Mitarbeiter, deren Daten Ihre Anwendung speichert, in einem bestimmten Monat Geburtstag haben.

Wenn das, was wir vorhaben, nicht mit `For/Next` funktioniert, vielleicht können wir dann aber eine Klasse schaffen, die einen Enumerator zur Verfügung stellt, sodass das Vorhaben mit `For/Each` gelingt. Diese Klasse sollte beim Instanziieren Parameter übernehmen, mit denen wir bestimmen können, welcher Datumsbereich in welcher Schrittweite durchlaufen werden soll. Damit Sie mit `For/Each` durch die Elemente einer Klasse iterieren können, muss die Klasse die Schnittstelle `IEnumerable` einbinden.

Das kann sie nur, wenn sie gleichzeitig eine Funktion `GetEnumerator` zur Verfügung stellt, die erst das Objekt mit dem eigentlichen Enumerator liefert. Doch eines nach dem anderen.

BEGLEITDATEIEN Schauen wir uns zunächst die Basisklasse an, die die Grundfunktionalität zur Verfügung stellt – Sie finden das Projekt im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 22\\Enumerators

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei); starten Sie das Programm mit `Strg+F5`.

```
Public Class Datumsaufzählung
    Implements IEnumerable
    Dim locDatumsaufzähler As Datumsaufzähler

    Sub New(ByVal StartDatum As Date, ByVal EndDatum As Date, ByVal Schrittweite As TimeSpan)
        locDatumsaufzähler = New Datumsaufzähler(StartDatum, EndDatum, Schrittweite)
    End Sub
    Public Function GetEnumerator() As System.Collections.IEnumerator _
        Implements System.Collections.IEnumerable.GetEnumerator
        Return locDatumsaufzähler
    End Function
End Class
```

Sie sehen, dass diese Klasse selbst nichts Großartiges macht – sie schafft durch die ihr übergebenen Parameter lediglich die Rahmenbedingungen und stellt die von `IEnumerable` verlangte Funktion `GetEnumerator` zur

Verfügung. Die eigentliche Aufgabe wird von der Klasse `Datumsaufzähler` erledigt, die auch als Rückgabewert von `GetEnumerator` zurückgegeben wird.

Eine Instanz dieser Klasse wird bei der Instanziierung von `Datumsaufzählung` erstellt. Was in dieser Klasse genau passiert, zeigt der folgende Code:

```
Public Class Datumsaufzähler
    Implements IEnumerable

    Private myStartDatum As Date
    Private myEndDatum As Date
    Private mySchrittweite As TimeSpan
    Private myAktuellesDatum As Date

    Sub New(ByVal StartDatum As Date, ByVal EndDatum As Date, ByVal Schrittweite As TimeSpan)
        myStartDatum = StartDatum
        myAktuellesDatum = StartDatum
        myEndDatum = EndDatum
        mySchrittweite = Schrittweite
    End Sub

    Public Property StartDatum() As Date
        Get
            Return myStartDatum
        End Get
        Set(ByVal Value As Date)
            myStartDatum = Value
        End Set
    End Property

    Public Property EndDatum() As Date
        Get
            Return myEndDatum
        End Get
        Set(ByVal Value As Date)
            myEndDatum = Value
        End Set
    End Property

    Public Property Schrittweite() As TimeSpan
        Get
            Return mySchrittweite
        End Get
        Set(ByVal Value As TimeSpan)
            mySchrittweite = Value
        End Set
    End Property

    Public ReadOnly Property Current() As Object Implements System.Collections.IEnumerable.Current
        Get
            Return myAktuellesDatum
        End Get
    End Property

    Public Function MoveNext() As Boolean Implements System.Collections.IEnumerable.MoveNext
        myAktuellesDatum = myAktuellesDatum.Add(Schrittweite)
        If myAktuellesDatum > myEndDatum Then
            Return False
        End If
    End Function
```

```

    Else
        Return True
    End If
End Function

Public Sub Reset() Implements System.Collections.IEnumerator.Reset
    myAktuellesDatum = myStartDatum
End Sub
End Class

```

Der Konstruktor und die beiden Eigenschaftsprozeduren sollen hier nicht so sehr interessieren. Vielmehr von Interesse ist, dass diese Klasse eine weitere Schnittstelle namens `IEnumerator` einbindet, und sie stellt die eigentliche Aufzählungsfunktionalität zur Verfügung. Sie muss dazu die Eigenschaft `Current`, die Funktion `MoveNext` sowie die Methode `Reset` implementieren.

Wenn eine `For/Each`-Schleife durchlaufen wird, dann wird das aktuell bearbeitete Objekt der Klasse durch die `Current`-Eigenschaft des eigentlichen `Enumerators` ermittelt. Anschließend zeigt `For/Each` mit dem Aufruf der Funktion `MoveNext` dem `Enumerator` an, dass es auf das nächste Objekt zugreifen möchte. Erst wenn `MoveNext` mit `False` als Rückgabewert anzeigt, dass es keine weiteren Objekte mehr zur Verfügung stellen kann (oder will), ist die umgebende `For/Each`-Schleife beendet.

In unserem Beispiel müssen wir bei `MoveNext` nur dafür sorgen, dass unsere interne Datums-Zähl-Variable um den Wert erhöht wird, den wir bei ihrer Instanziierung als Schrittweite bestimmt haben. Hat die Addition der Schrittweite auf diesen Datumszähler den Endwert noch nicht überschritten, liefern wir `True` als Funktionsergebnis zurück – `For/Each` darf mit seiner Arbeit fortfahren. Ist das Datum allerdings größer als der Datums-Endwert, wird die Schleife abgebrochen – der Vorgang wird beendet.

Das Programm, das von dieser Klasse Gebrauch machen kann, lässt sich nun sehr elegant einsetzen, wie die folgenden Beispielcodezeilen zeigen:

```

Module Enumerators
    Sub Main()
        Dim locDatumsaufzählung As New Datumsaufzählung(#12/24/2004#, _
            #12/31/2004#, _
            New TimeSpan(1, 0, 0, 0))

        For Each d As Date In locDatumsaufzählung
            Console.WriteLine("Datum in Aufzählung: {0}", d)
        Next

    End Sub
End Module

```

Das Ergebnis sehen Sie anschließend in Form einer Datumsfolge im Konsolenfenster.

Grundsätzliches zu Auflistungen (Collections)

Arrays haben im .NET Framework einen entscheidenden Nachteil. Sie können zwar dynamisch zur Laufzeit vergrößert oder verkleinert werden, aber der Programmieraufwand dazu ist doch eigentlich recht aufwändig. Wenn Sie sich schon früher mit Visual Basic beschäftigt haben, dann ist Ihnen »Auflistung« sicherlich ein Begriff.

Auflistungen erlauben es dem Entwickler, Elemente genau wie Arrays zu verwalten. Im Unterschied zu Arrays wachsen Auflistungen jedoch mit Ihren Speicherbedürfnissen.

Damit ist aber auch klar, dass das Indizieren mit Nummern zum Abrufen der Elemente nur bedingt funktionieren kann. Wenn ein Array 20 Elemente hat und Sie möchten das 21. Element hinzufügen, dann können Sie das dem Array nicht einfach so mitteilen. Ganz anders bei Auflistungen: Hier fügen Sie ein Element mit der Add-Methode hinzu.

Intern werden (fast alle) Auflistungen ebenfalls wie (oder besser: als) Arrays verwaltet. Rufen Sie eine neue Auflistung ins Leben, dann hat dieses Array, wenn nichts anderes gesagt wird, eine Größe von 16 Elementen. Wenn die Auflistungsklasse später, sobald Ihr Programm richtig »in Action« ist, »merkt«, dass ihr die Puste mengentechnisch ausgeht, dann legt sie ein Array nunmehr mit 32 Elementen an, kopiert die vorhandenen Elemente in das neue Array, arbeitet fortan mit dem neuen Array und tut ansonsten so, als wäre nichts gewesen.

Dieser anfängliche Load-Faktor erhöht sich bei jedem Neuanlegen des Arrays, um die Kopiervorgänge zu minimieren. Man geht einfach davon aus, dass, wenn der anfängliche Load-Faktor von 16 Elementen nicht ausreicht, 32 beim nächsten Mal auch zu wenig sind – und gerade in unserem Beispiel ist das ja auch richtig. So sind es beim zweiten Mal bereits 32 Elemente die dazukommen, beim nächsten Mal 64 usw. bis der maximale Load-Faktor mit 2048 Elementen erreicht ist.

In etwa entspricht also die Grundfunktionsweise einer Auflistung stark vereinfacht der folgenden Klasse:

```
Class DynamicList
    Implements IEnumerable

    Protected myStepIncreaser As Integer
    Protected myCurrentArraySize As Integer
    Protected myCurrentCounter As Integer
    Protected myArray() As Object

    Sub New()
        MyClass.New(16)
    End Sub

    Sub New(ByVal StepIncreaser As Integer)
        myStepIncreaser = StepIncreaser
        myCurrentArraySize = myStepIncreaser
        ReDim myArray(myCurrentArraySize)
    End Sub

    Sub Add(ByVal Item As Object)

        'Prüfen, ob aktuelle Arraygrenze erreicht wurde
        If myCurrentCounter = myCurrentArraySize - 1 Then
            'Neues Array mit mehr Speicher anlegen,
            'und Elemente hinüberkopieren. Dazu:

            'Neues Array wird größer:
            myCurrentArraySize += myStepIncreaser

            'temporäres Array erstellen
            Dim locTempArray(myCurrentArraySize - 1) As Object
            For i = 0 To myCurrentCounter
                locTempArray(i) = myArray(i)
            Next i
            myArray = locTempArray
        End If
        myCurrentCounter = myCurrentCounter + 1
    End Sub
```

```
'Elemente kopieren
'Wichtig: Um das Kopieren müssen Sie sich,
'anders als bei VB6, selber kümmern!
Array.Copy(myArray, locTempArray, myArray.Length)

'temporäres Array dem Memberarray zuweisen
myArray = locTempArray

'Beim nächsten Mal werden mehr Elemente reserviert!
myStepIncreaser *= 2
End If

'Element im Array speichern
myArray(myCurrentCounter) = Item

'Zeiger auf nächstes Element erhöhen
myCurrentCounter += 1

End Sub

'Liefert die Anzahl der vorhandenen Elemente zurück
Public Overridable ReadOnly Property Count() As Integer
    Get
        Return myCurrentCounter
    End Get
End Property

'Erlaubt das Zuweisen und Abfragen
Default Public Overridable Property Item(ByVal Index As Integer) As Object
    Get
        Return myArray(Index)
    End Get

    Set(ByVal Value As Object)
        myArray(Index) = Value
    End Set
End Property

'Liefert den Enumerator der Basis (dem Array) zurück
Public Function GetEnumerator() As System.Collections.IEnumerator Implements
System.Collections.IEnumerable.GetEnumerator
    Return myArray.GetEnumerator
End Function
End Class
```

Nun könnte man meinen, diese Vorgehensweise könnte sich zu einer Leistungsproblematik entwickeln, da alle paar Elemente der komplette Array-Inhalt kopiert werden muss.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 22\\DynamicList

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wie lange, glauben Sie, dauert es, ein Array mit 200.000 Zufallszahlen (mit Nachkommastellen) auf diese Weise anzulegen? 3 Sekunden? 2 Sekunden? Finden Sie es selbst heraus, indem Sie das Programm starten:

```
Anlegen von 200000 zufälligen Double-Elementen...
...in 21 Millisekunden!
```

Gerade mal 21 Millisekunden benötigt das Programm für diese Operation³ – beeindruckend, wie schnell Visual Basic dieser Tage ist, finden Sie nicht?

Nun muss ich Ihnen leider verraten: Die Klasse `DynamicList` werden Sie nie benötigen. Bestandteil des Frameworks ist nämlich eine Klasse, die das schon kann. Sogar noch ein kleines bisschen schneller. Und: Sie hat zusätzlich noch andere Möglichkeiten, die Ihnen die selbst gestrickte Klasse nicht bietet.

Im Beispielprojekt finden Sie eine Sub `Beispiel2`, die Ihnen die gleiche Prozedur mit der Klasse `ArrayList` demonstriert:

```
Sub Beispiel2()
    Dim locZeitmesser As New HighSpeedTimeGauge
    Dim locAnzahlElemente As Integer = 200000
    Dim locDynamicList As New ArrayList
    Dim locRandom As New Random(Now.Millisecond)

    Console.WriteLine("Anlegen von {0} zufälligen Double-Elementen...", locAnzahlElemente)
    locZeitmesser.Start()
    For count As Integer = 1 To locAnzahlElemente
        locDynamicList.Add(locRandom.NextDouble * locRandom.Next)
    Next
    locZeitmesser.Stop()
    Console.WriteLine("...in {0:#,##0} Millisekunden!", locZeitmesser.DurationInMilliseconds)
    Console.ReadLine()

End Sub
```

Dessen Geschwindigkeit ist auch nicht von schlechten Eltern:

```
Anlegen von 200000 zufälligen Double-Elementen...
...in 19 Millisekunden!
```

Im Prinzip arbeitet `ArrayList` nach dem gleichen Verfahren, das Sie in `DynamicList` kennengelernt haben. `ArrayList` verfährt auch mit dem gleichen Trick, um möglichst viel Leistung herauszuholen. Es verdoppelt die jeweils nächste Größe des neuen Arrays im Vergleich zu der vorherigen Größe des Arrays. Damit reduziert

³ Auf einem Intel Core 2 Quad Q6600 übrigens. Lesern des Vorgängerbuchs *Visual Studio 2005 – das Entwicklerbuch* (wie mehrfach schon erwähnt unter www.active-develop.de kostenlos herunterladbar!) wird auffallen, dass das Ergebnis sogar eine Millisekunde langsamer ist als auf dem Rechner, auf dem ich vor 3 Jahren das Visual Basic 2005-Buch schrieb. Sehr schön ist dabei zu sehen, dass neuere Rechner ihre Leistung kaum noch »vertikal«, über die Taktfrequenz, sondern nur noch horizontal über mehrere Prozessoren skalieren können – beim Laufenlassen dieses Beispiels wird nur einer von vier Prozessorkernen genutzt – $\frac{1}{4}$ des Prozessors liegen ungenutzt (schlimmer: in diesem Beispiel sogar unnutzbar!) brach. Um nicht nur alles, sondern überhaupt mehr an Leistung aus modernen Prozessoren herauszukitzeln ist daher Multithreading-Entwicklung sehr entscheidend und sollte bei Programmportierungen – beispielsweise wenn sie ältere Borland-, MFC- oder VB6-Anwendungen auf .NET portieren wollen oder müssen – *unbedingt* berücksichtigt werden!

sich der Gesamtaufwand des Kopierens erheblich. Da die Methode aber Bestandteil des Frameworks ist, muss sie nicht zur Laufzeit »geJITted« werden, was der Geschwindigkeit zusätzlich zugute kommt.

Im Übrigen werden die Daten der einzelnen Elemente ja nicht wirklich bewegt. Lediglich die Zeiger auf die Daten werden kopiert – vorhandene Elemente bleiben im Managed Heap an ihrem Platz. Kapitel 13 erklärte Ihnen im Abschnitt »New oder nicht New – Wieso es sich bei Objekten um Verweistypen handelt « übrigens mehr zu diesem Thema.

Wichtige Auflistungen der Base Class Library

Die BCL des Frameworks enthält eine ganze Reihe von Auflistungs-Typen, von denen Sie einen der wichtigsten – `ArrayList` – schon im Einsatz gesehen haben. In diesem Abschnitt möchte ich Ihnen die wichtigsten dieser Auflistungen kurz vorstellen und darauf hinweisen, für welchen Einsatz sie am besten geeignet sind oder welche Besonderheiten Sie bei ihrem Gebrauch beachten sollten. Für eine genauere Beschreibung ihrer Eigenschaften und Methoden verwenden Sie bitte die Online-Hilfe von Visual Studio.

ArrayList – universelle Ablage für Objekte

`ArrayList` können Sie als Container für Objekte aller Art verwenden. Sie instanzieren ein `ArrayList`-Objekt und weisen ihm mithilfe seiner `Add`-Funktion das jeweils nächste Element zu. Mit der `Default`-Eigenschaft `Item` können Sie schon vorhandene Elemente abrufen oder neu definieren. `AddRange` erlaubt Ihnen, die Elemente einer vorhandenen `ArrayList` einer anderen `ArrayList` hinzuzufügen.

Mit der `Count`-Eigenschaft eines `ArrayList`-Objektes finden Sie heraus, wie viele Elemente es beherbergt.

`Clear` löscht alle Elemente einer `ArrayList`. Mit `Remove` löschen Sie ein Objekt aus der `ArrayList`, das Sie als Parameter übergeben. Wenn mehrere gleiche Objekte (die `Equals`-Methode jedes Objektes wird dabei verwendet) existieren, wird das erste gefundene Objekt gelöscht. Mit `RemoveAt` löschen Sie ein Element an einer bestimmten Position. `RemoveRange` erlaubt Ihnen schließlich, einen ganzen Bereich von `ArrayList`-Elementen ab einer bestimmten Position im `ArrayList`-Objekt zu löschen.

`ArrayList`-Objekte können in einfache Arrays umgewandelt werden. Dabei ist jedoch einiges zu beachten: Die Elemente der `ArrayList` müssen ausnahmslos alle dem Typ des Arrays entsprechen, in den sie umgewandelt werden sollen. Die Konvertierung nehmen Sie mit der `ToArray`-Methode des entsprechenden `ArrayList`-Objektes vor. Dabei bestimmen Sie, wenn Sie in ein typdefiniertes Array (wie `Integer()` oder `String()`) umwandeln, den Grundtyp (nicht den Arraytyp!) als zusätzlichen Parameter. Wenn Sie ein Array in eine `ArrayList` umwandeln wollen, verwenden Sie den entsprechenden Konstruktor der `ArrayList` – die entsprechende Konstruktorroutine nimmt anschließend die Konvertierung in eine `ArrayList` vor.

HINWEIS Bitte schauen Sie sich dazu auch das weiter unten gezeigte Listing an, insbesondere was die Konvertierungshinweise von `ArrayList`-Objekten in Arrays betrifft.

`ArrayList` implementiert die Schnittstelle `IEnumerable`. Aus diesem Grund stellt die Klasse einen Enumerator zur Verfügung, mit dem Sie mithilfe von `For/Each` durch die Elemente der `ArrayList` iterieren können. Beachten Sie dabei, den richtigen Typ für die Schleifenvariable zu verwenden. `ArrayList`-Elemente sind nicht typsicher, und eine Typ-Verletzung ist nur dann ausgeschlossen, wenn Sie genau wissen, welche Typen gespeichert sind (das nachfolgende Beispiel demonstriert diesen Fehler recht anschaulich):

Hier die Beispiele, die das gerade Gesagte näher erläutern und den Code dazu dokumentieren:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 22\\CollectionsDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Sub ArrayListDemo()
    Dim locMännerNamen As String() = {"Jürgen", "Uwe", "Klaus", "Christian", "José"}
    Dim locFrauenNamen As New ArrayList
    Dim locNamen As ArrayList

    'ArrayList aus vorhandenem Array erstellen.
    locNamen = New ArrayList(locMännerNamen)

    'ArrayList mit Add auffüllen.
    locFrauenNamen.Add("Ute") : locFrauenNamen.Add("Miriam")
    locFrauenNamen.Add("Melanie") : locFrauenNamen.Add("Anja")
    locFrauenNamen.Add("Stephanie") : locFrauenNamen.Add("Heidrun")

    'ArrayList einer anderen ArrayList hinzufügen:
    locNamen.AddRange(locFrauenNamen)

    'ArrayList in eine ArrayList einfügen.
    Dim locHundenamen As String() = {"Hasso", "Bello", "Wauzi", "Wuffi", "Basko", "Franz"}
    'Einfügen *vor* dem 6. Element
    locNamen.InsertRange(5, locHundenamen)

    'ArrayList in ein Array zurückwandeln.
    Dim locAlleNamen As String()

    'Vorsicht: Fehler!
    'locAlleNamen = DirectCast(locNamen.ToArray, String())

    'Vorsicht: Ebenfalls Fehler!
    'locAlleNamen = DirectCast(locNamen.ToArray(GetType(String)), String())

    'So ist es richtig.
    locAlleNamen = DirectCast(locNamen.ToArray(GetType(String)), String())

    'Repeat legt eine ArrayList aus wiederholten Items an.
    locNamen.AddRange(ArrayList.Repeat("DubletteName", 10))

    'Ein Element im Array ändern.
    locNamen(10) = "Fiffi"
    'Mit der Item-Eigenschaft geht es auch:
    locNamen.Item(13) = "Miriam"

    'Löschen des ersten zutreffenden Elementes aus der Liste.
    locNamen.Remove("Basko")

    'Löschen eines Elementes an einer bestimmten Position.
    locNamen.RemoveAt(4)
```

```

'Löschen eines bestimmten Bereichs aus der ArrayList mit RemoveRange.
'Count ermittelt die Anzahl der Elemente in der ArrayList.
locNamen.RemoveRange(locNamen.Count - 6, 5)

'Ausgeben der Elemente über die Default-Eigenschaft der ArrayList (Item).
For i As Integer = 0 To locNamen.Count - 1
    Console.WriteLine("Der Name Nr. {0} lautet {1}", i, locNamen(i).ToString)
Next

'Anderes als ein String-Objekt der ArrayList hinzufügen,
'um den folgenden Fehler "vorzubereiten".
locNamen.Add(New FileInfo("C:\TEST.TXT"))

'Diese Schleife kann nicht bis zum Ende ausgeführt werden,
'da ein Objekt nicht vom Typ String mit von der Partie ist!
For Each einString As String In locNamen
    'Hier passiert irgendetwas mit dem String.
    'nicht von Interesse, deswegen kein Rückgabewert.
    einString.EndsWith("Peter")
Next
Console.ReadLine()
End Sub

```

Wenn Sie dieses Beispiel laufen lassen, dann sehen Sie zunächst die erwarteten Ausgaben auf dem Bildschirm. Doch der Programmcode erreicht nie die Anweisung `Console.ReadLine`, um auf Ihre letzte Bestätigung zu warten. Stattdessen löst er eine Ausnahme aus, etwa wie in Abbildung 22.2 zu sehen.

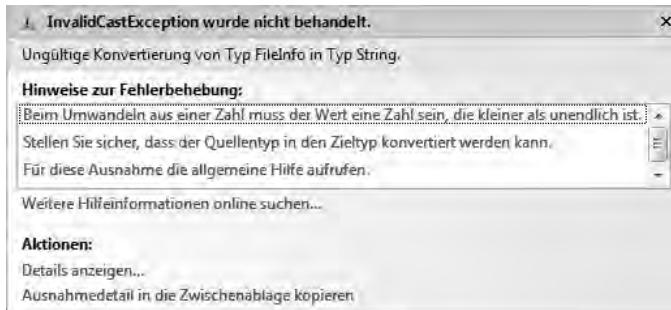


Abbildung 22.2 Achten Sie beim Iterieren mit `For/Each` durch eine Auflistung darauf, dass die Elemente dem SchleifenvariablenTyp entsprechen, um solche Ausnahmen zu vermeiden!

Der Grund dafür: Das letzte Element in der `ArrayList` ist kein `String`. Aus diesem Grund wird die Ausnahme ausgelöst. Achten Sie deshalb stets darauf, dass die Schleifenvariable eines `For/Each`-Konstruktks immer den Typen entspricht, die in einer Auflistung gespeichert sind.

Typsichere Auflistungen auf Basis von `CollectionBase`

Sie können dem Problem aus dem vorherigen Abschnitt entgehen, indem Sie eine Auflistung mithilfe von *Generics* (*Generika*, auf Microsoft-Deutsch) dazu zwingen, homogen zu sein (also nur Elemente des gleichen Typs zu verarbeiten). Für den späteren Einsatz von *LINQ to Objects* (siehe Kapitel 32) ist das sogar eine Notwendigkeit. Es gibt aber eine andere, klassische Möglichkeit, die eine Technologie unter Zuhilfenahme von Schnittstellen nutzt, um eine Basisklasse zu implementieren, mit der Sie ihre Ableitungen typsicher

machen, und die eben nicht auf Generics basieren. Diese Technik können Sie immer dann einsetzen, wenn Sie aus welchen Gründen auch immer auf Generics verzichten müssen – beispielsweise wenn Sie eine ältere Framework-Version verwenden müssen, die Generics nicht unterstützt, oder für bestimmte andere Plattformen entwickeln, bei denen Generics ebenfalls nicht zur Verfügung stehen.

Das Framework bietet zu diesem Zweck eine abstrakte Klasse namens `CollectionBase` als Vorlage an, die Sie für diese Zwecke ableiten und erweitern können.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 22\\TypeSafeCollections

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Der folgende Code zeigt, wie die `Item`-Eigenschaft und die `Add`-Methode in einer benutzerdefinierten Auflistung realisiert sind:

```
Public Class Adressen
    Inherits CollectionBase

    Public Overridable Function Add(ByVal Adr As Adresse) As Integer
        Return MyBase.List.Add(Adr)
    End Function

    Default Public Overridable Property Item(ByVal Index As Integer) As Adresse
        Get
            Return DirectCast(MyBase.List(Index), Adresse)
        End Get
        Set(ByVal Value As Adresse)
            MyBase.List(Index) = Value
        End Set
    End Property
End Class
```

Das entsprechende Beispielprogramm, das die Klasse testet, sieht folgendermaßen aus:

```
Module TypesafeCollections

Sub Main()
    Dim locAdressen As New Adressen
    Dim locAdresse As New Adresse("Christian", "Sonntag", "99999", "Trinken")
    Dim locAndererTyp As New FileInfo("C:\\Test.txt")

    'Kein Problem:
    locAdressen.Add(locAdresse)

    'Schon der Editor mault!
    'locAdressen.Add(locAndererTyp)

    'Auch kein Problem.
    locAdresse = locAdressen(0)
```

```

For Each eineAdresse As Adresse In locAdressen
    Console.WriteLine(eineAdresse)
Next
Console.ReadLine()
End Sub

End Module

```

Sie sehen anhand des Testprogramms, dass Typsicherheit in dieser Auflistung gewährleistet ist. Allerdings sollte dieses Programm, wenn Sie das Klassen-Kapitel dieses Buchs aufmerksam studiert haben, auch Fragen aufwerfen: Wenn Sie sich die Beschreibung zu `CollectionBase` in der Online-Hilfe ansehen, finden Sie dort folgenden Prototyp:

```

<Serializable>
MustInherit Public Class CollectionBase
    Implements IList, ICollection, IEnumerable
    .
    .
    .

```

`CollectionBase` bindet unter anderem die Schnittstelle `IList` ein. Die Schnittstelle `IList` schreibt vor, dass eine `Add`-Funktion mit der folgenden Signatur in der Klasse implementiert sein muss, die sie implementiert:

```
Function Add(ByVal value As Object) As Integer
```

Erkennen Sie die Unregelmäßigkeit? Unsere Klasse erbt von `CollectionBase`, `CollectionBase` bindet `IList` ein, `IList` verlangt die Methode `Add`, in unserer Klasse müsste demzufolge eine `Add`-Methode vorhanden sein, die wir zu überschreiben hätten! Aber sie ist es nicht, und das ist auch gut so, denn: Wäre sie vorhanden, dann müssten wir ihre Signatur zum Überschreiben übernehmen. Als Parameter nimmt sie aber ein `Object` entgegen – unsere Typsicherheit wäre dahin. Alternativ könnten wir sie überladen, aber in diesem Fall könnte man ihr dennoch ein `Object` übergeben – wieder wäre es aus mit der Typsicherheit.

Wir kommen dem Geheimnis auf die Spur, wenn wir uns die `Add`-Funktion unserer neuen Adressen-Auflistung ein wenig genauer anschauen. Dort heißt es:

```

Public Overridable Function Add(ByVal Adr As Adresse) As Integer
    Return MyBase.List.Add(Adr)
End Function

```

`MyBase` greift auf die Basisklasse zurück und verwendet letzten Endes die `Add`-Funktion des Objektes, das die `List`-Eigenschaft zurückliefert, um das hinzuzufügende Element weiterzureichen. Was aber macht `List`? Welches Objekt liefert es zurück? Um Sie vielleicht zunächst komplett zu verwirren: `List` liefert die Instanz unserer Klasse zurück⁴ – und wir betreiben hier Polymorphie in Vollendung!

⁴ Dieses Konstrukt erinnerte mich spontan an eine Star-Trek-Voyager-Episode, in der Tom Paris mit B'Elanna Torres das Holodeck besucht, um mit ihr in einem holografisch projizierten Kino einen »altertümlichen« Film mit 3-D-Brille zu sehen. Ihr Kommentar dazu: „Lass mich das mal klarstellen: Du hast diesen Aufwand betrieben, um eine dreidimensionale Umgebung so zu programmieren, das sie ein zweidimensionales Bild projiziert, und nun bittest Du mich, diese Brille zu tragen, damit es wieder dreidimensional ausschaut?« ...

Schauen wir, was die Online-Hilfe von Visual Studio zur Beschreibung von List zu sagen hat: »Ruft eine IList mit der Liste der Elemente in der CollectionBase-Instanz ab.« – Wow, das ist informativ!

Der ganze Umstand wird klar, wenn Sie sich das folgende Konstrukt anschauen. Behalten Sie dabei im Hinterkopf, dass eine Möglichkeit gefunden werden muss, die Add-Funktion einer Schnittstelle in einer Klasse zu implementieren, ohne einer sie einbindenden Klasse die Möglichkeit zu nehmen, eine Add-Funktion mit einer ganz anderen Signatur zur Verfügung zu stellen:

```
Interface ITest
    Function Add(ByVal obj As Object) As Integer
End Interface

MustInherit Class ITestKlasse
    Implements ITest

    Private Function ITestAdd(ByVal obj As Object) As Integer Implements ITest.Add
        Trace.WriteLine("ITestAdd:" + obj.ToString)
    End Function

End Class

Class ITestKlasseAbleitung
    Inherits ITestKlasse

    Public ReadOnly Property Test() As ITest
        Get
            Return DirectCast(Me, ITest)
        End Get
    End Property

    Public Sub Add(ByVal TypeSafe As Adresse)
        Test.Add(TypeSafe)
    End Sub
End Class
```

Die Schnittstelle dieser Klasse schreibt vor, dass eine Klasse, die diese Schnittstelle einbindet, eine Add-Funktion implementieren muss. Die abstrakte Klasse ITestKlasse bindet diese Schnittstelle auch ordnungsgemäß ein – allerdings stellt sie die Funktion nicht der Öffentlichkeit zur Verfügung; die Funktion ist dort nämlich als privat definiert. Außerdem – und das ist der springende Punkt – nennt sich die Funktion nicht Add, sondern ITestAdd – möglich wird das, da Schnittstelleneinbindungen in Visual Basic grundsätzlich explizit funktionieren, genauer gesagt durch das Schlüsselwort Implements am Ende der Funktionsdeklaration.

Halten wir fest: Wir haben nun eine saubere Schnittstellenimplementierung, *und* wir können die der Schnittstelle zugeordnete Funktion von außen nicht sehen. Wie rufen wir die Funktion ITestAdd der Basis-Klasse ITestKlasse dann überhaupt auf? Des Rätsels Lösung ist: Die Funktion ist nicht ganz so privat, wie sie scheint. Denn wir können über eine Schnittstellenvariable auf sie zugreifen. Wenn wir die eigene Instanz der Klasse in eine Schnittstellenvariable vom Typ ITest casten, dann können wir über die Schnittstellenfunktion ITest.Add dennoch auf die private Funktion ITestAdd der Klasse ITestKlasse zugreifen – und wir sind am Ziel!

Zu unserer Bequemlichkeit stellt die Klasse ITestKlasse bereits eine Eigenschaft (Property Test) zur Verfügung, die die aktuelle Klasseninstanz als ITest-Schnittstelle zurückliefert. Wir können uns dieser also direkt bedienen.

Genau das Gleiche haben wir in unserer aus CollectionBase entstandenen Klasse Adressen gemacht – um zum alten Beispiel zurückzukommen. Die List-Eigenschaft der Klasse CollectionBase entspricht der Test-Eigenschaft der Klasse ITestKlasse unseres Erklärungsbeispiels.

Hashtables – für das Nachschlagen von Objekten

Hashtable-Objekte sind das ideale Werkzeug, wenn Sie eine Datensammlung aufbauen wollen, aber die einzelnen Objekte nicht durch einen numerischen Index, sondern durch einen Schlüssel abrufen wollen. Ein Beispiel soll das verdeutlichen.

Angenommen, Sie haben eine Adressenverwaltung programmiert, bei der Sie einzelne Adressen durch eine Art Matchcode abrufen wollen (Kundennummer, Lieferantennummer, was auch immer). Bei der Verwendung einer ArrayList müssten Sie schon einigen Aufwand betreiben, um an ein Array-Element auf Basis des Matchcode-Namens zu gelangen: Sie müssten zum Finden eines Elements in der Liste für die verwendete Adressklasse eine CompareTo-Methode implementieren, damit die Liste mittels Sort sortiert werden könnte. Anschließend könnten Sie mit BinarySearch das Element finden – vorausgesetzt, die CompareTo-Methode würde eine Instanz der Adressklasse über ihren Matchcode-Wert vergleichen.

Hashtable-Objekte vereinfachen ein solches Szenario ungemein: Wenn Sie einer Hashtable ein Objekt hinzufügen, dann nimmt dessen Add-Methode nicht nur das zu speichernde Objekt (den hinzuzufügenden Wert) entgegen, sondern auch ein weiteres. Dieses zusätzliche Objekt (das genau genommen als erster Parameter übergeben wird) stellt den Schlüssel – den Key – zum Wiederauffinden des Objektes dar. Sie rufen ein Objekt aus einer Hashtable anschließend nicht wie bei der ArrayList mit

```
Element = eineArrayList(5)
```

ab, sondern mit dem entsprechenden Schlüssel, etwa:

```
Element = eineHashtable("ElementKey")
```

Voraussetzung bei diesem Beispiel ist natürlich, dass der Schlüssel zuvor ein entsprechender String gewesen ist.

Anwenden von Hashtables

BEGLEITDATEIEN

Im Verzeichnis

```
...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 22\\HashtableDemo01
```

finden Sie ein Beispielprojekt, das den Einsatz der Hashtable-Klasse demonstrieren soll. Es enthält eine Klasse, die eine Datenstruktur – eine Adresse – abbildet. Eine statische Funktion erlaubt darüber hinaus, eine beliebige Anzahl von Zufallsadressen zu erstellen, die zunächst in einer ArrayList gespeichert werden. Eine solche mit zufälligen Daten gefüllte ArrayList soll zum Experimentieren mit der Hashtable-Klasse dienen. Der Vollständigkeit halber (und des späteren einfacheren Verständnisses) möchte ich Ihnen diese Adresse-Klasse kurz vorstellen – wenn auch in verkürzter Form):

WICHTIG Sie benötigen mindestens 1 GByte-Arbeitsspeicher unter Windows XP, 2 GByte unter Windows Vista, damit Sie die folgenden Hashtable-Beispiele nachvollziehen können. Die Menge des zu reservierenden Speichers für das Beispiel ist deshalb so groß, weil durch die hohen Prozessorgeschwindigkeiten eine genügend große Anzahl von Elementen vorhanden sein muss, um überhaupt Geschwindigkeitsvergleiche anstellen zu können. Gleichzeitig muss aber soviel echter Speicher vorhanden sein, dass durch die schiere Masse an Daten die Auslagerungsdatei von Windows nicht in Anspruch genommen werden muss, was die Ergebnisse natürlich komplett verzerren würde.

```
Public Class Adresse
    'Member-Variablen, die die Daten halten:
    Protected myMatchcode As String
    Protected myName As String
    Protected myVorname As String
    Protected myPLZ As String
    Protected myOrt As String

    'Konstruktor - legt eine neue Instanz an.
    Sub New(ByVal Matchcode As String, ByVal Name As String, ByVal Vorname As String, _
            ByVal Plz As String, ByVal Ort As String)
        myMatchcode = Matchcode
        myName = Name
        myVorname = Vorname
        myPLZ = Plz
        myOrt = Ort
    End Sub
    'Mit Region ausgeblendet:
    'die Eigenschaften der Klasse, um die Daten offen zu legen.
    #Region "Eigenschaften"
        .
        .
        .

    #End Region
    Public Overrides Function ToString() As String
        Return Matchcode + ":" + Name + ", " + Vorname + ", " + PLZ + " " + Ort
    End Function

    Public Shared Function ZufallsAdressen(ByVal Anzahl As Integer) As ArrayList
        Dim locArrayList As New ArrayList(Anzahl)
        Dim locRandom As New Random(Now.Millisecond)

        Dim locNachnamen As String() = {"Heckhuis", "Löffelmann", "Thiemann", "Müller", _
                                         "Meier", "Tiemann", "Sonntag", "Ademmer", "Westermann", "Vüllers", _
                                         "Höllmann", "Vielstedde", "Weigel", "Weichel", "Weichelt", "Hoffmann", _
                                         "Rode", "Trouw", "Schindler", "Neumann", "Jungemann", "Hörstmann", _
                                         "Tinoco", "Albrecht", "Langenbach", "Braun", "Plenge", "Englisch", _
                                         "Clarke"}
        Dim locVornamen As String() = {"Jürgen", "Gabriele", "Uwe", "Katrín", "Hans", _
                                         "Rainer", "Christian", "Uta", "Michaela", "Franz", "Anne", "Anja", _
                                         "Theo", "Momo", "Katrín", "Guido", "Barbara", "Bernhard", "Margarete", _
                                         "Alfred", "Melanie", "Britta", "José", "Thomas", "Daja", "Klaus", "Axel", _
                                         "Lothar", "Gareth"}
        Dim locStädte As String() = {"Wuppertal", "Dortmund", "Lippstadt", "Soest", _
```

```

    "Liebenburg", "Hildesheim", "München", "Berlin", "Rheda", "Bielefeld", _
    "Braunschweig", "Unterschleißheim", "Wiesbaden", "Straubing", _
    "Bad Waldliesborn", "Lippetal", "Stirpe", "Erwitte" }

For i As Integer = 1 To Anzahl
    Dim locName, locVorname, locMatchcode As String
    locName = locNachnamen(locRandom.Next(locNachnamen.Length - 1))
    locVorname = locVornamen(locRandom.Next(locNachnamen.Length - 1))
    locMatchcode = locName.Substring(0, 2)
    locMatchcode += locVorname.Substring(0, 2)
    locMatchcode += i.ToString("0000000")
    locArrayList.Add(New Adresse(
        locMatchcode, _
        locName, _
        locVorname, _
        locRandom.Next(99999).ToString("00000"), _
        locStädte(locRandom.Next(locStädte.Length - 1))))
Next
Return locArrayList
End Function

Shared Sub AdressenAusgeben(ByVal Adressen As ArrayList)
    For Each Item As Object In Adressen
        Console.WriteLine(Item)
    Next
End Sub

End Class

```

Die eigentliche Klasse zum Speichern einer Adresse ist Kinderkram. Wichtig ist, dass Sie wissen, welche besondere Bewandtnis es mit dem Matchcode einer Adresse hat. Ein Matchcode ist in einem Satz von Adressen immer eindeutig. Die Prozedur in diesem Beispiel, die zufällige Adressen erzeugt, stellt das sicher. Der Matchcode setzt sich aus den ersten beiden Buchstaben des Nachnamens, den ersten beiden Buchstaben des Vornamens und einer fortlaufenden Nummer zusammen. Würden Sie 15 verschiedene Zufallsadressen mit diesem Code

```

Sub AdressenTesten()
    '15 zufällige Adressen erzeugen.
    Dim locDemoAdressen As ArrayList = Adresse.ZufallsAdressen(15)

    'Adressen im Konsolenfenster ausgeben.
    Console.WriteLine("Liste mit zufällig erzeugten Personendaten")
    Console.WriteLine(New String("=", 30))
    Adresse.AdressenAusgeben(locDemoAdressen)

End Sub

```

erstellen und ausgeben, sähen Sie etwa folgendes Ergebnis im Konsolenfenster:

```
Liste mit zufällig erzeugten Personendaten
=====
HeMo00000001: Heckhuis, Momo, 06549 Straubing
SoGu00000002: Sonntag, Guido, 21498 Liebenburg
ThA100000003: Thiemann, Alfred, 51920 Bielefeld
HöJü00000004: Hörstmann, Jürgen, 05984 Liebenburg
TiMa00000005: Tiemann, Margarete, 14399 München
TiAn00000006: Tiemann, Anja, 01287 Dortmund
ViGu00000007: Vielstede, Guido, 72762 Wuppertal
RoMe00000008: Rode, Melanie, 94506 Hildesheim
TiDa00000009: Tiemann, Daja, 54134 Lippstadt
BrJo00000010: Braun, José, 14590 Soest
WeJü00000011: Westermann, Jürgen, 83128 Wuppertal
HeKa00000012: Heckhuis, Katrin, 13267 Bad Waldliesborn
TrJü00000013: Trouw, Jürgen, 54030 Lippstadt
PlGa00000014: Plenge, Gabriele, 97702 Braunschweig
WeJü00000015: Weichel, Jürgen, 39992 Unterschleißheim
```

Falls Sie sich nun fragen, wieso ich Ihnen soviel Vorgeplänkel erzähle: Es ist wichtig für die richtige Anwendung von Hashtable-Objekten und das richtige Verständnis, wie Elemente in Hashtables gespeichert werden. Denn wenn Sie ein Objekt in einer Hashtable speichern, muss der Schlüssel eindeutig sein – anderenfalls hätten Sie nicht die Möglichkeit, wieder an alle Elemente heranzukommen. (Welches von zwei Elementen sollte die Hashtable schließlich durch einen Schlüssel indiziert zurückliefern, wenn die Schlüssel die gleichen wären?)

Verarbeitungsgeschwindigkeiten von Hashtables

Jetzt lassen Sie uns eine Hashtable in Aktion sehen und schauen, was sie zu leisten vermag. Aus den Elementen der ArrayList, die uns die ZufallsAdressen-Funktion liefert, bauen wir eine Hashtable mit nicht weniger als 1.000.000 Elementen auf. Gleichzeitig erzeugen wir ein weiteres Array mit 50 zufälligen Elementen der ArrayList und merken uns deren Matchcode. Diese Matchcodes verwenden wir anschließend, um uns Elemente aus der Liste herauszupicken und messen dabei die Zeit, die das Zugreifen benötigt. Das Programm dazu sieht folgendermaßen aus:

```
Module HashtableDemo

Sub Main()

    'AdressenTesten()
    'Console.ReadLine()
    'Return

    Dim locAnzahlAdressen As Integer = 1000000
    Dim locZugriffsElemente As Integer = 2
    Dim locMessungen As Integer = 3
    Dim locZugriffe As Integer = 1000000
    Dim locVorlageAdressen As ArrayList
    Dim locAdressen As New Hashtable
    Dim locTestKeys(locZugriffsElemente) As String
    Dim locZeitmesser As New HighSpeedTimeGauge
    Dim locRandom As New Random(Now.Millisecond)
```

```
'Warten auf Startschuss.  
Console.WriteLine("Drücken Sie Return, um zu beginnen", locZeitmesser.DurationInMilliSeconds)  
Console.ReadLine()  
  
'Viele Adressen erzeugen:  
Console.Write("Erzeugen von {0} zufälligen Adresseneinträgen...", locAnzahlAdressen)  
locZeitmesser.Start()  
locVorlageAdressen = adresse.ZufallsAdressen(locAnzahlAdressen)  
locZeitmesser.Stop()  
Console.WriteLine("fertig nach {0} ms", locZeitmesser.DurationInMilliSeconds)  
locZeitmesser.Reset()  
  
'Aufbauen der Hashtable.  
Console.Write("Aufbauen der Hashtable mit zufälligen Adresseneinträgen...", locAnzahlAdressen)  
locZeitmesser.Start()  
For Each adresse As Adresse In locVorlageAdressen  
    locAdressen.Add(adresse.Matchcode, adresse)  
Next  
locZeitmesser.Stop()  
Console.WriteLine("fertig nach {0} ms", locZeitmesser.DurationInMilliSeconds)  
locZeitmesser.Reset()  
  
'51 zufällige Adressen rauspicken.  
For i As Integer = 0 To locZugriffsElemente  
    locTestKeys(i) = DirectCast(locVorlageAdressen(locRandom.Next(locAnzahlAdressen)), _  
        Adresse).Matchcode  
Next  
  
Dim locTemp As Object  
Dim locTemp2 As Object  
  
'Zugreifen und messen, wie lange das dauert,  
'das ganze 5 Mal, um die Messung zu bestätigen.  
For z As Integer = 1 To locMessungen  
    Console.WriteLine()  
    Console.WriteLine("{0}. Messung:", z)  
    For i As Integer = 0 To locZugriffsElemente  
        Console.Write("{0} Zugriffe auf: {1} in ", locZugriffe, locTestKeys(i))  
        locTemp = locTestKeys(i)  
        locZeitmesser.Start()  
        For j As Integer = 1 To locZugriffe  
            locTemp2 = locAdressen(locTemp)  
        Next j  
        locZeitmesser.Stop()  
        locTemp = locTemp2.GetType()  
        Console.WriteLine("{0} ms", locZeitmesser.DurationInMilliSeconds)  
    Next  
  
'Zugriff auf ArrayList für Vergleich  
For i As Integer = 0 To locZugriffsElemente  
    Console.Write("{0} Zugriffe auf ArrayList-Element in ", locZugriffe)  
    locZeitmesser.Start()  
    For j As Integer = 1 To locZugriffe  
        locTemp2 = locVorlageAdressen(0)  
    Next j
```

```
    locZeitmesser.Stop()
    locTemp = locTemp2.GetType()
    Console.WriteLine("{0} ms", locZeitmesser.DurationInMilliseconds)
Next
Next

Console.ReadLine()
End Sub
```

Wenn Sie das Programm starten, erzeugt es eine Ausgabe, etwa wie im folgenden Bildschirmauszug:⁵

Drücken Sie Return, um zu beginnen

Erzeugen von 1000000 zufälligen Adresseneinträgen...fertig nach 2105 ms
Aufbauen der Hashtable mit zufälligen Adresseneinträgen...fertig nach 584 ms

1. Messung:

1000000 Zugriffe auf: WeMo00182238 in 102 ms
1000000 Zugriffe auf: PIMa00662840 in 130 ms
1000000 Zugriffe auf: WeLo00244369 in 129 ms
1000000 Zugriffe auf ArrayList-Element in 24 ms
1000000 Zugriffe auf ArrayList-Element in 28 ms
1000000 Zugriffe auf ArrayList-Element in 25 ms

2. Messung:

1000000 Zugriffe auf: WeMo00182238 in 141 ms
1000000 Zugriffe auf: PIMa00662840 in 134 ms
1000000 Zugriffe auf: WeLo00244369 in 109 ms
1000000 Zugriffe auf ArrayList-Element in 24 ms
1000000 Zugriffe auf ArrayList-Element in 24 ms
1000000 Zugriffe auf ArrayList-Element in 24 ms

3. Messung:

1000000 Zugriffe auf: WeMo00182238 in 102 ms
1000000 Zugriffe auf: PIMa00662840 in 131 ms
1000000 Zugriffe auf: WeLo00244369 in 104 ms
1000000 Zugriffe auf ArrayList-Element in 24 ms
1000000 Zugriffe auf ArrayList-Element in 25 ms
1000000 Zugriffe auf ArrayList-Element in 24 ms

Nach dem Programmstart legt das Programm hier im Beispiel 1.000.000 Testelemente an und baut daraus die Hashtable auf. Anschließend pickt es sich drei Beispieleinträge heraus und misst die Zeit, die es für 1.000.000 Zugriffe auf jeweils diese Elemente der Hashtable benötigt. Das Ganze macht es dreimal, um eine Konstanz in den Verarbeitungsgeschwindigkeiten sicherzustellen. Um im Gegenzug nachzuweisen, wie schnell ein indizierter Zugriff auf ein ArrayList-Element erfolgt, führt es eine Messung dazu anschließend durch. Das Ergebnis ist beeindruckend: Der Zugriff auf ein Element über seinen Schlüssel ist nur rund 4 mal so langsam wie der direkte Zugriff über den direkten Index. Dieses Ergebnis wäre nicht möglich, wenn man die gesamte Hashtable nach dem Key von vorne bis hinten durchsuchen würde. Und was auch auffällt:

⁵ Notebook mit 2,5 GHz Dual Core T9300, 6 MByte 2nd Level Cache mit 2,5 GHz Takt, 4 GByte Hauptspeicher auf 64-Bit-Vista und SP1 unter .NET Framework 3.5 SP1.

Wieso die Zugriffszeit auf Hashtable-Elemente nahezu konstant ist ...

Sie können die Parameter am Anfang des Programms verändern, um weitere Eindrücke der unglaublichen Geschwindigkeit von .NET zu sammeln. Sie werden bei allen Experimenten jedoch eines herausfinden: Ganz gleich, wie Sie auch an den Schrauben drehen, die Zugriffsgeschwindigkeit auf ein einzelnes Element bleibt nahezu konstant. In den seltensten Fällen benötigt der Zugriff auf ein Element das Doppelte der Durchschnittszeit – und diese Ausreißer sind vergleichsweise selten.

Das Geheimnis für die auf der einen Seite sehr hohe, auf der anderen Seite durchschnittlich gleich bleibende Geschwindigkeit beim Zugriff liegt am Aufbau der Hashtable und an der Behandlung der Schlüssel. Die schnellste Art und Weise, auf ein Element eines Arrays zuzugreifen, ist das direkte Auslesen des Elementes über seinen Index (auch das hat das vorherige Beispiel mit dem Zugriff auf die ArrayList-Elemente gezeigt). Daraus folgt, dass es am günstigsten ist, die Elemente der Hashtable nicht nach einem Schlüssel durchsuchen zu müssen, sondern die Positionsnummer eines Elementes der Hashtable irgendwie zu berechnen. Und genau hier setzt das *Hashing*-Konzept an. *Hashing* bedeutet eigentlich »zerhacken«. Das klingt sehr negativ, doch das Zerhacken des Schlüssels, das auf eine bestimmte Weise tatsächlich stattfindet, dient hier einem konstruktiven Zweck: Wenn der Schlüssel, der beispielsweise in Form einer Zeichenkette vorliegt, *gehashed* wird, geht daraus eine Kennzahl hervor, die Aufschluss über die Wertigkeit der Zeichenkette gibt. *Wertigkeit* in diesem Zusammenhang bedeutet, dass gleiche Zeichenketten nicht nur gleiche Hash-Werte ausweisen, sondern auch, dass größere Zeichenketten größere Hash-Werte bedeuten.

Ein einfaches Beispiel soll diese Zusammenhänge klarstellen: Angenommen, Sie haben 26 Wörter, die alle mit einem anderen Buchstaben beginnen. Diese Wörter liegen in unsortierter Reihenfolge vor. Nehmen wir weiter an, dass Ihr Hash-Algorithmus ganz einfach gestrickt ist: Der Hashcode einer Zeichenfolge entspricht der Nummer des Anfangsbuchstabens jedes Wortes. In dieser vereinfachten Konstellation haben Sie das Problem des Positionsberechnens bereits gelöst. Ihr Hashcode ist die Indexnummer im Array; sowohl das Einsortieren als auch das spätere Auslesen funktioniert in Windeseile über die Zeichenfolge selbst (genauer über seinen Anfangsbuchstaben).

Das Problem gestaltet sich in der Praxis natürlich nicht ganz so einfach. Mehr Zeichen (um beim Beispiel Zeichenfolgen für Schlüssel zu bleiben) müssen berücksichtigt werden, um den Hash zu berechnen, und bei langen Zeichenketten und begrenzter Hashcode-Präzision kann man nicht ausschließen, dass unterschiedliche Zeichenketten gleiche Hashcodes ergeben. In diesem Fall müssen Kollisionen bei der Indexberechnung berücksichtigt werden. So groß gestaltet sich das Problem aber gar nicht, denn wenn beispielsweise beim Einsortieren der Elemente der sich durch den Hashcode des Schlüssels ergebende Index bereits belegt ist, nimmt man eben den nächsten freien.

Um beim Beispiel zu bleiben: Sie haben nun 26 Elemente, von denen alle mit einem anderen Anfangsbuchstaben beginnen, mit einer Unregelmäßigkeit: Sie haben zwei A-Wörter, ein B-Wort und kein C-Wort. Der Hash-Algorithmus bleibt der gleiche. Sie sortieren das B-Wort in Slot 2, anschließend das erste A-Wort in Slot 1. Nun bekommen Sie das zweite A-Wort zum Einsortieren, und laut Hashcode zeigt es auch auf Slot 1. In diesem Fall fangen Sie an zu suchen und testen Slot 2, der durch das B-Wort schon belegt ist, aber Sie finden anschließend einen freien Slot 3. Der gehört eigentlich zu C, aber in diesem Fall ist er momentan nicht nur frei, sondern wird auch nie beansprucht werden, da es kein C-Wort in der einzusortierenden Liste gibt. Im ungünstigsten Fall gibt es in diesem Beispiel zwar ein C-Wort, dafür aber kein Z-Wort, und das zweite A-Wort wird als letztes Element eingesortiert. Jetzt verläuft die Suche über alle Elemente und landet schließlich auf dem letzten Element für das Z.

... und wieso Sie das wissen sollten!

Sie können sich die Verminderung solcher Fälle mit zusätzlichem Speicherplatz erkaufen. Angenommen, Sie reservieren in unserem Beispiel doppelt so viel Speicher für die Schlüssel, dann sind zwar ganz viele Slots leer, aber das zweite A-Wort muss nicht den ganzen Weg bis zum Z-Index antreten. Was ganz wichtig ist: Sie wissen jetzt, was ein *Load-Faktor* ist. So nennt man nämlich den Faktor, der das Verhältnis zwischen Wahrscheinlichkeiten von Zuordnungskollisionen und benötigtem Speicher angibt. Weniger Kollisionswahrscheinlichkeit erfordert höheren Speicher und umgekehrt. Und Sie können diesen Zusammenhang auch am Beispielprogramm austesten.

Am Anfang des Moduls in der Sub Main des Beispielprogramms finden Sie einen Block mit auskommentierten Deklarationsanweisungen. Tauschen Sie die Auskommentierung der beiden Blöcke, um folgende Parameter für den nächsten Versuch zu Grunde zu legen:

```
Dim locAnzahlAdressen As Integer = 1000000
Dim locZugriffsElemente As Integer = 25
Dim locMessungen As Integer = 3
Dim locZugriffe As Integer = 1000000
Dim locVorlageAdressen As ArrayList
Dim locAdressen As New Hashtable(100000, 1)
Dim locTestKeys(locZugriffsElemente) As String
Dim locZeitmesser As New HighSpeedTimeGauge
Dim locRandom As New Random(Now.Millisecond)
```

Mit diesem Block erhöhen wir die Anzahl der Elemente, die es zu testen gilt, auf 50, und damit erhöhen wir natürlich auch die Wahrscheinlichkeit, Elemente zu finden, die kollidieren. Gleichzeitig verändern wir – im Codelisting fett markiert – den Load-Faktor der Hashtable. Bei der Instanziierung einer Hashtable können Sie, neben der Anfangskapazität, als zweiten Parameter den Load-Faktor bestimmen. Gültig sind dabei Werte zwischen 0.1 und 1. Für den ersten Durchlauf bestimmen wir einen Load-Faktor von 1. Dabei wird auf Speicherplatz auf Kosten von vielen zu erwartenden Kollisionen und damit auf Kosten von Geschwindigkeit verzichtet (dieser Wert entspricht übrigens der Voreinstellung, die dann verwendet wird, wenn Sie keinen Parameter für den Load-Faktor angeben).

Starten Sie das Programm mit diesen Einstellungen und beenden Sie es zunächst **nicht**, nachdem der Messdurchlauf abgeschlossen wurde! Je nach Leistungsfähigkeit Ihres Rechners sehen Sie bei der Ausgabe der Testergebnisse Messungen, von denen ich nur einige ausschnittsweise im Folgenden zeigen möchte:

```
1000000 Zugriffe auf: BrFr00803543 in 126 ms
1000000 Zugriffe auf: HoK100971779 in 104 ms
1000000 Zugriffe auf: TiAn00984539 in 161 ms
1000000 Zugriffe auf: TiK100353084 in 102 ms
1000000 Zugriffe auf: HoKa00471671 in 108 ms
1000000 Zugriffe auf: ViBe00059525 in 105 ms
1000000 Zugriffe auf: WeKa00412662 in 117 ms
1000000 Zugriffe auf: TiBr00982230 in 152 ms
1000000 Zugriffe auf: BrBa00457658 in 104 ms
1000000 Zugriffe auf: HoAn00031026 in 104 ms
1000000 Zugriffe auf: TrMo00777434 in 175 ms
1000000 Zugriffe auf: ThMi00652362 in 110 ms
1000000 Zugriffe auf: WeUw00237110 in 115 ms
1000000 Zugriffe auf: HeBr00835839 in 104 ms
```

```
1000000 Zugriffe auf: VüBr00254760 in 131 ms
1000000 Zugriffe auf: LöMa00309629 in 129 ms
1000000 Zugriffe auf: ScUt00344455 in 101 ms
1000000 Zugriffe auf: WeBa00812888 in 104 ms
```

Sie erkennen, dass die Zugriffsdauer auf die Elemente in dieser Liste erheblich streut. Die Zugriffszeit auf einige Elemente liegt teilweise doppelt so hoch wie im Durchschnitt.

Rufen Sie nun, ohne das Programm zu beenden, den Task-Manager Ihres Betriebssystems auf. Dazu klicken Sie mit der rechten Maustaste auf die Task-Leiste (auf einen freien Bereich neben dem Startmenü) und wählen den entsprechenden Menüeintrag aus.

Wenn der Task-Manager-Dialog dargestellt ist, wählen Sie die Registerkarte *Prozesse*. Suchen Sie den Eintrag *HashtableDemo*, und merken Sie sich zunächst den Wert, der dort als Speicherauslastung angegeben wurde (etwa wie in Abbildung 22.3 zu sehen).

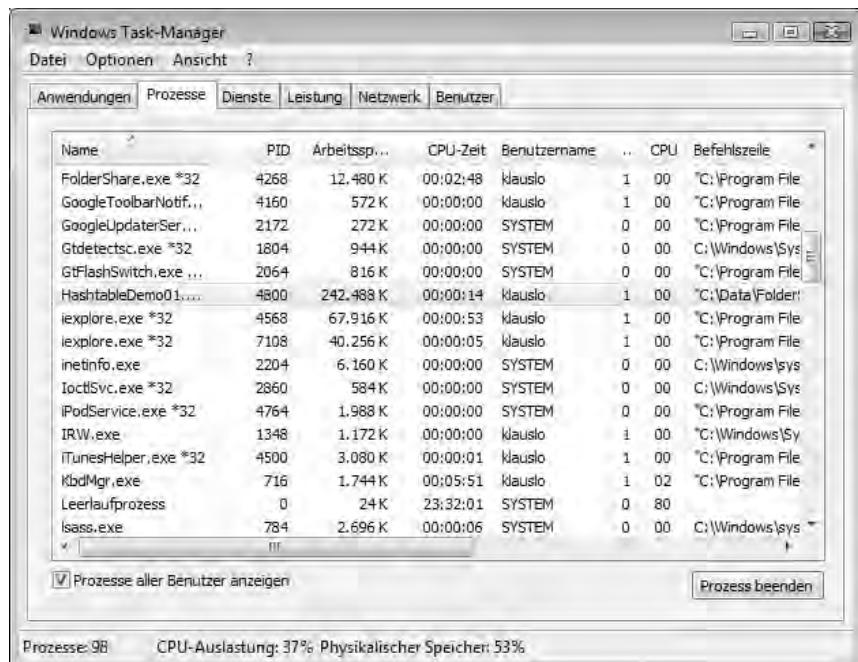


Abbildung 22.3 Bei einem großen Wert für den Parameter Load-Faktor einer Hashtable hält sich der Speicherbedarf in Grenzen, dafür gibt es mehr Zuordnungskollisionen, die Zeit kosten ...

Beenden Sie das Programm anschließend.

Jetzt verändern Sie den Load-Faktor der Hashtable auf den Wert 0.1 und wiederholen die Prozedur:

```
Dim locAdressen As New Hashtable(100000, 0.1)
```

Wenn Sie nicht gerade über mindestens 1.5 GByte Speicher in Ihrem Rechner verfügen, werden Sie den ersten deutlichen Unterschied bereits beim Aufbauen der Hashtable bemerken, das sehr viel mehr Zeit in Anspruch nimmt. Schuld daran ist in diesem Fall auch das Betriebssystem, das Teile des Hauptspeichers zunächst auf die Platte auslagern muss.

Beobachten Sie anschließend, wie sich der kleinere Load-Faktor auf die Zugriffsgeschwindigkeit der Elemente ausgewirkt hat:

```
1000000 Zugriffe auf: MüA100837928 in 103 ms
1000000 Zugriffe auf: HoDa00414300 in 103 ms
1000000 Zugriffe auf: JuGu00310325 in 105 ms
1000000 Zugriffe auf: AlBr00045195 in 104 ms
1000000 Zugriffe auf: NeJw00268129 in 103 ms
1000000 Zugriffe auf: TrFr00286598 in 104 ms
1000000 Zugriffe auf: HöJu00041436 in 106 ms
1000000 Zugriffe auf: HeJu00963087 in 108 ms
1000000 Zugriffe auf: WeCh00740139 in 104 ms
1000000 Zugriffe auf: AlGa00720899 in 104 ms
1000000 Zugriffe auf: BrUw00572080 in 104 ms
1000000 Zugriffe auf: WeRa00608848 in 102 ms
1000000 Zugriffe auf: AdMi00350918 in 107 ms
1000000 Zugriffe auf: TrKa00955107 in 104 ms
```

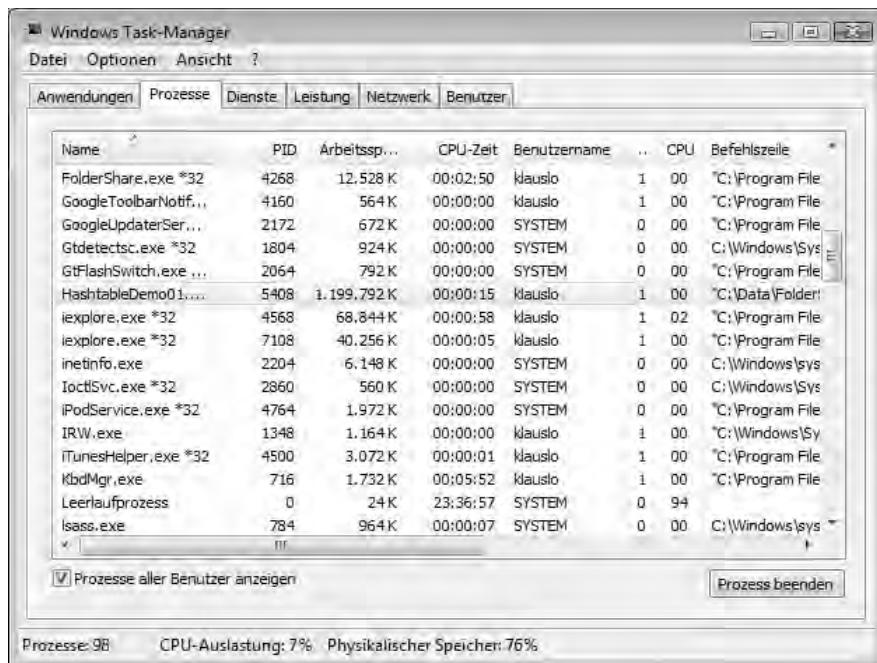


Abbildung 22.4 ... auf der anderen Seite bedeutet ein kleiner Load-Faktor zwar konstantere Zugriffszeiten, aber auf Kosten des Speichers

Sie sehen, dass im Gegensatz zum vorherigen Beispiel die Zugriffszeiten nahezu konstant sind. Natürlich kann es bei einer sehr unglücklichen Verteilung immer noch vorkommen, dass es den einen oder anderen Ausreißer gibt. Aber die Anzahl der Ausreißer ist deutlich gesunken.

Betrachten Sie anschließend den Speicherbedarf im Task-Manager, dann sehen Sie sofort, auf wessen Kosten die konstantere Zugriffsgeschwindigkeit erkauft wurde (Abbildung 22.4).

Der Speicherbedarf der Beispielapplikation hat sich nahezu verfünfacht!

Es gibt allerdings noch weiteres Beachtenswertes, das Sie im Hinterkopf behalten sollten, wenn Sie mit Hashtable-Objekten programmieren. Diese Dinge zu beachten wird Ihnen jetzt, da Sie wissen, wie Hash-tables prinzipiell funktionieren, abermals leichter fallen.

Verwenden eigener Klassen als Schlüssel (Key)

In allen bisherigen Beispielen zur Hashtable wurde eine Zeichenkette als Schlüssel eingesetzt. Der Index auf die Tabelle hat sich dabei aus dem Hashcode der als Schlüssel verwendeten Zeichenkette ergeben.

In unserem konkreten Beispiel ist das Ermitteln des Hashcodes des Strings eigentlich ein zu komplizierter und damit zu lange dauernder Algorithmus. Da wir eine eindeutige »Kundennummer« im Matchcode versteckt haben, können wir rein theoretisch auch diese Nummer als Hashcode verwenden, und der Zugriff auf die Elemente sollte damit – unabhängig vom Load-Faktor – gleich bleibend sein, denn die Kundennummer ist immer eindeutig (im Gegensatz zum Hashcode des Matchcodes, bei dem leicht Doublets und damit Kollisionen beim Einsortieren in die Hashtable auftreten können).

Wenn Sie Objekte eigener Klassen als Schlüssel verwenden wollen, müssen Sie die beiden folgenden Punkte beherzigen:

- Die Klasse muss ihre Equals-Funktionen überschreiben, damit die Hashtable-Klasse die Gleichheit zweier Schlüssel prüfen kann.
- Die Klasse muss die GetHashCode-Funktion überschreiben, damit die Hashtable-Klasse überhaupt einen Hashcode ermitteln kann.

WICHTIG Wenn Sie die Equals-Funktion überschreiben, achten Sie bitte darauf, die richtige Überladungsversion dieser Funktion zu überschreiben. Da die Basisfunktion (die Equals-Funktion von Object) sowohl als nicht statische als auch als statische Funktion implementiert ist, müssen Sie das Overloads-Schlüsselwort anwenden. Da Sie die Funktion überschreiben wollen, müssen Sie Overrides ebenfalls verwenden. Overrides alleine lässt der Compiler nicht zu, da die Funktion schon in die Basisklasse überladen ist. Allerdings – und das ist das Gefährliche – lässt er ein einzelnes Overloads zu, und das führt zu einer Überschattung der Basisfunktion, **ohne** sie zu überschreiben. Das Ergebnis: Der Compiler zeigt Ihnen keine Fehlermeldung (nicht einmal eine Warnung, obwohl er das sollte!), doch Ihre Funktion wird nicht polymorph behandelt und damit nie aufgerufen.

Mit diesem Wissen können wir nun unsere eigene Key-Klasse in Angriff nehmen. Bei unserem stark vereinfachten Beispiel bietet sich ein Teil der Matchcode-Zeichenkette an, direkt als *Hashcode* zu fungieren. Da die laufende Nummer einer Adresse Teil des Matchcodes ist, können wir genau diese als *Hashcode* verwenden. Der Code für eine eigene Key-Klasse könnte sich folgendermaßen gestalten:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 22\\HashtableDemo02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei). Veränderungen in Beispielcode sind im Folgenden fett markiert.

```
Public Class AdressenKey

    Private myMatchcode As String
    Private myKeyValue As Integer

    Sub New(ByVal Matchcode As String)
        myKeyValue = Integer.Parse(Matchcode.Substring(4))
        myMatchcode = Matchcode
    End Sub

    'Wird benötigt, um bei Kollisionen den richtigen
    'Schlüssel zu finden.
    Public Overloads Function Equals(ByVal obj As Object) As Boolean
        'If Not (TypeOf obj Is AdressenKey) Then
        '    Dim up As New InvalidCastException("AdressenKey kann nur mit Objekten gleichen Typs verglichen
        'werden")
        '    Throw up
        'End If
        Return myKeyValue.Equals(DirectCast(obj, AdressenKey).KeyValue)
    End Function

    'Wird benötigt, um den Index zu "berechnen".
    Public Overrides Function GetHashCode() As Integer
        Return myKeyValue
    End Function
    Public Overrides Function ToString() As String
        Return myKeyValue.ToString()
    End Function

    Public Property KeyValue() As Integer
        Get
            Return myKeyValue
        End Get
        Set(ByVal Value As Integer)
            myKeyValue = Value
        End Set
    End Property

End Class
```

Am ursprünglichen Testprogramm selbst sind nur einige kleinere Änderungen notwendig, um die neue Key-Klasse zu implementieren. Die Adresse-Klasse muss dabei überhaupt keine Änderung erfahren. Lediglich am Hauptprogramm müssen einige Zeilen geändert werden, um die Änderung wirksam werden zu lassen.

```
Module HashtableDemo

    Sub Main()

        Dim locAnzahlAdressen As Integer = 1000000
        Dim locZugriffsElemente As Integer = 50
        Dim locMessungen As Integer = 5
        Dim locZugriffe As Integer = 1000000
        Dim locVorlageAdressen As ArrayList
        Dim locAdressen As New Hashtable(100000, 0.1)
        Dim locTestKeys(locZugriffsElemente) As AdressenKey
```

```

Dim locZeitmesser As New HighSpeedTimeGauge
Dim locRandom As New Random(Now.Millisecond)
.
. ' Aus Platzgründen weggelassen.
.
' Aufbauen der Hashtable
Console.WriteLine("Aufbauen der Hashtable mit zufälligen Adresseneinträgen...", locAnzahlAdressen)
locZeitmesser.Start()
For Each adresse As Adresse In locVorlageAdressen
    'Änderung: Nicht den String, sondern ein Key-Objekt verwenden
    locAdressen.Add(New AdressenKey(adresse.Matchcode), adresse)
Next
locZeitmesser.Stop()
Console.WriteLine("fertig nach {0} ms", locZeitmesser.DurationInMilliSeconds)
locZeitmesser.Reset()

'51 zufällige Adressen rauspicken.
'Änderung: Die Keys werden abgespeichert, nicht der Matchcode.
For i As Integer = 0 To locZugriffsElemente
    locTestKeys(i) = New AdressenKey(
        DirectCast(locVorlageAdressen(locRandom.Next(locAnzahlAdressen)), Adresse).Matchcode)
Next
'Änderung: Kein Object mehr, sondern direkt ein AdressenKey.
Dim locTemp As AdressenKey
Dim locTemp2, locTemp3 As Object

'Zugreifen und messen, wie lange das dauert,
'Das ganze fünfmal, um die Messung zu bestätigen.
For z As Integer = 1 To locMessungen
    Console.WriteLine()
    Console.WriteLine("{0}. Messung:", z)
    For i As Integer = 0 To locZugriffsElemente
        Console.WriteLine("{0} Zugriffe auf: {1} in ", locZugriffe, locTestKeys(i))
        locTemp = locTestKeys(i)
        locZeitmesser.Start()
        For j As Integer = 1 To locZugriffe
            locTemp2 = locAdressen(locTemp)
        Next j
        locZeitmesser.Stop()
        locTemp3 = locTemp2.GetType
        Console.WriteLine("{0} ms", locZeitmesser.DurationInMilliSeconds)
    Next
    'Zugriff auf ArrayList für Vergleich
    .
    . ' Aus Platzgründen ebenfalls weggelassen.
    .
    Next
    Console.ReadLine()
End Sub

```

Die entscheidende Zeile im Beispielcode hat übrigens gar keine Änderung erfahren müssen. locTemp dient nach wie vor als Objektvariable für den Schlüssel, nur ist sie nicht mehr vom Typ String definiert, sondern als AdressenKey. locTemp3 (und ehemals locTemp) dient übrigens nur dazu, dass der Compiler die innere Schleife nicht wegoptimiert⁶ und damit Messergebnisse verfälscht.

⁶ Dieser Handgriff dient nur als Vorsichtsmaßnahme. Ich gebe zu, nicht wirklich überprüft zu haben, ob die Zeile wegoptimiert werden würde. Bei der Intelligenz moderner Compiler (oder JITter) ist das aber stark anzunehmen.

Wenn Sie dieses Programm starten, werden sie zwei Dinge feststellen. Der Zugriff auf die Daten ist spürbar schneller geworden, und: Der Zugriff auf die Daten erfolgt unabhängig vom Load-Faktor immer gleich schnell. Da der Schlüssel auf die Daten nun eindeutig ist, braucht sich die Hashtable nicht mehr um Kollisionen zu kümmern – es gibt nämlich keine mehr. Folglich bringt die Reservierung zusätzlicher Elemente auch keinen nennenswerten Vorteil mehr. Ganz im Gegenteil: Sie würden Speicher verschwenden, der gar nicht benötigt würde.

In diesem Beispiel sind die zu verwaltenden Datensätze nicht sonderlich groß gewesen. Wenn Sie überlegen, wie viele Zugriffe auf die Objekte der Hashtable tatsächlich notwendig gewesen sind, um überhaupt in einen messbaren Bereich zu gelangen, dann wird deutlich, dass sich der betriebene Aufwand für dieses Beispiel eigentlich nicht gelohnt hat. Dennoch: Denken Sie immer daran, dass Maschinen, auf denen Ihre Software später läuft, in der Regel nicht so leistungsfähig sind wie die Maschinen, auf denen sie entwickelt wird.

Schlüssel müssen unveränderlich sein!

Wenn Sie sich dazu entschlossen haben, eigene Klassen für die Verwaltung von Schlüsseln in einer Hashtable zu entwickeln, sollten Sie zusätzlich zum Gesagten Folgendes unbedingt beherzigen: Schlüssel müssen unveränderlich sein. Achten Sie darauf, dass Sie den Inhalt eines Key-Objektes nicht von außen verändern, solange er einer Hashtable zugeordnet ist. In diesem Fall würden Sie riskieren, dass die GetHashCode-Funktion unter Umständen einen falschen Hashcode für ein Key-Objekt zurückliefert. Der Nachschlagealgorithmus der Hashtable hätte dann keine Chance mehr, das Objekt in der Datentabelle wieder zu finden.

Enumrieren von Datenelementen in einer Hashtable

Die Enumeration einer Hashtable (das Iterieren durch sie mit For/Each) ist prinzipiell möglich. Allerdings müssen sie zwei Dinge dabei beachten:

- Datenelemente werden in einer Hashtable nicht in sequentieller Reihenfolge gespeichert. Da der Hashcode eines zu speichernden Objektes ausschlaggebend für die Position des Objektes innerhalb der Datentabelle ist, kann die wirkliche Position eines Objektes nur bei ganz einfachen Hashcode-Algorithmen vorausgesagt werden. Wenn Sie – und das wird wahrscheinlich am häufigsten der Fall sein – ein String-Objekt als Schlüssel verwenden, wirken die zu speichernden Objekte schon mehr oder weniger zufällig in der Tabelle verteilt.
- Objekte werden innerhalb einer Hashtable in so genannten Bucket-Strukturen gespeichert. Ein Schlüssel gehört untrennbar zu seinem eigentlichen Objekt, und beide werden in einem Bucket-Element in der Tabelle abgelegt. Eine Iteration durch die Datentabelle kann aus diesem Grund nur mit einem speziellen Objekt erfolgen – nämlich vom Typ DictionaryEntry (etwa: Wörterbucheintrag).

Eine Iteration durch eine Hashtable könnte für unser Beispiel folgendermaßen aussehen:

```
'Iterieren durch die Hashtable
For Each locDE As DictionaryEntry In locAdressen
    'in unserem Beispiel für den Key
    Dim locAdressenKey As AdressenKey = DirectCast(locDE.Key, AdressenKey)
    'in unserem Beispiel für das Objekt
    Dim locAdresse As Adresse = DirectCast(locDE.Value, Adresse)
Next
```

TIPP Schnelle, typsichere Auflistungen können Sie beispielsweise mithilfe der KeyedCollection implementieren, die Ihnen das .NET Framework ab der Version 2.0 als Basisklasse zur Verfügung stellt. Ein Beispiel für den Einsatz der KeyedCollection finden Sie im Kapitel 23.

DictionaryBase

Die Hashtable-Auflistung ermöglicht es, wie ArrayList selbst in ihrer Grundausstattung lediglich typunsichere Auflistungen von Objekten zu verwalten. Bis zum .NET Framework 2.0 war es lediglich mithilfe der Klasse DictionaryBase möglich, durch Vererbung eine typsichere Hashtable-Auflistung zu implementieren. Die Vorgehensweise dabei entspricht im Prinzip der gleichen, wie Sie sie für CollectionBase im Abschnitt »Typsichere Auflistungen auf Basis von CollectionBase« ab Seite 644 beschrieben finden – nur eben mit einer HashTable-Auflistung als InnerList und eben nicht mit einer ArrayList-Auflistung.

Mit dem .NET Framework 2.0 gibt es dank Generics (siehe Kapitel 23) eine sehr viel bequemere Möglichkeit, typsichere Wörterbuch/Hashtable-Implementierungen zu realisieren. Buchstäblicher Schlüssel zum Glück ist hier einerseits die generische Dictionary-Auflistung (Dictionary Of in Visual Basic), andererseits die so genannte KeyedCollection. Diese kann übrigens nur durch Vererbung konkret implementiert werden und dadurch sorgt sie selbstständig dafür, dass es eine homogene Schlüssel/Element-Auflistung gibt, indem sie dafür sorgt, den Schlüssel aus dem Element selbstständig zu ermitteln, das der Auflistung hinzugefügt wird. Ein Beispiel zu KeyedCollection finden Sie ebenfalls in Kapitel 23; bitte beachten Sie dabei unbedingt die Implementierungs-»Unzulänglichkeiten«, auf die dort auch im Zusammenhang mit KeyedCollection-Serialisierungen eingegangen wird.

Queue – Warteschlangen im FIFO-Prinzip

»First in first out« (als erstes rein, als erstes raus) – nach diesem Muster arbeitet die Queue-Klasse der BCL. Angewendet haben Sie dieses Prinzip selbst schon sicherlich einige Male in der Praxis – und zwar immer dann, wenn Sie unter Windows mehrere Dokumente hintereinander gedruckt haben. Das Drucken unter Windows funktioniert gemäß dem Warteschlangenprinzip. Das Dokument, das als Erstes in die Warteschlange eingereiht (*enqueue* – einreihen) wurde, wird als Erstes verarbeitet und anschließend wieder aus ihr entfernt (*dequeue* – ausreihen). Aus diesem Grund verwenden Sie die Methoden Enqueue, um Elemente der Queue hinzuzufügen und Dequeue, um sie zurückzubekommen und gleichzeitig aus der Warteschlange zu entfernen.

BEGLEITDATEIEN Falls Sie mit der Queue-Klasse experimentieren möchten, verwenden Sie dazu am besten nochmals das *CollectionsDemo*-Projekt, das Sie unter

...\\VB 2008 Entwicklerbuch\\D – Datenstrukturen\\Kapitel 22\\CollectionsDemo

finden. Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei). Verändern Sie das Programm so, dass es die Sub QueueDemo aufruft, um das folgende Beispiel nachzuvollziehen:

```
Sub QueueDemo()

    Dim locQueue As New Queue
    Dim locString As String

    locQueue.Enqueue("Erstes Element")
    locQueue.Enqueue("Zweites Element")
    locQueue.Enqueue("Drittes Element")
    locQueue.Enqueue("Viertes Element")

    'Nachschauen, was am Anfang steht, ohne es zu entfernen.
    Console.WriteLine("Element am Queue-Anfang:" + locQueue.Peek().ToString)
    Console.WriteLine()

    'Iterieren funktioniert auch.
    For Each locString In locQueue
        Console.WriteLine(locString)
    Next
    Console.WriteLine()

    'Alle Elemente aus Queue entfernen und Ergebnis im Konsolenfenster anzeigen.
    Do
        locString = CStr(locQueue.Dequeue)
        Console.WriteLine(locString)
    Loop Until locQueue.Count = 0
    Console.ReadLine()
End Sub
```

Wenn Sie dieses Programm ablaufen lassen, produziert es folgende Ausgabe im Konsolenfenster:

```
Element am Queue-Anfang:Erstes Element

Erstes Element
Zweites Element
Drittes Element
Viertes Element

Erstes Element
Zweites Element
Drittes Element
Viertes Element
```

Stack – Stapelverarbeitung im LIFO-Prinzip

Die Stack-Klasse arbeitet nach dem Prinzip »Last in first out« (»als letztes rein, als erstes raus«), arbeitet also genau umgekehrt zum FIFO-Prinzip der Queue-Klasse. Mit der Push-Methode schieben Sie ein Element auf den Stapel, mit Pull ziehen Sie es wieder herunter und erhalten es damit zurück. Das Element, das Sie zuletzt auf den Stapel geschoben haben, wird also mit Pull auch als Erstes wieder entfernt.

BEGLEITDATEIEN Sie finden die folgende Demo im gleichen Projekt wie die vorherige. Verändern Sie das Programm so, dass es die Sub StackDemo aufruft, um das folgende Beispiel nachzuvollziehen:

```
Sub StackDemo()
    Dim locStack As New Stack
    Dim locString As String

    locStack.Push("Erstes Element")
    locStack.Push("Zweites Element")
    locStack.Push("Drittes Element")
    locStack.Push("Viertes Element")

    'Nachschauen, was oben auf dem Stapel liegt, ohne das Element zu entfernen.
    Console.WriteLine("Element zu oberst auf dem Stapel: " + locStack.Peek.ToString())
    Console.WriteLine()

    'Iterieren funktioniert auch.
    For Each locString In locStack
        Console.WriteLine(locString)
    Next
    Console.WriteLine()

    'Alle Elemente vom Stack ziehen und Ergebnis im Konsolenfenster anzeigen.
    Do
        locString = CStr(locStack.Pop)
        Console.WriteLine(locString)
    Loop Until locStack.Count = 0
    Console.ReadLine()
End Sub
```

Wenn Sie dieses Programm ablaufen lassen, produziert es folgende Ausgabe im Konsolenfenster:

```
Element zu oberst auf dem Stapel: Viertes Element

Viertes Element
Dritttes Element
Zweites Element
Erstes Element

Viertes Element
Dritttes Element
Zweites Element
Erstes Element
```

SortedList – Elemente ständig sortiert halten

Wenn Sie Elemente schon direkt nach dem Einfügen in der richtigen Reihenfolge in einer Auflistung halten wollen, dann ist die `SortedList`-Klasse das richtige Werkzeug für diesen Zweck. Allerdings sollten Sie beachten: Von allen Auflistungsklassen ist die `SortedList`-Klasse diejenige, die die meisten Ressourcen verschlingt. Für zeitkritische Applikationen sollten Sie überlegen, ob Sie Ihre Daten auch anders organisieren und stattdessen lieber auf eine unsortierte `Hashtable` oder gar auf `ArrayList` zurückgreifen können.

Der Vorteil von `SortedList` ist, dass sie quasi aus einer Mischung von `ArrayList`- und `Hashtable`-Funktionen besteht (obwohl sie algorithmisch gesehen überhaupt nichts mit `Hashtable` zu tun hat). Sie können auf der einen Seite über einen Schlüssel, auf der anderen Seite aber auch über einen Index auf die Elemente von `SortedList` zugreifen.

Das folgende erste Beispiel zeigt den generellen Umgang mit `SortedList`.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 22\\SortedListDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei). Es besteht aus drei Codedateien. In der Datei *Daten.vb* finden Sie die schon bekannte Adresse-Klasse (bekannt, falls Sie die vorherigen Abschnitte ebenfalls durchgearbeitet haben) – allerdings in leicht veränderter Form. Der Matchcode der Zufallsadressen beginnt in dieser Version mit einer laufenden Nummer und endet mit der Buchstabenkombination des Nach- und Vornamens. Damit wird vermieden, dass eine Sortierung des Matchcodes grob auch die Adressen nach Namen und Vornamen sortiert und etwaige Nachweise eines bestimmten Programmverhaltens nicht geführt werden können.

```
Module SortedListDemo
    Sub Main()
        Dim locZufallsAdressen As ArrayList = Adresse.ZufallsAdressen(6)
        Dim locAdressen As New SortedList

        Console.WriteLine("Ursprungsanordnung:")
        For Each locAdresse As Adresse In locZufallsAdressen
            Console.WriteLine(locAdresse)
            locAdressen.Add(locAdresse.Matchcode, locAdresse)
        Next

        ' Zugriff per Index:
        Console.WriteLine()
        Console.WriteLine("Zugriff per Index:")
        For i As Integer = 0 To locAdressen.Count - 1
            Console.WriteLine(locAdressen.GetByIndex(i).ToString)
        Next

        Console.WriteLine()
        Console.WriteLine("Zugriff per Index:")
        ' Zugriff per Enumerator
        For Each locDE As DictionaryEntry In locAdressen
            Console.WriteLine(locDE.Value.ToString)
        Next
        Console.ReadLine()
    End Sub
End Module
```

Wenn Sie dieses Programm starten, generiert es in etwa die folgenden Ausgaben im Konsolenfenster (die Adressen werden zufällig generiert, deswegen kann die Darstellung in Ihrem Konsolenfenster natürlich wieder von der hier gezeigten abweichen).

Ursprungsanordnung:

```
00000005PlKa: Plenge, Katrin, 26201 Liebenburg
00000004PlKa: Plenge, Katrin, 93436 Liebenburg
00000003AlMa: Albrecht, Margarete, 65716 Bad Waldliesborn
00000002HoBa: Hollmann, Barbara, 96807 Liebenburg
00000001LoLo: Löffelmann, Lothar, 21237 Lippetal
00000000AdKa: Ademmer, Katrin, 49440 Unterschleißheim
```

Zugriff per Index:

```
00000000AdKa: Ademmer, Katrin, 49440 Unterschleißheim
00000001LoLo: Löffelmann, Lothar, 21237 Lippetal
00000002HoBa: Hollmann, Barbara, 96807 Liebenburg
00000003AlMa: Albrecht, Margarete, 65716 Bad Waldliesborn
00000004PlKa: Plenge, Katrin, 93436 Liebenburg
00000005PlKa: Plenge, Katrin, 26201 Liebenburg
```

Zugriff per Index:

```
00000000AdKa: Ademmer, Katrin, 49440 Unterschleißheim
00000001LoLo: Löffelmann, Lothar, 21237 Lippetal
00000002HoBa: Hollmann, Barbara, 96807 Liebenburg
00000003AlMa: Albrecht, Margarete, 65716 Bad Waldliesborn
00000004PlKa: Plenge, Katrin, 93436 Liebenburg
00000005PlKa: Plenge, Katrin, 26201 Liebenburg
```

Sie erkennen, dass die Liste in der Tat nach dem Schlüssel umsortiert wurde. Dies gilt sowohl für den Zugriff über den Index als auch über den Enumerator mit For/Each.

Kapitel 23

Generics (Generika) und generische Auflistungen

In diesem Kapitel:

Einführung	668
Lösungsansätze	669
Typengeneralisierung durch den Einsatz generischer Datentypen	671
Beschränkungen (Constraints)	674
Generische Auflistungen (Generic Collections)	684

Einführung

Generics – oder »Generika« wie Microsoft die so schöne deutsche Übersetzung dafür fand – sind nicht neu in Visual Basic 2008. Das erste Argument, sie dennoch in diesem Buch stattfinden zu lassen, ist, dass Generics – und vor allem generische Auflistungen – entscheidend für das Verständnis des nächsten Kapitels sind, in dem es um *Linq to Objects* geht. Das zweite Argument: Nicht jeder VB-Entwickler hat bereits mit Generics und generischen Auflistungen gearbeitet – eine Veranschaulichung der Konzepte kann daher so oder so nicht schaden, denn auch wenn Sie nicht vorhaben, in Zukunft mit *Linq to Objects* zu arbeiten: Das Verstehen von Generics und das Verwenden generischer Auflistungen wird Ihnen in jedem Fall helfen, Ihre Anwendungen sicherer zu machen und sie damit auch professionellen Standards genügen zu lassen.

Generics: Verwenden einer Codebasis für verschiedene Typen

Wenn Sie Methoden und Eigenschaften entwickeln, haben diese unter Umständen einen Nachteil: Sie verarbeiten, wenn sie typsicher sein sollen, nur einen bestimmten Datentyp.

Die Alternative dazu ist, dass Sie einen Typ schaffen, der die Aufnahme beliebiger Datentypen durch den auf Object basierenden Einsatz ermöglicht, doch ein solcher Typ ist dann nicht typsicher und kann zur Laufzeit Ausnahmen auslösen, mit denen Sie nicht rechnen.

BEGLEITDATEIEN Im Folgenden sehen Sie zunächst den Klassencode, der einen Adresseneintrag verwaltet – dieses Projekt finden Sie im Verzeichnis:

... \VB 2008 Entwicklerbuch\ D - Datenstrukturen\Kapitel 23\Generics01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Beispiel verwendet die aus dem vorherigen Kapitel bekannte Klasse *DynamicList* in leicht modifizierter Form. Das Modul verwendet eine Instanz von *DynamicList* und fügt ihr ein paar Elemente hinzu. Diese Elemente gibt es anschließend in einer For/Each-Schleife wieder im Konsolenfenster aus:

```
Module mdlMain

    Sub Main()
        Dim locListe As New DynamicList
        locListe.Add(123.32)
        locListe.Add(126.32)
        locListe.Add(124.52)
        locListe.Add(29.99)
        locListe.Add(13.54)
        'Der wird Probleme machen!
        locListe.Add(#12/31/2008 4:00:00 PM#)
        locListe.Add(43.32)
        For Each locItem As Double In locListe
            Console.WriteLine(locItem)
        Next
    End Sub
End Module
```

```
Console.WriteLine()
Console.WriteLine("Taste drücken zum Beenden!")
Console.ReadKey()
End Sub

End Module
```

Wenn Sie dieses Programm starten, sehen Sie anschließend Folgendes auf dem Bildschirm:

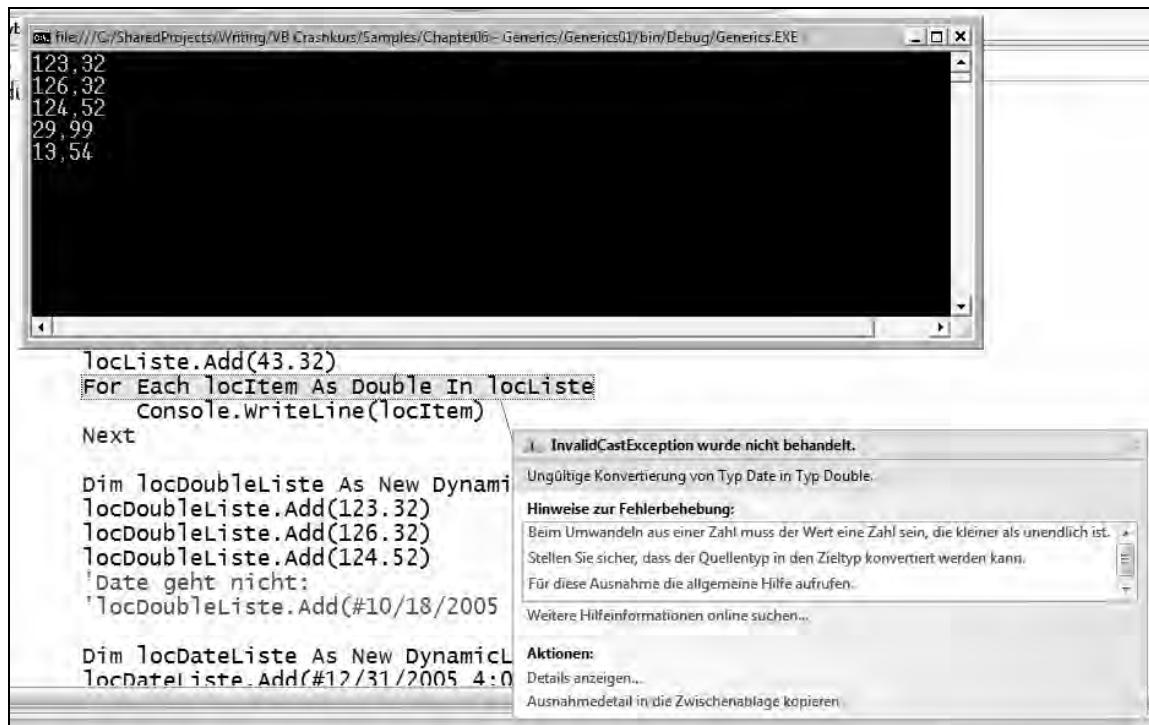


Abbildung 23.1 Die Liste wird nur bis zum *Date* Element ausgegeben; das *Date*-Element ist nämlich nicht in *Double* konvertierbar

Es ist klar, wieso das passiert. Wir haben eine Liste mit reinen Double-Elementen aufbauen wollen, aber uns ist ein Date-Element »dazwischengerutscht«. Solche Fehler sind in einem so einfachen Programm, wie wir es hier als Beispiel verwenden, noch auf den ersten Blick zu erkennen – doch in größeren Projekten sollte man solche Fehler von vornherein zu vermeiden versuchen.

Lösungsansätze

Und wie kann man das machen? Indem man eine Klasse wie die `DynamicList` typsicher macht. Indem man sie von vornherein erst gar keinen allgemein gültigen Datentyp wie `Object` akzeptieren lässt, sondern ausschließlich Werte vom Typ `Double`.

Und um das zu erreichen, nehmen wir uns den Quellcode der DynamicList vor, und führen die entsprechenden Änderungen durch. Konsequenterweise nennen wir diese Klasse dann auch `DynamicListDouble` (wie sie im angesprochenen Beispielprojekt übrigens bereits in einer eigenen Klassendatei vorhanden ist). Im folgenden Listing finden Sie all die Stellen in Fettschrift markiert, an denen der Datentyp `Object` in `Double` geändert wurde.

```
Class DynamicListDouble
    Implements IEnumerable

    Protected myStep As Integer = 4          ' Schrittweite, um die das Array erhöht wird.
    Protected myCurrentArraySize As Integer   ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer     ' Zeiger auf aktuelles Element
    Protected myArray() As Double           ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub

    Sub Add(ByVal Item As Double)
        'Element im Array speichern
        myArray(myCurrentCounter) = Item

        'Zeiger auf nächstes Element erhöhen
        myCurrentCounter += 1

        'Prüfen, ob aktuelle Arraygrenze erreicht wurde
        If myCurrentCounter = myCurrentArraySize - 1 Then
            'Neues Array mit mehr Speicher anlegen,
            'und Elemente hinüberkopieren. Dazu:
            'Neues Array wird größer:
            myCurrentArraySize += myStep

            'Temporäres Array erstellen
            Dim locTempArray(myCurrentArraySize - 1) As Double
            Array.Copy(myArray, locTempArray, myArray.Length)
            myArray = locTempArray
        End If
    End Sub
    .
    .
    .
End Class
```

Wenn Sie die Klasse auf diese Weise abgeändert und das eigentliche Programm wie folgt modifiziert haben,

```
Module mdlMain

    Sub Main()
        Dim locListe As New DynamicListDouble
```

```

locListe.Add(123.32)
locListe.Add(126.32)
.
.
```

zeigt Ihnen Visual Basic schon zur Entwurfszeit eine Fehlermeldung, wie Sie sie auch in Abbildung 23.2 sehen können.

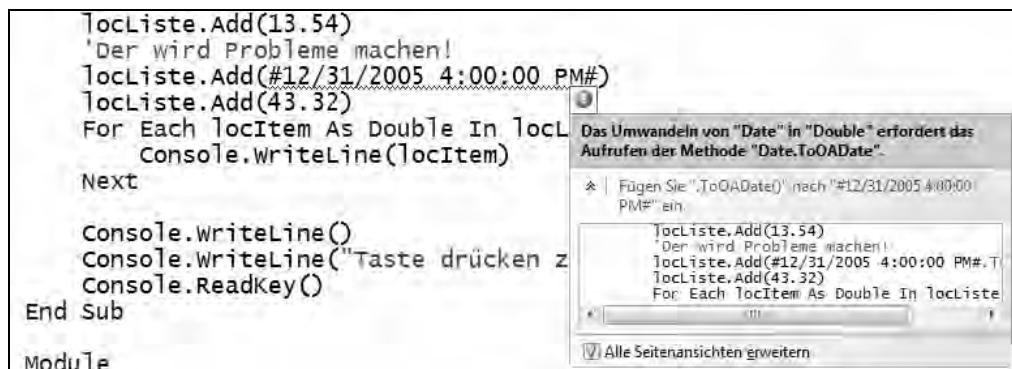


Abbildung 23.2 Bei typsicherer Klassen sehen Sie Fehlermeldungen bei »falschen« Typen bereits im Editor zur Entwurfszeit – wenngleich die Fehlermeldungen ab und an über das Ziel hinausschießen

Gut – diese Fehlermeldung an dieser Stelle schießt ein wenig über das Ziel hinaus; hätte es der Editor dabei belassen, uns über den falschen Typ an dieser Stelle zu informieren, wäre das ausreichend gewesen. Aber immerhin: Sie sehen auf jeden Fall durch die Verwendung der typsichereren Klasse schon zur Entwurfszeit, dass Sie einen Typen verwendet haben, den Sie mit dieser Klasse nicht verwenden dürfen.

Typengeneralisierung durch den Einsatz generischer Datentypen

Nun ist das Entwickeln einer Klasse wie `DynamicList` schon ein wenig aufwendiger. Und es wird in Ihrer späteren Entwicklerzeit Klassen und Strukturen geben, die noch viel, viel komplexer sind.

Doch gleichzeitig wird es gerade bei Klassen oder Strukturen, die große Datenmengen verwalten, immer wieder vorkommen, dass Sie sie für den Einsatz unterschiedlichster Typen verwenden wollen – für unser `DynamicList`-Beispiel trifft diese Aussage mehr als zu, denn:

Auch Zeichenketten ließen sich mit der Klasse wunderbar verwalten. Und auch `Integer`-Werte. Und auch `Decimals`. Und auch ... – eigentlich ist dieser Typ für alle Datentypen und Objektinstanzen geeignet, die Sie in größeren Mengen in einer Liste speichern wollen.

Doch als Programmierer sollten Sie aus den genannten Gründen immer auch auf Typsicherheit bestehen, und so bliebe Ihnen bislang eigentlich nur die Möglichkeit, ...

- ... für jeden benötigten Datentyp eine neue Version der verarbeitenden Klasse zu erstellen. Dieser Aufwand ist allerdings enorm groß, und zieht ein mindestens ebenso großes Problem nach sich: Finden Sie einen Fehler in einer Klasse, müssen Sie diesen in allen Klassen ändern, die sich nur durch ihren verarbeitenden Typ unterscheiden (DynamicListDouble, DynamicListString, DynamicListDate, und welche Klasse Sie auch immer sonst noch eingerichtet hätten).
- ... eine Klasse auf Basis einer Schnittstelle zu erstellen, mit der die Typen durch Vererbung anpassbar sind. Damit hält sich der Pflegeaufwand in Grenzen, da eine Fehlerbehebung in der Basisklasse sich natürlich auch in den Klassenableitungen widerspiegelt. Doch solche Klassen zu implementieren erfordert extrem abstraktes Denken und sorgfältige Planung, und dafür steht nicht unbedingt immer die Zeit zur Verfügung.

Mit generischen Datentypen wird das anders. Bei generischen Datentypen (auf englisch »Generics« – für den Fall, dass Sie mal nach englischen Artikeln im Internet recherchieren müssen) legen Sie sich während der Entwicklung überhaupt noch nicht fest, welchen Typ Ihre Klasse oder Struktur später einmal verarbeiten soll. Sie arbeiten stattdessen mit so genannten Typparametern, durch die der Typ – dem JITter sei Dank – erst bei der ersten Verwendung zur Laufzeit ersetzt wird.

Mit anderen Worten: Das, was Sie mit dem Kopieren und Typanpassen Ihres Codes zur Entwicklungszeit manuell machen, dafür sorgen JITter und die Technik der Generics zur Laufzeit automatisch. Vereinfacht gesagt: So, wie Sie bei Word mit Suchen und Ersetzen arbeiten können, wird der IML-Code Ihrer generischen Klasse kopiert, und alle Typparameter werden durch den angegebenen Typ ersetzt¹.

In der Praxis und für unser DynamicList-Beispiel sieht das wie folgt aus:

Anstelle sich von vornherein für einen Datentyp wie Double, Integer oder String zu entscheiden, platzieren Sie an den entscheidenden Stellen eine Art Platzhalter – einen Typparameter –, den Sie im Kopf der Klasse mit dem Zusatz Of benennen, etwa so:

```
Class DynamicList(Of flexibleDatentyp)
```

Und anstatt anschließend innerhalb der Klasse einen fixen Datentyp zu verwenden, setzen Sie diese Typparameter als Stellvertreter ein. Für unsere DynamicList-Klasse bedeutet das:

```
Class DynamicList(Of flexibleDatentyp)
    Implements IEnumerable

    Protected myStep As Integer = 4      ' Schrittweite, durch die das Array erhöht wird.
    Protected myCurrentArraySize As Integer ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer   ' Zeiger auf aktuelles Element
    Protected myArray() As flexibleDatentyp ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub
```

¹ Ganz so einfach geht es natürlich nicht. Es gibt für JITter bzw. Compiler durchaus die Möglichkeit, Codegemeinsamkeiten in generischen Klassen bestehen zu lassen, und nur dort neuen Code zu generieren, wo es nicht anders möglich ist.

```

Sub Add(ByVal Item As flexiblerDatentyp)
    myArray(myCurrentCounter) = Item
    myCurrentCounter += 1
    If myCurrentCounter = myCurrentArraySize - 1 Then
        myCurrentArraySize += myStep

        'temporäres Array erstellen
        Dim locTempArray(myCurrentArraySize - 1) As flexiblerDatentyp

        'Elemente kopieren;
        Array.Copy(myArray, locTempArray, myArray.Length)
        myArray = locTempArray
    End If
End Sub
'Liefert die Anzahl der vorhandenen Elemente zurück
Public Overridable ReadOnly Property Count() As Integer
    Get
        Return myCurrentCounter
    End Get
End Property

'Erlaubt das Zuweisen
Default Public Overridable Property Item(ByVal Index As Integer) As flexiblerDatentyp
    Get
        Return myArray(Index)
    End Get
    Set(ByVal Value As flexiblerDatentyp)
        myArray(Index) = Value
    End Set
End Property

Public Function GetEnumerator() As System.Collections.IEnumerator _
    Implements System.Collections.IEnumerable.GetEnumerator
    Dim locTempArray(myCurrentArraySize) As flexiblerDatentyp
    Array.Copy(myArray, locTempArray, myArray.Length)
    Return myArray.GetEnumerator
End Function
End Class

```

Und nun können Sie DynamicList für jeden Datentyp verwenden, den Sie möchten, und Sie müssen dabei nicht auf Typsicherheit verzichten, wie Abbildung 23.3 zeigt.

In der Grafik sehen Sie zweierlei. Zum einen, wie Sie einen generischen Datentyp anwenden. In der Sub Main des Moduls wird die generische Klasse einmal auf Basis des Datentyps Double definiert

```
Dim locDoubleListe As New DynamicList(Of Double)
```

und einmal auf Basis des Datentyps Date:

```
Dim locDateListe As New DynamicList(Of Date)
```

```

Dim locDoubleListe As New DynamicList(Of Double)
locDoubleListe.Add(123.32)
locDoubleListe.Add(126.32)
locDoubleListe.Add(124.52)
'Date geht nicht:
locDoubleListe.Add(#10/18/2005 3:20:00 PM#)

Dim locDateListe As New DynamicList(Of Date)
locDateListe.Add(#12/31/2005 4:00:00 PM#)
locDateListe.Add(#11/24/2005 6:20:00 PM#)
locDateListe.Add(#10/18/2005 3:20:00 PM#)
'Double geht nicht:
locDateListe.Add(124.52)

End Module

```

Beschreibung	Datei	Zeile	Spalte	Projekt
1 Das Umwandeln von 'Date' in 'Double' erfordert das Aufrufen der Methode 'Date.ToDouble()'	mdlMain.vb	29	26	Generics
2 Das Umwandeln von 'Double' in 'Date' erfordert das Aufrufen der Methode 'Date.FromOADate()'	mdlMain.vb	29	26	Generics

Abbildung 23.3 Mit *Of* bestimmen Sie, für welchen Datentyp Ihre generische Klasse zur Anwendung kommen soll

Und anhand der Fehlerliste, die Sie ebenfalls in der Grafik sehen können, erkennen Sie auch, dass beide »Typversionen« der Klasse auf ihre Weise typsicher sind. Sie können der einen nur Werte vom Typ Double und der anderen nur Werte vom Typ Date hinzufügen. Jeder Versuch, einer Liste einen jeweils anderen Typ unterzubringen, wird schon zur Entwurfszeit mit einer entsprechenden Fehlermeldung bestraft.

Beschränkungen (Constraints)

Im gezeigten Beispiel können Sie eine `DynamicList` so definieren, dass sie sich aus jedem beliebigen Typ bilden kann. Unter bestimmten Umständen kann das nicht erwünscht sein, und zwar genau dann, wenn Sie innerhalb einer generischen Klasse andere generische Typen verwenden, deren Typ Sie aber zur Entwicklungszeit noch nicht kennen.

BEGLEITDATEIEN

Das folgende Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 23\\Generics02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Beschränkungen für generische Typen auf eine bestimmte Basisklasse

Dazu ein Beispiel: Angenommen, Sie haben eine Anwendung geschaffen, die die verschiedensten Körper (Quader, Kugel, Pyramiden) berechnen, verwalten und darstellen muss. Sie möchten jetzt eine generische Klasse schaffen, die nicht nur die verschiedenen Typen von Körpern in einer Liste wie die `DynamicList` speichert, sondern Sie möchten, dass diese Klasse auch deren Gesamtvolumen berechnen soll.

Nehmen wir weiter an, dass es in unserem Beispiel eine Basisklasse gibt, auf der alle Körperklassen basieren. Diese Basisklasse speichert dann die Position und die Farbe eines Körpers. Die einzelnen Körperklassen leiten anschließend von dieser Körperbasisklasse ab, damit sie die für alle gleich bleibenden Eigenschaften nicht ständig wieder implementieren müssen. Eine solche Klassenerbfolge sieht dann in etwa folgendermaßen aus:

```
Imports System.Drawing

'Stellt die Grundeigenschaften eines Körpers bereit
Public MustInherit Class KörperBasis

    Private myFarbe As Color
    Private myPosition As Point

    MustOverride ReadOnly Property Volumen() As Double

    Public Property Farbe() As Color
        Get
            Return myFarbe
        End Get
        Set(ByVal value As Color)
            myFarbe = value
        End Set
    End Property

    Public Property Position() As Point
        Get
            Return myPosition
        End Get
        Set(ByVal value As Point)
            myPosition = value
        End Set
    End Property
End Class

'Stellt die Grundeigenschaften eines Körpers bereit
Public Class Quader
    Inherits KörperBasis

    Private mySeitenLänge_a As Double
    Private mySeitenLänge_b As Double
    Private mySeitenLänge_c As Double

    Sub New(ByVal a As Double, ByVal b As Double, ByVal c As Double)
        mySeitenLänge_a = a
        mySeitenLänge_b = b
        mySeitenLänge_c = c
    End Sub
End Class
```

```

Public Overrides ReadOnly Property Volumen() As Double
    Get
        Return mySeitenLänge_a * mySeitenLänge_b * mySeitenLänge_c
    End Get
End Property
End Class

Public Class Pyramide
    Inherits KörperBasis

    Private myGrundfläche As Double
    Private myHöhe As Double

    Sub New(ByVal Grundfläche As Double, ByVal Höhe As Double)
        myGrundfläche = Grundfläche
        myHöhe = Höhe
    End Sub

    Public Overrides ReadOnly Property Volumen() As Double
        Get
            Return (myGrundfläche * myHöhe) / 3
        End Get
    End Property
End Class

```

Rein theoretisch könnten wir natürlich nun die bereits vorhandene `DynamicList`-Klasse für die Speicherung der Körper-Objekte verwenden, wie die folgende Abbildung zeigt:

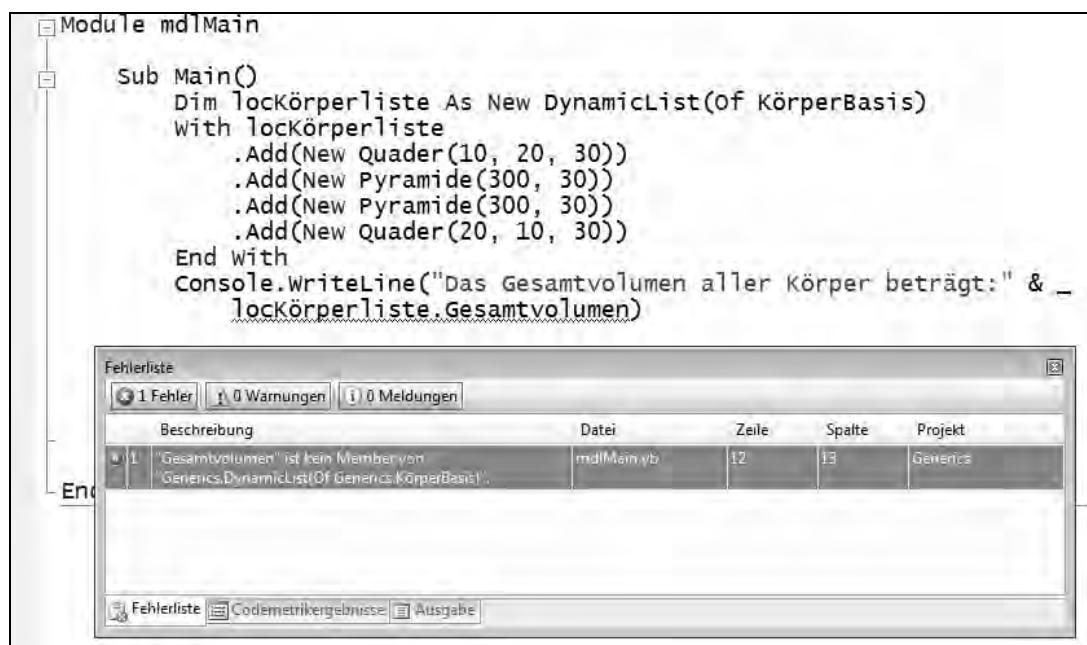


Abbildung 23.4 Die `DynamicList` soll auch eine `Gesamtvolume`-Eigenschaft zur Verfügung stellen, doch die ist zurzeit noch nicht implementiert

Doch dabei ergibt sich eine Kette von Problemen. Wie in Abbildung 23.4 zu sehen, möchten wir eine Eigenschaft der Liste verwenden, die es zu diesem Zeitpunkt noch nicht gibt. Und mit herkömmlichen Mitteln haben wir auch leider keine Chance, diese Eigenschaft zu implementieren, denn:

Innerhalb der generischen `DynamicList`-Klasse müssten wir eine Eigenschaft `Gesamtvolumen` erschaffen, die durch alle Elemente iteriert, die sie hält, und deren `Volumen`-Eigenschaft abfragt. Doch genau eine solche Prozedur können wir nicht implementieren, wie die folgende Grafik zeigt:

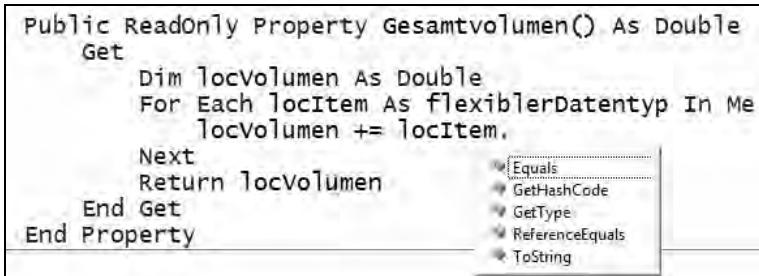


Abbildung 23.5 Die Eigenschaft, die zur Berechnung des Gesamtvolumens benötigt wird, ist nicht erreichbar!

Wenn wir die Schleife zum Iterieren durch die Elemente der `DynamicList` erstellen, dann müssen wir natürlich darauf achten, dass ein einzelnes Element dieser Iteration vom gleichen Typ ist, wie für die gesamte generische Klasse – und damit muss es vom Typ `flexiblerDatentyp` sein (andernfalls wäre die Klasse nicht mehr generisch, also allgemeingültig anwendbar).

`flexiblerDatentyp` kann aber jeder beliebige Datentyp sein, und deswegen sind auch nur die Eigenschaften und Methoden erreichbar, die von jedem Datentyp *gleichermaßen* zur Verfügung gestellt werden. Das sind logischerweise genau die Eigenschaften und Methoden, die `Object` zur Verfügung stellt und die jede neue Klasse implizit erbt. Das wiederum sind die Elemente, die Ihnen IntelliSense, wie in Abbildung 23.5 zu sehen ist, anzeigt.

Das ist nun der richtige Zeitpunkt, Beschränkungen ins Spiel zu bringen. Wenn wir dem generischen Datentyp »sagen«, »pass auf, du darfst nur solche Datentypen als Typparameter akzeptieren, die auf Körperbasis basieren«, dann kann das Framework ohne Probleme die Methoden und Eigenschaften über `locItem` anbieten, der vom Typ `flexiblerDatentyp` ist, die durch Körperbasis implementiert werden.

Diese Änderungen nehmen wir als Nächstes vor, allerdings nicht in der Klasse `DynamicList` selbst, denn: Damit wir nun nicht unsere `DynamicList` für alle Zeiten nur noch auf die Verwaltung von Körper-Objekten limitieren, implementieren wir diese Änderungen in einer Klasse, die identisch zur `DynamicList`-Klasse ist, jedoch nur die benötigten Änderungen noch zusätzlich innehaltet. Diese Klasse nennen wir `DynamicListKörper`, und die sieht folgendermaßen aus:

```

Class DynamicListKörper(Of flexibleDatentyp As KörperBasis)
    Implements IEnumerable

    Protected myStep As Integer = 4          ' Schrittweite, um die das Array erhöht wird.
    Protected myCurrentArraySize As Integer ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer   ' Zeiger auf aktuelles Element
    Protected myArray() As flexibleDatentyp ' Array mit den Elementen.

```

```

Sub New()
    myCurrentArraySize = myStep
    ReDim myArray(myCurrentArraySize - 1)
End Sub

Sub Add(ByVal Item As flexibleDatentyp)
    .
    .
    .
End Sub

Public ReadOnly Property GesamtVolumen() As Double
    Get
        Dim locVolumen As Double
        For Each locItem As flexibleDatentyp In Me
            locVolumen += locItem.Volumen
        Next
        Return locVolumen
    End Get
End Property
.
.
.

```

Aus Platzgründen sehen Sie in diesem Listing lediglich die Änderungen im Vergleich zur Klasse `DynamicList`. Sie können in der ersten Zeile des Klassenlistings sehen, wie Beschränkungen implementiert werden: Neben der Erweiterung (`Of flexibleDatentyp`, die den Typparameter für die weitere Verwendung des generischen Typs in der Klasse festlegt, gibt es obendrein den Zusatz `as KörperBasis`), der nun bestimmt, dass ausschließlich Klassen, die von `KörperBasis` abgeleitet wurden, als Basis für die Erstellung des Datentyps `DynamicListKörper` dienen dürfen. Das befähigt uns jetzt auch, die Eigenschaft `GesamtVolumen` zu implementieren. Da wir die Datentypen für die generische Klasse auf `KörperBasis` und deren Ableitungen beschränken, weiß das Framework, dass es alle Methoden, die `KörperBasis` anbietet, sicher für alle Objekte zur Verfügung stellen kann, die auf `flexibleDatentyp` basieren.

Nach der Implementierung dieser neuen Klasse, stellen wir das Testprogramm im Modul `mdlMain.vb` entsprechend um:

```

Module mdlMain

Sub Main()
    Dim locKörperliste As New DynamicListKörper(Of KörperBasis)
    With locKörperliste
        .Add(New Quader(10, 20, 30))
        .Add(New Pyramide(300, 30))
        .Add(New Pyramide(300, 30))
        .Add(New Quader(20, 10, 30))
    End With
    Console.WriteLine("Das Gesamtvolumen aller Körper beträgt:" & _
        locKörperliste.GesamtVolumen)

```

```

Console.WriteLine()
Console.WriteLine("Taste drücken zum Beenden!")
Console.ReadKey()
End Sub

End Module

```

Übrigens: Dass sich die Klasse wirklich nur noch verwenden lässt, wenn Sie sie tatsächlich auf einer Ableitung von KörperBasis basieren lassen, zeigt die folgende Abbildung.



Abbildung 23.6 Eine beschränkte generische Klasse können Sie tatsächlich nur noch auf Basis eines Datentyps instanziiieren, der die Beschränkung erfüllt

HINWEIS Das Framework erlaubt es nicht, mehrere Basisklassen als Beschränkung für einen generischen Typ zu definieren. Das würde die Technik der Mehrfachvererbung implizieren, die das Framework in der vorliegenden 2.0-Version nicht beherrscht (und wahrscheinlich auch nie beherrschen wird).

Beschränkungen auf Klassen, die bestimmte Schnittstellen implementieren

Ungleich flexibler können Sie generische Klassen gestalten, wenn sich diese in ihren Beschränkungen nicht auf bestimmte Basisklassen, sondern nur auf bestimmte Schnittstellen beschränken.

Darüber hinaus haben Schnittstellen den Vorteil, sich bereits in vielen »fertigen« Typen des Frameworks »zu befinden«, sodass Sie generische Klassen erstellen können, die nicht nur auf Ihren eigenen Typen basieren (deren Einschränkungen Sie ja gut steuern können, da Sie über deren Quellcode verfügen); vielmehr können Sie auch die Typen des Frameworks verwenden, die sich, da sie die unterschiedlichsten Schnittstellen implementieren, ebenfalls sehr gut selektieren lassen.

Ein Beispiel: Sie möchten die `DynamicList`-Klasse um eine Sortierfunktion erweitern. Zu diesem Zweck müssen Sie die Elemente, die die `DynamicList`-Klasse speichert, miteinander vergleichen können. Das Framework stellt für Typen, deren einzelne Instanzen sich miteinander vergleichen lassen sollen, die `IComparable`-Schnittstelle bereit. Wenn ein Typ diese Schnittstelle einbindet, zwingt ihn diese Schnittstelle damit auch, eine Funktion namens `CompareTo` einzubinden. Und wenn Sie eine generische Klasse schaffen, dann können Sie die Typen, aus denen diese hervorgehen soll, auch auf die `IComparable`-Schnittstelle beschränken. Damit bleibt die Klasse generisch, und kann dennoch jeden beliebigen Datentyp typsicher speichern – jedenfalls solange er die `IComparable`-Schnittstelle selbst implementiert. Wenn er das allerdings macht, können Sie auch vom Vorhandensein einer `CompareTo`-Funktion sicher ausgehen und damit beispielsweise eine Sortierfunktion in der generischen Klasse implementieren. Die `IComparable`-Schnittstelle wird übrigens von allen primitiven Datentypen wie `String`, `Long`, `Decimal`, `Date` etc. eingebunden.

BEGLEITDATEIEN

Ein Beispiel, das diesen Sachverhalt verdeutlichen soll, finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 23\\Generics03

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Schauen wir uns die veränderte (und aus Platzgründen wieder gekürzte) Version der `DynamicList` an, die nun den Namen `DynamicListSortable` trägt, und die – neben der Einschränkung bei der Definition des Klassennamens – um eine Sort-Methode ergänzt wurde:

```
Class DynamicListSortable(Of flexibleDatentyp As IComparable)
    Implements IEnumerable

    Protected myStep As Integer = 4      ' Schrittweite, die das Array erhöht wird.
    Protected myCurrentArraySize As Integer ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer   ' Zeiger auf aktuelles Element
    Protected myArray() As flexibleDatentyp ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub

    Sub Add(ByVal Item As flexibleDatentyp)
        .
        .
        .
    End Sub

    'Sortiert die Elemente, die die DynamicListSortable speichert
    Public Sub Sort()
        Dim locÄußererZähler, locInnererZähler As Integer
        Dim locDelta As Integer
        Dim locItemTemp As flexibleDatentyp

        locDelta = 1

        'Größten Wert der Distanzfolge ermitteln
        Do
```

```

locDelta = 3 * locDelta + 1
Loop Until locDelta > myCurrentCounter

Do
    locDelta \= 3

    'Shellsort's Kernalgorithmus
    For locÄuBererZähler = locDelta To myCurrentCounter - 1
        locItemTemp = Me.Item(locÄuBererZähler)
        locInnererZähler = locÄuBererZähler
        Do While (Me.Item(locInnererZähler - locDelta).CompareTo(locItemTemp) > 0)
            Me.Item(locInnererZähler) = Me.Item(locInnererZähler - locDelta)
            locInnererZähler = locInnererZähler - locDelta
            If (locInnererZähler <= locDelta) Then Exit Do
        Loop
        Me.Item(locInnererZähler) = locItemTemp
    Next
Loop Until locDelta = 0
End Sub

```

Möglich wird die Implementierung eines Sortieralgorithmus erst wegen der Beschränkung auf Typen, die die `IComparable`-Schnittstelle einbinden. Doch da die Typen die Schnittstelle einbinden müssen (denn andernfalls wäre gar keine Instanzierung der `DynamicListSortable` möglich), kann die Sort-Methode auch gefahrlos auf die `CompareTo`-Methode jedes Elements zurückgreifen (siehe fett hervorgehobene Zeile im oben stehenden Listing).

Da, wie schon gesagt, beispielsweise alle primitiven Datentypen in .NET `IComparable` einbinden, können wir nun diese auch mit unserer Liste verwenden, wie der Modulcode im Folgenden zeigt:

```

Module mdlMain

Sub Main()
    Dim locDoubleList As New DynamicListSortable(Of Double)
    locDoubleList.Add(124)
    locDoubleList.Add(1243)
    locDoubleList.Add(24)
    locDoubleList.Add(14)
    locDoubleList.Add(1)
    locDoubleList.Add(-32)
    locDoubleList.Add(231)
    locDoubleList.Add(143)

    locDoubleList.Sort()
    For Each locItem As Double In locDoubleList
        Console.WriteLine(locItem)
    Next
    Console.WriteLine()

    Dim locStringList As New DynamicListSortable(Of String)
    locStringList.Add("Klaus")
    locStringList.Add("Arnold")
    locStringList.Add("Sarah")
    locStringList.Add("Christiane")
    locStringList.Add("Jürgen")

```

```
locStringList.Add("Uta")
locStringList.Add("Helge")
locStringList.Add("Uwe")

locStringList.Sort()
For Each locItem As String In locStringList
    Console.WriteLine(locItem)
Next

Console.WriteLine()
Console.WriteLine("Taste drücken zum Beenden!")
Console.ReadKey()
End Sub

End Module
```

Auch hier sind die Zeilen in Fetschrift die eigentlich interessanten. Sie demonstrieren einerseits, dass die generische Liste auf unterschiedlichen Datentypen basieren kann und andererseits, dass dennoch solch komplexe Funktionen wie das Sortieren funktionieren, obwohl zum Zeitpunkt der Entwicklung der Liste noch gar nicht bekannt ist, mit welchen Typen es die Liste später zu tun haben wird.

Dass dieses Konzept auch funktioniert, zeigt die Ausführung des Programms, die folgendes Ergebnis ins Konsolenfenster zaubert:

```
-32
1
14
24
124
143
231
1243

Arnold
Christiane
Helge
Jürgen
Klaus
Sarah
Uta
Uwe

Taste drücken zum Beenden!
```

HINWEIS Einen Sortieralgorithmus in eigenen Auflistungsklassen zu implementieren ist im Übrigen genau so überflüssig, wie eigene Auflistungsklassen von Grund auf neu zu kreieren. Das Framework kennt nämlich eine Vielzahl von Auflistungsklassen für die unterschiedlichsten Zwecke. Doch es ist allemal interessant zu sehen, wie das Prinzip von Auflistungsklassen an sich funktioniert, und für das bessere Verständnis des vorherigen Kapitels, das die wichtigsten Auflistungen im .NET Framework vorstellt, bestimmt von Vorteil.

Beschränkungen auf Klassen, die über einen Standardkonstruktor verfügen

In einigen Fällen ist es notwendig, dass eine generische Klasse in der Lage ist, den Typ, auf dem sie basieren soll, auch zu instanziieren. Das kann sie dann nicht, wenn es sich beim Typ, auf dem sie basiert, um eine abstrakte Klasse handelt, um eine Schnittstelle oder um eine Klasse, die ausschließlich über parametisierte Konstruktoren verfügt.

Wenn Sie diese Fälle für den Typ ausschließen möchten, den eine generische Klasse einbindet, müssen Sie eine Beschränkung definieren, die vom Typ, auf dem sie basieren soll, einen Standardkonstruktor erfordert, und das geht folgendermaßen:

```
Public Class GenerischeKlasseMitInstanziierbarenTyp(Of flexibleDatentyp As New)

    Public Sub TestMethode()
        'Das geht nur auf Grund der angegebenen Beschränkung:
        Dim locTest As New flexibleDatentyp

        'Und hier ist locTest jetzt als Datentyp instanziert!
        Console.WriteLine(locTest.ToString)
    End Sub
End Class
```

Beschränkungen auf Wertetypen

Möchten Sie eine generische Klasse auf Wertetypen beschränken, verfahren Sie auf ähnliche Weise, wie im vorherigen Abschnitt beschrieben. Sie bestimmen durch As Structure, dass nur noch Wertetypen (in Visual Basic also Strukturen) innerhalb einer generischen Klasse als Typ zur Anwendung kommen dürfen, die auf ValueType basieren. Ein Beispiel:

```
Public Class GenerischeKlasseNurMitWertetypen(Of flexibleDatentyp As Structure)

    Public Sub TestMethode()
        'Ist Wertetyp – keine Instanzierung durch New erforderlich!
        Dim locWerteTyp As flexibleDatentyp

        'Und hier ist locTest jetzt als Datentyp instanziert.
        Console.WriteLine(locWerteTyp.ToString)
    End Sub
End Class
```

Kombinieren von Beschränkungen und Bestimmen mehrerer Typparameter

In Visual Basic sind alle Beschränkungen für generische Datentypen untereinander kombinierbar. Im Gegensatz zu Beschränkungen bei Basisdatentypen können Sie darüber hinaus auch Beschränkungen für mehrere Schnittstellen bestimmen.

Wenn Sie verschiedene Beschränkungen oder mehrere Schnittstellen für einen generischen Datentyp einrichten, fassen Sie die verschiedenen Vorschriften in geschweiften Klammern zusammen.

Ein Beispiel soll auch diesen Sachverhalt verdeutlichen:

```
Public Class GenerischeBeschränzungskombi(Of flexiblerDatentyp As {Structure, IComparable, IDisposable})

    Public Sub TestMethode()
        Dim locWerteTyp As flexiblerDatentyp
        Dim locWerteTyp2 As flexiblerDatentyp

        'Direkt verwendbar, da Wertetyp durch Struktur
        'Vergleichbar, dank IComparable
        locWerteTyp.CompareTo(locWerteTyp2)

        'Disposable, dank IDisposable
        locWerteTyp.Dispose()
        locWerteTyp2.Dispose()
    End Sub
End Class
```

Zusätzlich können Sie eine generische Klasse auch für die Verwendung von mehreren Typparametern einrichten. Falls Sie beispielsweise eine Auflistung entwickeln möchten, die als ein Wörterbuch fungiert, dann benötigen Sie einen Typ zum Nachschlagen (den Schlüssel) und einen für den eigentlichen Wert. (Programmieren Sie das aber nicht selbst, denn auch das gibt es schon beispielsweise mit der generischen KeyedCollection-Klasse, die sich im System.Collection.ObjectModel-Namespace befindet.) Die Einschränkungen lassen sich dann für jeden Typparameter einzeln festlegen:

```
Public Class GenerischesWörterbuch(Of Schlüsseltyp As {Structure, IComparable}, _
                                    Werttyp As {New, IComparable, IDisposable})

    Public Sub TestMethode()
        Dim locWerteTyp As Schlüsseltyp
        Dim locWerteTyp2 As Werttyp

        'Direkt verwendbar, da Wertetyp durch Struktur
        'Vergleichbar, dank IComparable
        locWerteTyp.CompareTo(locWerteTyp2)

        'Disposable, dank IDisposable
        locWerteTyp2.Dispose()
    End Sub
End Class
```

Generische Auflistungen (Generic Collections)

Generische Auflistungen haben im Vergleich zu »herkömmlichen« Auflistungen, die Sie im vorherigen Kapitel kennen gelernt haben, einen entscheidenden Vorteil: Sie sind grundsätzlich typsicher. Anders als »normale« Auflistungen wie beispielsweise die ArrayList-Klasse nehmen sie, da sie nicht auf Object basieren,

nicht jeden Datentyp als Element entgegen, sondern beschränken sich auf den Datentyp, der ihrer Definition zugrunde liegt.

Das bedeutet: Definieren Sie beispielsweise eine Collection auf Basis von Integer, etwa mit

```
Dim locGenColl As New Collection(Of Integer)
```

dann laufen Sie nicht Gefahr, später versehentlich der Auflistung ein Element hinzuzufügen, das nicht vom Typ Integer ist – oder mit anderen Worten: Die Zeile

```
locGenColl.Add("Ein Element")
```

würde bereits zur Entwurfszeit im Editor als fehlerhaft gekennzeichnet.

Ihre Programme werden durch den Einsatz von generischen Auflistungen somit robuster, und auch die Entwicklungszeit reduziert sich, da Sie sich nicht mit Laufzeitfehlern herumärgern müssen, sondern bereits zur Entwurfszeit Fehler korrigieren können. Und weniger Programmtests bedeuten weniger Entwicklungszeit und -kosten.

Nun gibt es nicht wenige generische Auflistungen seit dem Framework 2.0, und sie alle im Detail zu beschreiben ist nicht nur unnötig – schließlich funktionieren viele von ihnen wie ihre nicht-generischen Verwandten – es würde auch den Rahmen des Buchs sprengen.

Aus diesem Grund möchte ich mich auf die in meinen Augen wichtigen Besonderheiten beschränken, die Sie bei nicht-generischen Auflistungen nicht antreffen. Eine Tabelle, die Sie im Folgenden finden, gibt darüber hinaus Auskunft, welche generischen Auflistungen Ihnen zur Verfügung stehen, und zu welchem Zweck sie dienen.

Namespace	Auflistung	Beschreibung
System.Collection.ObjectModel	Collection(Of type)	<p>Stellt eine Standardauflistung für die einfache, unsortierte Verwaltung von Elementen eines bestimmten Typs zur Verfügung.</p> <p>Besonderheiten: Im Gegensatz zu List(Of type) gibt es überschreibbare Methoden, mit denen man das Verhalten beim Einfügen, Löschen und Neuzuweisen von Elementen der Auflistung in abgeleiteten Klassen beeinflussen kann.</p>
System.Collection.ObjectModel	KeyedCollection(Of key, type)	<p>Stellt eine Auflistung von Elementen zur Verfügung, die sowohl das Nachschlagen über einen Schlüssel (Key) eines bestimmten Typs sowie den Einsatz eines Indexes erlaubt.</p> <p>Besonderheiten: Diese Auflistung können Sie nur in Ableitungen verwenden, da der Typ, den Sie speichern, selber einen Standard-Schlüssel erzeugen muss, und für diesen Umstand müssen Sie in der Ableitung sorgen.</p> <p>WICHTIG: Vermeiden Sie den Einsatz von numerischen Schlüsseln (Integer, Long, etc.), da es hier beim Serialisieren der Auflistung zu Problemen kommt (Stand: Framework-Version 2.0.50727). Kapitel 24 erklärt im Zusatz zu KeyedCollection weitere Details. ►</p>

Namespace	Auflistung	Beschreibung
System.Collection.ObjectModel	ReadOnlyCollection(Of type)	<p>Stellt eine Auflistung dar, deren Elemente nur gelesen werden können.</p> <p>Besonderheiten: Elemente, die in einer anderen generischen Auflistung gleichen Typs vorliegen, können nur bei der Instanzierung im Konstruktor dieser Auflistung übergeben werden. Ansonsten sind die Elemente nur lesbar und können nach der Instanzierung nicht mehr verändert werden.</p>
System.Collections.Generic	Dictionary(Of key, type)	<p>Stellt eine Auflistung von Schlüsseln und Werten dar.</p> <p>Besonderheiten: Stellt eine Zuordnung von einem Satz von Schlüsseln zu einem Satz von Werten bereit. Jede Hinzufügung zum Wörterbuch besteht aus einem Wert und dem zugeordneten Schlüssel. Ein Wert kann anhand des zugehörigen Schlüssels sehr schnell abgerufen werden (beinahe ein O(1)-Vorgang), da die Dictionary-Klasse in Form einer Hashtable implementiert ist. Mehr zum Konzept von Hashtables finden Sie im vorherigen Kapitel.</p>
System.Collections.Generic	LinkedList(Of type)	<p>Stellt eine doppelt verknüpfte Liste dar.</p> <p>Besonderheiten: Ist eine verknüpfte Liste mit einzelnen Knoten vom Typ LinkedListNode; das Einfügen und Entfernen einzelner Elemente geht extrem schnell vonstatten.</p>
System.Collections.Generic	List(Of type)	<p>Stellt eine Standardauflistung für die einfache, unsortierte Verwaltung von Elementen eines bestimmten Typs zur Verfügung.</p> <p>Hinweis: Diese Klasse eignet sich nicht für Ableitungen in eigenen Auflistungsklassen, bei denen Sie mit Code Einfluss auf die Bearbeitung der Liste nehmen müssen. Verwenden Sie stattdessen die Collection(Of)-Auflistung (siehe oben).</p>
System.Collections.Generic	Queue(Of type)	<p>Stellt eine FIFO-Auflistung (First-In-First-Out) von Objekten dar.</p> <p>Hinweis: Prinzipiell funktioniert diese Auflistung wie ihr nicht generischer Verwandter. Mehr zur nicht generischen Version dieser Auflistung finden Sie im vorherigen Kapitel.</p>
System.Collections.Generic	SortedDictionary(Of key, type)	<p>Stellt eine Auflistung von Schlüssel-Wert-Paaren dar, deren Reihenfolge anhand des Schlüssels bestimmt wird.</p>
System.Collections.Generic	SortedList(Of key, type)	<p>Verwaltet eine sortierte Liste, deren Elemente über Schlüssel abrufbar sind.</p> <p>Hinweis: Prinzipiell funktioniert diese Auflistung wie ihr nicht generischer Verwandter. Mehr zur nicht generischen Version dieser Auflistung finden Sie im vorherigen Kapitel.</p> <p>Im Gegensatz zu SortedDictionary erfolgt die Sortierung über das Element und nicht über den Schlüssel!</p>
System.Collections.Generic	Stack(Of type)	<p>Stellt eine LIFO-Auflistung (Last-In-First-Out) von Objekten dar.</p> <p>Hinweis: Prinzipiell funktioniert diese Auflistung wie ihr nicht generischer Verwandter. Mehr zur nicht generischen Version dieser Auflistung finden Sie im vorherigen Kapitel.</p>

Tabelle 23.1: Die wichtigsten generischen Auflistungstypen

KeyedCollection – Schlüssel/Wörterbuch-Auflistung mit zusätzlichen Index-Abrufen

Ich gebe zu, ich bin ein großer Fan von KeyedCollection – sie ist in meinen eigenen Projekten neben der Collection(0f) am häufigsten anzutreffen. Warum? Sie hat zwei entscheidende Vorteile:

- Sie erlaubt es, einen beliebigen Typ (na ja – *fast* beliebigen Typ, doch dazu später mehr) als Schlüssel anzugeben. Damit können Sie sie als Wörterbuchaufstellung verwenden und ihre Elemente beispielsweise über einen Matchcode abfragen.
- Sie können sie zusätzlich wie eine ganz normale Collection behandeln, also mit For/Each durch ihre Elemente iterieren oder auch einzelne Elemente über einen numerischen Index abrufen.

Da KeyedCollection auch einen Schlüsselwert zum Abrufen jedes ihrer Elemente braucht, benötigt sie einen Mechanismus, der aus dem Element, das es hinzuzufügen gilt, einen Schlüssel erzeugt. Aus diesem Grund ist die KeyedCollection-Klasse auch eine abstrakte Basisklasse: Sie müssen die KeyedCollection vererben und ihre Methode GetKeyForItem überschreiben, in der Sie dann regeln, wie aus dem Element, das es hinzuzufügen gilt, ein eindeutiger Schlüssel kreiert werden soll. Die KeyedCollection-Klasse sollte deswegen nur Elemente verwalten, die über wenigstens eine Eigenschaft verfügen, mit der sich ein Element eindeutig von einem anderen unterscheiden lässt. Das kann beispielsweise eine Personalnummer, eine Kundennummer, ein eindeutiger Matchcode oder Ähnliches sein.

WICHTIG KeyedCollection hat aber auch einen entscheidenden Nachteil, oder besser: einen Design-Fehler. Wenn Sie ihren Inhalt serialisieren – also beispielsweise mit einem bestimmten im Framework eingebauten Mechanismus als XML-Datei abspeichern –, können Sie als Schlüssel *keinen* Integer-Datentyp verwenden, da das Framework beim Serialisierungsversuch mit einer Ausnahme »aussteigt«. Kapitel 24 geht näher auf dieses Problem ein und hält auch einen Workaround zu dieser »Design-Unzulänglichkeit« bereit.

BEGLEITDATEIEN Sie finden das Projekt für das folgende Beispiel im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 23\\KeyedCollectionDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Im Projekt finden Sie zwei Codedateien, *Daten.vb* und *KeyedCollection.vb*. *Daten.vb* enthält eine Adressenklasse, die Sie aus dem vorherigen Kapitel schon kennen. Aufgabe dieser Klasse ist es, die Daten einer Adresse zu speichern. Sie stellt darüber hinaus auch eine statische Funktion bereit, die eine bestimmbare Anzahl von Zufallsadressen generiert und in einer ArrayList zurückliefert (mehr zu ArrayList finden Sie ebenfalls im vorherigen Kapitel).

Damit die Klasse Adresse in diesem Beispiel als Verwaltungselement für die KeyedCollection in Frage kommt, muss sie eine Anforderung erfüllen: Sie muss über eine Eigenschaft verfügen, mit der sich ein Adresse-Element von jedem anderen Adresse-Element unterscheiden lässt.

HINWEIS Die Funktionalität zur Prüfung auf Eindeutigkeit kann die Adresse-Klasse natürlich nicht selbst implementieren, da sie die anderen Elemente einer Auflistung nicht kennt; deswegen ist die Bestimmung einer solchen Dateneigenschaft an dieser Stelle nur eine theoretische Definition. Es ist Aufgabe des späteren Hauptprogramms zu prüfen, ob es keine »Schlüsseldoublette« in einer Elementauflistung gibt, bevor ein neues Element (wie Adresse) der Auflistung hinzugefügt wird.

KeyedCollection ist eine *abstrakte* Basisklasse, was bedeutet, dass Sie sie nicht direkt verwenden können. Sie müssen sie vererben und um eine bestimmte Funktionalität erweitern. Deswegen definieren Sie – wie Sie es von abstrakten Klassen gelernt haben – zunächst einen neuen Klassennamen, etwa mit

```
Public Class Adressen
```

und fügen darunter die Zeile

```
Inherits KeyedCollection(Of String, Adresse)
```

ein. Mit dieser Zeile leiten Sie von KeyedCollection ab und bestimmten gleichzeitig, dass String der Typ für den Schlüssel und Adresse der Typ der zu verwaltenden Elemente sein soll.

In dem Moment, in dem Sie die Zeile mit abschließen, fügt der Visual Basic-Editor automatisch den Rumpf der Methode ein, den Sie in der Basisklasse überschreiben müssen:

```
Protected Overrides Function GetKeyForItem(ByVal item As Adresse) As String
End Function
```

Mit dieser Funktion »holt« sich KeyedCollection den Wert, der vom Schlüsseldatentyp sein muss, immer dann, wenn Sie ein neues Element der Auflistung (beispielsweise durch Add) hinzufügen oder ein Element verändern (zum Beispiel mit Item(Index)=New Adresse(...)). So wird sichergestellt, dass jedes Element über einen eindeutigen Schlüssel verfügt.

In unserem Beispiel gibt es eine Eigenschaft in der Klasse Adresse, die ein Element vom Typ Adresse eindeutig kennzeichnet: *Matchcode*. Diese Eigenschaft ist vom Typ String – also ist der Schlüsseltyp für die KeyedCollection ebenfalls String. Die komplett implementierte KeyedCollection sieht also folgendermaßen aus:

```
''' <summary>
''' Verwaltet Objekte vom Typ Adresse als Wörterbuch-Auflistung.
''' Abgeleitet von der abstrakten Basisklasse KeyedCollection.
''' </summary>
''' <remarks></remarks>
Public Class Adressen
    Inherits KeyedCollection(Of String, Adresse)

    'Diese Methode muss überschrieben werden, um zu garantieren,
    'dass für jedes Element ein Schlüssel existiert.
    Protected Overrides Function GetKeyForItem(ByVal item As Adresse) As String
        'Hier wird bestimmt, dass der eindeutige Key der Matchcode ist.
        Return item.Matchcode
    End Function
End Class
```

Diese Implementierung ist auch schon alles, was Sie brauchen, um anschließend mit der KeyedCollection arbeiten zu können. Das Hauptmodul *KeyedCollection.vb* demonstriert den Umgang und stellt insbesondere heraus, dass Elemente einer auf KeyedCollection basierenden Klasse sowohl über einen Index als auch über den entsprechenden Schlüssel abgerufen und bearbeitet werden können:

```
Imports System.Collections.ObjectModel

Module KeyedCollection

Sub Main()
    'Erstmal eine normale ArrayList definieren,
    'die aus 100 Zufallsadressen besteht.
    Dim locArrayList As ArrayList = Adresse.ZufallsAdressen(100)

    'Das ist die "selbst gestrickte" KeyedCollection
    Dim locKeyedAdressen As New Adressen

    'Mit den Elementen der ArrayList füllen wir die KeyedCollection
    For Each locAdressItem As Adresse In locArrayList
        locKeyedAdressen.Add(locAdressItem)
    Next

    'Abrufen der Elemente aus der KeyedCollection:
    'Variation Nr.1 - abrufen über den Index
    For c As Integer = 0 To 10
        Console.WriteLine(locKeyedAdressen(c).ToString)
    Next

    'Leerzeile - der besseren Übersicht wegen:
    Console.WriteLine()

    'Irgendeine herauspicken, um an den Matchcode zu kommen:
    Dim locMatchcode As String = locKeyedAdressen(10).Matchcode

    'Variation Nr. 2: Eine Adresse kann auch über den Key
    '(in diesem Fall über den Matchcode) abgerufen werden.
    Dim locAdresse As Adresse = locKeyedAdressen(locMatchcode)
    Console.WriteLine("Die Adresse mit dem Matchcode " & locMatchcode & " lautet:")
    Console.WriteLine(locAdresse.ToString)
    Console.WriteLine()

    Console.WriteLine("Taste drücken zum Beenden!")
    Console.ReadKey()

End Sub

End Module
```

Elementverkettungen mit LinkedList(Of)

Erwähnenswert bei den generischen Auflistungen ist die `LinkedList`-Klasse, die das erste Mal mit dem .NET Framework 2.0 die Möglichkeit zur Verwaltung von Elementen in Form von verketteten Listen einführt.

`LinkedList` stellt eine verknüpfte Liste mit einzelnen Knoten vom Typ `LinkedListNode` dar. Durch das Konzept von `LinkedList`, bei dem die zu speichernden Elemente jeweils selbst auf das nächste und das vorherige Elemente »zeigen«, kann das Einfügen bzw. Löschen von Elementen sehr schnell vonstatten gehen, da – sinnbildlich – die Kette an einer Stelle nur aufgeknüpft werden muss, um ein neues Glied in die Kette einzufügen.

Durch die Art der Elementverwaltung bietet `LinkedList` Methoden und Eigenschaften, die Sie in anderen Auflistungen nicht finden. Die folgende Tabelle gibt Aufschluss über die besonderen Methoden und Eigenschaften der `LinkedList`-Klasse.

Elemente werden listenintern nicht wie bei anderen Auflistungsklassen direkt gespeichert, sondern in einem Knoten-Objekt namens `LinkedListNode` abgelegt, das für die Verkettung zum nächsten bzw. vorherigen Knoten der Liste sorgt. Auch diese Klasse ist generisch ausgelegt. Knoten, die der Liste hinzugefügt werden, müssen dabei auf Basis desselben Typs definiert werden, wie die Liste selbst.

Methode	Beschreibung
<code>AddAfter</code>	Fügt nach einem vorhandenen Knoten in der <code>LinkedList</code> einen neuen Knoten oder Wert hinzu.
<code>AddBefore</code>	Fügt vor einem vorhandenen Knoten in der <code>LinkedList</code> einen neuen Knoten oder Wert hinzu.
<code>AddFirst</code>	Fügt am Anfang der <code>LinkedList</code> einen neuen Knoten oder Wert hinzu.
<code>AddLast</code>	Fügt am Ende der <code>LinkedList</code> einen neuen Knoten oder Wert hinzu.
<code>Find</code>	Sucht den ersten Knoten, der den angegebenen Wert enthält.
<code>FindLast</code>	Sucht den letzten Knoten, der den angegebenen Wert enthält.
<code>First</code>	Ruft den ersten Knoten der <code>LinkedList</code> ab. Diese Eigenschaft kann nur gelesen werden.
<code>Last</code>	Ruft den letzten Knoten der <code>LinkedList</code> ab. Diese Eigenschaft kann nur gelesen werden.
<code>Remove</code>	Entfernt das erste Vorkommen eines Knotens oder Werts aus der <code>LinkedList</code> .
<code>RemoveFirst</code>	Entfernt den Knoten am Anfang der <code>LinkedList</code> .
<code>RemoveLast</code>	Entfernt den Knoten am Ende der <code>LinkedList</code> .

Tabelle 23.2 Die besonderen Methoden und Eigenschaften der `LinkedList`-Klasse

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 23\\LinkedList

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module LinkedList

Sub Main()
    Dim locLinkedList As New LinkedList(Of Adresse)
    Dim locAdressen As ArrayList = Adresse.ZufallsAdressen(50)
    Dim locAdresse As Adresse = Nothing

    'Acht Adressen an das jeweilige Ende der LinkedList anfügen
    For c As Integer = 0 To 49

        'den 25. merken wir uns für die spätere Suche in der Liste
        If c = 25 Then
            locAdresse = DirectCast(locAdressen(c), Adresse)
        End If
    Next
End Sub
```

```
    locLinkedList.AddLast(DirectCast(locAdressen(c), Adresse))
Next

Dim locLinkedListNode As LinkedListNode(Of Adresse)
'Den Knoten finden, der dem 25. Adresseneintrag entsprach.
locLinkedListNode = locLinkedList.Find(locAdresse)
Console.WriteLine("Vor " & locLinkedListNode.Value.ToString & " kommt " &
                  locLinkedListNode.Previous.Value.ToString &
                  " und danach kommt " & locLinkedListNode.Next.Value.ToString)
Console.WriteLine()

'Eine neue Adresse vor dem 25. einfügen:
Console.WriteLine("Sarah Halek vorher einfügen!")
locLinkedList.AddBefore(locLinkedListNode,
                        New Adresse("SasiMatch", "Halek", "Sarah", "99999", "Musterhausen"))

Console.WriteLine()

'Das Gleiche nochmal ausgeben, es sollte die Änderungen jetzt widerspiegeln.
Console.WriteLine("Vor " & locLinkedListNode.Value.ToString & " kommt " &
                  locLinkedListNode.Previous.Value.ToString &
                  " und danach kommt " & locLinkedListNode.Next.Value.ToString)
Console.WriteLine()
Console.WriteLine("Taste drücken zum Beenden")
Console.ReadKey()
End Sub
End Module
```

Wenn Sie dieses Programm ausführen, gibt es in etwa Folgendes auf dem Bildschirm aus:

```
Vor "Hörstmann, Hans" kommt "Englisch, Margarete" und danach kommt "Löffelmann, Katrin"
Sarah Halek vorher einfügen!

Vor "Hörstmann, Hans" kommt "Halek, Sarah" und danach kommt "Löffelmann, Katrin"

Taste drücken zum Beenden
```

List(Of)-Auflistungen und Lambda-Ausdrücke

Die `List(Of)`-Klasse ist eine der am häufigsten (wenn nicht sogar die am häufigsten) eingesetzte generische Auflistungsklasse. Wenn Sie primitive Datentypen in einer Auflistung speichern, sollten Sie den Einsatz `List(Of type)` anderen Auflistungsklassen vorziehen – alleine schon aus Performance-Gründen. Das .NET Framework ist nämlich in der Lage, den eigentlich notwendigen Boxing-Vorgang bei `List(Of type)` zu umgehen, und damit ist `List(Of type)` die schnellere Alternative.

Und obwohl es sich bei dieser generischen Auflistung um solch eine populäre Klasse handelt, hält sie Funktionalitäten bereit, über die man – obwohl sie bereits im Framework 2.0 vorhanden waren – jedenfalls im Rahmen von Visual Basic bislang wenig Brauchbares erfahren konnte; wohl auch, weil sie förmlich nach der Verwendung von Lambda-Funktionen schreien, die erst mit der 2008er Version Einzug in Visual Basic hielten. Das folgende Beispiel setzt das Verständnis von Lambda-Funktionen übrigens voraus – und falls Sie sich mit diesen noch nicht auseinander gesetzt haben sollten: Kapitel 14 klärt deren grundsätzliches Verständnis.



Abbildung 23.7 Sortierung und Suche steuern Sie über die Spaltenköpfe in dieser Adressenliste

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 23\\ListOfDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie das Programm starten, sehen Sie einen Dialog, etwa wie in Abbildung 23.7 zu sehen. Die im Formular vorhandene ListView wird mit 50 Zufallsadressen gefüllt. Sie können die Liste sortieren, indem Sie auf den entsprechenden Spaltenkopf klicken – so, wie Sie es vom Windows-Explorer in der Details-Ansicht gewohnt sind.

Eine Suchfunktion, die Sie über die entsprechende Schaltfläche erreichen, gestattet es Ihnen, nach dem Begriff innerhalb der Liste zu suchen. Gefunden wird dabei der Eintrag, dessen Text in der Spalte, nach der zuletzt sortiert wurde, dem eingegebenen Suchbegriff entspricht.

Soweit ist dieses Beispiel noch nichts Besonderes. Doch wenn Sie einen Blick in die Umsetzung riskieren, wird klar, wie alternativ der Lösungsansatz ist. Das geht schon beim Schreiben der Zufallsadressen in die Liste los:

ForEach und die generische Action-Klasse

Wenn Sie den Inhalt eines Arrays oder einer Auflistung in einer Liste darstellen möchten, liegt die Vorgehensweise, um das zu erreichen, eigentlich auf der Hand: Sie iterieren mit einem ForEach-Konstrukt durch die Auflistung, verarbeiten damit jedes Element und bringen es mit geeigneten Methoden oder Eigenschaften in die sichtbare Liste eines Formulars.

Dieses Beispiel nutzt einen anderen Weg, wie im folgenden Listing-Ausschnitt zu sehen:

```
Sub ElementeDarstellen()

    'Unterdrückt Neuzeichnen-Ereignisse bis zum
    'nächsten EndUpdate; dadurch geht der Aufbau
    'der Elemente schneller und wackelt nicht.
    Me.ListViewAdressen.BeginUpdate()

    'Alle Elemente der ListView löschen.
    Me.ListViewAdressen.Items.Clear()

    'Für jedes Element der Liste wird ElementInListe aufgerufen.
    myAdressen.ForEach(New Action(Of Adresse)(AddressOf ElementInListe))

    'So werden die Spaltenbreiten optimal angepasst.
    For Each locCol As ColumnHeader In Me.ListViewAdressen.Columns
        locCol.Width = -2
    Next

    'Aufbau der ListView ist beendet.
    Me.ListViewAdressen.EndUpdate()
End Sub

'Der Action-Delegat: für jedes Element der Liste wird diese Aktion durchgeführt.
Sub ElementInListe(ByVal Element As Adresse)
    'Neues ListView-Element - Name kommt zuerst.
    Dim locLvwItem As New ListViewItem(Element.Name)

    'Die Untereinträge setzen
    With locLvwItem.SubItems
        .Add(Element.Vorname)
        .Add(Element.PLZ)
        .Add(Element.Ort)
    End With

    'Zum Wiederfinden Referenz in Tag
    locLvwItem.Tag = Element

    'Zur Listview hinzufügen
    ListViewAdressen.Items.Add(locLvwItem)
End Sub
```

Sie können die `ForEach`-Methode verwenden, um durch eine Auflistung iterieren *zu lassen* und dabei für jedes Element einen Delegaten aufrufen, der eine bestimmte Aktion ausführt. Genau das Verfahren nutzen wir an dieser Stelle, um die Liste aufzubauen. Der Delegat wird im Beispiel nicht durch ein Delegate-Objekt (mehr zu Delegaten finden Sie in Kapitel 14), sondern durch die direkte Angabe einer Prozedur mithilfe des Address Of-Operators angegeben – der Compiler sorgt dann für den richtigen Ersatz einer Delegatvariablen. Aber natürlich könnten an dieser Stelle auch »manuelle« Delegatvariablen zum Einsatz kommen, die in Abhängigkeit von bestimmten Programmzuständen andere Prozeduren verwenden, und genau darin besteht der Reiz des Einsatzes dieser Verfahren. Und – auch das ist möglich – mit Visual Basic 2008 können Sie anstelle eines Delegaten auch einen Lambda-Ausdruck an dieser Stelle einsetzen.

Wichtig ist in jedem Fall, dass die Routinen, die Sie mithilfe der Action-Klasse aufrufen, den Signaturanspruch des Delegaten erfüllen, den Sie im Konstruktor von Action angeben. Im Fall von ForEach und der Action-Klasse muss es sich bei der Delegatenprozedur um eine Methode handeln, die kein Funktionsergebnis hat (also um eine Visual Basic-Sub) und als Parameter ein Element entgegen nimmt, dessen Typ auch der Basis der generischen Action-Klasse entspricht – Adresse in unserem Beispiel.

Im Ergebnis erreichen wir also, dass für jedes Element des List(Of Adresse)-Objektes myAdressen die Methode ElementInListe aufgerufen wird.

Sort und die generische Comparison-Klasse

Prinzipiell funktioniert das nächste Pärchen ähnlich, das Sie verwenden, wenn Sie ein Array mit der Methode Sort ohne den Einsatz einer speziellen Comparer-Klasse (wie in Kapitel 22 gezeigt) sortieren möchten. In Abwandlung zur ersten Verwendung kommen hier jedoch Lambda-Ausdrücke zum Einsatz, sodass wir uns das buchstäbliche Delegieren an die richtige Sortierungsfunktion mit einer Delegatenvariable sparen können. Den relevanten Codeausschnitt des Beispiels finden Sie im Folgenden:

```
'Wird aufgerufen, wenn eine der Spalten angeklickt wird.
Private Sub lvwAdressen_ColumnClick(ByVal sender As Object, ByVal e As _
    System.Windows.Forms.ColumnEventArgs) Handles lvwAdressen.ColumnClick

    'Spaltennummer, die in e.Column steht, in AdressenSortierenNach konvertieren
    mySortierenNach = CType(e.Column, AdressenSortierenNach)

    Select Case mySortierenNach
        Case AdressenSortierenNach.Name
            myAdressen.Sort(Function(adr1 As Adresse, adr2 As Adresse) _
                String.Compare(adr1.Name, adr2.Name))

        Case AdressenSortierenNach.Vorname
            myAdressen.Sort(Function(adr1 As Adresse, adr2 As Adresse) _
                String.Compare(adr1.Vorname, adr2.Vorname))

        Case AdressenSortierenNach.PLZ
            myAdressen.Sort(Function(adr1 As Adresse, adr2 As Adresse) _
                String.Compare(adr1.PLZ, adr2.PLZ))

        Case AdressenSortierenNach.Ort
            myAdressen.Sort(Function(adr1 As Adresse, adr2 As Adresse) _
                String.Compare(adr1.Ort, adr2.Ort))

    End Select

    'Die Elemente neu sortiert darstellen
    ElementeDarstellen()
End Sub
```

Sort arbeitet hier in diesem Beispiel mit Lambda-Ausdrücken, bei denen jeder der vier Lambda-Ausdrücke nach einer anderen Eigenschaft der Adresse sortiert – in Abhängigkeit davon, welche der vier Spalten in der ListView zum Sortieren eingestellt wurde.

Find und die generische Predicate-Klasse

Das letzte generische »Dreamteam« schließlich kommt zum Einsatz, wenn es um das Finden eines bestimmten Objektes in einer generischen Liste geht: Find und der Delegat, der durch die generische Predicate-Klasse eingerichtet wird. Auch hier entspricht die grundsätzliche Vorgehensweise dem bereits Bekannten, wie der entsprechende Code im Beispiel zeigt:

```
Private Sub btnSuchen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnSuchen.Click

    'Suchbegriff abfragen
    Dim suchFormular As New frmSuchen
    Dim aktuellerSuchbegriff As String      ' Der zuletzt erfragte Suchbegriff

    'Den Suchbegriff merken, damit der Predicate-Delegat darauf
    'zugreifen kann.
    aktuellerSuchbegriff = suchFormular.Suchbegriff
    If String.IsNullOrEmpty(aktuellerSuchbegriff) Then
        Return
    End If

    'Hier kann die Suche beginnen!
    Dim locGefAdr As Adresse = Nothing

    Select Case mySortierenNach
        Case AdressenSortierenNach.Name
            locGefAdr = myAdressen.Find(Function(adr As Adresse) adr.Name = aktuellerSuchbegriff)
        Case AdressenSortierenNach.Vorname
            locGefAdr = myAdressen.Find(Function(adr As Adresse) adr.Vorname = aktuellerSuchbegriff)
        Case AdressenSortierenNach.PLZ
            locGefAdr = myAdressen.Find(Function(adr As Adresse) adr.PLZ = aktuellerSuchbegriff)
        Case AdressenSortierenNach.Ort
            locGefAdr = myAdressen.Find(Function(adr As Adresse) adr.Ort = aktuellerSuchbegriff)
    End Select

    'Wenn ein Element gefunden wurde, dieses markieren.
    If locGefAdr IsNot Nothing Then

        'Alle ListView-Elemente durchsuchen und überprüfen, ob ...
        For Each locLvwItem As ListViewItem In Me.lvwAdressen.Items

            '... die Tag-Referenz der Referenz des gesuchten Objekts entspricht.
            If locLvwItem.Tag Is locGefAdr Then

                'Gefunden! ListView-Element markieren,
                locLvwItem.Selected = True

                'und dafür sorgen, dass es im sichtbaren Bereich liegt.
                locLvwItem.EnsureVisible()
                Return
            End If
        Next
    End If
End Sub
```



Abbildung 23.8 Der Suchbegriff bezieht sich immer auf die Spalte, nach der zuletzt sortiert wurde

Auch hier machen Lambda-Ausdrücke, wie in den fett hervorgehobenen Codezeilen zu sehen, die Suche nach den richtigen Adressen vergleichsweise einfach. Je nach selektierter Sortierspalte wird einfach ein anderer Lambda-Ausdruck verwendet, der den Vergleich mit dem eingegebenen Suchbegriff übernimmt. Nur eine kleine Herausforderung ist dann noch das Selektieren des gefundenen Begriffs in der Liste – und das ist im Übrigen auch der Grund, wieso wir beim Aufbauen der Liste eine Referenz jedes Elements in der Tag-Eigenschaft jedes ListViewItem-Elements speichern: Wenn der Begriff durch Find gefunden wurde, liegt uns das entsprechende Adresse-Objekt vor. Mit diesem können wir anschließend durch die ListViewItem-Auflistung iterieren und auf Objektübereinstimmung durch Abfrage der Tag-Eigenschaft testen. Dieser Aufwand ist nötig, da es keine andere Möglichkeit gibt, das richtige ListViewItem-Element zu finden, und nur dieses erlaubt es aber, die richtige Zeile in der Liste durch seine Select-Eigenschaft zu selektieren.

Kapitel 24

Serialisierung von Typen

In diesem Kapitel:

Einführung in Serialisierungstechniken	698
Flaches und tiefes Klonen von Objekten	706
Serialisieren von Objekten mit Zirkelverweisen	712
XML-Serialisierung	715

Wenn Sie Anwendungen entwickeln, kommen Sie irgendwann zwangsläufig an den Punkt, an dem Sie die Objekte, mit deren Hilfe Sie die Daten Ihrer Anwendung verwalten, für den späteren Gebrauch sichern oder zur Weiterverarbeitung an eine andere Instanz übertragen müssen. Vom aktuellen »Zustand« eines Objektes muss in diesem Fall eine Art Momentaufnahme gemacht werden; der Speicherinhalt aller Eigenschaften und aller untergeordneten Objekte muss dabei gesichert werden. Dabei spielt es natürlich erst einmal keine Rolle, auf welche Weise die Daten gesichert werden: Sie können byteweise – so, wie sie im Arbeitsspeicher stehen – direkt dort ausgelesen und in eine Datei geschrieben werden; denkbar wäre auch, dass sie zuvor durch entsprechende Algorithmen komprimiert und erst dann in einer Datei gespeichert werden. Die Daten eines Objektes, wie Zahlen oder Datumswerte, könnten auch zuvor in ein vom Anwender lesbares Format umgewandelt und speziell formatiert werden, sodass ein Transfervorgang sie auch beispielsweise im XML- oder SOAP-Format direkt über das Internet zu einem anderen Server transportieren könnte.

Ganz gleich wie die Daten aus einem Objekt »geholt« und anschließend an eine andere Stelle verfrachtet werden – die Prozedur, die diese Aufgabe erfüllt, kann nur nach einem bestimmten Schema vorgehen: Sie muss *der Reihe nach* alle Eigenschaften und Variablen des Objektes abfragen, sie aufbereiten und anschließend mit den aufbereiteten Daten irgendetwas anstellen. Bei diesem Vorgang spricht man vom Serialisieren von Objekten.

Auch der umgekehrte Weg ist notwendig: Wenn Sie beispielsweise eine Reihe von Adressobjekten in eine Textdatei serialisiert haben, damit die Daten nach Beenden des Adressprogramms und Ausschalten des Computers erhalten bleiben, muss der umgekehrte Prozess stattfinden, sobald der Anwender den Computer einschaltet, die Adressverwaltung startet und mit den zuvor erfassten Adressen weiterarbeiten möchte. In diesem Fall müssen die Objekte wieder den gleichen Zustand annehmen, den sie vor dem Serialisieren hatten. Sie müssen jetzt aus der Textdatei *deserialisiert* werden.

Nun wäre das Framework nicht das Framework, wenn es Sie bei diesen Vorgängen, die natürlich in jeder Anwendung vorkommen, nicht unterstützen würde. Sie brauchen also nicht selbst Hand anzulegen und jede einzelne Eigenschaft eines Objektes auszulesen und in eine Textdatei zu schreiben. Umgekehrt müssen Sie Adressobjekte auch nicht selbst in Ihrer Anwendung komplett neu instanziieren, während Sie sie aus einer Textdatei wieder einlesen. Das Framework hält für diesen Zweck einige geniale Werkzeuge bereit, die Sie in den folgenden Abschnitten kennenlernen sollten, da sie Ihnen die Arbeit erheblich erleichtern können – nicht nur, wenn Sie die Adressen Ihrer Adressverwaltung auf der Festplatte Ihres Computers speichern wollen.

HINWEIS Wenn Sie Instanzen von Klassen serialisieren und an anderer Stelle wieder deserialisieren, benötigen Sie in jedem Fall die Klassendefinition in beiden Entitäten. Sie können mit den hier vorgestellten Techniken also nicht auch den kompletten Klassencode selbst übertragen, sondern nur die Zustände innerhalb eines Objektes.

Einführung in Serialisierungstechniken

Bevor Sie sich anschauen, was das Framework an Serialisierungstechniken zu bieten hat, lassen Sie uns zunächst einen Blick auf ein ganz simples Beispiel werfen. Dieses Beispielprogramm macht nichts weiter, als den Anwender eine Adresse eingeben zu lassen und diese durch Klick auf eine Schaltfläche in eine Datei zu serialisieren (sprich: zu sichern).

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 24\\Serialization01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).



Abbildung 24.1 Die Nur-Lesen-Eigenschaften werden beim Serialisieren nicht übernommen

Das Beispiel nutzt dabei zunächst noch keine Serialisierungstechniken des Frameworks, sondern liest die einzelnen Eigenschaften sozusagen manuell aus und speichert sie als Text in einer Datei. Beim Klicken auf die Schaltfläche *Serialisieren* generiert die Prozedur, die das Click-Ereignis behandelt, ein Adresse-Objekt aus den Feldinhalten des Dialoges. Dieses Objekt übergibt die Routine einer weiteren Prozedur, die eine Datei zum Schreiben öffnet, die Eigenschaften nacheinander ausliest und sie als String in die Datei schreibt:

```
Private Sub btnSerialisieren_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnSerialisieren.Click
    Dim locAdresse As New Adresse(txtVorname.Text, _
        txtNachname.Text, _
        txtStraße.Text, _
        txtPLZOrt.Text)
    'Einen "unmöglichen" Dateinamen verwenden, damit, wie es der Zufall will,
    'nicht eine andere wichtige Datei gleichen Namens überschrieben wird.
    Adresse.SerializeToFile(locAdresse, "C:\\serializedemo_f4e3w21.txt")

    'Info über den Datensatz anzeigen.
    txtErstelltAm.Text = locAdresse.ErfasstAm.ToString("dd.MM.yyyy HH:mm:ss")
    txtErstelltVon.Text = locAdresse.ErfasstVon
End Sub
```

Die Adresse-Klasse hält die notwendigen Elemente zum Speichern der Adressdaten und die statische Prozedur zum Serialisieren der Objektdaten in eine Datei bereit:

```
Public Class Adresse
    Private myName As String
    Private myVorname As String
    Private myStraße As String
    Private myPLZOrt As String
    Private myErfasstAm As DateTime
    Private myErfasstVon As String
```

```

Sub New(ByVal Vorname As String, ByVal Name As String, ByVal Straße As String, ByVal PLZOrt As String)
    'Konstruktor legt alle Member-Daten an.
    myName = Name
    myVorname = Vorname
    myStraße = Straße
    myPLZOrt = PLZOrt
    myErfasstAm = DateTime.Now
    myErfasstVon = Environment.UserName
End Sub

'Alle öffentlichen Felder in die Datei schreiben.
Public Shared Sub SerializeToFile(ByVal adr As Adresse, ByVal Filename As String)
    Dim locStreamWriter As New StreamWriter(Filename, False, System.Text.Encoding.Default)
    With locStreamWriter
        .WriteLine(adr.Vorname)
        .WriteLine(adr.Name)
        .WriteLine(adr.Straße)
        .WriteLine(adr.PLZOrt)
        .Flush()
        .Close()
    End With
End Sub

'Aus der Datei lesen und daraus ein neues Adressobjekt erstellen.
Public Shared Function DeserializeFromFile(ByVal Filename As String) As Adresse
    'Interessiert an dieser Stelle nicht, deswegen ausgelassen.
End Function

Public Property Name() As String
    Get
        Return myName
    End Get
    Set(ByVal Value As String)
        myName = Value
    End Set
End Property

'Die beiden folgenden Eigenschaften haben "Nur-Lesen-Status", da auch
'der Entwickler das Anlegen-Datum nicht manipulieren darf!
Public ReadOnly Property ErfasstAm() As DateTime
    Get
        Return myErfasstAm
    End Get
End Property

Public ReadOnly Property ErfasstVon() As String
    Get
        Return myErfasstVon
    End Get
End Property

#Region "Die anderen Eigenschaften"
    'Aus Platzgründen ausgelassen.
#End Region
End Class

```

Sie sehen anhand dieses Codelistings, dass es zwei Eigenschaften gibt, die zwar gelesen, aber nicht geschrieben werden dürfen: Diese Eigenschaften, die Auskunft darüber geben, wer das `Adresse`-Objekt zu welchem Zeitpunkt angelegt hat, dürfen ausschließlich bei der Objekterstellung definiert werden.

Bei Serialisieren auf die herkömmliche Art und Weise, wie hier im Beispiel zu sehen, ergibt sich hier schon ein Problem: Wenn Sie das Programm nämlich beenden, anschließend erneut starten und die Deserialisierenschaltfläche betätigen, dann wird zwar aus der Datei ein neues `Adresse`-Objekt erzeugt. Dieses Objekt entspricht dem ursprünglichen aber nicht in allen Details, denn: Das Erstellungsdatum und der »Ersteller« des Objektes selber hätten zwar noch mitgesichert werden können; der Deserialisierungs-Algorithmus hat aber aufgrund der Beschaffenheit dieser zusätzlichen Eigenschaften keine Möglichkeit, die Originalzustände dieser Eigenschaft wieder einzulesen. Er könnte diese Zusatzinformationen dem Objekt schlichtweg nicht zuordnen.

Diese Tatsache spiegelt sich im Programm wider: Wann immer Sie das ursprünglich gespeicherte Objekt durch Klick auf die entsprechende Schaltfläche deserialisieren, steht in den unteren Infofeldern nicht das ursprüngliche Erstellungsdatum, sondern das Erstellungsdatum des Objektes zum Zeitpunkt des Deserialisierens.

Serialisieren mit SoapFormatter und BinaryFormatter

Anders sieht es aus, wenn Sie das Serialisieren und das Deserialisieren mit bestimmten Hilfsmitteln aus dem Framework durchführen. Sie brauchen sich in diesem Fall nämlich nicht um das Auslesen der Eigenschaften der Objekte selbst zu kümmern – das Framework macht das automatisch für Sie. Und noch mehr: Einige Klassen des Frameworks, die für das Serialisieren von Objektdaten zuständig sind, erlauben auch das Serialisieren und Deserialisieren von privaten Eigenschaften einer Klasse.

WICHTIG Für alle Objekte, die serialisiert werden sollen, gilt: Sie müssen mit einem besonderen Attribut namens `Serializable` gekennzeichnet werden. Wenn diese Voraussetzungen erfüllt sind, können Sie zwei der Serializer-Klassen für die Serialisierung und Deserialisierung einer so gekennzeichneten Objekt-Instanz verwenden, die auch private Member-Variablen einer Klasse berücksichtigen.

- **SoapFormatter-Klasse:** Stellt Serialisierungs- und Deserialisierungsfunktionen zur Verfügung, die das `SOAP`-Format verwenden. Die Dateninhalte eines Objektes werden dabei in reinen Text umgewandelt, der auf der einen Seite auch für den Anwender verständlich mit einem Texteditor gelesen und angezeigt und auf der anderen Seite – da er das `SOAP`-Format einhält – auch problemlos über das Internet transportiert werden kann. Der Nachteil: Diese Art der Datenspeicherung ist nicht sonderlich effizient, eben durch die Konvertierung nativer Daten in lesbaren Text.
- **BinaryFormatter-Klasse:** Stellt Serialisierungs- und Deserialisierungsfunktionen zur Verfügung, die das Binärformat verwenden. Die Dateninhalte eines Objektes werden dabei so verwendet, wie sie im Arbeitsspeicher vorliegen. Werden die Daten eines Objektes unter Verwendung dieses `Serializers` beispielsweise in einer Datei gespeichert, erfolgt die Datenspeicherung sehr kompakt. Eine auf diese Weise generierte Datei ist aber für den Anwender direkt nicht lesbar, da sie die Daten des Objektes im binären Format enthält.

BEGLEITDATEIEN

Das folgende Beispiel demonstriert den Einsatz mit diesen beiden Klassen. Sie finden dieses Beispiel unter:

... \VB 2008 Entwicklerbuch\ D - Datenstrukturen\ Kapitel 24\ Serialization02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).



Abbildung 24.2 In dieser Version können Sie zwischen Soap- und Binärserialisierung wählen. Die Nur-Lesen-Eigenschaften bleiben jetzt beim Deserialisieren erhalten

Wenn Sie dieses Programm starten, wählen Sie für das Serialisieren bzw. Deserialisieren das gewünschte Format aus. Wenn Sie sich für das *SOAP*-Format entscheiden, generiert die *SoapFormatter*-Klasse eine Datei, die der folgenden entspricht (vorausgesetzt natürlich, Sie haben die gleichen Daten eingegeben, wie in Abbildung 24.2 zu sehen):

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
  ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"> 
<SOAP-ENV:Body>
<a1:Adresse id="ref-1">
  xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Serialization02/Serialization02%20Version%3D1.0.0.
  0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
    <myName id="ref-3">Thiemann</myName>
    <myVorname id="ref-4">Uwe</myVorname>
    <myStraße id="ref-5">Autorenstraße 34</myStraße>
    <myPLZOrt id="ref-6">99999 Buchhausen</myPLZOrt>
    <myErfasstAm>2006-02-15T11:29:24.1295355+01:00</myErfasstAm>
    <myErfasstVon id="ref-7">ACTIVEDEVELOP\loeffel</myErfasstVon>
  </a1:Adresse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Sie werden überrascht sein, wenn Sie sehen, mit wie wenig Aufwand die Serialisierung im Programm realisiert wurde. Wenn Sie einen Blick in den Projektmappen-Explorer werfen, finden Sie dort im Gegensatz zum vorherigen Beispiel zwei weitere Klassendateien. Sie enthalten die Kapselungen von *SoapFormatter* und *BinaryFormatter* zur Serialisierung (und Deserialisierung) beliebiger, serialisierbarer Objekte in Dateien, die folgendermaßen aussehen:

Universeller Soap-Datei-De-/Serializer

```
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Soap
Imports System.IO

Public Class ADSoapSerializer

    Shared Sub SerializeToFile(ByVal FileInfo As FileInfo, ByVal [Object] As Object)

        Dim locFs As FileStream = New FileStream(FileInfo.FullName, FileMode.Create)
        Dim locSoapFormatter As New SoapFormatter(Nothing,
            New StreamingContext(StreamingContextStates.File))
        locSoapFormatter.Serialize(locFs, [Object])
        locFs.Flush()
        locFs.Close()

    End Sub

    Shared Function DeserializeFromFile(ByVal FileInfo As FileInfo) As Object

        Dim locObject As Object

        Dim locFs As FileStream = New FileStream(FileInfo.FullName, FileMode.Open)
        Dim locSoapFormatter As New SoapFormatter(Nothing,
            New StreamingContext(StreamingContextStates.File))
        locObject = locSoapFormatter.Deserialize(locFs)
        locFs.Close()
        Return locObject

    End Function

End Class
```

Wenn Sie SoapFormatter verwenden wollen, müssen Sie einen Verweis in das entsprechende Projekt einbauen. Dazu öffnen Sie im Projektmappen-Explorer über dem Projektnamen oder dem Ordner *Verweise* (wenn das Symbol *alle Dateien anzeigen aktiviert* ist) mit der rechten Maustaste das Kontextmenü und wählen dort den Eintrag *Verweise*. Im Dialog, der sich jetzt öffnet, wählen Sie den Eintrag *System.Runtime.Serialization.Formatters.Soap* per Doppelklick aus (etwa wie in Abbildung 24.3 zu sehen).

Verlassen Sie den Dialog mit *OK*. Achten Sie ebenfalls darauf, die Anweisungen

```
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Soap
```

als erste Zeilen in der Codedatei zu platzieren, in der Sie SoapFormatter verwenden möchten.

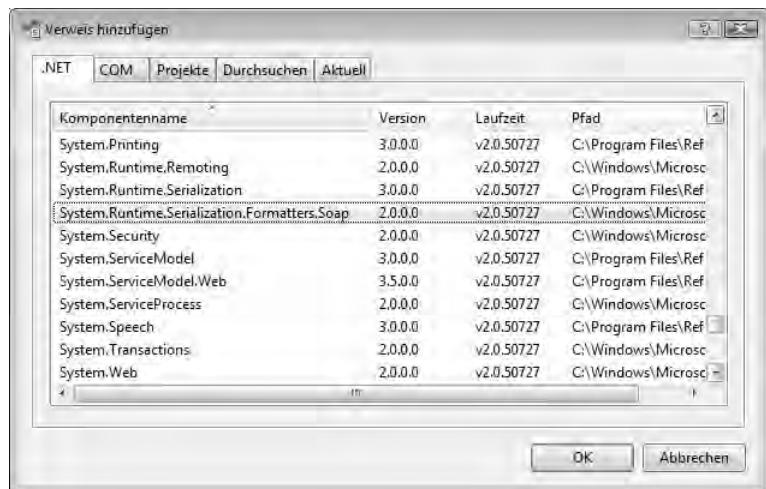


Abbildung 24.3 Wenn Sie mit der SoapFormatter-Klasse Objektdaten im SOAP-Format serialisieren wollen, binden Sie diesen Verweis in Ihr Projekt ein

Universeller Binary-Datei-De-/Serializer

```

Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary
Imports System.IO

Public Class ADBinarySerializer

    Shared Sub SerializeToFile(ByVal FileInfo As FileInfo, ByVal [Object] As Object)

        Dim locFs As FileStream = New FileStream(FileInfo.FullName, FileMode.Create)
        Dim locBinaryFormatter As New BinaryFormatter(Nothing,
            New StreamingContext(StreamingContextStates.File))
        locBinaryFormatter.Serialize(locFs, [Object])
        locFs.Flush()
        locFs.Close()

    End Sub

    Shared Function DeserializeFromFile(ByVal FileInfo As FileInfo) As Object

        Dim locObject As Object

        Dim locFs As FileStream = New FileStream(FileInfo.FullName, FileMode.Open)
        Dim locBinaryFormatter As New BinaryFormatter(Nothing,
            New StreamingContext(StreamingContextStates.File))
        locObject = locBinaryFormatter.Deserialize(locFs)
        locFs.Close()
        Return locObject

    End Function

End Class

```

Achten Sie darauf, die Anweisungen

```
Imports System.Runtime.Serialization  
Imports System.Runtime.Serialization.Formatters.Binary
```

als erste Zeilen in der Codedatei zu platzieren, in der Sie BinaryFormatter verwenden möchten. Einen speziellen Verweis brauchen Sie, anders als bei SoapFormatter, nicht in das Projekt einzubinden.

Funktionsweise der Datei-Serializer-Klassen

Beide Klassen funktionieren exakt nach dem gleichen Prinzip, sie verwenden lediglich unterschiedliche Formatter (also Klassen, die Datenaufbereitungslogiken zur Verfügung stellen), um unterschiedliche Formate zu erzeugen bzw. für die Rekreation des Ursprungobjektes.

Sie öffnen beim Serialisieren zunächst einen Dateistrom, über den die Daten über den jeweiligen Formatter aus dem Objekt in die Datei gelangen. In der anschließenden Instanziierung des Formatters wird dieser durch die Übergabe einer StreamingContext-Instanz darauf vorbereitet, welches Ziel die Objektserialisierung haben wird (in diesem Fall werden die Objektdaten in eine Datei serialisiert). Die eigentliche Serialisierung des Objektes geschieht dann durch die einzige Zeile:

```
locSoapFormatter.Serialize(locFs, [Object])
```

Der Rest der Prozedur ist obligatorisch: Alle Stromdaten werden mit Flush aus dem internen Puffer geleert, und die Datei wird anschließend geschlossen. Das Ergebnis: Die Objektdaten wurden einschließlich ihrer privaten Member in die Datei geschrieben. Der umgekehrte Weg beim Deserialisieren erfolgt äquivalent.

TIPP Wenn Sie diese Art der Serialisierung in Dateien in Ihren eigenen Programmen verwenden wollen, kopieren Sie die Codedateien ADBinarySerializer.vb bzw. ADSoapSerializer.vb einfach in Ihr Projektverzeichnis und fügen sie anschließend Ihrem Projekt hinzu. Sie können sie dann auf genauso einfache Weise verwenden, wie sie im Beispielprojekt vom Adressobjekt verwendet wurden (siehe in Fettschrift gesetzte Zeilen am Ende des Codelistings):

```
<Serializable()>  
Public Class Adresse  
  
    Private myName As String  
    Private myVorname As String  
    Private myStraße As String  
    Private myPLZOrt As String  
    Private myErfasstAm As DateTime  
    Private myErfasstVon As String  
  
    Sub New(ByVal Vorname As String, ByVal Name As String, ByVal Straße As String, ByVal PLZOrt As  
String)  
        'Konstruktor legt alle Member-Daten an.  
        myName = Name  
        myVorname = Vorname  
        myStraße = Straße  
        myPLZOrt = PLZOrt  
        myErfasstAm = DateTime.Now  
        myErfasstVon = Environment.UserName  
    End Sub
```

```
'Alle öffentlichen Felder in die Datei schreiben - Soap-Format
Public Shared Sub SerializeSoapToFile(ByVal adr As Adresse, ByVal Filename As String)
    ADSoapSerializer.SerializeToFile(New FileInfo(Filename), adr)
End Sub

'Aus der Datei lesen und daraus ein neues Adressobjekt erstellen - Soap-Format.
Public Shared Function DeserializeSoapFromFile(ByVal Filename As String) As Adresse
    Return CType(ADSoapSerializer.DeserializeFromFile(New FileInfo(Filename)), Adresse)
End Function

'Alle öffentlichen Felder in die Datei schreiben - Binary-Format.
Public Shared Sub SerializeBinToFile(ByVal adr As Adresse, ByVal Filename As String)
    ADBinarySerializer.SerializeToFile(New FileInfo(Filename), adr)
End Sub

'Aus der Datei lesen und daraus ein neues Adressobjekt erstellen - Binary-Format.
Public Shared Function DeserializeBinFromFile(ByVal Filename As String) As Adresse
    Return CType(ADBinarySerializer.DeserializeFromFile(New FileInfo(Filename)), Adresse)
End Function
.
.
```

WICHTIG Wenn Sie die Objektserialisierung anwenden, achten Sie darauf, dass alle Objekte, die Sie serialisieren wollen, das Serializable-Attribut tragen – so wie auch im vorherigen Codelisting zu sehen (siehe erste beiden Zeilen). Wenn die Klasse, die Sie serialisieren möchten, oder eine Objektinstanz, die diese Klasse einbindet, dieses Attribut nicht trägt, löst das Framework beim Serialisierungsversuch durch eines der zu serialisierenden Objekte eine Ausnahme aus, etwa wie in Abbildung 24.4 zu sehen.

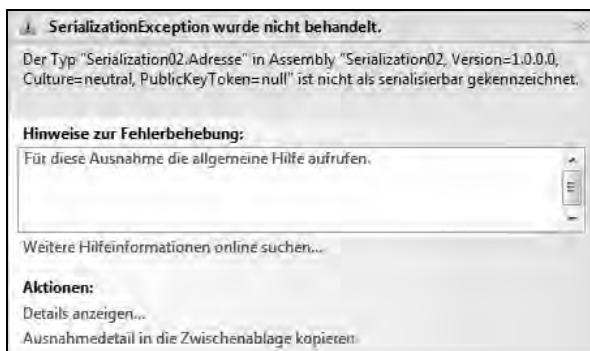


Abbildung 24.4 Wenn eine zu serialisierende Klasse oder ein Objekt, das sie einbindet, nicht mit dem Attribut Serializable gekennzeichnet ist, löst das Framework beim Serialisierungsversuch diese Ausnahme aus

Flaches und tiefes Klonen von Objekten

Der Vorteil beim Serialisieren über Funktionen des Frameworks ist, dass die Serialisierungsalgorithmen in der Lage sind, automatisch so genannte »tiefen Klons« (vollständige Kopien) eines Objektes zu erstellen.

Dazu folgender Hintergrund: Angenommen, Sie haben ein Objekt, das die Daten Ihrer Applikation speichert. Dieses Objekt verfügt dann beispielsweise über eine Eigenschaft, die eine ArrayList zur Verfügung

stellt, in der weitere Elemente gespeichert sind. Um eine komplette Kopie dieses Objektes zu erstellen, würde es nicht ausreichen, die Elemente, die diese ArrayList-Eigenschaft beinhaltet, zu kopieren, wie das folgende Beispiel zeigt:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 24\\DeepCloning

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module mdlMain
Sub Main()
    Dim locAdrOriginal As New Adresse("Hans", "Mustermann", "Musterstraße 22", "59555 Lippstadt")
    Dim locAdrKopie As Adresse
    With locAdrOriginal.BefreundetMit
        .Add(New Adresse("Uwe", "Thiemann", "Autorstr. 33", "49595 Buchhausen"))
        .Add(New Adresse("Gaby", "Halek", "Autorstr. 34", "49595 Buchhausen"))
        .Add(New Adresse("Gabriele", "Löffelmann", "Autorstr. 35", "49595 Buchhausen"))
    End With
    'Originaladresse ausgeben.
    Console.WriteLine("Original:")
    Console.WriteLine("=====")
    Console.WriteLine(locAdrOriginal)

    'Kopie anlegen.
    With locAdrOriginal
        locAdrKopie = New Adresse(.Vorname, .Name, .Straße, .PLZOrt)
        locAdrKopie.Name += " (Kopie)"
        locAdrKopie.BefreundetMit = .BefreundetMit
    End With

    'Kopie ausgeben.
    Console.WriteLine("Kopie:")
    Console.WriteLine("=====")
    Console.WriteLine(locAdrKopie)

    'Änderungen im Original:
    CType(locAdrOriginal.BefreundetMit(1), Adresse).Name = "Löffelmann-Halek"
    CType(locAdrOriginal.BefreundetMit(2), Adresse).Name = "Löffelmann-Halek"

    'Kopie nach Änderungen im Original:
    Console.WriteLine("Kopie nach Änderung im Original:")
    Console.WriteLine("=====")
    Console.WriteLine(locAdrKopie)
    Console.ReadLine()
End Sub
End Module
```

Dieses Beispiel nutzt die leicht abgewandelte `Adresse`-Klasse, die Sie schon aus dem vorherigen Beispiel kennen. Sie verfügt über eine zusätzliche Eigenschaft namens `BefreundetMit`. Diese Eigenschaft entspricht einer `ArrayList` mit der Aufgabe, andere Adressen aufzunehmen, die bestimmen, mit wem dieser Kontakt befreundet ist.

```
<Serializable()>
Public Class Adresse

    Private myName As String
    Private myVorname As String
    Private myStraße As String
    Private myPLZOrt As String
    Private myErfasstAm As DateTime
    Private myErfasstVon As String
    Private myBefreundetMit As ArrayList

    Sub New(ByVal Vorname As String, ByVal Name As String, ByVal Straße As String, ByVal PLZOrt As String)
        'Konstruktor legt alle Member-Daten an.
        myName = Name
        myVorname = Vorname
        myStraße = Straße
        myPLZOrt = PLZOrt
        myErfasstAm = DateTime.Now
        myErfasstVon = Environment.UserName
        myBefreundetMit = New ArrayList
    End Sub

    Public Property BefreundetMit() As ArrayList
        Get
            Return myBefreundetMit
        End Get
        Set(ByVal Value As ArrayList)
            myBefreundetMit = Value
        End Set
    End Property

    #Region "Die anderen Eigenschaften"
    'Aus Platzgründen ausgelassen.
    #End Region

    Public Overrides Function ToString() As String
        Dim locTemp As String
        locTemp = Name + ", " + Vorname + ", " + Straße + ", " + PLZOrt + vbNewLine
        locTemp += "--- Befreundet mit: ---" + vbNewLine
        For Each locAdr As Adresse In BefreundetMit
            locTemp += " * " + locAdr.ToStringShort() + vbNewLine
        Next
        locTemp += vbNewLine
        Return locTemp
    End Function

    Public Function ToStringShort() As String
        Return Name + ", " + Vorname
    End Function
End Class
```

Zusätzlich gibt es in dieser Klasse zwei Funktionen – `ToString` und `ToStringShort` – die eine Adresse-Instanz in einen String umwandeln, damit die Ausgabe einfacher wird.

Das Programm macht nun Folgendes: Es legt eine Originaladresse an und fügt ihr weitere Adressobjekte hinzu, die in der `ArrayList`, die durch `BefreundetMit` offengelegt wird, gespeichert werden. Anschließend erzeugt es eine identische Kopie der Originaladresse im Objekt `locAdrKopie`.

Das Problem: An dieser Stelle wird eine so genannte flache Kopie des Objektes erstellt. Nur die Eigenschaften der oberen Ebene werden in die Kopie übernommen. Das wird auch deutlich, wenn Sie das Beispielprogramm starten:

```
Original:  
=====  
Mustermann, Hans, Musterstraße 22, 59555 Lippstadt  
--- Befreundet mit: ---  
* Thiemann, Uwe  
* Halek, Gaby  
* Löffelmann, Gabriele  
  
Kopie:  
=====  
Mustermann (Kopie), Hans, Musterstraße 22, 59555 Lippstadt  
--- Befreundet mit: ---  
* Thiemann, Uwe  
* Halek, Gaby  
* Löffelmann, Gabriele  
  
Kopie nach Änderung im Original:  
=====  
Mustermann (Kopie), Hans, Musterstraße 22, 59555 Lippstadt  
--- Befreundet mit: ---  
* Thiemann, Uwe  
* Löffelmann-Halek, Gaby  
* Löffelmann-Halek, Gabriele
```

Beim Ändern der Elemente der `BefreundetMit-ArrayList` werden auch die Elemente der Kopie verändert. Und das muss so sein, denn: Die Eigenschaft `BefreundetMit` stellt ja nicht wirklich selbst eine `ArrayList` dar, sondern verweist lediglich auf sie. Tatsächlich gibt es die Elemente `ArrayList` nur ein einziges Mal. In diesem Beispiel ist also nur eine flache Kopie (»shallow clone« bzw. »shallow copy« auf englisch) des Adresse-Objektes erstellt worden.

Anders wird das, wenn Sie eine tiefe Kopie (»deep clone« bzw. »deep copy« auf englisch) erstellen. Hier werden die Elemente der `ArrayList`, um beim Beispiel zu bleiben, wirklich kopiert – es gibt nach Abschluss der Kopieerstellung wirklich zwei völlig unabhängige Objekte, mit ebenso unabhängigen Elementen.

Was bei diesem Beispiel mit noch vergleichsweise wenig Aufwand durchführbar wäre, sieht bei wirklich komplexen Objekten schon anders aus. Wollten Sie eine `DeepCopy`-Routine selber implementieren, bedeutete dies einen enormen Aufwand. Obendrein könnten Sie eine solche Routine mit normalen Mitteln nicht universal gestalten; sie würde sich nur auf das aktuelle Objekt beziehen. Bei einer Klassenableitung, die die Basisklasse um weitere Eigenschaften ergänzen würde, müssten Sie die `DeepCopy`-Routine schon wieder modifizieren.

Doch diesen Aufwand müssen Sie auch gar nicht betreiben, denn mit den Serialierungsfunktionen nimmt Ihnen das Framework diesen kompletten Aufwand ab. Beim Serialisieren erstellt das Framework nämlich eine tiefe Kopie.¹ Und da Serialisierung natürlich nicht notwendigerweise bedeutet, das Objekt in einer Datei zwischenspeichern, können Sie mit einem kleinen Trick eine universelle DeepCopy-Routine kreieren, die mit jedem serialisierbaren Objekt funktioniert, wie das folgende Beispiel zeigt.

Universelle DeepClone-Methode

Das folgende Beispiel verwendet ebenfalls den `BinarySerializer` für das Serialisieren und das Deserialisieren eines Objektes, doch nutzt er einen anderen Träger im Vergleich zum letzten Beispiel. Ein Objekt, das es zu serialisieren gilt, wird nicht in eine Datei, sondern in ein `MemoryStream`-Objekt geschrieben, das schließlich in ein Byte-Array konvertiert wird. Im umgekehrten Fall erzeugt der `BinarySerializer` aus einem Byte-Array, das in ein `MemoryStream`-Objekt umgewandelt wird, wieder das ursprüngliche Objekt. Da der `BinarySerializer` grundsätzlich auch verschachtelte Eigenschaften verarbeitet, lässt sich daraus auf einfache Art und Weise eine Klasse erstellen, die von jedem beliebigen Objekt, das selbst als serialisierbar gekennzeichnet ist und auch nur selbst serialisierbare Objekte verwendet, eine tiefe Kopie anfertigen kann.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 24\\UniversalDeepCloning

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary
Imports System.IO

Public Class ADObjectCloner

    Public Shared Function DeepCopy(ByVal [Object] As Object) As Object
        Return DeserializeFromByteArray(SerializeToByteArray([Object]))
    End Function

    Shared Function SerializeToByteArray(ByVal [Object] As Object) As Byte()
        Dim retByte() As Byte
        Dim locMs As MemoryStream = New MemoryStream
        Dim locBinaryFormatter As New BinaryFormatter(Nothing,
            New StreamingContext(StreamingContextStates.Clone))
        locBinaryFormatter.Serialize(locMs, [Object])
        locMs.Flush()
        locMs.Close()
        retByte = locMs.ToArray()
        locMs.Dispose()
        Return retByte
    End Function
End Class
```

¹ Wobei diese Aussage nicht hundertprozentig richtig ist, denn eigentlich erstellt das Framework ja keine Kopie des Objektes, sondern liest beim Serialisieren zunächst nur seine Daten aus. Das macht es aber bis in die unterste Ebene, sodass man den kombinierten Vorgang von Serialisieren eines Objektes in einen Datenstrom und Deserialisieren dieses Datenstroms in ein neues Objekt durchaus »tiefes Kopieren« des Objektes nennen kann.

```
retByte = locMs.ToArray()
Return retByte

End Function

Shared Function DeserializeFromByteArray(ByVal by As Byte()) As Object
    Dim locObject As Object

    Dim locFs As MemoryStream = New MemoryStream(by)
    Dim locBinaryFormatter As New BinaryFormatter(Nothing,
        New StreamingContext(StreamingContextStates.File))
    locObject = locBinaryFormatter.Deserialize(locFs)
    locFs.Close()
    Return locObject

End Function

End Class
```

Das Hauptprogramm verwendet im folgenden Beispiel nun nicht mehr eigenen Code zum Kopieren des Objektes, sondern benutzt die DeepCopy-Funktion der Klasse (die geänderte Passage ist fett hervorgehoben):

```
Module mdlMain

    Sub Main()
        Dim locAdrOriginal As New Adresse("Hans", "Mustermann", "Musterstraße 22", "59555 Lippstadt")
        Dim locAdrKopie As Adresse
        With locAdrOriginal.BefreundetMit
            .Add(New Adresse("Uwe", "Thiemann", "Autorstr. 33", "49595 Buchhausen"))
            .Add(New Adresse("Gaby", "Halek", "Autorstr. 34", "49595 Buchhausen"))
            .Add(New Adresse("Gabriele", "Löffelmann", "Autorstr. 35", "49595 Buchhausen"))
        End With

        'Originaladresse ausgeben.
        Console.WriteLine("Original:")
        Console.WriteLine("=====")
        Console.WriteLine(locAdrOriginal)

        'Kopie anlegen.
        locAdrKopie = CType(ADObjectCloner.DeepCopy(locAdrOriginal), Adresse)

        'Kopie ausgeben.
        Console.WriteLine("Kopie:")
        Console.WriteLine("=====")
        Console.WriteLine(locAdrKopie)

        'Änderungen im Original:
        CType(locAdrOriginal.BefreundetMit(1), Adresse).Name = "Löffelmann-Halek"
        CType(locAdrOriginal.BefreundetMit(2), Adresse).Name = "Löffelmann-Halek"

        'Kopie nach Änderungen im Original:
        Console.WriteLine("Kopie nach Änderung im Original:")
        Console.WriteLine("=====")
```

```
Console.WriteLine(locAdrKopie)
Console.ReadLine()

End Sub

End Module
```

Wenn Sie das Programm anschließend starten, erkennen Sie den Unterschied zum vorherigen Beispiel. Die beiden erzeugten Objekte sind wirklich komplett unabhängig voneinander. Änderungen an der ArrayList des Ausgangsobjektes beeinflussen das Ergebnis der Ausgabe der Objektkopie in keiner Weise:

```
Original:
=====
Mustermann, Hans, Musterstraße 22, 59555 Lippstadt
--- Befreundet mit: ---
* Thiemann, Uwe
* Halek, Gaby
* Löffelmann, Gabriele

Kopie:
=====
Mustermann, Hans, Musterstraße 22, 59555 Lippstadt
--- Befreundet mit: ---
* Thiemann, Uwe
* Halek, Gaby
* Löffelmann, Gabriele

Kopie nach Änderung im Original:
=====
Mustermann, Hans, Musterstraße 22, 59555 Lippstadt
--- Befreundet mit: ---
* Thiemann, Uwe
* Halek, Gaby
* Löffelmann, Gabriele
```

Serialisieren von Objekten mit Zirkelverweisen

So genannte Zirkelverweise bereiten Speicheralgorithmen erfahrungsgemäß die größten Schwierigkeiten. Zirkelverweise kennen Sie vielleicht aus der Tabellenkalkulation. Sie treten dann auf, wenn beispielsweise Zelle A auf Zelle B, diese auf Zelle C, und diese wieder auf Zelle A verweisen soll. Was bei Tabellenkalkulationen grundsätzlich verboten ist, gestattet das Framework mit Objektreferenzen dagegen sehr wohl. Und – um bei unserem bisherigen Beispiel zu bleiben – im Szenario des Programms aus dem letzten Abschnitt könnte das sogar recht schnell passieren, denn: Es ist nicht nur denkbar, sondern wahrscheinlich, dass Person A Person B zu seinem Freundeskreis zählt und umgekehrt.

Welche Auswirkungen Zirkelverweise normalerweise haben, zeigt das folgende Beispiel.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 24\\Zirkelverweise

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

In diesem Beispielprogramm sind im Gegensatz zum vorherigen die folgenden Änderungen vorgenommen worden, was die Adresse-Klasse betrifft. Die `ToString`-Funktion benutzt selbst nicht mehr die Kurzform der Adressen für die Ausgabe der `BefreundetMit`-Eigenschaft, sondern erzeugt den Ausgabe-String ebenfalls mit der `ToString`-Funktion, die allerdings eine kleine Änderung erfahren hat. Welche das ist, sehen Sie, wenn Sie das Programm starten und sich anschließend das Ergebnis anschauen:

```
Erste Adresse:  
=====  
Halek, Gaby, Musterstraße 24, 32132 Buchhausen  
--- Befreundet mit: ---  
    Raubein, Petra, Autorenstr. 12, 32132 Buchhausen  
  
Thiemann, Uwe, Autorenstr. 22, 32132 Buchhausen  
--- Befreundet mit: ---  
    Koch, Manuela, Autorenstr. 22, 32132 Buchhausen
```

```
Zweite Adresse:  
=====  
Thiemann, Uwe, Autorenstr. 22, 32132 Buchhausen  
--- Befreundet mit: ---  
    Koch, Manuela, Autorenstr. 22, 32132 Buchhausen
```

Das Programm, das diese Ausgabe hervorgebracht hat, sieht folgendermaßen aus:

```
Module mdlMain  
Sub Main()  
    Dim locErsteAdr As New Adresse("Gaby", "Halek", "Musterstraße 24", "32132 Buchhausen")  
    Dim locZweiteAdr As New Adresse("Uwe", "Thiemann", "Autorenstr. 22", "32132 Buchhausen")  
    locErsteAdr.BefreundetMit.Add(New Adresse("Petra", "Raubein", "Autorenstr. 12", "32132  
Buchhausen"))  
    locErsteAdr.BefreundetMit.Add(locZweiteAdr)  
    locZweiteAdr.BefreundetMit.Add(New Adresse("Manuela", "Koch", "Autorenstr. 22", "32132  
Buchhausen"))  
    'Wenn Sie diese Zeile einbauen, erstellen Sie einen Zirkelverweis.  
    'locZweiteAdr.BefreundetMit.Add(locErsteAdr)  
  
    'Originaladresse ausgeben.  
    Console.WriteLine("Erste Adresse:")  
    Console.WriteLine("=====")  
    Console.WriteLine(locErsteAdr)  
  
    Console.WriteLine("Zweite Adresse:")  
    Console.WriteLine("=====")
```

```

Console.WriteLine(locZweiteAdr)

'Kopie anlegen.
Dim locAdrKopie As Adresse = CType(ADObjectCloner.DeepCopy(locErsteAdr), Adresse)
Console.ReadLine()

End Sub
End Module

```

Sie erkennen am Ergebnis und dem Programmtext, dass das Programm auch verschachtelte Daten in der Eigenschaft `BefreundetMit` (und der dahinter steckenden `ArrayList`) bei der Ausgabe berücksichtigt. Wenn es die Liste der befreundeten Personen erstellt, und eine Person dieser Liste wieder Personeneinträge unter `BefreundetMit` führt, werden diese bei der Ausgabe ebenfalls berücksichtigt.

Eine solche Vorgehensweise kann für ein Programm allerdings wirkliche fatale Folgen haben. Denn wenn eine der Personen der `BefreundetMit`-Liste in ihrer Liste die Person führt, die ihr ebenfalls zugeordnet ist (Person A ist befreundet mit Person B, und Person B ist ebenfalls befreundet mit Person A – was ja durchaus nichts Ungewöhnliches ist), dann haben Sie es mit einem Zirkelverweis zu tun.

Der `ToString`-Algorithmus versagt in diesem Fall, weil er sich durch den Zirkelverweis immer wieder selbst aufruft. Sie können dieses Verhalten testen, indem Sie die Auskommentierung der im Listing fett gesetzten Zeile aufheben und den Auflösungsversuch eines Zirkelverweises initiieren. Wenn Sie das Programm anschließend starten, erscheint nach einer Weile im Ausgabefenster die Meldung

Eine Ausnahme (erste Chance) des Typs "System.OutOfMemoryException" ist in mscorlib.dll aufgetreten.

Das Programm »hängt« eine Weile, und je nachdem, wie gut oder schlecht Ihr System mit der nur noch ungenügend zur Verfügung stehenden Menge an Speicher klarkommt, sehen Sie kurz darauf folgende Ausnahme:

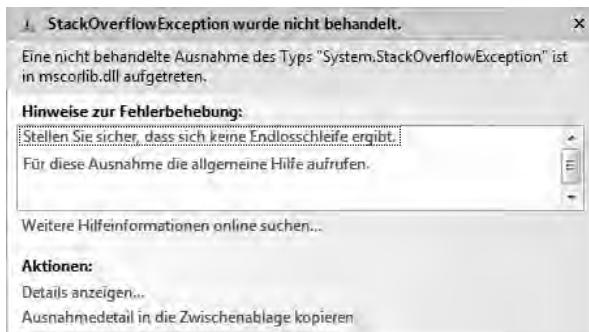


Abbildung 24.5 Durch den Zirkelverweis versagt `ToString` der Beispielklasse, da es sich immer wieder selbst aufruft und dabei allen verfügbaren Speicher verbraucht

Viele Serialisierungsalgorithmen versagen ebenfalls durch eine solche Konstellation der Datenklassen. Dass die `DeepCopy`-Methode jedoch funktioniert und der .NET-Serializer keine Fehlermeldung auslöst, können Sie ganz einfach testen, indem Sie alle `Console.WriteLine`-Befehle auskommentieren. Das Programm passiert nach einem erneuten Start anstandlos die Zeile

```
Dim locAdrKopie As Adresse = CType(ADObjectCloner.DeepCopy(locErsteAdr), Adresse)
```

beendet anschließend ordnungsgemäß und beweist so, dass der Serialisierungsalgorithmus des Frameworks über Zirkelverweise erhaben ist.

Serialisierung von Objekten unterschiedlicher Versionen beim Einsatz von BinaryFormatter oder SoapFormatter-Klassen

Einer sehr wichtigen Sache sollten Sie sich bewusst sein: Wenn Sie zur Serialisierung BinaryFormatter oder SoapFormatter verwenden, um Objekte in Dateien zu serialisieren, dann können Sie bei einem Update eines Programms leicht Gefahr laufen, in Versionskonflikte zu geraten.

Angenommen, Sie verfügen über eine Datenebene in Ihrer Applikation, die durch ein Objekt repräsentiert wird. Dieses Objekt kapselt alle Daten Ihrer Anwendung. Da Sie dem Anwender natürlich die Möglichkeit geben müssen, die Daten als Datei auf einem Datenträger zu speichern, nutzen Sie dafür die Serialisierungsfunktionen von .NET.

Nun erweitern Sie durch ein Update Ihr Programm, und es kommen einige Eigenschaften zur Datenklasse hinzu. Da sich die Deserialisierungsfunktionen beider Serialisierungsklassen am Objekt orientieren, erwartet es die Daten zu einigen Eigenschaften, die es im alten Objektmodell natürlich nicht gegeben hat. Der Deserialisierungsversuch schlägt in diesem Fall mit einer Ausnahme fehl.

Eine Möglichkeit, das zu verhindern, besteht durch den Einsatz der so genannten `SerializationBinder`-Klasse, die für die Lösung des Versionsproblems konzipiert wurde. Mehr über dieses Objekt erfahren Sie in der Visual Studio-Online-Hilfe.

Eine andere Möglichkeit, das Versionsproblem in den Griff zu bekommen, ist die Anwendung der XML-Serialisierung, die im folgenden Abschnitt besprochen wird.

XML-Serialisierung

Die XML-Serialisierung erfolgt generell nach dem gleichen Prinzip, wie Sie es beim Einsatz von BinaryFormatter und SoapFormatter kennengelernt haben. Die XML-Serialisierung hat im Gegensatz dazu entscheidende Vorteile:

- Das Problem mit der Serialisierung von Klassen unterschiedlicher Versionen betrifft sie nicht, denn es werden nur die Daten deserialisiert, die einerseits vorhanden sind, andererseits auch in den zu deserialisierenden Klassen zur Verfügung stehen.
- XML-Code ist nicht nur leichter lesbar, sondern kann auch von den verschiedensten Programmen importiert und weiterverarbeitet werden.

Allerdings müssen Sie beim Erstellen von Klassen, die Sie im XML-Format serialisieren wollen, auch einige Kompromisse eingehen:

- Wie bei anderen Klassen, die Sie mit .NET-Serialisierungstechniken serialisieren, müssen die Klassen mit dem `Serializable`-Attribut gekennzeichnet werden.

- Eine Klasse, die im XML-Format serialisiert werden soll, muss über einen parameterlosen Konstruktor (`Sub New()`) verfügen.
 - Nur Member-Variablen bzw. Eigenschaften, die öffentlichen Zugriff haben, können serialisiert werden.
- Wenn Ihre Klassen diese Voraussetzungen erfüllen, steht einer Serialisierung nichts mehr im Weg.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 24\\XMLSerialization

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Beispielprojekt entspricht im Großen und Ganzen dem letzten Beispiel aus Kapitel 23. Es erlaubt das Generieren von Zufallsadressen und deren Darstellung in einer Liste. Im Gegensatz zum Beispiel aus Kapitel 23 verfügt es jedoch über eine richtige Menüstruktur und in dieser Ausbaustufe über ein paar zusätzliche Funktionen, die über dieses Menü erreichbar sind:

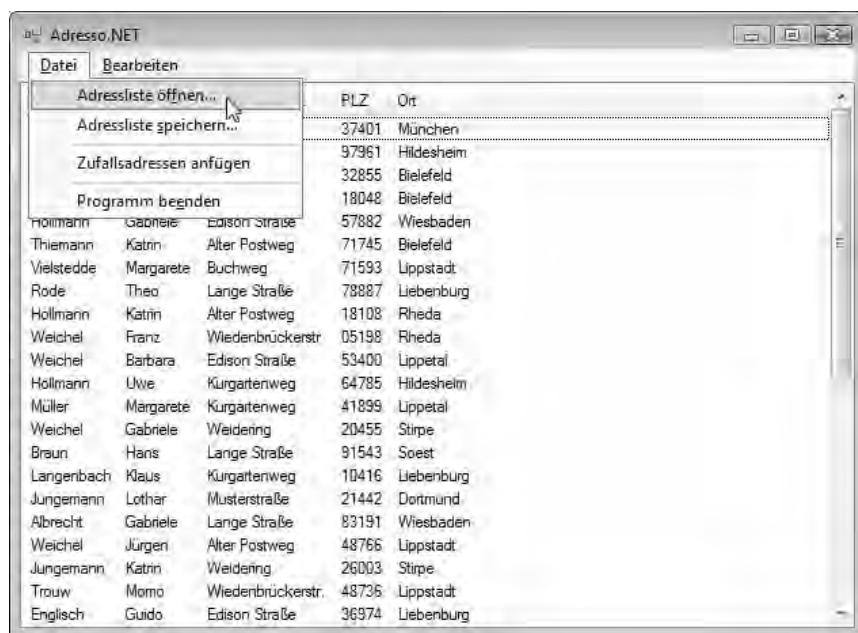


Abbildung 24.6 Die Adressenverwaltung »Adresso.NET« erlaubt es in dieser Ausbaustufe Zufallsadressen zu generieren und erstellte Listen in Dateien abzuspeichern und zu laden

Sie können eine beliebige Anzahl zufälliger Adressen erstellen, indem Sie den Menüpunkt *Zufallsadressen anfügen* wiederholt aufrufen. Mit *Adressliste speichern* erstellen Sie die so generierte Adressliste als Datei im XML-Format. Über den umgekehrten Weg können Sie eine Liste, die als XML-Datei vorliegt, über den Menüpunkt *Adressliste öffnen* in die entsprechenden Adresse-Objekte umwandeln, die dann wieder im ListView-Steuerelement des Programms angezeigt werden.

Neben der Darstellung der Liste und der Auswertung der Menüereignisse, die weitestgehend dem Beispiel aus Kapitel 37 entsprechen, und auf die ich aus diesem Grund an dieser Stelle nicht nochmals eingehen möchte, besteht der Code dieses Beispiels aus zwei zentralen Klassen.

Die Klasse Adresse kennen Sie aus vorherigen Beispielen bereits, und ihre Codebesonderheiten sind schnell erklärt:

```
<Serializable()>
Public Class Adresse

    'Membervariablen, die die Daten halten:
    Protected myMatchcode As String
    Protected myName As String
    Protected myVorname As String
    Protected myStraße As String
    Protected myPLZ As String
    Protected myOrt As String

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse.
    ''' </summary>
    ''' <remarks>Ein parameterloser Konstruktor wird benötigt,
    ''' um XML-Serialisierung zu ermöglichen.</remarks>
    Sub New()
        MyBase.New()
    End Sub

    'Konstruktor - legt eine neue Instanz an
    Sub New(ByVal Matchcode As String, ByVal Name As String, ByVal Vorname As String, _
            ByVal Straße As String, ByVal Plz As String, ByVal Ort As String)
        myMatchcode = Matchcode
        myName = Name
        myVorname = Vorname
        myStraße = Straße
        myPLZ = Plz
        myOrt = Ort
    End Sub

    Public Overridable Property Matchcode() As String
        Get
            Return myMatchcode
        End Get
        Set(ByVal Value As String)
            myMatchcode = Value
        End Set
    End Property
    .
    . 'Der Code der anderen Eigenschaften wurde aus Platzgründen ausgelassen.
    .
End Class
```

Drei Dinge sind wichtig, damit die XML-Serialisierung mit Auflistungen von Elementen dieses Typs funktionieren:

- Das Attribut `Serializable` muss auf die Klasse angewendet werden (siehe oberste, fett hervorgehobene Listingzeile).
- Die Klasse muss über einen parameterlosen Konstruktor verfügen (darunter, ebenfalls in Fettschrift).
- Alle Eigenschaften, die serialisiert werden sollen, müssen öffentlich zugänglich sein. Im Beispielisting ist die `Matchcode`-Eigenschaft (die im Übrigen in der Formularliste nicht dargestellt wird) deswegen auch mit dem `Public`-Modifizierer ausgestattet.

Gespeichert werden die einzelnen Adresse-Elemente in einer Auflistung namens `Adressen`, die aus der generischen Klasse `List(Of Adresse)` abgeleitet wird (mehr zur generischen Auflistung erfahren Sie ebenfalls in Kapitel 23). Diese Klasse sieht folgendermaßen aus:

```
<Serializable()>
Public Class Adressen
    Inherits List(Of Adresse)

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse.
    ''' </summary>
    ''' <remarks></remarks>
    Sub New()
        MyBase.New()
    End Sub

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse
    ''' und ermöglicht das Übernehmen einer vorhandenen Auflistung in diese.
    ''' </summary>
    ''' <param name="collection">Die generische Adresse-Auflistung, deren Elemente
    ''' in diese Auflistungsinstanz übernommen werden sollen.</param>
    ''' <remarks></remarks>
    Sub New(ByVal collection As ICollection(Of Adresse))
        MyBase.New(collection)
    End Sub

    ''' <summary>
    ''' Serialisiert alle Elemente dieser Auflistung in eine XML-Datei.
    ''' </summary>
    ''' <param name="Dateiname">Der Dateiname der XML-Datei.</param>
    ''' <remarks></remarks>
    Sub XMLSerialize(ByVal Dateiname As String)
        Dim locXmlWriter As New XmlSerializer(GetType(Adressen))
        Dim locXmlDatei As New StreamWriter(Dateiname)
        locXmlWriter.Serialize(locXmlDatei, Me)
        locXmlDatei.Flush()
        locXmlDatei.Close()
    End Sub

    ''' <summary>
    ''' Generiert aus einer XML-Datei eine neue Instanz dieser Auflistungsklasse.
    ''' </summary>
    ''' <param name="Dateiname">Name der XML-Datei, aus der die Daten für
    ''' diese Auflistungsinstanz entnommen werden sollen.</param>
    ''' <returns></returns>
```

```

'<remarks></remarks>
Public Shared Function XmlDeserialize(ByVal Dateiname As String) As Adressen
    Dim locXmlLeser As New XmlSerializer(GetType(Adressen))
    Dim locXmlDatei As New StreamReader(Dateiname)
    Return CType(locXmlLeser.Deserialize(locXmlDatei), Adressen)
End Function

'<summary>
'<summary> Liefert eine Instanz dieser Klasse mit Zufallsadressen zurück.
'</summary>
'<param name="Anzahl">Anzahl der Zufallsadressen, die erzeugt werden sollen.</param>
'<returns></returns>
'<remarks></remarks>
Public Shared Function ZufallsAdressen(ByVal Anzahl As Integer) As List(Of Adresse)
.
.
.
    ' Ausgelassener Code; unwichtig an dieser Stelle
.
.
.
End Function
End Class

```

Auch diese Klasse, die die Adresse-Elemente speichert, erfüllt die Voraussetzungen für die XML-Serialisierung – Sie verfügt über das notwendige Attribut, über einen parameterlosen Konstruktor und über öffentliche Eigenschaften. Die einzige öffentliche Eigenschaft, auf die es bei dieser Klasse ankommt, ist übrigens im Code nicht zu erkennen: Es ist Items, die durch die Basisklasse List(Of Adresse) definiert wurde und per Vererbung natürlich auch in der neuen Adressen-Klasse zur Verfügung steht. Sie erlaubt dem XML-Serialisierer Zugriff auf die einzelnen Adresse-Elemente und damit letzten Endes auf jedes einzelne Datenfeld von Adresse.

Beim ersten Blick auf den Code dieser Klasse mag es vielleicht verwirren, dass die Methode zum Serialisieren des Klasseninhalts eine nicht statische, die zum Deserialisieren hingegen eine statische ist. Doch das ist klar: Beim Deserialisieren gibt es noch keine Instanz der Adressen-Klasse; also gibt es auch keine Member-Methode die aufgerufen werden könnte, denn die Instanz mit den Adresse-Elementen geht ja erst durch den Deserialisierungsprozess hervor!

Anders ist es beim Serialisieren: Hier sind die einzelnen Elemente bereit in der Instanz von Adressen vorhanden, und aus diesem Grund kann es auch eine Member-Methode sein, die das Serialisieren vornimmt; sie greift auf ihre eigenen Elemente zu.

Das Anwenden dieser beiden Methoden ist schließlich eine Kleinigkeit. Die beiden folgenden Ereignishandlungsmethoden, die das Auswählen der entsprechenden Menüeinträge auswerten, sind dafür verantwortlich (relevante Codeteile sind wieder fett hervorgehoben):

```

'Wird aufgerufen, wenn Anwender Datei/Adresslist öffnen wählt.
Private Sub tsmAdresslisteLaden_Click(ByVal sender As System.Object,
                                ByVal e As System.EventArgs) Handles tsmAdresslisteLaden.Click

    Dim dateiÖffnenDialog As New OpenFileDialog
    With dateiÖffnenDialog
        .CheckFileExists = True
        .CheckPathExists = True
        .DefaultExt = "*.xml"
        .Filter = "Adresse XML-Dateien" & _

```

```

    " (" & "*.&adrxml" & ")|" & "*.&adrxml" & "|Alle Dateien (*.*)|*.*"
Dim dialogErgebnis As DialogResult = .ShowDialog
If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
    Exit Sub
End If

'Adressen deserialisieren
myAdressen = Adressen.XmlDeserialize(.FileName)

'Liste neu darstellen
ElementeDarstellen()
End With
End Sub

'Wird aufgerufen, wenn Anwender Datei/Adressliste speichern wählt.
Private Sub tsmAdresslisteSpeichern_Click(ByVal sender As System.Object,
                                         ByVal e As System.EventArgs) Handles tsmAdresslisteSpeichern.Click

    Dim dateiSpeichernDialog As New SaveFileDialog

    With dateiSpeichernDialog
        .CheckPathExists = True
        .DefaultExt = "*.xml"
        .Filter = "Adresse XML-Dateien" & " (" & "*.&adrxml" & ")|" & _
                  "*.&adrxml" & "|Alle Dateien (*.*)|*.*"
        Dim dialogErgebnis As DialogResult = .ShowDialog
        If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
            Exit Sub
        End If

        'Adressen serialisieren
        myAdressen.XMLSerialize(.FileName)
    End With
End Sub
End Class

```

Prüfen der Versionsunabhängigkeit der XML-Datei

Die XML-Datei, die dieses Programm generiert, sieht auszugweise etwa wie in Abbildung 24.7 aus, und Sie können sich »Ihre« Version mit dem Notepad anschauen.

Wenn Sie ein Datenfeld aus dieser XML-Datei entfernen, etwa wie in Abbildung 24.7 zu sehen, und anschließend abspeichern, können Sie diese XML-Datei dennoch deserialisieren lassen – für die Ort-Eigenschaft des betroffenen Adresse-Objektes wird dann beim Deserialisieren eben keine Zuordnung vorgenommen. Der Deserialisierungsprozess schlägt aber nicht fehl.

```

<?xml version="1.0" encoding="utf-8"?>
<ArrayOfAdresse xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Adresse>
    <Matchcode>ViMa00000001</Matchcode>
    <Name>Vielstedde</Name>
    <Vorname>Margarete</Vorname>
    <Straße>Mühlweg</Straße>
    <PLZ>37401</PLZ>
    <Ort>München</Ort>
  </Adresse>
  <Adresse>
    <Matchcode>TrBr00000002</Matchcode>
    <Name>Trouw</Name>
    <Vorname>Britta</Vorname>
    <Straße>Alter Postweg</Straße>
    <PLZ>97961</PLZ>
    <Ort>Hildesheim</Ort>
  </Adresse>
</ArrayOfAdresse>

```

Abbildung 24.7 Auch wenn Sie ein Datenfeld aus der XML-Datei entfernen, lässt sich die Datei anschließend noch deserialisieren

Genau so könnten Sie der Adresse-Klasse eine zusätzliche Eigenschaft hinzufügen. »Alte« Versionen der Serialisierungsdateien ließen sich dennoch weiterverwenden.

TIPP Wenn Sie XML-Serialisierungslogik in eigenen Klassen implementieren wollen, nutzen Sie am besten die Codeausschnittsbibliothek (über die Sie in Kapitel 8 mehr erfahren). Sie finden die entsprechenden Codeausschnitte unter *XML/Klassendaten aus einer XML-Datei* lesen sowie *XML/Klassendatei in eine XML-Datei schreiben*.

Serialisierungsfehler bei KeyedCollection

Die generische KeyedCollection-Klasse eignet sich als Auflistungsbasis für Geschäftsobjekte ideal, denn:

- Sie stellt, da sie generisch ist, eine typsichere Auflistung des jeweiligen Elementes dar.
- Objekte lassen sich wahlweise über einen Index *oder* über einen Schlüssel abrufen. Den letzteren muss das Objekt allerdings selber zur Verfügung stellen – Kapitel 23 liefert im entsprechenden Abschnitt mehr zu diesem Thema.
- Da die KeyedCollection auch über einen Indexer verfügt, stellt sie auch eine Enumerierungstechnik für `ForEach` zur Verfügung, obwohl Sie zusätzlich Wörterbuchfunktionalität implementiert.

An Flexibilität ist diese Klasse also in Sachen Auflistung kaum zu übertreffen.

Allerdings ist diese Kombination auch gerade dann eine Krux, wenn sich Schlüsseldatentyp und Indexer typtechnisch ins Gehege kommen, denn:

Stellen Sie sich vor, Sie haben eine Klasse wie die folgende, die den eindeutigen Schlüssel eines Geschäftsobjektes über eine ID zur Verfügung stellt (und dieser Anwendungsfall ist keinesfalls konstruiert, denn gerade in der Datenbankprogrammierung haben Tabellen oft den Datentyp Integer als primären Schlüssel und Hauptindex):

BEGLEITDATEIEN

Das Projekt zu diesem Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 24\\KeyedCollection

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Public Class Adresse

    Private myIDAdresse As Integer
    Private myNachname As String
    Private myVorname As String

    Sub New(ByVal IDAdresse As Integer, ByVal Nachname As String, ByVal Vorname As String)
        myIDAdresse = IDAdresse
        myNachname = Nachname
        myVorname = Vorname
    End Sub

    Public Property IDAdresse() As Integer
        Get
            Return myIDAdresse
        End Get
        Set(ByVal value As Integer)
            myIDAdresse = value
        End Set
    End Property

    Public Property Nachname() As String
        Get
            Return myNachname
        End Get
        Set(ByVal value As String)
            myNachname = value
        End Set
    End Property

    Public Property Vorname() As String
        Get
            Return myVorname
        End Get
        Set(ByVal value As String)
            myVorname = value
        End Set
    End Property
End Class
```

Dieses Geschäftsobjekt speichert also eine simple Adresse – wesentliche Eigenschaften sind hier aus Platzgründen ausgelassen.

Was viel wichtiger ist: Ein Geschäftsobjekt dieses Typs identifiziert sich eindeutig durch seine IDAdresse-Eigenschaft, und das Problem dabei ist: Diese ist vom Typ Integer. Warum das ein Problem darstellt, zeigt die Implementierung einer KeyedCollection-Auflistung auf Basis dieser Klasse, denn es bietet sich mehr als an, dass IDAdresse auch als Quelle für die Schaffung des eindeutigen Schlüssels verwendet wird:

```

Public Class Adressen
    Inherits System.Collections.ObjectModel.KeyedCollection(Of Integer, Adresse)

    Protected Overrides Function GetKeyForItem(ByVal item As Adresse) As Integer
        Return item.IDAdresse
    End Function
End Class

```

Während der Aufbau einer KeyedCollection in dieser Struktur noch ohne Probleme möglich ist,

```

Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
    MyBase.OnLoad(e)
    myAdressen.Add(New Adresse(2, "Löffelmann", "Klaus"))
    myAdressen.Add(New Adresse(4, "Heckhuis", "Jürgen"))
    myAdressen.Add(New Adresse(6, "Sonntag", "Miriam"))
    myAdressen.Add(New Adresse(8, "Heckhuis", "Jürgen"))
    myAdressen.Add(New Adresse(10, "Vielstedde", "Anja"))
End Sub

```

zeigt sich beim Abrufen der Elemente bereits eine Besonderheit, wenn es sich beim Schlüssel um den Datentyp Integer handelt:

```

Private Sub btnObjekteAusgeben_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnObjekteAusgeben.Click
    For count As Integer = 0 To 4
        Debug.Print(myAdressen()
    Next

```

Item (key As Integer) As KeyedCollection.Adresse
key:
 The key of the element to get.

Abbildung 24.8 Handelt es sich um einen Integer-Schlüssel, bietet die KeyedCollection-Auflistung nur eine Signatur zum Abrufen der Elemente an. Dass IntelliSense hier übrigens die englischen Hilfetexte liefert, scheint an »unzureichender« Übersetzung der verwendeten Visual Studio-Version zu liegen!

Soweit macht das auch Sinn. Da Indexer und Schlüssel in diesem Fall vom gleichen Datentyp sind, könnten weder Compiler noch Framework unterscheiden, welche »Version« der Abruffunktion sie verwenden sollen – die durch den Indexer oder die über den Schlüssel. Einzig und allein die Methodensignatur (also welche Parametertypen übergeben werden) entscheidet nämlich darüber, und da es in diesem Fall die gleichen Typen sind, kann keine Signaturenunterscheidung stattfinden. Ergo: Es gibt nur eine Möglichkeit, an das Element heranzukommen.

Anders wäre das, wenn unser Schlüssel beispielsweise vom Datentyp Long wäre.² In diesem Fall würde der Indexer über einen Integer, der Schlüssel über den Datentyp Long angesprochen werden, und dementsprechend würde Ihnen schon IntelliSense zwei Möglichkeiten zum Abrufen der Elemente anbieten:

² Lesen Sie aber bitte diesen Abschnitt zunächst zu Ende, bevor Sie jetzt schon alle Ihre KeyedCollection-basierten Auflistungsklassen auf den Datentyp Long als Schlüssel umstellen!

```

Private Sub btnObjekteAusgeben_Click(ByVal sender As System.Object, ByVal e As EventArgs)
    Dim myAdressen As KeyedCollection(Of Long, Adresse)
    For count As Integer = 0 To 4
        Debug.Print(myAdressen(key))
    Next
End Sub

```

key: The key of the element to get.
index: The zero-based index of the element to get or set.

Abbildung 24.9 Unterscheiden sich Schlüssel und Indexer im Datentyp, gibt es bei der KeyedCollection tatsächlich zwei Möglichkeiten, an ein einzelnes Element heranzukommen

Diese Integer/Integer-Problematik wäre noch nicht weiter tragisch, denn durch den von KeyedCollection implementierten Enumerator gäbe es immer noch die Möglichkeit, die Objekte einerseits per Schlüssel, andererseits nacheinander abzurufen. Für letztere Methode würden Sie einfach eine ForEach-Schleife verwenden.

Richtig problematisch wird es aber, wenn Sie versuchen, die Daten zu serialisieren. Denn dann passiert Folgendes:

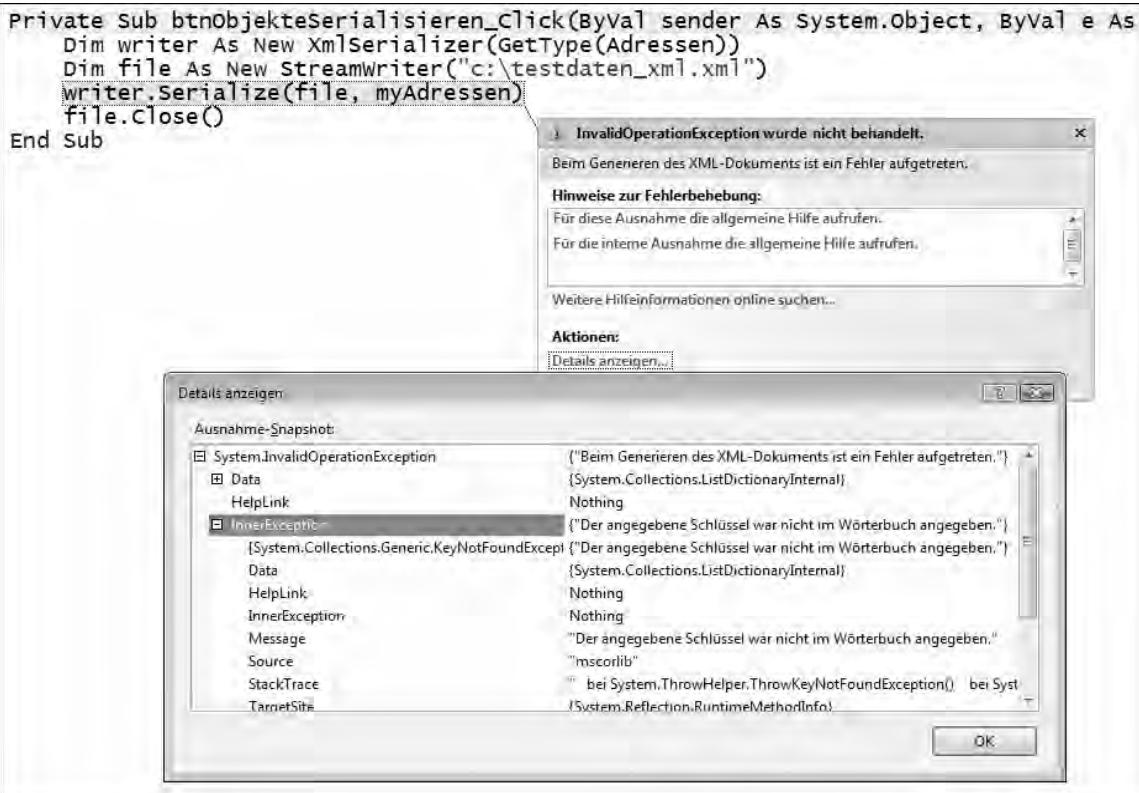


Abbildung 24.10 Beim Serialisieren stellen sich massive Probleme ein, wenn Schlüsseltyp und Indexer gleichermaßen vom Typ Integer sind!

Der Serialisierungsalgorithmus kommt hier offensichtlich nicht mehr mit der Doppelbelegung der Datentypen zurecht. Offensichtlich versucht er über eine Zählvariable der Reihe nach auf die Objekte zuzugreifen, die aber der Indexer von KeyedCollection fälschlicherweise als Schlüssel interpretiert. Da in unserem Beispiel die IDs nicht der Nummerierungsreihenfolge der Objekte entsprechen, schlägt dieser Versuch fehl, und das Programm steigt mit einer Ausnahme, wie in der Abbildung zu sehen, aus.

Workaround

Abhilfe können Sie bei diesem Szenario auf folgende Weise schaffen:

Sie schreiben sich eine Helfer-Klasse, die nur zur Signaturenunterscheidung dient, die aber nichts weiter macht, als den Integer-Datentyp zu kapseln. Diese könnte beispielsweise folgendermaßen ausschauen:

```
<Serializable()>
Public Class IntKey

    Private myKey As Integer

    'Wichtig für die XML-Serialisierung: Parameterloser Konstruktor!
    Sub New()
        MyBase.new()
    End Sub

    Sub New(ByVal Key As Integer)
        myKey = Key
    End Sub

    Public ReadOnly Property Key() As Integer
        Get
            Return myKey
        End Get
    End Property
End Class
```

HINWEIS Auf den ersten Blick könnte man meinen, dass auch die Verwendung des Long-Datentyps als Schlüssel ausreichend sein müsste, um das Problem in den Griff zu bekommen. Ich persönlich wäre dabei aber vorsichtig, da durch die implizite Konvertierungsmöglichkeit von Integer in Long unter Umständen dieselben Effekte zu Tage treten könnten. Mit dem hier vorgestellten Workaround funktioniert es jedoch einwandfrei.

Die geänderte Eigenschaft in der Adresse-Klasse sieht nun wie folgt aus:

```
Public Class Adresse

    Private myIDAdresse As IntKey
    Private myNachname As String
    Private myVorname As String

    'Den brauchen wir fürs Deserialisieren!
    Sub New()
        MyBase.new()
    End Sub

    Sub New(ByVal IDAdresse As IntKey, ByVal Nachname As String, ByVal Vorname As String)
        myIDAdresse = IDAdresse
        myNachname = Nachname
        myVorname = Vorname
    End Sub

    Public Property IDAdresse() As IntKey
        Get
            Return myIDAdresse
        End Get
    End Property
End Class
```

```

End Get
Set(ByVal value As IntKey)
    myIDAdresse = value
End Set
End Property
.
.
.

```

Und dementsprechend hat sich auch die KeyedCollection geändert:

```

Public Class Adressen
    Inherits System.Collections.ObjectModel.KeyedCollection(Of IntKey, Adresse)

    Protected Overrides Function GetKeyForItem(ByVal item As Adresse) As IntKey
        Return item.IDAdresse
    End Function
End Class

```

Das Hinzufügen von Elementen zur Auflistung bereitet zwar ein kleines bisschen mehr Aufwand:

```

Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
    MyBase.OnLoad(e)
    myAdressen.Add(New Adresse(New IntKey(2), "Löffelmann", "Klaus"))
    myAdressen.Add(New Adresse(New IntKey(4), "Heckhuis", "Jürgen"))
    myAdressen.Add(New Adresse(New IntKey(6), "Sonntag", "Miriam"))
    myAdressen.Add(New Adresse(New IntKey(8), "Heckhuis", "Jürgen"))
    myAdressen.Add(New Adresse(New IntKey(10), "Vielstedde", "Anja"))
End Sub

```

Aber dafür können Sie die einzelnen Elemente nunmehr zielsicher und ohne Zweideutigkeiten sowohl über den Indexer (siehe Grafik) *als auch* über den Schlüssel abrufen:

```

Private Sub btnObjekteAusgeben_Click(ByVal sender As System.Object,
    For count As Integer = 0 To 4
        Debug.Print(myAdressen(count).Nachname & ", " & myAdressen(
    Next
End Sub

```

▲ 2 von 2 Item (index As Integer) As KeyedCollection.Adresse
 index: The zero-based index of the element to get or set.

Abbildung 24.11 Mit der Workaround-Lösung funktioniert nun der Abruf über Schlüssel und Indexer perfekt

BEGLEITDATEIEN

Eine Version des Programms, die dieses Verfahren nutzt, finden Sie im Verzeichnis

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 24\\KeyedCollection2

und Sie werden sehen, dass es auch beim Serialisieren und Deserialisieren einwandfrei funktioniert.

Kapitel 25

Attribute und Reflection – Wenn Klassen ein Bewusstsein entwickeln

In diesem Kapitel:

Genereller Umgang mit Attributen	728
Einführung in Reflection	732
Erstellung benutzerdefinierter Attribute und deren Erkennen zur Laufzeit	737

Wenn Sie größere Anwendungen entwickeln, kommen Sie an Attributen nicht vorbei. Attribute sind besondere Klassen, die keine spezielle Funktion ausführen, sondern nur ein Hinweis für eine andere Instanz sind, ein bestimmtes Element Ihrer Anwendung auf besondere Weise zu behandeln.

Sie können nicht nur auf die im Framework vorhandenen Attribute zurückgreifen, sondern auch eigene Attributklassen entwerfen. In diesem Fall müssen Sie allerdings auch Techniken beherrschen, die das Erkennen von Attributen zur Laufzeit ermöglichen – und an dieser Stelle kommt *Reflection* ins Spiel.

Die Reflection-Techniken im Framework stellen im Prinzip den Psychologen Ihres Programms dar, und sie helfen Ihnen, Informationen über bestimmte Assemblies, Klassen, Methoden, Eigenschaften und weitere Elemente zur Laufzeit Ihres Programms zu erhalten. Zu diesen Elementen gehören auch Attribute. Wenn Sie herausfinden wollen, ob eine bestimmte Methode einer Klasse oder Klasseninstanz beispielsweise mit einem bestimmten Attribut ausgestattet ist, verwenden Sie die Techniken der Reflection, um diese Attribute zu ermitteln. Gerade benutzerdefinierte Attribute und Reflection sind also zwei Themenbereiche, die eng miteinander verknüpft sind, und aus diesem Grund finden Sie diese beiden Themen auch als in einem gemeinsamen Kapitel an dieser Stelle.

Damit klarer wird, welche enormen Möglichkeiten Ihnen Attribute und Reflection bieten können, finden Sie im Folgenden ein paar praktische Beispiele:

- Sie möchten, dass eine bestimmte Klasse Ihrer Datenbankanwendung automatisch die vorhandenen Datenbankfelder synchronisiert: Bestimmte Klassen sollen Tabellen darstellen, die Eigenschaften dieser Klassen die Datenfelder. Durch Attribute hätten Sie die Möglichkeit, diese Klassen zu kennzeichnen und die Datenbankdatei zur Laufzeit zu synchronisieren.
- Formulare könnten sich in Abhängigkeit bestimmter Klassen selber erstellen und die Eingabe bzw. Änderung einer Klasseninstanz übernehmen.
- Eine Ableitung des *ListView*-Steuerelements könnte ein Array mit Instanzen besonders gekennzeichneter Klassen automatisch anzeigen. Attribute würden bestimmen, welche Eigenschaften eines Array-Elementes als Spalte verwendet würden. Die Reihenfolge der Spalten ließe sich durch weitere Attribute bestimmen. Dieses Beispiel finden Sie übrigens am Ende dieses Kapitels als Steuerelement realisiert.

Sie sehen: Beispiele gibt es viele, und vielleicht ahnen Sie schon, welch mächtiges Werkzeug Ihnen das Framework mit Attributen und Reflection in die Hände legt.

Genereller Umgang mit Attributen

Wie in der Einführung schon kurz angerissen, und wie Sie es in den zahlreichen vergangenen Beispielprogrammen schon zigfach gesehen haben, werden Attribute ganz anders als herkömmliche Klassen verwendet (wenn sie auch auf dieselbe Weise erstellt werden). Attribute statthen Klassen oder Prozeduren mit besonderen Eigenschaften aus. Genauso, wie die Verwendung von **Fettschrift innerhalb eines Absatzes dieses Buchs** selbst nicht den Sinn des Geschriebenen verändert, so erfüllt das Attribut Fettschrift dennoch seinen Zweck: Es ist der Hinweis für Sie, die so markierte Textstelle aufmerksamer zu lesen.

Attribute »markieren« eine Klasse oder eine Prozedur in Visual Basic, indem sie in Kleiner-/Größerzeichen eingeschlossen vor die Klassen- bzw. Prozedurdefinition gesetzt werden, etwa:

```
<MeinAttribute(MöglichParameter)> Class MeineKlasse  
End Class
```

Da Zeilen, in denen Attribute verwendet werden, auf diese Weise unnötig lang und damit schlecht lesbar sind, verwendet man in Visual Basic üblicherweise das *Underscore*-Zeichen (»_«), um die Zeile zu umbrechen:

```
<MeinAttribute(MöglichParameter)> _  
Class MeineKlasse  
End Class
```

HINWEIS Achten Sie bei der Verwendung des Umbruchzeichens darauf, dass vor dem Umbruchzeichen ein Leerzeichen platziert wird!

Die Auswirkungen, die ein Attribut auf den folgenden Code hat, werden nicht durch die Attribut-Klasse gesteuert, sondern ausschließlich von den Instanzen, die die Elemente unter die Lupe nehmen, die mit Attributen versehen sind. Attribut-Klassen sind in der Regel Klassen, die keine wirkliche Funktionalität zur Verfügung stellen; sie dienen lediglich als Container für Informationen. Behalten Sie diese wichtige Aussage im Hinterkopf, wenn Sie den Einsatz von Attributen planen.

WICHTIG Attributklassen enden grundsätzlich auf den Namen ... *Attribute*. Dennoch müssen Sie den Namenszusatz *Attribute* nicht mit angeben, um eine *Attribute*-basierte Klasse für Kennzeichnungszwecke zu verwenden.

Einsatz von Attributen am Beispiel von **ObsoleteAttribute**

Ein Beispiel soll das verdeutlichen: Die *ObsoleteAttribute*-Klasse dient beispielsweise dazu, eine Prozedur oder Klasse als »überholt« (d.h. sie wird in zukünftigen Versionen eventuell nicht mehr verfügbar sein) zu kennzeichnen. Gesetzt den Fall, Sie haben vor einem Jahr eine Klassenbibliothek für .NET entwickelt, die Sie Ihren Kunden nun in erweiterter und überarbeiteter Form zugänglich machen wollen. Im Rahmen der Umbauarbeiten haben Sie festgestellt, dass Sie bestimmte Funktionen nicht mehr benötigen, weil Ihre Klassenbibliothek entweder sehr viel automatisierter arbeiten kann, oder es Sinn ergibt, bestimmte Funktionen durch neuere Funktionen zu ersetzen, da diese viel effizienter arbeiten.

Natürlich können Sie in der neuen Version die alten, überflüssigen Funktionen nicht einfach entfernen. Würde der Anwender Ihrer Klassenbibliothek nämlich anschließend sein Programm mit der neuen Version kompilieren, wären Fehlfunktionen vorprogrammiert (das Programm ließe sich wahrscheinlich gar nicht erst kompilieren). Mithilfe des *Obsolete*-Attributes können Sie den Entwickler aber gefahrlos darauf aufmerksam machen, eine bestimmte Funktion nicht mehr zu verwenden. Sie setzen in diesem Fall das *ObsoleteAttribute* vor die entsprechende Klasse/Prozedur und geben zusätzlich eine Hinweismeldung an.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

... \VB 2008 Entwicklerbuch \ D - Datenstrukturen \ Kapitel 25 \ AttributesDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Bei diesem Beispielprogramm kommt es gar nicht darauf an, das Programm zu starten. Wenn Sie es geladen haben, betrachten Sie vielmehr den Quellcode und wie die Aufgabenliste auf die Verwendung einer bestimmten Eigenschaft reagiert:



The screenshot shows the Microsoft Visual Studio IDE. On the left, the code editor displays a module named 'mdlMain' and a class named 'TestKl'. The 'TestKl' class contains a property 'myEigenschaft' with both 'Get' and 'Set' methods. The 'Get' method returns the value of 'myEigenschaft', and the 'Set' method assigns a new value to it. The 'mdlMain' module contains a 'Main' subroutine that creates an instance of 'TestKl', sets its 'myEigenschaft' to 'Dies ist ein Test', and then prints the values of 'AlteEigenschaft' and 'NeueEigenschaft' to the console. A tooltip is visible over the 'AlteEigenschaft' line, indicating it is obsolete.

The error list window on the right shows one warning:

Beschreibung	Datei	Zeile	Spalte	Projekt
'Public Property AlteEigenschaft() As String' ist veraltet. 'Sie sollten die AlteEigenschaft-Eigenschaft nicht mehr verwenden. Verwenden Sie stattdessen NeueEigenschaft.'	mdlMain.vb	16	77	AttributesDemo

Abbildung 25.1 Die Verwendung eines durch das Obsolete-Attribut gekennzeichnetes Element ruft mindestens eine entsprechende Warnmeldung hervor

Die Eigenschaft `AlteEigenschaft` der Klasse `TestKlasse` ist hier im Beispiel mit dem `Obsolete`-Attribut ausgestattet. Der Visual Basic Compiler erkennt dieses Attribut und zeigt in der Aufgabenliste eine Warnung, die den Entwickler auf diese Tatsache hinweist. Die `ObsoleteAttribute`-Klasse selbst hat aber nichts mit der Ausgabe des Textes zu tun, außer, dass sie den Text, der ausgegeben werden soll, zur Verfügung stellt. Die eigentliche Ausgabe des Textes erfolgt vom Visual Basic Compiler bzw. von der Entwicklungsumgebung.

TIPP Obwohl die `ObsoleteAttribute`-Klasse im Beispiel zur Kennzeichnung der `AlteEigenschaft`-Eigenschaft verwendet worden ist, lässt sich die Eigenschaft verwenden und das Programm damit auch kompilieren und starten. Wenn Sie möchten, dass der Einsatz einer veralteten Eigenschaft zum Compiler-Fehler führt, der dafür sorgt, dass sich das Programm nicht mehr starten lässt, ändern Sie hinter dem Meldungsstring im `ObsoleteAttribute`-Konstruktoren den zweiten (booleschen) Parameter in `True`, der die Kompilierung des Programms verhindert, das diese alte Version der Eigenschaft verwendet.

Die speziell in Visual Basic verwendeten Attribute

Die wichtigsten Attribute haben Sie im Laufe der vergangenen Kapitel themenbezogen schon kennengelernt. Es gibt allerdings einige spezielle Attribute für Visual Basic selbst,¹ die in der folgenden Tabelle zusammengefasst sind:

Attribut	Zweck
<code>COMClassAttribute</code> -Klasse	Weist den Compiler an, die Klasse als COM-Objekt anzuseigen.
<code>VBFixedStringAttribute</code> -Klasse	Gibt die Größe einer Zeichenfolge mit fester Länge in einer Struktur an, die mit Dateiein- und -ausgabefunktionen verwendet werden soll. Spezifisch für Visual Basic .NET.
<code>VBFixedArrayAttribute</code> -Klasse	Gibt die Größe eines festen Arrays in einer Struktur an, die mit Dateiein- und -ausgabefunktionen verwendet werden soll. Spezifisch für Visual Basic .NET.
<code>WebMethodAttribute</code> -Klasse	Ermöglicht das Aufrufen einer Methode mit dem SOAP-Protokoll. Wird in XML-Webdiensten verwendet.
<code>SerializableAttribute</code> -Klasse	Gibt an, dass eine Klasse serialisiert werden kann.
<code>MarshalAsAttribute</code> -Klasse	Stellt fest, wie ein Parameter zwischen dem verwalteten Code von Visual Basic .NET und nicht verwaltetem Code z.B. einer Windows-API gemarshallt werden soll. Wird von der Common Language Runtime verwendet.
<code>AttributeUsageAttribute</code> -Klasse	Gibt die Verwendungweise eines Attributs an.
<code>DllImportAttribute</code> -Klasse	Gibt an, dass die attributierte Methode als Export aus einer nicht verwalteten DLL implementiert ist.

Tabelle 25.1 Die speziellen Visual Basic-Attribute

¹ Das bedeutet nicht, dass Sie nur diese Attribute in Visual Basic verwenden dürfen. Sie können natürlich alle Attribute des Frameworks auch in Ihren eigenen Visual Basic-Anwendungen verwenden. Um eine Liste aller im Framework enthaltenen Attribute zu erhalten, rufen Sie die Online-Hilfe für die `Attribute`-Klasse auf und lassen sich anschließend alle abgeleiteten Klassen anzeigen.

Einführung in Reflection

Bevor wir im nächsten Schritt die beiden Themengebiete Reflection und Attribute miteinander verheiraten, lassen Sie uns zunächst einen Blick auf die Möglichkeiten der Reflection-Klassen selbst werfen.

Reflection selbst ist, wie am Anfang des Kapitels schon kurz zu erfahren war, der Oberbegriff für Techniken, die es einem Programm ermöglichen, etwas über Klassen zu erfahren, Klassen zu analysieren und auch Instanzen von Klassen programmtechnisch zu erstellen.

Natürlich ist es keine Kunst, eine Klasse programmtechnisch zu erstellen. Sie machen das immer, wenn Sie den Konstruktor einer Klasse verwenden. Aber darum geht es in diesem Fall auch gar nicht. Es geht darum, Klassen zu verwenden, von denen das Programm zum Zeitpunkt, an dem es gestartet wird, noch nichts weiß.

Angenommen, Sie möchten eine Funktion zur Verfügung stellen, die ein beliebiges Objekt als Parameter übernimmt und den Wert jeder einzelnen Eigenschaft auf dem Bildschirm ausgibt. Mit herkömmlichen Mitteln könnten Sie dieses Vorhaben nicht realisieren, denn: Da Sie im Vorfeld nicht wissen, welchen Objekttyp Sie zu erwarten haben, kennen Sie dessen Eigenschaftenamen auch nicht. Folglich können Sie die Werte dieser Eigenschaften auch nicht abrufen.

Sie müssen also Mittel und Wege finden, ein Objekt zu analysieren und zunächst zu ermitteln, über welche Eigenschaften es verfügt. Erst im zweiten Schritt können Sie, wenn Ihr Programm die Namen der Eigenschaften herausgefunden hat, die Inhalte der Eigenschaften herausfinden und schließlich auf dem Bildschirm ausgeben.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 25\\Reflection01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Zentraler Bestandteil dieses Programms ist die Adresse-Klasse, die Sie aus vergangenen Kapiteln kennen. Die Main-Prozedur dieser Konsolenanwendung macht nichts weiter, als eine neue Adresse-Instanz zu erstellen und die Untersuchungsergebnisse im Konsolenfenster darzustellen, etwa wie im folgenden Bildschirmauszug zu sehen:

```
Attribute der Klasse:Reflection.Adresse
Standardattribute:
    *AutoLayout, AnsiClass, NotPublic, Public, Serializable

Member-Liste:
    *GetHashCode, Method
    *Equals, Method
    *ToString, Method
    *get_Name, Method
    *set_Name, Method
    *get_ErfasstAm, Method
    *get_ErfasstVon, Method
    *get_BefreundetMit, Method
    *set_BefreundetMit, Method
    *get_Vorname, Method
```

```
*set_Vorname, Method
*get_Straße, Method
*set_Straße, Method
*get_PLZOrt, Method
*set_PLZOrt, Method
*ToStringShort, Method
*GetType, Method
*.ctor, Constructor
*Name, Property
Wert: Halek
*ErfasstAm, Property
Wert: 12.04.2004 09:06:27
*ErfasstVon, Property
Wert: Administrator
*BefreundetMit, Property
Wert: System.Collections.ArrayList
*Vorname, Property
Wert: Gaby
*Straße, Property
Wert: Buchstraße 223
*PLZOrt, Property
Wert: 32154 Autorhausen
```

Sie sehen, dass dieses Programm nicht nur in der Lage ist, eine (fast) vollständige Member-Liste der übergebenden Objekt-Instanz zu ermitteln; es kann darüber hinaus auch die Inhalte der Eigenschaften des Objektes anzeigen.

Bevor wir die Funktionsweise dieses Programms genauer unter die Lupe nehmen, lassen Sie mich zunächst ein paar Grundlagen zum Thema Typen und Reflection zwecks späteren besseren Verständnisses klären.

Die Type-Klasse als Ausgangspunkt für alle Typenuntersuchungen

Den Schlüssel dazu bilden Klassen und Methoden, die Sie im Reflection-Namespace vom Framework finden. Ausgangspunkt für alle Reflection-Operationen bildet dabei die so genannte Type-Klasse, die den Typ einer Objektvariablen zur Laufzeit ermitteln kann.

Die Type-Klasse selbst können Sie nicht instanziieren, da sie eine abstrakte Klasse darstellt. Sie können sie aber verwenden, um den Typ eines bestehenden Objekts zu ermitteln und festzuhalten. Zwar gehört die Type-Klasse selbst nicht zum Reflection-Namespace – Sie benötigen sie aber, um mit weiteren Klassen dieses Namensbereichs und deren Funktionen Informationen über die jeweilige Typdeklaration eines zu untersuchenden Objektes abzurufen, z.B. Konstruktoren, Methoden, Felder, Eigenschaften und Ereignisse. Auch Informationen über Modul und Assembly, in denen die Klasse bereitgestellt wird, lassen sich anschließend mit diesen Funktionen ermitteln.

Jedes Objekt im Framework stellt eine GetType-Funktion zur Verfügung, mit der ihr zugrundeliegendes Type-Objekt ermittelt werden kann. Darüber hinaus können Sie mit der statischen Funktionsvariante auch ein Type-Objekt erstellen, wenn nur der voll qualifizierte Name des Typs bekannt ist. Mit TypeOf in Zusammenhang mit dem Is-Operator können Sie einen Typenvergleich durchführen. Alternativ funktioniert das auch mit GetType, welches auch als Operator eingesetzt werden kann. Die folgenden Codeaussüge demonstrieren den generellen Einsatz dieser Funktionen:

```

'Ein paar Type-Experimente:
Dim locTest As New Adresse("Klaus", "Löffelmann", "Urlaubsgasse 17", "59555 Lippende")
Dim locType1, locType2 As Type
locType1 = locTest.GetType

'Wichtig: TypeOf funktioniert nur zusammen mit dem Is-Operator:
If TypeOf locTest Is Reflection01.Adresse Then
    Console.WriteLine("Adresse-Typ erkannt!")
Else
    Console.WriteLine("Adresse-Typ nicht erkannt!")
End If

'Und auch das ist eine Alternative, die dasselbe bewirkt:
If locTest.GetType Is GetType(Reflection01.Adresse) Then
    Console.WriteLine("Adresse-Typ erkannt!")
Else
    Console.WriteLine("Adresse-Typ nicht erkannt!")
End If

'So funktioniert's, wenn zwei Typobjekte
'miteinander verglichen werden sollen:

locType2 = GetType(Adresse)           ' Der Normalfall, GetType als Operator
locType2 = GetType(Reflection01.Adresse) ' Alternativ: mit Namespace
locType2 = Type.GetType("Reflection01.Adresse") ' Alternativ: aus String

Console.WriteLine("Der Typ " + locType1.FullName +
    CStr(IIf(locType1 Is locType2, " entspricht ", " entspricht nicht ")) + _
    "dem Typ " + locType2.FullName)

Console.ReadLine()

```

Dieses Programm würde die folgende Ausgabe produzieren:

```

Der Typ ReflectionDemo.Adresse entspricht dem Typ ReflectionDemo.Adresse
Adresse-Typ erkannt!

```

HINWEIS Dieser Codeschnipsel ist ebenfalls im besprochenen Beispielprojekt vorhanden (direkt im Anschluss an die *Main*-Prozedur, die mit `Exit Sub` endet). Möchten Sie mit ihm experimentieren, kommentieren Sie das `Exit Sub` einfach aus.

Wenn Sie auf diese Weise ein Type-Objekt ermittelt haben, können Sie mithilfe seiner Funktionen weitere Informationen über den entsprechenden Typen erhalten.

Klassenanalysefunktionen, die ein Type-Objekt bereitstellt

Die folgende Tabelle gibt Ihnen einen Überblick über die wichtigsten Funktionen und Klassentypen, die von der Type-Klasse bereitgestellt werden, und die dazu dienen, nähere Informationen zum entsprechenden Typ zu erhalten:

Funktion der Type-Klasse	Aufgabe
Assembly	Ruft die Assembly ab, in der der Typ definiert ist. Rückgabetyp: <i>System.Reflection.Assembly</i>
AssemblyQualifiedName	Ruft den voll qualifizierten Assembly-Namen ab. Rückgabetyp: <i>String</i>
Attributes	Ruft eine Bitkombination (<i>Flags-Enum</i>) ab, die Auskunft über alle nicht-benutzerdefinierten Attribute gibt. Rückgabetyp: <i>TypeAttributes</i>
BaseType	Ruft den Typ ab, von dem der angegebene Typ direkt vererbt wurde. Rückgabetyp: <i>Type</i>
FullName	Ruft den voll qualifizierten Namen des angegebenen Typs ab. Rückgabetyp: <i>String</i>
GetCustomAttributes	Ruft ein Array mit allen benutzerdefinierten Attributen ab. Wurden für den Typ keine benutzerdefinierten Attribute definiert, wird <i>Nothing</i> zurückgegeben. Rückgabetyp: <i>Object()</i>
GetEvent	Ruft das <i>EventInfo</i> -Objekt eines bekannten Ereignisses ab. Der Ereignisname wird als String übergeben. Rückgabetyp: <i>EventInfo</i>
GetEvents	Ruft eine Liste (als Array) mit allen Ereignissen des Objektes ab. Rückgabetyp: <i>EventInfo()</i>
GetField	Ruft das <i>FieldInfo</i> -Objekt eines bekannten öffentlichen Feldes ab. Der Feldname wird als String übergeben. Rückgabetyp: <i>FieldInfo</i>
GetFields	Ruft eine Liste (als Array) mit allen öffentlichen Feldern des Objektes ab. Rückgabetyp: <i>FieldInfo()</i>
GetMember	Ruft das <i>MemberInfo</i> -Objekt eines bekannten <i>Members</i> des Objektes an. Ein <i>Member</i> ist eine Oberkategorie eines Objektelementes, wie eine Eigenschaft, eine Methode, ein Ereignis oder ein Feld. Mit der <i>MemberType</i> -Eigenschaft eines <i>MemberInfo</i> -Objektes können Sie feststellen, um was für ein Objektelement es sich handelt. Rückgabetyp: <i>MemberInfo</i>
GetMembers	Ruft eine Liste (als Array) aller Elemente (<i>Member</i>) des Objektes ab. Rückgabetyp: <i>MemberInfo()</i>
GetProperty	Ruft das <i> PropertyInfo</i> -Objekt einer bekannten Eigenschaft ab. Der Eigenschaftenname wird als String übergeben. Rückgabetyp: <i> PropertyInfo</i>
GetProperties	Ruft eine Liste (als Array) aller Eigenschaften des Objektes ab. Rückgabetyp: <i> PropertyInfo()</i>

Tabelle 25.2 Die wichtigsten Funktionen, mit denen Sie Informationen über einen Typ abrufen können

Mit diesen Informationen können wir nun das Beispielprogramm betrachten. Es nutzt im Wesentlichen die *GetMembers*-Funktion des *Type*-Objektes, um die Informationen über ein beliebiges Objekt zu ermitteln:

```
Module md1Main
    Sub Main()
        Dim locAdresse As New Adresse("Gaby", "Halek", "Buchstraße 223", "32154 Autorhausen")
        PrintObjectInfo(locAdresse)
        Console.ReadLine()
    End Sub

    Sub PrintObjectInfo(ByVal [Object] As Object)
        'Den Objekttypen ermitteln, um auf die Objektinhalte zuzugreifen.
        Dim locTypeInstanz As Type = [Object].GetType

        'Die nicht benutzerdefinierten Attribute ausgeben:
        Console.WriteLine("Attribute der Klasse:" + locTypeInstanz.FullName)
```

```

Console.WriteLine("Standardattribute:")
Console.WriteLine("    *" + locTypeInstanz.Attributes.ToString())
Console.WriteLine()
'Member und deren mögliche Attribute ermitteln.
Dim locMembers() As MemberInfo
locMembers = locTypeInstanz.GetMembers()
Console.WriteLine("Member-Liste:")
For Each locMember As MemberInfo In locMembers
    Console.WriteLine("    *" + locMember.Name + ", "
                    + locMember.MemberType.ToString)
    If locMember.GetCustomAttributes(True).Length > 0 Then
        Console.WriteLine("        " + New String("-c", locMember.Name.Length))
    End If

    If locMember.MemberType = MemberTypes.Property Then
        Dim locPropertyInfo As PropertyInfo = CType(locMember, PropertyInfo)
        Console.WriteLine("        Wert: " + locPropertyInfo.GetValue([Object], Nothing).ToString)
    End If
    Next
End Sub
End Module

```

TIPP Wenn Sie alle Elemente eines Typs ermitteln wollen, verwenden Sie die Funktion GetMembers, wie hier im Beispiel gezeigt (erster in Fettschrift gesetzter Block). Sie können anschließend MemberType jedes einzelnen Elementes überprüfen, um herauszufinden, um was für einen Elementtyp es sich genau handelt (Eigenschaft, Methode etc.).

HINWEIS Beachten Sie auch, dass zu jeder Eigenschaft auch Methoden existieren. Framework-intern werden Eigenschaften wie solche behandelt; sie bekommen dann entweder das Präfix set_ oder get_ verpasst, um die Zugriffsart unterscheiden zu können. Aus diesem Grund finden Sie in der Elementliste, die Sie mit GetMembers ermitteln, drei Elemente für jede Eigenschaft (zwei Methoden, eine Eigenschaft – vorausgesetzt natürlich, dass es sich dabei nicht um eine Nur-Lesen- oder um eine Nur-Schreiben-Eigenschaft handelt).

Objekthierarchie von MemberInfo und Casten in den spezifischen Info-Typ

MemberInfo ist die Basisklasse für alle spezifischeren Reflection-XXXInfo-Objekte. Von ihr abgeleitet sind:

- EventInfo zur Speicherung von Informationen über Ereignisse,
- FieldInfo zur Speicherung von Informationen über öffentliche Felder einer Klasse,
- MethodInfo zur Speicherung von Informationen über Methoden einer Klasse,
- PropertyInfo zur Speicherung von Informationen über die Eigenschaften einer Klasse.

Wenn Sie Informationen über eine Klasse oder ein Objekt mit der GetMembers-Funktion ermitteln, dann sind die spezifischeren Info-Objekte in den einzelnen MemberInfo-Objekten des MemberInfo-Arrays geboxt. Sie können sie mit einer CType oder DirectCast-Anweisung in den eigentlichen Infotyp zurückwandeln, um auf spezielle Eigenschaften des Infotyps zugreifen zu können, etwa mit:

```
If locMember.MemberType = MemberTypes.Property Then  
    Dim loc PropertyInfo As PropertyInfo = CType(locMember, PropertyInfo)  
End If
```

Ermitteln von Eigenschaftswerten über PropertyInfo zur Laufzeit

Wenn Sie auf die beschriebene Weise ein PropertyInfo-Objekt ermittelt haben, können Sie auch den eigentlichen Wert der Eigenschaft abrufen. Voraussetzung dafür ist, dass es einen entsprechenden Typ gibt, der zuvor instanziert wurde. Sie verwenden dazu die GetValue-Methode des PropertyInfo-Objektes. Das Beispielprogramm verwendet diese Vorgehensweise, um den aktuellen Inhalt einer Eigenschaft als Text anzuzeigen.

TIPP Da jede Klasse über eine zumindest standardmäßig implementierte ToString-Funktion verfügt, ist die Ermittlung des Wertes als String sicher. Inwieweit ToString ein brauchbares Ergebnis zurückliefert, hängt natürlich von der Implementierung der ToString-Methode des jeweiligen Objektes ab.

Ein Beispiel für die Ermittlung von Eigenschaftsinhalten von Objekten zuvor nicht bekannten Typs finden Sie ebenfalls im Beispielprogramm:

```
If locMember.MemberType = MemberTypes.Property Then  
    Dim loc PropertyInfo As PropertyInfo = CType(locMember, PropertyInfo)  
    Console.WriteLine("    Wert: " + loc PropertyInfo.GetValue([Object], Nothing).ToString)  
End If
```

GetValue erfordert mindestens zwei Parameter: Der erste Parameter bestimmt, von welchem Objekt die angegebene Eigenschaft ermittelt werden soll. Da es Eigenschaften mit Parametern gibt, übergeben Sie im zweiten Parameter ein Object-Array, das diese Parameter enthält. Wenn die Eigenschaft parameterlos verwendet wird, übergeben Sie Nothing als zweiten Parameter, wie im Beispiel zu sehen.

Erstellung benutzerdefinierter Attribute und deren Erkennen zur Laufzeit

Neben der manuellen Serialisierung von Daten ist das Erkennen und Reagieren auf benutzerdefinierte Attribute zur Laufzeit die wohl häufigste Anwendung von Reflection-Techniken. Attribute dienen, wie eingangs erwähnt, zur Kennzeichnung von Klassen oder Klassenelementen; Attributklassen erfüllen in der Regel aber keine weitere wirkliche Funktionalität, da nur in seltenen Fällen ihr Klassencode ausgeführt wird.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2008 Entwicklerbuch\0 - Datenstrukturen\Kapitel 25\Reflection02
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Innerhalb der Codedatei *mdlMain.vb* finden Sie die Definition einer Attribute-Klasse, die folgendermaßen ausschaut.

```
'Benutzerdefiniertes Attribut erstellen.
<AttributeUsage(AttributeTargets.All)> Public Class MeinAttribute
    Inherits Attribute
    Private myName As String

    Public Sub New(ByVal name As String)
        myName = name
    End Sub 'New

    Public ReadOnly Property Name() As String
        Get
            Return myName
        End Get
    End Property
End Class
```

Zwei Sachen fallen hier auf: Zum einen ist der Klassename nicht wirklich deutsch (Attribut wird im Deutschen nicht mit »e« am Ende geschrieben). Achten Sie aber darauf, wenn Sie selber Attribute-Klassen entwerfen, dass die Klassen grundsätzlich auf »... Attribute« enden. Der Entwickler, der das Attribut anschließend verwendet, kann in diesem Fall den verkürzten Namen verwenden. Anstelle von `MeinAttribute` reichte also die Angabe von `Mein`.

Die zweite Auffälligkeit: Eine Attribute-Klasse kann für den Gebrauch von nur bestimmten Elementen einer Klasse reglementiert werden, und dazu dient `AttributeUsageAttribute`. Sie bestimmen, wie im oben gezeigten Beispielcode, mit der `AttributeTargets`-Enum, für welche Elemente einer Klasse Ihre Attribute-Klasse zutreffend ist.

Wichtig ist auch, dass Sie eine benutzerdefinierte Attribute-Klasse mit `Inherits` von der Attribute-Basisklasse ableiten, damit die Reflection-Funktionen sie später überhaupt als Attribute ausmachen können.

Für das erweiterte Beispielprogramm kommt ebenfalls wieder die bekannte Adresse-Klasse zum Einsatz; allerdings verfügt sie an einigen Stellen über Attribute-Kennzeichnungen, und wir haben die neue `MeinAttribute`-Klasse dafür verwendet. Wie Sie im oben gezeigten Codeausschnitt sehen können, übernimmt die `MeinAttribute`-Klasse einen Parameter, der in diesem Beispiel nur informative Zwecke hat. In der Adresse-Klasse nutzen wir den String-Parameter, um darüber zu informieren, in welchem Kontext wir das Attribut eingesetzt haben. Die mit dem benutzerdefinierten Attribut versehene Adresse-Klasse sieht folgendermaßen aus (die Attribute-Verwendungen sind fett markiert).

HINWEIS Sie sehen anhand dieses Beispiels, dass der Namenszusatz `Attribute` weggelassen werden kann, wenn er korrekt buchstabiert wurde.

```
<Serializable(), Mein("Über der Klasse")> _
Public Class Adresse

    Private myName As String
    Private myVorname As String
    Private myStraße As String
    Private myPLZOrt As String
    Private myErfasstAm As DateTime
    Private myErfasstVon As String
    Private myBefreundetMit As ArrayList
```

```

<Mein("Über dem Konstruktor")> _
Sub New(ByVal Vorname As String, ByVal Name As String, ByVal Straße As String, ByVal PLZOrt As
String)
    'Konstruktor legt alle Member-Daten an.
    myName = Name
    myVorname = Vorname
    myStraße = Straße
    myPLZOrt = PLZOrt
    myErfasstAm = DateTime.Now
    myErfasstVon = Environment.UserName
    myBefreundetMit = New ArrayList
End Sub

<Mein("Über einer Eigenschaft")> _
Public Property Name() As String
    Get
        Return myName
    End Get
    Set(ByVal Value As String)
        myName = Value
    End Set
End Property

#Region "Die anderen Eigenschaften"
    'Aus Platzgründen hier nicht gezeigt.
#End Region

<Mein("Über einer Methode")>
Public Overrides Function ToString() As String
    Dim locTemp As String
    locTemp = Name + ", " + Vorname + ", " + Straße + ", " + PLZOrt + vbNewLine
    locTemp += "--- Befreundet mit: ---" + vbNewLine
    For Each locAdr As Adresse In BefreundetMit
        locTemp += "    * " + locAdr.ToStringShort() + vbNewLine
    Next
    locTemp += vbNewLine
    Return locTemp
End Function

Public Function ToStringShort() As String
    Return Name + ", " + Vorname
End Function
End Class

```

Ziel von Reflection ist es nun, die Attribute zur Laufzeit zu erkennen. Schauen wir uns zunächst das Ergebnis vorweg an, damit das eigentliche Auswertungsprogramm, das die Attribute findet, anschließend leichter zu verstehen ist.

Wenn Sie das Programm starten, produziert es die folgende Bildschirmausgabe:

```

Attribute der Klasse:Reflection02.Adresse
Standardattribute:
    *AutoLayout, AnsiClass, NotPublic, Public, Serializable

```

```
Benutzerattribute:  
  * Reflection02.MeinAttribute  
  
Member-Liste:  
  *GetHashCode, Method  
  *Equals, Method  
  *ToString, Method  
  -----  
  * Reflection02.MeinAttribute  
    Name: Über einer Methode  
    TypeId: Reflection02.MeinAttribute  
  *get_Name, Method  
  *set_Name, Method  
  *get_ErfasstAm, Method  
  *get_ErfasstVon, Method  
  *get_BefreundetMit, Method  
  *set_BefreundetMit, Method  
  *get_Vorname, Method  
  *set_Vorname, Method  
  *get_Straße, Method  
  *set_Straße, Method  
  *get_PLZOrt, Method  
  *set_PLZOrt, Method  
  *ToStringShort, Method  
  *GetType, Method  
  *.ctor, Constructor  
  -----  
  * Reflection01.MeinAttribute  
    Name: Über dem Konstruktor  
    TypeId: Reflection02.MeinAttribute  
*Name, Property  
  -----  
  * Reflection02.MeinAttribute  
    Name: Über einer Eigenschaft  
    TypeId: Reflection02.MeinAttribute  
Wert: Halek  
*ErfasstAm, Property  
  Wert: 12.04.2004 11:43:58  
*ErfasstVon, Property  
  Wert: Administrator  
*BefreundetMit, Property  
  Wert: System.Collections.ArrayList  
*Vorname, Property  
  Wert: Gaby  
*Straße, Property  
  Wert: Buchstraße 223  
*PLZOrt, Property  
  Wert: 32154 Autorhausen
```

Die Passagen, bei denen sowohl das Attribut selbst als auch sein jeweils gültiger Parameter erkannt wurden, sind im Bildschirmauszug fett markiert.

Ermitteln von benutzerdefinierten Attributen zur Laufzeit

Um benutzerdefinierte Attribute zu ermitteln, gibt es die Funktion `GetCustomAttributes`. Diese Funktion ist sowohl auf einen Typ (die Klasse oder Struktur selbst) als auch auf einzelne Member anwendbar. Als Parameter übernimmt sie entweder einen booleschen Wert, der bestimmt, ob in der Hierarchieliste des Objektes vorhandene Typen ebenfalls auf Attribute untersucht werden sollen (`True`), oder zum einen den Typ des Attributes, nach dem gezielt gesucht werden soll, und zum anderen den booleschen Wert für das Durchsuchen der Hierarchieliste zusätzlich.

Da die Parameter einer Attributklasse in der Regel als Eigenschaften abrufbar sind, können Sie, nachdem Sie das Attribut ermittelt haben, mit `GetType` seine Typ-Instanz abrufen, anschließend seine Eigenschaften mit `GetProperties` auflisten und schließlich mit `GetValue` den eigentlichen Wert einer Attributeigenschaft auslesen.

Das modifizierte Beispielprogramm zeigt, wie es geht. Es liest am Anfang sowohl ein mögliches, benutzerdefiniertes Attribut aus, das für die gesamte Klasse gilt, und untersucht anschließend jeden einzelnen Klassen-Member auf Attribute:

```
Module mdlMain

Sub Main()
    Dim locAdresse As New Adresse("Gaby", "Hälek", "Buchstraße 223", "32154 Autorhausen")
    PrintObjectInfo(locAdresse)
    Console.ReadLine()
End Sub

Sub PrintObjectInfo(ByVal [Object] As Object)
    'Den Objekttypen ermitteln, um auf die Objektinhalte zuzugreifen
    Dim locTypeInstanz As Type = [Object].GetType

    'Die nicht Benutzerdefinierten Attribute anzeigen:
    Console.WriteLine("Attribute der Klasse:" + locTypeInstanz.FullName)
    Console.WriteLine("Standardattribute:")
    Console.WriteLine("    *" + locTypeInstanz.Attributes.ToString())
    Console.WriteLine()

    'Benutzerdefinierte Attribute der Klasse ermitteln.
    '(Es können auf diese Weise *nur* benutzerdefinierte Attribute ermittelt werden)
    Console.WriteLine("Benutzerattribute:")
    For Each locAttribute As Attribute In locTypeInstanz.GetCustomAttributes(True)
        Console.WriteLine("    * " + locAttribute.ToString())
    Next
    Console.WriteLine()

    'Member und deren mögliche Attribute ermitteln.
    Dim locMembers() As MemberInfo
    locMembers = locTypeInstanz.GetMembers()
    Console.WriteLine("Member-Liste:")
    For Each locMember As MemberInfo In locMembers
        Console.WriteLine("    *" + locMember.Name + ", "
                        + locMember.MemberType.ToString())
        If locMember.GetCustomAttributes(True).Length > 0 Then
            Console.WriteLine("        " + New String("-"c, locMember.Name.Length))
            For Each locAttribute As Attribute In locMember.GetCustomAttributes(False)
                Console.WriteLine("            * " + locAttribute.ToString())
            Next
        End If
    Next
End Sub
```

```
For Each loc PropertyInfo As PropertyInfo In locAttribute.GetType.GetProperties
    Console.Write("      " + loc PropertyInfo.Name)
    Console.WriteLine(": " + loc PropertyInfo.GetValue(locAttribute,
        Nothing).ToString)
Next
Next
End If

If locMember.MemberType = MemberTypes.Property Then
    Dim loc PropertyInfo As PropertyInfo = CType(locMember, PropertyInfo)
    Console.WriteLine("      Wert: " + loc PropertyInfo.GetValue([Object], Nothing).ToString)
End If
Next
End Sub
End Module
```

HINWEIS Bitte beachten Sie, dass Sie mit GetCustomAttributes ausschließlich benutzerdefinierte Attribute auslesen können. Möchten Sie wissen, ob eine Klasse beispielsweise serialisierbar ist, können Sie entweder mit einer der IsXxx-Funktionen ihres Type-Objektes (IsSerializable beispielsweise) oder mit der Attributes-Eigenschaft an diese Informationen gelangen.

Kapitel 26

Kultursensible Klassen entwickeln

In diesem Kapitel:

Allgemeines über Format Provider in .NET	744
Kulturabhängige Formatierungen mit CultureInfo	745
Formatierung durch Formatzeichenfolgen	748
Gezielte Formatierungen mit Format Providern	758
Kombinierte Formatierungen	760
So helfen Ihnen benutzerdefinierte Format Provider, Ihre Programme zu internationalisieren	763

Beim Durcharbeiten der vergangenen Kapitel sind Sie bereits auf die eine oder andere Funktionalität von .NET gestoßen, Zahlen oder Zeitwerte aufzubereiten, um sie in bestimmten Darstellungsformaten auszugeben. Auch die Vorgehensweise zur Vorgabe von Zeichenmustern für das Parsen von Zeichenketten, um sie in entsprechende Datentypen umzuwandeln, konnten Sie schon in einigen Beispielen kennen lernen. Doch glauben Sie mir: Sie haben bislang nur die Spitze des Eisberges gesehen.

Die folgenden Abschnitte beschäftigen sich intensiver mit der textlichen Aufbereitung von Daten, und insbesondere der Umgang mit den verfügbaren Format Providern bis hin zur Realisierung eigener Format Provider wird für Sie sicherlich von großem Interesse sein.

Allgemeines über Format Provider in .NET

Wenn Sie einen Zahlenwert in eine Zeichenkette umwandeln möchten, dann machen Sie das in der Regel mit der `ToString`-Funktion, da sie Ihnen die flexibelste Steuerung der Umwandlung bietet. Für numerische Werte stehen Ihnen folgende Möglichkeiten zur Verfügung:

- `DoubleVariable.ToString()`: Der Wert der Zeichenkette wird mit den aktuellen Kultureinstellungen ausgegeben, die von Ihren Betriebssystemeinstellungen abhängig sind.
- `DoubleVariable.ToString("formatzeichenfolge")`: Die Formatzeichenfolge bestimmt, wie der Zahlenwert in eine Zeichenkette umgewandelt werden soll.
- `DoubleVariable.ToString(IFormatProvider)`: Ein so genannter Format Provider (etwa: Formatierungsanbieter) bestimmt, wie die Zeichenfolge formatiert werden soll.
- `DoubleVariable.ToString("formatzeichenfolge", IFormatProvider)`: Eine Formatzeichenfolge bestimmt, wie der Zahlenwert in eine Zeichenkette umgewandelt werden soll; der zusätzlich angegebene Format Provider regelt nähere Konventionen.

Das Gleiche gilt für die Konvertierung anderer primitiver Datentypen, beispielsweise bei der Ausgabe von Datums- bzw. Zeitwerten, wenn also der Datentyp `Date` in eine Zeichenkette umgewandelt werden soll.

Von der ersten Möglichkeit haben Sie sicherlich schon oft Gebrauch gemacht. Wichtig zu wissen: Die kulturabhängigen Besonderheiten werden bei der Umwandlung berücksichtigt. Wenn Sie auf einem deutschen .NET-System den Code

```
Module FormatDemos  
  
Sub Main()  
  
    Dim locDouble As Double = 1234.56789  
    Dim locDate As Date = #12/24/2008 10:33:00 PM#  
  
    Console.WriteLine(locDouble.ToString())  
    Console.WriteLine(locDate.ToString())  
    Console.ReadLine()  
  
End Sub  
  
End Module
```

ablaufen lassen, erhalten Sie die folgende Ausgabe:

```
1234,56789  
24.12.2008 22:33:00
```

Wenn Sie das Programm kompilieren und auf einem englischen oder amerikanischen Computer¹ laufen lassen (und das, ohne es neu zu kompilieren), sieht die Ausgabe schon ganz anders aus:

```
1234.56789  
12/24/2008 10:33:00 PM
```

Der Hintergrund: Auch wenn Sie keinen zusätzlichen Parameter bei der Verwendung von `Tostring` angegeben haben, muss es irgendein Regelwerk in .NET geben, das bestimmt, wann ein Dezimalkomma wirklich ein Komma und wann ein Punkt ist, so wie auf englischen oder amerikanischen Systemen. Diese Bestimmungen werden von den Format Providern geregelt. Sie können auch auf einem deutschen System so tun, als würden Sie Englisch formatieren. Standardmäßig werden einfach die Ländereinstellungen aus der Systemsteuerung des Rechners benutzt, auf dem das Programm läuft.

Kulturabhängige Formatierungen mit CultureInfo

Um länderspezifische Formatierungen bei der Umwandlung vorzunehmen, verwenden Sie einen Format Provider namens `CultureInfo`. Auf diese Klasse können Sie zugreifen, wenn Sie den Namensbereich `System.Globalization` in Ihr Programm eingebunden haben. Verändern Sie das vorherige Programm wie folgt (Änderungen sind fett hervorgehoben):

```
Module FormatDemosCultureInfo  
  
Sub Main()  
  
    Dim locDate As Date = #12/24/2008 10:33:00 PM#  
    Dim locDouble As Double = 1234.56789  
    Dim locCultureInfo As New CultureInfo("en-US")  
  
    Console.WriteLine(locDouble.ToString(locCultureInfo))  
    Console.WriteLine(locDate.ToString(locCultureInfo))  
    Console.ReadLine()  
  
End Sub  
  
End Module
```

Wenn Sie dieses Programm starten, werden Sie sehen, dass die Ausgabe exakt dem entspricht, was auf dem US-Computer bei der Verwendung des vorherigen Programms ausgegeben wurde.

¹ Es gilt das zuvor Gesagte: Die Ländereinstellungen des Betriebssystems sind dafür verantwortlich.

Wenn Sie keinen Format Provider bei der `ToString`-Methode eines primitiven Datentyps angeben, verwendet .NET automatisch den, der der eingestellten Kultur entspricht. Sie können den Format Provider der aktuellen Kultureinstellung mit der statischen Funktion `CurrentCulture` der `CultureInfo`-Klasse ermitteln. Wenn Sie die beiden Zeilen

```
Console.WriteLine(locDouble.ToString(locCultureInfo))
Console.WriteLine(locDate.ToString(locCultureInfo))
```

gegen die Zeilen

```
Console.WriteLine(locDouble.ToString(CultureInfo.CurrentCulture))
Console.WriteLine(locDate.ToString(CultureInfo.CurrentCulture))
```

austauschen und wieder einmal auf einem deutschen und einmal auf einem amerikanischen System laufen lassen, bekommen Sie exakt das Ergebnis, das Sie auch beim Beispiel beobachten konnten, bei dem `ToString` gar kein Parameter übergeben wurde.

Wie Sie im vorherigen Beispiel sehen konnten, erstellen Sie eine neue `CultureInfo`-Klasseninstanz, indem Sie eine Buchstabenkombination angeben, die die entsprechende Kultur bezeichnet. Die folgende Tabelle zeigt Ihnen die wichtigsten Einstellungen. Eine vollständige Tabelle können Sie aus der Visual Studio-Online-Hilfe erhalten.

Kulturkürzel	Kulturcode	Sprache-Land/Region
"" (leere Zeichenfolge)	0x007F	Invariante Kultur
nl	0x0013	Niederländisch
nl-BE	0x0813	Niederländisch – Belgien
nl-NL	0x0413	Niederländisch – Niederlande
en	0x0009	Englisch
en-AU	0x0C09	Englisch – Australien
en-CB	0x2409	Englisch – Karibik
en-IE	0x1809	Englisch – Irland
en-GB	0x0809	Englisch – Großbritannien
en-US	0x0409	Englisch – USA
fr	0x000C	Französisch
fr-BE	0x080C	Französisch – Belgien
fr-CA	0x0C0C	Französisch – Kanada
fr-FR	0x040C	Französisch – Frankreich
fr-LU	0x140C	Französisch – Luxemburg
fr-MC	0x180C	Französisch – Monaco
fr-CH	0x100C	Französisch – Schweiz

Kulturkürzel	Kulturcode	Sprache-Land/Region
de	0x0007	Deutsch
de-AT	0x0C07	Deutsch – Österreich
de-DE	0x0407	Deutsch – Deutschland
de-LI	0x1407	Deutsch – Liechtenstein
de-LU	0x1007	Deutsch – Luxemburg
de-CH	0x0807	Deutsch – Schweiz
it	0x0010	Italienisch
it-IT	0x0410	Italienisch – Italien
it-CH	0x0810	Italienisch – Schweiz
es	0x000A	Spanisch
es-AR	0x2C0A	Spanisch – Argentinien
es-PR	0x500A	Spanisch – Puerto Rico
es-ES	0x0C0A	Spanisch – Spanien
es-VE	0x200A	Spanisch – Venezuela

Tabelle 26.1 Diese Einstellungen verwenden Sie für CultureInfo-Objekte

Vermeiden von kulturabhängigen Programmfehlern

Auf Programmentwicklungen hat das unter Umständen Auswirkungen, nämlich dann, wenn Sie numerische Daten oder Zeitdaten in Textdateien speichern und diese kulturübergreifend ausgetauscht werden müssen. Und: Das Definieren von konstanten Werten sollte im Programm nur direkt mit Literalen und nie durch Zeichenketten und entsprechende Konvertierungsfunktionen erfolgen. Dazu ein Beispiel:

Die Codezeilen

```
Dim locDoubleTest As Double = CDbl("1.234")
Console.WriteLine(locDoubleTest)
```

ergeben auf einem deutschen System den Wert 1234; auf einem englischsprachigen System jedoch den Wert 1,234 (aber 1.234 angezeigt), da hier der Punkt nicht als Tausendergruppierung, sondern eben als Dezimalkomma (*Decimal Point*) angesehen wird. Mehr Überlegungen zu diesem Thema finden Sie auch im vorherigen Kapitel. Das dort Gesagte gilt gleichermaßen auch für Datumswerte. Die Umwandlung aus einer Zeichenfolge wirkt dabei noch konstruiert, aber denken Sie daran, wie oft Eingaben aus einer TextBox mit TextBox.Text (eine Methode, die einen String zurückgibt) geparsst werden müssen, auch und gerade für numerische Werte.

Wenn Sie Formatierungen für das Serialisieren (Speichern) vornehmen müssen, verwenden Sie dazu am besten ein so genanntes *invariantes CultureInfo-Objekt*, das Sie mit folgender Anweisung ermitteln können:

```
Dim locCultureInfo As CultureInfo = CultureInfo.InvariantCulture
```

Wenn Sie alle Konvertierungen innerhalb Ihres Programms (also sowohl das Parsen eines Strings, um ihn in einen entsprechenden Datentyp umzuwandeln, als auch das Konvertieren eines Datentyps in einen String) damit durchführen, sind Sie auf der sicheren Seite.

Formatierung durch Formatzeichenfolgen

Sowohl die `ToString`-Methode als auch die `Parse`- bzw. die `ParseExact`-Methode unterstützen neben dem Einsatz von Format Providern auch die Formatierung durch direkte Mustervorlagen, die durch Strings oder String-Arrays übergeben werden. Sie haben schon an einigen Beispielen gesehen, dass bestimmte Buchstaben, zu Gruppen zusammengeführt, bestimmte Formatierungen eines Datenblocks (Tage, Monate oder Jahre beispielsweise) bewirken. Die folgenden Abschnitte demonstrieren den Einsatz von Formatzeichenfolgen.

Formatierung von numerischen Ausdrücken durch Formatzeichenfolgen

BEGLEITDATEIEN Die Begleitdateien für die Beispiele, die in diesem Abschnitt behandelt werden, finden Sie im Verzeichnis:

```
...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 26\\FormatProvider - Num
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie die Codezeilen

```
Dim locDouble As Double  
locDouble = Math.PI + 10000 * Math.PI  
Console.WriteLine("locDouble hat den Wert:" + locDouble.ToString())
```

ausführen, erhalten Sie die Ausgabe

```
locDouble hat den Wert: 31419,0681285515
```

In vielen Fällen ist es aber erwünscht, beispielsweise den Tausendergruppierungspunkt ebenfalls anzeigen zu lassen oder nur eine bestimmte Anzahl von Nachkommastellen auszugeben. Dann können Sie die Ausgabe durch Formatzeichenfolgen reglementieren.

Wenn Sie beispielsweise die Zeile

```
Console.WriteLine("locDouble hat den Wert:" + locDouble.ToString())
```

des obigen Beispiels durch die folgende ersetzen,

```
Console.WriteLine("locDouble hat den Wert: " + locDouble.ToString("#,###.00"))
```

sieht die Ausgabe schon sehr viel ordentlicher aus:

```
locDouble hat den Wert: 31.419,07
```

In diesem Fall ist der übergebene Parameter als Regelvorgabe für die Formatierung übergeben worden. Jedes »#«-Zeichen stellt dabei eine mögliche Ziffer dar, jede 0 eine Muss-Ziffer. Das bedeutet: Die Zahl 304 würde keine zusätzliche führende 0 bei der Ausgabe bekommen, da die Formatzeichenfolge zwar vier Zeichen (»#,###«) vorgibt, durch die Verwendung des »#«-Zeichens aber keine zwingende Verwendung aller Ziffern vorgeschrieben wird.

Anders sieht es bei dem gleichen Wert und seinen Nachkommastellen nach der Formatierung aus: Da hier »0« und nicht »#« angegeben wird, lauten die Nachkommastellen nach der Formatierung ».00«, obwohl zur Wertdarstellung eigentlich keine Nachkommastellen nötig wären.

Die folgende Tabelle zeigt, welche Formatzeichen Sie bei der Zusammenstellung von Formatzeichenfolgen für numerische Werte verwenden können:

Formatzeichen	Name	Beschreibung
0	0-Platzhalter	<p>Gibt es an der Stelle des 0-Platzhalters eine korrelierende Ziffer im aufzubereitenden Wert, dann wird die Ziffer des Wertes an der Stelle platziert, die der 0-Platzhalter durch seine Position vorgibt.</p> <p>Wenn der zu formatierende Wert die vorgegebene Ziffernposition nicht hergibt (bei »000,00« als Vorgabe gäbe es für den Wert 34,1 nicht alle vorgegebenen Positionen), wird eine 0 angezeigt.</p> <p>Die Positionen der »0«, die am weitesten links vor dem Dezimaltrennzeichen steht, und der »0«, die am weitesten rechts hinter dem Dezimaltrennzeichen steht, bestimmen also den Bereich der Ziffern, die immer in der Ergebniszeichenfolge enthalten sind.</p> <p>Falls im zu formatierenden Wert mehr Nachkommastellen vorhanden sind, als Nullen Positionen vorgeben, wird auf die entsprechende Anzahl auf Stellen gerundet.</p>
#	Ziffernplatzhalter	<p>Verfügt der zu formatierende Wert über eine Ziffer an der Stelle, an der »#« in der Formatzeichenfolge steht, wird diese Ziffer in der Ergebniszeichenfolge angezeigt, anderenfalls nicht.</p> <p>Beachten Sie, dass dieses Formatzeichen nie die Anzeige von 0 bewirkt, wenn es sich nicht um eine signifikante Ziffer handelt, selbst wenn 0 die einzige Ziffer in der Zeichenfolge ist. 0 wird jedoch angezeigt, wenn es sich um eine signifikante Ziffer in der angezeigten Zahl handelt (bei 0304,030 beispielsweise sind die äußeren Nullen nicht signifikant, d.h. Weglassen verändert den Wert nicht).</p>
.	Dezimaltrennzeichen	<p>Das erste ».«-Zeichen in der Formatzeichenfolge bestimmt die Position des Dezimaltrennzeichens im formatierten Wert. Weitere ».«-Zeichen werden ignoriert.</p> <p>Das für das Dezimaltrennzeichen verwendete Zeichen wird durch die <i>NumberDecimalSeparator</i>-Eigenschaft der <i>NumberFormatInfo</i> bestimmt, die die Formatierung steuert, allerdings nur für die Ausgabe, nie für die Vorgabe! WICHTIG: Verwechseln Sie ».,« und »,.« nicht bei der Erstellung der Formatzeichenfolge. Die amerikanisch/englische Formatvorgabe ist hier maßgeblich – beispielsweise »1,000,000.23« für <i>Einemillionkommazweidrei</i>. ►</p>

Formatzeichen	Name	Beschreibung
,	Tausendertrennzeichen und Zahlenskalierung	Das »,«-Zeichen erfüllt zwei Aufgaben. Erstens: Wenn die Formatzeichenfolge ein »,«-Zeichen zwischen zwei Ziffernplatzhaltern enthält (»0« oder »#«) und einer der Platzhalter links neben dem Dezimaltrennzeichen steht, werden in der Ausgabe Tausendertrennzeichen zwischen jeder Gruppe von drei Ziffern links neben dem Dezimaltrennzeichen eingefügt. Das in der Ergebniszeichenfolge als Dezimaltrennzeichen verwendete Zeichen wird durch die <i>NumberGroupSeparator</i> -Eigenschaft der aktuellen <i>NumberFormatInfo</i> bestimmt, die die Formatierung steuert. Zweitens: Wenn die Formatzeichenfolge ein oder mehr »,«-Zeichen direkt links neben dem Dezimaltrennzeichen enthält, wird die Zahl durch die Anzahl der »,«-Zeichen multipliziert mit 1000 dividiert, bevor sie formatiert wird. Die Formatzeichenfolge »0,« stellt 100 Millionen z.B. als 100 dar. Verwenden Sie das »,«-Zeichen, um anzugeben, dass die Skalierung keine Tausendertrennzeichen in der formatierten Zahl enthält. Um also eine Zahl um 1 Million zu skalieren und Tausendertrennzeichen einzufügen, verwenden Sie die Formatzeichenfolge »#,##0,«. WICHTIG: Verwechseln Sie »,« und »,« nicht bei der Erstellung der Formatzeichenfolge. Die amerikanisch/englische Formatvorgabe ist hier maßgeblich – beispielsweise »1,000,000.23« für <i>Einemillionkommazweidrei</i> .
%	Prozentplatzhalter	Enthält eine Formatzeichenfolge ein »%«-Zeichen, wird die Zahl vor dem Formatieren mit 100 multipliziert. Das entsprechende Symbol wird in der Zahl an der Stelle eingefügt, an der »%« in der Formatzeichenfolge steht. Das verwendete Prozentzeichen ist von der aktuellen <i>NumberFormatInfo</i> -Klasse abhängig.
E0 E+0 E-0 e0 e+0 e-0	Wissenschaftliche Notation	Enthält die Formatzeichenfolge die Zeichenfolgen »E«, »E+«, »E-«, »e«, »e+« oder »e-« und folgt direkt danach mindestens ein »0«-Zeichen, wird die Zahl mit der wissenschaftlichen Notation formatiert und ein »E« bzw. »e« zwischen der Zahl und dem Exponenten eingefügt. Die Anzahl der »0«-Zeichen nach dem entsprechenden Formatzeichen für die wissenschaftliche Notation bestimmt die Mindestanzahl von Ziffern, die für den Exponenten ausgegeben werden. Das »E+«-Format und das »e+«-Format geben an, dass immer ein Vorzeichen (Plus oder Minus) vor dem Exponenten steht. Die Formate »E«, »E-«, »e« oder »e-« geben an, dass nur vor negativen Exponenten ein Vorzeichen steht.
'ABC' "ABC"	Zeichenfolgenliteral	Zeichen, die in einfachen bzw. doppelten Anführungszeichen stehen, werden direkt in die Ergebniszeichenfolge kopiert, ohne die Formatierung zu beeinflussen.
;	Abschnittstrennzeichen	Mit dem »;«-Zeichen werden Abschnitte für positive und negative Zahlen sowie Nullen in der Formatzeichenfolge voneinander getrennt. Lesen Sie dazu auch die Anmerkung am Ende der Tabelle.
Sonstige	Alle anderen Zeichen	Alle anderen Zeichen werden als Literale an der angegebenen Position in die Ergebniszeichenfolge kopiert.

Tabelle 26.2 Für numerische Formatzeichenfolgen verwenden Sie diese Formatzeichen

HINWEIS Bitte beachten Sie, dass Sie für negative und positive Werte sowie für den Wert 0 jeweils eine individuelle Zeichenfolge bestimmen können, die durch das Semikolon getrennt werden. Das folgende Beispiel verdeutlicht ihre Anwendung:

```
Dim locDouble As Double
Dim locFormat As String = "#,###.00;-#,###.0000;+-0.00000"
locDouble = Math.PI + 10000 * Math.PI
Console.WriteLine("locDouble hat den Wert: " + locDouble.ToString(locFormat))
```

```
locDouble *= -1
Console.WriteLine("locDouble hat den Wert: " + locDouble.ToString(locFormat))
locDouble = 0
Console.WriteLine("locDouble hat den Wert: " + locDouble.ToString(locFormat))
```

Dieser Codeausschnitt würde das folgende Ergebnis auf dem Bildschirm produzieren:

```
locDouble hat den Wert: 31.419,07
locDouble hat den Wert: -31.419,0681
locDouble hat den Wert: +-0,00000
```

Formatierung von numerischen Ausdrücken durch vereinfachte Formatzeichenfolgen

Für den einfachen Einsatz gibt es in .NET einige vereinfachte Formatzeichenfolgen, die in der Regel nur aus einem einzigen Zeichen bestehen und einen Typ von Formatierung bezeichnen. Anstatt z. B. eine Formatzeichenfolge zu erstellen, die eine Währungsformatierung vorgibt, können Sie einfach das »C«-Zeichen als Formatzeichenfolge verwenden. In Kombination mit dem entsprechenden *CultureInfo*-Format-Provider brauchen Sie sich obendrein noch nicht einmal um das Finden des entsprechenden Währungssymbols zu kümmern:

```
Dim locDouble As Double
locDouble = 12234.346
Console.WriteLine("Sie bekommen {0} aus einem Lottogewinn.", locDouble.ToString("#,###.00"))
Console.WriteLine("Sie bekommen {0} aus einem Lottogewinn.", locDouble.ToString("C", New
CultureInfo("en-US")))
```

Beide `Console.WriteLine`-Anweisungen zeigen in diesem Beispiel das exakt gleiche Ergebnis im Konsolenfenster an, nämlich:²

```
Sie bekommen $12.234,35 aus einem Lottogewinn.
Sie bekommen $12,234.35 aus einem Lottogewinn.
```

Die folgende Tabelle zeigt Ihnen, welche Kurzformen für Formatzeichenfolgen .NET für die Formatierung von numerischen Werten kennt. Bitte beachten Sie, dass, wie im Beispiel gezeigt, die Resultate unter Umständen kulturabhängig sind und sich deswegen mit einem *CultureInfo*-Objekt entsprechend steuern lassen.

Formatzeichen	Beschreibung
c, C	Währungsformat
d, D	Dezimales Format
e, E	Wissenschaftliches Format (Exponentialformat)

² Ich habe bei Währungsangaben bewusst auf die Euro-Währung verzichtet, da Konsolenanwendungen das Euro-Zeichen nicht ohne weiteres darstellen können.

Formatzeichen	Beschreibung
f, F	Festkommaformat
g, G	Allgemeines Format
n, N	Zahlenformat
r, R	Schleifenformat, das sicherstellt, dass in Zeichenfolgen konvertierte Zahlen denselben Wert haben, wenn sie wieder in Zahlen konvertiert werden.
x, X	Hexadezimales Format

Tabelle 26.3 Für vereinfachte numerische Formatzeichenfolgen verwenden Sie diese Formatzeichen

Formatierung von Datums- und Zeitausdrücken durch Formatzeichenfolgen

BEGLEITDATEIEN

Die Begleitdateien zu folgenden Beispielen finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 26\\FormatProvider - Date

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Was für numerische Datentypen gilt, ist auch auf den Datumstyp *Date* anwendbar: Es gibt eine umfangreiche Unterstützung durch bestimmte Formatzeichenfolgen, die bestimmen, wie ein Datumswert in eine Zeichenkette umgewandelt werden kann.

Wenn Sie die Codezeilen

```
Dim locDate As Date
locDate = #12/24/2008 10:33:00 PM#
Console.WriteLine("locDate hat den Wert: " + locDate.ToString())
```

ausführen, erhalten Sie die Ausgabe

```
locDate hat den Wert: 24.12.2008 22:33:00
```

In den meisten Fällen benötigen Sie aber keine kombinierte Ausgabe des Zeitwertes von Uhrzeit und Datum, sondern nur eine von beiden. Außerdem verlangen viele Anwendungen, das Datum etwas ausführlicher darzustellen – beispielsweise indem die ersten Buchstaben des Monats dargestellt werden oder der Wochentag des Datums mit ausgegeben wird.

Wenn Sie beispielsweise die Zeile

```
Console.WriteLine("locDate hat den Wert: " + locDate.ToString())
```

des obigen Beispiels durch die folgende ersetzen,

```
Console.WriteLine("locDate hat den Wert: " + locDate.ToString("ddd, dd. MMM yyyy - HH:mm"))
```

erkennen Sie, dass Sie durch den Einsatz einer Datums-Formatzeichenfolge sehr viel mehr Einfluss auf die Art der Umwandlung des Datumswertes in eine Zeichenkette genommen haben, denn die Ausgabe sieht nun folgendermaßen aus:

```
locDate hat den Wert: Mittwoch, 24. Dez 2008 - 22:33
```

In diesem Fall haben Sie der `ToString`-Methode eine Zeichenkette mit Formattanweisungen für die Formatierung übergeben. Jedes der verwendeten Zeichen der Formatzeichenfolge spiegelt dabei eine Datengruppe innerhalb eines `Date`-Datentyps wider.

Die folgende Tabelle zeigt, welche Formatzeichen Sie bei der Zusammenstellung von Formatzeichenfolgen für numerische Werte verwenden können.

WICHTIG Einige Formatzeichen der folgenden Tabelle können auch den Kurzformatzeichen der übernächsten Tabelle entsprechen. Achten Sie deswegen darauf, dass Sie die folgenden Zeichen im beschriebenen Kontext nicht alleine verwenden. Und: Je nach zusätzlich verwendetem `CultureInfo` produzieren die verwendeten Formatzeichenkombinationen unterschiedliche Resultate.

Formatzeichen	Beschreibung
d	Zeigt den aktuellen Tag des Monats als Zahl zwischen 1 und 31 an. Wenn die Nummer des Tages nur einstellig ist, wird diese nicht mit einer führenden Null versehen.
dd	Zeigt den aktuellen Tag des Monats als Zahl zwischen 1 und 31 an. Wenn die Nummer des Tages nur einstellig ist, wird ihr eine 0 vorangestellt.
ddd	Zeigt den abgekürzten Namen des Tages für den angegebenen <i>Date</i> -Wert an.
dddd	Zeigt den vollständigen Namen des Tages für den angegebenen <i>Date</i> -Wert an.
f	Zeigt die Bruchteile von Sekunden als eine Ziffer an.
ff	Zeigt die Bruchteile von Sekunden als zwei Ziffern an.
fff	Zeigt die Bruchteile von Sekunden als drei Ziffern an.
ffff	Zeigt die Bruchteile von Sekunden als vier Ziffern an.
fffff	Zeigt die Bruchteile von Sekunden als fünf Ziffern an.
fffffff	Zeigt die Bruchteile von Sekunden als sechs Ziffern an.
fffffff	Zeigt die Bruchteile von Sekunden als sieben Ziffern an.
g oder gg	Zeigt den Zeitraum für den angegebenen <i>Date</i> -Wert an. Wenn Sie mit einer deutschen Kultureinstellung (durch <code>CultureInfo</code> bestimmt) arbeiten, ist das beispielsweise der Vermerk »n. Christ.«.
H	Zeigt die Stunde des angegebenen <i>Date</i> -Wertes im Bereich 1–12 an. Die Stunde stellt die ganzen Stunden dar, die seit Mitternacht (als 12 dargestellt) oder Mittag (ebenfalls als 12 dargestellt) vergangen sind. Wenn dieses Format einzeln verwendet wird, können die jeweils gleichen Stunden vor und nach Mittag nicht unterschieden werden. Wenn die Stundenzahl nur einstellig ist, wird diese auch als einzelne Ziffer angezeigt. Wichtig: Für das 24-Stunden-Format wählen Sie das große »H«.

Formatzeichen	Beschreibung
Hh	Zeigt die Stunde des angegebenen <i>Date</i> -Wertes im Bereich 1–12 an. Die Stunde stellt die ganzen Stunden dar, die seit Mitternacht (als 12 dargestellt) oder Mittag (ebenfalls als 12 dargestellt) vergangen sind. Wenn dieses Format einzeln verwendet wird, können die jeweils gleichen Stunden vor und nach Mittag nicht unterschieden werden. Wenn die Stundenzahl nur einstellig ist, wird eine führende Null vorangestellt. Wichtig: Für das 24-Stunden-Format wählen Sie das große »H«.
H	Zeigt die Stunde des angegebenen <i>Date</i> -Wertes im Bereich 0–23 an. Die Stunde stellt die seit Mitternacht (als 0 angezeigt) vergangenen Stunden dar. Wenn die Stundenzahl nur einstellig ist, wird sie auch nur als einzelne Ziffer angezeigt.
HH	Zeigt die Stunde des angegebenen <i>Date</i> -Wertes im Bereich 0–23 an. Die Stunde stellt die seit Mitternacht (als 0 angezeigt) vergangenen Stunden dar. Wenn die Stundenzahl einstellig ist, wird dieser Ziffer eine führende 0 vorangestellt.
m	Zeigt die Minute des angegebenen <i>Date</i> -Wertes im Bereich 0–59 an. Die Minute stellt die seit der letzten Stunde vergangenen ganzen Minuten dar. Wenn die Minute nur einstellig ist, wird diese auch nur als einzelne Ziffer angezeigt.
mm	Zeigt die Minute des angegebenen <i>Date</i> -Wertes im Bereich 0–59 an. Die Minute stellt die seit der letzten Stunde vergangenen ganzen Minuten dar. Wenn die Minute nur eine einzelne Ziffer ist (0–9), wird dieser Ziffer bei der Formatierung eine 0 (01–09) vorangestellt.
M	Zeigt den Monat als Zahl zwischen 1 und 12 an. Die Ausgabe erfolgt bei einstelligen Werten ohne Führungsnull. Wichtig: Denken Sie daran, hier Großbuchstaben zu verwenden, da Sie sonst eine Minutenausgabe erwirken.
MM	Zeigt den Monat als Zahl zwischen 1 und 12 an. Die Ausgabe erfolgt bei einstelligen Werten mit Führungsnull. Wichtig: Denken Sie daran, hier Großbuchstaben zu verwenden, da Sie sonst eine Minutenausgabe erwirken.
MMM	Zeigt den abgekürzten Namen des Monats für den angegebenen <i>Date</i> -Wert an. Wichtig: Denken Sie daran, hier Großbuchstaben zu verwenden, da Sie sonst eine Minutenausgabe erwirken.
MMMM	Zeigt den vollständigen Namen des Monats für den angegebenen <i>Date</i> -Wert an. Wichtig: Denken Sie daran, hier Großbuchstaben zu verwenden, da Sie sonst eine Minutenausgabe erwirken.
s	Zeigt die Sekunden im Bereich 0–59 an. Die Sekunde stellt die seit der letzten Minute vergangenen ganzen Sekunden dar. Die Ausgabe erfolgt bei einstelligen Werten ohne Führungsnull.
ss	Zeigt die Sekunden im Bereich 0–59 an. Die Sekunde stellt die seit der letzten Minute vergangenen ganzen Sekunden dar. Die Ausgabe erfolgt bei einstelligen Werten mit Führungsnull.
t	Zeigt das erste Zeichen des A.M./P.M.-Bezeichners für den angegebenen <i>Date</i> -Wert an. Wichtig: Beim deutschen <i>CultureInfo</i> -Format-Provider (entspricht auf deutschen .NET-Systemen der Standardeinstellung, wenn kein Format Provider angegeben wurde), bleibt dieses Formatzeichen unberücksichtigt – wird also ignoriert.
tt	Zeigt den vollständigen A.M./P.M.-Bezeichner für den angegebenen <i>Date</i> -Wert an. Wichtig: Beim deutschen <i>CultureInfo</i> -Format-Provider (entspricht auf deutschen .NET-Systemen der Standardeinstellung, wenn kein Format Provider angegeben wurde), bleibt dieses Formatzeichen unberücksichtigt – wird also ignoriert.
y	Zeigt das Jahr an. Die ersten beiden Ziffern des Jahres werden ausgelassen. Die Ausgabe erfolgt bei einstelligen Werten ohne Führungsnull.
yy	Zeigt das Jahr an. Die ersten beiden Ziffern des Jahres werden ausgelassen. Die Ausgabe erfolgt bei einstelligen Werten mit Führungsnull.
yyyy	Zeigt das Jahr an. Wenn das Jahr aus weniger als vier Ziffern besteht, werden diesem Nullen vorangestellt, damit es vier Ziffern enthält.

Formatzeichen	Beschreibung
z	Zeigt das Offset der aktuellen Zeitzone des Systems in ganzen Stunden an. Das Offset wird immer mit einem vorangestellten Plus- bzw. Minuszeichen angezeigt (Null wird als »+0« angezeigt), das die Stunden vor (+) GMT (Greenwich Mean Time) bzw. nach (-) GMT angibt. Der Wertebereich liegt zwischen -12 und +13 Stunden. Ist das Offset einstellig, erfolgt die Anzeige ebenfalls nur einstellig; es werden keine führenden Nullen vorangestellt. Die Einstellung für die Zeitzone wird als +X oder -X angegeben, wobei X der Offset zur GMT in Stunden ist. Das angezeigte Offset kann durch die Sommer-/Winterzeitumstellungen beeinflusst werden.
zz	Wie vorher; das Offset wird bei einstelligen Werten jedoch mit führender Null angezeigt.
zzz	Wie vorher; das Offset wird aber in Stunden und Minuten angezeigt.
:	Trennzeichen für Zeitangaben.
/	Trennzeichen für Datumsangaben. WICHTIG: Wenn Sie für das Parsen eines Datumswertes einen Schrägstrich verlangen, müssen Sie »\« verwenden, um den Schrägstrich als Gruppentrennzeichen zu bestimmen.
"	Zeichenfolge in Anführungszeichen. Zeigt den literalen Wert einer Zeichenfolge zwischen zwei Anführungszeichen an, denen ein Escapezeichen (/) vorangestellt ist.
'	Zeichenfolge in Anführungszeichen. Zeigt den literalen Wert einer Zeichenfolge zwischen zwei '«-Zeichen an.
Jedes andere Zeichen	Andere Zeichen werden als Literale direkt in die Ergebniszzeichenfolge geschrieben.

Tabelle 26.4 Für Formatzeichenfolgen zur Aufbereitung von Datums- und Zeitwerten verwenden Sie diese Formatzeichen

Die folgenden Beispiele sollen den Umgang mit den Formatzeichenfolgen verdeutlichen:

```
Module FormatDemosFormatzeichenfolgen
Sub Main()
    Dim locDate As Date = #12/24/2008 10:32:22 PM#
    Dim locFormat As String

    'Normale Datumsausgabe; große Ms ergeben den Monat.
    locFormat = "dd.MM.yyyy"
    Console.WriteLine("'{0}' : {1}", _
                      locFormat, _
                      locDate.ToString(locFormat, CultureInfo.CurrentCulture))

    'Falsche Ausgabe; anstelle des Monats werden Minuten ausgegeben.
    locFormat = "dd.mm.yyyy"
    Console.WriteLine("'{0}' : {1}", _
                      locFormat, _
                      locDate.ToString(locFormat, CultureInfo.CurrentCulture))

    'Komplette Ausgabe, ausführlicher geht's nicht.
    locFormat = "dddd, dd.MMMM.yyyy - HH:mm:ss:fffffff ""(Offset:\"" zzz)""
    Console.WriteLine("'{0}' : {1}", _
                      locFormat, _
                      locDate.ToString(locFormat, CultureInfo.CurrentCulture))

    'Falsche Ausgabe; der Text "Uhrzeit" steht nicht in Anführungszeichen.
    locFormat = "Uhrzeit: HH:mm:ss"
    Console.WriteLine("'{0}' : {1}", _
                      locFormat, _
                      locDate.ToString(locFormat, CultureInfo.CurrentCulture))
```

```

'So geht es richtig:
locFormat = """Uhrzeit:"" HH:mm:ss"
Console.WriteLine("'{0}' : {1}", _
                  locFormat,
                  locDate.ToString(locFormat, CultureInfo.CurrentCulture))
'PM-Anzeige funktioniert nicht bei deutscher...
locFormat = """Uhrzeit:"" hh:mm:ss tt"
Console.WriteLine("'{0}' : {1}", _
                  locFormat,
                  locDate.ToString(locFormat, New CultureInfo("de-DE")))

'...aber beispielsweise bei amerikanischer Kultureinstellung
locFormat = """Uhrzeit:"" hh:mm:ss tt"
Console.WriteLine("'{0}' : {1}", _
                  locFormat,
                  locDate.ToString(locFormat, New CultureInfo("en-US")))

'Englisches Datumsformat trotz deutscher Kultureinstellung
locFormat = """Date:"" MM\dd\yyyy"
Console.WriteLine("'{0}' : {1}", _
                  locFormat,
                  locDate.ToString(locFormat, New CultureInfo("de-DE")))

'Backslash davor nicht vergessen, sonst:
locFormat = """Date:"" MM/dd/yyyy"
Console.WriteLine("'{0}' : {1}", _
                  locFormat,
                  locDate.ToString(locFormat, New CultureInfo("de-DE")))

'Aber nur so mit englischen Texten:
locFormat = """Date:"" MMMM, ""the"" dd. yyyy"
Console.WriteLine("'{0}' : {1}", _
                  locFormat,
                  locDate.ToString(locFormat, New CultureInfo("en-US")))

Console.ReadLine()
End Sub
End Module

```

Lassen Sie dieses Programm laufen, ergibt sich folgende Ausgabe im Konsolenfenster:

```

'dd.MM.yyyy' : 24.12.2008
'dd.mm.yyyy' : 24.32.2008
'dddd, dd.MMMM.yyyy - HH:mm:ss:fffffff "(Offset: " zzz)' : Mittwoch, 24.Dezember.2008 - 22:32:22:000000
(Offset: +01:00)
'Uhrzeit: HH:mm:ss' : U10r+lei: 22:32:22
'"Uhrzeit:" HH:mm:ss' : Uhrzeit: 22:32:22
'"Uhrzeit:" hh:mm:ss tt' : Uhrzeit: 10:32:22
'"Uhrzeit:" hh:mm:ss tt' : Uhrzeit: 10:32:22 PM
'"Date:" MM\dd\yyyy' : Date: 12/24/2008
'"Date:" MM/dd/yyyy' : Date: 12.24.2008
'"Date:" MMMM, "the" dd. yyyy' : Date: December, the 24. 2008

```

Formatierung von Zeitausdrücken durch vereinfachte Formatzeichenfolgen

Für die bequeme Realisierung von Zeit- und Datumsformatierungen gibt es in .NET vereinfachte Formatzeichenfolgen. Anstatt beispielsweise gezielt die Formatzeichenkombinationen zusammenzustellen, die ein Datum in Langform formatieren, reicht das Zurückgreifen auf bestimmte einzelne Zeichen. In Kombination mit dem entsprechenden `CultureInfo`-Format-Provider brauchen Sie sich obendrein noch nicht einmal um die in der Kultur übliche Darstellung zu kümmern:

```
Sub main()
    Dim locDate As Date = #12/24/2008 10:32:22 PM#
    Dim locFormat As String

    locFormat = "dddd, dd. MMMM yyyy"
    Console.WriteLine("Datumsformatierung mit Formatzeichen:" + locDate.ToString(locFormat))
    Console.WriteLine("...und mit vereinfachten Formatzeichen:" + locDate.ToString("D"))
    Console.ReadLine()
End Sub
```

Die folgende Tabelle zeigt, welche vereinfachten Formatzeichenfolgen Sie verwenden können, um Datums- und Zeitformatierungen durchzuführen. Viele der vereinfachten Formatzeichen haben äquivalente Formatzeichenfolgen, die mit entsprechenden (in der Tabelle angegebenen) Eigenschaften mithilfe eines `DateTimeFormatInfo`-Objektes ermittelt werden können.

Formatzeichen	Zugeordnete Eigenschaft/Beschreibung
d	Kurzform des Datums. Entspricht der Formatzeichenfolge, die die Eigenschaft <code>ShortDatePattern</code> zurückliefert.
D	Langform des Datums. Entspricht der Formatzeichenfolge, die die Eigenschaft <code>LongDatePattern</code> zurückliefert.
f	Vollständiges Datum und Uhrzeit (Langform des Datums und 24-Stunden-Zeitformat).
F	Langform des Datums und der Uhrzeit. Entspricht der Formatzeichenfolge, die die Eigenschaft <code>FullDateTimePattern</code> zurückliefert.
g	Allgemein (Kurzform des Datums sowie 24-Stunden-Zeitformat).
G	Allgemein (Kurzform des Datums und Langform der Zeit).
m, M	Ergibt den Tag und den ausgeschriebenen Monatsnamen (beispielsweise 24. Dezember). Entspricht der Formatzeichenfolge, die die Eigenschaft <code>MonthDayPattern</code> zurückliefert.
r, R	Ergibt Datum und Zeit in der so genannten RFC1123-Norm. Dabei hat die Zeiteingabe das Format beispielsweise von »Wed, 24 Dec 2008 22:32:22 GMT«. Entspricht der Formatzeichenfolge, die die Eigenschaft <code>RFC1123Pattern</code> zurückliefert. Diese Form wird bei der Kommunikation über das Internet verwendet, beispielsweise bei der Header-Dokumentierung von Mail-Dateien.
s	Ergibt ein auf der Grundlage von ISO 8601 unter Verwendung der Ortszeit formatiertes sortierbares Datum (beispielsweise »2008-12-24T22:32:22«). Entspricht der Formatzeichenfolge, die die Eigenschaft <code>SortableDateTimePattern</code> zurückliefert.
t	Kurzform der Zeit. Entspricht der Formatzeichenfolge, die die Eigenschaft <code>ShortTimePattern</code> zurückliefert.
T	Langform der Zeit. Entspricht der Formatzeichenfolge, die die Eigenschaft <code>LongTimePattern</code> zurückliefert.

Formatzeichen	Zugeordnete Eigenschaft/Beschreibung
u	Ergibt ein unter Verwendung des Formats zur Anzeige der koordinierten Weltzeit formatiertes sortierbares Datum (beispielsweise »2008-12-24 22:32:22Z«). Entspricht der Formatzeichenfolge, die die Eigenschaft <i>UniversalSortableDateTimePattern</i> zurückliefert.
U	Vollständiges Datum und Uhrzeit (langes Datumsformat und langes Zeitformat) unter Verwendung der koordinierten Weltzeit.
y, Y	Monat und Jahreszahl (etwa »Dezember 2008«). Entspricht der Formatzeichenfolge, die die Eigenschaft <i>YearMonthPattern</i> zurückliefert.

Tabelle 26.5 Für vereinfachte Formatzeichen zur Aufbereitung von Datums- und Zeitwerten verwenden Sie diese Tabelle

Gezielte Formatierungen mit Format Providern

Das Ziel bei der Implementierung von Format Providern in .NET war es, dem Entwickler ein möglichst universelles Werkzeug zum Aufbereiten von Daten zum Zweck der übersichtlichen Ausgabe auf Bildschirmen oder in Dateien zu bieten. Gleichzeitig sollten kulturabhängige Formatierungseigenarten berücksichtigt werden.

Letzteres haben Sie bereits in Form des *CultureInfo*-Objektes kennen gelernt. Wenn Sie der *Tostring*-Funktion eines primitiven Datentyps ein *CultureInfo*-Objekt übergeben, das unter Angabe eines bestimmten Kulturkennzeichens instanziiert wurde, passt sich die Ausgabe an die in der Kultur übliche Datendarstellungsweise an.

Neben dem *CultureInfo*-Objekt kennt das Framework zwei weitere Format Provider, die die Formatierung von numerischen Werten (*NumberFormatInfo*) und Datums- und Zeitwerten (*DateTimeFormatInfo*) genauer spezifizieren. Diese Format Provider arbeiten in Zusammenarbeit mit den vereinfachten Formatzeichen.

Gezielte Formatierungen von Zahlenwerten mit NumberFormatInfo

Angenommen, Sie möchten eine Reihe von Zahlen als Währung formatiert untereinander ausgeben. Sie möchten dabei, dass die Ziffern der einzelnen Zahlen zwar als Tausender gruppiert werden (also 1.000.000 und nicht 1000000), aber Sie möchten nicht den Punkt, sondern das Leerzeichen als Gruppierungstrennzeichen verwenden. In diesem Fall verwenden Sie eine *NumberFormatInfo*-Instanz, um die Formatierungsregel genauer zu spezifizieren:

```
Sub main()
    'Den zu verwendenden Wert definieren.
    Dim locDouble As Double = 1234567.23
    'Kultureinstellungen sind englisch/britisches.
    Dim locCultureInfo As New CultureInfo("en-GB")
    'Die Einstellungen, die CultureInfo auf Grund der Kultur schon
    'geleistet hat, übernehmen wir in ein NumberFormatInfo...
    Dim locNumFormatInfo As NumberFormatInfo = locCultureInfo.NumberFormat

    '...dessen anzuwendende Regeln wir nun genauer spezifizieren können:
    locNumFormatInfo.CurrencyGroupSeparator = " "
    Console.WriteLine("Als Währung formatiert: " + locDouble.ToString("C", locNumFormatInfo))
```

```
'Auf die normale Fließkommadarstellung hat diese Einstellung keinen Einfluss:  
Console.WriteLine("Als Fließkommazahl formatiert: " + locDouble.ToString("n", locNumFormatInfo))  
  
'Jetzt schon!  
locNumFormatInfo.NumberGroupSeparator = ""  
Console.WriteLine("Als Fließkommazahl formatiert: " + locDouble.ToString("n", locNumFormatInfo))  
End Sub
```

Wenn Sie diese kleine Prozedur laufen lassen, produziert sie folgende Ausgabe:

```
Als Währung formatiert: £1 234 567.23  
Als Fließkommazahl formatiert: 1,234,567.23  
Als Fließkommazahl formatiert: 1 234 567.23
```

NumberFormatInfo stellt Ihnen für die Spezialisierung von Formatierungen im hier gezeigten Stil eine ganze Reihe von Eigenschaften zur Verfügung, die Sie nach Belieben vor dem Einsatz in ToString einstellen können. Welche Eigenschaften welche Änderung bewirken, entnehmen Sie bitte der Online-Hilfe von Visual Studio.

Gezielte Formatierungen von Zeitwerten mit DateTimeFormatInfo

Was für die Spezialisierung von numerischen Formatierungen gilt, ist auch für die Datums- und Zeitenformatierung gültig. Sie verwenden die DateTimeFormatInfo-Klasse, um Formatierungen dieses Datentyps zu spezialisieren.

Auch hier soll ein Beispiel dem besseren Verständnis dienen. Normalerweise gibt es keine AM/PM-Designatoren (Tagesbereichsbezeichner beim englisch/amerikanischem Datumsformat) für die deutschen Kultureinstellungen. Das folgende kleine Beispielprogramm richtet die Designatoren mit einem DateTimeFormatInfo-Objekt gezielt mit deutschen Bezeichnungen ein, und sie werden anschließend durch die Angabe der entsprechenden Formatzeichen in der Formatzeichenkette mit ToString bei der Umwandlung des Datums ausgegeben.

```
Sub main()  
  
'Zu verwendenden Wert definieren.  
Dim locDate As Date = #12/24/2008 1:12:23 PM#  
'Kultureinstellungen sind deutsch.  
Dim locCultureInfo As New CultureInfo("de-DE")  
'Die Einstellungen, die CultureInfo auf Grund der Kultur schon  
'geleistet hat, übernehmen wir in ein DateTimeFormatInfo...  
Dim locDateTimeFormatInfo As DateTimeFormatInfo = locCultureInfo.DateTimeFormat  
  
'...dessen anzuwendende Regeln wir nun genauer spezifizieren können:  
locDateTimeFormatInfo.AMDesignator = "Vormittag"  
locDateTimeFormatInfo.PMDesignator = "Nachmittag"  
Console.WriteLine("Mit deutschen AM/PM-Designatoren: "  
    + locDate.ToString("dd.MM.yyyy hh:mm:ss - tt", locDateTimeFormatInfo))  
'12 Stunden dazu addieren:  
locDate = locDate.AddHours(12)  
Console.WriteLine("Mit deutschen AM/PM-Designatoren: "  
    + locDate.ToString("dd.MM.yyyy hh:mm:ss - tt", locDateTimeFormatInfo))  
End Sub
```

Kombinierte Formatierungen

Kombinierte Formatierungen sind ein spezielles Feature von .NET, das die Einbindung von zu formatierendem bzw. umzuwandelndem Text gezielt an bestimmte Stellen innerhalb einer Zeichenkette ermöglicht. Dazu ein Beispiel:

Das folgende kleine Programm definiert einen bestimmten Ausgangszeitpunkt. Es addiert anschließend 15 Mal in Folge eine zufällige Zeitspanne zwischen 0 Minuten und 23 Stunden und 59 Minuten zum Ausgangsdatum und zeigt das entsprechende Ergebnis an. Das Programm dafür sieht folgendermaßen aus:

BEGLEITDATEIEN

Die Begleitdateien zu folgenden Beispielen finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 26\\CompositeFormatting

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Sub Main()
    Dim locBasisDatum As Date = #12/30/2008 1:12:32 PM#
    Dim locOffset As TimeSpan
    Dim locRandom As New Random(Now.Millisecond)
    'Backslash vor ', damit es gedruckt und nicht als Steuerzeichen interpretiert wird!
    Console.WriteLine("Es ist " +
        locBasisDatum.ToString("dd\\dd, \"der\" d. MMMM \\'yy, HH:mm") + _
        "...")

    '15 Wiederholungen
    For count As Integer = 1 To 15
        locOffset = New TimeSpan(locRandom.Next(23), locRandom.Next(59), 0)
        locBasisDatum = locBasisDatum.Add(locOffset)
        Console.WriteLine("...und " + Math.Floor(locOffset.TotalHours).ToString() + _
            " Std. und " + locOffset.Minutes.ToString() + _
            " Min. später ist " +
            locBasisDatum.ToString("dd\\dd, \"der\" d. MMMM \\'yy, HH:mm"))
    Next
End Sub
```

Es produziert etwa folgende Ausgabe:

```
Es ist Dienstag, den 30. Dezember '08, 13:12...
...und 19 Std. und 13 Min. später ist Mittwoch, der 31. Dezember '08, 08:25
...und 18 Std. und 8 Min. später ist Donnerstag, der 1. Januar '09, 02:33
...und 3 Std. und 20 Min. später ist Donnerstag, der 1. Januar '09, 05:53
...und 14 Std. und 3 Min. später ist Donnerstag, der 1. Januar '09, 19:56
...und 0 Std. und 11 Min. später ist Donnerstag, der 1. Januar '09, 20:07
...und 1 Std. und 27 Min. später ist Donnerstag, der 1. Januar '09, 21:34
...und 18 Std. und 26 Min. später ist Freitag, der 2. Januar '09, 16:00
...und 1 Std. und 36 Min. später ist Freitag, der 2. Januar '09, 17:36
...und 12 Std. und 15 Min. später ist Samstag, der 3. Januar '09, 05:51
...und 15 Std. und 34 Min. später ist Samstag, der 3. Januar '09, 21:25
...und 11 Std. und 44 Min. später ist Sonntag, der 4. Januar '09, 09:09
...und 7 Std. und 31 Min. später ist Sonntag, der 4. Januar '09, 16:40
...und 19 Std. und 28 Min. später ist Montag, der 5. Januar '09, 12:08
...und 16 Std. und 54 Min. später ist Dienstag, der 6. Januar '09, 05:02
...und 16 Std. und 51 Min. später ist Dienstag, der 6. Januar '09, 21:53
```

Zwei Dinge fallen auf: Zum Einen ist der Programmtext ein einziges Chaos. Man muss sich Buchstabe für Buchstabe durch das Listing hangeln, um erst nach geraumer Zeit festzustellen, was die Zeilen überhaupt bewirken. Das zweite Problem: Auch die Ausgabe ist nicht wirklich sauber formatiert.

Schauen Sie sich jetzt die folgende Version des Programms an, das die exakt gleiche Ausgabe produziert:

```
Sub ZweiteVersion()

    Dim locBasisDatum As Date = #12/30/2008 1:12:32 PM#
    Dim locOffset As TimeSpan
    Dim locRandom As New Random(Now.Millisecond)

    With locBasisDatum
        Console.WriteLine("Es ist {0}, der {1}...", _
            .ToString("ddd"),
            .ToString("d. MMMM \\'yy, HH:mm"))

        '15 Wiederholungen
        For count As Integer = 1 To 15
            locOffset = New TimeSpan(locRandom.Next(23), locRandom.Next(59), 0)
            locBasisDatum = locBasisDatum.Add(locOffset)
            Console.WriteLine("...und {0} Std. und {1} Min. später ist {2}", _
                Math.Floor(locOffset.TotalHours).ToString(),
                locOffset.Minutes.ToString(),
                locBasisDatum.ToString("ddd, \"der\" d. MMMM \\'yy, HH:mm"))
        Next
    End With

End Sub
```

Was ist passiert? Sie haben durch Platzhalter in geschweiften Klammern bestimmt, an welchen Positionen innerhalb des angegebenen Textes, der die Platzhalter enthält, die folgenden Parameter eingesetzt werden sollen. Das Ergebnis kann sich sehen lassen: Das Listing ist leicht lesbar, und man versteht fast auf Anhieb, welchem Zweck es dient.

Sie können kombinierte Formatierungen nicht nur in der `Console.WriteLine`-Anweisung einsetzen. Alle Ableitungen der `TextWriter`-Klasse verstehen mit ihrer `WriteLine`-Anweisung ebenfalls kombinierte Formatierungen. Und zu guter Letzt haben Sie mit der statischen `String.Format`-Methode die Möglichkeit, einen neuen String zu produzieren, der durch die Möglichkeiten der kombinierten Formatierung aufbereitet werden kann.

Ausrichtungen in kombinierten Formatierungen

Die Funktionalität von kombinierten Formatierungen geht noch weiter. Sie können in den so genannten Indexkomponenten – so nennen sich die in geschweiften Klammern eingefügten Platzhalter – angeben, wie der Text mit führenden Leerzeichen ausgerichtet werden soll. Dazu geben Sie mit Komma getrennt die Anzahl der Zeichen ein, auf die der Text durch das Einfügen von führenden Leerzeichen verlängert werden soll, sodass die einzufügenden Zeichen auf einer bestimmten Position enden. Angewendet auf das schon vorhandene Beispiel, ergibt sich folgende Version des Programms:

```

Sub DritteVersion()

    Dim locBasisDatum As Date = #12/30/2008 1:12:32 PM#
    Dim locOffset As TimeSpan
    Dim locRandom As New Random(Now.Millisecond)

    With locBasisDatum
        Console.WriteLine("Es ist {0}, der {1}...", _
            .ToString("dddd"),_
            .ToString("d. MMMM \yy, HH:mm"))
        '15 Wiederholungen
        For count As Integer = 1 To 15
            locOffset = New TimeSpan(locRandom.Next(23), locRandom.Next(59), 0)
            locBasisDatum = locBasisDatum.Add(locOffset)
            Console.WriteLine("...und {0,2} Std. und {1,2} Min. später ist {2,11}, der {3}", _
                Math.Floor(locOffset.TotalHours).ToString(),_
                locOffset.Minutes.ToString(),_
                .ToString("dddd"),_
                .ToString("dd. MMMM \yy, HH:mm") _
            )
        Next
    End With
End Sub

```

Wenn Sie diese Prozedur laufen lassen, ergibt sich die folgende Ausgabe auf dem Konsolenfenster:

```

Es ist Dienstag, der 30. Dezember '08, 13:12...
...und 21 Std. und 25 Min. später ist Mittwoch, der 31. Dezember '08, 10:37
...und 0 Std. und 39 Min. später ist Mittwoch, der 31. Dezember '08, 11:16
...und 18 Std. und 56 Min. später ist Donnerstag, der 01. Januar '09, 06:12
...und 14 Std. und 21 Min. später ist Donnerstag, der 01. Januar '09, 20:33
...und 6 Std. und 51 Min. später ist Freitag, der 02. Januar '09, 03:24
...und 1 Std. und 10 Min. später ist Freitag, der 02. Januar '09, 04:34
...und 13 Std. und 28 Min. später ist Freitag, der 02. Januar '09, 18:02
...und 17 Std. und 45 Min. später ist Samstag, der 03. Januar '09, 11:47
...und 12 Std. und 16 Min. später ist Sonntag, der 04. Januar '09, 00:03
...und 9 Std. und 56 Min. später ist Sonntag, der 04. Januar '09, 09:59
...und 2 Std. und 18 Min. später ist Sonntag, der 04. Januar '09, 12:17
...und 21 Std. und 4 Min. später ist Montag, der 05. Januar '09, 09:21
...und 0 Std. und 24 Min. später ist Montag, der 05. Januar '09, 09:45
...und 14 Std. und 44 Min. später ist Dienstag, der 06. Januar '09, 00:29
...und 18 Std. und 11 Min. später ist Dienstag, der 06. Januar '09, 18:40

```

Zugegeben: Schöner ist nur die Formatierung der ersten beiden Parameter geworden. Ob das Einrücken der Wochentage wirklich zur Lesbarkeit beiträgt, ist strittig. Zur Demonstration der Funktion ist es allemal ein brauchbares Ergebnis, denn: Sie können leicht erkennen, wie sich die Erweiterung der Indexkomponenten um die Angabe der Gesamtlänge der Buchstaben (die entsprechende Zeile im Listing ist fett markiert) auf das Ergebnis auswirken.

Angeben von Formatzeichenfolgen in den Indexkomponenten

Zu guter Letzt gibt es eine weitere Möglichkeit, die Indexkomponenten zu parametrisieren. Sie können die Formatzeichenfolge, die sich in den bisherigen Versionen bei den zu formatierenden Parametern selbst befand, ebenfalls in jeder Indexkomponente angeben. Dazu trennen Sie die Formatzeichen per Doppel-

punkt von den weiteren Parametern. Das folgende Listing zeigt die letzte Version der Prozedur, bei der sie von dieser Möglichkeit Gebrauch macht:

```
Sub VierteVersion()
    Dim locBasisDatum As Date = #12/30/2008 1:12:32 PM#
    Dim locOffset As TimeSpan
    Dim locRandom As New Random(Now.Millisecond)

    Console.WriteLine("Es ist {0:ddd}, der {1:d. MMMM \'yy, HH:mm}...", _
        locBasisDatum, _
        locBasisDatum)

    '15 Wiederholungen
    For count As Integer = 1 To 15
        locOffset = New TimeSpan(locRandom.Next(23), locRandom.Next(59), 0)
        locBasisDatum = locBasisDatum.Add(locOffset)

        Console.WriteLine("...und {0,2} Std. und {1,2} Min. später ist {2,11:ddd}, der {3:dd. MMMM \'yy, HH:mm}", _
            Math.Floor(locOffset.TotalHours).ToString(), _
            locOffset.Minutes.ToString(), _
            locBasisDatum, _
            locBasisDatum - )
    Next
End Sub
```

WICHTIG Achten Sie bei dieser Version auf zwei wesentliche Änderungen. Da zum Einen die Formatierungsanweisungen in die Indexkomponenten verschoben wurden, dürfen Sie jetzt nicht mehr die `ToString`-Funktion der einzelnen zu formatierenden Daten verwenden, da diese zuvor für die korrekte Formatierung zuständig war. Würden Sie die `ToString`-Funktion beibehalten, so würde die Formatierungsfunktion, die durch `WriteLine` ins Leben gerufen wird (der so genannte *Formatter*), versuchen, die Formatzeichen auf eine Zeichenkette und nicht auf den `Date`-Typ anzuwenden. Er würde natürlich ins Leere laufen, da Zeichenketten selbst nicht mit den Formatzeichen für den `Date`-Typ zu formatieren sind.

Beachten Sie auch, dass das bündige Ausrichten durch Leerzeichen nur mit nicht-proportionalen Zeichensätzen möglich ist. Bei proportionalen Zeichensätzen, bei denen Buchstaben nicht gleich groß sind (»www« ist viel länger als »iii«, mit anderen Worten: Ich schreibe gerade in einer proportionalen Schrift), schlägt das Formatieren mit Leerzeichen natürlich fehl.

So helfen Ihnen benutzerdefinierte Format Provider, Ihre Programme zu internationalisieren

.NET erlaubt das Erstellen von benutzerdefinierten Format Providern. Um zu verstehen, wie Sie Format Provider in .NET entwickeln, lassen Sie mich das Pferd anhand des Ergebnisses eines Beispiels von hinten aufzäumen: Stellen Sie sich vor, Sie müssten ein Programm entwickeln, dass nicht nur die Konvertierung von Maßeinheiten vornehmen kann, sondern den Entwickler seiner Klasse auch bei der Aufbereitung der Werte unterstützt.

Ein Programm soll folgendes Problem lösen: Es erlaubt seinem Anwender, einen Wert in Metern einzugeben; das Programm wird anschließend die Werte in die in deutsch- und englischsprachigen Kulturen üblichen Maßeinheiten umrechnen, etwa wie im folgenden Beispiel zu sehen:

Geben Sie einen Wert in Metern zur Umrechnung ein: 550			
mm 550.000,00000	cm 55.000,00000	m 550,00000	km 0,55000
lines 259.820,00000	inches 21.653,50000	yards 601,70000	miles 0,34177

Soweit ist das Programm absolut nichts Besonderes – jeder Basic-Nieueinsteiger programmierte so etwas schon vor Jahrzehnten nach ein paar Stunden.

Allerdings ist der Weg dorthin schon bemerkenswerter. Das folgende Programm zeigt, wie die einzelnen Zeilen Zustände gekommen sind.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 26\\CustomFormatProvider01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module CustomFormatProvider

Sub Main()

    Dim locEngKultur As New CultureInfo("en-US")

    Console.WriteLine("Geben Sie einen Wert in Metern zur Umrechnung ein: ")
    Dim locLaenge As New Laengen(Decimal.Parse(Console.ReadLine()))

    'Umgerechneten Wert anzeigen:
    Console.WriteLine("      mm      |      cm      |      m      |      km      |")
    Console.WriteLine("{0,17} |{1,17} |{2,17} |{3,17} |", _
        locLaenge.ToString("s-d;#,##0.00000"), _
        locLaenge.ToString("k-d;#,##0.00000"), _
        locLaenge.ToString("m-d;#,##0.00000"), _
        locLaenge.ToString("g-d;#,##0.00000"))

    Console.WriteLine()
    'Umgerechneten Wert anzeigen:
    Console.WriteLine("      lines      |      inches      |      yards      |      miles      |")
    Console.WriteLine("{0,17} |{1,17} |{2,17} |{3,17} |", _
        locLaenge.ToString("s;#,##0.00000", locEngKultur), _
        locLaenge.ToString("k-e;#,##0.00000"), _
        locLaenge.ToString("m;#,##0.00000", _
            New LaengenFormatInfo(LaengenKultur.EnglischAmerikanisch, LaengenAufloesung.Mittel)), _
        locLaenge.ToString("g;#,##0.00000", locEngKultur))
    Console.ReadLine()

End Sub

End Module
```

Sie können leicht erkennen, dass Sie in diesem Programm quasi keine einzige Berechnung finden können. Es werden auch keine Eigenschaften oder Funktionen einer speziellen Klasse aufgerufen – alle Konvertierungen finden ausschließlich über die Steuerung entweder von Formatzeichenfolgen oder – was auf den ersten Blick noch nicht offensichtlich ist – über mehr oder weniger spezielle Format Provider statt.

Die Konvertierungen werden aber nichtsdestotrotz durch eine spezielle Klasse realisiert – sie nennt sich für dieses Beispiel schlicht und einfach Laengen. Dies ist eine von mir selbst geschriebene Klasse, suchen Sie sie also nicht im .NET Framework, dazu später mehr. Sie stellt auf der einen Seite die wirklich simplen Konvertierungsfunktionen zur Verfügung (mit Methoden wie ToMile, ToInch, etc.). Auf der anderen Seite erweitert sie die ToString-Funktion der Basisklasse (sie ist direkt von Object abgeleitet). ToString nimmt wahlweise einen oder zwei Parameter entgegen, und zwar in dem Stil, den Sie in den vergangenen Abschnitten schon kennen gelernt haben. Sie verarbeitet Formatzeichenfolgen – wahlweise in Kombination mit einem Format Provider.

Die Formatzeichenfolgen akzeptieren Formatzeichenfolgen, die eigentlich Kombinationen aus zweien sind – oder zumindest sein können. Der erste Teil einer Formatzeichenfolge steuert, welche Maßeinheit bei der Ausgabe verwendet werden soll; der zweite Teil – und jetzt kommt der internationalisierende Part – bestimmt, welche Kulturvorgaben dabei verwendet werden sollen. Aus diesem Grund bestimmen Sie mit der Angabe von den Formatzeichenfolgen auch nicht direkt »Zentimeter« oder »Inch« (die englische Einheit für Zoll), sondern geben vielmehr eine Skalierungsbezeichnung an, wahlweise in Kombination mit einem Kulturbuchstaben. Die Skalierung habe ich der Einfachheit halber in *sehr klein, klein, mittel* und *groß* festgelegt. Diese Version der Klasse unterscheidet ferner die beiden Kulturen *deutsch* und *amerikanisch-englisch*.

Damit haben Sie, ohne direkte Bezeichnungen definieren zu müssen, kulturabhängig die Möglichkeit, verschiedene Skalierungen zu verwenden. Mit den folgenden Anweisungen lassen Sie beispielsweise eine Zeile ausgeben, die die *sehr kleine* Skalierung mit einem Wert für die Klasse Laenge von 1 verwendet:³

```
Sub Spielchen()
    'Definiert eine Laengen-Instanz mit 1 (einem Meter).
    Dim locLaenge As New Laengen(1)
    'Gibt auf einem deutschen System den Klasseninstanzwert in Millimeter aus.
    Console.WriteLine(locLaenge.ToString("s"))

    'Auf einem englischen oder amerikanischen System würde die vorherige Zeile
    'die gleiche Ausgabe wie die folgende bewirken:
    Console.WriteLine(locLaenge.ToString("s", New CultureInfo("en-US")))
    Console.ReadLine()
End Sub
```

Dieses Programm gibt die folgenden Zeilen aus:

```
1000
472,4
```

³ Die einzelnen Testroutinen befinden sich alle innerhalb des Moduls. Wenn Sie sie selber ausprobieren wollen, brauchen Sie am Anfang des Moduls lediglich die Kommentare der ersten beiden Zeilen zu entfernen und den Aufruf der jeweils vorgestellten Prozedur dort einzusetzen.

Sehr klein im Deutschen bedeutet bei diesem Beispiel *Millimeter*. Da die Klasse in der Einheit *Meter* definiert wird, druckt die entsprechende Programmzeile korrekt 1000 (für 1000 Millimeter) aus und in der englischen Version, in der *sehr klein* die Einheit *Lines* bedeutet, korrekt 472,4.

Welche Kultur Sie bei der Ausgabe berücksichtigen, lässt sich bei der Laengen-Klasse nicht nur mit dem CultureInfo-Objekt steuern, wie im letzten Beispiel gesehen. Sie haben nämlich auch die Möglichkeit, die Kultur in einer Erweiterung der Formatzeichenfolge zu bestimmen, etwa wie im folgenden Beispiel, das kein CultureInfo-Objekt verwendet:

```
Sub Spielchen2()
    'Definiert eine Laengen-Instanz mit 1 (einem Meter).
    Dim locLaenge As New Laengen(1)
    'Gibt auf jedem System den Klasseninstanzwert in Millimeter aus.
    Console.WriteLine(locLaenge.ToString("s-d"))
    'Gibt auf jedem System den Klasseninstanzwert in Lines aus.
    Console.WriteLine(locLaenge.ToString("s-e"))

    Console.ReadLine()
End Sub
```

Die Ausgabe ist dieselbe wie im vorherigen Beispiel.

Die Steuerung mit Formatzeichen erlaubt aber noch mehr. Mit Semikolon getrennt können Sie eine Formattierung für die Werte an sich bestimmen, wie Sie es im Abschnitt »Formatierung von numerischen Ausdrücken durch Formatzeichenfolgen« ab Seite 748 schon kennen gelernt haben. Ein weiteres Beispiel zeigt diese Verwendung der Formatzeichen:

```
Sub Spielchen3()
    'Definiert eine Laengen-Instanz mit 1 (einem Meter).
    Dim locLaenge As New Laengen(1)
    'Gibt auf jedem System den Klasseninstanzwert in Millimeter aus.
    Console.WriteLine(locLaenge.ToString("s-d;#,##0.00"))
    'Gibt auf jedem System den Klasseninstanzwert in Lines aus.
    Console.WriteLine(locLaenge.ToString("s-e;#,##0.00"))

    Console.ReadLine()
End Sub
```

Die Ausgabe lautet jetzt:

```
1.000,00
472,40
```

Die Zahlen wurden den Formatzeichen entsprechend formatiert.

Die folgende Tabelle zeigt Ihnen, welche Zeichenkombinationen die `ToString`-Funktion meiner `Laengen`-Klasse auswerten kann:

Formatzeichen	Bedeutung	deutsche Maßeinheit (-d)	englisch-amerikanische Maßeinheit (-e)
s	sehr klein	Millimeter	Lines
k	klein	Zentimeter	Inches
m	mittel	Meter	Yard
g	groß	Kilometer	Miles

Tabelle 26.6 Die Laengen-Klasse versteht diese Formatzeichen

Bislang haben Sie von eigentlichen benutzerdefinierten Format Providern noch nichts gesehen. Die Formatsteuerung mit Formatzeichen in Kombination mit dem `CultureInfo`-Objekt schien zwar schon ganz gut zu funktionieren, aber die Ausgabeform eines Laengen-Klassenwertes konnten Sie nur auf Kulturseite beeinflussen.

Zusätzlich zur Laengen-Klasse gibt es im Beispielprogramm aber auch einen richtigen Format Provider – er nennt sich passenderweise `LaengenFormatInfo`. Seine Funktionsweise demonstriert das folgende Beispiel:

```
Sub Spielchen4()

    'Definiert eine Laengen-Instanz mit 1 (einem Meter).
    Dim locLaenge As New Laengen(1)
    Dim locLaengenFormatInfo As New LaengenFormatInfo

    locLaengenFormatInfo.Aufloesung = LaengenAufloesung.SehrKlein
    locLaengenFormatInfo.Kultur = LaengenKultur.Deutsch

    'Gibt auf jedem System den Klasseninstanzwert in Millimeter aus.
    Console.WriteLine(locLaenge.ToString(locLaengenFormatInfo))
    'Gibt auf jedem System den Klasseninstanzwert in Lines aus.
    locLaengenFormatInfo.Kultur = LaengenKultur.EnglischAmerikanisch
    Console.WriteLine(locLaenge.ToString(locLaengenFormatInfo))

    Console.ReadLine()

End Sub
```

Die Verwendungsweise lehnt sich an die bekannten Format Provider `NumberFormatInfo` und `DateTimeFormatInfo` an. Sie instanziieren die Klasse, setzen bestimmte Eigenschaften (im Beispieldiagramm fett gekennzeichnet), und wenn Sie die Klasseninstanz der `ToString`-Funktion der Laengen-Klasse übergeben, passt sie die Formatierung des Ausgabetextes entsprechend an.

Das Ergebnis dieses Beispiels ist wieder das des vorherigen.

Nachdem Sie die Laengen-Klasse nun umfassend anzuwenden gelernt haben, will ich Ihnen die genaue Funktionsweise nicht vorenthalten.

Die Klasse besteht zunächst einmal aus dem Konstruktor, und einer ganzen Menge einfacher Umrechnungsfunktionen, die folgendermaßen implementiert sind:

```
Public Class Laengen
    'Speichert die Länge in Meter.
    Private myLaenge As Decimal

    Sub New(ByVal Meter As Decimal)
        myLaenge = Meter
    End Sub

    Public Shared Function FromMile(ByVal Mile As Decimal) As Laengen
        Return New Laengen(Mile * 1609D)
    End Function

    Public Shared Function FromYard(ByVal Yard As Decimal) As Laengen
        Return New Laengen(Yard * 0.9144D)
    End Function

    Public Shared Function FromInch(ByVal Inch As Decimal) As Laengen
        Return New Laengen(Inch * 0.0254D)
    End Function

    Public Shared Function FromLine(ByVal Line As Decimal) As Laengen
        Return New Laengen(Line * 0.002117D)
    End Function

    Public Shared Function FromKilometer(ByVal Kilometer As Decimal) As Laengen
        Return New Laengen(Kilometer * 1000D)
    End Function

    Public Shared Function FromCentimeter(ByVal Centimeter As Decimal) As Laengen
        Return New Laengen(Centimeter * 0.01D)
    End Function

    Public Shared Function FromMillimeter(ByVal Millimeter As Decimal) As Laengen
        Return New Laengen(Millimeter * 0.001D)
    End Function

    Public Function ToMeter() As Decimal
        Return myLaenge
    End Function

    Public Function ToKilometer() As Decimal
        Return myLaenge * 0.001D
    End Function

    Public Function ToCentimeter() As Decimal
        Return myLaenge * 100D
    End Function

    Public Function ToMillimeter() As Decimal
        Return myLaenge * 1000D
    End Function

    Public Function ToMile() As Decimal
        Return myLaenge * 0.0006214D
    End Function
```

```

End Function

Public Function ToYard() As Decimal
    Return myLaenge * 1.094D
End Function

Public Function ToInch() As Decimal
    Return myLaenge * 39.37D
End Function

Public Function ToLine() As Decimal
    Return myLaenge * 472.4D
End Function

```

Das Interessante sind anschließend die verschiedenen Überladungen der `ToString`-Funktionen, mit denen der Inhalt der Klasseninstanz in Zeichenketten umgewandelt werden kann:

```

Public Overloads Function ToString(ByVal format As String) As String
    Return ToString(format, Nothing)
End Function

Public Overloads Function ToString(ByVal formatProvider As System.IFormatProvider) As String
    Return ToString(Nothing, formatProvider)
End Function

Public Overloads Function ToString(ByVal formatChars As String,
    ByVal formatProvider As System.IFormatProvider) As String
    Trace.WriteLine("ToString (Formattable-Signatur) wurde aufgerufen!")

    If (TypeOf formatProvider Is CultureInfo) Or formatProvider Is Nothing Then
        formatProvider = LaengenFormatInfo.FromFormatProvider(formatProvider)
    ElseIf Not (TypeOf formatProvider Is LaengenFormatInfo) Then
        Dim up As New _
            FormatException("Der Format Provider wird für die Klasse Laengen nicht unterstützt!")
        Throw up
    End If

    'LaengenFormatInfo-Provider enthält die Format-Aufbereitungsroutine
    Return DirectCast(formatProvider, LaengenFormatInfo).Format(formatChars, Me, formatProvider)
End Function

End Class

```

Erwähnenswert an dieser Stelle ist die Vorgehensweise zum Erkennen des Typs des übergebenen Format Providers am Anfang des Listings. Hier wird nämlich kein fester Typ als Parameter übernommen, sondern eine Schnittstelle. Eine Schnittstelle deswegen, damit wahlweise ein `CultureInfo`-Objekt, ein `LaengenFormatInfo`-Objekt oder `Nothing` übergeben werden kann. Mit `Type Of` überprüft `ToString` dabei, um welchen Typ es sich bei der Schnittstelle genau handelt (die relevanten Stellen sind im Listing wieder fett markiert). Sollte `Nothing` oder eine `CultureInfo` übergeben worden sein, dann kümmert sich die statische Methode `FromFormatProvider` der `LaengenFormatInfo`-Klasse darum, dass im Anschluss nur mit eben diesem Format Provider gearbeitet wird. Und eigentlich macht genau diese Funktionsweise die Internationalisierung des

Programms aus: Wenn kein Format Provider übergeben wurde, dann legt – wie wir später noch sehen werden – die statische Funktion nicht etwa direkt ein LaengenFormatInfo-Objekt an, sondern zunächst ein CultureInfo-Objekt. Dieses CultureInfo-Objekt ist aber nicht irgendeins, sondern es spiegelt die voreingestellte Kultur des aktuellen Threads wider – und damit die Grundeinstellung des Rechners. Erst jetzt passiert die Konvertierung in ein LaengenFormatInfo-Objekt – mit dem Ergebnis, dass auf einem englischen System automatisch englische und auf einem deutschen System automatisch deutsche Maßeinheiten verwendet werden.

Die Formatierungsroutine selbst befindet sich ebenfalls in unserem LaengenFormatInfo-Objekt. Da das zu diesem Zeitpunkt, zu dem wir es zum Aufruf von Format benötigen, aber unbedingt vorhanden ist (alle anderen möglicherweise artfremden Format Provider sind zu diesem Zeitpunkt entweder konvertiert worden oder haben eine Ausnahme ausgelöst), können wir die Schnittstellenvariable formatProvider, ohne eine Ausnahme zu riskieren, in ein LaengenFormatInfo-Objekt casten, um uns den Zugang zu dessen Format-Methode zu erschließen.

Die komplette Aufbereitungsfunktionalität an sich findet anschließend in der LaengenFormatInfo-Klasse in eben dieser Funktion statt. Diese Klasse ist im folgenden Listing zu sehen.

```
Public Enum LaengenAufloesung
    SehrKlein      ' Millimeter oder Line
    Klein          ' Zentimeter oder Inch
    Mittel         ' Meter oder Yard
    Groß           ' Kilometer oder Mile
End Enum

Public Enum LaengenKultur
    EnglischAmerikanisch
    Deutsch
End Enum
```

Diese beiden Enums (mehr zu Enums finden Sie in Kapitel 21) dienen lediglich dazu, dem Entwickler den Umgang mit der im Anschluss besprochenen LaengenFormatInfo-Klasse zu erleichtern; er muss sich dann keine Nummern für Parametereinstellungen merken, sondern kann per Namen darauf zugreifen. Die eigentliche LaengenFormatInfo-Klasse verwendet die Enums an verschiedenen Stellen.

```
Public Class LaengenFormatInfo
    Implements IFormatProvider
    Private myKultur As LaengenKultur
    Private myAufloesung As LaengenAufloesung

    Sub New()
        myKultur = LaengenKultur.Deutsch
        myAufloesung = LaengenAufloesung.Mittel
    End Sub

    Sub New(ByVal Kultur As LaengenKultur)
        myKultur = Kultur
        myAufloesung = LaengenAufloesung.Mittel
    End Sub

    Sub New(ByVal Kultur As LaengenKultur, ByVal Aufloesung As LaengenAufloesung)
```

```

myKultur = Kultur
myAufloesung = Aufloesung
End Sub
Public Shared Function FromFormatProvider(ByVal formatProvider As IFormatProvider) As
LaengenFormatInfo

    Dim retLaengenFormatInfo As LaengenFormatInfo

    If formatProvider Is Nothing Then
        formatProvider = CultureInfo.CurrentCulture
    End If

    If DirectCast(formatProvider, CultureInfo).ThreeLetterISOLanguageName = "deu" Then
        retLaengenFormatInfo =
            New LaengenFormatInfo(LaengenKultur.Deutsch, LaengenAufloesung.Mittel)
    Else
        retLaengenFormatInfo =
            New LaengenFormatInfo(LaengenKultur.EnglischAmerikanisch, LaengenAufloesung.Mittel)
    End If
    Return retLaengenFormatInfo

End Function

```

Erste Station: die schon angesprochene statische Funktion `FromFormatProvider`, die dafür sorgt, dass jeder ankommende Format Provider automatisch in einen `LaengenFormatInfo`-Format-Provider umgewandelt wird. Sie sorgt dafür – wie schon gesagt –, dass die Klasse `Laengen` sich ohne weiteres Eingreifen durch den Entwickler automatisch internationalisiert.

```

Public Function GetFormat(ByVal formatType As System.Type) As Object _
    Implements System.IFormatProvider.GetFormat

    Trace.WriteLine("Ausgabe von GetFormat:" + formatType.Name)

End Function

```

Da durch die `Implements`-Anweisung am Anfang der Klasse die `IFormatProvider`-Schnittstelle eingebunden wird, muss diese Funktion `GetFormat` ebenfalls vorhanden sein. Momentan enthält sie nur eine einzelne Anweisung, die – sollte sie von wem oder was auch immer aufgerufen werden – uns darüber im Ausgabefenster (nicht Konsolenfenster!) informieren wird. Dadurch, dass wir die `IFormatProvider`-Schnittstelle einbinden, ermöglichen wir, sie auch als Parameter für `Tostring` der `Laengen`-Klasse zuzulassen. Auf diese Weise schaffen wir die Möglichkeit, die Schnittstelle zu übergeben, ohne uns bei der Parameterübergabe ausschließlich auf ein `LaengenInfoFormat`-Objekt festlegen zu müssen.

TIPP Wenn Sie das Kapitel über Schnittstellen (noch) nicht gelesen haben, dann beachten Sie Folgendes, um die benötigten Routinenrümpfe (Stubs) für die Schnittstelle `IFormatProvider` einzufügen, falls Sie ähnlichen Code selber erstellen: Tippen Sie **Implements IFormatProvider** und gehen Sie in die nächste Zeile mit (nicht mit den Cursortasten!). Fertig.

```

Public Function Format(ByVal formatChars As String, _
    ByVal arg As Object, _
    ByVal formatProvider As System.IFormatProvider) As String _
Dim locLaengen As Laengen

Trace.WriteLine("Format (CustomFormatter-Signatur) wurde aufgerufen!")
'Dafür sorgen, dass das zu formatierende Element und der Format Provider übereinstimmen.
If Not TypeOf arg Is Laengen Then
    Return String.Format(formatProvider, formatChars, arg)
End If

locLaengen = DirectCast(arg, Laengen)

'Dafür sorgen, dass die Formatzeichenfolge nie "nichts" ist.
If formatChars Is Nothing Then
    formatChars = ""
End If

'Mit Semikolon können Formatzeichen für die Formatierung des eigentlichen Wertes folgen.
Dim locSemikolonPos As Integer = formatChars.IndexOf(";"c)

'Standardzeichen für die Formatzeichen zur Werteformatierung vorgeben.
Dim locNumFormat As String = "G"

'Doppelpunkt gefunden.
If locSemikolonPos > -1 Then
    'Das ist die Formatzeichenfolge für die Werteformatierung
    locNumFormat = formatChars.Substring(locSemikolonPos + 1)

    'Das für die Wahl der Längeneinheit
    formatChars = formatChars.Substring(0, locSemikolonPos)

    'Leerstring kommt nicht in Frage.
    If locNumFormat = "" Then
        locNumFormat = "G"
    End If
End If

'Nur noch kein, ein oder drei Zeichen kommen in Frage.
If formatChars.Length <> 0 And formatChars.Length <> 1 And formatChars.Length <> 3 Then
    Dim up As New FormatException("Ungültige(s) Formatzeichen für die Kulturbestimmung!")
    Throw up
End If

'Wenn drei Zeichen, dann wird die Einstellung des FormatProviders ignoriert;
'das Formatzeichen ist der Bestimmen!
If formatChars.Length = 3 Then
    If formatChars.ToUpper.EndsWith("-D") Then
        formatProvider = New LaengenFormatInfo(LaengenKultur.Deutsch)
        formatChars = formatChars.Substring(0, 1)
    ElseIf formatChars.ToUpper.EndsWith("-E") Then
        formatProvider = New LaengenFormatInfo(LaengenKultur.EnglischAmerikanisch)
        formatChars = formatChars.Substring(0, 1)
    Else
        Dim up As New FormatException("Ungültiges Formatzeichen für die Kulturbestimmung!")
    End If
End If

```

```
        Throw up
    End If
End If

'Zu diesem Zeitpunkt ist formatProvider unbedingt eine LaengenformatInfo,
'Das folgende Casting kann also nicht schiefgehen:
Dim locLaengenFormatInfo As LaengenFormatInfo = DirectCast(formatProvider, LaengenFormatInfo)

'formatChars besteht aus (jetzt noch) nur einem Zeichen.

If formatChars.Length = 1 Then
    'S' für 'Sehr klein'
    If formatChars.ToUpper.StartsWith("S") Then
        locLaengenFormatInfo.Aufloesung = LaengenAufloesung.SehrKlein
    End If

    'K' für 'klein'
    If formatChars.ToUpper.StartsWith("K") Then
        locLaengenFormatInfo.Aufloesung = LaengenAufloesung.Klein
    End If

    'M' für 'Mittel'
    If formatChars.ToUpper.StartsWith("M") Then
        locLaengenFormatInfo.Aufloesung = LaengenAufloesung.Mittel
    End If

    'G' für 'groß'
    If formatChars.ToUpper.StartsWith("G") Then
        locLaengenFormatInfo.Aufloesung = LaengenAufloesung.Groß
    End If
End If

With locLaengenFormatInfo
    'Und alle Stringausgaben anhand des Providers durchführen
    If .Kultur = LaengenKultur.Deutsch Then
        If .Aufloesung = LaengenAufloesung.SehrKlein Then
            Return locLaengen.ToMillimeter.ToString(locNumFormat)
        ElseIf .Aufloesung = LaengenAufloesung.Klein Then
            Return locLaengen.ToCentimeter.ToString(locNumFormat)
        ElseIf .Aufloesung = LaengenAufloesung.Mittel Then
            Return locLaengen.ToMeter.ToString(locNumFormat)
        ElseIf .Aufloesung = LaengenAufloesung.Groß Then
            Return locLaengen.ToKilometer.ToString(locNumFormat)
        End If
    Else
        If .Aufloesung = LaengenAufloesung.SehrKlein Then
            Return locLaengen.ToLine.ToString(locNumFormat)
        ElseIf .Aufloesung = LaengenAufloesung.Klein Then
            Return locLaengen.ToInch.ToString(locNumFormat)
        ElseIf .Aufloesung = LaengenAufloesung.Mittel Then
            Return locLaengen.ToYard.ToString(locNumFormat)
        ElseIf .Aufloesung = LaengenAufloesung.Groß Then
            Return locLaengen.ToMile.ToString(locNumFormat)
        End If
    End If
End If
```

```
End With  
End Function
```

Und hier findet sie nun statt, die Aufbereitung der Zeichenkette für die formatierte Ausgabe. Sie macht unseren Format Provider erst wirklich zu einem *Format* Provider. Doch im Grunde genommen sind die knapp 100 Zeilen, in denen die Aufbereitung des Wertes und die Auswertung der Formatzeichenfolgen stattfindet, nichts Besonderes. Die eine oder andere Zeichenkettenanalyse, ein paar Bedingungsauswertungen – das war es schon.

```
Public Property Kultur() As LaengenKultur  
    Get  
        Return myKultur  
    End Get  
    Set(ByVal Value As LaengenKultur)  
        myKultur = Value  
    End Set  
End Property  
Public Property Aufloesung() As LaengenAufloesung  
    Get  
        Return myAufloesung  
    End Get  
    Set(ByVal Value As LaengenAufloesung)  
        myAufloesung = Value  
    End Set  
End Property  
End Class
```

Und damit ist das Geheimnis der Funktionsweise unserer Laengen-Klasse und ihres eigenen Format Providers auch schon gelüftet. Eine Sache fehlt allerdings noch, und die hat es in sich:

Benutzerdefinierte Format Provider durch **IFormatProvider** und **ICustomFormatter**

Was bislang kein Beispielprogramm gezeigt hat, ist ein Feature, auf das Sie bei der Formatierung von Datumswerten und numerischen Daten zurückgreifen können: die direkte Einbindung von Formatzeichenfolgen beispielsweise in `WriteLine` oder `String.Format`. Da wir im jetzigen Stand einen Format Provider implementiert haben, schauen wir, was passiert, wenn wir diesen Format Provider zusammen mit unserem Datentyp in einer solchen Kombination einsetzen:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 26\\CustomFormatProvider02
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```

Sub FormatterTest()
    'Definiert eine Laengen-Instanz mit 1 (einem Meter).
    Dim locLaenge As New Laengen(1)
    Dim locLaengenFormatInfo As New LaengenFormatInfo

    locLaengenFormatInfo.Aufloesung = LaengenAufloesung.SehrKlein
    locLaengenFormatInfo.Kultur = LaengenKultur.EnglischAmerikanisch

    'Testen der Format-Funktion.
    Dim locStr As String = String.Format(locLaengenFormatInfo, _
        "Testausgabe {0:; ##0.00} eines Laengen-Objektes", _
        locLaenge)
    Console.WriteLine(locStr)

    'Testen des Formatters.
    Console.WriteLine("Testausgabe {0:g-d; ##0.00} eines Laengen-Objektes", locLaenge)

    Console.ReadLine()

End Sub

```

Doch leider passiert nach dem Programmstart nicht das, was wir eigentlich erwarten. Die Ausgabe lautet schlicht:

```

Testausgabe CustomFormatProvider.Laengen eines Laengen-Objektes
Testausgabe CustomFormatProvider.Laengen eines Laengen-Objektes

```

Anstatt die formatierten Werte auszugeben, haben beide Zeilen lediglich den Namen des Objektes in das Konsolenfenster geschrieben. Allerdings: Im Ausgabefenster (nicht im Konsolenfenster!) erscheint ein Hinweis, der uns bei der Lösung dieses Problems einen Schritt weiter bringt.

Dort ist nämlich

```
GetFormat (IFormatProvider-Signatur) wurde aufgerufen:ICustomFormatter
```

zu lesen, und genau diese Zeile drucken wir durch die Anweisung

```
Trace.WriteLine("GetFormat (IFormatProvider-Signatur) wurde aufgerufen:" + formatType.Name)
```

in der GetFormat-Methode der LaengenFormatInfo-Klasse aus. Die Frage, die sich jetzt stellt: *Was* hat diese Funktion aufgerufen und *warum*?

Um genau zu sein: Die AppendFormat-Methode des StringBuilder-Objektes, die .NET-intern für die Aufbereitung einer zu formatierenden Zeichenfolge mithilfe eines Format Providers zuständig ist, war für den Aufruf von GetFormat verantwortlich. Sie selbst wurde über den Umweg String.Format aufgerufen und fragt uns, welcher benutzerdefinierte Format Provider die eigentliche Formatierung vornehmen soll. Da unsere Routine GetFormat z.Z. noch Nothing zurückgibt (genau genommen gibt sie gar nichts zurück – aber das entspricht ja buchstäblich *Nothing* ...), interpretiert sie das als Antwort »keiner«. Sie greift deswegen auf ein Notprogramm zurück und ruft die parameterlose ToString-Funktion des aufzubereitenden Objektes (Instanz von Laengen) auf. Wir haben diese aber nicht überschrieben, und deswegen liefert sie – da von Object abgeleitet und somit unverändert übernommen – nur den Namen der Klasse zurück, und genau das haben wir als Ausgabe im Konsolenfenster sehen können.

Nur wenn wir innerhalb von GetFormat einen für unseren Datentyp gültigen Formatter zurückliefern, wird AppendFormat diesen verwenden und anschließend dessen Format-Routine aufrufen. Unsere Aufgabe ist es also lediglich, einen Formatter zu implementieren und eine Instanz als Funktionsergebnis von GetFormat an AppendFormat zurückzuliefern. AppendFormat weiß dann, wen es zur Formatierung heranziehen soll. Die einzige sinnvolle Instanz, die GetFormat zu dieser Zeit kennt, ist aber die eigene.

Das bedeutet, dass die LaengenFormatInfo-Klasse auch die ICustomFormatter-Schnittstelle implementieren muss, damit sie zum Formatter wird und ein erlaubtes Funktionsergebnis zurückliefern kann.

Einige Handgriffe reichen, um zum Ziel zu gelangen: Am Anfang des Codes der LaengenFormatInfo-Klasse muss die Implements-Anweisung um die IFormatProvider-Schnittstelle ergänzt werden:

```
Public Class Laengen
    Implements IFormatProvider

    'Speichert die Länge in Meter.
    Private myLaenge As Decimal
    .
    .

```

Die IFormatProvider-Schnittstelle verlangt, dass eine Format-Funktion in der Klasse existiert, die die Formatierung durchführt. Sie muss als Signatur einen String mit Formatzeichen und das zu formatierende Objekt sowie ein weiteres Objekt entgegennehmen, das die IFormatProvider-Schnittstelle implementiert. Zufälligerweise⁴ haben wir genau so eine Version von Format schon im Programm. Es reicht, diese Funktion mit der Implements-Anweisung zu versehen, damit das Implementieren der IFormattable-Schnittstelle abgeschlossen ist:

```
Public Function Format(ByVal formatChars As String, _
                      ByVal arg As Object,
                      ByVal formatProvider As System.IFormatProvider) As String
    Implements ICustomFormatter.Format

    Trace.WriteLine("Format (CustomFormatter-Signatur) wurde aufgerufen!")
    'Dafür sorgen, dass das zu formatierende Element und der FormatProvider übereinstimmen
    If Not TypeOf arg Is Laengen Then
        Return String.Format(formatProvider, formatChars, arg)
    End If
    .
    .

```

Und zu guter Letzt teilen wir der AppendFormat-Methode, die GetFormat der LaengenFormatInfo aufruft, noch mit, dass sie unsere LaengenFormatInfo-Klasse als Formatter verwenden kann. Das machen wir folgendermaßen:

```
Public Function GetFormat(ByVal formatType As System.Type) As Object
    Implements System.IFormatProvider.GetFormat
    Trace.WriteLine("Ausgabe von GetFormat:" + formatType.Name)
    'Wird mein Typ verlangt?
    If formatType.Name = "ICustomFormatter" Then

```

⁴ ;-) – vielleicht nicht ganz so zufällig...

```
'Ja, diese Instanz ist erlaubt zu handeln!
Return Me
Else
  'Falscher Typ, diese Instanz darf nichts machen, denn
  'wenn sie als Provider einem nicht kompatiblen Typ
  'übergeben wird, geht's in die Hose.
  Return Nothing
End If
End Function
```

Wenn wir das Programm nun abermals starten, sieht das Ergebnis viel besser aus:

```
Testausgabe 472,40000 eines Laengen-Objektes
Testausgabe CustomFormatProvider.Laengen eines Laengen-Objektes
```

String.Format liefert jetzt brauchbare Ergebnisse – WriteLine direkt allerdings noch nicht.

Wir haben im vergangenen Abschnitt gesehen, dass AppendFormat sich richtig Mühe gibt, korrekte Formatierungen auch mit Objekten durchzuführen, die dem Framework fremd sind. Und diese Bemühungen gehen noch einen Schritt weiter. Denn bevor alle Stricke reißen und AppendFormat auf die parameterlose ToString-Funktion eines Objektes zurückgreift, sucht es automatisch nach einer weiteren Schnittstelle, die das Objekt implementieren könnte. Ihr Name: IFormattable.

Automatisch formatierbare Objekte durch Einbinden von IFormattable

Die IFormattable-Schnittstelle ist in ihrer Handhabung eigentlich sogar noch einfacher als die FormatProvider-Methode – leider auch nicht ganz so flexibel, denn sie muss auf eine korrelierende FormatProvider-Klasse beim Aufbereiten des Strings verzichten und sich ganz auf Formatzeichenfolgen verlassen.

Für unser Beispiel ist der Aufwand umso geringer, als wir eine Formatierungs-Engine, die Formatzeichen auswerten kann, von vornherein implementiert haben.

Damit AppendFormat, das in letzter Instanz auch von Console.WriteLine für das Zusammenbauen einer Parameterzeichenfolge (»text {0} text {1} ...«) verwendet wird, auf diese Formatierungs-Engine unserer Beispielklasse zurückgreift, brauchen wir lediglich die IFormattable-Schnittstelle in unsere Laengen-Klasse einzubinden:

```
Public Class Laengen
  Implements IFormattable

  'Speichert die Länge in Meter.
  Private myLaenge As Decimal

  Sub New(ByVal Meter As Decimal)
    myLaenge = Meter
  End Sub
  .
  .
  .
```

IFormattable verlangt, dass es eine ToString-Funktion mit entsprechender Signatur in der sie einbindenden Klasse gibt. Die Signatur verlangt, dass eine Formatzeichenfolge und ein IFormatProvider übergeben werden können. Aber auch eine solche ToString-Version gibt es in unserer Klasse bereits:

```
Public Overloads Function ToString(ByVal formatChars As String,
    ByVal formatProvider As System.IFormatProvider) As String Implements
IFormattable.ToString

    Trace.WriteLine("ToString (Formattable-Signatur) wurde aufgerufen!")

    If (TypeOf formatProvider Is CultureInfo) Or formatProvider Is Nothing Then
        .
        .
        .
    End If
```

Wir waren gut vorbereitet, was? Denn das war es auch schon. Wenn Sie die Formatierungsprozedur

```
Sub FormatterTest()

    'Definiert eine Laengen-Instanz mit 1 (einem Meter).
    Dim locLaenge As New Laengen(1)
    Dim locLaengenFormatInfo As New LaengenFormatInfo

    locLaengenFormatInfo.Aufloesung = LaengenAufloesung.SehrKlein
    locLaengenFormatInfo.Kultur = LaengenKultur.EnglischAmerikanisch

    'Testen der Format-Funktion.
    Dim locStr As String = String.Format(locLaengenFormatInfo,
        "Testausgabe {0:; ########.0.00} eines Laengen-Objektes", _
        locLaenge)
    Console.WriteLine(locStr)
    'Testen des Formatters.
    Console.WriteLine("Testausgabe {0:g-d; ########.0.00} eines Laengen-Objektes", locLaenge)

    Console.ReadLine()
End Sub
```

anschließend abermals starten, sehen Sie folgendes Ergebnis auf dem Bildschirm:

```
Testausgabe 472,40000 eines Laengen-Objektes
Testausgabe ,00100 eines Laengen-Objektes
```

Sie sehen: Sowohl String.Format mit Unterstützung eines eigenen Format Providers als auch WriteLine funktionieren jetzt perfekt!

Kapitel 27

Reguläre Ausdrücke (Regular Expressions)

In diesem Kapitel:

RegExperimente mit dem RegExplorer	780
Erste Gehversuche mit Regulären Ausdrücken	782
Programmieren von Regulären Ausdrücken	796
Regex am Beispiel: Berechnen beliebiger mathematischer Ausdrücke	800

Reguläre Ausdrücke (*Regular Expressions*) sind eine mächtige Erweiterung der String-Verarbeitung in Visual Basic (eigentlich im Framework). Ungewöhnlich ist auch die Geschichte, die sich dahinter verbirgt, nämlich wie Reguläre Ausdrücke ihren Weg in das Framework gefunden haben. Sie müssen wissen: Die verschiedenen Themengebiete innerhalb des Frameworks werden von eigenen, sehr unabhängigen Entwicklungsteams bei Microsoft entwickelt. Reguläre Ausdrücke waren ursprünglich »nur« eine Erweiterung bzw. ein Werkzeug, die bzw. das im ASP.NET-Team gebraucht wurde. Erst in teamübergreifenden Meetings erkannten auch andere Teams das Vorhandensein von Regulären Ausdrücken, und nun begann ein Tauziehen darum, in welchem Namensbereich die Regex-Klasse, mit der Sie Reguläre Ausdrücke anwenden, letzten Endes ihr zu Hause fand.

Das Ergebnis kennen Sie: Sie finden die Regex-Klasse im Bereich `System.Text.RegularExpressions`. Das bedeutet: Sie müssen die Anweisung

```
Imports System.Text.RegularExpressions
```

an den Anfang einer Klassen-Quellcodedatei setzen, damit Sie auf die Klassen zugreifen können.

Die große Frage, die sich vielen stellt: Was genau sind Reguläre Ausdrücke? Die Wurzeln von Regulären Ausdrücken gehen zurück auf die Arbeiten eines gewissen Stephen Kleene. Stephen Kleene war ein amerikanischer Mathematiker und darüber hinaus einer derjenigen, die die Entwicklung der theoretische Informatik maßgeblich beeinflusst und vorangetrieben haben. Er erfand eine Schreibweise für die, wie er sie nannte, »Algebra regelmäßiger Mengen«. Im Kontext von Suchaufgaben mit dem Computer war das »*«-Zeichen deshalb bis vor einiger Zeit auch unter dem Namen »Kleene-Star« bekannt.

Und damit sind wir auch schon beim Thema, denn das »*«-Zeichen als *Joker* oder *Wildcard* hat jeder von Ihnen sicherlich schon einmal unter DOS, zumindest aber in der Konsole verwendet. Wenn Sie in der Konsole beispielsweise alle Dateien anzeigen lassen möchten, die mit ».TXT« enden, geben Sie den Befehl

```
dir *.txt
```

ein. Sie können also bestimmte Sonderzeichen verwenden, um Zeichenfolgen zu finden, welche Regeln unterliegen, die von diesen Sonderzeichen definiert werden. Genau dazu dienen Reguläre Ausdrücke. Dummerweise ist das Demonstrieren von Regulären Ausdrücken mit simplen Konsolen-Anwendungen nicht sehr anschaulich, vor allen Dingen auch recht mühsam. Denn bei aller Leistungsfähigkeit von Regulären Ausdrücken haben diese doch einen Nachteil: Sie sind vergleichsweise schwer zu lesen. Wenn Sie sich an die Zusammensetzung von Regulären Ausdrücken gewöhnt haben, dann wird Ihnen beim zeichenweisen Analysieren klar, wieso die Zeichenfolge

```
[\d, .] + [S]*
```

alle Zahlenkonstanten in einer Formel finden kann. Doch wenn Sie sie anschauen, werden Sie auch mit einiger Übung nicht direkt auf den ersten Blick ihre Funktionsweise durchschauen.

RegExperimente mit dem RegExplorer

Aus diesem Grund finden Sie in den Begleitdateien (respektive im entsprechenden Verzeichnis Ihrer Festplatte) ein Projekt, mit dem Sie Reguläre Ausdrücke testen können.

BEGLEITDATEIEN

Sie finden dieses Projekt unter dem Namen *RegExplorer* im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\D - Datenstrukturen\\Kapitel 27\\RegExplorer

Wenn Sie dieses Projekt laden und starten, sehen Sie einen Dialog, wie auch in Abbildung 27.1 zu sehen.

Kurz zur Beschreibung: Nach dem Start des Programms können Sie Texte unter *Quelltext* erfassen, oder Sie können einen Text mit *Datei/Quelltextdatei laden* aus einer beliebigen Datei laden und in der Textbox *Quelltext* anzeigen lassen.

Unter *Suchmuster* können Sie einen Regulären Ausdruck eingeben; ein Mausklick auf die Schaltfläche *Suchen* löst dann die Suche mit dem angegebenen Suchbegriff aus.

Die verschiedenen Suchergebnisse zeigt Ihnen das TreeView-Steuerelement, das Sie in der linken, unteren Ecke des Programmfensters sehen. Ein Klick auf den Wurzeleintrag bringt die komplette Datei in das rechts daneben stehende Ergebnisfenster.

Ein Mausklick auf einen der untergeordneten Zweige (nur 2. Ebene) zeigt Ihnen Informationen über den gefundenen Begriff an. Gleichzeitig wird der Suchbegriff im Quelltext markiert.

Der RegExplorer hat eine Bibliothek mit häufig verwendeten Regulären Ausdrücken. Sie können einen Ausdruck aus der Bibliothek auswählen, indem Sie auf die Schaltfläche mit der Aufschrift »...« klicken, die sich neben der Schaltfläche *Suchen* befindet.

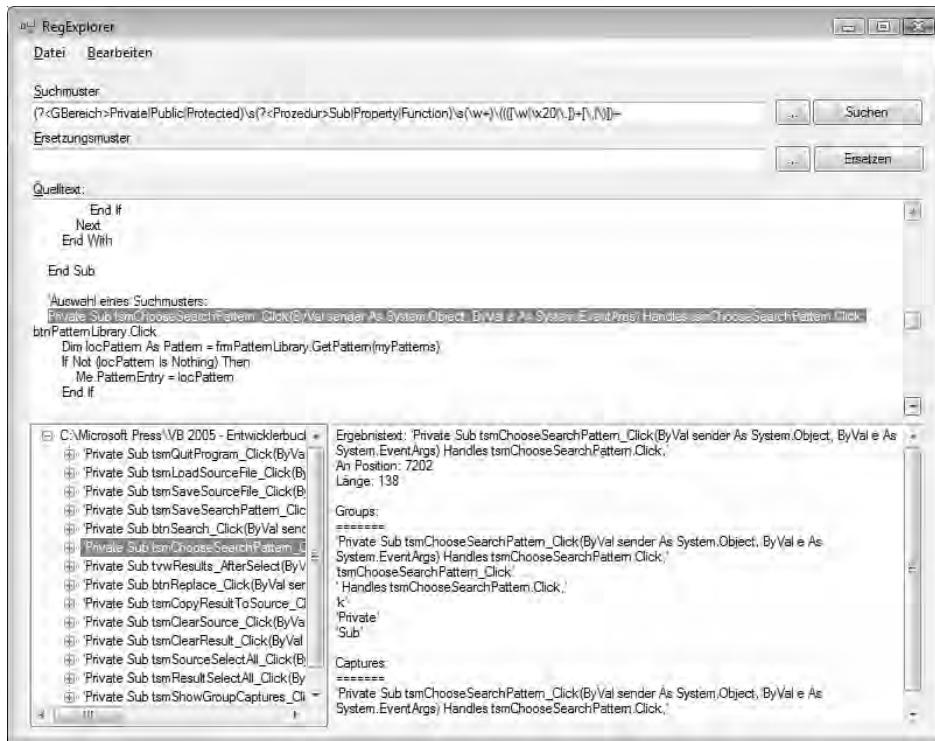


Abbildung 27.1 Mit dem RegExplorer können Sie sich mit Regulären Ausdrücken nach Herzenslust austoben, ohne eine Zeile Code schreiben zu müssen!

Möchten Sie einen neuen Bibliothekseintrag anlegen, klicken Sie auf die darunter stehende Schaltfläche, oder wählen Sie aus dem Menü *Datei* den Menüpunkt *Suchmuster speichern*. Der RegExplorer zeigt Ihnen einen weiteren Dialog an, mit dem Sie den neuen Bibliothekseintrag erfassen können (siehe Abbildung 27.2).

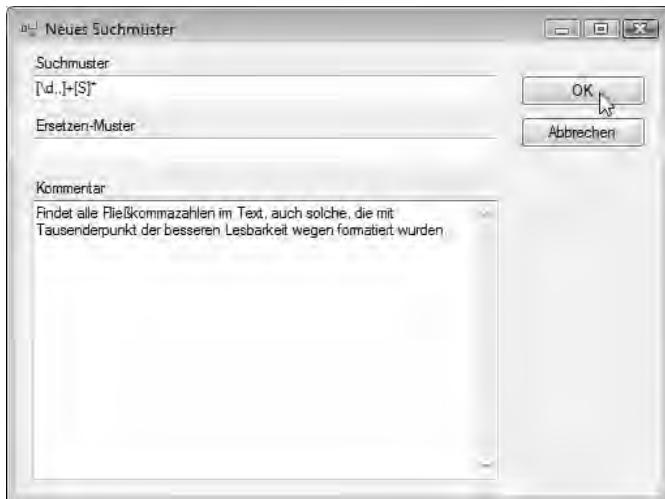


Abbildung 27.2 Mit diesem Dialog fügen Sie Reguläre Ausdrücke zur Bibliothek hinzu. Dabei müssen mindestens Suchmuster und Kommentar angegeben sein

Klicken Sie auf *Speichern*, um den neuen Eintrag in die Bibliothek aufzunehmen.

Sie können das Programmfenster übrigens nach Belieben in der Größe anpassen und auch die verschiedenen Bereiche innerhalb des Fensters mit den SplitterContainer-Komponenten sowohl horizontal (oberer Parameter- und unterer Ergebnisbereich) als auch vertikal (Verhältnis zwischen linker, unterer ErgebnisTreeView und rechter, unterer Ergebnis-TextBox) verändern.

Erste Gehversuche mit Regulären Ausdrücken

Für die ersten Gehversuche laden Sie am besten den Quellcode des Programms selbst in den Quelltextbereich, um damit experimentieren zu können. Wählen Sie dazu *Quelltext laden* aus dem Menü *Datei*. Im Dateiauswahl-dialog wählen Sie anschließend den Dateityp *VB-Quelldateien*. Öffnen Sie anschließend die Datei *frmMain.vb*.

Einfache Suchvorgänge

Zunächst einmal können Sie einfache Zeichenfolgen verwenden, wenn Sie eine exakte Entsprechung für diese im Quelltext finden wollen. Geben Sie beispielsweise

Imports

als Suchbegriff ein und klicken anschließend auf die Schaltfläche *Suchen*, finden Sie in der darunter stehenden Ergebnisliste ausschließlich die Entsprechungen dieses Worts. Bis hierhin ist die Suchfunktion noch nichts Besonderes. Reguläre Ausdrücke werden erst dann interessant, wenn mit Steuerzeichen bestimmte Funktionen beim Suchen (und später auch beim Ersetzen) mit einbezogen werden können.

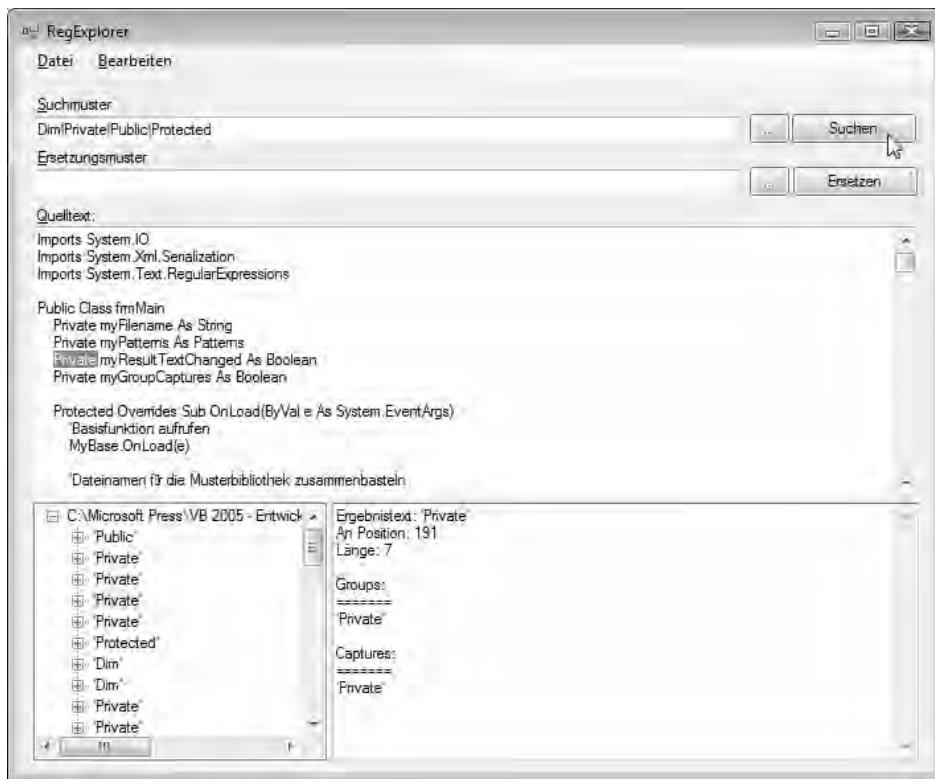


Abbildung 27.3 Wenn Sie nach alternativen Begriffen suchen, trennen Sie durch das »|«-Zeichen. Ein Klick auf den Begriff in der linken, unteren Ergebnisliste markiert übrigens das Wort im Quelltext.

Dazu ein simples Beispiel. Mithilfe des Oder-Operatorzeichens (»|«) können Sie aus einer Alternative von Zeichenketten Treffer generieren. Suchen Sie beispielsweise nach Wörtern, die wahlweise *Dim*, *Private*, *Protected* oder *Public* heißen sollen, dann formulieren Sie den Suchbegriff folgenderweise:

Dim|Private|Protected|Public

Wenn Sie anschließend auf *Suchen* klicken, sehen Sie ein Ergebnis, etwa wie es Abbildung 27.3 zeigt.

Einfache Suche nach Sonderzeichen

Nicht alle Zeichen lassen sich über die Tastatur eingeben. Möchten Sie beispielsweise nach doppelten Absätzen suchen, bekommen Sie bei der Eingabe über die Tastatur schon Probleme.

Escape-Zeichen	Beschreibung
Normale Zeichen	Andere Zeichen als \$ ^ { [() * + ? \ stehen für sich selbst.
\a	Entspricht einem Klingelzeichen (Warnsignal) \u0007 (Bell).

Escape-Zeichen	Beschreibung
\b	Entspricht in einer []-Zeichenklasse einem Rücktastenzeichen \u0008 (Backspace). Wichtig: Das Escape-Zeichen \b ist ein Sonderfall: In einem Regulären Ausdruck markiert \b eine Wortbegrenzung (zwischen \w und \W-Zeichen), ausgenommen innerhalb einer []-Zeichenklasse, bei der \b das Rücktastenzeichen darstellt. In einem Ersetzungsmuster kennzeichnet \b immer ein Rücktastenzeichen.
\t	Entspricht einem Tabulator \u0009.
\r	Entspricht dem Wagenrücklaufzeichen \u000D (Carriage Return).
\v	Entspricht dem vertikalen Tabstopnzeichen \u000B, das aber in der Windows-Welt in der Regel keine Anwendung findet.
\f	Entspricht einem Seitenwechselzeichen \u000C (Form Feed).
\n	Entspricht einem Zeilenvorschub \u000A (Line Feed).
\e	Entspricht einem Escape-Zeichen \u001B.
\o40	Entspricht einem beliebigen ASCII-Zeichen, das durch eine Oktalzahl (bis zu drei Stellen) repräsentiert wird. Zahlen ohne voranstehende Null sind Rückverweise, wenn sie nur eine Ziffer enthalten oder einer Aufzeichnungsgruppennummer entsprechen. Beispielsweise stellt das Zeichen \o40 ein Leerzeichen dar.
\x20	Entspricht einem ASCII-Zeichen in hexadezimaler Darstellung (genau zwei Stellen).
\cC	Entspricht einem ASCII-Steuerzeichen. Beispiel: \cC ist Control-C.
\u0020	Entspricht einem Unicode-Zeichen in hexadezimaler Darstellung (genau vier Stellen).
\	Wird dieses Zeichen von einem Zeichen gefolgt, das nicht als Escape-Zeichen erkannt wird, entspricht es diesem Zeichen. So stellt \. beispielsweise den Punkt oder \\ den Backslash dar.

Tabelle 27.1 Gültige Sonderzeichen für Reguläre Ausdrücke

Mit Sonderzeichen, die Sie der Tabelle entnehmen, können Sie das Problem recht simpel lösen:

```
\r\n\r\n
```

Diese Sonderzeichen weisen die *Regular Expressions Engine* an, nach dem fortlaufenden Vorkommen von *Carriage Return*, *Line Feed*, *Carriage Return* und *LineFeed* zu suchen – und diese Folge entspricht zwei aufeinander folgenden Absätzen.

Komplexere Suche mit speziellen Steuerzeichen

Noch flexibler können Sie Ihre Suchvorgänge gestalten, wenn Sie von Steuerzeichen Gebrauch machen, wie sie die folgende Tabelle darstellt:

Zeichenklasse	Beschreibung
.	Entspricht allen Zeichen mit Ausnahme von \n. Bei Modifikation durch die Singleline-Option entspricht ein Punkt einem beliebigen Zeichen. Weitere Informationen hierzu finden Sie unter Optionen für Reguläre Ausdrücke.
[aeiou]	Entspricht einem beliebigen einzelnen Zeichen, das in dem angegebenen Satz von Zeichen enthalten ist.
[^aeiou]	Entspricht einem beliebigen einzelnen Zeichen, das nicht in dem angegebenen Satz von Zeichen enthalten ist. ►

Zeichenklasse	Beschreibung
[0-9a-fA-F]	Durch die Verwendung eines Bindestrichs (-) können aneinander grenzende Zeichenbereiche angegeben werden.
\p{name}	Entspricht einem beliebigen Zeichen in der durch {name} angegebenen benannten Zeichenklasse. Unterstützte Namen sind Unicodegruppen und Blockbereiche. Beispielsweise Ll, Nd, Z, IsGreek, IsBoxDrawing.
\P{name}	Entspricht Text, der nicht in Gruppen und Blockbereichen enthalten ist, die in {name} angegeben werden.
\w	Entspricht einem beliebigen Wortzeichen. Entspricht den Unicode-Zeichenkategorien [\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]. Wenn mit der ECMAScript-Option ECMAScript-konformes Verhalten angegeben wurde, ist \w gleichbedeutend mit [a-zA-Z_0-9].
\W	Entspricht einem beliebigen Nichtwortzeichen. Entspricht den Unicodekategorien [^\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]. Wenn mit der ECMAScript-Option ECMAScript-konformes Verhalten angegeben wurde, ist \W gleichbedeutend mit [^a-zA-Z_0-9].
\s	Entspricht einem beliebigen Leerraumzeichen. Entspricht den Unicode-Zeichenkategorien [\f\n\r\t\v\x85\p{Z}]. Wenn mit der ECMAScript-Option ECMAScript-konformes Verhalten angegeben wurde, ist \s gleichbedeutend mit [\f\n\r\t\v].
\S	Entspricht einem beliebigen Nicht-Leerraumzeichen. Entspricht den Unicode-Zeichenkategorien [^\f\n\r\t\v\x85\p{Z}]. Wenn mit der ECMAScript-Option ECMAScript-konformes Verhalten angegeben wurde, ist \S gleichbedeutend mit [^ \f\n\r\t\v].
\d	Entspricht einer beliebigen Dezimalziffer. Gleichbedeutend mit \p{Nd} für Unicode und [0-9] für Nicht-Unicode mit ECMAScript-Verhalten.
\D	Entspricht einer beliebigen Nichtziffer. Gleichbedeutend mit \P{Nd} für Unicode und [^0-9] für Nicht-Unicode mit ECMAScript-Verhalten.
^	Bestimmt, dass der Vergleich am Anfang der Zeichenfolge oder der Zeile erfolgen muss.
\$	Bestimmt, dass der Vergleich am Ende der Zeichenfolge, vor einem \n am Ende der Zeichenfolge oder am Ende der Zeile erfolgen muss.
\A	Bestimmt, dass der Vergleich am Anfang der Zeichenfolge erfolgen muss (die Multiline-Option wird ignoriert).
\Z	Bestimmt, dass der Vergleich am Ende der Zeichenfolge oder vor einem \n am Ende der Zeichenfolge erfolgen muss (die Multiline-Option wird ignoriert).
\z	Bestimmt, dass der Vergleich am Ende der Zeichenfolge erfolgen muss (die Multiline-Option wird ignoriert).
\G	Bestimmt, dass der Vergleich an dem Punkt erfolgen muss, an dem der vorherige Vergleich beendet wurde. Beim Verwenden mit Match.NextMatch() wird sichergestellt, dass alle Übereinstimmungen aneinander grenzend sind.
\b	Bestimmt, dass der Vergleich an einer Begrenzung zwischen \w (alphanumerischen) und \W (nicht alphanumerischen) Zeichen erfolgen muss. Der Vergleich muss bei Wortbegrenzungen erfolgen, d. h. beim ersten oder letzten Zeichen von Wörtern, die durch beliebige nicht alphanumerische Zeichen voneinander getrennt sind.
\B	Bestimmt, dass der Vergleich nicht bei einer \b-Begrenzung erfolgen darf.

Tabelle 27.2 Steuer- bzw. Befehlszeichen für Reguläre Ausdrücke

Angenommen, Sie möchten alle Begriffe finden, die mit der Zeichenfolge »Me.« beginnen und anschließend vier Buchstaben aufweisen, dann wäre, wenn Sie nach der oben stehenden Tabelle logisch vorgingen, die Suchzeichenfolge

```
Me\.\w\w\w\w
```

auf den ersten Blick die richtige Vorgehensweise. Aber die Ergebnisliste entspricht wahrscheinlich mit dem Ergebnis, wie es auch in Abbildung 27.4 zu sehen ist, nicht dem, was Sie im Hinterkopf hatten.



Abbildung 27.4 Beim ersten Versuch entspricht die Ergebnisliste manchmal nicht dem, was Sie sich vorgestellt haben

Ihre Vorstellung ist es eher gewesen, dass das Wort nach den vier Buchstaben auch zu Ende ist. Aber auch diese Vorstellung müssen Sie dem Computer mitteilen.

Ergänzen Sie die Suchabfrage um das Steuerzeichen »\s«, und verwenden damit den Suchbegriff

```
Me\.\w\w\w\w\s
```

entspricht das Ergebnis vermutlich schon eher Ihren Vorstellungen.

Verwendung von Quantifizierern

Nun ist es vermutlich nicht gerade praxisnah, nach Begriffen zu suchen, deren Zeichenanzahl Sie vorher schon kennen. Oder, um beim vorherigen Beispiel zu bleiben: Sie möchten schon eher nach einem Wort suchen, dass mit »Me.« anfängt, dessen Buchstabenanzahl Sie aber nicht wissen. Sie möchten also der

Such-Engine mitteilen, dass sie im Anschluss an »Me.« nach mindestens einem beliebigen weiteren Zeichen suchen soll, das unter die Kategorie »\w« fällt. Die Lösung zu diesem Problem sind die so genannten Quantifizierer, die Sie in der folgenden Tabelle aufgelistet finden:

Quantifizierer	Funktion
*	Setzt keine oder mehr Übereinstimmungen voraus. Beispiel: Entsprechendes Vorhandensein vorausgesetzt, findet »Me.\w*« sowohl die Zeichenfolge »Me.« als auch »Me.Close«. Dieser Quantifizierer ist gleichbedeutend mit {0,}.
+	Setzt eine oder mehr Übereinstimmungen voraus. Beispiel: »Me.\w+« findet sowohl die Zeichenfolge »Me.Close« als auch »Me.Panel1« aber nicht »Me.« oder »Mehl«.
?	Setzt keine oder eine Übereinstimmung voraus.
{n}	Setzt exakt n Übereinstimmungen voraus. Beispiel: »(\w+.){2}« findet in dem String Me.components = New System.ComponentModel.Container Me.Splitter2 = New System.Windows.Forms.Splitter die Begriffe »System.ComponentModel.« und »System.Windows.«
{n,}	Setzt mindestens n Übereinstimmungen voraus.
{n,m}	Setzt mindestens n, jedoch höchstens m Übereinstimmungen voraus.
*?	Setzt die erste Übereinstimmung voraus, die so wenige Wiederholungen wie möglich verwendet. Dieser Befehl wird auch »faules *« (lazy *) genannt.
+?	»Faules +«: Setzt so wenige Wiederholungen wie möglich voraus, jedoch mindestens eine.
??	»Faules ?«: Setzt keine Wiederholungen voraus, falls möglich, oder eine Wiederholung.
{n}?	Gleichbedeutend mit {n}.
{n,}?	Setzt so wenige Wiederholungen wie möglich voraus, jedoch mindestens n Wiederholungen.
{n,m}?	Setzt so wenige Wiederholungen wie möglich zwischen n und m voraus.

Tabelle 27.3 Mit diesen Quantifizierern steuern Sie Anweisungen für Zeichenwiederholungen

Sie können als Suchbegriff beispielsweise

Me\.\w+

eingeben, um zum gewünschten Ziel zu gelangen. Warum? Analysieren wir den Suchbegriff Zeichen für Zeichen. »Me« bestimmt zunächst die ersten beiden Zeichen des Präfixes der gesuchten Begriffe. Den Punkt können wir nicht im Klartext schreiben, da er selbst ein Steuerzeichen darstellt. Damit wird der vorangestellte Backslash nötig, um die Such-Engine den Punkt als bloßes Suchzeichen betrachten zu lassen. Mit »\w« teilen wir der Engine anschließend mit, dass wir nach einem weiteren Buchstabenzeichen suchen. Das Plus schließlich erweitert die Anweisung: Die Engine sucht damit nicht nach einem Buchstabenzeichen sondern nach mindestens einem Buchstabenzeichen. Das Ergebnis lässt nicht lange auf sich warten. Klicken Sie nach Eingabe dieses Suchstrings auf die Schaltfläche *Suchen*, sehen Sie ein Ergebnis, etwa wie in Abbildung 27.5 zu sehen.



Abbildung 27.5 Dieser Suchbegriff ist geeignet, mit den Quantifizierern herumzuexperimentieren

Verändern Sie den Suchbegriff beispielsweise in

Me\.\w*

dann zählt auch »kein Zeichen« als Kriterium für die Begriffserkennung. Das mag zunächst verwirrend sein, denn wie kann »kein Zeichen« als Kriterium gelten?

Wenn Sie alle Zeichenfolgen finden wollen, die mit »Me.« beginnen und weitere Zeichen haben, die aber auch nur aus »Me.« selbst bestehen dürfen, müssen Sie der Such-Engine mitteilen, dass nach dem Suchbegriff Zeichen folgen dürfen, aber nicht müssen. Und das Mitteilen des »nicht müssen« entspricht dem Kriterium »kein Zeichen«.

Gruppen

Gruppen bilden Sie, wenn Sie einen großen Suchbegriff in mehrere kleine Gruppen unterteilen möchten und die Ergebnisse der kleinen Gruppen auch einzeln abrufen wollen. Sie verwenden zur Gruppenbildung runde Klammern. Dazu ein Beispiel:

Angenommen, Sie möchten den Quelltext eines Programms nach allen Prozeduren durchsuchen lassen, ganz gleich ob es sich um *Properties*, *Subs* oder *Functions* handelt. Sie suchen aber nach bestimmten, nämlich solchen die entweder als *Private*, *Public* oder *Protected* deklariert sind. Und: Sie möchten ohne große Umschweife die Ergebnisse der beiden Gruppen wissen, nämlich um welchen Gültigkeitsbereich es sich handelt und was Sie deklariert haben.

Dazu entwickeln Sie einen Suchbegriff, der die Entscheidungskriterien für den Gültigkeitsbereich in der ersten Gruppe

(Private|Public|Protected)

ein Trennzeichen dazwischen und die Prozedurart mit

(Sub|Property|Function)

in der zweiten Gruppe enthält. Fügen Sie diese einzelnen Komponenten zu einem Gesamtsuchbegriff zusammen, etwa mit

(Private|Public|Protected)\s(Sub|Property|Function)

dann erhalten Sie, angewendet auf den Beispiel-Quellcode, ein Ergebnis, etwa wie in Abbildung 27.6 zu sehen.

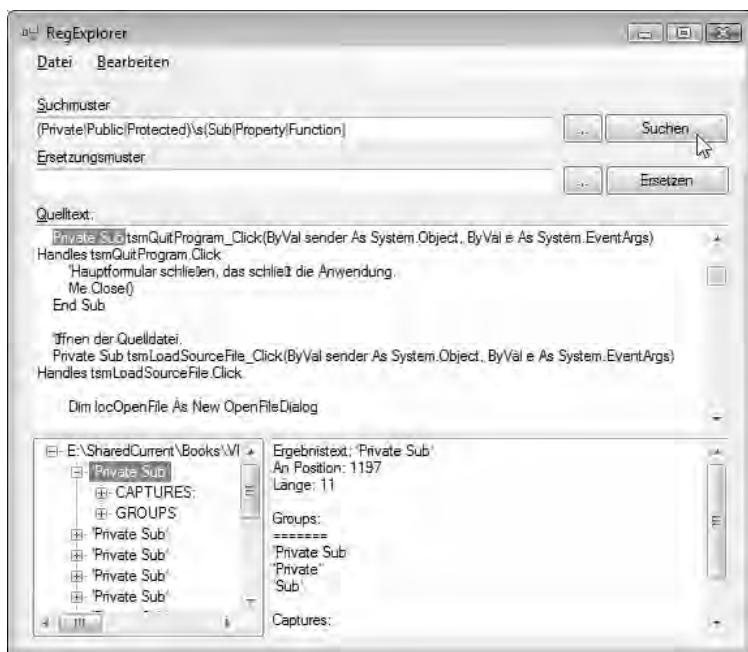


Abbildung 27.6 Durch das Gruppieren von Suchbegriffen können Sie auch programmtechnisch elegant auf Teilergebnisse zugreifen

Das Tolle daran: Sie haben mit Gruppen Zugriff auf – in diesem Beispiel – alle drei Gruppen. Drei? Genau. Mit einer (der ersten) Gruppe arbeiten Sie grundsätzlich – denn sobald Sie einen Suchbegriff eingeben, erstellen Sie bereits die erste Gruppe. Das gesamte Suchergebnis entspricht deswegen auch immer dem Ergebnis der 1. Gruppe. In diesem Beispiel haben wir explizit zwei weitere Gruppen definiert – die entsprechenden Ergebnisse spiegeln sich in Gruppe 2 und 3 wider, was Sie in der Abbildung auch sehr schön erkennen können.

Gruppen eignen sich nicht nur dazu, beim Programmieren mit Regulären Ausdrücken elegant auf Teilergebnisse zugreifen zu können. Insbesondere beim Ersetzen von Begriffen leisten sie hervorragende Dienste. Durch Gruppen, die Sie im Übrigen auch benennen können, lassen sich Ersetzungen individualisieren. Dazu benötigen Sie allerdings weitere Informationen über Steuerzeichen, die Sie im *Ersetzen*-Ausdruck für Reguläre Ausdrücke verwenden können.

Suchen und Ersetzen

Diese speziellen Steuerzeichen können Sie ausschließlich beim Ersetzen einsetzen. Die nachstehende Tabelle zeigt, welche Steuerzeichen die Regular-Expression-Engine für das Ersetzen von Ausdrücken versteht:

Zeichen	Beschreibung
\$number	Ersetzt die letzte untergeordnete Zeichenfolge, die der Gruppennummer number (dezimal) entspricht.
\${name}	Ersetzt die letzte untergeordnete Zeichenfolge, die einer (?<name>)-Gruppe entspricht.
\$\$	Ersetzt ein einzelnes "\$"-Literal.
\$&	Ersetzt eine Kopie der gesamten Entsprechung.
\$`	Ersetzt den gesamten Text der Eingabezeichenfolge vor der Entsprechung.
\$'	Ersetzt den gesamten Text der Eingabezeichenfolge nach der Entsprechung.
\$+	Ersetzt die zuletzt erfasste Gruppe.
\$_	Ersetzt die gesamte Eingabezeichenfolge.

Tabelle 27.4 Ersetzungs-Steuerzeichen für Reguläre Ausdrücke

Um beim vorhandenen Beispiel zu bleiben: Wenn Sie alle Prozeduren, ganz gleich, wie Sie sie zuvor definiert haben, durch den Gültigkeitsbereichsbezeichner *Private* ersetzen wollen, verwenden Sie als Suchbegriff den bereits bekannten

```
(Private|Public|Protected)\s(Sub|Property|Function)
```

und als Ersetzungsbegriff folgenden:

```
Private $2
```

Mit »\$2« greifen Sie auf das Ergebnis der zweiten Gruppe zu – in diesem Beispiel die Prozedurenart, denn sie ist an zweiter Stelle definiert worden. Das Ergebnis der ersten Gruppe interessiert Sie nicht, denn Sie ersetzen es ohnehin durch die Zeichenfolge »Private«.

Das gleiche Beispiel mit benannten Gruppen sähe folgendermaßen aus:

```
(?<GBereich>Private|Public|Protected)\s(?<Prozedur>Sub|Property|Function)
```

würden Sie hierbei als Suchstring und

```
Private ${Prozedur}
```

als Ersetzungszeichenfolge verwenden.

Nun könnten wir dieses Beispiel noch weiter spinnen und eine Ersetzungsroutine entwickeln, die den vormals vorhandenen Gültigkeitsbereich als Kommentar hinter die Definition setzt. Dazu müssen wir den Suchbegriff um eine weitere Gruppe erweitern, die den Rest der Zeile als insgesamt zu findende Zeichenfolge mit einschließt, etwa folgendermaßen:

Die Suchzeichenfolge:

```
(?<GBereich> Public|Protected)\s(?<Prozedur>Sub|Property|Function)(?<Rest>[^r^v^n]*)
```

Die Ersetzenzeichenfolge:

```
Private ${Prozedur}${Rest} ' Vormals: ${GBereich}
```

Und das Ergebnis sehen Sie in Abbildung 27.7.

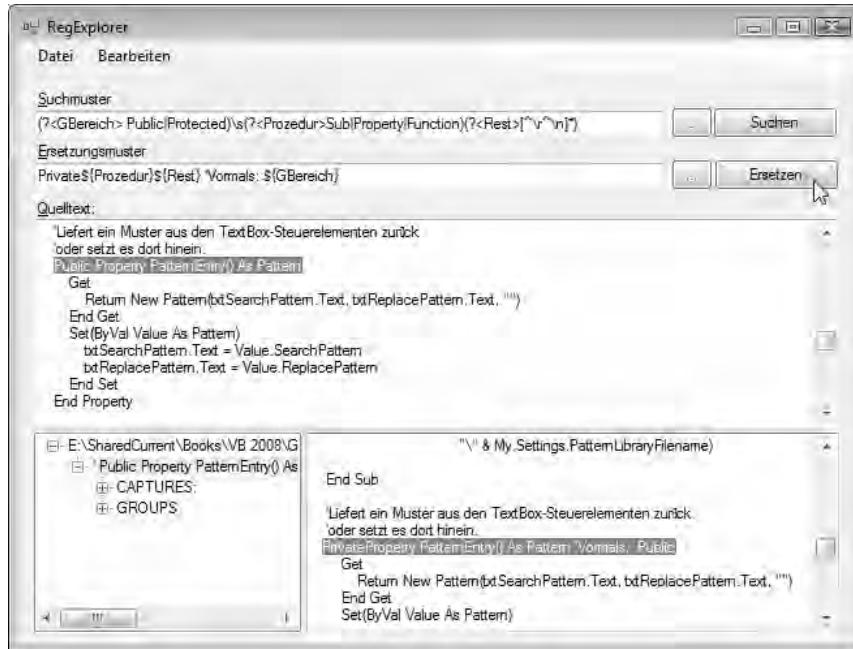


Abbildung 27.7 Das Ersetzen von Text mit Regulären Ausdrücken macht Programmieren fast überflüssig!

Captures

Captures sind dann interessant, wenn Quantifizierer bei Gruppenoperationen ins Spiel kommen. Um auch hier wieder beim Beispiel zu bleiben: Sie möchten wissen, aus welchen Variablen – so vorhanden – die einzelnen Prozedurenaufrufe bestehen. In diesem Fall legen Sie eine Regel an, die die einzelnen Variablen erfassen kann – und zwar so, dass ein Quantifizierer auf den gesamten Ausdruck anwendbar wird. Zum Beispiel:

```
\(((\w|\x20|\.)+[\,|\\])+
```

Schauen wir uns den Ausdruck einmal genauer an. Erste Regel: »\(`) – er muss mit einer Klammer beginnen. Dann beginnt der eigentliche zu wiederholende Ausdruck:

```
((\w|\x20|\.)+[\,|\\])+
```

Dieser besteht wiederum aus zwei Ausdrücken, nämlich

```
(\w|\x20|\.)+
```

und

```
[\,|\\]
```

Ausdruck Nummer eins legt alle nach der Klammer vorkommenden Zeichen so fest, dass sie aus Buchstaben, Leerzeichen oder dem Punkt bestehen dürfen. Das anschließende Quantifizierungszeichen »+« definiert, dass diese Zeichen sich beliebig wiederholen dürfen, aber mindestens einmal vorhanden sein müssen.

Der zweite Ausdruck regelt das Ende eines Parameterblocks, der mit einem Komma oder einer schließenden Klammer enden kann. Beide Ausdrücke zusammengefügt ergeben die komplette Parameterregel innerhalb der Klammer. Und jetzt kommt der Trick: Da wir nicht wissen, wie viele Parameter innerhalb eines Prozedurenprototypen definiert sind, setzen wir den »+«-Quantifizierer ans Ende, und schon kann uns egal sein, wie viele Parameter folgen – der Quantifizierer sorgt dafür, dass entsprechend viele gefunden werden.

Anschließend schnappen wir uns den Suchstring aus dem vorherigen Beispiel und modifizieren ihn so, dass eine weitere Gruppe für den Funktionsnamen gefunden werden kann. Das ist vergleichsweise einfach: Lediglich der Ausdruck

```
\s(\w+)
```

muss noch eingefügt werden. Das »\s« regelt die Trennung zwischen Prozedurentypnamen und Funktionsnamen; die Gruppe »(\w+)« deckt den Funktionsnamen ab. Packen wir alles zusammen, erhalten wir folgenden Gesamtsuchstring:

```
(?<GBereich>Private|Public|Protected)\s(?<Prozedur>Sub|Property|Function)\s(\w+)\(((\w|\x20|\.)+[\,|\\])+|
```

Eindrucksvoll, oder nicht? Ganz ehrlich: So einfach, wie ich diese Beschreibung hier herunter geschrieben habe, war das Austüfteln dieses Strings nicht – es hat mich immerhin drei Stunden gekostet. Nichtsdestotrotz hat sich der Aufwand gelohnt, denn: Spätestens an dieser Stelle werden Sie den Nutzen der *Captures* kennen lernen.

Um die *Captures* der einzelnen Gruppen auch sichtbar zu machen, wählen Sie aus dem Menü *Datei* die Option *GroupCaptures anzeigen*.

Wenn Sie diesen String auf die zuvor geladene Visual Basic-Datei anwenden (keine Angst, Sie können diese Mammutkonstruktion aus der Bibliothek holen), können Sie sich die *Captures* der einzelnen Gruppen ebenfalls in der Ergebnisliste betrachten – etwa wie in Abbildung 27.8 zu sehen.

Da wir die Parameterfindung so allgemeingültig gehalten haben, dass wir einen Quantifizierer einsetzen konnten, kommen wir über die Captures an jede einzelne Zeichenfolge, die durch die Quantifizierer in Kombination mit dem eigentlichen Gruppensuchstring gefunden wurde. Und das sind bei der 3. Gruppe eben die einzelnen Parameter.

Sie sehen, dass wir nur durch die Anwendung von Regulären Ausdrücken schon auf dem besten Wege sind, einen kompletten Cross-Referenzer zu kreieren – und bislang haben wir noch nicht eine einzige Zeile Code dafür schreiben müssen!

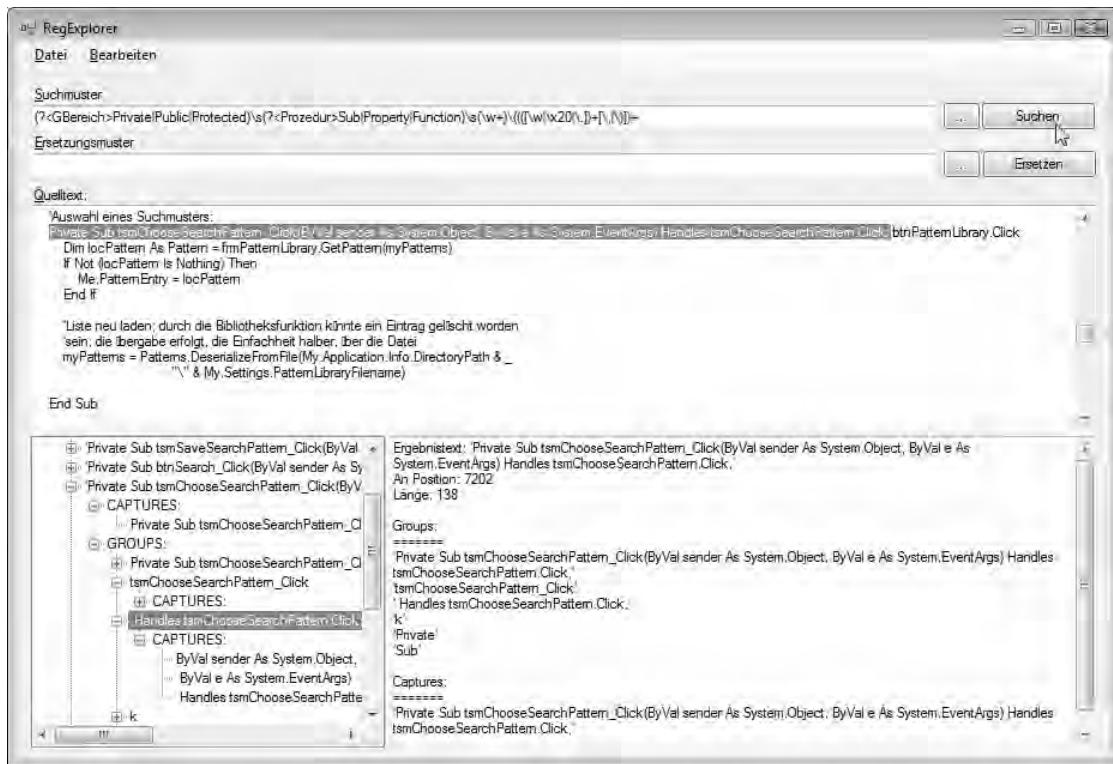


Abbildung 27.8 Richtig angewendet entlocken Sie der Regular Expression Engine mit Captures die durch die Quantifizierer entstandenen Suchergebnisse

Doch das wird sich gleich ändern. Im Abschnitt »Programmieren von Regulären Ausdrücken« ab Seite 796 erfahren Sie mehr darüber, wie Sie Reguläre Ausdrücke in Ihren eigenen Programmen einsetzen können.

Um die Tabellen zum Nachschlagen komplett zu halten, habe ich zuvor aber noch zwei für Sie.

Optionen bei der Suche

Optionen für die Suche können Sie später, bei der Programmierung von Regulären Ausdrücken, bequem über die Klassenparameter einstellen. Allerdings können Sie diese Parameter auch mit Steuerzeichen direkt in den Suchstring einbauen. Die folgende Tabelle gibt Ihnen Auskunft darüber, mit welchem Steuerzeichen Sie welchen Parameter aktivieren bzw. deaktivieren können.

RegexOption-Member	Inline-Zeichen	Beschreibung
None	Nicht vorhanden	Gibt an, dass keine Optionen festgelegt wurden.
IgnoreCase	i	Gibt an, dass bei Übereinstimmungen die Groß-/Kleinschreibung berücksichtigt werden soll.
Multiline	m	Bestimmt den Mehrzeilenmodus. Das ändert die Bedeutung von ^ und \$, sodass sie jeweils dem Anfang und dem Ende einer beliebigen Zeile innerhalb des zu durchsuchenden Strings und nicht nur dem Anfang und dem Ende der gesamten Zeichenfolge entsprechen.
ExplicitCapture	n	Gibt an, dass die einzigen gültigen Aufzeichnungen ausdrücklich benannte oder nummerierte Gruppen in der Form (?<name>...) sind. Dadurch können Klammern als nicht aufzeichnende Gruppen eingesetzt werden, ohne dass die umständliche Syntax des Ausdrucks (?...) benötigt wird.
Compiled	c	Gibt an, dass der Reguläre Ausdruck in eine Assembly kompiliert wird. Generiert MSIL (Microsoft Intermediate Language)-Code für den Regulären Ausdruck und ermöglicht eine schnellere Ausführung, jedoch auf Kosten der kurzen Startdauer.
Singleline	s	Gibt den Einzeilenmodus an. Ändert die Bedeutung des Punktes (), sodass dieser jedem Zeichen entspricht (und nicht jedem Zeichen mit Ausnahme von \n).
IgnorePatternWhitespace	x	Gibt an, dass Leerraum ohne Escape-Zeichen aus dem Muster ausgeschlossen wird und ermöglicht Kommentare hinter einem Nummernzeichen (#). Beachten Sie, dass niemals Leerraum aus einer Zeichenklasse eliminiert wird.
RightToLeft	r	Gibt an, dass die Suche von rechts nach links und nicht, wie standardmäßig, von links nach rechts durchgeführt wird. Ein Regulärer Ausdruck mit dieser Option steht links von der Anfangsposition und nicht rechts davon. (Daher sollte die Anfangsposition als das Ende der Zeichenfolge angegeben werden.) Diese Option kann nicht mitten in der Suchmusterzeichenfolge angegeben werden, um zu verhindern, dass Reguläre Ausdrücke mit Endlosschleifen auftreten. Die (?<)-Lookbehind-Konstrukte bieten jedoch eine ähnliche Funktionalität, die als Teilausdruck verwendet werden können. RightToLeft ändert lediglich die Suchrichtung. Die gesuchte untergeordnete Zeichenfolge an sich wird nicht umgekehrt.
ECMAScript	Nicht vorhanden	Gibt an, dass für den Ausdruck so genanntes ECMAScript-konformes Verhalten aktiviert ist (bestimmtes, standardisiertes <i>RegEx</i> -Verhalten). Diese Option kann nur in Verbindung mit dem <i>IgnoreCase</i> - und dem <i>Multiline</i> -Flag verwendet werden. Bei Verwendung dieser Option mit anderen Flags wird eine Ausnahme ausgelöst.
CultureInvariant	Nicht vorhanden	Gibt an, dass kulturelle Unterschiede bei der Sprache ignoriert werden.

Tabelle 27.5 Options-Steuerzeichen für Reguläre Ausdrücke

Steuerzeichen zu Gruppdefinitionen

Auch bei der Definition von Gruppen gibt es weitere Kombinationsmöglichkeiten. Die folgende Tabelle verrät Ihnen, welche es gibt:

Gruppenkonstrukt	Beschreibung
()	Zeichnet die übereinstimmende Teilzeichenfolge auf (oder die nicht aufzeichnende Gruppe). Aufzeichnungen mit () werden gemäß der Reihenfolge der öffnenden Klammern automatisch nummeriert, beginnend mit 1. Die erste Aufzeichnung, Aufzeichnungselement Nummer 0, ist der Text, dem das gesamte Muster für den Regulären Ausdruck entspricht. Auch wenn Sie also keine Gruppe mit einer Klammer gebildet haben, gibt es immer mindestens Gruppe 1.
(?<name>)	Zeichnet die übereinstimmende Teilzeichenfolge in einem Gruppennamen oder einem Nummernnamen auf. Die Zeichenfolge für »name« darf keine Satzzeichen enthalten und nicht mit einer Zahl beginnen. Sie können anstelle von spitzen Klammern einfache Anführungszeichen verwenden. Beispiel: »(?'name')«.
(?<name1-name2>)	Ausgleichsgruppdefinition. Löscht die Definition der zuvor definierten Gruppe »name2« und speichert in Gruppe »name1« das Intervall zwischen der zuvor definierten Gruppe »name2« und der aktuellen Gruppe. Wenn keine Gruppe »name2« definiert ist, wird die Übereinstimmung rückwärts verarbeitet. Da durch Löschen der letzten Definition von »name2« die vorherige Definition von »name2« angezeigt wird, kann mithilfe dieses Konstrukts der Aufzeichnungsstapel für die Gruppe »name2« als Zähler für die Aufzeichnung von geschachtelten Konstrukten, z.B. Klammern, verwendet werden. In diesem Konstrukt ist »name1« optional. Sie können anstelle von spitzen Klammern einfache Anführungszeichen verwenden. Beispiel: »(?'name1-name2')«.
(?:)	Nicht aufzeichnende Gruppe.
(?imnsx-imnsx:)	Aktiviert oder deaktiviert die angegebenen Optionen innerhalb des Teilausdrucks. Beispielsweise aktiviert »(?-s:)« die Einstellung, dass Groß-/Kleinschreibung nicht beachtet wird, und deaktiviert den Einzeilenmodus. Weitere Informationen hierzu finden Sie unter Optionen für Reguläre Ausdrücke.
(?=)	Positive Lookahead-Anweisung mit einer Breite von Null. Der Vergleich wird nur dann fortgesetzt, wenn der Teilausdruck rechts von dieser Position übereinstimmt. Beispiel: »w+(?=\\d)« entspricht einem Wort, gefolgt von einer Ziffer, wobei für die Ziffer keine Übereinstimmung gesucht wird. Dieses Konstrukt wird nicht rückwärts verarbeitet.
(?!)	Negative Lookahead-Anweisung mit einer Breite von Null. Der Vergleich wird nur dann fortgesetzt, wenn der Teilausdruck rechts von dieser Position nicht übereinstimmt. Beispiel: »\\b(?!un)\\w+\\b« entspricht Wörtern, die nicht mit »un« beginnen.
(?<=)	Positive Lookbehind-Anweisung mit einer Breite von Null. Der Vergleich wird nur dann fortgesetzt, wenn der Teilausdruck links von dieser Position übereinstimmt. Beispiel: »(?<=19)99« entspricht Instanzen von 99, die auf 19 folgen. Dieses Konstrukt wird nicht rückwärts verarbeitet.
(?<!)	Negative Lookbehind-Anweisung mit einer Breite von Null. Der Vergleich wird nur dann fortgesetzt, wenn der Teilausdruck links von dieser Position nicht übereinstimmt.
(?>)	Nicht zurückverfolgende Teilausdrücke (so genannte »gierige« Teilausdrücke). Für den Teilausdruck wird einmal eine volle Übereinstimmung gesucht, dann wird der Teilausdruck nicht stückweise in die Rückwärtsverarbeitung einbezogen. (D.h. der Teilausdruck entspricht nur Zeichenfolgen, für die durch den Teilausdruck allein eine Übereinstimmung gesucht werden würde.)

Tabelle 27.6 Spezielle Gruppensteuerzeichen bei Regulären Ausdrücken

Programmieren von Regulären Ausdrücken

Alle Grundlagen zu Regulären Ausdrücken sind jetzt an vielen Beispielen geklärt, und damit liegt die aufwändigste Lernarbeit bereits hinter Ihnen. Sich das Programmieren selbst anzueignen ist jetzt nur noch ein Klacks, und die Beispiele, die nun folgen, sind direkt aus dem Programm entnommen, mit dem Sie die ganze Zeit gearbeitet haben.

Die Klasse Regex bildet den Schlüssel zu den Funktionen von Regulären Ausdrücken. Sie können Sie auf zwei verschiedene Weisen verwenden: Entweder Sie instanziierten sie und nutzen ihre Member-Funktionen. Oder Sie nutzen ausschließlich ihre statischen Funktionen.

HINWEIS Aufgepasst jedoch, wenn Sie die statischen Funktionen der Regex-Klasse verwenden. Fehler, die zum Beispiel in Ausdrücken vorkommen, führen nicht zu Ausnahmen (*Exceptions*)! Ich habe keine Ahnung, was sich die Entwickler der Klassen dabei gedacht haben, aber ob Sie es glauben oder nicht: Wenn Sie die statischen Funktionen verwenden und Fehler dabei auftreten, zeigen die entsprechenden Funktionen lediglich eine MessageBox (!!!) – Ihr Programm bekommt davon aber nichts mit, es wird nicht durch eine Ausnahme unterbrochen und läuft weiter, als wäre nichts passiert. Deswegen gilt der Grundsatz: Instanziiieren Sie grundsätzlich die Regex-Klasse, und vermeiden Sie die Nutzung der statischen Funktionen, wo Sie können, denn Sie können sonst auf mögliche Fehler keinen Einfluss nehmen!

Zu Demonstrationszwecken (die statischen Funktionen wollen ja dennoch gezeigt werden) habe ich mich an diesen Grundsatz in den folgenden Beispielen übrigens ein paar Mal selbst nicht gehalten.

Ergebnisse im Match-Objekt

Beginnen wir direkt mit einer statischen Funktion: Sie möchten nach einem Regulären Ausdruck in einem String suchen. Nichts leichter als das, Sie schreiben einfach:

```
Dim match As Match = Regex.Match("Dieser wird durchsucht", "hiernach")
```

Das Match-Objekt selber wird durch die Match-Funktion zurückgeliefert. Möchten Sie auf die nicht statischen Member-Funktionen der Regex-Klasse zugreifen, müssen Sie das Regex-Objekt zunächst – wie jede andere Klasse auch – instanziiieren. Das gleiche Ergebnis würden Sie folgendermaßen erzielen:

```
Dim RegexInstanz As New Regex("Hiernach")
Dim match As Match = RegexInstanz.Match("Dieser wird durchsucht")
```

Da das Suchmuster bereits bei der Klasseninstanziierung bestimmt wird, geben Sie es im Unterschied zur Verwendung der statischen Version nicht mehr als Parameter an, wenn Sie nach Übereinstimmungen mit der Match-Methode suchen. Nun liegt es in der Natur von Regulären Ausdrücken, dass ein String höchstens ausreichen würde, den ersten Treffer im Text widerzuspiegeln. Sie benötigen allerdings mehr Informationen, um wirklich Brauchbares mit dem Treffer – oder vielmehr: den Treffern – anstellen zu können. Deswegen hält das Match-Objekt einige Eigenschaften parat, die diese Informationen liefern:

Eigenschaft des Match-Objektes	Funktion
NextMatch	Liefert ein neues Match-Objekt zurück, das Informationen über den nächsten Treffer enthält.
Value	Gibt den String zurück, der den Treffer darstellt.
Index	Gibt die Position innerhalb des Suchstrings an, an dem der Treffer aufgetreten ist. Die Positionszählung beginnt dabei, wie bei allen Strings, an der Position 0.
Length	Gibt an, aus wie vielen Zeichen der Treffer-String besteht.
Success	Ermittelt, ob der Treffer erfolgreich war.
Groups	Liefert eine <i>Collection</i> aus <i>Group</i> -Objekten zurück, die die Teilergebnisse der einzelnen Gruppen enthalten.
Captures	Liefert eine <i>Collection</i> aus <i>Capture</i> -Objekten zurück, die <i>Captures</i> enthalten, etwa wie die in den vorangegangenen Suchbegriff-Beispielen beschriebenen.

Tabelle 27.7 Die wichtigsten Eigenschaften des Match-Objektes

Die Matches-Auflistung

Nun wäre es nicht des schönen Programmierstils würdig, den .NET ermöglicht, müsste man den jeweils nächsten Treffer einer Regex.Match-Abfrage in einer Do/Loop-Schleife ermitteln, bis NextMatch schließlich Nothing zurückliefert, weil es keine weiteren Treffer mehr gibt. Aus diesem Grund bietet die Regex-Klasse die so genannte Matches-Collection an, die alle Treffer als Array von Match-Objekten erhält, und durch die sich vor allen Dingen elegant mit For/Each iterieren lässt.

Der Programmcode bis zur äußeren Schleife, die anschließend die TreeView im RegExplorer aufbaut, sieht aus diesem Grund folgendermaßen aus:

```
Private Sub btnSearch_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnSearch.Click

    Dim locRegEx As Regex

    Try
        locRegEx = New Regex(txtSearchPattern.Text)
    Catch ex As Exception
        MessageBox.Show("Fehler beim Anlegen des RegEx-Objektes!" + ex.Message,
                      "Fehler in Ausdruck:", MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
    End Try

    Dim locRootNode As TreeNode

    'Die Ergebnisstruktur in der TreeView anzeigen.
    tvwResults.Nodes.Clear()
    With tvwResults.Nodes

        'Dateiname ist Wurzelknoten der TreeView.
        locRootNode = .Add(myFilename)
        'Alle Match-Objekte (Treffer) durchlaufen...
    End With
End Sub
```

```
For Each locMatch As Match In locRegEx.Matches(txtSourceText.Text).
```

```
.
```

```
.
```

```
Next
```

Wenn der Anwender auf die *Suchen*-Schaltfläche klickt, verzweigt das Programm in Sub `btnSearch_Click`. Die Variable `locRegEx` wird anschließend als Typ `Regex` definiert. Da wir die `Regex`-Klasse instanziiieren, ergibt es Sinn, die Instanziierungsanweisung in einen Try/Catch-Block zu packen, da genau hier der Zeitpunkt in Frage kommt, zu dem die *Regular Expression Engine* eine Ausnahme auslösen kann, die ein anwendерfreundliches Programm natürlich abfangen muss. Die eigentliche Auswertung erfolgt mit dem Abruf von `locRegEx.Matches(txtSourceText.Text)` in der Schleifendefinition. Die Textbox `txtSourceText` enthält den Quelltext; den Suchbegriff, der aus der Textbox `txtSearchPattern` stammt, haben wir beim Instanziieren des `Regex`-Objektes Zeilen zuvor schon definiert. Die `Matches`-Eigenschaft liefert nun alle Treffer als Match-Auflistung zurück, durch die mit For/Each wunderbar iteriert werden kann. `locMatch` enthält dabei wieder alle Informationen über einen Treffer.

Abrufen von Captures und Gruppen eines Match-Objektes

Das Abrufen von vorhandenen *Captures* und Gruppen eines Match-Objektes gestaltet sich ebenfalls recht simpel: Für *Captures* verwenden Sie die `Captures`-Eigenschaft, die eine Captures-Auflistung zurückliefert, die wiederum aus einzelnen Capture-Objekten aufgebaut ist.

Für Gruppen verwenden Sie die `Groups`-Eigenschaft, die eine Groups-Auflistung aus Group-Objekten zurückliefert.

Das Iterieren durch diese Elemente ist ebenfalls recht einfach, wie der erste Teil der inneren Schleife des Programmteils zum Aufbau der `TreeView` zeigt:

```
'Alle Match-Objekte (Treffer) durchlaufen...
For Each locMatch As Match In locRegEx.Matches(txtSourceText.Text)

    'und in der TreeView darstellen.
    Dim locMainNode As New TreeNode("'" + locMatch.Value + "'")
    locMainNode.Tag = locMatch
    locRootNode.Nodes.Add(locMainNode)

    'Falls es zu einem Match Captures gab...
    If locMatch.Captures.Count > 0 Then
        Dim locCaptureNode As TreeNode = locMainNode.Nodes.Add("CAPTURES:")
        For Each locCC As Capture In locMatch.Captures
            '...auch diese unter jedem Match darstellen.
            Dim locNode As TreeNode = locCaptureNode.Nodes.Add(locCC.Value)
            locNode.Tag = locCC
        Next
    End If

    'Das Gleiche gilt für Groups.
    If locMatch.Groups.Count > 0 Then
        Dim locGroupNode As TreeNode = locMainNode.Nodes.Add("GROUPS:")
        For Each locGroup As Group In locMatch.Groups
            Dim locNode As TreeNode = locGroupNode.Nodes.Add(locGroup.Value)
            locNode.Tag = locGroup
        Next
    End If
End If
```

```
'Captures der einzelnen Gruppen nur im Bedarfsfall zeigen
If myGroupCaptures Then
    If locGroup.Captures.Count > 0 Then
        Dim locCaptureNode As TreeNode = locNode.Nodes.Add("CAPTURES:")
        For Each locCC As Capture In locGroup.Captures
            Dim locGCNode As TreeNode = locCaptureNode.Nodes.Add(locCC.Value)
            locGCNode.Tag = locCC
        Next
    End If
End If
Next
End If
Next
```

Damit die *Captures* der einzelnen Gruppen nur im Bedarfsfall angezeigt werden, gibt es ein Flag, das die Anzeige steuert. Dieses Flag wird gesetzt in Abhängigkeit der Einstellung, die der Anwender des Programms mit der Option *Group Captures anzeigen* im Menü *Datei* festlegt:

```
'An- und abschalten der Anzeige der Group Captures:
Private Sub tsmShowGroupCaptures_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmShowGroupCaptures.Click
    'Xor vertauscht Bits.
    myGroupCaptures = myGroupCaptures Xor True
    'Durch ein Häkchen widerspiegeln lassen:
    tsmShowGroupCaptures.Checked = myGroupCaptures
End Sub
```

Um die *Group Captures* letzten Endes darzustellen, wenn der Anwender diese Option gewählt hat, verfährt das Programm so wie im vorherigen Listing (fett markiert) zu sehen.

Sowohl die *Group-* als auch die *Capture*-Objekte, die dabei verwendet werden, sind dem *Match*-Objekt in ihrer Anwendung sehr ähnlich. Das *Capture*-Objekt verfügt über die folgenden wichtigen Eigenschaften:

Wichtige Eigenschaften des Capture-Objektes	Funktion
Value	Gibt den String zurück, der das Capture darstellt.
Index	Gibt die Position innerhalb des Suchstrings an, an dem das Capture aufgetreten ist. Die Positionszählung beginnt dabei, wie bei allen Strings, an der Position 0.
Length	Gibt an, aus wie vielen Zeichen der Capture-String besteht.

Tabelle 27.8 Die wichtigsten Eigenschaften des Capture-Objektes

Noch ähnlicher zum *Match*-Objekt ist das *Group*-Objekt, das quasi die gleiche Funktionalität wie das *Match*-Objekt aufweist, nur dass es – logischerweise – keine *Groups*-Eigenschaft besitzt und auch die *NextMatch*-Eigenschaft vermissen lässt.

Eigenschaft des Group-Objektes	Funktion
Value	Gibt den String zurück, der den Treffer der Gruppe darstellt.
Index	Gibt die Position innerhalb des Suchstrings an, an dem der Treffer der Gruppe aufgetreten ist. Die Positionszählung beginnt dabei, wie bei allen Strings, an der Position 0.
Length	Gibt an, aus wie vielen Zeichen der Gruppentreffer-String besteht.
Success	Ermittelt, ob der Gruppentreffer erfolgreich war.
Captures	Liefert eine Collection aus Capture-Objekten zurück, die Captures enthalten, etwa wie die in den vorangegangenen Suchbegriff-Beispielen beschriebenen.

Tabelle 27.9 Die wichtigsten Eigenschaften der Match-Klasse

Mit diesen Infos sind Sie für die Programmierung von Regulären Ausdrücken bestens gerüstet. Sie sehen selbst, dass sich die eigentliche Problemlösung mit Regulären Ausdrücken nicht in der Anwendung der Klassen verbirgt – das ist der weitaus weniger aufwändiger Teil. Das eigentliche Problem – oder besser: die eigentliche Übung, die Sie benötigen – liegt in der Anwendung der Regulären Ausdrücke selbst, also: Wie müssen Sie Ihre Such- bzw. Ersetzungsstrings gestalten, damit Sie die nötigen Informationen innerhalb einer Zeichenkette finden, um beispielsweise Benutzereingaben zu strukturieren und entsprechend auszuwerten?

Ein gutes Beispiel für den Einsatz von Regulären Ausdrücken zeigt das folgende Beispiel.

Regex am Beispiel: Berechnen beliebiger mathematischer Ausdrücke

Es gibt hunderte von denkbaren Anwendungen, bei denen Sie mathematische Ausdrücke innerhalb eines Programms auswerten müssen. Denken Sie beispielsweise an Aufmaßprogramme, die die Bausubstanz eines Hauses berechnen. Oder einen Funktionsplotter, der die Graphen beliebiger Funktionen darstellen kann. Überhaupt kann es immer nur von Vorteil sein, wenn Sie dem Anwender an geeigneten Stellen die Möglichkeit geben, eine beliebige Formel zu berechnen. Er braucht dann nämlich nicht für jede Kleinigkeit seinen Taschenrechner zu bemühen.

Leider ist das Parsen – also das Analysieren und Berechnen – eines mathematischen Ausdrucks keine wirklich triviale Sache, denn es gibt einige Dinge zu berücksichtigen:

Da sind zunächst mal Klammern, die die Priorität der Berechnungsreihenfolge ändern. Der Ausdruck

`2*2+2`

ergibt natürlich was völlig anderes als

`2*(2+2)`

Im ersten Fall lautet das Ergebnis 6, im zweiten 8. Noch komplizierter wird es, wenn man die Hierarchie der Operatoren berücksichtigt. Sie können eine Formel nicht einfach von links nach rechts auseinander nehmen, sondern müssen die Operatorprioritäten berücksichtigen:

Der Ausdruck

`2+2*2^2`

ist dafür ein gutes Beispiel. Hier werden erst die Potenz, anschließend das Produkt und schließlich die Summe berechnet, nicht umgekehrt.

Ebenfalls nicht trivial sind Funktionen, am besten mit unterschiedlich langen Funktionsnamen und beliebig vielen Parametern. Denken Sie beispielsweise an einen Ausdruck wie

`2+2*(2+Max(123;234;345;456))`

In dieser Formel müssen nicht nur der Funktionsname, sondern auch die Anzahl der Parameter bestimmt werden, die innerhalb des Funktionsnamens auftreten.

Und dann gibt es zu guter Letzt auch noch das Problem der negativen Vorzeichen. Der Parser muss so clever sein, dass er den Ausdruck

`-2*-1^(-2-1)`

in den Ausdruck

`((0-1)*2)*((0-1)*1)^(((0-1)*2)-((0-1)*1))`

umwandeln kann, wenn er nicht eine komplizierte Funktionalität zum Erkennen von negativen Zahlen bereitstellen will.



Abbildung 27.9 Mit dem Formelparser können Sie in Ihren eigenen Programmen beliebige Formeln errechnen

BEGLEITDATEIEN

Im Verzeichnis

`...\VB 2008 Entwicklerbuch\0 - Datenstrukturen\Kapitel 27\FormelParser`

finden Sie das Projekt *FormelParser*.

Dieses Programm enthält die Klasse `ADFormularParser`, die in der Lage ist, genau diese Auswertungen durchzuführen. Das einzige Formular des Projektes ist nur das Drumherum, damit Sie die Klasse ohne umständliche Kommandozeilenparameter austesten können.

Der Formelparser

Wenn Sie dieses Programm starten, sehen Sie einen Dialog, wie er auch in Abbildung 27.9 zu sehen ist. Eine Testformel ist bereits vorgegeben. Sie können unter *Formel* einen beliebigen Ausdruck eingeben und durch Mausklick auf *Berechnen* berechnen lassen. Das Ergebnis zeigt das Programm anschließend im Beschriftungsfeld unterhalb des Eingabefensters an.

Der Programmcode selber dazu ist denkbar gering und lautet wie folgt:

```
Private Sub btnCalculate_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnCalculate.Click

    Dim locFormPars As New ADFormularParser(txtFormular.Text)
    lblResult.Text = locFormPars.Result.ToString

End Sub
```

Sobald der Anwender die Schaltfläche betätigt, instanziert das Programm die Klasse `ADFormularParser` in `locFormPars`. Der eigentliche Ausdruck liegt dabei als String aus dem Textfeld `txtFormular` vor und wird dem Konstruktor dieser Klasse übergeben. Für das Berechnen dieses Ausdrucks ist nur ein einziger Methodenaufruf erforderlich: `Result`. Sie liefert das Ergebnis als Double zurück; damit das Ergebnis im Label angezeigt werden kann, wird es mit `Tostring` in eine Zeichenkette umgewandelt.

Die Klasse `ADFormularParser`

Interessant zu sehen ist es, wie die Klasse an sich arbeitet. Obwohl sie einen recht großen Funktionsumfang besitzt, ist die eigentliche Auswertungslogik durch die konsequente Anwendung von Regulären Ausdrücken recht klein gehalten.

Bevor Sie sich das dokumentierte Listing dieser Klasse anschauen, vielleicht noch ein paar Worte zur generellen Funktionsweise:

Bei der Entwicklung dieser Klasse stand ihre beliebige Erweiterbarkeit im Vordergrund. Das heißt im Klartext: Ein Entwickler, der diese Klasse benutzt, sollte durch Vererbung die Möglichkeit haben, auf einfache Weise den Funktionsvorrat zu erweitern. Diese Möglichkeit sollte aber nicht nur für normale Funktionen, sondern auch für Operatoren gelten. Aus diesem Grund besteht die Klasse eigentlich aus zwei wichtigen Klassen. Die erste Klasse speichert Operatoren in einem bestimmten Format; und die zweite Klasse ist schließlich für die eigentliche Funktionsauswertung verantwortlich.

Die Klasse zur Speicherung der Operatoren und Funktionen finden Sie im Folgenden abgedruckt. Sie befindet sich im Projekt in der Codedatei `ADFunction.vb`.

```
''' <summary>
''' Speichert die Parameter für eine Funktion, die vom FormelParser berücksichtigt werden kann.
''' </summary>
''' <remarks></remarks>
Public Class ADFunction
    Implements IComparable

    Public Delegate Function ADFunctionDelegate(ByVal parArray As Double()) As Double

    Protected myFunctionname As String
    Protected myParameters As Integer
    Protected myFunctionProc As ADFunctionDelegate
    Protected myConsts As ArrayList
    Protected myIsOperator As Boolean
    Protected myPriority As Byte

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse.
    ''' Verwenden Sie diese Überladungsversion, um Operatoren zu erstellen, die aus einem Zeichen
    bestehen,
    ''' </summary>
    ''' <param name="OperatorChar">Das Zeichen, das den Operator darstellt.</param>
    ''' <param name="FunctionProc">Der ADFunctionDelegat für die Berechnung durch diesen
    Operator.</param>
    ''' <param name="Priority">Die Operatorpriorität (3= Potenz, 2=Punkt, 1=Strich).</param>
    ''' <remarks></remarks>
    Sub New(ByVal OperatorChar As Char, ByVal FunctionProc As ADFunctionDelegate, ByVal Priority As
Byte)

        If Priority < 1 Then
            Dim Up As New ArgumentException("Priority kann für Operatoren nicht kleiner 1 sein.")
            Throw Up
        End If

        myFunctionname = OperatorChar.ToString
        myParameters = 2
        myFunctionProc = FunctionProc
        myIsOperator = True
        myPriority = Priority
    End Sub

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse. Verwenden Sie
    ''' diese Überladungsversion, um Funktionen zu erstellen, die aus mehreren Zeichen bestehen.
    ''' </summary>
    ''' <param name="FunctionName">Die Zeichenfolge, die den Funktionsnamen darstellt.</param>
    ''' <param name="FunctionProc">Der ADFunctionDelegat für die Berechnung durch diese
    Funktion.</param>
    ''' <param name="Parameters">Die Anzahl der Parameter, die diese Funktion entgegen nimmt.</param>
    ''' <remarks></remarks>
    Sub New(ByVal FunctionName As String, ByVal FunctionProc As ADFunctionDelegate, _
           ByVal Parameters As Integer)
        myFunctionname = FunctionName
        myFunctionProc = FunctionProc
        myParameters = Parameters
    End Sub
End Class
```

```
myIsOperator = False
myPriority = 0
End Sub

''' <summary>
''' Liefert den Funktionsnamen bzw. das Operatorenzeichen zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property FunctionName() As String
    Get
        Return myFunctionname
    End Get
End Property

''' <summary>
''' Liefert die Anzahl der zur Anwendung kommenden Parameter für diese Funktion zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property Parameters() As Integer
    Get
        Return myParameters
    End Get
End Property

''' <summary>
''' Zeigt an, ob es sich bei dieser Instanz um einen Operator handelt.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property IsOperator() As Boolean
    Get
        Return myIsOperator
    End Get
End Property

''' <summary>
''' Ermittelt die Priorität, die dieser Operator hat. (3=Potenz, 2=Punkt, 1=Strich)
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property Priority() As Byte
    Get
        Return myPriority
    End Get
End Property

''' <summary>
''' Ermittelt den Delegaten, der diese Funktion oder diesen Operator berechnet.
''' </summary>
```

```

''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property FunctionProc() As ADFunctionDelegate
    Get
        Return myFunctionProc
    End Get
End Property
''' <summary>
''' Ruft den Delegaten auf, der diese Funktion (diesen Operator) berechnet.
''' </summary>
''' <param name="parArray">Das Array, dass die Argumente der Funktion enthält.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Function Operate(ByVal parArray As Double()) As Double
    If Parameters > -1 Then
        If parArray.Length <> Parameters Then
            Dim Up As New ArgumentException _
                ("Anzahl Parameter entspricht nicht der Vorschrift der Funktion " & FunctionName)
            Throw Up
        End If
    End If
    Return myFunctionProc(parArray)
End Function

''' <summary>
''' Vergleicht zwei Instanzen dieser Klasse anhand ihres Prioritätswertes.
''' </summary>
''' <param name="obj">Eine ADFunction-Instanz, die mit dieser Instanz verglichen werden
soll.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Function CompareTo(ByVal obj As Object) As Integer Implements System.IComparable.CompareTo
    If obj.GetType Is GetType(ADFunction) Then
        Return myPriority.CompareTo(DirectCast(obj, ADFunction).Priority) * -1
    Else
        Dim up As _
            New ArgumentException("Nur ActiveDev.Function-Objekte können verglichen/sortiert
werden")
        Throw up
    End If
End Function
End Class

```

Wichtig zu wissen: Da Funktionen sich von Entwicklern, die die Klasse verwenden möchten, zu späterer Zeit hinzufügen lassen sollen, ohne dabei den Quellcode verändern zu müssen, arbeitet die Klasse ADFormularParser nicht mit fest »verdrahteten« Funktionen. Sie benutzt vielmehr Delegaten, um die Funktionsaufrufe durchzuführen. Zum besseren Verständnis des Beispielcodes sollten Sie daher den Abschnitt über Delegaten in Kapitel 14 studiert haben.

Kurz zur Wiederholung: Durch die Verwendung von Delegaten werden Funktionen nicht als feste Ziele durch Funktionsnamen angegeben. Funktionen selbst können in Variablen gespeichert werden (natürlich nicht die Funktionen selbst – intern werden lediglich die Zeiger auf die Adressen der Funktionen gespeichert). Sie definieren einen Delegaten mit einer Deklarationsanweisung ähnlich wie die Prozedur einer

Klasse. Im Gegensatz zu einer solchen Prozedur erreichen Sie durch das Schlüsselwort Delegate, dass sich eine Prozedur, die über die gleiche Signatur wie der Delegat verfügt, in einer Delegaten-Variable speichern lässt. Im Beispiel wird der Delegat namens ADFunctionDelegate mit der folgenden Anweisung deklariert:

```
Public Delegate Function ADFunctionDelegate(ByVal parArray As Double()) As Double
```

Sie können nun eine Objektvariable vom Typ ADFunctionDelegate erstellen und ihr eine Funktion zuweisen. Der Zeiger auf die Funktion wird dabei in dieser Objektvariablen gespeichert. Angenommen, es gibt, wie im Beispiel, irgendwo die folgende Prozedur:

```
Public Shared Function Addition(ByVal Args() As Double) As Double
    Return Args(0) + Args(1)
End Function
```

Dann können Sie sie, da sie über die gleiche Signatur wie der Delegat verfügt, in einer Delegatenvariablen speichern – etwa wie im folgenden Beispiel:

```
Dim locDelegate As ADFunctionDelegate
locDelegate = New ADFunctionDelegate(AddressOf Addition)
```

Die Verwendung von AddressOf (etwa: *Adresse von*) macht deutlich, dass hier die Adresse (also ein Zeiger) der Funktion in der Delegatenvariable gespeichert wird.

Es gibt anschließend zwei Möglichkeiten, die Funktion Addition aufzurufen: einmal direkt über den Funktionsnamen und über den Delegaten. Der direkte Aufruf mit

```
Dim locErgebnis As Double = Addition(New Double() {10, 10})
```

bewirkt also das Gleiche wie der Umweg über den Delegaten mit

```
Dim locErgebnis As Double = locDelegate(New Double() {10, 10})
```

bloß mit einem zusätzlichen Vorteil: Sie können erst zur Laufzeit durch entsprechende Variablenzuweisung bestimmen, welche Funktion aufgerufen werden soll, wenn es mehrere Funktionen gleicher Signatur gibt.

Um beim Beispiel zu bleiben: Genau das ist der Grund, wieso den Funktionen, die das Programm berechnen kann, die Argumente als Double-Array und als fortlaufende Parameter übergeben werden. So kann gewährleistet werden, dass alle Funktionen die gleichen Signaturen haben. Damit kann jede Funktion in einer Delegatenvariablen gespeichert werden (vordefiniert in einer generischen List(Of ADFunction)-Auflistung, im nachfolgenden Listing fett markiert). Die Auswertungsroutine kann dann mithilfe der Auflistung nicht nur den Funktionsnamen finden, sondern die zugeordnete mathematische Funktion auch direkt über ihren Delegaten aufrufen.

Ansonsten hat die Klasse lediglich die Aufgabe, die Rahmendaten einer Funktion (oder eines Operators) zu speichern. Für Operatoren gibt es eine besondere Eigenschaft namens Priority, die es erlaubt, die Stellung eines Operators in der Hierarchieliste aller Operatoren zu bestimmen. Diese Priorität ermöglicht, die Regel »Potenz-vor-Klammer-vor-Punkt-vor-Strich« einzuhalten. Bestimmte Operatoren (wie beispielsweise »^« für die Potenz) haben eine höhere Priorität als andere (wie beispielsweise »+« für die Addition).

```
Imports System.Text.RegularExpressions

Public Class ADFormularParser

    Protected myFormular As String
    Protected myFunctions As List(Of ADFunction)
    Protected myPrioritizedOperators As ADPrioritizedOperators
    Protected Shared myPredefinedFunctions As List(Of ADFunction)
    Protected myResult As Double
    Protected myIsCalculated As Boolean
    Protected myConsts As ArrayList
    Private myConstEnumCounter As Integer

    Protected Shared myXVariable As Double
    Protected Shared myYVariable As Double
    Protected Shared myZVariable As Double

    'Definiert die Standardfunktionen statisch bei der ersten Verwendung dieser Klasse.
    Shared Sub New()

        myPredefinedFunctions = New List(Of ADFunction)

        With myPredefinedFunctions
            .Add(New ADFunction("+"c, AddressOf Addition, CByte(1)))
            .Add(New ADFunction("-"c, AddressOf Subtraction, CByte(1)))
            .Add(New ADFunction("*"c, AddressOf Multiplication, CByte(2)))
            .Add(New ADFunction("/"c, AddressOf Division, CByte(2)))
            .Add(New ADFunction("\\"c, AddressOf Remainder, CByte(2)))
            .Add(New ADFunction("^"c, AddressOf Power, CByte(3)))
            .Add(New ADFunction("PI", AddressOf PI, 1))
            .Add(New ADFunction("Sin", AddressOf Sin, 1))
            .Add(New ADFunction("Cos", AddressOf Cos, 1))
            .Add(New ADFunction("Tan", AddressOf Tan, 1))
            .Add(New ADFunction("Max", AddressOf Max, -1))
            .Add(New ADFunction("Min", AddressOf Min, -1))
            .Add(New ADFunction("Sqrt", AddressOf Sqrt, 1))
            .Add(New ADFunction("Tanh", AddressOf Tanh, 1))
            .Add(New ADFunction("LogDec", AddressOf LogDec, 1))
            .Add(New ADFunction("XVar", AddressOf XVar, 1))
            .Add(New ADFunction("YVar", AddressOf YVar, 1))
            .Add(New ADFunction("ZVar", AddressOf ZVar, 1))
        End With
    End Sub

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse.
    ''' </summary>
    ''' <param name="Formular">Die auszuwertende Formel, die als Zeichenkette vorliegen muss.</param>
    ''' <remarks></remarks>
    Sub New(ByVal Formular As String)

        'Vordefinierte Funktionen übertragen
        myFunctions = myPredefinedFunctions
        myFormular = Formular
        OnAddFunctions()

    End Sub
```

Die Parser-Klasse verfügt über zwei Konstruktoren – über einen statischen und einen nicht statischen. Im statischen Konstruktor, der dann aufgerufen wird, wenn die Klasse innerhalb einer Assembly das erste Mal zur Anwendung kommt, werden die Grundfunktionen der Klasse definiert, deren Code sich in der Projektdatei *ADFormularParser.Math.vb* befindet.

Bei der eigentlichen Instanziierung der Klasse werden die so schon vorhandenen Funktionen der Klasseninstanz zugewiesen. Indem der Anwender die Klasse vererbt und die Funktion *OnAddFunction* überschreibt, kann er im gleichen Stil weitere Funktionen der Klasse hinzufügen. Ein Beispiel dafür folgt am Ende der Codebeschreibung.

```
'Mit dem Überschreiben dieser Funktion kann der Entwickler eigene Funktionen hinzufügen
Public Overridable Sub OnAddFunctions()
    'Nichts zu tun in der Basisversion
    Return
End Sub

'Interne Funktion, die das Berechnen startet.
Private Sub Calculate()

    Dim locFormular As String = myFormular
    Dim locOpStr As String = ""

    'Operatorenliste anlegen
    myPriorizedOperators = New ADPrioritizedOperators
    For Each adf As ADFunction In myFunctions
        If adf.IsOperator Then
            myPriorizedOperators.AddFunction(adf)
        End If
    Next

    'Operatoren Zeichenkette zusammenbauen

    For Each ops As ADFunction In myFunctions
        If ops.IsOperator Then
            locOpStr += "\\" + ops.FunctionName
        End If
    Next

    'White-Spaces entfernen
    'Syntax-Check für Klammern
    'Negativ-Vorzeichen verarbeiten
    locFormular = PrepareFormular(locFormular, locOpStr)

    'Konstanten 'rausparsen
    locFormular = GetConsts(locFormular)

    myResult = ParseSimpleTerm(Parse(locFormular, locOpStr))
    IsCalculated = True

End Sub
```

Diese Routine ist die »Zentrale« der Parser-Klasse. Hier werden alle weiteren Funktionen aufgerufen, die benötigt werden, um einen Ausdruck korrekt auszuwerten.

```
'Überschreibbare Funktion, die die Formelauswertung steuert.
Protected Overrides Function Parse(ByVal Formular As String, ByVal OperatorRegEx As String) As
String

    Dim locTemp As String
    Dim locTerm As Match
    Dim locFuncName As Match
    Dim locMoreInnerTerms As MatchCollection
    Dim locPreliminaryResult As New ArrayList
    Dim locFuncFound As Boolean
    Dim locOperatorRegEx As String = "\(([^\d;]" + OperatorRegEx + "[^]*\)"

    Dim adf As ADFunction

    locTerm = Regex.Match(Formular, locOperatorRegEx)
    If locTerm.Value <> "" Then
        locTemp = Formular.Substring(0, locTerm.Index)

        'Befindet sich ein Funktionsname davor?
        locFuncName = Regex.Match(locTemp, "[a-zA-Z]*", RegexOptions.RightToLeft)

        'Gibt es mehrere, durch ; getrennte Parameter?
        locMoreInnerTerms = Regex.Matches(locTerm.Value, "[\d" + OperatorRegEx + "]*[;|\])")

        'Jeden Parameterterm auswerten und zum Parameter-Array hinzufügen
        For Each locMatch As Match In locMoreInnerTerms
            locTemp = locMatch.Value
            locTemp = locTemp.Replace(";", "").Replace(")", "")
            locPreliminaryResult.Add(ParseSimpleTerm(locTemp))
        Next

        'Möglicher Syntaxfehler: Mehrere Parameter, aber keine Funktion
        If locFuncName.Value = "" And locMoreInnerTerms.Count > 1 Then
            Dim up As New SyntaxErrorException _
                ("Mehrere Klammerparameter aber kein Funktionsname angegeben!")
            Throw up
        End If

        If locFuncName.Value <> "" Then
            'Funktionsnamen suchen
            locFuncFound = False
            For Each adf In myFunctions
                If adf.FunctionName.ToUpper = locFuncName.Value.ToUpper Then
                    locFuncFound = True
                    Exit For
                End If
            Next

            If locFuncFound = False Then
                Dim up As New SyntaxErrorException("Der Funktionsname wurde nicht gefunden")
                Throw up
            Else
                Formular = Formular.Replace(locFuncName.Value + locTerm.Value, _
                    myConstEnumCounter.ToString("000"))
                Dim locArgs(locPreliminaryResult.Count - 1) As Double
```

```

    locPreliminaryResult.CopyTo(locArgs)
    'Diese Warnung bezieht sich auf einen hypothetischen Fall,
    'der aber nie eintreten kann!
    myConsts.Add(adf.Operate(locArgs))
    myConstEnumCounter += 1
End If
Else
    Formular = Formular.Replace(locTerm.Value, myConstEnumCounter.ToString("000"))
    myConsts.Add(CDbl(locPreliminaryResult(0)))
    myConstEnumCounter += 1
End If
Else
    Return Formular
End If
Formular = Parse(Formular, OperatorRegEx)
Return Formular

End Function

'Überschreibbare Funktion, die einen einfachen Term
'(ohne Funktionen, nur Operatoren) auswertet.
Protected Overridable Function ParseSimpleTerm(ByVal Formular As String) As Double

    Dim locPos As Integer
    Dim locResult As Double

    'Klammern entfernen
    If Formular.IndexOfAny(New Char() {"(", ")"}) > -1 Then
        Formular = Formular.Remove(0, 1)
        Formular = Formular.Remove(Formular.Length - 1, 1)
    End If

    'Die Prioritäten der verschiedenen Operatoren von oben nach unten durchlaufen
    For locPrioCount As Integer = myPriorizedOperators.HighestPriority To _
        myPriorizedOperators.LowestPriority Step -1
        Do
            'Schauen, ob *nur* ein Wert
            If Formular.Length = 3 Then
                Return CDbl(myConsts(Integer.Parse(Formular)))
            End If

            'Die Operatorenzeichen einer Ebene ermitteln
            Dim locCharArray As Char() = myPriorizedOperators.OperatorChars(CByte(locPrioCount))
            If locCharArray Is Nothing Then
                'Gibt keinen Operator dieser Ebene, dann nächste Hierarchie.
                Exit Do
            End If

            'Nach einem der Operatoren dieser Hierarchieebene suchen
            locPos = Formular.IndexOfAny(locCharArray)
            If locPos = -1 Then
                'Kein Operator dieser Ebene mehr in der Formel vorhanden - nächste Hierarchie.
                Exit Do
            Else
                Dim locDblArr(1) As Double

```

```

'Operator gefunden - Teilterm ausrechnen
locDblArr(0) = CDb1(myConsts(Integer.Parse(Formular.Substring(locPos - 3, 3))))
locDblArr(1) = CDb1(myConsts(Integer.Parse(Formular.Substring(locPos + 1, 3)))

'Die entsprechende Funktion aufrufen, die durch die Hilfsklassen
'anhand Priorität und Operatorzeichen ermittelt werden kann.
Dim locOpChar As Char = Convert.ToChar(Formular.Substring(locPos, 1))
locResult = myPriorizedOperators.OperatorByChar(
    CByte(locPrioCount), locOpChar).Operate(locDblArr)

'Und den kompletten Ausdruck durch eine neue Konstante ersetzen
myConsts.Add(locResult)
Formular = Formular.Remove(locPos - 3, 7)
Formular = Formular.Insert(locPos - 3, myConstEnumCounter.ToString("000"))
myConstEnumCounter += 1
End If
Loop
Next
End Function

```

Parse und ParseSimpleTerm sind die eigentlichen Arbeitspferde der Klasse. Die Funktionsweise von Parse ergibt sich aus den Kommentaren – weitere Erklärungen dazu sind deswegen überflüssig. Interessant wird es bei ParseSimpleTerm, vor allen Dingen, wenn Sie zu den Lesern gehören, die den Vorgänger dieses Buchs gelesen haben: In der ersten Version dieses Beispiels zu *Visual Basic .NET – Das Entwicklerbuch* (2003) hatte sich nämlich ein gemeiner Fehler eingeschlichen:¹ ParseSimpleTerm dient dazu, dass Teilterme des gesamten Ausdrucks, die nur aus Operatoren bestehen, gemäß der Operatorenhierarchie ausgewertet werden. Dabei müssen Operatoren gleicher Priorität von links nach rechts ausgewertet werden. In der Vorgängerversion dieses Beispiels kamen Operatoren gleicher Hierarchie aber auch *nacheinander* zur Anwendung (also erst wurden Additionen, dann wurden Subtraktionen berechnet, nicht, wie es sein sollte, Additionen und Subtraktionen in einem Zug). Bei bestimmten Konstellationen hatte das falsche Ergebnisse zur Folge (beispielsweise bei dem Ausdruck $-2+1$ der -3 ergab, da zunächst alle Additionen ($2+1$) und anschließend alle Subtraktionen ($*-1$) durchgeführt wurden).

Mit zwei neu geschaffenen Hilfsklassen, die Sie in der Codedatei *AuxilliaryClasses.vb* finden, arbeitet ParseSimpleTerm nun so, dass Operatoren gleicher Ebene so berechnet werden, wie sie von links nach rechts betrachtet auftreten.

Die folgende Routine nutzt die Text-Analyse-Fähigkeit von Regulären Ausdrücken, um einen konstanten Ausdruck in der zu analysierenden Formel zu ermitteln. An diesem Beispiel wird einmal mehr deutlich, wie das geschickte Verwenden von Regulären Ausdrücken eine ganze Menge Programmieraufwand sparen kann.

```

'Überschreibbare Funktion, die die konstanten Zahlenwerte in der Formel ermittelt.
Protected Overrides Function GetConsts(ByVal Formular As String) As String
    Dim locRegEx As New Regex("[\d,.]+[S]*")
    'Alle Ziffern mit Komma oder Punkt aber keine Whitespaces
    myConstEnumCounter = 0
    myConsts = New ArrayList
    Return locRegEx.Replace(Formular, AddressOf EnumConstsProc)

```

¹ Mein Dank gilt deswegen an dieser Stelle Andreas Schlegel, der mich schon in der 2005er Version dieses Buchs auf diesen Fehler aufmerksam gemacht hat!

```

End Function
'Rückruffunktion für das Auswerten der einzelnen Konstanten (siehe vorherige Zeile).
Protected Overridable Function EnumConstsProc(ByVal m As Match) As String
    Try
        myConsts.Add(Double.Parse(m.Value))
        Dim locString As String = myConstEnumCounter.ToString("000")
        myConstEnumCounter += 1
        Return locString
    Catch ex As Exception
        myConsts.Add(Double.NaN)
        Return "ERR"
    End Try
End Function

'Hier werden vorbereitende Arbeiten durchgeführt.
Protected Overridable Function PrepareFormular(ByVal Formular As String, _
    ByVal OperatorRegEx As String) As String
    Dim locBracketCounter As Integer
    'Klammern überprüfen
    For Each locChar As Char In Formular.ToCharArray
        If locChar = "("c Then
            locBracketCounter += 1
        End If
        If locChar = ")"c Then
            locBracketCounter -= 1
            If locBracketCounter < 0 Then
                Dim up As New SyntaxErrorException
                    ("Zu viele Klammer-Zu-Zeichen.")
                Throw up
            End If
        End If
    Next
    If locBracketCounter > 0 Then
        Dim up As New SyntaxErrorException
            ("Eine offene Klammer wurde nicht ordnungsgemäß geschlossen.")
        Throw up
    End If

    'White-Spaces entfernen
    Formular = Regex.Replace(Formular, "\s", "")

    'Vorzeichen verarbeiten
    If Formular.StartsWith("-") Or Formular.StartsWith("+") Then
        Formular = Formular.Insert(0, "0")
    End If

    'Sonderfall negative Klammer
    Formular = Regex.Replace(Formular, "\(-\(", "(0-(")

    Return Regex.Replace(Formular, _
        "(?<operator>[" + OperatorRegEx + "]\)-(?<zah1>[\d\.\.,]*), _"
        "${operator}((0-1)*${zah1})")
End Function

```

Und auch für die Auswertung von Vorzeichen verwendet das Programm wieder die Hilfe von Regulären Ausdrücken (im oben stehenden Listingauszug fett dargestellt).

Dabei wird, wie schon eingangs erwähnt, beispielsweise der Ausdruck

-2*-1^(-2-1)

in den Ausdruck

((0-1)*2)*((0-1)*1)^(((0-1)*2)-((0-1)*1))

umgewandelt – mit diesem kleinen Trick sind negative Vorzeichen berücksichtigt. Die Routine nutzt dazu die Suchen-und-Ersetzen-Funktion Replace der Regex-Klasse.

```
Public Property Formular() As String
    Get
        Return myFormular
    End Get
    Set(ByVal Value As String)
        IsCalculated = False
        myFormular = Value
    End Set
End Property

Public ReadOnly Property Result() As Double
    Get
        If Not IsCalculated Then
            Calculate()
        End If
        Return myResult
    End Get
End Property

Public Property IsCalculated() As Boolean
    Get
        Return myIsCalculated
    End Get
    Set(ByVal Value As Boolean)
        myIsCalculated = Value
    End Set
End Property

Public Property Functions() As ArrayList
    Get
        Return myFunctions
    End Get
    Set(ByVal Value As ArrayList)
        myFunctions = Value
    End Set
End Property
```

Alle fest implementierten mathematischen Operatoren und Funktionen folgen ab diesem Punkt. Beachten Sie, dass auch die hier implementierten Funktionen die Signatur des Delegaten **ADFunctionDelegate** voll erfüllen. Falls Sie den Formel Parser um eigene Operatoren oder Funktionen ergänzen wollen, können Sie diese prinzipiell hier entlehnen.

Wie Sie genau vorgehen, um den Formel Parser um eigene Funktionen zu erweitern, erfahren Sie im letzten Abschnitt dieses Kapitels. Die mathematischen Funktionen finden Sie übrigens ausgelagert in der Codedatei *ADFormularParser_Math.vb*.

HINWEIS Zwar gibt es zwei Codedateien – *ADFormularParser.vb* sowie *ADFormularParser_Math.vb* – aber beide Codedateien enthalten dennoch Code für nur *eine* Klasse. Möglich wird das durch das Schlüsselwort *Partial*, mit dem Sie Klassen mit umfangreichen Code auf mehrere Codedateien verteilen können. Mehr zu partiellen Klassen finden Sie auch in Kapitel 14.

```
Partial Public Class ADFormularParser
    Public Shared Function Addition(ByVal Args() As Double) As Double
        Return Args(0) + Args(1)
    End Function

    Public Shared Function Subtraction(ByVal Args() As Double) As Double
        Return Args(0) - Args(1)
    End Function

    Public Shared Function Multiplication(ByVal Args() As Double) As Double
        Return Args(0) * Args(1)
    End Function

    Public Shared Function Division(ByVal Args() As Double) As Double
        Return Args(0) / Args(1)
    End Function

    Public Shared Function Remainder(ByVal Args() As Double) As Double
        Return Decimal.Remainder(New Decimal(Args(0)), New Decimal(Args(1)))
    End Function

    Public Shared Function Power(ByVal Args() As Double) As Double
        Return Args(0) ^ Args(1)
    End Function

    Public Shared Function Sin(ByVal Args() As Double) As Double
        Return Math.Sin(Args(0))
    End Function

    Public Shared Function Cos(ByVal Args() As Double) As Double
        Return Math.Cos(Args(0))
    End Function
    Public Shared Function Tan(ByVal Args() As Double) As Double
        Return Math.Tan(Args(0))
    End Function

    Public Shared Function Sqrt(ByVal Args() As Double) As Double
        Return Math.Sqrt(Args(0))
    End Function

    Public Shared Function PI(ByVal Args() As Double) As Double
        Return Math.PI
    End Function
    Public Shared Function Tanh(ByVal Args() As Double) As Double
        Return Math.Tanh(Args(0))
    End Function
```

```
Public Shared Function LogDec(ByVal Args() As Double) As Double
    Return Math.Log10(Args(0))
End Function

Public Shared Function XVar(ByVal Args() As Double) As Double
    Return XVariable
End Function

Public Shared Function YVar(ByVal Args() As Double) As Double
    Return YVariable
End Function

Public Shared Function ZVar(ByVal Args() As Double) As Double
    Return ZVariable
End Function

Public Shared Function Max(ByVal Args() As Double) As Double
    Dim retDouble As Double
    If Args.Length = 0 Then
        Return 0
    Else
        retDouble = Args(0)
        For Each locDouble As Double In Args
            If retDouble < locDouble Then
                retDouble = locDouble
            End If
        Next
    End If
    Return retDouble
End Function

Public Shared Function Min(ByVal Args() As Double) As Double
    Dim retDouble As Double
    If Args.Length = 0 Then
        Return 0
    Else
        retDouble = Args(0)
        For Each locDouble As Double In Args
            If retDouble > locDouble Then
                retDouble = locDouble
            End If
        Next
    End If
    Return retDouble
End Function

Public Shared Property XVariable() As Double
    Get
        Return myXVariable
    End Get
    Set(ByVal Value As Double)
        myXVariable = Value
    End Set
End Property
```

```

Public Shared Property YVariable() As Double
    Get
        Return myYVariable
    End Get
    Set(ByVal Value As Double)
        myYVariable = Value
    End Set
End Property

Public Shared Property ZVariable() As Double
    Get
        Return myZVariable
    End Get
    Set(ByVal Value As Double)
        myZVariable = Value
    End Set
End Property

End Class

```

TIPP Ein paar Worte zu den Funktionen XVariable, YVariable und ZVariable möchte ich an dieser Stelle verlieren: Sie dienen dazu, mit Variablen innerhalb einer Formel zu arbeiten, die Sie wiederum vom Programm aus steuern können. Wenn Sie innerhalb der zu verarbeitenden Formel die Ausdrücke XVar, YVar und ZVar verwenden, werden deren Funktionsergebnisse aus den hier zu sehenden Funktionen »entnommen«. Auf diese Weise haben Sie die Möglichkeit, beispielsweise einen Funktionsplotter mit frei bestimmbarer Formeln zu realisieren oder einfach nur ein simples Programm, das Ihnen Wertetabellen von Funktionen erstellt. Sie müssen dabei lediglich die entsprechenden Werte für XVariable, YVariable und ZVariable innerhalb Ihres Programms setzen.

Wichtig für das korrekte Funktionieren der Auswertung für einfache Teilterme sind die folgenden beiden Hilfsklassen, die Sie in der Codedatei *AuxilliaryClasses.vb* finden.

```

Imports System.Collections.ObjectModel

''' <summary>
''' Auflistung, in der alle Operatoren gleicher Priorität gesammelt werden, damit
''' es die Möglichkeit gibt, sie von links nach rechts (in einem Rutsch) zu verarbeiten.
''' </summary>
''' <remarks></remarks>
Public Class ADOperatorsOfSamePriority
    Inherits Collection(Of ADFunction)
    Private myPriority As Byte

    Sub New()
        MyBase.New()
    End Sub

    Protected Overrides Sub InsertItem(ByVal index As Integer, ByVal item As ADFunction)
        If Not item.IsOperator Then
            Dim locEx As New
                ArgumentException("Nur Operatoren (keine Funktionen) können dieser Auflistung hinzugefügt
werden!")
            Throw locEx
        End If
        If Me.Count = 0 Then

```

```
    myPriority = item.Priority
Else
    'Überprüfen, ob es dieselbe Priorität ist, sonst Ausnahme!
    If item.Priority <> myPriority Then
        Dim locEx As New _
            ArgumentException("Nur Operatoren der Priorität " & myPriority _
                & " können dieser Auflistung hinzugefügt werden!")
        Throw locEx
    End If
End If
 MyBase.InsertItem(index, item)
End Sub

Protected Overrides Sub SetItem(ByVal index As Integer, ByVal item As ADFunction)
    Dim locEx As New _
        ArgumentException("Elemente können in dieser Auflistung nicht ausgetauscht werden!")
    Throw locEx
End Sub

''' <summary>
''' Liefert die Priorität dieser Operatorenauflistung zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property Priority() As Byte
    Get
        Return myPriority
    End Get
End Property
End Class
''' <summary>
''' Fasst alle Operatorenlisten nach Priorität kategorisiert in einer übergeordneten Auflistung
zusammen.
''' </summary>
''' <remarks></remarks>
Public Class ADPrioritizedOperators
    Inherits KeyedCollection(Of Byte, ADOperatorsOfSamePriority)
    Private myHighestPriority As Byte
    Private myLowestPriority As Byte

    ''' <summary>
    ''' Fügt einer der untergeordneten Auflistungen einen neuen Operator hinzu,
    ''' in Abhängigkeit von seiner Priorität.
    ''' </summary>
    ''' <param name="Function"></param>
    ''' <remarks></remarks>
    Public Sub AddFunction(ByVal [Function] As ADFunction)
        'Feststellen, ob es schon eine Auflistung für diese Operator-Priorität gibt
        If Me.Contains([Function].Priority) Then
            'Ja - dieser hinzufügen,
            Me([Function].Priority).Add([Function])
        Else
            'Nein - anlegen und hinzufügen.
        End If
    End Sub
End Class
```

```
Dim locOperatorsOfSamePriority As New ADOperatorsOfSamePriority()
locOperatorsOfSamePriority.Add([Function])
Me.Add(locOperatorsOfSamePriority)
End If
End Sub

Protected Overrides Function GetKeyForItem(ByVal item As ADOperatorsOfSamePriority) As Byte
    Return item.Priority
End Function

Protected Overrides Sub InsertItem(ByVal index As Integer, ByVal item As ADOperatorsOfSamePriority)

    If Me.Count = 0 Then
        myHighestPriority = item.Priority
        myLowestPriority = item.Priority
        MyBase.InsertItem(index, item)
        Return
    End If

    MyBase.InsertItem(index, item)

    If myHighestPriority < item.Priority Then
        myHighestPriority = item.Priority
    End If

    If myLowestPriority > item.Priority Then
        myLowestPriority = item.Priority
    End If
End Sub
''' <summary>
''' Liefert alle Operatorzeichen einer bestimmten Priorität als Char-Array zurück.
''' </summary>
''' <param name="Priority">Die Priorität, deren Operatoren zusammengestellt werden sollen.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Function OperatorChars(ByVal Priority As Byte) As Char()
    If Me.Contains(Priority) Then
        Dim locChars As New List(Of Char)
        For Each locFunction As ADFunction In Me(Priority)
            locChars.Add(Convert.ToChar(locFunction.FunctionName))
        Next
        Return locChars.ToArray
    End If
    Return Nothing
End Function
''' <summary>
''' Liefert die Funktion zurück, die sich durch ein Operator-Zeichen einer bestimmten Priorität
ergibt.
''' </summary>
''' <param name="Priority">Die Priorität, die den Operatoren entspricht, die ...</param>
''' <param name="OperatorChar">Das Operatorzeichen mit der angegebenen ....</param>
''' <returns></returns>
''' <remarks></remarks>
```

```

Public Function OperatorByChar(ByVal Priority As Byte, ByVal OperatorChar As Char) As ADFunction
    If Me.Contains(Priority) Then
        For Each locFunction As ADFunction In Me(Priority)
            If OperatorChar = Convert.ToChar(locFunction.FunctionName) Then
                Return locFunction
            End If
        Next
    End If
    Return Nothing
End Function

''' <summary>
''' Liefert die höchste Prioritätennummer zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property HighestPriority() As Byte
    Get
        Return myHighestPriority
    End Get
End Property

''' <summary>
''' Liefert die kleinste Prioritätennummer zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property LowestPriority() As Byte
    Get
        Return myLowestPriority
    End Get
End Property
End Class

```

Diese beiden Klassen dienen dazu, Operatoren nach Priorität einzuordnen. Gleichzeitig stellen sie Funktionen bereit, mit deren Hilfe man einerseits alle Operatorzeichen einer Hierarchieebene anhand der Prioritätennummer, andererseits die Funktionsklasse (ADFunction) einer Funktion mithilfe von Priorität und Operatorzeichen ermitteln kann.

Verwendet werden beide Funktionen anschließend, um folgende Konstrukte bei der Formelauswertung in den Griff zu bekommen:

5–2*3+7/2+1

Im ersten Schritt werden die Ausdrücke $2*3$ und $7/2$ verarbeitet – und zwar von links nach rechts innerhalb eines Durchlaufs. Um die Suche nach beiden Operatorzeichen dieser Ebene zu finden, verfügt die Klasse über die Methode `OperatorChars`, die ein Char-Array für die entsprechende Hierarchieebene zurückliefert – im Beispiel also die Zeichen »*« und »/«. Mit der String-Funktion `IndexOfAny` können also beide Operatoren tatsächlich auf gleicher Ebene gefunden werden.

Um nun aber herauszufinden, welche der beiden Operatoren bei der Auswertung des Teilterms tatsächlich zur Anwendung kommen müssen, dient dann die Funktion `OperatorByChar`, die – gekapselt in `ADFunction` – den eigentlichen Delegaten zur Durchführung der Berechnung zurückliefert. Übergeben wird dieser Funktion das Operatorzeichen und die Priorität, und sie liefert anschließend die entsprechende `ADFunction`-Instanz zurück, mit der die Berechnung dann durchgeführt werden kann.

Auf diese Weise ist einerseits die beliebige Erweiterung durch zusätzliche Operatoren und andererseits die Einhaltung der Hierarchien garantiert.

Vererben der Klasse `ADFormularParser`, um eigene Funktionen hinzuzufügen

Sie können die Klasse vererben, um auf einfache Weise eigene Funktionen zum Auswerten in Formeln hinzuzufügen. Das umgebende Beispielprogramm demonstriert das anhand einer einfachen Funktion, die Sie dann im Ausdruck verwenden können, wenn Sie die zweite Schaltfläche zum Auswerten verwenden:

```
Private Sub btnCalculateEx_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnCalculateEx.Click
    Dim locFormPars As New ModifiedFormularParser(txtFormular.Text)
    lblResult.Text = locFormPars.Result.ToString
End Sub
End Class

Public Class ModifiedFormularParser
    Inherits ADFormularParser

    Sub New(ByVal Formular As String)
        MyBase.New(Formular)
    End Sub

    Public Overrides Sub OnAddFunctions()
        'Benutzerdefinierte Funktion hinzufügen.
        Functions.Add(New ADFunction("Double", AddressOf [Double], 1))
    End Sub

    Public Shared Function [Double](ByVal Args() As Double) As Double
        Return Args(0) * 2
    End Function
End Class
```

Die Klasse `ModifiedFormularParser` ist, wie hier im Code gezeigt, aus der `ADFormularParser`-Klasse hervorgegangen. Durch Überschreiben der Funktion `OnAddFunctions` können Sie eigene Funktionen der Klasse hinzufügen.

Beim Überschreiben der Klasse müssen Sie nur zwei Punkte beachten:

- Funktionen, die die Klasse zusätzlich behandeln sollen, müssen, wie hier im Beispiel zu sehen, als statische Funktionen eingebunden werden (sie müssen also mit dem Attribut `Shared` definiert werden).

In der Methode `OnAddFunctions`, die Sie überschreiben müssen, können Sie die Funktionsvorschriften durch das Instanziieren zusätzlicher `ADFunction`-Klasse der `Functions`-Auflistung hinzufügen.

Teil E

Entwicklungsvereinfachungen in Visual Basic 2008

In diesem Teil:

Eine philosophische Betrachtung der Visual Basic-spezifischen Vereinfachungen	823
Der My-Namespace	831
Das Anwendungs-Framework	849

Kapitel 28

Eine philosophische Betrachtung der Visual Basic- spezifischen Vereinfachungen

In diesem Kapitel:

- | | |
|---|-----|
| Die VB2008-Vereinfachungen am Beispiel DotNetCopy | 824 |
| Die prinzipielle Funktionsweise von DotNetCopy | 829 |

Es gibt exklusiv in Visual Basic 2008 einige Hilfsmittel (man könnte sie pessimistisch eingestellt auch als »Eigentümlichkeiten« bezeichnen), die auf den ersten Blick das OOP-Konzept des .NET Framework zu umgehen scheinen. Es gibt obendrein einige Dinge, bei denen OOP-Puristen den VB6lern der alten Riege vorwerfen, man hätte die neuen Hilfsmittel, die es erstmalig in VB 2005 gab, eigentlich nicht implementieren dürfen, weil es VB2005 wieder zurück in den Pool der »Baby-Programmiersprachen« wirft, und – Zitat – »man dürfe die Blödheit oder Sturheit der prozeduralen Fraktion nicht noch unterstützen«.

Ich gebe zu, ich sah mich eine Weile mehr in der letzteren Fraktion der OOP-Puristen als in der, der alten prozeduralen VB6-Riege. Und auch ich wollte nicht, dass mein schönes neues jetzt endlich auch erwachsenes VB.NET durch »nachgemachte« VB6-Features aufgeweicht und der Ruf »meines« VB 2005 dadurch vielleicht wieder angeknackst wird – Visual Basic hatte (und hat!) es schließlich in bestimmten Gemeinden eh schwer genug, sich als professionelle Programmiersprache zu etablieren.

Doch wissen Sie was? Diese ganze Diskussion ist eigentlich völlig überflüssig, und mit den folgenden Argumenten werde ich bewusst wahrscheinlich für Umsatzeinbußen in so manchen Kneipen sorgen, in denen Diskussionen über exakt dieses Thema geführt werden, denn:

OOP-Puristen, lasst euch gesagt sein: Visual Basic hat all das, was zum OOP-Programmieren benötigt wird. Und ich persönlich, der ich in C# und Visual Basic 2008 eigentlich gleichermaßen gut zurecht komme, programmiere in VB2008 lieber, weil meiner Meinung nach Visual Basic-Editor und -Background-Compiler ein schnelleres und effizienteres Arbeiten als die C#-Äquivalente gestatten. Wenn ihr Funktionalitäten, die in diesem Buchteil besprochen werden, wie beispielsweise das Anwendungsframework oder den My-Namespace nicht nutzen wollt – niemand wird dazu gezwungen! Macht euch die Arbeit halt schwerer als sie sein müsste – und braucht dann eben länger zum Ziel. Oder kurz gesagt: Ich finde Visual Basic einfach viel cooler!

Euer Hauptargument, dass für die Vereinfachung Komponenten verwendet werden, die eigentlich nicht zum .NET Framework gehören, ist – mit Verlaub gesagt – ziemlicher Unsinn. Die *VisualBasic.dll* ist Bestandteil des Frameworks, und alle Vereinfachungen, mit denen wir vielleicht auch die »alten« VB6ler überzeugen könnten, bei »uns mitzumachen«, sind dort implementiert. Und wenn ihr sagt, ihr wollt auf diese Vereinfachungen verzichten, weil ihr Gerüchten glaubt, die sagen, dass diese Visual Basic-eigenen Dinge in Zukunft aus dem Framework verbannt werden, dann dürft ihr übrigens gar nicht mehr in Visual Basic programmieren. Denn, und das ist wichtig: Ohne die *VisualBasic.dll* läuft überhaupt kein Visual Basic-Programm, egal welchen Ansatz (einfach oder .NET-puristisch) ihr verfolgt oder welcher Komponenten aus dem Framework ihr euch bedient. Und wenn ihr wisst, dass die *VisualBasic.dll* erstens sowieso genauso zentraler Bestandteil des Frameworks ist, wie alle anderen .NET-Assemblies und sich zweitens diese Vereinfachungen wie Anwendungsframework und My-Namespace auch dort befinden, riskiert doch mal einen Blick in die folgenden Seiten. Viele Probleme lassen sich damit wirklich einfach und schnell lösen – ich kann's am Beispiel beweisen!

Die VB2008-Vereinfachungen am Beispiel DotNetCopy

Jeder, der mit seinem Computer regelmäßig viele neue und wichtige Dateien generiert – und zu dieser Riege gehören wir Softwareentwickler ja zwangsläufig wohl alle – weiß, wie wichtig Datensicherungen sind. Aber getreu dem Motto »Der Schuster hat die schlechtesten Schuhe« gibt's hier und da den einen oder anderen, der dieses Gebiet ein wenig vernachlässigt – ist's nicht so? Der Autor dieser Zeilen will sich da gar nicht von

frei sprechen. Ich persönlich kopiere meine Daten jeden Abend auf einen Backup-Rechner, und das mithilfe eines simplen Batchs, der in der Eingabeaufforderung von Windows XP läuft. Eine regelrechte differenzierende Bandsicherung mit einem Band für jeden Wochentag empfehle ich lediglich meinen Kunden ...

So wäre es immerhin ein Fortschritt, gäbe es ein Programm, das die Dateien bestimmter Verzeichnisse nicht nur kopiert, sondern auch beliebige Backup-Historien der Dateien erstellt, die vor einem erneuten Sichern einer aktualisierten Datei versionsmäßig »nach oben« rutschen, wie etwa in Abbildung 28.1 zu sehen.

Das Beispiel zu diesem Buchteil kann genau das, und es kann noch viel, viel mehr: Sie können DotNetCopy so einstellen, dass es nur Dateien kopiert, die im Zielverzeichnis entweder nicht vorhanden oder älter sind als die Dateien, die es zu sichern gilt. Und im Bedarfsfall werden die im Zielverzeichnis vorhandenen Dateien nicht einfach ersetzt, sodass die alte Version auf Nimmerwiedersehen verschwindet. Vielmehr können Sie bestimmen, wie viele Backup-Stufen der älteren Dateien automatisch vorgehalten werden sollen – Abbildung 28.1 zeigt auch das in Form einer Dateien-Backup-Historie, wie das später auf dem Backup-Laufwerk aussieht.

BEGLEITDATEIEN

Sie finden dieses Projekt übrigens unter dem Namen *DotNetCopy.sln* im Verzeichnis

...\\VB 2008 Entwicklerbuch\\E - Vereinfachungen\\Kapitel 28\\DotNetCopy

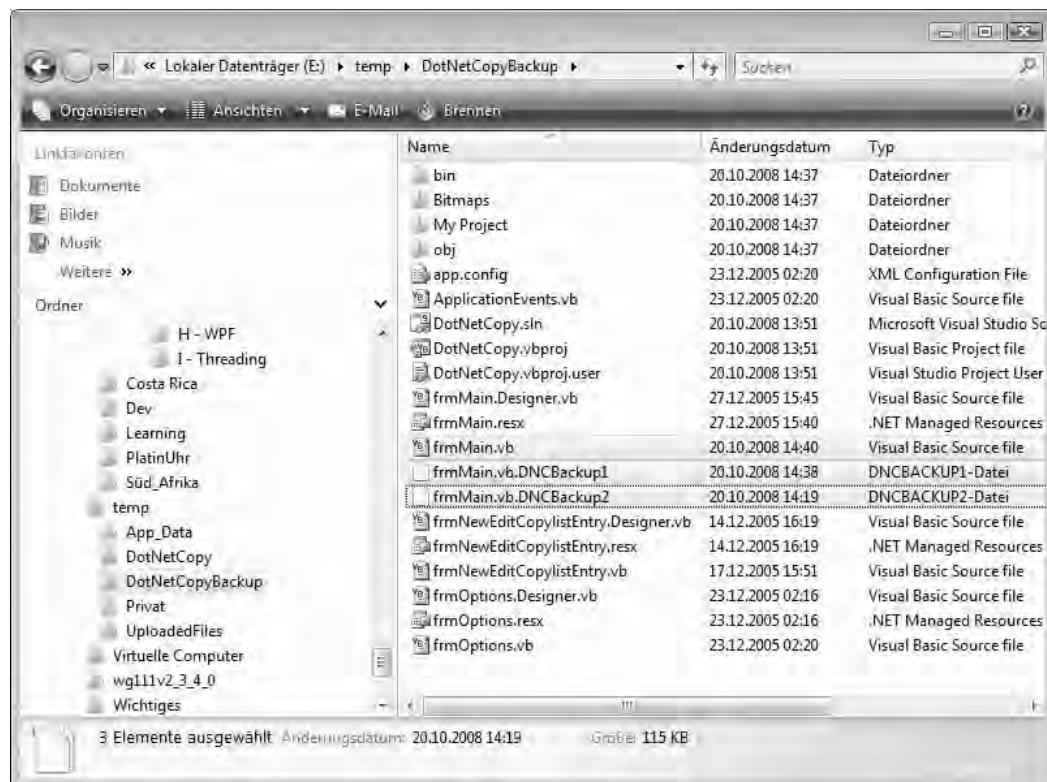


Abbildung 28.1 DotNetCopy kopiert Dateien im Bedarfsfall nur, wenn eine Datei neuer ist als eine vorhandene im Zielverzeichnis und legt automatisch Backup-Historien an

Der Optionsdialog von *DotNetCopy* erlaubt es, das Kopierverhalten genau zu kontrollieren:

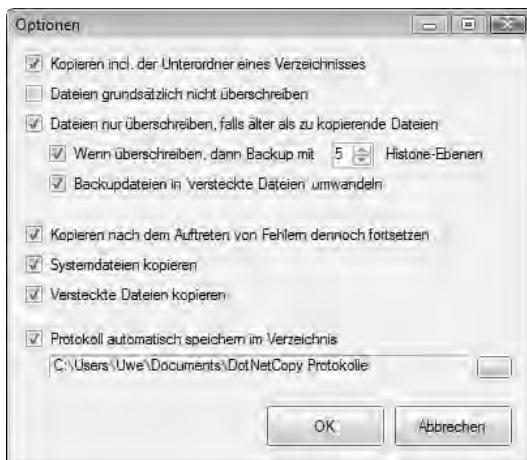


Abbildung 28.2 DotNetCopy erlaubt es, bestimmte Kopieroptionen im Optionen-Dialog zu konfigurieren

Sie erstellen mit DotNetCopy anschließend eine Kopierliste, die das Programm darüber informiert, welches Quellverzeichnis wohin kopiert (also gesichert) werden soll. Kopierlisten, die Sie auf diese Art und Weise erstellen, lassen sich sogar abspeichern und laden, was die Anwendung des Programms recht flexibel macht:

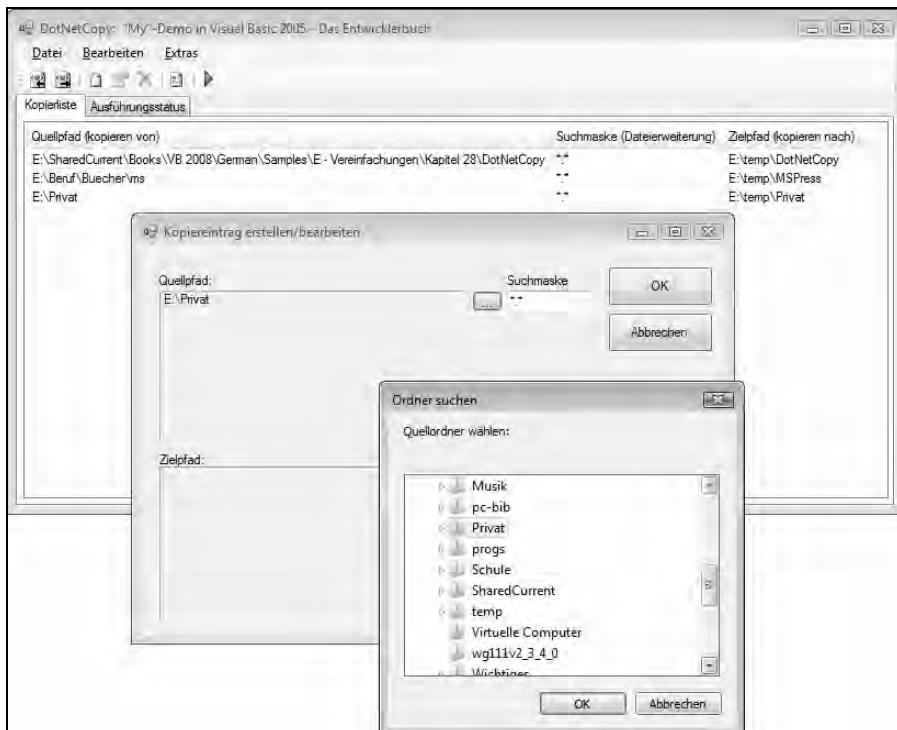


Abbildung 28.3 Erstellen Sie mit DotNetCopy Kopierlisten, die beliebig viele Quell- und Zielpfade enthalten und geladen sowie gespeichert werden können

Und wenn Sie eine Kopierliste erstellt und für die spätere Wiederverwendung gespeichert haben, drücken Sie einfach aufs Knöpfchen, und der Kopievorgang kann beginnen, wie in der folgenden Grafik zu sehen:

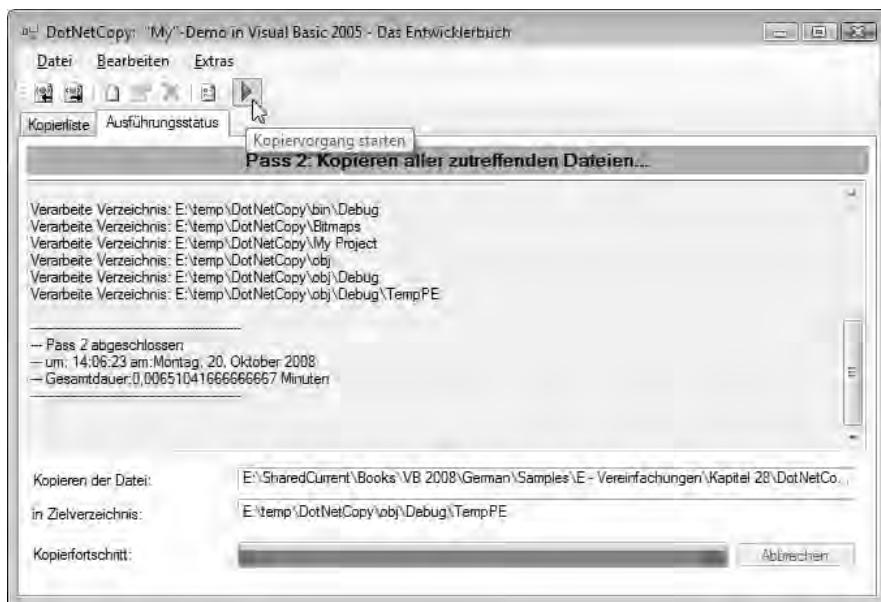


Abbildung 28.4 DotNetCopy nach erfolgreichem Backup-Lauf

DotNetCopy mit /Autostart und /Silent-Optionen

Damit dieses Tool aber auch für Administratoren interessant ist, die es vielleicht in eigene Batch-Dateien an der Befehlszeile einbinden möchten, lässt es sich auch im Autostart- sowie im Silent-Modus starten.

- Der Autostart-Modus erlaubt es, eine Kopierlistendatei beim Aufruf des Programms anzugeben. Diese Kopierlistendatei wird dann anschließend geladen und der Backup-Vorgang beginnt unmittelbar. Der Aufruf von DotNetCopy im Kopierlistenmodus erfolgt dazu an der Eingabeaufforderung oder über Ausführen im Start-Menü. Sie geben zusätzlich zum Programmnamen *DotNetCopy* die Option */Autostart:kopierlistendatei.ols* an, also beispielsweise

```
Dotnetcopy /autostart:backup.ols
```

- Zusätzlich können Sie das Programm so starten, dass es ohne sichtbare Benutzeroberfläche läuft. In diesem Fall verwenden Sie die Option */silent*, die Sie durchaus mit der zuvor beschriebenen Option */autostart* kombinieren können. Um also beispielsweise die obige als Beispiel gezeigte Kopierliste automatisch anzeigen zu lassen, würden Sie die Befehlszeile

```
Dotnetcopy /autostart:backup.ols /silent
```

dazu verwenden können.

Nachdem Sie die Funktionsweise von DotNetCopy kennengelernt haben, geht's als nächstes darum zu erfahren, was die Kapitel dieses Buchs mit der ganzen Geschichte zu tun haben.

Und falls Sie denken, Sie müssten zum Verständnis des Programms tief in die Abgründe des .NET Framework hinabsteigen – weit gefehlt. DotNetCopy benötigt dank der Visual Basic-eigenen Vereinfachungen überraschend wenig Code.

Und selbst Visual Basic 6-Umsteigern werden lediglich zwei Klassen des Framework neu sein – fast die gesamte andere Funktionalität basiert komplett auf Funktionalitäten, die durch den My-Namespace und das Anwendungsframework zur Verfügung gestellt werden.

Exkurs: Die Klassen FileInfo- und DirectoryInfo zur Pflege von Verzeichnissen und Dateien

Um Zugriff auf Dateien und Verzeichnisse der Festplatte oder eines anderen Datenträgers Ihres Computers zu nehmen, stellt Ihnen das .NET Framework die Klassen `FileInfo`- und `DirectoryInfo` zur Verfügung.

Die `FileInfo`-Klasse dient zur »Bearbeitung« von Dateien im Sinne des Windows-Explorers; `DirectoryInfo` stellt im Gegensatz dazu die äquivalente Funktionalität für Verzeichnisse zur Verfügung: Stellen Sie sich beide Klassen am besten wie einen Windows-Datei-Explorer vor, der nur eben keine sichtbare Benutzeroberfläche hat: Mit der `FileInfo`-Klasse haben Sie beispielsweise die Möglichkeit, Attribute von Dateien abzurufen, Dateien umzubenennen, sie an einen anderen Ort zu verschieben, auf deren Vorhandensein zu überprüfen und Ähnliches. Mit `DirectoryInfo` arbeiten Sie prinzipiell genau so – nur eben mit Verzeichnissen.

Das Schwierige bei Dateioperationen ist in der Regel gar nicht so sehr das Aufrufen bestimmter Funktionen, wenn Sie erst einmal den exakten Pfad und Dateinamen einer Datei kennen.

Es ist das richtige Zusammenbauen des Pfades, das Extrahieren des eigentlichen Dateinamens aus der Zeichenkettenkombination von Pfad und Dateinamen oder auch das Ermitteln der Dateiendung einer kompletten Pfad-/Dateinamenkombination. Und auch dabei helfen Ihnen die Klassen `FileInfo`- und `DirectoryInfo`. Beim Erstellen neuer Unterordner in mehreren Verzeichnisebenen brauchen Sie sich obendrein auch nicht darum zum kümmern, ob Ordner in Teilebenen schon existieren – wenn Sie beispielsweise mit der `DirectoryInfo`-Klasse ein Verzeichnis namens `C:\Ordner1\UnterOrdner\EigentlicherOrdner` anlegen wollen, werden im Bedarfsfall alle benötigten Ordner (`Ordner1, UnterOrdner`) gleich mit angelegt. Die folgenden Codezeilen geben einige Beispiele für die Anwendung der `FileInfo`- bzw. `DirectoryInfo`-Klasse:

Beispiel DirectoryInfo-Objekt:

```
'Neues DirectoryInfo-Objekt aus Pfad-Zeichenkette erstellen:  
Dim einDirInfo As New DirectoryInfo("C:\Ordner1\Unterordner")  
'Existiert das Verzeichnis?  
If Not einDirInfo.Exists Then  
    Debug.Print("Ordner existiert nicht, wird erstellt:")  
    'Verzeichnis mit allen benötigten Unterverzeichnissen erstellen  
    einDirInfo.Create()  
End If
```

Beispiel FileInfo-Objekt:

```
'Neues FileInfo-Objekt aus Pfad-/Dateinamenzeichenkette erstellen  
Dim einFileInfo As New FileInfo("c:\Order1\Unterordner\textfile.txt")
```

```
'Jedes FileInfo-Objekt enthält auch ein DirectoryInfo-Objekt, das
'über die Directory-Eigenschaft abrufbar ist:
If Not einFileInfo.Directory.Exists Then
    'Pfad zur Datei im Bedarfsfall erstellen
    einFileInfo.Directory.Create()
End If
'Auf Vorhandensein der Datei prüfen:
If Not einFileInfo.Exists Then
    'Eine simple Textdatei erstellen.
    My.Computer.FileSystem.WriteAllText(einFileInfo.FullName, "Dateiinhalt", False)
End If
Debug.Print("Die Erweiterung der Datei lautet: " & einFileInfo.Extension)
Debug.Print("Der reine Dateiname lautet: " & einFileInfo.Name)
Debug.Print("Voller Pfad und Dateiname zur Datei lauten: " & einFileInfo.FullName)
Debug.Print("Die Datei hat folgende Attribute: " & einFileInfo.Attributes)
Debug.Print("Die Datei wird jetzt ins Hauptverzeichnis kopiert!")
'Wichtig: Pfad und neuer Dateiname müssen als Ziel beide angegeben werden!
einFileInfo.CopyTo("C:\\" & einFileInfo.Name)
Debug.Print("Die Ausgangsdatei wird gelöscht!")
einFileInfo.Delete()
```

Dieser Code würde folgende Ausgabe erzeugen:

```
Die Erweiterung der Datei lautet: .txt
Der reine Dateiname lautet: textfile.txt
Voller Pfad und Dateiname zur Datei lauten: c:\Order1\Unterordner\textfile.txt
Die Datei hat folgende Attribute: 32
Die Datei wird jetzt ins Hauptverzeichnis kopiert!
Die Ausgangsdatei wird gelöscht!
```

Die prinzipielle Funktionsweise von DotNetCopy

Wie gesagt: DotNetCopy dient dazu, die beiden Besonderheiten von Visual Basic 2005 in Vergleich zu den anderen .NET Framework-Programmiersprachen zu demonstrieren. Dazu gehören die Funktionalitäten, die durch das so genannte Anwendungsframework zur Verfügung gestellt werden, sowie der My-Namespace, der die Aufgabe hat, bestimmte thematisch kategorisierte Probleme, die Ihnen bei den täglichen Entwicklungsaufgaben immer wieder begegnen, einfacher lösen zu können.

Beim Anwendungsframework, dessen Funktionalität und Handhabung das folgende Kapitel beschreibt, bedient sich DotNetCopy dreierlei Dinge: Dem Zur-Verfügung-Stellen der Visual Styles von Windows XP, den Anwendungseignissen und einer Funktion, die verhindert, dass eine zweite Instanz von DotNetCopy gestartet werden kann, wenn eine Instanz von DotNetCopy bereits aktiv ist.¹

¹ Ob dieser letzte Punkt für das richtige Funktionieren von DotNetCopy tatsächlich erforderlich ist, darüber lässt sich sicherlich streiten – vielleicht möchte auch der ein oder andere, dass DotNetCopy in mehreren Instanzen gleichzeitig laufen kann. Für die Demonstration des Anwendungsframeworks ist es alle Male geeignet.

Da DotNetCopy die Option */Silent* kennt, muss es einen Mechanismus geben, der quasi noch vor dem Anzeigen des ersten Formulars greift, und der eben in diesem Modus verhindert, dass das erste Formular angezeigt wird. Das Anwendungsframework stellt zu diesem Zweck eine Reihe von Ereignissen zur Verfügung, und unter anderem auch ein Ereignis, das ausgelöst wird, nachdem das Programm gestartet ist. Ob es zur Anzeige des Hauptformulars kommt oder nicht, wird in dieser Ereignisbehandlungsroutine namens *MyApplication_Startup* geregelt, die sich in der Codedatei *ApplicationEvents.vb* des Projektes befindet.

Im Silent-Modus wird anschließend zwar das Hauptformular, das auch die komplette Funktionalität für den Backup-Vorgang in *frmMain.vb* enthält, instanziert, aber nicht dargestellt. Für die korrekte Funktionsweise ist das jedoch kein Nachteil – selbst der Code für die Formularaktualisierungen (Fortschrittsanzeige und Protokollausgabe aktualisieren, aktuell bearbeitetes Verzeichnis anzeigen etc.) funktioniert, nur dass der Anwender sie während des Backup-Vorgangs nicht sieht, da das Formular selbst eben nicht angezeigt wird.

Die weitere Funktionsweise ist daher auch in allen Modi gleich. Nachdem eine Kopierliste geladen wurde – die Funktion *LoadCopyEntryList* ist dafür zuständig, und sie wird entweder durch die Funktion *HandleAutoStart* (ihrerseits durch *MyApplication_Startup* aufgerufen) oder durch das Auswählen des entsprechenden Menüpunkts oder Symbols vom Anwender getriggert – startet der Kopievorgang in der Methode *CopyFiles* (auch entweder automatisch durch *HandleAutoStart* oder durch das Auswählen des entsprechenden Menüpunkts oder Symbols durch den Anwender).

Und in *CopyFiles* und allen Methoden, die dort aufgerufen werden, kommt das zweite Thema dieses Buchteils intensiv zum Einsatz – Funktionen aus dem My-Namespace.

Wenn Sie einen Blick in die Projektdateien werfen, werden Sie sehen, dass das Programm ausführlich kommentiert ist. Zudem werden Sie die wichtigen Ausschnitte in den nächsten Kapiteln wiederfinden.

Obendrein haben Sie durch .NET-Copy nicht nur ein anschauliches Beispiel, wie einfach sich auch komplexere Anwendungen mit VB und den »VB-Vereinfachungen« entwickeln lassen, sondern auch ein Tool, das der eine oder andere von Ihnen vielleicht auch wirklich gebrauchen kann. Bei mir hat DotNetCopy jedenfalls seit seiner Fertigstellung meine alten Batch-Prozesse abgelöst, und die Backup-Historie von Dateien, die es generiert, habe ich schon mehr als einmal gut gebrauchen können.

Und vielleicht noch eine letzte Anmerkung zum Thema OOP und den Visual Basic-Vereinfachungen: DotNetCopy ist sicherlich kein Glanzbeispiel für OOP. Aber es ist brauchbar, stabil und war schnell entwickelt. Das gleiche Tool in der Sprache C#, in der es die VB-Vereinfachungen nicht gibt, hätte sicherlich einen Tag mehr Entwicklungsaufwand bedeutet. OOP ist gerade bei größeren Anwendungen sicherlich ein Muss, auch in Visual Basic und trotz der Tatsache, dass es My und das Anwendungsframework gibt. Aber OOP verleitet auch dazu, mehr zu machen, als eigentlich gefordert ist. Denken Sie daran, dass Sie für ein Einfamilienhaus auch kein Fundament für einen Wolkenkratzer ausheben würden, und diese Analogie sollten Sie im Hinterkopf behalten, wenn Sie die Klassenplanung für ein größeres Projekt in Angriff nehmen.

Es ist nichts Verwerfliches dabei, die speziellen VB-Vereinfachungen zu verwenden. Wenn Sie damit Zeit sparen können, und absehen können, dass Sie sich damit für spätere Erweiterungen den Weg nicht verbauen, nutzen Sie sie. Und lassen Sie die C#- oder VB-Entwickler, die Ihnen etwas anderes weismachen wollen, ruhig reden ...

Kapitel 29

Der My-Namespace

In diesem Kapitel:

Formulare ohne Instanziierung aufrufen	833
Auslesen der Befehlszeilenargumente mit My.Application.CommandLineArgs	835
Gezieltes Zugreifen auf Ressourcen mit My.Resources	837
Internationalisieren von Anwendungen mithilfe von Ressource-Dateien und dem My-Namespace	840
Vereinfachtes Durchführen von Dateioperationen mit My.Computer.FileSystem	843
Verwenden von Anwendungseinstellungen mit My.Settings	846

Der My-Namespace wurde in Visual Basic 2005 eingeführt, um den Umgang mit bestimmten Funktionalitäten zu vereinfachen – um sozusagen »Abkürzungen« zu bestimmten Zielen mit Funktionalitäten im .NET Framework zur Verfügung zu stellen, die Sie normalerweise nur auf verschlungenen Pfaden erreichen würden.

So Sie das Beispielszenario aus Kapitel 5 komplett durchexerziert haben, sind Sie auch schon in den Genuss dieser Abkürzungen gekommen – Sie haben dort nämlich kennen gelernt, wie einfach Sie auf selbst speichernde Anwendungseinstellungen durch `My.Settings` zurückgreifen können.

Doch der My-Namespace verfügt über noch weit mehr wirklich coole Features. Welche das allerdings genau sind, lässt sich nicht verbindlich sagen, denn die Features, die Ihnen durch My zur Verfügung stehen, hängen von dem Projekttyp ab, den Sie gerade bearbeiten.

Generell teilt sich die Funktionalität im My-Namespace in die folgenden Kategorien auf:

- **My.Application:** Stellt Zugriff auf Informationen über die aktuelle Anwendung bereit, wie beispielsweise den Pfad zur ausführbaren Datei, die Programmversion, derzeitige Kulturinformationen oder den Benutzer-Authentifizierungsmodus.
- **My.Computer:** Ermöglicht Ihnen den Zugriff auf mehrere untergeordnete Objekte, die Ihnen wiederum das Abrufen von computerbezogenen Informationen gestatten, wie beispielsweise über das verwendete Dateisystem, über verwendete Audio- und Videospezifikationen, angeschlossene Drucker oder generelle I/O-Hardware wie Maus, Tastatur, Speicher, die verwendete Netzwerkumgebung, serielle Schnittstellen und Weiteres.
- **My.Forms:** Gestattet Ihnen das Abrufen von Standardinstanzen aller Formulare einer Windows Forms-Anwendung.
- **My.Resources:** Ermöglicht den einfachen Zugriff auf eingebettete Ressourcen, wie beispielsweise auf Zeichenkettentabellen, Bitmaps und Ähnliches.
- **My.Settings:** Ermöglicht einerseits, auf Anwendungseinstellungen Zugriff zu nehmen, und sorgt andererseits dafür, dass diese Anwendungseinstellungen im Bedarfsfall benutzerabhängig beim Programmstart wiederhergestellt und beim Programmende gesichert werden.
- **My.User:** Ermöglicht das Abrufen von Infos über den zurzeit angemeldeten Benutzer und erlaubt ferner das Implementieren benutzerdefinierter Authentifizierungsmechanismen.
- **My.WebServices:** Stellt eine Eigenschaft für jeden Webservice zur Verfügung, den das aktuelle Projekt referenziert, und erlaubt so auf Webservices Zugriff zu nehmen, ohne eine explizite Proxy-Klasse für den jeweiligen Webservice erstellen zu müssen.

In einer Konsolenanwendung, die nun einmal keine Formulare verwendet, steht Ihnen natürlich die Kategorie `My.Forms` nicht zur Verfügung. Und so hängt die tatsächliche Verfügbarkeit der My-Features durchweg von den Projekttypen ab, mit denen Sie es gerade zu tun haben, wie die folgende Tabelle zeigt:

My-Objekt	Windows-Anwendung	Klassenbibliothek	Konsolenanwendung	Windows-Steuerelementbibliothek	Web-Steuerelementbibliothek	Windows-Dienst
My.Application	Ja	Ja	Ja	Ja	Nein	Ja
My.Computer	Ja	Ja	Ja	Ja	Ja	Ja
My.Forms	Ja	Nein	Nein	Ja	Nein	Nein ►

My-Objekt	Windows-Anwendung	Klassenbibliothek	Konsolenanwendung	Windows-Steuerelementbibliothek	Web-Steuerelementbibliothek	Windows-Dienst
My.Resources	Ja	Ja	Ja	Ja	Ja	Ja
My.Settings	Ja	Ja	Ja	Ja	Ja	Ja
My.User	Ja	Ja	Ja	Ja	Ja	Ja
My.WebServices	Ja	Ja	Ja	Ja	Ja	Ja

Tabelle 29.1 Verfügbarkeit der Bereiche des My-Namespace bei den verschiedenen Projekttypen

Nun finde ich, es würde an dieser Stelle keinen Sinn machen, die komplette Referenz aller Funktionalitäten jedes einzelnen Bereichs herunterzubeten – im Gegenteil: Schließlich dient `My` eben genau dazu, auch ohne großes Blättern und nur mithilfe von IntelliSense und dynamischer Hilfe, schnell an die gewünschte Funktionalität zu gelangen.

Stattdessen sollten Sie gerade von einem Entwicklerbuch erwarten können, dass Ihnen der Autor Sachverhalte im Rahmen einer Softwareentwicklung näher bringt – und das hat er für die Demonstration von `My` auch getan. Die in den folgenden Abschnitten beschriebenen `My`-Funktionalitäten erheben deshalb keinen Anspruch auf Vollständigkeit – sie sollen es auch gar nicht. Ziel ist es vielmehr, zum einen generelle Vorgehensweisen beim Umgang mit dem `My`-Namespace zu vermitteln. Die richtigen Funktionen für Ihre eigenen Bedürfnisse zu finden, wird dank IntelliSense und Online-Hilfe sicherlich nicht *das* Problem sein, und würde an dieser Stelle nur wertvollen Platz beschränken.

Es gibt zum anderen auch einiges Erwähnenswertes zum `My`-Namespace und seinen Funktionalitäten, was Sie nicht in der Online-Hilfe finden, und auch diesen Punkten widmen sich die folgenden Abschnitte.

BEGLEITDATEIEN Viele der hier gezeigten Features können Sie sich direkt am Beispiel von `DotNetCopy` (vorgestellt im vorherigen Kapitel) anschauen, und innerhalb der Abschnitte werden Sie auch immer wieder Codeausschnitte aus diesem Beispiel finden. Sie finden dieses Projekt unter dem Namen `DotNetCopy.sln` im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\E - Vereinfachungen\\Kapitel 28\\DotNetCopy

So Sie das vorherige Kapitel noch nicht gelesen haben, sollten Sie es zunächst tun, um die Beispiele besser nachvollziehen zu können.

Formulare ohne Instanziierung aufrufen

Die Änderungen an Visual Basic, um die es jetzt geht, haben in der bis dato etablierten VB.NET-Gemeinde für das meiste Aufsehen gesorgt,¹ denn: Es geht um die Vereinfachung von Formularaufrufen, und das Hauptargument dabei ist: Damit nähert sich Visual Basic politisch betrachtet wieder einer »Kindersprache«, bei der man dem »unmündigen« VB-Programmierer nicht zumuten kann, die buchstäblichen Basics der OOP zu kennen und anzuwenden.

¹ Vergleichen Sie dazu bitte auch den Einführungstext zu diesem Buchteil.

Worum geht es genau?

Auch schon in VB6 mussten Sie, um mit einem Objekt arbeiten zu können, es zunächst instanzieren. Haben Sie also eine Collection verwendet, in der Sie andere Objekte speichern wollten, waren folgende Zeilen notwendig:

```
Dim auflistung As Collection  
Set auflistung = New Collection  
auflistung.Add 5
```

Es ist klar, dass diese Zeilen nicht funktioniert hätten:

```
Collection.Add 10  
Collection.Add 15  
Collection.Add 20
```

Bei VB6-Formularen passiert aber genau das: Hier können Klassenname und Instanz das Gleiche sein. Diese Version funktioniert:

```
Dim f As Form1  
f.Show
```

genau wie die Verwendung der Klasse Form1 als Instanzvariable:

```
Form1.Show
```

Das entspricht natürlich nicht den Vorschriften zur objektorientierten Programmierung, war aber wahrscheinlich für viele VB6-Programmierer einfacher zu verstehen, und aus diesem Grund wurde der Basic-Compiler so frisiert, dass eine solche Verwendung einer Formularklasse möglich war.

Schon in Visual Basic 6 passierte »unter der Haube« dazu etwas, was nunmehr auch seit VB2005 wieder möglich war.

Der VB-Compiler sorgt nämlich bei der Kompilierung einer Windows Forms-Anwendung dafür, dass versteckter Quellcode, der im Compiler fest verdrahtet ist, mit in Ihre Anwendung kompiliert wird. Der VB-Compiler »erfindet« also quasi einen Haufen von Quellcode und fügt diesen, unsichtbar für den Entwickler, dem eigentlichen Windows-Projekt hinzu.

So können Sie auch in Visual Basic 2008 wieder folgende Zeile verwenden, um das unter dem Klassennamen Form2 erstellte Formular beispielsweise als modalen Dialog ins Leben zu rufen:

```
Form2.ShowDialog
```

Doch es sieht hier nur so aus, als würden Sie eine statische Klassenmethode direkt verwenden, denn in Wirklichkeit ist Form2 eine Eigenschaft, die eine Instanz vom Typ Form2 zurückliefert. Denn was hier eigentlich passiert, ist, dass der Compiler diese Zeile nur vervollständigt, und er im Grunde genommen folgenden Quellcode für die Generierung der eigentlichen ausführbaren Datei kompiliert:

```
MyProject.Forms.Form2.ShowDialog
```

Die Klasse MyProject ist dabei allerdings eine Klasse, die Sie unter diesem Namen nicht erreichen können, weil sie mit entsprechenden Attributen so gekennzeichnet ist, dass IntelliSense sie nicht »sieht«. Sie ist aber dennoch vorhanden. Quasi versteckt legt der Compiler also für jedes Formular, das Sie Ihrem Projekt hinzufügen, eine Eigenschaft vom Typ des jeweiligen Formulars in der (ebenfalls unsichtbaren) MyForms-Klasse an, die eine eingebettete Klasse der Klasse MyProject selbst ist, deren Instanz durch die Forms-Eigenschaft der MyProject-Klasse abgerufen werden kann. Und erst diese Eigenschaft, die den Namen des Formulars trägt, sorgt dafür, dass die gewünschte Formulkarklasse vor Gebrauch instanziert und dann ohne Probleme (bzw. NullReference-Ausnahmen) verwendet werden kann.

Sie können das ausprobieren. Tippen Sie **My.**, dann werden Sie sehen, dass IntelliSense MyProject nicht zur Auswahl anbietet.

Komplettieren Sie jedoch die Zeile mit **MyProject.Forms.Form2.ShowDialog**, meldet der Visual Basic-Compiler dennoch keinen Fehler.

Zur Veranschaulichung: Die Programmzeile

```
MyProject.Forms.Form2.ShowDialog
```

bedeutet im Grunde genommen:

```
(MyProject-Klasse). (Forms-Eigenschaft der MyProject-Klasse vom Typ MyForms).  
(Form2-Eigenschaft der MyForms-Klasse vom Typ Form2). (ShowDialog-Methode der Form2-Klasse).
```

Auslesen der Befehlszeilenargumente mit My.Application.CommandLineArgs

Anwendungen, die unter Windows laufen, können Sie auf unterschiedliche Weise starten. Die einfachste und bekannteste ist, sie durch einen Doppelklick auf eine Verknüpfung beispielsweise aus dem Startmenü zum Laufen zu bewegen. Sie können jede Windows- oder Befehlszeilenanwendung aber auch direkt durch die Angabe des Namens der ausführbaren Datei aus der Befehlszeile oder durch den *Ausführen*-Befehl des Start-Menüs heraus in Gang bekommen. Und viele Anwendungen erlauben es hier, weitere Parameter anzugeben, die das Startverhalten steuern. So starten Sie beispielsweise den Windows-Explorer mit

```
Explorer c:\
```

um zu definieren, dass der Explorer nicht nur gestartet wird, sondern dass er auch direkt den Inhalt des Wurzelverzeichnisses von Laufwerk *c:* anzeigt.

Parameter, die Sie Anwendungen auf diese Weise übergeben, nennt man Befehlszeilenargumente, und mithilfe des **My**-Namespace können Sie Befehlszeilenargumente auf einfache Weise ermitteln und auswerten.

Das **My**-Beispiel DotNetCopy arbeitet ebenfalls mit solchen Befehlszeilenargumenten, nämlich um die Anwendung in den Autostart- oder Silent-Modus zu versetzen. Der folgende Code, der sich im StartUp-Ereignis des **MyApplication**-Objektes befindet, und der ausgelöst wird, sobald die Anwendung startet, demonstriert den Umgang mit Befehlszeilenargumenten.

HINWEIS Sie finden diesen Code in der Datei *ApplicationEvents.vb* des Projektes. Mehr zu Anwendungseignissen erfahren Sie übrigens im nächsten Kapitel.

```
Partial Friend Class MyApplication

    Private Sub MyApplication_Startup(ByVal sender As Object, ByVal e As
        Microsoft.VisualBasic.ApplicationServices.StartupEventArgs) Handles Me.Startup

        'Das Verzeichnis für die Protokolldatei beim ersten Mal setzen...
        If String.IsNullOrEmpty(My.Settings.Option_AutoSaveProtocolPath) Then
            My.Settings.Option_AutoSaveProtocolPath =
                My.Computer.FileSystem.SpecialDirectories.MyDocuments & "\DotNetCopy Protokolle"
            Dim locDi As New DirectoryInfo(My.Settings.Option_AutoSaveProtocolPath)

            'Überprüfen und
            If Not locDi.Exists Then
                'im Bedarfsfall anlegen
                locDi.Create()
            End If

            'Settings speichern
            My.Settings.Save()
        End If

        Dim locFrmMain As New frmMain

        'Kommandozeile auslesen
        'Sind überhaupt Befehlszeilenargumente vorhanden?
        If My.Application.CommandLineArgs.Count > 0 Then
            'Durch jedes einzelne Befehlszeilenargument durchiterieren
            For Each locString As String In My.Application.CommandLineArgs

                'Alle unnötigen Leerzeichen entfernen und
                'Groß-/Kleinschreibung 'Unsensibilisieren'
                'HINWEIS: Das funktioniert nur in der Windows-Welt;
                'kommt die Kopierlistendatei von einem Unix-Server, bitte darauf achten,
                'dass der Dateiname dafür auch komplett in Großbuchstaben gesetzt ist,
                'da Unix- (und Linux-) Derivate Groß-/Kleinschreibung berücksichtigen!!!
                locString = locString.ToUpper.Trim

                If locString = "/SILENT" Then
                    locFrmMain.SilentMode = True
                End If

                If locString.StartsWith("/AUTOSTART") Then
                    locFrmMain.AutoStartCopyList = locString.Replace("/AUTOSTART:", "")
                    locFrmMain.AutoStartMode = True
                End If
            Next
        End If
    .
    .

```

Gezieltes Zugreifen auf Ressourcen mit My.Resources

Ressource-Dateien speichern bestimmte Elemente einer Anwendung in einer separaten Datei. Bei solchen Elementen handelt es sich beispielsweise um Bitmaps, Symbole, Zeichenketten oder Ähnliches. Diese Elemente können mithilfe von My.Resources zur Laufzeit aus Ressource-Dateien gelesen und anschließend im Kontext verwendet werden.

Der Vorteil dieser Vorgehensweise: Anwendungen können auch im Nachhinein angepasst werden, ohne dem Bearbeiter den eigentlichen Code der Anwendung zur Verfügung zu stellen. Denken Sie dabei nur zum Beispiel an die Lokalisierung einer Anwendung, wie bei Microsoft Office-Programmen wie Word zum Beispiel, in eine andere Sprache. Wären die Texte der Anwendung fest verdrahtet, müsste man den Übersetzern die kompletten Quellen der Anwendung zur Verfügung stellen und die entsprechenden Texte, die dann als Zeichenketten irgendwo im Quellcode eingebettet sind, heraus suchen. Ein weiterer Nachteil wäre, dass es verschiedene Versionen des Quellcodes gäbe, von denen jede einzelne bei Änderungen oder Fehlerbehebungen in der Anwendung bearbeitet werden müsste – ein Aufwand, der bei größeren Projekten nicht zu leisten ist.

Anlegen und Verwalten von Ressource-Elementen

In unserer Beispielanwendung werden die Texte für die Spaltennamen der ListView, die die Kopierlisteneinträge beinhaltet, aus einer Ressource-Datei entnommen. Die Ressource-Datei selber wird ähnlich verwaltet, wie Sie es bei den Anwendungseinstellungen (den *Application Settings*) bereits im 5. Kapitel kennen gelernt haben.

- Rufen Sie, um die Ressource-Datei einer Anwendung zu bearbeiten, einfach die Projekteigenschaften über das Kontext-Menü des Projektmappen-Explorers auf.
- Wählen Sie anschließend die Registerkarte *Ressourcen*.
- Wählen Sie aus der linken Schaltfläche, die Sie aufklappen können (siehe Abbildung 29.1), den Typ Ressource, den Sie einpflegen möchten. Bei Zeichenketten erfassen Sie die Texte wie bei Anwendungseinstellungen in Form von Texttabellen. Bei anderen Ressource-Typen, wie beispielsweise Bitmaps, Symbolen oder Audio-Clips, verwenden Sie die Schaltfläche *Ressource hinzufügen*, um eine entsprechende Datei der Ressource-Datei hinzuzufügen.

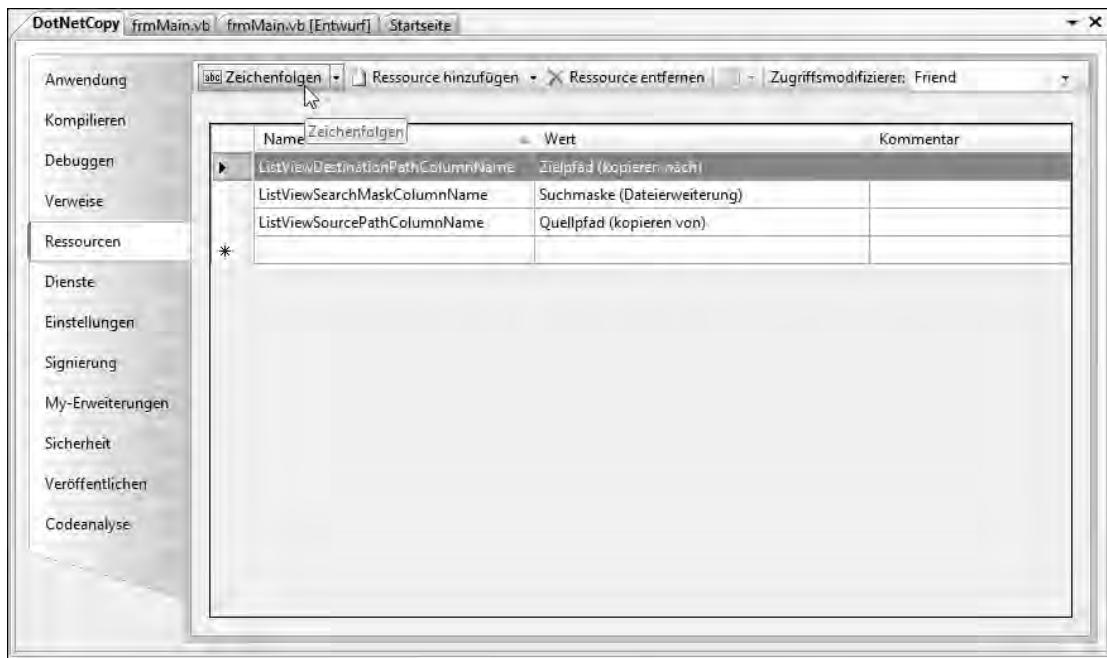


Abbildung 29.1 Bestimmen Sie wie hier den Typ Ressource, den Sie der Ressource-Datei hinzufügen möchten. Bei anderen Typen als Zeichenketten verwenden Sie Ressource hinzufügen, um Ressource-Elemente wie Bitmaps, Symbole oder Audio-Clips der Ressource-Datei hinzuzufügen.

Abrufen von Ressourcen mit My.Resources

Wenn Sie Ressourcen auf die im letzten Abschnitt beschriebene Weise Ihrem Projekt hinzugefügt haben, ist das Abrufen der Ressource-Elemente zur Laufzeit dank *My* ein Einfaches. Im Beispielprojekt DotNetCopy wird das beispielsweise im Ereignisbehandler `form_Load` des Formulars gemacht; hier werden beispielhaft die Spaltentexte der `ListView` eingerichtet:

```

    ''' <summary>
    ''' Wird aufgerufen, wenn das Formular geladen wird, und enthält die
    ''' Initialisierung der ListView sowie das Anstoßen des Kopievorgangs
    ''' (und das zuvor notwendige Laden der Kopierliste), wenn das Programm
    ''' im Autostart (aber nicht im Silent) -Modus läuft.
    ''' (Silent-Modus wird in ApplicationEvents.vb gehandelt).
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
Private Sub frmMain_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles Me.Load
    'Listview einrichten
    With Me.lvwCopyEntries.Columns
        'Die Texte für die ListView-Spalten aus der Ressource-Datei entnehmen
        .Add(My.Resources.ListViewSourcePathColumnName)
        .Add(My.Resources.ListViewSearchMaskColumnName)
        .Add(My.Resources.ListViewDestinationPathColumnName)
    End With
End Sub

```

```
'Spalten ausrichten  
AlignColumns()  
End With  
. . .
```

Natürlich könnte man dieses Verfahren auch beispielsweise auf die angezeigten Texte im Backup-Protokoll ausweiten. Der Einfachheit halber habe ich die Verwendung von Text-Ressourcen auf diesen Bereich beschränkt.

HINWEIS Im Übrigen machen WinForms-Anwendungen beim Zuweisen von Texten oder Bilddaten ebenfalls implizit von Ressource-Dateien Gebrauch. Zwar verwendet der Windows Forms-Designer für den generierten Code nicht die Funktionalität aus dem My-Namespace, aber das Holen beispielsweise von Bitmap-Dateien aus einer Ressource, um dieser ein Symbol etwa einer Symbolleistenschaltfläche hinzuzufügen, funktioniert prinzipiell auf die gleiche Weise. Sie können sich selbst ein Bild davon machen: Wenn Sie im Projektmappen-Explorer die ausgeblendeten Dateien des Projektes mit dem Symbol *Alle Dateien anzeigen* (der Tooltip hilft beim Finden des Symbols) einblenden, können Sie einen Blick in den Code werfen, der zum Aufbau des Formulars führt. Dazu doppelklicken Sie anschließend auf die Datei *frmMain.Designer.vb*, die sich jetzt im Zweig unterhalb der Formulardatei *frmMain.vb* befindet.

```
'. . .  
'  
'tsbLoadCopyList – Auszug aus der FormsDesigner generierten Datei zum Zuweisen  
'eines Images an ein Symbol in DotNetCopy:  
Me.tsbLoadCopyList.DisplayStyle = System.Windows.Forms.ToolStripItemDisplayStyle.Image  
Me.tsbLoadCopyList.Image = CType(resources.GetObject("tsbLoadCopyList.Image"), _  
System.Drawing.Image)  
  
Me.tsbLoadCopyList.ImageTransparentColor = System.Drawing.Color.Magenta  
. . .
```

Die Zuweisung eines Symbols mithilfe des My-Namespace würde folgendermaßen ausschauen:

```
Me.tsbLoadCopyList.Image = My.Resources.LoadCopyList
```

Voraussetzung dafür wäre, dass eine entsprechende Image-Datei auf die zuvor beschriebene Weise der Ressource-Datei hinzugefügt wurde, etwa wie in Abbildung 29.2 zu sehen.

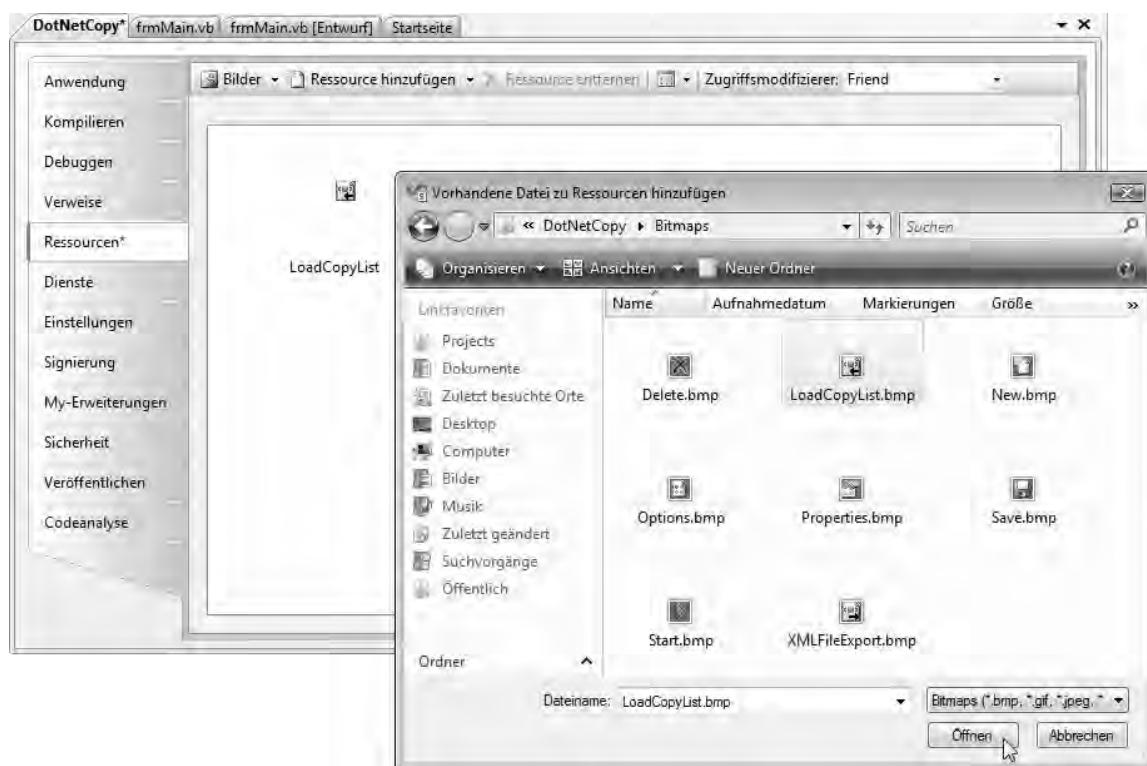


Abbildung 29.2 Klappen Sie die Schaltfläche Ressource hinzufügen auf (nicht darauf klicken!), und wählen Sie aus der Aufklappliste Datei hinzufügen, um beispielsweise eine Bitmap der Ressource-Datei hinzuzufügen, die Sie dann zur Laufzeit mit My.Resource und dem Element Laden zuweisen können

Internationalisieren von Anwendungen mithilfe von Ressource-Dateien und dem My-Namespace

Der Einsatz von Ressource-Dateien macht natürlich nur dann Sinn, wenn Sie planen, eine Anwendung tatsächlich auch in einer anderen Sprache zur Verfügung zu stellen. Für diesen Vorgang des Lokalisierens müssen Sie weitere Ressource-Dateien erstellen, die auf Basis eines bestimmten Schemas benannt und dem Projekt hinzugefügt werden.

Leider gibt es in Visual Studio 2008 für Windows Forms-Anwendungen in Visual Basic keine Designer-Unterstützung, die Ihnen wirklich dabei hilft, neue Ressource-Dateien für andere Kulturen aus schon vorhandenen zu erstellen. Sie müssen dazu selbst Hand anlegen und die IDE auch ein wenig austricksen. Die Unterstützung im Designer beschränkt sich lediglich auf Formulare selbst, bei denen Sie einfach mit Ihrer Localizable-Eigenschaft bestimmen können, ob das Formular lokalisiert werden soll oder nicht.

HINWEIS Wenn Sie die folgenden Schritte nachvollziehen möchten, empfehle ich Ihnen, die Projektdateien zum »Kaputt-experimentieren« zunächst an einen anderen Ort auf Ihrer Festplatte zu kopieren. Sollte bei den Versuchen dann etwas schief gehen, können Sie das Quellprojekt einfach wieder dort rüberkopieren und ohne Verlust von vorne beginnen.

- Öffnen Sie als Erstes die »Experimentier-Kopie« von DotNetCopy und lassen Sie den Projektmappen-Explorer mit **Strg Alt L** darstellen, falls dieser noch nicht angezeigt wird.
- Klicken Sie auf das entsprechende Symbol im Projektmappen-Explorer, um alle Dateien des Projektes anzeigen zu lassen. Öffnen Sie den Zweig, der sich neben dem jetzt sichtbaren Eintrag *My Project* befindet.
- Die Datei *Resources.resx*, in der sich die Ressourcen für die deutsche Kultur befinden, wird nun neben anderen angezeigt.
- Klicken Sie mit der rechten Maustaste auf diese Datei, um das Kontextmenü zu öffnen, und wählen Sie dort den Eintrag *Kopieren*.
- Im nächsten Schritt müssen Sie die IDE ein wenig austricksen: Sie müssen nun die Datei exakt an der Stelle einfügen, und damit eine Kopie der Ressource-Datei erstellen, an der die Ausgangsdatei ebenfalls platziert war. Klicken Sie allerdings wieder mit der rechten Maustaste, dann befindet sich im Kontextmenü kein Eintrag namens *Einfügen*. Aus diesem Grund klicken Sie die Datei *Resources.resx* mit der linken Maustaste an, um sie zu selektieren, und wählen Sie anschließend *Einfügen* aus dem Menü *Bearbeiten* der Visual Studio IDE. Eine Kopie der Datei steht anschließend ebenfalls im Zweig *My Project*.
- Nun benennen Sie die Ressource-Datei um. Das .NET Framework wird dann später, wenn es Ihr Programm ausführen muss, anhand des Namens der Ressource-Datei wissen, welche Datei es für welche zugrunde liegende Kultur verwenden soll. Auf deutschen Systemen wird, wenn Sie auf einer deutschen Windows-Plattform entwickelt haben, immer die Ressource-Datei verwendet, mit der Sie die Anwendung entwickelt haben. Für ein englisches System (bzw. US-amerikanisches) benennen Sie die gerade erstellte Kopie der Ressource-Datei in *Resources.en.resx* um, für französisch in *Resources.fr.resx*, für italienisch in *Resources.it.resx* usw.

TIPP Die gängigen Sprachkürzel entsprechen denen, wie sie bei der Programmierung von kulturabhängigen Format Providern verwendet werden. Sie können deswegen die entsprechende Liste aus Kapitel 26 verwenden.

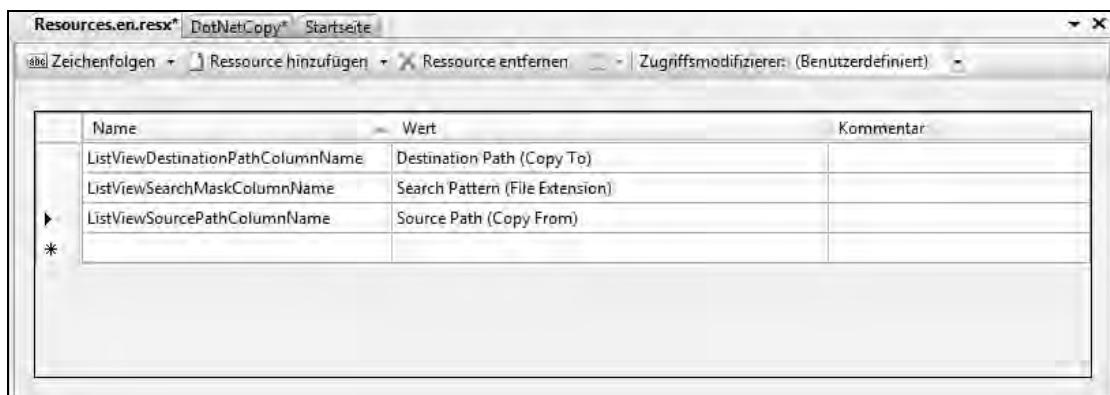


Abbildung 29.3 Wenn Sie die zusätzliche Ressource-Datei ins Projekt kopiert haben, können Sie sich an das Übersetzen der Ressourcen machen

- Wenn Sie die Datei umbenannt haben,² müssen Sie noch mit ein paar Handgriffen dafür sorgen, dass diese neue Ressource-Datei nicht dazu herhält, eine weitere Codedatei zu erzeugen, bei der dann, bedingt durch die schon vorhandene deutsche Version, Codeelemente für das Abrufen der Ressourcen doppelt definiert würden und so Fehler generierten. Dazu klicken Sie die umbenannte Datei an und löschen im Eigenschaftenfenster die Werte für *Benutzerdefiniertes Tool* und *Namespace des benutzerdefinierten Tools*.
- Doppelklicken Sie anschließend auf die unbenannte Datei, um die Einträge, die sie beinhaltet, zu lokalisieren – etwa wie in Abbildung 29.3 zu sehen.
- Speichern Sie die Änderungen anschließend und lassen Sie das komplette Projekt neu erstellen.

Sie werden anschließend sehen, dass im Verzeichnis der ausführbaren Dateien des Projektes (standardmäßig ist das der Pfad *Projekt\bin\Debug*) ein weiteres Unterverzeichnis hinzugekommen ist, das eine so genannte Satelliten-Assembly beinhaltet. Diese Satelliten-Assembly beinhaltet die lokalisierten Ressourcen für jede Ausgangsressource-Datei und wird vom .NET Framework automatisch verwendet, wenn der Name des Unterordners mit dem entsprechenden Kulturkürzel der Kultur des Systems übereinstimmt, auf dem die Anwendung ausgeführt wird (*en* für Englisch, *fr* für Französisch usw.). Das bedeutet: Wenn Sie die Satelliten-Assemblies auf diese Weise erzeugt haben, dann brauchen Sie sich selbst um nichts Weiteres mehr zu kümmern – sorgen Sie lediglich dafür, dass sie in entsprechenden Unterverzeichnissen vorhanden sind.

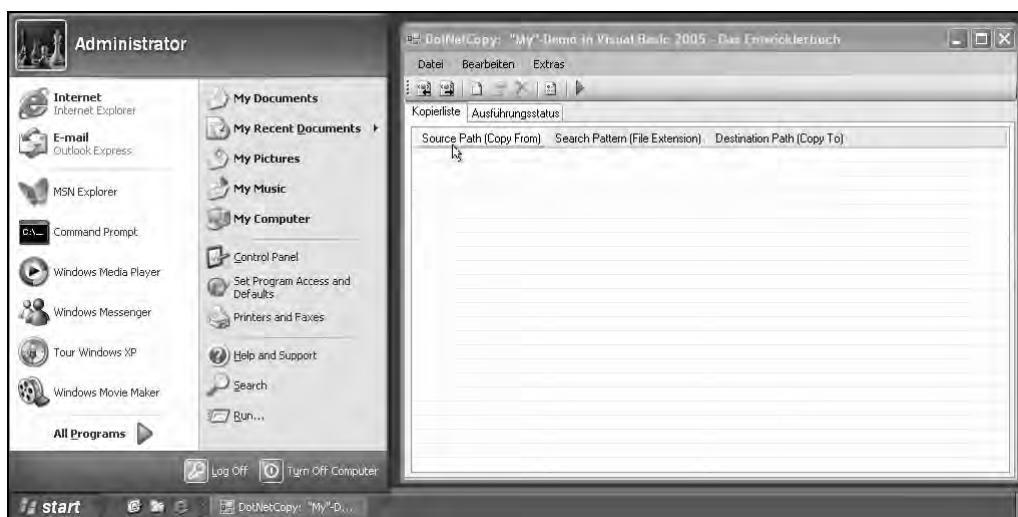


Abbildung 29.4 Auf diesem ohne Zweifel englischen Betriebssystem wird automatisch – so vorhanden – die englische Satelliten-Assembly verwendet; in der Ressource-Datei zuvor lokalisierte Texte erscheinen dann auch in der Landessprache

Auf einem englischen System würde die Anwendung dann automatisch so erscheinen, wie es Abbildung 29.4 auch wiedergibt (achten Sie hier auf die Bezeichnungen der ListView-Spalten, denn nur diese werden beispielhaft bei DotNetCopy aus der Ressource-Datei entnommen).

² Laut Aussagen des Entwicklerteams ist dieser Schritt seit der finalen Version von Visual Studio 2005 nicht mehr nötig; ein Auslassen dieses Schrittes hat aber bei einigen Experimenten Probleme verursacht, und so kann es nicht schaden, diesen zusätzlichen Schritt dennoch zu erledigen.

Vereinfachtes Durchführen von Dateioperationen mit My.Computer.FileSystem

FileInfo und DirectoryInfo unterstützten Sie bei der Ausführung von Dateioperationen schon auf eine sehr komfortable Art und Weise. Doch bestimmte Funktionen im My-Namespace können das noch besser, und sie machen es erst möglich, dass unser Backup-Tool DotNetCopy trotz der vergleichsweise großen Flexibilität dennoch relativ wenig Code benötigt.

Die Funktionen, von denen DotNetCopy gerade bei der Durchführung des eigentlichen Backup-Vorgangs sehr intensiv Gebrauch macht, finden sich in My.Computer.FileSystem. Besonders bemerkenswert sind hier Funktionen, mit denen ganze Verzeichnisstrukturen bzw. die Namen der sich dort befindlichen Dateien rekursiv eingelesen werden können. Sie kennen das vielleicht von dem Dir-Befehl der Befehlszeile von Windows: Nutzen Sie diesen mit der Option /S, werden nicht nur die Verzeichnisse bzw. Dateien des Verzeichnisses aufgelistet, das Sie angegeben haben bzw. in dem Sie sich derzeitig befinden, sondern es werden auch alle Unterverzeichnisse berücksichtigt.

Die eigentliche Kopierroutine von DotNetCopy macht von dieser GetFiles-Funktion Gebrauch, indem sie alle Unterverzeichnisse samt der darin enthaltenen Dateien ermittelt, zwischenspeichert und gemäß der so erstellten Liste im zweiten Schritt all diese Dateien kopiert. Der folgende Codeausschnitt demonstriert den Umgang:

```
''' <summary>
''' Die eigentliche Kopierroutine, die durch die My.Settings-Optionen gesteuert wird.
''' </summary>
''' <remarks></remarks>
Public Sub CopyFiles()

    'Aus Platzgründen gekürzt

    'In dieser Schleife werden zunächst alle Dateien ermittelt,
    'die es daraufhin zu untersuchen gilt, ob sie kopiert werden
    'müssen oder nicht.
    For locItemCount As Integer = 0 To lvwCopyEntries.Items.Count - 1
        Dim locCopyListEntry As CopyListEntry
        'CType ist notwendig, damit .NET weiß, welcher Typ
        'in der Tag-Eigenschaft gespeichert war.
        locCopyListEntry = CType(lvwCopyEntries.Items(locItemCount).Tag, CopyListEntry)
        'UI aktualisieren
        lblCurrentPass.Text = "Pass 1: Kopiervorbereitung durch Zusammenstellung der Pfade..."
        lblSourceFileInfo.Text = "Unterverzeichnisse suchen in:"
        lblDestFileInfo.Text = ""
        lblProgressCaption.Text = "Fortschritt Vorbereitung:"
        lblCurrentSourcePath.Text = locCopyListEntry.SourceFolder.ToString
        LogInProtocolWindow(locCopyListEntry.SourceFolder.ToString)
        pbPrepareAndCopy.Value = locItemCount + 1

        'Windows die Möglichkeit geben, die Steuerelemente zu aktualisieren
        My.Application.DoEvents()
    Next
End Sub
```

```
'GetFiles aus My.Computer.FileSystem liefert die Namen aller
'Dateien im Stamm- und in dessen Unterverzeichnissen.
Dim locFiles As ReadOnlyCollection(Of String)
Try
    locNotCaught = False
    locFiles = My.Computer.FileSystem.GetFiles(
        locCopyListEntry.SourceFolder.ToString(),
        FileIO.SearchOption.SearchAllSubDirectories,
        locCopyListEntry.SearchMask)
Catch ex As Exception
   .LogError(ex)
    'Wenn schon beim Zusammenstellen der Dateien Fehler aufgetreten sind,
    'sollte das Verzeichnis ausgelassen werden. In diesem Fall sollte dieser
    'schwerwiegende Fakt im Anwendungsprotokoll protokolliert werden!
    My.Application.Log.WriteLineException(ex, TraceEventType.Error,
        "DotNetCopy konnte das Verzeichnis " &
        locCopyListEntry.SourceFolder.ToString & " nicht durchsuchen., " & _
        "Das Verzeichnis wurde daher nicht berücksichtigt!")
    ' Es müsste wie "Catch" einen "NotCaught"-Zweig geben...
    locNotCaught = True
End Try
.
.
.
```

Eine weitere Funktionalität, die ebenfalls sehr sinnvoll und praktisch ist, stellt `My.Computer.FileSystem` in Form der `CopyFile`-Methode zur Verfügung. Anders als die Methode `CopyTo` der `FileInfo`-Klasse greift diese exakt auf die gleiche Betriebssystemfunktion von Windows zurück, die auch beispielsweise der Windows-Explorer verwendet. Und das bedeutet – auch wenn `DotNetCopy` in diesem Beispiel davon keinen Gebrauch macht – dass Sie bei der Verwendung von `CopyFile` aus dem `My`-Namespace automatisch den bekannten Fortschrittsdialog anzeigen lassen können.

```
''' <summary>
''' Interne Kopierroutine, die den eigentlichen 'Job' des Kopierens übernimmt.
''' </summary>
''' <param name="SourceFile">Quelldatei, die kopiert werden soll.</param>
''' <param name="DestFile">Zielpfad, in den die Datei hineinkopiert werden soll.</param>
''' <remarks></remarks>
Private Sub CopyFileInternal(ByVal SourceFile As FileInfo, ByVal DestFile As FileInfo)
    'Falls die Zieldatei noch gar nicht existiert,
    'dann in jedem Fall kopieren
    If Not DestFile.Exists Then
        Try
            'Datei kopieren, ohne Windows-Benutzeroberfläche anzuzeigen.
            My.Computer.FileSystem.CopyFile(SourceFile.ToString, DestFile.ToString)
            LogInProtocolWindow("Kopiert, OK: " & SourceFile.ToString)
        Catch ex As Exception
           .LogError(ex)
        End Try
        Exit Sub
    End If
```

```

'Datei nur kopieren, wenn Sie jünger als die zu überschreibende ist
If My.Settings.Option_OnlyOverwriteIfOlder Then
    If SourceFile.LastWriteTime > DestFile.LastWriteTime Then
        'Wenn das Historien-Backup aktiviert ist...
        If My.Settings.Option_EnableBackupHistory Then
            '...dieses durchführen.
            ManageFileBackupHistory(DestFile)
        End If
    Try
        'Datei kopieren.
        My.Computer.FileSystem.CopyFile(SourceFile.ToString, DestFile.ToString)
        LogInProtocolWindow("Kopiert, OK: " & SourceFile.ToString)
    Catch ex As Exception
        LogError(ex)
    End Try
End If
End If
End Sub

```

HINWEIS Leider werden bei der Verwendung von CopyFile in Windows-Anwendungen, bei denen – wie es standardmäßig bei Windows Forms-Anwendungen geschieht – Formulare an eine Meldungswarteschlange gebunden werden, auch Fehlermeldungen als modale Dialoge dargestellt. Gerade bei Fehlern, die das Programm behandeln soll, kann das lästig und unerwünscht sein. Wichtig ist es aber auch zu wissen, dass bei Anwendungen, die keine Meldungswarteschlange initiieren, Fehlermeldungen nicht vom Betriebssystem in Form eines sichtbaren modalen Dialogs abgefangen werden, oder mit anderen Worten: Wenn Sie dafür sorgen, dass Ihr Programm keine sichtbare Benutzeroberfläche hat, dann erscheint auch keine sichtbare Fehlermeldung, selbst wenn bei CopyFile ein Fehler auftritt.

Im Falle unseres Beispiels DotNetCopy bedeutet das: Wir können im Silent-Modus keine sichtbaren Fehlermeldungen gebrauchen. Die gibt es aber auch bei CopyFile in diesem Modus sowieso nicht, denn es gibt keine sichtbare Benutzeroberfläche. Zwar verwenden wir die gleiche Formulkarklasse, die wir auch verwenden, wenn DotNetCopy nicht im Silent-Modus läuft, aber wir zeigen das Formular nicht an.

Der folgende Code, der sich im StartUp-Ereignis des MyApplication-Objektes befindet, und der ausgelöst wird, sobald die Anwendung startet, demonstriert das:

```

'Hinweis: locFrmMain wurde weiter oben im Code schon instanziiert, aber nicht dargestellt!
'Silentmode bleibt nur "an", wenn AutoStart aktiv ist.
locFrmMain.SilentMode = locFrmMain.SilentMode And locFrmMain.AutoStartMode

'Und wenn Silentmode, erfolgt keine Bindung des Formulars an den Anwendungskontext!
If locFrmMain.SilentMode Then
    'Alles wird in der nicht sichtbaren Instanz des Hauptforms durchgeführt,
    locFrmMain.HandleAutoStart()
    'und bevor das "eigentliche" Programm durch das Hauptformular gestartet wird,
    'ist der ganze Zauber auch schon wieder vorbei.
    e.Cancel = True
Else
    'Im Nicht-Silent-Modus wird das Formular an die Anwendung gebunden,
    'und los geht's!
    My.Application.MainForm = locFrmMain
End If
End Sub
End Class

```

Verwenden von Anwendungseinstellungen mit My.Settings

DotNetCopy verwendet Anwendungseinstellungen (*ApplicationSettings*), um die Einstellungen auch nach dem Beenden des Programms zu erhalten, die der Anwender vornehmen kann, wenn er aus dem Menü *Extras* den Menüpunkt *Optionen* auswählt. Da schon Kapitel 5 sich intensiv mit Anwendungseinstellungen auseinandergesetzt hat, erfolgt an dieser Stelle lediglich eine kompakte Zusammenfassung des dazu Bekannten:

Anwendungseinstellungen pflegen Sie – ähnlich wie die Zeichenkettentabellen der Ressourcen – über eine Tabelle, die Sie über die Projekteigenschaften erreichen.

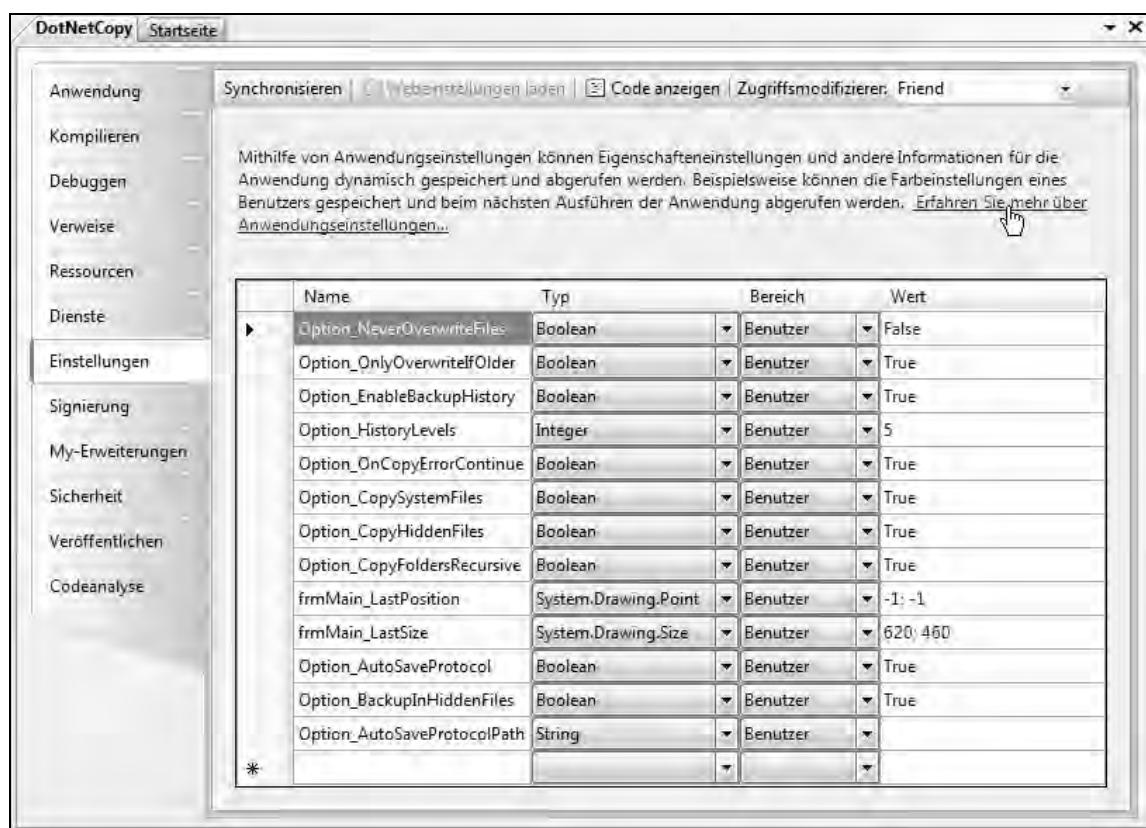


Abbildung 29.5 Anwendungseinstellungen, die Sie über My.Settings abrufen können, verwalten Sie wie Ressourcen in den Projekteigenschaften unter der Registerkarte Einstellungen

Für jeden Namen, den Sie in der Tabelle anlegen, wird dabei eine über My.Settings zu erreichende Objektvariable von dem Typ angelegt, den Sie in der Tabelle unter *Typ* bestimmt haben. Sie können für die Variablen, die Sie an dieser Stelle anlegen, aus zwei Bereichen auswählen: *Benutzer* wählen Sie, wenn Sie den Variableninhalt zur Laufzeit verändern wollen, und der neue Inhalt nach Programmende auch automatisch gesichert

werden soll; *Anwendung* wählen Sie, wenn es sich um eine Konstante handeln soll, die nur Sie zur Entwurfszeit in den Projekteigenschaften (in der in Abbildung 29.5 gezeigten Tabelle) einstellen können, die man aber zur Laufzeit nicht verändern darf.

Das Abspeichern oder Auslesen von Anwendungseinstellungen ist schließlich eine Kleinigkeit. Der folgende Code zeigt, wie von Anwendungseinstellungen intensiv Gebrauch gemacht wird, wenn der Anwender im Optionsdialog Änderungen vornimmt. Die folgende Routine, die aufgerufen wird, wenn der Optionsdialog geladen wird, sorgt zunächst dafür, dass die Steuerelemente die Anwendungseinstellungen widerspiegeln:

```
Private Sub frmOptions_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
    'Anwendungseinstellungen im Dialog widerspiegeln lassen:
    'Auch dazu wird My.Settings verwendet.
    chkCopyFoldersRecursive.Checked = My.Settings.Option_CopyFoldersRecursive
    chkCopyHiddenFiles.Checked = My.Settings.Option_CopyHiddenFiles
    chkCopySystemFiles.Checked = My.Settings.Option_CopySystemFiles
    chkEnableBackupHistory.Checked = My.Settings.Option_EnableBackupHistory
    nudHistoryLevels.Value = My.Settings.Option_HistoryLevels
    chkNeverOverwriteFiles.Checked = My.Settings.Option_NeverOverwriteFiles
    chkOnCopyErrorContinue.Checked = My.Settings.Option_OnCopyErrorContinue
    chkOnlyOverwriteIfOlder.Checked = My.Settings.Option_OnlyOverwriteIfOlder
    chkBackupInHiddenFiles.Checked = My.Settings.Option_BackupInHiddenFiles
    chkAutoSaveProtocol.Checked = My.Settings.Option_AutoSaveProtocol
    lblProtocolPath.Text = My.Settings.Option_AutoSaveProtocolPath
End Sub
```

Beim Verlassen des Dialogs mit *OK* erfolgt der umgekehrte Weg: Die Werte oder Zustände der Steuerelemente werden ausgelesen und in den Anwendungseinstellungen gespeichert.

```
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
    'Die Einstellungen des Options-DIALOGS in den Anwendungseinstellungen speichern.
    'Auch dazu wird My.Settings verwendet.
    My.Settings.Option_CopyFoldersRecursive = chkCopyFoldersRecursive.Checked
    My.Settings.Option_CopyHiddenFiles = chkCopyHiddenFiles.Checked
    My.Settings.Option_CopySystemFiles = chkCopySystemFiles.Checked
    My.Settings.Option_EnableBackupHistory = chkEnableBackupHistory.Checked
    My.Settings.Option_HistoryLevels = CInt(nudHistoryLevels.Value)
    My.Settings.Option_NeverOverwriteFiles = chkNeverOverwriteFiles.Checked
    My.Settings.Option_OnCopyErrorContinue = chkOnCopyErrorContinue.Checked
    My.Settings.Option_OnlyOverwriteIfOlder = chkOnlyOverwriteIfOlder.Checked
    My.Settings.Option_BackupInHiddenFiles = chkBackupInHiddenFiles.Checked
    My.Settings.Option_AutoSaveProtocol = chkAutoSaveProtocol.Checked
    My.Settings.Option_AutoSaveProtocolPath = lblProtocolPath.Text

    'Das Setzen des Dialogergebnisses bei einem modal aufgerufenen Dialog
    'bewirkt gleichzeitig, dass dieser geschlossen wird!
    Me.DialogResult = Windows.Forms.DialogResult.OK
End Sub
```

Das Ablegen von Informationen in den Anwendungseinstellungen wird auch vom Designer der Windows-Anwendungen und der dort benutzten Steuerelemente verwendet. Wenn Sie beispielsweise die Beschriftung eines Formulars (also die Text-Eigenschaft einer Windows Forms Klasse) außerhalb des Quellcodes der Anwendung festlegen und ändern wollen, ist dies über die Oberfläche einfach zu realisieren. Öffnen Sie ein beliebiges Formular und wechseln Sie dort ins Eigenschaftsfenster (**F4**). Wenn Sie die Eigenschaften nach Kategorien anzeigen lassen, gehen Sie zur Kategorie *Daten*, dort ist der erste Eintrag, in alphabetischer Sortierung ist es von vornherein der erste Eintrag: (*Application Settings*). Die wichtigsten Eigenschaften der Klasse lassen sich direkt (wie Text bei Windows Forms) eintragen, wählen Sie (*Neu*) in der Klappliste ganz unten. Unter (*PropertyBinding*) lassen sich aber beinahe alle Eigenschaften auslagern. Im folgenden Dialog geben Sie einfach einen Namen an, unter dem Sie die Einstellung abspeichern wollen. Unter *Scope* wählen Sie, ob die Einstellungen immer für die Anwendung gelten sollen, oder ob diese pro Benutzer konfiguriert werden können. Dazu mehr im folgenden Kapitel.

Speichern von Anwendungseinstellungen mit dem Bereich Benutzer

Den Code zum Abspeichern der Anwendungseinstellungen werden Sie übrigens im Optionsdialog vergebens suchen – es gibt ihn nicht, da er im Grunde genommen nicht notwendig ist. Wenn eine Windows Forms-Anwendung beendet wird, für die sowohl das Anwendungsframework als auch die Option *Eigene Einstellungen beim Herunterfahren speichern* in den Projekteigenschaften unter *Anwendung* aktiviert ist (siehe auch Kapitel 30), dann regelt das Framework das Speichern der Anwendungseinstellungen für den Bereich *Benutzer* automatisch.

In einigen Fällen kann es aber sinnvoll sein, die Anwendungseinstellungen selbst oder schon vor dem Beenden des Programms zu speichern. In DotNetCopy ist das beispielsweise der Fall, wenn das Programm zum allerersten Mal gestartet wird. In diesem Fall werden die Einstellungen für die Position und die Größe des Hauptformulars festgelegt und direkt gespeichert – und das erfolgt mit der Methode `My.Settings.Save`, wie das folgende Beispiel zeigt:

```
Private Sub frmMain_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
    '
    . ' Aus Platzgründen ausgelassen
    .
    'Fenstergröße und -position wiederherstellen, wenn
    'nicht im Silent-Modus
    If Not mySilentMode Then
        'Beim ersten Aufruf ist die Größe -1, dann bleibt die Default-
        'Position, die Settings werden aber sofort übernommen...
        If My.Settings.frmMain_LastPosition.X = -1 Then
            My.Settings.frmMain_LastPosition = Me.Location
            Me.Size = My.Settings.frmMain_LastSize
            My.Settings.Save()

        'anderenfalls werden die Settings-Einstellungen in die Formular-
        'Position übernommen
        Else
            Me.Location = My.Settings.frmMain_LastPosition
        End If
        Me.Size = My.Settings.frmMain_LastSize
    End If
```

Kapitel 30

Das Anwendungsframework

In diesem Kapitel:

Die Optionen des Anwendungsframeworks	851
Eine Codedatei implementieren, die Anwendungsereignisse behandelt	852

Das, was Microsoft als »Anwendungsframework« bezeichnet, ist eine Reihe von Funktionalitäten, die unter anderem auch dafür verantwortlich sind, dass Anwendungseinstellungen nach dem Beenden eines Programms abgespeichert werden oder auch dass Anwendungsereignisse zur Verfügung gestellt werden (siehe Abschnitt »Eine Codedatei implementieren, die Anwendungsereignisse behandelt« ab Seite **Fehler! Textmarke nicht definiert.**).

Um das zu erreichen, bedient sich Visual Basic übrigens zwei Mitteln: Zum Einen ergänzt der Visual Basic-Compiler Ihren Quellcode um weiteren Code, den Sie selbst aber nie zu Gesicht bekommen. Dieser Quellcode ist quasi fest im Compiler hinterlegt, und werden bestimmte Funktionalitäten benötigt, die Sie durch die Optionen des Anwendungsframeworks in den Projekteigenschaften für Ihr Projekt aktivieren, ergänzt der Compiler dafür während des Kompilierens den entsprechenden Quellcode, der dann wiederum dafür sorgt, dass Ihr Programm die entsprechenden Funktionalitäten zur Verfügung stellt. Zum Anderen benötigt Visual Basic dazu bestimmte Funktionsaufrufe, die durch die *VisualBasic.dll* des Frameworks zur Verfügung gestellt werden.

Für Windows-Anwendungen ist das Anwendungsframework standardmäßig aktiviert. Sie können die Aktivierung des Anwendungsframeworks in den Projekteigenschaften unter der Registerkarte *Anwendung* steuern (siehe Abbildung 30.1).

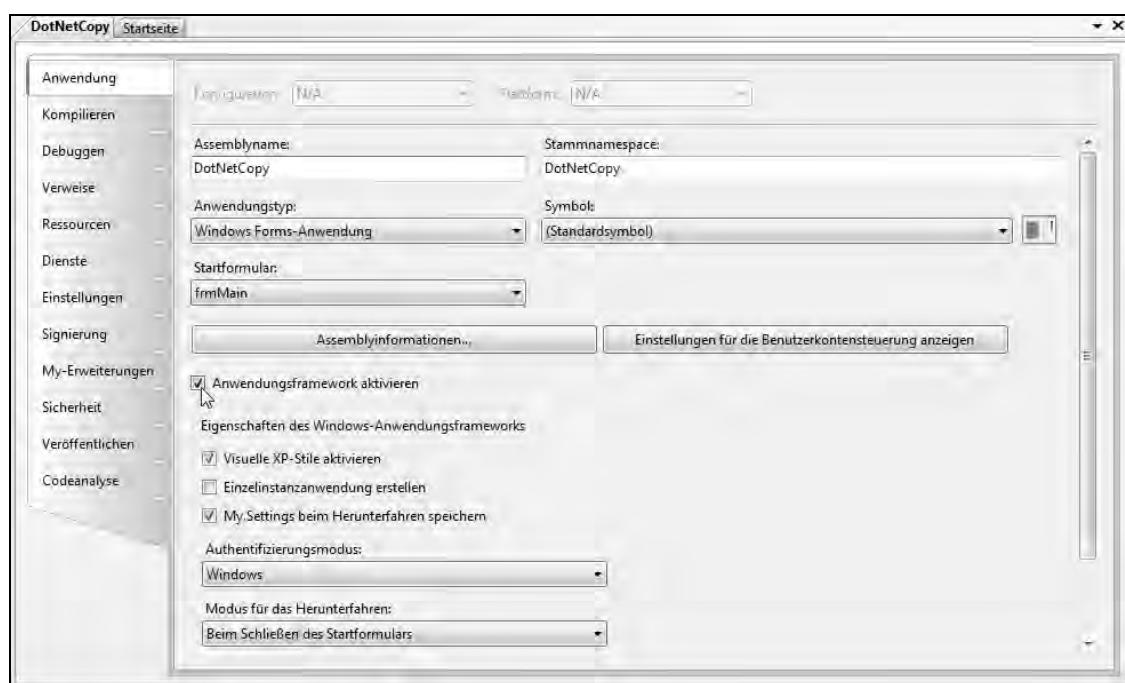


Abbildung 30.1 Erst durch das aktivierte Anwendungsframework stehen viele der hier beschriebenen Funktionalitäten zur Verfügung. Bei Windows Forms-Anwendungen ist das Anwendungsframework standardmäßig aktiviert.

Wenn das Kontrollkästchen *Anwendungsframework aktivieren* aktiviert ist, verwendet die Anwendung dann eine quasi »versteckte« Sub Main (die durch den versteckten Quellcode des Compilers in Ihr Projekt »gelangt«), mit der Ihre Windows-Anwendung startet, und die es erst ermöglicht, dass alle weiteren Optionen des Anwendungsframeworks funktionieren können. In diesem Fall müssen Sie ein Startformular auswählen, das

von dieser versteckten Sub Main aufgerufen wird, wenn alle nötigen Vorbereitungen für das Zur-Verfügung-Stellen aller anderen Funktionen des Anwendungsframeworks abgeschlossen sind.

Wenn dieses Kontrollkästchen deaktiviert ist, verwendet die Anwendung die benutzerdefinierte Sub Main, die Sie unter Startformular angegeben haben, und die Sie in einem Modul Ihres Projektes platzieren müssen. In diesem Fall können Sie entweder ein Startobjekt (eine benutzerdefinierte Sub Main in einer Methode oder Klasse) oder ein Formular festlegen. Die Optionen im Bereich *Eigenschaften des Windows-Anwendungsframeworks* sind in diesem Fall nicht verfügbar.

Die Optionen des Anwendungsframeworks

Die folgenden Einstellungen dienen der Konfiguration des Abschnitts *Eigenschaften* des Windows-Anwendungsframeworks. Diese Optionen sind nur verfügbar, wenn das Kontrollkästchen *Anwendungsframework aktivieren* aktiviert ist.

Windows XP Look and Feel für eigene Windows-Anwendungen – Visuelle XP-Stile aktivieren

Aktivieren oder deaktivieren Sie die visuellen Windows XP-Stile, die auch Windows XP-Designs genannt werden. Die visuellen Windows XP-Stile lassen z. B. Steuerelemente mit gerundeten Ecken und dynamischen Farben zu. Die Option ist standardmäßig aktiviert; eine entsprechende visuelle Anpassung wird auch für Vista vorgenommen.

Verhindern, dass Ihre Anwendung mehrfach gestartet wird – Einzelinstanzanwendung erstellen

Aktivieren Sie dieses Kontrollkästchen, um zu verhindern, dass Benutzer mehrere Instanzen der Anwendung ausführen. Dieses Kontrollkästchen ist standardmäßig deaktiviert, wodurch mehrere Instanzen der Anwendung ausgeführt werden können.

MySettings-Einstellungen automatisch sichern – Eigene Einstellungen beim Herunterfahren speichern

Durch das Aktivieren dieses Kontrollkästchens wird festgelegt, dass die My.Settings-Einstellungen der Anwendung beim Herunterfahren gespeichert werden. Die Option ist standardmäßig aktiviert. Wenn diese Option deaktiviert wird, können Sie diese Festlegung durch Aufrufen von My.Settings.Save manuell durchführen.

Bestimmen, welcher Benutzer-Authentifizierungsmodus verwendet wird

Wählen Sie Windows (Standard) aus, um die Verwendung der Windows-Authentifizierung für die Identifikation des gerade angemeldeten Benutzers festzulegen. Diese Informationen können zur Laufzeit übrigens mit dem My.User-Objekt abgerufen werden. Wählen Sie *Anwendungsdefiniert* aus, wenn Sie eigenen Code anstatt der Windows-Standardauthentifizierungsmethoden für die Authentifizierung von Benutzern verwenden möchten.

Festlegen, wann eine Anwendung »zu Ende« ist – Modus für das Herunterfahren

Wählen Sie *Beim Schließen des Startformulars (Standard)* aus, um festzulegen, dass die Anwendung beendet wird, wenn das als Startformular festgelegte Formular geschlossen wird. Dies gilt auch, wenn andere Formulare geöffnet sind. Wählen Sie *Beim Schließen des letzten Formulars* aus, um festzulegen, dass die Anwendung beendet wird, wenn das letzte Formular geschlossen oder wenn `My.Application.Exit` oder die End-Anweisung explizit aufgerufen wird.

Einen Splash-Dialog beim Starten von komplexen Anwendungen anzeigen – Begrüßungsdialog

Wählen Sie das Formular aus, das als Begrüßungsbildschirm angezeigt werden soll. Zuvor müssen Sie einen Begrüßungsbildschirm mithilfe eines Formulars oder einer Vorlage (in dem Sie z.B. unter dem Projekt *Hinzufügen, Neues Element..., Begrüßungsbildschirm*) erstellt haben.

Eine Codedatei implementieren, die Anwendungsereignisse behandelt

In den vorangegangenen Abschnitten haben Sie schon hier und da am Rande mitbekommen können, dass es offensichtlich möglich sein muss, bestimmte Anwendungereignisse »mithören« zu können. *DotNetCopy* klinkt sich so beispielsweise in den Programmstart ein und steuert das weitere Schicksal des Programms aufgrund der übergebenen Optionen sogar.

Wenn bei bestimmten Anwendungereignissen Ihr eigener Code ausgeführt werden soll, müssen Sie dazu eine kleine Vorbereitung treffen:

- Rufen Sie dazu die Eigenschaften Ihres Projektes auf, und wählen Sie die Registerkarte *Anwendung*.
- Klicken Sie auf die Schaltfläche *Anwendungereignisse anzeigen* (siehe Abbildung 30.1).
- Die Visual Studio-IDE fügt Ihrem Projekt nun eine Codedatei namens *ApplicationEvents.vb* hinzu und öffnet diese im Editor.
- Um den Funktionsrumpf für die Behandlungsroutine in eine der fünf existierenden Anwendungereignisse einzufügen, wählen Sie im Codeeditor aus der linken Aufklappliste am oberen Rand des Editors zunächst den Eintrag `myApplicationEvents`.
- Wählen Sie anschließend in der rechten Liste das Ereignis, das Sie behandeln wollen, und für das der Funktionsrumpf in den Editor eingefügt werden soll. Orientieren Sie sich dabei auch an Abbildung 30.2.

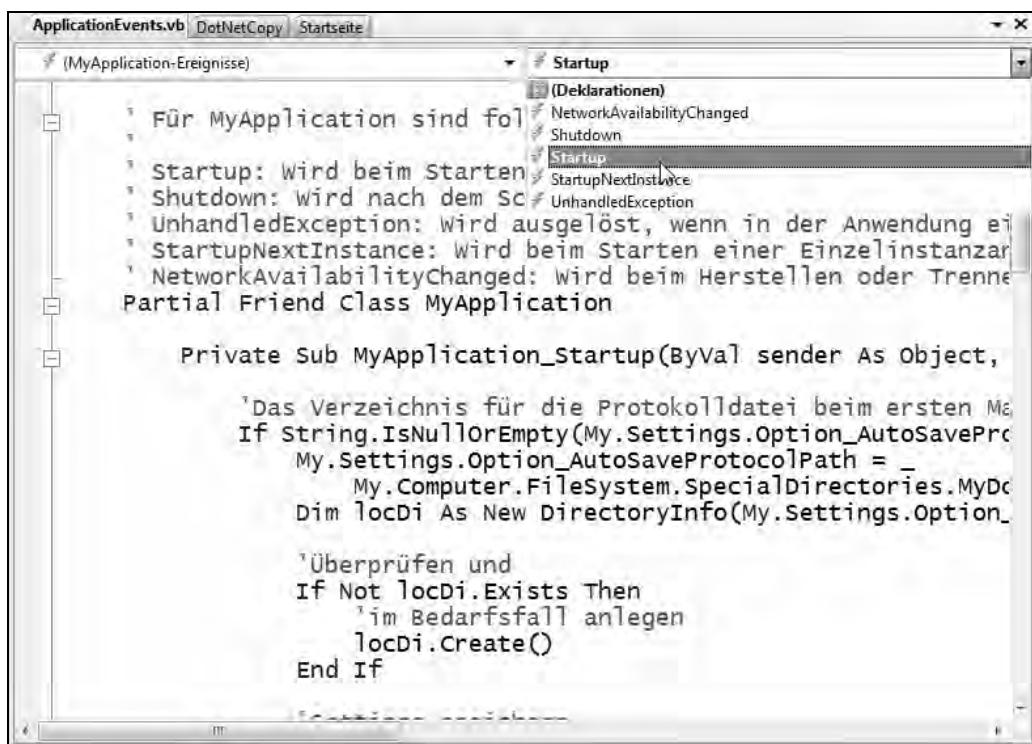


Abbildung 30.2 Wählen Sie zum Einfügen des Rumpfs der Ereignisbehandlungsroutine eines der fünf existierenden Ereignisse aus der Aufklappliste aus

- Formulieren Sie den Behandlungscode aus.

HINWEIS Beachten Sie auch, dass einige Ereignisse besondere Ableitungen der EventArgs-Klasse als Instanz übergeben, mit denen Sie das weitere Programmverhalten beeinflussen können. Die folgende Tabelle verrät mehr zu den vorhandenen Ereignissen, und wie Sie in ihren Rahmen den weiteren Programmverlauf beeinflussen können:

Ereignisname	Beschreibung	Steuerungsmöglichkeiten
Startup	Wird beim Starten der Anwendung noch vor dem Erstellen des Startformulars und dessen Binden an den Anwendungskontext ausgelöst.	Ja: Der Start der Anwendung kann durch Setzen von Cancel der StartupEventArgs-Instanz verhindert werden. In CommandLineArgs dieser Instanz werden ferner die Befehlszeilen-Argumente an die Anwendung als String-Array übergeben.
Shutdown	Wird nach dem Schließen aller Anwendungsfenster ausgelöst. Dieses Ereignis wird nicht ausgelöst, wenn die Anwendung nicht normal beendet wird, also beispielsweise durch einen Fehler, der eine Ausnahme ausgelöst hat.	Nein.

Ereignisname	Beschreibung	Steuerungsmöglichkeiten
StartUpNextInstance	<p>Wird beim Starten einer Einzelinstanzanwendung ausgelöst, wenn diese bereits aktiv ist.</p> <p>HINWEIS: Sie bestimmen, dass eine Anwendung nur in einer Instanz (also nicht mehrmals gleichzeitig) gestartet werden darf, wenn Sie in den Projekteigenschaften Ihres Windows Forms-Projektes auf der Registerkarte <i>Anwendungen</i> die Option <i>Einzelinstanzanwendung erstellen</i> aktivieren.</p>	<p>Ja: Mit dem Setzen der <i>BringToFront</i>-Eigenschaft der <i>StartupNextInstanceEventArgs</i>-Instanz können Sie festlegen, dass nach Verlassen der Ereignisbehandlungsroutine die schon laufende Instanz Ihrer Anwendung wieder zur aktiven Anwendung wird.</p> <p>In <i>CommandLineArgs</i> dieser Instanz werden ferner die Befehlszeilen-Argumente an die Anwendung als String-Array übergeben (und zwar für die, die zum erneuten Startversuch und damit auch zum Auslösen des Ereignisses führen).</p>
NetworkAvailabilityChanged	Wird beim Herstellen oder Trennen der Netzwerkverbindung ausgelöst.	Nein. Sie können jedoch mit der Nur-Lesen-Eigenschaft <i>IsNetworkAvailable</i> abfragen, ob das Netzwerk zur Verfügung steht oder nicht.
UnhandledException	Wird ausgelöst, wenn in der Anwendung auf Grund eines nicht behandelten Fehlers eine unbehandelte Ausnahme auftritt.	<p>Ja: Das Beenden der Anwendung aufgrund des Fehlers kann durch Setzen von <i>Cancel</i> der <i>UnhandledExceptionEventArgs</i>-Instanz verhindert werden.</p> <p>Darüber hinaus steht Ihnen durch die <i>Exception</i>-Eigenschaft dieser Instanz ein <i>Exception</i>-Objekt zur Verfügung, das Auskunft über die aufgetretene unbehandelte Ausnahme gibt.</p>

Tabelle 30.1 Diese Ereignisse des Anwendungsframeworks stehen Ihnen zur Verfügung

Der folgende Beispielcode demonstriert den Umgang mit den Anwendungsereignissen:

```

Private Sub MyApplication_NetworkAvailabilityChanged(ByVal sender As Object, _
    ByVal e As Microsoft.VisualBasic.Devices.NetworkAvailableEventArgs) _
    Handles Me.NetworkAvailabilityChanged
    MessageBox.Show("Die Netzwerkverfügbarkeit hat sich geändert!" &
        "Das Netzwerk ist " & IIf(e.IsNetworkAvailable, "verfügbar", "nicht
verfügbar").ToString)
End Sub

Private Sub MyApplication_Shutdown(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles Me.Shutdown
    Debug.Print("Die Anwendung wird jetzt beendet!")
End Sub

Private Sub MyApplication_StartupNextInstance(ByVal sender As Object, _
    ByVal e As Microsoft.VisualBasic.ApplicationServices.StartupNextInstanceEventArgs) _
    Handles Me.StartupNextInstance
    MessageBox.Show("Sie können keine 2. Instanz dieser Anwendung starten!")
    e.BringToFront = True
End Sub

Private Sub MyApplication_UnhandledException(ByVal sender As Object, _
    ByVal e As Microsoft.VisualBasic.ApplicationServices.UnhandledExceptionEventArgs) _
    Handles Me.UnhandledException
    'Eine bislang nicht behandelte Ausnahme ist aufgetreten!

```

```
Dim locDr As DialogResult =  
    MessageBox.Show("Eine unbehandelte Ausnahme ist aufgetreten!" & _  
        vbNewLine & "Soll die Anwendung beendet werden?", _  
        "Unbehandelte Ausnahme", MessageBoxButtons.YesNo, _  
        MessageBoxIcon.Error, MessageBoxDefaultButton.Button1)  
If locDr = DialogResult.No Then  
    e.ExitApplication = False  
End If  
End Sub
```

Ein dokumentiertes Beispiel für das Ereignis StartUp, das beim Starten einer Windows Forms-Anwendung ausgelöst wird, finden Sie auch im Beispielprogramm DotNetCopy – der Vollständigkeit halber finden Sie im Folgenden den kompletten Code dieser Ereignisbehandlungsroutine:

BEGLEITDATEIEN Viele der hier gezeigten Features können Sie sich direkt am Beispiel von DotNetCopy (vorgestellt in Kapitel 28) anschauen. Sie finden dieses Projekt unter dem Namen *DotNetCopy.sln* im Verzeichnis

...\\VB 2008 Entwicklerbuch\\E - Vereinfachungen\\Kapitel 28\\DotNetCopy

So Sie Kapitel 28 noch nicht gelesen haben, sollten Sie es zunächst tun, um das Beispiel besser nachvollziehen zu können.

```
Private Sub MyApplication_Startup(ByVal sender As Object,  
    ByVal e As Microsoft.VisualBasic.ApplicationServices.StartupEventArgs) Handles Me.Startup  
  
'Das Verzeichnis für die Protokolldatei beim ersten Mal setzen...  
If String.IsNullOrEmpty(My.Settings.Option_AutoSaveProtocolPath) Then  
    My.Settings.Option_AutoSaveProtocolPath =  
        My.Computer.FileSystem.SpecialDirectories.MyDocuments & "\\DotNetCopy Protokolle"  
    Dim locDi As New DirectoryInfo(My.Settings.Option_AutoSaveProtocolPath)  
  
    'Überprüfen und  
    If Not locDi.Exists Then  
        'im Bedarfsfall anlegen  
        locDi.Create()  
    End If  
  
    'Settings speichern  
    My.Settings.Save()  
End If  
  
Dim locFrmMain As New frmMain  
  
'Kommandozeile auslesen  
If My.Application.CommandLineArgs.Count > 0 Then  
    For Each locString As String In My.Application.CommandLineArgs  
  
        'Alle unnötigen Leerzeichen entfernen und  
        'Groß-/Kleinschreibung 'Unsensibilisieren'  
        'HINWEIS: Das funktioniert nur in der Windows-Welt;  
        'kommt die Kopierlistendatei von einem Unix-Server, bitte darauf achten,  
        'dass der Dateiname dafür auch komplett in Großbuchstaben gesetzt ist,  
        'da Unix- (und Linux-) Derivate Groß-/Kleinschreibung berücksichtigen!!!  
        locString = locString.ToUpper.Trim
```

```
If locString = "/SILENT" Then
    locFrmMain.SilentMode = True
End If

If locString.StartsWith("/AUTOSTART") Then
    locFrmMain.AutoStartCopyList = locString.Replace("/AUTOSTART:", "")
    locFrmMain.AutoStartMode = True
End If
Next
End If

'Silentmode bleibt nur "an", wenn AutoStart aktiv ist.
locFrmMain.SilentMode = locFrmMain.SilentMode And locFrmMain.AutoStartMode

'Und wenn Silentmode, erfolgt keine Bindung des Formulars an den Anwendungskontext!
If locFrmMain.SilentMode Then
    'Alles wird in der nicht sichtbaren Instanz des Hauptforms durchgeführt,
    locFrmMain.HandleAutoStart()
    'und bevor das "eigentliche" Programm durch das Hauptformular gestartet wird,
    'ist der ganze Zauber auch schon wieder vorbei.
    e.Cancel = True
Else
    'Im Nicht-Silent-Modus wird das Formular an die Anwendung gebunden,
    'und los geht's!
    My.Application.MainForm = locFrmMain
End If
End Sub
End Class

End Namespace
```

Teil F

Language Integrated Query – LINQ

In diesem Teil:

Einführung in Language integrated Query (LINQ)	859
LINQ to Objects	879
LINQ to XML	901
LINQ to SQL	911
LINQ to Entities –Programmieren mit dem Entity Framework	979

Kapitel 31

Einführung in Language Integrated Query (LINQ)

In diesem Kapitel:

Wie »geht« LINQ prinzipiell	862
Die Where-Methode	867
Die Select-Methode	868
Die OrderBy-Methode	872
Die GroupBy-Methode	875
Kombinieren von LINQ-Erweiterungsmethoden	876
Vereinfachte Anwendung von LINQ – Erweiterungsmethoden mit der LINQ-Abfragesyntax	878

Falls Sie sich in Ihrer Entwicklerkarriere jemals schon mit Datenbankprogrammierung auseinandergesetzt haben, dann wissen Sie, dass alle Konzepte, die Microsoft in diesem Zusammenhang vorgestellt hat, zwar immer brauchbar, aber nie wirklich das Gelbe vom Ei waren.

Es geht unter anderem um das Fehlen einer richtigen Umsetzung eines Konzeptes, das man unter dem Schlagwort O/ORM kennt. O/ORM steht für Object Relational Mapping, und dabei handelt es sich um eine Programmiertechnik, um Daten zwischen den beiden verschiedenen, eigentlich nicht kompatiblen Typsystemen »Datenbanken« und denen der objektorientierten Programmierung mehr oder weniger automatisch zu konvertieren.

Wie Sie es in den vorangehenden Kapiteln schon erfahren konnten, verwendet man am besten Business-Objekte, um mit Daten in einer objektorientierten Programmiersprache umzugehen. Unter »Business-Objekt« versteht man im einfachsten Fall eine simple Klasse, die keine weitere Aufgabe hat, als eine Daten-Entität wie eine Kontaktadresse oder einen Artikel zu speichern. Mehrere dieser Adressen oder Artikel legt man dann in Auflistungen ab.

Datenbanken hingegen speichern ihre einzelnen Datensätze (also eine einzelne Adresse oder einen einzelnen Artikel) in Tabellen. In der Regel verwendet man dann hier die berühmten SQL-Abfragen, um Daten in verschiedenen Tabellen miteinander zu kombinieren, zu filtern und zu selektieren.

Die Aufgabe einer O/ORM ist es nun, diese beiden Welten auf eine möglichst einfache Art und Weise miteinander zu kombinieren – denn bislang war das Aufgabe des Entwicklers, und die sah exemplarisch folgendermaßen aus: Wollte man beispielsweise die Adressen eines bestimmten Postleitzahlbereichs aus einer SQL Server-Datenbank in einer Windows-Forms-Anwendung darstellen (wobei es hier im Folgenden nicht um die konkrete Programmierung sondern nur um das grundlegende Konzept geht), war in etwa Folgendes zu tun:

- Eine Windows Forms-Anwendung baut eine Verbindung zu einer Datenbank auf. Dabei muss sie sich an bestimmte Konventionen halten und vor allem provider-spezifische Klassen verwenden, um diese Verbindung herzustellen. Eine Verbindung zu einem SQL-Server erfordert so unter dem .NET Framework das `SqlConnection`-Objekt, eine Verbindung zu einer Access-Datenbank das `OleDbConnection`-Objekt und eine Verbindung zu einer Oracle-Datenbank eben ein `OracleConnection`-Objekt.
- Dieses Objekt überträgt eine Selektionsabfrage in Form eines `Command`-Objekts. Auch dieses `Command`-Objekt ist wieder provider-spezifisch (`OleDbCommand`, `OracleCommand`, `SqlCommand`); zumal können sich die einzelnen Abfragetexte – auch wenn sie alle ANSI-SQL-basierend sind – schon ein wenig voneinander unterscheiden.¹ Um beispielsweise alle Adressen der Postleitzahlgebiete zwischen 50000 und 59999 abzufragen und die Ergebnisse nach Ort und Namen zu sortieren, lautet die entsprechende Abfrage:

```
SELECT [Name], [Vorname], [Strasse], [PLZ], [ORT] FROM [Adressen]
WHERE [PLZ]>='50000' AND [PLZ] <='59999'
ORDER BY [Ort], [Name]
```

- Zudem muss der SQL String, mit dem die Abfragen oder UPDATES gesendet werden, syntaktisch korrekt formuliert werden, d.h. Sie müssen SQL »können«. VB.NET hat aber keinerlei Kenntnis über die korrekte SQL-Syntax der entsprechenden Datenbank, sondern übergibt die Zeichenfolge genauso an

¹ Beispiel Datensätze löschen: In Access heißt es `DELETE * FROM Tabelle WHERE...`, in Oracle `DELETE Tabelle WHERE...` in SQL Server `DELETE FROM Tabelle WHERE...` (auch wenn SQL Server auch die Oracle-Syntax »kann«, da das ANSI-SQL ist).

die Datenbank. Sollte sich in dieser ein Fehler befinden, so tritt dieser erst zur Laufzeit auf. Die Entwicklungsumgebung, hier also das Visual Studio, hat keine Möglichkeiten, die Syntax bereits zur Entwurfszeit zu überprüfen. Ganz anders ist es aber mit VB-Code selbst: schreiben Sie die Zeile

```
dim as Integer=6-+2//8
```

werden Sie natürlich schon zur Entwurfszeit, also im Moment des Schreibens darüber informiert, dass hier etwas nicht stimmen kann.

- Anschließend können die Daten feldweise ausgelesen werden. Im einfachsten Fall funktioniert das zellenweise – also nacheinander kommen Name des ersten Datensatzes, dann Vorname, dann Straße, dann Postleitzahl und schließlich der Ort. Weiter geht es mit Name des zweiten Datensatzes, usw. Und diese Ergebnistexte, die nacheinander vom SQL Server zurückgeliefert werden, müssen jetzt wiederum irgendwo gespeichert werden – beispielsweise in einer entsprechenden Auflistung, deren einzelne Elemente idealerweise den Schemainformationen der Datenbanktabelle entsprechen (die Eigenschaften eines Elements sollten sich also auf die einzelnen Spalten der Datenbanktabelle abbilden).
- Wenn das Programm den Anwender dann anschließend die Daten in den Business-Objekten hat verändern lassen – beispielsweise indem die Daten in einer Maske editierbar gemacht wurden – müssen sie aus den Masken zurück in die Business-Objekte und schließlich dann zurück in die Datenbank. Dazu muss das Programm alle Business-Objekte durchlaufen, schauen, ob sich die Daten dort geändert haben (dazu sollte es die Ursprungsdaten natürlich auch gespeichert haben, da es sonst ja gar nicht entscheiden kann, ob es Daten an den Business-Objekten gegeben hat), und dann für jedes Business-Objekt, das sich geändert hat, eine entsprechende SQL-INSERT- bzw. UPDATE-Anweisung generieren, damit die Daten auch in den entsprechenden Tabellen der Datenbank aktualisiert werden.

Würde man das alles manuell machen, wäre das ein ziemlicher Coding-Aufwand. Und aus diesem Grund gab es auch bisher schon entsprechende Hilfsmittel – beispielsweise DataSets – mit deren Hilfe sich der ganze Kommunikationsvorgang extrem vereinfachen ließ. Abfragen an den SQL Server waren allerdings immer noch an den SQL Server zu stellen – und das Selektieren oder Sortieren von Business-Daten im Speicher funktionierte damit völlig anders, als die eleganten SQL-Abfragen, die an die verschiedenen SQL-Server gestellt wurden.

Das wird mit LINQ anders. Mit LINQ formulieren Sie SQL-Abfragen innerhalb des Quellcodes und zwar unabhängig von der benutzten Datenbank und deren spezieller Syntax. Und wenn Sie vorher durch entsprechende Einstellungen gesagt haben, dass diese sich an eine SQL-Server-Datenbank richten sollen, dann sorgt LINQ dafür, dass diese im Quelltext hinterlegte Abfrage eben SQL Server-kompatibel dort hingelangt. LINQ erlaubt es aber mit der gleichen Abfragesyntax, auch XML-Dokumente abzufragen. Oder simple Auflistungen. Oder Inhalte von DataSets.

Diese verschiedenen Dinge, gegen die Sie eine LINQ-Abfrage richtigen können, nennen sich LINQ-Datenprovider. SQL Server gibt es als LINQ-Datenprovider. Auflistungen (*Collections*) auch. XML kann auch ein solcher sein. Und DataSets können auch LINQ-Datenprovider sein. So gibt es nicht nur LINQ, sondern immer auch etwas, auf das sich LINQ bezieht. Und im Moment² gibt es die folgenden Provider für LINQ:

² Stand: 20.10.2008

- *LINQ to Objects*: Damit können Sie SQL-ähnliche Abfragen an Auflistungen stellen. Und zwar an alle Auflistungen, die die Schnittstelle `IEnumerable` implementieren – und das sind die meisten. Kapitel 32 beschäftigt sich mit LINQ to Objects.
- *LINQ to SQL*: Damit können Sie SQL-ähnliche Abfragen direkt an den SQL Server stellen. Kapitel 34 zeigt Ihnen, wie das funktioniert.
- *LINQ to XML*: Mit XML als Zielprovider können Sie Abfragen direkt auf XML-Dokumente loslassen, um eine Auflistung entsprechender Elemente zu bekommen. Kapitel 33 zeigt Ihnen, wie das funktioniert.
- *LINQ to DataSets*: Funktioniert in etwa wie *LINQ to Objects*, nur dass eben `DataSets` abgefragt werden können, und dass es zusätzliche Abfragemöglichkeiten gibt, bei der aus einer LINQ-Abfrage ein `DataSet` bzw. eine `DataTable` zurückgeliefert wird. Aus Platzgründen werden wir auf diese Technologie nicht näher eingehen; wenn Sie allerdings Erfahrungen mit ADO.NET haben, werden Sie nach dem Studium dieses und des nächsten Kapitels problemlos LINQ to `DataSets` einsetzen können.
- *LINQ to Entities*: Funktioniert prinzipiell wie *LINQ to SQL*, ist aber in Sachen SQL eine andere, alternative Implementierung, die aber zusätzlich den Vorteil hat, dass nicht nur der SQL Server der Abfrageempfänger sein muss, sondern es auch weitere auf ADO.NET basierende Provider dafür gibt. Darüber hinaus lassen sich bei diesem LINQ-Provider viel detailliertere Mapping-Szenarien zwischen dem verwendeten Objektmodell (konzeptionelles Modell) und dem Datenbankmodell (Speichermodell) konstruieren. LINQ to Entities finden Sie in Kapitel 35 beschrieben. Wichtig dafür zu wissen: LINQ to Entities gibt es erst ab Service Pack 1 für Visual Studio 2008 bzw. mit dem SP1 für das .NET Framework 3.5. Mehr dazu in Kapitel 35.

Wie »geht« LINQ prinzipiell

LINQ erweitert die .NET-Sprachen um Abfrageausdrücke. Was dabei genau abgefragt wird, ist Sache des jeweiligen LINQ-Providers – das haben wir im vergangenen Abschnitt schon erfahren. Das *Wie* ist dabei aber eher entscheidend, denn es ist für alle LINQ-fähigen Entitäten (Objects, XML, SQL, etc.) gleich.

BEGLEITDATEIEN

Im Verzeichnis

...\\VB 2008 Entwicklerbuch\\E - LINQ\\Kapitel 31\\ErsteLinqDemo

finden Sie eine Konsolen-Beispielanwendung, die mit Business-Objekten die Fähigkeiten von LINQ to Objects eindrucksvoll demonstriert. Das Programm konstruiert dabei folgendes Szenario: Es erstellt eine Kundenaufstellung – die einzelnen Elemente dieser Auflistung werden dabei in entsprechenden Business-Objekten gespeichert, deren Schema genau dem der folgenden Tabelle entspricht. Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Relation zwischen Artikel und Käufer-Tabellen										
ID	Nachname	Vorname	Straße	PLZ	Ort					
1	Heckhuis	Jürgen	Wiedenbrücker Str. 47	59555	Lippstadt					
2	Wördehoff	Angela	Erwitter Str. 33	01234	Dresden					
3	Dröge	Ruprecht	Douglas-Adams-Str. 42	55544	Ratingen					
4	Dröge	Ada	Douglas-Adams-Str. 42	55544	Ratingen					
5	Halek	Gaby „Doc“	Krankenhausplatz 1	59494	Soest					
6										
IdGekauftVon										
ArtikelNummer										
1	9-445	Die Tore der Welt	Bücher, Belletristik	19,90	2					
3	3-537	Visual Basic 2005 - Das Entwicklerbuch	Bücher, EDV	59,00	2					
3	3-123	SQL Server 2000 - So gelingt der Einstieg	Bücher, EDV	19,90	1					
5	5-312	SQL Server 2008 - Das Profi-Buch	Bücher, EDV	39,90	2					

Abbildung 31.1 Das LINQ-Beispielprogramm legt zwei Tabellen an, die nur logisch über die Spalten-ID miteinander verknüpft sind

Das Beispielprogramm definiert ferner eine zweite Tabelle, die Daten mit gekauften Artikeln enthält. In dieser Tabelle ist jedoch der Name des Käufers nicht mehr enthalten; stattdessen gibt es nur eine ID in der Gekauften-Artikel-Tabelle, mit der man in der ersten nachschlagen und so die eigentlichen Käuferdaten ermitteln kann.

Und jetzt kommt das eigentliche Problem, bzw. die eigentliche Aufgabe: Das Programm soll die Daten so aufbereiten, dass die Kunden nicht nur nacheinander nach Nachnamen sortiert angezeigt werden; das Programm soll ebenfalls ermitteln, welcher Kunde wie viel Umsatz gemacht hat, und es soll die gekauften Artikel untereinander auflisten. Dabei soll es aber nur die Kunden berücksichtigen, die mehr als 1.000 Euro erzielt haben und in einem definierten Postleitzahlgebiet wohnen, etwa wie folgt:

- 5: Lehnert, Theo - 11 Artikel zu insgesamt 1064,10 Euro
 - Details:
 - 1-234: Das Leben des Brian(3 Stück für insgesamt 254,85 Euro)
 - 1-234: Das Leben des Brian(1 Stück für insgesamt 14,95 Euro)
 - 3-123: Das Vermächtnis der Tempelritter(1 Stück für insgesamt 29,95 Euro)
 - 3-537: Visual Basic 2005 - Das Entwicklerbuch(2 Stück für insgesamt 169,90 Euro)
 - 3-537: Visual Basic 2005 - Das Entwicklerbuch(1 Stück für insgesamt 89,95 Euro)
 - 5-506: Visual Basic 2008 - Das Entwicklerbuch(1 Stück für insgesamt 79,95 Euro)
 - 5-518: Visual Basic 2008 - Neue Technologien - Crashkurs(1 Stück für insgesamt 4,95 Euro)
 - 7-321: Das Herz der Hölle(2 Stück für insgesamt 99,90 Euro)
 - 9-009: Die Wächter(2 Stück für insgesamt 39,90 Euro)
 - 9-009: Die Wächter(3 Stück für insgesamt 254,85 Euro)
 - 9-646: Was diese Frau so alles treibt(1 Stück für insgesamt 24,95 Euro)
- 10: Weigel, Momo - 14 Artikel zu insgesamt 1118,70 Euro
 - Details:
 - 4-444: Harry Potter und die Heiligtümer des Todes(1 Stück für insgesamt 34,95 Euro)
 - 1-234: Das Leben des Brian(1 Stück für insgesamt 49,95 Euro)
 - 2-424: 24 - Season 6 [UK Import - Damn It!](2 Stück für insgesamt 89,90 Euro)
 - 2-424: 24 - Season 6 [UK Import - Damn It!](3 Stück für insgesamt 74,85 Euro)
 - 2-424: 24 - Season 6 [UK Import - Damn It!](1 Stück für insgesamt 4,95 Euro)
 - 3-534: Mitten ins Herz(3 Stück für insgesamt 59,85 Euro)

```

3-537: Visual Basic 2005 - Das Entwicklerbuch(3 Stück für insgesamt 74,85 Euro)
3-543: Microsoft Visual C# 2005 - Das Entwicklerbuch(1 Stück für insgesamt 44,95 Euro)
3-543: Microsoft Visual C# 2005 - Das Entwicklerbuch(1 Stück für insgesamt 29,95 Euro)
5-401: Programmieren mit dem .NET Compact Framework(1 Stück für insgesamt 29,95 Euro)
7-321: Das Herz der Hölle(1 Stück für insgesamt 89,95 Euro)
9-009: Die Wächter(3 Stück für insgesamt 179,85 Euro)
9-423: Desperate Housewives - Staffel 2, Erster Teil(2 Stück für insgesamt 99,90 Euro)
9-445: Die Tore der Welt(3 Stück für insgesamt 254,85 Euro)

```

LINQ ermöglicht es, aus zwei Auflistungen, die mit den Schemainformationen, wie sie auch in Abbildung 31.1 zu sehen sind, eine neue Auflistung zu erstellen, die das hier Gezeigte mit den genannten Konventionen einfach hintereinander ausgibt. Das entsprechende Listing dazu sieht folgendermaßen aus:

```

Module LinqDemo
Sub Main()
    Dim adrListe = Kontakt.Zufallskontakte(10)
    Dim artListe = Artikel.Zufallsartikel(adrListe)

    Dim adrListeGruppiert = From adrElement In adrListe
                           Join artElement In artListe
                           On adrElement.ID Equals artElement.IDGekauftVon
                           Select adrElement.ID, adrElement.Nachname,
                                  adrElement.Vorname, adrElement.PLZ,
                                  artElement.ArtikelNummer, artElement.ArtikelName,
                                  artElement.Anzahl, artElement.Einzelpreis,
                                  Postenpreis = artElement.Anzahl * artElement.Einzelpreis
                           Order By Nachname, ArtikelNummer
                           Where (PLZ > "0" And PLZ < "50000")
                           Group ArtikelNummer, ArtikelName,
                                 Anzahl, Einzelpreis, Postenpreis
                           By ID, Nachname, Vorname
                           Into Artikelliste = Group, AnzahlArtikel = Count(),
                                 Gesamtpreis = Sum(Postenpreis)
                           Where Gesamtpreis > 1000

    For Each KundenItem In adrListeGruppiert
        With KundenItem
            Console.WriteLine(.ID & ": " & .Nachname & ", " & .Vorname & " - " &
                            .AnzahlArtikel & " Artikel zu insgesamt " & .Gesamtpreis & " Euro")
            Console.WriteLine("      Details:")
            For Each ArtItem In KundenItem.Artikelliste
                With ArtItem
                    Console.WriteLine("          " & .ArtikelNummer & ": " & .ArtikelName &
                                    "(" & .Anzahl & " Stück für insgesamt "
                                    & (.Einzelpreis * .Anzahl).ToString("#,##0.00") & " Euro)")
                End With
            Next
        End With
    Next
    Console.ReadKey()
End Sub
End Module

```

Wenn Sie sich dieses Listing anschauen, dann geht es Ihnen wahrscheinlich wie jedem, der sich das erste Mal mit SQL-Abfragen oder – wie hier im Beispiel von LINQ – mit an SQL angelehnten Abfragen beschäftigt, und es stellen sich Ihnen zwei Fragen:

- Wie formuliere ich eine Abfrage, um genau das Datenresultat zu bekommen, das ich mir vorstelle?
- Wie funktioniert LINQ, und wie gliedert der Compiler die LINQ-Abfrage in das Klassenkonzept und die Sprachelemente des .NET Frameworks ein?

Wir beginnen mit der Beantwortung der letzten Frage, denn die erste wird sich automatisch erledigen, wenn Sie verstanden haben, wie der Compiler diese Art von Abfragen verarbeitet und was im Grunde genommen dabei passiert.

Intern lebt LINQ und leben LINQ-Abfragen von Lambda-Ausdrücken. Und um zu verstehen, wie LINQ funktioniert, sollten Sie Lambda-Ausdrücke (letzter Abschnitt in Kapitel 14), Generics (Kapitel 23 – die Nullable-Datentypen in Kapitel 11 sind auch einen Blick wert!), Auflistungen und gerade auch generische Auflistungen (Kapitel 22 und auch Kapitel 23) verstanden haben.

Dabei sollten Sie im Auge behalten, dass LINQ für seine Ausdrücke eine Syntax benutzt, die zwar reservierte Worte der SQL Sprache benutzt (Select, From etc.) aber nicht SQL ist – auch kein SQL-Dialekt. Auch die Logik, die Art und Weise, wie Abfragen formuliert werden, weicht spätestens bei komplexen Abfragen doch sehr stark von SQL ab. Insofern werden Kenner der SQL-Syntax mit LINQ ebensoviele, wenn auch andere Probleme haben wie Neueinsteiger.

Erweiterungsmethoden als Basis für LINQ

Wie in Kapitel 11 schon angedeutet, basiert LINQ als allererstes auf einem Satz an Funktionen, die als Erweiterungsmethoden implementiert sind. Mit ihrer Hilfe können Entwickler bereits definierte Datentypen um benutzerdefinierte Funktionen erweitern, *ohne* eben einen neuen, vererbten Typ zu erstellen. Erweiterungsmethoden sind eine besondere Art von statischen Methoden, die Sie jedoch wie Instanzmethoden für den erweiterten Typ aufrufen können: Für in Visual Basic (und natürlich auch C#) geschriebenen Clientcode gibt es keinen sichtbaren Unterschied zwischen dem Aufrufen einer Erweiterungsmethode und den Methoden, die in einem Typ tatsächlich definiert sind.

Erweiterungsmethoden sind also im Grunde genommen nur eine Mogelpackung, denn sie bleiben das, was sie sind: Simple statische Methoden, die sich, wie gesagt, lediglich wie Member-Methoden der entsprechenden Typen *anfühlen*. Ein kleiner Code-Ausschnitt soll das verdeutlichen:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\E - LINQ\\Kapitel 31\\ErweiterungsmethodenDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

'Wichtig: Das Extension-Attribut ist in diesem Namespace versteckt
Imports System.Runtime.CompilerServices

'Wichtig: Der Namespace mit der Erweiterungsmethode für einen Typ muss importiert werden!
Imports ErweiterungsmethodenDemo.MeineErweiterungsmethode

```

Module Hauptmodul

Sub Main()
    Dim t = "Dies ist die erste Zeile!\nUnd dieses hier die zweite."
    Console.WriteLine(t.Formatted)

    'Auf Taste warten.
    Console.ReadKey()
End Sub

End Module

Namespace MeineErweiterungsmethode

Module ErweiterungsmethodenModul

'Das Extension-Attribut kennzeichnet eine Methode
'als Erweiterungsmethode. Der Typ des ersten Parameters
'bestimmt den Typ, der erweitert werden soll.
<Extension()
    Public Function Formatted(ByVal text As String) As String
        'Aus \n ein CR+LF machen.
        Return text.Replace("\n", vbNewLine)
    End Function

End Module
End Namespace

```

Dieses kleine Beispiel erweitert den String-Datentyp um eine Methode `Formatted`. Diese Methode soll einen Text, der eine Formatierungsanweisung á la C# und C++ enthält (nämlich »\n« für das Einfügen eines Zeilenumbruchs) umsetzen. Im Beispiel kann also über die `Formatted`-Methode ein String ausgegeben werden, der einen Zeilenumbruch als Steuerzeichen enthält.³ Damit der String-Datentyp eine neue Methode erhalten kann, ist folgendes notwendig:

- Die Methode muss als statische Methode definiert werden. Da in Visual Basic .NET alle Methoden statisch sind, die sich in einem Modul befinden, packt man Erweiterungsmethoden in Visual Basic .NET am besten in ein solches Modul, wie hier im Beispieldatei zu sehen.
- Damit die Methode als Erweiterungsmethode zum Datentyp erkannt wird, benötigt man ein Attribut und zwar das `ExtensionAttribute`. Mit diesem Attribut wird die Erweiterungsmethode markiert. Damit Sie auf das `ExtensionAttribute` zugreifen können, müssen Sie den Namespace `System.Runtime.CompilerServices` einbinden.
- Damit der Typ auf die Erweiterungsmethode zugreifen kann, muss der Namespace, der die Erweiterungsmethode enthält, importiert werden. Das gilt natürlich nur für den Fall, dass sich die Namespaces des Kontextes, in dem sich der erweiterte Typ befindet, von der Erweiterungsmethode, um die erweitert wird, unterscheiden. Deswegen finden Sie die entsprechende Imports-Anweisung am Anfang der Codedatei.

³ Wichtig zu wissen: Das ist damit aber nicht dasselbe wie in C# oder C++, bei denen die Umsetzung dieser Steuerzeichen schon zur Compile-Zeit »ausgetauscht« werden. Hier in unserem Beispiel verschwenden wir Prozessorzeit, um diese Umsetzung der Steuerzeichen zur Laufzeit vorzunehmen.

Das Erweitern von vorhandenen Typen auf diese Weise mag ja für »Convenience Purposes« (siehe Kapitel 19) schon einigermaßen sinnvoll sein. Aber was die Infrastrukturschaffung für LINQ anbelangt, kommt jetzt der geniale Trick: Da Erweiterungsmethoden alle Typen als ersten Parameter aufweisen können, können ihnen auch Schnittstellen als Typ übergeben werden. Das führt aber zwangsläufig dazu, dass alle Typen, die diese Schnittstelle implementieren, in Folge auch mit den entsprechenden scheinbar zusätzlichen Methoden ausgestattet sind. In Kombination mit Generics und Lambda-Ausdrücken ist das natürlich eine coole Sache, da die eigentliche Funktionalität durch – je nach Ausbaustufe der Überladungsversionen der statischen Erweiterungsmethoden – einen oder mehrere Lambda-Ausdrücke gesteuert wird und der eigentlich zu verarbeitende Typ eben durch die Verwendung von Generics erst beim späteren Typisieren (Typ einsetzen) ans Tageslicht kommt. Mit diesem Kunstgriff schafft man im Handumdrehen eine komplette, typsichere Infrastruktur beispielsweise für das Gruppieren, Sortieren, Filtern, Ermitteln oder sich gegenseitige Ausschließen von Elementen aller Auflistungen, die eine bestimmte, um Erweiterungsmethoden ergänzte Schnittstelle implementieren – beispielsweise bei `IEnumerable(T)` um die statischen Methoden der Klasse `Enumerable`. Oder kurz zusammengefasst: Alle Auflistungen, die `IEnumerable(T)` einbinden, werden scheinbar um Member-Methoden ergänzt, die in `Enumerable` zu finden sind, und das sind genau die, die für LINQ benötigt werden, um Selektionen, Sortierungen, Gruppierungen und weitere Funktionalitäten von Daten durchzuführen. Es ist aber nur scheinbar so – die ganze Funktionalität von LINQ wird letzten Endes in eine rein prozedural aufgebaute Klasse mit einer ganzen Menge statischer Funktionen delegiert. Bevor wir uns im Detail mit der LINQ-typischen Abfragesyntax beschäftigen, wollen wir als erstes diese Erweiterungsmethoden ein wenig genauer unter die Lupe nehmen.

Dabei sei Folgendes angemerkt: Die kommenden Abschnitte beschreiben die wichtigsten Erweiterungsmethoden an verschiedenen Beispielen; sie erheben aber keinen Anspruch auf Vollständigkeit. Vielmehr geht es in den Abschnitten hauptsächlich darum, ein Verständnis für den Kontext zu vermitteln, in dem diese Methoden später bei der Verwendung der konkreten LINQ-Syntax durch den Visual Basic-Compiler zum Einsatz gelangen.

Die Where-Methode

Das folgende Listing demonstriert die `Where`-Methode. Diese Methode ist eine Erweiterungsmethode für `IEnumerable(Of t)`, und steht damit auch »normalen« Arrays zu Verfügung. Sie iteriert automatisch durch alle Elemente, und ruft für jedes Element eine *Predicat*-Bedingung (einen Delegaten, der für die Bedingungsprüfung jedes Elements zuständig ist – mehr zu Predicats, oder Prädikaten, finden Sie im Kapitel 23) auf. Anstelle einer Delegatenvariablen kann hier natürlich auch ein Lambda-Ausdruck angegeben werden. Wird dieser Ausdruck als *wahr* ausgewertet – trifft die entsprechende Bedingung also zu –, wird das betroffene Element in eine neue Ergebnisliste überführt, die vom generischen Typ `IEnumerable` des entsprechenden Quelltyps ist: Im ersten Beispiel ist der Quelltyp vom Typ `Integer`, im zweiten vom Typ `String`.

```
Public Sub WhereBeispiel()
    ' Where iteriert durch alle Elemente der angegebenen
    ' Auflistung und führt für jedes Element die Prüfung in der Lambda-Funktion
    ' durch. Trifft sie zu, wird das entsprechende Element
    ' in die neue Auflistung überführt.
    ' Hier der erste Versuch mit Integer-Elementen ...
    Dim fnums = ziffern.Where(Function(einIntElement) einIntElement < 6)
    Console.WriteLine("Ziffern < 6")
    For Each intElement In fnums
        Console.WriteLine(intElement)
```

```

Next
Console.WriteLine("-----")
' ... und hier ein weiterer mit Strings.
Dim fStrings = Namen.Where(Function(einName) einName.StartsWith("K"c))
Console.WriteLine("Namen die mit 'K' beginnen")
For Each strElement In fStrings
    Console.WriteLine(strElement)
Next
End Sub

```

Die Select-Methode

Weiter geht es mit der Select-Methode. Die Select-Methode ist in der Lage, ein Element der einen Auflistung gegen das entsprechende Elemente einer anderen Auflistung anderer Typs aufgrund seiner Ordinalnummer auszutauschen.

Auch hierzu ein Beispiel: Wir haben zwei Arrays, nämlich die folgenden beiden:

```

Private ziffern As Integer() = New Integer() {9, 1, 4, 2, 5, 8, 6, 7, 3, 0}
Private ziffernNamen As String() = New String() {"null", "eins", "zwei", "drei",
                                                "vier", "fünf", "sechs", "sieben", _
                                                "acht", "neun"}

```

Mithilfe der Select-Anweisung können wir nun eine Auflistung aufbauen, die aufgrund der Inhalte der Elemente des ersten Arrays die Elemente des zweiten Arrays findet. Die folgende Methode gibt daher ...

```

Public Sub SelectBeispiel()
    ' Hier wird Select verwendet, um ein Element auf Grund seiner
    ' Wertigkeit durch ein zweites in einer anderen Auflistung zu ersetzen.
    Dim ziffernListe = ziffern.Select(Function(einIntElement) ziffernNamen(einIntElement))

    Console.WriteLine("Ziffern:")
    For Each s As String In ziffernListe
        Console.WriteLine(s)
    Next
End Sub

```

... das folgende Ergebnis aus ...

```

Ziffern:
neun
eins
vier
zwei
fünf
acht
sechs
sieben
drei
null

```

... das exakt der Reihenfolge der Inhalte des ersten Arrays entsprach.

Anonyme Typen

Nun können wir dieses Spielchen mit der Select-Methode noch ein kleines bisschen weitertreiben, und lernen in diesem Zusammenhang das erste Mal den sinnvollen Einsatz anonymer Typen kennen. Anonyme Typen sind Klassen, die keinen Namen haben und direkt als Rückgabetypen – beispielsweise im Rahmen einer Select-Methode – definiert werden können.

Nun werden Sie denken: Wenn der Typ keinen Namen hat – wie soll ich dann eine Variable definieren von diesem nicht benannten Typen, oder gar eine Auflistung, die auf Basis eines anonymen Typs definiert wird? Möglich wird das auf Basis von lokalen Typrückschlüssen (siehe Kapitel 11) und Generics (Kapitel 23). Wir erinnern uns: Lokaler Typrückschluss bewirkt, dass Typen aufgrund ihrer initialen Zuweisung während der Deklarierung definiert werden – und das funktioniert natürlich auch für Typen, die keinen Namen haben, anonyme Typen eben:

```
Public Sub AnonymeklasseDemo()
    Dim element = New With {.Elementname = "Ein Element", _
                           Dim element As <anonymer Typ> .ElementID = 42}
    element.
    End Sub
```

Abbildung 31.2 Anonyme Klassen erlauben das Definieren von Typen, die keine Namen haben. Dank lokalem Typrückschluss lassen sich solche Typen aber dennoch an Objektvariablen zuweisen.

Typrückschluss für generische Typparameter

Für unser erweitertes Select-Beispiel brauchen wir eine weitere Technik, die sich Typrückschluss für generische Typparameter nennt. Dabei geht es darum, dass generische Funktionen ihre Typparameter über die ihnen tatsächlich übergebenen Parameter rückschließen, beispielsweise wie es in der folgenden Methode passiert:

```
Public Function ListeMitEinemElement(Of TSource)(ByVal dasEineElement As TSource) As
                                         List(Of TSource)
    Dim lst As New List(Of TSource)
    lst.Add(dasEineElement)
    Return lst
End Function
```

Über den Sinn oder Unsinn dieser Routine kann man natürlich streiten, denn ihre Aufgabe ist folgende: Ihr wird eine Instanz beliebigen Typs übergeben, und die Methode macht daraus eine Auflistung, der sie exakt dieses Element hinzufügt. Es geht dabei auch weniger um die Aufgabe der Methode, sondern wie ihr Parameter übergeben werden können. Um diese Routine ohne Typrückschluss für generische Typparameter zu verwenden, müssten Sie sie folgendermaßen verwenden:

```
Dim dasEineElement As Integer = 10
Dim liste As List(Of Integer) = ListeMitEinemElement(Of Integer)(dasEineElement)
```

Nun wird dasEineElement als Integer definiert, und die Tatsache, dass Sie der Methode eine Integer-Variable übergeben, reicht dem Compiler, die Information zu ergänzen, um aus der generischen Methode eben eine vom Typ Integer zu machen. Ausreichend ist also:

```
Dim dasEineElement As Integer = 10
Dim liste As List(Of Integer) = ListeMitEinemElement(dasEineElement)
```

Dank Typrückschluss kann es jetzt noch einen Schritt weitergehen: Die im oben gezeigten Listing angegebenen Typdefinitionen sind eigentlich auch redundant: Die Konstante 10 definiert dasEineElement ausreichend als Integer. Und damit funktioniert der Typrückschluss für Generics natürlich auch bei der Zuweisung der zurückgegebenen Auflistung. Damit ergibt sich folgende, immer noch typmäßig ausreichend aufschlussreiche Sparversion des Beispiels:

```
Dim dasEineElement = 10
Dim liste = ListeMitEinemElement(dasEineElement)
```

Und da wir auf diese Weise sämtliche explizite Typangaben eliminieren konnten, funktioniert diese Sparversion natürlich auch für anonyme Typen, wie die folgende Abbildung zeigt:

```
Dim liste = ListeMitEinemElement(New With {.ID = 42, .Elementname = "Adams"})
Dim liste As System.Collections.Generic.List(Of <anonymer Typ>)
```

Abbildung 31.3 Ist der übergebene Parameter ein alternder Typ, wird auch der generische Typparameter der generischen Methode anonym – genau wie die Auflistung, die die Methode in dessen Abhängigkeit zurückliefert

Mit diesem Wissen ist das Verständnis der folgenden Routine ebenfalls kein Problem mehr. Wenn Sie das folgende Beispiel laufen lassen ...

```
Public Sub SelectBeispielAnonym()
    ' Hier wird Select verwendet, um ein Element auf Grund seiner
    ' Wertigkeit durch ein zweites, anonymes in einer anderen Auflistung zu ersetzen.
    Dim anonymeListe = ziffern.Select(Function(einIntElement)
        New With {.OriginalWert = einIntElement,
                  .Ziffernname = ziffernNamen(einIntElement)})
    Console.WriteLine("Inhalt der anonymen Klassenelemente:")
    For Each anonymerTyp In anonymeListe
        Console.WriteLine(anonymerTyp.OriginalWert & ", " & anonymerTyp.Ziffernname)
    Next
End Sub
```

erhalten Sie das folgende Ergebnis:

```
Inhalt der anonymen Klassenelemente:
9, neun
1, eins
4, vier
2, zwei
5, fünf
8, acht
```

```
6, sechs  
7, sieben  
3, drei  
0, null
```

Und für diejenigen, die es genau interessiert, wie Methoden wie `Select` implementiert werden, sei die folgende Methode als Beispiel gezeigt, mit deren Hilfe sich Elemente einer Auflistung beliebigen Typs nummerieren lassen:

```
Public Function ItemNumberer(Of TSource, TResult)(ByVal source As IEnumerable(Of TSource), _  
    ByVal selector As Func(Of TSource, Integer, TResult)) As IEnumerable(Of TResult)  
  
    Dim retColl As New List(Of TResult)  
    Dim count = 1  
    For Each item In source  
        retColl.Add(selector(item, count))  
        count += 1  
    Next  
    Return retColl.AsEnumerable  
End Function
```

Diese Methode übernimmt eine generische `IEnumerable`-Auflistung eines beliebigen Typs und einen Lambda-Ausdruck, und liefert eine `IEnumerable`-Auflistung zurück, die durch den Rückgabewert des Lambda-Ausdrucks definiert wurde. Ziel ist es, durch alle Elemente der Quellaufstellung hindurchzuiterieren, und der aufrufenden Methode über den Lambda-Ausdruck die Möglichkeit zu geben, eine aktuelle Elementnummer (`count`) in die Zielaufstellung einzubauen. Mithilfe eines anonymen Typs könnte die Verwendung dieser Methode folgendermaßen aussehen:

```
Public Sub ItemNumbererDemo()  
    Dim numList = ItemNumberer(New String() {"eins", "zwei", "drei", "vier", "fünf", _  
        "sechs", "sieben", "acht", "neun", "zehn"}, _  
        Function(element, count) New With {.ID = count, _  
            .Element = element})  
  
    For Each numListItem In numList  
        Console.WriteLine(numListItem.ID & ": " & numListItem.Element)  
    Next  
  
End Sub
```

Auch bei Aufruf dieser Methode wird wieder mit Rückschließen generischer Typparameter gearbeitet. Die Methode `ItemNumberer` wird nicht mit `(Of Irgendwas, Irgendwas)` genauer spezifiziert. Stattdessen leitet sich der Typ `TSource` durch das übergebene String-Array, der Rückgabewert durch den übergebenen, anonymen Typ ab.

Die anschließende Ausführung der Methode ergibt folgendes Ergebnis:

```
1: eins  
2: zwei  
3: drei  
4: vier
```

```
5: fünf
6: sechs
7: sieben
8: acht
9: neun
10: zehn
```

Die OrderBy-Methode

Mit der generischen OrderBy-Erweiterungsmethode können Sie Auflistungen quasi nach beliebigen Kriterien sortieren. Das zu sortierende Element wird als Argument dem Lambda-Ausdruck übergeben, und der Lambda-Ausdruck entscheidet dann quasi, nach welchem Kriterium die Auflistung sortiert wird.

Bei einer einfachen Auflistung aus String-Elementen, ist das Kriterium ...

```
Public Sub OrderBy()
    'Für jedes Element liefert OrderBy das Sortierelement - bei einer Liste
    'von Strings ist das jedes Element selbst.
    Dim sortierteNamen = Namen.OrderBy(Function(einName) einName)
    
    For Each s In sortierteNamen
        Console.WriteLine(s)
    Next
End Sub
```

... gleich einem der Elemente, die sortiert werden sollen.

Anders wird das, wenn Sie Klasseninstanzen sortieren, wie das folgende Beispiel zeigt.

Zunächst die Klasse, die die Elemente stellt:

```
Public Class Kontakt
    Private myID As Integer
    Private myVorname As String
    Private myNachname As String
    Private myOrt As String
    
    Public Property ID() As Integer
        Get
            Return myID
        End Get
        Set(ByVal value As Integer)
            myID = value
        End Set
    End Property
    
    Public Property Vorname() As String
        Get
            Return myVorname
        End Get
        Set(ByVal value As String)
            myVorname = value
        End Set
    End Property
```

```

End Property

Public Property Nachname() As String
    Get
        Return myNachname
    End Get
    Set(ByVal value As String)
        myNachname = value
    End Set
End Property

Public Property Ort() As String
    Get
        Return MyOrt
    End Get
    Set(ByVal value As String)
        myOrt = value
    End Set
End Property
End Class

```

Dann die Definition der Elemente:

```

Module Module1

Private ziffern As Integer() = New Integer() {9, 1, 4, 2, 5, 8, 6, 7, 3, 0}
Private ziffernNamen As String() = New String() {"null", "eins", "zwei", "drei", _
                                                "vier", "fünf", "sechs", "sieben", _
                                                "acht", "neun"}

Private Namen As String() = New String() {"Jürgen", "Denise", "Angela", "Gaby", _
                                         "Andreas", "Katrin", "Klaus", "Arnold", _
                                         "Kerstin", "Anja", "Miriam", "Uta", "Momo", _
                                         "Leonie-Gundula"}

Private Kontakte() As Kontakt = New Kontakt() {
    New Kontakt With {.ID = 1, .Vorname = "Jürgen", .Nachname = "Heckhuis", .Ort = "Demohausen"}, _
    New Kontakt With {.ID = 2, .Vorname = "Ruprecht", .Nachname = "Dröge", .Ort = "Lippstadt"}, _
    New Kontakt With {.ID = 3, .Vorname = "Klaus", .Nachname = "Löffelmann", .Ort = "Demohausen"}, _
    New Kontakt With {.ID = 4, .Vorname = "Andreas", .Nachname = "Belke", .Ort = "Lippstadt"}, _
    New Kontakt With {.ID = 5, .Vorname = "Kerstin", .Nachname = "Lehnert", .Ort = "Ratingen"}, _
    New Kontakt With {.ID = 6, .Vorname = "Denise", .Nachname = "Wagner", .Ort = "Demohausen"}, _
    New Kontakt With {.ID = 7, .Vorname = "Anja", .Nachname = "Vielstedde", .Ort = "Lippstadt"}, _
    New Kontakt With {.ID = 8, .Vorname = "Angela", .Nachname = "Wördehoff", .Ort = "Demohausen"}, _
    New Kontakt With {.ID = 9, .Vorname = "Momo", .Nachname = "Weichel", .Ort = "Ratingen"}, _
    New Kontakt With {.ID = 10, .Vorname = "Leonie-Gundula", .Nachname = "Beckmann", _
                     .Ort = "Ratingen"}, _
    New Kontakt With {.ID = 11, .Vorname = "Miriam", .Nachname = "Sonntag", .Ort = "Lippstadt"} _
}
```

.

.

.

Und schließlich die OrderBy-Routine, die die entsprechenden Elemente sortiert:

```

Public Sub OrderBy()
    'Bei Klasseninstanzen liefert OrderBy die Instanz zum Sortieren,
    'und hier bestimmt die Eigenschaft, nach was sortiert wird.
    Dim sortierteKontakte = Kontakte.OrderBy(Function(einKontakt) einKontakt.Nachname)

    For Each kontaktElement In sortierteKontakte
        Console.WriteLine(kontaktElement.ID & ": " &
                           kontaktElement.Nachname & ", " &
                           kontaktElement.Vorname & ", " &
                           kontaktElement.Ort)
    Next
End Sub

```

Sortieren nach Eigenschaften, die per Variable übergeben werden

In vielen Anwendungen stößt man auf Szenarien, bei denen es notwendig wird, dass beispielsweise eine Auflistung von Objekten nach wechselnden Kriterien sortiert wird, die sich zur Laufzeit festlegen lassen sollen.

In diesem Fall müssen Sie den eigentlichen Sortierstring – beispielsweise die Eigenschaft, die diesen für ein zu sortierendes Element ermittelt – dynamisch abrufen. Das funktioniert nur über .NET-Reflection, eine Technik, die Sie in Kapitel 25 beschrieben finden.

Die OrderBy-Methode per Reflection mit dem entsprechenden Sortierstring zu versorgen, ist jedenfalls nicht nur möglich, sondern auch kein großes Problem.

Dabei wird der Inhalt des Sortierkriteriums nicht direkt mit dem Eigenschaftennamen des Elements gelesen, sondern über einen Umweg:

```

Public Sub OrderBy(ByVal propertyName As String)
    'Wenn die "Sortierspalte" frei wählbar sein soll, kann
    'das jeweilige Element nur über Reflection ermittelt werden:
    Dim sortierteNamen = Kontakte.OrderBy(Function(einKontakt)
                                             einKontakt.GetType.GetProperty(propertyName).GetValue(einKontakt, Nothing))

    'Funktioniert aber auch!
    For Each s In sortierteNamen
        Console.WriteLine(s.Nachname)
    Next
End Sub

```

1. Der Typ des Elements wird per GetType ermittelt.
2. GetProperty ermittelt anschließend ein so genanntes PropertyInfo-Objekt, das alle Informationen der Eigenschaft enthält und auch Zugriff auf den eigentlichen Wert gestattet. Da GetProperty der Eigenschaftenname als Argument übergeben wird, können die Eigenschafteninfos wie gewünscht dynamisch zur Laufzeit ermittelt werden. Man könnte diese Vorgehensweise quasi als »manuelles Late-Binding« bezeichnen.
3. GetValue ermittelt anschließend den eigentlichen Wert, und damit das Sortierkriterium.

Das Ergebnis zeigt, dass das Verfahren funktioniert, wenn die Methode folgendermaßen aufgerufen wird:

```
OrderBy("Nachname")
```

... führt zu folgender Ausgabe:

```
10: Beckmann, Leonie-Gundula, Ratingen
4: Belke, Andreas, Lippstadt
2: Dröge, Ruprecht, Lippstadt
1: Heckhuis, Jürgen, Demohausen
5: Lehnert, Kerstin, Ratingen
3: Löffelmann, Klaus, Demohausen
11: Sonntag, Miriam, Lippstadt
7: Vielstedde, Anja, Lippstadt
6: Wagner, Denise, Demohausen
9: Weichel, Momo, Ratingen
8: Wördehoff, Angela, Demohausen
```

Die GroupBy-Methode

GroupBy ermöglicht es, Elemente einer Auflistung nach bestimmten Kriterien zu selektieren und in mehrere, verschiedene Auflistungsinstanzen zu überführen. Die einfachste Möglichkeit, die GroupBy-Erweiterungsmethode zu verwenden, ist im folgenden Beispiel durchgeführt:

```
Public Sub GroupBy()
    Dim gruppierteNamen = Kontakte.GroupBy(Function(einKontakt) einKontakt.Ort)

    For Each gruppenElement In gruppierteNamen
        Console.WriteLine(gruppenElement.Key)
        For Each kontaktElement In gruppenElement
            Console.WriteLine("----> " & kontaktElement.Nachname & ", " & kontaktElement.Vorname)
        Next
    Next
End Sub
```

Wenn Sie dieses Beispiel ausführen, sehen Sie folgendes Ergebnis:

```
Demohausen
----> Heckhuis, Jürgen
----> Löffelmann, Klaus
----> Wagner, Denise
----> Wördehoff, Angela
Lippstadt
----> Dröge, Ruprecht
----> Belke, Andreas
----> Vielstedde, Anja
----> Sonntag, Miriam
Ratingen
----> Lehnert, Kerstin
----> Weichel, Momo
----> Beckmann, Leonie-Gundula
```

GroupBy hat also eine Auflistung zum Ergebnis, die ihrerseits wieder eine Auflistung kapselt. In der oberen Hierarchieebene gibt es eine Key-Eigenschaft, die dem Element entspricht, nach dem zuvor gruppiert wurde, und das den Zugriff auf die untergeordneten – also gruppierten – Elemente ermöglicht. Dementsprechend ist es am Geschicktesten, mit zwei ineinander verschachtelten For/Each-Schleifen durch alle Elemente zu iterieren, wie es das Beispiel auch zeigt.

In Kombination mit einer Select-Anweisung ist es darüber hinaus nicht nur möglich, sondern auch sinnvoll, eine neue Klasse anzulegen, die weiterführende Informationen über die Gruppen enthalten kann.

Das folgende Beispiel zeigt, wie eine solche Kombination realisiert werden kann:

```
Public Sub GroupBy()
    Dim gruppierteKontakte = Kontakte.GroupBy(Function(einKontakt)
                                                einKontakt.Ort).Select(Function(eineGruppe) _
                                         New With {.Ort = eineGruppe.Key, _
                                         .Anzahl = eineGruppe.Count, _
                                         .Elemente = eineGruppe.ToList})

    For Each gruppenElement In gruppierteKontakte
        Console.WriteLine(gruppenElement.Ort & " (" & gruppenElement.Anzahl & ")")
        For Each kontaktElement In gruppenElement.Elemente
            Console.WriteLine("---> " & kontaktElement.Nachname & ", " & kontaktElement.Vorname)
        Next
    Next
End Sub
```

In diesem Beispiel werden die vorhandenen Elemente in der Auflistung Kontakte nicht nur nach der Eigenschaft Ort gruppiert; die dabei entstehende Auflistung wird vielmehr in eine neue Auflistung übertragen, deren Schema durch einen anonymen Typ per Select-Methode definiert wird. Schön ist dabei auch zu erkennen, dass hier – am Beispiel der Eigenschaft Anzahl – Aggregat-Funktionen zum Einsatz kommen können, die Aufschluss über die eigentlichen Kontakt-Elemente geben, die den entsprechenden Gruppen-elementen zugeordnet sind. Die Elemente selber werden ebenfalls »weitergereicht«, indem sie durch die Eigenschaft `ToList`, die eine generische Kontakte-Auflistung zurückgibt – in der Gruppenauflistung gruppierteKontakte durch die Elemente-Eigenschaft verfügbar gemacht wird.

Kombinieren von LINQ-Erweiterungsmethoden

Nachdem Sie nunmehr ein paar der wichtigsten LINQ-Erweiterungsmethoden kennen gelernt haben, lassen Sie uns als nächstes einen Blick darauf werfen, wie sich diese Methoden miteinander vertragen.

Wie wir gelernt haben, geben alle diese Methoden Auflistungen zurück. Und diese Auflistungen haben alle eines gemeinsam: Sie implementieren, genau wie ihre Quellauflistungen – also die, aus denen sie entstanden sind – `IEnumerable(Of t)`. Das bedeutet aber weiterhin, dass die Ergebnisse der Erweiterungsmethoden sich wieder als Argument für eine weitere LINQ-Erweiterungsmethode nutzen lassen.

Ein Beispiel soll auch dies verdeutlichen:

```
Public Sub KombinierteErweiterungsmethoden()

    Dim ergebnisListe = Kontakte.Where(Function(einKontakt) einKontakt.Ort = "Lippstadt" _ 
        ).Select(Function(einKontakt) _
            New With {.ID = einKontakt.ID, _
                .LastName = einKontakt.Nachname, _
                .FirstName = einKontakt.Vorname, _
                .City = einKontakt.Ort} _ 
            ).OrderBy(Function(Contact) Contact.LastName)

    For Each contactItem In ergebnisListe
        Console.WriteLine(contactItem.ID & ": " & _
            contactItem.LastName & ", " & _
            contactItem.FirstName & " living in " & _
            contactItem.City)
    Next
End Sub
```

Wenn Sie die Erweiterungsmethoden dieses Beispiels in dieser Reihenfolge nacheinander angewendet sehen, ahnen Sie bestimmt bereits, was bei LINQ-Abfragen, wie Sie sie am Anfang dieses Kapitels kennen gelernt haben, wirklich passiert.

Aber lassen Sie uns nicht zwei Schritte auf einmal machen, sondern die Methodenkombination, die Sie im obigen Listing fett hervorgehoben sehen, erkläzungstechnisch Schritt für Schritt auseinander nehmen:

- Als erstes verarbeitet das Programm die Ausgangsliste Kontakte mithilfe der Where-Erweiterungsmethode. In diesem konkreten Beispiel generiert die Where-Methode eine neue, `IEnumerable(Of t)`-implementierende Auflistung, die nur jene Elemente enthält, deren Ort der Zeichenkette »Lippstadt« entspricht.
- Das Ergebnis dieser Auflistung ist Gegenstand der nächsten Erweiterungsmethode: Select. Select macht nichts anderes, als ein Element der ersten Auflistung durch ein anderes Element auszutauschen und dieses neue Element in einer abermals neu erstellten Auflistung einzugliedern. Die Select-Methode dieses konkreten Beispiels erstellt dazu eine anonyme Klasse, die prinzipiell den alten Elementen ähnlich ist, aber englische Eigenschaftsbezeichnungen enthält. Die alten Objekte, auf Basis der Klasse Kontakte, werden also als Basis für eine Reihe von neuen Objekten verwendet, die von der Select-Methode nacheinander angelegt werden. Wir erhalten so eine Auflistung mit anonymen Klasseninstanzen, die sich durch englische Eigenschaftsbezeichner auszeichnen.
- Diese Auflistung wird abermals Gegenstand einer Methode – dieses Mal ist es die Erweiterungsmethode OrderBy. Sie nimmt nun die anonymen Klasseninstanzen mit den englischen Eigenschaftsbezeichnern und sortiert diese nach Nachnamen (`Contact.LastName`) in einer wiederum neuen Auflistung ein.
- Diese zuletzt generierte Auflistung ist schließlich die Ergebnisauflistung, die der Objektvariablen `ergebnisListe` zugewiesen und in der anschließenden For/Each-Schleife ausgegeben wird:

```
4: Belke, Andreas living in Lippstadt
2: Dröge, Ruprecht living in Lippstadt
11: Sonntag, Miriam living in Lippstadt
7: Vielstedde, Anja living in Lippstadt
```

Vereinfachte Anwendung von LINQ – Erweiterungsmethoden mit der LINQ-Abfragesyntax

OK – viel zu sagen gibt es jetzt nicht mehr, ich hoffe, dass Ihnen – um es mit einer Metapher zu sagen – mein vielleicht etwas von hinten aufgezäumtes Pferd didaktisch genehm war.

Ein Blick auf das folgende Listing reicht meiner Meinung nach jetzt aus, um zu verstehen, was der Visual Basic-Compiler aus einem LINQ-Ausdruck macht, und wie er ihn – jedenfalls im Fall von *LINQ to Objects* – in eine Kombination aus Erweiterungsmethoden umsetzt.

Die Abfrage des folgenden Listings entspricht nämlich exakt ...

```
Public Sub KombinierteErweiterungsmethoden_à_la_LINQ()

    Dim ergebnisListe = From einKontakt In Kontakte
        Where einKontakt.Ort = "Lippstadt"
        Select ID = einKontakt.ID,
            LastName = einKontakt.Nachname,
            FirstName = einKontakt.Vorname,
            City = einKontakt.Ort
        Order By LastName

    For Each contactItem In ergebnisListe
        Console.WriteLine(contactItem.ID & ": " &
            contactItem.LastName & ", " &
            contactItem.FirstName & " living in " &
            contactItem.City)

    Next

End Sub
```

... der Methodenkombination des Listings im vorherigen Abschnitt. Und natürlich liefert es auch das exakt gleiche Ergebnis – probieren Sie es aus!

Kapitel 32

LINQ to Objects

In diesem Kapitel:

Einführung in LINQ to Objects	880
Der Aufbau einer LINQ-Abfrage	881
Kombinieren und verzögertes Ausführen von LINQ-Abfragen	887
Verbinden mehrerer Auflistungen zu einer neuen	891
Gruppieren von Auflistungen	894
Aggregatfunktionen	898

Einführung in LINQ to Objects

Nach der Lektüre des vorherigen Kapitels wissen Sie ja im Grunde genommen schon, wie LINQ to Objects funktioniert, was gibt es also in Sachen Einführung zu LINQ to Objects noch zu sagen? – Was die generelle, interne Funktionsweise anbelangt, im Grunde gar nichts. Was die Auswirkungen von LINQ to Objects auf Ihre tägliche Routine anbelangt vielleicht eine ganze Menge!

Letzten Endes geht es nämlich beim Einsatz von LINQ to Objects nicht primär darum, möglichst eine Datenbank mit Business-Objekten quasi im Managed Heap Ihrer .NET-Anwendung nachzubilden – auch wenn die Beispiele dieses Kapitels das vielleicht vermuten lassen.

Nein? – Nein!

LINQ to Objects soll es Ihnen ermöglichen, die im letzten Kapitel vorgestellte, an die SQL-Abfragesprache angelehnte Syntax auch für ganz andere Zwecke, an die Sie im Moment vielleicht noch gar nicht denken, in ihre Programme einzubauen. Doch dazu müssen Sie wissen, mit welchen Komponenten einer Abfrage Sie welche Ergebnisse erzielen können – und dabei soll Ihnen dieses Kapitel behilflich sein. Aus diesem Grund wählen wir als Beispielträger auch einfachste Objektauflistungen (übrigens exakt die des letzten Kapitels), damit Sie sich bei den Beispielen auf LINQ konzentrieren können und nicht von Funktionalitäten anderer Auflistungsklassen des Systems abgelenkt werden. Dennoch können Sie LINQ nicht nur für ihre selbstgeschriebenen Business-Klassen und –Auflistungen einsetzen. Natürlich lassen sich auch Verzeichnis-, Grafik- oder Steuerelement-Auflistungen mit LINQ bearbeiten und organisieren.

HINWEIS Dieses Kapitel kann übrigens dabei keinen Anspruch auf Vollständigkeit erheben – dazu ist das Thema LINQ einfach zu mächtig und umfangreich. Es wird Ihnen aber auf jeden Fall ausreichend Informationen, Beispiele und Anregungen zum Thema liefern, sodass Sie Ihr Wissen durch eigenes Ausprobieren und aufmerksames Lesen der Online-Hilfe sicherlich perfektionieren können.

Übrigens: Der Einsatz von LINQ ist immer sinnvoll, wenn Sie mit Auflistungen arbeiten müssen. Das gilt vor allen Dingen unter Beachtung des folgenden Aspektes, dem ich – wegen seiner Zukunftsträchtig- und -wichtigkeit – einen eigenen Abschnitt widmen will; sicherlich ein wenig spekulativ, aber deswegen nicht minder wahrscheinlich – jedenfalls nach meinem bescheidenen Ermessen!

Verlieren Sie die Skalierbarkeit von LINQ nicht aus den Augen!

Denken Sie daran: LINQ als *Engine* zur Verarbeitung von internen .NET-Auflistungen ist skalierbar. Worauf ich hinaus will, möchte ich an einem einfachen Denkmodell demonstrieren:

Stellen Sie sich vor, Sie benötigen in Ihrer Anwendung einen Algorithmus zur Sortierung einer Auflistung. Dann können Sie, was gleichermaßen unnötig und unklug wäre, einen eigenen Algorithmus dazu schreiben. Unnötig, weil es bereits ausreichend Sortieralgorithmen im .NET Framework für alle möglichen Auflistungstypen gibt, und unklug, weil Sie selbst dafür zuständig wären, Ihren Algorithmus zu optimieren, beispielsweise weil Sie bemerken, dass Sie – was passieren wird – es nur noch mit Multi-Core-Prozessoren zu tun haben, und Ihre Anwendung, weil sie ständig (aber nur auf einem Core des Prozessors) sortiert, nur 50%, 25% oder gar 12,5% der gesamt zur Verfügung stehenden Prozessorleistung ausschöpft. Ihr Algorithmus arbeitet nämlich nicht multithreaded.

Es gibt bei LINQ bereits eine Erweiterungsbibliothek im Beta-Stadium, – die so genannten Parallel Extensions to .NET Framework 3.5¹ – die durchzuführenden Operationen mit mehreren Prozessoren (oder eben mehreren Prozessor-Cores) zu parallelisieren – jedenfalls soweit das algorithmisch möglich ist. Eine LINQ-Abfrage würde dann eine Kombination von Erweiterungsmethoden »auslösen«, die ihre eigentliche Arbeit in mehrere Threads und damit mehrere Cores aufteilen – das Resultat wäre eindrucksvoll: Gleicher Programmaufwand, doppelte, vierfache oder sogar achtfache Leistung!

LINQ anstelle normaler, selbstgestrickter For/Each-Auflistungsverarbeitungsschleifen lohnt sich aus planungstechnischer Sicht also auf jeden Fall – mal ganz abgesehen davon, dass Ihr Coding-Aufwand auch geringer wird und damit die Fehleranfälligkeit Ihres Programms abnimmt.

Der Aufbau einer LINQ-Abfrage

LINQ-Abfragen beginnen stets² mit der Syntax

From bereichsVariable In DatenAuflistung

und offen gesagt: Wenn ich auch die Syntax aus Programmiersicht nachvollziehen konnte, so verstand ich zunächst nicht, warum es unter rein sprachlichen Aspekten ausgerechnet From heißen musste – was wird denn da eigentlich von der Bereichsvariablen genommen, entwendet oder verwendet?

Also begann ich, ein wenig im Internet zu recherchieren und stellte fest, dass nicht nur ich mir, sondern sich auch englische Muttersprachler diese Frage stellten, und so bemühte ich mich um ein wenig mehr Hintergrundinfos zur Entstehungsgeschichte der LINQ-Syntax bei jemanden, von dem ich glaubte, dass er es wissen müsse:

Lisa Feigenbaum vom Visual Basic Team über die Syntax von LINQ-Konstrukten

Je mehr ich darüber nachdachte, wieso ausgerechnet From (und nicht vielleicht Through oder For) eine LINQ-Abfrage einleitet, desto mehr ließ mich auch die Frage nicht mehr in Ruhe, wieso eine LINQ-Abfrage nicht vielmehr mit Select eingeleitet würde – so, wie wir Entwickler es schließlich schon seit Jahren von SQL Server-Abfragen gewohnt waren. In der glücklichen Lage, mit einen Insider, der es wissen musste, diskutieren zu können, frage ich Lisa Feigenbaum vom Visual Basic Team direkt zu diesem Thema. Und eine Antwort ließ nicht lange auf sich warten:

»From« deutet auf Quelle oder Ursprungsort hin. Der From-Ausdruck als solcher ist die Stelle, an der man die Abfragequelle bestimmt (beispielsweise den Datenkontext, eine Auflistung im Speicher, XML, etc.). Aus diesem Grund fanden wir diesen Terminus als angemessen. Darüber hinaus handelt es sich bei »From« auch um ein SQL-Schlüsselwort. Wir haben uns bei LINQ um größtmöglichen SQL-Erhält bemüht, sodass LINQ-Newbie, die bereits über SQL-Erfahrung verfügen, die Ähnlichkeit bereits »erfühlen« konnten. ►

¹ Diese Erweiterung, bei der zur Drucklegung dieses Buchs die Juni 2008 Community Technology Preview verfügbar war, können Sie unter dem IntelliLink F3201 herunterladen. Bitte beachten Sie, dass diese Erweiterung Beta-Status hat!

² Ausnahmen bilden reine Aggregat-Abfragen, die auch mit der Klausel Aggregate beginnen, wie im entsprechenden Abschnitt dieses Kapitels beschrieben.

Ein interessanter Unterschied zwischen LINQ und SQL, den wir jedoch einbauen mussten, ist die Reihenfolge [der Elemente] innerhalb des Abfrageausdrucks. Bei SQL steht Select vor From. In LINQ ist es genau umgekehrt.

Einer der großen Vorteile von LINQ ist, dass man mit verschiedenen Datenquellen in einem gemeinsamen Modell arbeiten kann. Innerhalb einer Abfrage arbeitet man nur mit Objekten, und wir können IntelliSense für diese Objekte zur Verfügung stellen. Der Grund dafür, dass wir From als erstes benötigen, ist, dass wir als erstes wissen müssen, von woher die Quelle stammt, über die wir die Abfrage durchführen, bevor wir die IntelliSense-Information zur Verfügung stellen können.³

Sprachlich zu verstehen ist From also im Kontext des gesamten ersten Abfrageteilausdrucks – also quasi From (bereichsVariable in Auflistung) – und bezieht sich *nicht* nur auf die Bereichsvariable, die hinter From steht. Und nachdem dieser Ausdruck nun auch sprachlich geklärt ist, fassen wir zusammen:

From leitet eine LINQ-Abfrage ein, und er bedeutet ins »menschliche« übersetzt: »Hier, laufe mal bitte durch alle Elemente, die hinter dem Schlüsselwort In stehen, und verwende die Bereichsvariable hinter From, um mit deren Eigenschaften bestimmen zu können, wie die Daten der Auflistung selektiert, sortiert, gruppiert und dabei in eine neue Elementliste (mit – bei Bedarf – Instanzen anderer Elementklassen) überführt werden können.

WICHTIG Wichtig für die Profi-T-SQL-ler unter Ihnen ist übrigens: LINQ-Abfragen beginnen niemals mit Select, ja, sie müssen noch nicht einmal ein Select aufweisen. DELETE, INSERT und UPDATE gibt es im Übrigen auch nicht – auch nicht wie im übernächsten Kapitel zu lesen, in *LINQ to SQL*.

Mit dem Wissen des vorherigen Kapitels schauen wir uns jetzt mal die folgenden beiden Abfragen an und überlegen uns, wie sie sich im Ergebnis unterscheiden.

BEGLEITDATEIEN Im Verzeichnis

...\\VB 2008 Entwicklerbuch\\F – LINQ\\Kapitel 32\\LinqToObjectSamples

finden Sie eine Konsolen-Beispielanwendung, die mit Business-Objekten weitere Möglichkeiten von *LINQ to Objects* erklären soll. Das Programm entspricht wieder dem aus dem letzten Kapitel: Es generiert per Zufallsgenerator zwei Auflistungen mit Business-Objekten (eine Kunden-, eine Artikelbestellaufstellung), die logisch zueinander in Relation stehen. Das vorherige 31. Kapitel erklärt mehr zum Aufbau des Beispiels.

OK. Wir schauen uns zunächst das folgende Listing des Beispielprogramms an, und das schaut so aus:

```
Sub LINQAbfragenAufbau()
    Console.WriteLine("Ergebnisliste 1:")
    Console.WriteLine("-----")
    Dim ergebnisListe = From adrElement In adrListe _
```

³ Quelle: Lisa Feigenbaum in einer E-Mail vom 01.02.2008. Lisa Feigenbaum ist VB IDE Program Manager im Visual Basic-Team bei Microsoft. Unter dem IntelliLink **F3202**, der Team Blog-Seite, erreichen Sie Lisa Feigenbaum und das Visual Basic Team (Stand 29.10.2008). Die Original-E-Mail lesen Sie unter www.activedevelop.de in der Bücher-Sektion.

```
Order By adrElement.Nachname, adrElement.Vorname Descending _
Select KontaktId = adrElement.ID, _
adrElement.Nachname, _
adrElement.Vorname, _
PlzOrtKombi = adrElement.PLZ & " " & adrElement.Ort _
Where KontaktId > 50 And KontaktId < 100 And _
Nachname Like "L*"

For Each ergebnisItem In ergebnisListe
With ergebnisItem
    Console.WriteLine(.KontaktId & ": " &
                      .Nachname & ", " & .Vorname &
                      " - " & .PlzOrtKombi)
End With
Next

Console.WriteLine()
Console.WriteLine("Ergebnisliste 2:")
Console.WriteLine("-----")

Dim ergebnisListe2 = From adrElement In adrListe _
Where adrElement.ID > 50 And adrElement.ID < 100 And _
adrElement.Nachname Like "L*" _
Select KontaktId = adrElement.ID, _
adrElement.Nachname, _
adrElement.Vorname, _
PlzOrtKombi = adrElement.PLZ & " " & adrElement.Ort _
Order By Nachname, Vorname

For Each ergebnisItem In ergebnisListe2
With ergebnisItem
    Console.WriteLine(.KontaktId & ": " &
                      .Nachname & ", " & .Vorname &
                      " - " & .PlzOrtKombi)
End With
Next
End Sub
```

Sie sehen, dass sich die Abfragen syntaktisch und damit auf den ersten Blick schon einmal gar nicht gleichen. Und dennoch, wenn Sie das Beispiel starten, so sehen Sie ein Ergebnis etwa wie im folgenden Bildschirmauszug...

Ergebnisliste 1:

```
-----
81: Langenbach, Barbara - 78657 Wiesbaden
76: Langenbach, Klaus - 61745 Dortmund
54: Lehnert, Katrin - 82730 Wiesbaden
96: Löffelmann, Barbara - 63122 Rheda
68: Löffelmann, Gabriele - 54172 Bad Waldliesborn
```

Ergebnisliste 2:

```
-----
81: Langenbach, Barbara - 78657 Wiesbaden
```

```
76: Langenbach, Klaus - 61745 Dortmund
54: Lehnert, Katrin - 82730 Wiesbaden
96: Löffelmann, Barbara - 63122 Rheda
68: Löffelmann, Gabriele - 54172 Bad Waldliesborn
```

... und obwohl diese Liste laut Codelisting mal definitiv auf zwei verschiedenen Ereignisauflistungen beruht, die aus zwei verschiedenen Abfragen entstanden sind, ist das Ergebnis dennoch augenscheinlich dasselbe. Zufall? Keinesfalls – im Gegenteil, pure Absicht.

Denn dieses Beispiel demonstriert sehr schön, wie LINQ-Abfragen im Code funktionieren und wie sehr sie im Grunde genommen nichts mit den klassischen SQL-Datenbankabfragen zu tun haben. Gut, zugegeben, Ähnlichkeiten sind vorhanden, aber das war's auch schon.

Nun lassen Sie uns mal beide Abfragen auseinandernehmen und dabei schauen, wieso so unterschiedlich Formulierte dennoch zum gleichen Ergebnis führen kann. Los geht's:

- Die erste Abfrage startet, wie jede LINQ-Abfrage, mit der `From`-Klausel, und diese definiert `adrElement` als *Bereichsvariable* für alle kommenden Parameter bis, so vorhanden, zum ersten `Select`. Die Bereichsvariable ist also bis zum ersten `Select`-Befehl die Variable, mit der quasi intern durch die komplette Auflistung hindurchiteriert wird, und an der damit auch die Eigenschaften bzw. öffentlichen Felder der Auflistungselemente »hängen«.
- Es folgt hier im Beispiel (aber natürlich nicht notwendigerweise) die Sortieranweisung `Order By`, der die Felder bzw. Eigenschaften, nach denen die Elemente der Auflistung sortiert werden sollen, als Argumente über die Bereichsvariable übergeben werden: das sind in diesem Beispiel `adrElement.Nachname` und `adrElement.Vorname`. Die Bereichsvariable dient also dazu, auf die Eigenschaften zuzugreifen, über die durch die `Order By`-Klausel festgelegt werden kann, nach welchen Feldern sortiert wird.

TIPP Wenn nichts anderes gesagt wird, sortiert `Order By` *aufsteigend*. Durch die Angabe des Schlüsselwortes `Descending` können Sie den Sortierausdruck ändern, sodass die Elementauflistung *absteigend* sortiert wird. Wollte man – um beim Beispiel zu bleiben – die erste Liste nach Namen aufsteigend und nach Vornamen absteigend sortieren, hieße der LINQ-Abfrageausdruck folgendermaßen:

```
Dim ergebnisListe = From adrElement In adrListe _
    Order By adrElement.Nachname, adrElement.Vorname Descending _
    Select KontaktId = adrElement.ID, _
        adrElement.Nachname, _
        adrElement.Vorname, _
        PlzOrtKombi = adrElement.PLZ & " " & adrElement.Ort _
    Where KontaktId > 50 And KontaktId < 100 And _
        Nachname Like "L*"
```

-
- Jetzt folgt ein `Select`, und Achtung: Wir haben ja im letzten Kapitel erfahren, dass `Select` dazu dient, ein Element einer Auflistung durch ein anderes zu ersetzen, das in einer anderen Auflistung »landet«. Und dieses zu ersetzende Element wird eine Instanz einer neuen anonymen Klasse mit exakt den Feldern sein, die Sie hinter dem `Select` bestimmen.

TIPP Spätestens hier sehen Sie, dass ein Select für eine Abfrage, die sich auf nur eine Tabelle bezieht, und die keine Gruppierungen durchführt, eigentlich ein reiner Zeitkiller ist, denn Sie können natürlich mit den Ausgangselementen weiterarbeiten, die *ohnehin* schon da sind und in der späteren Ergebnisliste *schlimmstenfalls* ein paar redundante Informationen aufweisen. Anders als bei SQL-Abfragen benötigen Sie das Select-Schlüsselwort bei LINQ *nicht*, um Abfragen für Filterungen, Selektierungen oder Gruppierungen einzuleiten, sondern *nur*, um eine *neue Auflistung* mit Elementen zu erstellen, die aus einem neuen, anonymen Typ bestehen, der nur die Eigenschaften offenlegt, die Sie als Feldnamen angeben. Beherzigen Sie deswegen auch das, was der nächste Abschnitt über Select zu berichten weiß!

- Nachdem wir nun das Select hinter uns gelassen und der Abfrage-Engine damit mitgeteilt haben, dass wir nur noch mit KontaktId, Nachname, Vorname und einem PlatzOrtKombi-Feld weitermachen wollen, können sich weitere LINQ-Abfrageelemente auch nur noch auf diese, durch Select neu eingeführte anonyme Klasse beziehen – dementsprechend würde hinter der Where-Klausel ein adrElement.Id als Argument nicht mehr funktionieren; Select hat adrElement der Auflistung schließlich durch Instanzen einer anonymen Klasse ersetzt, und die haben nur noch die Eigenschaft KontaktID. Where dient jetzt dazu, die neue Auflistung mit den Elementen der anonymen Klasse zu filtern: nur Elemente mit KontaktID > 50 und KontaktID < 100 sind mit von der Partie, so wie die Elemente, deren Nachname mit »L« beginnt.

Damit ist die erste Abfrage durch. Und jetzt schauen wir uns die zweite an.

- Hier geht es nach der obligatorischen Festlegung von Auflistung und Bereichsvariable, die als Element-Iterator dienen soll, zunächst los mit der Where-Klausel, die die Elemente auf jene IDs einschränkt, die größer als 50 und kleiner als 100 sind. Die Einschränkung wird dann noch weitergeführt, nämlich auf Nachnamen, die mit dem Buchstaben »L« beginnen. Hier kann Where direkt auf die Eigenschaften zurückgreifen, die über die Bereichsvariable erreichbar sind, denn sie entsprechen einem Element der Ausgangsaufstellung.
- Das geht solange, bis es auch hier wieder ein Select gibt, was dem ein Ende setzt. Auch hier wird wieder ein Ersetzungsvorgang durchgeführt – unnötigerweise und nur zu Demonstrationszwecken –, der die Elemente der vorhandenen Auflistung durch neue einer anonymen Klasse ersetzt, die wiederum die angegebenen Eigenschaften haben.
- Das anschließende Where schränkt diese Auflistung wieder ein, und zwar prinzipiell in gleicher Weise wie im ersten Beispiel – lediglich der Zeitpunkt, wann die Elemente oder welche Elemente eingeschränkt werden, ist ein anderer.

Eines ist in diesem Zusammenhang sehr wichtig zu erwähnen:

WICHTIG LINQ-Abfragen werden immer verzögert ausgeführt und niemals direkt. Das bedeutet: Nicht die eigentliche Abfrage ist ausschlaggebend für das »Anstoßen« der Verarbeitung, sondern erst das erste Zugreifen auf die Ergebnisliste löst das Ausführen der eigentlichen Abfrage aus. Mehr dazu erfahren Sie im Abschnitt »Kombinieren und verzögertes Ausführen von LINQ-Abfragen«.

Die Ausführung von Select ergibt bei der Anwendung von nur einer Auflistung nur eingeschränkt Sinn. Das folgende Beispiel zeigt, wieso Select Rechenleistung kosten kann ohne einen nachvollziehbaren Nutzen dabei zu bringen.

Dieses Beispiel arbeitet mit einem Hochgeschwindigkeitszeitmesser, um die Laufdauer bestimmter Auswertungen zu messen. Dabei wird eine Auflistung mit 2.000.000 Kundenelementen einmal mit und einmal ohne Select-Abfrage ausgeführt.

HINWEIS Achten Sie beim Einrichten dieses Beispiels darauf, die ersten Zeilen des Beispielprojektes folgendermaßen abzuändern:

```
Module LinqDemo
```

```
Private adrListe As List(Of Kontakt) = Kontakt.Zufallskontakte(2000000)
Private artListe As List(Of Artikel) '= Artikel.Zufallsartikel(adrListe)
```

Achten Sie darauf, diese Zeilen für die anderen Beispiele wieder zurückzubauen.

Die eigentliche Methode, die das Beispiel enthält, sieht wie folgt aus:

```
Sub SelectSpeedCompare()
    Dim hsp As New HighSpeedTimeGauge
    Console.WriteLine("Starte Test")
    hsp.Start()
    Dim ergebnisListe = From adrElement In adrListe
        Order By adrElement.Nachname, adrElement.Vorname
        Select KontaktId = adrElement.ID,
            adrElement.Nachname,
            adrElement.Vorname,
            PLzOrtKombi = adrElement.PLZ & " " & adrElement.Ort

    Dim ersteAnzahl = ergebnisListe.Count
    hsp.Stop()
    Dim dauer1 = hsp.DurationInMilliSeconds

    hsp.Reset()
    hsp.Start()
    Dim ergebnisListe2 = From adrElement In adrListe
        Order By adrElement.Nachname, adrElement.Vorname

    Dim zweiteAnzahl = ergebnisListe2.Count
    hsp.Stop()
    Dim dauer2 = hsp.DurationInMilliSeconds

    Console.WriteLine("Abfragedauer mit Select: " & dauer1 & " für " & ersteAnzahl & " Elemente.")
    Console.WriteLine("Abfragedauer ohne Select: " & dauer2 & " für " & zweiteAnzahl & " Elemente.")
End Sub
```

Das Ergebnis dieses Tests offenbart den Unterschied. Bei 2.000.000 Elementen kommt es ...

```
Starte Test
Abfragedauer mit Select: 16939 für 2000000 Elemente.
Abfragedauer ohne Select: 16061 für 2000000 Elemente.
```

... bis zum Unterschied von einer Sekunde. Natürlich ist das kein Wert, an dem man sich für die Praxis in irgendeiner Form orientieren sollte; das Beispiel ist zudem natürlich auch praxisfern, denn wer zwei Millionen Datensätze im Speicher hält, hat entweder eine Datenbankengine programmiert oder das Konzept von Datenbanken und Datenbankabfragen nicht verstanden. Aber es geht nur um Tendenzen und den Tipp, Prozessorleistung nicht an Stellen unnötig zu verheizen, an denen es nicht nötig ist.

Übrigens, und besser könnte diese Überleitung zum nächsten Abschnitt nicht sein, wenn Sie sich das vorherige Listing anschauen, sollte Ihnen etwas auffallen, das Sie erstaunen sollte. Sie sehen es nicht? OK – dann

überlegen Sie sich mal, ob es Zufall ist, dass die Count-Eigenschaft noch innerhalb der Zeitmessung abgefragt wird. Denn: Das ist nicht nur bewusst so gemacht, anderenfalls würden Sie komplett andere Ergebnisse bekommen, denn wenn Sie das Listing folgendermaßen abändern ...

```
Sub SelectSpeedCompare()  
    .  
    .  
    hsp.Stop()  
    Dim dauer1 = hsp.DurationInMilliseconds  
    Dim ersteAnzahl = ergebnisListe.Count  
  
    hsp.Reset()  
    hsp.Start()  
    Dim ergebnisListe2 = From adrElement In adrListe _  
        Order By adrElement.Nachname, adrElement.Vorname  
  
    hsp.Stop()  
    Dim dauer2 = hsp.DurationInMilliseconds  
    Dim zweiteAnzahl = ergebnisListe2.Count  
  
    .  
    .  
End Sub
```

... erhalten Sie auf einmal das folgende Ergebnis!

```
Starte Test  
Abfragedauer mit Select: 1 für 2000000 Elemente.  
Abfragedauer ohne Select: 0 für 2000000 Elemente.
```

Ein Fehler? Nein – das ist genau das Ergebnis, was bei der Ausführung herauskommen muss!

Kombinieren und verzögertes Ausführen von LINQ-Abfragen

Eines vorweg: LINQ-Abfragen werden immer verzögert ausgeführt, die Überschrift könnte also insofern in die Irre führen, als dass sie impliziert, der Entwickler, der sich LINQ-Abfragen bedient, hätte eine Wahl. Er hat sie nämlich nicht.

Wann immer Sie eine Abfrage definieren, und sei sie wie die folgende ...

```
Dim adrListeGruppiert = From adrElement In adrListe _  
    Join artElement In artListe  
    On adrElement.ID Equals artElement.IDGekauftVon  
    Select adrElement.ID, adrElement.Nachname, _  
        adrElement.Vorname, adrElement.PLZ, _  
        artElement.ArtikelNummer, artElement.ArtikelName, _  
        artElement.Anzahl, artElement.Einzelpreis, _  
        Postenpreis = artElement.Anzahl * artElement.Einzelpreis _  
    Order By Nachname, ArtikelNummer  
    Where (PLZ > "0" And PLZ < "50000") _
```

```

Group ArtikelNummer, ArtikelName,
      Anzahl, Einzelpreis, Postenpreis _
By ID, Nachname, Vorname
Into Artikelliste = Group, AnzahlArtikel = Count(), _
      Gesamtpreis = Sum(Postenpreis)

```

... noch so lang; die Abfragen selbst werden nicht zum Zeitpunkt ihres »Darüberfahrens« ausgeführt. Im Gegenteil: In adrListeGruppiert wird – um bei diesem Beispiel zu bleiben – im Prinzip nur eine Art *Ausführungsplan* generiert, also eine Liste der Methoden, die nacheinander ausgeführt werden, sobald ein Element aus der (noch zu generierenden!) Ergebnisliste abgerufen wird, oder Eigenschaften bzw. Funktionen der Ergebnisliste abgerufen werden, die in unmittelbarem Zusammenhang mit der Ergebnisliste selbst stehen – wie beispielsweise die Count-Eigenschaft.

Und es wird noch besser: Verschiedene LINQ-Abfragen lassen sich so nacheinander »aufreihen«, wie das folgende Beispiel eindrucksvoll zeigt.

HINWEIS Achten Sie beim Einrichten dieses Beispiels darauf, die ersten Zeilen des Beispielprojektes folgendermaßen abzuändern:

Module LinqDemo

```

Private adrListe As List(Of Kontakt) = Kontakt.Zufallskontakte(500000)
Private artListe As List(Of Artikel) '= Artikel.Zufallsartikel(adrListe)

```

Achten Sie auch darauf, diese Zeilen für die anderen Beispiele wieder zurückzubauen.

```

Sub SerialLinqsCompare()

    Dim hsp As New HighSpeedTimeGauge
    Console.WriteLine("Starte Test")
    hsp.Start()
    Dim ergebnisListe = From adrElement In adrListe
                         Order By adrElement.Nachname, adrElement.Vorname

    Dim ergebnisListe2 = From adrElement In ergebnisListe
                         Where adrElement.Nachname Like "L*"

    ergebnisListe2 = From adrElement In ergebnisListe2
                     Where adrElement.ID > 50 And adrElement.ID < 200

    Dim ersteAnzahl = ergebnisListe2.Count
    hsp.Stop()
    Dim dauer1 = hsp.DurationInMilliseconds

    hsp.Reset()
    hsp.Start()
    Dim ergebnisListe3 = From adrElement In adrListe
                         Order By adrElement.Nachname, adrElement.Vorname
                         Where adrElement.Nachname Like "L*" And
                               adrElement.ID > 50 And adrElement.ID < 200
    Dim zweiteAnzahl = ergebnisListe3.Count
    hsp.Stop()

```

```
Dim dauer2 = hsp.DurationInMilliseconds

Console.WriteLine("Abfragedauer serielle Abfrage: " & dauer1 & " für " & ersteAnzahl &
    " Ergebniselemente.")
Console.WriteLine("Abfragedauer kombinierte Abfrage: " & dauer2 & " für " & zweiteAnzahl &
    " Ergebniselemente.")
End Sub
```

Der zweite, in Fettschrift markierte Block entspricht im Grunde genommen dem ersten, nur dass hier Abfragen hintereinandergeschaltet, aber eben nicht ausgeführt werden. Die eigentliche Ausführung findet statt, wenn auf die zu entstehende Elementauflistung zugegriffen wird – im Beispiel also die Count-Eigenschaft der »Auflistung« abgerufen wird, die die Anzahl der Elemente nur dann zurückgeben kann, wenn es eine Anzahl an Elementen gibt.

Dass sich die beiden Ausführungspläne nicht nennenswert unterscheiden, zeigt das folgende Ergebnis ...

```
Starte Test
Abfragedauer serielle Abfrage: 3531 für 17 Ergebniselemente.
Abfragedauer kombinierte Abfrage: 3561 für 17 Ergebniselemente.
```

... das mit 30ms Unterschied bei 500.000 Elementen wirklich nicht nennenswert ist.

WICHTIG Die Ausführungspläne, die Sie durch die Abfragen erstellen, werden jedes Mal ausgeführt, wenn Sie auf eine der Ergebnislisten zugreifen. Wiederholen Sie die letzte fettgeschriebene Zeile im obigen Listing ...

```
.
.
.
Dim zweiteAnzahl = ergebnisListe3.Count
zweiteAnzahl = ergebnisListe3.Count
hsp.Stop()
Dim dauer2 = hsp.DurationInMilliseconds
.
.
```

... ergibt sich das folgende Ergebnis:

```
Starte Test
Abfragedauer serielle Abfrage: 3675 für 12 Ergebniselemente.
Abfragedauer kombinierte Abfrage: 7104 für 12 Ergebniselemente.
```

Faustregeln für das Erstellen von LINQ-Abfragen

Die Faustregeln für das Erstellen von Abfragen lauten:

- LINQ-Abfragen bestehen nur aus Ausführungsplänen – die eigentlichen Abfragen werden durch ihre Definition nicht ausgelöst!
- Wenn das Ergebnis einer Abfrage als Ergebnisliste Gegenstand einer weiteren Abfrage wird, kommt der erste Abfrageplan auch nicht zur Auslösung; beide Abfragepläne werden miteinander kombiniert.

- Erst der Zugriff auf ein Element der Ergebnisaufstellung (über For/Each oder den Indexer) löst die Abfrage und damit das Erstellen der Ergebnisliste aus.
- Ein erneuter Zugriff auf eine elementeabhängige Eigenschaft oder auf die Elemente selbst löst – und das ist ganz wichtig – abermals das Erstellen der Ergebnisliste aus. Das gilt auch für Abfragen, die durch mehrere Abfragen kaskadiert wurden.

Kaskadierte Abfragen

Das Beispiel, was ich Ihnen im letzten Listing vorgestellt habe, hat nämlich noch zwei weitere, auskommene Zeilen, die die Kaskadierungsfähigkeit (das Hintereinanderschalten) von Abfragen eindrucksvoll demonstrieren. Wenn Sie das Ergebnis dieser beiden zusätzlichen Zeilen des Listings ...

```
adrListe.Add(New Kontakt(51, "Löffelmann", "Klaus", "Wiedenbrücker Straße 47", "59555", "Lippstadt"))
Console.WriteLine("Elemente der seriellen Abfrage: " & ergebnisListe2.Count)
```

... verstanden haben, dann haben Sie auch das Prinzip von LINQ verstanden!

```
Starte Test
Abfragedauer serielle Abfrage: 3515 für 7 Ergebniselemente.
Abfragedauer kombinierte Abfrage: 7101 für 7 Ergebniselemente.
Elemente der seriellen Abfrage: 8
```

Hier wird der Originalauflistung ein weiteres Element hinzugefügt, das exakt der Kriterienliste der kaskadierten Abfrage entspricht. Durch das Abfragen der Count-Eigenschaft von ergebnisListe2 wird die *komplette* Abfragekaskade ausgelöst, denn von vorher 7 Elementen befinden sich anschließend 8 Elemente in der Ergebnismenge, was nicht der Fall wäre, würde LINQ nur die letzte Ergebnismenge, nämlich ergebnisListe2 selbst auswerten, denn dieser haben wir das Element nicht hinzugefügt.

Gezieltes Auslösen von Abfragen mit ToArray oderToList

Nun haben wir gerade erfahren, dass Abfragen bei jedem Zugriff auf die Ergebnisliste mit For/Each oder direkt über eine Elementeigenschaft bzw. den Index zu einer neuen Ausführung des Abfrageplans führen. Das kann, wie im Beispiel des vorherigen Abschnittes, durchaus wünschenswert sein; in vielen Fällen können sich daraus aber echte Performance-Probleme entwickeln.

Aus diesem Grund implementierten die Entwickler von LINQ spezielle Methoden, die eine auf IEnumerable basierende Ergebnisliste wieder in eine »echte« Auflistung bzw. in ein »echtes« Array umwandeln können.

Am häufigsten wird dabei sicherlich die Methode `ToList` zur Anwendung kommen, die das Ergebnis einer wie auch immer gearteten LINQ-Abfrage in eine generische `List(Of t)` umwandelt – und dabei ist es völlig gleich, ob es sich um eine *LINQ to Objects*, *LINQ to SQL*, *LINQ to XML* oder eine *LINQ to sonst was*-Abfrage handelt. Das Ausführen von `ToList` auf eine solche Ergebnisliste hat zwei Dinge zur Folge:

- Die LINQ-Abfrage wird ausgeführt.
- Eine `List(Of {Ausgangstyp|Select-Anonymer-Typ})` wird zurückgeliefert.

Die Elemente, die anschließend zurückkommen, sind völlig ungebunden – Sie können sie anschließend sofort indizieren, mit For/Each durchiterieren, ihre Count-Eigenschaft abfragen wie Sie wollen – mit der ursprünglichen LINQ-Abfrage haben sie nichts mehr zu tun.

ToList funktioniert dabei denkbar einfach:

```
Dim reineGenerischeListe = ergebnisListe.ToList
```

Und `ToList` ist auch nicht die einzige Methode, mit der Sie eine Ergebnisliste von ihrer Ursprungsabfrage trennen können. Das können Sie – in Form von Ergebnislisten unterschiedlichen Typs – auch mit den folgenden Methoden machen. `t` ist dabei immer der Typ eines Elements der Ausgangsaufstellung oder ein anonymer Typ, der in der Abfrage beispielsweise durch die `Select`-Klausel entstanden ist.

- `ToList`: Überführt die Abfrageergebnisliste in eine generische `List(Of t)`.
- `ToArray`: Überführt die Abfrageergebnisliste in ein Array vom Typ `t`.
- `ToDictionary`: Überführt die Abfrageliste in eine generische Wörterbuchaufstellung vom Typ `Dictionary(Of t.key, t)`. `t.key` muss dabei durch die Angabe eines Lambda-Ausdrucks festgelegt werden, also etwa

```
Dim reineGenerischeListe = ergebnisListe.ToDictionary(Function(einKontakt) einKontakt.ID)
```

um beim Beispiel zu bleiben, und die ID zum Nachschlageschlüssel zu machen.

- `ToLookup`: Überführt die Abfrageliste in eine generische `Lookup`-Auflistung vom Typ `Lookup(Of t.key, t)`. `t.key` muss dabei durch die Angabe eines Lambda-Ausdrucks festgelegt werden, also etwa

```
Dim reineGenerischeListe = ergebnisListe.ToLookup(Function(einKontakt) einKontakt.ID)
```

HINWEIS Die `ToLookup`-Methode gibt ein generisches `Lookup`-Element zurück, ein $1:n$ -Wörterbuch, das Auflistungen von Werten Schlüssel zuordnet. Ein `Lookup` unterscheidet sich von `Dictionary`, das eine $1:1$ -Zuordnung von Schlüsseln zu einzelnen Werten ausführt.

Verbinden mehrerer Auflistungen zu einer neuen

LINQ kennt mehrere Möglichkeiten, Auflistungen miteinander zu verbinden. Natürlich ist es nicht unbedingt sinnig, das willkürlich zu tun: Die Auflistungen müssen schon in irgendeiner Form miteinander in logischer Relation stehen – dann aber kann man sich durch geschicktes Formulieren eine Menge an Zeit sparen. Wie eine solche Relation ausschauen kann, in der zwei Auflistungen zueinander stehen, demonstriert Ihnen der Abschnitt »Wie „geht« LINQ prinzipiell« des vorherigen Kapitels – und diese Relation soll auch noch mal Gegenstand der Beispiele dieses Abschnittes sein:

Relation zwischen Artikel und Käufer-Tabellen					
ID	Nachname	Vorname	Straße	PLZ	Ort
1	Heckhuis	Jürgen	Wiedenbrücker Str. 47	59555	Lippstadt
2	Wördehoff	Angela	Erwitter Str. 33	01234	Dresden
3	Dröge	Ruprecht	Douglas-Adams-Str. 42	55544	Ratingen
4	Dröge	Ada	Douglas-Adams-Str. 42	55544	Ratingen
5	Halek	Gaby „Doc“	Krankenhausplatz 1	59494	Soest
6	IDGekauftVon	ArtikelNummer	ArtikelName	Kategorie	Einzelpreis
	1	9-445	Die Tore der Welt	Bücher, Belletristik	19,90
	3	3-537	Visual Basic 2005 - Das Entwicklerbuch	Bücher, EDV	59,00
	3	3-123	SQL Server 2000 - So gelingt der Einstieg	Bücher, EDV	19,90
	5	5-312	SQL Server 2008 - Das Profi-Buch	Bücher, EDV	39,90

Abbildung 32.1 Das LINQ-Beispielprogramm legt zwei Tabellen an, die nur logisch über die Spalten-ID miteinander verknüpft sind

Implizite Verknüpfung von Auflistungen

Die einfachste Möglichkeit, zwei Auflistungen zu gruppieren, zeigt die folgende Abbildung.

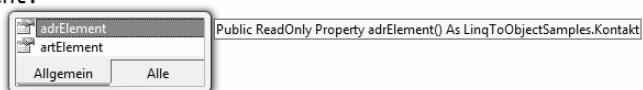
```
Sub Auflistungsvereinigung()
    Dim ergebnisliste = From adrElement In adrListe, artElement In artListe
    For Each element In ergebnisliste
        element.
        
        with element
            Console.WriteLine(.adrElement.ID & ":" & .adrElement.Nachname & _
                ", " & .adrElement.Vorname & ":" & _
                .artElement.IDGekauftVon & ":" & .artElement.ArtikelName)
        End With
    Next
End Sub
```

Abbildung 32.2 Schon mit der From-Klausel können Sie zwei Auflistungen per LINQ miteinander kombinieren

From kombiniert zwei Auflistungen miteinander. Im Ergebnis erhalten Sie eine neue Auflistung, bei der jedes Element der ersten mit jedem Element der zweiten Auflistung kombiniert wurde. Die Ausgabe dieser Auflistung, die über zwei Eigenschaften verfügt, die jeweils Zugriff auf die Originalelemente gestatten, zaubert dann folgendes Ergebnis auf den Bildschirm (aus Platzgründen gekürzt):

```
6: Clarke, Christian: 1: Visual Basic 2008 - Das Entwicklerbuch
6: Clarke, Christian: 1: Das Herz der Hölle
6: Clarke, Christian: 2: Visual Basic 2005 - Das Entwicklerbuch
6: Clarke, Christian: 2: Das Vermächtnis der Tempelritter
6: Clarke, Christian: 3: Das Leben des Brian
```

```
6: Clarke, Christian: 3: Die Tore der Welt  
.  
. .  
7: Ademmer, Lothar: 1: Visual Basic 2008 - Das Entwicklerbuch  
7: Ademmer, Lothar: 1: Das Herz der Hölle  
7: Ademmer, Lothar: 2: Visual Basic 2005 - Das Entwicklerbuch  
7: Ademmer, Lothar: 2: Das Vermächtnis der Tempelritter  
7: Ademmer, Lothar: 3: Das Leben des Brian  
7: Ademmer, Lothar: 3: Die Tore der Welt
```

An der Ergebnisliste können Sie erkennen, wie redundant und nicht informativ diese Liste ist, denn jeder Artikel der Artikelliste wird einfach und stumpf mit jedem Kontakt kombiniert.

Wichtiger wäre es, Zuordnungen ausdrücklich bestimmen zu können, um zu sagen, dass ein Artikel mit einer bestimmten ID auch nur dem logisch dazugehörigen Kontakt zugeordnet werden soll. Und das funktioniert folgendermaßen:

```
Sub Auflistungsvereinigung()  
    Dim ergebnisliste = From adrElement In adrListe, artElement In artListe _  
        Where adrElement.ID = artElement.IDGekauftVon  
  
    For Each element In ergebnisliste  
        With element  
            Console.WriteLine(.adrElement.ID & ":" & .adrElement.Nachname &  
                ", " & .adrElement.Vorname & ":" &  
                .artElement.IDGekauftVon & ":" & .artElement.ArtikelName)  
        End With  
    Next  
End Sub
```

In dieser Version werden durch die `Where`-Klausel nur die Elemente in die Auflistung übernommen, die die gleiche ID haben wie das korrelierende Element der anderen Auflistung (`IDGekauftVon`). Das Ergebnis, was dann zu sehen ist, macht natürlich viel mehr Sinn, da es eine Aussagekraft hat (nämlich: welcher Kunde hat welche Artikel gekauft):

```
7: Sonntag, Uwe: 7: Visual Basic 2008 - Das Entwicklerbuch  
7: Sonntag, Uwe: 7: The Da Vinci Code  
7: Sonntag, Uwe: 7: O.C., California - Die komplette zweite Staffel (7 DVDs)  
7: Sonntag, Uwe: 7: Desperate Housewives - Staffel 2, Erster Teil  
7: Sonntag, Uwe: 7: Die Rache der Zwerge  
8: Vüllers, Momo: 8: Programmieren mit dem .NET Compact Framework  
9: Tinoco, Daja: 9: Das Herz der Hölle  
9: Tinoco, Daja: 9: Mitten ins Herz  
9: Tinoco, Daja: 9: The Da Vinci Code  
9: Tinoco, Daja: 9: Der Schwarm  
9: Tinoco, Daja: 9: Desperate Housewives - Staffel 2, Erster Teil  
9: Tinoco, Daja: 9: Harry Potter und die Heiligtümer des Todes  
9: Tinoco, Daja: 9: Der Teufel trägt Prada  
9: Tinoco, Daja: 9: O.C., California - Die komplette zweite Staffel (7 DVDs)  
10: Lehnert, Michaela: 10: Abgefahren - Mit Vollgas in die Liebe  
10: Lehnert, Michaela: 10: Das Herz der Hölle
```

HINWEIS Die Verknüpfung zweier oder mehrerer Auflistungen mit In als Bestandteil der From-Klausel einer Abfrage nennt man implizite Verknüpfung von Auflistungen, da dem Compiler nicht ausdrücklich mitgeteilt wird, welche Auflistung auf Grund welchen Elementes mit einer anderen verknüpft wird. Eine ausdrückliche oder explizite Verknüpfung stellen Sie mit Join her, das im nächsten Abschnitt beschrieben wird. Explizit oder implizit hier im Beispiel ist aber letzten Endes einerlei – das Ergebnis ist in beiden Fällen dasselbe.

Explizite Auflistungsverknüpfung mit Join

Im Gegensatz zu impliziten Auflistungsverknüpfungen, die Sie, wie im letzten Abschnitt beschrieben, mit In als Teil der From-Klausel bestimmen, erlaubt Ihnen die Join-Klausel so genannte explizite Auflistungsverknüpfungen festzulegen. Die generelle Syntax der Join-Klausel lautet:

```
Dim ergebnisliste = From bereichsVariable In ersterAuflistung _
    Join verknüpfungsElement In zweiterAuflistung _
        On bereichsVariable.JoinKey Equals verknüpfungsElement.ZweiterJoinKey
```

Join verknüpft die erste mit der zweiten Tabelle über einen bestimmten Schlüssel (JoinKey, ZweiterJoinKey), der beide Tabellen zueinander in Relation stellt. In unserem Beispiel wird für jede Bestellung in der Artikel-tabelle ein Schlüssel (ein *Key*, eine *ID*) definiert, der der Nummer der *ID* in der Kontakttabelle entspricht.

Im Vergleich zur impliziten Verknüpfung von Tabellen ändert sich im Ergebnis nichts; die Umsetzung des vorherigen Beispiels mit Join gestaltet sich folgendermaßen:

```
Sub JoinDemo()
    Dim ergebnisliste = From adrElement In adrListe _
        Join artElement In artListe _
            On adrElement.ID Equals artElement.IDGekauftVon

    For Each element In ergebnisliste
        With element
            Console.WriteLine(.adrElement.ID & ":" & .adrElement.Nachname & _
                ", " & .adrElement.Vorname & ":" & _
                .artElement.IDGekauftVon & ":" & .artElement.ArtikelName)
        End With
    Next
End Sub
```

Es gibt die Möglichkeit, mit der Klausel GroupJoin Verknüpfungen mehrerer Tabellen auf bestimmte Weise in Gruppen zusammenzufassen. Wie das funktioniert, erfahren Sie im Abschnitt »Group Join« ab Seite 897.

Gruppieren von Auflistungen

Die Klausel GroupBy erlaubt es, Dubletten von Elementen einer oder mehrerer Auflistungen in Gruppen zusammenzufassen. Sie möchten also beispielsweise eine Auflistung von Kontakten nach Nachnamen gruppieren, und dann in einer geschachtelten Schleife durch die Namen und innen durch alle den Namen zugeordneten Kontakte iterieren. Mit der GroupBy-Klausel können Sie genau das erreichen, wie das folgende Beispiel zeigt:

```
Sub GroupByDemo()
    Dim ergebnisliste = From adrElement In adrListe _
        Group By adrElement.Nachname Into Kontaktliste = Group _
        Order By Nachname

    For Each element In ergebnisliste
        With element
            Console.WriteLine(element.Nachname)
            For Each Kontakt In element.Kontaktliste
                With Kontakt
                    Console.WriteLine(.ID & ": " & .Nachname & ", " & .Vorname)
                End With
            Next
            Console.WriteLine()
        End With
    Next
    End Sub
```

Wörtlich ausformuliert hieße die Group By-Klausel dieses Beispiels: »Erstelle eine Liste aller Nachnamen (Group By adrElement.Nachname) und mache diese unter der Nachname-Eigenschaft in der Liste zugänglich.⁴ Fasse alle Elemente der Ausgangsliste in die jeweiligen Auflistungen zusammen, die dem Nachnamen zugehörig sind (Into ... = Group), und mache diese Auflistung über die Eigenschaft Kontaktliste verfügbar.«

HINWEIS

Falls Sie keine Aliasbenennung der Gruppe vornehmen (der Ausdruck würde dann einfach

```
Dim ergebnisliste = From adrElement In adrListe _
    Group By adrElement.Nachname Into Group _
    Order By Nachname
```

heißen), würde die Eigenschaft, mit der Sie die zugeordneten Elemente erreichen können, einfach Group heißen.

Wenn Sie das Beispiel starten, sehen Sie das gewünschte Ergebnis, etwa wie im folgenden Bildschirmauszug (aus Platzgründen gekürzt):

```
Weichert
19: Weichert, Anne
39: Weichert, Gabriele
47: Weichert, Uta
69: Weichert, Franz
97: Weichert, Hans

Weigel
37: Weigel, Lothar
40: Weigel, Rainer
43: Weigel, Lothar
```

⁴ Wenn nichts anderes gesagt wird, heißt das Feld bzw. die Eigenschaft, nach der Sie gruppieren, in der späteren Auflistung so wie das Ausgangsfeld. Wenn Sie das nicht wünschen, können Sie das durch das Vorsetzen eines neuen Namens etwa mit Group By Lastname = adrElement.Nachname Into Kontaktliste = Group – an Ihre Wünsche anpassen. Anders als im Beispiel wäre hier Lastname die Eigenschaft, mit der Sie später die Nachnamen abfragen könnten.

```

58: Weigel, Hans
60: Weigel, Anja

Westermann
11: Westermann, Margarete
21: Westermann, Alfred
28: Westermann, Alfred
41: Westermann, Alfred
77: Westermann, Guido
98: Westermann, José
100: Westermann, Michaela

Wördehoff
14: Wördehoff, Bernhard

```

Gruppieren von Listen aus mehreren Auflistungen

Group By eignet sich auch sehr gut dazu, mit Join kombinierte Listen zu gruppieren und auszuwerten. Stellen Sie sich vor, Sie möchten eine Liste mit Kontakten erstellen, mit der Sie über jeden Kontakt wieder auf eine Liste mit Artikeln zugreifen können, um auf diese Weise herauszufinden, welche Kunden welche Artikel gekauft hätten. Die entsprechende Abfrage und die anschließende Iteration durch die Ergebniselemente sähen dann folgendermaßen aus:

```

Sub GroupByJoinedCombined()
    Dim ergebnisliste = From adrElement In adrListe _
        Join artElement In artliste _
        On adrElement.ID Equals artElement.IDGekauftVon _
        Group artElement.IDGekauftVon, artElement.ArtikelNummer, _
            artElement.ArtikelName _
        By artElement.IDGekauftVon, adrElement.Nachname, adrElement.Vorname _
        Into Artikelliste = Group, AnzahlArtikel = Count() Order By Nachname

    For Each kontaktElement In ergebnisliste
        With kontaktElement
            Console.WriteLine(.IDGekauftVon & ":" & .Nachname & .Vorname)
            For Each Artikel In .Artikelliste
                With Artikel
                    Console.WriteLine(" " & .ArtikelNummer & ":" & .ArtikelName)
                End With
            Next
            Console.WriteLine()
        End With
    Next
End Sub

```

Hier sehen Sie eine Zusammenfassung dessen, was wir in den letzten Abschnitten kennen gelernt haben. Die Abfrage beginnt mit einem Join und vereint Artikel und Kundenliste zu einer flachen Auflistung, die sowohl die Namen als auch die Artikel für jeden Namen beinhaltet. Und dann wird gruppiert: Anders als beim ersten Gruppierungsbeispiel, in dem alle Elemente in der untergeordneten Liste einbezogen werden, geben wir hier zwischen Group und By die Felder an, die in der inneren Auflistung als Eigenschaften erscheinen

sollen, wir ändern also, ähnlich dem Select-Befehl, das Schema der inneren Auflistung. Die Elemente, die wir anschließend hinter dem By angeben, sind die, nach denen gruppiert wird, und die damit auch in der äußeren Auflistung verfügbar sind. Das Ergebnis entspricht dann unseren Erwartungen:

```
21: Wördehoff, Theo
    4-444: The Da Vinci Code
    2-424: 24 - Season 6 [UK Import - Damn It!]
    2-134: Abgefahren - Mit Vollgas in die Liebe
    3-534: Mitten ins Herz
    3-333: Der Schwarm
    3-537: Visual Basic 2005 - Das Entwicklerbuch
    4-444: Harry Potter und die Heiligtümer des Todes
    5-554: O.C., California - Die komplette zweite Staffel (7 DVDs)
    2-134: Abgefahren - Mit Vollgas in die Liebe
    7-321: Das Herz der Hölle

75: Wördehoff, Katrin
    2-134: Abgefahren - Mit Vollgas in die Liebe
    2-134: Abgefahren - Mit Vollgas in die Liebe
    2-134: Abgefahren - Mit Vollgas in die Liebe
    3-123: Das Vermächtnis der Tempelritter
    2-134: Abgefahren - Mit Vollgas in die Liebe
    1-234: Das Leben des Brian
    3-543: Microsoft Visual C# 2005 - Das Entwicklerbuch
    3-543: Microsoft Visual C# 2005 - Das Entwicklerbuch
    9-423: Desperate Housewives - Staffel 2, Erster Teil
    7-321: Das Herz der Hölle
    5-506: Visual Basic 2008 - Das Entwicklerbuch
    5-513: Microsoft SQL Server 2008 - Einführung in Konfiguration, Administration, Programmierung
    9-646: Was diese Frau so alles treibt
    5-506: Visual Basic 2008 - Das Entwicklerbuch
    2-321: Die Rache der Zwerge
    9-445: Die Tore der Welt
    4-444: Harry Potter und die Heiligtümer des Todes

77: Wördehoff, Theo
    7-321: Das Herz der Hölle
    1-234: Das Leben des Brian
    9-009: Die Wächter
    3-123: Das Vermächtnis der Tempelritter
    5-518: Visual Basic 2008 - Neue Technologien - Crashkurs
```

Group Join

Exakt das gleiche Ergebnis bekommen Sie, allerdings mit sehr viel weniger Aufwand, wenn Sie sich der Group Join-Klausel bedienen, die Join und Group By miteinander kombiniert – das folgende Beispiel zeigt, wie's geht:

```
Sub GroupJoin()
    Dim ergebnisliste = From adrElement In adrListe
        Group Join artElement In artListe
            On adrElement.ID Equals artElement.IDGekauftVon Into Artikelliste = Group
```

```

For Each kontaktElement In ergebnisliste
    With kontaktElement
        Console.WriteLine(.adrElement.ID & ":" &
            .adrElement.Nachname & ", " -
            & .adrElement.Vorname)
        For Each Artikel In .Artikelliste
            With Artikel
                Console.WriteLine(" " & .ArtikelNummer & ":" & .ArtikelName)
            End With
        Next
        Console.WriteLine()
    End With
Next
End Sub

```

Aggregatfunktionen

Aggregatfunktionen sind Funktionen, die ein bestimmtes Feld einer Auflistung aggregieren – zum Beispiel aufsummieren, den Durchschnitt berechnen, das Maximum oder das Minimum ermitteln oder einfach nur zählen, wie viele Elemente in einer Auflistung vorhanden sind. Anders als bei »normalen« Abfragen werden beim Aggregieren also die Elemente der Auflistung einer Aggregatfunktion alle nacheinander übergeben, und diese Funktion arbeitet bzw. rechnet dann mit jedem einzelnen dieser Elemente, betrachtet sie aber als Menge.

HINWEIS LINQ-Abfragen fangen grundsätzlich mit der FROM-Klausel an – reines Aggregieren von Auflistungen sind allerdings die einzige Ausnahme. Wenn Sie eine reine Aggregierung auf eine Auflistung ausführen wollen, leiten Sie die LINQ-Abfrage nicht mit From sondern mit der Klausel Aggregate ein; die folgenden Beispiele werden es gleich demonstrieren.

Lassen Sie uns ganz einfach beginnen. Die erste Demo ...

```

Sub AggregateDemo()
    Dim scheinbarerUmsatz = Aggregate artElement In artListe
        Into Summe = Sum(artElement.Einzelpreis)
    Console.WriteLine("Scheinbar beträgt der Umsatz {0:#,##0.00} Euro", scheinbarerUmsatz)

```

... aggregiert unsere Artikelliste in eine Gesamtsumme und ermittelt so scheinbar, wie viel Gesamtumsatz mit den Artikeln erzielt wurde, etwa wie hier zu sehen:

Scheinbar beträgt der Umsatz 47.117,60 Euro

Doch diese Aussage stimmt wirklich nur scheinbar. Denn bei der Abfrage wurden nur die Artikelpreise in einer Decimal-Variable (scheinbarerUmsatz) summiert – die Anzahl eines Artikel blieb unberücksichtigt.

Dieses Manko behebt das direkt anschließende Beispiel ...

```

Dim gesamtUmsatz = Aggregate artElement In artListe
    Let Artikelumsatz = artElement.Einzelpreis * artElement.Anzahl
    Into Summe = Sum(Artikelumsatz)
Console.WriteLine("Der Umsatz beträgt {0:#,##0.00} Euro", gesamtUmsatz)

```

... das mit einem sehr, sehr alten Bekannten – nämlich der Let-Klausel – eine neue Variable einführt, die das *Produkt* von Anzahl und Preis aufnimmt und anschließend als Ergebnis aufsummiert:

Der Umsatz beträgt 93.306,00 Euro

Zurückliefern mehrerer verschiedener Aggregierungen

Wenn eine Aggregat-Abfrage nur eine einzige Aggregat-Funktion verwendet, dann ist das Abfrageergebnis eine einzige Variable, die dem Rückgabetyp der Aggregatfunktion entspricht. Liefert beispielsweise die *Sum*-Aggregatfunktion ein Ergebnis vom Typ *Decimal*, dann ist das Abfrageergebnis ebenfalls vom Typ *Decimal*.

Sie können aber auch mehrere Aggregate innerhalb einer einzigen Abfrage verwenden, wie das dritte Beispiel zeigt:

```
Dim mehrereInfos = Aggregate artElement In artListe
    Let Artikelumsatz = artElement.Einzelpreis * artElement.Anzahl
    Into Summe = Sum(Artikelumsatz),
        Minimalpreis = Min(artElement.Einzelpreis),
        MaximalPreis = Max(artElement.Einzelpreis)
Console.WriteLine("Der Umsatz beträgt {0:#,#0.00} Euro" & vbNewLine &
    "Minimal- und Maximalpreis jeweils " &
    "{1:#,#0.00} und {2:#,#0.00} Euro",
    mehrereInfos.Summe, mehrereInfos.Minimalpreis,
    mehrereInfos.MaximalPreis)
```

In diesem Beispiel werden »in einem Rutsch« Artikelpostensumme, Maximalpreis und Minimalpreis ermittelt; die RückgabevARIABLE ist in dem Fall vom Typ *anonyme Klasse* mit den drei Eigenschaften, die über die entsprechenden Variablen innerhalb der Abfrage (*Summe*, *Minimalpreis* und *Maximalpreis*) definiert wurden. Hätten Sie diese Variablendefinitionen weggelassen und stattdessen nur die Aggregatfunktionen angegeben, hätte der Compiler für die Eigenschaften des anonymen Rückgabetypen Namen gewählt, die den Namen der Aggregatfunktionen entsprächen.

WICHTIG Anders als bei Standardabfragen werden reine Aggregatabfragen, wie Sie sie hier in den letzten drei Beispielen kennen gelernt haben, *nicht* verzögert sondern sofort ausgeführt.

Kombinieren von gruppierten Abfragen und Aggregatfunktionen

Das Gruppieren von Abfragen eignet sich naturgemäß gut für den Einsatz von Aggregatfunktionen, denn beim Gruppieren von Datensätzen werden verschiedene Elemente zusammengefasst, auf die sich dann die unterschiedlichsten Aggregatfunktionen anwenden lassen.

Ein Beispiel soll auch das verdeutlichen: Angenommen, Sie möchten herausfinden, welcher Kunde durch seine Käufe wie viel Umsatz produziert hat. In diesem Fall würden Sie die Abfrage nach Artikeln gruppieren, und zwar nach dem Feld *IDGekauftVon* – denn dieses Feld repräsentiert »*wer* hat gekauft«. Damit in der Ergebnisliste nicht nur eine nichtssagende Nummer sondern ein greifbarer Vor- sowie ein Nachname zu finden sind, verknüpfen Sie Artikeltabelle und Kundentabelle mit einer *Join*-Abfrage oder – noch besser – fassen Gruppierung und Zusammenfassung der Tabellen mit einem *Group Join* zusammen. In die *Group*

Join-Abfrage bauen Sie gleichzeitig noch die schon bekannte Order By-Klausel ein, mit der Sie die Ergebnisliste nach dem errechneten Umsatz sortieren – durch die Angabe von Descending als Schlüsselwort sogar absteigend, um eine Top-Ranking-Umsatzliste zu bekommen. Die eigentliche Aggregierung der Umsätze erfolgt dann mit der hinter Into stehenden Klausel, in der ebenfalls bestimmt wird, mit welcher Eigenschaft später die innere Liste (JoinedList) im Bedarfsfall durchiteriert werden kann (und dabei dann einzelne artListe-Elemente vom Typ Artikel zurückliefert). Die Aggregatfunktion Sum, die dafür notwendig ist, summiert im Beispiel das Produkt von Menge und Artikeleinzelpreis auf, was – für jeden Kontakt gruppiert – exakt dem Umsatz jedes Kunden entspricht.

TIPP Wie ebenfalls im folgenden Beispiel zu sehen, verwendet die Abfrage die Take-Klausel, um die Ergebnisliste auf 10 Elemente zu beschränken. Es gibt weitere Klauseln, die Sie für solche Einschränkungen verwenden können, wie beispielsweise Take While oder Skip – deren Anwendungsweise ist ähnlich und über dieses Beispiel leicht ableitbar. Die Online-Hilfe verrät Ihnen zu diesen Klauseln mehr.

```
Sub JoinAggregateDemo()
    Dim ergebnisListe = From adrElement In adrListe _
        Group Join artElement In artListe _
        On adrElement.ID Equals artElement.IDGekauftVon _
        Into _
        KundenUmsatz = Sum(artElement.Einzelpreis * artElement.Anzahl) _
        Order By KundenUmsatz Descending _
        Take 10

    For Each item In ergebnisListe
        With item.adrElement
            Console.WriteLine(.ID & ":" & .Nachname & ", " _
                & .Vorname & " --- Umsatz:" _
                & item.KundenUmsatz.ToString("#,##0.00") & _
                " Euro")
        End With
    Next
End Sub
```

Wenn Sie dieses Beispiel ausführen, produziert es in etwa die folgenden Zeilen:

```
3: Braun, Franz --- Umsatz:2.358,00 Euro
34: Plenge, Franz --- Umsatz:2.287,80 Euro
91: Jungemann, Katrin --- Umsatz:2.183,05 Euro
94: Tinoco, Anne --- Umsatz:2.128,00 Euro
100: Hollmann, Gareth --- Umsatz:2.108,15 Euro
18: Westermann, Anja --- Umsatz:2.098,05 Euro
66: Hörstmann, Katrin --- Umsatz:1.923,00 Euro
54: Albrecht, Uta --- Umsatz:1.843,40 Euro
97: Hollmann, Daja --- Umsatz:1.798,50 Euro
27: Westermann, Alfred --- Umsatz:1.788,15 Euro
```

Kapitel 33

LINQ to XML

In diesem Kapitel:

Einführung in LINQ to XML	902
XML-Dokumente verarbeiten – Visual Basic 2005 im Vergleich mit Visual Basic 2008	903
XML-Literale – XML direkt im Code ablegen	904
Erstellen von XML mithilfe von LINQ	905
Abfragen von XML-Dokumenten mit LINQ to XML	907
IntelliSense-Unterstützung für LINQ to XML-Abfragen	908

Einführung in LINQ to XML

XML steht für *Extensible Markup Language*, was auf Deutsch soviel wie *erweiterbare Auszeichnungssprache* bedeutet. XML ist eine Untermenge von SGML, welches bereits 1986 vom World Wide Web Consortium (W3C) standardisiert wurde.

Markup Languages (etwa: *Auszeichnungssprachen*) dienen dazu, mit einem Satz von Attributen eine Dokumentenstruktur inhaltlich zu gestalten, oder zu definieren, wie die Inhalte dieses Dokumentes dargestellt werden sollen – oder natürlich auch beides. Die *Hypertext Markup Language* (wir kennen sie eher unter ihrer Abkürzung *HTML*) ist dabei wohl die derzeit am häufigsten Angewendete für das Speichern und Darstellen von Inhalten im Web; aber auch die *Extensible Application Markup Language* (kurz XAML) ist Ihnen im Zusammenhang mit dem Thema *Windows Presentation Foundation* im Laufe dieses Buchs schon hier und da begegnet. Generell können XML-Dokumente einfacher Struktur auch dazu dienen, einfache Daten wie Adressen oder Artikel strukturiert zu speichern – das X (*extensible*, etwa: *erweiterbar*) in XML beschreibt schon, dass die Extensible Markup Language eben die Basis für konkrete Implementierungen für Markup Languages bildet. XML hat aber noch einen weiter gehenden Anspruch. Mit XML lassen sich prinzipiell alle Arten von Daten beschreiben und übertragen.

LINQ to XML wiederum ist, vereinfacht ausgedrückt, eine Implementierung, die dazu dient, dass Daten, die in XML-Dokumenten gespeichert werden, mithilfe der schon vorgestellten Language Integrated Query verarbeitet und selektiert werden können.

Bevor wir uns mit dem eigentlichen Kapitelthema *LINQ to XML* beschäftigen, lassen Sie uns kurz XML als solches rekapitulieren: Ein typisches XML-Dokument besteht, vereinfacht gesagt, aus Elementen, Attributen und Werten.

BEGLEITDATEIEN

Sie finden die Begleitdateien für die folgenden Abschnitte im Verzeichnis:

... \VB 2008 Entwicklerbuch\F - LINQ\Kapitel 33\LinqToXmlDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Beispiel:

```
<software Installierbar = "Ja" >Visual Studio 2008 </software>
```

Das Element Software besitzt ein Attribut Installierbar und den Wert Visual Studio 2008. Elemente in XML beginnen mit einem Starttag `<name>` und enden mit dem Endtag `</name>`, wobei *name* durch ein beliebiges Wort getauscht werden kann (Achtung: dabei dürfen keine Leerzeichen verwendet werden!).

Zwischen `<name>` und `</name>` steht der Wert. Ist kein Wert vorhanden, kann anstelle von `<name></name>` auch `<name/>` geschrieben werden.

Attribute werden in dem Element notiert, etwa:

```
<person name="Müller" vorname="Hans" />
```

Dieses Beispiel definiert ein Element ohne Wert, das zudem die Attribute *name* und *vorname* beinhaltet. Bei all diesen Beispielen handelt es sich genau genommen um ein XML-Fragment, kein vollständiges XML-

Dokument. Jedes XML Dokument sollte z. B. mit einer Auszeichnung beginnen, die deutlich macht, dass es sich um ein XML Dokument handelt: <?xml version="1.0"?>

Eine detaillierte Beschreibung von XML würde den Umfang dieses Buchs sprengen, daher sei hier auf die gute deutsche Einführung von *SelfHtml*¹ bzw. auf die Seiten von *W3C*² verwiesen.

XML-Dokumente verarbeiten – Visual Basic 2005 im Vergleich mit Visual Basic 2008

Um unter Visual Basic 2005 eine XML-Struktur zu erstellen oder zu ändern, mussten Sie schon einiges an Aufwand betreiben. Um das zu demonstrieren, schauen wir uns an, was zu tun ist, um folgende XML-Struktur, die Daten für Fahrzeuge eines Fuhrpark speichert, um das Element *baujahr* zu erweitern:

```
<fuhrpark>
<kennzeichen>MS - VB 2008</kennzeichen>
<ladung menge="10 Tonnen">Salz</ladung>
<hersteller>MAN</hersteller>
<baujahr>1998</baujahr>
</fuhrpark>
```

Der Code lautete hierzu für Visual Basic 2005:

```
Dim xml As New XmlDocument
xml.Load(Application.StartupPath + "\fuhrpark1.xml")
Dim nodeList As XmlNodeList = xml.GetElementsByTagName("fuhrpark")

'Für den Fall, dass es mehrere gäbe: Schleife
For Each node As XmlNode In nodeList
    Dim xmlElement As XmlElement = xml.CreateElement("baujahr")
    xmlElement.InnerText = "1998"
    node.AppendChild(xmlElement)
Next
```

In Visual Basic 2008 gibt es viele Vereinfachungen bezüglich der XML-Verarbeitung, denn das Framework wurde um einige recht leistungsfähige Klassen erweitert. Die wichtigsten dabei sind *XDocument*, *XElement* und *XAttribute*.

Um nunmehr mit Visual Basic 2008 das oben stehende XML-Dokument zu erhalten, reicht unter Zuhilfenahme dieser neuen Klassen folgender Code aus:

```
Dim xml As New XDocument
xml.Add(New XElement("fuhrpark",
    New XElement("kennzeichen", "MS - VB 2008"),
    New XElement("ladung", _
```

¹ Mehr dazu unter dem IntelliLink **A3301**

² Weiteres unter dem IntelliLink **A3302**

```

        New XAttribute("menge", "10 Tonnen"),
        "Salz" -
    ),
    New XElement("hersteller", "MAN") -
))

```

Unterhalb des Elementes `fuhrpark` werden die Elemente `kennzeichen`, `ladung` und `hersteller` gespeichert. Zudem erhält das Element `ladung` ein weiteres Attribut namens `menge`. Um nun ein weiteres Element `baujahr` hinzuzufügen, verwenden Sie einfach die `Add`-Methode, etwa wie in folgender Codezeile zu sehen:

```
xml.Element("fuhrpark").Add(New XElement("baujahr", 1998))
```

Möchten Sie den Wert eines `XElements` (z.B. 1998) ermitteln, verwenden Sie dazu die `Value`-Eigenschaft, etwa auf die folgende Art und Weise:

```

'Mit der Value-Eigenschaft Werte ermitteln
'Gesamter "Wert":
Console.WriteLine(xml.Element("fuhrpark").Value)

'Nur das Baujahr:
Console.WriteLine(xml.Element("fuhrpark").Element("baujahr").Value)
Console.ReadKey()

```

Die letzten Zeilen dieses Beispiel erzeugen folgende Ausgabe auf dem Bildschirm:

```
MS - VB 2008SalzMAN1998
1998
```

XML-Literale – XML direkt im Code ablegen

Ab Visual Basic 2008 gibt es die Möglichkeit, für `XDocument`-und `XElement`-Objekte die entsprechenden Dokumente direkt im Code zu erstellen – diese XML-Dokumente, die im Code abgelegt werden können, nennt man *XML-Literale*. Wie das funktioniert, zeigt das folgende Beispiel:

```
Dim fuhrpark As XElement = <fuhrpark>
    <kennzeichen>MS - VB 2008</kennzeichen>
    <ladung menge="10 Tonnen">Salz</ladung>
    <hersteller>MAN</hersteller>
</fuhrpark>
```

Beim `fuhrpark`-Objekt handelt es sich um ein gewöhnliches `XElement` und `baujahr` kann nun wie gewohnt hinzugefügt werden:

```
fuhrpark.Add(New XElement("baujahr", 1998))
```

Mit XML-Literalen geht das sogar noch einfacher:

```
fuhrpark.Add(<baujahr>1998</baujahr>)
```

Einbetten von Ausdrücken in XML-Literalen

XML-Dokumente, die Sie im Code direkt ablegen, müssen nicht komplett statisch sein, sondern können auch sich dynamisch anpassende Elemente erhalten. Es ist so möglich, Ausdrücke mit <%= %> als XML in den Code einzubetten, wie die folgenden beiden Zeilen zeigen:

```
Dim myBaujahr As Integer = 1998  
fuhrpark.Add(<baujahr><%= myBaujahr %></baujahr>)
```

Beim Aufrufen der Add-Methode wird der Wert des baujahr-Elements durch die Variable myBaujahr ersetzt.

Erstellen von XML mithilfe von LINQ

Mithilfe von LINQ können ebenso XML-Dokumente bzw. -Elemente erstellt werden.

Aus einer bestehenden Liste, die durch die folgenden Codezeilen erstellt wird ...

```
Dim fahrzeugliste As New List(Of Fahrzeug)  
fahrzeugliste.Add(New Fahrzeug With {.Kennzeichen = "MS-VB 2008", .Hersteller = "MAN", -  
    .Ladung = New Fahrzeug.LadungsBeschreibung With  
        {.Menge = 10000, .Gut = "Salz"})})  
fahrzeugliste.Add(New Fahrzeug With {.Kennzeichen = "MS-C# 2008", .  
    Hersteller = "Mercedes-Benz",  
    .Ladung = New Fahrzeug.LadungsBeschreibung With  
        {.Menge = 20000, .Gut = "Erdnüsse"})})  
fahrzeugliste.Add(New Fahrzeug With {.Kennzeichen = "MS-J# 2008", .Hersteller = "DAF",  
    .Ladung = New Fahrzeug.LadungsBeschreibung With  
        {.Menge = 5000, .Gut = "Wackeldackel"})})  
  
Private Class Fahrzeug  
    Public Class LadungsBeschreibung  
        Private _menge As Double  
        Private _gut As String  
  
        Public Property Menge() As Double  
            ...  
        End Property  
  
        Public Property Gut() As String  
            ...  
        End Property  
    End Class  
  
    Private _kennzeichen As String  
    Private _ladung As LadungsBeschreibung  
    Private _hersteller As String  
  
    Public Property Kennzeichen() As String  
        ...  
    End Property  
  
    Public Property Ladung() As LadungsBeschreibung  
        ...  
    End Property
```

```

    Public Property Hersteller() As String
    ...
End Property
End Class

```

... soll folgendes XML-Element erstellt werden:

```

<fuhrpark>
  <fahrzeug>
    <kennzeichen>MS-VB 2008</kennzeichen>
    <hersteller>MAN</hersteller>
    <ladung menge="10000">Salz</ladung>
  </fahrzeug>
  <fahrzeug>
    <kennzeichen>MS-C# 2008</kennzeichen>
    <hersteller>Mercedes-Benz</hersteller>
    <ladung menge="20000">Erdnüsse</ladung>
  </fahrzeug>
  <fahrzeug>
    <kennzeichen>MS-J# 2008</kennzeichen>
    <hersteller>DAF</hersteller>
    <ladung menge="5000">Wackeldackel</ladung>
  </fahrzeug>
</fuhrpark>

```

Das Anlegen der XML-Strukturen erfolgt dabei mit XElement und XAttribute:

```

Dim fuhrpark = New XElement("fuhrpark", _
    From fzg In fahrzeugliste _
    Select New XElement("fahrzeug", _
        New XElement("kennzeichen", fzg.Kennzeichen), _
        New XElement("hersteller", fzg.Hersteller), _
        New XElement("ladung", fzg.Ladung.Gut, _
            New XAttribute("menge", fzg.Ladung.Menge)) _
    )
)

```

Dem XElement mit dem Namen Fuhrpark wird eine Liste von XElementen (fahrzeug) hinzugefügt. Diese Liste (genaugenommen IEnumerable(of XElement)) wird mit *LINQ to Objects* erstellt. Das fahrzeug-Element wird zudem um die entsprechenden Unterelemente erweitert.

Auch hier ist wieder die Verwendung von Literalen möglich:

```

fuhrpark = <fuhrpark>
    <%= From fzg In fahrzeugliste _
    Select <fahrzeug>
        <kennzeichen><%= fzg.Kennzeichen %></kennzeichen>
        <hersteller><%= fzg.Hersteller %></hersteller>
        <ladung menge=<%= fzg.Ladung.Menge %>
            <%= fzg.Ladung.Gut %>
        </ladung>
    </fahrzeug> _
%
</fuhrpark>

```

Abfragen von XML-Dokumenten mit LINQ to XML

LINQ bietet mit *LINQ to XML* auch eine Abfrageunterstützung für XML-Strukturen an.

Ein Beispiel:

```
Dim fuhrpark = <?xml version="1.0"?>
    <fuhrpark>
        <fahrzeug>
            <kennzeichen>MS-VB 2008</kennzeichen>
            <hersteller>MAN</hersteller>
            <ladung menge="10000">Salz</ladung>
        </fahrzeug>
        <fahrzeug>
            <kennzeichen>MS-C# 2008</kennzeichen>
            <hersteller>Mercedes-Benz</hersteller>
            <ladung menge="20000">Erdnüsse</ladung>
        </fahrzeug>
        <fahrzeug>
            <kennzeichen>MS-J# 2008</kennzeichen>
            <hersteller>DAF</hersteller>
            <ladung menge="5000">Wackeldackel</ladung>
        </fahrzeug>
    </fuhrpark>
Dim manFahrzeuge = From fahrzeug In fuhrpark...<fahrzeug>
Where fahrzeug.<hersteller>.Value = "MAN"
For Each fahrzeug In manFahrzeuge
    Console.WriteLine(fahrzeug)
Next
```

Bei der Variablen `fuhrpark` handelt es sich um ein `XDocument`. In der `From`-Klausel wird auf alle `XElemente` mit dem Namen `Fahrzeug` zugegriffen. Die `Where`-Klausel prüft den `Hersteller` des Fahrzeugs auf den Eintrag »`MAN`«. Der Wert eines `XElements` muss über die `Value` Property erfragt werden.

Es werden zwei Zugriffsmöglichkeiten für Elemente bereitgestellt:

Der Zugriff auf direkte Unterelemente erfolgt über `.<Element-name>`. Zudem kann auf darunterliegende Elemente mit `...<Element-name>` zugegriffen werden, wobei sich das Element durchaus tief in den XML-Strukturen befinden darf.

Auf den Inhalt eines Attribut kann über `.@<Attribut-name>` zugegriffen werden.

Beispiel: Es sollen alle schwer beladenen Fahrzeuge ermittelt werden:

```
Dim schwerBeladen = From fahrzeug In fuhrpark...<fahrzeug>
    Where CInt(fahrzeug.<ladung>.@menge) > 15000
For Each fahrzeug In schwerBeladen
    Console.WriteLine(fahrzeug)
Next
```

ergibt folgende Ausgabe:

```
<fahrzeug>
    <kennzeichen>MS-C# 2008</kennzeichen>
    <hersteller>Mercedes-Benz</hersteller>
    <ladung menge="20000">Erdnüsse</ladung>
</fahrzeug>
```

ACHTUNG **Aufgepasst bei der Typsicherheit bei LINQ to XML:** Auch wenn Sie – wie Sie beim Nachschauen der Beispieldateien erkennen können – für Ihr XML-Dokument mithilfe einer XSD-Datei die Typen des Dokumentes definiert haben, gilt: Werte der LINQ-Abfragen bekommen Sie grundsätzlich als String zurück – das gilt sowohl für Elemente als auch für Attribute. Wie im Listing zu sehen, müssen Sie also stets entsprechende Casting-Operatoren einsetzen, um den wirklichen Typ abzufragen, und dabei können natürlich Fehler auftreten.

Achten Sie deswegen besonders darauf, gerade wenn Sie LINQ to XML verwenden, grundsätzlich typsicher zu entwickeln – Option Explicit muss also eingeschaltet sein! Andernfalls laufen Sie Gefahr, dass Sie Strings verarbeiten, obwohl Sie eigentlich numerische Typen vergleichen sollten, und dann kann es ruck zuck passieren, dass 4.500 größer als 15.000 ist!

IntelliSense-Unterstützung für LINQ to XML-Abfragen

Bei der Arbeit mit XML-Daten ist jede Unterstützung der IDE sehr hilfreich. Visual Basic 2008 bietet genau diese an, wie in der folgenden Abbildung zu sehen:

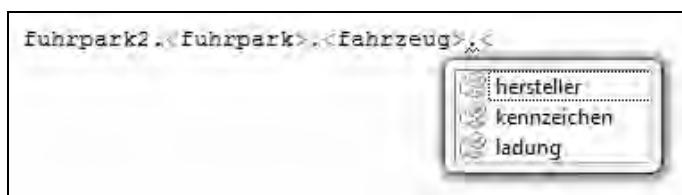


Abbildung 33.1 XML IntelliSense-Unterstützung

Diese Unterstützung erhält man, sobald eine XML Schema Definition (XSD)-Datei mit Imports in die Codedatei importiert wird.

Aber fangen wir mal vorne an. Eine XSD-Datei kann Visual Studio selbst erstellen. Hierzu erstellt man zunächst die komplette XML-Datei mit allen möglichen Ausprägungen.

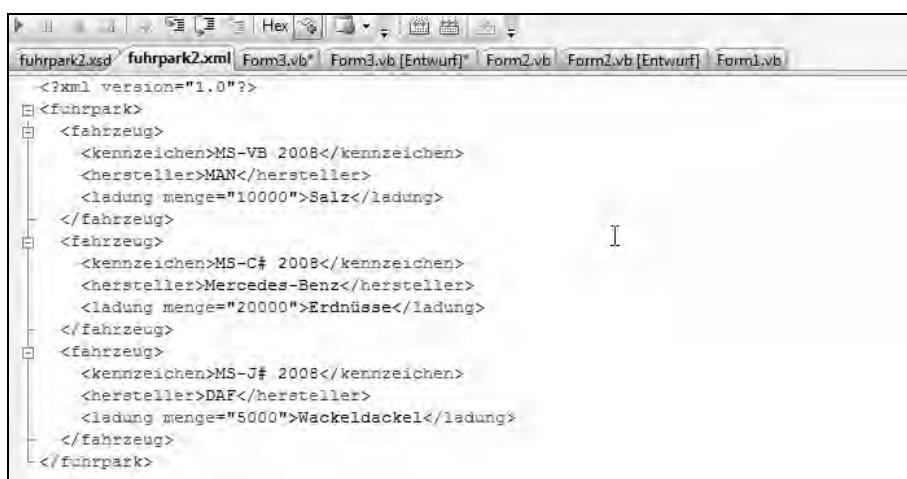


Abbildung 33.2 Eine XML-Datei in Visual Studio

Danach kann man mithilfe des gleichlautenden Befehls im Menü *XML* ein Schema erstellen:

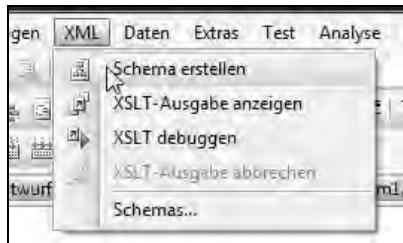


Abbildung 33.3 Erstellen einer XML Schema Definition (XSD)

Das erstellte Schema muss nun im Projektverzeichnis gespeichert und in das Projekt mit aufgenommen werden.

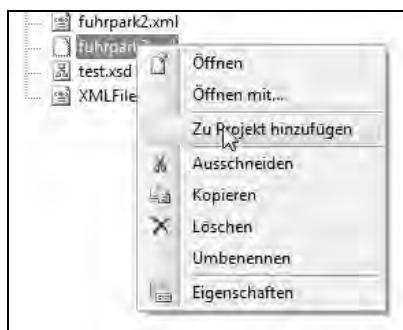


Abbildung 33.4 Hinzufügen einer Datei im Projektmappen-Explorer

In der soeben erstellten XSD-Datei muss nun noch ein Ziel-Namensbereich (»Targetnamespace«) angegeben werden. Dieser Namensbereich muss nicht existieren, sondern dient lediglich der Klassifikation. In diesem Fall wurde <http://mysample.local.de/fuhrpark2> verwendet.

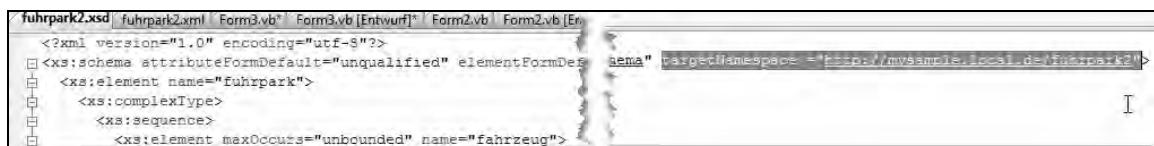


Abbildung 33.5 Erweitern einer XSD-Datei um den Ziel-Namensbereich

In der VB-Datei, die eine XML-Unterstützung erhalten soll, kann jetzt der soeben festgelegte Ziel-Namensbereich importiert werden.

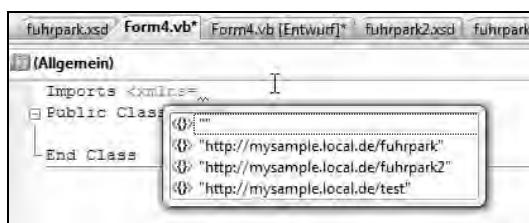


Abbildung 33.6 Hinzufügen einer Imports-Anweisung für XML Schema-Dateien

Werden mehrere Namensräume verwendet, bietet es sich an, den Namensraum mit einem Präfix zu versehen. Das Präfix wird nach `xmlns` notiert und mit einem Doppelpunkt von `xmlns` getrennt. In diesem Fall wird als Präfix `fuhrpark` verwendet. Bei nur einem Namensraum, der in die VB-Codedatei importiert wird, ist das nicht erforderlich.

```
Imports <xmlns:fuhrpark="http://mysample.local.de/fuhrpark2">
Imports <xmlns:artikel="http://mysample.local.de/artikel">
```

Verwendung von Präfixen (`fuhrpark` und `artikel`)

Die XML-Strukturen müssen in diesem Fall auch um die Präfixe erweitert werden.

```
Dim fuhrpark = <fuhrpark:fuhrpark>
    <fuhrpark:fahrzeug>
        <fuhrpark:kennzeichen>MS-VB 2008</fuhrpark:kennzeichen>
        <fuhrpark:hersteller>MAN</fuhrpark:hersteller>
        <fuhrpark:ladung menge="10000">Salz</fuhrpark:ladung>
    </fuhrpark:fahrzeug>
    <fuhrpark:fahrzeug>
        <fuhrpark:kennzeichen>MS-C# 2008</fuhrpark:kennzeichen>
        <fuhrpark:hersteller>Mercedes-Benz</fuhrpark:hersteller>
        <fuhrpark:ladung menge="20000">Erdnüsse</fuhrpark:ladung>
    </fuhrpark:fahrzeug>
    <fuhrpark:fahrzeug>
        <fuhrpark:kennzeichen>MS-J# 2008</fuhrpark:kennzeichen>
        <fuhrpark:hersteller>DAF</fuhrpark:hersteller>
        <fuhrpark:ladung menge="5000">Wackeldackel</fuhrpark:ladung>
    </fuhrpark:fahrzeug>
</fuhrpark:fuhrpark>
```

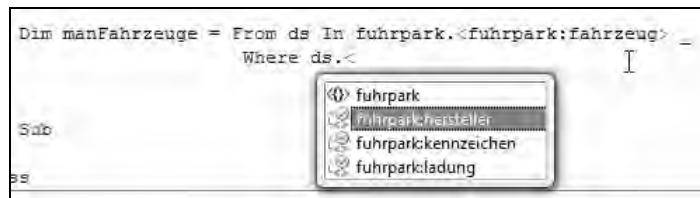


Abbildung 33.7 XML-Unterstützung der IDE

Kapitel 34

LINQ to SQL

In diesem Kapitel:

Einleitung	912
Object Relational Mapper (O/RM)	912
Microsoft SQL Server	914
Voraussetzungen für die Beispiele dieses und des nächsten Kapitels	918
LINQ to SQL oder LINQ to Entities – was ist besser, was ist die Zukunft?	935
Wie es bisher war – ADO.NET 2.0 vs. LINQ in .NET 3.5	939
LINQ to SQL am Beispiel – Die ersten Schritte	941
Verzögerte Abfrageausführung und kaskadierte Abfragen	952
Eager und Lazy-Loading – Steuern der Ladestrategien bei 1:n-Relationen	955
Trennen des Abfrageergebnisses vom Kontext	960
Daten verändern, speichern, einfügen und löschen	961
Transaktionen	971
Was, wenn LINQ einmal nicht reicht	973
LINQ-Abfragen dynamisch aufbauen	974

Einleitung

Als ich mit den ersten Beta-Versionen meine ersten Gehversuche mit LINQ machte, war ich begeistert. Nach und nach stellte ich fest, welche enormen Möglichkeiten sich mir mit LINQ auf einmal boten – und ich hatte zu diesem Zeitpunkt doch noch gar nichts gesehen! Während dieser Gehversuche mit einer der frühen Betas von Visual Studio 2008 wusste ich nämlich zum damaligen Zeitpunkt, irgendwann im April 2007, noch gar nicht, wie sehr ich erst an der Spitze des Eisberges 'rumdokterte.

Doch nach und nach begriff ich, wohin »uns« diese ganze LINQ-Geschichte führen sollte, und irgendwann war es dann auch so weit, dass es die ersten Visual Studio Betas gab, die uns Entwicklern vorläufige Versionen von *LINQ to SQL* vorführten. Wir hier im Entwicklungsteam waren alle total begeistert; sollte uns Microsoft mit LINQ to SQL endlich den jahrelang erwarteten O/RM bescheren? So hatte es zumindest den Anschein, und LINQ to SQL war nur der Anfang.

Das Idee hinter LINQ to SQL ist eigentlich ganz einfach: Sie »sagen«: »Ich programmiere im Folgenden genau so, wie ich es mit LINQ to Objects gelernt habe, die Quelle meiner Daten ist jedoch keine Auflistung, sondern eine SQL Server-Datenbank. Ansonsten bleibt alles genau so«.

Und so einfach soll das sein? Keine neuen Objekte, Klassen und Verfahren, die man lernen muss? Nichts Spezielles, was man zu beachten hat? Gibt es keine Falltüren, auf die man aufpassen sollte?

Doch, na klar, ein paar gibt es, denn Sie wollen Daten ja nicht nur aus dem SQL Server »abholen«, Sie wollen sie ja schließlich nach ihrer Verarbeitung auch wieder zurück in den Server bekommen, und dazu musste die reine LINQ-Infrastruktur ein wenig »aufgebohrt« und erweitert werden, damit sie das gestattet. Man muss sich darüber hinaus auch an ein paar bestimmte Konventionen zu halten, die man dazu wiederum kennen muss, damit dann solche Sachen wie gespeicherte Prozeduren (Stored Procedures) auch berücksichtigt werden müssen.

Die Kehrseite der Medaille ist jedoch: Es gibt nicht nur LINQ to SQL; mit dem Service Pack 1 für Visual Basic 2008 kam ein weiterer LINQ-Provider, und der nennt sich LINQ to Entities. Diese Tatsache ist für viele Entwickler ein wenig verwirrend, denn in LINQ to Entities und LINQ to SQL gibt es so einiges an Schnittmengen. Man kann nicht pauschal sagen, LINQ to Entities ist besser als LINQ to SQL; Sie müssen diese Bewertung immer im Kontext dessen sehen, was Sie mit welchen Erweiterungs- bzw. Ausbaumöglichkeiten erreichen wollen. Doch dazu verrät Ihnen der übernächste Abschnitt mehr.

Object Relational Mapper (O/RM)

Ganz so trivial, wie in der Einleitung beschrieben, sind die beiden Technologien LINQ to SQL bzw. LINQ to Entities – auf Letztere gehen wir im nächsten Kapitel noch detaillierter ein – natürlich nicht. Denn beide erfüllen die Aufgabe eines sogenannten Object Relational Mappers, der gerne auch mit O/RM abgekürzt wird. Und um was geht's dabei?

Prinzipielle Anforderung eines O/RMs ist es, eine Infrastruktur zu bieten, mit der eine in einer OOP-Sprache wie Visual Basic .NET geschriebene Anwendung ihre Geschäftsobjekte in einer Datenbank ablegen kann bzw. aus dieser auslesen kann. Perfekt würde das aber nur funktionieren, würde man auch tatsächlich auf eine Objektorientierte Datenbank zurückgreifen, bei der Datenbanktabellen wie Objekte in Visual Basic vererbbar wären.

Das entspricht aber leider nicht der Realität, da

- Entwickler auf die Datenbankplattformen zurückgreifen müssen, die sich als quasi Industriestandard durchgesetzt haben, und das sind eben Microsoft SQL Server, Oracle, IBM's DB2 oder MySql (Access und ähnliche dateibasierten Datenbanksysteme klammere ich hier bewusst aus, denn Access bietet keine Datenbankserver-Funktionalität – Abfragen werden hier auf jedem Client berechnet, während ein Datenbankserver Datenbankoperationen auf seiner (dedizierten) Maschine mit deren Ressourcen durchführt).
- Datenbanken sind in der Regel zunächst vorhanden, und der Entwickler hat durch so genanntes Reverse Mapping die Aufgabe, Klassen in seiner Anwendung zu erstellen (oder eben erstellen zu lassen), mit der sich eine möglichst gute Relation zwischen den so entstehenden Business-Objekten und den Datenbanktabellen bzw. Sichten (Views) der Datenbank ergibt.
- Die verwendeten Datenbanken sind in der Regel eben nicht objektorientiert aufgebaut, sodass besondere Abbildungstechniken dafür sorgen müssen, dass der objektorientierte Anspruch nicht verloren geht. In der Regel passiert bei einer solchen Abbildung/Zuordnung (eben: Mapping) Folgendes:
 - Der O/RM stellt eine grundsätzliche Verwaltungsklasse zur Verfügung, die die Kommunikation mit dem Datenbankserver regelt.
 - Tabellen oder Sichten einer Datenbank werden als Businessobjekte abgebildet. Eine Tabellenzeile (oder eine Datenzeile einer Sicht) entspricht dabei der Instanz einer Klasse. Die Tabellenspalten (also die Datenfelder) mappen zu jeweils einer Eigenschaft dieser Instanz. Eine Tabelle (oder das Ergebnis einer Frage) ist dann eine Auflistung (eine Collection) vom Typ dieser Klasse. Damit wirkliches O/RM möglich wird, gehen viele O/RMs einen Weg über eine Zwischenschicht, den sogenannten *Conceptual Layer*, der eine Art »Umleitungs-Verbindung« schafft zwischen Objekten und der Datenbank, und diese Beziehung eben nicht 1:1 abbildet. Um es vorweg zu nehmen: Das ist der größte Unterschied zwischen LINQ to SQL (1:1) und LINQ to Entities (bietet die Möglichkeit eines Conceptual Layers).
- Die von O/RMs zur Verfügung gestellte Programmierinfrastruktur muss dann zur Laufzeit natürlich auch dafür sorgen, die Verbindung zur Datenbank zu öffnen, die Auflistungen der von ihnen zuvor generierten Businessobjekte mit Daten zu befüllen, Änderungen an ihnen festzustellen und Aktualisierungslogiken bereitzustellen, damit geänderte Daten in den Business-Objekten auch wieder ihren Weg zurück in die Datenbank finden.

Objekt-relationale Unverträglichkeit – Impedance Mismatch

Wenn Sie ein erfahrender objektorientierter Entwickler sind, und auch schon Datenbankanwendungen entwickelt haben, dann wissen Sie, dass O/RMs eigentlich immer unvollkommen arbeiten müssen, wenn nicht eine wirkliche objektorientierte Datenbank als Grundlage für die Objektpersistierung (also für die Speicherung der Inhalte der Businessobjekte) vorhanden ist. Objekte einer OO-Programmiersprache und relationale Datenbanken sind einfach komplett unterschiedliche Topologien, bei denen Konstellationen auftreten können, die ein System abbilden können und andere eben nicht.

Denken Sie beispielsweise an eine Datenbank, die Autoren und Bücher verwaltet. Natürlich gibt es eine Menge Bücher und eine Menge Autoren. Würde es ein Gesetz geben, das verbieten würde, dass ein Buch von mehreren Autoren geschrieben werden kann, gäbe es in Sachen Abbildung Datenbank → Objektmodell

keine Probleme. Sie können jedem Buch *einen* Autor zuweisen, und Sie haben damit eine 1:N-Relation zwischen Autor und Büchern geschaffen (ein Autor (1) kann mehrere Bücher (n) schreiben, aber ein Buch kann nur von einem Autor geschrieben werden).

Wenn ich mich allerdings nun dazu entschließe, mit meinem Kumpel Ruprecht Dröge ein Buch zu schreiben, dann kommen wir mit dieser Zuordnung nicht mehr hin – wollten wir unser Buch später in der Datenbank ablegen. Wir brauchen jetzt eine Zuordnung, die viele Bücher (n) mit vielen Autoren (m) verknüpft. In einem Objektmodell ist das kein Problem. Eine Instanz der Klasse Autor hat eine Auflistung Bücher, und eine Instanz der Klasse Buch hat eine Auflistung Autoren. Die Elemente zweier Auflistungen können ohne Probleme Teile der Auflistung referenzieren. In einer relationalen Datenbank ist das ein Problem: Da eine Tabellenzeile nur einen Verweis auf einen Datensatz einer anderen Tabelle halten kann, muss man eine weitere Datentabelle zu Hilfe nehmen, die die Relation zwischen den Beziehungen »viele Bücher« und »viele Autoren« abbildet. Dies wird dann zumeist durch eine Zwischentabelle realisiert, zu der sowohl die Autoren, als auch die Büchertabelle eine 1:n Beziehung aufbaut. Einen anderen Weg gibt es zurzeit nicht.

Und genau hier unterscheiden sich zum Beispiel die beiden O/RMs von Microsoft voneinander: LINQ to SQL ist beispielsweise nicht in der Lage, solche Zwischentabellen so aufzulösen, dass sich wieder zwei abhängige Auflistungsklassen ergeben (denn die reichten ja eigentlich aus!) – LINQ to Entities hat das aber drauf!

Microsoft SQL Server

Wenn Anwendungen entwickelt werden, dann müssen diese immer in irgendeiner Form mit Daten hantieren. Bei Datenbankanwendungen greift dann nicht nur *ein* Computer auf die Daten zu, sondern *mehrere* Computer müssen zur gleichen Zeit mit einer Datenquelle arbeiten. Zwar ist das Framework durchaus in der Lage, auch komplexere Datenstrukturen über Rechnergrenzen hinaus zu verwalten, doch der Einsatz von Datenbanksystemen ist für solche Lösungen angezeigt.

Aber auch für Einzelplatzlösungen ist der Einsatz von Datenbankanwendungen eine sinnvolle Angelegenheit. Dank SQL (*Structured Query Language*, etwa: strukturierte Abfragesprache) können im Zusammenspiel mit LINQ (oder auch mit dem »alten« reinen ADO.NET 2.0) Daten auf einfache Weise verwaltet und Datenbank-Engine-unterstützt sortiert und gefiltert werden, und da Daten sowieso nach einer Arbeitssitzung gespeichert werden müssen, bietet sich der Einsatz einer Datenbank auch für Insellösungen mehr als an.

HINWEIS Einige der in diesem Kapitel vorgestellten Funktionen können Sie leider nicht mit der Visual Basic Express Edition durchführen, weil ihr die entsprechende Funktionalität fehlt (wie beispielsweise der Server-Explorer). Das bedeutet natürlich nicht, dass Sie keine Datenbankprogrammierung mit Visual Basic Express durchführen können! Visual Basic Express leistet sprachtechnisch das Gleiche, wie jede andere Version von Visual Basic bzw. Visual Studio, und dazu gehört natürlich auch ADO.NET. Ihnen fehlt hier und da lediglich die Designer-Unterstützung.

Sowohl LINQ to SQL als auch LINQ to Entities sind für diese Art der Datenanwendungen ausgelegt; wenn wir also wissen wollen, wie die beiden Systeme funktionieren, dann benötigen wir natürlich auch eine Datenbankplattform – und was liegt da näher, als eine zu verwenden, die zu den besten und professionellsten zählt und obendrein nichts kostet!

SQL Server 2008 Express Edition with Tools

SQL Server 2008 Express ist der Nachfolger von SQL Server 2005 Express Edition, der wiederum aus der Microsoft SQL Server Desktop Engine, vielen eher bekannt unter dem Namen MSDE, hervorgegangen ist. Alle drei Generationen basieren auf ihren größeren Brüdern, SQL Server 2008, SQL Server 2005 bzw. SQL Server 2000, ihnen fehlte bis jetzt jedoch einiges in Sachen Funktionalität für Administrationsaufgaben. Außerdem fehlen Ihnen, bis auf eine Ausnahme, auf die wir weiter unten noch einmal zu sprechen kommen, weitere Dienste der Kaufversionen im Bereich der Business Intelligence Lösungen. Auch einige Funktionen im Bereich der Datenbankdienste, besonders solche, die man braucht, um auf sehr große Datenmengen effektiv zuzugreifen, sind in der Express Version nicht verfügbar.

Für SQL Server 2005 gibt es in Ergänzung zum eigentlichen Kernprodukt bislang das Management Studio Express, das Sie neben der eigentlichen Datenbank-Engine gesondert herunterladen konnten.¹

Mit SQL Server 2008 Express wurde das ein wenig anders. Es gibt zu diesem Produkt das Management Studio, mit dem Sie auch den großen Bruder administrieren, in einer speziellen Version, die den Postfix »with Tools« (etwa: mit Werkzeugen) trägt. So fühlt sich, nachdem Sie SQL Server 2008 Express Edition with Tools herunter geladen und installiert haben (siehe Abschnitt »Download und Installation von SQL Server 2008 Express Edition with Tools« ab Seite 919), diese Version zunächst wie der nächst größere Bruder an – aber das ist leider nicht so.

Die Express Edition erfuhr nämlich, wie schon ihre Vorgänger, ein paar Einschnitte in ihrer Leistungsfähigkeit, aber dafür stellt Microsoft dieses Datenbanksystem unentgeldlich zur Verfügung, und sie lassen sich damit als professionelle Datenbankplattformen in Ihren eigenen Anwendungen einsetzen.

SQL Server Express erfuhr die folgenden Einschränkungen:

- Datenbanken sind auf maximal 4 GByte beschränkt.
- Maximal ein Prozessor eines Systems wird verwendet.
- Maximal ein Gigabyte Hauptspeicher eines Systems kommt zur Anwendung.

HINWEIS Denken Sie daran, dass SQL Server Express sich auch auf Computern installieren lässt, die über mehr Hauptspeicher oder mehrere Prozessoren verfügen. Diese werden dann lediglich nicht genutzt.

Denken Sie auch daran, dass Sie sich bei Microsoft explizit registrieren lassen müssen, bevor Sie eine eigene Anwendung auf Basis von SQL Server Express vertreiben. Das kostet zwar nichts (Stand der Drucklegung dieses Buchs), Sie müssen es aber halt machen. Der IntelliLink **F3401** bringt Sie direkt zu dieser Registrierungsseite (Sie benötigen dafür ein Microsoft Passport Konto).

Von diesen Einschränkungen abgesehen, steht Ihnen mit SQL Server Express eine Datenbankplattform zur Verfügung, die dem großen Bruder – was die relationale Datenbank angeht – in beinahe nichts nachsteht. Umfangreiche, höchst performante Datenbanken lassen sich mit dieser Software realisieren und die volle Programmierfähigkeit mit gespeicherten Prozeduren (*Stored Procedures*), Funktionen und auch die Ausführungsfähigkeit von verwaltetem Code steht Ihnen mit SQL Server Express zur Verfügung.

¹ Falls Sie es im Rahmen einer SQL Server 2005 Express-Installation noch benötigen, Sie finden es unter dem IntelliLink **F3402**.

Dazu gibt es eine um Reporting Services abermals erweiterte Version, die sogenannte SQL Server 2008 Express Edition With Advanced Services, die kostenfrei das komplette Reporting-Paket des großen Bruders zusammen mit den Features der hier beschriebenen »with Tools«-Express Edition in sich vereint. Diese Version – die wir in dem Kontext des Buchs aus Platzgründen leider nicht besprechen können – ist für viele Entwickler übrigens mit der 2008er Version noch mal attraktiver geworden, da SQL Server 2008 in Sachen Reporting Services gänzlich auf den Internet Information Server als Pflichtvoraussetzung verzichten kann. Hintergrund: In vielen Betrieben war und ist der Einsatz des IIS aus Angst vor bösen, bösen Menschen, die durch das Internet ins System einbrechen und eine ganze Firma kompromittieren, leider immer noch groß, auch wenn der betroffene Server vielleicht gar nicht am externen »Internet« hängt. Aber auch dieses kleine, meiner Meinung nach nicht unwichtige, Detail unterbricht die Argumentation gewisser Entscheidungsträger nicht in deren Argumentationskette – na ja, dank SQL Server 2008 ist das glücklicherweise nun ein wenig »egal« geworden.

Im Übrigen beinhaltet diese Advanced Services-Version von SQL Server 2008 auch einen Dienst zur sogenannten Volltextsuche in Datenbanken, eine Funktionalität, die bei der reinen With-Tools-Edition nicht vorhanden ist. Wichtig in diesem Zusammenhang:

ACHTUNG Es gibt von der Firma Codeplex, die von Microsoft zum Hosting der Beispieldatenbanken beauftragt wurde, Datenbankbeispiele sowohl zu SQL Server 2005 als auch zur 2008er-Version. Zur Drucklegung dieses Buchs war leider keine 2008er-Version der Beispiele verfügbar, die ohne die Funktionalität der Volltextsuche auskommen würde. Aus diesem Grund schlägt die Installation der Beispiele fehl, wenn Sie die 2008er-Beispiele für die With-Tools-Version von SQL Server 2008 Express Edition installieren wollen. Ihre Alternative: Sie installieren die SQL Server 2008 Express Edition with Advanced Services und beziehen später, während der Installation die Volltextsuche als Komponente mit in den Installationsvorgang ein, oder Sie bedienen sich der 2005er Beispiele, die auch ohne Probleme auf SQL Server 2008 laufen.

Wir entscheiden uns hier für die letzte Variante, damit Sie die folgenden Beispiele obendrein auch dann nachvollziehen können, wenn Sie sich – beispielsweise aus Abwärtskompatibilitätsgründen zu Windows Server 2000 SP4 – für den Einsatz von SQL Server 2005 Express Edition entscheiden.

TIPP Noch mehr Möglichkeiten bietet Ihnen die Developer Edition von SQL Server 2005 bzw. SQL Server 2008. Diese können Sie entweder für kleines Geld, etwa um \$100, bei Microsoft beziehen, oder Sie finden Sie im Rahmen eines MSDN-Entwickler-Abos (für Entwickler sehr empfehlenswert! – mehr dazu in Kapitel 1 und IntelliLink [A0102](#) bzw. [A0103](#)) zum Herunterladen. Die Developer Edition entspricht der ganz großen Enterprise Edition, Sie können sie allerdings anders als die Enterprise Edition auch auf Nicht-Serverbetriebssystemen, wie Windows XP, Windows Vista oder Windows 7 installieren. Sie müssen dabei allerdings beachten, dass Sie diese Version nur zum Entwickeln und Testen, nicht aber in Produktionsumgebungen einsetzen dürfen.

Einsatz von SQL Server Express in eigenen Anwendungen

SQL Server Express wird ab Visual Studio 2005 Standard Edition direkt mitinstalliert. Leider wird auch mit Visual Studio 2008 noch SQL Server Express 2005 mitinstalliert, sodass Sie sich um die Installation von SQL Server 2008 mal in jedem Fall kümmern müssen, aber der folgende Abschnitt klärt ja zum Glück, wo sie es her bekommen und wie sie es am geschicktesten installieren.

HINWEIS SQL Server 2008-Derivate lassen sich leider nicht mehr unter Windows 2000 installieren. Falls Sie Windows 2000-Server beim Kunden vorfinden, sollten Sie überlegen, ob Sie Ihre Datenbankanwendung noch auf Basis von SQL Server 2005 entwickeln und austesten wollen. Eine Installationsanleitung zu SQL Server 2005 Express Edition finden Sie in der Vorgängerversion zu diesem Buch, die Sie sich unter dem IntelliLink **F3403** herunterladen können. Dort finden Sie im Übrigen auch eine Einführung in ADO.NET 2.0, auf deren Wiederholung und weiteren Ausbau wir zugunsten von LINQ und SQL Server 2008 verzichtet haben.

Wichtig, wie die Installation vonstatten geht, ist auch dann, wenn Sie SQL Server Express später in einer Produktivumgebung, also beim Kunden einsetzen wollen. Dort begegnen Ihnen dann unter Umständen Herausforderungen, mit denen Sie sich nicht erst zum Zeitpunkt der Installation Ihrer Anwendung beschäftigen sollten, denn:

Anwendungen, die auf SQL Server Express basieren, lassen sich solange problemlos installieren, wie SQL Server Express auf dem gleichen Rechner wie Ihre Anwendung selbst läuft. Etwas aufwändiger wird es, wenn ...

- ... Ihre Anwendung von mehreren Rechnern aus auf eine SQL Server Express-Instanz zugreifen soll.
- ... Ihnen kein Server-Betriebssystem, das als Active Directory eingerichtet ist, als Plattform für SQL Server Express zur Verfügung steht, oder Sie SQL Server Express nicht zumindest auf einem Computer (XP Professional, eine Windows Vista-Version mit Domänenanbindungsmöglichkeit) installieren können, der Mitglied einer Domäne ist.

Der Grund: Standardmäßig wird SQL Server Express (wie übrigens auch seine großen Brüder) so installiert, dass es die integrierte Sicherheit des Betriebssystems für Anmeldungen an einer SQL Server-Instanz oder einer Datenbank nutzt. Solange wie ein Benutzer (oder eine Anwendung) sich auf dem gleichen System an der SQL Server-Instanz anmeldet, auf dem SQL Server Express selbst auch installiert wurde, ist das kein Problem, dann nämlich wird der lokale Rechner als AnmeldeDomäne benutzt. Problematisch wird bei dieser Anmeldemethode die Anmeldung von Rechnern des Netzwerkes, wenn kein Domänencontroller zur Verfügung steht, der eine vertraute Verbindung zwischen dem Client-Computer und dem Computer, der die SQL Server-Instanz beherbergt, bestätigen könnte.²

Für solche Fälle ist die so genannte *Gemischter Modus*-Authentifizierung angezeigt. Bei dieser Art der Authentifizierung können Sie sowohl die integrierte Sicherheit von Windows als auch die Authentifizierungsmethode verwenden, die vom SQL Server-System selbst zur Verfügung gestellt wird. Der Nachteil: Letzterer besteht in geringeren Sicherheitsstandards. Um eine Verbindung auf diese Weise zu einem SQL Server aufzubauen, müssen Kennwort und Benutzername beim Verbindungsauflauf angegeben werden. Die Wahrscheinlichkeit, dass Anmeldedaten dadurch kompromittiert werden könnten, ist daher ungleich höher, wenn sie nicht auf geschickte Weise in Ihrer Software verschlüsselt werden.

² Jedenfalls wenn Sie als Administrator am Rechner angemeldet sind – was die meisten Entwickler ja leider häufig sind. Als »normaler« Benutzer muss man auch erst einmal gezielt Zugriff vom SQL Server erteilt bekommen... CREATE LOGIN bzw. CREATE USER FROM LOGIN lauten hier die Stichworte, die genauer zu beschreiben den Rahmen an dieser Stelle allerdings sprengen würden.

Voraussetzungen für die Beispiele dieses und des nächsten Kapitels

Dieses Kapitel nennt sich LINQ to SQL.³ Und mit SQL im Part dieses Namens ist nicht etwa irgendein SQL gemeint, sondern SQL in Form von Microsoft SQL Server. Mit diesem LINQ-Provider können Sie nur diesen SQL Server bedienen; andere SQL-Server wie Oracle, DB2 oder MySQL bleiben außen vor. Sollten Sie also mit LINQ auch auf andere Server-Systeme zugreifen müssen, dann ist LINQ to SQL nichts für Sie. In diesem Fall sollten Sie sich mit dem nächsten Kapitel auseinander setzen, mit LINQ to Entities.⁴

HINWEIS LINQ to Entities ist erst mit dem Service Pack 1 von Visual Studio 2008 bzw. mit dem Service Pack 1 zum .NET Framework 3.5 verfügbar. Anhand der folgenden Abbildung können Sie erkennen, ob wenigstens Service Pack 1 bereits für Ihr Visual Studio installiert wurde. Falls das nicht der Fall ist, können Sie sich das DVD-ISO-Image von Visual Studio 2008 unter dem IntelliLink **A0104** herunterladen.



Abbildung 34.1 Um festzustellen, ob das Service Pack 1 installiert ist, wählen Sie aus dem Menü Hilfe von Visual Studio den Eintrag Info. Überprüfen Sie Ihr Visual Studio, deren Servicepacks wie hier im Dialog ausgewiesen sein müssen.

³ Übrigens: Tatsächlich sollte es »Link tuh Ess Kju Äll« ausgesprochen werden, aber genau wie bei Microsofts SQL Server, den (fast) jeder »βiekwäl Sörwä« ausspricht, nennen es die meisten »Link to βiekwäl«.

⁴ Wobei, auch dazu muss zunächst etwas Einschränkendes gesagt werden: Zur Drucklegung des Buchs gibt es nämlich weder von Microsoft noch von den Datenbankherstellern selbst konkrete Implementierungen als Entity Client Data Provider. Das Einzige, was derzeit bekannt und halbwegs brauchbar ist, war zum Zeitpunkt, zu dem diese Zeilen entstehen, eine Treibersammlung für LINQ to Entities von einem Dritthersteller, den Sie unter dem IntelliLink **F3404** erreichen können. Dieser enthält auch eine getestete Oracle-Unterstützung für Oracle XE. Von Microsoft selber gibt es einen Sample Entity Framework Provider für Oracle unter dem **F3405**, der die Oracle 10g Express Edition unterstützt, die Sie unter dem IntelliLink **F3406** herunterladen können.

Download und Installation von *SQL Server 2008 Express Edition with Tools*

Fassen wir noch mal zusammen: Um die Beispiele dieses Kapitels und des nächsten Kapitels nachvollziehen zu können, benötigen Sie eine funktionsfähige SQL Server-Instanz – am besten in der Version 2008, mindestens aber in der Version 2005. SQL Server 2008 in der Express Edition ist dazu ausreichend, diese kostet nichts, und reicht für viele Datenbank-Anwendungen ohnehin aus. Sie können Sie sogar mit ihrer Anwendung, die Sie vielleicht auf Basis von LINQ to SQL entwickeln, weiter vertreiben.

- *SQL Server 2008 Express Edition with Tools* können Sie über den IntelliLink **F3407** herunterladen.

ACHTUNG Je nach Aktualität und bereits installierten Komponenten auf Ihrem Computer, benötigen Sie ferner folgende Komponenten, die Sie herunterladen und installieren müssen, *bevor* Sie SQL Server 2008 Express Edition installieren.

Stellen Sie sicher, dass Sie das .NET Framework 3.5 mit Service Pack 1 installiert haben. Der IntelliLink dazu lautet: **F3408**. Wenn Sie Visual Studio 2008 auf dem Zielrechner bereits installiert haben, brauchen Sie diese Komponente nicht erneut zu installieren (vorausgesetzt, Visual Studio *inklusive Service Pack 1* ist auf dem Zielrechner vorhanden).

Laden Sie unter dem IntelliLink **F3409** den *Microsoft Installer 4.5* herunter und installieren Sie diesen. Beim Herausfinden des richtigen Dateinamens ist ein wenig Puzzlelei angesagt: Für Windows Vista und Windows Server 2008 laden Sie bitte die für Ihre Prozessorarchitektur relevante Version herunter, die mit *Windows 6.0* beginnt; Windows XP und Windows Server 2003-Versionen sind entsprechend abweichend gekennzeichnet (bitte auch auf die Prozessorarchitektur achten! *x64* für 64-Bit-Betriebssysteme, *x86* für 32-Bit-Betriebssysteme verwenden). Für Vista 64-Bit würden Sie also die Datei *Windows6.0-KB942288-v2-x64.msu* herunterladen.

Laden Sie unter dem IntelliLink **F3410** die PowerShell-Installation herunter, und installieren Sie diese Komponente ebenfalls. Das ist erforderlich, selbst wenn Sie Visual Studio 2008 zuvor installiert haben. Für Windows Server 2008 müssen Sie diesen Download nicht durchführen, da PowerShell hier bereits im System enthalten ist. Für Windows Vista laden Sie vom angegebenen Link die für Sie zutreffende Version, die unter *Windows Vista RTM* angegeben ist (*x86* für 32-Bit, *x64* für 64-Bit Betriebssystemversion). Für Windows XP und Windows 2003 die jeweils unter der gleichnamigen Kategorie für Ihre Busbreite aufgeführte Version. Denken Sie auch daran, die für Sie wichtige Sprachversion herunter zu laden.

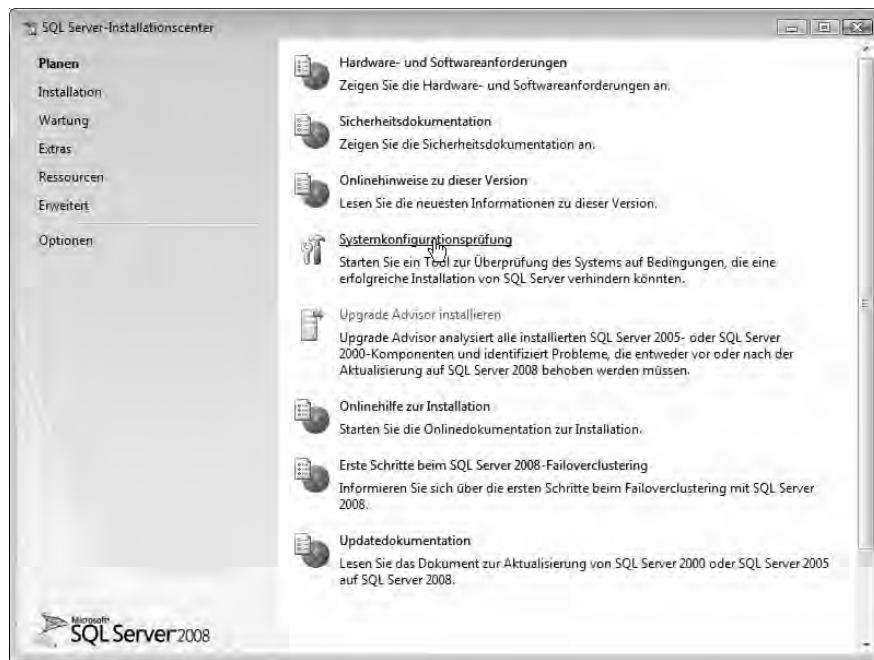


Abbildung 34.2 Nach dem Start der SQL-Server-Express-Installationsdatei gelangen Sie zunächst in das SQL Server-Installationscenter. Führen Sie erst eine *Systemkonfigurationsprüfung* durch bevor Sie die eigentliche Installation starten.

Haben Sie alle erforderlichen Komponenten installiert, können Sie sich an die Installation von *SQL Server 2008 Express Edition with Tools* machen.

1. Dazu starten Sie die heruntergeladene Installationsdatei, und Sie sollten anschließend den Dialog vorfinden, den Sie auch in Abbildung 34.2 erkennen können.
2. Klicken Sie anschließend auf den Link *Systemkonfigurationsprüfung*, um abzuklären, ob alle Installationsvoraussetzungen, die Sie in der vorherigen Liste beschrieben finden, auf dem Zielsystem erfüllt sind. Bessern Sie im Bedarfsfall die fehlenden Komponenten nach.
3. Klicken Sie anschließend im linken Bereich des Dialogs aus Abbildung 34.3 auf *Installation*. Der Inhalt des Dialogs ändert sich, etwa wie in Abbildung 34.4 zu sehen.

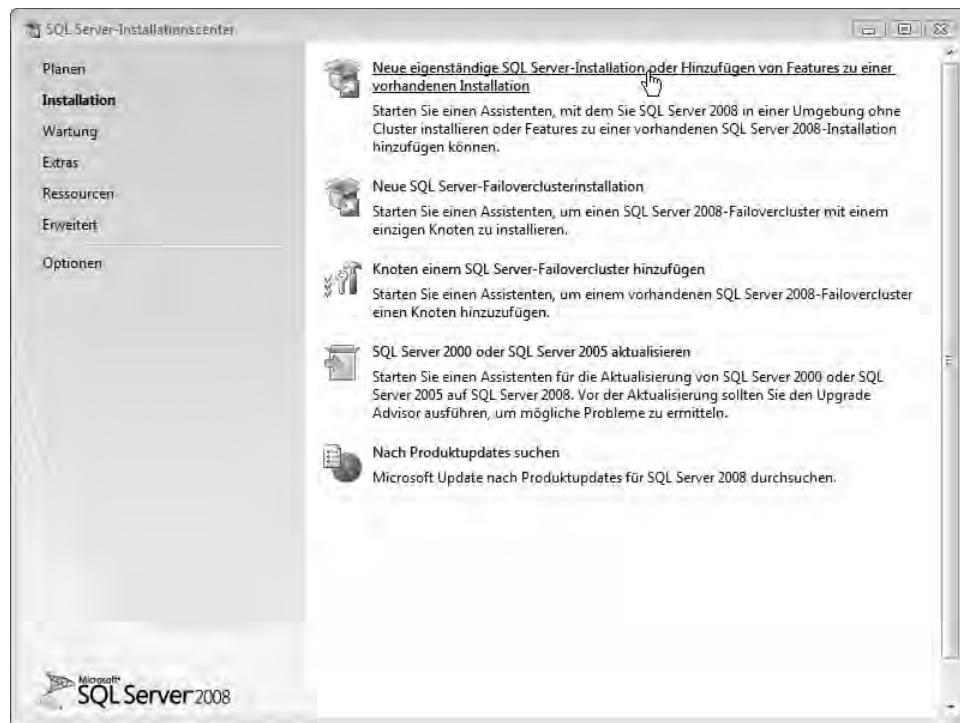


Abbildung 34.3 Klicken Sie im linken Bereich auf Installieren und anschließend im rechten Bereich des Dialogs auf den oberen Link, um die Installation der Express-Edition von SQL Server 2008 zu starten

4. Klicken Sie hier auf *Neue eigenständige SQL Server-Installation oder Hinzufügen von Features zu einer vorhandenen Installation*, um die eigentliche Installation der Express-Edition zu starten.
5. Die Installation startet jetzt die Überprüfung der Setup-Unterstützungsregeln, die keine Fehler mehr ergeben sollte. Verlassen Sie den Dialog mit *OK*.
6. Klicken im SQL Server 2008-Setup-Assistenten, der jetzt erscheint (und sich vielleicht unter dem Ausgangsdialog befindet; holen Sie den Dialog im Bedarfsfall in den Vordergrund) auf *Weiter*, akzeptieren Sie die Lizenzbedingungen im nächsten Schritt und klicken Sie abermals auf *Weiter*.
7. Im Schritt *Setup-Unterstützungdateien*, der jetzt wie in Abbildung 34.4 erscheinen sollte, klicken Sie auf *Installieren*. Dieser Vorgang kann einige Minuten in Anspruch nehmen.

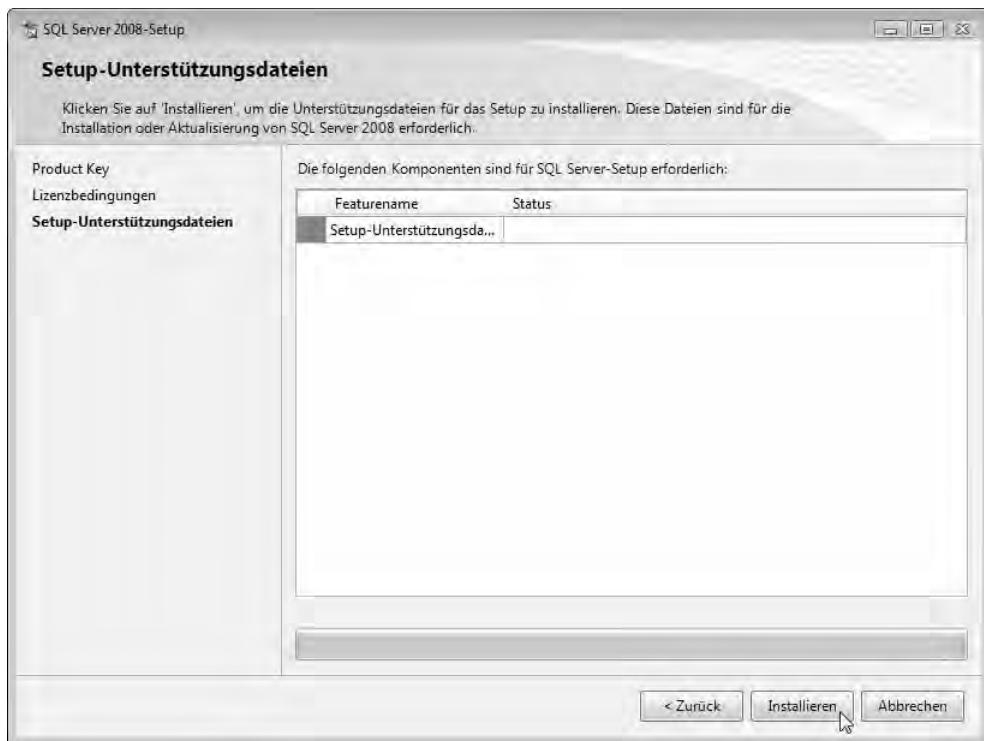


Abbildung 34.4 Klicken Sie im Schritt Setup-Unterstützungsdateien auf *Installieren*, um die Setup-Unterstützungsdateien zu starten

8. Sie sollten bei der anschließenden Zusammenfassung lediglich eine Windows-Firewall-Warnung erhalten. Diese Warnung können Sie ignorieren, solange Sie Ihre SQL Server-Datenbanken nur lokal, also von Ihrem Computer aus verwenden. Erst wenn Sie auf die SQL Server-Instanzen von einem anderen Rechner aus über das Netzwerk zugreifen wollen, müssen Sie die entsprechenden Ports der Windows-Firewall öffnen (SQL Server-Standard: 1433). Klicken Sie daher auf *Weiter*, um zum nächsten Schritt zu gelangen.
9. Sie gelangen anschließend zur Featureauswahl (siehe Abbildung 34.34). Hier bestimmen Sie, welche Features im Rahmen der Installation mitinstalliert werden sollen. Beziehen Sie die *Datenbankmoduldienste* und die *Verwaltungstools* in jedem Fall mit in die Installation ein. Die *SQL Server-Replikation* können Sie optional installieren, wenn Sie Inhalte der Datenbank zu Replikationszwecken regelmäßig in andere Datenbanken übertragen wollen.

HINWEIS Sollten Sie sich für die Installation für SQL Server 2008 Express Edition *with Advanced Services* entschieden haben, stellen Sie an dieser Stelle sicher, dass Sie die Funktionalität der Volltextsuche mit in die Liste der zu installierenden Features mit einbeziehen!

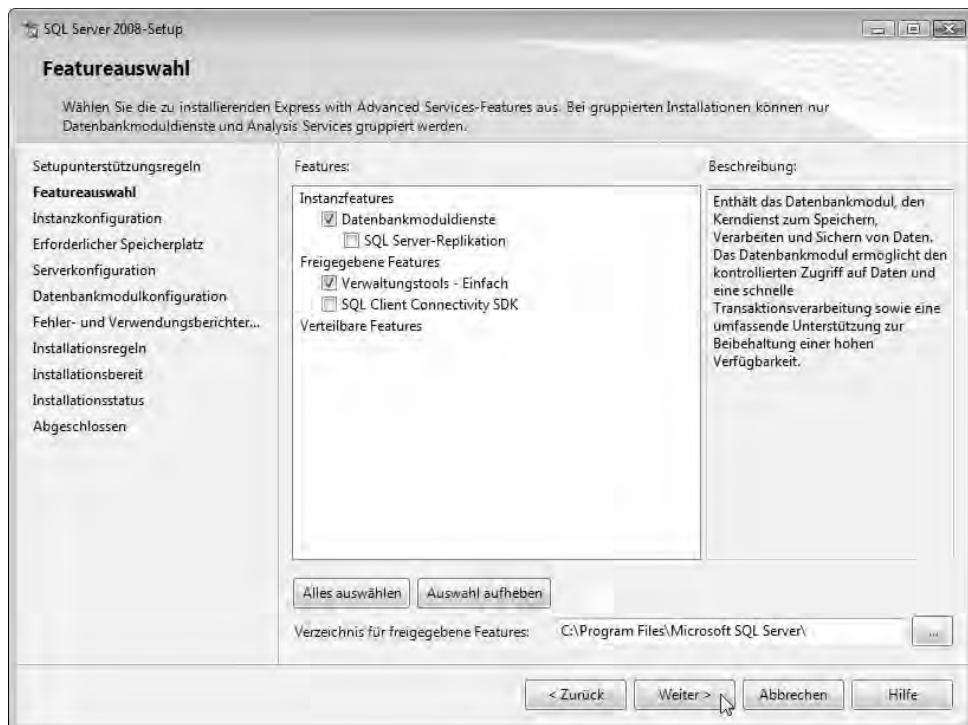


Abbildung 34.5 In diesem Schritt wählen Sie die Komponenten aus, die Setup mitinstallieren soll. Die hier markierten müssen Sie in jedem Fall auswählen.

10. Klicken Sie anschließend auf *Weiter*.

11. Bestimmen Sie im nächsten Dialog den Namen der SQL Server-Instanz, die Ihre Datenbanken später verwalten soll. Sie können die Voreinstellung *Benannte Instanz: SQLEXPRESS* belassen wie sie ist, wenn es nicht bereits eine SQL Server Instanz unter diesem Namen auf demselben Rechner gibt; dies wäre z.B. dann der Fall, wenn Sie mit Visual Studio auch eine Version des SQL Server 2005 Express Edition installiert haben. Geben Sie dann einfach einen anderen, beliebigen Namen ein.

TIPP Falls Sie sich mit einer »alten« Anwendung, die beispielsweise ODBC verwendet, gegen den SQL Server verbinden wollen, kann die Benutzung einer benannten Instanz Probleme bereiten. Benutzen Sie dann die Standardinstanz. Falls Sie aber ausschließlich mit .NET und ADO.NET oder LINQ (to SQL, to Entities) arbeiten, bieten die benannten Instanzen eher nur Vorteile. Sie dienen dazu, Datenbanken so voneinander zu isolieren, dass sie sich nicht gegenseitig ins Gehege kommen – in etwa so, als wenn Sie eine Software wie Word oder Excel mehrfach auf Ihrem Computer installieren würden, und für jede Installation (Instanz) komplett unabhängige Grundeinstellungen vornehmen können. So könnte man beispielsweise alle Datenbanken zum Verkauf in der Instanz »Verkauf« unterbringen usw. Solche Instanzen arbeiten dann wie unabhängige SQL Server und können einzeln gestartet und gestoppt werden, und sie verfügen über eine eigene Benutzerverwaltung, etc.

Produktiv-Datenbanken lassen sich, als weiteres Beispiel, so von Entwicklungsdatenbanken samt kompletter Umgebung voneinander trennen und kommen sich nicht ins Gehege. Die voreingestellte Instanz *SQLExpress* können Sie an dieser Stelle so belassen, wie sie ist. Geben Sie dem Kind allerdings einen deutlich anderen Namen, falls es bereits eine Express-Edition mit diesem Instanznamen auf dem System gibt oder sie eine weitere Express-Instanz installieren wollen (zum Beispiel *SQL2008Express*).

-
12. Klicken Sie anschließend auf *Weiter*.
 13. Der nächste Schritt zeigt Ihnen eine Zusammenfassung der Installationsparameter. Wenn Sie mit diesen einverstanden sind, bestätigen Sie auch diesen Dialog mit *Weiter*.
 14. Damit Sie die SQL Server-Instanz später beispielsweise mit dem Verbindungsdialog des Server-Explorers von Visual Studio finden können, stellen Sie, wie in der folgenden Abbildung zu sehen, den SQL Browser-Dienst auf *Automatisch*. Sie müssen in diesem Dialog ferner die Systemkonten bestimmen, in deren Kontext sowohl die Datenbank-Engine als auch der Browser-Dienst laufen sollen.

TIPP Für den professionellen Einsatz von SQL Server sollten Sie Folgendes beherzigen: Das Systemkonto *Netzwerkdienst* ist das Konto mit den geringsten Rechten. Damit wird es böswilligen Angreifern erheblich erschwert, über den SQL Server Zugriff auf Ihr System zu erlangen. Alternativ können Sie hier auch das Konto *System* bestimmen oder – unter Zuhilfenahme des Eintrags *Durchsuchen* – auch das Konto *Lokales System* ausfindig machen. Das Konto *Netzwerkdienst* ist aber in Produktivumgebungen beiden Konten auf jeden Fall vorzuziehen.

Für weitergehende Möglichkeiten sollten Sie jedoch ein eigenes, spezielles »Dienstkonto« einrichten und dann hier im Setup bestimmen. Man könnte auf die Idee kommen, dass auch ein späterer Wechsel auf ein Dienstkonto z.B. über die Systemsteuerung und dem Eintrag »Dienste« möglich wäre. Dies ist auf den ersten Blick auch richtig; denken Sie allerdings daran: das Kontextkonto für SQL Server benötigt aber vom Start weg für bestimmte Aufgaben sehr »spezielle« Rechte, wie zum Beispiel für das Ausführen von Festplattenwartungen (nur dann etwa funktioniert das neue Feature der schnellen Dateiinitialisierung).

Für Standardaufgaben und zum Entwickeln in Nicht-Produktiv-Systemen sind die standardmäßig angebotenen Konten aber in jedem Falle ausreichend.

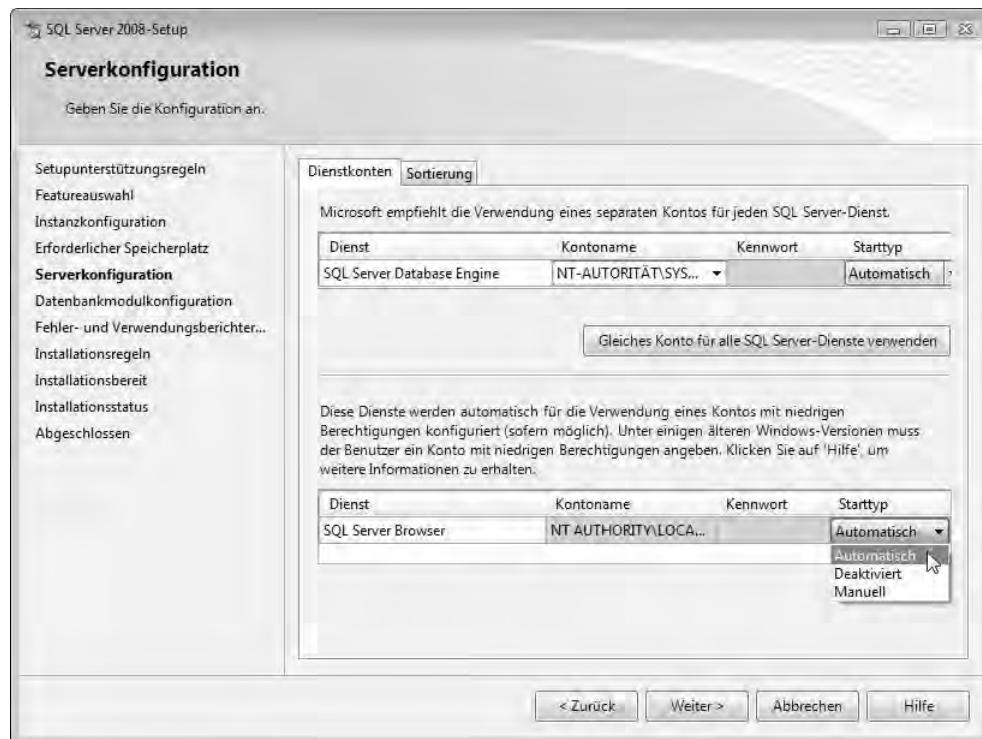


Abbildung 34.6 Bestimmen Sie hier die Konten in deren Kontext Datenbank-Engine und Browser-Dienst laufen sollen

15. Im nächsten Dialog (Abbildung 34.34) bestimmen Sie den Authentifizierungsmodus, mit dem sich Clients an der Instanz und an von ihr verwalteten Datenbanken anmelden können. Wenn Ihre Anwendung ausschließlich auf der gleichen Maschine läuft, die auch die SQL Server-Instanz enthält oder SQL Server auf einer Maschine läuft, die Teil einer Active Directory Domäne ist, wählen Sie die sichere Authentifizierungsmethode *Windows-Authentifizierungsmodus*. Ist keine Domäne vorhanden, müssen Sie zur Anmeldung an einer SQL Server Instanz Anmeldeinformationen übergeben; das funktioniert nur, wenn die Instanz zuvor in den *gemischten Modus* geschaltet wurde. In diesem Fall bestimmen Sie ebenfalls ein Systemadministrator-Kennwort (»sa«), mit dem Sie sich später an der Instanz anmelden können.

HINWEIS Ein sehr bekannter Virus (SQL Slammer) benutzte beim SQL Server 2000 die menschlich verständliche Schwäche, dass Benutzer damals zumeist KEIN Kennwort für das sa-Konto angaben; schon deshalb warnt das Setup hier deutlich. Vergeben Sie daher auf jeden Fall ein sicheres Kennwort (mind. 6 Zeichen, Groß- und Kleinschreibung, Sonderzeichen und Zahlen, kein Wort der deutschen Sprache, also beispielsweise im Stil von »P@a\$\$w0rd«)

16. Klicken Sie anschließend auf *Weiter*.

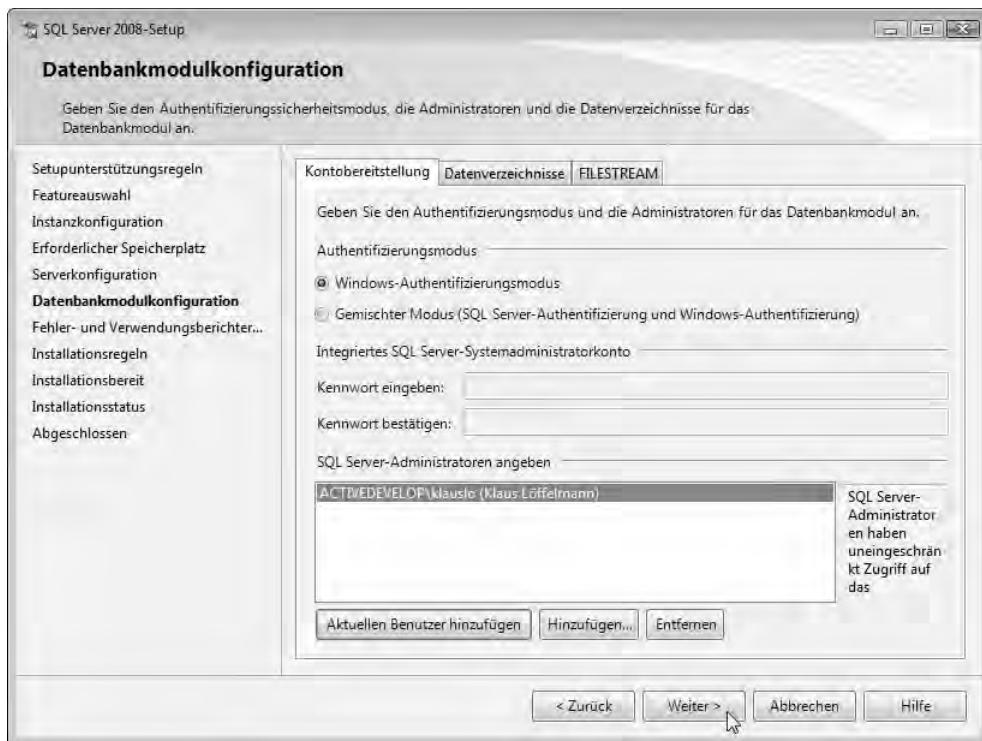


Abbildung 34.7 Bestimmen Sie in diesem Dialog den Authentifizierungsmodus und fügen Sie der Datenbankinstanz mindestens einen Benutzer als SQL Server-Administrator hinzu

17. Bestätigen Sie den nächsten Schritte des Setup-Assistenten mit *Weiter*, nachdem Sie den Dialog nach Belieben eingestellt haben – die Optionen, die sich Ihnen hier bieten, sind durch ausreichende Hilfstexte beschrieben.
18. Sie sollten anschließend die Installationsregeln überprüft und gecheckt vorfinden, etwa wie in der folgenden Abbildung zu sehen. Danach sind die Datenbankdienste zur Installation bereit, und Sie können deren Installation beginnen, indem Sie abermals auf *Weiter* klicken ...

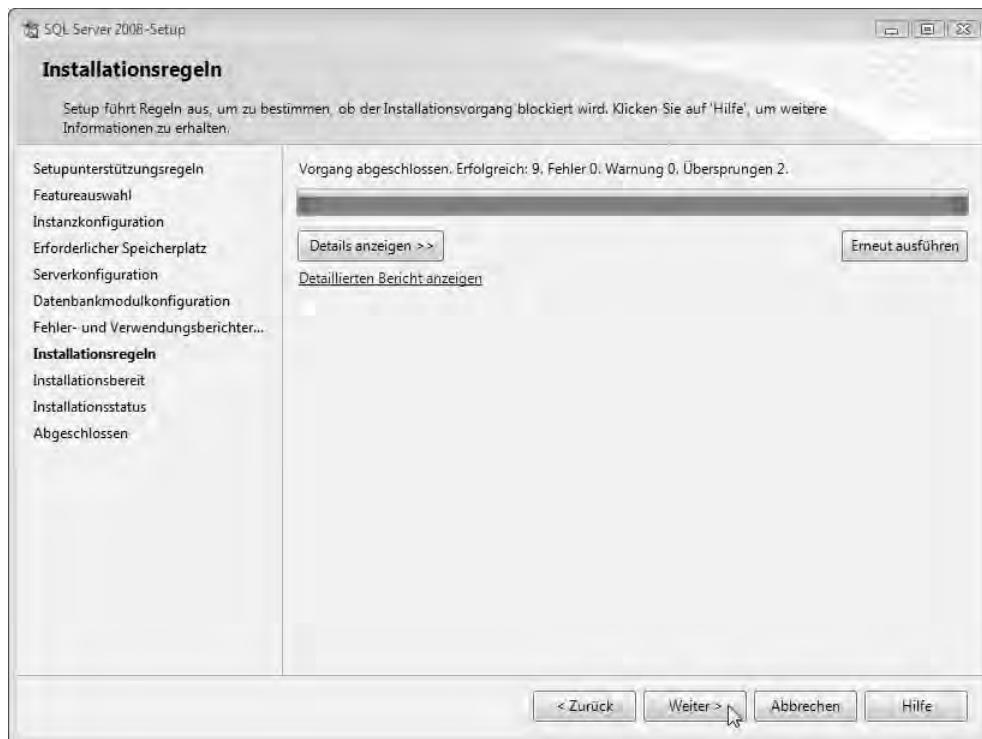


Abbildung 34.8 Die Installationsregeln sollten hier erfolgreich bestätigt worden sein; die Datenbankdienste sind nun zur Installation bereit

19. ... nicht allerdings, bevor Sie sich im letzten Dialog die Zusammenfassung anschauen und mit einem beherzten Klick auf *Installieren* bestätigen müssen.

Die Installation dauert nun eine ganze Weile. Nach dem Abschluss der Installation werden Sie in den meisten Fällen aufgefordert, Ihren Rechner durchzustarten. Machen Sie das, nachdem Sie zunächst den Dialog mit der Meldung *Die Installation ist abgeschlossen* mit *Weiter* bestätigt haben, und im Abschlussdialog das entsprechende Meldungsfeld mit *OK* bestätigen (Abbildung 34.34). Der Rechner wird nicht automatisch neu gestartet, Sie müssen das Durchbooten nur selbst anstoßen, da das durch den Abschluss der Installation nicht automatisch initiiert wird.

TIPP Sollten Sie gerade das Kapitel eines Buchs schreiben, sorgen Sie dafür, dass das Kapitel abgespeichert ist, bevor Sie den Rechner neu durchbooten. Andernfalls müssen Sie erfahrungsgemäß gegebenenfalls einige Absätze, die Sie bereits geschrieben haben, noch einmal schreiben. Das gilt natürlich auch für ähnliche andere Arbeiten, die Sie gerade durchführen, und die Sie in irgendeiner Form erhaltenswert finden.



Abbildung 34.9 Es ist vollbracht! Nach dem Durchstarten des Computers können Sie SQL Server 2008 Express with Tools verwenden

Installation der Beispieldatenbanken

Darüber hinaus brauchen Sie für die Beispiele dieses Kapitels auch die Beispieldatenbank *AdventureWorks*. Diese wird bei *codeplex* gehostet, und sie ist sowohl als 2005er als auch als 2008er Version verfügbar.

HINWEIS Nach derzeitigem Stand können Sie die dedizierten 2008er *AdventureWorks*-Beispiele nur auf dem großen SQL Server 2008 bzw. auf SQL Server 2008 with Advanced Services installieren, da nur in diesen Versionen die zur Installation notwendige Volltextsuche funktionell implementiert ist.

- **Beispieldatenbanken für Sql Server 2005 (auch für 2008):** Sie finden die Beispieldateien von *AdventureWorks*, die sich auch für SQL Server 2008 verwenden lassen, unter dem IntelliLink **F3411**. Verwenden Sie dort das zu oberst stehende MSI-Installerpaket *AdventureWorksDB.msi*. Diese Beispieldatenbank werden wir auch im Folgenden verwenden.
- **Beispieldatenbanken für SQL Server 2008 (für die Beispiele in diesem Buch):** Die Sample-Datenbanken für SQL Server 2008 finden Sie unter dem IntelliLink **F3412**. Sie können, wie schon erwähnt, diese nur dann installieren, wenn Sie einen der großen SQL Server-Versionen verwenden, oder SQL Server 2008 with Advanced Services und die Komponente Volltextsuche mitinstalliert haben.

Wenn Sie bereits mit Microsoft SQL Server-Datenbanksystemen gearbeitet haben, dann wissen Sie, dass jede SQL Server-Datenbank fest an eine SQL Server-Instanz gebunden wird. Die Fachbegriffe hierfür lauten »Attachen«⁵ oder »Anfügen« einer Datenbankdatei an eine Instanz bzw. »Detach« oder »Trennen« einer Datenbankdatei. Während Sie beispielsweise in Access eine Datenbank einfach öffnen, bearbeiten und wieder schließen können, hält der SQL Server-Datenbankdienst einen ständigen Zugriff auf die Datenbank, auch, wenn sie von keinem Client aus geöffnet ist.

Das ist für die Beispieldateien, die Sie heruntergeladen und deren Installer Sie haben laufen lassen, ebenfalls wichtig zu wissen. Denn das alleinige Installieren ist nur die halbe Miete: Sie müssen die Datenbanken nach der Installation eben noch an die Instanz binden, sie also »attachen« (anfügen). Wie das funktioniert, zeigt die folgende Schritt-für-Schritt-Anleitung.

1. Wenn Sie die Installationsdateien der Beispieldatenbank AdventureWorks heruntergeladen haben, starten Sie das Installerpaket durch Doppelklick.



Abbildung 34.10 Nach dem Start des Installerpaketes begrüßt Sie das Setup der Beispieldatenbanken »AdventureWorks« mit diesem Dialog

2. Klicken Sie auf *Next* und bestätigen Sie im nächsten Schritt die Lizenzbedingungen. Klicken Sie anschließend abermals auf *Next*.

⁵ Sprich: »Ätätsch« bzw. »Ättätschen« als Anglizismus sowie »Diehtätsch« bzw. »Diehtätschen«.

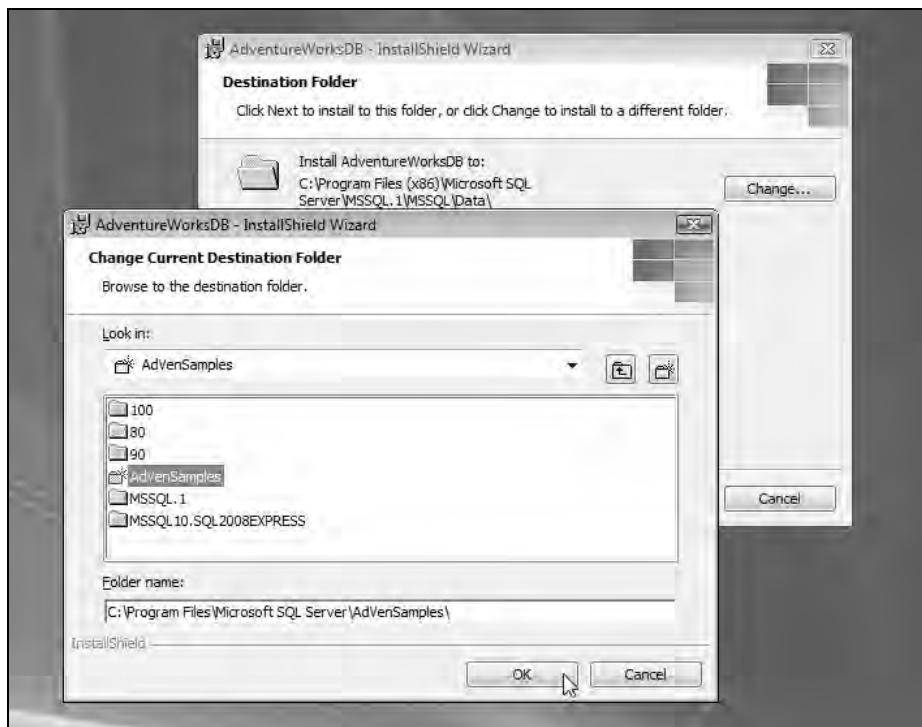


Abbildung 34.11 In diesem Schritt bestimmen Sie, in welcher Datenbankinstanz die Beispieldatenbanken attached werden sollen

3. Übernehmen Sie das anschließend vorgeschlagene Verzeichnis, oder legen Sie, wie hier in der Abbildung zu sehen, ein neues an. Klicken Sie dazu im Ausgangsdialog auf die Schaltfläche *Change...* (*Ändern*), bestimmten Sie ein neues Verzeichnis, und beenden Sie den Dialog mit *OK*.

TIPP Merken Sie sich das Installationsverzeichnis an dieser Stelle gut, damit Sie es im nächsten Schritt, beim Anfügen der Datenbankdateien an die SQL Server-Instanz, wiederfinden!

4. Klicken Sie im Ausgangsdialog auf *Next*.
5. Klicken Sie anschließend auf *Install*, um mit der eigentlichen Installation der Beispieldatenbanken zu beginnen, und beenden Sie den Installationsvorgang schließlich mit Mausklick auf die Schaltfläche *Finish*.

Anfügen (»Attachen«) der Beispieldatenbanken an die SQL Server-Instanz

Die Datenbankdateien aller AdventureWorks-Beispiele befinden sich anschließend im Verzeichnis, das Sie während der Installation angegeben haben. Ihre Aufgabe ist es nun nur noch, die für die folgenden Kapitel relevante Beispieldatenbankdatei an die Instanz des SQL Servers zu hängen (zu »attachen«). Und das geht folgendermaßen:

1. Starten Sie das Management Studio des SQL Server 2008, das ihm Rahmen der SQL Server 2008-Installation mitinstalliert worden ist.

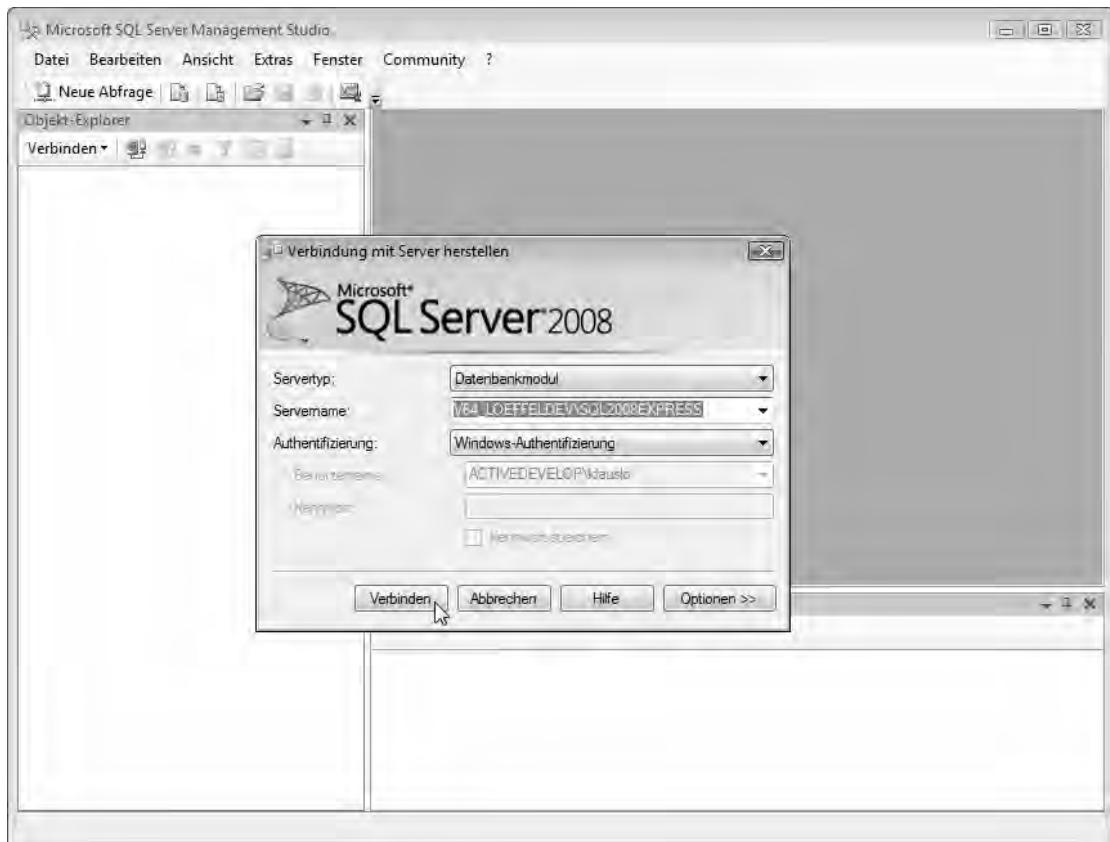


Abbildung 34.12 Mit dem Management-Studio verbinden Sie sich als erstes mit der entsprechenden SQL Server-Instanz, an der Sie die Beispieldatenbank AdventureWorks attachen (anfügen) möchten

2. Unter *Servername* wählen Sie die Instanz aus, mit der Sie sich verbinden möchten. Das kann im einfachsten Fall eine, nämlich die Instanz der lokalen Express-Edition sein; Sie können sich aber natürlich auch über das Netzwerk mit einer SQL Server-Instanz einer ganz anderen Maschine verbinden. Falls Sie den Netzwerknamen nicht kennen, können Sie die Klappliste auch öffnen und auf <Suche fortsetzen> klicken. In diesem Fall öffnet sich ein weiterer Dialog, der auch in Abbildung 34.13 zu sehen ist, mit dem Sie auf der Registerkarte *Netzwerkservicer* alle im lokalen Netzwerk erreichbaren SQL Server-Instanzen auflisten und auswählen können.

TIPP Anhand der Versionsnummer, die hinter dem jeweiligen Instanzeintrag steht, können Sie erkennen, um welche SQL Server-Version es sich handelt. 8.0 entspricht SQL Server 2000, 9.0 entspricht SQL Server 2005 und 10.0 einer SQL Server 2008-Instanz.

HINWEIS Denken Sie daran, dass dieser Dialog die Server auf dem Rechner nur dann findet, wenn der SQL Browser-Dienst auf den entsprechenden Rechnern installiert ist, und die Suche nicht durch eine etwaige installierte Firewall geblockt wird. Aber auch ohne SQL Browser können Sie sich natürlich mit der Instanz verbinden, Sie müssen dann Recher- und Instanzname durch den Backslash getrennt einfach manuell eingeben.

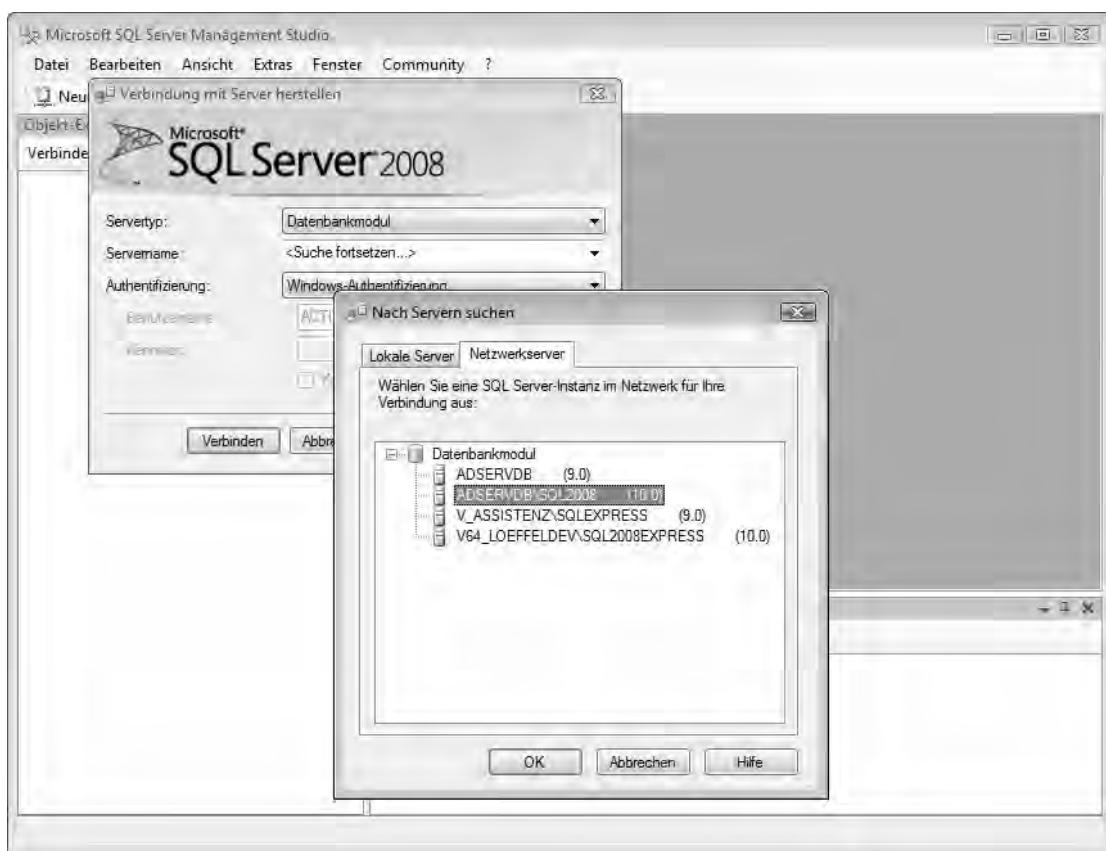


Abbildung 34.13 Mit *Suche fortsetzen* erreichen Sie diesen Dialog, mit dem Sie alle im lokalen Netzwerk erreichbaren SQL Server-Instanzen auflisten können

HINWEIS Die Verbindungszeichenfolgen zu einer Server-Instanz bestehen immer aus *Servername\Instanzname*. Sollten Sie auf einem Server eine unbenannte Instanz installiert haben (wie in der Abbildung in der Liste ganz oben zu sehen), wird nur der Servername (also der Name des Computers) angezeigt. Sie können natürlich diese Kombination auch manuell eingeben und müssen dazu nicht unbedingt diesen Dialog konsultieren.

Kleiner Tipp dabei: Anstelle des lokalen Computernamens (wie V64_LOEFFELDEV – der Name meines Arbeitsplatzrechners – im Beispieldialog), reicht es auch einen Punkt einzugeben. Die Zeichenfolge .SQL2008EXPRESS würde es also in diesem Fall ebenfalls tun.

3. Klicken Sie auf *Verbinden*, wenn Sie die entsprechende Instanz ausgewählt haben.
4. Wenn die Verbindung hergestellt worden ist, öffnen Sie den Zweig des Servers, etwa in wie in Abbildung 34.14 zu sehen.
5. Öffnen Sie das Kontextmenü und wählen Sie den Eintrag *Anfügen...*

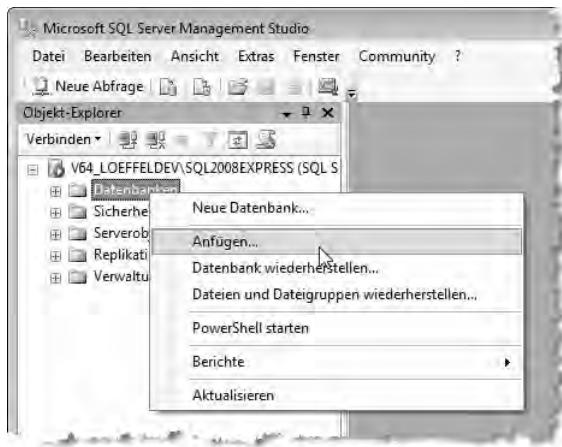


Abbildung 34.14 Auf dem Zweig *Datenbanken* öffnen Sie das Kontextmenü, um eine neue Datenbankdatei der Instanz anzufügen

6. Im Dialog, der jetzt erscheint, klicken Sie auf *Hinzufügen*. Wählen Sie aus der Verzeichnis- und Dateienliste den Speicherort, den Sie zuvor bei der Installation der Datenbankbeispieldateien festgelegt haben, und dort die Datei *AdventureWorks_Data.mdf*.

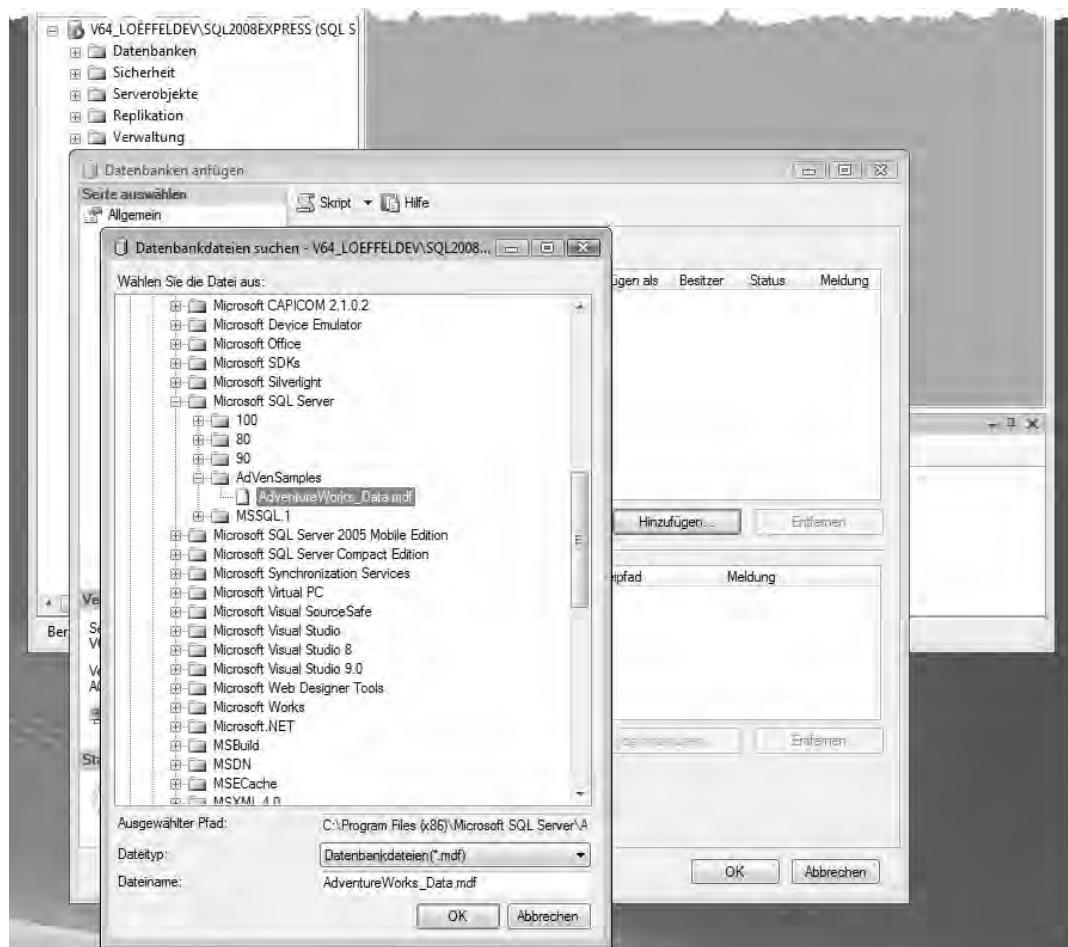


Abbildung 34.15 In diesem Dialog wählen Sie die AdventureWorks-Datenbankdatei, um sie an die SQL Server-Instanz anzuhängen

7. Verlassen Sie den aktuellen und den untergeordneten Dialog mit **OK**.
8. Meldungen über nicht ausgewählte Volltextkataloge können Sie in diesem Fall ignorieren, und auch diesen Meldungsdialog mit **OK** bestätigen.

Die AdventureWorks-Beispieldatenbank befindet sich nun in der Gewalt der SQL Server-Instanz. Wenn Sie sich nun beispielsweise Inhalte der Datei anschauen bzw. bearbeiten wollen, öffnen Sie den Zweig *Datenbanken, AdventureWorks, Tabellen* und öffnen beispielsweise für die Tabelle *Person.Contact* das Kontextmenü mit der rechten Maustaste. Wählen Sie hier den Eintrag *Oberste 200 Zeilen bearbeiten*.

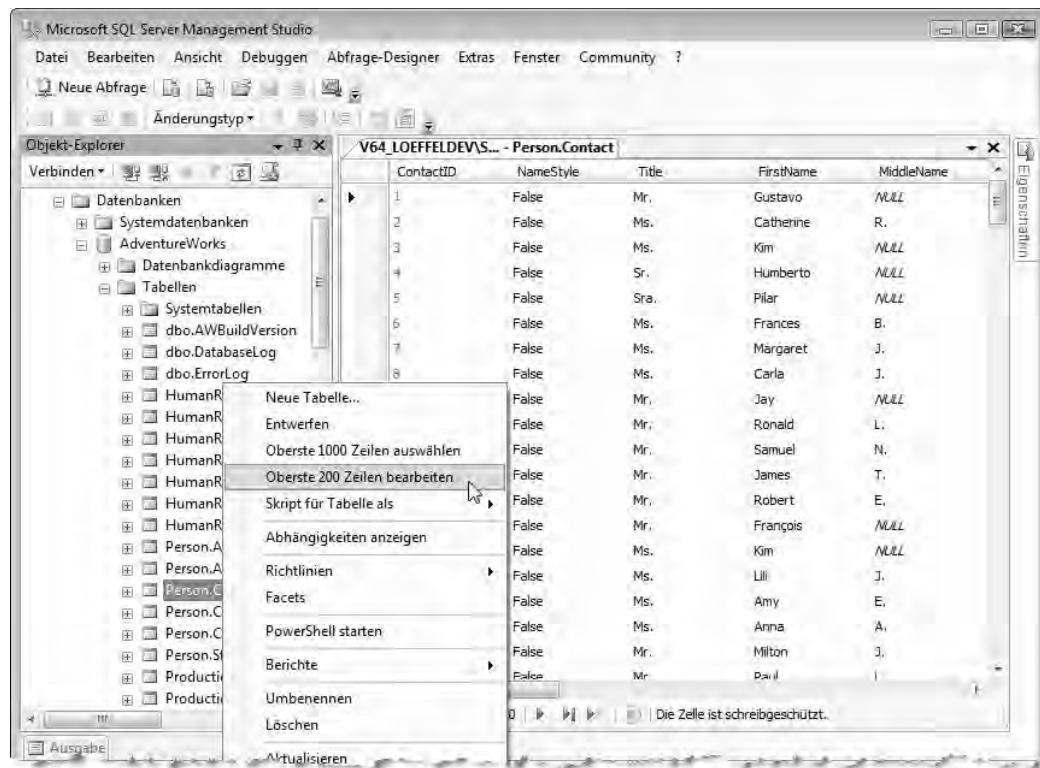


Abbildung 34.16 Management Studio erlaubt Ihnen auch das direkte Bearbeiten von Tabellendaten

Sie können, nachdem Sie sich so von der korrekten Funktionsweise der Datenbank überzeugt haben, das Management Studio einfach schließen – die Verbindung zur Datenbank wird dabei automatisch geschlossen. Die AdventureWorks-Datenbank steht Ihnen nun für die folgenden Beispiele zum Einsatz bereit.

TIPP Wenn Sie andere Datenbankdateien an der Serverinstanz anfügen wollen, verfahren Sie auf die gleiche Art und Weise, wie hier beschrieben. Wie schon gesagt, hat der SQL Server-Datenbank-Dienst anschließend exklusiven Zugriff auf die Datei, sodass diese sich beispielsweise nicht an einen anderen Ort verschieben oder kopieren lässt.

Möchten Sie eine Datenbankdatei anschließend von der Instanz wieder lösen, eben damit Sie sie beispielsweise verschieben oder kopieren können, verfahren Sie prinzipiell auf die gleiche Art und Weise, nur öffnen Sie in diesem Fall das Kontextmenü der Datenbank und wählen den Menüpunkt *Datenbank trennen*.

LINQ to SQL oder LINQ to Entities – was ist besser, was ist die Zukunft?

Nun wissen Sie auf alle Fälle schon einmal, wann Sie LINQ to SQL verwenden können, und wann nicht. Falls Sie sich allerdings für Microsofts SQL Server entscheiden, bedeutet das im Umkehrschluss nicht, dass Sie auch unbedingt LINQ to SQL verwenden müssen – es gibt nämlich eine ebenbürtige Alternative, LINQ

to Entities könnte für Sie ebenfalls in Frage kommen. LINQ to Entities hat einige entscheidende Vorteile gegenüber LINQ to SQL, allerdings auch einige Nachteile. Was die rein technischen Unterschiede zum Zeitpunkt .NET Framework 3.5 SP1 anbelangt, dazu soll Ihnen die Tabelle im Abschnitt »LINQ to SQL oder LINQ to Entities – was ist besser, was ist die Zukunft? « ab Seite 935 bei Ihrer Entscheidung helfen. Doch es gibt einen weiteren Entscheidungspunkt, der ungleich schwerer wiegen könnte.

Hat LINQ to SQL eine Zukunft?

Die Diskussion sollte – abgesehen von den Features, bei denen die eine LINQ-Technologie mitbringt und die andere eben nicht, oder welche der Features in der Schnittmenge beider Technologien einfach besser sind – auch in Richtung Zukunftsorientiertheit gehen.

Und um eine mögliche Aussage dazu vorwegzunehmen, und diese auch noch politisch klug zu formulieren (oder zumindest einen solchen Formulierungsversuch zu starten): Das Entwicklungsteam von ActiveDevelop⁶ stand jüngst ebenfalls vor der Entscheidung, sich für ein großes Softwareprojekt für eine der beiden Technologien entscheiden zu müssen. Die Wahl fiel auf LINQ to Entities, und sie wurde in erster Linie davon geprägt, dass sich, ...

- ... grundsätzlich erstmal alle wesentlichen Problemen in *beiden* Technologien lösen lassen und
- ... damit das Entscheidende ist, welche der beiden Technologien auf Dauer intensiver weiter gepflegt wird!

Es gibt einen schönen Artikel dazu im Blog vom ADO.NET Team (leider nur englischsprachig), den Sie unter dem IntelliLink **F3413** einsehen können. Nicht nur die Darstellung der Fakten durch das ADO.NET Team selbst war für ActiveDevelop dafür ausschlaggebend, die Entscheidung für LINQ to Entities zu treffen. Aber machen Sie sich selbst ein Bild. Tim Mallalieu, Program Manager LINQ to SQL und LINQ to Entities, entgegnet auf die Frage:

Hat LINQ to SQL eine Zukunft? – Tim Mallalieu, Program Manager LINQ to SQL/Entities

*Wir werden basierend auf Kunden-Feedback fortfahren, Einiges in LINQ to SQL zu investieren. Dieser Post (IntelliLink **F3413**) diente dazu, unsere Absichten für zukünftige Innovationen klar zu stellen und darzulegen, dass ab .NET 4.0 LINQ to Entities unsere Empfehlung für LINQ-zu-relationalen-Szenarien sein wird. Wie schon erwähnt, haben wir auf diese Entscheidung die letzten paar Monate hingearbeitet. Als wir diese Entscheidung trafen, erachteten wir es als erforderlich, die Entwicklergemeinde sofort darüber zu informieren. Wir wussten, dass das ein heftiges Echo seitens der Entwicklergemeinde hervorrufen würde, aber uns war es wichtig, unsere Absichten so transparent wie möglich und so früh wie möglich verfügbar zu machen. Wir möchten, dass diese Information die Entwickler erreicht, damit sie rechtzeitig wissen, wohin wir wollen, wenn sie ihre Entscheidungen für das Entwickeln ihrer zukünftigen .NET-Anwendungen treffen. [...].*

Doch an dieser Stelle will ich auf jeden Fall auch eine Lanze für LINQ to SQL brechen: Natürlich gilt auch im Rahmen von LINQ to SQL: Was einmal im Framework drin ist, wird aus der Version, in der es drin ist, auch nicht wieder herausgenommen. Es kann also auf keinen Fall passieren, dass Ihre LINQ to SQL-

⁶ Die Entwicklungsfirma des Autors: www.activedevelop.de.

Anwendungen irgendwann nicht mehr laufen.⁷ Die Frage ist eigentlich nicht mehr, welche der Technologien wird den Evolutionskampf überleben, denn die Feature-Gegenüberstellung in der nächsten Tabelle zeigt auch, dass die Schnittmenge vergleichsweise groß ist, und welche Technologie die andere vermutlich überleben wird, zeigt der Blog meiner Meinung nach deutlich. Doch zum derzeitigen Zeitpunkt ist auch klar, dass LINQ to SQL den Bedarf für die nächsten 10 Jahre für viele Ansprüche an ein Datenbank-Entwicklungssystem decken wird, zumal LINQ to Entities sicherlich noch verbesserungswürdig ist, gerade was die Designer-Unterstützung anbelangt. Die Performance von LINQ to SQL ist derzeitig der von LINQ to Entities auf jeden Fall überlegen.

Jetzt geraten Sie aber bitte nicht in Rage und schimpfen auf unkoordinierte Microsoft-Entwicklerteams, sondern betrachten Sie die Entstehungsgeschichte dieser Technologien im historischen Kontext. Dass es bei Microsoft überhaupt »so weit kommen« konnte, hat nämlich einen solchen Hintergrund:

Einige unter uns alten Hasen werden sich sicherlich noch ein, zwei bereits gescheiterte Versuche Microsofts erinnern können, einen O/R-Mapper zu früheren Zeiten der Entwicklergemeinde zur Verfügung zu stellen. Der erste war eine Technologie, die damals unter dem Code-Namen *Object Spaces* für Furore sorgte, und diese Technologie schaffte es immerhin bis in eine fröhe Whidbey-Version (der damalige Codename von Visual Studio 2005). Parallel gab es die Entwicklung von *Windows Future Storage*, besser bekannt unter seiner Abkürzung WinFs,⁸ dem sagenumwobenen und komplett neuartigen Windows-Dateisystem, das erstmalig in Windows Vista und Windows Server 2008 Einzug halten sollte. Auch dieses System kannte eine Plattform für das O/R-Mapping, und es ergab damals keinen Sinn, Object Spaces an WinFs vorbei zu entwickeln, und so wurde es kurzerhand eingestellt.

Das eigentliche LINQ wird aber logischerweise nicht erst seit Visual Basic 2005 entwickelt, sondern dessen Planung gibt es ebenfalls schon früher. Und dass eine LINQ to SQL-Implementierung nicht unbedingt nur als O/RM-Ersatz mit LINQ auszuliefern war, ist eigentlich ebenfalls eine logische Schlussfolgerung.

Nur schaffte es leider WinFs dann bekanntermaßen ebenfalls nicht, das Licht der Welt zu erblicken, aber viele Komponenten dieser Technologie wurden zu eigenständigen Plattformen ganz neuer Technologien – so beispielsweise auch der O/R-Mapper von WinFs, der die Basis der heutigen Entity-Technologie von LINQ to Entities wurde. Es gab also quasi LINQ to Entities und LINQ to SQL; beide Produkte sind in der Form also niemals direkt und nebeneinander auf dem Reißbrett entstanden. Microsoft wurde vielmehr von der eigenen Entwicklung überholt, und wie schnell solche Dinge geschehen können, wird jedem klar sein, der bereits größere Softwareproduktentwicklungen über eine längere Zeit mit großen Teams betreut hat. Mir selbst ist es ehrlicherweise sogar schon passiert, dass ich im Abstand von drei Jahren zwei komplett neue Bibliotheken für einen gleichen Zweck implementiert habe, und ich einige Geschehnisse dabei als Déjà-vu abgetan habe...

Man sollte also Microsoft nicht pauschal den Vorwurf machen, unkoordiniert zu arbeiten (auch wenn man das in einigen Fällen sicherlich mit Recht könnte), sondern immer einen derartig problematischen Entwicklungsverlauf aus der Vogelperspektive und mit ausreichend einbezogener Historie betrachten. Es ist keinesfalls so, dass es hier Teams gegeben hätte, die die beiden LINQ-Technologien unabsprachegemäß parallel entwickelt hätten, und »auf einmal« entdeckte jemand, dass es zwei Konkurrenzprodukte aus gleichem Hause gab. »So isses nun nich« – wie wir Westfalen sagen!

⁷ Na ja – besser niemals nie sagen! Visual Basic 6.0-Anwendungen werden, so wie es ausschaut, schließlich auch nicht mehr in der nächsten Version von Windows, Windows 7, laufen. Solange ein Betriebssystem aber das .NET Framework 3.5 oder 4.0 noch unterstützt, sind Sie auf der sicheren Seite – und das wird sicherlich die nächsten 12 wenn nicht 15 Jahre der Fall sein.

⁸ Einen ziemlich interessanten Artikel gibt es zu diesem Thema übrigens bei Wikipedia, leider nur in ihrer englischen Version (der deutsche unterschlägt viele Fakten; Stand 8.11.2008). Wen es interessiert: Der IntelliLink **F3414** liefert mehr Infos.

Zwischenfazit

Tatsache bleibt: Es gibt diese beiden konkurrierenden Technologien nun einmal, und die Wahrscheinlichkeit, dass LINQ to Entities das evolutionäre Rennen machen wird, ist nach derzeitigem Stand der Dinge einfach viel, viel größer. LINQ to SQL ist aber einfacher zu erlernen, hat zurzeit noch ein paar Vorteile, so zum Beispiel, dass es performantere Abfragen als LINQ to Entities erstellt, und *natürlich* können Sie es bedenkenlos für Datenbank-Anwendungen einsetzen, die Microsoft SQL Server (2000, 2005, 2008) als Datenplattform benötigen.

Entscheidungshilfe – Gegenüberstellung der wichtigsten Features von LINQ to SQL und LINQ to Entities

Trotz aller politischen Diskussion kann LINQ to SQL also dennoch die richtige Wahl für kleinere und mittelkomplexe Anwendungen sein, vor allen Dingen wenn Entwicklungen schnell passieren müssen. Und hier wird LINQ to SQL erstmal – und erstmal bedeutet aller Voraussicht nach bis .NET 4.0 – auch sicherlich die erste Wahl bleiben, denn natürlich gilt, dass keine Features aus einer vorhandenen Framework-Version wieder entfernt werden. Wir werden also sowieso im .NET Framework 3.5 SP1 und auch im nächsten .NET Framework 4.0 noch in den Genuss einer gepflegten LINQ to SQL-Version kommen. Deswegen macht es durchaus Sinn, der folgenden Feature-Gegenüberstellung ein paar Minuten Denkkraft zu opfern.

HINWEIS Und denken Sie daran: Diese Liste ist sowieso nur dann für Sie wichtig, wenn Sie sich ohnehin für SQL Server von *Microsoft* als ausschließliche Datenbankplattform entschieden haben. Andere Datenbankserver können Sie zurzeit nur mit LINQ to Entities bedienen, aber Achtung! – Auch hier gilt das in der 2. Fußnote auf Seite 918 gesagte.

LINQ to SQL	LINQ to Entities
<p>Grundsätzlich gilt: Beide Technologien erlauben, dass Datenbankschemata für Tabellen und Sichten in Form von Klassen in Ihrem .NET-Projekt gemappt werden. Eine Tabellendatenzeile entspricht dabei vereinfacht gesprochen einer Instanz einer Klasse, wobei die Spalten der Tabelle (natürlich auch einer Sicht) den Eigenschaften dieser Klasse entsprechen.⁹ Abfragen passieren bei beiden Verfahren über LINQ, und beide Methoden stellen eine übergeordnete Verwaltungsinstanz zur Verfügung (<code>DataContext</code> bei LINQ to SQL, <code>ObjectContext</code> bei LINQ to Entities), über die die Tabellen und Sichtdefinitionen einerseits in Form von generischen Auflistungen zur Verfügung gestellt werden, die andererseits die Überwachung von Aktualisierungsbedarf geänderter/gelöschter/hinzugekommener Daten übernimmt. Dazu gehören natürlich auch das Herstellen der Datenbankverbindungen, das Befüllen der Businessobjektauflistungen, das Protokollieren der generierten SQL-Befehle aus den LINQ- oder Aktualisierungskonstrukten sowie das Bemerken und Auflösen von Concurrency-Exceptions (Behandeln von Änderungskonflikten).</p> <p>Dabei sind beide Technologien in der Lage, Tabellen <i>und</i> Sichten (Views) von Datenbanken im Objektmodell zu berücksichtigen, und auch Gespeicherte Prozeduren (Stored Procedures) lassen sich entweder zur Abfrage von Daten oder auch für spezielle Aktualisierungslogiken nutzen.</p>	
Vorteil: Vergleichsweise einfach zu erlernen	Nachteil: Benötigtes Hintergrundwissen zum Nutzen aller Möglichkeiten ist ungleich größer als bei LINQ to SQL – was danach natürlich von Vorteil ist, weil Entwickler flexibler werden.
Nachteil: Datenbank → Objektmodell-Mappings werden überwiegend 1:1 abgebildet. Eine Tabellenstruktur (Tabellschema) in LINQ to SQL muss also einer Klasse entsprechen, die entsprechende Eigenschaften aufweist, um mit dem Spaltenschema einer Tabelle kompatibel zu sein.	Vorteil: Durch Verwendung des konzeptionellen Datenmodells erlaubt es vereinfacht ausgedrückt einen Zwischenlayer, Relationen zwischen Tabellen und Business-Objekten eben nicht nur 1:1 abzubilden. (Mehr Infos dazu gibt es im nächsten Kapitel) – damit ist es schemaunabhängig. ►

⁹ Das ist standardmäßig bei LINQ to Entities so, muss aber bei LINQ to Entities nicht so sein, wie dann die spezifischen Gegenüberstellungen auch zeigen.

LINQ to SQL	LINQ to Entities
Nachteil: Auf Microsoft SQL Server beschränkt	Vorteil: Bietet die Möglichkeit, weitere auf dem ADO.NET 2.0-Treibermodell basierende Provider zu verwenden, und damit den Zugriffscode auf die darunterliegende Datenbankplattform zu »legalisieren« (Ein Code, unterschiedliche Datenbankplattformen). Es gilt aber zurzeit noch das in der Fußnote auf Seite 918 Gesagte.
Vor- und Nachteil: Sehr einfache Designmöglichkeiten, aber dafür schnell erlernbar	Vor- und Nachteil: Weit komplexere Designmöglichkeiten, wie beispielsweise Datentabellenvererbung und Assoziationen, die aber auch eine längere Einarbeitungszeit erfordern.
Nachteil: N:m-Verbindungen werden, wie in der Datenbank, nur durch Zwischentabellen realisiert	Vorteil: Auflösung von Zwischentabellen bei N:M-Verbindungen in zwei Objektlisten
Vorteil: Lazy-Loading und Eager-Loading möglich. Transparentes Lazy-Loading ist standardmäßig aktiviert.	Nachteil: Sowohl Lazy- als auch Eager-Loading sind möglich, allerdings ist beim Lazy-Loading nur manuell angestoßenes Nachladen von verknüpften Tabelle möglich.
Vorteil: Businessobjekte, die die Tabellenzeilen einer Datentabelle oder einer Sicht widerspiegeln, müssen nicht von einer Basisklasse abgeleitet werden, sondern werden nur über .NET-Attributklassen zugeordnet. POCOs (<i>Plain Old CLR Objects</i> , etwa: <i>gute alte CLR-Objekte</i>) als Zuordnung zu einem Tabellen- oder Sichtschema ist also möglich. Sie müssen allerdings die Schnittstellen <code>INotifyPropertyChanging</code> sowie <code>INotifyPropertyChanged</code> einbinden, damit Änderungen an Eigenschaften der Businessobjekte entsprechend nachvollziehbar sind.	Nachteil: Im derzeitigen Release-Stand müssen Business-Objekte von <code>EntityObject</code> abgeleitet werden, um sich in die Entity-Infrastruktur einzufügen. Reine POCO-Objekte können nicht direkt einem Tabellen- oder Sichtschema zugeordnet werden, weil POCO-Objekte – wie der Name schon sagt – eben nicht von <code>EntityObject</code> sondern gar nicht abgeleitet sind (außer von <code>Object</code> selbst natürlich, denn jede Klasse wird implizit von <code>Object</code> abgeleitet).
Vorteil: Forward Mapping ist möglich. Forward Mapping bedeutet, dass Sie ein Objektmodell entwerfen und sich daraus eine Datenbank erstellen lassen können. Reverse Mapping ist zwar die gängigere Methode, bei der also zunächst das Datenbankschema existiert, woraus das Objektmodell erstellt wird – in einigen Szenarien kann aber Forward Mapping durchaus Sinn machen.	Nachteil: Nur Reverse Mapping ist im derzeitigen Releasestand möglich. Sie können also ein Entity-Objektmodell nur aus einer vorhandenen Datenbank erstellen lassen, der umgekehrte Weg ist nicht möglich.
Nachteil: Businessobjekte sind zwar selbst serialisierbar, aber Auflistungen mit Verknüpfungen durch Foreign-Key-Relations leider nicht. Der Serialisierungssupport, wenn man Objektbäume also beispielsweise zwischenspeichern oder für Webservices serialisieren muss, ist also quasi nicht vorhanden.	Vorteil: Ganze Objektbäume, also Auflistungen mit den entsprechenden Business-Objekten, lassen sich serialisieren. Das gilt allerdings nicht für die »Verwalterklasse« <code>ObjectContext</code> , und auch die XML-Serialisierung funktioniert im gegenwärtigen Release-Stand nur eingeschränkt.
Nachteil: Die Delegation einer Abfrage über WebServices ist nicht möglich.	Vorteil: Die ADO.NET Data Services erlauben es, LINQ to Entities-Abfragen über einen Webservice auch auf einem Remote-Server ausführen zu lassen.

Tabelle 34.1 Vergleichstabelle als Entscheidungshilfe zwischen LINQ to SQL und LINQ to Entities

Wie es bisher war – ADO.NET 2.0 vs. LINQ in .NET 3.5

Auch in Visual Studio 2005 war es natürlich schon möglich, Client/Server-Anwendungen auf Basis von SQL Server 2000/2005 zu erstellen. Man bediente sich dabei regelmäßig eines interaktiven DataSet-Designers, der es ermöglichte, so genannte typisierte DataSets zu erstellen. Diese übernahmen die Infrastruktur für den Austausch von Daten zwischen Client und SQL-Server.

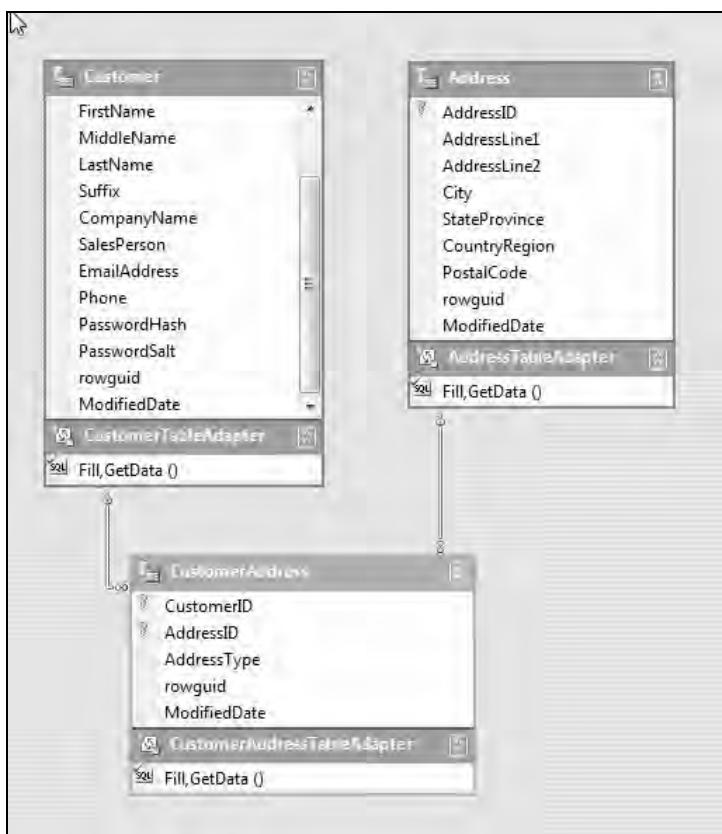
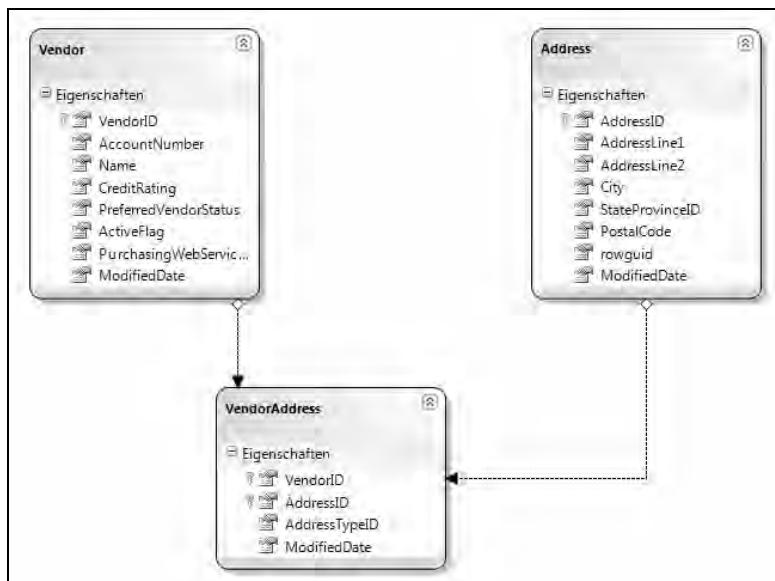


Abbildung 34.17 Der DataSet-Designer von VS 2005

Pro Tabelle wurde eine DataTable-Klasse erstellt und zudem noch jeweils eine Klasse mit dem TableAdapter. Letzterer enthielt die Methoden, um den Austausch der Daten (Selektieren, Aktualisieren, Einfügen und Löschen) mit dem SQL Server zu gewährleisten.

HINWEIS Visual Studio 2008 bietet auch die Möglichkeit zu *LINQ to DataSets*. Dabei wird die Kommunikation zwischen SQL Server und der Client-Anwendung wie in bisherigen Versionen über DataTable-Objekte und DataSet-Objekte abgewickelt, und auch die Aktualisierungslogik läuft ganz normal, wie gewohnt, über Datasets. Die Abfrage, Selektierung und Sortierung der Daten *innerhalb* von Datasets und DataTable-Objekten kann dann aber über LINQ-Abfragen erfolgen. Durch die sehr große Ähnlichkeit zu *LINQ to Objects* und fehlendem Platz wollen wir dieses Thema im Rahmen dieses Buchs nicht weiter vertiefen.

Auch für LINQ to SQL gibt es einen Designer, der dem seines Vorgängers vergleichsweise ähnlich sieht:

**Abbildung 34.18**

Der O/R-Designer von VS 2008

Wie Sie sehen, fehlen – mal ganz davon abgesehen, dass es sich im zweiten Fall um andere Tabellen handelt – in den Zusätzen die jeweiligen TableAdapter, und Verhaltensweisen, die man zuvor über Tabellenattribute einstellen konnte, werden nun über Eigenschaften gesteuert.

Und damit enden die Gemeinsamkeiten auch schon – lassen Sie sich deswegen nicht über die oberflächlichen Ähnlichkeiten hinwegtäuschen: LINQ to SQL verfolgt einen gänzlich anderen Ansatz als die typisierten DataSets (mit denen Sie, falls Sie es wünschten, natürlich auch in dieser Version von Visual Basic noch weiterarbeiten könnten).

LINQ to Objects haben Sie bereits kennen gelernt, und falls nicht, sollten Sie sich auf alle Fälle das vorherige Kapitel zu Gemüte führen. Die gleiche Syntax und ein sehr, sehr ähnlicher Ansatz wird nämlich auch von LINQ to SQL verwendet.

LINQ to SQL am Beispiel – Die ersten Schritte

Learning by doing ist immer noch der beste Weg, neue Technologien kennen zu lernen, deswegen lassen Sie uns im Folgenden ein kleines LINQ to SQL-Beispiel erstellen.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis

...\\VB 2008 Entwicklerbuch\\F - LINQ\\Kapitel 34\\LinqToSqlDemo

falls Sie die Beispiele nicht »händisch« nachvollziehen wollen. Denken Sie daran, dass Sie für das Nachvollziehen der Beispiele Zugriff auf eine Instanz von SQL Server 2008 bzw. SQL Server 2005 haben müssen und dort die *AdventureWorks*-Beispiele eingerichtet sind.

1. Erstellen Sie ein neues Visual Basic-Projekt (als Konsolenanwendung).
2. Fügen Sie mithilfe des Projektmappen-Explorers ein neues Element in die Projektmappe ein – den entsprechenden Dialog erreichen Sie über das Kontext-Menü des Projektnamens –, und wählen Sie für das neue Element, wie in der Abbildung zu sehen, die Vorlage *LINQ to SQL-Klasse* aus.
3. Nennen Sie sie AdventureWorks.

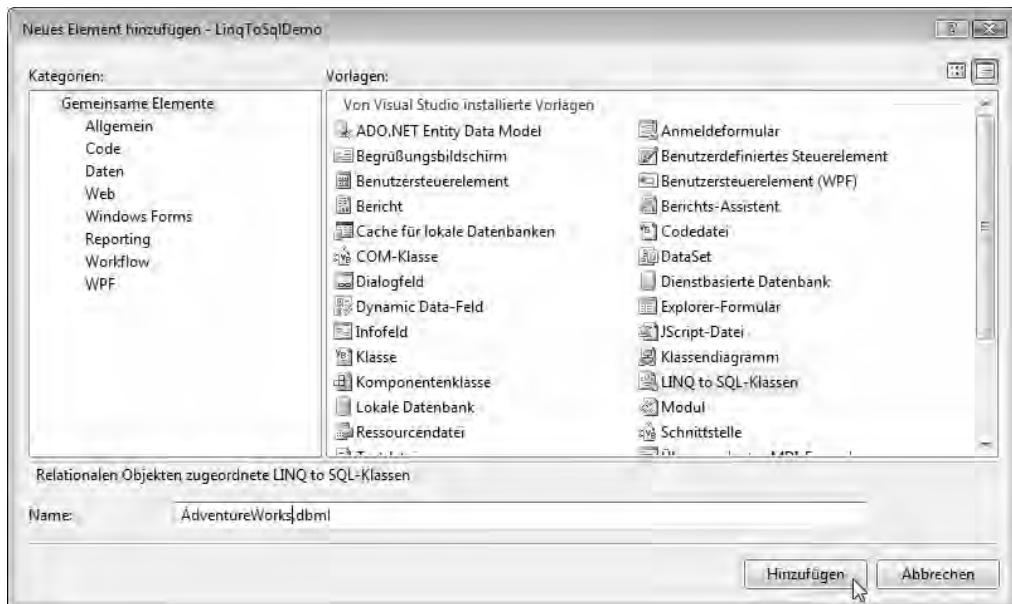


Abbildung 34.19 Hinzufügen einer LINQ to SQL-Klasse

4. Schließen Sie den Dialog mit Mausklick auf *Hinzufügen* ab.
5. Im nächsten Schritt sehen Sie den so genannten Object Relational Designer (O/R Designer), der es Ihnen gestattet, neue Business-Objektklassen auf Basis von Datenbankobjekten zu entwerfen. Öffnen Sie den Server-Explorer (falls Sie ihn nicht sehen, lassen Sie ihn über das *Ansicht*-Menü anzeigen), öffnen Sie das Kontextmenü von *Datenverbindungen*, und fügen Sie eine neue Verbindung hinzu.

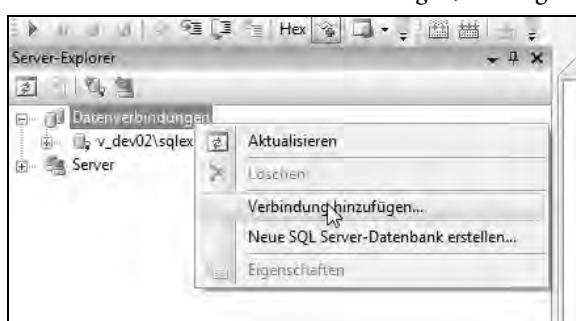


Abbildung 34.20 Erstellen einer neuen Datenverbindung

6. Wählen Sie in dem nun folgenden Dialog die SQL Server-Instanz sowie die Datenbank AdventureWorks aus, die wir im folgenden Beispiel verwenden wollen.



Abbildung 34.21 Auswählen von SQL Server-Instanz und Datenbank

HINWEIS Wenn Sie die Klappliste *Servername* öffnen, um einen Netzwerkserver an dieser Stelle auszuwählen, Sie ihn aber nicht sehen, überprüfen Sie ob a) die Firewall-Einstellungen (Port 1433 für die Standardinstanz bzw. die erste auf dem Rechner installierte Instanz) auf dem SQL Server die Kommunikation verhindern und b) der Browser-Dienst aktiviert ist. Und c), ganz wichtig: Natürlich müssen auf dem Zielserver auch die entsprechenden Netzwerkprotokolle aktiviert sein; diese sind nach einer Installation nämlich standardmäßig ausgeschaltet. Verwenden Sie, um die Netzwerkprotokolle zu aktivieren, den *SQL Server Configuration Manager*, den Sie in der Programmgruppe *SQL Server 2008* und *Konfigurationstools* finden. Aktivieren Sie die Protokolle *TCP/IP* sowie *Named Pipes*, wie in der folgenden Abbildung zu sehen.

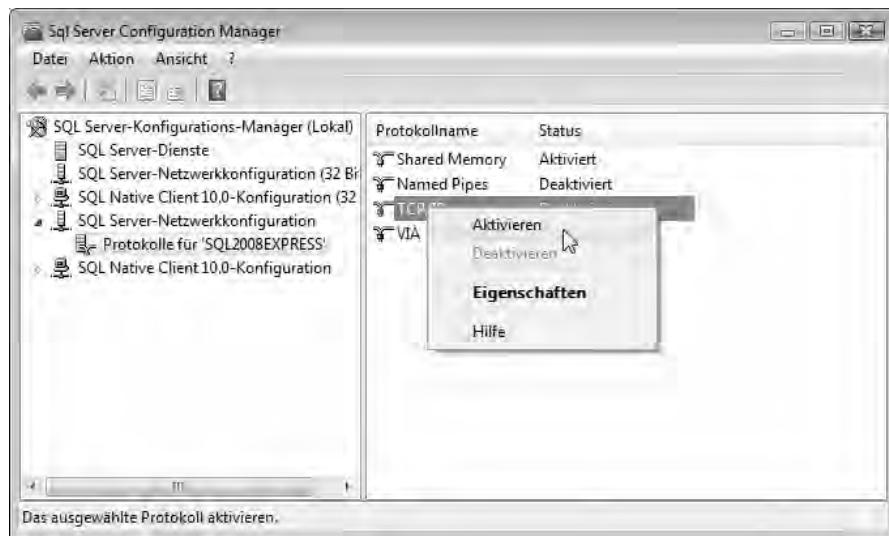


Abbildung 34.22 Damit ein SQL Server über das Netzwerk erreichbar ist, müssen Sie einstellen, welche Netzwerkprotokolle zur Kommunikation verwendet werden sollen. Denken Sie auch an das Öffnen des entsprechenden Ports (1433) in der Firewall!

- Sobald Sie die Verbindung getestet und den Dialog mit *OK* beendet haben, können Sie sich im *Server-Explorer* durch die Datenbankstrukturen bewegen. Ziehen Sie die Tabellen *Vendor*, *VendorAddress* und *Address* per Drag and Drop in den O/R-Designer. Sie erhalten eine ähnliche Ansicht wie in Abbildung 34.18.

HINWEIS Das Tabellschema in der AdventureWork-Datenbank ist in der englischen Sprache gehalten; es gibt zurzeit leider keine lokalisierten Versionen dieser Beispieldatenbank. Deswegen seien Ihnen hier die am wahrscheinlichsten unklaren Begriffe/Bezeichnungen der Datenbank kurz erklärt:

- **Customer:** Entspricht dem *Kunden*. Alle kundenrelevanten Daten werden in Tabellen, die diese Namenskategorie aufweisen, gespeichert.
- **Vendor:** Entspricht dem *Lieferanten*. Entsprechende Daten werden in Tabellen mit dieser Namenskategorie gespeichert.
- **PurchaseOrder:** Entspricht dem *Kaufauftrag*.
- **Contact:** Entspricht dem *Ansprechpartner*, also eine für eine Aufgabe definierte Person innerhalb des Unternehmens, oder eine Zielperson bei einer Tabellenrelation.
- **Employee:** Entspricht dem *Mitarbeiter*.
- **Currency:** Entspricht der *Währung*.
- **Sales/Order:** Sind die *Verkäufe/Verkaufsaufträge*.
- **SpecialOffer:** Dabei handelt es sich um *Sonderangebote*.

Alle weiteren Namensgebungen (wie *Address*) lassen sich problemlos aus dem englischen Wortlaut ableiten.

Dabei ist es zunächst übrigens völlig gleich, ob Sie Tabellen der Datenbank oder Sichten in den O/R-Designer ziehen – das hat bestenfalls Einfluss auf die entsprechenden generierten Klassen. Auch gespeicherte Prozeduren lassen sich auf diese Weise mit dem O/R-Designer verarbeiten, jedoch landen diese, anders als Sichten oder Tabellen, nicht im Hauptbereich sondern im rechten Bereich des O/R-Designers, wie in Abbildung 34.23 gekennzeichnet.

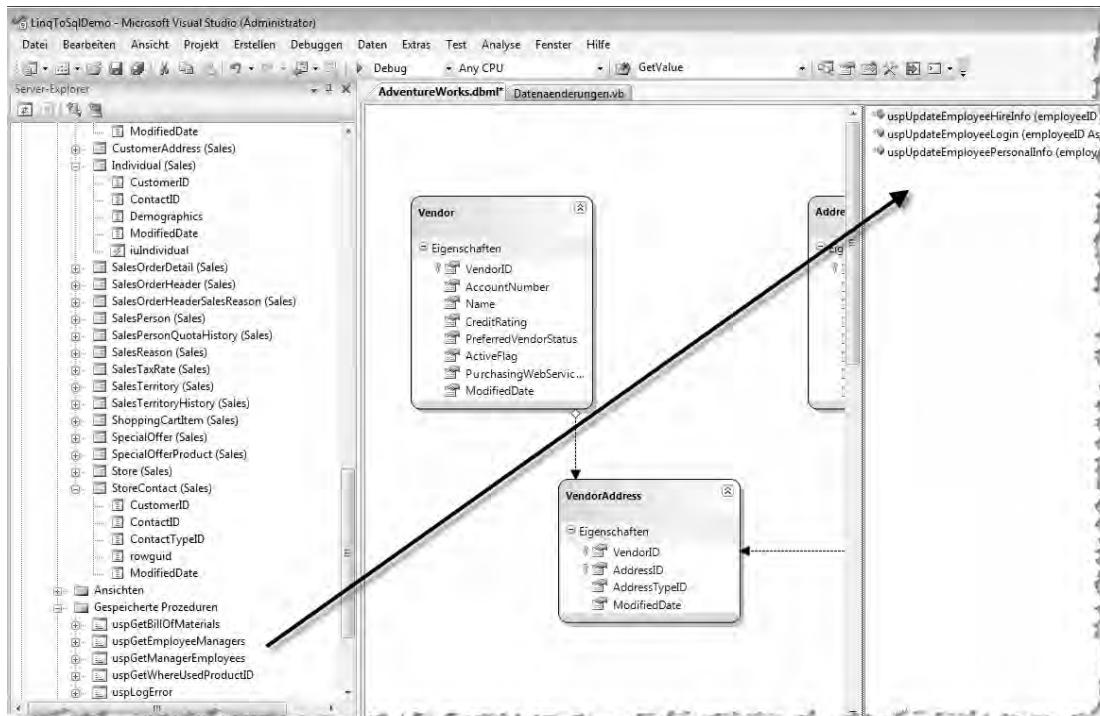


Abbildung 34.23 Hinzufügen von gespeicherten Prozeduren

8. Speichern Sie Ihre Design-Änderungen ab. Wenn Sie mit dem entsprechenden Symbol des Projektmappen-Explorers *Alle Dateien* im Projektmappen-Explorer eingeblendet haben, finden Sie dort eine Datei namens *AdventureWorks.designer.vb*.

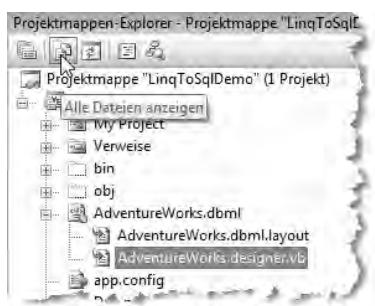


Abbildung 34.24 Der O/R-Designer erstellt eine VB-Codedatei – klicken Sie auf das Symbol *Alle Dateien anzeigen*, falls die Datei nicht zu sehen sein sollte!

Diese Datei beinhaltet unter anderem den Quellcode für den schon erwähnten *DataContext*. Erinnern wir uns: Der *DataContext* ist die Verwaltungsinstanz bei LINQ to SQL, etwas, was wir bislang bei LINQ to Objects nicht kennen gelernt haben, weil es nicht benötigt wurde: Der *DataContext* verwaltet beispielsweise die Verbindungszeichenfolge für den Verbindungsaufbau zur Datenbank, er kümmert sich um die Aktualisierungslogiken und übernimmt die Transaktionssteuerung.

Doch lassen Sie uns die Designer-Codedatei ein wenig genauer unter die Lupe nehmen und schauen, welchen Code der LINQ to SQL O/R-Designer für uns erzeugt hat:

Da wäre zunächst einmal der Prolog der Codedatei, der die notwendigen Namespaces einbindet. Und dann beginnt die Datei mit der Verwalterklasse, dem *DataContext*:

```
'-----
' <auto-generated>
'   Dieser Code wurde von einem Tool generiert.
'   Laufzeitversion:2.0.50727.3053
'
'   Änderungen an dieser Datei können falsches Verhalten verursachen und gehen verloren, wenn
'   der Code erneut generiert wird.
' </auto-generated>
'-----
```

```
Option Strict On
Option Explicit On

Imports System
Imports System.Collections.Generic
Imports System.ComponentModel
Imports System.Data
Imports System.Data.Linq
Imports System.Data.Linq.Mapping
Imports System.Linq
Imports System.Linq.Expressions
Imports System.Reflection

<System.Data.Linq.Mapping.DatabaseAttribute(Name:="AdventureWorks")> _
Partial Public Class AdventureWorksDataContext
    Inherits System.Data.Linq.DataContext
```

Weiter unten erfolgen dann die Definitionen der entsprechenden Tabellen-Eigenschaften, die als generische Auflistung der vom O/RM erstellten Businessobjekte implementiert sind. Also: Für eine Tabellenzeile der Ausgangstabelle *Vendor* dient im Objektmodell die Klasse *Vendor*. Ein ganzer Tabelleninhalt oder Zeilenergebnisse einer Abfrage dieser Tabelle werden durch eine Auflistung dieses Typs repräsentiert und können durch die Eigenschaft *Vendors* abgerufen werden. Das gilt für die anderen Tabellen, die wir in den Designer gezogen haben, gleichermaßen, und deswegen schneiderte der O/R-Designer folgenden Code daraus:

```
Public ReadOnly Property Vendors() As System.Data.Linq.Table(Of Vendor)
    Get
        Return Me.GetTable(Of Vendor)
    End Get
End Property
```

```

Public ReadOnly Property VendorAddresses() As System.Data.Linq.Table(Of VendorAddress)
    Get
        Return Me.GetTable(Of VendorAddress)
    End Get
End Property

Public ReadOnly Property Addresses() As System.Data.Linq.Table(Of Address)
    Get
        Return Me.GetTable(Of Address)
    End Get
End Property

```

TIPP Hier an der Stelle können Sie übrigens sehen, dass die automatische Pluralisierung von Tabellenzeile und Tabelle bzw. Objekt und Objektauflistung bei der Namensgebung durchaus Sinn macht. Aus *Vendor* wird die Eigenschaft *Vendors* als Auflistung, aus *VendorAddress* wird *VendorAddresses*, usw. Was hier im Englischen immer gut klappt, hört sich im Deutschen seltsam an, falls man nicht gerade den sehr charmanten Ruhrgebietssdialet gewohnt ist und liebt.¹⁰ Die Eigenschaft für die Auflistung von Kunden-Objekten beispielsweise *Kundens* zu nennen, würde man Ihnen wahrscheinlich nur entlang der B1 zwischen Dortmund und Essen durchgehen lassen.

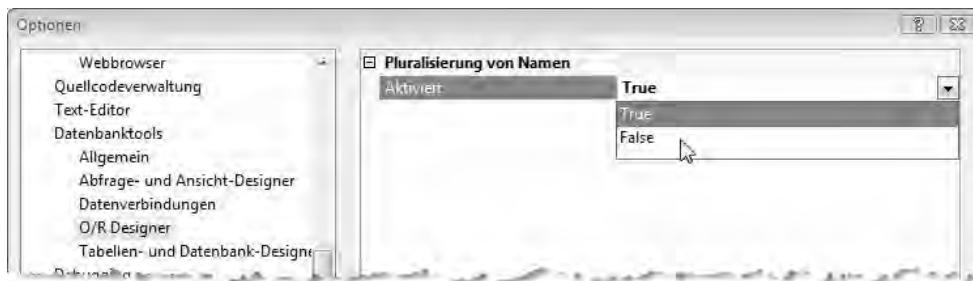


Abbildung 34.25 Im Optionsdialog von Visual Studio können Sie die Pluralisierung von Auflistungseigenschaften beim Tabellen-Mapping ein- und ausschalten

Das ist der Grund, weswegen Sie die Pluralisierung ein- und ausschalten können. Dazu wählen Sie aus dem Menü *Extras* den Befehl *Optionen* und navigieren in der Optionsliste zu *Datenbanktools* und *O/R Designer*. Ändern Sie in der Eigenschaftenliste dann die Pluralisierung von Namen entsprechend.

Sie sehen: Sämtliche Tabellenstrukturen werden in der Ableitung der *DataContext*-Klasse definiert, die in unserem Fall *AdventureWorksDataContext* lautet. Der *DataContext* übernimmt zur Laufzeit übrigens in etwa die Rolle, die *TableAdapter* bei typisierten *DataSets* spielt – er kümmert sich ebenfalls um die Aktualisierungslogik.

Nun fügen wir ein einfaches Modul in das Beispiel ein, in dem wir unsere erste kleine Abfrage starten und das Ergebnis in das Konsolenfenster ausgeben.

¹⁰ Der Vorarbeiter einer Zeche treibt seine Mitarbeiter an: »Kommt endlich mit die Wagens, KOMMT MIT DIE WAGENS«. Der Chef der Zeche hört das, und stellt seinen Vorarbeiter zur Rede: »Hör'n Se 'mal Sablowski, Sie könn' doch eigentlich ganz gut Deutsch, wieso sagen Se denn immer ›kommt mit die Wagens?‹ – Sablowski entgegnet knapp: »Chef, wenn ich sage, ›kommt mit den Wagen, komm 'se doch nur mit einem!«.

1. Wechseln Sie daher mithilfe des Projektmappen-Explorers per Doppelklick zur Codedatei Modul *Modul1.vb*.
2. Als erstes deklarieren wir den benötigten Datenkontext, der in unserem Fall vom Typ AdventureWorks-DataContext ist:

```
' Neuen DataContext anlegen
Dim awDatacontext As New AdventureWorksDataContext
```

Mithilfe von IntelliSense können wir schon jetzt erkennen, wie in der folgenden Abbildung zu sehen, dass die gemappten Tabellen, die wir zuvor im O/R-Designer angelegt haben, jetzt in Form von Eigenschaften über diesen DataContext zu erreichen sind.

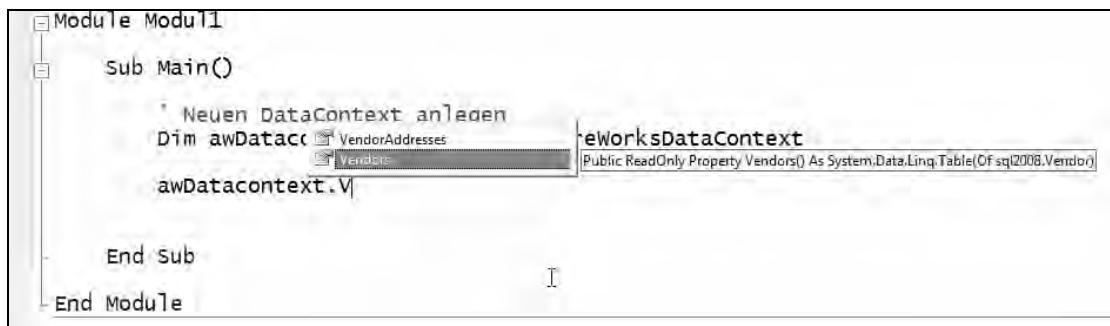


Abbildung 34.26 Die gemappten Tabellen sind in Form von generischen Auflistungen über den DataContext zu erreichen

3. Damit haben wir nun die Möglichkeit, beispielsweise die Tabellen der Lieferanten (Vendors) in eine LINQ-Abfrage einzubauen. Und jetzt kommt das Schöne an LINQ: Wir bauen unsere LINQ-Abfragen nun genau so, wie wir es bei LINQ to Objects kennen gelernt haben. Dadurch, dass wir eine Tabellenauf-listungseigenschaft des AdventureWorks-Datenkontext verwenden, wird diese Abfrage automatisch auf den SQL-Server »gelenkt«. Probieren wir es. Fügen Sie die Zeilen:

```
Module Modul1

Sub Main()

    ' Neuen DataContext anlegen
    Dim awDatacontext As New AdventureWorksDataContext

    Dim LieferantenListe = From LieferantItem In awDatacontext.Vendors _
                           Where LieferantItem.Name.StartsWith("W") _
                           Order By LieferantItem.Name

    For Each Lieferant In LieferantenListe
        Console.WriteLine(Lieferant.AccountNumber & ": " & Lieferant.Name)
    Next

```

```

Console.WriteLine()
Console.WriteLine("Taste drücken, zum Beenden")
Console.ReadKey()
End Sub

End Module

```

Übrigens, haben Sie es bemerkt: Auch beim Eingeben der im Listing fett gesetzten eigentlichen LINQ-Abfrage werden Sie wieder proaktiv von IntelliSense unterstützt. Da die Objekte, die wir hier verwenden, aus der Datenbank 1:1 gemappt wurden, brauchen Sie sich um Feldnamen und Typen keine Gedanken mehr zu machen: IntelliSense zeigt Ihnen direkt bei der Ausformulierung der LINQ-Abfrage an – wie auch in der folgenden Abbildung zu sehen:

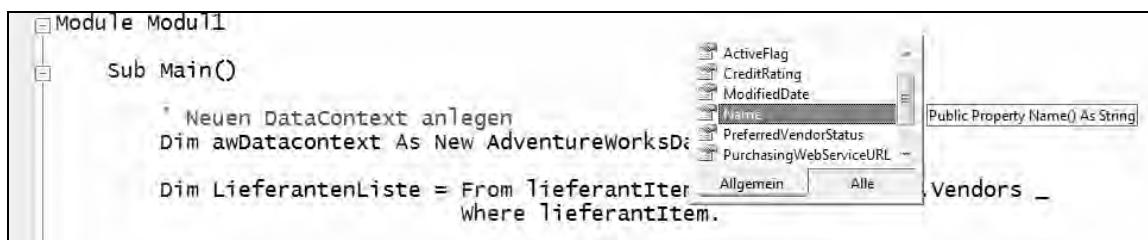


Abbildung 34.27 Auch bei LINQ to SQL gilt – volle IntelliSense-Unterstützung entlastet Sie aktiv. Tabellschemata in Form von Felddefinitionen und -Typen

Und damit können wir unsere erste kleine LINQ-Abfrage schon ausprobieren. Wenn Sie diese Anwendung laufen lassen (beim ersten Start kann es ein paar Sekunden länger dauern, bevor was passiert, da der SQL Server die entsprechenden Datenseite cacht, was bei allen folgenden Starts dann viel schneller geht!).

```

WESTJUN0001: West Junction Cycles
WESTAMER0001: WestAmerica Bicycle Co.
WIDEWOR0001: Wide World Importers
WOODFIT0001: Wood Fitness

Taste drücken, zum Beenden

```

Das Ergebnis entspricht genau dem, was zu erwarten war!

Wechseln Sie nun zum Modul *JoinAnonym.vb*, und schauen Sie sich die dort vorhandenen Codezeilen an:

```

Module JoinAnonym

''' <summary>
''' Beispiel: Abfragen von Daten mit Hilfe eines Joins
''' Das Ergebnis wird in anonymen Typen gespeichert
''' </summary>
''' <remarks></remarks>
Sub Main()
    Dim datacontext As New AdventureWorksDataContext

    Dim LieferantenAdressen = From adr In datacontext.Addresses _
                                Join lieferantenAdressItem In datacontext.VendorAddresses _
                                On adr.AddressID Equals lieferantenAdressItem.AddressID _

```

```

Join lieferantItem In datacontext.Vendors
On lieferantItem.VendorID Equals lieferantenAdressItem.VendorID
Where lieferantItem.Name.StartsWith("H")
Select New With {.KontoNr = lieferantItem.AccountNumber,
                 .Lieferantename = lieferantItem.Name,
                 .Adresszeile1 = lieferantenAdressItem.Address.AddressLine1,
                 .Adresszeile2 = lieferantenAdressItem.Address.AddressLine1,
                 .PLZ = lieferantenAdressItem.Address.PostalCode,
                 .Ort = lieferantenAdressItem.Address.City}

For Each Lieferant In LieferantenAdressen
    Console.WriteLine("Konto: {0}; Name: {1}; Ort {2} {3} {4} {5}", _
                      Lieferant.KontoNr,
                      Lieferant.Lieferantename,
                      Lieferant.Adresszeile1,
                      Lieferant.Adresszeile2,
                      Lieferant.PLZ,
                      Lieferant.Ort)
Next
Console.WriteLine("Bitte mit Return bestätigen")
Console.ReadLine()
End Sub

End Module

```

Die Syntax von LINQ haben Sie bereits in Kapitel 32 »LINQ to Objects« kennen gelernt – und jetzt lernen Sie die Stärken von LINQ kennen: Kennen Sie eine, kennen Sie alle! (Einmal von den Feinheiten abgesehen, die Sie in Ergänzung kennen müssen.)

In dieser Abfrage werden die drei Tabellen *Address*, *VendorAddress* und *Vendor* durch eine *Join*-Klausel miteinander verknüpft. Zudem wird die Abfrage auf alle Firmen beschränkt, deren Name mit »H« beginnt. Für alle gefundenen Datensätze werden anschließend Informationen zu dem Lieferanten und seiner Adresse ausgegeben – für die Ergebnismenge wird dazu mit *Select* eine Auflistung mit Instanzen einer entsprechenden anonymen Klasse erstellt.

Damit das Beispielprojekt jetzt auch mit dieser *Main*-Methode startet, stellen Sie bitte in den Projekteigenschaften das entsprechende Startobjekt ein. Die Projekteigenschaften erreichen Sie, indem Sie im Projektmappen-Explorer auf dem Projektnamen das Kontextmenü öffnen und Eigenschaften auswählen (Sie erhalten folgende Maske).

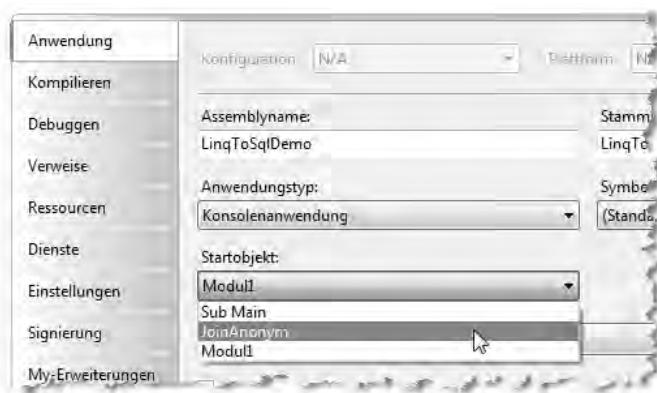


Abbildung 34.28 So bestimmen Sie, in welchem Modul die Sub Main zum Starten der Anwendung aufgerufen werden soll

Die Ausgabe dieses Programms liefert:

```
Konto: HILLSBI0001; Name: Hill's Bicycle Service; Ort 3664 Colt Ct. 94801 Richmond  
Konto: HYBRIDB0001; Name: Hybrid Bicycle Center; Ort 6959 Lakewood Court 94015 Daly City  
Konto: HILLBIC0001; Name: Hill Bicycle Center; Ort 87 Pheasant Circle 94611 Oakland  
Konto: HOLIDAY0001; Name: Holiday Skate & Cycle; Ort 3294 Buena Vista 85284 Lemon Grove
```

Taste drücken, zum Beenden

Protokollieren der generierten T-SQL-Befehle

Und jetzt kommt das Entscheidende: Im Gegensatz zu LINQ to Objects wird das Ergebnis dieser Abfrage nicht auf dem Rechner ermittelt, auf dem das Programm läuft, sondern aus der LINQ-Abfrage wird ein SQL-Kommando erstellt, zur Datenbank versendet und das Ergebnis wird anschließend von der Datenbank quasi »abgeholt«.

Sie können das einfach überprüfen, indem Sie vor der For/Each-Schleife der Log-Eigenschaft des Datenkontexts eine TextWriter-Instanz hinzufügen. Wir nutzen hierfür Console.Out:

```
Console.WriteLine()  
datacontext.Log = Console.Out
```

Die Ausgabe des Programms liefert nun etwas mehr Informationen, und verrät das Geheimnis, wie oder vielmehr mit welchen SQL-Statements die Kommunikation zwischen dem Client und dem SQL Server über den Datenkontext vorstatten geht:

```
SELECT [t2].[AccountNumber] AS [KontoNr], [t2].[Name] AS [Lieferantename], [t3].[AddressLine1] AS  
[Adresszeile1], [t3].[AddressLine2] AS [Adresszeile2], [t3].[P  
ostalCode] AS [PLZ], [t3].[City] AS [Ort]  
FROM [Person].[Address] AS [t0]  
INNER JOIN [Purchasing].[VendorAddress] AS [t1] ON [t0].[AddressID] = [t1].[AddressID]  
INNER JOIN [Purchasing].[Vendor] AS [t2] ON [t1].[VendorID] = [t2].[VendorID]  
INNER JOIN [Person].[Address] AS [t3] ON [t3].[AddressID] = [t1].[AddressID]  
WHERE [t2].[Name] LIKE @p0  
-- @p0: Input NVarChar (Size = 2; Prec = 0; Scale = 0) [H%]  
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1
```

```
Konto: HILLSBI0001; Name: Hill's Bicycle Service; Ort 3664 Colt Ct. 94801 Richmond  
Konto: HYBRIDB0001; Name: Hybrid Bicycle Center; Ort 6959 Lakewood Court 94015 Daly City  
Konto: HILLBIC0001; Name: Hill Bicycle Center; Ort 87 Pheasant Circle 94611 Oakland  
Konto: HOLIDAY0001; Name: Holiday Skate & Cycle; Ort 3294 Buena Vista 85284 Lemon Grove
```

Taste drücken, zum Beenden

Wie Sie sehen, werden nur die Daten selektiert, die wir auch in dem Select-Ausdruck festgelegt haben. Zudem wird unsere Where-Klausel ebenso an den SQL Server übermittelt.

Natürlich arbeitet der LINQ to SQL-Provider dabei mit Parametern, und setzt diese nicht als Konstanten direkt in die Abfrage ein. Deswegen ist die WHERE-Klausel auch als

```
WHERE [t2].[Name] LIKE @p0
```

formuliert. Welcher Parameter dann für @p0 eingesetzt wird, protokolliert das Log in den nächsten beiden fett gesetzten Zeilen im Listing. Das hat den Vorteil, dass der SQL Server einen so genannten Ausführungsplan erstellen kann. Das bedeutet, dass er einen nicht unerheblichen Aufwand betreibt, die beste Strategie auszuarbeiten, um schnellstmöglich an die abgefragten Daten zu kommen. Lässt man ihn dabei mit Parametern arbeiten, kann er diesen Ausführungsplan wiederverwenden, was nicht möglich wäre, würde man ihm die Abfragekonditionen direkt als Konstanten und nicht als Parameter übergeben.

Diese Umwandlung von LINQ to SQL in eine Select-Anweisung wird durch einen datenbankspezifischen LINQ-Provider durchgeführt, und ich mag Ihnen empfehlen, ein wenig mit verschiedenen Abfragen gegen die AdventureWorks-Datenbank zu experimentieren, um einerseits ein Gefühl für LINQ to SQL an sich, andererseits aber auch eines für die Umsetzung von lokalem LINQ auf »echtes« T-SQL zu bekommen.

Verzögerte Abfrageausführung und kaskadierte Abfragen

Aufgrund der verzögerten Ausführung von LINQ-Abfragen, die für LINQ to SQL genau so gilt wie für LINQ to Objects (Kapitel 32 hält mehr dazu bereit), können sehr einfache und übersichtliche Sub-Select-Abfragen mit LINQ to SQL durchgeführt werden.

Im folgenden Beispiel wiederholen wir eine Adressabfrage für einen Lieferanten. Dieses Mal interessieren uns jedoch nicht die Lieferantendaten, sondern lediglich die Adresse des Lieferanten:

```
Module KaskadierteAbfragen

''' <summary>
''' Beispiel: Zusammenstellen einer Abfrage aus mehreren Abfragen
''' </summary>
''' <remarks></remarks>
Sub main()

    Dim GesuchterLieferant = "Hybrid Bicycle Center"

    Dim datacontext As New AdventureWorksDataContext

    ' Abfrage für Lieferant erstellen
    Dim lieferant = From lieferantItem In datacontext.Vendors _
                    Where lieferantItem.Name = GesuchterLieferant

    ' Abfrage für Lieferantenadresse erstellen
    Dim lieferantAdressZuordnung = From zuordnungItem In datacontext.VendorAddresses _
                                    Where lieferant.Any(Function(c) c.VendorID = zuordnungItem.VendorID)

    ' Wichtig: Wir könnten natürlich auch die Where-Klausel formulieren als
    ' ... WHERE lieferant(0).VendorID=zuordnungsItem.VendorID
    ' doch dann griffen wir auf ein Element (das einzige!) der ersten Ergebnisaufstellung zurück,
    ' und würden dort bereits die Abfrage ausführen und die Kaskade unterbrechen.
```

```
' Abfrage für Adresse erstellen
Dim lieferantAdresse = From adressItem In datacontext.Addresses _
    Where lieferantAdresse.Zuordnung.Any(Function(a) a.AddressID = adressItem.AddressID)

'Alle Datenbank-Aktionen auf Console.Out ausgeben
datacontext.Log = Console.Out

' Abfrage ausführen (und Objekt vom Datenkontext trennen!)
Dim adressListe = lieferantAdresse.ToList

' Gefundene Datensäze ausgeben
For Each adressItem In adressListe
    Console.WriteLine("Adresse von {0}: {1} {2} {3}", GesuchterLieferant, _
        adressItem.AddressLine1, _
        adressItem.PostalCode, _
        adressItem.City)
Next

Console.WriteLine()
Console.WriteLine("Taste drücken, zum Beenden")
Console.ReadKey()
End Sub

End Module
```

HINWEIS Denken Sie daran, das Startobjekt auf das Modul *KaskadierteAbfragen* im Beispielprojekt zu ändern (gemäß Abbildung 34.34 und der entsprechenden dort zu findenden Beschreibung), damit dieses Beispiel ausgeführt wird.

Als Erstes selektieren wir die Lieferantendaten des gesuchten Lieferanten. Diese Abfrage wird in `lieferant` gespeichert – und das ist für den Moment äußerst richtig formuliert. Die eine Ergebnismenge liefert `lieferant` erst dann, wenn auf das erste Element zugegriffen würde, denn erst dann führt LINQ und damit der SQL-Server die Abfrage aus. Bis das geschieht, ist `lieferant` also quasi nur ein Platzhalter für die bis dahin zusammengestellt Abfrage.

Als Nächstes selektieren wir aus der Tabelle `VendorAddresses` den Datensatz, der zur Lieferanten-ID des gesuchten Lieferanten passt. Diese Abfrage wird in `lieferantAddressZuordnung` gespeichert.

Nun können wir in der Tabelle `Addresses` suchen, und wir benutzen dazu die `Any`-Klausel, um den entsprechenden Datensatz zu finden, der den Kriterien entspricht, da die Ausgangsliste natürlich ebenfalls komplett durchsucht werden muss – und das ist genau das, was `Any` leistet: `Any` iteriert durch die Elemente und überprüft, ob wenigstens ein Element der Bedingung entspricht, die durch den angegebenen Lambda-Ausdruck vorgegeben wird. Diese Abfrage wird in `lieferantAdresse` gespeichert.

HINWEIS Noch einmal zur Erinnerung: Das Prinzip der verzögerten Ausführung bestimmt, dass bislang noch keine Abfrage ausgeführt wurde. Es wurden lediglich die Abfragen definiert. Das gilt natürlich gleichermaßen für die Ausführung von `Any`. Erst durch den Aufruf der `ToList`-Methode werden die drei Abfragen zu einer zusammengesetzt und ausgeführt, weil – wie wir es bei LINQ to Objects bereits kennen gelernt haben – das *erste Mal* der Versuch gestartet wird, ein Element der Ergebnismenge abzurufen.

Als Ergebnis erhalten wir die Anschrift des gesuchten Lieferanten (und, da das Logging für die SQL-Abfragegenerierung aktiviert wurde, auch die eigentliche SQL-Abfrage, die an den SQL-Server gesendet wird).

```

SELECT [t0].[AddressID], [t0].[AddressLine1], [t0].[AddressLine2], [t0].[City], [t0].[StateProvinceID],
[t0].[PostalCode], [t0].[rowguid], [t0].[ModifiedDate]
FROM [Person].[Address] AS [t0]
WHERE EXISTS(
    SELECT NULL AS [EMPTY]
    FROM [Purchasing].[VendorAddress] AS [t1]
    WHERE ([t1].[AddressID] = [t0].[AddressID]) AND (EXISTS(
        SELECT NULL AS [EMPTY]
        FROM [Purchasing].[Vendor] AS [t2]
        WHERE ([t2].[VendorID] = [t1].[VendorID]) AND ([t2].[Name] = @p0)
    )))
)
-- @p0: Input NVarChar (Size = 21; Prec = 0; Scale = 0) [Hybrid Bicycle Center]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

Adresse von Hybrid Bicycle Center: 6959 Lakewood Court 94015 Daly City

Taste drücken, zum Beenden

```

Übrigens: Wenn Sie schon ein wenig mit den verschiedenen Abfragemöglichkeiten von LINQ to Objects bzw. LINQ to SQL herumexperimentiert haben, haben Sie sich möglicherweise gefragt, warum wir hier im Beispiel die Any-Methode von den Abfragen `Lieferant` und `LieferantAdressZuordnung` verwendet haben, obwohl `All` in diesem Zusammenhang sinnvoller erscheinen könnte. Um Ihnen diesen Unterschied vor Augen zu führen, ändern wir den Code ab und verwenden nun die `All`-Methode für das weitere Selektieren der Daten:

```

.
.
.

' Abfrage für Lieferantenadresse erstellen
Dim lieferantAdressZuordnung = From zuordnungItem In datacontext.VendorAddresses
    Where lieferant.All(Function(c) c.VendorID = zuordnungItem.VendorID)

' Wichtig: Wir könnten natürlich auch die Where-Klausel auch formulieren als
'           ... WHERE lieferant(0).VendorID=zuordnungItem.VendorID
' doch dann griffen wir auf ein Element (das einzige!) der ersten Ergebnisaufstellung zurück,
' und würden dort bereits die Abfrage ausführen und die Kaskade unterbrechen.

' Abfrage für Adresse erstellen
Dim lieferantAdresse = From adressItem In datacontext.Addresses
    Where lieferantAdressZuordnung.All(Function(a) a.AddressID = adressItem.AddressID)
.
.
.
```

Wenn das Programm nun gestartet wird, erhalten wir folgende Ausgabe:

```
SELECT [t0].[AddressID], [t0].[AddressLine1], [t0].[AddressLine2], [t0].[City], [t0].[StateProvinceID],  
[t0].[PostalCode], [t0].[rowguid], [t0].[ModifiedDate]  
FROM [Person].[Address] AS [t0]  
WHERE NOT (EXISTS  
    SELECT NULL AS [EMPTY]  
    FROM [Purchasing].[VendorAddress] AS [t1]  
    WHERE ((  
        (CASE  
            WHEN [t1].[AddressID] = [t0].[AddressID] THEN 1  
            ELSE 0  
        END)) = 0) AND (NOT (EXISTS(  
            SELECT NULL AS [EMPTY]  
            FROM [Purchasing].[Vendor] AS [t2]  
            WHERE ((  
                (CASE  
                    WHEN [t2].[VendorID] = [t1].[VendorID] THEN 1  
                    ELSE 0  
                END)) = 0) AND ([t2].[Name] = @p0)  
            )))  
        ))  
-- @p0: Input NVarChar (Size = 21; Prec = 0; Scale = 0) [Hybrid Bicycle Center]  
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1  
  
Adresse von Hybrid Bicycle Center: 6959 Lakewood Court 94015 Daly City  
  
Taste drücken, zum Beenden
```

Diese Abfrage funktioniert ebenfalls, aber wie Sie sehen, ist das erstellte T-SQL-SELECT deutlich komplexer. Und das liegt einfach daran, dass die All-Methode jeden Datensatz betrachtet, während Any nur die Datensätze betrachtet, die den Kriterien entsprechen.

Eager und Lazy-Loading – Steuern der Ladestrategien bei 1:n-Relationen

Stellen Sie sich vor, Sie möchten in einer Datenbank häufig Daten einer Tabelle abrufen, die in einer 1:n-Beziehung mit einer anderen Tabelle verknüpft sind. Sie brauchen dazu, wenn Sie die entsprechenden Relationen der Tabellen im O/R-Designer eingefügt haben, keinen zusätzlichen Aufwand zu betreiben, da LINQ to SQL die entsprechenden Maßnahmen für Sie unternimmt.

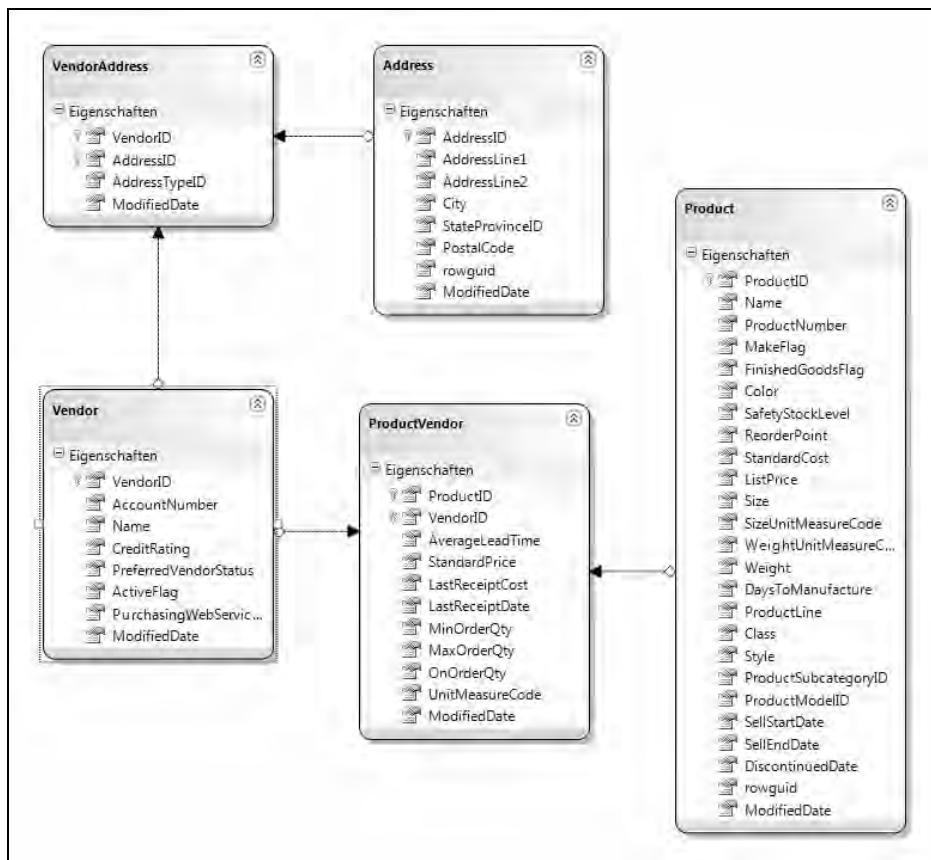


Abbildung 34.29 Um das verzögerte Abfragen von 1:n-Relationen zu demonstrieren, verwenden wir zusätzlich die Tabelle mit den Produkten der Lieferanten (*ProductVendor*), die wiederum mit der Liste aller Produkte (*Product*) verknüpft ist

So fragt der folgende Codeausschnitt (im Modul *EagerLazyLoading* des Beispielprojektes zu finden) alle Lieferanten ab, die mit dem Buchstaben »H« beginnen.

```

Dim LieferantenListe = From lieferantItem In dataContext.Vendors _
    Where lieferantItem.Name.StartsWith("H") _
    Order By lieferantItem.Name

'Rausfinden: Welche Produkte liefert der Lieferant
For Each Lieferant In LieferantenListe
    Console.WriteLine(Lieferant.AccountNumber & ":" & Lieferant.Name)
    Console.WriteLine(New String("c", 70))
    For Each produkt In Lieferant.ProductVendors
        Console.Write("ID:" & produkt.ProductID & " ")
        Console.WriteLine("Name:" & produkt.Product.Name)
    Next
    Console.WriteLine()
Next
  
```

Das Ergebnis dieser Zusammenstellung sieht nun folgendermaßen aus:

```
HILLBIC0001: Hill Bicycle Center
=====
ID:524 Name:HL Spindle/Axle

HILLSBI0001: Hill's Bicycle Service
=====
ID:911 Name:LL Road Seat/Saddle
ID:912 Name:ML Road Seat/Saddle

HOLIDAY0001: Holiday Skate & Cycle
=====

HYBRIDB0001: Hybrid Bicycle Center
=====
ID:910 Name:HL Mountain Seat/Saddle

Taste drücken, zum Beenden
```

Soweit ist dieser Vorgang super einfach zu handeln. Doch wie genau, und noch wichtiger, wann genau erhält die Anwendung die untergeordneten Produktdaten zu jeder Bestellung? Wir können diese Frage vergleichsweise schnell klären, wenn wir das Logging zu dieser Abfrage wieder aktivieren, und uns das Ergebnis im Ausgabefenster anschauen (wegen der großen Zeilenzahl gekürzt):

```
Produktliste zu jedem Lieferanten immer direkt laden? [J/N]n
SELECT [t0].[VendorID], [t0].[AccountNumber], [t0].[Name], [t0].[CreditRating],
[t0].[PreferredVendorStatus], [t0].[ActiveFlag], [t0].[PurchasingWebServiceURL],
[t0].[ModifiedDate]
FROM [Purchasing].[Vendor] AS [t0]
WHERE [t0].[Name] LIKE @p0
ORDER BY [t0].[Name]
-- @p0: Input NVarChar (Size = 2; Prec = 0; Scale = 0) [H%]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

HILLBIC0001: Hill Bicycle Center
=====
SELECT [t0].[ProductID], [t0].[VendorID], [t0].[AverageLeadTime], [t0].[StandardPrice],
[t0].[LastReceiptCost], [t0].[LastReceiptDate], [t0].[MinOrderQty], [t0]
.[MaxOrderQty], [t0].[OnOrderQty], [t0].[UnitMeasureCode], [t0].[ModifiedDate]
FROM [Purchasing].[ProductVendor] AS [t0]
WHERE [t0].[VendorID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [96]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

ID:524 SELECT [t0].[ProductID], [t0].[Name], [t0].[ProductNumber], [t0].[MakeFlag],
[t0].[FinishedGoodsFlag], [t0].[Color], [t0].[SafetyStockLevel], [t0].[Reord
erPoint], [t0].[StandardCost], [t0].[ListPrice], [t0].[Size], [t0].[SizeUnitMeasureCode],
[t0].[WeightUnitMeasureCode], [t0].[Weight], [t0].[DaysToManufacture],
[t0].[ProductLine], [t0].[Class], [t0].[Style], [t0].[ProductSubcategoryID], [t0].[ProductModelID],
[t0].[SellStartDate], [t0].[SellEndDate], [t0].[Discontinue
dDate], [t0].[rowguid], [t0].[ModifiedDate]
FROM [Production].[Product] AS [t0]
```

```
WHERE [t0].[ProductID] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [524]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1
Name:HL Spindle/Axle
```

Sie sehen, dass hier für die Verknüpfung über die insgesamt drei Tabellen auch drei SQL-Anweisungen erforderlich sind, die dann dafür sorgen, dass beim Zugriff auf die entsprechenden Eigenschaften für die verknüpften Tabellen die dazugehörigen Daten aus dem SQL-Server geladen werden.

Das kann unter Umständen für einige Szenarien zu wirklichen Performance-Problemen führen, denn wenn nicht so viele Daten aus Relationen beispielsweise nur ein einziges Mal und möglichst schnell benötigt werden, wäre es sehr viel günstiger, die Abfrage aller Daten mithilfe (aus SQL-Sicht) einer JOIN-Abfrage zu realisieren.

Dazu müssen Sie das Load-Verhalten von Lazy-Loading¹¹ auf Eager-Loading umstellen, und Sie haben bei LINQ to SQL die Möglichkeit, dieses Verhalten für bestimmte Relationen gezielt zu bestimmen.

HINWEIS Denken Sie daran, das Startobjekt auf das Modul *EagerLazyLoading* im Beispielprojekt zu ändern (gemäß Abbildung 34.34 und der entsprechenden dort zu findenden Beschreibung), damit dieses Beispiel ausgeführt wird.

Betrachten wir das folgende Listing:

```
Imports System.Data.Linq

Module EagerLazyLoading

Sub main()

    Dim verzögertesLaden = True

    ' Neuen DataContext anlegen
    Dim dataContext As New AdventureWorksDataContext
    Console.WriteLine("Produktliste zu jedem Lieferanten immer direkt laden? [J/N]")
    If Console.ReadLine.ToLower = "j" Then
        verzögertesLaden = False
    End If

    ' Alle Datenbank Aktionen auf Console.Out ausgeben
    dataContext.Log = Console.Out

    If Not verzögertesLaden Then
        Dim ladeOptionen = New DataLoadOptions
        ladeOptionen.LoadWith(Of Vendor)(Function(v) v.ProductVendors)
        ladeOptionen.LoadWith(Of ProductVendor)(Function(p) p.Product)
        dataContext.LoadOptions = ladeOptionen
    End If
End Sub
```

¹¹ Merken Sie sich die Begriff am besten so: lazy → faul – westfälischer Spruch dazu: »Kommse heut nich, kommse moagn«. Eager → eifrig. »Streber« auf Englisch heißt »eager beaver«.

```

Dim LieferantenListe = From lieferantItem In dataContext.Vendors _
    Where lieferantItem.Name.StartsWith("H") _
    Order By lieferantItem.Name

'Rausfinden: Welche Produkte liefert der Lieferant
For Each Lieferant In LieferantenListe
    Console.WriteLine(Lieferant.AccountNumber & ": " & Lieferant.Name)
    Console.WriteLine(New String("c", 70))
    For Each produkt In Lieferant.ProductVendors
        Console.Write("ID:" & produkt.ProductID & " ")
        Console.WriteLine("Name:" & produkt.Product.Name)
    Next
    Console.WriteLine()

Next

Console.WriteLine()
Console.WriteLine("Taste drücken, zum Beenden")
Console.ReadKey()
End Sub

End Module

```

Hier kann der Anwender zu Beginn der Anwendung bestimmen, ob er die zugeordneten Produkte der Lieferanten in einem Rutsch beim Laden der Lieferanten mit ermitteln lassen möchte. Will er das, wird die Steuervariable *verzögertes Laden* auf *False* gesetzt. Und falls das dann weiterhin der Fall ist (siehe fett markierter Abschnitt im Listing), definiert die Anwendung eine Instanz der Klasse *DataLoadOptions*. Mit Hilfe dieser Instanz können Sie definieren, welche Relationen Sie vom Standardverhalten des Lazy-Loadings auf Eager-Loading umstellen möchten. Im Beispiel machen wir das mit beiden Relationen, also mit Lieferanten → Lieferantenprodukte und Lieferantenprodukte → Produktdetails gleichermaßen. Diese Relationen werden übrigens wieder mit Lambda-Ausdrücken realisiert: Sie rufen die *LoadWith*-Methode wiederholt mit *dem* Typ als Typvariable auf, der die Mastertabelle abbildet und übergeben ihr eine Lambda-Funktion, die nichts weiter macht, als die entsprechende Details-Tabelle zurückzugeben).

Lassen Sie das Programm laufen, und beantworten Sie, um das Standardverhalten zu ändern, die Nachfrage der Anwendung mit *Ja*, sieht die generierte SQL-Anweisung schon ganz anders aus, und ist gerade für dieses Beispiel die weitaus bessere Alternative:

```

Produktliste zu jedem Lieferanten immer direkt laden? [J/N]j
SELECT [t0].[VendorID], [t0].[AccountNumber], [t0].[Name], [t0].[CreditRating],
[t0].[PreferredVendorStatus], [t0].[ActiveFlag], [t0].[PurchasingWebServiceURL],
[t0].[ModifiedDate], [t1].[ProductID], [t1].[VendorID] AS [VendorID2], [t1].[AverageLeadTime],
[t1].[StandardPrice], [t1].[LastReceiptCost], [t1].[LastReceiptDate],
[t1].[MinOrderQty], [t1].[MaxOrderQty], [t1].[OnOrderQty], [t1].[UnitMeasureCode],
[t1].[ModifiedDate] AS [ModifiedDate2], [t2].[ProductID] AS [ProductID2],
[t2].[Name] AS [Name2], [t2].[ProductNumber], [t2].[MakeFlag], [t2].[FinishedGoodsFlag],
[t2].[Color], [t2].[SafetyStockLevel], [t2].[ReorderPoint], [t2].[StandardCost],
[t2].[ListPrice], [t2].[Size], [t2].[SizeUnitMeasureCode], [t2].[WeightUnitMeasureCode],
[t2].[Weight], [t2].[DaysToManufacture], [t2].[ProductLine],
[t2].[Class], [t2].[Style], [t2].[ProductSubcategoryID], [t2].[ProductModelID], [t2].[SellStartDate],
[t2].[SellEndDate], [t2].[DiscontinuedDate], [t2].[rowg]

```

```

uid], [t2].[ModifiedDate] AS [ModifiedDate3], (
    SELECT COUNT(*)
    FROM [Purchasing].[ProductVendor] AS [t3]
    INNER JOIN [Production].[Product] AS [t4] ON [t4].[ProductID] = [t3].[ProductID]
    WHERE [t3].[VendorID] = [t0].[VendorID]
) AS [value]
FROM [Purchasing].[Vendor] AS [t0]
LEFT OUTER JOIN ([Purchasing].[ProductVendor] AS [t1]
    INNER JOIN [Production].[Product] AS [t2] ON [t2].[ProductID] = [t1].[ProductID]) ON [t1].[VendorID]
= [t0].[VendorID]
WHERE [t0].[Name] LIKE @p0
ORDER BY [t0].[Name], [t0].[VendorID], [t1].[ProductID]
-- @p0: Input NVarChar (Size = 2; Prec = 0; Scale = 0) [H%]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

HILLBIC0001: Hill Bicycle Center
=====
ID:524 Name:HL Spindle/Axle

HILLSBI0001: Hill's Bicycle Service
=====
ID:911 Name:LL Road Seat/Saddle
ID:912 Name:ML Road Seat/Saddle

HOLIDAY0001: Holiday Skate & Cycle
=====

HYBRIDB0001: Hybrid Bicycle Center
=====
ID:910 Name:HL Mountain Seat/Saddle

Taste drücken, zum Beenden

```

HINWEIS Falls Sie das Standardverhalten des *gesamten* Datenkontexts von Lazy- auf Eager-Loading ändern möchten, setzen Sie einfach seine DeferredLoadingEnabled-Eigenschaft auf False:

```
dataContext.DeferredLoadingEnabled = False
```

Trennen des Abfrageergebnisses vom Kontext

Wenn Sie mit den aus den Abfragen erhaltenen Daten weiterarbeiten möchten, ohne sie jedes Mal neu zu selektieren, wandeln Sie das Ergebnis der Abfrage in eine Liste oder ein Array um. Dieses erfolgt beispielsweise mit den Methoden `ToList` oder `ToArray` – wir haben das im LINQ to Objects-Kapitel bereits kennengelernt.

HINWEIS Das gilt für LINQ to SQL umso mehr, als dass Sie es nur in den wenigsten Fällen wirklich wünschen, dass die eigentliche SQL-Abfrage immer und immer wieder zur Ausführung an den SQL-Server übermittelt wird. In den meisten Fällen wird es reichen, die Abfrage ein Mal auszuführen, und die Ergebnisliste schließlich mit `ToArray` bzw. `ToList` von der eigentlichen Abfrage zu trennen.

```
Dim datacontext As New AdventureWorksDataContext  
Dim cust = From row In datacontext.Customers _  
    Where row.LastName = "Geist"  
datacontext.Log = Console.Out  
Dim custlist = cust.ToList
```

Sie können jetzt mit der Liste `custlist` arbeiten, ohne dass die Daten bei jedem Zugriff neu geladen werden.

ACHTUNG Wenn Sie die Daten auf diese Weise von der Liste getrennt haben, stehen die einzelnen Ergebnisobjekte natürlich auch nicht mehr unter der Schirmherrschaft des Datenkontexts. Achten Sie also darauf, dass Sie nicht den Fehler begehen, Änderungen in den losgelösten Objekten vorzunehmen und zu erwarten, dass diese Objekte dann mit den entsprechenden Anweisungen wieder den Weg in die Datenbank finden. Sie werden es nämlich nicht!

Daten verändern, speichern, einfügen und löschen

Auf welche Weise Sie Daten von einer SQL-Server-Datenbank abfragen, haben die letzten Abschnitte nun von verschiedenen Seiten beleuchtet. Doch in einer Datenbankanwendung ist das natürlich nur die halbe Miete, denn Daten müssen auch geändert werden.

Auch dabei hilft Ihnen LINQ to SQL, und unterscheidet sich bei seiner prinzipiellen Arbeitsweise gar nicht so sehr von ADO.NET, denn: Wann immer Sie Daten innerhalb einer Auflistung, die an einen Datenkontext gebunden sind, ändern, merkt sich der Datenkontext nicht nur die neuen, sondern auch die alten Eigenschaften. In einer Liste von Geschäftsobjekten müssen dann natürlich bei einem Update der Daten nicht alle Daten dieser Liste zurück in den SQL Server übertragen werden, sondern nur die Elemente, deren Eigenschaften sich auch verändert haben. Da der Datenkontext aber alte und neue Werte kennt, kann er diese Entscheidung selbstständig treffen.

In Anlehnung daran funktioniert das auch mit dem Einfügen neuer Datensätze oder dem Löschen von Daten. Vorgänge, die Sie in den gemappten Business-Objekten durchführen, werden, sobald Sie es sagen, in den SQL Server repliziert.

Auch das Einfügen von verknüpften Daten handelt der Datenkontext, ohne Probleme zu machen: Wenn Sie einen neuen Lieferanten hinzufügen wollen, dann erweitern Sie die Adressauflistung und den Lieferanten. Das Übertragen des Fremdschlüssels in die jeweilige Tabelle läuft auch dabei im Hintergrund völlig automatisiert ab.

Und schließlich gibt es entsprechende Überprüfungen für den so genannten Concurrency-Check; wenn in einer Netzwerkumgebung gleiche Datensätze verändert und nun zurückgeschrieben werden – welche Instanz gewinnt dann, und wie soll Ihre Anwendung darauf reagieren. Eine entsprechende Infrastruktur, um auch solche Fälle sauber zu behandeln, finden Sie ebenfalls in den folgenden Abschnitten beschrieben.

Datenänderungen mit SubmitChanges an die Datenbank übermitteln

Schauen wir uns das Beispiel an, das sich im Modul *DatensätzeÄndern* befindet.

HINWEIS Denken Sie auch hier wieder daran, das Startobjekt auf das Modul *DatensätzeÄndern* im Beispielprojekt zu ändern (gemäß Abbildung 34.34 und der entsprechenden dort zu findenden Beschreibung), damit dieses Beispiel ausgeführt wird.

Das folgende Beispiel ruft einen Lieferanten ab (und zwar den Einzigen, der mit der Zeichenfolge »Hybrid« beginnt), liefert Ihnen diesen aber nicht als Auflistung mit einem Element, sondern durch den Einsatz der Methode Single (siehe Listing) bereits als Instanz der Klasse Vendor:

```
Module DatensätzeÄndern
    ''' <summary>
    ''' Beispiel: Abfragen von Daten
    ''' zum Ändern eines Datensatzes
    ''' </summary>
    ''' <remarks></remarks>
    Sub Main()

        ' Neuen DataContext anlegen
        Dim datacontext As New AdventureWorksDataContext

        ' Sämtliche SQL-Kommandos auf Console.Out ausgeben
        datacontext.Log = Console.Out

        'Mit Single bekommt man genau das eine Element der Auflistung zurück,
        'oder eine Exception, falls es keines oder mehrere gab.
        Dim lieferant = (From lieferantItem In datacontext.Vendors
                         Where lieferantItem.Name.ToLower.StartsWith("hybrid")).Single

        Console.WriteLine("Gefunden: " & lieferant.Name)
        Console.WriteLine("wird nun geändert!")

        'Hin und her ändern, damit es nicht nur einmal funktioniert:
        If lieferant.Name = "Hybrid Bicycle Center" Then
            lieferant.Name = "Hybrid Fahrrad Center"
        Else
            lieferant.Name = "Hybrid Bicycle Center"
        End If

        'Änderungen zurückschreiben
        datacontext.SubmitChanges()

        Console.WriteLine()
        Console.WriteLine("Taste drücken, zum Beenden")
        Console.ReadKey()
    End Sub

End Module
```

Im Anschluss daran wird die Eigenschaft Name der Vendor-Klasse geändert. Und im Prinzip ist das auch schon alles, was Sie machen müssen, wenn Sie Änderungen von gemappten Tabellen durchführen möchten: Sie ändern lediglich, wie hier im Beispiel, die korrelierende Eigenschaft, die der Tabellenspalte entspricht.

Um die Änderungen anschließend in die Datenbank zurückzuschreiben, rufen Sie lediglich die Methode SubmitChanges des Datenkontextes auf – die dafür notwendigen T-SQL-UPDATE-Befehle generiert dann der LINQ to SQL-Provider selbstständig, wie der Bildschirmauszug des Beispiels demonstriert:

```

SELECT [t0].[VendorID], [t0].[AccountNumber], [t0].[Name], [t0].[CreditRating],
[t0].[PreferredVendorStatus], [t0].[ActiveFlag], [t0].[PurchasingWebServiceURL],
[t0].[ModifiedDate]
FROM [Purchasing].[Vendor] AS [t0]
WHERE LOWER([t0].[Name]) LIKE @p0
-- @p0: Input NVarChar (Size = 7; Prec = 0; Scale = 0) [hybrid%]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

Gefunden: Hybrid Fahrrad Center
wird nun geändert!
UPDATE [Purchasing].[Vendor]
SET [Name] = @p5
WHERE ([VendorID] = @p0) AND ([AccountNumber] = @p1) AND ([Name] = @p2) AND ([CreditRating] = @p3) AND
([PreferredVendorStatus] = 1) AND ([ActiveFlag] = 1) AND
([PurchasingWebServiceURL] IS NULL) AND ([ModifiedDate] = @p4)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [52]
-- @p1: Input NVarChar (Size = 11; Prec = 0; Scale = 0) [HYBRIDB0001]
-- @p2: Input NVarChar (Size = 20; Prec = 0; Scale = 0) [Hybrid Fahrrad Center]
-- @p3: Input TinyInt (Size = 1; Prec = 0; Scale = 0) [1]
-- @p4: Input DateTime (Size = 0; Prec = 0; Scale = 0) [25.02.2002 00:00:00]
-- @p5: Input NVarChar (Size = 21; Prec = 0; Scale = 0) [Hybrid Bicycle Center]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

```

Taste drücken, zum Beenden

Die SELECT-Abfrage ganz am Anfang des Auszugs ermittelt den Datensatz aus der Lieferantentabelle, die UPDATE-Anweisung sorgt dann dafür, dass sich die Änderungen an unserem gemappten Business-Objekt auch in der Datenbank widerspiegeln.

Einfügen von Datensätzen mit InsertOnSubmit

Das Einfügen von Datensätzen funktioniert bei LINQ to SQL nicht, wie vielleicht erwartet, weil Sie es vielleicht bei ADO.NET so kennen gelernt haben: Sie werden vergeblich nach einer Add-Methode an einer Business-Objektauflistung, die auf Table(Of) basiert, suchen, mit der Sie einen neuen Datensatz hinzufügen können. Und Sie bekommen auch keine Instanz einer Datenzeile mit NewLine.

Es ist aber auch nicht schwerer, diese Aufgabe mit LINQ to SQL zu lösen, nur anders. Und im Grunde genommen noch einfacher: Sie erstellen eine zunächst völlig losgelöste Instanz Ihres gemappten Business-Objektes, fügen dieses mit InsertOnSubmit der Auflistung hinzu, und wenn dann das nächste SubmitChanges erfolgt, generiert LINQ to SQL für Sie die entsprechenden T-SQL-INSERT-Anweisungen, die die neuen Datensätze in der Datenbank gemäß der Vorgaben in die Business-Objekte einfügen, etwa wie folgt:

```

Sub DatensätzeEinfügen()
    ' Neuen DataContext anlegen
    Dim datacontext As New AdventureWorksDataContext

    ' Sämtliche SQL-Kommandos auf Console.Out ausgeben
    datacontext.Log = Console.Out

```

```
'Einen Datensatz einfügen
Dim neuerLieferant As New Vendor With {.Name = "ActiveDevelop", .CreditRating = 1,
                                         .AccountNumber = "ACTDEV00001", .ModifiedDate = Now}

'Der Tabelle der Lieferanten hinzufügen
datacontext.Vendors.InsertOnSubmit(neuerLieferant)

'Änderungen zurückschreiben
datacontext.SubmitChanges()

Console.WriteLine()
Console.WriteLine("Taste drücken, zum Beenden")
Console.ReadKey()
End Sub
```

Auch hier können wir wieder beobachten, welche T-SQL-Anweisungen die Ausführung dieser Routine zur Folge hat:

```
INSERT INTO [Purchasing].[Vendor]([AccountNumber], [Name], [CreditRating], [PreferredVendorStatus],
[ActiveFlag], [PurchasingWebServiceURL], [ModifiedDate])
VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6)

SELECT CONVERT(Int,SCOPE_IDENTITY()) AS [value]
-- @p0: Input NVarChar (Size = 11; Prec = 0; Scale = 0) [ACTDEV00001]
-- @p1: Input NVarChar (Size = 13; Prec = 0; Scale = 0) [ActiveDevelop]
-- @p2: Input TinyInt (Size = 1; Prec = 0; Scale = 0) [1]
-- @p3: Input Bit (Size = 0; Prec = 0; Scale = 0) [False]
-- @p4: Input Bit (Size = 0; Prec = 0; Scale = 0) [False]
-- @p5: Input NVarChar (Size = 0; Prec = 0; Scale = 0) [Null]
-- @p6: Input DateTime (Size = 0; Prec = 0; Scale = 0) [10.11.2008 18:47:04]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1
```

Taste drücken, zum Beenden

Ziemlich einfach, finden Sie nicht?

TIPP Verwenden Sie die Methode `InsertAllOnSubmit`, wenn Sie eine Auflistung aus entsprechenden Business-Objekten übergeben wollen, die alle in einem Rutsch eingefügt werden sollen.

Datensätze in Tabellen mit Foreign-Key-Relations einfügen

Viele Szenarien verlangen allerdings, dass Daten nicht nur in einer, sondern in einer weiteren, in Relation zur ersten stehenden Tabelle eingefügt werden. In unserem Fall ist das sogar bei gleich drei Tabellen der Fall. Die Vorgehensweise in einem solchen Fall zeigt das folgende Beispiel:

```
Sub VerknüpfteDatensätzeEinfügen()
    ' Neuen DataContext anlegen
    Dim datacontext As New AdventureWorksDataContext

    ' Sämtliche SQL-Kommandos auf Console.Out ausgeben
    datacontext.Log = Console.Out
```

```
'Einen Datensatz einfügen
Dim neuerLieferant As New Vendor With {.Name = "ActiveDevelop", .CreditRating = 1,
                                         .AccountNumber = "ACTDEV00001", .ModifiedDate = Now}

'Der Tabelle der Lieferanten hinzufügen
datacontext.Vendors.InsertOnSubmit(neuerLieferant)

'Das neue Produkt erstellen
Dim neuesProdukt As New Product With {.Name = "Visual Basic 2008 - Das Entwicklerbuch", _
                                         .ProductNumber = "", _
                                         .ListPrice = 59, _
                                         .ModifiedDate = Now, _
                                         .DiscontinuedDate = #12/31/2011#, _
                                         .SellStartDate = #12/10/2008#, _
                                         .SellEndDate = #12/31/2011#, _
                                         .SafetyStockLevel = 5, _
                                         .ReorderPoint = 2}

'Neues Lieferantenprodukt erstellen, das Produkt mit diesem,
'und dieses mit dem Lieferanten verknüpfen
neuerLieferant.ProductVendors.Add(New ProductVendor() _
                                         With {.Product = neuesProdukt, _
                                         .StandardPrice = 59, _
                                         .ModifiedDate = Now, _
                                         .UnitMeasureCode = "PC", _
                                         .AverageLeadTime = 19, _
                                         .MinOrderQty = 1, _
                                         .MaxOrderQty = 10})

'Alles zurückschreiben
datacontext.SubmitChanges()

Console.WriteLine()
Console.WriteLine("Taste drücken, zum Beenden")
Console.ReadKey()
End Sub
```

Sie sehen, dass Sie sich hier als Entwickler gar nicht großartig um Foreign-Key-Relations kümmern müssen – jedenfalls nicht um die Entscheidung, welche Datensätze der verschiedenen Tabellen Sie in welcher Reihenfolge zuerst aktualisieren. Wir erstellen hier im Beispiel zunächst einen neuen Lieferanten, dann eine Instanz, die die Produktdetails darstellt und schließlich fügen wir dem neuen Lieferanten über die entsprechende ProductVendors-Eigenschaft und damit über die korrelierenden Lieferantprodukte-Auflistung ein neues Lieferantprodukt hinzu, das die Produktdetails wieder als Eigenschaft aufnimmt.

Das Hinzufügen dieser Datensätze funktioniert, anders als beim Hauptdatensatz, wie bei jeder anderen Auflistung hier mit Add. Um die IDs der Fremdschlusseinträge brauchen wir uns dabei genau so wenig zu kümmern, wie um die Reihenfolge, in der wir die Eigenschaften der in Relation zueinander stehenden Tabellen zuweisen. Ein Blick auf die generierte SQL-Anweisung zeigt, dass LINQ to SQL uns diese Aufgabe komplett abnimmt:

```

INSERT INTO [Purchasing].[Vendor]([AccountNumber], [Name], [CreditRating], [PreferredVendorStatus],
[ActiveFlag], [PurchasingWebServiceURL], [ModifiedDate])
VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6)

SELECT CONVERT(Int,SCOPE_IDENTITY()) AS [value]
-- @p0: Input NVarChar (Size = 11; Prec = 0; Scale = 0) [ACTDEV00001]
-- @p1: Input NVarChar (Size = 13; Prec = 0; Scale = 0) [ActiveDevelop]
-- @p2: Input TinyInt (Size = 1; Prec = 0; Scale = 0) [1]
-- @p3: Input Bit (Size = 0; Prec = 0; Scale = 0) [False]
-- @p4: Input Bit (Size = 0; Prec = 0; Scale = 0) [False]
-- @p5: Input NVarChar (Size = 0; Prec = 0; Scale = 0) [Null]
-- @p6: Input DateTime (Size = 0; Prec = 0; Scale = 0) [10.11.2008 20:08:34]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

INSERT INTO [Production].[Product]([Name], [ProductNumber], [MakeFlag], [FinishedGoodsFlag], [Color],
[SafetyStockLevel], [ReorderPoint], [StandardCost], [ListPrice],
[Size], [SizeUnitMeasureCode], [WeightUnitMeasureCode], [Weight], [DaysToManufacture],
[ProductLine], [Class], [Style], [ProductSubcategoryID], [ProductModelID],
[SellStartDate], [SellEndDate], [DiscontinuedDate], [rowguid], [ModifiedDate])
VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8, @p9, @p10, @p11, @p12, @p13, @p14, @p15, @p16,
@p17, @p18, @p19, @p20, @p21, @p22, @p23)

SELECT CONVERT(Int,SCOPE_IDENTITY()) AS [value]
-- @p0: Input NVarChar (Size = 38; Prec = 0; Scale = 0) [Visual Basic 2008 - Das Entwicklerbuch]
-- @p1: Input NVarChar (Size = 0; Prec = 0; Scale = 0) []
-- @p2: Input Bit (Size = 0; Prec = 0; Scale = 0) [False]
-- @p3: Input Bit (Size = 0; Prec = 0; Scale = 0) [False]
-- @p4: Input NVarChar (Size = 0; Prec = 0; Scale = 0) [Null]
-- @p5: Input SmallInt (Size = 0; Prec = 0; Scale = 0) [5]
-- @p6: Input SmallInt (Size = 0; Prec = 0; Scale = 0) [2]
-- @p7: Input Money (Size = 0; Prec = 19; Scale = 4) [0]
-- @p8: Input Money (Size = 0; Prec = 19; Scale = 4) [59]
-- @p9: Input NVarChar (Size = 0; Prec = 0; Scale = 0) [Null]
-- @p10: Input NChar (Size = 3; Prec = 0; Scale = 0) [Null]
-- @p11: Input NChar (Size = 3; Prec = 0; Scale = 0) [Null]
-- @p12: Input Decimal (Size = 0; Prec = 8; Scale = 2) [Null]
-- @p13: Input Int (Size = 0; Prec = 0; Scale = 0) [0]
-- @p14: Input NChar (Size = 2; Prec = 0; Scale = 0) [Null]
-- @p15: Input NChar (Size = 2; Prec = 0; Scale = 0) [Null]
-- @p16: Input NChar (Size = 2; Prec = 0; Scale = 0) [Null]
-- @p17: Input Int (Size = 0; Prec = 0; Scale = 0) [Null]
-- @p18: Input Int (Size = 0; Prec = 0; Scale = 0) [Null]
-- @p19: Input DateTime (Size = 0; Prec = 0; Scale = 0) [10.12.2008 00:00:00]
-- @p20: Input DateTime (Size = 0; Prec = 0; Scale = 0) [31.12.2011 00:00:00]
-- @p21: Input DateTime (Size = 0; Prec = 0; Scale = 0) [31.12.2011 00:00:00]
-- @p22: Input UniqueIdentifier (Size = 0; Prec = 0; Scale = 0) [00000000-0000-0000-0000-000000000000]
-- @p23: Input DateTime (Size = 0; Prec = 0; Scale = 0) [10.11.2008 20:08:34]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

INSERT INTO [Purchasing].[ProductVendor]([ProductID], [VendorID], [AverageLeadTime], [StandardPrice],
[LastReceiptCost], [LastReceiptDate], [MinOrderQty], [MaxOrderQty], [OnOrderQty], [UnitMeasureCode],
[ModifiedDate])
VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8, @p9, @p10)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1007]
-- @p1: Input Int (Size = 0; Prec = 0; Scale = 0) [116]
-- @p2: Input Int (Size = 0; Prec = 0; Scale = 0) [19]
-- @p3: Input Money (Size = 0; Prec = 19; Scale = 4) [59]
-- @p4: Input Money (Size = 0; Prec = 19; Scale = 4) [Null]
-- @p5: Input DateTime (Size = 0; Prec = 0; Scale = 0) [Null]
-- @p6: Input Int (Size = 0; Prec = 0; Scale = 0) [1]
-- @p7: Input Int (Size = 0; Prec = 0; Scale = 0) [10]

```

```
-- @p8: Input Int (Size = 0; Prec = 0; Scale = 0) [Null]
-- @p9: Input NChar (Size = 3; Prec = 0; Scale = 0) [PC]
-- @p10: Input DateTime (Size = 0; Prec = 0; Scale = 0) [10.11.2008 20:08:34]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1
```

Taste drücken, zum Beenden

Die Vorgehensweise beim Erstellen der erforderlichen T-SQL-Anweisungen ist dabei wie folgt:

- Die INSERT INTO-Anweisung für den Hauptdatensatz wird erstellt und ausgeführt.
- SELECT CONVERT(Int,SCOPE_IDENTITY()) ermittelt dann den Auto-ID-Wert dieses neuen Datensatzes.
- Die INSERT INTO-Anweisung für das Einfügen des Datensatzes in der Produktdetails-Tabelle *Product* wird erstellt.
- Auch hier dient wieder SCOPE_IDENTITY zum Ermitteln des Auto-ID-Wertes des neuen Datensatzes.
- Schließlich wird die INSERT INTO-Anweisung für das Erstellen des Lieferantprodukt-Datensatzes ausgeführt, in die dann beide ermittelten IDs einfließen.

Daten löschen mit DeleteOnSubmit

Wie beim Einfügen von Daten kennt der Datenkontext auch eine spezielle Anweisung für das Löschen von Daten. Der Name dieser Methode lautet `DeleteOnSubmit`. Die Anwendung dieser Methode ist auch ein Kinderspiel, wie der folgende Listingausschnitt zeigt:

```
Sub DatensätzeLöschen()
    ' Neuen DataContext anlegen
    Dim datacontext As New AdventureWorksDataContext
    ' Erst mal löschen wir den Trigger, damit das Delete funktioniert
    VendorTriggerEntfernen(datacontext)

    ' Sämtliche SQL-Kommandos auf Console.Out ausgeben
    datacontext.Log = Console.Out

    'Den einen Datensatz ermitteln
    Dim activeDevelop = (From lieferantItem In datacontext.Vendors
                         Where lieferantItem.AccountNumber = "ACTDEV00001").Single

    'Datensatz löschen
    datacontext.Vendors.DeleteOnSubmit(activeDevelop)

    'Änderungen zurückschreiben
    datacontext.SubmitChanges()

    Console.WriteLine()
    Console.WriteLine("Taste drücken, zum Beenden")
    Console.ReadKey()
    VendorTriggerWiederHerstellen(datacontext)
End Sub
```

Hier wird der Datensatz, der gelöscht werden soll, zunächst mit einer LINQ-Abfrage ermittelt und über die Single-Methode als einzelne Business-Objekt-Instanz zurückgeliefert. Dieses Objekt dient anschließend der `DeleteOnSubmit`-Methode als Parameter, bestimmt also, welcher Datensatz in der Datentabelle des SQL Servers gelöscht werden soll. Dieser Löschevorgang wird auch erst wieder dann ausgeführt, sobald die Methode `SubmitChanges` des entsprechenden `DataContext`-Objektes aufgerufen wird, und diese sorgt dann wiederum für die Generierung der korrekten T-SQL-Anweisung, wie der folgende Bildschirmauszug beweist.

```

SELECT [t0].[VendorID], [t0].[AccountNumber], [t0].[Name], [t0].[CreditRating],
[t0].[PreferredVendorStatus], [t0].[ActiveFlag], [t0].[PurchasingWebServiceURL],
[t0].[ModifiedDate]
FROM [Purchasing].[Vendor] AS [t0]
WHERE [t0].[AccountNumber] = @p0
-- @p0: Input NVarChar (Size = 11; Prec = 0; Scale = 0) [ACTDEV00001]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

DELETE FROM [Purchasing].[Vendor] WHERE ([VendorID] = @p0) AND ([AccountNumber] = @p1) AND ([Name] =
@p2) AND ([CreditRating] = @p3) AND (NOT ([PreferredVendorS
tatus] = 1)) AND (NOT ([ActiveFlag] = 1)) AND ([PurchasingWebServiceURL] IS NULL) AND ([ModifiedDate] =
@p4)
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [106]
-- @p1: Input NVarChar (Size = 11; Prec = 0; Scale = 0) [ACTDEV00001]
-- @p2: Input NVarChar (Size = 13; Prec = 0; Scale = 0) [ActiveDevelop
] -- @p3: Input TinyInt (Size = 1; Prec = 0; Scale = 0) [1]
-- @p4: Input DateTime (Size = 0; Prec = 0; Scale = 0) [10.11.2008 15:44:29]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

```

Taste drücken, zum Beenden

TIPP Verwenden Sie die Methode `DeleteAllOnSubmit`, wenn Sie eine Auflistung aus entsprechenden Business-Objekten übergeben wollen, die alle in einem Rutsch gelöscht werden sollen.

HINWEIS Damit das Löschen in diesem Beispiel überhaupt funktionieren kann, gibt es zwei Methoden, die einen SQL-Trigger, der an der Lieferantentabelle hängt, löschen und später wieder einfügen – diese Tatsache soll uns aber nicht vom eigentlichen Problem ablenken. Die beiden dafür zuständigen Methoden im Quellcode senden zu diesem Zweck eigentlich nur eine DROP-Trigger- und später wieder eine CREATE TRIGGER-Anweisung an die SQL Server-Datenbank (Letztere steht übrigens in der Ressourcendatei). Zum Hintergrund: Die AdventureWorks-Datenbank ist standardmäßig so aufgebaut, dass Lieferanten aus der `Vendor`-Tabelle nicht physisch gelöscht sondern nur als gelöscht gekennzeichnet werden können – der standardmäßig aktivierte Trigger verhindert das physische Löschen der Datensätze. Den Trigger mit `ALTER TRIGGER triggername DEACTIVATE` kurz zu deaktivieren, wäre auch eine Alternative gewesen – so sehen Sie aber direkt, wie Trigger programmtechnisch erstellt werden können.

Concurrency-Checks (Schreibkonfliktprüfung)

Solange, wie Sie Daten nur von einem Client aus im SQL-Server geändert haben, wird Ihre Anwendung beim Aktualisieren von Daten kaum Probleme bekommen. Was geschieht aber, wenn dieselben Daten in einer Netzwerkumgebung von zwei Anwendern Ihrer Software gleichzeitig geändert wurden?

In diesem Fall wird eine ChangeConflictException ausgelöst, und zwar dann, wenn Sie die Änderungen mit SubmitChanges zurück zur Datenbank übermitteln wollen. LINQ überprüft nämlich beim Aktualisieren, ob der zu ändernde Datensatz noch in der Version vorliegt, in der er ursprünglich geladen wurde. Ist das nicht der Fall, wird die Ausnahme ausgelöst. Standardmäßig werden alle Felder einer Tabelle überprüft. Sie können dieses Verhalten jedoch mithilfe des O/R-Designers für jedes Feld einer Tabelle individuell einstellen:

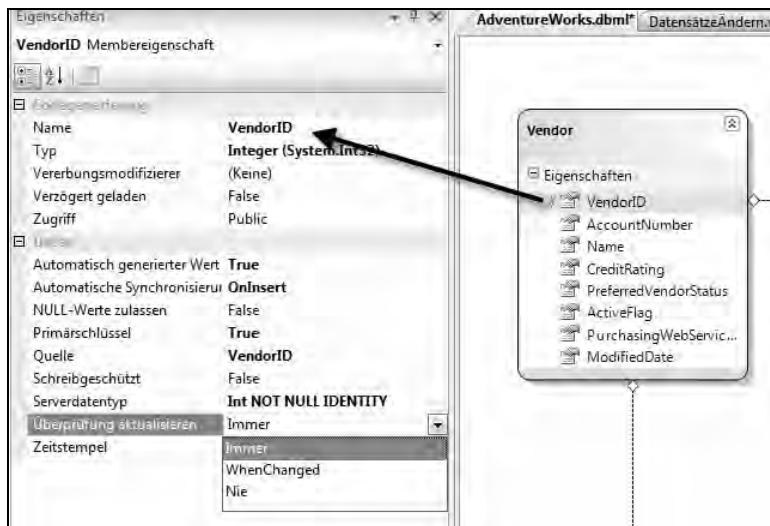


Abbildung 34.30 Ändern der Kollisionsüberprüfung eines Attributes

Wenn eine Tabelle einen Änderungszeitstempel besitzt oder eine Versionsnummer des Datensatzes gepflegt wird, können die Daten auch anhand dieser Angaben auf Eindeutigkeit geprüft werden. Hierzu muss die Zeitstempel Eigenschaft bei dem jeweiligen Attribut auf true gesetzt werden.

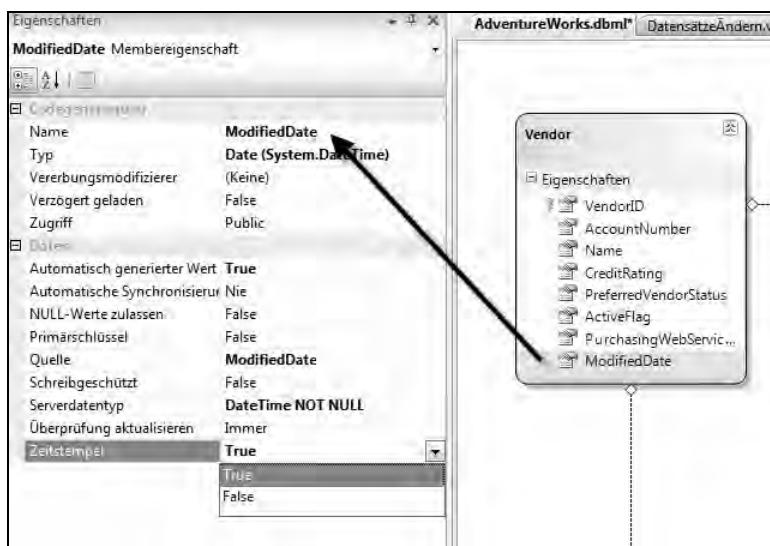


Abbildung 34.31 Verwenden eines Zeitstempels zur Kollisionsüberprüfung

HINWEIS In der Designer-Datei *AdventureWorks.designer.vb* spiegelt sich die Zeitstempel-Eigenschaft in dem ColumnAttribut unter IsVersion wider. Es handelt sich also letzten Endes um die gleiche Eigenschaft, die im Designer nur »über Umwege« mit anderem Namen angesprochen wird.

WICHTIG Wird bei einem Feld eine Tabelle Zeitstempel auf true gesetzt, werden für alle anderen Felder die Update Check-Angaben ignoriert. Zudem wird automatisch die Eigenschaft Automatisch generierter Wert auf True gesetzt. LINQ übernimmt nun die Aktualisierung des Feldes.

Ändern Sie für unser Beispiel im O/R Designer das ModifiedDate der Tabelle Vendor. Setzen Sie die Eigenschaft Zeitstempel auf true. Verwenden Sie die folgenden Zeilen Code (im Beispiel in der Datei *ConcurrencyCheck.vb* zu finden):

```
' Neuen DataContext anlegen
Dim datacontext As New AdventureWorksDataContext

' Sämtliche SQL-Kommandos auf Console.Out ausgeben
datacontext.Log = Console.Out

Dim electronicBikeRepair = (From lieferant In datacontext.Vendors
                           Where lieferant.Name = "Electronic Bike Repair & Supplies").First

electronicBikeRepair.CreditRating += CByte(1)

Console.WriteLine()
Console.WriteLine("Verwenden Sie das Management Studio und führen folgendes Update aus:")
Console.WriteLine("update AdventureWorks.Purchasing.Vendor")
Console.WriteLine("set CreditRating=CreditRating+1,")
Console.WriteLine("ModifiedDate = GETDATE()")
Console.WriteLine("where Name = 'Electronic Bike Repair & Supplies'")
Console.WriteLine("Ist dieses erfolgt, drücken Sie bitte Return. Sie werden eine Fehlermeldung erhalten.")
Console.ReadKey()

datacontext.SubmitChanges()
' die folgenden Zeilen werden nur ausgeführt, falls das Update nicht durchgeführt wurde.
Console.WriteLine("Die Änderungen wurden gespeichert. Bitte Return drücken, um Programm zu beenden.")
Console.ReadKey()
```

Die Ausgabe im Konsolenfenster schaut dann folgendermaßen aus:

```
SELECT TOP (1) [t0].[VendorID], [t0].[AccountNumber], [t0].[Name], [t0].[CreditRating], [t0].[PreferredVendorStatus], [t0].[ActiveFlag], [t0].[PurchasingWebServiceURL], [t0].[ModifiedDate]
FROM [Purchasing].[Vendor] AS [t0]
WHERE [t0].[Name] = @p0
-- @p0: Input NVarChar (Size = 33; Prec = 0; Scale = 0) [Electronic Bike Repair & Supplies]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1
```

Verwenden Sie das Management Studio und führen folgendes Update aus:
 update AdventureWorks.Purchasing.Vendor
 set CreditRating=CreditRating+1,

```
ModifiedDate = GETDATE()
where Name = 'Electronic Bike Repair & Supplies'
Ist dieses erfolgt, drücken Sie bitte Return. Sie werden eine Fehlermeldung erhalten..
```

Eine weitere Möglichkeit, die Kollisionsüberprüfung zu steuern, liegt in der Methode SubmitChanges selbst. Es existiert eine überladene Version, bei der eine ConflictMode-Einstellung angegeben werden kann.



Abbildung 34.32 Mögliche ConflictModes-Einstellungen

Mögliche Werte sind:

- **ContinueOnConflict:** Es bedeutet *nicht*, dass Fehler durch parallele Zugriffe ignoriert werden, sondern es ist vielmehr so, dass auftretende Kollisionen quasi »gesammelt« und erst zum Schluss durch eine entsprechende Ausnahme »bekannt gegeben« werden.
- **FailOnFirstConflict:** Schon beim Auftreten des *ersten* Konfliktes wird eine Ausnahme ausgelöst.

Transaktionen

Standardmäßig werden Änderungen an die Datenbank durch den Datenkontext automatisch als Transaktion übermittelt, falls keine explizite Transaktion im Gültigkeitsbereich vorgefunden wird. Um eine übergreifende Transaktion zu nutzen, stehen zwei Möglichkeiten zur Verfügung.

TransactionScope (Transaktionsgültigkeitsbereich)

Um Transaktionen mit TransactionScope nutzen zu können, muss die `System.Transactions.dll` Assembly in das Projekt aufgenommen werden.

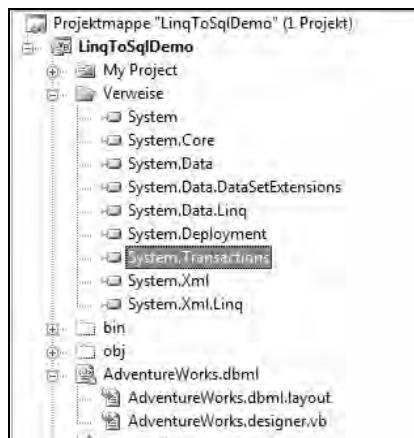


Abbildung 34.33 Einbinden der `System.Transactions.dll` Assembly

```

' neuen Transaktionsgültigkeitsbereich anlegen
Dim ts As New Transactions.TransactionScope
Using ts
    ' CreditRating einlesen, ändern und speichern
    Dim electronicBikeRepair = (From lieferant In datacontext.Vendors
                                 Where lieferant.Name = "Electronic Bike Repair & Supplies").First
    electronicBikeRepair.CreditRating = 1
    datacontext.SubmitChanges()

    ' den Transaktionsgültigkeitsbereich beenden
    ' also das Commit absetzen.
    ts.Complete()
    Console.WriteLine("Die Änderungen wurden gespeichert. Bitte Return drücken, um Programm zu
beenden.")
    Console.ReadKey()
End Using

```

Hier wird das obige Beispiel nochmals innerhalb eines Transaktionsgültigkeitsbereiches durchgeführt.

`TransactionScope` sorgt automatisch für das Durchführen eines Rollbacks, falls Fehler in der Transaktion auftreten. Sie müssen lediglich dafür sorgen, dass zum richtigen Zeitpunkt die `Complete`-Methode des `TransactionScope`-Objekts aufgerufen wird, um die geänderten Daten in der Datenbank mit einem *Commit* festzuschreiben.

Verwenden der Transaktionssteuerung des DataContext

Gerade für Anwendungen die noch mehr ADO.NET-orientiert sind, existiert eine weitere Möglichkeit mit Transaktionen zu arbeiten.

`DataContext` besitzt eine `Transaction`-Eigenschaft. Es ist ebenso möglich, über diese Eigenschaft Transaktionen zu steuern. Jedoch ist hier ein deutlicher Mehraufwand notwendig – der folgende Codeauszug soll das exemplarisch demonstrieren (zur Ausführung müssen Sie die Prozedur `main_ADO` in `Transaktion.vb` in `main` umbenennen):

```

Dim datacontext As New AdventureWorksDataContext
datacontext.Connection.Open()
datacontext.Transaction = datacontext.Connection.BeginTransaction()
datacontext.Log = Console.Out
Try

    ' CreditRating einlesen, ändern und speichern
    Dim electronicBikeRepair = (From lieferant In datacontext.Vendors
                                 Where lieferant.Name = "Electronic Bike Repair & Supplies").First
    electronicBikeRepair.CreditRating = 1
    datacontext.SubmitChanges()

    datacontext.Transaction.Commit()
    Console.WriteLine("Die Änderungen wurden gespeichert. Bitte Return drücken, um Programm zu
beenden.")
    Console.ReadKey()
Catch ex As Exception
    datacontext.Transaction.Rollback()

```

```
Finally  
    datacontext.Transaction = Nothing  
End Try
```

Als Erstes muss eine neue Transaktion eingeleitet werden. Dazu wird auf dem Connection-Objekt des Data-Contexts die BeginTransaction-Methode aufgerufen. Sie erstellt ein neues Transaktionsobjekt. Änderungen am Datenbankhaushalt müssen nun in einem Try-Catch-Block durchgeführt werden. Im Fehlerfall wird eine Ausnahme ausgelöst. Sie muss abgefangen werden, um das Rollback durchzuführen.

Zudem sollte im Finally-Block das Transaktionsobjekt wieder auf Nothing gesetzt werden. So startet der Datenkontext automatisch eine neue Transaktion bei einem SubmitChanges, sofern nicht explizit über BeginTransaction eine neue Transaktion begonnen wird.

Was, wenn LINQ einmal nicht reicht

Unter Umständen kommen Sie einmal in eine Situation, in der Sie eine Select-Abfrage mit LINQ nicht abbilden können. In diesem Fall können Sie selbst das T-SQL-Kommando für SQL Server erstellen, um die Datenbank abzufragen. Das Schöne daran ist, dass Sie, nachdem die Daten geladen wurden, nicht auf die LINQ-Features zu verzichten brauchen.

Der DataContext stellt eine Methode namens ExecuteQuery bereit, mit dem beliebige T-SQL-Abfragen direkt an den SQL Server geschickt werden können:

```
Dim datacontext As New AdventureWorksDataContext  
datacontext.Log = Console.Out  
  
Dim allrows = From ds In datacontext.Vendors  
Dim cmd As Common.DbCommand = datacontext.GetCommand(allrows)  
Dim sel = cmd.CommandText & " where Name = {0}"  
Dim res = datacontext.ExecuteQuery(Of Vendor)(sel, New Object() {"Electronic Bike Repair &  
Supplies"})  
For Each ds In res  
    Console.WriteLine("Electronic Bike Repair & Supplies' hat die VendorID {0}", ds.VendorID)  
Next  
Console.WriteLine("Bitte Return drücken, um Programm zu beenden.")  
Console.ReadKey()
```

Wir definieren hier im Beispielcode eine Abfrage auf der Tabelle Vendors und speichern sie in allrows. DataContext stellt die Methode GetCommand zur Verfügung, mit der man das generierte Select-Kommando erfragen kann.

Dieses wird als cmd gespeichert. Der CommandText enthält das eigentliche Select-Kommando im Klartext.

Wir erweitern es um

```
" where Name = {0}"      ' Name muss groß geschrieben werden
```

Bei {0} handelt es sich um einen Parameter, und zwar genauer gesagt, um den ersten Parameter, welcher der Abfrage übergeben wurde. Den zweiten erreichen Sie mit {1}, usw. Beim Aufrufen der Methode ExecuteQuery muss die Klasse angegeben werden, die einen Datensatz der Tabelle aufnehmen kann. In diesem Fall die

Klasse `Vendor` (demgegenüber steht `Vendors`, welches eine *Tabelle* von `Vendor`-Instanzen ist). Als Parameter wird das `Select`-Kommando übergeben, sowie ein Objekt-Array, das die im `Select` festgelegten Parameter in der entsprechenden Reihenfolge enthält (die Parameter können ebenso kommassepariert – also ohne Verwendung eines Arrays – übergeben werden). In diesem Fall wird nur ein Parameter vom Typ `String` übergeben, nämlich »Electronic Bike Repair & Supplies«. Als Ergebnis erhalten wir:

```
'Electronic Bike Repair & Supplies' hat die VendorID 2
```

Die Methode `ExecuteQuery` ist für Datenbankabfragen gedacht. Für Datenbankänderungen existiert die Methode `ExecuteCommand` des `DataContext`.

Als Beispiel:

```
Dim datacontext As New AdventureWorksDataContext  
datacontext.Log = Console.Out  
  
datacontext.ExecuteCommand("update AdventureWorks.Purchasing.Vendor set CreditRating={0} where  
Name = {1}", 3, "Electronic Bike Repair & Supplies")
```

LINQ-Abfragen dynamisch aufbauen

Bei all den Vorteilen, die Sie bisher im Rahmen von LINQ kennengelernt haben, bleibt bei vielen Entwicklern doch immer noch ein kleiner Beigeschmack: Dadurch, dass LINQ-Abfragen typsicher sind, können sie grundsätzlich erst einmal nicht dynamisch aufgebaut werden, sie sind statisch.

Sie können also keine Abfrage formulieren, die beispielsweise folgendermaßen aufgebaut ist:

```
Dim whereList As String() = {"Name.StartsWith("H")"}  
Dim OrderByList As String() = {"Name"}  
  
Dim whereText = "lieferanten.name.StartsWith("H")"  
Dim orderText = "lieferanten.name"  
  
Dim Ergebnisliste = From lieferanten In datacontext.Vendors _  
    Where whereText _  
    Order "Option Strict On" lässt keine impliziten Konvertierungen von String in Boolean zu.
```

Abbildung 34.34 Dynamische Abfragen – also solche, die anstelle der festen Typen beispielsweise Zeichenketten übernehmen – lassen sich in der LINQ Standardimplementierung nicht erstellen

Im Grunde genommen ist das eigentlich gar nicht so dramatisch, da LINQ-Abfragen kaskadierbar sind. Man kann durchaus im Rahmen von `Select Case`-Blöcken den Abfragebaum stetig erweitern. Das geht aber nur solange gut, wie Ihnen die Schemainformationen des Objektes, das Sie zur Laufzeit abfragen wollen, schon zur Entwurfszeit bekannt sind. Es hört dann auf, wenn Sie eine Abfrage so formulieren wollen, dass sowohl das Objekt als auch die angegebenen Parameter frei definierbar sind, und Sie diese Abfrage dann vielleicht noch in einer Methode einer Bibliothek auslagern möchten.

```
Public Function DynamicWhereAndOrderQuery (ByVal queryObject As LinqAbfragbar, _
    ByVal WhereList As List(Of String),
    ByVal OrderByList As List(Of String)) As LinqAbfragbar
    .
    .
    .
    Return Ergebnisliste
End Function
```

Sie sehen, dass es hier gleich zwei Probleme gibt:

- Wie lösen wir das Problem, beliebige, in LINQ abfragbare Objekte zu bestimmen? – Denn es gibt natürlich keine Basisklasse die LinqAbfragbar heißt, so wie es der Codeblock hier nur verdeutlichen will.
- Wie können wir die Parameter zum Selektieren durch Where oder zum Sortieren durch Order By in die LINQ-Abfrage »reinbekommen«, da diese ja keine Zeichenkette sondern nur konkrete Typen erwartet?

Abfragen und IQueryable

Objektauflistungen, die Sie mit LINQ abfragen können, müssen `IEnumerable` implementieren – das haben wir bereits im LINQ to Objects-Kapitel gelernt. Wenn die eigentliche Abfrage aber an einen anderen Anbieter ausgelagert werden soll (wie hier im Fall von LINQ to SQL an den SQL ADO.NET-Datenprovider), dann ändert sich diese Anforderung: In diesem Fall muss eine Auflistungsklasse `IQueryable` implementieren.

Und das ist auch so bei allen Auflistungsklassen, die im Rahmen von LINQ to SQL mithilfe des O/R-Designers entstehen. Jede gemappte Tabelle basiert zunächst einmal auf einem generierten Business-Objekt, das dem Schema der Tabelle entspricht; die *Zeilen* werden dann durch die generische Auflistung `Table(Of)` dargestellt. Und eben diese generische Auflistung implementiert die Schnittstelle `IQueryable`.

Um also eine Methode zu entwerfen, der man beliebige abfragbare Auflistungen übergeben kann, müssen wir nur definieren (um typsicher zu bleiben), von welchem Typ die Elemente sein sollen, die in der Abfrage liste enthalten sein sollen, und dass die Liste, die die Elemente enthält, auf jeden Fall die Schnittstelle `IQueryable` implementieren muss.

Wenn Sie sich das Generics-Kapitel dieses Buchs zu Gemüte geführt haben, dann wissen Sie, dass wir solche Typrestriktionen mit *Type Constraints* (Typbeschränkungen) erreichen können.

Unser Methodenrumpf kann also folgendermaßen ausschauen:

```
Public Function DynamicWhereAndOrderQuery(Of baseQueryType,
    baseQueryListType As IQueryable(Of baseQueryType)) _
    (ByVal queryObject As baseQueryListType,
    ByVal WhereList As IEnumerable(Of String),
    ByVal OrderByList As IEnumerable(Of String)) As IQueryable(Of baseQueryType)
```

Wir definieren hier unsere Methode so, dass sie als generische Methode aufgerufen werden muss, und diese generische Methode definiert zunächst zwei Typen: `baseQueryType` und `baseQueryListType`. Wenn wir also unsere Lieferantentabelle abfragen wollen, dann ...

- ... entspricht `baseQueryType` dem Typ `Vendor`
- ... und `baseQueryListType` einem `IQueryable(Of Vendor)` – `Table(Of Vendor)` tut dieser Vorschrift, so wie wir es benötigen, genüge.

Durch diese generische Definition der Methode erreichen wir, dass wir eine typsichere Auflistung zurückbekommen können, genau so, wie auch jede herkömmliche LINQ-Abfrage an einen Datenprovider wieder eine Auflistung vom Typ `IQueryable(Of Type)` hervorruft, damit diese wieder abfragbar bleibt und sich Abfragen kaskadieren lassen.

Dynamische Abfragen durch `DynamicExtension.vb`

Gut versteckt vor Blicken von der Außenwelt liefert Microsoft in einem LINQ-Beispiel zur Visual Studio-Hilfe ein Modul namens `DynamicExtensions.vb` aus, das die Schnittstelle `IQueryable(Of Type)` um einige sehr brauchbare Methoden erweitert:¹² Nämlich der Fähigkeit, Abfrageausdrücke bzw. die Parameter dazu als Zeichenketten zu übergeben. Damit lassen sich diese zwar nicht mehr als konkrete LINQ-Abfragen formulieren, aber sie lassen sich immerhin dynamisch formulieren. Dadurch dass Abfragen kaskadierbar sind, können Sie LINQ-Abfragen auch in statische und dynamische mischen, sodass der Nachteil, auf Sprachen-LINQ zu verzichten, nicht ganz so schwer wiegt.

Unser Beispiel, das demonstrieren soll, wie diese Art von dynamischer LINQ-Abfragengestaltung funktioniert, sieht mit dieser zweiten Voraussetzungen dann folgendermaßen aus:

```
Imports System.Data.Linq

Module DynamischeAbfragen

    Sub main()

        Dim datacontext As New AdventureWorksDataContext

        Dim ladeOptionen = New DataLoadOptions
        ladeOptionen.LoadWith(Of Vendor)(Function(v) v.ProductVendors)
        ladeOptionen.LoadWith(Of ProductVendor)(Function(p) p.Product)
        datacontext.LoadOptions = ladeOptionen

        'Und Debug-Ausgaben
        datacontext.Log = Console.Out

        Dim whereList As String() = {"Name.StartsWith("H")"}
        Dim OrderByList As String() = {"Name"}

        Dim LieferantenListe = DynamicWhereAndOrderQuery(Of Vendor,
            Table(Of Vendor))(datacontext.Vendors, whereList, OrderByList)

        'Rausfinden: Welche Produkte liefert der Lieferant
        For Each Lieferant In LieferantenListe
            Console.WriteLine(Lieferant.AccountNumber & ": " & Lieferant.Name)
            Console.WriteLine(New String("=", 70))
            For Each produkt In Lieferant.ProductVendors
                Console.Write("ID:" & produkt.ProductID & " ")
            Next
        Next
    End Sub
End Module
```

¹² Alle Beispiele zum Visual Studio, die über die Visual Studio-Hilfe angeboten werden bzw. auf die die Visual Studio-Hilfe verweist, finden Sie unter dem IntelliLink *F3415*.

```
Console.WriteLine("Name:" & produkt.Product.Name)
Next
Console.WriteLine()

Next
Console.ReadLine()
End Sub

Public Function DynamicWhereAndOrderQuery(Of baseQueryType,
                                             baseQueryListType As IQueryable(Of baseQueryType)) _
    (ByVal queryObject As baseQueryListType,
     ByVal WhereList As IEnumerable(Of String),
     ByVal OrderByList As IEnumerable(Of String)) As IQueryable(Of baseQueryType)

    'Where-String zusammenbauen
    Dim whereString = ""
    For c = 0 To WhereList.Count - 1
        whereString &= WhereList(c)
        If c < WhereList.Count - 1 Then
            whereString &= " AND "
        End If
    Next

    'OrderBy-String zusammenbauen
    Dim orderString = ""
    For c = 0 To OrderByList.Count - 1
        orderString &= OrderByList(c)
        If c < OrderByList.Count - 1 Then
            whereString &= ", "
        End If
    Next

    Return queryObject.Where(whereString).OrderBy(orderString)
End Function

End Module
```

Schön hier im Listing zu sehen ist die Tatsache, dass Sie auch komplette Visual Basic-Methodenaufrufe dynamisch in den entsprechenden Kriterien-String-Arrays übergeben können.

Wir haben mit dieser Art der Programmierung also erreicht, eine Funktionslibrary für beliebige Quelllisten abhandeln zu können, die obendrein noch dynamisch selektiert, sortiert und wie sonst noch ausgewertet werden können.

Und dass hier tatsächlich die notwendigen Selektierungen vom SQL-Server und nicht durch interne Selektierungen durchgeführt werden, zeigt die Ausgabe, die diese kleine Beispielanwendung erzeugt:

```
SELECT [t0].[VendorID], [t0].[AccountNumber], [t0].[Name], [t0].[CreditRating],
[t0].[PreferredVendorStatus], [t0].[ActiveFlag], [t0].[PurchasingWebServiceURL],
[t0].[ModifiedDate], [t1].[ProductID], [t1].[VendorID] AS [VendorID2], [t1].[AverageLeadTime],
[t1].[StandardPrice], [t1].[LastReceiptCost], [t1].[LastReceiptDate],
[t1].[MinOrderQty], [t1].[MaxOrderQty], [t1].[OnOrderQty], [t1].[UnitMeasureCode],
```

```
t1].[ModifiedDate] AS [ModifiedDate2], [t2].[ProductID] AS [ProductID]
], [t2].[Name] AS [Name2], [t2].[ProductNumber], [t2].[MakeFlag], [t2].[FinishedGoodsFlag],
[t2].[Color], [t2].[SafetyStockLevel], [t2].[ReorderPoint], [t2].[St
andardCost], [t2].[ListPrice], [t2].[Size], [t2].[SizeUnitMeasureCode], [t2].[WeightUnitMeasureCode],
[t2].[Weight], [t2].[DaysToManufacture], [t2].[ProductLine
], [t2].[Class], [t2].[Style], [t2].[ProductSubcategoryID], [t2].[ProductModelID], [t2].[SellStartDate],
[t2].[SellEndDate], [t2].[DiscontinuedDate], [t2].[rowg
uid], [t2].[ModifiedDate] AS [ModifiedDate3], (
    SELECT COUNT(*)
    FROM [Purchasing].[ProductVendor] AS [t3]
    INNER JOIN [Production].[Product] AS [t4] ON [t4].[ProductID] = [t3].[ProductID]
    WHERE [t3].[VendorID] = [t0].[VendorID]
) AS [value]
FROM [Purchasing].[Vendor] AS [t0]
LEFT OUTER JOIN ([Purchasing].[ProductVendor] AS [t1]
    INNER JOIN [Production].[Product] AS [t2] ON [t2].[ProductID] = [t1].[ProductID]) ON [t1].[VendorID]
= [t0].[VendorID]
WHERE [t0].[Name] LIKE @p0
ORDER BY [t0].[Name], [t0].[VendorID], [t1].[ProductID]
-- @p0: Input NVarChar (Size = 2; Prec = 0; Scale = 0) [H%]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 3.5.30729.1

HILLBIC0001: Hill Bicycle Center
=====
ID:524 Name:HL Spindle/Axle

HILLSBI0001: Hill's Bicycle Service
=====
ID:911 Name:LL Road Seat/Saddle
ID:912 Name:ML Road Seat/Saddle

HOLIDAY0001: Holiday Skate & Cycle
=====

HYBRIDB0001: Hybrid Bicycle Center
=====
ID:910 Name:HL Mountain Seat/Saddle
```

Kapitel 35

LINQ to Entities – Programmieren mit dem Entity Framework

In diesem Kapitel:

Voraussetzungen für das Verstehen dieses Kapitels	981
LINQ to Entities – ein erstes Praxisbeispiel	983
Abfrage von Daten eines Entitätsmodells	990
Daten verändern, speichern, einfügen und löschen	1004
Ausblick	1012

Als eine Entität bezeichnet man in der Datenmodellierung ein eindeutiges Objekt, dem Informationen zugeordnet werden können, und dass diese Begriffsdefinition nicht einer heftigen Abstraktheit entbehrt, impliziert schon, dass es sich bei LINQ to Entities um ein Verfahren handeln muss, das das, was es erreichen will, auf eine sehr flexible Weise macht.

Dem ist auch so. Nur: was macht es überhaupt?

Heutzutage werden viele datenorientierte Anwendungen auf der Grundlage von relationalen Datenbanken geschrieben. Diese Anwendungen müssen also irgendwann mit diesen Daten interagieren. Der logisch orientierte Aufbau von Datenbanktabellen und Datensichten eines üblichen Datenbanksystems, wie sie beispielsweise Microsofts SQL Server, Oracles Oracle Database oder IBMs DB2 darstellen, bietet allerdings nur eine suboptimale Grundlage, die konzeptionellen Modelle objektorientiert entwickelter Anwendungen abzubilden: *Impedance Mismatch*, in etwa: *objekt-relationale Unverträglichkeit*, ist der Grund, dass Abbildungsversuche zwischen den beiden verschiedenen Welten eigentlich immer unvollkommen sein müssen, wenn nicht eine wirkliche objektorientierte Datenbank als Grundlage für die Speicherung der Anwendungsobjekte vorhanden ist – und das ist bei den großen, genannten kommerziellen Datenbanksystemen eben nicht der Fall.

Das Entitätsdatenmodell (*Entity Data Modell*, kurz: *EDM*) kann hier Abhilfe schaffen, da es ein konzeptuelles Datenmodell beschreibt, welches es erlaubt, durch flexibles Mapping eine Brücke zwischen den beiden Welten zu bauen.

LINQ to SQL macht das im Ansatz ebenso; auch LINQ to SQL hilft dem Anwender, eine Reihe von Business-Objekten zu erstellen, die sich weitestgehend an den Tabellen- bzw. Sichtenschemata der Datenbank orientieren. Für den Entwickler eines Datenbanksystems mittlerer Größenordnung ist das sicherlich oftmals genau die Hilfe, die er benötigt, und in vielen Fällen ausreichend.

Doch die Zuordnungsmöglichkeiten zwischen dem logischen, relationalen Datenmodell der Datenbanken und dem konzeptionellen, objektorientierten Modell in Anwendungen gehen in LINQ to SQL einfach nicht weit genug, sowohl was die Zuordnung von Tabellen zu Businessobjekten anbelangt, als auch was die Abstraktion des oder der darunterliegenden Datenprovider betrifft, denn: LINQ to SQL kann lediglich mit der Microsoft SQL Server-Plattform umgehen (mit den Versionen SQL Server 2000, 2005 und 2008), und Tabellen und Sichten können lediglich 1:1 auf ein Objektmodell abgebildet werden. Der Abschnitt »*Objekt-relationale Unverträglichkeit – Impedance Mismatch*« des vorherigen Kapitels veranschaulicht diesen Sachverhalt an einem praktischen Beispiel.

Diese Unzulänglichkeiten löst LINQ to Entities, indem es eine weitere Schicht zwischen den generierten Business-Objekten und der ursprünglichen Datenbank einführt – die so genannte *konzeptionelle* Schicht. Dadurch wird LINQ to Entities gegenüber LINQ to SQL ungleich flexibler, aber auch nicht ganz so einfach zu handhaben wie LINQ to SQL.

Es gibt einige zusätzliche Konzepte, die man zum richtigen Nutzen kennen und begreifen muss. Diese Konzepte jedoch versteht und lernt man am besten mit der Methode Learning by doing. Und deswegen schauen wir uns in der folgenden Schritt-für-Schritt-Anleitung als erstes einmal an, wie man LINQ to Entities einsetzt, um ein konzeptionelles Entitätsmodell für eine Datenbank zu erstellen und dieses anschließend nutzt, um Daten abzufragen.

Die Elemente, die dabei verwendet werden, lernen Sie im entsprechenden Kontext kennen.

Voraussetzungen für das Verstehen dieses Kapitels

Um die recht komplexe Materie dieses Kapitels einfacher verstehen zu können, sind zwei grundsätzliche Dinge erforderlich.

- Sie sollten von technischer Seite her in der Lage sein, die folgenden Beispiele nachzuvollziehen. Dazu benötigen Sie eine erreichbare SQL Server-Instanz, sowie die Beispieldatenbank, die diesen Beispielen zugrunde liegt.
- Es gibt ein paar Fachbegriffe bzw. Grundkonzepte, die im Vorfeld verstanden sein sollten. Ich selber mag es eigentlich nicht, mir in Vorführungen erst stundenlang Theorieabhandlungen anhören zu müssen, bevor es dann an die eigentlichen Beispieldemos geht. Deswegen gelobe ich: So wenig wie möglich Theorie, aber – da müssen wir durch – so viel wie nötig. Aber nur so viel.

Technische Voraussetzungen

Die Beispiele, die Sie im Folgenden beschrieben finden, verwenden wie im vorherigen Kapitel die AdventureWorks-Beispieldatenbank von Microsoft auf Basis einer lokal installierten SQL Server Express with Tools-Instanz. Die Abschnitte »Microsoft SQL Server« sowie »Voraussetzungen für die Beispiele dieses und des nächsten Kapitels« sollten Sie deswegen unbedingt durcharbeiten, damit Sie die Voraussetzungen für die folgenden Abschnitte schaffen.

Für die Entscheidungsgrundlage, welche der beiden Techniken zum Entwerfen von Datenbankanwendungen (LINQ to SQL vs. LINQ to Entities) für Sie die Bessere ist, lesen Sie den Abschnitt »LINQ to SQL oder LINQ to Entities – was ist besser, was ist die Zukunft?« des vorherigen Kapitels.

ACHTUNG Auch wenn dieser Hinweis bereits in den gerade genannten Abschnitten des vorherigen Kapitels erfolgt ist, sei hier nochmals explizit darauf hingewiesen: LINQ to Entities ist erst ab Visual Studio 2008 mit Service Pack 1 verfügbar. Ohne die Installation von Service Pack 1 für Visual Basic 2008 gibt's auch kein LINQ to Entities.

Prinzipielle Funktionsweise eines Entity Data Model (EDM)

LINQ to Entities bezeichnet die Möglichkeit, mithilfe der Abfragetechnologie LINQ Daten-Entitäten abzufragen, die durch ein Entitätsdatenmodell (Entity Data Model) beschrieben werden. LINQ to Entities ist aber natürlich nicht das Entitätsdatenmodell, und LINQ to Entities beinhaltet auch nicht die Werkzeuge, dieses Modell zu erstellen.

Zuhause ist ein Entitätsdatenmodell im Entity Framework, oder genauer gesagt, im ADO.NET Entity Framework, und eine Elementvorlage dieses Typs fügen Sie auch ein, wenn Sie ein solches Modell Ihrem Projekt hinzufügen wollen.

Der »Modell«-Teil im Namen impliziert, dass hier ein komplexes Gebilde erschaffen wird, das weit über das hinausgeht, was wir bei LINQ to SQL kennengelernt haben. Bei LINQ to SQL gibt es die Schnittstelle zur Datenbank an einem Ende, das Businessmodell am anderen, und das war es.

Das Entity Framework möchte in seinem Modell sehr viel mehr erreichen:

- Der Datenprovider, der letzten Endes für die Kommunikation zwischen Ihrer Anwendung und dem Datenbankserver zuständig ist, soll austauschbar sein, ohne dass sich Abfragen gegen das Entity-Modell wirklich ändern müssen.
- Die Tabellen, die das Datenbankmodell liefert, sollen sich nicht ausschließlich 1:1 gegen die sich ergebenden Businessobjekte mappen lassen müssen, sondern *beliebig* zuordenbar sein.

Aus diesen Gründen basiert das Entity-Modell aus drei Einzelmodellen, nämlich aus dem ...

- ... **Physikalischen Speichermodell** (*Storage Model*), das all das abbildet, was sich aus den Schemainformationen der eigentlichen Entitäten (Tabellen, Sichten) der Datenbank ergibt und was zur Kommunikation mit der Datenplattform erforderlich ist.
- ... **Konzeptionellen Modell** (*Conceptual Model*), aus dem später die Business-Objekte generiert werden, und gegen das Sie mit LINQ to Entities abfragen.
- ... **Mapping**, das die Zuordnungen zwischen den beiden Modellen regelt und definiert.

WICHTIG Ganz wichtig zu wissen in diesem Zusammenhang: Diese drei Modelle (bzw. diese zwei Modelle und das Mapping) werden beim Erstellen eines Entitätsdatenmodels (EDM) in Form einer XML-Datei aufgebaut, die die Datei-Endung *.edmx* trägt – für *Entity Data Model eXtended Markup Language*. Beim Erstellen des Projektes landen sie automatisch in drei verschiedenen XML-Dateien, die als eingebettete Ressourcen in der .NET-Assembly abgelegt werden. Auf diese Weise sind die Definitionen zwar nicht frei zugänglich, können aber durch Austauschen der Ressourcen angepasst werden. Und nicht nur das: Mehrere dieser Definitionen können natürlich ebenfalls vorhanden sein, denn der eigentliche Vorgang des Mapping zwischen den beiden Modellen findet nicht zur Compile- sondern zur Laufzeit statt – wie diese drei Dateien dabei zur Verwendung »bestimmt« werden, sehen Sie gleich im Praxisteil!

Diese drei Dateien werden übrigens in der Standardeinstellung genau wie der Gesamtcontainer genannt, unterscheiden sich aber durch ihre Dateiendungen:

- **.CSDL:** Enthält die XML-Datei, die das konzeptionelle Modell beschreibt (*Conceptual Schema Definition Language*).
- **.SSDL:** Enthält die XML-Datei, die das physikalische Speichermodell beschreibt (*Store Schema Definition Language*).
- **.MSL:** Enthält die Mappingdatei (*Mapping Specification Language*).

TIPP **Einzelne Erzeugung der drei Modell-Dateien:** Falls Sie diese drei Dateien als Projekt-Output doch in separaten Dateien enthalten wollen, weil Sie sie beispielsweise für weitere Tools benötigen, öffnen Sie den EDM-Designer. Selektieren Sie das Modell selbst, indem Sie auf einen freien Bereich im Designer klicken. Im Eigenschaftenfenster stellen Sie anschließend die Eigenschaft *Verarbeitung der Metadatenartefakte* auf *In Ausgabeverzeichnis kopieren*.

In der deutschen SP1-Version von Visual Studio 2008 Development/Professional Edition erscheint dann übrigens eine Fehlermeldung, die wohl auf ein Lokalisierungsproblem der Eigennamen/-einstellungen hindeutet lässt. Die Ausgabe der drei Dateien erfolgt aber anschließend dennoch in das entsprechende Ausgabeverzeichnis (*bin/Debug/* bei der Projektgrundeinstellung).

Das ist übrigens ein weiterer Hinweis, dass es an den Tools zum Entity Framework noch Nachbesserungsbedarf gibt. Das gilt insbesondere für die gesamte Lokalisierung von SP1, die an vielen Stellen deutlich zu wünschen übrig lässt. Falls es Sie nicht stört, eine englische Version zu verwenden, nehmen Sie diese. Sie ist zurzeit einfach stabiler.

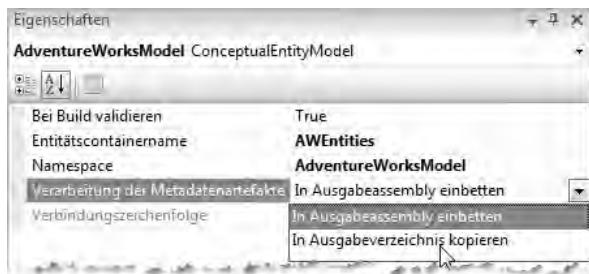


Abbildung 35.1 Um die Erstellung der drei Modell-Dateien ins Ausgabeverzeichnis zu forcieren, wählen Sie diese Einstellung. Ignorieren Sie die anschließende Fehlermeldung in der deutschen Version von Visual Studio – die Einstellung funktioniert!

Mit diesen Voraussetzungen wissen wir vorerst ausreichend viel, um mit dem Aufbau des ersten Praxisbeispiels beginnen zu können.

LINQ to Entities – ein erstes Praxisbeispiel

Lassen Sie uns der LINQ to Entities mit einem Praxisbeispiel nähern, anhand dessen wir dann die einzelnen Elemente erforschen können.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis

...\\VB 2008 Entwicklerbuch\\F - LINQ\\Kapitel 35\\LinqToEntitiesDemo

Denken Sie daran, dass Sie für das Nachvollziehen der Beispiele Zugriff auf eine Instanz von SQL Server 2008 bzw. SQL Server 2005 haben müssen und dort die *AdventureWorks*-Beispiele eingerichtet sind. Die entsprechenden Abschnitte im vorherigen Kapitel zeigen wie's geht.

1. Erstellen Sie ein neues Visual Basic-Projekt (als Konsolenanwendung).
2. Fügen Sie mithilfe des Projektmappen-Explorers ein neues Element in die Projektmappe ein – den entsprechenden Dialog erreichen Sie über das Kontext-Menü des Projektnamens –, und wählen Sie für das neue Element, wie in der Abbildung zu sehen, die Vorlage *ADO.NET Entity Data Model* aus.
3. Nennen Sie das neue Datenmodell *AdventureWorks.edmx*.

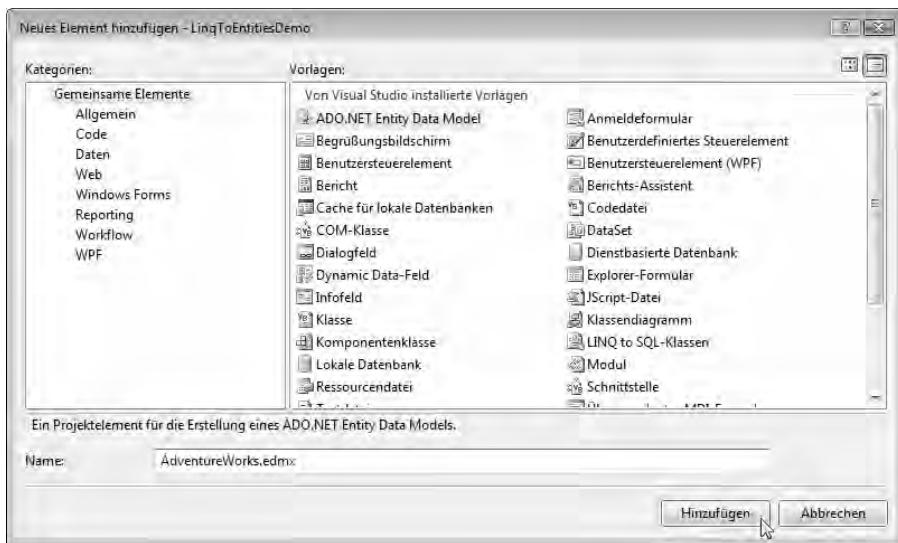


Abbildung 35.2 Hinzufügen eines ADO.NET Entity Data Models

4. Schließen Sie den Dialog mit Mausklick auf *Hinzufügen* ab.
5. Im nächsten Schritt zeigt Visual Studio Ihnen einen Assistenten, der Ihnen hilft, die Modellinhalte entweder aus einer Datenbank generieren zu lassen oder ein leeres Modell zu erzeugen. Wählen Sie hier *Aus Datenbank generieren*.

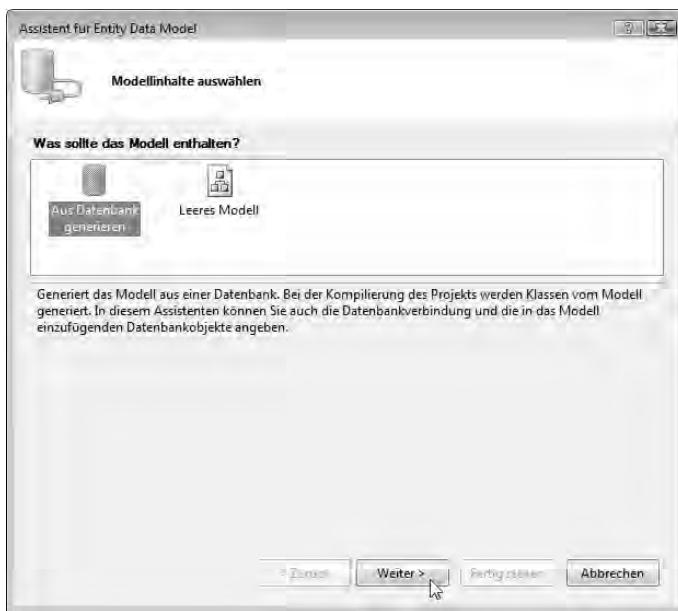


Abbildung 35.3 Bestimmen Sie im ersten Schritt des Assistenten, ob Sie ein Modell aus der Datenbank generieren lassen möchten oder ein leeres Modell verwenden

HINWEIS Falls Sie schon mit dem O/R-Designer von LINQ to SQL gearbeitet haben, stellen Sie spätestens an dieser Stelle den beschriebenen Unterschied zwischen den beiden Konzepten fest. Tatsächlich definieren Sie hier zunächst ein Modell, das dann als Vorlage für die Codegenerierung der Business-Objekte dient. Dieses Modell können Sie komplett selbst entwerfen, und dann Stückchen für Stückchen auf eine Datenquelle abbilden, oder eben ein Rahmenmodell aus einer Datenbank generieren lassen. Das ist – so ganz nebenbei erwähnt – eine wirklich bemerkenswerte Leistung des ADO.NET-Teams, denn schließlich müssen die Werkzeuge, die dieses Modell aus dem Datenbankschema erstellen, natürlich nicht nur mit SQL Server klarkommen. Funktionieren dabei müssen auch Informix, Oracle, DB2, MySQL ...¹

Quintessenz: Das Modell ist in erster Linie für die Generierung der Businessobjekte zuständig, nicht das Schema der Datenbank selbst. Eine erste Modellvorlage ergibt sich lediglich aus dem Datenbankschema.

6. Klicken Sie auf *Weiter*.



Abbildung 35.4 Bestimmen Sie in diesem Schritt die Entitätsverbindungszeichenfolge, und wie und ob diese Verbindungszeichenfolge hinterlegt werden soll

- In diesem Assistentenschritt bestimmen Sie nun die Verbindung zu Ihrem Datenprovider. Sobald weitere Treiber verfügbar werden, können Sie mithilfe der Schaltfläche *Neue Verbindung* auch andere Datenbank-Plattformen als den SQL Server-Datenprovider nutzen. Richten Sie Ihre Verbindung zur Datenbank-Instanz ein, und wählen Sie an dieser Stelle die Datenbank aus, die Sie als Grundlage der ersten Schemaerstellung Ihres neuen Entitätsmodells erstellen lassen wollen. Für unser Beispiel verwenden wir wieder die Instanz der AdventureWorks-Datenbank, deren Einrichtung am Anfang des vorherigen Kapitels beschrieben wurde.

¹ Treiber für andere Datenbankplattformen gibt es zwar zur Zeit der Drucklegung noch nicht von den jeweiligen Herstellern als finale Release-Versionen, die Beta-Versionen, die beispielsweise auf der PDC 2008 zu sehen waren, versprachen aber schon sehr viel!

HINWEIS Sie sehen jetzt in der Vorschau der Entitätsverbindungszeichenfolgenvorschau, wie die Schema- bzw. Spezifikationsdateien des Entitätsmodells und die physische Verbindung zur Datenbankinstanz in einer Verbindungszeichenfolge zusammengefasst werden. Die Verbindungszeichenfolge für das EDMs bestimmen also einerseits Konzeptionelles Modell, Speichermodell und Mapping und obendrein noch die verwendete Datenbank.

Sie können, falls Sie das wünschen, die Entitätsverbindungseinstellungen auch in der *App.Config*-Datei speichern, und brauchen sich dann später, beim Instanziieren des Objektkontextes, der u. a. den Verbindungsaufbau zur Datenbank-Instanz managet, nicht mehr um das Übergeben dieser ellenlangen Zeichenfolge zu kümmern. Sie sollten dann wiederum dieses Verfahren nur dann anwenden, wenn Sie keine sicherheitskritischen Anmeldeinformationen in der Verbindungszeichenfolge verwenden; falls Sie die integrierte Windows-Sicherheit zur Anmeldung an einem Microsoft SQL Server-System verwenden, befinden Sie sich auf der sicheren Seite. Sollten Sie SQL Server im Mixed Mode betreiben, und Anmeldeinformationen in der SQL Server-Verbindungszeichenfolge übergeben müssen, ist es besser, die Verbindungszeichenfolge in der Anwendung zu hinterlegen, oder die entsprechenden sicherheitsrelevanten Abschnitte der *app.config* zu verschlüsseln.²

8. Klicken Sie auf Weiter, um zum nächsten Schritt zu gelangen.

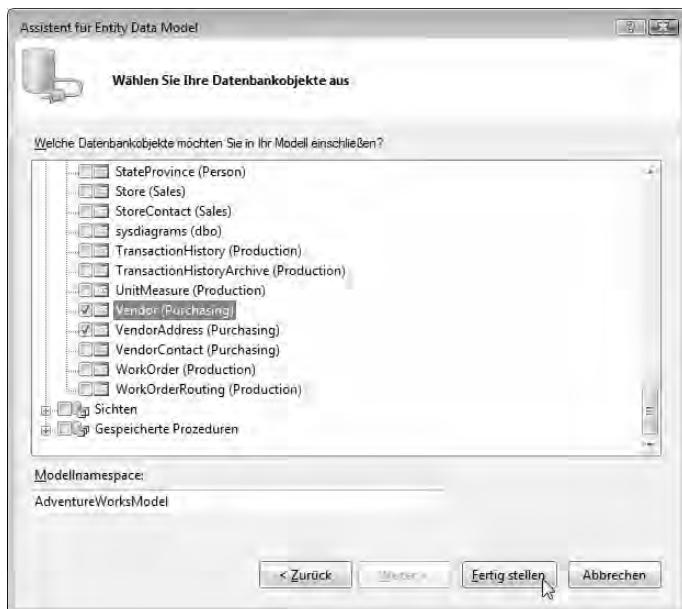


Abbildung 35.5 In diesem Schritt wählen Sie aus, welche Objekte als Grundlage für die Erstellung der Schema- und Mappingdateien dienen sollen

9. In diesem Schritt bestimmen Sie, welche Datenbankobjekte als Grundlage für die Erstellung der Schema- und Mappingdateien dienen sollen. Im einfachsten Fall entspricht die Zuordnung des konzeptionellen Modells über ein 1:1-Mapping genau dem Speichermodell. Ihre Aufgabe im Bedarfsfall wird es dann

² Dieser MSDN-Forumsbeitrag, den Sie durch den IntelliLink **F3501** erreichen, liefert wertvolle Tipps zur Vorgehensweise.

immer sein, ein entsprechendes Mapping vorzunehmen, das für das generierte Objektmodell Ihrer Anwendung am besten ist. Für das Beispiel in diesem Kapitel wählen Sie bitte folgende Tabellen aus:

- Address (Adressendetails-Tabelle)
- Product (Produktdetails-Tabelle)
- ProductVendor (Lieferantenprodukte-Tabelle)
- Vendor (Lieferanten-Tabelle)
- VendorAddress (Lieferantenadressen-Tabelle)

10. Klicken Sie auf *Fertig stellen*, wenn Sie Ihre Auswahl getroffen haben

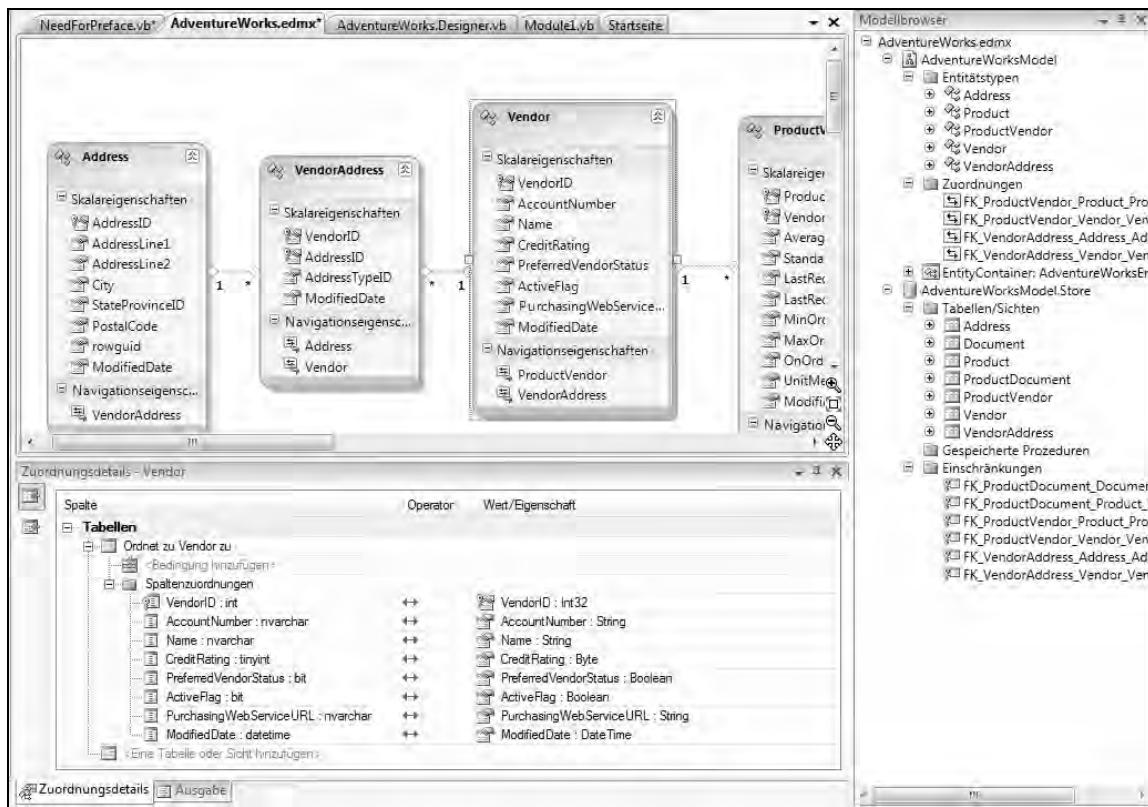


Abbildung 35.6 Anders als beim O/R-Designer von LINQ to SQL können Sie beim Bearbeiten des Entitätsmodells die Zuordnung zwischen konzeptionellem Modell und Speichermodell beliebig umgestalten

In der Designer-Ansicht können Sie übrigens sehen, dass die Darstellung der Eigenschaften der einzelnen Entitäten wesentlich durchsichtiger ist als bei LINQ to SQL. Neben den Skalareigenschaften, die das Mapping auf die eigentlichen Felder (also Spalten) in der Datenbank ermöglichen, sind die Navigationseigenschaften gesondert aufgeführt. Mithilfe der Navigationseigenschaften einer Entität gelingt beispielsweise später, bei der Programmierung mit den korrelierenden Business-Objekten, der Zugriff auf verknüpfte Tabellen, auf Tabellen also, die durch das Speichermodell beschrieben in Relation zueinander stehen.

HINWEIS Der Designer ist dabei übrigens so clever, reine Hilfstabellen, die in der Datenbank dem ausschließlichen Zweck dienen, eine n:m-Verbindung herzustellen, komplett durch die Einstellungen in den Zuordnungdetails des Mappings ganz aufzulösen. Das funktioniert natürlich dann nicht mehr, wenn es nur ein weiteres Feld in der Hilfstabelle gibt, das eine Skalareigenschaft enthält, wie es leider bei der AdventureWorks-Datenbank durch die ModifiedDate-Eigenschaft in jeder solchen Tabelle geschieht. Mithilfe der Northwind-Datenbank, die Sie natürlich trotz Ihres fortgeschrittenen Alters immer noch an eine SQL Server 2008-Instanz knüpfen können, haben Sie aber die Möglichkeit, dieses automatische Wegmappen einer Hilfstabelle direkt in Aktion zu sehen.

TIPP Ein Skript, das die Northwind-Datenbank ohne Fehlermeldungen unter SQL Server 2005/2008 installiert, findet man unter <http://blogs.sqlserverfaq.de/Lists/Beitraege/Post.aspx?ID=21>.

Und nachdem diese Grundvoraussetzungen erledigt sind, steht, wie in der obigen Abbildung zu sehen, das erste Entitätsmodell für unsere zukünftigen Demos, und wir können uns den programmiertechnischen Aspekten zuwenden.

Nachträgliches Ändern des Entitätscontainernamens

In vielen Fällen mögen Sie das EDM so nennen wie auch die Datenbank heißt. In vielen Fällen macht es aber Sinn, den Namen des Entitätscontainers zu ändern, aus denen sich auch der programmtechnische Objektkontext ergibt, den Sie in Ihren Anwendungen benötigen, um auf die Entitätsobjekte des konzeptionellen Modells beispielsweise für die Erstellung von Abfragen mit LINQ to Entities zuzugreifen.

Die Änderung ist dabei vergleichsweise simpel: Sie öffnen den EDM-Designer, klicken auf einen freien Bereich im Designer um das Modell selbst »auszuwählen«, und können dann im Eigenschaftenfenster unter Entitätscontainername einen neuen Namen festlegen.

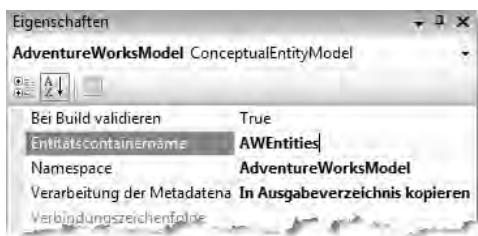


Abbildung 35.7 So ändern Sie den Namen eines Entitätscontainers im EDM-Designer

Gegebenenfalls wird es danach notwendig, auch die *App.Config* anzupassen, falls Sie dort die Verbindungszeichenfolge hinterlegt haben (fett geschrieben im unten stehenden Auszug der Datei).

```

<!--<add name="EventLog" type="System.Diagnostics.EventLogTraceListener"
initializeData="APPLICATION_NAME"/> -->
</sharedListeners>
</system.diagnostics>
<connectionStrings><add name="AWEntities"
connectionString="metadata=res://*/AdventureWorks.csdl|res://*/AdventureWorks.ssdl|res://*/AdventureWork
s.msl;provider=System.Data.SqlClient;provider connection string="Data
Source=V64_LOEFFELDEV\SQL2008EXPRESS;Initial Catalog=AdventureWorks;Integrated
Security=True;MultipleActiveResultSets=True"" providerName="System.Data.EntityClient" /><add
name="AWEntities"
connectionString="metadata=res://*/AdventureWorks.csdl|res://*/AdventureWorks.ssdl|res://*/AdventureWork
s.msl;provider=System.Data.SqlClient;provider connection string="Data
Source=V64_LOEFFELDEV\SQL2008EXPRESS;Initial Catalog=AdventureWorks;Integrated
Security=True;MultipleActiveResultSets=True"" providerName="System.Data.EntityClient"
/></connectionStrings>
</configuration>

```

ACHTUNG Die Designer-Tools des Entity Frameworks scheinen an einigen Stellen noch Probleme mit der Konsistenz zu haben. So ist es bei Experimenten ein paar Mal passiert, dass beispielsweise das Löschen einer Entität aus dem Designer nicht alle Referenzen im Speichermodell bzw. im entsprechenden Mapping nach sich zog – Inkonsistenzen und Compiler-Fehler waren dann der Fall; ob das ein Bug oder ein Feature ist wird die Zukunft zeigen. **Die nicht lokalisierte, englische Version von Visual Studio erwies sich hier übrigens als sehr viel stabiler!**

Seien Sie deswegen schon beim Zusammenstellen der Entitätsobjekte, die aus den Datenbankschemata abgerufen werden, sehr aufmerksam, und überlegen Sie sich gut, welche Tabellen, Sichten und gespeicherten Prozeduren Sie benötigen und welche nicht, wenn Sie nicht in mühsamer Kleinarbeit später die *edmx*-Dateien des Entitätsmodells mit dem XML-Editor nachbearbeiten wollen. Falls dies doch einmal nötig werden sollte: keine Panik, Ruhe bewahren! Klappen Sie den Tisch vor sich hoch, warten Sie bis die Sauerstoffmasken herunterfallen, ... – nein, Scherz beiseite. In diesem Fall schließen Sie einfach den Entitätsdesigner und öffnen die ihm zugrunde liegende Entitätsmodell-XML-Datei einfach mit dem XML-Editor. Dazu wählen Sie aus dem Kontextmenü der *edmx*-Datei *Öffnen mit...*...

Wählen Sie im jetzt erscheinenden Dialog *XML-Editor* aus der Liste aus (siehe nächste Abbildung), und klicken Sie auf *OK*, um das Modell als XML-Datei zu editieren.

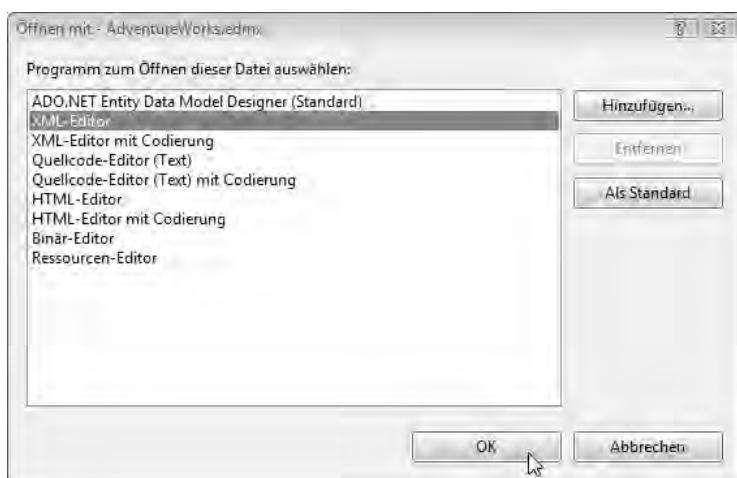


Abbildung 35.8 Falls der Entitätsdesigner einmal streiken sollte oder Dummheiten macht, schließen Sie das EDM in der Designansicht und öffnen Sie die *edmx*-Datei, die Konzeptionelles Modell, Speichermodell und Mapping enthält, mit dem XML-Editor.

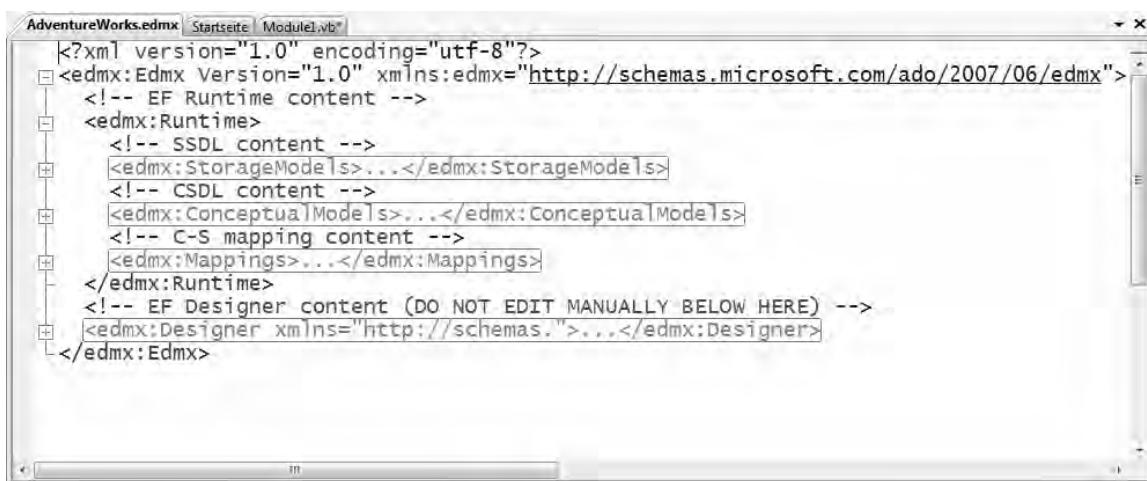


Abbildung 35.9 Konzeptionelles Modell, Speichermodell und Mapping können Sie im Bedarfsfall auch als XML einsehen und bearbeiten. Bitte nehmen Sie keine Änderungen unterhalb des Abschnittes *EF Designer content* vor!

In der obenstehenden Abbildung sehen Sie die Abschnitte, in denen die Inhalte der SSDL-Datei (Speichermodell), CSDL-Datei (konzeptionelles Modell) und MSL (Mapping) dargestellt sind, die Sie bedenkenlos editieren können. Lassen Sie aber unbedingt die Finger vom Abschnitt, der sich unterhalb des Kommentars *EF Designer content. DO NOT EDIT MANUALLY BELOW HERE (AB HIER NICHT MEHR HÄNDISCH EDITIEREN!)* sollten Sie sehr, sehr wörtlich nehmen!

Abfrage von Daten eines Entitätsmodells

Die Codedateien, die aus dem Konzeptionellen Modell des Entitätsmodell entstehen, werden in einer Codedatei abgelegt, die Sie unterhalb des Datenmodells im Projektmappen-Explorer eingeordnet finden, wenn Sie die Projektmappen-Explorer alle Dateien anzeigen lassen.

Auch bei der Programmierung mit dem Entity-Framework gibt es für die Business-Objekte, die aus dem Konzeptionellen Modell entstehen, eine zentrale Verwalterklasse, die sich anders als bei LINQ to SQL nicht Datenkontext, sondern konsequenterweise Objektkontext nennt.

```
'''<summary>
'''Es gibt keine Kommentare für AdventureWorksEntities im Schema.
'''</summary>
Partial Public Class AdventureWorksEntities
    Inherits Global.System.Data.Objects.ObjectContext
```

Der Objektkontext stellt zum Einen die Businessobjekt-Abfrageauflistungen in Form von `ObjectQuery(of Type)-Klassen` dar. Eine solche generische Klasse, wie im unten stehenden Listing zu sehen, repräsentiert also eine Abfrage, die eine Auflistung von Objekten eines bestimmten Typs zurückliefer, wenn diese Abfrage entweder mithilfe einer so genannten Entity-SQL-Anweisung oder einer LINQ-Abfrage erstellt und ausgeführt wird. Dabei führt sie die Abfragen immer dann aus, wenn ...

- ... ihr Iterator verwendet wird, wenn Sie also beispielsweise mit For/Each durch die Elemente ihrer Ergebnisaufstellung hindurch iterieren,
- ... sie an eine generische Liste (List(Of Type)) zugewiesen wird oder
- ... wenn ihre Execute-Methode ausdrücklich aufgerufen wurde.

```
'''<summary>
'''Es gibt keine Kommentare für Product im Schema.
'''</summary>
Public Readonly Property Product() As Global.System.Data.Objects.ObjectQuery(Of Product)
    Get
        If (Me._Product Is Nothing) Then
            Me._Product = MyBase.CreateQuery(Of Product)("[Product]")
        End If
        Return Me._Product
    End Get
End Property
```

Die Business-Objekte werden, wie ebenfalls im Listing in der zweiten in Fettschrift gesetzten Zeile zu sehen, so definiert, dass bei der ersten Verwendung automatisch eine Instanz mit einer Abfrage definiert wird, die die gesamten Elemente der zugeordneten Entität (Tabelle, View) abruft.

Zum Anderen steuert der Objektkontext natürlich auch den Aufbau zur Datenbank, managet die Änderungsverfolgung und stellt eine Infrastruktur zum Concurrency-Check zur Verfügung.

Abfrage von Daten mit LINQ to Entities-Abfragen

Eine einfache Abfrage zeigt das folgende Listing, das sich in *Module1.vb* des folgenden verwendeten Beispielprojektes befindet.

BEGLEITDATEIEN

Die Begleitdateien zu den folgenden Beispielen finden Sie im Verzeichnis:

```
...\VB 2008 Entwicklerbuch\F - LINQ\Kapitel 35\LinqToEntities
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Module Module1

Sub Main()

    'Wir verwenden in den Beispielen die Verbindungszeichenfolge aus der App.Config.
    'Ändern Sie die Zeichenfolge im Abschnitt ConnectionStrings, um die Beispiele
    'auf Ihre SQL Server-Instanz anzupassen. Sie müssen dazu nur den Provider-
    'ConnectionString innerhalb des Gesamt-ConnectionStrings anpassen.

    Dim awContext As New AWEntities()

    Dim lieferanten = From lieferant In awContext.Vendor
                      Where lieferant.Name.StartsWith("H") _
                      Order By lieferant.Name
```

```

For Each lieferant In lieferanten
    Console.WriteLine(lieferant.AccountNumber & ":" & lieferant.Name)
    Console.WriteLine(New String("c", 70))

    'Zu jedem Lieferanten die Produkte ausliefern
    For Each produkt In lieferant.ProductVendor
        Console.Write("ID:" & produkt.ProductID & " ")
        Console.WriteLine("Name:" & produkt.Product.Name)
    Next
    Console.WriteLine()

Next
End Sub

End Module

```

Falls Sie das letzte Kapitel über LINQ to SQL ebenfalls durchgearbeitet haben, wird Ihnen das Beispiel bekannt vorkommen – die Ähnlichkeiten sind frappierend! In folgenden Punkten gibt es allerdings Unterschiede:

- Es gibt – wie eingangs schon gesagt – keinen Datenkontext wie bei LINQ to SQL sondern ein verwaltetes Objekt, das von `ObjectContext` abgeleitet wurde. In unserem Fall ist das die `AEEntities`-Klasse.
- Die Pluralisierung der Abfrageeigenschaften findet hier nicht automatisch statt. Die Eigenschaften nennen sich also standardmäßig genau so, wie die Entitäten, auf denen sie basieren. Sie können das aber natürlich im EDM-Designer im konzeptionellen Modell entsprechend ändern. Der folgende Abschnitt zeigt, wie es geht.
- Wenn Sie das Programm laufen lassen, werden Sie feststellen, dass es lediglich die folgende Ausgabe auf den Bildschirm zaubert:

```

HILLBIC0001: Hill Bicycle Center
=====
HILLSBI0001: Hill's Bicycle Service
=====
HOLIDAY0001: Holiday Skate & Cycle
=====
HYBRIDB0001: Hybrid Bicycle Center
=====
```

Die innere Schleife wird also gar nicht ausgeführt, bzw. liefert keine Ergebnisse. Das liegt an der Art und Weise, wie Lazy-Loading im Entity-Framework implementiert ist. *Lazy-Loading* (»faules« Laden) ist dabei nämlich eigentlich sogar ein »Lazy-Lazy-Loading«, ein quasi »ganz faules« Laden: Inhalte verknüpfter Entitäten werden nicht nur *nicht* direkt mitgeladen, sie werden *auch nicht* geladen, wenn Sie darauf zugreifen. Der Abschnitt »Generierte SQL-Anweisungen unter die Lupe nehmen« ab Seite 994 hält mehr zu diesem Thema bereit.

- Zu guter Letzt erlaubt es das Entity-Framework auch nicht, auf einfache Art und Weise Debug-Ausgaben auf der Konsole auszugeben. Wenn Sie beispielsweise die Developer-Edition von SQL Server zur Hand haben, können Sie sich hier mit dem SQL Profiler aus der Patsche helfen. Wie Sie den Profiler

verwenden, um sich generiertes SQL anzuschauen (Hilfe zur Selbsthilfe kann beim Entity-Framework nicht schaden!), finden Sie im Abschnitt »Generierte SQL-Anweisungen unter die Lupe nehmen« ab Seite 994 beschrieben.

HINWEIS Wie bei LINQ üblich, werden Abfragen auch bei LINQ to Entities verzögert ausgeführt, und sie lassen sich auch kaskadieren. Die LINQ to Objects- und LINQ to SQL-Kapitel bieten hier ausreichend Beispiele. Auch bei LINQ to Entities gilt: Sie können eine Abfrageergebnisliste beispielsweise mit `ToList` oder `ToArray` von ihrem Abfrageobjekt lösen. Im Gegensatz zu LINQ to SQL ist die Liste damit aber nicht automatisch von ihrem Datenkontext gelöst, sondern nur von der Abfragekette, die dann unterbrochen ist.

Wie Abfragen zum Datenprovider gelangen – Entity-SQL (eSQL)

Anders als LINQ to SQL werden im Entity-Framework aus LINQ-Abfragen nicht direkt zum Datenprovider geschickt, der sich übrigens im vollen Namen *ADO.NET Entity Client Data Provider* nennt. Vielmehr haben die Redmonder Entwicklerkollegen einen Dialekt namens Entity-SQL entwickelt, der das konzeptionelle Modell eines EDMs abfragt. Dadurch, dass eSQL nicht direkt auf dem Datenprovider arbeitet, wird eSQL Datenbankplattform-unabhängig – wenn Sie später also Ihre Programme an andere Datenprovider anpassen müssen, ist der Code, der ja gegen das konzeptionelle Modell des EDM abfragt, davon nicht betroffen. Im günstigsten Fall. Im ungünstigsten Fall kann es sein, dass Sie bei Abfragen gegen Ihr Entitätsmodell Features verwendet haben, die der eine oder andere Datenbanktreiber – wenn er denn erstmal verfügbar ist – vielleicht nicht umsetzen kann. Das jedoch ist bei den Providern für die großen Anbieter von Datenbanksystemen (Oracle, IBM, etc.) wohl eher nicht zu erwarten.

Anpassen des Namens der Entitätenmenge

Der Name einer Entitätsmenge ist mit dem Entitätsdesigner übrigens schnell geändert. Wenn es Ihnen nicht passt, dass aus einer Tabelle *Vendor* die Abfrageeigenschaft *Vendor* des Objektkontexts hervorgegangen ist, und Sie den Namen lieber pluralisiert hätten, ändern Sie den Namen einfach:

1. Doppelklicken auf das EDM im Projektmappen-Explorer, um den Designer zu öffnen.
2. Klicken Sie die Entität (zum Beispiel *Vendor*) an.
3. Ändern Sie die Eigenschaft *Name der Entitätenmenge* im Eigenschaftenfenster dementsprechend.

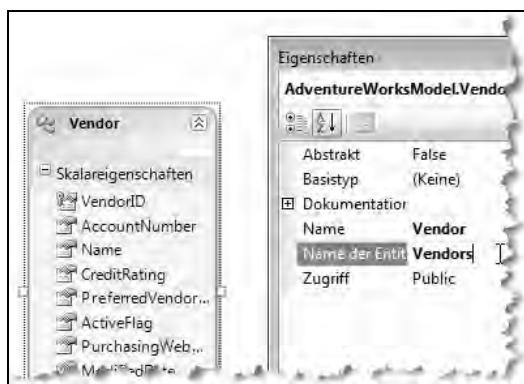


Abbildung 35.10 Der Name einer Entitätenmenge lässt sich mit dem EDM-Designer in Nullkommanichts ändern

HINWEIS Eine automatische Pluralisierung von Auflistungsnamen wie bei LINQ to SQL gilt nicht für das Entity-Framework, jedenfalls nicht in der gegenwärtigen, deutsch-lokalisierten Version. Sie können dies also nur auf die hier beschriebene Art und Weise erreichen.

Generierte SQL-Anweisungen unter die Lupe nehmen

In LINQ to SQL können Sie es mit so gut wie gar keinem Aufwand erreichen, dass die generierten T-SQL-Abfragen, die sich aus einer LINQ to SQL-Abfrage ergeben, automatisch in einem Protokoll erstellt werden. Mit LINQ to Entities geht das nicht so einfach. Aber es geht. Sie haben zwei Möglichkeiten:

- Sie haben die Möglichkeit, Ihre Abfrage in das ursprüngliche Abfrage-Objekt des Objektkontexts zu casten, und dann die Methode `ToTraceString` aufzurufen, um herauszufinden, welche Befehle das Entity Framework *plant*, zum Datenprovider zu schicken. Sie könnten also das obige Beispielprogramm um diese Zeilen ergänzen...

```
.
.
.

Dim awContext As New AWEntities()

Dim lieferanten = From lieferant In awContext.Vendors
    Where lieferant.Name.StartsWith("H") _
    Order By lieferant.Name

'Mit dieser Anweisung können wir rausfinden,
'welcher SQL-Text zur Anwendung kommen wird.
Console.WriteLine(CType(lieferanten, ObjectQuery(Of Vendor)).ToTraceString)
Console.WriteLine()

.
.
.
```

... um die Ausgabe um folgende Zeilen zu ergänzen:

```
SELECT
[Extent1].[VendorID] AS [VendorID],
[Extent1].[AccountNumber] AS [AccountNumber],
[Extent1].[Name] AS [Name],
[Extent1].[CreditRating] AS [CreditRating],
[Extent1].[PreferredVendorStatus] AS [PreferredVendorStatus],
[Extent1].[ActiveFlag] AS [ActiveFlag],
[Extent1].[PurchasingWebServiceURL] AS [PurchasingWebServiceURL],
[Extent1].[ModifiedDate] AS [ModifiedDate]
FROM [Purchasing].[Vendor] AS [Extent1]
WHERE (CAST(CHARINDEX(N'H', [Extent1].[Name]) AS int)) = 1
ORDER BY [Extent1].[Name] ASC
```

```
HILLBIC0001: Hill Bicycle Center
=====
```

```
HILLSBI0001: Hill's Bicycle Service
=====
HOLIDAY0001: Holiday Skate & Cycle
=====
HYBRIDB0001: Hybrid Bicycle Center
=====

Taste drücken, zum Beenden
```

- Die zuverlässigere aber aufwändigere Methode, da sie den tatsächlichen IST-Zustand widerspiegelt: Sie verwenden das Profiler-Werkzeug einer großen SQL Server-Version. Wenn Sie beispielsweise die Möglichkeit haben, in Ihrer Firma auf eine Vollversion von SQL Server per Remote-Desktop zuzugreifen, haben Sie auch die Möglichkeit, den Profiler mit der Instanz Ihrer lokalen SQL Express-Version zu verbinden. Das Profiler-Werkzeug gibt Ihnen genau Auskunft über die gesendeten Daten zwischen der Datenbankinstanz und in der auf dem Client geöffneten Session. Ein Beispiel sehen Sie in folgender Abbildung.

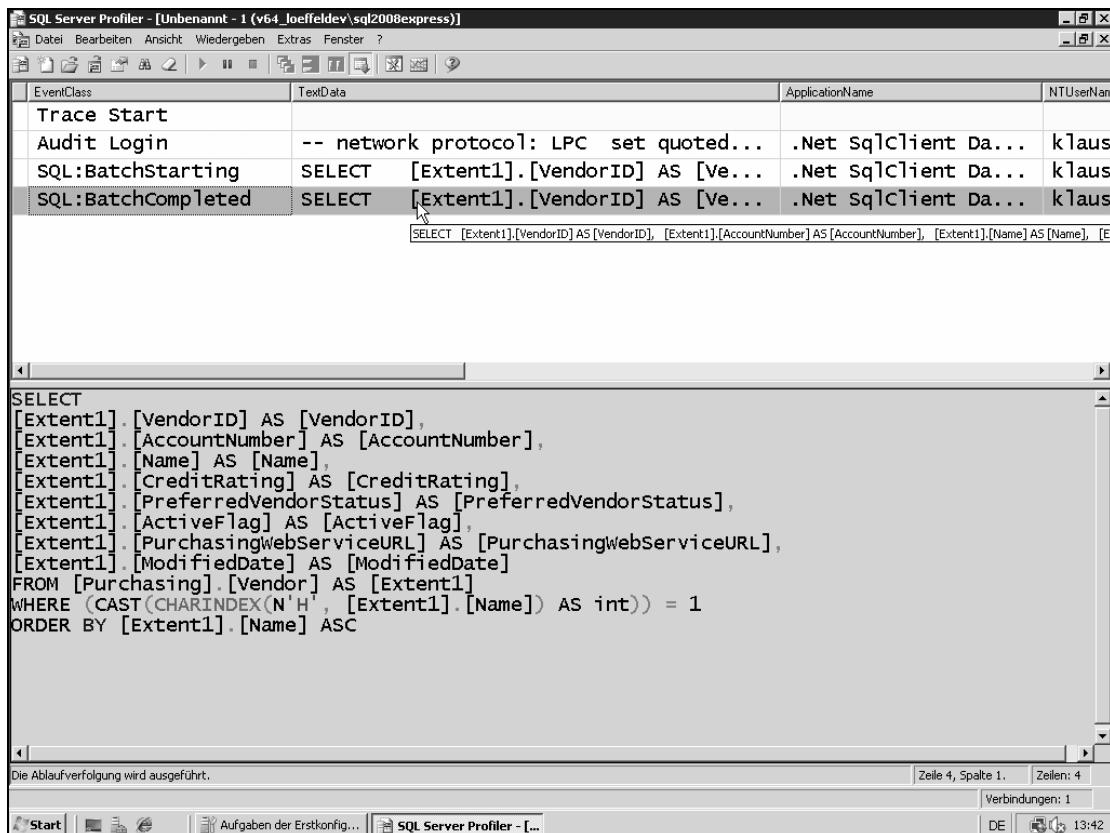


Abbildung 35.11 Das Profiler-Tool eines großen SQL-Servers können Sie auch auf Ihre Express-Instanz ansetzen. Dabei muss sich der Profiler nicht mal auf der gleichen Maschine befinden – wenn es Ihre Netzwerk-Einstellungen zulassen.

TIPP Damit Sie den Profiler des großen Visual Studios mit Ihrer Express-Instanz verbinden können, muss Ihre Firewall das erlauben und Sie müssen natürlich auch die Netzwerkprotokolle entsprechend eingerichtet haben. Das vorherige Kapitel erklärt im Rahmen der SQL Server 2008 Express-Installation, wie das funktioniert.

Lazy- und Eager-Loading im Entity Framework

Das Ladeverhalten unterscheidet sich von LINQ to SQL ganz erheblich – das haben wir bei dem bislang verwendeten Beispiel schon gesehen. Während bei LINQ to SQL transparentes Lazy-Loading standardmäßig verwendet wird – die Daten, die Sie benötigen, werden also erst im Bedarfsfall vom Provider abgerufen – passiert bei LINQ to Entities gar nichts, wenn Sie auf die untergeordneten Daten einer Tabellenverknüpfung zugreifen. Aber natürlich kommen Sie an die Daten ran – Sie müssen nur ein wenig mehr Aufwand betreiben. Grundsätzlich haben Sie zwei Möglichkeiten:

- Sie laden die Daten, die Sie benötigen, manuell nach.
- Sie verwenden Eager-Loading für bestimmte Relationen.

Daten beim Lazy-Loading manuell nachladen

Im Beispielprojekt der Anwendung finden Sie ein weiteres Modul mit dem Namen *LazyEagerLoading.vb*. Es demonstriert, wie Sie vorgehen müssen, um Daten aus Tabellen-Relationen manuell nachladen zu können.

HINWEIS Damit das Beispielprojekt jetzt auch mit dieser Main-Methode startet, stellen Sie bitte in den Projekteigenschaften das entsprechende Startobjekt ein. Die Projekteigenschaften erreichen Sie, indem Sie im Projektmappen-Explorer auf dem Projektnamen das Kontextmenü öffnen und Eigenschaften auswählen (Sie erhalten folgende Maske).

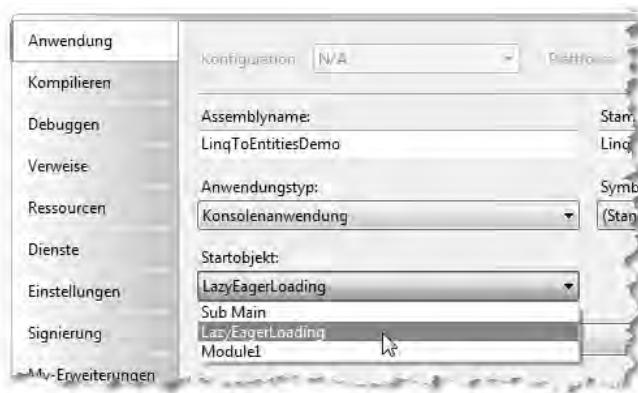


Abbildung 35.12 So bestimmen Sie, in welchem Modul die Sub Main zum Starten der Anwendung aufgerufen werden soll

```
Imports System.Data.Objects
Module LazyEagerLoading
    Sub main()
        Dim awContext As New AWEntities()
```

```
Dim lieferanten = From lieferant In awContext.Vendors
    Where lieferant.Name.StartsWith("H") _
    Order By lieferant.Name

'Mit dieser Anweisungen können wir rausfinden,
'welcher SQL-Text zur Anwendung kommen wird.
Console.WriteLine(CType(lieferanten, ObjectQuery(Of Vendor)).ToTraceString)
Console.WriteLine()

For Each lieferant In lieferanten
    Console.WriteLine(lieferant.AccountNumber & ": " & lieferant.Name)
    Console.WriteLine(New String("="c, 70))

    If Not lieferant.ProductVendor.IsLoaded Then
        lieferant.ProductVendor.Load()

        'Zu jedem Lieferanten die Produkte ausliefern
        For Each produkt In lieferant.ProductVendor
            If Not produkt.ProductReference.IsLoaded Then
                produkt.ProductReference.Load()
                Console.Write("ID: " & produkt.ProductID & " ")
                Console.WriteLine("Name: " & produkt.Product.Name)
            End If
            Next
            Console.WriteLine()
        End If

        Next
    End Sub

End Module
```

Von Relevanz sind hier drei Dinge:

- Sie stellen mit `IsLoaded` fest, ob eine bestimmte Entität, die eine Verknüpfung von oder zu einer anderen darstellt, schon geladen wurde.
- Sie laden die Inhalte der Verknüpfung mit der `Load`-Methode nach.
- Zu einem `:1`-Endpunkt besteht immer eine *Referenz*. Das bedeutet, dass Sie nicht den eigentlichen Entitätsnamen-Eigenschaft zum Nachladen der Daten verwenden, sondern den Namen mit der Endung `Reference`. Da die Tabelle `ProductVendors` (alle Lieferantenprodukte) mit der Tabelle `Products` (Produktdetails) `1:1` verknüpft ist, verwenden Sie also `ProductReference` zum Laden der Inhalte. Da hingegen `Vendor` und `ProductVendor` `1:n` verknüpft sind, (erster fettgesetzter Codeblock im Listing), verwenden Sie hier *nicht* das Postfix `Reference`.

HINWEIS Es gibt übrigens einen Generator der Code generiert, um Unterstützungscode für transparentes Lazy-Loading zu generieren. Es gibt allerdings zwei Haken bei der Sache. Er ist experimentell und er generiert zurzeit nur C#-Code. Falls Sie sich ihn dennoch anschauen wollen: Der IntelliLink **F3502** verrät mehr.

Eager-Loading für bestimmte Relationen verwenden

Die andere Möglichkeit, die in bestimmten Szenarien sinniger ist – insbesondere dann, wenn vergleichsweise wenig viele Daten von vorne herein zur Verfügung stehen sollen – ist, das Framework anzuweisen, die Daten für die untergeordneten Tabellen in einem Rutsch durch eine JOIN-Verknüpfung sofort mitzuladen.

Sie erreichen das durch die Verwendung der `Include`-Methode, der Sie den Pfad der untergeordneten Tabellen einfach durch »« getrennt mitgeben. Wenn Sie also die Tabelle *Vendor* abfragen, und möchten, dass *ProductVendor*-Daten und die denen zugeordneten *Product*-Daten automatisch mitgeliefert werden, dann lautet der Suchpfad eben »*ProductVendor.Product*«.

WICHTIG Die Ausgangstabelle (im Beispiel *Vendor*) geben Sie bitte nicht mit im Suchpfad an!

Das auf diese Weise abgeänderte Beispiel sieht dann folgendermaßen aus:

```
Sub EagerLoading()
    Dim awContext As New AWEntities()

    Dim lieferanten = From lieferant In awContext.Vendors.Include("ProductVendor.Product") _
                      Where lieferant.Name.StartsWith("H") _
                      Order By lieferant.Name

    'Mit dieser Anweisungen können wir rausfinden,
    'welcher SQL-Text zur Anwendung kommen wird.
    Console.WriteLine(CType(lieferanten, ObjectQuery(Of Vendor)).ToTraceString)
    Console.WriteLine()

    For Each lieferant In lieferanten
        Console.WriteLine(lieferant.AccountNumber & ": " & lieferant.Name)
        Console.WriteLine(New String("="c, 70))

        'Zu jedem Lieferanten die Produkte ausliefern
        For Each produkt In lieferant.ProductVendor
            Console.Write("ID:" & produkt.ProductID & " ")
            Console.WriteLine("Name:" & produkt.Product.Name)
        Next
        Console.WriteLine()

        Next

        Console.WriteLine()
        Console.WriteLine("Taste drücken, zum Beenden")
        Console.ReadKey()
    End Sub
```

Damit ändert sich dementsprechend auch der generierte SQL-Text, der dann wie folgt ausschaut (gekürzt):

```
SELECT
[Project2].[VendorID] AS [VendorID],
[Project2].[AccountNumber] AS [AccountNumber],
[Project2].[Name] AS [Name], [Project2].[CreditRating] AS [CreditRating],
[Project2].[PreferredVendorStatus] AS [PreferredVendorStatus],
```

```

[Project2].[ActiveFlag] AS [ActiveFlag], [Project2].[PurchasingWebServiceURL] AS
[PurchasingWebServiceURL], [Project2].[ModifiedDate] AS [ModifiedDate],
[Project2].[C1] AS [C1], [Project2].[C2] AS [C2],
[Project2].[ProductID] AS [ProductID], [Project2].[VendorID1] AS [VendorID1],
[Project2].[AverageLeadTime] AS [AverageLeadTime], [Project2].[StandardPrice] AS [StandardPrice],
[Project2].[LastReceiptCost] AS [LastReceiptCost], [Project2].[LastReceiptDate] AS [LastReceiptDate],
[Project2].[MinOrderQty] AS [MinOrderQty], [Project2].[MaxOrderQty] AS [MaxOrderQty],
[Project2].[OnOrderQty] AS [OnOrderQty], [Project2].[UnitMeasureCode] AS [UnitMeasureCode],
[Project2].[ModifiedDate1] AS [ModifiedDate1], [Project2].[ProductID1] AS [ProductID1],
[Project2].[Name1] AS [Name1], [Project2].[ProductNumber] AS [ProductNumber],
[Project2].[MakeFlag] AS [MakeFlag], [Project2].[FinishedGoodsFlag] AS [FinishedGoodsFlag],
[Project2].[Color] AS [Color], [Project2].[SafetyStockLevel] AS [SafetyStockLevel],
[Project2].[ReorderPoint] AS [ReorderPoint], [Project2].[StandardCost] AS [StandardCost],
[Project2].[ListPrice] AS [ListPrice], [Project2].[Size] AS [Size],
[Project2].[SizeUnitMeasureCode] AS [SizeUnitMeasureCode], [Project2].[WeightUnitMeasureCode] AS
[WeightUnitMeasureCode], [Project2].[Weight] AS [Weight],
[Project2].[DaysToManufacture] AS [DaysToManufacture], [Project2].[ProductLine] AS [ProductLine],
[Project2].[Class] AS [Class], [Project2].[Style] AS [Style],
[Project2].[ProductSubcategoryID] AS [ProductSubcategoryID],
[Project2].[ProductModelID] AS [ProductModelID], [Project2].[SellStartDate] AS [SellStartDate],
[Project2].[SellEndDate] AS [SellEndDate], [Project2].[DiscontinuedDate] AS [DiscontinuedDate],
[Project2].[rowguid] AS [rowguid], [Project2].[ModifiedDate2] AS [ModifiedDate2]
FROM ( SELECT
    [Extent1].[VendorID] AS [VendorID], [Extent1].[AccountNumber] AS [AccountNumber],
    [Extent1].[Name] AS [Name], [Extent1].[CreditRating] AS [CreditRating],
    [Extent1].[PreferredVendorStatus] AS [PreferredVendorStatus],
    [Extent1].[ActiveFlag] AS [ActiveFlag],
    [Extent1].[PurchasingWebServiceURL] AS [PurchasingWebServiceURL],
    [Extent1].[ModifiedDate] AS [ModifiedDate], 1 AS [C1],
    .
    . (* Gekürzt – Schema ist zu umfangreich! *)
    .
    [Project1].[SellStartDate] AS [SellStartDate],
    [Project1].[SellEndDate] AS [SellEndDate],
    [Project1].[DiscontinuedDate] AS [DiscontinuedDate],
    [Project1].[rowguid] AS [rowguid],
    [Project1].[ModifiedDate1] AS [ModifiedDate2],
    [Project1].[C1] AS [C2]
    FROM [Purchasing].[Vendor] AS [Extent1]
    LEFT OUTER JOIN (SELECT
        [Extent2].[ProductID] AS [ProductID],
        [Extent2].[VendorID] AS [VendorID],
        [Extent2].[AverageLeadTime] AS [AverageLeadTime],
        [Extent2].[StandardPrice] AS [StandardPrice],
        [Extent2].[LastReceiptCost] AS [LastReceiptCost],
        .
        . (* Gekürzt – Schema ist zu umfangreich! *)
        .
        [Extent3].[DiscontinuedDate] AS [DiscontinuedDate],
        [Extent3].[rowguid] AS [rowguid],
        [Extent3].[ModifiedDate] AS [ModifiedDate1],
        1 AS [C1]
        FROM [Purchasing].[ProductVendor] AS [Extent2]
        LEFT OUTER JOIN [Production].[Product] AS [Extent3] ON [Extent2].[ProductID] =
        [Extent3].[ProductID] ) AS [Project1] ON [Extent1].[VendorID] = [
        Project1].[VendorID]

```

```

        WHERE (CAST(CHARINDEX(N'H', [Extent1].[Name]) AS int)) = 1
    ) AS [Project2]
ORDER BY [Project2].[Name] ASC, [Project2].[VendorID] ASC, [Project2].[C2] ASC

```

HILLBIC0001: Hill Bicycle Center

ID:524 Name:HL Spindle/Axle

HILLSBI0001: Hill's Bicycle Service

ID:911 Name:LL Road Seat/Saddle

ID:912 Name:ML Road Seat/Saddle

HOLIDAY0001: Holiday Skate & Cycle

HYBRIDB0001: Hybrid Bicycle Center

ID:910 Name:HL Mountain Seat/Saddle

Taste drücken, zum Beenden

Anonymisierungsvermeidung bei Abfragen in verknüpften Tabellen

Sie können sowohl bei LINQ to SQL als auch bei LINQ to Entities Join-Abfragen verwenden, um Daten aus mehreren Tabellen in einer flachen Ergebnisliste zusammenzuführen. In vielen Fällen möchten Sie aber, dass Sie nur deswegen auf verknüpfte Tabellen zugreifen, um bestimmte Datensätze in der Haupttabelle in Abhängigkeit bestimmter Werte der Tabellen zu selektieren, mit denen die Haupttabelle verknüpft ist. Und in diesen Fällen können Sie Join nicht gebrauchen, weil Join Ihnen automatisch eine Ergebnisliste auf Basis einer anonymen Klasse zurückliefert. Sie möchten in diesen Fällen aber in der Regel eine Ergebnisliste auf Basis einer konkreten Entitätsklasse zurückgeliefert bekommen.

Ein Beispiel: Angenommen Sie möchten alle Lieferanten ermitteln, die über mindestens ein Produkt verfügen, dessen Name mit dem Buchstaben »W« beginnt. Diese Abfrage mit einer Join-Abfrage zu formulieren wäre, nachdem Sie sich das LINQ to Objects-Kapitel zu Gemüte geführt haben, sicherlich kein Problem.

Doch das Problem dabei ist: Sie bekommen eine Ergebnisliste auf Basis einer anonymen Klasse zurück. Sie möchten aber eine Ergebnisliste auf Basis von Vendor (*Lieferant*) zurückgeliefert bekommen.

In diesem Fall verwenden Sie eine Abfragemethode, die Sie im Modul *SpezielleAbfrageMethoden.vb* im Beispielprojekt finden, und die mehrere FROM-Abfragen ineinander verschachtelt und schließlich mit Distinct dafür sorgt, dass Dubletten in der Ergebnisliste ausgeschlossen werden.

HINWEIS	Das folgende Beispiel verwendet übrigens Lazy-Loading mit manuellem Nachladen der Datensätze, wie wir es im vorherigen Abschnitt kennengelernt haben, um die untergeordneten Datensätze zu ermitteln.
---------	---

```
Sub AnonymisierungsvermeidungBeiVerknüpftenAbfragen()

    Dim awContext As New AWEntities()

    'Fragt alle Lieferanten ab, die Produkte haben,
    'deren Namen ihrer Produktdetailsbeschreibungen mit "W" beginnen.
    Dim lieferanten = From lieferantItems In awContext.Vendors _
        From lieferantProduktItem In lieferantItems.ProductVendor _
            From productItem In awContext.Product
                Where lieferantItems.VendorID = lieferantProduktItem.VendorID
                Where productItem.ProductID = lieferantProduktItem.ProductID
                Where productItem.Name.StartsWith("W") _
                    Select lieferantItems Distinct

    'Manuelles Lazy-Loading für die Gesamtausgabe verwenden:
    For Each lieferant In lieferanten
        Console.WriteLine(lieferant.AccountNumber & ":" & lieferant.Name)
        Console.WriteLine(New String("="c, 70))

        If Not lieferant.ProductVendor.IsLoaded Then
            lieferant.ProductVendor.Load()

            'Zu jedem Lieferanten die Produkte ausliefern
            For Each produkt In lieferant.ProductVendor
                If Not produkt.ProductReference.IsLoaded Then
                    produkt.ProductReference.Load()
                    Console.Write("ID:" & produkt.ProductID & " ")
                    Console.WriteLine("Name:" & produkt.Product.Name)
                End If
            Next
            Console.WriteLine()
        End If
    Next

    Console.WriteLine()
    Console.WriteLine("Taste drücken, zum Beenden")
    Console.ReadKey()

End Sub
```

Wenn wir diese Methode laufen lassen, liefert sie die gewünschte Ergebnisliste, wie im folgenden Bildschirmauszug zu sehen:

```
GREENLA0001: Green Lake Bike Company
=====
ID:870 Name:Water Bottle - 30 oz.
ID:871 Name:Mountain Bottle Cage
ID:872 Name:Road Bottle Cage
ID:873 Name:Patch Kit/8 Patches
ID:876 Name:Hitch Rack - 4-Bike
ID:877 Name:Bike Wash - Dissolver
ID:878 Name:Fender Set - Mountain
ID:879 Name:All-Purpose Bike Stand
ID:880 Name:Hydration Pack - 70 oz.
```

```
TEAMATH0001: Team Athletic Co.
=====
ID:858 Name:Half-Finger Gloves, S
ID:859 Name:Half-Finger Gloves, M
ID:860 Name:Half-Finger Gloves, L
ID:861 Name:Full-Finger Gloves, S
ID:862 Name:Full-Finger Gloves, M
ID:863 Name:Full-Finger Gloves, L
ID:864 Name:Classic Vest, S
ID:865 Name:Classic Vest, M
ID:866 Name:Classic Vest, L
ID:867 Name:Women's Mountain Shorts, S
ID:868 Name:Women's Mountain Shorts, M
ID:869 Name:Women's Mountain Shorts, L
```

```
FITNESS0001: Fitness Association
=====
```

```
ID:849 Name:Men's Sports Shorts, M
ID:850 Name:Men's Sports Shorts, L
ID:851 Name:Men's Sports Shorts, XL
ID:852 Name:Women's Tights, S
ID:853 Name:Women's Tights, M
ID:854 Name:Women's Tights, L
ID:855 Name:Men's Bib-Shorts, S
ID:856 Name:Men's Bib-Shorts, M
ID:857 Name:Men's Bib-Shorts, L
```

Taste drücken, zum Beenden

Interessant dabei ist auch ein Blick auf das Ergebnis des SQL-Profilers, der uns Einblick in die generierte SQL-Anweisung erlaubt, die die Ergebnisliste ermittelt:

```
[Distinct1].[PurchasingWebServiceURL] AS [PurchasingWebServiceURL],
[Distinct1].[ModifiedDate] AS [ModifiedDate]
FROM ( SELECT DISTINCT
    [Extent1].[VendorID] AS [VendorID],
    [Extent1].[AccountNumber] AS [AccountNumber],
    [Extent1].[Name] AS [Name],
    [Extent1].[CreditRating] AS [CreditRating],
    [Extent1].[PreferredVendorStatus] AS [PreferredVendorStatus],
    [Extent1].[ActiveFlag] AS [ActiveFlag],
    [Extent1].[PurchasingWebServiceURL] AS [PurchasingWebServiceURL],
    [Extent1].[ModifiedDate] AS [ModifiedDate]
    FROM [Purchasing].[Vendor] AS [Extent1]
    INNER JOIN [Purchasing].[ProductVendor] AS [Extent2] ON
    ([Extent1].[VendorID] = [Extent2].[VendorID]) AND ([Extent1].[VendorID] = [Extent2].[VendorID])
    INNER JOIN [Production].[Product] AS [Extent3] ON [Extent3].[ProductID] = [Extent2].[ProductID]
    WHERE (CAST(CHARINDEX(N'W', [Extent3].[Name]) AS int)) = 1
) AS [Distinct1]
```

Kompilierte Abfragen

LINQ to Entities ermöglicht es aus Gründen der Performance-Optimierung, so genannte optimierte Abfragen einzusetzen. Eine kompilierte Abfrage besteht quasi aus einem Delegaten zu einer vorkompilierten Instanz eines ObjectQuery(of Type)-Objektes, die dann eine entsprechende Ergebnisliste zurückliefert – natürlich wieder vom Typ IQueryable(of Type). Mithilfe des Einsatzes entsprechender Parameter in dem dazu notwendigen Lambda-Ausdruck, der beim Einrichten einer kompilierten Abfrage dem Konstruktor der CompiledQuery-Klasse übergeben werden muss, können Sie dabei kompilierte Abfragen für verschiedene Abfrageparameter wieder verwendbar gestalten.

Das folgende Beispiel demonstriert den Einsatz einer kompilierten Abfrage, die Lieferanten mit einem bestimmten Anfangsbuchstaben ermittelt. Diese Abfrage wird parametrisiert kompiliert, und sie kann durch Übergabe des Anfangsbuchstabens beim Aufruf der Abfrage immer wieder eingesetzt werden:

```
Sub KompilierteAbfragen()

    Dim awContext As New AWEntities()

    Dim kompilierteLieferantenAbfrage = CompiledQuery.Compile(Of AWEntities,
        String, IQueryable(Of Vendor))( _
        Function(context, Anfangsbuchstabe) From lieferant In context.Vendors _
        Where lieferant.Name.StartsWith(Anfangsbuchstabe) _
        Order By lieferant.Name _
        Select lieferant)

    Dim abfrageNach_H = kompilierteLieferantenAbfrage(awContext, "H")

    For Each lieferant In abfrageNach_H
        Console.WriteLine(lieferant.AccountNumber & ": " & lieferant.Name)
        Console.WriteLine(New String("="c, 70))

        'Zu jedem Lieferanten die Produkte ausliefern
        For Each produkt In lieferant.ProductVendor
            Console.Write("ID:" & produkt.ProductID & " ")
            Console.WriteLine("Name:" & produkt.Product.Name)
        Next
        Console.WriteLine()
    Next

    Dim abfrageNach_W = kompilierteLieferantenAbfrage(awContext, "W")

    For Each lieferant In abfrageNach_W
        Console.WriteLine(lieferant.AccountNumber & ": " & lieferant.Name)
        Console.WriteLine(New String("="c, 70))

        'Zu jedem Lieferanten die Produkte ausliefern
        For Each produkt In lieferant.ProductVendor
            Console.Write("ID:" & produkt.ProductID & " ")
            Console.WriteLine("Name:" & produkt.Product.Name)
        Next
        Console.WriteLine()
    Next
```

```
Console.WriteLine()  
Console.WriteLine("Taste drücken, zum Beenden")  
Console.ReadKey()  
  
End Sub  
  
End Module
```

Die erste fett gesetzte Zeile erstellt die kompilierte Abfrage mithilfe einer `CompiledQuery`-Klasse. Diese wird beim Instanziieren auf den Typ des Objektkontexts festgelegt, auf den Typ des Parameters, sowie auf den Rückgabetyp. Der übergebende Lambda-Ausdruck definiert dann nach diesem Typschema die eigentliche Abfrage, die Sie mit dem Lambda-Ausdruck als LINQ-Abfrage definieren.

HINWEIS Mit diesem Schritt haben Sie nicht die Abfrage selbst definiert, sondern nur einen Funktions-Delegaten der eine Abfrage zurück liefert, wenn Sie ihm die entsprechenden Parameter übergeben.

Die Nutzung der Abfrage ist dann anschließend leicht: Sie definieren eine weitere Abfragevariable (die dann natürlich selbst wieder vom Typ `IQueryable(Of Type)` definiert sein wird), die die vorkompilierte Abfrage zurück liefert, auf die der Funktions-Delegat zeigt, und bei der Zuweisung übergeben Sie die Parameter für die Abfragedurchführung. Wie das in der Praxis genau ausschaut, sehen Sie anhand der beiden weiteren fett gesetzten Codezeilen im Beispieldressing.

Daten verändern, speichern, einfügen und löschen

Wie bei LINQ to SQL, ist auch bei LINQ to Entities das Abfragen von Daten das Eine, das Modifizieren von Daten aber eine ganz andere Geschichte. Auf welche Weise Sie Daten abfragen, haben die letzten Abschnitte von verschiedenen Seiten beleuchtet. Alle Operationen, die sich mit dem Ändern von Daten beschäftigen, besprechen die folgenden Abschnitte.

LINQ to Entities unterscheidet sich hier von LINQ to SQL. Während bei LINQ to SQL der Datenkontext mit den an ihm hängenden `Table(Of T)`-Auflistungen für die Änderungsverfolgung zuständig ist, übernehmen diese Aufgabe in LINQ to Entities die Entitätsobjekte selbst. Das können sie deswegen, weil diese in LINQ to Entities nicht nur einfache POCOs (*Plain Old CLR Objects* – also reine, nur von `Object` abgeleitete Business-Objekte) darstellen, sondern von der Klasse `EntityObject` abgeleitet sind.

Die folgenden Abschnitte beschreiben, wie Sie mithilfe des Entity Frameworks Datenänderungen in dem durch das EDM vorgegebene Objektmodell programmtechnisch vornehmen, und die entsprechenden Änderungen in der Datenbank fest schreiben.

Datenänderungen mit `SaveChanges` an die Datenbank übermitteln

Schauen wir uns das Beispiel an, das sich im Modul *DatenBearbeiten* befindet.

HINWEIS Denken Sie auch hier wieder daran, das Startobjekt auf das Modul *DatenBearbeiten* im Beispielprojekt zu ändern!

Das folgende Beispiel ruft einen Lieferanten ab (und zwar den Einzigen, der mit der Zeichenfolge »Hybrid« beginnt), liefert Ihnen diesen aber nicht als Auflistung mit einem Element sondern durch den Einsatz der Methode `First` (siehe Listing) bereits als Instanz der Klasse `Vendor`:

HINWEIS Single wie in LINQ to SQL können Sie bei LINQ to Entities-Abfragen *nicht* verwenden. Sie verwenden dazu die `First`-Methode. Die `First`-Methode sorgt bei der SQL Server-Implementierung des Entity Data Providers dafür, dass die TOP 1-Klausel vor der eigentlichen SELECT-Abfrage platziert wird. `First` löst, im Gegensatz zu `Single`, auch keine Ausnahme aus, wenn nicht genau *ein* Datensatz in der Ergebnisliste zurückgeliefert wird.

```
Sub DatenÄndern()
    Dim awContext As New AWEntities()

    Dim lieferant = (From lieferantItem In awContext.Vendors
                     Where lieferantItem.Name.ToLower.StartsWith("hybrid") _
                     Order By lieferantItem.Name).First

    'Hin- und her ändern, damit es nicht nur einmal funktioniert:
    If lieferant.Name = "Hybrid Bicycle Center" Then
        lieferant.Name = "Hybrid Fahrad Center"
    Else
        lieferant.Name = "Hybrid Bicycle Center"
    End If

    'Änderungen zurückschreiben
    awContext.SaveChanges()

    Console.WriteLine()
    Console.WriteLine("Taste drücken, zum Beenden")
    Console.ReadKey()

End Sub
```

Im Anschluss an die Abfrage wird die Eigenschaft `Name` der `Vendor`-Klasse geändert. Und im Prinzip ist das auch schon alles, was Sie machen müssen, wenn Sie Änderungen durchführen möchten: Sie ändern lediglich, wie hier im Beispiel, die korrelierende Eigenschaft, die der Eigenschaft im konzeptionellen Modell entspricht – das Mapping auf die eigentliche Datenbank erledigt das Entity Framework.

Um die Änderungen anschließend in die Datenbank zurückzuschreiben, rufen Sie, anders als bei LINQ to SQL, die Methode `SaveChanges` des Objektkontextes auf – die dafür notwendigen T-SQL-UPDATE-Befehle generiert dann der ADO.NET Entity Data Provider selbstständig.

Diese Befehle, die für das Aktualisieren der Datensätze generiert werden, sind bei LINQ to Entities nach derzeitigem Wissensstand nur noch mit dem SQL-Profiler-Tool einsehbar (siehe auch Abschnitt »Generierte SQL-Anweisungen unter die Lupe nehmen« auf Seite 994). Die Abfrage des Zieldatensatzes erfolgt im Beispiel mit:

```
SELECT TOP (1)
[Extent1].[VendorID] AS [VendorID],
[Extent1].[AccountNumber] AS [AccountNumber],
[Extent1].[Name] AS [Name],
```

```
[Extent1].[CreditRating] AS [CreditRating],  
[Extent1].[PreferredVendorStatus] AS [PreferredVendorStatus],  
[Extent1].[ActiveFlag] AS [ActiveFlag],  
[Extent1].[PurchasingWebServiceURL] AS [PurchasingWebServiceURL],  
[Extent1].[ModifiedDate] AS [ModifiedDate]  
FROM [Purchasing].[Vendor] AS [Extent1]  
WHERE (CAST(CHARINDEX(N'hybrid', LOWER([Extent1].[Name])) AS int)) = 1  
ORDER BY [Extent1].[Name] ASC
```

Das Aktualisieren übernimmt beim Aufruf von `SaveChanges` dann folgende Anweisung, die dazu vom Entity Framework generiert wird.

```
exec sp_executesql N'update [Purchasing].[Vendor]
set [Name] = @0
where ([VendorID] = @1)
',N'@0 nvarchar(21),@1 int',@0=N'Hybrid Bicycle Center',@1=52
```

Einfügen von verknüpften Daten in Datentabellen

Das Einfügen von Daten in Tabellen geschieht durch den Objektkontext, bzw. durch die Navigationseigenschaften der Entitätsobjekte. Das heißt im Klartext:

- Sie fügen einen Datensatz in einem Entitätsobjekt ein, indem Sie eine Instanz erstellen, und die `AddToEntitätsname`-Methode des Objektkontextes aufrufen, um sie hinzuzufügen. Einen Lieferanten würden Sie also mit `AddToVendors`, einen Kunden mit `AddToCustomer` hinzufügen.
 - Sie fügen einen Datensatz in einer Navigationseigenschaft einfach mit der `Add`-Methode ein – also anders wie bei LINQ to SQL wirklich mit dem standardmäßigen »Hinzufügen«-Befehl einer klassischen Auflistung. Möchten Sie also beispielsweise im Ergebnis nicht nur ein Lieferantenprodukt der Tabelle `ProductVendor` hinzufügen, sondern auch gleichzeitig die Relation zur Lieferanten-Tabelle herstellen, würden Sie eine neue `ProductVendor`-Instanz mit der `Add`-Methode der `ProductVendor`-Navigationseigenschaft einer `Vendor`-Entitätsinstanz hinzufügen.

Auf diese Weise bauen Sie zunächst die Verknüpfungen der neuen Datensätze untereinander auf, und erst ganz zum Schluss sorgen Sie mit der `SaveChanges`-Methode des Objektkontextes dafür, dass die neuen Objekte durch entsprechende SQL-`INSERT`-Anweisungen in die Datenbank gelangen. Das folgende Beispiel zeigt, wie es geht:

```

        .DiscontinuedDate = #12/31/2011#, _
        .SellStartDate = #12/10/2008#, _
        .SellEndDate = #12/31/2011#, _
        .SafetyStockLevel = 5, _
        .ReorderPoint = 2}

'Neues Lieferantenprodukt erstellen, das Produkt mit diesem,
'und dieses mit dem Lieferanten verknüpfen soll

Dim neuesLieferantenProdukt = New ProductVendor With {.Product = neuesProdukt, _
        .StandardPrice = 59, _
        .ModifiedDate = Now, _
        .UnitMeasureCode = "PC", _
        .AverageLeadTime = 19, _
        .MinOrderQty = 1, _
        .MaxOrderQty = 10}

awContext.AddToVendors(neuerLieferant)
neuerLieferant.ProductVendor.Add(neuesLieferantenProdukt)
awContext.SaveChanges()

End Sub

```

Sie sehen, dass das Beispiel hier das Hinzufügen der einzelnen neuen Entitätsobjekte bewusst in einer der Relationsvorschrift der Datenbank entgegengesetzten Richtung vornimmt. Ein Einfügen der neuen Datensätze in dieser Reihenfolge auf Datenbankebene würde natürlich sofort zu einer Foreign-Key-Relation-Verletzung führen. Doch darum müssen Sie sich im Entity Framework nicht kümmern, denn der Objektkontext kennt natürlich die durch das Storage Modell definierten Relationen der Tabelle und generiert die INSERT-Anweisungen in der Reihenfolge, in der sie diese Beziehungen nicht verletzen können. So produziert – durch den Profiler überprüft – das obige Beispiel folgende INSERT-Anweisungen:

```

exec sp_executesql N'insert [Production].[Product]([Name], [ProductNumber], [MakeFlag],
[FinishedGoodsFlag], [Color], [SafetyStockLevel], [ReorderPoint], [StandardCost], [ListPrice], [Size],
[SizeUnitMeasureCode], [WeightUnitMeasureCode], [Weight], [DaysToManufacture], [ProductLine], [Class],
[Style], [ProductSubcategoryID], [ProductModelID], [SellStartDate], [SellEndDate], [DiscontinuedDate],
[rowguid], [ModifiedDate])
values (@0, @1, @2, @3, null, @4, @5, @6, @7, null, null, null, null, @8, null, null, null, null, null,
@9, @10, @11, @12, @13)
select [ProductID]
from [Production].[Product]
where @@ROWCOUNT > 0 and [ProductID] = scope_identity()',N'@0 nvarchar(38),@1 nvarchar(4000),@2 bit,@3
bit,@4 smallint,@5 smallint,@6 decimal(19,4),@7 decimal(19,4),@8 int,@9 datetime2(7),@10
datetime2(7),@11 datetime2(7),@12 uniqueidentifier,@13 datetime2(7)',@0=N'Visual Basic 2008 - Das
Entwicklerbuch',@1=N'',@2=0,@3=0,@4=5,@5=2,@6=0,@7=59.0000,@8=0,@9='2008-12-10 00:00:00',@10='2011-12-31
00:00:00',@11='2011-12-31 00:00:00',@12='00000000-0000-0000-0000-000000000000',@13='2008-11-13
23:05:53.3700741'

exec sp_executesql N'insert [Purchasing].[Vendor]([AccountNumber], [Name], [CreditRating],
[PreferredVendorStatus], [ActiveFlag], [PurchasingWebServiceURL], [ModifiedDate])
values (@0, @1, @2, @3, @4, null, @5)
select [VendorID]
from [Purchasing].[Vendor]

```

```

where @@ROWCOUNT > 0 and [VendorID] = scope_identity()',N'@0 nvarchar(11),@1 nvarchar(13),@2 tinyint,@3
bit,@4 bit,@5 datetime2(7)',@0=N'ACTDEV00001',@1=N'ActiveDevelop',@2=1,@3=0,@4=0,@5='2008-11-13
23:05:53.3680739'

exec sp_executesql N'insert [Purchasing].[ProductVendor] ([ProductID], [VendorID], [AverageLeadTime],
[StandardPrice], [LastReceiptCost], [LastReceiptDate], [MinOrderQty], [MaxOrderQty], [OnOrderQty],
[UnitMeasureCode], [ModifiedDate])
values (@0, @1, @2, @3, null, @4, @5, null, @6, @7)
',N'@0 int,@1 int,@2 int,@3 decimal(19,4),@4 int,@5 int,@6 nchar(2),@7
datetime2(7)',@0=1014,@1=124,@2=19,@3=59.0000,@4=1,@5=10,@6=N'PC',@7='2008-11-13 23:05:53.3830754'

```

Daten aus Tabellen löschen

Auch das Löschen von Entitätsobjekten aus bestehenden Auflistungen könnte nicht einfacher sein – wie beim Einfügen von Daten spielt die Reihenfolge, mit der Sie Entitätsobjekte von ihren übergeordneten Entitäten trennen, quasi keine Rolle, denn das Entity-Framework kümmert sich um die Generierung der richtigen Reihenfolge der DELETE-Statements, damit es keine Foreign-Key-Verletzungen gibt.

In der Regel werden Sie durch irgendeine Abfrage ein Objekt ermitteln, das es aus einer Datenbanktabelle zu löschen gilt. Und dabei wird es oft vorkommen, dass entsprechende untergeordnete Zeilen in anderen Tabellen direkt mitgelöscht werden müssen.

Das Angenehme beim Entity Framework ist, dass es dabei kaskadierendes Löschen unterstützt. Wenn Sie also einen Datensatz einer Haupttabelle löschen, werden in der Standardeinstellung die durch Foreign-Keys verknüpften Datensätze in untergeordneten Tabellen direkt mitgelöscht.

Im folgenden Beispiel ermitteln wir zunächst ein Lieferantenprodukt anhand seines Namens. Auch hier verwenden wir, wie schon im Abschnitt »Anonymisierungsvermeidung bei Abfragen in verknüpften Tabellen« auf Seite 1000 eine verschachtelte From-LINQ-Abfrage, um ein Entitätsobjekt zu ermitteln, dessen Abfragekriterium allerdings in einer verknüpften Tabelle liegt (wir selektieren nach dem Namen des Produktes, wollen aber einen Datensatz aus der Lieferantenprodukte-Tabelle ermitteln. In den Lieferantenprodukten gibt es aber kein Namens-Feld, nur in der verknüpften Produkte-Tabelle, die die Details wiedergibt). Wir bekommen also auf diese Weise ein Lieferantenprodukt zurück. Ziel ist es aber nun, nicht nur das Lieferantenprodukt zu löschen, sondern auch die korrelierenden Detailinformationen aus der Tabelle *Product* sowie den Lieferanten aus der Tabelle *Vendor* selbst.

Wir bedienen uns zur Ermittlung der gesuchten Entitätsobjekte wieder des manuellen Lazy-Loadings, speichern dann allerdings die Referenzen auf *Product*- und *Vendor*-Objekt zunächst zwischen. Der Hintergrund: Sie entfernen Objekte mit *DeleteObject* aus den entsprechenden Navigationseigenschaftslisten bzw. aus dem Objektkontext, und lösen damit später die DELETE-Anweisung für den SQL Server aus. Allerdings gehen mit *DeleteObject* auch die .NET-Objektreferenzen³ der Navigationseigenschaftsobjekte verloren. Aus diesem Grund ist es sinnvoll, die .NET-Referenzen mit Hilfsvariablen zu sichern, und diese dann *DeleteObject* durch die temporären Objektvariablen zu übergeben. Im unten stehenden Beispiel würden Sie anderenfalls eine Ausnahme auslösen.

³ Ich spreche hier bewusst von .NET-Referenzen, um Verwirrungen zwischen logischen Referenzen zwischen Datentabellen und Referenzen auf Objekte aus CLR-Sicht zu vermeiden.

```
Sub DatenLöschen()

    Dim awContext As New AWEntities()

    Dim lieferantProduct = (From lieferantProduktItem In awContext.ProductVendor _
                           From productItem In awContext.Product _
                           Where productItem.ProductID = lieferantProduktItem.ProductID _
                           Where productItem.Name = "Visual Basic 2008 - Das Entwicklerbuch" _
                           Select lieferantProduktItem).First

    Console.WriteLine(lieferantProduct.ProductID)
    lieferantProduct.ProductReference.Load()
    lieferantProduct.VendorReference.Load()

    Dim tmpProduct = lieferantProduct.Product
    Dim tmpVendor = lieferantProduct.Vendor

    awContext.DeleteObject(tmpVendor)
    awContext.DeleteObject(tmpProduct)
    awContext.SaveChanges()

End Sub
```

Das kaskadierende Löschen und die Einhaltung der richtigen Reihenfolge kann man an dieser Stelle übrigens sehr gut beobachten: Obwohl wir mit `DeleteObject` nur zwei Entitäten löschen, zieht dies laut SQL-Profiler drei DELETE-Anweisungen nach sich. Das Lieferantenprodukt in der Tabelle *ProductVendor* steht nämlich in Abhängigkeit zu seinem Lieferanten in der Tabelle *Vendor*. So muss also zunächst das Lieferantenprodukt gelöscht werden, bevor der Lieferant aus der *Vendor*-Tabelle entfernt werden kann – das Entity Framework kümmert sich darum automatisch.

```
exec sp_executesql N'delete [Purchasing].[ProductVendor]
where ([ProductID] = @0) and ([VendorID] = @1)',N'@0 int,@1 int',@0=1012,@1=123

exec sp_executesql N'delete [Production].[Product]
where ([ProductID] = @0)',N'@0 int',@0=1012

exec sp_executesql N'delete [Purchasing].[Vendor]
where ([VendorID] = @0)',N'@0 int',@0=123
```

HINWEIS Damit das Löschen in diesem Beispiel funktionieren kann, müssen Sie den Trigger, der an der *Vendor*-Tabelle hängt, entweder deaktivieren oder löschen. Im LINQ to SQL-Kapitel finden Sie zwei Methoden, die zeigen, wie Sie das Löschen und Wiederherstellen des Triggers vornehmen können.

Alternativ können Sie den Trigger auch mit `ALTER TRIGGER Triggername INACTIVE;` deaktivieren. Mit `ALTER TRIGGER Triggername ACTIVE;` können Sie ihn wieder aktivieren. Das Management Studio kann Ihnen beispielsweise dabei helfen.

Concurrency-Checks (Schreibkonfliktprüfungen)

Es gilt das schon für LINQ to SQL gesagte: Solange, wie Daten nur von einem Client aus im SQL-Server geändert werden, wird Ihre Anwendung beim Aktualisieren von Daten kaum Probleme bekommen. Was geschieht aber, wenn dieselben Daten in einer Netzwerkumgebung von zwei Anwendern Ihrer Software gleichzeitig geändert wurden?

Standardmäßig implementiert das Entity Framework das sogenannte Optimistische Konfliktprüfungsmodell. Die darunter liegenden Objektdienste speichern Änderungen in den Objekten, ohne eine Konfliktprüfung durchzuführen.

Um dieses Standardverhalten für bestimmte Eigenschaften zu ändern, ist es angebracht, ihre Einstellung mithilfe des Designers im Konzeptionellen Modell entsprechend anzupassen, und das ist eine ganz einfache Geschichte. Sie selektieren die Eigenschaft in der Entität und stellen ihren Parallelitätsmodus einfach auf *Fixed*.

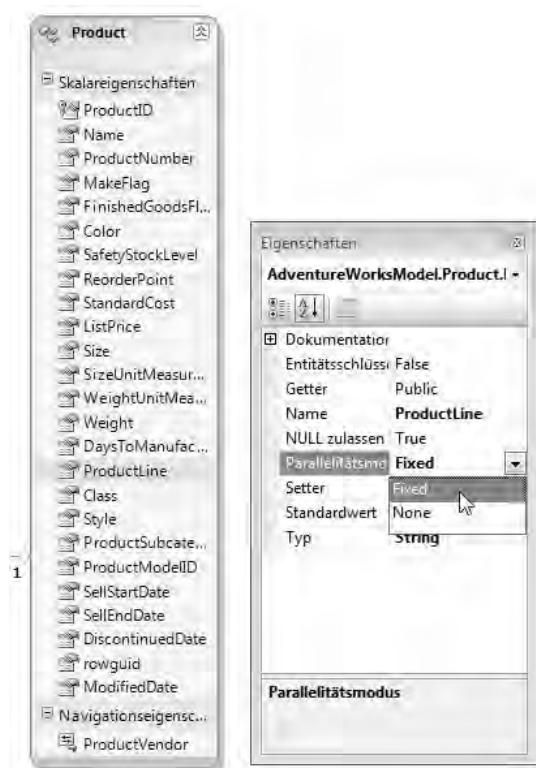


Abbildung 35.13 Um Konfliktprüfung für bestimmte Eigenschaften einer Entität zu aktivieren, stellen Sie ihren Parallelitätsmodus einfach auf Fixed

Wenn diese Einstellung zur Anwendung kommt, führen die Objektdienste eine Änderungsüberprüfung durch, bevor die Änderungen in die Datenbank übertragen werden. Gibt es dabei Konflikte, wird eine `OptimisticConcurrencyException` ausgelöst.

Wenn Sie in Szenarien, die zu häufigen Änderungskonflikten neigen, Änderungen an die Datenbank übermitteln, ist es empfehlenswert, häufiger zwischendurch die Refresh-Methode des Objektkontextes zu verwenden. Refresh definiert, wie Änderungen sich auswirken sollen. Übergeben Sie die Option `StoreWins` (in etwa: *die Speichereinheit – also die Datenbank – gewinnt!*), überschreiben die Objektdienste die Daten im Entitätsobjekt. Die Alternative: Sie übergeben die Option `ClientWins`, damit die Daten, die in den Entitätsobjekten lagern, ihren Siegeszug in die Datenbank antreten können.

Ein Beispiel dafür kann folgendermaßen ausschauen:

```
Dim awContext As New AWEntities()

Dim lieferant = (From lieferantItem In awContext.Vendors
                  Where lieferantItem.Name.ToLower.StartsWith("hybrid") _
                  Order By lieferantItem.Name).First

'Hin- und her ändern, damit es nicht nur einmal funktioniert:
If lieferant.Name = "Hybrid Bicycle Center" Then
    lieferant.Name = "Hybrid Fahrad Center"
Else
    lieferant.Name = "Hybrid Bicycle Center"
End If

Try
    Dim anzahl = awContext.SaveChanges()
    Console.WriteLine("Keine Konflikte bei der Speicherung. " & anzahl & " Aktualisierungen gespeichert.")

    Catch ex As OptimisticConcurrencyException
        awContext.Refresh(RefreshMode.ClientWins, lieferant)

        awContext.SaveChanges()
        Console.WriteLine("Konfliktfehler wurde abgefangen, Änderungen wurden gespeichert!")
    End Try

    Console.WriteLine()
    Console.WriteLine("Taste drücken, zum Beenden")
    Console.ReadKey()

End Sub
```

Wichtig für das Funktionieren dieses Beispiels ist es, dass Sie den Parallelitätsmodus der `Name`-Eigenschaft des Entitätsobjektes `Vendor` zuvor auf `Fixed` gestellt haben, wie in der obenstehenden Abbildung gezeigt.

Ausblick

Das Entity Framework ist ein mächtiges Instrument zum Entwickeln auch der komplexesten Datenbankszenarien. Es ist der erste Wurf und dass es deshalb noch in vielerlei Hinsicht verbesserungswürdig ist, mögen wir Microsoft verzeihen – bis hier hin ist es mehr als vielversprechend. Der Zugang zum Entity Framework ist für Viele sicherlich schwieriger, als der vergleichsweise einfache Einstieg in LINQ to SQL. Und auch wenn LINQ to SQL einerseits leichter zu erlernen ist und im Moment noch stabiler und schneller wirkt und ist, gerade was die Designerunterstützung anbelangt, so dürfen wir nicht vergessen, dass es bis zum Framework 4.0 nicht mehr lange dauert.

Im Rahmen dieses Buchs konnte ich natürlich nicht auf alle Features des Entity-Frameworks eingehen. Im Entity Framework gehört die Verwendung von gespeicherten Prozeduren natürlich zum täglichen Handwerkszeug. Tabellenvererbung und ausgedehnte Mapping-Funktionalitäten konnten an dieser Stelle aus Platzgründen ebenfalls nicht beschrieben werden, genauso wenig wie Transaktionen, die sich aber ganz einfach mithilfe von TransactionScope-Objekten der Enterprise-Services des .NET Frameworks realisieren lassen.

Ich denke aber, dass dieses Kapitel Sie auf alle Fälle in die richtige Richtung gedreht hat und Sie nun losmarschieren können. Sie haben mit dem hier erworbenen Wissen sicherlich das Zeug, leistungsfähige Datenbanken auf Entity Framework-Basis zu entwickeln, die vor allen Dingen eines sind: zukunftssicher! Es gibt noch viel zu entdecken, viel auszuprobieren, sicherlich nicht nur für Sie – auch für uns Autoren. Und ich jedenfalls habe auf dem Weg hierher schon eines in meinen ersten EDM-Projekten feststellen dürfen: Mit dem Entity Framework zu programmieren hat mir, bei allen Macken, die es in dieser derzeitigen ersten Version noch hat, schon jetzt enormen Spaß gemacht.

Teil G

SmartClient-Anwendungen entwickeln

In diesem Teil:

Einführung in SmartClient-Entwicklung	1015
Programmieren mit Windows Forms	1031
Im Motorraum von Formularen und Steuerelementen	1097
Individuelles Gestalten von Elementinhalten mit GDI+	1139
Entwickeln von Steuerelementen für Windows Forms	1169

Kapitel 36

Einführung in SmartClient-Entwicklung

In diesem Kapitel:

Einführung in SmartClient-Entwicklung	1015
Die Gretchenfrage: Windows Forms oder WPF?	1016
Vorüberlegungen zur Smart Client-Entwicklung	1018
Strukturieren von Daten in Windows Forms-Anwendungen	1019

Zentraler Bestandteil einer jeden SmartClient-Anwendung sind Windows- oder WPF-Formulare – Ersteres wird im .NET-Jargon kurz *WinForms* genannt, und um dieses Thema soll es sich primär in diesem Buchteil drehen. Einige Leser meiner bisher publizierten Bücher stellten mir in diesem Zusammenhang häufiger die Frage, wieso ich nicht wenigstens ein Kapitel in Petto habe, in dem ich die Komponenten, aus denen Windows Forms-Anwendungen aufgebaut werden, einzeln erkläre. Die Antwort ist ganz einfach: Es wäre redundant. Ein Entwicklerbuch sollte meiner Meinung nach immer den Anspruch haben, die Möglichkeiten einer Entwicklungsumgebung im Kontext dessen zu erklären, was Sie mit ihr erreichen können. Ihre einzelnen Elemente erklärt die Referenz zu Visual Basic bzw. Visual Studio in ausreichendem Maße. Und mir sei es in diesem Zusammenhang auch einmal gestattet, eine Lanze für die technischen Autoren von Microsoft zu brechen. Denn die Hilfe von Visual Studio ist weitaus besser als ihr Ruf. Wir Entwickler können, wie ich finde, nicht einerseits beschweren, wieso die deutsche Version immer (jetzt wirklich: nur!) ein, zwei Monate auf sich warten lässt, und uns dann lautstark beschweren, wenn es innerhalb eines automatisierten Prozesses hier und da Übersetzungsfehler gegeben hat, weil der Übersetzer das, was er übersetzen musste, vielleicht nicht im Kontext erkennen konnte. In zwei Monaten eine mehrere Gigabyte umfassende Online-Hilfe in die verschiedensten Sprachen zu lokalisieren, ist eine Meisterleistung, die Jedermanns Hochachtung verdient!

Als SmartClient-Anwendungen werden solche Anwendungen bezeichnet, die die lokale Rechenleistung des Computers nutzen, um Daten irgendwelcher Art zu verarbeiten. Es ist eigentlich die einfachste Art des parallelen Rechnens, wenn man so will. Web-Anwendungen haben im Gegensatz dazu eigentlich nur einen ungeheuren Vorteil, nämlich den des einfachen Auslieferns – das findet als Solches nämlich gar nicht statt. SmartClients müssen immer auf dem Zielrechner installiert werden, und auch wenn .NET-Techniken wie das ClickOnce-Deployment sehr stark in Richtung »Null-Aufwand« gehen, ist es bei Webanwendungen immer noch einfacher, da hier einfach das Aufrufen der entsprechenden Startseite Zugriff auf die Anwendung nehmen lässt. Doch Web-Anwendungen haben einen entscheidenden Nachteil, der mit Techniken des Web 2.0 auch nicht wirklich besser geworden ist: Anwendungen, die Validierung von vielen Eingaben aufgrund bestimmter Algorithmen vornehmen müssen, gehen oft zu langsam. Durch Postbacks zum Webserver sind die Reaktionszeiten dabei oft zu groß. Und: Webanwendungen haben einen viel höheren Design-Aufwand – ein Vorteil, der sich allerdings bei WPF-Anwendungen, zum großen Teil relativiert. Doch heutzutage sind auch die einfachsten Bürocomputer so leistungsstark, dass es fast schon eine Schande ist, deren Ressourcen ungenutzt zu lassen, und dem Webserver die ganze Business-Logik zu überlassen.

Die Gretchenfrage: Windows Forms oder WPF?

In Kapitel 6 im Rahmen der Einführung zu WPF habe ich es schon erwähnt: Gerade bei der Migration von VB6-Anwendungen, die derzeit in großen Mengen portiert werden müssen, weil sie dank Windows 7 nach letzten Erkenntnissen gar nicht mehr funktionieren werden, stellt sich natürlich auch für Neuentwicklungen die Frage, welches die richtige Plattform für die Entwicklung von Geschäftsanwendungen ist. Die Stimmen dazu gehen auch bei Microsoft stark auseinander. Auch wenn WPF grundsätzlich erstmal fast alles ermöglicht, was WinForms auch kann, gibt es berechtigerweise immer noch eine ganze Reihe grundsätzlicher Bedenken, sich für WPF und gegen WinForms bei Neuentwicklungen zu entscheiden. Und die sind:

- Die Einarbeitung in WPF verlangt einen sehr großen Aufwand, und es vergeht einiges an Zeit, bis Teams in der Lage sind, wirklich anspruchsvolle UIs auf Basis der WPF zu entwerfen.
- Entwickler sind keine Designer. Die WPF berücksichtigt das durch das XAML-Konzept, zu dem Sie im nächsten Abschnitt mehr erfahren können. Aber wirklich professionelle UIs bedeuten gerade deswegen, dass Sie einen Designer in Ihrem Team benötigen.

- Die Designer-Unterstützung ist in der SP1-Version von Visual Studio 2008 zwar besser geworden, aber man möchte sagen: Eigentlich immer noch nur rudimentär, jedenfalls wenn man sie mit Windows Forms vergleicht. Auch wenn Sie als Entwickler bereits richtig firm im Umgang mit der WPF sind, Sie werden für gleiche Formularaufbauten in der WPF immer mehr Zeit als unter Windows Forms benötigen. Denn für viele Dinge, wie beispielsweise das Aufbauen von Pulldown-Menübüumen, gibt es einfach noch keine gescheite Designer-Unterstützung – Sie müssen hier mit dem XAML-Editor XAML-Code manuell pflegen. Die XAML-Powertoys von Karl Shifflet können bei der Erstellung von Formularen mit Assistentenunterstützung oder zur automatischen Generierung aus Business-Objektklassen übrigens eine riesengroße Hilfe sein (IntelliLink **G3601**).
- Und wie schon im entsprechenden Hinweiskasten am Anfang dieses Kapitels erwähnt: Wenn Sie professionelle WPF-Anwendungen entwickeln wollen, werden Sie um den Einsatz einer speziell auf WPF abgestimmten Designer-Software nicht drum herumkommen. Ihr Name: *Expression Blend* – und es gibt sie inzwischen in der zweiten Version. Der unverständliche Nachteil: Expression Blend ist NICHT in Visual Studio 2008 enthalten – Sie müssen es wohl oder übel dazu kaufen. Spätestens hier bietet sich der Kauf eines so genannten MSDN-Abos an: Schon in der MSDN-Professional Edition dieses Abonnements sind sowohl Visual Studio 2008 Professional als auch Expression Blend 2 enthalten. Mehr zum Thema MSDN erfahren Sie unter dem IntelliLink **A0603**.
- Anwendungen müssen in vielen Fällen immer noch Windows 2000SP4-Clients bedienen können, – entscheidet sich ein Entwicklungsteam für die WPF, entscheidet es sich auch gleichzeitig für mindestens Windows XP SP2 als Basis für seine Anwendungen. Und auch da gibt es Unterschiede zwischen Windows Vista/Windows 7 und Windows XP: Die Performance von Vista ist im Gegensatz zu Windows XP ungleich höher – und das liegt nicht nur daran, dass für Windows Vista/Windows 7-Geräte in der Regel modernere Grafikkomponenten verbaut wurden.
- Gerade die sollten Sie aber berücksichtigen – alte Onboard-Grafikkarten lassen bestimmte UI-Szenarien, die mit der WPF entwickelt worden sind, unter Windows XP wirklich wie lahme Krücken aussehen.
- Es fehlen zurzeit immer noch Steuerelemente in der WPF, die zum Aufbau von WPF-Anwendungen aber sehr wichtig wären. Das Wichtigste dabei ist sicherlich die Entsprechung einer *DataGridView*, die Microsoft selbst mit dem SP1 noch nicht als WPF-Version herausgebracht hat. Es gibt allerdings hier von einem Fremdhersteller eine Alternative, die mittlerweile von vielen Entwicklungsteams verwendet wird: Der Hersteller XCEED bietet eine wirklich leistungsfähige Komponente an, die in ihrer professionellen Ausführung allerdings auch einiges kostet. Der Hersteller bietet aber eine Light-Variante umsonst an, mit der bereits sehr viel machbar ist, und frühzeitiges Herunterladen und Registrieren der Komponente ist hier wichtig, denn nur dann dürfen Sie diese Komponente auch in Zukunft ohne das Abführen von Lizenzgebühren in Ihren Anwendungen weiter vertreiben. Der IntelliLink **A0608** bringt Sie auf die Seite des Herstellers. *DateTimePicker* und vollwertiges Kalender-Control werden ebenfalls schmerzlich vermisst – das gleiche gilt für ein *RibbonBar*-Control. Abhilfe schafft hier die vorläufige Version des WPF-Toolkits, mit dem man aber schon recht stabile Anwendungen schreiben kann (siehe IntelliLinks **G3602** und **G3603**). Das Schöne daran: Es liefert nicht nur diese Steuerelemente, sondern auch ein WPF-Grid direkt aus dem Hause Microsoft. Diese Komponenten liegen obendrein auch im Source-Code vor, sodass man auch noch eine ganze Menge über das Design solcher Steuerelemente lernen kann. Und die letzte gute Nachricht in diesem Zusammenhang: Die Kalender-Steuerelemente sind *endlich Nullable-fähig!*

Trotz dieser Einschränkungen sollten Sie aber auch die Pro-Punkte berücksichtigen:

- Schon in der 3.5-Version des Frameworks hat Microsoft in Sachen Windows Forms nichts mehr unternommen – sie unterscheidet sich faktisch gar nicht von der 2.0-Version. Das lässt darauf schließen, dass Windows Forms zwar sicherlich noch gepflegt, aber bestimmt nicht mehr wesentlich erweitert wird.
- WPF-Anwendungen sind zukunftssicher, und ihre Technik wird sicherlich in großem Maße in neue Betriebssysteme Einzug halten. WPF-Anwendungen lassen sich deshalb in Zukunft sicherlich leichter ausbauen, auch im Hinblick auf neue Eingabetechniken, wie sie Windows 7 mit Touchscreens und Fingergesten einführt.
- Und ein netter Indikator für die Entwicklungsrichtung ist sicherlich auch dieser Punkt: Nach derzeitigem Stand der Dinge wird Visual Studio 2010 eine Benutzeroberfläche bekommen, die komplett auf der Windows Presentation Foundation basiert. Alleine diese Tatsache spricht Bände, wie ich meine.

Vorüberlegungen zur Smart Client-Entwicklung

Jedes Windows-Formular dient als Träger für Steuerelemente, mit denen der Anwender eine SmartClient-Anwendung letzten Endes bedienen kann. Dabei gibt es im Grunde genommen nur drei Arten von Formularen, die in Ihren Programmen zum Einsatz kommen:

- Formulare, die Daten darstellen,
- Formulare, die der Datenerfassung dienen, und
- Formulare, mit denen Sie Programmoptionen einstellen.

Und eigentlich sind die Formulare, die unter die letzte Kategorie fallen, auch nichts anderes als Formulare, die der Datenerfassung dienen.

Typische SmartClient-Anwendungen unter Windows Forms (und natürlich auch unter WPF) haben dabei immer den gleichen Aufbau: Sie enthalten ein Hauptanwendungsfenster, das den eigentlichen Arbeitsbereich der Anwendung darstellt. Und von diesem Formular aus verzweigt der Funktionsbereich der Anwendung mithilfe weiterer Formulare, die entweder dazu dienen, weitere Daten (vielleicht anderen Typs) zu erfassen oder die Parameter bestimmter Funktionen einzustellen, die auf die Hauptdaten der Anwendung in irgendeiner Form angewendet werden (doch auch das ist ja wieder eine Art von Datenerfassung).

Unterscheiden kann man schließlich noch zwei Grundformen der Datenaufbereitung und Form der Verarbeitung durch den Anwender: Es gibt Anwendungstypen, die ihre Daten in mehrere Dokumente aufteilen, und in so genannten MDI-Anwendungen (*Multi Dokument Interface* – etwa: Mehrdokumenten-Benutzerschnittstelle) für den Anwender zur Bearbeitung zur Verfügung halten. Verschiedene Datensammlungen gleicher Art (Textdokumente, Bilder, Adresskarten, etc.) können dabei in verschiedenen untergeordneten Fenstern zur gleichen Zeit dargestellt und zur Bearbeitung angeboten werden.

HINWEIS Eine klassische Unterscheidung zwischen MDI- und SDI-Anwendungen gibt es in der derzeitigen Ausbaustufe der Windows Presentation Foundation leider nicht.

Im Gegensatz dazu gibt es die häufiger vorkommenden SDI-Anwendungen (*Single Document Interface*-Anwendungen), bei denen die Daten eben nicht über mehrere Dokumentenfenster verteilt werden, sondern lediglich ein einzelnes Fenster durch das die Anwendung den vorgegebenen Datentyp zur Bearbeitung anbietet.

Für welche Art der Benutzeroberfläche sich ein Entwickler entscheidet, ist eigentlich mehr Geschmackssache als funktionelle Notwendigkeit. Viele Grafikprogramme sind als MDI-Anwendungen ausgelegt – der Anwender kann also »gleichzeitig« mehrere Grafiken als untergeordnete Fenster öffnen; kann er sie aber wirklich gleichzeitig bearbeiten? Moderne Computer ließen das, was die Performance anbelangt, sicherlich zu. Die »Schwachstelle«, wenn man das als solches überhaupt bezeichnen kann, ist dabei eher der Anwender. Ich jedenfalls würde mir es nicht zutrauen, zwei Texte wirklich gleichzeitig zu schreiben, und Microsoft Word ist wahrscheinlich aus ähnlichen Überlegungen bestes Beispiel dafür, wie aus einer MDI-Anwendung (bis Word 97) sinnvollerweise eine SDI-Anwendung (ab Word 2000) werden kann.

Doch ganz gleich welchen Aufbau einer SmartClient-Anwendung Sie auch wählen und welche Art von Daten Ihre Anwendung auch bearbeitet: Entscheiden Sie sich für den Einsatz von Windows-Formularen unter .NET, stehen Sie immer wieder vor der Lösung derselben Detailprobleme. Um nur einige zu nennen:

- Wie strukturieren SmartClient-Anwendungen am besten die Daten, die sie verwalten und dem Anwender zur Bearbeitung anbieten?
- Wie kann man Formulare aufrufen, die Parameter für andere Formulare »einholen«?
- Wie kann die Dateneingabe für den Anwender benutzerfreundlich erfolgen?
- Welche Techniken kann man anwenden, um eine Benutzeroberfläche ergonomischer zu gestalten und sie – getreu dem Motto: »das Auge isst mit« – ohne großen Aufwand verschönern?
- Wie geht man vor, um eine Benutzeroberfläche auch in anderen Sprachen anbieten zu können?

Dieses Kapitel setzt sich exakt mit dieser Problematik auseinander, und Sie sehen, dass es einen anderen Ansatz verfolgt, als Sie ihn vielleicht aus vielen anderen Büchern kennen.

Natürlich kann ein Windows Forms-Kapitel, wie eingangs schon gesagt, hunderte von Seiten füllen, indem es beschreibt, wie das 76. Steuerelement im Detail funktioniert und über welche Eigenschaften, Methoden und Ereignisse es verfügt. Doch damit sind Sie noch keinen Schritt weiter, denn es geht ja schließlich darum, das Gelernte im Kontext der Anwendungsentwicklung umzusetzen. Und außerdem können Sie sich diese Art von Information auch aus der Online-Hilfe von Visual Studio beschaffen.

Viel interessanter ist es doch, Lösungen für verschiedene, häufig auftretende Problemstellungen zu bekommen, und genau diesen Ansatz möchte dieses Kapitel verfolgen. Technische Details, die natürlich für das Verständnis des einen oder anderen Ansatzes notwendig sind, kommen in den jeweiligen Erklärungen natürlich nicht zu kurz.

Strukturieren von Daten in Windows Forms-Anwendungen

Stellen Sie sich vor, Sie stehen vor der Aufgabe, eine kleine, einfache Adressenverwaltung zu entwickeln. Und um das Szenario wirklich einfach zu halten, gehen wir bei diesem Beispiel für den Moment davon aus, dass Sie die Daten nicht in einer Datenbank sondern in einer einfachen Datei speichern. Die Frage, die sich vielen als erstes stellt, lautet dann oftmals: Wie schaffe ich es, dass meine Anwendung von allen relevanten Stellen aus an die Daten herankommt? Und damit stehen viele bereits vor dem ersten wirklich schwerwiegenden Problem.

Das eigentliche Problem an dieser Stelle ist aber gar nicht die noch nicht bekannte Antwort. Im Grunde genommen ist es nämlich schon die Frage, die falsch gestellt wurde: Auch wenn sie die Klassenprogrammierung bereits aus dem Effeff kennen, fällt es vielen Entwicklern schwer, die OOP-Programmierung auch »im großen Rahmen« umzusetzen. Sie möchten nämlich, dass die Daten irgendwo zentral (und vor allen Dingen: global zugänglich) irgendwo in der Anwendung beherbergt werden. Und wenn, um beim Eingangsbeispiel zu bleiben, eine *Adressen-Bearbeiten*-Funktion aufgerufen wird, dann wird einfach ein neuer Dialog aufgerufen, der dann die Adressen entsprechend bearbeitet und dazu selbst auf den Gesamtdatenbestand der Anwendung zugreift. Soll eine neue Adresse erfasst werden, dann passiert der gleiche Datenzugriff eben aus dem »*Adresse-Neu-Anlegen*-Dialog« heraus. Jede Programmfunction greift also nach Gutdünken auf den Datenbestand zu. Das Chaos ist dabei buchstäblich vorprogrammiert.

Der Ansatz sollte ein anderer sein. In einer gelungenen Windows Forms-Anwendung sollte eine bestimmte Komponente die wesentlichen Daten, die der Anwender bearbeiten können soll, kapseln. Und nur diese Komponente sollte in der Lage sein dürfen, bestimmte Daten zum Bearbeiten »herauszurücken« oder Änderungen bestimmter Daten wieder entgegenzunehmen. Andere Komponenten stellen die Datenstrukturen selbst dar. Und wieder andere Komponenten übernehmen die Kommunikation mit dem Anwender.

Aus diesem Grund ist es bei der OOP-Programmierung gängige Praxis, Anwendungen in verschiedene Ebenen bzw. Schichten – (englisch: *Layer* oder *Tier*) zu unterteilen:

- Eine Schicht stellt dabei die Datenschicht bereit. Das sind in der Regel Klassen, die die eigentlichen Daten beherbergen.
- Eine andere Schicht stellt die Anwendungslogik (auch *Geschäftslogik* oder neudeutsch: *Business Logic*) bereit. Das sind wiederum Klassen, die die Klassen, die von der Datenschicht zur Verfügung gestellt werden, einbinden und kapseln und die Funktionalität zur Verfügung stellen, um die Daten im Sinne des Anwenders aufzubereiten und zu verarbeiten.
- Eine letzte Schicht stellt die Schnittstelle zwischen dem Benutzer und der Anwendungslogik her. Sie hat nur Delegatenfunktion, was in diesem Zusammenhang bedeutet: Sie nimmt Anwenderanforderungen entgegen und leitet sie an die Schicht weiter, die die Anwendungslogik enthält.

Die Einteilung in solche Schichten bringt den Entwicklern dann gleich mehrere Vorteile, die sich vor allen Dingen längerfristig auswirken:

- Verschiedene Ihrer Anwendungen können die einzelnen Komponenten bzw. Schichten, aus denen eine mehrschichtige Anwendung besteht, ebenfalls verwenden.
- Umfangreiche Anwendungen bzw. Anwendungen, die komplexe Datenstrukturen zu verwalten haben, lassen sich einfacher in kleinere Portionen aufteilen.
- Ihre Anwendung kann relativ schnell einem Facelift unterzogen werden: Ohne dass die eigentliche Anwendungslogik wirklich ersetzt werden müsste, können Sie ihr ein neues Make-up verpassen und sie sieht damit in null Komma nichts neuer und moderner aus.
- Verschiedene Schichten können einfacher im Team entwickelt werden. Durch die Aufteilung einer Anwendung in Schichten bzw. Komponenten lassen sich komplexe Anwendungen viel besser im Team entwickeln.

Wenn Sie große Teile dieses Buchs bereits studiert haben, dann sind Sie vielleicht schon einem Ansatz einer kleinen Adressverwaltung hier und da begegnet. Ging es in diesen Kapiteln noch darum, bestimmte Details an plausiblen Beispielen zu erklären (Kapitel 23, generische Auflistungen, Kapitel 24, XML-Serialisierung),

so will das folgende Beispiel diese technischen Puzzleteile aufgreifen, neu strukturieren, wieder zusammensetzen und daraus eine kleine, unkomplizierte aber dennoch vollständige Musteranwendung schaffen, die nicht nur prinzipiell und theoretisch zeigt, wie eine saubere OOP-Entwicklung einer SmartClient-Anwendung aussehen sollte.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\ G - WinForms\\Kapitel36\\Adresso01\\

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

So Sie sich die Kapitel 23 und 24 schon zu Gemüte geführt haben, werden Sie bei diesem Projekt in Sachen technischen Gegebenheiten zunächst nichts Neues entdecken. Dennoch unterscheiden sich sowohl Projekt als auch Code deutlich von der letzten Version dieses Beispiels aus Kapitel 23, und das wird bereits beim Betrachten des Projektmappen-Explorers deutlich, denn:

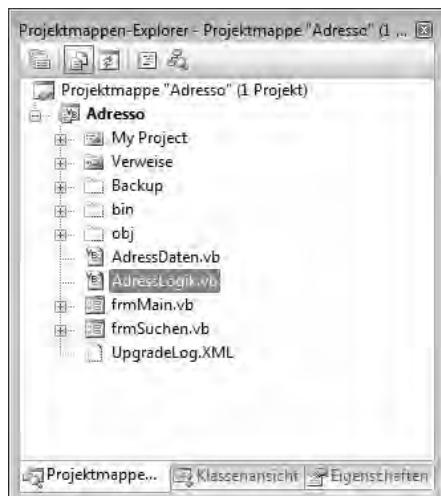


Abbildung 36.1 Schon im Projektmappen-Explorer wird deutlich, wie das Programm in drei Schichten eingeteilt ist

- Die Codedatei *AdressDaten.vb* enthält die Datenschicht der Anwendung – die Adresse-Klasse, die eine einzelne Adresse speichert.
- Die Codedatei *AdressLogik.vb* sorgt dafür, dass die einzelnen Adresse-Datensätze in einer Auflistung zusammengefasst und dort funktionell verwaltet werden. Und hier zeigt sich auch das erste Mal der Unterschied zwischen dieser Version von »Adresso« und der, die Sie noch aus Kapitel 23 kennen. Die gesamte Funktionalität des Programms – beispielsweise zum Sortieren oder Suchen von Adressen – findet nun an dieser Stelle statt, und nicht mehr, wie in der »Vorgängerversion«, lustig verteilt mal bei der Benutzeroberflächenschicht in *frmMain* und *mal*, wie das Serialisieren, in der abgeleiteten Auflistung Adressen. Alle Aufgaben sind in dieser Version, so wie es geplant war, sauber voneinander getrennt (wo-von Sie sich im anschließenden Codelisting auch selber überzeugen können).

- In *frmMain.vb* schließlich, der dritten Schicht, finden nur noch Delegationsaufgaben statt. Funktionen, die der Anwender beispielsweise aus dem Pulldown-Menü abruft, werden einfach an die Adresslogik weitergereicht. Das Komplizierteste, was diese Benutzeroberflächenschicht noch leisten muss, ist die Darstellung der durch die Adresslogik verwalteten Adresse-Objekte im ListView-Steuerelement.

Übrigens: Das Umgestalten des Codes vom Beispiel aus Kapitel 23 zu der Version, wie Sie sie hier sehen, ist ein Vorgang, den man unter dem Begriff *Refactoring* versteht.

Die Codedateien sehen in dieser Version des Beispiels nun folgendermaßen aus:

Datenschicht: AdressDaten.vb

AdressDaten.vb enthält nur noch die Datenstruktur zur Speicherung einer einzelnen Adresse in Form von Adresse-Klasse:

```
<Serializable()>
Public Class Adresse

    'Membervariablen, die die Daten halten:
    Protected myMatchcode As String
    Protected myName As String
    Protected myVorname As String
    Protected myStraße As String
    Protected myPLZ As String
    Protected myOrt As String
    Protected myGeburtsdatum As Date

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse.
    ''' </summary>
    ''' <remarks>Ein parameterloser Konstruktor wird benötigt,
    ''' um XML-Serialisierung zu ermöglichen.</remarks>
    Sub New()
        MyBase.New()
    End Sub

    'Konstruktor - legt eine neue Instanz an
    Sub New(ByVal Matchcode As String, ByVal Name As String, ByVal Vorname As String, _
            ByVal Straße As String, ByVal Plz As String, ByVal Ort As String, _
            ByVal Geburtsdatum As Date)
        myMatchcode = Matchcode
        myName = Name
        myVorname = Vorname
        myStraße = Straße
        myPLZ = Plz
        myOrt = Ort
        myGeburtsdatum = Geburtsdatum
    End Sub

    Public Overridable Property Matchcode() As String
        Get
            Return myMatchcode
        End Get
        Set(ByVal Value As String)
    End Set
End Class
```

```
    myMatchcode = Value
End Set
End Property

Public Overridable Property Name() As String
Get
    Return myName
End Get
Set(ByVal Value As String)
    myName = Value
End Set
End Property

Public Overridable Property Vorname() As String
Get
    Return myVorname
End Get
Set(ByVal Value As String)
    myVorname = Value
End Set
End Property

Public Overridable Property Straße() As String
Get
    Return myStraße
End Get
Set(ByVal value As String)
    myStraße = value
End Set
End Property

Public Overridable Property PLZ() As String
Get
    Return myPLZ
End Get
Set(ByVal Value As String)
    myPLZ = Value
End Set
End Property

Public Overridable Property Ort() As String
Get
    Return myOrt
End Get
Set(ByVal Value As String)
    myOrt = Value
End Set
End Property

Public Overridable Property Geburtsdatum() As Date
Get
    Return myGeburtsdatum
End Get
Set(ByVal Value As Date)
```

```

        myGeburtsdatum = Value
    End Set
End Property

Public Overrides Function ToString() As String
    Return Matchcode & ":" & Name & ", " & Vorname
End Function
End Class

```

Sie werden übrigens feststellen, jedenfalls wenn Sie das Beispiel aus Kapitel 23 kennen, dass in dieser Version ein weiteres Adressdatenfeld hinzugekommen ist – auch wenn es nicht in der Adressenliste des Formulars zu sehen ist. Das Geburtsdatum ist nicht nur ein wesentlicher Bestandteil jeder noch so kleinen Adressverwaltung – dieses Feld eignet sich insbesondere auch zur Demonstration von Eingabeüberprüfungen bei neuen Adressen, mit dem sich der nächste Abschnitt beschäftigt.

TIPP Wenn Sie wissen wollen, wie im Programm die dazugehörigen »Zufallsdatums« erzeugt werden, werfen Sie einen Blick in die Codedatei *AdressLogik.vb* und dort in die Funktion Zufallsadressen. Die Erklärung finden Sie in den Kommentaren am Ende dieser Prozedur.

Anwendungslogik: AdressLogik.vb

Alle Funktionen der Anwendung sind in dieser Codedatei zusammengefasst. Sie bildet damit den Code, der am intensivsten refaktoriert wurde. Die Änderungen im Vergleich zum Beispiel aus Kapitel 23 sind fett hervorgehoben.

```

<Serializable()>
Public Class Adressen
    Inherits List(Of Adresse)

    Private mySortierenNach As AdressenSortierenNach

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse.
    ''' </summary>
    ''' <remarks></remarks>
    Sub New()
        MyBase.New()
    End Sub

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse
    ''' und ermöglicht das Übernehmen einer vorhandenen Auflistung in diese.
    ''' </summary>
    ''' <param name="collection">Die generische Adresse-Auflistung, deren Elemente
    ''' in diese Auflistungsinstanz übernommen werden sollen.</param>
    ''' <remarks></remarks>
    Sub New(ByVal collection As ICollection(Of Adresse))
        MyBase.New(collection)
    End Sub

```

```

'<summary>
'<summary> Gibt alle Adresse-Objekte als Instanz dieser Klasse zurück,
'<summary> in denen der angegebene Suchbegriff vorkam.
'</summary>
'<param name="Text">Suchbegriff, nachdem diese Auflistung durchsucht werden soll.</param>
'<returns></returns>
'<remarks></remarks>
Public Function Suchen(ByVal Text As String) As Adressen

    'In dieser Adressen-Auflistung wird das Suchergebnis gesammelt.
    Dim locAdressen As New Adressen()

    For Each locAdresse As Adresse In Me
        If locAdresse.Name Like Text Then
            locAdressen.Add(locAdresse)
            'Direkt zum 'Next' springen.
            Continue For
        ElseIf locAdresse.Vorname Like Text Then
            locAdressen.Add(locAdresse)
            Continue For
        ElseIf locAdresse.Straße Like Text Then
            locAdressen.Add(locAdresse)
            Continue For
        ElseIf locAdresse.Ort Like Text Then
            locAdressen.Add(locAdresse)
            Continue For
        ElseIf locAdresse.PLZ.ToString Like Text Then
            locAdressen.Add(locAdresse)
        End If
    Next

    Return locAdressen
End Function

```

Die Suchenfunktion wurde im Gegensatz zum Beispiel aus Kapitel 23 auch funktionell abgeändert: Sie findet nun nicht nur die erste Adresse, auf die der Suchbegriff zutraf, sondern liefert eine Adressen-Auflistung zurück, die alle Adressen enthält, in denen der Suchbegriff vorkam. Darüber hinaus findet es den Suchbegriff nicht nur in einer bestimmten einstellbaren Datenspalte, sondern in der gesamten Adresszeile.

```

Public Sub Sortieren(ByVal SortierenNach As AdressenSortierenNach)
    'Das Array mithilfe eines Comparison-Delegaten sortieren
    mySortierenNach = SortierenNach
    Me.Sort(New Comparison(Of Adresse)(AddressOf AdressenVergleicher))

End Sub

'Der Comparison-Delegat zum Vergleichen zweier Elemente.
Private Function AdressenVergleicher(ByVal x As Adresse, ByVal y As Adresse) As Integer

    Try
        'Sortierung in Abhängigkeit zur Suchspalte durchführen
        Select Case mySortierenNach
            Case AdressenSortierenNach.Matchcode
                Return x.Matchcode.CompareTo(y.Matchcode)
            Case AdressenSortierenNach.Name
                Return x.Name.CompareTo(y.Name)
            Case AdressenSortierenNach.Ort
                Return x.Ort.CompareTo(y.Ort)
        End Select
    Catch ex As Exception
        'Fehlerbehandlung
    End Try
End Function

```

```

        Case AdressenSortierenNach.PLZ
            Return x.PLZ.CompareTo(y.PLZ)
        Case AdressenSortierenNach.Straße
            Return x.Straße.CompareTo(y.Straße)
        Case Else
            Return x.Vorname.CompareTo(y.Vorname)
        End Select
    Catch ex As Exception
        'Afangen, dass einer der Suchstrings Nothing ist.
        'Der ist dann immer kleiner als alle anderen!
        Return -1
    End Try
End Function

Serialisiert alle Elemente dieser Auflistung in eine XML-Datei.
Der Dateiname der XML-Datei.

Sub XMLSerialize(ByVal Dateiname As String)
    Dim locXmlWriter As New XmlSerializer(GetType(Adressen))
    Dim locXmlDatei As New StreamWriter(Dateiname)
    locXmlWriter.Serialize(locXmlDatei, Me)
    locXmlDatei.Flush()
    locXmlDatei.Close()
End Sub

Generiert aus einer XML-Datei eine neue Instanz dieser Auflistungsklasse.
Name der XML-Datei, aus der die Daten für diese Auflistungsinstanz entnommen werden sollen.


Public Shared Function XmlDeserialize(ByVal Dateiname As String) As Adressen
    Dim locXmlLeser As New XmlSerializer(GetType(Adressen))
    Dim locXmlDatei As New StreamReader(Dateiname)
    Return CType(locXmlLeser.Deserialize(locXmlDatei), Adressen)
End Function

Liefert eine Instanz dieser Klasse mit Zufallsadressen zurück.
Anzahl der Zufallsadressen, die erzeugt werden sollen.


Public Shared Function ZufallsAdressen(ByVal Anzahl As Integer) As List(Of Adresse)
    'Aus Platzgründen ausgelassen
End Function
End Class

Public Enum AdressenSortierenNach
    Matchcode
    Name
    Vorname
    Straße
    PLZ
    Ort
End Enum

```

Benutzeroberflächenschicht: frmMain.vb

Und schließlich gibt es die letzte Schicht der Anwendung, die, wie schon gesagt, nur noch Delegationsaufgaben erfüllt. Alle Funktionen, die das Formular durch entsprechende Ereignisse vom Benutzer entgegennimmt, leitet es direkt an die Klasse Adressen weiter, in der dann die eigentliche Problemlösung vollzogen wird.

Die Stellen, an denen es die Programmkontrolle zur eigentlichen Durchführung der »Geschäftslogik« an die Anwendungslogikschicht übergibt, sind im folgenden Listing fett markiert. Nicht relevante Codeauszüge sind aus Platzgründen ausgeblendet.

Auf einen funktionellen Unterschied zum Beispiel aus Kapitel 23 möchte ich in diesem Zusammenhang noch besonders hinweisen: Das Anklicken einer Adresszeile in der Liste bewirkt die Anzeige einiger Detaildaten in der Statuszeile des Formulars. Der entsprechende Code, der dieses Verhalten steuert, befindet sich ganz am Ende des dokumentierten Listings, das Sie im Folgenden finden.

```
Public Class frmMain

    Private myAdressen As Adressen          ' Alle Adressen

    'Dieses hier nicht als Ereignis einbinden - wäre von hinten, durch die Brust, ins Auge...
    'Besser überschreiben - frmListenForm ist schließlich von Form abgeleitet!
    Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
        'Basismethode aufrufen, damit das Formular machen kann,
        ''was es machen muss, um alles einzurichten.
        MyBase.OnLoad(e)

        'Instanziieren.
        myAdressen = New Adressen

        'Die Listview-Spalten einrichten.
        Vorbereitungen()

        'Die Adressenliste darstellen.
        ElementeDarstellen()
    End Sub

    'Richtet lediglich die Listview ein.
    Sub Vorbereitungen()
        'Aus Platzgründen ausgelassen
    End Sub

    Sub ElementeDarstellen()

        'Unterdrückt Neuzeichnen-Ereignisse bis zum
        'nächsten EndUpdate; dadurch geht der Aufbau
        ''der Elemente schneller und wackelt nicht.
        Me.lvAdressen.BeginUpdate()

        'Alle Elemente der ListView löschen.
        Me.lvAdressen.Items.Clear()

        'Für jedes Element der Liste wird ElementInListe aufgerufen.
        myAdressen.ForEach(New Action(Of Adresse)(AddressOf ElementInListe))
    End Sub
End Class
```

```

'So werden die Spaltenbreiten optimal angepasst.
For Each locCol As ColumnHeader In Me.lvAdressen.Columns
    locCol.Width = -2
Next

'Aufbau der ListView ist beendet.
Me.lvAdressen.EndUpdate()
End Sub

'Der Action-Delegat: für jedes Element der Liste wird diese Aktion durchgeführt.
Sub ElementInListe(ByVal Element As Adresse)
    'Neues ListView-Element - Matchcode kommt zuerst.
    Dim locLvwItem As New ListViewItem(Element.Matchcode)

    'Die Untereinträge setzen
    With locLvwItem.SubItems
        .Add(Element.Name)
        .Add(Element.Vorname)
        .Add(Element.Straße)
        .Add(Element.PLZ)
        .Add(Element.Ort)
    End With

    'Zum Wiederfinden Referenz in Tag
    locLvwItem.Tag = Element

    'Zur Listview hinzufügen
    lvAdressen.Items.Add(locLvwItem)
End Sub

'Wird aufgerufen, wenn eine der Spalten angeklickt wird.
Private Sub lvAdressen_ColumnClick(ByVal sender As Object, ByVal e As System.Windows.Forms.ColumnEventArgs) Handles lvAdressen.ColumnClick
    'Spaltennummer, die in e.Column steht, in AdressenSortierenNach konvertieren
    Dim locNach As AdressenSortierenNach = CType(e.Column, AdressenSortierenNach)

    'Sortieren
    myAdressen.Sortieren(locNach)

    'Die Elemente neu sortiert darstellen
    ElementeDarstellen()
End Sub

Private Sub tsmAdresseSuchen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmAdresseSuchen.Click
    Dim locSuchFormular As New frmSuchen
    Dim locErsterGefundener As Boolean

    'Den Suchbegriff merken, damit der Predicate-Delegat darauf
    'zugreifen kann.
    Dim locSuchbegriff As String = locSuchFormular.Suchbegriff
    If String.IsNullOrEmpty(locSuchbegriff) Then
        Return
    End If

```

```
'Alle gefundenen Elemente durchlaufen, und...
For Each locAdresse As Adresse In myAdressen.Suchen(locSuchbegriff)

    'jeweils alle ListView-Elemente durchsuchen und überprüfen, ob ...
    For Each locLvwItem As ListViewItem In Me.lvAdressen.Items

        '... die Tag-Referenz der Referenz des gesuchten Objekts entspricht.
        If locLvwItem.Tag Is locAdresse Then

            'Gefunden! ListView-Element markieren,
            locLvwItem.Selected = True

            'und dafür sorgen, dass der erste gefundene
            'Eintrag im sichtbaren Bereich liegt.
            If Not locErsterGefundener Then
                locLvwItem.EnsureVisible()
                locErsterGefundener = True
            End If

            'Gefunden, wir müssen in der ListView
            'nicht weitersuchen!
            Exit For
        End If
    Next
End Sub

Private Sub tsmAdressenLöschen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmAdressenLöschen.Click

    If lvwAdressen.SelectedItems IsNot Nothing AndAlso lvwAdressen.SelectedItems.Count > 0 Then

        Dim locDr As DialogResult = MessageBox.Show(My.Resources.MB_Nachfrage_Löschen_Body, _
            My.Resources.MB_Nachfrage_Löschen_Titel, MessageBoxButtons.YesNo, _
            MessageBoxIcon.Question)
        If locDr = Windows.Forms.DialogResult.Yes Then
            For Each lvwItem As ListViewItem In lvwAdressen.SelectedItems
                myAdressen.Remove(DirectCast(lvwItem.Tag, Adresse))
            Next
            ElementeDarstellen()
        End If
    End If
End Sub

Private Sub tsmZufallsadressenAnfügen_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles tsmZufallsadressenAnfügen.Click
    ' Aus Platzgründen ausgelassen
End Sub

'Wird aufgerufen, wenn Anwender Datei/Adressliste speichern wählt.
Private Sub tsmAdresslisteSpeichern_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles tsmAdresslisteSpeichern.Click

    Dim dateiSpeichernDialog As New SaveFileDialog
```

```

With dateiSpeichernDialog
    'Ermitteln des Dateinamens aus Platzgründen ausgelassen

    'Adressen serialisieren
    myAdressen.XMLSerialize(.FileName)
End With
End Sub

'Wird ausgelöst, wenn der Anwender einen Eintrag oder mehrere Einträge der Liste selektiert
Private Sub lvwAdressen_SelectedIndexChanged(ByVal sender As System.Object,
                                         ByVal e As System.EventArgs) Handles lvwAdressen.SelectedIndexChanged

    If lvwAdressen.SelectedItems.Count = 0 Then

        'Kein Eintrag selektiert
        tsslAusgewählteAdresse.Text = "Keine Adresse selektiert"
    ElseIf lvwAdressen.SelectedItems.Count > 1 Then

        'Mehrere Einträge selektiert
        tsslAusgewählteAdresse.Text = "Mehrere Adressen selektiert"
    Else

        'Genau ein Eintrag selektiert, dann die Grunddaten und das Geburtsdatum darstellen
        Dim locAdresse As Adresse

        'Das ursprüngliche Adresse-Objekt aus der Liste holen.
        'Dieses wurde beim Erstellen der ListViewItems in deren
        'Tag-Eigenschaften gespeichert, und auf diese Weise wird
        'die Relation zwischen Listeneintrag und eigentlichem Objekt hergestellt.
        locAdresse = DirectCast(lvwAdressen.SelectedItems(0).Tag, Adresse)

        'Text für die Darstellung in der Statuszeile zusammenbasteln:
        Dim locStatusText As String
        locStatusText = locAdresse.Matchcode & ": "
        locStatusText &= locAdresse.Name & ", " & locAdresse.Vorname & " - Geboren am "
        locStatusText &= locAdresse.Geburtsdatum.ToString("dd/MM/yyyy")

        'Text in der Statuszeile darstellen.
        tsslAusgewählteAdresse.Text = locStatusText
    End If
End Sub

End Class

```

Wie Sie sehen, ist das Komplizierteste, was diese Schicht noch übernehmen muss, die Darstellung der Daten im ListView-Steuerelement. Für den vorhandenen Funktionsumfang ist der reine Formularcode aber nunmehr vergleichsweise kompakt. Genaueres über die Darstellung von Auflistungsklassen im ListView-Steuerelement erfahren Sie im folgenden Kapitel.

Kapitel 37

Programmieren mit Windows Forms

In diesem Kapitel:

Formulare zur Abfrage von Daten verwenden – Aufruf von Formularen aus Formularen	1032
Überprüfung auf richtige Benutzereingaben in Formularen	1043
Anwendungen über die Tastatur bedienbar machen	1047
Über das »richtige« Schließen von Formularen	1051
Grundsätzliches zum Darstellen von Daten aus Auflistungsklassen in Steuerelementen	1053
Darstellen von Daten aus Auflistungen im ListView-Steuerelement	1054
Verwalten von Daten aus Auflistungen mit dem DataGridView-Steuerelement	1059
Entwickeln von MDI-Anwendungen	1081
Vererben von Formularen	1089

Formulare zur Abfrage von Daten verwenden – Aufruf von Formularen aus Formularen

Die seit Erscheinen von Visual Basic .NET wohl häufigste Frage zu Windows Forms-Anwendungen, der man in den verschiedensten Newsgroups begegnet, lautet: »Wie rufe ich aus einem Formular ein Formular auf – beispielsweise um Daten für die Hauptanwendung vom Anwender zu erfragen?«

Zweifelsohne gibt es auf diese Frage nicht *die eine* Antwort; es gibt natürlich mehrere Wege, Formulare zu realisieren, die zum Abfragen von bestimmten Parametern dienen und diese an das aufrufende Formular zurückliefern.

Die Technik, die ich Ihnen im Folgenden vorstellen möchte, ist in Absprache mit vielen Entwicklerkollegen aber wohl eine der gängigsten. Sie hat den Vorteil, dass sie ein Muster darstellt, das Sie in Anwendungen, die diese Technik des »Dateneinsammelns« erfordern, immer wieder einsetzen können.

Bei den Realisierungsüberlegungen und zum späteren besseren Verständnis des Codes, der diese Aufgabe übernimmt, sollte man sich die folgenden Punkte vor Augen führen:

Formulare, die zum Einlesen bestimmter Parameter dienen (im schon bekannten Beispielprojekt könnte das zum Beispiel das Erfassen einer neuen Adresse oder das Editieren einer vorhandenen sein), sollten ...

- ... als modale Dialoge dargestellt werden (siehe nächster Abschnitt).
- ... Daten, mit denen das Formular vorbelegt werden muss, vom aufrufenden Formular entgegen nehmen können, und die durch den Anwender veränderten Daten wieder zurückliefern.
- ... möglichst als Funktion aufgerufen werden können, und selbst dafür sorgen, dass sie sich darstellen und schließlich wieder schließen.
- ... vor ihrem Schließen die Richtigkeit der durch den Anwender eingegebenen Daten überprüfen.

Der Umgang mit modalen Formularen

Bevor wir uns mit dem Erstellen eines grundsätzlichen Musters für das Erfragen von Daten mit modalen Formularen beschäftigen, sollten wir uns für ein besseres Verständnis zunächst mit modalen Dialogen an sich beschäftigen. Es gibt hier nämlich Einiges zu entdecken, was nicht allgemeinhin bekannt ist.

Modale Dialoge verhalten sich wie Meldungsfelder (MessageBox-Dialoge), was bedeutet, dass Sie die darunter liegenden Dialoge nicht erreichen können, solange der modale Dialog aktiv ist. Ein Anwender muss also den Dialog erst ausfüllen und bestätigen (oder alternativ abbrechen, wenn er den Dialog versehentlich aufgerufen hat), um mit der Anwendung weiterarbeiten zu können. Im Gegensatz dazu gibt es übrigens nicht-modale Dialoge, die sich auf gleicher Ebene mit anderen Fenstern befinden, also parallel zu anderen Fenstern bedient werden können.

Modale Formulare haben eine eigene Nachrichtenwarteschlange

In Windows selbst funktioniert die Kommunikation zwischen verschiedenen Komponenten auf Basis so genannter Nachrichtenwarteschlangen (das nächste Kapitel hält dazu mehr bereit, falls Sie sich für die Details interessieren). Wenn Sie eine Windows Forms-Anwendung mit einem Hauptformular starten, dann richtet das Anwendungsframework für diese Anwendung genau eine dieser Nachrichtenwarteschlangen ein.

Sobald ein Ereignis eintritt, wird dieses Ereignis Windows-intern dieser Nachrichtenschlange hinzugefügt und letzten Endes von entsprechenden .NET-Klassen des Anwendungsframeworks bzw. des Hauptformulars der Anwendung bearbeitet.

Das ändert sich in dem Moment, in dem Sie einen modalen Dialog ins Leben rufen. In diesem Fall wird eine neue Nachrichtenwarteschlange für den modal darzustellenden Dialog eingerichtet. Der Programmablauf stoppt dann an der Stelle, an der Sie den entsprechenden modalen Dialog aufrufen:

```
Dim locFrm As New IrGeneeinFormular  
locFrm.ShowDialog()  
'Hier geht es erst weiter, wenn der Dialog beendet wurde.'
```

Steuern von modalen Dialogen mit der DialogResult-Eigenschaft

Da sich modale Dialoge wie Meldungsfelder verhalten (oder besser: es sind), kommt einer ihrer Eigenschaften eine besondere Funktion zu: der Eigenschaft DialogResult. Das Formular bestimmt durch Setzen dieser Eigenschaft, wie das »Ergebnis« der Formularbedienung aussah. Doch das Setzen dieser Eigenschaft im Formular bewirkt noch mehr: Sobald Sie diese Eigenschaft im Rahmen eines Ereignisses auf einen anderen Wert als None setzen, wird der Dialog beendet!

Das bedeutet: Verfügt Ihr Dialog, den Sie modal aufrufen möchten, beispielsweise über eine OK-Schaltfläche, dann genügt eine einzige Zeile Code, nicht nur um das Dialogergebnis zu setzen, sondern auch, um den Dialog zu schließen – etwa auf die folgende Weise:

```
'Wird aufgerufen beim Auslösen von OK-Schaltfläche.  
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles  
btnOK.Click  
  
'Das Zuordnen eines Wertes für DialogResult beendet den Dialog.  
Me.DialogResult = Windows.Forms.DialogResult.OK  
  
End Sub
```

HINWEIS Wichtig dabei ist es zu wissen, dass beim bloßen Setzen von DialogResult die Ressourcen des Dialoges nicht implizit wieder freigegeben werden. Wenn das Formular kritische Ressourcen blockiert, sollte es also in der Anwendung eine Instanz geben, die für das explizite Aufrufen von Dispose des Formulars sorgt. Eine gute Technik ist es, das Formular in einen Using...End Using-Block einzuschließen. Mehr zum Thema Dispose erfahren Sie in Kapitel 18. Der Umgang mit Using wird in Kapitel 10 beschrieben (Abschnitt: »Gezieltes Freigeben von Objekten mit Using«).

Ein Muster für modale Formulare implementieren, die sich selber verwalten können

Nachdem diese wichtigen Fakten bekannt sind, schreiten wir zur eigentlichen Aufgabe. Ziel ist es, ein Muster für einen Dialog zu schaffen, der, wie eingangs erwähnt, ...

- ... eine Funktion zur Verfügung stellt, die Parameter zur Bearbeitung entgegennimmt, den eigentlichen Dialog darstellt und die vom Anwender eingegebenen und geprüften Daten zurückliefert,
- ... OK und Abbrechen implementiert, mit denen der Dialog geschlossen wird,
- ... eine zentrale Routine anbietet, in der die Daten überprüft werden und
- ... dafür sorgt, dass die Ressourcen des Dialogs wieder freigegeben werden.

Beginnen wir mit dem ersten Teil, einer Funktion, die wir im Dialog als Member-Methode implementieren und aufrufen können, sobald uns eine Instanz des Dialogs vorliegt. Die Funktion innerhalb des Formular-codes sieht in etwa wie folgt aus:

```
Public Class frmFürDatenabfrage

    'Datenmember, auf die FormClosing zugreifen und die eine Funktion zurückliefern können
    Private myDatentyp As Datentyp

    'Member-Funktion des Formulars, die das Bearbeiten von Daten regelt.
    Public Function DatenBearbeiten(ByVal adr As Datentyp) As Datentyp
        'Damit wird dafür gesorgt, dass die Ressourcen für das Formular
        'nach Aufruf sofort wieder freigegeben werden!
        Using Me
            'TODO: Daten in die Maske kopieren.

            'Dialog modal darstellen. Das Programm "hält" an dieser
            'Stelle quasi an, und nur noch Ereignisse innerhalb
            'dieses Formulars werden verarbeitet.
            Me.ShowDialog()

            'Hier geht's erst weiter, wenn OK oder Abbrechen geklickt wurde.
            Return myDatentyp
        End Using
    End Function
    .
    .
    .
End Class
```

Diese Funktion deckt den ersten und den letzten Punkt unserer Todo-Liste bereits ab. Mit der Anweisung

```
Me.ShowDialog()
```

sorgt sie dafür, dass der Dialog als modaler Dialog dargestellt wird. Wenn die Funktion aufgerufen wird, nachdem eine Instanz des Formulars, das sie beherbergt, eingerichtet wurde, hält die Programmausführung an exakt dieser Stelle an und übergibt die Kontrolle der Nachrichtenwarteschlange des Formulars. Alle Ereignisse, die durch den Anwender bei der Bedienung des Dialogs auftreten können, werden von diesem Zeitpunkt an vom Formular verarbeitet, die entsprechenden Ereignisbehandlungsroutinen auch aufgerufen. Erst wenn der Dialog auf irgendeine Art und Weise wieder geschlossen wird, geht es an dieser Stelle hinter dem ShowDialog-Aufruf weiter, und das Formular kann schließlich die Daten, die die aufrufende Instanz angefordert hat (oder Nothing, wenn der Dialog abgebrochen wurde) zurückliefern.

Damit die Ressourcen des Formulars freigegeben werden, wenn die Funktion beendet ist, klammern wir den gesamten Funktionscode in Using mit Bezug auf das eigene Formular (Me) ein. Dieses Konstrukt sorgt dafür, dass jedes Objekt, das das Formular selbst als Member-Variable speichert, einerseits noch zurückgegeben werden kann, die Formularinstanz anschließend aber (und *erst dann*) noch »entsorgt« wird.

Implementieren müsste diese Formularklasse nun noch lediglich eine Methode, die die übergebenden Daten, die bearbeitet werden sollen, in die entsprechenden Steuerelemente des Formulars schreibt.

Die aufrufende Instanz würde diese Funktion nun wie folgt verwenden:

```
Dim locFrm As New frmFürDatenabfrage
Dim locDatentypZumBearbeiten As Datentyp = IrgendwasWasBearbeitetWerdenSoll
Dim locDatentypZurück As Datentyp = locFrm.DatenBearbeiten(locDatentypZumBearbeiten)
```

Auffällig ist die Objektvariable `myDatentyp`, die zu Beginn des Formulars als Klassen-Member deklariert wurde; sie global für das ganze Formular zu deklarieren, scheint auf den ersten Blick unnötig zu sein. Klarer wird ihr Gültigkeitsbereich, wenn wir den fehlenden Formularcode, der die Datenüberprüfung und das Schließen (bzw. Abbrechen) des Formulars regelt, im Kontext betrachten:

```
'Wird aufgerufen beim Auslösen von OK-Schaltfläche.
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
    'Das Zuordnen eines Wertes für DialogResult beendet den Dialog.
    Me.DialogResult = Windows.Forms.DialogResult.OK
End Sub
'Wird aufgerufen beim Auslösen von Abbrechen-Schaltfläche.
Private Sub btnAbbrechen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnAbbrechen.Click
    'Das Zuordnen eines Wertes für DialogResult beendet den Dialog.
    Me.DialogResult = Windows.Forms.DialogResult.Cancel
End Sub

'Wird aufgerufen, wenn das Formular geschlossen werden soll.
Protected Overrides Sub OnClosing(ByVal e As System.ComponentModel.CancelEventArgs)
    MyBase.OnClosing(e)
    'Überprüfung des Formulars nur, wenn OK geklickt wurde:
    If Me.DialogResult = Windows.Forms.DialogResult.OK Then
        myDatentyp = DatenDerMaskeÜberprüfenUndAlsObjektErmitteln
        'Wenn die ermittelten Daten Nothing und damit NICHT in Ordnung waren, ...
        If myDatentyp Is Nothing Then
            '... bleibt der Dialog, der ja eigentlich geschlossen werden soll,
            'offen - und das erreichen wir durch Setzen von
            e.Cancel = True
        End If
    Else
        'Diese Zeile kann nur erreicht werden, wenn Abbrechen
        'ausgelöst wurde. Dann wird auf jeden Fall Nothing als
        'Funktionsergebnis zurückgegeben.
        myDatentyp = Nothing
    End If
End Sub
```

Das Beenden des Dialogs wird eingeleitet, wenn eine der vorhandenen Schaltflächen *OK* oder *Abbrechen* des Formulars betätigt wird. Die jeweiligen Ereignisbehandlungs routinen machen dazu nichts weiter, als die `DialogResult`-Eigenschaft des Formulars zu ändern, und damit das Schließen des Formulars anzustoßen.

Das Schließen durch *OK* darf allerdings nur dann erfolgen, wenn die Daten, die der Anwender in der Maske zuvor eingegeben hat, auch validiert werden konnten, anderenfalls muss nicht nur eine entsprechende Fehlermeldung ausgegeben werden oder eine Kennzeichnung der falsch eingegebenen Datenfelder erfolgen – das Formular darf vor allen Dingen nicht geschlossen werden.

Das *Closing*-Ereignis des Formulars verfügt über die Fähigkeit, das Schließen des Formulars zu verhindern (wenn es nicht gerade mit *Form.Dispose* sozusagen mit Brachialgewalt abgeschossen wurde, doch das sollte eh die Ausnahme bleiben). Also ergibt es Sinn, die Prüfung auf Richtigkeit der Anwendereingaben in exakt diese Ereignisbehandlungsroutine zu verlegen.

Übrigens: Denken Sie daran, dass wir uns in der OOP-Programmierung befinden, und dass Formulare im Grunde genommen auch nichts anderes als Klassen sind. Es wäre also von hinten durch die Brust ins Auge, wenn wir innerhalb eines Formulars ein Formularereignis einbänden. Stattdessen ist es guter Programmierstil, die Methode zu überschreiben, die das Ereignis auslöst – damit bekommt Ihr Formularcode das Ereignis natürlich schon ein paar Schritte vorher zu Gesicht. Im Codeauszug selbst passiert exakt das mit *Protected Overrides Sub OnClosing*.

Der Code dieser Methode ist nun auch von entscheidender Bedeutung: Nur wenn das Schließen des Formulars mit *OK* eingeleitet wurde (*DialogResult* war *OK*), muss eine Überprüfung (und am besten auch noch die gleichzeitige Erstellung der Instanz einer Datentypklasse aus den Formulardatenfeldern) erfolgen. Im Ereignisbehandlungscode wird das durch die Zeilen

```
If Me.DialogResult = Windows.Forms.DialogResult.OK Then
    myDatentyp = DatenDerMaskeÜberprüfenUndAlsObjektErmitteln
```

erledigt. Und hier zeigt sich auch, wieso das Vorhalten einer Objektvariablen für die Datenrückgabe als Klassen-*Member* so wichtig ist: *OnClosing* muss in diese Objektvariable das Ergebnis mit den vom Anwender eingegebenen Daten ablegen, und damit der Ausgangscode der Formularaufruffunktion *DatenBearbeiten* diese Objektvariable als Funktionsergebnis zurückliefern kann, muss auch ihr der Zugriff auf die Objektvariable gestattet sein.

Hat die Datenüberprüfungsfunktion Fehler in der Anwendereingabe gefunden, dann sollte sie zu diesem Zeitpunkt schon entsprechende Hinweismeldungen im Formular gezeigt haben. Darüber hinaus gibt sie *OnClosing* in diesem Fall *Nothing* zurück, und *OnClosing* weiß anhand dessen, dass ein Fehler aufgetreten ist. Es verwendet nun folgendes Verfahren, um das Schließen des Formulars zu verhindern:

```
If myDatentyp Is Nothing Then
    '... bleibt der Dialog, der ja eigentlich geschlossen werden soll,
    'offen - und das erreichen wir durch Setzen von
    e.Cancel = True
End If
```

Durch Setzen der *Cancel*-Eigenschaft im Ereignisparameter von *OnClosing* lässt sich das Schließen des Formulars an dieser Stelle verhindern.

TIPP Falls Sie übrigens mehr zur Kommunikation zwischen Komponenten in .NET-Anwendungen in Form von Ereignissen erfahren wollen, werfen Sie einen Blick in Kapitel 20.

Implementierung des Musters in einer Anwendung

Theorie ist das eine, Praxis das andere. Wie sähe ein solcher Code nun aus, den wir in unsere schon bekannte Adresso-Anwendung implementieren wollten? Werfen wir dazu zunächst einen Blick auf die nächste Implementierungsstufe dieses Beispiels.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\ G - WinForms\\Kapitel 37\\Adresso02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Sie entdecken nach dem Programmstart in diesem Beispielprojekt zwei neue Funktionen im Menü *Bearbeiten*: *Neue Adresse eingeben* und *Adresse bearbeiten* (Siehe Abbildung 37.1). Die letzte Funktion lässt sich natürlich nur dann aufrufen, wenn Sie zuvor eine Adresse in der Liste selektiert haben. Ein und dasselbe Formular in der Anwendung deckt letztes Ende beide Funktionen ab.

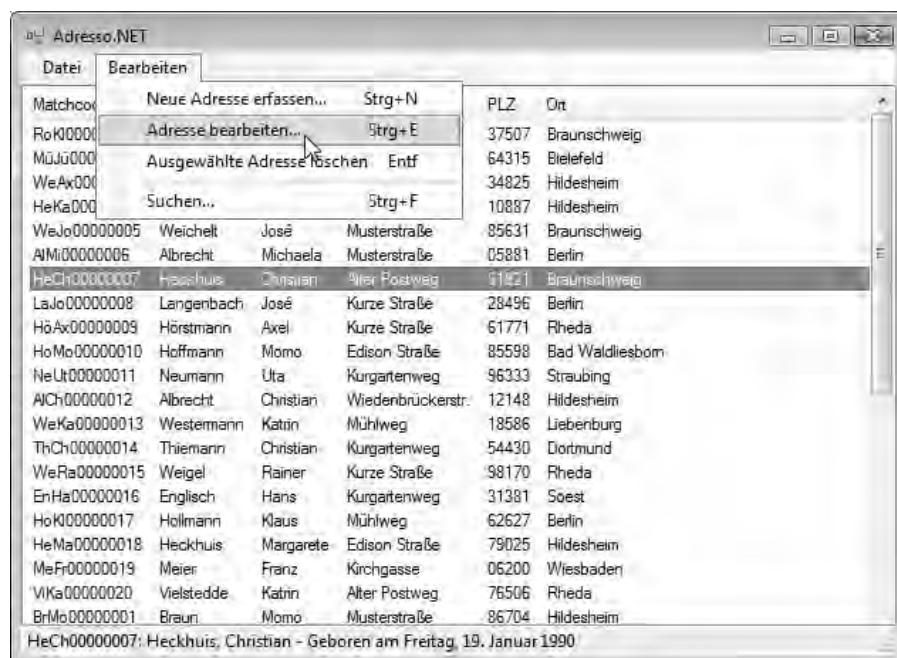


Abbildung 37.1 In dieser Ausbaustufe von Adresso können Sie die vorhandene Liste der Adressen bearbeiten und um neue ergänzen

Dennoch unterscheidet sich die Funktionsweise des Formulars bei beiden Funktionen marginal: Beim Bearbeiten einer vorhandenen Adresse lässt sich der Matchcode nicht ändern. Beim Neuanlegen muss er mit eingegeben werden. Der modale Dialog zum Ändern bzw. Erfassen einer Adresse sieht wie folgt aus:



Abbildung 37.2 Mit diesem modal dargestellten Dialog werden Adress-daten in Adresso bearbeitet oder neu angelegt

Damit das im letzten Kapitel postulierte Prinzip der mehrschichtigen Programmentwicklung gewahrt bleibt, befinden sich die Aufrufe des Formulars nicht im Code des Hauptformulars sondern im Logikteil der Anwendung.

Der Aufruf des Formulars zum Bearbeiten gestaltet sich wie folgt:

```

'<summary>
'<> Ruft den Dialog zum Bearbeiten einer Adresse auf.
'<> Liefert bei erfolgreicher Bearbeitung True zurück.
'</summary>
'<param name="adr">Adresse, die in dieser Instanz vorhanden sein muss
'<> und bearbeitet werden soll.</param>
'<returns></returns>
'<remarks></remarks>
Public Function AdresseBearbeiten(ByVal adr As Adresse) As Boolean
    'Zu suchende Adresse merken, damit der Find-Predicate den
    'richtigen Index ausfindig machen kann.
    myAktuelleAdresse = adr

    'Index ermitteln
    Dim locIndex As Integer = Me.FindIndex(New Predicate(Of Adresse)(AddressOf IndexFinder))

    'Adresse ist nur bei Index > -1 in Auflistung vorhanden.
    'Die Nummer dieser Adresse merken wir uns in locIndex.
    If locIndex > -1 Then

        'Adresse bearbeiten.
        Dim locFrm As New frmAdresseNeuUndBearbeiten
        Dim locAdresse As Adresse = locFrm.AdresseBearbeiten(adr)

        'Die Adresse wurde geändert...
        If locAdresse IsNot Nothing Then
            '...und kann gegen die ursprüngliche ausgetauscht werden.
            Me(locIndex) = locAdresse

            'True zurückliefern, damit die aufrufende Instanz weiß, dass
            'es die Liste aktualisieren muss, um die Änderungen widerzuspiegeln.
            Return True
        End If
    End If
    'Keine Änderung
    Return False
End Function

```

Der relevante Codeteil ist hier fett hervorgehoben. Sie werden sehen, dass das Muster zum Aufruf des Dialogs prinzipiell dem entspricht, wie wir es im vorherigen Abschnitt (ab Seite 1033) besprochen haben.

Übrigens: Um den Index des in der Liste markierten Objektes zu finden, bedient sich die Prozedur der `FindIndex`-Methode, die, angewendet auf die generische `List(Of)`-Auflistung, genauso wie deren `Find`-Methode zusammen mit einer `Predicate`-Instanz arbeitet. Genaueres zu diesem Thema finden Sie in Kapitel 23.

Ein wenig komplizierter ist das Neuanlegen einer Adresse, da sich hier ein besonderes Problem stellt: Wenn eine neue Adresse angelegt wird, darf der Matchcode, den der Anwender bestimmt, natürlich keinem schon vorhandenen Matchcode der Adressenliste entsprechen. Aus diesem Grund muss es eine Möglichkeit geben, dass der Dialog, in dem die Eingabe einer neuen Adresse erfolgt, die Adressen auf einem bereits vorhandenen gleichen Matchcode kontrolliert. Nun entspricht es aber nicht der Aufteilung einer Anwendung in Schichten, wenn ein Erfassungsdialog Zugriff auf die komplette Datenschicht nehmen darf. Abermals helfen uns Delegaten (besprochen in Kapitel 14) aus dem Dilemma. Anstatt dem Dialog eine Referenz auf die kompletten Adressendaten zu geben, mit denen er anschließend eine Kontrolle auf einen vorhandenen Matchcode in der Liste durchführen könnte, übergeben wir ihm einfach die *Adresse einer Funktion*, die das für ihn erledigt. Zu gegebener Zeit ruft er über den ihm übergebenden Delegaten einfach eine Prozedur in der Anwendungslogikschicht auf, die diese Überprüfung für ihn vornimmt – das Prinzip der strikten Schichttrennung bleibt damit gewahrt.

Der Code in der Anwendungslogik, der das Neuanlegen einer Adresse einleitet, sieht damit wie folgt aus:

```
''' <summary>
''' Delegat zum Überprüfen eines schon vorhandenen Matchcodes.
''' </summary>
''' <param name="Matchcode">Matchcode, der überprüft werden soll.</param>
''' <returns>True, wenn der Matchcode bereits existierte.</returns>
''' <remarks></remarks>
Public Delegate Function MatchcodeCheckenDelegate(ByVal Matchcode As String) As Boolean

''' <summary>
''' Ruft den Dialog zum Hinzufügen einer neuen Adresse auf.
''' Liefert bei erfolgreicher Eingabe True zurück.
''' </summary>
''' <returns>True, wenn die Adresse dieser Auflistung hinzugefügt wurde.</returns>
''' <remarks></remarks>
Public Function NeueAdresse() As Boolean
    'Formularinstanz erstellen
    Dim locFrm As New frmAdresseNeuUndBearbeiten

    'Neue Adresse ermitteln. Dazu die Routine als Delegat übergeben,
    'die überprüft, ob der Matchcode in dieser Auflistung bereits vorhanden ist.
    Dim locAdresse As Adresse = locFrm.NeueAdresse(AddressOf MatchcodeChecken)

    'Nur wenn eine gültige Adresse ermittelt wurde...
    If locAdresse IsNot Nothing Then

        '...diese dieser Auflistung hinzufügen.
        Me.Add(locAdresse)

        'True zurückliefern, damit die aufrufende Instanz weiß, dass
        'es die Liste aktualisieren muss, um die Änderungen widerzuspiegeln.
        Return True
    End If
End Function
```

```

End If
'Keine Änderung
Return False
End Function

Überprüft, ob ein vorhandener Matchcode bereits in der Auflistung vorhanden ist.
As String
Boolean

Public Function MatchcodeChecken(ByVal Matchcode As String) As Boolean
    For Each locItem As Adresse In Me
        If locItem.Matchcode = Matchcode Then
            Return True
        End If
    Next
    Return False
End Function

```

Delegat und korrelierende Funktion entsprechen der ersten und letzten fett hervorgehobenen Listingzeile im oben zu sehenden Codeausschnitt. Der »mittlere Teil« entspricht fast wörtlich dem im vorherigen Abschnitt besprochenen Muster.

Mit dieser Kapselung der Funktionalität beschränken sich die Änderungen, die an der Benutzeroberfläche in *frmMain.vb* nötig sind, auf die folgenden beiden Ereignisbehandlungsmethoden:

```

'Wird aufgerufen, wenn der Anwender im Menü Bearbeiten
'den Menüpunkt Neue Adresse erfassen anklickt.
Private Sub tsmNeueAdresseErfassen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles tsmNeueAdresseErfassen.Click

    'Nur aktualisieren, wenn das Neuanlegen erfolgreich war.
    If myAdressen.NeueAdresse() Then
        ElementeDarstellen()
    End If

    End Sub

    'Wird aufgerufen, wenn der Anwender im Menü Bearbeiten
    'den Menüpunkt Adresse bearbeiten anklickt.
    Private Sub tsmAdresseBearbeiten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
        Handles tsmAdresseBearbeiten.Click

        'Herausfinden, ob es überhaupt eine selektierte Adresse gibt
        If lvwAdressen.SelectedItems.Count > 0 Then

            'Erste selektierte Adresse wird bearbeitet
            Dim locAdresse As Adresse = DirectCast(lvwAdressen.SelectedItems(0).Tag, Adresse)

            'Nur aktualisieren, wenn das Bearbeiten erfolgreich war.
            If myAdressen.AdresseBearbeiten(locAdresse) Then
                ElementeDarstellen()
            End If
        End If
    End Sub

```

Nun ist die eigentliche Realisierungsweise des Formularcodes zum Erfassen einer Adresse natürlich von besonderem Interesse. Abgesehen von der Überprüfung der Formulardaten, auf die der nächste Abschnitt eingehen will, entspricht dieser Code ebenfalls zu 99% dem vorgestellten Muster:

```
Public Class frmAdresseNeuUndBearbeiten

    'Hier steht das Ergebnis der beiden Funktionen
    Private myAdresse As Adresse

    'Delegat, mit dessen Hilfe ein doppelter Matchcode überprüft wird.
    Private myMatchcodeCheckenProc As Adressen.MatchcodeCheckenDelegate

    ''' <summary>
    ''' Stellt diesen Dialog dar, und ermittelt das Objekt einer neuen Adresse.
    ''' </summary>
    ''' <param name="matchcodeChecken">Delegat, der auf eine Prozedur verweist, die doppelte
    ''' Matchcodes in der Auflistung überprüft.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Function NeueAdresse(ByVal matchcodeChecken As Adressen.MatchcodeCheckenDelegate) As Adresse

        'Damit wird dafür gesorgt, dass die Ressourcen für das Formular
        'nach Aufruf sofort wieder freigegeben werden!
        Using Me
            'Dialogtitel anpassen
            Me.Text = "Adresse bearbeiten"

            'Delegaten zuordnen - den brauchen wir später, um auf doppelte Matchcodes zu prüfen.
            myMatchcodeCheckenProc = matchcodeChecken

            'Dialog modal darstellen. Das Programm "hält" an dieser
            'Stelle quasi an, und nur noch Ereignisse innerhalb
            'dieses Formulars werden verarbeitet.
            Me.ShowDialog()

            'Hier geht's erst weiter, wenn OK oder Abbrechen geklickt wurde.
            Return myAdresse
        End Using
    End Function

    ''' <summary>
    ''' Stellt diesen Dialog dar, lässt den Anwender eine Adresse bearbeiten und liefert
    ''' ein neues Adresse-Objekt zurück, das die Änderungen widerspiegelt.
    ''' </summary>
    ''' <param name="adr">Adresse-Objekt, das bearbeitet wird.
    ''' ACHTUNG! Das Rückgabeobjekt stellt eine neuen Instanz dar!</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Function AdresseBearbeiten(ByVal adr As Adresse) As Adresse

        'Damit wird dafür gesorgt, dass die Ressourcen für das Formular
        'nach Aufruf sofort wieder freigegeben werden!
        Using Me
            'Dialogtitel anpassen
            Me.Text = "Adresse bearbeiten"
```

```
'Daten in die Maske kopieren.  
DatenInMaske(adr)  
  
'Matchcode darf nicht editiert werden:  
txtMatchcode.ReadOnly = True  
  
'Dialog modal darstellen. Das Programm "hält" an dieser  
'Stelle quasi an, und nur noch Ereignisse innerhalb  
'dieses Formulars werden verarbeitet.  
Me.ShowDialog()  
  
'Hier geht's erst weiter, wenn OK oder Abbrechen geklickt wurde.  
Return myAdresse  
End Using  
  
End Function  
  
'Wird aufgerufen beim Auslösen von OK-Schaltfläche.  
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles  
btnOK.Click  
  
'Das Zuordnen eines Wertes für DialogResult beendet den Dialog.  
Me.DialogResult = Windows.Forms.DialogResult.OK  
  
End Sub  
  
'Wird aufgerufen beim Auslösen von Abbrechen-Schaltfläche.  
Private Sub btnAbbrechen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _  
Handles btnAbbrechen.Click  
  
'Das Zuordnen eines Wertes für DialogResult beendet den Dialog.  
Me.DialogResult = Windows.Forms.DialogResult.Cancel  
  
End Sub  
  
'Wird aufgerufen, wenn das Formular geschlossen werden soll.  
Protected Overrides Sub OnClosing(ByVal e As System.ComponentModel.CancelEventArgs)  
 MyBase.OnClosing(e)  
'Überprüfung des Formulars nur, wenn OK geklickt wurde:  
If Me.DialogResult = Windows.Forms.DialogResult.OK Then  
  
'Daten aus Maske gibt ein Adresse-Objekt nur dann zurück,  
'wenn die Daten in Ordnung waren.  
myAdresse = DatenAusMaske()  
  
'Wenn myAdresse also Nothing und damit NICHT in Ordnung war, ...  
If myAdresse Is Nothing Then  
  
'... bleibt der Dialog, der ja eigentlich geschlossen werden soll,  
'offen - und das erreichen wir durch Setzen von  
e.Cancel = True  
End If  
Else  
'Diese Zeile kann nur erreicht werden, wenn Abbrechen  
'ausgelöst wurde. Dann wird auf jeden Fall Nothing als
```

```
'Funktionsergebnis zurückgegeben.  
myAdresse = Nothing  
End If  
End Sub  
  
'Überprüft die Eingaben im Formular, und liefert  
'im Erfolgsfall ein fix-und-fertiges Adresse-Objekt  
'aus den Eingabefeldern zurück.  
Private Function DatenAusMaske() As Adresse  
    'Wird im nächsten Abschnitt beschrieben – daher hier ausgelassen.  
End Function  
  
'Kopiert ein vorhandenes Adresse-Objekt in die Maske  
'für das weitere Bearbeiten.  
Private Sub DatenInMaske(ByVal adr As Adresse)  
    txtMatchcode.Text = adr.Matchcode  
    txtVorname.Text = adr.Vorname  
    txtNachname.Text = adr.Name  
    txtStraße.Text = adr.Straße  
    mtbPlz.Text = adr.PLZ  
    txtOrt.Text = adr.Ort  
    mtbGeburtsdatum.Text = adr.Geburtsdatum.ToShortDateString  
    lblWochentag.Text = adr.Geburtsdatum.ToString("dddd")  
End Sub  
  
Private Sub mtbGeburtsdatum_LostFocus(ByVal sender As Object, ByVal e As System.EventArgs) _  
    Handles mtbGeburtsdatum.LostFocus  
    Dim locGebDatum As Date  
    If Date.TryParse(mtbGeburtsdatum.Text, locGebDatum) Then  
        lblWochentag.Text = locGebDatum.ToString("dddd")  
    End If  
End Sub  
End Class
```

Überprüfung auf richtige Benutzereingaben in Formularen

Die Qualität Ihres Arbeitsalltags fällt und wächst mit dem Aufwand, den Sie zu Zeiten der Softwareentwicklung in das Thema Eingabeprüfungen stecken. Wenn Sie professionell Softwareentwicklung betreiben, dann können Sie mir glauben, dass Sie, wenn Sie sich morgens an Ihren Entwicklungsrechner setzen, Sie kein Fax mit Inhalt dessen auf dem Schreibtisch liegen haben möchten, was Sie auch in Abbildung 37.3 sehen können.

Gerade bei der Datenerfassung, also wenn es um die Schnittstelle zwischen Ihrer Software und dem Anwender geht, können bei der Entwicklung viele Fehler passieren. Und Sie sollten nach Möglichkeit extrem kreativ werden, um Vorahnungen zu entwickeln, welche Kombinationen unglücklicher Umstände dazu führen können, dass Ihre Software jahrelang funktioniert und plötzlich nicht mehr.

Anekdoten aus der Praxis

So wurde ich als Berater zur Analyse eines Phänomens einer Software bestellt – einer Erfassung von Mitarbeiterzeiten in Produktionsbetrieben – die jahrelang anstandslos ihren Dienst verrichtet hatte. Die Frage an den Kunden, ob in den vergangenen Tagen etwas Ungewöhnliches passiert sei, beantwortete er mit einem klaren Nein.

Nach einigem Recherchieren stellte sich allerdings heraus, dass doch etwas Außergewöhnliches vorgefallen war – wenn auch nicht im Betrieb vor Ort: Der Schwesterbetrieb dieses Berliner Unternehmens, das seinen Sitz in Magdeburg hatte, war nämlich tragischerweise direkt vom Elbe-Hochwasser 2002 betroffen.

Der Konzern bewies aber gute Führungsqualitäten in Form von schneller Reaktion und gutem Improvisationstalent, und so verlegte man kurzerhand die Produktion der wichtigsten Produkte von Magdeburg nach Berlin; die Berliner Mitarbeiter und der Betriebsrat erklärten sich sofort bereit, eine noch nie da gewesene 3. Schicht zu fahren, um den betroffenen Magdeburgern den Vortritt zu lassen.

Damit trat eine Konstellation ein, die der Software das erste Mal abverlangte, datumsübergreifende Buchungen durchführen zu müssen, und da das Tagesdatum nicht bei der Erfassung von Start- und Endzeit sondern nur als Buchungsdatum berücksichtigt wurde, sah es aus Sicht der Software so aus, als ob die Anfangszeit (20:00, 24.8.2002) zeitlich *hinter* der Endzeit (2:30, aber 25.8.2002) lag.

Die Software quittierte das mit einem Absturz bei den anschließenden Auswertungen; alle zuvor ermittelten Zeiten konnten aber glücklicherweise mit angepasstem Filter abermals aus den Zeiterfassungsterminals übernommen werden.

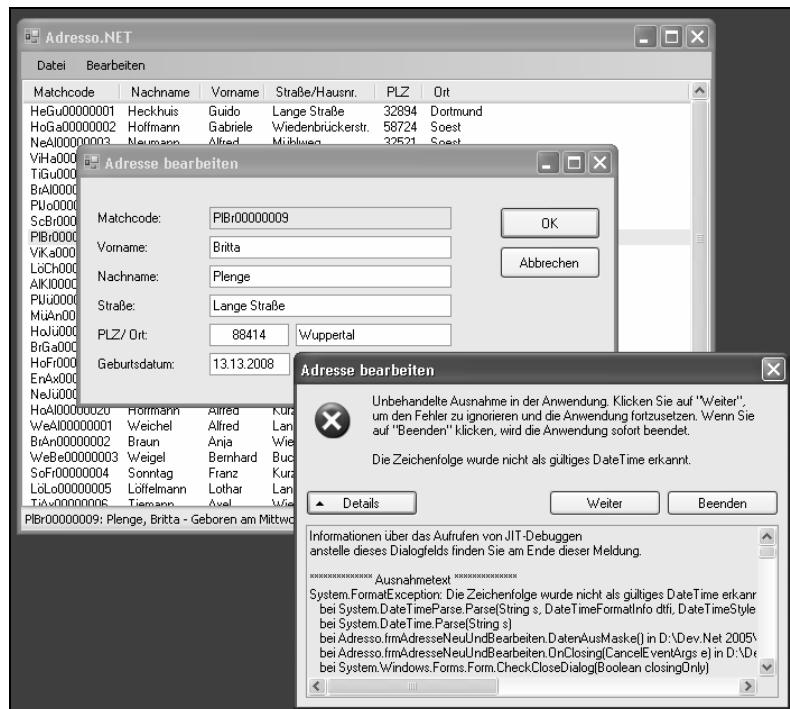


Abbildung 37.3 Wenn ein Anwender Ihrer Software auf seinem Arbeitsrechner einen solchen »Programmzustand« provozieren kann, liegt der Fehler leider in Ihrer Software ...

Es spielt keine Rolle, woher Ihre Software Ihre Daten bezieht – ob nun von Maschinen oder von Menschen. Sie sollten nicht zuletzt im eigenen Interesse (ruhiger schlafen, bessere Reputation) sehr großen Wert darauf legen, das größtmögliche Augenmerk auf das Sichern von Datenschnittstellen gegen Fahrlässigkeit und groben Unfug zu legen.

Was die manuelle Eingabe von Daten anbelangt, bietet das .NET Framework eine Komponente an, die dem Anwender Falscheingaben auf eine – meiner Meinung nach – recht eigenwillige aber dennoch nicht minder plausible Weise vor Augen führt.

Abbildung 37.4 zeigt dieses Verfahren im Einsatz: Durch die ErrorProvider-Komponente lassen sich beim Auftreten von Fehlern in Eingabefeldern von Formularen diese betroffenen Felder mit einem roten Ausrufungszeichen markieren. Tritt der Fehler auf, blinken diese Ausrufungszeichen sogar für eine bestimmte (einstellbare) Weile.



Abbildung 37.4 Durch die ErrorProvider-Komponente können Sie Anwender auf Fehler in Eingabefeldern gezielt aufmerksam machen

Fährt der Anwender anschließend mit dem Mauszeiger auf eines der Ausrufungszeichen, zeigt der ErrorProvider eine zuvor definierbare Fehlermeldung als Tooltip an.



Abbildung 37.5 Eine Komponente (ein zur Entwurfzeit nicht sichtbares Steuerelement) ziehen Sie bei Bedarf ebenfalls ins Formular; der Designer legt es dann im Komponentenfach ab

Um ein Formular mit einer ErrorProvider-Komponente auszustatten, brauchen Sie sie lediglich aus der Toolbox ins Formular zu ziehen. Der Visual Basic-Designer legt dann – so dies die erste Komponente war – ein Komponentenfach unterhalb des Formulars an und legt die Komponente dort ab.

Der ErrorProvider verfügt nämlich über keine eigene Benutzeroberfläche. Er erweitert einfach die Steuerelemente, die sich auf dem Container (dem Formular in diesem Fall) befinden, um die entsprechende Funktionalität.

Zum Anzeigen eines Eingabefehlers neben einem Steuerelement des Formulars genügt anschließend ein einziger Methodenaufruf, etwa in dieser Form:

```
ErrorProvider.SetError(steuerelementName, "Fehlermeldungstext für den Tooltip!")
```

Wenn Sie möchten, dass alle Fehlermarkierungen wieder aufgehoben werden, verwenden Sie die folgende Methode:

```
ErrorProvider.Clear()
```

Das Adresso-Beispiel verwendet diese Vorgehensweise, und der entsprechende Code, der von OnFormClosing (siehe Beispiele der vorherigen Abschnitte) aus aufgerufen wird, sieht folgendermaßen aus:

```
'Überprüft die Eingaben im Formular, und liefert
'im Erfolgsfall ein fix-und-fertiges Adresse-Objekt
'aus den Eingabefeldern zurück.
Private Function DatenAusMaske() As Adresse

    Dim locFehler As Boolean

    'Alle möglichen vorherigen Fehler zurücksetzen
    ErrProv.Clear()

    'Wenn keine Eingabe im Feld gemacht wurde,
    If String.IsNullOrEmpty(txtMatchcode.Text) Then

        'Fehlermeldung setzen.
        ErrProv.SetError(txtMatchcode, "Fehlende Eingabe!")
        locFehler = locFehler Or True

    Else

        'Der Delegat ist nur beim Neuanlegen einer Adresse vorhanden,
        'deswegen auf Vorhandensein überprüfen!
        If myMatchcodeCheckenProc IsNot Nothing Then
            'Feststellen, ob der Matchcode in der Auflistung schon existiert!
            'Das passiert in unserem Fall über einen Delegaten.
            If myMatchcodeCheckenProc.Invoke(txtMatchcode.Text) Then
                ErrProv.SetError(txtMatchcode, "Dieser Matchcode ist schon vorhanden!")
                locFehler = locFehler Or True
            End If

        End If
    End If
End If
'Ähnlich bei allen anderen vorgehen.
```

```
If String.IsNullOrEmpty(txtVorname.Text) Then
    ErrProv.SetError(txtVorname, "Fehlende Eingabe!")
    locFehler = locFehler Or True
End If

If String.IsNullOrEmpty(txtNachname.Text) Then
    ErrProv.SetError(txtNachname, "Fehlende Eingabe!")
    locFehler = locFehler Or True
End If

If String.IsNullOrEmpty(txtStraße.Text) Then
    ErrProv.SetError(txtStraße, "Fehlende Eingabe!")
    locFehler = locFehler Or True
End If

Dim locGebDate As Date
If Not Date.TryParse(mtbGeburtsdatum.Text, locGebDate) Then
    ErrProv.SetError(mtbGeburtsdatum, "Falsches Datumsformat!")
    locFehler = locFehler Or True
End If

If String.IsNullOrEmpty(txtOrt.Text) Then
    ErrProv.SetError(txtOrt, "Fehlende Eingabe!")
    locFehler = locFehler Or True
End If

'Fehler war vorhanden - Nothing zurückliefern
If locFehler Then Return Nothing

'Alles war OK, es gibt eine neue Adresse.
Return New Adresse(txtMatchcode.Text, txtNachname.Text,
    txtVorname.Text, txtStraße.Text,
    mtbPlz.Text, txtOrt.Text, locGebDate)
End Function
```

Anwendungen über die Tastatur bedienbar machen

Dieses Buch wäre schätzungsweise einen Monat später fertig geworden, könnte man die Funktionen von Microsoft Word ausschließlich über die Maus bedienen. Wenn ich selbst den typografischen Satz eines Buches vornehme, kann ich die Batterien aus meiner Funkmaus herausnehmen – ich brauche sie oft über mehrere Stunden gar nicht. Zwar ist Windows eine grafische Benutzeroberfläche, doch ist es vielen Softwareentwicklern gar nicht bewusst, wie wichtig es für einen Anwender ist, der täglich mit einer Software arbeiten muss, diese nur ausschließlich über die Maus bedienen zu können. Es kostet wahnsinnig viel Zeit, von der Tastatur, die man natürlich für Texteingaben benötigt, zur Maus umzugreifen, weil man nur so an eine bestimmte Programmfunction herankommen kann.

Dabei bedarf es nicht viel, um Dialoge oder Menüs auch über die Tastatur bedienbar zu machen. Es gibt unter Windows definierbare Schnellzugriffstasten, deren Implementierung in die eigene Software im Handumdrehen erledigt ist.

Definition von Schnellzugriffstasten per »&«-Zeichen

Beim Erstellen von Menüs genügt ein Voranstellen des »&«-Zeichens, um dieses Menü mit dem dahinter stehenden Buchstaben über die Tastatur (in Verbindung mit [Alt]) aktivieren zu können (siehe Abbildung 37.6).

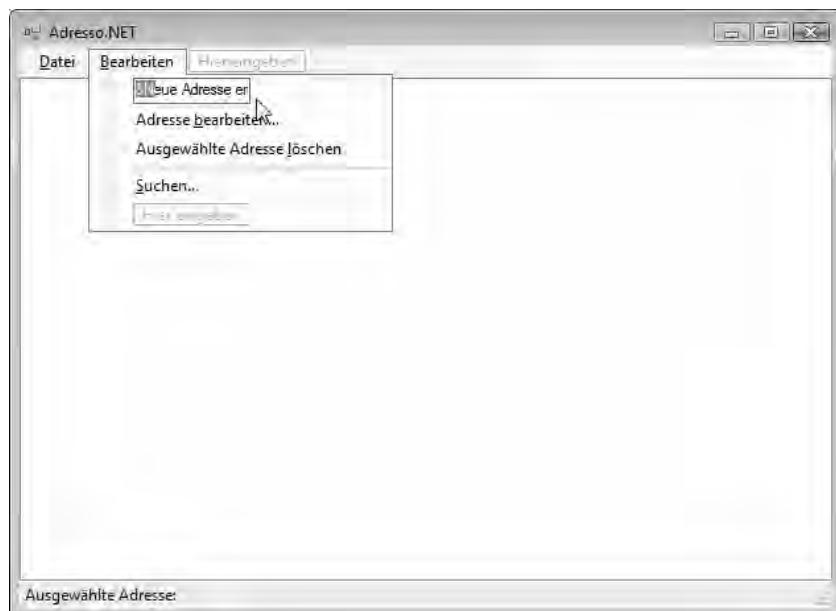


Abbildung 37.6 Mit dem Kaufmannsund (»&«) definieren Sie Schnellzugriffstasten in Menüs ...

Für jeden Menüeintrag können Sie im Bedarfsfall zudem eine Funktionstaste definieren, mit der Sie den Menübefehl (und damit die dahinter stehende Funktion) direkt auslösen können. Dazu klicken Sie einfach auf den entsprechenden Menüeintrag und stellen im Eigenschaftenfenster seine `ShortcutKeys`-Eigenschaft ein. Standardmäßig blendet dann der Menüeintrag neben dem eigentlichen Menütext einen Hinweis auf diese Tastenkombination für den Anwender ein. Sollte Ihnen diese Tastenkombinationsbeschreibung nicht passen, können Sie sie mit dem `ShortcutKeyDisplayString` nach Gutdünken anpassen.¹ Und falls Sie es überhaupt nicht wünschen, dass die Taste oder Tastenkombination neben dem Menütext angezeigt wird, setzen Sie die `ShowShortcutKeys`-Eigenschaft des entsprechenden Eintrags auf `False`.

¹ Und in diesem Zusammenhang sei mir die bissige Bemerkung erlaubt: »Strg« steht auf deutschen Tastaturen nicht für »String«, nicht für »Strong«, nicht für »Struck« oder sonst einen Blödsinn. Es ist die vielleicht nicht ganz gelungene Abkürzung von »S t e u e - r u n g«. Und falls Sie wissen, welche Geschichte sich hinter der Abkürzung »S-Abfr« auf älteren Tastaturen verbirgt (unterhalb der Druck-Taste) – für eine kurze E-Mail wäre ich dankbar ... ;-)

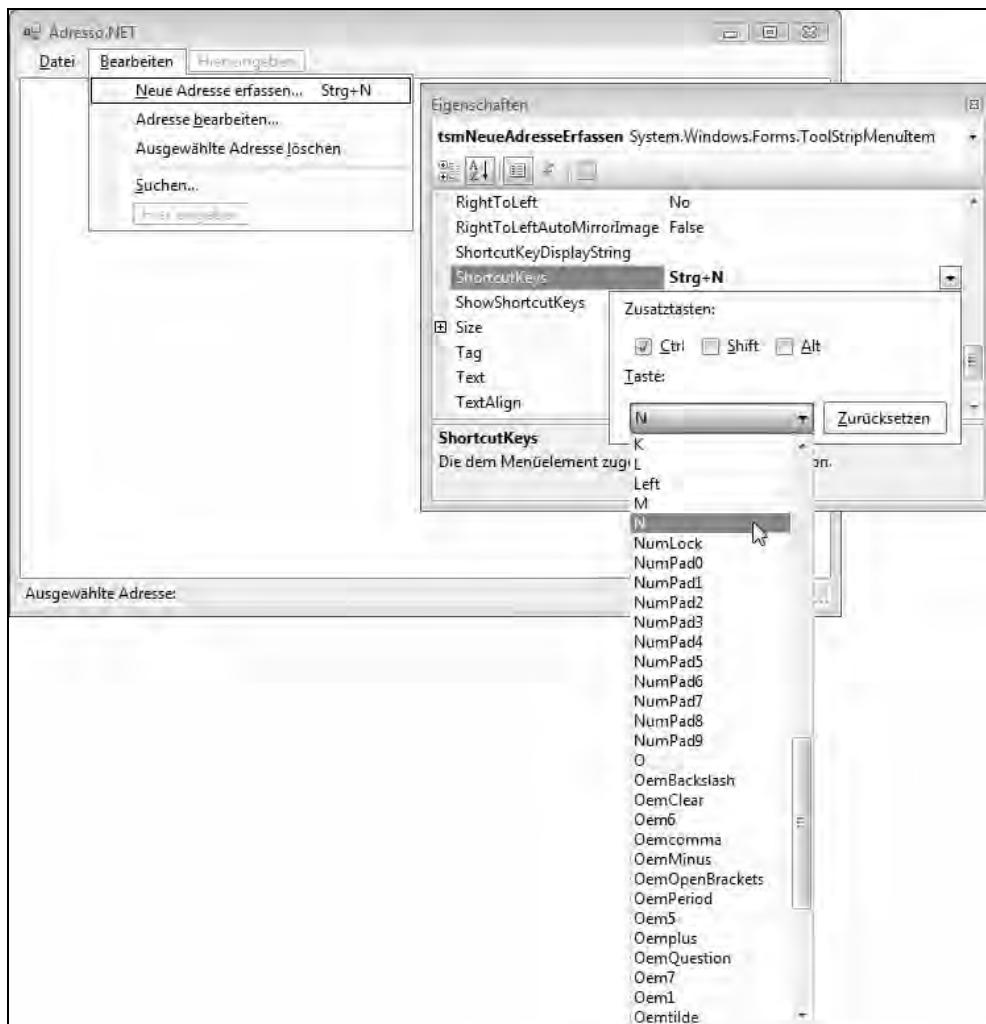


Abbildung 37.7 Neben den Schnellzugriffstasten lassen sich für jeden Menübefehl auch Funktionstasten (»Abkürzungstasten«) einrichten – einfach und schnell über das Eigenschaftenfenster ...

Bei Textfeldern in Formularen beispielsweise kommen zwei Faktoren ins Spiel, um das Eingabefeld per Tastatur zur fokussieren: Ein beschreibendes Label-Steuerelement, das sich vor dem eigentlich zu aktivierenden Steuerelement befindet, und eine richtig eingestellte Aktivierungsreihenfolge.



Abbildung 37.8 ... und mithilfe davor stehender Beschriftungen (Label) auch für Steuerelemente, die über keine selbst beschreibende Beschriftung verfügen – wie beispielsweise TextBox-Steuerelemente

Da das Label vor dem Eingabesteuerelement (z.B. TextBox) nur eine beschreibende Funktion hat, selbst aber nicht aktiviert werden kann, wird beim Auslösen der Zugriffstaste das in der Aktivierungsreihenfolge hinter dem Label-Steuerelement stehende Steuerelement fokussiert.

Die Aktivierungsreihenfolge können Sie übrigens einstellen, indem Sie das Formular durch Anklicken im Designer selektieren, und anschließend aus dem Menü *Ansicht* den Menüpunkt *Aktivierungsreihenfolge* auswählen. Der Visual Studio-Designer schaltet anschließend in einen besonderen Bearbeitungsmodus, der das Bestimmen der Aktivierungsreihenfolge (englisch: *Tab Ordner* für *Tabulatorreihenfolge*) durch simples Nacheinander-Anklicken der Steuerelemente auf dem Formular erlaubt. Eine geschickte Auswahl der Aktivierungsreihenfolge ermöglicht, wie in Abbildung 37.9 zu sehen, dabei auch Konstellationen, in denen eine Beschriftung für zwei Eingabefelder gilt.

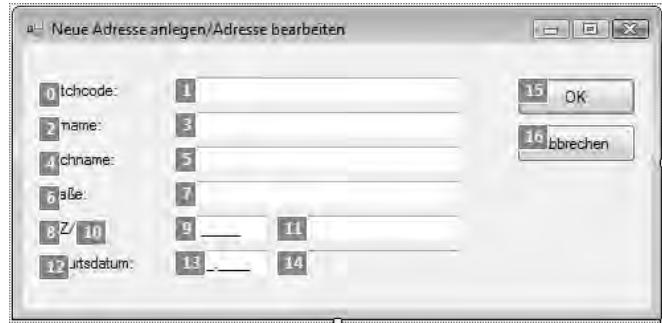


Abbildung 37.9 Mit der Funktion Aktiviererienfolge bestimmen Sie die Zugriffsreihenfolge für das Fokussieren von Steuerelementen per **[Tab]**-Taste durch simples nacheinander Anklicken

Accept- und Cancel-Schaltflächen in Formularen

Viele modale Dialoge lassen sich mit den Tasten **[Esc]** und **[←]** beenden.

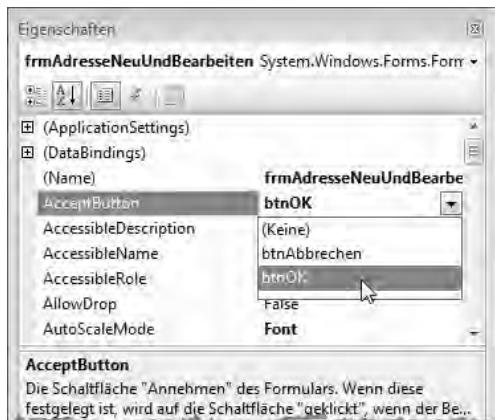


Abbildung 37.10 Die Einstellung der Schaltflächen für **[Esc]** und **[←]** nehmen Sie über **AcceptButton** und **CancelButton** am Formular vor

Für .NET-Formulare richten Sie diese Tasten für Schaltflächen (und ausschließlich für Schaltflächen) über die Formulareigenschaften **AcceptButton** (für Schaltflächen, die mit **[←]** auszulösen sind) und **CancelButton** (für Schaltflächen, die mit **[Esc]** auszulösen sind) ein.

HINWEIS Dies ist eine für VB6-Entwickler unübliche Vorgehensweise, da sich diese Art der Formularsteuerung in VB6 über Eigenschaften der Schaltflächen und nicht über die des Formulars selbst einstellen ließen. Achten Sie also darauf, die beschriebenen Eigenschaften am Formular und nicht an den Schaltflächen im Eigenschaftenfenster zu suchen!

Im Normalfall entspricht **AcceptButton** der *OK*-Schaltfläche, **CancelButton** der *Abbrechen*-Schaltfläche eines Formulars. Sinn ergibt das Setzen dieser Eigenschaften übrigens nur in modal darzustellenden Formularen.

Über das »richtige« Schließen von Formularen

Es mag auf den ersten Blick nicht eines ganzen Abschnittes wert sein; besorgte Nachfragen im Usenet machen mich aber glauben, dass ein paar Sätze über das richtige Schließen eines Formulars nicht schaden können.

Um ein Formular programmtechnisch zu schließen, gibt es vier Möglichkeiten:

- `formInstance.Hide`
- `formInstance.Close`
- `formInstance.Dispose`
- `formInstance.DialogResult = [einWert]>>DialogResult.None` (nur in modalen Dialogen – lesen Sie dazu bitte den Abschnitt »Der Umgang mit modalen Formularen« ab Seite 1032).

Unsichtbarmachen eines Formulars mit Hide

Die `Hide`-Methode macht nichts weiter, als die `Visible`-Eigenschaft eines Formulars auf `False` zu setzen. Damit gibt es die eigentliche Instanz eines Formulars zwar noch, das Formular befindet sich nur nicht mehr sichtbar auf dem Bildschirm.

Allerdings gibt es etwas zu beachten, wenn Sie Formulare modal darstellen:

Das nämlich bedeutet für das Anwenden der `Hide`-Methode (oder auch für das Setzen der `Visible`-Eigenschaft des Formulars auf `False`): Wenn die Darstellung des Formulars zuvor modal erfolgte, wird in diesem Moment die Warteschlange beendet und die Kontrolle an die aufrufende Instanz zurückgegeben, die Ressourcen des Formulars werden aber nicht sofort freigegeben.

Schließen des Formulars mit Close

Das Schließen des Formulars mit `Close`, »emuliert« sozusagen das Schließen des Formulars durch den Anwender. Dabei hat die das Formular einbindende Instanz (oder das Formular selbst) die Möglichkeit, den Vorgang des Schließens zu verhindern.

Entweder durch Überschreiben von `OnClosing` (soweit es das Formular selbst betrifft) oder durch Einbinden des `Closing`-Ereignisses (soweit es die das Formular einbindende Instanz betrifft) besteht die Möglichkeit, durch die `CancelEventArgs` des Ereignisses den kompletten Schließen-Vorgang abzubrechen:

```
Private Sub frmMain_Closing(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs)
    Handles MyBase.Closing
    'Formular soll nicht geschlossen werden
    e.Cancel = True
End Sub
```

Allerdings: Wenn nicht programmtechnisch interveniert wurde, das Formular zu schließen, ist es nicht nur unsichtbar geworden, sondern wird durch den Garbage Collector bei der nächsten Gelegenheit entsorgt. Die `Close`-Methode entspricht also quasi einem intervenierbaren `Dispose` (siehe nächster Abschnitt).

Was passiert bei `Form.Close` intern:

Für die Puristen unter Ihnen hier eine Ereignisabfolge, die beschreibt, in welcher Reihenfolge Dinge beim Schließen eines Formulars mit `Close` passieren (lesen Sie im Bedarfsfalls zunächst die entsprechenden Abschnitte im nachfolgenden Kapitel, um mehr Grundsätzliches über die interne Verwaltung und Verarbeitung von Nachrichten bei Windows-Anwendungen zu erfahren).

- `Close` wird aufgerufen, das Formular sendet Windows-intern die Nachricht `WM_Close` mit seinem eigenen Window Handle an die Nachrichtenwarteschlange.
- Wenn die Nachrichtenschlange verarbeitet wird, schickt diese die Nachricht weiter an `WndProc` des Formulars.
- `WndProc` ruft die Formular-interne Methode `WMCclose` auf.
- `WMCclose` ruft die `Dispose`-Methode des Formulars auf, nachdem `OnClosing` und `OnClose` aufgerufen wurden, die ihrerseits die entsprechenden Ereignisse ausgelöst haben. `Dispose` wird aber nur dann aufgerufen, wenn `OnClosing` den Schließvorgang nicht abgebrochen hat.

TIPP Wenn sich Ihr Formular selber schließen soll, es dies aber in einer Methode macht, die einen Rückgabewert an die aufrufende Instanz zurückliefern muss, der aus einem Member der Formularklasse hervorgeht, schließen Sie es mit Close, und wenden Sie dann Return an. Nur so ist gewährleistet, dass es den zurückzugebenden Member noch gibt, wenn Return ausgeführt wird. Dispose würde das Formular sofort zur Entsorgung freigeben; Sie könnten nicht sicher sein, dass es den zurückzugebenden Wert beim Return noch gibt.

Da diese Anforderungen in der Regel nur für modale Dialoge bestehen, verwenden Sie noch besser das Programmiermuster, über das der Abschnitt »Der Umgang mit modalen Formularen« ab Seite 1032 genau Auskunft gibt.

Entsorgen des Formulars mit Dispose

Dispose macht mit einem Formular, was es auch mit jedem anderen Objekt macht: es zur Entsorgung durch den Garbage Collector freigeben. Das bedeutet: Dispose lässt ein Formular unaufhaltsam und für immer verschwinden. Das Closing-Ereignis wird *nicht* aufgerufen, und weder die das Formular einbindende Instanz noch das Formular selbst haben die Möglichkeit, etwas dagegen zu tun. Das gilt natürlich auch für alle Member des Formulars. Lesen Sie deswegen bitte auch die Ausführungen des vorherigen Abschnittes.

Grundsätzliches zum Darstellen von Daten aus Auflistungsklassen in Steuerelementen

Fast alle denkbaren Anwendungen verlangen es, dass Datenauflistungen in irgendeiner Form visualisiert werden, und das geschieht in der Regel in Form von Listen. Das ListView-Steuerelement eignet sich dazu nur bedingt, da es im Prinzip nur eine Eigenschaft eines Datenelements darstellen kann. Wenn es darum geht, umfangreiche Mengen von Daten in SmartClient-Anwendungen übersichtlich darzustellen, bieten sich dazu zwei Steuerelemente an:

- Das ListView-Steuerelement.
- Das DataGridView-Steuerelement.

Das ListView-Steuerelement eignet sich hervorragend zur reinen Textdarstellung von tabellarischen Daten, wenn Sie seine detaillierte Listendarstellung verwenden. Sie kennen es aus dem Windows-Explorer und der Dateianzeige: Es ist ein unglaublich mächtiges Steuerelement, das die verschiedensten Darstellungsformen kennt. So können Sie sich auch im Windows-Explorer für eine Miniaturansicht von Daten, für die gekachelte Darstellung, die symbolische Darstellung, die einfache Listendarstellung und in die detaillierte Listendarstellung entscheiden. In den anschließenden Beispielen werden wir uns auf diesen letzten Darstellungsstil beschränken.

Das DataGridView-Steuerelement ist ein komplett neues Steuerelement im .NET Framework 2.0 und dann interessant, wenn nicht nur reine Textdaten, sondern komplexere Datenvisualisierungen in verschiedenen Spalten umgesetzt werden müssen. Im Gegensatz zum ListView-Steuerelement erlaubt es auch das Bearbeiten von Daten direkt in Tabellenform. Dafür ist der Programmieraufwand zur Verwaltung der Daten im DataGridView-Steuerelement auch größer, um Daten abzubilden und dem Anwender die Möglichkeit zu geben, diese zu bearbeiten.

Die folgenden Abschnitte beschreiben die grundsätzliche Vorgehensweise beim Abbilden von Auflistungsklassen in diesen Steuerelementen.

Darstellen von Daten aus Auflistungen im ListView-Steuerelement

Wenn Sie sich für die tabellarische Darstellung der Daten von Auflistungsklassen entscheiden, ist die detaillierte Listendarstellung des ListView-Steuerelements genau das Richtige. Um eine Instanz des ListView-Steuerelements für diese Form der Darstellung vorzubereiten, verwenden Sie die folgenden Eigenschaften (lvwAdressen sei dabei im Folgenden der Name eines ListView-Steuerelements im Formular):

lvwAdressen.View = Details	'Einstellung des Steuerelements auf detaillierte Listendarstellung
lvwAdressen.FullRowSelect = True	'Selektierung der gesamten Zeile, nicht nur des ersten Elements
lvwAdressen.HideSelection = False	'Beim Verlieren des Fokus bleibt die Selektierung erhalten
lvwAdressen.GridLines = True	'Zwischen den Zeilen werden kleine Abtrennungslinien eingezeichnet

Diese Einstellungen können Sie natürlich auch mit dem Eigenschaftenfenster zur Entwurfzeit einstellen, so wie es in den vergangenen und zukünftigen Beispielen dieses Kapitels auch der Fall ist.

Spaltenköpfe

Die Details-Ansicht des ListView-Steuerelements zeichnet sich dadurch aus, dass Daten in mehreren Spalten pro Zeile angeordnet werden können. Die Spalten selbst werden durch die ColumnHeaderCollection-Auflistung² des Steuerelements eingerichtet und gesteuert. Um dem Steuerelement verschiedene Spalten hinzuzufügen, verfahren Sie beispielsweise wie folgt:

```
With Me.lvwAdressen
    With .Columns
        'Alle Spaltenköpfe löschen.
        .Clear()
        'Spaltenüberschriften einrichten.
        .Add("Spalte1", -2, HorizontalAlignment.Left)      'Linksbündige Darstellung in der Spalte
        .Add("Spalte2", -2, HorizontalAlignment.Right)     'Rechtsbündige Darstellung
        .Add("Spalte3", -2, HorizontalAlignment.Center)   'Zentrierte Darstellung
        .Add("Spalte4", -2)                                'Ohne Angabe: Standard ist linksbündig
    End With
End With
```

Mit der Columns-Eigenschaft erhalten Sie Zugriff auf die ColumnHeaderCollection-Auflistung, durch die Sie dann die einzelnen Spaltenköpfe wie bei einer »normalen« Auflistung mit der Add-Methode hinzufügen können. Die einfachste Methode, das zu erreichen, wird hier demonstriert: Als ersten Parameter bestimmen Sie den Text eines Spaltenkopfes, als zweiten seine Breite. Wenn Sie möchten, dass die Spaltenbreite automatisch an die Breite des Textes angepasst wird, übergeben Sie als Spaltenbreite den Wert -2. Wichtig dabei ist zu wissen, dass Sie die Spaltenbreiten abermals auf -2 setzen müssen (obwohl sich ja eigentlich dieser Wert

² Kleine Anmerkung am Rande: Die Benennung einer Auflistung mit dem endenden Wortteil »Collection« widerspricht übrigens Microsofts Richtlinien zur Namensvergabe von Auflistungen – Microsoft verstößt bei der Namensgebung von ColumnHeaderCollection also gegen seine eigenen Richtlinien. Am besten fahren Sie mit Begriffen, bei denen Sie die Datenklasse im Singular benennen (Adresse), eine entsprechende Auflistung, die diese Daten als Menge verwaltet, im Plural (Adressen).

bereits in der Eigenschaft befindet), nachdem Sie die Liste mit Daten gefüllt haben. Dann wird die Layout-Logik abermals für jede Spalte aufgerufen; dieses Mal werden die Breiten der Spalten dann jedoch aufgrund der Breite der Spaltenköpfe *und* aufgrund der Breite der einzelnen Listeneinträge neu berechnet und visuell angepasst.

Schließlich können Sie einen dritten Parameter übergeben, der die Ausrichtung der Texte in den Spalten bestimmt.

HINWEIS Beachten Sie, dass die erste Spalte Texte grundsätzlich nur linksbündig darstellen kann, egal, welchen Parameter Sie hier angeben. Wenn Sie keinen Parameter übergeben, wird die linksbündige Darstellung der Texte in der entsprechenden Spalte angenommen.

Hinzufügen von Daten

Jede Zeile eines ListView-Steuerelements wird durch eine ListViewItem-Instanz repräsentiert. Ein ListViewItem-Objekt enthält also die Texte jeder Spalte einer Zeile, die im ListView-Steuerelement dargestellt werden sollen. Den Text, der in der ersten Spalte dargestellt wird, bestimmt die Text-Eigenschaft eines ListViewItem-Objekts.

Diesen Text können Sie direkt beim Instanziieren einer Instanz dieses Objektes angeben. Die Texte aller weiteren Spalten einer Zeile werden durch die SubItems-Elemente des ListViewItem-Objekts festgelegt. Da es sich bei SubItems wieder um eine Auflistung handelt, können Sie die Texte der einzelnen Spalten ebenfalls mit deren Add-Methode einrichten, etwa wie im Folgenden zu sehen:

```
Dim locLvwItem As New ListViewItem("Text, Spalte 1")

'Die Untereinträge setzen
With locLvwItem.SubItems
    .Add("Text, Spalte 2")
    .Add("Text, Spalte 3")
    .Add("Text, Spalte 4")
End With
```

Wenn Sie eine ListViewItem-Instanz auf diese Weise aufbereitet haben, stellen Sie die komplette Zeile im ListView-Steuerelement dar, indem Sie diese Instanz der Items-Auflistung des ListView-Steuerelements hinzufügen, etwa mit:

```
'Zur Listview hinzufügen
lvwAdressen.Items.Add(locLvwItem)
```

Beschleunigen des Hinzufügens von Elementen

Das Hinzufügen der Elemente auf die so beschriebene Weise erfolgt sichtbar, und das heißt: Sobald Sie ein Element hinzugefügt haben, wird der komplette Inhalt des ListView-Steuerelements neu gezeichnet. Bei hunderten von Elementen kann das sehr lange dauern – unnötigerweise.

Sie können mit zwei Methoden aufrufen – BeginUpdate sowie EndUpdate – festlegen, dass das Neuzeichnen der Elemente für die Dauer ihres Hinzufügens zur Liste ausgesetzt wird, und das funktioniert folgendermaßen:

```
'Unterdrückt Neuzeichnen-Ereignisse bis zum nächsten EndUpdate;
'dadurch geht der Aufbau der Elemente schneller und "wackelt" nicht.
Me.lvAdressen.BeginUpdate()
'Hier fügen Sie alle Elemente der Liste hinzu:

'Aufbau der ListView ist beendet.
Me.lvAdressen.EndUpdate()
```

Automatisches Anpassen der Spaltenbreiten nach dem Hinzufügen aller Elemente

Nachdem alle Elemente der Liste hinzugefügt wurden, können Sie mit einem kleinen Trick dafür sorgen, dass sich die Spaltenbreiten automatisch an die Breiten der Texte bzw. der Spaltenköpfe anpassen (die längsten Texte in den Köpfen und Datenzeilen einer jeden Spalte werden als Maß verwendet). Dazu setzen Sie die Breiten aller Spalten einfach komplett nochmals auf den Wert -2. In Form von Code sieht das so aus:

```
'So werden die Spaltenbreiten optimal angepasst.
For Each locCol As ColumnHeader In Me.lvAdressen.Columns
    locCol.Width = -2
Next
```

Sie sollten diese Aktion auch zwischen BeginUpdate und EndUpdate durchführen, damit ein Neuzeichnen des gesamten Steuerelements erst anschließend stattfindet.

Zuordnen von ListViewItems und Objekten, die sie repräsentieren

Anders als bei der einfachen ListBox, der Sie eine komplette Objektreferenz als Listeneintrag selbst übergeben können, deren Textdarstellung anschließend über die ToString-Funktion des Objektes geregelt wird, und bei der eine selektierte Zeile auch gleichzeitig einem selektierten Objekt entspricht, gibt es beim ListView-Steuerelement keine automatische Zuordnung zwischen einem Eintrag in der Liste und einem Objekt, das diese Zeile visuell darstellen soll. Die Frage lautet also: Wenn der Anwender eine Zeile angeklickt hat, wie kann ich dann herausfinden, welchem Objekt sie ursprünglich entsprach?

Um diesem Problem zu entgehen, verfügt jedes ListViewItem-Objekt, das eine Zeile der ListView darstellt, über eine Tag-Eigenschaft (sprich: »Tähg«, etwa: Kennzeichnung, Markierung). Diese Eigenschaft kann beliebige Objekte, sprich: »alles« aufnehmen. Der Trick besteht also darin, aus einem Objekt Texte für die Darstellung über eine ListView-Instanz zu generieren und die Objektreferenz zusätzlich noch in der Tag-Eigenschaft eines ListViewItem zu speichern.

Wenn Sie später eine bestimmte Zeile der ListView als ListViewItem über die Items-Auflistung abrufen, steht Ihnen so auch das Ursprungsobjekt, aus dem die Texte für die ListViewItem-Instanz entstanden sind, wieder zur Verfügung.

Eine Zuordnung eines beliebigen Objekts sieht in etwa wie folgt aus:

```
'Zum Wiederfinden: Referenz in Tag speichern
locLvwItem.Tag = locElement
```

Wenn Sie später wieder an das Ausgangsobjekt herankommen wollen, brauchen Sie es nur aus der Tag-Eigenschaft wieder auszulesen (und im Bedarfsfall in den Ursprungstyp zurückzukasten). Der folgende Codeschnipsel macht genau das (Voraussetzung dafür ist natürlich, dass mindestens ein Element in der ListView selektiert wurde):

```
'Das ursprüngliche Adresse-Objekt aus der Liste holen.  
locElement = DirectCast(lvwAdressen.SelectedItems(0).Tag, ElementType)
```

Übrigens: Auch bei komplexen Objekten bedeutet diese Vorgehensweise natürlich kein Aasen mit Arbeitsspeicher, weil, wie Sie im Klassenteil dieses Buches erfahren konnten, natürlich nur eine *Referenz* auf das eigentliche Objekt gespeichert wird. Da es die Tag-Eigenschaft sowieso gibt (und der entsprechende Speicher für eine Referenz sowieso reserviert wird, auch wenn sie standardmäßig auf Nothing »zeigt«), kostet das Speichern jedes Objektes in der Tag-Eigenschaft jedes ListViewItem der Liste kein einziges Byte an zusätzlichen Speicher.

Feststellen, dass ein Element der Liste selektiert wurde

Wenn Sie eine ListView verwenden, wird der Anwender irgendwann einmal ein Element dieser Liste auswählen, um es beispielsweise bearbeiten zu können. In diesem Fall muss Ihre Anwendung natürlich nicht nur wissen, welches Element angeklickt wurde, sondern unter Umständen auch, wann dieses Ereignis auftrat, um zum Beispiel eine bestimmte Statusinformation (etwa in einer Statuszeile des Fensters) zu aktualisieren.

Mit dem SelectedIndexChanged-Ereignis informiert Sie ein ListView-Steuerelement, dass sich die Elementselektierungen in der Liste geändert haben. Das Ereignis wird immer dann ausgelöst, wenn ...

- ... ein Element selektiert wurde und zuvor keines selektiert war
- ... kein Element mehr selektiert ist, vorher aber mindestens ein Element selektiert war
- ... ein weiteres Element selektiert wurde, wenn das ListView-Steuerelement zuvor mithilfe der MultiSelect-Eigenschaft so eingestellt wurde (True), dass mehrere Elemente in der Liste selektiert werden konnten.

HINWEIS Wenn Sie sehr zeitintensive Benutzeroberflächenaktualisierungen durchführen müssen, sobald sich die Selektion in der Liste ändert, beachten Sie dabei, dass dieses Ereignis beim Wechsel einer Elementselektierung zweimal ausgelöst wird: Einmal für die Deselektierung des ersten Elements und ein anderes Mal für die Selektierung des neuen Elements.

Mit dem ItemSelectionChanged-Ereignis (neu im Framework 2.0) werden Sie darüber hinaus informiert, wenn sich der Auswahlzustand eines Elementes ändert.

Die Feststellung, welche Elemente im ListView-Steuerelement selektiert sind, kann mithilfe zweier Eigenschaften erfolgen:

- Die SelectedItems-Eigenschaft enthält eine Auflistung aller Elemente, die zurzeit im ListView-Steuerelement selektiert sind.
- Die SelectedIndices-Eigenschaft enthält eine Auflistung aller Indizes der Elemente, die zurzeit im ListView-Steuerelement selektiert sind.

Der folgende Code implementiert eine Routine zum Löschen aller selektierten Elemente im ListView-Steuerelement aus einer korrelierenden Auflistung, die die Ursprungsobjekte zur Darstellung im ListView-Steuerelement bereithält.

```
If lvwAdressen.SelectedItems IsNot Nothing AndAlso lvwAdressen.SelectedItems.Count > 0 Then
    For Each lvwItem As ListViewItem In lvwAdressen.SelectedItems
        myAdressen.Remove(DirectCast(lvwItem.Tag, Adresse))
    Next
    'Hier wird die Liste neu aufgebaut:
    ElementeDarstellen()
End If
End If
```

Selektieren von Elementen in einem ListView-Steuerelement

Wichtig für Anwendungen ist es nicht nur, herausfinden zu können, welche Elemente eines ListView-Steuerelements selektiert sind. Hier und da müssen auch Elemente programmtechnisch selektiert werden.

HINWEIS Hierfür können Sie die `SelectedItems`-Eigenschaft *nicht* verwenden, da sie nur die selektierten Elemente eines ListView-Steuerelements *widerspiegelt*. Wenn Elemente eines ListView-Steuerelements selektiert werden sollen, erreichen Sie das ausschließlich über die `Selected`-Eigenschaft eines ListViewItem-Elements der Liste.

Der Code, um Elemente eines ListView-Steuerelements zu selektieren, in deren Tag-Eigenschaft sich Objekte einer korrelierenden Auflistung befinden, lautet folgendermaßen:

```
'Alle zu selektierenden Elemente durchlaufen, und...
For Each locAdresse As Adresse In zuSelektierendeAdressenObjekte

    'jeweils alle ListView-Elemente durchsuchen und überprüfen, ob ...
    For Each locLvwItem As ListViewItem In Me.lvwAdressen.Items

        '... die Tag-Referenz der Referenz des gesuchten Objekts entspricht.
        If locLvwItem.Tag Is locAdresse Then

            'Gefunden! ListView-Element markieren,
            locLvwItem.Selected = True

            'und wir müssen in der ListView
            'nicht weitersuchen!
            Exit For
        End If
    Next
Next
```

TIPP Wenn Sie möchten, dass ein bestimmtes Element nicht nur selektiert wird, sondern es sich darüber hinaus auch auf jeden Fall im sichtbaren Bereich des ListView-Steuerelements befinden soll, verwenden Sie die `EnsureVisible`-Methode des entsprechenden ListViewItem-Objekts.

Verwalten von Daten aus Auflistungen mit dem DataGridView-Steuerelement

Wenn es um die reine Darstellung von Daten einer Auflistung geht, verrichtet das ListView-Steuerelement sicherlich gute Dienste, zumal es bei der Anzeige auch vieler Daten eine gute Performance an den Tag legt.

Flexibler für die Darstellung von Daten, wenn auch nicht so performant, ist ein Steuerelement, das neu mit dem Framework 2.0 eingeführt wurde: das DataGridView-Steuerelement. Der Name dieses Steuerelements ist vielleicht ein wenig irreführend, weil es durch die Silbe »View« (etwa: *Ansicht, Darstellung, Ausblick*) im Namen die Vorstellung erweckt, es könne Daten lediglich *anzeigen*. Doch das ist überhaupt nicht der Fall.

Ganz im Gegenteil: Mit dem DataGridView-Steuerelement legt Ihnen Microsoft eine visuelle Komponente in die Hände, die an Mächtigkeit in Sachen Datendarstellung und Datenerfassung eigentlich nicht mehr zu übertreffen ist.

Doch es ist wie bei allen Dingen des Lebens: Wo viel Licht ist, ist auch viel Schatten. Um bei dieser Analogie zu bleiben: Das DataGridView-Steuerelement ist ein 10.000-Watt-Suchscheinwerfer; deswegen gibt es auch eine Menge Falltüren, die es clever zu umschiffen gilt, wenn Sie für den Komfort des Anwenders das Letzte aus diesem Steuerelement herausholen wollen.

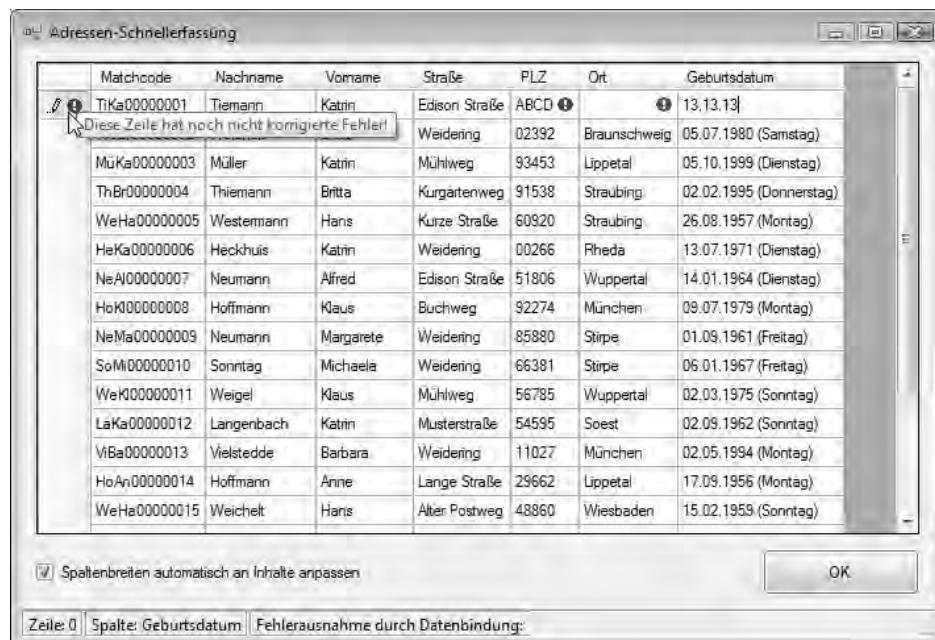


Abbildung 37.11 So komfortabel sollte eine Datenerfassungsanwendung mindestens sein, die Sie den Anwendern Ihrer Software über das DataGridView-Steuerelement zur Verfügung stellen

Um es vorweg zu sagen: Dem DataGridView-Steuerelement könnte man locker nicht nur ein eigenes Kapitel, sondern fast schon ein kleines Buch widmen, und wenn ich persönlich auch dieses Thema extrem spannend finde, so muss ich mich aus Platzgründen dennoch auf Wesentliches zu diesem Thema beschränken. Neben

den grundlegendsten Dingen, die Sie für den Start benötigen, habe ich deswegen versucht, möglichst viele der Themen herauszuarbeiten, die Sie nicht ohnehin aus der Online-Hilfe erfahren könnten.

Und das impliziert meinen ersten Vorschlag: Die Lektüre der Online-Hilfe zum Thema `DataGridView` enthält wirklich umfangreiches und gar nicht so trocken geschriebenes Material. Wenn Sie sich viel mit Datenbindung und Datenaufbereitung in Ihren Anwendungen beschäftigen müssen, sollten Sie sich wirklich einen halben Tag Zeit nehmen und die Online-Hilfe von Visual Basic zu diesem Thema gründlich studieren.

Um Daten in einem `DataGridView`-Steuerelement anzeigen oder bearbeiten zu können, stehen Ihnen folgende Vorgehensweisen zur Verfügung:

- Sie sorgen komplett per Programmcode für das Einrichten der verschiedenen Spalten, für die Datentypen, die in den Zellen der einzelnen Spalten verarbeitet werden, für das Anzeigen der Daten in den Zellen und für die Aufbereitung und spätere Übernahme in den Datenklassen ihrer Anwendung.
- Sie binden das `DataGridView`-Steuerelement an eine Datenquelle, sorgen sich nicht um Tabelleneinrichtung, Datendarstellung und Datenüberprüfung und programmieren gar nicht.
- Sie finden den goldenen Mittelweg, verwenden die Datenbindung, aber greifen bei entscheidenden Prozessen per Programmcode in das Geschehen ein. Die folgenden Abschnitte und darin enthaltenen Beispiele gehen diesen Mittelweg, da er den meiner Meinung nach besten Kosten-/Nutzen-Faktor darstellt.

Über Datenbindung, Bindungsquellen, die `BindingSource`-Komponente und warum `Currency` nicht unbedingt Währung heißt

Ganz simpel ausgedrückt: Bei der Datenbindung von Framework-Komponenten wird die Eigenschaft einer konsumierenden Datenquelle mit der Eigenschaft einer Daten vorgebenden Komponente verknüpft. Ändert sich der Wert der Eigenschaft der letzteren, ändert sich gleichzeitig der Wert der ersten – im Bedarfsfall auch umgekehrt.

Bei größeren Bindungsszenarien, die nicht nur auf Eigenschaften (also Datenfeldebene) beschränkt sind, bedeutet das beispielsweise für ein `DataGridView`-Steuerelement: Man kann auch eine ganze `DataGridView`-Instanz an eine Datenquelle binden. Die Datenquelle liefert Datenlisten, die `DataGridView` zeigt sie an, erlaubt das Bearbeiten, und die Änderungen laufen im Bedarfsfall wieder zurück in die Datenquelle. Eine Datenquelle kann dabei eine Komponente einer Datenbank sein. Beispielsweise eine Access-Datentabelle. Oder besser: Eine Tabelle oder ein Resultset³ einer SQL Server-Datenbank. Oder, und das ist das Tolle, auch etwas, was man im Microsoft-Jargon als *Business Object* (Geschäftsobjekt) bezeichnet, und dabei handelt es sich in der Regel um eine Auflistungsklasse, wie Sie sie beispielsweise in unserer ständig wachsenden Adresso-Anwendung kennen gelernt haben. Nicht nur einzelne Eigenschaften werden also dabei gebunden, sondern mehrere Objekte, die ihrerseits über mehrere Eigenschaften verfügen. Eine Tabellenzeile stellt also genau ein Objekt einer Auflistung dar (oder, im Falle von Datenbanken, einen Datensatz). Eine Tabellenzelle entspricht einer Eigenschaft eines Objektes der Auflistung (oder dem Inhalt eines Datenfeldes eines Datensatzes im Fall von Datenbanken). Die ganze `DataGridView`-Tabelle entspricht der gesamten Auflistung (oder – bei Datenbanken – einem kompletten Resultset). Beim Binden von Auflistungen spricht man übrigens nur von Datenbindungen, beim Binden von Datenbanktabellen von *komplexen* Datenbindungen.

³ ... was soviel wie *Abfrageergebnissatz* bedeutet und einer wie auch immer gearteten Datentabelle entspricht.

Doch egal, welches Objekt an welches andere Objekt gebunden wird, sie benötigen immer eine vermittelnde Instanz. Im Fall von simplen Eigenschaftsverknüpfungen übernimmt diese Aufgabe das so genannte `PropertyManager`-Objekt; im Falle von Listen das `CurrencyManager`-Objekt.

HINWEIS Lassen Sie sich dabei nicht von dem englischen Begriff *Currency* ins Bockshorn jagen, für den Sie vielleicht nur die deutsche Übersetzung *Währung* parat haben. *Currency* bedeutet nämlich auch *Zeitnähe* (von *current*, etwa: *aktuell*); ein `CurrencyManager` ist also eine überwachende Instanz, die dafür sorgt, dass Dinge *zeitnah* passieren – im .NET Framework die zeitnahe Umsetzung der Datenbindung.

Bis zur Version 1.1 verlief das Binden einer Komponente an eine Datenquelle, die mehrere Listenelemente anbot, dubios und irgendwie im Hintergrund, ohne dass man so recht wusste, was genau passiert. *Was* wirklich passierte, war: Beim Zuweisen einer Datenquelle an eine Komponente – beispielsweise für eine Listendarstellung dieser Elemente – wurde implizit ein `CurrencyManager`-Objekt erstellt. Dieses `CurrencyManager`-Objekt sorgte dann dafür, dass – sehr vereinfacht ausgedrückt – beim Editieren einer Zeile auch das korrelierende Objekt in der Auflistung »getroffen« wurde.

Im .NET 2.0-Framework ist dieser Vorgang nicht nur offensichtlicher geworden – er wurde auch um Designer-Unterstützung ergänzt: Sie können mithilfe des Designers eine Datenquelle erstellen, oder besser: dem Designer mitteilen, welches Objekt (Datenbank, Auflistungsklasse) eine Datenquelle *sein soll*. Und dann können Sie interaktiv diese Datenquelle an eine Komponente wie beispielsweise eine `DataGridView`-Instanz binden. Der Designer erstellt dabei eine so genannte `BindingSource`-Komponente (auf deutsch etwa *Bindungsquelle*, eine Art *CurrencyManager XXL*), und legt diese Komponente im Komponentenfach ab.

Diese `BindingSource`-Komponente ist also im .NET Framework 2.0 das Bindeglied zwischen der die Daten anbietenden und der die Daten konsumierenden Komponente. Damit das reibungslos funktioniert, stellt sie (ganz ähnlich wie früher `CurrencyManager`) Methoden wie `MoveNext` (*zum Nächsten bewegen*), `MovePrevious` (*zum Vorherigen bewegen*), `AddNew` (*Neuen anlegen*) und Ähnliches bereit. Und wozu? Ganz einfach:

Wenn Sie später beim Bearbeiten von Daten in Ihrer `DataGridView`-Instanz mit dem Cursor eine Zeile nach unten fahren, dann ruft die `DataGridView` die `MoveNext`-Methode der `BindingSource` auf. Die `BindingSource` sorgt dann dafür, dass ein interner Zeiger auf die einzelnen Elemente der Datenquelle ebenfalls weitergerückt wird. Die `DataGridView` kann sich also das nächste (*next*) Objekt aus der Datenquelle holen – im Falle einer Auflistung eben das nächste Objekt in der Liste.

Fahren Sie mit dem Cursor in der Tabelle eine Zeile nach oben, ruft die `DataGridView`-Instanz `MovePrevious` der `BindingSource` auf, und sie bekommt, um beim Beispiel zu bleiben, wieder über den Vermittler `BindingSource`, das vorherige (*previous*) Objekt der Auflistung.

Dieser Fall geht auch umgekehrt. Wenn Sie eine Objektauflistung als Datenquelle über eine `BindingSource`-Instanz an eine `DataGridView`-Instanz gebunden haben, können Sie auch selber `MoveNext` der `BindingSource` aufrufen, um einerseits den Zeiger zum nächsten Objekt der Datenaufstellung, andererseits den Cursor der `DataGridView`-Instanz eine Zeile nach unten zu verschieben. Und ein Aufruf von `MovePrevious` macht das Gleiche bei beiden involvierten Komponenten, nur in die andere Richtung. Übrigens: Eine weitere im .NET Framework 2.0 neue Komponente macht von dieser Methode regen Gebrauch, und sie nennt sich `BindingNavigator`. Ein Beispiel dafür finden Sie auf dem alten E-Book (2005er Entwicklerbuch), das sich ebenfalls auf der Begleit-DVD befindet, und dort im ADO-Kapitel.

HINWEIS Das ListView-Steuerelement sieht das Binden von Auflistungsdaten über eine BindingSource-Komponente übrigens nicht vor.

Wie das interaktive Erstellen einer solchen Korrelation in der Praxis ausschaut, und wie einfach das vonstatten geht, zeigt der folgende Abschnitt.

Erstellen einer gebundenen DataGridView mit dem Visual Basic-Designer

Für das folgende Beispiel können Sie zwei Beispielprojekte der Begleitdateien verwenden – ein fix und fertiges und eines, mit dem Sie die folgenden Schritte nachvollziehen können.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 37\\Adresso03

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

- Öffnen Sie das Projekt und rufen Sie das schon teilweise vorbereitete Formular *frmSchnellerfassung.vb* im Designer auf. Ihnen bietet sich anschließend ein Bild, wie Sie es in etwa auch in Abbildung 37.12 sehen können.



Abbildung 37.12 Suchen Sie das DataGridView-Steuerelement in der Toolbox und ziehen Sie es ins Formular

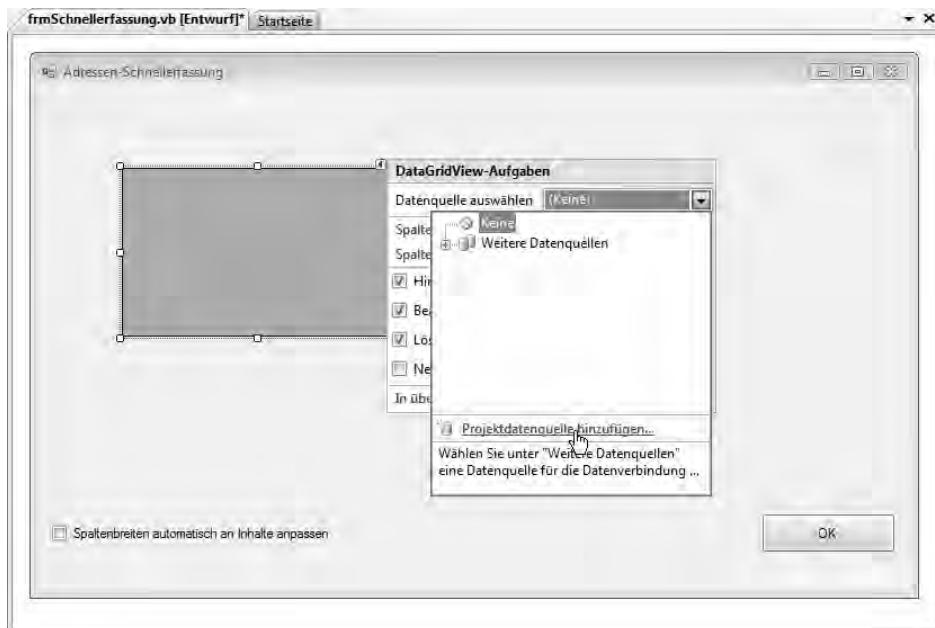


Abbildung 37.13 Wählen Sie aus der Aufgabenliste der DataGridView als Erstes Datenquelle auswählen, und klicken Sie – da es noch keine definierten Datenquellen gibt – auf den Link *Projektdatenquelle hinzufügen*



Abbildung 37.14 Der Assistent zum Konfigurieren von Datenquellen hilft Ihnen beim Erstellen einer neuen Datenquelle auf Auflistungsklassenbasis. Dazu wählen Sie im Assistentendialog das Objekt-Element

- Suchen Sie in der Toolbox das DataGridView-Steuerelement. Ziehen Sie es anschließend ins Formular. Die Aufgabenliste des DataGridView-Steuerelements wird jetzt angezeigt.
- Klappen Sie die Liste *Datenquelle auswählen* auf.
- Da es noch keine Projektdatenquellen gibt, klicken Sie auf den Link *Projektdatenquelle hinzufügen*, um eine neue Datenquelle einzurichten. Ziel dabei wird es in den folgenden Schritten sein, die Adressen-Klasse, die die Auflistungsklasse für alle Adressen unserer kleinen Adressverwaltung darstellt, zur Datenquelle für das DataGridView-Steuerelement zu promovieren.
- Der Visual Basic-Designer zeigt nun einen Assistenten an, mit dem Sie Datenquellen konfigurieren können. Im ersten Assistentenschritt, etwa wie in Abbildung 37.14 zu sehen, wählen Sie als Datenquellentyp *Objekt*.

**Abbildung 37.15**

Bestimmen Sie in der TreeView, wo in der Objekt- und Namespace-Hierarchie Ihres Projektes die Klasse, die als Datenquelle fungieren soll, zu finden ist

- Bestätigen Sie diesen Schritt des Assistenten mit *Weiter*.
- Im nächsten und auch schon letzten Schritt (Abbildung 37.15) bestimmen Sie, wo innerhalb Ihrer Projektmappe sich das Objekt (sprich: die Auflistungsklasse) befindet, dessen Instanz als Datenquelle verwendet werden soll. Klappen Sie die entsprechenden Zweige in der Treeview auf, und selektieren Sie die *Adressen*-Auflistungsklasse.
- Klicken Sie anschließend auf *Fertig stellen*, um den Assistenten zu beenden.
- Wenn Sie das DataGridView-Steuerelement anschließend vergrößern, und es mehr oder weniger flächenfüllend auf dem Formular anordnen, erkennen Sie, dass ein Prozess stattgefunden haben muss, der die Adressenklasse analysiert und aus den verschiedenen Eigenschaften eines Elementes (*Adresse*) die Grobdefinition für die Tabelle erstellt hat (Abbildung 37.16).

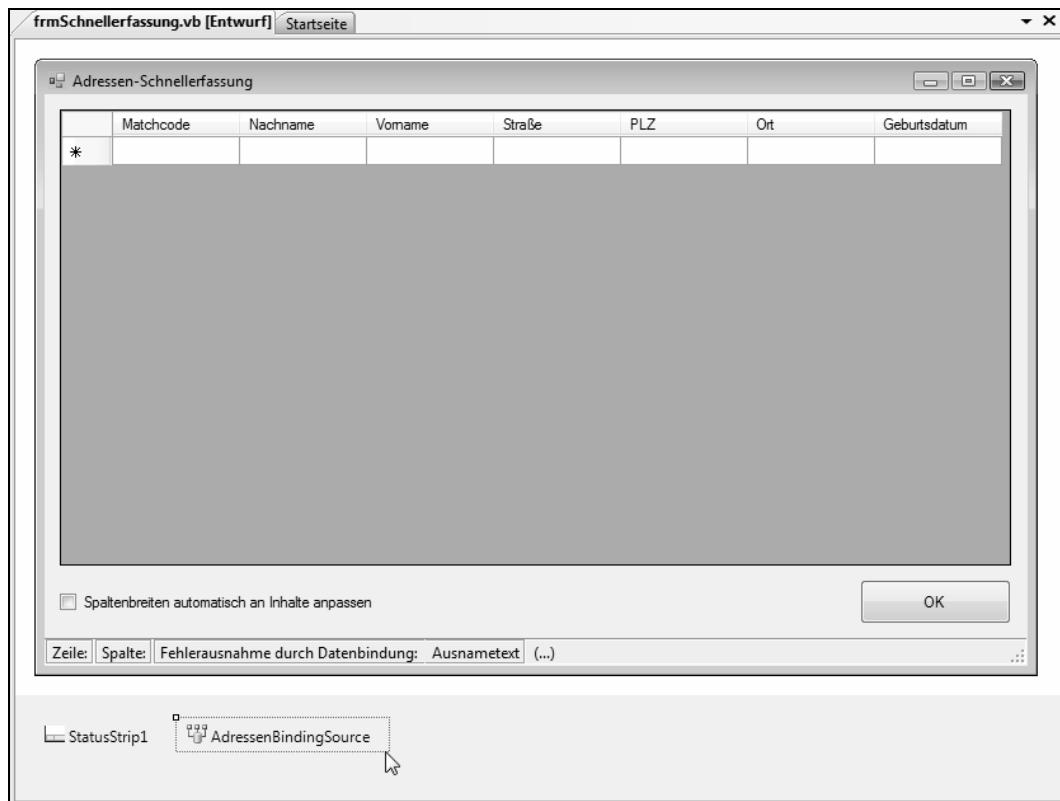


Abbildung 37.16 Nach dem Vergrößern des DataGridView-Steuerelements können Sie sehen, dass die Schemainformationen der Adresse-Klasse als Basis für das Einrichten der Tabellenspalten diente

In dieser Abbildung sehen Sie ebenfalls, dass der Designer die BindingSource-Komponente Adressen-BindingSource erstellt und im Komponentenfach des Formulars untergebracht hat. Und diese stellt ab sofort das Bindeglied zwischen der Auflistungsklasse Adressen und dem DataGridView-Steuerelement dar.

Das Beste ist: Mit einer einzigen Zeile Code können Sie nun dafür sorgen, dass die Daten im DataGridView-Steuerelement angezeigt werden können.

- Dazu wechseln Sie zum Hauptformular des Projektes *frmMain.vb*, und öffnen Sie dieses im Designer.
- Klicken Sie im MenuStrip-Steuerelement des Formulars zunächst auf *Bearbeiten*, und doppelklicken Sie anschließend auf *Schnellerfassung von Adressen*.

Im Codeeditor können Sie nun die folgenden Zeilen eingeben, sodass sich für die Ereignisbehandlungsroutine Folgendes ergibt:

```
Private Sub tsmSchnellerfassung_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmSchnellerfassung.Click
    'Neue Instanz des Dialogs
```

```

Dim locfrm As New frmSchnellerfassung

'Eine Zeile reicht aus, um die Adressen darzustellen!
Locfrm.AdressenBindingSource.DataSource = myAdressen

'Dialog modal darstellen
locfrm.ShowDialog()
End Sub

```

Wenn Sie diese Zeilen eingegeben haben und das Programm anschließend starten, wählen Sie zuerst aus dem Menü *Datei* den Menüpunkt *Zufallsadressen einfügen*. Wählen Sie anschließend aus dem Menü *Bearbeiten* den Menüpunkt *Schnellerfassung von Adressen*. Was Sie anschließend sehen, sollte nicht nur Abbildung 37.17 entsprechen, sondern Sie vor Begeisterung auch umhauen ...



Abbildung 37.17 Mit einer einzigen Codezeile steht Ihnen ein komfortabler Editor zur Bearbeitung des Adress-Pools zur Verfügung

Doch der Teufel steckt wie immer im Detail, denn es gibt noch ein paar Punkte, die keinesfalls dem entsprechen, was man unter einer komfortablen Benutzeroberfläche verstehen könnte:

- Die Datenspalten sind willkürlich angeordnet; wir sollten versuchen, eine plausible Ordnung in die Datenspalten zu bekommen.
- Die Statuszeile sollte als Navigationshilfe die Position des Cursors im DataGridView-Steuerelement anzeigen.
- Sie können in jedem Feld jede beliebige Eingabe durchführen – Eingaben werden überhaupt nicht kontrolliert, und bei Feldern, die datentypbedingt ein bestimmtes Eingabeformat verlangen, kann man dabei auf die Nase fallen (siehe Abbildung 37.18).

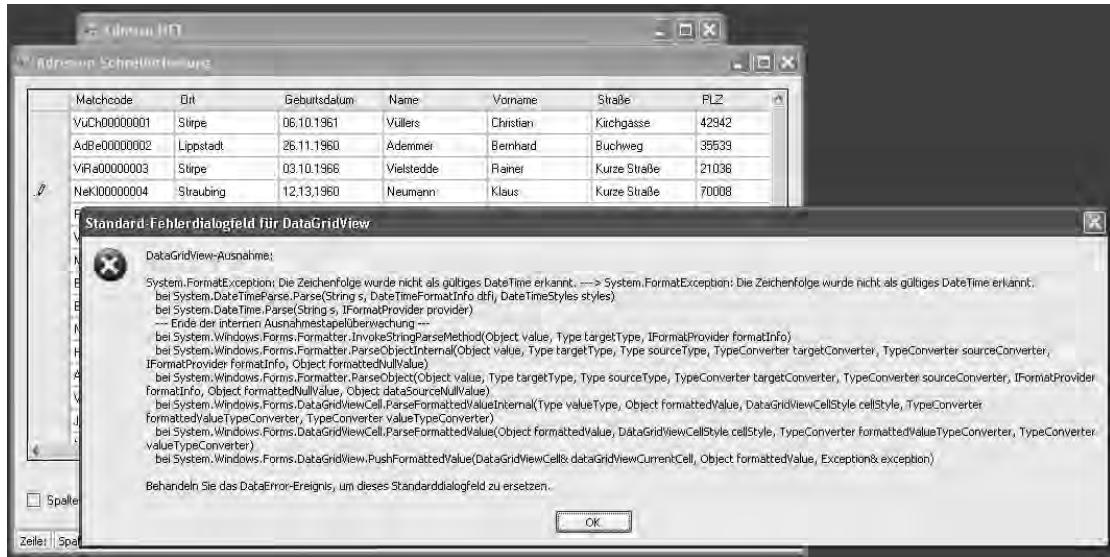


Abbildung 37.18 Solche Fehler sollten in der Release-Version Ihrer Anwendung nach Möglichkeit nicht mehr vorkommen

- Bei unserem Beispiel sollte der Matchcode nicht editierbar, auf jeden Fall aber neu eingebbar sein. Und: Auch hier dürfen doppelte Matchcodes nicht vorkommen und müssen im Rahmen einer Eingabeprüfung bei Neueingaben von Adressen ausgeschlossen werden.
- Und schließlich gibt es auch noch eine kleinen, aber gemeinen Störfaktor bei der Eingabe: Wenn Sie eine Zelle editiert haben und Ihre Änderungen mit bestätigen, springt der Cursor nicht in das nächste Feld rechts des gerade bearbeiteten Feldes, sondern in das darunter. Damit ist der Anwender gar nicht in der Lage, Adressen eine nach der anderen (also zeilenweise) neu zu erfassen.

Die nächsten Abschnitte beschreiben, wie Sie mit gar nicht soviel Codierungsaufwand diese Ziele erreichen können.

Sortieren und Benennen der Spalten eines DataGridView-Steuerelements

Wenn Sie ein DataGridView-Steuerelement an eine Datenquelle gebunden haben, werden die Schemainformationen (welche Datenspalten mit welchen Datentypen gibt es) aus der Datenquelle ermittelt. Auf Basis dieser Schemainformationen werden die Spaltenköpfe eingerichtet. Programmtechnisch rufen Sie die Spaltendefinitionen einer Tabelle mithilfe der `Columns`-Eigenschaft (vom Typ `DataGridViewColumnCollection`) ab, die eine Auflistung mit `DataGridViewColumn`-Objekten enthält. Ein einzelnes `DataGridViewColumn`-Objekt dieser Auflistung bestimmt letzten Endes die Beschaffenheit einer Spalte, indem sie Spaltennamen, Beschriftung, darstellenden Typ und Weiteres definiert.

Die Einrichtung bzw. Modifizierung nehmen Sie am einfachsten mit dem Designer vor. Die folgende Anleitung beschreibt, wie Sie ...

- ... definieren, welche Operationen (*Neuer Datensatz*, *Datensatz bearbeiten*, *Datensatz löschen*, *Umsortieren von Spalten*) mit dem DataGridView-Steuerelement durchgeführt werden dürfen,
- ... Spaltenreihenfolgen anordnen,

- ... die Beschriftung und Benennung von Spalten ändern,
- ... die Datentypen von Spalten einrichten bzw. verändern können.

BEGLEITDATEIEN Um die Schritte der folgenden Unterabschnitte nachzuvollziehen, die zunächst noch den Visual Basic-Designer involvieren, verwenden Sie das bisher verwendete Beispiel im Verzeichnis

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 37\\Adresso03 - zum Experimentieren

Dabei wird davon ausgegangen, dass Sie die Bindung der Datenquelle, wie im vorherigen Abschnitt beschrieben, bereits vorgenommen haben.

Öffnen der Liste der häufigen Aufgaben

Das DataGridView-Steuerelement lässt sich natürlich wie jedes andere Steuerelement über das Eigenschaftenfenster des Visual Basic-Designers konfigurieren. Einfacher ist es allerdings, bestimmte Einstellungen über die Liste der häufigen Aufgaben auszuführen. Um diese Liste zu öffnen, verfahren Sie wie folgt:

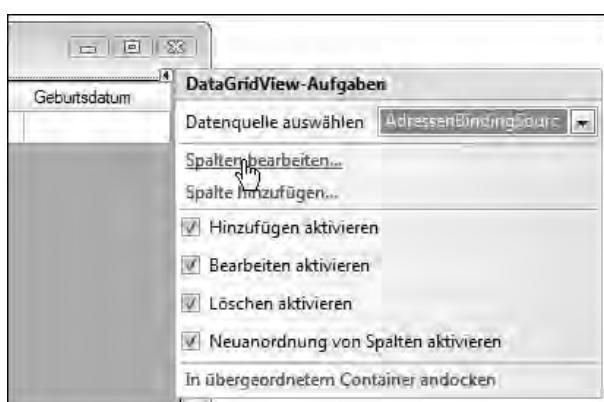


Abbildung 37.19 Mit dem Smarttag des Steuerelements erreichen Sie die Liste der häufigen Aufgaben

- Klicken Sie auf das DataGridView-Steuerelement im Formular, um es zu selektieren.
- Klicken Sie auf den Smarttag des Steuerelements (der kleine nach rechts weisende Pfeil an der rechten oberen Ecke), um die Liste häufiger Aufgaben des Steuerelements zu öffnen.

Bestimmen, welche Aufgaben mit dem DataGridView-Steuerelement erledigt werden dürfen

- Öffnen Sie die Liste häufiger Aufgaben durch Klick auf das Smarttag des zuvor selektierten DataGridView-Steuerelements (siehe Abbildung 37.19).
- Wählen Sie im Dialogfeld durch An- oder Abwählen der entsprechenden Kontrollkästchen, welche Funktionen das DataGridView-Steuerelement zu Verfügung stellen soll.

HINWEIS Auch wenn Sie hier das Hinzufügen von Datensätzen erlauben, wird diese Funktion zur Laufzeit nur dann zur Verfügung stehen, wenn die AllowNew-Eigenschaft der entsprechenden BindingSource-Komponente (oder – bei programmtechnischer Verknüpfung – auch die CurrencyManager-Komponente) auf True gesetzt wurde.

Wenn Sie später zur Laufzeit feststellen, dass beide Eigenschaften auf True gesetzt wurden, Sie aber dennoch die Funktionalität zum Hinzufügen von Datensätzen vermissen, stellen Sie sicher, ob Sie die Daten einer Auflistung nicht versehentlich mithilfe ihrer DataSource-Eigenschaft *direkt* an das DataGridView-Steuerelement anstatt an die verknüpfte BindingSource-Komponente zugewiesen haben.⁴

Anordnen der Spalten eines DataGridView-Steuerelements mit Designer-Unterstützung

- Öffnen Sie die Liste häufiger Aufgaben durch Klick auf das Smarttag des zuvor selektierten DataGridView-Steuerelements (siehe Abbildung 37.19).
- Klicken Sie auf *Spalten bearbeiten*. Sie sehen einen Dialog, etwa wie auch in Abbildung 37.20 zu sehen.

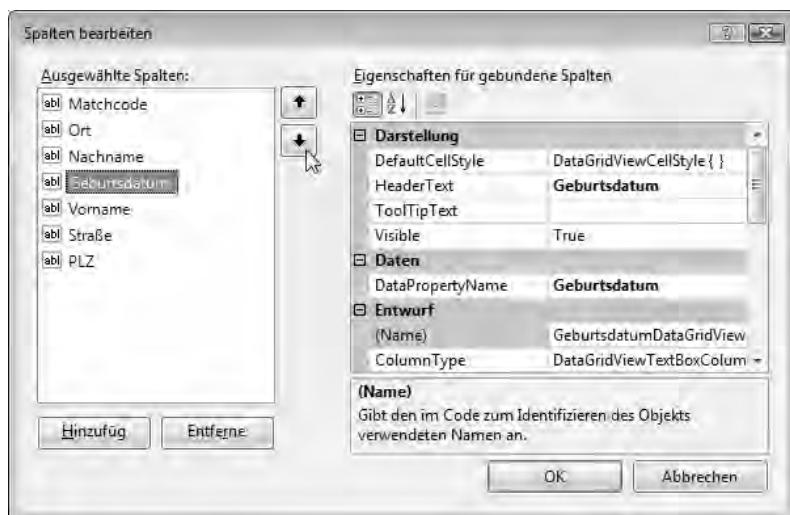


Abbildung 37.20

Mit den Pfeilschaltflächen (hier durch den Mauszeiger gekennzeichnet) ordnen Sie die Reihenfolge der Spalten neu an

- Mit den Pfeilschaltflächen ordnen Sie, wie in der Abbildung zu sehen, die Reihenfolge der Spalten des DataGridView-Steuerelements neu an. Der hier in der Liste ganz oben stehende Eintrag entspricht der ganz links im Steuerelement erscheinenden Spalte.

Die Beschriftung und Benennung von Spalten ändern

Insgesamt drei Eigenschaften eines DataGridViewColumn-Objekts, das für die Eigenschaften einer DataGridView-Spalte zuständig ist, haben etwas mit Benennungen bzw. Beschriftungen zu tun. Gerade wenn die Spaltendefinitionen aus Schemainformationen einer Datenquelle stammen, kann es anfangs verwirren, welche Eigenschaften für welche Aufgabe zuständig sind.

⁴ Und nun raten Sie mal, wieso dieses Kapitel einen halben Tag eher fertig gestellt sein können...

Um diese Einstellungen mit dem Designer durchzuführen, verfahren Sie wie folgt:

- Öffnen Sie die Liste häufiger Aufgaben durch Klick auf das Smarttag des zuvor selektierten DataGridView-Steuerelements (siehe Abbildung 37.19).
- Klicken Sie auf *Spalten bearbeiten*. Sie sehen einen Dialog, etwa wie auch in Abbildung 37.20 zu sehen.
- In der Eigenschaftenliste bestimmen Sie folgende Eigenschaften:
- **Headertext:** Legen Sie mit dieser Zeichenfolge fest, welche Überschrift die entsprechende Spalte im DataGridView-Steuerelement aufweisen soll.
- **DataPropertyName:** Bestimmen Sie mit dieser Zeichenfolge, welcher Objekteigenschaft der Datenquelle diese Spalte zugeordnet sein soll. Sie sollten diese Eigenschaft nur dann ändern, wenn sich die Eigenschaft eines Objektes der als Datenquelle fungierenden Auflistung geändert hat. Belassen Sie andererseits diese Eigenschafteneinstellung so, wie sie ist.
- **Name:** Bestimmen Sie mit dieser Zeichenfolge, mit welchem Textschlüssel Sie auf ein Element der DataGridViewColumnCollection oder der DataGridViewCellCollection (einer DataGridViewRow-Auflistung) zugreifen. Diese Eigenschafteneinstellung ist also dann wichtig, wenn Sie programmtechnisch entweder auf eine Spaltendefinition oder eine Zelle einer DataGridView-Instanz zugreifen wollen.

Zugriff auf ein Element der DataGridViewColumnCollection (eine Spaltendefinition):

```
'Index einer Spalte aus dem Spaltennamen ermitteln  
myGebDatColumnIndex = dgvAdressen.Columns("GeburtsdatumDataGridViewTextBoxColumn").Index
```

Zugriff auf ein Element der DataGridViewCellCollection (einer Zelle der Tabelle):

```
'Eine bestimmte Zelle aus Zeilennummer und Spaltennamen ermitteln  
Dim locRow As DataGridViewRow = dgvAdressen.Rows(Zeilennummer)  
Dim locCell As DataGridViewCell = locCurrentRow.Cells("MatchcodeDataGridViewTextBoxColumn")
```

Bestimmen der Spaltentypen einer DataGridView-Instanz

Mit der **ColumnType**-Eigenschaft des *Spalten bearbeiten*-Dialogs bestimmen Sie, mit welchem Steuerelementtyp die Spalte ausgestattet werden soll.

HINWEIS Die **ColumnType**-Eigenschaft bestimmt dadurch *nicht* notwendigerweise, welcher Datentyp in einer Spalte verarbeitet wird. So wählen Sie für numerische Werte, Datumswerte oder Texte als **ColumnType**-Eigenschaft wahrscheinlich immer **DataGridViewTextBoxColumn** (**DataGridViewComboBoxColumn** würden Sie dabei wählen, um aus einer vorgegebenen Liste mit Werten nur bestimmte zur Auswahl zuzulassen, oder wenn Sie eine Verknüpfung mit einer anderen Datentabelle herstellen wollen).

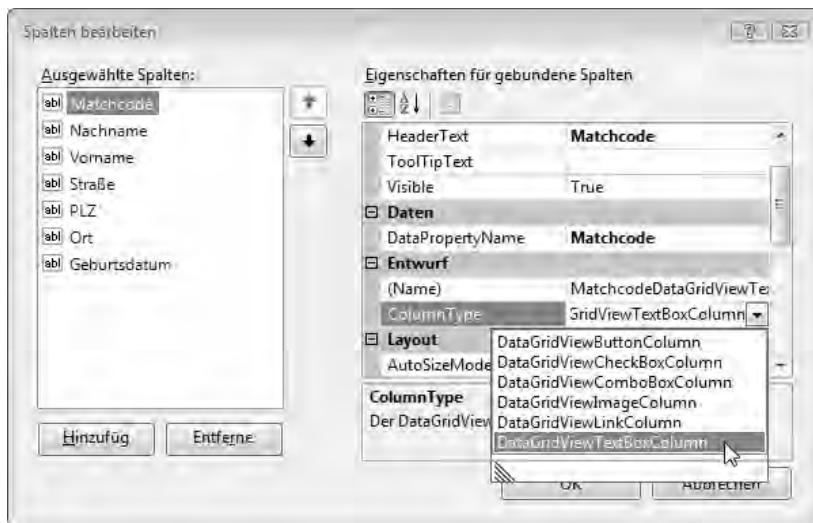


Abbildung 37.21 Mit der **ColumnType**-Eigenschaft bestimmen Sie den Steuerelementtyp (nicht Datentyp!), den eine DataGridView-Spalte verarbeiten soll

Insgesamt 6 verschiedene Spaltentypen stehen standardmäßig zur Verfügung:

ColumnType	Aufgabe
DataGridViewButtonColumn	Implementiert Schaltflächen in den Datenzeilen für die entsprechende Spalte
DataGridViewCheckBoxColumn	Implementiert Kontrollkästchen in den Datenzeilen für die entsprechende Spalte
DataGridViewComboBoxColumn	Implementiert Aufklapplisten in den Datenzeilen für die entsprechende Spalte
DataGridViewImageColumn	Implementiert Bilddarstellungen in den Datenzeilen für die entsprechende Spalte
DataGridViewLinkColumn	Implementiert einen Web-Link in den Datenzeilen für die entsprechende Spalte
DataGridViewTextBoxColumn	Implementiert ein Texteingabefeld in den Datenzeilen für die entsprechende Spalte

Tabelle 37.1 Die standardmäßig vorhandenen Steuerelemente, die als Darstellungs- und Anzeigeeinstanzen innerhalb von DataGridView-Tabellen fungieren können

HINWEIS Sie werden die *ColumnType*-Eigenschaft vergeblich im *DataGridViewColumn*-Objekt suchen – es handelt sich nämlich um eine – nennen wir sie mal – »virtuelle« Designereigenschaft, die sich nur auf die Erstellung des Designer-Codes (das nächste Kapitel erklärt mehr dazu) bezieht. Wenn Sie für ein gebundenes DataGridView-Steuerelement eine Spaltentypumstellung vornehmen wollen, platzieren Sie den entsprechenden Code dazu am besten direkt unterhalb von *InitializeComponent* im Standardkonstruktor des Formulars, das die *DataGridView* beinhaltet.

Programmtechnisch legen Sie das zu verwendende Steuerelement für die entsprechende Spalte mit der *CellTemplate*-Eigenschaft fest. Das Umdefinieren einer Spalte führen Sie am besten im Standardkonstruktor des Formulars durch, etwa wie folgt:

```
Public Class frmSchnellerfassung
    Sub New()
        ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    End Sub
```

```

InitializeComponent()

' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.

'Eines der möglichen Steuerelemente für die Spalte manuell zuordnen.
dgvAdressen.Columns("NameDataGridViewTextBoxColumn").CellTemplate = New
DataGridViewTextBoxColumn()
End Sub
.
.
.
End Class

```

TIPP Wenn Sie eigene Steuerelemente für Tabellenzellen mit individuellen Verhaltensweisen implementieren wollen, leiten Sie entweder eines der vorhandenen Steuerelemente der oben stehenden Tabelle ab, erweitern es und binden es auf die hier beschriebene Weise ein. Oder Sie leiten es von der Basis aller Steuerelemente ab, die in Tabellenzeilen einer DataGridView zur Anwendung kommen können. Verwenden Sie dazu die abstrakte Basisklasse DataGridViewCell. Wichtig: Achten Sie dabei darauf, die `Clone`-Methode zu überschreiben und neu zu implementieren, da die Definition für alle Zellen einer Spalte aus der ersten Instanz erstellt wird, die man über die `CellTemplate`-Eigenschaft definiert. Dies gilt nur dann, wenn Sie einem vorhandenen Steuerelement weitere Eigenschaften hinzufügen, deren Inhalten natürlich zusätzlich zu den bereits vorhandenen kopiert werden müssen. Dies gilt aber auf jeden Fall, wenn Sie das Steuerelement auf Basis von `DataGridViewCell` komplett neu implementieren.

Programmtechnisches Abrufen der aktuellen Zeile und aktuellen Spalte

Die aktuelle Zeile und die aktuelle Spalte, also die Position innerhalb der DataGridView-Tabelle, in der sich der Cursor gerade befindet, rufen Sie mit der Eigenschaft `CurrentCell` ab. Diese Eigenschaft liefert ein Objekt auf Basis der `DataGridViewCell`-Klasse zurück.

Um die aktuelle Zeile zu ermitteln, verwenden Sie die `RowIndex`-Eigenschaft dieses Objektes. Die aktuelle Spalte ermitteln Sie mit `ColumnIndex`.

Feststellen, dass sich die aktuelle Zelle geändert hat

Wenn Sie informiert werden möchten, wenn sich die aktuelle Position des Cursors in der Tabelle geändert hat, binden Sie das `CurrentCellChanged`-Ereignis des `DataGridView`-Steuerelements ein. Das folgende Beispiel demonstriert, wie Sie Zeile und Spalte in einer Statuszeile anzeigen lassen können; diese Anzeige wird aktualisiert, sobald sich die Cursorposition im `DataGridView`-Steuerelement verändert.

Hinweis Sie finden diese Prozedur als Teil des Projektes im Verzeichnis:

```
...\\VB 2008 Entwicklerbuch\\ G - WinForms\\Kapitel 37\\Adresso03\\
```

in der Codedatei *frmSchnellerfassung.vb*.

```

'Wird aufgerufen, wenn eine neue Zelle zur aktuellen Zelle wird.
Private Sub dgvAdressen_CurrentCellChanged(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles dgvAdressen.CurrentCellChanged
    With dgvAdressen

```

```
    ts1Zeile.Text = "Zeile: " & .CurrentCell.RowIndex.ToString  
    ts1Spalte.Text = "Spalte: " & .Columns(.CurrentCell.ColumnIndex).HeaderText  
End With  
End Sub
```

Formatieren von Zellen

Damit Daten in einem DataGridView-Steuerelement übersichtlich aufbereitet werden können, lassen sie sich durch eine Formatzeichenfolge formatieren (mehr zum generellen Umgang mit Formatzeichenfolgen lesen Sie in Kapitel 26). Möchten Sie zum Beispiel für eine Spalte, die Datumswerte anzeigt, bestimmen, dass diese im Format »ddd, dd. MMM yyyy« dargestellt werden (was für den 24.2. des Jahres 2006 die Zeichenfolge »Fr, 24 Feb 2006« ergäbe), dann stellen Sie dieses Format global für alle Zellen der entsprechenden Spalte wie folgt ein:

```
dgvAdressen.Columns("GeburtsdatumDataGridViewTextBoxColumn").DefaultCellStyle.Format = "ddd, dd. MMM yyyy"
```

Wenn Ihr DataGridView-Steuerelement Zellen lediglich darstellen, aber nicht bearbeiten muss, dann genügt diese Art der Formatierungsfestlegung für die entsprechende(n) Spalten(n) im Regelfall.

Wenn sich die Formatierung dabei nicht über eine Formatzeichenfolge realisieren lässt, können Sie das CellFormatting-Ereignis des DataGridView-Steuerelements behandeln, um die Formatierung dort vorzunehmen. Zwei Möglichkeiten stehen Ihnen nun zur Formatierung zur Verfügung:

- Bestimmen Sie über die Eigenschaftenkombination `CellStyle.Format` des Ereignisparameters `e` die individuell anzuwendende Formatzeichenfolge. Oder:
- Lesen Sie den zu formatierenden Wert über die `Value`-Eigenschaft des Ereignisparameters `e` aus, konvertieren Sie ihn im Bedarfsfall in den entsprechenden Datentyp, formatieren Sie ihn (in der Regel in Form einer Zeichenkette) und schreiben Sie ihn anschließend zurück in die `Value`-Eigenschaft. Setzen Sie in diesem Fall die `FormattingApplied`-Eigenschaft des Ereignisparameters `e` auf `True`, um dem DataGridView-Steuerelement anzugeben, dass Sie die Formatierung der Zelle komplett selbst übernommen haben.

HINWEIS Denken Sie bitte daran, dass das `CellFormatting`-Ereignis für jede darzustellende Zelle ausgelöst wird. Damit Sie die richtige zu formatierende Zelle »treffen«, testen Sie mit dem Ereignisparameter `RowIndex`, ob sich das gerade ausgelöste Ereignis auf die entsprechende Spalte bezieht. Ein Beispiel dafür finden Sie im nächsten Abschnitt.

Wenn Ihre individuell formatierte Zelle auch die Bearbeitung von Werten zulässt, lesen Sie bitte auch den nächsten Abschnitt.

Problemlösung für das Parsen eines Zellenwerts mit einem komplexen Anzeigeformat

Bei der Datenerfassung von anderen Typen als reine Zeichenketten stellt sich immer das Problem der Datenkonvertierung. Für die wichtigsten Datentypen implementiert das DataGridView-Steuerelement so genannte Parsing-Algorithmen, die den meisten Konvertierungsansprüchen Genüge tun.

Das Problem bei der Verarbeitung anderer Typen als reine Zeichenketten ist, dass der Datentyp für die Darstellung in eine Zeichenkette umgewandelt werden muss. Bearbeitet der Anwender eine Zelle, in der ein

anderer Datentyp als eine Zeichenkette erfasst werden soll, editiert er natürlich zunächst einmal ebenfalls eine Zeichenkette, die anschließend wieder zurück in den eigentlichen Datentyp konvertiert werden muss.

Solange wie Sie ein sehr einfaches Standardformat für die Darstellung des Datentyps verwenden, treten dabei keine Probleme auf. Handelt es sich aber um eine sehr informative Aufbereitung eines Datentyps, dann wird der Standardparser des DataGridView-Steuerelements dieses Formats nicht mehr hergeben. Oder, um ein Beispiel zu bemühen:

Sie stellen in einer Spalte, in der Datumswerte angezeigt und bearbeitet werden sollen, die Werte im Format »dd.MM.yyyy« dar. Die Darstellung ist zwar sehr simpel, aber der Parser hat mit dieser Darstellung keine Probleme – er kann also einen in diesem Format editierten Wert ohne Probleme wieder zurück in den eigentlichen Datentyp umwandeln.

Entscheiden Sie sich allerdings für die Darstellung von »dd.MM.yyyy (dddd)« – der Wochentag wird hier also in Klammern neben dem eigentlichen Datum angezeigt –, und Sie lassen den Anwender die Zelle auch in diesem Format editieren, wird der Parser mit dieser komplexen Darstellung des Datums nicht mehr zurechtkommen.

Nun gibt es zwei generelle Möglichkeiten, dieses Problem zu umgehen: Sie implementieren für die entsprechende Datenspalte eine Ereignisbehandlungsroutine für das CellParsing-Ereignis und sorgen dort selbst dafür, dass der Parsing-Vorgang auch mit dem komplex formatierten Datentyp nicht fehlschlägt. Das kann aber unter Umständen eine ganze Menge Programmieraufwand bedeuten.

Oder, und das wäre meine empfohlene Vorgehensweise, sie formatieren zum Editieren die entsprechende Zelle anders als für die reine Anzeige. Und das funktioniert, weil sowohl beim Anzeigen als auch kurz vor dem Editieren das CellFormatting-Ereignis ausgelöst wird. Das folgende Beispiel zeigt, wie es funktioniert:

Hinweis Sie finden diese Prozedur als Teil des Projektes im Verzeichnis:

... \VB 2008 Entwicklerbuch\G - WinForms\Kapitel 37\Adresso03\

in der Codedatei *frmSchnellerfassung.vb*.

```
'Wird für jede Zelle aufgerufen, wenn ihr Inhalt formatiert wird.
Private Sub dgvAdressen_CellFormatting(ByVal sender As Object,
 ByVal e As System.Windows.Forms.DataGridViewCellFormattingEventArgs) Handles dgvAdressen.CellFormatting

    'Nur die Geburtsdatums-Spalte wird besonders formatiert.
    If e.ColumnIndex = myGebDatColumnIndex Then
        'Die Zelle ermitteln. Aus Geschwindigkeitsgründen sollte das in einem
        'Rutsch erfolgen; hier getrennt, damit es deutlicher wird.
        Dim locCurrentRow As DataGridViewRow = dgvAdressen.Rows(e.RowIndex)
        Dim locCurrentCell As DataGridViewCell = locCurrentRow.Cells(e.ColumnIndex)

        'Wenn ein Geburtsdatum zum Bearbeiten formatiert wird,
        'dann ein "parse"-bares Format darstellen.
        If locCurrentCell.IsInEditMode Then
            eCellStyle.Format = "dd.MM.yyyy"
        Else
            'Nur für die Anzeige ein "mehr informatives Format" darstellen.
            eCellStyle.Format = "dd.MM.yyyy (dddd)"
        End If
    End If
End Sub
```

Das Ergebnis dieses Aufwands können Sie betrachten, wenn Sie das Beispielprojekt starten, ein paar Zufallsadressen erzeugen und diese zum Schnellbearbeiten darstellen lassen. Wenn Sie anschließend das Geburtsdatum eines Datensatzes bearbeiten, sehen Sie, dass sich das Format zur Eingabe ändert.

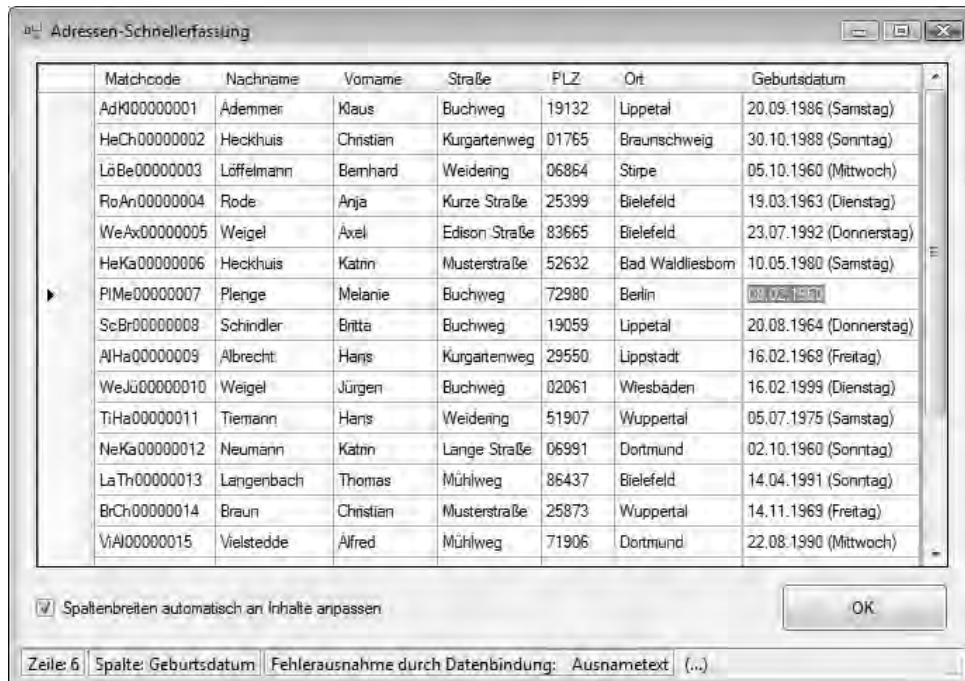


Abbildung 37.22 Durch geschickte Behandlung des CellFormatting-Ereignisses wird für die Bearbeitung ein anderes Format als für die Anzeige verwendet

HINWEIS Da das CellFormatting-Ereignis für jede einzelne darzustellende Zelle im Anzeigebereich des DataGridView-Steuerelements aufgerufen wird, sollten Sie sehr auf eine effiziente Programmierung achten und Code formulieren, der so wenig Zeit wie möglich benötigt.

Verhindern des Editierens bestimmter Spalten aber Gestatten Ihrer Neueingabe

Falls Sie das Adresso-Szenario bis hierher verfolgt haben, dann wissen Sie, dass das Beispiel das Erfassen eines Matchcodes beim Neuerfassen einer Adresse erlaubt, es aber verbietet, den Matchcode einer vorhandenen Adresse zu editieren.

Mit dem Setzen simpler Eigenschaften können Sie das DataGridView-Steuerelement nicht dazu bringen, solche Teilreglementierungen umzusetzen.

Zwar ist es möglich, beispielsweise eine Spalte mit der Anweisung

```
dgvAdressen.Columns("MatchcodeDataGridViewTextBoxColumn").ReadOnly = True
```

für die Bearbeitung ganz zu sperren. Doch das entspricht nicht dem Sinne des Erfinders, jedenfalls nicht bei unserem Beispiel.

Mit dem CellBeginEdit-Ereignis können Sie die Kontrolle über das Editieren einer Zelle in die eigenen Hände nehmen. Innerhalb dieses Ereignisses brauchen Sie nämlich nur festzustellen, ob eine neue Zeile eingegeben oder eine vorhandene editiert wird, und um welche Datenspalte es sich dabei handelt.

```
'Wird aufgerufen, bevor das Bearbeiten einer Zelle startet.
Private Sub dgvAdressen_CellBeginEdit(ByVal sender As Object, ByVal e As _
    System.Windows.Forms.DataGridViewCellCancelEventArgs) Handles dgvAdressen.CellBeginEdit

    'Verhindern, dass der Matchcode bearbeitet werden kann.
    If e.ColumnIndex = dgvAdressen.Columns("MatchcodeDataGridViewTextBoxColumn").Index Then

        'Neue Matchcodes können zwar bearbeitet werden, aber keine vorhandenen.
        'IsNewRow zeigt ein, ob es sich um die "Neue-DataGridView-Zeile" handelt.
        If Not dgvAdressen.CurrentRow.IsNewRow Then
            'Wie gesagt: NICHT neue Zeile, dann Bearbeitung nicht zulassen.
            e.Cancel = True
        End If
    End If
End Sub
```

Schlüssel bei diesem Beispiel sind zwei Komponenten: Zum Einen die Eigenschaft IsNewRow des CurrentRow-Objektes des DataGridView-Steuerelements, die Auskunft darüber gibt, ob eine vorhandene Zeile bearbeitet oder eine neue eingegeben wird. Zum Anderen die Cancel-Eigenschaft des Ereignisparameters e des CellBeginEdit-Ereignisses, mit dem Sie durch Setzen auf True verhindern können, dass die Zeile editiert wird.

Überprüfen der Richtigkeit von Eingaben auf Zellen- und Zeilenebene

Ein Blick auf Abbildung 37.11 offenbart, dass es beim DataGridView-Steuerelement so etwas wie ErrorProvider auf Zellen- und Zeilenebene gibt (falls Sie diese ErrorProvider-Klasse nicht kennen, der Abschnitt »Überprüfung auf richtige Benutzereingaben in Formularen« ab Seite 1043 weiß mehr dazu).

Und in der Tat: Idealerweise erlauben Sie Falscheingaben zunächst zwar auf Zellenbasis, machen den Anwender aber im Moment des Verlassens einer »falsch« bearbeiteten Zelle auf diesen Missstand aufmerksam. Sie erlauben aber nicht das Verlassen einer Zeile, solange diese noch nicht korrigierte Fehler enthält.

Wenn der Anwender eine Eingabe durchgeführt hat, muss die Richtigkeit seiner Eingabe überprüft werden. Diesen Vorgang nennt man Validieren. Dementsprechend nennen sich die Ereignisse des DataGridView-Steuerelements, die ausgelöst werden, wenn Eingabeüberprüfungen auf Zellen- bzw. Zeilenebene anstehen, CellValidating und RowValidating.

Hinweis Sie finden diese Prozedur als Teil des Projektes im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\ G - WinForms\\Kapitel 37\\Adresso03\\

in der Codedatei *frmSchnellerfassung.vb*.

Die folgenden beiden Ereignisbehandlungsroutine implementieren die Eingabereglementierungen für unsere Adresso-Anwendung auf bekannte Weise:

- Es dürfen keine doppelten Matchcodes vorhanden sein.
- Jedes Feld muss ausgefüllt werden.
- Postleitzahlen müssen mindestens 4-stellig und dürfen höchstens 5-stellig sein, und sie müssen aus Ziffern bestehen.
- Nur gültige Datumsformate dürfen eingegeben werden.

Die prinzipielle Arbeitsweise beider Routinen ist wie folgt:

- Als erstes wird das `CellValidatingEreignis` aufgerufen, wenn der Cursor das Eingabefeld verlässt. **Wichtig:** Dieses Ereignis wird auch dann aufgerufen, wenn keine Eingabe oder Änderung in einem Feld erfolgte, also auch dann, wenn der Cursor nur über das Feld hinweg bewegt wurde. Die Validierung wird in der unten stehenden Ereignisbehandlungsroutine deswegen nur dann ausgeführt, wenn die Zelle mit ihrer `IsInEditMode`-Eigenschaft angezeigt, dass sie tatsächlich bearbeitet wurde (oder besser: *wird* – denn zum Zeitpunkt der Validierung läuft die Bearbeitung noch und das Beenden der Bearbeitung könnte auch mit `e.Cancel = True` verhindert werden!).
- Wurde ein Fehler in einem der Eingabefelder festgestellt, wird die Zelle durch Zuweisen eines Textes an ihre `ErrorText`-Eigenschaft als fehlerhaft markiert. Das bewirkt, dass ein kleines rotes Ausrufungszeichen an das Ende der Zelle gesetzt wird. Den Fehlertext bekommt der Anwender als Tooltip angezeigt, wenn er mit dem Mauszeiger auf das Ausrufungszeichen fährt. War das Eingabefeld okay, wird die `ErrorText`-Eigenschaft auf `Nothing` gesetzt. Ein etwaiger vorhandener Fehlertext aus einer vorherigen Falscheingabe wird auf diese Weise zurückgesetzt. An dieser Stelle wird das Verlassen einer Zelle nach rechts oder links allerdings noch nicht verhindert.
- Versucht der Anwender eine neue Zeile »zu betreten«, tritt das `RowValidating-Ereignis` ein. Jetzt hat das Programm die Möglichkeit, die Richtigkeit der kompletten Zeile zu überprüfen. Das ist dann der Fall, wenn keine einzelne Zelle der betroffenen Zeile mehr über eine gesetzte `ErrorText`-Eigenschaft verfügt, und auf genau das überprüft der Code in dieser Ereignisbehandlungsroutine. Sollte er allerdings dabei auf noch nicht korrigierte Fehler stoßen, verhindert er einerseits das Verlassen der Zeile durch Setzen von `e.Cancel = True` und setzt ebenfalls einen Fehlerhinweis (dieses Mal auf Zeilenebene), dass es in der Zeile noch nicht korrigierte Fehler gibt.

```
'Wird aufgerufen, wenn der Inhalt einer Zelle überprüft werden soll.  
Private Sub dgvAdressen_CellValidating(ByVal sender As System.Object, ByVal e As  
    System.Windows.Forms.DataGridViewCellValidatingEventArgs) Handles dgvAdressen.CellValidating  
  
    'Aktuell zu validierende Zeile und Zelle ermitteln  
    Dim locCurrentRow As DataGridViewRow = dgvAdressen.Rows(e.RowIndex)  
    Dim locCurrentCell As DataGridViewCell = locCurrentRow.Cells(e.ColumnIndex)  
  
    'Zellen werden nur validiert, wenn sie bearbeitet wurden.  
    '(Und beim Validieren *werden* sie noch bearbeitet)  
    If locCurrentCell.IsInEditMode Then  
  
        'Postleitzahleninhalt überprüfen  
        If e.ColumnIndex = dgvAdressen.Columns("PLZDataGridViewTextBoxColumn").Index Then
```

```

'Postleitzahlen dürfen nicht mehr als 5-stellig sein,
'und nur aus Ziffern bestehen.
If (e.FormattedValue.ToString.Length < 6 And -
    e.FormattedValue.ToString.Length > 4 And -
    IsNumeric(e.FormattedValue.ToString)) Then
    locCurrentCell.ErrorText = Nothing
Else
    locCurrentCell.ErrorText = "Die Postleitzahl hat das falsche Format!"
End If

ElseIf e.ColumnIndex = dgvAdressen.Columns("GeburtsdatumDataGridViewTextBoxColumn").Index
Then

    'Geburtsdatumsformat überprüfen
    Dim locDate As Date
    If Not Date.TryParse(e.FormattedValue.ToString, locDate) Then
        locCurrentCell.ErrorText = "Das eingegebene Datum hat das falsche Format!"
    Else
        'Zurücksetzen
        locCurrentCell.ErrorText = Nothing
    End If
Else
    'Alle anderen Felder nur auf nicht vorhandene Eingabe überprüfen
    If String.IsNullOrEmpty(e.FormattedValue.ToString) Then
        locCurrentCell.ErrorText = "Fehlende Eingabe!"
    Else
        locCurrentCell.ErrorText = Nothing
    End If
End If
End Sub

'Wird aufgerufen, wenn beim Wechseln der Zeile
'die Richtigkeit der Inhalte *aller* Zellen überprüft werden sollen.
Private Sub dgvAdressen_RowValidating(ByVal sender As Object, ByVal e As _
    System.Windows.Forms.DataGridViewCellCancelEventArgs) Handles dgvAdressen.RowValidating

    'Alle Zellen der betroffenen Zeile durchlaufen
    For Each locCell As DataGridViewCell In dgvAdressen.Rows(e.RowIndex).Cells

        'Schauen, ob noch irgendwo ein Fehlerertext vermerkt ist
        If Not String.IsNullOrEmpty(locCell.ErrorText) Then
            'Falls ja, dann auch einen Fehler auf Zeilenbasis eintragen
            dgvAdressen.Rows(e.RowIndex).ErrorText =
                "Diese Zeile hat noch nicht korrigierte Fehler!"
            e.Cancel = True
            Return
        End If
    Next

    'Keine der Zellen wies einen Fehler auf --> Zeilenfehler zurücksetzen.
    dgvAdressen.Rows(e.RowIndex).ErrorText = Nothing
End Sub

```

Ändern des Standardpositionierungsverhaltens des Zellencursors nach dem Editieren einer Zelle

Das DataGridView-Steuerelement hat ein etwas seltsames Verhalten, das bei Anwendern, die viele Eingaben mit seiner Hilfe vornehmen müssen, auf wenig Gegenliebe stößt. Nachdem Sie eine Zelle im Steuerelement geändert haben, springt der Cursor nicht in die rechts daneben stehende Zelle, sondern er springt eine Zelle nach unten.

Da es Anwender nun einmal gewohnt sind, eine Eingabe auch mit der Taste und nicht mit der Cursor-nach-rechts-Taste zu beenden, macht sich hier schnell Unmut breit.

Dummerweise verfügt das DataGridView-Steuerelement zwar über die meisten Einstellmöglichkeiten aller Steuerelemente überhaupt, aber über keine, um dieses Verhalten direkt zu ändern.

Die Klassenvererbung ist die einzige Möglichkeit, diesem Problem zu entgehen. Das bedeutet: Sie vererben das DataGridView-Steuerelement, das im Prinzip ja nichts anderes ist als eine Klasse, in eine neue Klasse und kommen auf diese Weise an bestimmte Methoden und Eigenschaften heran, die Ihnen bei der direkten Verwendung eines DataGridView-Objektes nicht zugänglich sind. Und dabei handelt es sich um genau die Elemente, die als Protected so gekennzeichnet sind, dass auf sie nur von der Klasse selbst und von jeder geerbten Klasse aus darauf zugegriffen werden kann.

Wenn Sie ein Steuerelement in Visual Basic 2008 vererben, erstellen Sie automatisch ein neues Steuerelement. Damit hat dieser Abschnitt in diesem Kapitel eigentlich gar nichts verloren, denn erst Kapitel 30 beschäftigt sich mit diesem Thema detaillierter. Mir schien es aber sinnvoll, gerade dieses herbe Manko des DataGridView-Steuerelements im Kontext des Themas *DataGridView* zu erklären. Wenn Sie genauer interessiert, wie Sie eigene Steuerelemente mit Visual Basic 2008 für die Vereinfachung von Programmierungen Ihrer eigenen Anwendungen erstellen, lesen Sie eben Kapitel 30. Die Erklärung des Programms an dieser Stelle erfolgt deswegen auch nur in aller Kürze.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 37\\Adresso04\\

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Zentrale Änderung zum vorherigen Beispielprojekt ist eine neue Klassendatei namens *DataGridViewEx.vb*, die die Erweiterung des DataGridView-Steuerelements in Form eines neuen Steuerelements namens *DataGridViewEx* implementiert.

Um dieses erweiterte Steuerelement mit der »richtig« funktionierenden -Taste in Ihren zukünftigen Projekten zu verwenden, kopieren Sie diese Klassendatei einfach in Ihr Projektverzeichnis, lassen anschließend mit Klick auf das entsprechende Symbol im Projektexplorer alle Dateien anzeigen, wählen die nun sichtbar gewordene Klassendatei aus und rufen aus dem Kontextmenü des Projektexplorers den Befehl *Zum Projekt hinzufügen* auf. Nach dem ersten Komplizieren des Projekts erscheint das neue Steuerelement automatisch in der Toolbox, und Sie können es wie eine normale DataGridView verwenden.

Möchten Sie, dass dieses Steuerelement anstelle bestehender DataGridView-Steuerelemente verwendet wird, dann schließen Sie zunächst ein möglicherweise im Designer geöffnetes Formular, das die »betroffene« DataGridView beinhaltet, und öffnen Sie den Designer-Code dieses Formulars. Den Designer-Code erreichen Sie,

indem Sie im Projektmappen-Explorer mit dem entsprechenden Symbol *Alle Dateien anzeigen* lassen und anschließend den Zweig vor dem Formular erweitern, das das DataGridView-Steuerelement erhält. Öffnen Sie nun den Code der unter der eigentlichen Formularklasse stehenden Codedatei mit der Endung *.designer.vb*. Suchen Sie hier alle System.Windows.Forms.DataGridView-Referenzen (per DataGridView dürften es nicht mehr als 2 sein) und ersetzen Sie sie durch DataGridViewEx. Kompilieren Sie das Projekt anschließend neu.

Den dokumentierten Code des erweiterten Steuerelements finden Sie im Folgenden:

```
Public Class DataGridViewEx
    Inherits DataGridView

    'Verarbeitet Tasten, z. B. TAB, ESC, die EINGABETASTE und Pfeiltasten,
    'die zum Steuern von Dialogfeldern verwendet werden.
    Protected Overrides Function ProcessDialogKey(ByVal keyData As Keys) As Boolean

        'Nur die Keycodes interessieren, Rest wird ausmaskiert.
        Dim key As Keys = (keyData And Keys.KeyCode)

        'Eingabe-Taste gedrückt?
        If key = Keys.Enter Then

            'Ja, dann in Cursor-rechts umleiten, wo
            'Eingabe-Taste nochmal gesondert behandelt wird.
            Return MyBase.ProcessRightKey(keyData)
        End If

        'Andere Keys bleiben nicht betroffen
        Return MyBase.ProcessDialogKey(keyData)
    End Function

    'Die alte Cursor-Rechts-Funktion wird überschattet, damit die
    'Polymorphie dieser Funktion an dieser Stelle unterbrochen wird.
    Public Shadows Function ProcessRightKey(ByVal keyData As Keys) As Boolean

        'Nur die Keycodes interessieren, Rest wird ausmaskiert.
        Dim key As Keys = (keyData And Keys.KeyCode)

        'Wenn es sich um die umgeleitete Eingabe-Taste handelte...
        If key = Keys.Enter Then

            'Feststellen, ob sich der Cursor am Ende der letzten Spalte befand, ...
            If MyBase.CurrentCell.ColumnIndex = (MyBase.ColumnCount - 1) Then
                '...dann den Cursor nach vorn und eine Zeile nach unten verschieben.
                MyBase.CurrentCell = MyBase.Rows(MyBase.CurrentCell.RowIndex + 1).Cells(0)
                Return True
            End If
        End If

        'Bei allen anderen Positionen reicht es aus,
        'das Standardverhalten von Cursor-rechts statt Eingabe zu verwenden.
        Return MyBase.ProcessRightKey(keyData)
    End Function

    'Verarbeitet Tasten, die zum Navigieren in der DataGridView verwendet werden.
    Protected Overrides Function ProcessDataGridViewKey(ByVal e As KeyEventArgs) As Boolean
        If e.KeyCode = Keys.Enter Then
            Return MyBase.ProcessRightKey(e.KeyData)
        End If
        Return MyBase.ProcessDataGridViewKey(e)
    End Function
End Class
```

Entwickeln von MDI-Anwendungen

MDI ist die Abkürzung für *Multi Document Interface*, und das bedeutet etwas freier übersetzt: Benutzeroberfläche für mehrere Dokumente. Was dieser Ausdruck meint: MDI-Anwendungen verfügen über ein Hauptformular (oder, um meinen Fachlektor zufrieden zu stellen, über ein *Hauptfenster*), das die generische Grundfunktionalität der gesamten Anwendung anbietet. Es bindet aber gleichzeitig für jedes eigentliche Dokument, das Sie mit ihm bearbeiten, mehrere Unterfenster (die aus dem Englischen übernommene Bezeichnung müsste eigentlich *Kindfenster* – von *Child Window* – heißen, hat sich aber nicht durchgesetzt) ein, die jeweils das eigentliche Dokument darstellen.

Was dabei genau ein »Dokument« darstellt, bestimmt die Anwendung. Das können Bitmaps, Zeichnungen, Kalkulationstabellen oder auch Textdokumente sein, etwa wie in der folgenden Abbildung zu sehen:

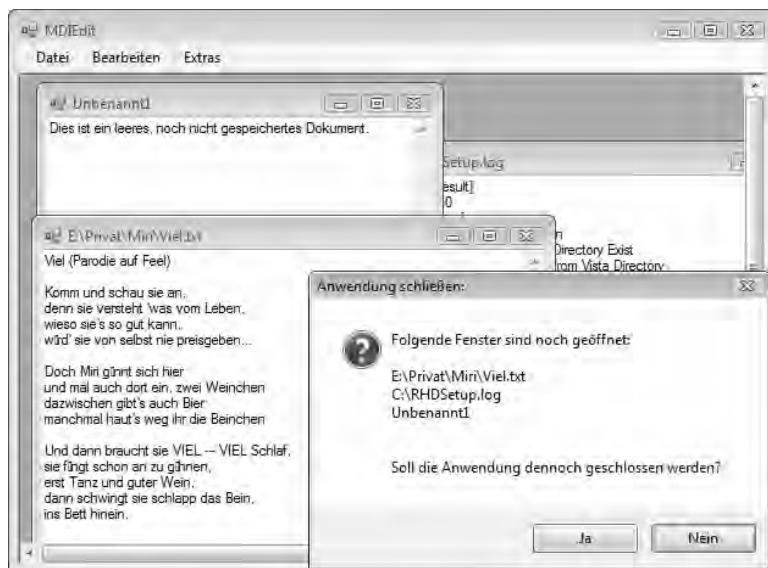


Abbildung 37.23 Eine typische MDI-Anwendung – hier zum Bearbeiten von Textdokumenten

Das eigentliche Darstellen eines untergeordneten Fensters in einem übergeordneten Fenster (eigentlich: *Elternfenster*, da von *Parent Window* abgeleitet), ist noch das kleinere Problem bei der Entwicklung bzw. Konzeption von MDI-Anwendungen. Mit insgesamt zwei Eigenschaften der Formularklasse sorgen Sie nämlich dafür, ein Formular einerseits zum Container für untergeordnete Fenster zu machen (`FormInstanz.IsMdiContainer = True`), andererseits dafür, dass eine neue Instanz eines Formulars zum untergeordneten Formular eines MDI-Hauptformulars wird (`UntergeordneteForm.MdiParent = Elternform`). Ein Aufruf der `Show`-Methode des untergeordneten Formulars genügt anschließend, um das Formular als Kindfenster einer MDI-Anwendung darzustellen.

Größere Probleme bilden bei der Entwicklung von MDI-Anwendungen die folgenden Punkte:

- Welche Funktionen werden von der Hauptanwendung, welche von den untergeordneten Formularklassen idealerweise übernommen?
- Wie lassen sich die Funktionsaufrufmöglichkeiten der beiden Instanzen visuell vereinen?

Lassen Sie uns beim Beispiel aus Abbildung 37.23 bleiben, um diese Problemstellungen zu erläutern.

Beim Entwickeln einer MDI-Texteditor-Anwendung wird recht schnell klar, dass globale Funktionen wie das Öffnen einer Datei, das Zur-Freigabe-Stellen eines globalen Optionsdialogs oder das Beenden des Programms in der Klasse des Hauptformulars implementiert werden müssen. Alle weiteren Funktionen wie das Speichern einer Datei, das Drucken, Suchen und Ersetzen im Text müssen von der jeweiligen Dokumentenklasse bereitgestellt werden, denn diese zuletzt genannten Funktionen beziehen sich immer auf die verschiedenen Dokumentinstanzen.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 37\\MDIEdit\\

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Um das Beispiel einfach zu halten, beschränken wir uns auf folgende Funktionalität:

Für das globale Hauptformular:

- Öffnen einer Datei
- Erstellen einer neuen Datei
- Beenden des Programms (und Schließen aller noch geöffneten Dokumente mit Sicherheitsabfrage und Abbruchmöglichkeit)
- Aufruf eines Options-Dialogs.

Für jede Dokumentklasse:

- Speichern der Datei
- Aufruf einer Suchfunktion.

In dieser Beispielanwendung gibt es nur eine einzige Dokumentenklasse. Denkbar wären aber auch zwei Dokumentenklassen mit unterschiedlicher Funktionalität – eine beispielsweise für das Bearbeiten reiner Textdateien, eine andere für das Bearbeiten von Richt-Text-Dokumenten, also solchen, die mit Formatierungen ausgestattet werden können.

Und hier ergibt sich nun das nächste Problem, nämlich das der unterschiedlichen Benutzeroberfläche. Eine Dokumentenklasse, die Richt-Text-Dokumente bearbeitet, muss beispielsweise in der Hauptanwendung ein *Format*-Menü bereitstellen; eine einfache Textdokumentklasse benötigt dieses nicht.

Elegant wäre es, wenn jede Dokumentenklasse ihre eigene Benutzeroberfläche in Form von Menü und Toolbar mitbrächte, die dann mit der Hauptanwendung verschmelzen würde, sobald man eine ihrer Instanzen aktivierte.

Das Ergebnis sähe dann so aus wie in den folgenden beiden Abbildungen zu sehen:



Abbildung 37.24 Solange es keine geöffneten Dokumente gibt (und damit keine instanziierten Dokumentklassen), steht nur der Grundpool an Funktionen zur Verfügung, der über die Menüstruktur zu erreichen ist. Dieser wird ...

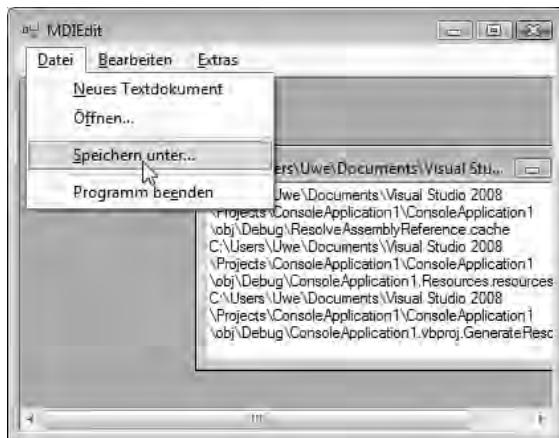


Abbildung 37.25 ... aber durch die jeweils aktive Dokumentklasseninstanz erweitert – die beiden Funktionssätze verschmelzen sozusagen.

Um diese Technik zu erreichen, bieten sowohl das Menustrip-Steuerelement (mit dem Sie eine Menüstruktur im Fenster darstellen lassen können) als auch das ToolStrip-Steuerelement (mit dem Sie Toolbars anzeigen lassen können) besondere Eigenschaften an: Menüs (und auch Toolbars) von untergeordneten Formularen können dynamisch und nur durch Aktivierung zur Laufzeit mit denen von übergeordneten Formularen zusammengeführt werden.

Wichtig zu wissen sind dabei die folgenden Punkte:

- Sie erstellen ein Menü oder eine Toolbar für das Hauptfenster und müssen nur darauf achten, dass die standardmäßig gesetzte AllowMerge-Eigenschaft des jeweiligen Steuerelements (MenuStrip, ToolStrip) auch wirklich gesetzt ist.
- Sie erstellen ein Menü oder eine Toolbar für jedes untergeordnete Fenster und »tun« zunächst so, als wäre das Menü oder die Toolbar ausschließlich für dieses Fenster gedacht.
- Für untergeordnete Fenster bestimmen Sie anschließend über die MergeAction und MergeIndex-Eigenschaften, auf welche Weise und an welcher Stelle im übergeordneten Fenster die Elemente vereinigt werden sollen. Bei Elementen mit Unterelementen ergibt sich dabei eine besonders knifflige

Problematik, wenn Elemente untergeordneter Ebenen zusammengeführt werden sollen. In diesem Fall müssen Sie die MergeIndex-Eigenschaft der Elemente der übergeordneten Ebene auf die gleiche Indexnummer des entsprechenden Elements im übergeordneten Fenster setzen, und als MergeAction den Wert MergeOnly einstellen. Möchten Sie aus dem untergeordneten Formular bestimmte Elemente übernehmen, legen Sie für diese als MergeIndex die Positionen fest, die der Position im Hauptfenster entsprechen, und für MergeAction bestimmen Sie den Wert Insert.

- Wenn das untergeordnete Formular zur Laufzeit seine sämtlichen Elemente (Menüs, Toolbars) mit dem übergeordneten verschmelzen lassen soll, dann legen Sie schon zur Entwurfszeit die Visible-Eigenschaft des Steuerelements auf False fest.

ACHTUNG Aber aufgepasst dabei: Wenn Sie die Visible-Eigenschaft eines ToolStrip- oder eines MenuStrip-Steuerelements zur Entwurfszeit auf False setzen, dann verschwindet es auch im Designer. Das ist ungewöhnlich, aber logisch: Denn Ihnen muss beim Gestalten des Formulars ja schließlich der Platz für das unsichtbare Steuerelement »gutgeschrieben« werden. Das Problem: Wie verändern Sie die Elemente eines unsichtbaren Steuerelements, denn Sie können es ja nicht mehr anklicken!? Abbildung 37.26 hat die Lösung: Sie können es für den Augenblick der Bearbeitung wieder sichtbar machen, indem Sie es mithilfe des Eigenschaftenfensters (obere Aufklappliste) selektieren. Sobald das Steuerelement anschließend seinen Fokus wieder verliert, weil Sie ein anderes angeklickt haben, ist es auch schon wieder verschwunden.

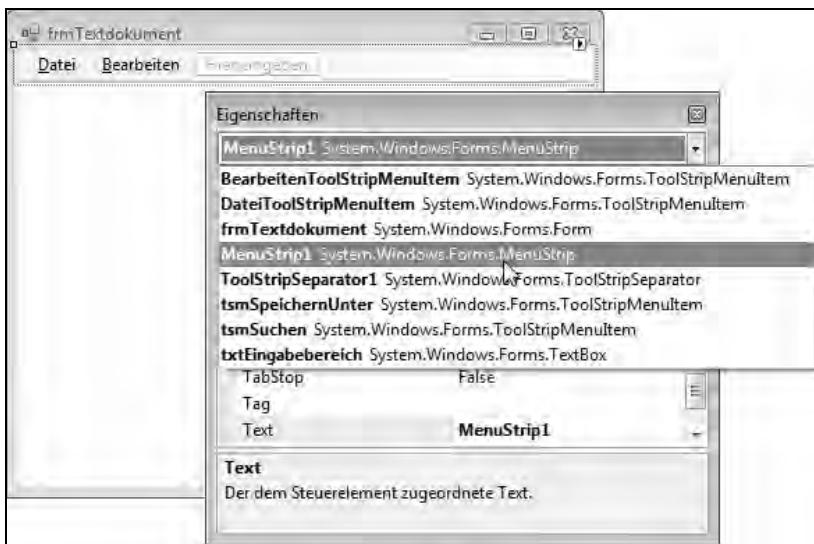


Abbildung 37.26: Ein ToolStrip- oder MenuStrip-Steuerelement, das Sie mit seiner Visible-Eigenschaft unsichtbar machen mussten, erreichen Sie zum Selektieren über das Eigenschaftenfenster

- Die Einstellungen von MergeIndex und MergeAction für das jeweilige Steuerelement im übergeordneten Formular bleiben völlig unberührt.

HINWEIS Viele Entwickler sind der Meinung, dass korrelierende Indexnummern der MergeIndex-Eigenschaften der Steuerelemente im untergeordneten und übergeordneten Fenster die Korrelation bestimmen. Das ist falsch. Die MergeIndex-Nummer des Steuerelements des untergeordneten Formulars bezieht sich immer nur auf die Nummer schon *vorhandener* Elemente von Steuerelementen des übergeordneten Formulars, und nur auf *gleicher Ebene*!

Und das bedeutet: Wenn das Menü des untergeordneten Steuerelements die Einträge:

- *Datei*
- *Datei / Trennlinie*
- *Datei / Speichern*
- *Bearbeiten*
- *Bearbeiten / Suche*

aufweist, und das übergeordnete Formular bereits die Menüeinträge

- *Datei (0)*
- *Datei / Neues Dokument (0) (0)*
- *Datei / Dokument öffnen (0) (1)*
- *Datei / Trennlinie (0) (2)*
- *Datei / Programm beenden (0) (3)*
- *Extras (1)*
- *Extras / Optionen (1) (0)* enthält,

so ist die MergeAction-Eigenschaft für *Datei* des untergeordneten Steuerelements auf MergeOnly und die MergeIndex-Eigenschaft auf 0 zu setzen. Der erste Menüeintrag auf dieser Ebene (*Datei*) im untergeordneten Formular soll nämlich nur mit dem ersten Menüeintrag gleicher Ebene im übergeordneten Formular verschmolzen werden.

Für *Datei/Trennlinie* und *Datei/Speichern* des untergeordneten Formulars setzen Sie die MergeAction-Eigenschaft auf Insert, weil diese Menüpunkte im Menü des übergeordneten Formulars eingefügt werden sollen, und zwar hinter *Datei/Dokument öffnen* und vor *Datei/Trennlinie*. Da bei Insert immer vor einem Element eingefügt wird, und oben angegebene Nummerierung im übergeordneten Formular gilt, bestimmen Sie für *Datei/Trennlinie* den Wert 2 als MergeIndex für *Datei speichern*.

Für *Bearbeiten* bestimmen Sie aus gleichen Gründen 1 für MergeIndex und Insert für MergeAction, weil das komplette *Bearbeiten*-Menü dann vor dem *Extras*-Menü eingefügt wird.

Und damit ist die Einrichtung der Benutzeroberfläche auch schon vollbracht! Sie können sich nun um die programmtechnische Umsetzung der eigentlichen Funktionen kümmern. Die ist in unserem Beispiel vergleichsweise simpel, wie die beiden folgenden Listings des Hauptformulars und das des Formulars zeigen, das die Dokumentenklasse in der Beispielanwendung ausmacht.

Listing von frmMain.vb (übergeordnetes Hauptformular)

```
Public Class frmMain

    'Der "Unbenannt"-Dateinamenzähler.
    Private myUnbekanntesDokumentZähler As Integer

    Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
        MyBase.OnLoad(e)

        'Den "Unbenannt"-Dateinamenzähler beim Laden des Formulars initialisieren.
    End Sub
End Class
```

```
myUnbekanntesDokumentZähler = 1
End Sub

'Wird aufgerufen, wenn der Anwender aus dem Menü Datei den Menüpunkt Neues Dokument auswählt.
Private Sub NeuesDokument_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles tsmNeuesDokument.Click
    'Neue Instanz des MDI-Dokuments erstellen.
    Dim locFrmTextdokument As New frmTextdokument

    'Dieser Instanz mitteilen, dass es ein MDI-untergeordnetes Fenster werden soll.
    locFrmTextdokument.MdiParent = Me

    'Die Funktion zum Anzeigen des neuen untergeordneten Fensters aufrufen.
    locFrmTextdokument.NeuesTextdokument(myUnbekanntesDokumentZähler)

    'Den "Unbenannt"-Dateinamenzähler um eins erhöhen.
    myUnbekanntesDokumentZähler += 1
End Sub

'Wird aufgerufen, wenn der Anwender aus dem Menü Datei den Menüpunkt Öffnen auswählt.
Private Sub tsmÖffnen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles tsmÖffnen.Click
    Dim dateiÖffnenDialog As New OpenFileDialog
    With dateiÖffnenDialog
        .CheckFileExists = True
        .CheckPathExists = True
        .DefaultExt = "*.txt"
        .Filter = "Textdateien" & " (" & "*.txt" & ")|" & "*.txt" & "|Alle Dateien (*.*)|*.*"

        'Dateinamen der zu öffnenden Textdatei ermitteln.
        Dim dialogErgebnis As DialogResult = .ShowDialog
        If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
            Exit Sub
        End If

        'Neue Instanz des MDI-Dokuments erstellen.
        Dim locFrmTextdokument As New frmTextdokument

        'Dieser Instanz mitteilen, dass es ein MDI-untergeordnetes Fenster werden soll.
        locFrmTextdokument.MdiParent = Me

        'Die Funktion zum Laden des Textes und Anzeigen des untergeordneten Fensters aufrufen.
        locFrmTextdokument.TextdokumentÖffnen(.FileName)
    End With
End Sub

'Wird aufgerufen, wenn der Anwender aus dem Menü Extras den Menüpunkt Optionen auswählt.
Private Sub tsmOptionen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles tsmOptionen.Click
    MessageBox.Show("Hier für die Optionen-Funktion implementiert werden")
End Sub

'Wird aufgerufen, wenn der Anwender aus dem Menü Datei den Menüpunkt Programm beenden auswählt.
```

```

Private Sub tsmProgrammBeenden_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmProgrammBeenden.Click
    Me.Close()
End Sub

'Wird beim Schließen des Formulars aufgerufen
Protected Overrides Sub OnFormClosing(ByVal e As System.Windows.Forms.FormClosingEventArgs)
    MyBase.OnFormClosing(e)

    'Gibt es untergeordnete Fenster?
    If Me.MdiChildren IsNot Nothing AndAlso Me.MdiChildren.Length > 0 Then
        Dim locFenstertitel As String = ""

        'Die Fenstertitel der untergeordneten Fenster ermitteln
        For Each locForm As Form In Me.MdiChildren
            locFenstertitel &= locForm.Text & vbCrLf
        Next

        'Warnhinweis mit der Liste der noch offenen Fenster.
        Dim locDr As DialogResult = MessageBox.Show("Folgende Fenster sind noch geöffnet:" & _
            vbCrLf & vbCrLf & locFenstertitel & _
            vbCrLf & vbCrLf & _
            "Soll die Anwendung dennoch geschlossen werden?", _
            "Anwendung schließen:", MessageBoxButtons.YesNo, _
            MessageBoxIcon.Question, MessageBoxDefaultButton.Button2)
        If locDr = Windows.Forms.DialogResult.No Then
            'Der Anwender hat Nein gesagt!
            'Das Fenster bleibt offen.
            e.Cancel = True
        End If
    End If
End Sub
End Class

```

Listung von frmTextdokument.vb (untergeordnete Dokumentenklasse)

```

Public Class frmTextdokument
    ''' <summary>
    ''' Zeigt dieses Formular mit einem leeren Textfeld an
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub NeuesTextdokument(ByVal DokumentNr As Integer)
        txtEingabebereich.Text = Nothing

        'Dokumentnamen "Unbenannt_X" im Titel anzeigen.
        Me.Text = "Unbenannt" & DokumentNr

        'Formular nicht-modal anzeigen.
        Me.Show()
    End Sub

    ''' <summary>
    ''' Zeigt dieses Formular an, und lädt eine Textdatei in den Eingabebereich.
    ''' </summary>
    ''' <param name="Dateiname"></param>

```

```
''' <remarks></remarks>
Public Sub TextdokumentÖffnen(ByVal Dateiname As String)

    'Textdatei in den Eingabebereich laden.
    txtEingabebereich.Text = My.Computer.FileSystem.ReadAllText(Dateiname)

    'Dateinamen als Dokumentnamen im Titel anzeigen.
    Me.Text = Dateiname

    'Formular nicht-modal anzeigen.
    Me.Show()
End Sub

'Wird aufgerufen, wenn der Anwender aus dem Menü Datei den Menüpunkt Speichern unter auswählt.
Private Sub tsmSpeichernUnter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmSpeichernUnter.Click
    Dim dateiSpeichernDialog As New SaveFileDialog
    With dateiSpeichernDialog

        'Pfad muss vorhanden sein!
        .CheckPathExists = True

        'Warnen, wenn die Datei schon existiert!
        .OverwritePrompt = True

        'Dateinamenerweiterungen einrichten:
        .DefaultExt = "*.txt"
        .Filter = "Textdateien" & " (" & "*.txt" & ")|" & "*.txt" & "|Alle Dateien (*.*)|*.*"
        Dim dialogErgebnis As DialogResult = .ShowDialog
        If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
            Exit Sub
        End If

        'Textdatei speichern
        My.Computer.FileSystem.WriteAllText(.FileName, txtEingabebereich.Text, False)

        'Fenstertitel ist Dateiname
        Me.Text = .FileName
    End With
End Sub

'Wird aufgerufen, wenn der Anwender im Menü Bearbeiten auf Suchen klickt.
Private Sub tsmSuchen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmSuchen.Click
    MessageBox.Show("Hier würde die Suchefunktion implementiert werden!")
End Sub
End Class
```

Vererben von Formularen

Da Formulare nichts anderes sind als Klassen, können Sie sie natürlich auch vererben. Im Prinzip machen Sie das automatisch, sobald Sie eine neue Windows Forms-Anwendung anlegen: Das Formular, das Ihrer Anwendung automatisch hinzugefügt wird, erbt aus der Basisklasse *Form*.

Allerdings gibt es eine besondere Möglichkeit, Formulare zu vererben; in Visual Studio .NET spricht man dabei von der visuellen Vererbung von Formularen. Die IDE bezeichnet solche Formulare schlicht als *geerbte Formulare*.

BEGLEITDATEIEN Wenn Sie, anstatt die folgenden Schritte selbst durchzuführen, lieber auf das fertige Projekt zurückgreifen wollen: Sie finden Sie es unter:

...\\VB 2008 Entwicklerbuch\\ G - WinForms\\Kapitel 37\\FormVererbung

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Die Vorgehensweise dazu ist denkbar einfach:

- Erstellen Sie ein neues *Windows Forms*-Projekt, beispielsweise unter dem Namen *FormVererbung*.
- Platzieren Sie ein paar *TextBox*-Steuerelemente auf dem Formular, etwa wie in Abbildung 37.27 zu sehen.

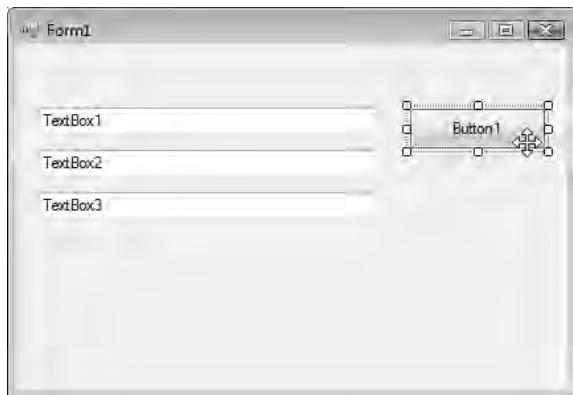


Abbildung 37.27 Dieses Formular dient als Basis für die visuelle Vererbung

- Doppelklicken Sie auf die Schaltfläche *OK*, um den Codeeditor zu öffnen, und fügen Sie den folgenden Programmcode ein:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles Button1.Click

    Dim locString As String

    For Each locControl As Control In Me.Controls
        If TypeOf locControl Is TextBox Then
            locString += "Inhalt von " + locControl.Name + ": "
            locString += DirectCast(locControl, TextBox).Text + vbNewLine
        End If
    Next
End Sub
```

```

    End If
Next
locString = "Inhalt der TextBox-Komponenten im Formular:" + _
            vbCrLf + vbCrLf + locString
MessageBox.Show(locString, "Hinweis:")
End Sub

```

- Starten Sie das Programm anschließend, geben Sie ein paar Zeichen in das TextBox-Steuerelement ein, und beobachten Sie, welche Aktion ein Mausklick auf OK anschließend auslöst (siehe Abbildung 37.28).

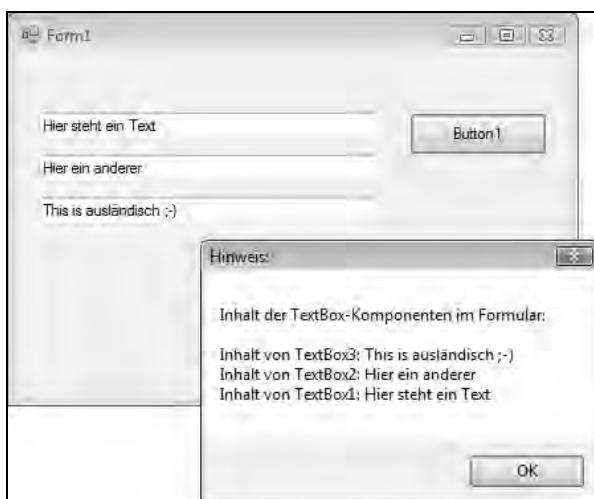


Abbildung 37.28 Formulare kommen übrigens auch ohne Steuerelemente-Arrays aus. Das hat zwar nichts mit dem Thema dieses Abschnitts zu tun, ergibt sich aber durch das Demo für die visuelle Vererbung so ganz nebenbei.

ControlCollection vs. Steuerelemente-Array aus VB6

Auch wenn es nichts mit dem eigentlichen Thema dieses Abschnittes zu tun hat, so zeigt dieses Beispielprogramm dennoch auf einfache Weise, wie unnötig Steuerelemente-Arrays in .NET sind und wieso es sie aus diesem Grund auch nicht mehr gibt. Denn auch die `Controls`-Auflistung, die vordergründig dazu dient, das Formular mit Steuerelementen auszustatten, lässt sich wie jede andere Auflistung, die über einen Enumerator verfügt, mit `For/Each` durchlaufen. Anhand des Typen, den Sie wie im Beispiel mit `Type Of` ermitteln können, haben Sie die Möglichkeit, durch eine entsprechende Typkonvertierung auf die gewünschte Eigenschaft des Steuerelements zuzugreifen.

Kleiner Tipp am Rande: Falls Sie weitere, spezifische Informationen in einem Steuerelement abspeichern möchten, die nur der Identifizierung dienen, verwenden Sie die `Tag5`-Eigenschaft, die jedes auf `Control` basierende Steuerelement zur Verfügung stellt. Anders als in Visual Basic 6.0 können Sie in der `Tag`-Eigenschaft nicht nur Strings, sondern beliebige Objekte speichern.

⁵ Engl. Tag (ausgesprochen: »Tähg«), auf deutsch etwa: *Markierung, Kennzeichnung*.

Für die eigentliche visuelle Vererbung dieses Formulars beenden Sie bitte das Programm, indem Sie auf die Schließenschaltfläche des Formulars klicken. Um ein geerbtes Formular dem Projekt hinzuzufügen, verfahren Sie folgendermaßen:

- Fahren Sie mit der Maus in den Projektmappen-Explorer, klicken Sie mit der rechten Maustaste über dem Projektnamen, und wählen Sie aus dem Kontextmenü, das sich jetzt öffnet, den Eintrag *Hinzufügen* und *Geerbtes Formular hinzufügen*.
- Geben Sie im Dialog, der jetzt erscheint, *GeerbtesFormular* als neuen Formularnamen ein.
- Klicken Sie anschließend auf *Öffnen*.

Visual Studio zeigt Ihnen anschließend einen Dialog, wie Sie ihn auch in Abbildung 37.29 sehen können. In diesem Dialog werden alle Formulare der Projektmappe angezeigt, die sie visuell vererben können. Wählen Sie für dieses Beispiel den einzigen vorhandenen Eintrag aus und klicken Sie abschließend auf *OK*.

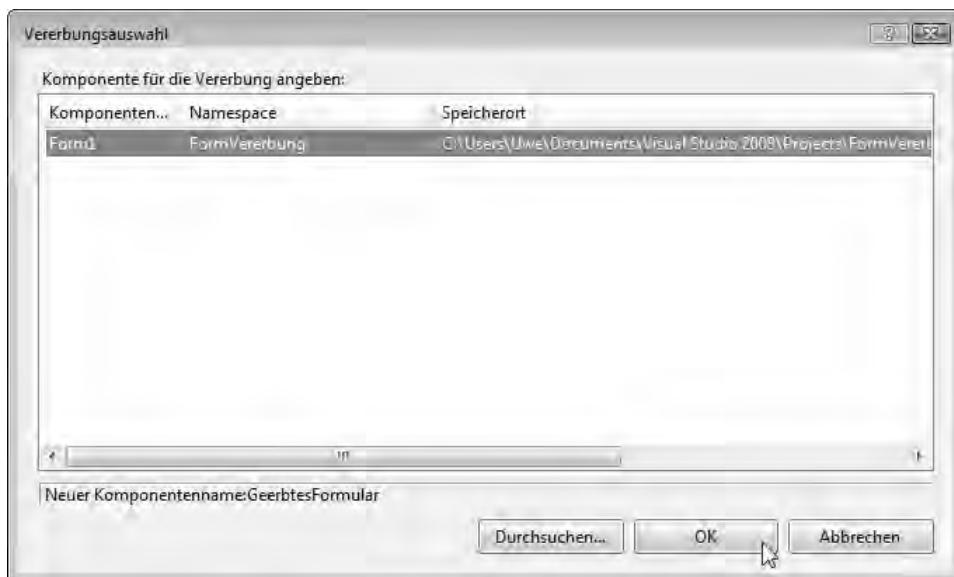


Abbildung 37.29 In dieser Liste, die alle Formulare der Projektmappe zeigt, wählen Sie das Formular, aus dem das neue Formular abgeleitet werden soll

Wenn Sie diesen Vorgang ausgeführt haben, gibt es anschließend einen zweiten Dialog in Ihrer Projektmappe mit dem Namen *GeerbtesFormular.vb*. Klicken Sie ihn doppelt an, um ihn darzustellen und sich davon zu überzeugen, dass er seine äußerliche Ähnlichkeit vom Vater-Dialog geerbt hat (siehe Abbildung 37.30).

Wenn Sie Ihre ersten Experimente mit dem vererbten Steuerelement unternehmen, wird Ihnen eines gleich auffallen: Zwar sind alle Steuerelemente im Formular vorhanden, doch lassen sie sich nicht verändern. Weder können Sie ihre Eigenschaften editieren, noch lassen sie sich umpositionieren oder in ihrer Größe verändern (was letzten Endes auch nichts anderes ist als ein Verändern der Eigenschaften).

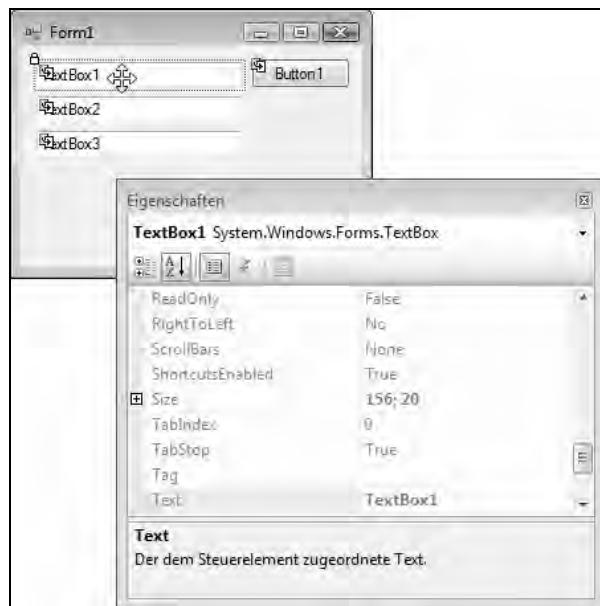


Abbildung 37.30 Die Steuerelemente des Basisdialogs sind zwar alle vorhanden, aber ihre Eigenschaften lassen sich in diesem Zustand nicht verändern, da sie standardmäßig als Friend deklariert, damit nicht überschreibbar und somit aus dem vererbten Formular heraus nicht manipulierbar sind

Fürs Erste behalten Sie diese Erkenntnis einfach nur im Hinterkopf – ich werde später darauf zurückkommen.

Lassen Sie uns vorerst herausfinden, wie wir aus dem Vererben des Formulars nun einen Nutzen ziehen können. Zu diesem Zweck fügen Sie unter den vorhandenen TextBox-Steuerelementen zwei weitere ein – denn das ist durchaus möglich!

Öffnen Sie anschließend die Projekteigenschaften (rechte Maustaste für das Kontextmenü im Projektmappen-Explorer über dem Projektnamen), und wählen Sie als Startobjekt das neue, vererbte Formular *GeerbtesFormular* aus. Starten Sie das Programm anschließend.

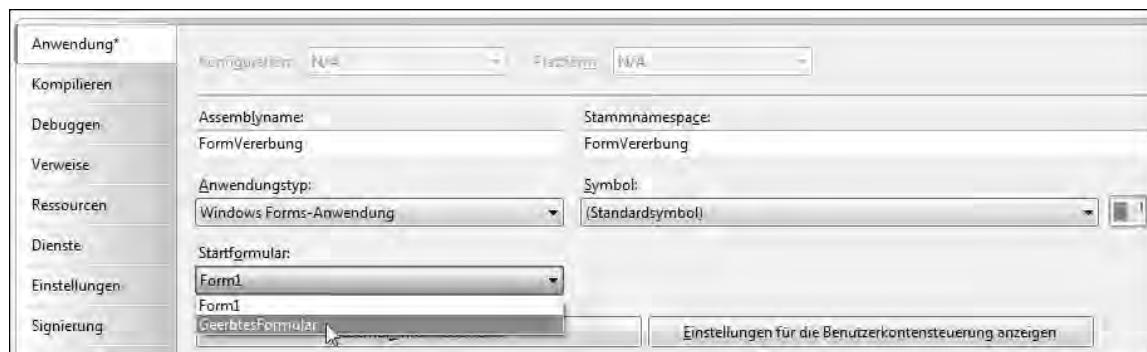


Abbildung 37.31 Legen Sie hier das geerbte Formular als neues Startformular fest

Geben Sie in alle Eingabefelder einen beliebigen Text ein und beobachten Sie, was geschieht, wenn Sie auf die Schaltfläche klicken (siehe Abbildung 37.32).

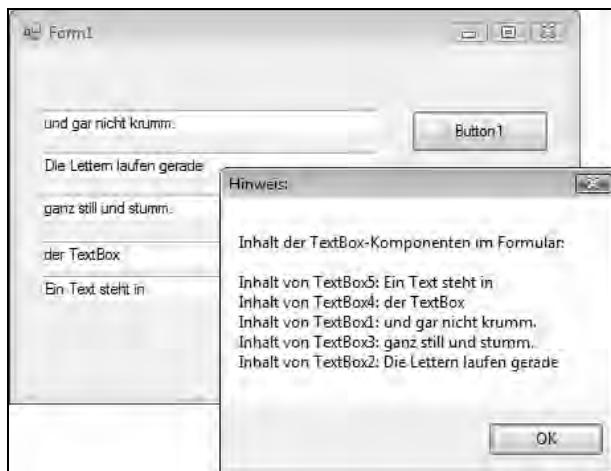


Abbildung 37.32 Keine Zeile Code ist hinzugekommen, und dennoch funktioniert das Programm automatisch auch mit den neu hinzugefügten Komponenten im abgeleiteten Formular!

Überrascht? Sie haben nicht eine einzige Zeile am Code geändert und dennoch – dank Polymorphie – hat der ursprünglich in *Form1* implementierte Code auch in dieser Ableitung seine volle Gültigkeit.

Ein erster Blick auf den Designer-Code eines Formulars

Vielleicht sind Sie sogar ein weiteres Mal überrascht, wenn Sie sich jetzt, nachdem Sie das Programm beendet haben, auf die Suche nach dem Code in der abgeleiteten Klasse begeben. Wechseln Sie mit dem Projektmappen-Explorer in die Codeansicht von *GeerbtesFormular.vb*, und bestaunen Sie, was der Visual Studio-Designer aus dem Code gemacht hat:

```
Public Class GeerbtesFormular
End Class
```

Nichts, absolut gar nichts an zusätzlichem Code ist hier zu sehen. Im Übrigen nicht einmal der Hinweis auf eine Vererbung?!

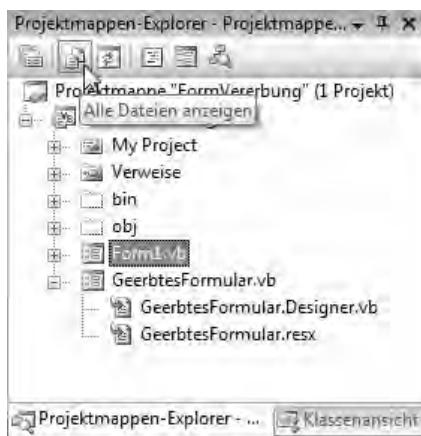


Abbildung 37.33 Nur mit Klick auf dieses Symbol können Sie alle Dateien eines Projektes anzeigen lassen. Bei mehreren Projekten in einer Projektmappe müssen Sie diese Einstellung für jedes Projekt einer Projektmappe wiederholen.

Des Rätsels Lösung liegt darin, dass der Designer den Designercode in Visual Basic 2008 in einer besonderen Codedatei versteckt, die Sie normalerweise gar nicht zu Gesicht bekommen.

Um Sie dennoch zu sehen, klicken Sie im Projektmappenexplorer auf das Symbol *Alle Dateien anzeigen* (siehe Abbildung 37.33). Erst dann können Sie den Zweig vor jedem Formular aufklappen, und Sie werden feststellen, dass jedes in Visual Basic 2008 erstellte Formular eigentlich mindestens aus zwei Dateien besteht. Der Designer-Code, also der Code, der zum Einen vom Designer erstellt wurde und der zum Anderen dafür zuständig ist, dass alle Steuerelemente des Formulars rechtzeitig instanziert und auf dem Formular dargestellt werden, befindet sich in der Datei mit der Endung *.Designer.vb*.

```
<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
Partial Class GeerbtesFormular
    Inherits FormVererbung.Form1

    'Das Formular überschreibt den Löschvorgang, um die Komponentenliste zu bereinigen.
    <System.Diagnostics.DebuggerNonUserCode()> _
    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing AndAlso components IsNot Nothing Then
            components.Dispose()
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Wird vom Windows Form-Designer benötigt.
    Private components As System.ComponentModel.IContainer

    'Hinweis: Die folgende Prozedur ist für den Windows Form-Designer erforderlich.
    'Das Bearbeiten ist mit dem Windows Form-Designer möglich.
    'Das Bearbeiten mit dem Codeeditor ist nicht möglich.
    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()
        Me.TextBox4 = New System.Windows.Forms.TextBox
        Me.TextBox5 = New System.Windows.Forms.TextBox
        Me.SuspendLayout()
        '
        'TextBox4
        '
        Me.TextBox4.Location = New System.Drawing.Point(16, 147)
        Me.TextBox4.Name = "TextBox4"
        Me.TextBox4.Size = New System.Drawing.Size(256, 20)
        Me.TextBox4.TabIndex = 5
        '
        'TextBox5
        '
        Me.TextBox5.Location = New System.Drawing.Point(16, 173)
        Me.TextBox5.Name = "TextBox5"
        Me.TextBox5.Size = New System.Drawing.Size(256, 20)
        Me.TextBox5.TabIndex = 6
        '
        'GeerbtesFormular
        '
        Me.ClientSize = New System.Drawing.Size(416, 262)
        Me.Controls.Add(Me.TextBox5)
        Me.Controls.Add(Me.TextBox4)
    End Sub
End Class
```

```
Me.Name = "GeerbtesFormular"
Me.Controls.SetChildIndex(Me.TextBox4, 0)
Me.Controls.SetChildIndex(Me.TextBox5, 0)
Me.ResumeLayout(False)
Me.PerformLayout()

End Sub
Friend WithEvents TextBox4 As System.Windows.Forms.TextBox
Friend WithEvents TextBox5 As System.Windows.Forms.TextBox

End Class
```

Mithilfe des Konzepts der partiellen Klassen (mehr zum Schlüsselwort `Partial` finden Sie in Kapitel 14) kann der Klassencode den manchmal störenden Designer-Code in einer eigenen Codedatei ablegen. Die eigentliche Vererbungsanweisung befindet sich, wie Sie sich anhand der ersten fett hervorgehobenen Zeilen dieses Listings selbst überzeugen können, ebenfalls in dieser Codedatei.

Ein Formular ist ja nichts weiter als eine ganz normale Klasse, und wenn Sie sie aus einer Basisklasse ableiten, erbt sie alle Eigenschaften dieser – und damit natürlich auch die Ereignisbehandlungsmethode des Click-Ereignisses.

Das abgeleitete Formular verfügt über einen Unterschied zur Basisklasse, den allerdings nicht wir, sondern der Designer als Code in das Listing eingefügt hat:

Es sind dies, wie Sie hier sehen können, die zusätzlichen Definitionen der beiden `TextBox`-Steuerelemente. Was passiert nun genau, wenn Sie das Programm starten und im vererbten Formular auf die Schaltfläche klicken? Die nachstehende Abfolgenbeschreibung macht das deutlich:

- Wenn Sie das Programm starten, wird der Konstruktor der abgeleiteten Klasse aufgerufen. Dieser ruft zunächst den Konstruktor der Basisklasse auf, der seinerseits `InitializeComponent` der Basisklasse aufruft. Damit sind die ursprünglichen drei `TextBox`-Steuerelemente im Formular vorhanden.
- Anschließend ruft der Konstruktor `InitializeComponent` seiner eigenen Klasse auf. Da es für jedes Formular nur eine `Controls`-Auflistung gibt (sie enthält die Steuerelemente eines Formulars und ist aus der Basisklasse entstanden), werden die beiden neuen `TextBox`-Steuerelemente zu den bereits vorhandenen in der Auflistung hinzugefügt.
- Klickt der Anwender zur Laufzeit auf die Schaltfläche, werden alle Texte aus den in der `Controls`-Auflistung vorhandenen `TextBox`-Steuerelementen ausgelesen – dazu gehören jetzt auch die beiden neuen `TextBox`-Steuerelemente.

Das Ergebnis: Die Texte aller fünf Steuerelemente werden im Nachrichtenfeld dargestellt – Polymorphie ist eine feine Sache, finden Sie nicht?

Modifizierer von Steuerelementen in geerbten Formularen

Wenn Sie sich das zuvor abgedruckte Listing des Designer-Codes genau anschauen, dann werden Sie feststellen, dass die Variablen, die die `TextBox`-Steuerelemente instanziieren, mit dem `Friend`-Modifizierer gekennzeichnet sind. Nun eignet sich der `Friend`-Modifizierer genau wie `Protected`, `Public` oder `Protected Friend` gleichermaßen, einer geerbten Klasse in der gleichen Assembly (in der gleichen DLL oder der gleichen EXE-Datei) Zugriff auf derart deklarierte Member der Basisklasse zu verschaffen. Dennoch sieht es so

aus, als wenn dem Designer der Zugriff verwehrt wäre – denn Sie können Steuerelemente, die als Friend deklariert sind, offensichtlich nicht in der abgeleiteten Klasse verändern. Das liegt daran, dass der Designer, der die Klasse zur Darstellung auf seine eigene Weise verwendet, sich in einer anderen Assembly befindet als das Formular, das Sie ableiten: Er instanziert es nicht nur, sondern leitet es auf seiner Basis neu ab – durch den Friend-Zugriffsmodifizierer kann er dann natürlich nicht mehr auf die Eigenschaften der Steuerelemente (die nunmehr versteckte Member der Ursprungsklasse sind) zugreifen. Anders ist das, wenn Sie die Modifiers-Eigenschaft eines Steuerelements in Protected, Protected Friend oder Public ändern. Jetzt kann der Designer auch auf die so definierten Steuerelemente des abgeleiteten Formulars zugreifen, und Sie können die Eigenschaften dieser Steuerelemente verändern und sie damit auch beispielsweise positionstechnisch verändern.

Probieren Sie es aus:

- Klicken Sie auf die Registerkarte *Form1.vb [Entwurf]*, um das Basisformular in der Entwurfsansicht anzeigen zu lassen.
- Klicken Sie die erste TextBox an, und ändern Sie im Eigenschaftenfenster ihre Modifiers-Eigenschaft auf Protected. Sobald Sie diese Änderung vorgenommen haben, zeigt Ihnen die Aufgabenliste einen Hinweis an, etwa wie in Abbildung 37.34 zu sehen.

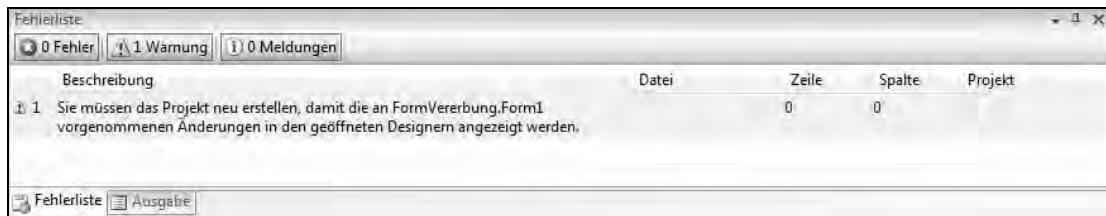


Abbildung 37.34 Wenn Sie Änderungen am Basisformular vorgenommen haben, müssen Sie die Anwendung neu erstellen, damit sich die Änderungen auf die Ableitungen auswirken

- Erstellen Sie die Anwendung neu, indem Sie aus dem Menü *Erstellen* den Menüpunkt *Projektmappe neu erstellen* wählen.
- Lassen Sie anschließend den Designer zum vererbten Form *GeerbtesForm.vb [Entwurf]* anzeigen.

Sie sehen, dass Sie die erste TextBox jetzt nach Belieben verändern können. Sowohl die Position lässt sich beliebig anpassen als auch andere Eigenschaften der TextBox. Der Name des Steuerelements ist natürlich nach wie vor unveränderlich.

Kapitel 38

Im Motorraum von Formularen und Steuerelementen

In diesem Kapitel:

Über das Vererben von Form und die Geheimnisse des Designer-Codes	1098
Ereignisbehandlung von Formularen und Steuerelementen	1104
Wer oder was löst welche Formular- bzw. Steuerelementereignisse wann aus?	1110

Nachdem Sie Ihre ersten Windows Forms-Anwendungen mit Visual Basic erstellt haben, gehen Ihnen bestimmte Prinzipien beim Verdrahten von Benutzeroberflächen mit den eigentlichen Funktionalitäten Ihrer Programme in Fleisch und Blut über. So wissen Sie nach einer Weile einfach, wie Sie dafür zu sorgen haben, dass beim Klicken einer Schaltfläche eine entsprechende Ereignisbehandlungsmethode ausgeführt werden soll.

Doch was passiert eigentlich genau, wenn der Benutzer ein Element bedient? In welcher Reihenfolge treten welche Ereignisse auf, wenn ein Formular oder ein Steuerelement dargestellt wird? Und wie gliedern sich .NET Framework-Windows-Anwendungen in das »alte« Betriebssystem (also vor Windows Vista) eigentlich ein? Dieses Kapitel möchte ein wenig Licht ins Dunkel bringen und auf diese Fragen Antworten geben.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel38\\PictureViewer

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Über das Vererben von Form und die Geheimnisse des Designer-Codes

Immer wenn Sie Ihrem Projekt ein neues Windows-Formular hinzufügen, legt Visual Basic nicht nur eine einfache Klassendatei an, die den Formularaufbau enthält. Die Wahrheit ist: Wenn Sie ein neues Formular anlegen, und es im Codeeditor öffnen, dann sehen Sie zunächst außer dem Klassenrumpf erst einmal gar nichts. Im Beispielprojekt gibt es das Formular *LeeresFormular.vb*, das, wenn Sie es im Codeeditor öffnen, den folgenden Code bereits hält:

```
Public Class LeeresFormular  
End Class
```

Öffnen Sie das Formular jedoch mit dem Designer, stellen Sie fest, dass es bereits eine ganze Reihe von Steuerelementen gibt. Nur: Wo ist denn nur die Information für diese Steuerelemente untergebracht?

Gerade für Visual Basic .NET-Umsteiger, die ihre ersten Gehversuche mit Visual Basic 2008 und Windows Forms-Anwendungen machen, sieht diese Tatsache auf den ersten Blick geradezu erschreckend aus, weil die für .NET-Programmiersprachen postulierte »OOP-Konsequenz« gebrochen zu sein scheint.

Es gibt weder einen Hinweis darauf, dass die Grundfunktionalität des Formulars sich aus irgendeiner im .NET Framework schon vorhandenen Klasse ableitet, noch, an welcher Stelle die Platzierung der Steuerelemente im Formular zur Laufzeit erfolgt.

Wenn Sie das vorherige Kapitel aufmerksam studiert haben, wissen Sie es bereits:

Des Rätsels Lösung liegt darin, dass der spezielle Code zur eigentlichen Darstellung der Steuerelemente eines jeden Formulars in Visual Basic 2008 in einer besonderen Codedatei versteckt ist, die Sie normalerweise gar nicht zu Gesicht bekommen.

Um Sie dennoch zu sehen, klicken Sie im Projektmappenexplorer auf das Symbol *Alle Dateien anzeigen* (siehe Abbildung 38.1).

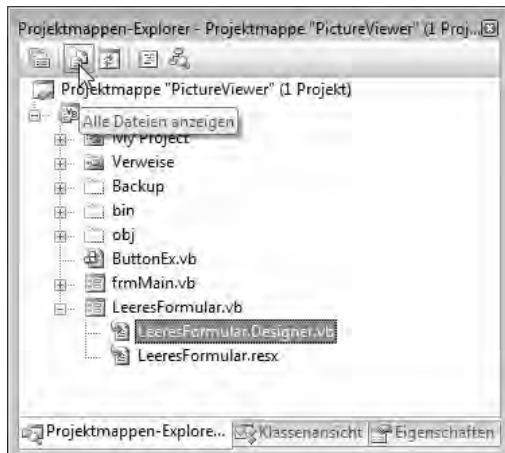


Abbildung 38.1 Nur mit Klick auf dieses Symbol können Sie alle Dateien eines Projektes anzeigen lassen, und damit auch Zugriff auf den Designer-Code eines Formulars nehmen. Bei mehreren Projekten in einer Projektmappe müssen Sie diese Einstellung für jedes Projekt einer Projektmappe wiederholen.

Erst dann können Sie den Zweig vor jedem Formular aufklappen, und Sie werden feststellen, dass jedes in Visual Basic 2008 erstellte Formular eigentlich mindestens aus zwei Dateien besteht. Der Designer-Code, also der Code, der zum Einen vom Designer erstellt wurde und der zum Anderen dafür zuständig ist, dass alle Steuerelemente des Formulars rechtzeitig instanziert und auf dem Formular dargestellt werden, befindet sich in der Datei mit der Endung *.Designer.vb*.

Und ohne dass der Entwickler im Formular *LeeresFormular.vb* nur eine einzige Zeile selbst programmiert hat, beträgt der Umfang Ihrer Formularklasse zu diesem Zeitpunkt schon weit über 100 Zeilen, denn:

In der Datei *LeeresFormular.Designer.vb* befinden sich die Codezeilen, die der Windows-Designer (eigentlich: *die* Designer, denn jedes Steuerelement verfügt streng genommen über seinen eigenen – siehe dazu den entsprechenden grauen Kasten in Kapitel 5) beim Anlegen des Projektes und bei jedem Hinzufügen der Komponenten und dem dazugehörigen Einstellen der Eigenschaften produziert hat.

Auch die Angabe der Klassenableitung mit *Inherits* steht in diesem Designer-Teil des Formularcodes – und mit dieser Technik wird zweierlei erreicht:

- Das OOP-Konzept für Framework-Klassen ist auch in Visual Basic .NET nicht verletzt.
- Dennoch ist die Codedatei, in der Sie Ihren eigenen programmtechnischen Teil zum Funktionieren des Formulars beitragen, sauber aufgeräumt und bleibt stets übersichtlich. Der Designer-Code steht buchstäblich völlig außen vor, und sie bekommen ihn nur zu Gesicht, wenn Sie es ausdrücklich wollen.

Es ist aber dennoch wichtig, dass Sie verstehen, was beim Anzeigen eines Formulars zur Laufzeit passiert, denn nur dann haben Sie die Möglichkeit, ins Geschehen einzugreifen, wenn mal etwas nicht so funktioniert, wie Sie es sich gedacht haben, oder wenn Sie Erweiterungen implementieren müssen, die über die Standardimplementierung hinausgehen.

Die folgenden Abschnitte beschreiben daher die Funktionsweise des Designer-Codes am Beispiel.

Geburt und Tod eines Formulars – New und Dispose

Wenn Sie Ihrem Formular ein neues Formular hinzufügen, dann bekommt es automatisch vom Designer eine Dispose-Methode verpasst.

Was in Visual Basic 2008 erstellten Formularen allerdings völlig fehlt, ist ein Konstruktor, und – nehmen wir die Tatsache vorweg, dass gerade der fehlende Konstruktor dafür zuständig ist, die Methode InitializeComponent aufzurufen, mit dem das eigentliche Einrichten und Platzieren der Steuerelemente des Formulars erfolgt – es stellt sich die Frage, wer oder was ruft InitializeComponent überhaupt auf?

Eine offensichtliche Antwort könnte lauten: Der Konstruktor der Basisklasse, denn der wird, wie Sie wissen, wenn Sie den Klassenteil dieses Buchs aufmerksam studiert haben, in vererbten Klassen implizit aufgerufen. Doch das ist nicht der Fall.

Die Wahrheit ist: Der Visual Basic-Compiler kompiliert »virtuellen« Code beim Kompilieren in die Klasse hinein. Würden Sie das Kompilat des Beispielprogramms mit einem geeigneten Tool wieder zurück in seinen Visual Basic-Quelltext übersetzen, dann ergäbe sich nämlich Folgendes:

```
<DebuggerNonUserCode> _
Public Sub New()
    'Diese Zeile hilft beim schnellen Wiederauferstehenlassen des Formulars,
    'falls es doch nochmal benötigt werden sollte.
    frmMain.ENCLIST.Add(New WeakReference(Me))
    'Hier wird InitializeComponent aufgerufen
    Me.InitializeComponent
End Sub
```

Dieses Verhalten wird auch dadurch unterstrichen, dass etwas Außergewöhnliches passiert, wenn Sie in der eigentlichen Formulkarklasse einen parameterlosen Konstruktor einfügen. Der Editor zaubert nämlich aus der einfachen Eingabe von **Sub New** einen Coderumpf, wie Sie ihn auch in Abbildung 38.2 sehen können.

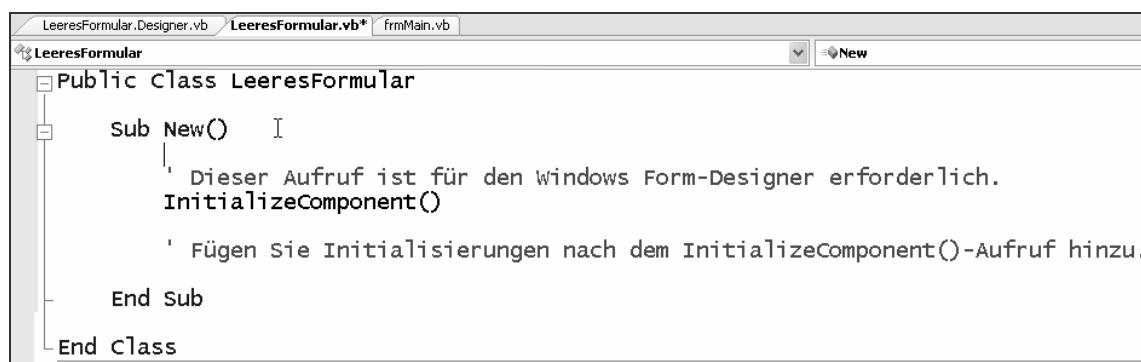


Abbildung 38.2 Wenn Sie einen Konstruktor durch die Eingabe von Sub New im Formularcode (nicht Designer-Code!) einfügen, ergänzt der Editor den Code um einen wesentlichen Aufruf

Sub New des Formulars

Egal, wo also die Sub New des Formulars also letzten Endes herkommt – sie gestaltet sich grundsätzlich folgendermaßen:

```

Public Sub New()
    MyBase.New()

    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()

    ' Initialisierungen nach dem Aufruf InitializeComponent() hinzufügen.
End Sub

```

Der Konstruktor – repräsentiert durch Sub New – ist existenziell für das saubere Funktionieren des Formulars. Der Formularkonstruktor wird nämlich nicht nur durch den Designer aufgerufen, wenn Sie das Formular zur Entwurfszeit bearbeiten, er wird natürlich auch vom Anwendungsframework benötigt, das das Formular instanziert und darstellt, wenn das Formular das Startformular der Anwendung ist.

InitializeComponent des Formulars

Der Konstruktor des Formulars ruft anschließend die Prozedur InitializeComponent auf. Und hier wird es interessant, denn InitializeComponent erledigt den Job, das Formular mit sinnvollen Inhalten zu füllen.

Um das zu tun, müssen die Komponenten und Steuerelemente, mit denen der Anwender später das Formular bedient, für den Gebrauch im Formular vorbereitet werden. Steuerelemente und Komponenten sind wie alle anderen Elemente in .NET Objekte. Und Objekte entstehen aus Klassen. Das heißt: Für jedes Steuerelement, das Sie zur Entwurfszeit auf das Formular gezogen haben, hat der Designer eine Objektvariable bereitgestellt. Die Deklaration dieser Objektvariablen finden Sie im Beispiel ganz am Ende des Designer-Klassencodes:

```

Friend WithEvents picViewArea As System.Windows.Forms.PictureBox
Friend WithEvents btnNextBitmap As PictureViewer.ButtonEx
Friend WithEvents btnOpenBitmap As PictureViewer.ButtonEx
Friend WithEvents btnQuitProgram As System.Windows.Forms.Button
Friend WithEvents btnSaveBitmap As System.Windows.Forms.Button
Friend WithEvents pnlPicture As System.Windows.Forms.Panel

```

Sie werden feststellen, dass die Namensgebung der Objektvariablen genau der entspricht, die Sie im Eigenschaftenfenster mit der Name-Eigenschaft vorgenommen haben. Sie stellen aber ebenfalls fest, dass die Komponentenvariablen mit dem Schlüsselwort WithEvents deklariert wurden. Diese Deklaration zeigt an, dass es über die jeweilige Objektvariable möglich ist, Ereignisse zu empfangen, die an eine zum Ereignis kompatible Prozedur delegiert werden kann. Mehr zu diesem Thema finden Sie übrigens in Kapitel 20.

Schauen wir uns als nächstes den ersten Teil von InitializeComponent genauer an:

```

'Hinweis: Die folgende Prozedur ist für den Windows Form-Designer erforderlich.
'Das Bearbeiten ist mit dem Windows Form-Designer möglich.
'Das Bearbeiten mit dem Codeeditor ist nicht möglich.
<System.Diagnostics.DebuggerStepThrough()> _
Private Sub InitializeComponent()
    Me.picViewArea = New System.Windows.Forms.PictureBox
    Me.btnNextBitmap = New PictureViewer.ButtonEx
    Me.btnOpenBitmap = New PictureViewer.ButtonEx
    Me.btnExitProgram = New System.Windows.Forms.Button
    Me.btnSaveBitmap = New System.Windows.Forms.Button
    Me.pnlPicture = New System.Windows.Forms.Panel

```

Alle innerhalb des Formulars verwendeten Komponenten werden in diesem Teil instanziert. Das Attribut DebuggerStepThrough¹ gibt dem Debugger an, dass er nicht in diesen Teil des Programms schrittweise eintreten darf (auch, wenn Sie das Programm im Einzelschrittmodus debuggen und er dies eigentlich sollte).

Aussetzen der Layout-Logik – SuspendLayout und ResumeLayout

Um die nächsten beiden Codezeilen ranken sich in der Entwicklergemeinde mehr Mythen als Wahrheiten. Es geht um die Methode SuspendLayout, und Sie finden sie im besprochenen Beispielcode in den folgenden Codezeilen:

```
Me.pnlPicture.SuspendLayout()
Me.SuspendLayout()
```

Viele Entwickler glauben, dass sie mit SuspendLayout verhindern können, dass ein Steuerelement, das einem Container (das kann ein Formular oder ein Steuerelement sein, das andere Steuerelemente enthält, wie beispielsweise das GroupBox-Steuerelement) zugeordnet ist, gezeichnet wird, während es sich im *Suspend*-Zustand befindet. Das ist falsch. SuspendLayout, angewendet auf einen Container, der andere Steuerelemente enthält, sorgt lediglich dafür, dass das Layout-Ereignis² des Steuerelements nicht ausgelöst wird. Das Layout-Ereignis tritt dann ein, wenn einem Steuerelement, das als Container fungiert, weitere Steuerelemente hinzugefügt werden, wenn Steuerelemente aus ihm entfernt werden oder wenn sich die Begrenzungen eines Steuerelements ändern.

Wieso ist es aber so wichtig, das Layout-Ereignis zu unterdrücken? Aus zweierlei Gründen. Zum Einen ergibt es gerade beim Initialisieren eines Steuerelements aus Geschwindigkeitsgründen keinen Sinn, auf jede Eigenschaft zu reagieren, die das Layout verändert. Es reicht, wenn sich ein Steuerelement an das neue Layout anpasst, wenn alle seine Eigenschaften vollständig gesetzt sind. Zum Anderen kann es gerade beim Initialisieren zum Dilemma kommen, wenn bestimmte Eigenschaften sich gegenseitig beeinflussen, und eine das Layout beeinflussende Eigenschaft, die weiter hinten im Programmcode ausgeführt wird, durch das Layout-Ereignis indirekt eine Eigenschaft verändert, die bereits gesetzt wurde und so Zirkelaufufe entstehen könnten.

Die restlichen Zeilen im InitializeComponent-Code sind übrigens lediglich dafür verantwortlich, die geänderten Eigenschaften für die Steuerelemente so zu setzen, wie Sie sie im Eigenschaftenfenster zur Entwurfszeit definiert haben. Teilweise holpert der Code dabei ein wenig, wie beispielsweise beim Setzen der *Anchor*-Eigenschaft, dessen umständliche Formulierung

```
Me.pnlPicture.Anchor = CType(((System.Windows.Forms.AnchorStyles.Top Or
                               System.Windows.Forms.AnchorStyles.Bottom) _
                               Or System.Windows.Forms.AnchorStyles.Left) _
                               Or System.Windows.Forms.AnchorStyles.Right),
System.Windows.Forms.AnchorStyles)
```

¹ Sinngemäß übersetzt: »Debugger, überspring dies«.

² Mehr Informationen zu Ereignissen erfahren Sie zu einem späteren Zeitpunkt in diesem Kapitel. Fürs Erste reicht es zu wissen, dass Ereignisse dazu da sind, automatisch bestimmte Prozeduren auszuführen, wenn ein bestimmter Zustand eintritt. Die dem Click-Ereignis zugewiesene Prozedur beispielsweise wird also dann automatisch aufgerufen, wenn der Zustand »Anwender hat Schaltfläche angeklickt« eintritt.

auch einfach nur mit der Zeile

```
Me.pnlPicture.Anchor = AnchorStyles.Top Or AnchorStyles.Left Or AnchorStyles.Right Or AnchorStyles.Top
```

funktionieren würde; aber natürlich wurde diese Zeile nicht von Menschenhand, sondern durch eine Maschine erzeugt. Und vermutlich, um allgemein gültige Algorithmen bei der Codegenerierung einsetzen zu können, gibt es hier und da schon einmal Typ-Castings, wo keine nötig wären, aber offensichtlich schadet es auch nicht.

Steuerelemente auf dem Formular mit der ControlCollection sichtbar machen

Ungleich interessanter sind die letzten Zeilen von InitializeComponent, die dafür sorgen, dass die zu dieser Zeit bereits instanzierten Steuerelemente auch auf dem Formular sichtbar werden. Jedes von der Control-Klasse abgeleitete Objekt – und dazu gehört auch Form – verfügt über eine so genannte **Controls-Auflistung**, die die Steuerelemente enthält, die dieses als Container fungierende Steuerelement enthält. Ein instanziiertes Formular ist nichts weiter als ein erweitertes Control-Objekt, also gilt das für ein Formular gleichermaßen. In dem Moment, in dem ein Objekt mit einer instanzierten Control-Klasse (oder einer von Control abgeleiteten) der **Controls-Auflistung** einem Control-Objekt mit Add hinzugefügt wurde, wird das Steuerelement auch im Container (im Beispiel also dem Formular) sichtbar. Voraussetzung dafür ist natürlich, dass es sich um eine sichtbare Komponente handelt und dessen **Visible**-Eigenschaft auf **True** gesetzt wurde. Mal abgesehen von den nicht in diesen Zusammenhang passenden Zeilen

```
Me.AutoScaleDimensions = New System.Drawing.SizeF(6.0!, 13.0!)
Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font
Me.ClientSize = New System.Drawing.Size(582, 431)
```

die kurz vorher noch die Größeneinstellungen des Formulars vornehmen, dienen die folgenden Codezeilen schließlich dazu, den Komponenten die Möglichkeit zu geben, im Formular sichtbar zu werden.

```
Me.Controls.Add(Me.btnExitBitmap)
Me.Controls.Add(Me.btnOpenBitmap)
Me.Controls.Add(Me.btnExitProgram)
Me.Controls.Add(Me.btnSaveBitmap)
Me.Controls.Add(Me.pnlPicture)
Me.Name = "LeeresFormular"
Me.Text = "LeeresFormular"
Me.pnlPicture.ResumeLayout(False)
Me.ResumeLayout(False)
```

Die beiden letzten Zeilen schließlich sorgen dafür, dass die Layout-Ereignisse für Panel und das gesamte Formular wieder in der gewohnten Weise stattfinden können.

Um das Verhalten zu verdeutlichen: Wenn Sie der **Controls-Auflistung** eines Formulars eine Instanz einer TextBox-Komponente mit Add hinzufügen, ist die TextBox (entsprechend eingestellte Eigenschaften vorausgesetzt) direkt nach dem Ausführen der Add-Methode im Formular sichtbar und einsatzbereit.

Ereignisbehandlung von Formularen und Steuerelementen

Ereignisse haben gerade bei der Programmierung von Windows-Anwendungen eine ganz zentrale Bedeutung. Wann immer der Anwender Ihres Programms im weitesten Sinne »Irgendetwas« macht, löst er damit ein Ereignis aus, auf das Sie reagieren können.



Abbildung 38.3 Mit dem PictureViewer können Sie beliebige Grafiken eines Verzeichnisses betrachten

Das wohl einfachste Beispiel ist der Klick auf eine Schaltfläche, und es mag Ihnen als erfahrenem Entwickler ein wenig überflüssig vorkommen, etwas über die Funktionsweise des *Click*-Ereignisses zu erfahren, doch wissen Sie genau, was unter der Haube passiert?

Dazu werfen wir an dieser Stelle einen ersten Blick auf das Beispielprogramm selbst, indem wir es starten. Nach dem Start des Programms sehen Sie einen Dialog auf dem Bildschirm, wie er auch in Abbildung 38.3 zu erkennen ist.

Auf den ersten Blick scheint es überflüssig zu sein, die Funktionen dieses Beispiels zu erklären, schließlich wirkt jede Funktion so offensichtlich!

Doch eine Besonderheit hat das Programm, von der Sie Notiz nehmen können, wenn Sie mit dem Mauszeiger über eine der Schaltflächen *Grafik öffnen* oder *Nächste Grafik* fahren. Achten Sie dazu auf das, was im Ausgabefenster von Visual Studio angezeigt wird.

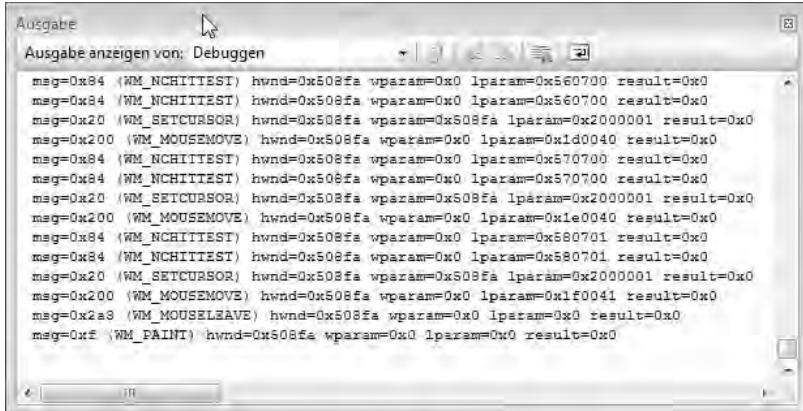


Abbildung 38.4 Wenn Sie eine der beiden Schaltflächen Grafik öffnen oder Nächste Grafik mit der Maus berühren, über sie hinwegfahren und diese betätigen, zeigt das Ausgabefenster die nativen Nachrichten der Windows-Nachrichtenschlange an

Um zu verstehen, was genau passiert, und dieses Verständnis für die funktionelle Erweiterung von Steuerelementen zu nutzen, ist allerdings ein detailliertes Verständnis für die internen Abläufe bei auf Windows-Nachrichten basierenden Ereignissen von Steuerelementen erforderlich. Der folgende Abschnitt liefert diese Grundlagen, doch nehmen Sie sich für seine Lektüre ein wenig Zeit: Er hat es nämlich »in sich«!

Vom Programmstart mithilfe des Anwendungsframeworks über eine Benutzeraktion zur Ereignisauslösung

Solange Sie nichts anderes sagen, verwenden Windows Forms-Anwendungen das Anwendungsframework, um ein Windows-Programm ans Laufen zu bekommen. Ganz vereinfacht ausgedrückt, startet dabei zunächst eigentlich nicht Ihr eigenes Programm, sondern ein völlig anderer Prozess, der mit Ihrem Programm noch nichts zu tun hat. Das ist notwendig, damit eine automatisierte Aktualisierung einer SmartClient-Anwendung (über das Thema ClickOnce-Deployment, das hier den Rahmen weit sprengen würde, hält die Online-Hilfe Genaueres bereit) überhaupt funktionieren kann – denn Programmdateien können nicht ausgetauscht werden, wenn das Programm, dessen Dateien ausgetauscht werden müssen, bereits läuft.

Dieser Prozess ist übrigens auch dafür zuständig, einen eventuell darzustellenden Begrüßungsdialog anzuzeigen (den Sie ganz einfach in den Projekteinstellungen definieren können) und schließlich auf den eigentlichen Prozess Ihrer Anwendung zu initiieren und eine so genannte Windows-Nachrichtenwarteschlange für das Startformular Ihrer Anwendung einzurichten.

Diese Warteschlange ist, stark vereinfacht ausgedrückt, eine Endlosschleife, die zunächst nichts weiter macht, als zu überprüfen, ob es irgendwelche neuen Meldungen vom Windows-System gibt, die an den Thread gerichtet sind, in dem sie ausgeführt werden.

Das Hauptformular spielt in der Warteschlange dabei eigentlich gar keine besondere Rolle – jedenfalls was die Verarbeitung der Warteschlange an sich anbelangt. Es wird mit seiner Dispose-Methode nur an den so genannten *Application Context* (etwa: Anwendungskontext) »gebunden«, und das bewirkt, dass die Warteschlangenroutine des laufenden Prozesses (und damit letzten Endes auch die Anwendung) beendet wird, wenn das Formular geschlossen wird.

Der Schlüssel der ganzen Ereignissesteuerung für Windows-Nachrichten liegt eigentlich in dem Zusammenspiel zwischen der `Control`-Klasse und einer intern verwendeten Klasse, die sich `NativeWindow` nennt – und damit sind wir mitten im Thema, nämlich was passiert, wenn der Anwender beispielsweise eine Schaltfläche (die ja eine wenn auch abgeleitete `Control`-Klasse darstellt) betätigt.

Wichtig ist es zunächst einmal zu wissen, dass jedes noch so kleine Element im Windows-Betriebssystem (Schaltfläche, `PictureBox` – sogar ein angezeigter `Tooltip`) ein Window darstellt, genauso wie ein Fenster, dass Sie selbst erst als Window bezeichnen würden. Der Ereignisverlauf von einem richtigen »Windows« ist also prinzipiell kein anderer als der einer Schaltfläche, die ja schließlich auch ein Window im Windows-Betriebssystemsinne darstellt.

Schon vor .NET gab es OOP und wenn Sie sich einmal die MFC (Microsoft Foundation Class) unter C++ anschauen (Windows als Betriebssystem wurde fast vollständig in C und C++ geschrieben), sehen Sie, dass ein Button wirklich ein Fenster IST – es ist nämlich eine abgeleitete Klasse, die Sie auch wieder zu einem Window downcasten können.

Sobald die `Visible`-Eigenschaft eines `Control`-Objekts oder eines von `Control` abgeleiteten Objekts auf `True` gesetzt wird und damit das erste Mal die Notwendigkeit besteht, ein Window im Sinne vom Windows-Betriebssystem ins Leben zu rufen, legt die `Control`-Klasse eine Member-Variable auf Basis der Klasse `NativeWindow` (etwa: »grundlegendes Windows«) an. `NativeWindow` wird dabei die eigene Instanz von `Control` übergeben – es gibt damit einen Zirkelverweis zwischen der `Control`-Instanz (unserer Schaltfläche `Grafik öffnen`, um beim Beispiel zu bleiben) und ihrem `NativeWindow`-Member. Zirkelverweis in diesem Zusammenhang bedeutet: Das `Control`-Objekt kann nicht nur auf die `NativeWindow`-Instanz zugreifen, sondern die `NativeWindow`-Instanz weiß auch, zu welchem `Control`-Objekt sie gehört.

`NativeWindow` ist deswegen so wichtig für `Control`, weil es die Schnittstelle zur Warteschlange bildet und zwar auf eine ganz raffinierte Art und Weise: `NativeWindow` selbst erstellt bei seiner Instanziierung ein Objekt der so genannten `WindowClass`-Klasse. Diese Klasse können Sie selbst nicht verwenden, sie steht nur dem Framework zur Verfügung und bildet eine Art Verwalter zwischen dem Windows-Subsystem und dem darüber liegenden .NET Framework. Wenn eine `WindowClass`-Klasse instanziert wird, dann zu dem Zweck, ein Window im Windows-Betriebssystem-Sinne zu erstellen. Dazu legt sie zunächst durch spezielle Betriebssystemaufrufe ein so genanntes *Window Handle*³ an (jede Schaltfläche, jedes Fenster, jede Liste – ja sogar jeder Auswahlbereich *unterhalb* einer aufklappbaren Liste unter Windows ist, wie schon gesagt, ein Window im Betriebssystem-Sinne). Anschließend trägt sie sowohl dieses Window Handle als auch das `NativeWindow`-Objekt (das sie kennt, da es ihr als Konstruktorparameter übergeben wurde) in eine Tabelle ein. Auf diese Tabelle kann die Warteschlange Zugriff nehmen, die vom Anwendungsframework (oder, für Puristen, mit der Methode `Application.Run`) beim Start der Applikation eingerichtet worden ist.

Irgendwann bekommt die Warteschlange eine für sie bestimmte Nachricht, beispielsweise wenn der Anwender unseres Beispielprogramms auf die Schaltfläche `Grafik öffnen` klickt. Sie prüft nun, welches spezielle Window Handle in der Nachricht als Kennung gespeichert ist und durchforstet die Tabelle mit der Zuordnung Window Handle/`NativeWindow`-Instanz nach diesem. Auf diese Weise findet sie das entsprechende `NativeWindow`-Objekt, ruft dessen `Callback`⁴-Funktion auf und übergibt der Funktion dabei die Nachricht.

³ Eine eindeutige ID, die jedes Windows-Element unter Windows zugewiesen bekommt, wenn es ins Leben gerufen wird.

⁴ Es ist nicht nur eine Callback-Funktion im klassischen C-Sinne, die Funktion heißt tatsächlich so. Kleine Randnotiz: Wenn sich das .NET-Programm im Debug-Modus befindet, ruft sie eine andere Funktion namens `DebuggableCallback` auf.

Callback macht seinerseits wiederum ein so genanntes Message-Objekt daraus, das für die Nachrichtenverarbeitung in .NET verwendet wird, und ruft die `WndProc`-Routine seiner Instanz auf.

Nun wird es Zeit für eine weitere Information, die ich Ihnen bisher verschwiegen habe, um die bisherigen Zusammenhänge nicht zu undurchschaubar werden zu lassen. `Control` arbeitet nämlich eigentlich gar nicht mit einer Instanz von `NativeWindow`, sondern mit einer davon abgeleiteten Klasse namens `ControlNativeWindow`. `ControlNativeWindow` überschreibt die `WndProc`-Routine seiner Basisklasse `NativeWindow` und leitet damit den Aufruf zur `OnMessage`-Funktion um (das Programm befindet sich zu diesem Zeitpunkt immer noch in der `ControlNativeWindow`-Instanz). `OnMessage` greift anschließend auf die zuvor gespeicherte Instanz des `Controls` zurück und ruft schließlich die `WndProc`-Routine von `Control` auf – die Nachricht ist jetzt bei der richtigen Komponente angekommen.

Die Aufgabe von `Control` ist jetzt nur noch, die Nachricht zu filtern, und das daraus resultierende Ereignis auszulösen. Das bedeutet aber auch, dass `WndProc` die unterste Basis jeder `Control`-Instanz (und damit auch jedes Formulars) darstellt, sich in die Ereigniskette hineinzuhängen.

In unserem Beispiel war das `Control` die Schaltfläche *Grafik öffnen*. Dessen `Click`-Ereignis haben wir in unserem Programm eingebunden, mit einer entsprechenden Auswertungslogik versehen, und dass das Ergebnis tadellos funktioniert, können Sie sehen, wenn Sie das Programm starten und auf die Schaltfläche klicken.

Doch lassen Sie uns an dieser Stelle herausfinden, was es mit der Anzeige der Nachrichten im Ausgabefenster auf sich hat. Dazu werfen Sie einen Blick auf das Hauptformular des Beispielprojekts, und betrachten die *Grafik öffnen*-Schaltfläche ein wenig genauer im Designer (siehe Abbildung 38.5).

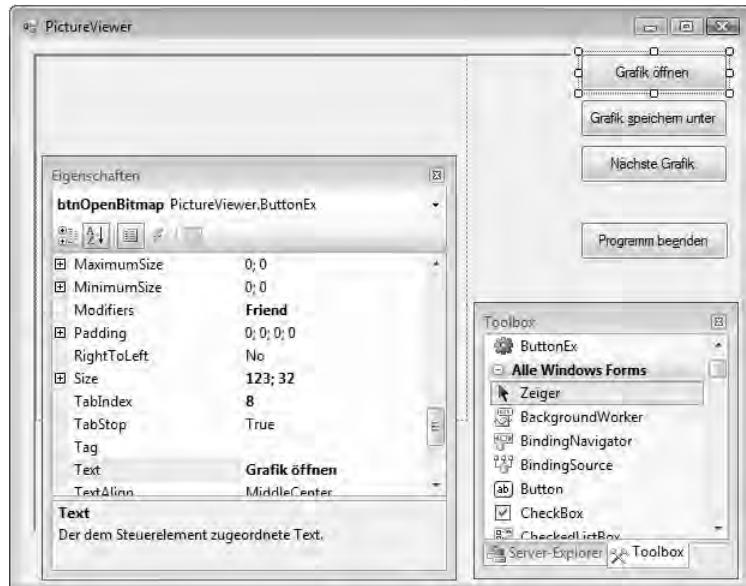


Abbildung 38.5 Bei genauer Betrachtung wird deutlich, dass es sich bei der »Grafik öffnen«-Schaltfläche nicht um ein Button-, sondern um ein ButtonEx-Steuerelement handelt

Sie sehen, dass es sich bei der Schaltfläche im Formular gar nicht um eine »herkömmliche« Schaltfläche, sondern um eine Erweiterung handelt. Eine zusätzliche Codedatei namens *ButtonEx.vb* klärt, wieso sowohl Toolbox als auch Formular über eine eigentlich unbekannte Schaltfläche verfügen können:

```
Public Class ButtonEx
    Inherits Button

    Private Const WM_RBUTTONDOWN As Integer = &H204
    Private Const WM_RBUTTONUP As Integer = &H205
    Private myDownFlag As Boolean

    Public Event RightClick(ByVal Sender As Object, ByVal e As EventArgs)

    Protected Overrides Sub WndProc(ByRef m As System.Windows.Forms.Message)
        Debug.Print(m.ToString)
        If m.Msg = WM_RBUTTONDOWN Then
            myDownFlag = True
        End If
        If m.Msg = WM_RBUTTONUP And myDownFlag Then
            myDownFlag = False
            OnRightClick(Me, EventArgs.Empty)
        End If
        MyBase.WndProc(m)
    End Sub

    Protected Overridable Sub OnRightClick(ByVal Sender As Object, ByVal e As EventArgs)
        RaiseEvent RightClick(Sender, e)
    End Sub
End Class
```

Das Erstellen einer Klassendatei mit diesem Namen und das Einfügen dieses Codes haben ausgereicht, um nach dem ersten Kompilieren das »neue« Steuerelement in der Toolbox anzeigen zu lassen und es sofort anschließend im Formular verwenden zu können.

Für ein weiteres Experiment setzen Sie nun in der im oben stehenden Klassencode in der fett ausgezeichneten Zeile mit der Funktionstaste **[F9]** einen Haltepunkt. Starten Sie das Programm anschließend und klicken Sie mit der *rechten* Maustaste auf die Schaltfläche *Grafik öffnen*.

Sie können im Ausgabefenster nun in aller Ruhe die Nachrichtenhistorie der Schaltfläche betrachten (siehe Abbildung 38.5). *hwnd* im Fenster bezeichnet übrigens das Window Handle, von dem in den vorherigen Erklärungen die Rede war und das die Warteschlangenroutine verwendet hat, um die .NET-Schaltfläche wieder zu finden. Der Debugger bietet Ihnen die Möglichkeit, die Aufruf-Hierarchie der aktuellen Funktion aufzulisten, in der das Programm gerade »steckt«. Um die Aufrufliste darzustellen, drücken Sie einfach die Tastenkombination **[Strg][Alt][C]** (alternativ wählen Sie aus dem Menü *Debuggen* den Menüpunkt *Fenster* und weiter den Untermenüpunkt *Aufrufliste*).

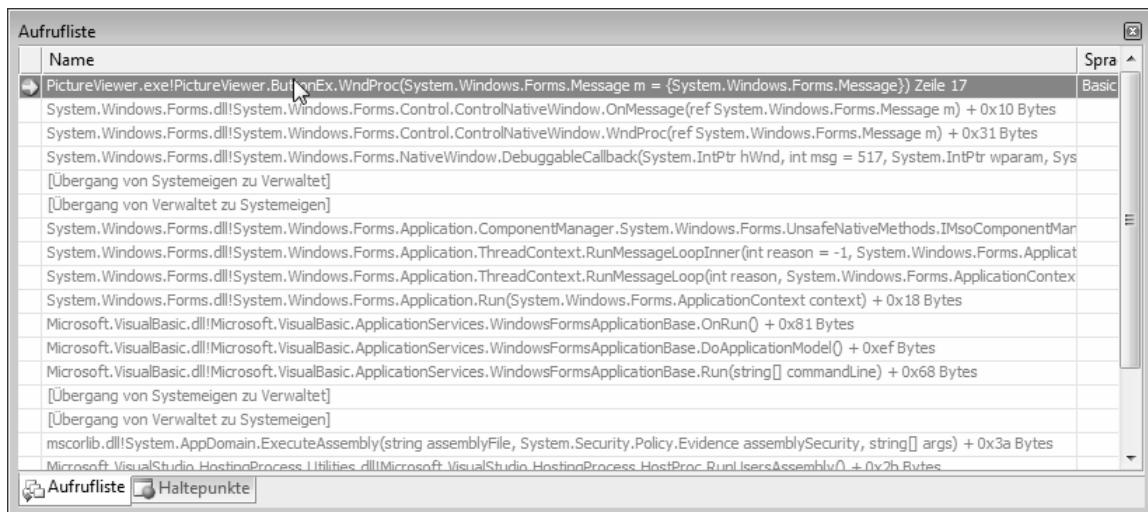


Abbildung 38.6 Die Aufrufliste zeigt die Quelle des Funktionsaufrufs

Die Aufrufliste verifiziert die vorhin geschilderte Erklärung. Fast lückenlos sind die verschiedenen Funktionsaufrufe in der Kette zu verfolgen, die letzten Endes zum Ausführen der Routine *WndProc* des jeweiligen Steuerelements oder Formulars führen.

HINWEIS In der Standardeinstellung von Visual Studio werden Aufrufe, die nicht von Ihrer Anwendung stammen, nicht im Ausgabefenster angezeigt. In diesem Fall öffnen Sie das Kontextmenü der Aufrufliste mit der rechten Maustaste und wählen den Eintrag *Nicht-benutzerseitigen Code anzeigen* aus.

Implementieren neuer Steuerelementereignisse auf Basis von Warteschlangenauswertungen

Nun wissen Sie, wie Nachrichten des Windows-Betriebssystems in .NET-WinForms-Anwendungen verarbeitet werden. Dieses Wissen können Sie sich zunutze machen, um vorhandene Steuerelemente um eigene Ereignisse zu erweitern.

So könnte es für den Entwickler beispielsweise von Vorteil sein, ein RightClick-Ereignis zu erhalten, wenn der Anwender über der Schaltfläche die rechte Maustaste drückt.

Wenn ein Mausklickereignis ein bestimmtes Objekt erreichen soll, dann reicht es nicht aus, die Nachrichtenwarteschlange von Windows daraufhin abzufragen. Eine Klick-Nachricht gibt es nämlich in dieser Form überhaupt nicht. Ein Klick besteht aus einer direkten Folge aus Nachrichten vom Nachrichtentyp *WM_LMOUSEDOWN*⁵ und *WM_LMOUSEUP* – jedenfalls soweit das die linke Maustaste betrifft.

⁵ Die Bezeichnung von Ereignissen in Form von Konstanten hat eine lange Geschichte und reicht zurück bis zur Programmierung von Windows 2.11 unter C. WM_ steht dabei natürlich für Windows Message. In .NET sind diese Konstanten leider nicht vordefiniert, aber spätere Beispiele zeigen, wie Sie an die entsprechenden Definitionen gelangen.

In der Routine *WndProc*, die die Windows-Nachrichten schon gefiltert für die eigene Instanz der Klasse (*TestButton* im Beispiel) bekommt, sollte es deswegen eine Member-Variable geben, die sich merkt, ob die rechte Maustaste bereits gedrückt wurde. Dann kann der Auswertungsabschnitt für die Nachricht des Loslassens der rechten Maustaste beide Aktionen als »Mausklick rechts« interpretieren und das *RightClick*-Ereignis auslösen.

HINWEIS Mir ist klar, dass es mehrere Möglichkeiten gibt, ein solches Ereignis zur Verfügung zu stellen. Alternativ zum Abfangen der Windows-Nachrichten in *WndProc* ließe sich auch *PreProcessMessage* überschreiben – eine Funktion, die den Vorteil hat, bereits fix und fertig aufbereitete Windows-Nachrichten im .NET-üblichen *Message*-Format zu empfangen. Auch das Überschreiben von *OnMouseDown* und *OnMouseUp* wäre denkbar – aber: Es geht an dieser Stelle mehr um die Demonstration der grundlegenden Verfahren als um die beste Form der Implementierung. Die schnellste ist die *WndProc*-Methode allemal, denn alle anderen Funktionen werden mehr oder weniger direkt aus *WndProc* heraus aufgerufen.

Der Beispielcode auf Seite 1108 zeigt die komplette Implementierung, die letzten Endes zum Ausführen des Ereignisses führt. Mehr über die Grundlagen von Ereignissen und zum Auslösen von Ereignissen erfahren Sie übrigens in Kapitel 20.

Entfernen Sie nun den Haltepunkt, den Sie im vorherigen Abschnitt gesetzt haben, und starten Sie die Beispielanwendung erneut. Und nun beobachten Sie, was passiert, wenn Sie die Schaltflächen *Grafik öffnen* und *Nächste Grafik* mit der rechten Maustaste bedienen.

- Wenn Sie die Schaltfläche *Grafik öffnen* normal betätigen, wird standardmäßig jeweils das Verzeichnis in der Dateiauswahl angezeigt, das sie als Letztes verwendet haben. Klicken Sie auf diese Schaltfläche jedoch mit der *rechten* Maustaste, sehen Sie grundsätzlich die Verzeichnisauswahl Ihres *Eigene Bilder*-Verzeichnisses.
- Wenn Sie die Schaltfläche *Nächste Grafik* normal betätigen, wird automatisch das nächste Bild im Verzeichnis dargestellt. Betätigen Sie jedoch die *rechte* Maustaste, dann wird die erste Bilddatei des Verzeichnisses dargestellt.

Wer oder was löst welche Formular- bzw. Steuerelementereignisse wann aus?

Es gibt eine Vielzahl von Ereignissen, die durch Benutzeraktionen bei Formularen (und den Schaltflächen, die sie beherbergen) ausgelöst werden können. Auch, wenn Sie sich schon eine Weile mit diesem Thema beschäftigt haben, bleibt es immer noch aufwändig herauszufinden, welche Aktion des Benutzers welches Ereignis wann auslöst.

Ich habe mir lange Gedanken darüber gemacht, was Ihnen beim Finden des richtigen Ereignisses und beim Verstehen der richtigen Zusammenhänge bei Ereignissen am besten helfen kann. Das Ergebnis ist das folgende Programm, das die Geheimnisse um Ereignisse sowohl für Formulare als auch für Komponenten auf seine Weise löst.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel38\\Ereignistester

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Programm soll gleich zweierlei Zwecke erfüllen. Auf der einen Seite soll es Ihnen als eine Art Nachschlagewerk dienen. Sämtliche Methoden, die es überschreibt, sind dokumentiert, und sie beschreiben sozusagen direkt vor Ort, welche Überschreibung welchen Zweck erfüllt. Wenn Sie dennoch nicht sicher sind, in welchem Zusammenhang die verschiedenen Ereignisse stehen, können Sie es auf der anderen Seite auch als Testprogramm verwenden, um mit den Ereignissen zu experimentieren. Die wichtigsten Ereignisse sind nämlich überschrieben, und wenn Sie das Programm starten, generiert es ein Testformular, das auch eine Testkomponente enthält. Durch Verschieben, Vergrößern, Verkleinern, Darüberfahren mit der Maus, Daraufklicken und das Ausführen anderer Aktionen sehen Sie im Ausgabefenster, welches Ereignis zu welcher Zeit aufgerufen wird.

Wenn Sie dieses Programm starten, sehen Sie zunächst einen Dialog, etwa wie in Abbildung 38.7 auf dem Bildschirm:



Abbildung 38.7 Im Hauptdialog der Anwendung nehmen Sie die Einstellungen für die Elemente und Ereignisse vor, die Sie nachverfolgen bzw. testen möchten

Dieser Dialog erlaubt Ihnen zu bestimmen, welche Ereigniskategorie im Ausgabefenster protokolliert werden soll. Die verschiedenen Ereignisse sind dabei in Gruppen eingeteilt. Durch die Kontrollkästchen im Bereich *Testeinstellungen für Ereignisse* wählen Sie die zu protokollierenden aus. Im darunter liegenden Bereich bestimmen Sie, ob die Ereignisse nur für das Formular, nur für die Testkomponente oder für beide ausgegeben werden sollen.

In der Rubrik *Einstellungen für das Testformular* bestimmen Sie, mit welchen Attributen Sie das Testformular ausstatten möchten. Diese Optionen repräsentieren die wichtigsten Eigenschaften, die Sie auch zur Entwurfszeit für ein Formular einstellen können.

Wählen Sie nun bitte die Einstellungen so aus, wie Sie sie auch in Abbildung 38.7 sehen können. Klicken Sie anschließend auf die Schaltfläche *Testform mit TestControl* erzeugen.

Sie sehen anschließend ein Formular mit einer wunderschönen, selbst gestrickten Komponente, wie Sie auch in Abbildung 38.8 zu sehen ist.

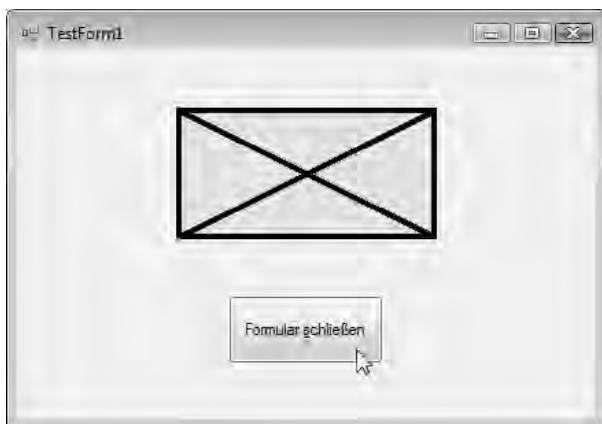


Abbildung 38.8 Mit diesem Testformular und seiner benutzerdefinierten Komponente können Sie Ihre Experimente durchführen

Und jetzt tobten Sie sich aus! Bewegen Sie das Formular über den Bildschirm. Fahren Sie mit dem Mauszeiger über Formular und Komponenten. Vergrößern und verkleinern Sie das Formular. Wechseln Sie mit der **[Esc]**-Taste den Fokus der beiden Komponenten. Klicken Sie in das Formular. Halten Sie den Mausknopf über dem Formular gedrückt und bewegen Sie dabei die Maus.

Wann immer Sie Aktionen durchführen, sehen Sie im Ausgabefenster eine entsprechende Kommentierung dazu, wie etwa in folgendem Protokollauszug zu sehen:

```
'FormEventChain.exe': 'c:\windows\assembly\gac\system.xml\1.0.5000.0_b77a5c561934e089\system.xml.dll'
geladen, keine Symbole geladen.
FormEventChain.frmTest, Text: frmTest: OnLayout: AffectedControl=FormEventChain.frmTest, Text: frmTest;
AffectedProperty=Bounds
FormEventChain.frmTest, Text: frmTest: OnLayout: AffectedControl=FormEventChain.frmTest, Text: frmTest;
AffectedProperty=Bounds
FormEventChain.frmTest, Text: TestForm1: OnLayout: AffectedControl=; AffectedProperty=
FormEventChain.TestControl: OnLayout: AffectedControl=FormEventChain.TestControl;
AffectedProperty=Bounds
FormEventChain.TestControl: OnLayout: AffectedControl=FormEventChain.TestControl;
AffectedProperty=Bounds
FormEventChain.frmTest, Text: TestForm1: OnLayout: AffectedControl=FormEventChain.TestControl;
AffectedProperty=Parent
FormEventChain.frmTest, Text: TestForm1: OnLayout: AffectedControl=System.Windows.Forms.Button, Text:
Formular & schließen; AffectedProperty=Parent
FormEventChain.frmTest, Text: TestForm1: OnHandleCreated
FormEventChain.frmTest, Text: TestForm1: OnActivated
FormEventChain.frmTest, Text: TestForm1: OnInvalidate: InvalidRect={X=0,Y=0,Width=390,Height=236}
FormEventChain.TestControl: OnHandleCreated
FormEventChain.TestControl: OnCreateControl
FormEventChain.frmTest, Text: TestForm1: OnLoad
FormEventChain.frmTest, Text: TestForm1: OnCreateControl
FormEventChain.frmTest, Text: TestForm1: OnLayout: AffectedControl=; AffectedProperty=
```

```
FormEventChain.frmTest, Text: TestForm1: OnPaintBackground:{X=0,Y=0,Width=390,Height=236}
FormEventChain.TestControl: OnInvalidated: InvalidRect={X=0,Y=0,Width=195,Height=79}
FormEventChain.frmTest, Text: TestForm1: SetVisibleCore: value=True
FormEventChain.frmTest, Text: TestForm1: OnPaint:{X=0,Y=0,Width=390,Height=236}
FormEventChain.TestControl: OnPaintBackground:{X=0,Y=0,Width=195,Height=79}
FormEventChain.TestControl: OnPaint:{X=0,Y=0,Width=195,Height=79}
FormEventChain.TestControl: OnPaint:{X=0,Y=0,Width=195,Height=79}
FormEventChain.TestControl: OnPaint:{X=0,Y=0,Width=195,Height=79}
FormEventChain.TestControl: OnPaintBackground:{X=0,Y=0,Width=195,Height=79}
FormEventChain.TestControl: OnPaint:{X=0,Y=0,Width=195,Height=79}
'FormEventChain.exe':
'c:\windows\assembly\gac\microsoft.visualbasic\7.0.5000.0__b03f5f7f11d50a3a\microsoft.visualbasic.dll'
geladen, keine Symbole geladen.
FormEventChain.frmTest, Text: TestForm1: OnDeactivate
FormEventChain.frmTest, Text: TestForm1: OnHandleDestroyed
FormEventChain.TestControl: OnHandleDestroyed
FormEventChain.TestControl: Dispose
FormEventChain.frmTest, Text: : Dispose
```

Es ist interessant zu sehen, wie die einzelnen Ereignisse sich gegenseitig bedingen, finden Sie nicht?

Noch interessanter ist, welche Ereignisse mit welchen Prozeduren abgefangen werden können. Prinzipiell spielt es natürlich keine Rolle, ob Sie ein Ereignis als Event im Sinne eines .NET Framework-Events behandeln oder, wenn es im Kontext möglich ist, eine entsprechende Prozedur durch Überschreiben verwenden. Zu diesem Zweck sollen Ihnen die folgenden Seiten dienen, die das kommentierte Listing enthalten. Es ist gemäß den Kategorien der Ereignisse in verschiedene Sektionen unterteilt, die auch mit entsprechenden Überschriften versehen sind. Dadurch können Sie dieses Listing auch als Nachschlagewerk verwenden, wenn Sie später, beim Entwickeln Ihrer eigenen Komponenten und Anwendungen, schnell eine geeignete Ereignisroutine finden müssen.

HINWEIS Die Definitionszeilen der einzelnen Prozeduren sind fett formatiert, damit Sie sie leichter im Listing erkennen können. Darüber hinaus sind die Prozeduren – soweit möglich – nach der Reihenfolge ihres Auftretens innerhalb der einzelnen Kategorien sortiert.

Auch wenn das Abdrucken beider Teile – der des Formulars und der der Control-Klasse – auf den ersten Blick doppelt und damit überflüssig erscheint: Formulare und Steuerelemente lösen in gleichen Situationen oft verschiedene Ereignisse aus – das ist der Grund.

Die Kommentare sind zu Gunsten der leichteren Lesbarkeit im folgenden Listing in normalen Fließtext umgewandelt worden. Im Code selbst finden Sie die Kommentare in gleichem Wortlaut als Kommentarzeilen.

Kategorie Erstellen und Zerstören des Formulars

```
'*****
'Erstellen, Aktivieren, Deaktivieren und Zerstören
*****
```

OnHandleCreated: Wird aufgerufen, nachdem das *Window-Handle* für die Formular-Instanz erstellt wurde. Ab diesem Zeitpunkt ist das Formular von der Nachrichtenwarteschlange erkennbar.

```
Protected Overrides Sub OnHandleCreated(ByVal e As System.EventArgs)
    MyBase.OnHandleCreated(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnHandleCreated")
    End If
End Sub
```

OnLoad: Tritt ein, kurz bevor die Formular-Instanz das erste Mal sichtbar wird. Sie haben hier die Möglichkeit, Initialisierungen für das Formular vorzunehmen. Beachten Sie, dass das Fokussieren von Steuerelementen zu dieser Zeit noch nicht funktioniert und eine Ausnahme auslösen würde.

```
Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
    MyBase.OnLoad(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnLoad")
    End If
End Sub
```

OnCreateControl: Tritt ein, nachdem die Framework-seitigen Ressourcen für das Formular erstellt wurden. Die Basisfunktion muss in den Framework-Versionen 1.0 und 1.1 nicht notwendigerweise aufgerufen werden; aus Aufwärtskompatibilitätsgründen sollte es aber dennoch passieren.

```
Protected Overrides Sub OnCreateControl()
    MyBase.OnCreateControl()
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnCreateControl")
    End If
End Sub
```

```
Protected Overrides Sub OnActivated(ByVal e As System.EventArgs)
    MyBase.OnActivated(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnActivated")
    End If
End Sub
```

SetVisibleCore: Wird aufgerufen, um einer ableitenden Klasse beim Initialisierungsvorgang die Möglichkeit zu geben, die Sichtbarkeit (durch die `Visible`-Eigenschaft gesteuert) zu ändern. Eigentlich ist diese Routine kein richtiges Ereignis, sondern nur die ausführende Unterfunktion einer Eigenschaft.

```
Protected Overrides Sub SetVisibleCore(ByVal value As Boolean)
    MyBase.SetVisibleCore(value)
    If myShowCreationDestroy Then
        Debug.WriteLine(String.Format(Me.ToString + ": SetVisibleCore: value={0}", value))
    End If
End Sub
```

OnClosing: Wird aufgerufen, wenn der Schließen-Vorgang des Formulars beginnt. Sie können das Schließen verhindern, indem Sie die Cancel-Eigenschaft von e auf True setzen.

```
Protected Overrides Sub OnClosing(ByVal e As System.ComponentModel.CancelEventArgs)
    MyBase.OnClosing(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnClosing")
    End If
End Sub
```

OnClosed: Wird aufgerufen, wenn das Formular geschlossen wurde.

```
Protected Overrides Sub OnClosed(ByVal e As System.EventArgs)
    MyBase.OnClosed(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnClosed")
    End If
End Sub
```

OnDeactivate: Wird aufgerufen, wenn das Formular deaktiviert wurde.

```
Protected Overrides Sub OnDeactivate(ByVal e As System.EventArgs)
    MyBase.OnDeactivate(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnDeactivate")
    End If
End Sub
```

OnHandleDestroyed: Wird aufgerufen, wenn das *Window-Handle* des Formulars zerstört wurde.

```
Protected Overrides Sub OnHandleDestroyed(ByVal e As System.EventArgs)
    MyBase.OnHandleDestroyed(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnHandleDestroyed")
    End If
End Sub
```

Dispose: Das Formular überschreibt den Löschgong der Basisklasse, um Komponenten zu bereinigen. Diese Routine wird in der Regel durch den Formular-Designer implementiert.

```
Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
    If disposing Then
        If Not (components Is Nothing) Then
            components.Dispose()
        End If
    End If
    MyBase.Dispose(disposing)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": Dispose")
    End If
End Sub
```

Kategorie Mausereignisse des Formulars

```
'*****
'*Mausereignisse
'*****
```

OnMouseDown: Wird aufgerufen, wenn ein Mausbutton gedrückt wird und sich die Maus über einem Bereich des Formulars, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) befindet.

```
Protected Overrides Sub OnMouseDown(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseDown(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnMouseDown: x={0}; y={1}; delta={2}; button={3}; clicks={4}" +
            , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnClick: Wird aufgerufen, wenn ein Mausklick mit der linken Maustaste über einem Bereich des Formulars, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) durchgeführt wird.

```
Protected Overrides Sub OnClick(ByVal e As System.EventArgs)
    MyBase.OnClick(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnClick")
    End If
End Sub
```

OnDoubleClick: Wird aufgerufen, wenn ein Doppelklick mit der linken Maustaste über einem Bereich des Formulars, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) durchgeführt wird.

```
Protected Overrides Sub OnDoubleClick(ByVal e As System.EventArgs)
    MyBase.OnDoubleClick(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnDoubleClick")
    End If
End Sub
```

OnMouseUp: Wird aufgerufen, wenn ein Mausbutton losgelassen wird und sich die Maus über einem Bereich des Formulars, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) befindet.

```
Protected Overrides Sub OnMouseUp(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseUp(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnMouseUp: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
            , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnMouseEnter: Wird aufgerufen, wenn der Mauszeiger den Bereich des Formulars, aber nicht einen *ChildWindow*-Bereich (andere Komponente) betritt.

```
Protected Overrides Sub OnMouseEnter(ByVal e As System.EventArgs)
    MyBase.OnMouseEnter(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseEnter:" + e.ToString)
    End If
End Sub
```

OnMouseHover: Wird aufgerufen, wenn der Mauszeiger das erste Mal nach dem Betreten des Formularbereichs zur Ruhe gekommen ist.

```
Protected Overrides Sub OnMouseHover(ByVal e As System.EventArgs)
    MyBase.OnMouseHover(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseHover:" + e.ToString)
    End If
End Sub
```

OnMouseMove: Wird aufgerufen, wenn der Mauszeiger über dem Bereich des Formulars, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) bewegt wird.

```
Protected Overrides Sub OnMouseMove(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseMove(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnMouseMove: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
            , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnMouseLeave: Wird aufgerufen, wenn der Mauszeiger den Bereich des Formulars verlässt.

```
Protected Overrides Sub OnMouseLeave(ByVal e As System.EventArgs)
    MyBase.OnMouseLeave(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseLeave:" + e.ToString)
    End If
End Sub
```

OnMouseWheel: Wird aufgerufen, wenn das Mausrad über dem Bereich des Formulars bewegt wird. Dieses Ereignis wird für alle untergeordneten Komponenten (*ChildWindows*) ebenfalls ausgelöst!

```
Protected Overrides Sub OnMouseWheel(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseWheel(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnMouseWheel: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
            , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

Kategorie Tastaturereignisse des Formulars

```
'*****
'*Tastatur
'*****
```

OnKeyDown: Wird aufgerufen, wenn eine Taste gedrückt wird. Wird allerdings nicht aufgerufen, wenn es eine weitere, fokussierte Komponente im Formular gibt und die KeyPreview-Eigenschaft auf False gesetzt wurde bzw. die überschriebene ProcessKeyPreview-Methode (s.u.) das Ereignis schon verarbeitet hat.

```
Protected Overrides Sub OnKeyDown(ByVal e As System.Windows.Forms.KeyEventArgs)
    MyBase.OnKeyDown(e)
    If myShowKeyboard Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnKeyDown: KeyCode={0}; KeyData={1}; KeyValue={2}; Modifiers={3}" _
            , e.KeyCode, e.KeyData, e.KeyValue, e.Modifiers))
    End If
End Sub
```

OnKeyPress: Wird aufgerufen, wenn eine Taste gedrückt wird; wird nicht aufgerufen, wenn eine Steuerungstaste (wie **Strg** oder **Shift**) alleine oder in Kombination mit einer anderen gedrückt wird. Diese Prozedur wird auch dann nicht aufgerufen, wenn es eine weitere, fokussierte Komponente im Formular gibt und die KeyPreview-Eigenschaft auf False gesetzt wurde bzw. die überschriebene ProcessKeyPreview-Methode (s.u.) das Ereignis schon verarbeitet hat.

```
Protected Overrides Sub OnKeyPress(ByVal e As System.Windows.Forms.KeyPressEventArgs)
    MyBase.OnKeyPress(e)
    If myShowKeyboard Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnKeyPress: KeyChar={0}" _
            , e.KeyChar))
    End If
End Sub
```

OnKeyUp: Wird aufgerufen, wenn eine Taste wieder losgelassen wird. Diese Prozedur wird nicht aufgerufen, wenn es eine weitere, fokussierte Komponente im Formular gibt und die KeyPreview-Eigenschaft auf False gesetzt wurde bzw. die überschriebene ProcessKeyPreview-Methode (s.u.) das Ereignis schon verarbeitet hat.

```
Protected Overrides Sub OnKeyUp(ByVal e As System.Windows.Forms.KeyEventArgs)
    MyBase.OnKeyUp(e)
    If myShowKeyboard Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnKeyUp: KeyCode={0}; KeyData={1}; KeyValue={2}; Modifiers={3}", _
            e.KeyCode, e.KeyData, e.KeyValue, e.Modifiers))
    End If
End Sub
```

Kategorie Position und Größe des Formulars

```
'*****
'*Größe und Position
'*****
```

OnMove: Wird aufgerufen, wenn die Formularposition verändert wird. Diese Methode wird kontinuierlich aufgerufen, während der Anwender das Formular verschiebt und die Anzeigeneinstellung so vorgenommen wurde, dass der Fensterinhalt beim Ziehen mit verschoben wird.

```
Protected Overrides Sub OnMove(ByVal e As System.EventArgs)
    MyBase.OnMove(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnMove:" + e.ToString)
    End If
End Sub
```

OnLocationChanged: Wird aufgerufen, wenn sich die Position des Formulars verändert hat. Diese Methode wird leider ebenfalls kontinuierlich aufgerufen, wenn die Anzeigeneinstellung so vorgenommen wurde, dass der Fensterinhalt beim Ziehen mit verschoben wird, sodass ein Ende der Verschiebeaktion hiermit nicht festgestellt werden kann. Um das zu erreichen, müssten Sie WndProc überschreiben und die empfangene Nachricht dort auf WM_EXITSIZEMOVE überprüfen. Ein Beispiel dazu finden Sie in Kapitel 39.

```
Protected Overrides Sub OnLocationChanged(ByVal e As System.EventArgs)
    MyBase.OnLocationChanged(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnLocationChanged:" + e.ToString)
    End If
End Sub
```

OnResize: Wird aufgerufen, wenn die Formulargröße verändert wird. Diese Methode wird kontinuierlich aufgerufen, während der Anwender das Formular vergrößert oder verkleinert und die Anzeigeneinstellung so vorgenommen wurde, dass der Fensterinhalt beim Ziehen mit verschoben wird.

```
Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
    MyBase.OnResize(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnResize:" + e.ToString)
    End If
End Sub
```

OnSizeChanged: Wird aufgerufen, wenn sich die Größe des Formulars verändert hat. Diese Methode wird leider ebenfalls kontinuierlich aufgerufen, wenn die Anzeigeneinstellung so vorgenommen wurde, dass der Fensterinhalt beim Ziehen mit verschoben wird, sodass ein Abschluss der Größenänderung hiermit nicht festgestellt werden kann. Um das zu erreichen, müssten Sie WndProc überschreiben und die empfangene Nachricht dort auf den Wert WM_EXITSIZEMOVE überprüfen. Ein Beispiel dazu finden Sie in Kapitel 39.

```
Protected Overrides Sub OnSizeChanged(ByVal e As System.EventArgs)
    MyBase.OnSizeChanged(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnSizeChanged:" + e.ToString)
    End If
End Sub
```

Kategorie Anordnen der Komponenten und Neuzeichnen des Formulars

```
*****
'Anordnen und Neuzeichnen
*****
```

OnInvalidated: Wird aufgerufen, wenn eine Entität das Neuzeichnen des Formularinhalts mit Invalidate anfordert. Invalidate sollte in der Regel von OnResize aufgerufen werden, wenn der Inhalt des Fensters in Abhängigkeit von der Fenstergröße komplett neu gezeichnet werden muss. Ausgenommen sind Änderungen am Verhalten durch SetStyle (Kapitel 39).

```
Protected Overrides Sub OnInvalidated(ByVal e As System.Windows.Forms.InvalidateEventArgs)
    MyBase.OnInvalidated(e)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnInvalidated: InvalidRect={0}", _
            e.InvalidRect))
    End If
End Sub
```

OnLayout: Wird aufgerufen, wenn das Formular anzeigt, dass seine beinhaltenden Steuerelemente aus irgendwelchen Gründen neu angeordnet werden müssen.

HINWEIS Für Änderungen an einem Steuerelement, z.B. Größenänderungen, Ein- oder Ausblenden sowie Hinzufügen oder Entfernen untergeordneter Steuerelemente ist es notwendig, dass das Layout der untergeordneten Steuerelemente vom Steuerelement festgelegt wird. Der diesem Ereignis mitgegebene Parameter LayoutEventArgs gibt das geänderte untergeordnete Steuerelement und die davon betroffene Eigenschaft an. Wenn z.B. ein Steuerelement seit dem letzten Layoutvorgang sichtbar gemacht wurde, ist davon die Visible-Eigenschaft betroffen. Die AffectedControl- und AffectedProperty-Eigenschaften werden auf Nothing festgelegt, wenn beim Aufruf der PerformLayout-Methode keine Werte bereitgestellt wurden. Dieses Ereignis erfolgt nicht, wenn das Formular das Layout-Ereignis mit SuspendLayout außer Kraft gesetzt hat.

```
Protected Overrides Sub OnLayout(ByVal levent As System.Windows.Forms.LayoutEventArgs)
    MyBase.OnLayout(levent)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(String.Format( _
```

```

        Me.ToString + ": OnLayout: AffectedControl={0}; AffectedProperty={1}", _
        levent.AffectedControl, levent.AffectedProperty))
End If
End Sub

```

OnPaintBackground: Wird aufgerufen, wenn der Hintergrund des Formulars neu gezeichnet werden muss. Das Graphics-Objekt, das mit dem Parameter vom Typ PaintEventArgs dem Ereignis übergeben wird, ist ausschließlich auf den Bereich *geclipped*, der neu gezeichnet werden muss. Wenn durch die Vergrößerung des Fensters der Fensterinhalt komplett neu gezeichnet werden muss, dann sollte OnResize bzw. das Resize-Ereignis die Methode Invalidate aufrufen, damit den Paint-Ereignissen ein ungeclippter Bereich für das Neuzeichnen des kompletten Inhalts übergeben wird. Beispiele dafür gibt es auch in Kapitel 39.

```

Protected Overrides Sub OnPaintBackground(ByVal pevent As System.Windows.Forms.PaintEventArgs)
    MyBase.OnPaintBackground(pevent)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(Me.ToString + ": OnPaintBackground:" + pevent.ClipRectangle.ToString)
    End If
End Sub

```

OnPaint: Wird aufgerufen, wenn der Fensterinhalt neu gezeichnet werden muss. Das Graphics-Objekt, das mit dem Parameter vom Typ PaintEventArgs dem Ereignis übergeben wird, ist ausschließlich auf den Bereich *geclipped*, der neu gezeichnet werden muss. Wenn durch die Vergrößerung des Fensters der Fensterinhalt komplett neu gezeichnet werden muss, dann sollte OnResize bzw. das Resize-Ereignis die Methode Invalidate aufrufen, damit den Paint-Ereignissen ein ungeclippter Bereich für das Neuzeichnen des kompletten Inhalts übergeben wird. Beispiele dafür gibt es auch in Kapitel 39.

```

Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    MyBase.OnPaint(e)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(Me.ToString + ": OnPaint:" + e.ClipRectangle.ToString)
    End If
End Sub

```

Kategorie Fokussierung des Formulars

```

'*****
'*Fokussierung
'*****

```

OnEnter: Wird aufgerufen, wenn das Formular aktiviert wird, aber nur, wenn es mindestens eine weitere Komponente beinhaltet, die den Fokus beim Aktivieren bekommen kann.

```

Protected Overrides Sub OnEnter(ByVal e As System.EventArgs)
    MyBase.OnEnter(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnEnter:" + e.ToString)
    End If
End Sub

```

OnGotFocus: Wird aufgerufen, wenn das Formular aktiviert wird, aber nur, wenn es kein weiteres Steuerelement beinhaltet, das den Fokus beim Aktivieren bekommen könnte.

```
Protected Overrides Sub OnGotFocus(ByVal e As System.EventArgs)
    MyBase.OnLostFocus(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnGotFocus:" + e.ToString)
    End If
End Sub
```

OnLostFocus: Wird aufgerufen, wenn das Formular deaktiviert wird (zum Beispiel, weil ein anderes Fenster in den Vordergrund geklickt wurde), aber nur, wenn es keine weitere Komponente beinhaltet, die den Fokus beim Deaktivieren verlieren könnte.

```
Protected Overrides Sub OnLostFocus(ByVal e As System.EventArgs)
    MyBase.OnLostFocus(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnLostFocus:" + e.ToString)
    End If
End Sub
```

OnLeave: Wird aufgerufen, wenn das Formular deaktiviert wird, aber nur, wenn es mindestens eine weitere Komponente beinhaltet, die den Fokus beim Deaktivieren verlieren kann.

```
Protected Overrides Sub OnLeave(ByVal e As System.EventArgs)
    MyBase.OnLeave(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnLeave:" + e.ToString)
    End If
End Sub
```

Kategorie Tastaturvorverarbeitungsnachrichten des Formulars

```
'*****
'Nachrichtenverarbeitung
*****
```

ProcessCmdKey: Wird ausgelöst, wenn eine Befehlstaste (z.B. **Alt** + Anfangsbuchstabe) gedrückt wurde.

```
Protected Overrides Function ProcessCmdKey(ByRef msg As System.Windows.Forms.Message, _
                                         ByVal keyData As System.Windows.Forms.Keys) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessCmdKey:" + _
                      msg.ToString + ": KeyData: " + keyData.ToString)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funktionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessCmdKey(msg, keyData)
End Function
```

ProcessDialogChar: Wird ausgelöst, wenn eine Dialogtaste (auch Steuerungstaste) gedrückt wurde.

```
Protected Overrides Function ProcessDialogChar(ByVal charCode As Char) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessDialogChar: " + charCode)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funktionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessDialogChar(charCode)
End Function
```

ProcessDialogKey: Wird ausgelöst, wenn eine Dialog-Taste (aber keine Steuerungstaste) gedrückt wurde.

```
Protected Overrides Function ProcessDialogKey(ByVal keyData As System.Windows.Forms.Keys) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessDialogKey:" + _
                      ": KeyData: " + keyData.ToString)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funktionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessDialogKey(keyData)
End Function
```

ProcessKeyPreview: Wird bei jedem Tastenereignis des Formulars ausgelöst und regelt bei Formularen, ob in Abhängigkeit der KeyPreview-Eigenschaft Tastatur-Ereignis-Prozeduren aufgerufen werden.

```
Protected Overrides Function ProcessKeyPreview(ByRef m As System.Windows.Forms.Message) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessKeyPreview:" + m.ToString)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funktionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde
    Return False
End Function
```

WndProc: Wird bei jeder Nachricht aufgerufen, die das Fenster in irgendeiner Form betrifft. Um erweiterte Ereignisse selbst auszulösen, überschreiben Sie diese Prozedur. Rufen Sie die Basisfunktion im Anschluss nur dann auf, wenn Sie möchten, dass die Nachrichten, die Sie bereits verarbeitet haben, von der Basisklasse auch verarbeitet werden sollen.

```
Protected Overrides Sub WndProc(ByRef m As System.Windows.Forms.Message)
    If myShowWndProcMessages Then
        Console.WriteLine(m)
    End If
    MyBase.WndProc(m)
End Sub
```

Kategorie Erstellen/Zerstören des Controls (des Steuerelements)

OnHandleCreated: Wird aufgerufen, nachdem das *Window-Handle* für die Steuerelement-Instanz erstellt wurde.

```
Protected Overrides Sub OnHandleCreated(ByVal e As System.EventArgs)
    MyBase.OnHandleCreated(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnHandleCreated")
    End If
End Sub
```

OnCreateControl: Tritt ein, nachdem die Framework-seitigen Ressourcen für das Steuerelement erstellt wurden. Die Basisfunktion muss in den Framework-Versionen 1.0 und 1.1 nicht notwendigerweise aufgerufen werden; aus Aufwärtskompatibilitätsgründen sollte das aber dennoch passieren.

```
Protected Overrides Sub OnCreateControl()
    MyBase.OnCreateControl()
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnCreateControl")
    End If
End Sub
```

OnHandleDestroyed: Wird aufgerufen, wenn das *Window-Handle* zerstört wurde.

```
Protected Overrides Sub OnHandleDestroyed(ByVal e As System.EventArgs)
    MyBase.OnHandleDestroyed(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnHandleDestroyed")
    End If
End Sub
```

Dispose: Wird aufgerufen, wenn das Steuerelement entweder durch den Garbage Collector oder durch Dispose des einbindenden *Controls/Formulars* entsorgt wird.

```
Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
    MyBase.Dispose(disposing)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": Dispose")
    End If
End Sub
```

Kategorie Mausereignisse des Controls

```
*****
'Mausereignisse
*****
```

OnMouseDown: Wird aufgerufen, wenn ein Mausbutton gedrückt wird und sich die Maus über einem Bereich des *Controls*, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) befindet.

```
Protected Overrides Sub OnMouseDown(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseDown(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnMouseDown: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
            , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnClick: Wird aufgerufen, wenn ein Mausklick mit der linken Maustaste über einem Bereich des *Controls*, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) durchgeführt wird.

```
Protected Overrides Sub OnClick(ByVal e As System.EventArgs)
    MyBase.OnClick(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnClick")
    End If
End Sub
```

OnDoubleClick: Wird aufgerufen, wenn ein Doppelklick mit der linken Maustaste über einem Bereich des *Controls*, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) durchgeführt wird.

```
Protected Overrides Sub OnDoubleClick(ByVal e As System.EventArgs)
    MyBase.OnDoubleClick(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnDoubleClick")
    End If
End Sub
```

OnMouseUp: Wird aufgerufen, wenn ein Mausbutton losgelassen wird und sich die Maus über einem Bereich des *Controls*, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) befindet.

```
Protected Overrides Sub OnMouseUp(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseUp(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnMouseUp: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
            , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnMouseEnter: Wird aufgerufen, wenn der Mauszeiger den Bereich des *Controls*, aber nicht einen *ChildWindow*-Bereich (andere Komponente) betritt.

```
Protected Overrides Sub OnMouseEnter(ByVal e As System.EventArgs)
    MyBase.OnMouseEnter(e)
    If myShowMouse Then
```

```

        Debug.WriteLine(Me.ToString + ": OnMouseEnter:" + e.ToString)
    End If
End Sub

```

OnMouseHover: Wird aufgerufen, wenn der Mauszeiger das erste Mal nach dem Betreten des Steuerelement-Bereichs zur Ruhe gekommen ist.

```

Protected Overrides Sub OnMouseHover(ByVal e As System.EventArgs)
    MyBase.OnMouseHover(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseHover:" + e.ToString)
    End If
End Sub

```

OnMouseMove: Wird aufgerufen, wenn der Mauszeiger über dem Bereich des *Controls*, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) bewegt wird.

```

Protected Overrides Sub OnMouseMove(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseMove(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnMouseMove: x={0}; y={1}; delta={2}; button={3}; clicks={4}" +
            , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub

```

OnMouseLeave: Wird aufgerufen, wenn der Mauszeiger den Bereich des *Controls* verlässt.

```

Protected Overrides Sub OnMouseLeave(ByVal e As System.EventArgs)
    MyBase.OnMouseLeave(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseLeave:" + e.ToString)
    End If
End Sub

```

OnMouseWheel: Wird aufgerufen, wenn das Mausrad bewegt wird. Wichtig: Alle Komponenten des Formulars empfangen dieses Ereignis!

```

Protected Overrides Sub OnMouseWheel(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseWheel(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnMouseWheel: x={0}; y={1}; delta={2}; button={3}; clicks={4}" +
            , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub

```

Kategorie Tastaturereignisse des Controls

```
*****  
'Tastatur  
*****
```

OnKeyDown: Wird aufgerufen, wenn eine Taste gedrückt wird und das Steuerelement den Fokus hat. Fungiert das Steuerelement als Container (von Scrollable- oder ContainerControl abgeleitet), verwenden Sie die Tastaturvorverarbeitungsnachrichten-Ereignisse, um die Tastaturereignisse auszuwerten, da sie selbst in diesem Fall nicht ausgelöst werden.

```
Protected Overrides Sub OnKeyDown(ByVal e As System.Windows.Forms.KeyEventArgs)  
    MyBase.OnKeyDown(e)  
    If myShowKeyboard Then  
        Debug.WriteLine(String.Format(  
            Me.ToString + ": OnKeyDown: KeyCode={0}; KeyData={1}; KeyValue={2}; Modifiers={3}", _  
            e.KeyCode, e.KeyData, e.KeyValue, e.Modifiers))  
    End If  
End Sub
```

OnKeyPress: Wird aufgerufen, wenn eine Taste gedrückt wird und das Steuerelement fokussiert ist; wird nicht aufgerufen, wenn eine Steuerungstaste (wie **Strg** oder **Alt**) alleine oder in Kombination mit einer anderen gedrückt wird. Fungiert das Steuerelement als Container (von Scrollable- oder ContainerControl abgeleitet), verwenden Sie die Tastaturvorverarbeitungsnachrichten-Ereignisse, um die Tastaturereignisse auszuwerten, da sie selbst in diesem Fall nicht ausgelöst werden.

```
Protected Overrides Sub OnKeyPress(ByVal e As System.Windows.Forms.KeyPressEventArgs)  
    MyBase.OnKeyPress(e)  
    If myShowKeyboard Then  
        Debug.WriteLine(String.Format(  
            Me.ToString + ": OnKeyPress: KeyChar={0}", _  
            e.KeyChar))  
    End If  
End Sub
```

OnKeyUp: Wird ausgerufen, wenn eine Taste wieder losgelassen wird und das Steuerelement den Fokus hat. Fungiert das Steuerelement als Container (von Scrollable- oder ContainerControl abgeleitet), verwenden Sie die Tastaturvorverarbeitungsnachrichten-Ereignisse, um die Tastaturereignisse auszuwerten, da sie selbst in diesem Fall nicht ausgelöst werden.

```
Protected Overrides Sub OnKeyUp(ByVal e As System.Windows.Forms.KeyEventArgs)  
    MyBase.OnKeyUp(e)  
    If myShowKeyboard Then  
        Debug.WriteLine(String.Format(  
            Me.ToString + ": OnKeyUp: KeyCode={0}; KeyData={1}; KeyValue={2}; Modifiers={3}", _  
            e.KeyCode, e.KeyData, e.KeyValue, e.Modifiers))  
    End If  
End Sub
```

Kategorie Größe und Position des Controls

```
'*****
'Größe und Position
*****
```

OnMove: Wird aufgerufen, wenn die Steuerelement-Position verändert wird.

```
Protected Overrides Sub OnMove(ByVal e As System.EventArgs)
    MyBase.OnMove(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnMove:" + e.ToString)
    End If
End Sub
```

OnLocationChanged: Wird aufgerufen, wenn sich die Position des *Controls* verändert hat.

```
Protected Overrides Sub OnLocationChanged(ByVal e As System.EventArgs)
    MyBase.OnLocationChanged(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnLocationChanged:" + e.ToString)
    End If
End Sub
```

OnResize: Wird aufgerufen, wenn sich die Ausmaße des *Controls* ändern. Wenn das Steuerelement seinen Inhalt in Abhängigkeit seiner Größe verändert, sollte an dieser Stelle ein Aufruf an `Invalidate` erfolgen, damit das parallel automatisch ausgelöste Paint-Ereignis (nur beim Vergrößern) verhindert wird und stattdessen ein neues Paint-Ereignis ausgelöst wird, das dann aber in der Lage ist, den Neuaufbau des *gesamten* Client-Bereichs durchzuführen. Hintergrund: Das Standard-Paint-Ereignis kann nur die neu zu zeichnenden Bereiche verarbeiten und wird gar nicht ausgelöst, wenn das Steuerelement nur verkleinert wird (siehe auch Kapitel 39 und Kapitel 40).

```
Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
    MyBase.OnResize(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnResize:" + e.ToString)
    End If
    Invalidate()
End Sub
```

OnSizeChanged: Wird aufgerufen, wenn sich die Ausmaße eines *Controls* geändert haben (siehe auch Kapitel 39 und Kapitel 40).

```
Protected Overrides Sub OnSizeChanged(ByVal e As System.EventArgs)
    MyBase.OnSizeChanged(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnSizeChanged:" + e.ToString)
    End If
End Sub
```

SetBoundsCore: Diese Prozedur dient zweierlei Dingen: Zum Einen wird sie von der Basisklasse bei jedem Ereignis aufgerufen, das durch das Ändern der Größe oder der Position des *Controls* aufgerufen wird. Klassen, die diese Routine überschreiben, können die Position und Ausmaße des *Controls* durch das Verändern der Parameter auf der anderen Seite reglementieren. Wenn Sie also beispielsweise nicht wollen, dass die Ausmaße des *Controls* eine bestimmte Größe überschreiten, definieren Sie in dieser Funktion für den entsprechenden Parameter einen neuen Wert, bevor Sie die Basisfunktion mit den geänderten Werten aufrufen. Der Parameter *BoundsSpecified* informiert Sie darüber, welcher Parameter durch ein Ereignis geändert wurde. Ein richtiges Beispiel dafür finden Sie in Kapitel 40.

```
Protected Overrides Sub SetBoundsCore(ByVal x As Integer, ByVal y As Integer, ByVal width As
Integer,
ByVal height As Integer, ByVal specified As System.Windows.Forms.BoundsSpecified)
    MyBase.SetBoundsCore(x, y, width, height, specified)
    If myShowPositioning Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": SetBoundsCore: X={0}; y={1}; width={2}; height={3}; specified={4}" _ ,
            x, y, width, height, specified))
    End If
End Sub
```

SetClientSize: Wird aufgerufen, wenn sich die Größe des *Controls* durch das Setzen der *ClientSize*-Eigenschaft ändern soll.

```
Protected Overrides Sub SetClientSizeCore(ByVal x As Integer, ByVal y As Integer)
    MyBase.SetClientSizeCore(x, y)
    If myShowPositioning Then
        Debug.WriteLine(String.Format(Me.ToString + ": SetClientSizeCore: X={0}; y={1}", x, y))
    End If
End Sub
```

Kategorie Neuzeichnen des Controls und Anordnen untergeordneter Komponenten

```
*****  
'Anordnen und Neuzeichnen  
*****
```

OnInvalidated: Wird aufgerufen, wenn eine Entität das Neuzeichnen des Steuerelement-Inhalts durch *Invalidate* anfordert. Hier im Beispielprogramm geschieht das durch *Resize*, *GotFocus* und *LostFocus*.

```
Protected Overrides Sub OnInvalidated(ByVal e As System.Windows.Forms.InvalidateEventArgs)
    MyBase.OnInvalidated(e)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnInvalidated: InvalidRect={0}", _
            e.InvalidRect))
    End If
End Sub
```

OnLayout: Wird aufgerufen, wenn das Steuerelement anzeigt, dass seine beinhaltenden Steuerelemente aus irgendwelchen Gründen neu angeordnet werden müssen.

```
Protected Overrides Sub OnLayout(ByVal levent As System.Windows.Forms.LayoutEventArgs)
    MyBase.OnLayout(levent)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(String.Format(
            Me.ToString + ": OnLayout: AffectedControl={0}; AffectedProperty={1}",
            levent.AffectedControl, levent.AffectedProperty))
    End If
End Sub
```

OnPaintBackground: Wird aufgerufen, wenn der Hintergrund des *Controls* neu gezeichnet werden muss.

```
Protected Overrides Sub OnPaintBackground(ByVal pevent As System.Windows.Forms.PaintEventArgs)
    MyBase.OnPaintBackground(pevent)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(Me.ToString + ": OnPaintBackground:" + pevent.ClipRectangle.ToString)
    End If
    DrawControlBackground(pevent.Graphics)
End Sub
```

OnPaint: Wird aufgerufen, wenn der Fensterinhalt neu gezeichnet werden muss.

```
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    MyBase.OnPaint(e)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(Me.ToString + ": OnPaint:" + e.ClipRectangle.ToString)
    End If
    DrawControl(e.Graphics)
End Sub
```

Kategorie Fokussierung des Controls

```
*****
'Fokussierung
*****
```

OnEnter: Wird aufgerufen, wenn das Steuerelement dabei ist, den Fokus zu erhalten.

```
Protected Overrides Sub OnEnter(ByVal e As System.EventArgs)
    MyBase.OnEnter(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnEnter:" + e.ToString)
    End If
End Sub
```

OnGotFocus: Wird aufgerufen, wenn das Steuerelement fokussiert wird, aber nicht, wenn es ein ContainerControl ist, das weitere Komponenten enthält! (In diesem Fall verwenden Sie OnEnter, um das Ereignis zu empfangen.)

```
Protected Overrides Sub OnGotFocus(ByVal e As System.EventArgs)
    MyBase.OnLostFocus(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnGotFocus:" + e.ToString)
    End If
    'Das fokussierte Control sieht anders aus als das nicht-fokussierte;
    'deswegen: alles NeuZeichnen
    Invalidate()
End Sub
```

OnLeave: Wird aufgerufen, wenn das Steuerelement dabei ist, den Fokus zu verlieren.

```
Protected Overrides Sub OnLeave(ByVal e As System.EventArgs)
    MyBase.OnLeave(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnLeave:" + e.ToString)
    End If
End Sub
```

OnLostFocus: Wird aufgerufen, wenn das Steuerelement den Fokus verloren hat, aber nicht, wenn es ein ContainerControl ist, das weitere Komponenten enthält! (In diesem Fall verwenden Sie OnLeave, um das Ereignis zu empfangen.)

```
Protected Overrides Sub OnLostFocus(ByVal e As System.EventArgs)
    MyBase.OnLostFocus(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnLostFocus:" + e.ToString)
    End If
    'Das fokussierte Control sieht anders aus als das nicht fokussierte;
    'deswegen: alles neu zeichnen.
    Invalidate()
End Sub
```

Kategorie Tastaturnachrichtenvorverarbeitung des Controls

```
'*****
'Tastatur-Vorverarbeitung
*****
```

ProcessCmdKey: Wird ausgelöst, wenn eine Befehlstaste (z.B. **Alt** + Anfangsbuchstabe) gedrückt wurde.

```
Protected Overrides Function ProcessCmdKey(ByRef msg As System.Windows.Forms.Message, _
                                         ByVal keyData As System.Windows.Forms.Keys) As Boolean
    If myShowPreProcessing Then
```

```

        Debug.WriteLine(Me.ToString + ": ProcessCmdKey:" +
                      msg.ToString + ": KeyData: " + keyData.ToString)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funktionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessCmdKey(msg, keyData)
End Function

```

ProcessDialogChar: Wird ausgelöst, wenn eine Dialogtaste (auch Steuerungstaste) gedrückt wurde.

```

Protected Overrides Function ProcessDialogChar(ByVal charCode As Char) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessDialogChar: " + charCode)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funktionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessDialogChar(charCode)
End Function

```

ProcessDialogKey: Wird ausgelöst, wenn eine Dialogtaste (aber keine Steuerungstaste) gedrückt wurde.

```

Protected Overrides Function ProcessDialogKey(ByVal keyData As System.Windows.Forms.Keys) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessDialogKey:" +
                      ": KeyData: " + keyData.ToString)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funktionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessDialogKey(keyData)
End Function

```

Die Steuerungs routinen des Beispielprogramms

Der Vollständigkeit halber finden Sie an dieser Stelle auch die Listings der Programmabschnitte, die das Testformular ins Leben rufen, den Code des Testformulars selbst und auch den Code der TestControl-Klasse, deren Instanz die Anwendung zur Laufzeit erstellt.

Soviel an Information vorweg: Sowohl TestControl als auch das Testformular haben zwei zusätzliche Konstruktoren, denen eine Sammlung mit Flags übergeben wird. Diese Flags steuern, welche der Kategorien im Ausgabefenster von Visual Studio protokolliert werden.

Programmcode von frmTest:

```

Public Class frmTest
    Inherits System.Windows.Forms.Form

    Private myShowCreationDestroy As Boolean

```

```

Private myShowMouse As Boolean
Private myShowKeyboard As Boolean
Private myShowPositioning As Boolean
Private myShowRepaintAndLayout As Boolean
Private myShowFocussing As Boolean
Private myShowPreProcessing As Boolean
Private myShowWndProcMessages As Boolean

'<Vom Windows Form Designer generierter Code (ausgeblendet)>

Public Sub New(ByVal ShowCreationDestroy As Boolean, _
              ByVal ShowMouse As Boolean, _
              ByVal ShowKeyboard As Boolean, _
              ByVal ShowPositioning As Boolean, _
              ByVal ShowRepaintAndLayout As Boolean, _
              ByVal ShowFocussing As Boolean, _
              ByVal ShowPreProcessing As Boolean, _
              ByVal ShowWndProcMessages As Boolean)
    MyBase.New()
    myShowCreationDestroy = ShowCreationDestroy
    myShowMouse = ShowMouse
    myShowKeyboard = ShowKeyboard
    myShowPositioning = ShowPositioning
    myShowRepaintAndLayout = ShowRepaintAndLayout
    myShowFocussing = ShowFocussing
    myShowPreProcessing = ShowPreProcessing
    myShowWndProcMessages = ShowWndProcMessages
    InitializeComponent()
End Sub

```

Steuerungscode von TestControl:

```

Public Class TestControl
    Inherits Control

    Private myShowCreationDestroy As Boolean
    Private myShowMouse As Boolean
    Private myShowKeyboard As Boolean
    Private myShowPositioning As Boolean
    Private myShowRepaintAndLayout As Boolean
    Private myShowFocussing As Boolean
    Private myShowPreProcessing As Boolean

    'Standardkonstruktor: Basiskonstruktor aufrufen
    Public Sub New()
        MyBase.new()
    End Sub

    'Erweiterter Konstruktor: Parameter für die Debug-Ausgaben setzen.
    Public Sub New(ByVal ShowCreationDestroy As Boolean, _
                  ByVal ShowMouse As Boolean, _
                  ByVal ShowKeyboard As Boolean, _
                  ByVal ShowPositioning As Boolean, _
                  ByVal ShowRepaintAndLayout As Boolean, _
                  ByVal ShowFocussing As Boolean, _

```

```
ByVal ShowPreProcessing As Boolean)
 MyBase.New()
 myShowCreationDestroy = ShowCreationDestroy
 myShowMouse = ShowMouse
 myShowKeyboard = ShowKeyboard
 myShowPositioning = ShowPositioning
 myShowRepaintAndLayout = ShowRepaintAndLayout
 myShowFocussing = ShowFocussing
 myShowPreProcessing = ShowPreProcessing
End Sub
'Wird von OnBackgroundPaint aufgerufen, damit der Hintergrund
'des Controls gelöscht wird. Zeichnet hier im Beispiel
'einen gelben Hintergrund.
Protected Overridable Sub DrawControlBackground(ByVal g As Graphics)
    Dim locBrush As New SolidBrush(Color.Yellow)
    g.SetClip(Me.ClientRectangle, Drawing2D.CombineMode.Replace)
    g.FillRectangle(locBrush, Me.ClientRectangle)
End Sub

'Wird von OnPaint aufgerufen, damit das TestControl einen sichtbaren Inhalt hat.
'Zeichnet hier im Beispiel ein umrandetes Kreuz mit einer bestimmten Stiftdicke,
'die von der Fokussierung der Komponente abhängig ist.
Protected Overridable Sub DrawControl(ByVal g As Graphics)
    Dim locPenWidth As Integer
    Dim locClientRecPenWidthIncluded As Rectangle

    'Wenn das Control fokussiert ist,
    If Me.Focused Then
        locPenWidth = 4
    Else
        locPenWidth = 2
    End If

    'Die Dicke des Pens bei den Koordinaten berücksichtigen!
    locClientRecPenWidthIncluded = New Rectangle(
        Me.ClientRectangle.X + locPenWidth \ 2, -
        Me.ClientRectangle.Y + locPenWidth \ 2, -
        Me.ClientRectangle.Width - locPenWidth, -
        Me.ClientRectangle.Height - locPenWidth)

    'Pen zum Malen.
    Dim locPen As New Pen(Color.Black, locPenWidth)

    'Rahmen zeichnen.
    g.DrawRectangle(locPen, locClientRecPenWidthIncluded)

    'Kreuz malen.
    g.DrawLine(locPen, locClientRecPenWidthIncluded.X, locClientRecPenWidthIncluded.Y, -
        locClientRecPenWidthIncluded.Right, locClientRecPenWidthIncluded.Bottom)

    g.DrawLine(locPen, locClientRecPenWidthIncluded.Right, locClientRecPenWidthIncluded.Y, -
        locClientRecPenWidthIncluded.X, locClientRecPenWidthIncluded.Bottom)

End Sub
```

Programmcode von frmMain:

```
Public Class frmMain
    Inherits System.Windows.Forms.Form

    '<Vom Windows Form Designer generierter Code (ausgeblendet)>

    Private Sub btnCreateWithTestControl_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles btnCreateWithTestControl.Click
        Dim locfrmTest As frmTest
        Dim locTestControl As TestControl
        Dim locButton As Button

        'Einstellungen gelten für das Formular...
        If chkFormular.Checked Then
            locfrmTest = New frmTest(
                chkCreateDestroy.Checked, chkMouse.Checked, chkKeyboard.Checked,
                chkPositioning.Checked, chkRepaintLayout.Checked, chkFocussing.Checked,
                chkPreProcessing.Checked, chkWndProcMessages.Checked)
        Else
            locfrmTest = New frmTest
        End If

        '...und/oder für die TestButton-Komponente.
        If chkSchaltfläche.Checked Then
            locTestControl = New TestControl(
                chkCreateDestroy.Checked, chkMouse.Checked, chkKeyboard.Checked,
                chkPositioning.Checked, chkRepaintLayout.Checked, chkFocussing.Checked,
                chkPreProcessing.Checked)
        Else
            locTestControl = New TestControl
        End If

        'Einstellungen für das Formular durchführen.

        'Das Formular hat ein Viertel der Bildschirmgröße
        'und soll in der Bildschirmmitte des primären Bildschirms erscheinen.
        Dim locBounds As Rectangle = Screen.PrimaryScreen.Bounds
        locfrmTest.Width = locBounds.Width \ 4
        locfrmTest.Height = locBounds.Height \ 4
        locfrmTest.StartPosition = FormStartPosition.CenterScreen
        locfrmTest.Text = txtFormText.Text

        'Einstellungen entsprechend der CheckBox-Controls im Formular
        locfrmTest.ControlBox = chkControlBox.Checked
        locfrmTest.MinimizeBox = chkMinimizeBox.Checked
        locfrmTest.MaximizeBox = chkMaximizeBox.Checked
        locfrmTest.HelpButton = chkHelpButton.Checked
        locfrmTest.ShowInTaskbar = chkShowInTaskbar.Checked
        locfrmTest.TopMost = chkTopMost.Checked
        locfrmTest.KeyPreview = chkKeyPreview.Checked
        locfrmTest.AutoScroll = chkScrollBars.Checked

        'TestControl mittig und im oberen Drittel des Formulars platzieren
```

```

' - nicht zu klein oder zu groß.
locTestControl.Width = CInt(locfrmTest.ClientSize.Width / 2)
locTestControl.Height = CInt(locfrmTest.ClientSize.Height / 3)
locTestControl.Location =
    New Point(CInt(locfrmTest.ClientSize.Width / 2 - locTestControl.Width / 2),
              CInt(locfrmTest.ClientSize.Height / 3 - locTestControl.Height / 2))

'TestControl verankern, wenn die AutoScroll-Funktion des Formulars nicht gewünscht wird
If Not chkScrollBars.Checked Then
    locTestControl.Anchor = AnchorStyles.Bottom Or AnchorStyles.Top Or _
                           AnchorStyles.Left Or AnchorStyles.Right
End If

'Schließschaltfläche im unteren Drittel positionieren.
locButton = New Button
locButton.Width = CInt(locfrmTest.ClientSize.Width / 3)
locButton.Height = CInt(locfrmTest.Height / 6)
locButton.Location =
    New Point(CInt(locfrmTest.ClientSize.Width / 2 - locButton.Width / 2),
              CInt(locfrmTest.ClientSize.Height / 4 * 3 - locButton.Height / 2))

locButton.Text = "Formular &schließen"
'Schließschaltfläche verankern, wenn die AutoScroll-Funktion des Formulars nicht gewünscht wird.
If Not chkScrollBars.Checked Then
    locButton.Anchor = AnchorStyles.Bottom Or _
                      AnchorStyles.Left Or AnchorStyles.Right
End If

'Return und Escape lösen Click-Ereignis des Buttons aus.
Me.AcceptButton = locButton
Me.CancelButton = locButton

'Zur Laufzeit einstellen, dass das Click-Ereignis der Schließschaltfläche
'ven TestButton-Click behandelt wird.
AddHandler locButton.Click, AddressOf TestButton_Click

'Beide Controls der Formular-ControlCollection hinzufügen;
'damit werden die beiden Komponenten windowstechnisch angelegt und dargestellt.
locfrmTest.Controls.Add(locTestControl)
locfrmTest.Controls.Add(locButton)

locfrmTest.Show()
End Sub

'Ereignis-Routine des Buttons; wird zur Laufzeit eingebunden (s.o.).
Sub TestButton_Click(ByVal Sender As Object, ByVal e As EventArgs)

    Dim locButton As Button

    'Könnte schief gehen, wenn Sender nicht die Schaltfläche ist,
    'deswegen sichergehen durch Try/Catch.
    Try
        'Das sendende Objekt herausfinden
        locButton = DirectCast(Sender, Button)
    Catch ex As Exception

```

```
    Return
End Try
'Sendendes Objekt war der Button selbst, dann dessen Parent (das Formular) entsorgen.
'Damit wird das Formular, das den Button enthält, geschlossen.
locButton.Parent.Dispose()
End Sub
End Class
```

Anmerkungen zum Beispielprogramm

Dem einen oder anderen unter Ihnen könnte es sich nicht auf den ersten Blick erschließen, wie der Inhalt von TestControl letzten Endes auf den Bildschirm gelangt. Die Zeichenroutinen sind ja offensichtlich die, die sich in OnPaint befinden. Das würde ja bedeuten, dass das Steuerelement sich jedes Mal neu zeichnen muss, wenn es beispielsweise durch ein anderes Fenster zuvor verdeckt wurde – und man könnte meinen, dass das sehr lange dauerte. Die Frage, die sich aus diesem Grund vielleicht stellt: Gibt es keinen Weg, den Inhalt von TestControl nur einmal zeichnen zu müssen, und der bleibt dann bestehen?

Genau das ist aber die Vorgehensweise, wenn Sie benutzerdefinierte Inhalte in Formularen oder sichtbaren Komponenten (Steuerelementen) anzeigen lassen wollen. Wenn Sie – wie in einem vorherigen Beispiel zu sehen war – beispielsweise die Image-Eigenschaft des PictureBox-Steuerelements bestimmen, brauchen Sie sich um das ständige Neuzeichnen nicht selbst zu kümmern. Dafür muss das PictureBox in der OnPaint-Prozedur selbst erledigen. Es gibt unter Windows keine festen Inhalte in *der Form*. Wenn ein Fenster von einem anderen überdeckt wird und dann wieder sichtbar wird, muss es seinen Inhalt neu zeichnen – diese Vorgehensweise muss jedes Windows-Programm anwenden, egal in welcher Sprache es entwickelt wurde. Selbst wenn es so aussieht, als sei der Inhalt »fest«, gibt es dennoch irgendwo eine Routine, die dafür sorgt, dass er nur »fest wirkt«. Das nächste Kapitel und Kapitel 40 verraten Ihnen weitere Geheimnisse zu diesem Thema.

Kapitel 39

Individuelles Gestalten von Elementinhalten mit GDI+

In diesem Kapitel:

Einführung in GDI+	1141
Flimmerfreie, fehlerfreie und schnelle Darstellungen von GDI+-Zeichnungen	1155
Was Sie beim Zeichnen von breiten Linienzügen beachten sollten	1161

GDI+ ist eine Klassenbibliothek, die das Zeichnen verschiedener Elemente in Windows (Windows im Sinne vom Windows-Betriebssystem) erlaubt. GDI steht als Abkürzung von *Graphic Device Interface* – etwa *grafische Geräteschnittstelle* – und das Pluszeichen hinter dem Akronym lässt schon vermuten, dass es sich um etwas Weiterentwickeltes handeln muss.

Allerdings ist das im Grunde genommen nur halb richtig – jedenfalls in der gegenwärtigen Version des GDI+. GDI+ ist der von Microsoft erklärte Nachfolger des GDI, der, wenn auch in ständig weiterentwickelter Form, schon zu 16-Bit-Windows-Zeiten sozusagen den ausführenden Produzenten für alles darstellte, was in irgendeiner Form auf dem Bildschirm erscheinen sollte. Aus diesem Grund erfuhren viele Grundfunktionen des GDI eine umfangreiche Unterstützung durch die Treiber von Grafikkarten der verschiedensten Hersteller. Das heißt im Klartext: Wenn Sie eine Linie von einem zum anderen Punkt auf dem Bildschirm mit GDI zeichnen, dann ist es nicht eine bestimmte Prozedur im GDI, die die eigentlichen Punkte setzt, sondern die Grafikkarte selbst, die diese Aufgabe übernimmt. Das GDI teilt dem Treiber lediglich mit, dass es eine Linie gezeichnet haben möchte. Und genau das ist zurzeit noch der Unterschied zum GDI+. Es bietet viel umfassendere und vor allen Dingen auch einfacher zu handhabende Zeichenfunktionen, doch viele davon sind momentan noch nicht hardwareunterstützt. Und das ist der Grund, weswegen Microsoft mit GDI+ grundsätzlich auf dem richtigen Weg ist, es sich für einige wenige Anwendungen unter Windows aus Geschwindigkeitsgründen aber einfach noch nicht hundertprozentig zur Verwendung eignet.

Nun ist das Framework für die Zukunft konzipiert, und in zukünftigen .NET- und Windows-Betriebssystemversionen werden sich definitiv leistungsfähigere Werkzeuge für die Darstellung von Grafik befinden – die *Windows Presentation Foundation* (auch bekannt unter seinem Codenamen »Avalon«) wird es nach Fertigstellung dieses Buchs sowohl noch für Windows XP als natürlich auch für den Windows XP-Nachfolger Windows Vista geben. Aus diesem Grund hat es wohl für die Entwickler des Frameworks bislang keinen Sinn ergeben, die ältere Version des GDI (ohne Plus) als Klassenlibrary in .NET zu implementieren.

Außer Acht gelassen, was die wirklichen Gründe für die Entscheidung zu diesem Schritt waren, gilt: Es ist derzeit jedenfalls nicht vorhanden, und es wird aller Wahrscheinlichkeit nach auch niemals implementiert werden. Für den Moment müssen wir für den Inhalt der Dokumente unserer .NET-Programme mit der aktuellen Version des GDI+ vorlieb nehmen.¹

HINWEIS GDI+ liefert genug Stoff, um ein eigenes Buch damit zu füllen. Charles Petzold hat das mit seiner Core-Reference,² die ein wenig GDI+-lastig geworden ist, eindrucksvoll unter Beweis gestellt. Aus diesem Grund möchte ich mich an dieser Stelle nur auf grundlegendste Erklärungen zum GDI+ beschränken – gerade auf so viel, dass es für andere Projekte ausreicht – um beispielweise Benutzersteuerelemente mit sinnvollen Inhalten zu füllen. In den nächsten Abschnitten finden Sie deswegen zunächst die wichtigsten Informationen, die Sie als Grundlage benötigen, um die ersten Gehversuche mit dem GDI+ bewältigen zu können, in sehr kompakter Form. Aus Platzgründen möchte ich mich nicht in endlos langen Beschreibungen von

¹ Natürlich gibt es auch die Möglichkeit, direkte Betriebssystemaufrufe an das herkömmliche GDI aus .NET heraus durchzuführen – doch das sind dann grundsätzlich so genannte *Unsafe Calls* (unsichere Aufrufe). Unsicher deswegen, weil das Programm das behütete, sichere .NET-Zuhause verlässt. Und solche Schritte sind nur in einer voll vertrauenswürdigen Umgebung erlaubt. Das heißt, dass alle Programme, die unsichere Aufrufe durchführen oder sonst irgendeinen unsicheren Code ausführen, mit den Standardsicherheitseinstellungen nur auf dem Computer selbst ausgeführt werden können. Starten Sie ein solches Programm beispielsweise von einer Netzwerk-Ressource, löst es beim Erreichen des unsicheren Codes eine Sicherheitsausnahme aus.

² Ein sehr empfehlenswertes Buch, gerade wenn es um GDI+ und native Textausgabe-/formatierung geht. Englische Ausgabe (.NET 1.0/1.1), ISBN: 0-7356-1799-6. Deutsche Ausgabe: Windows-Programmierung mit Visual Basic .NET, ISBN: 3-86063-691-X.

einzelnen Funktionen des GDI+ verlieren. Vielmehr möchte ich mich auf die *Anwendung* des GDI+ zur Lösung bestimmter Probleme beim Entwickeln von Komponenten und Anwendungen konzentrieren. Ersatzweise verwenden Sie für Fragen, die die korrekte Anwendung einer Methode oder eines Konstruktors betreffen, die Online-Hilfe von Visual Studio. Sie leistet gerade beim GDI+ als Referenz ausgezeichnete Arbeit. Durch die IntelliSense-Funktion des Editors werden selbst viele Blicke in die Online-Hilfe überflüssig.

Einführung in GDI+

Wie schon an mehreren Stellen in diesem Buch beschrieben, ist alles, was Sie in ein Window zeichnen, hoch volatil. Es hält genau so lange, wie Sie etwas anderes über das Fenster schieben. Dann ist sein Inhalt verschwunden. Das gilt sogar für das Fenster selbst, das den Inhalt darstellt: Wenn Sie ein Fenster in einer bestimmten Größe auf den Bildschirm gebracht haben, seinen Inhalt zeichnen, es anschließend verkleinern und wieder auf seine alte Größe bringen, ist der Ursprungsinhalt ebenfalls verloren.

Das liegt am Prinzip, wie Windows-Inhalte gemanagt werden: Jedes Fenster ist für das Zeichnen seines Inhaltes selbst verantwortlich. Alle auf Control basierenden Komponenten (dazu gehören auch Formulare) müssen deswegen ihr Paint-Ereignis behandeln oder noch besser, da schneller, die OnPaint-Methode ihrer Basisklasse überschreiben. In beiden Fällen wird den Prozeduren das so genannte Graphics-Objekt mit dem PaintEventArgs-Objekt übergeben, das den Dreh- und Angelpunkt für alle Grafikoperationen bildet.

Die Zeichenroutinen, die dieses Graphics-Objekt nun zur Verfügung stellt, beziehen sich auf den so genannten *Client*-Bereich des Fensters. Das ist das Innere des Fensters, also der Bereich, ohne Rahmen, Titel oder Rollbalken.

Sie können ein Graphics-Objekt übrigens nicht selbst direkt durch seinen Konstruktor erstellen; sie können es nur durch bestimmte Funktionen ermitteln – der Graphics-Parameter, der Ihnen durch OnPaint mit PaintEventArgs geliefert wird, ist nur ein (wenn auch das am häufigsten auftretende) Beispiel dafür.

Sie haben zwar auch die Möglichkeit, das für eine Control-Ableitung gültige Graphics-Objekt auch ohne die OnPaint-Ereignisparameter zu bekommen, doch ist das Anwenden dieser Vorgehensweise eher selten der Fall. Außerdem führt es oft zu einer falschen Vorgehensweise, wie das folgende Beispiel zeigt:

BEGLEITDATEIEN

Sie finden dieses Projekt im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 39\\SimpleGdi01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Public Class frmMain

    Private Sub btnLinieZeichnen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnLinieZeichnen.Click

        Dim g As Graphics

        'Ermittelt das Graphics-Objekt, das zu einem bestimmten
        'Window gehört, dessen Handle (ID) zur Identifizierung dient.
        g = Graphics.FromHwnd(Me.Handle)
```

```
'Schwarze, ein Pixel dünne Linie zeichnen von (0,0) zu (500,500).
'Koordinaten werden standardmäßig in Pixel angegeben;
'(0,0) liegt in der linken, oberen Ecke des Client-Bereichs.
g.DrawLine(New Pen(Color.Black), 0, 0, 500, 500)

End Sub

End Class
```

Wenn Sie dieses Programm starten, werden Sie nach dem Anklicken der einzigen Schaltfläche zwei Dinge feststellen: a) Das ermittelte Graphics-Objekt bezieht sich tatsächlich nur auf das Fenster, für das es ermittelt wurde. Denn obwohl der Linienpfad die Schaltfläche kreuzt, zeichnet der Befehl die Linie im Bereich der Schaltfläche nicht (siehe Abbildung 39.1). Und b) Wenn Sie irgendetwas über das Formular bewegen (der Windows-Taschenrechner eignet sich für solche Experimente am besten), ist die Linie verschwunden.

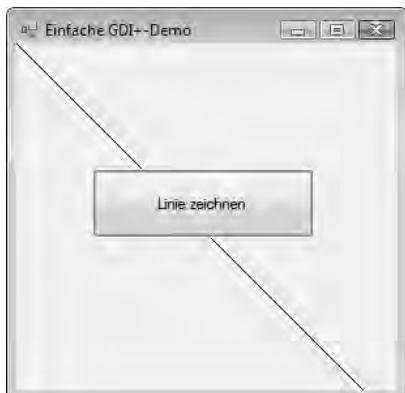


Abbildung 39.1 Zeichenoperationen mit einem Graphics-Objekt beziehen sich nur auf das Fenster, für das das Graphics-Objekt ermittelt wurde

Wenn Sie wollen, dass die Linie auch noch vorhanden ist, *nachdem* sich ein anderes Objekt über dem Formular befunden hat, müssen Sie ein anderes Verfahren verwenden, das grob skizziert folgendermaßen funktioniert:

- Alle Zeichenroutinen finden in der Paint-Ereignisbehandlungsroutine des Formulars statt. Dazu können Sie – wie schon gesagt – entweder das Ereignis im Formular einbinden oder, der zu empfehlende Weg, die Basisprozedur OnPaint überschreiben, die das Zeichnen der Linie vornimmt.
- Damit die Linie nur dann gezeichnet wird, wenn die Schaltfläche vom Anwender angeklickt wurde, muss es eine Art Informationsspeicher geben (ein Flag, das von überall im Formular zugänglich ist – also einen Klassen-Member), der OnPaint darüber informiert, ob die Linie im Falle eines neu zeichnen Müssens überhaupt gemalt werden soll.
- Damit die Linie auch dann gezeichnet wird, wenn der Anwender die Schaltfläche angeklickt hat, muss der Code zur Behandlung des Klickereignisses für die Schaltfläche nicht nur das Flag setzen, sondern auch den kompletten Client-Bereich des Formulars für ungültig erklären. Geschieht das mit einer bestimmten Methode namens Invalidate, wird automatisch das Paint-Ereignis ausgelöst, damit OnPaint aufgerufen und die Linie gezeichnet werden kann.

BEGLEITDATEIEN

Sie finden das Projekt, das diese Modifizierungen enthält im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 39\\SimpleGdi02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Der dazugehörige Formularcode sieht folgendermaßen aus:

```
Public Class frmMain
    Inherits System.Windows.Forms.Form
    Private myDrawLineFlag As Boolean

    'Klassenweiter Member, der von Click und OnPaint aus zugänglich ist.
    Private myDrawLineFlag As Boolean

    Private Sub btnLinieZeichnen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
        Handles _
            btnLinieZeichnen.Click

        'Ab sofort darf gezeichnet werden!
        myDrawLineFlag = True
        'Client-Bereich für ungültig erklären --> OnPaint wird ausgelöst,
        'und damit, da myDrawLineFlag jetzt true ist, die Linie gezeichnet.
        Me.Invalidate()

    End Sub

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)

        If myDrawLineFlag Then
            Dim g As Graphics = e.Graphics

            'Ermittelt das Graphics-Objekt, das zu einem bestimmten
            'Window gehört, dessen Handle (ID) zur Identifizierung dient.
            g = Graphics.FromHwnd(Me.Handle)

            'Schwarze, ein Pixel dünne Linie zeichnen von (0,0) zu (500,500).
            'Koordinaten werden standardmäßig in Pixel angegeben;
            '(0,0) liegt in der linken, oberen Ecke des Client-Bereichs.
            g.DrawLine(New Pen(Color.Black), 0, 0, 500, 500)
        End If
    End Sub
End Class
```

Wenn Sie diese Version des Programms starten, funktioniert es auf den ersten Blick wie die erste Version des Programms, mit einem entscheidenden Unterschied: Das Formular zeichnet sich in den entscheidenden Schritten neu, und das ist schließlich auch genau das, was es muss.

Sie sollten dieses einfache Prinzip in Ihren eigenen Anwendungen berücksichtigen, wenn Sie selbst gezeichnete Inhalte in Formularen darstellen müssen.

Linien, Flächen, Pens und Brushes

Im Beispiel des letzten Abschnittes haben Sie bereits teilweise sehen können, dass für alle Zeichenoperationen gilt: Sie benötigen entweder ein Pen-Objekt (einen Stift) oder ein Brush-Objekt (einen Pinsel), um Linien oder Flächen zu zeichnen.

Für Linienfiguren benötigen Sie ein Pen-Objekt. Ein Pen bestimmt, wie ein Linienzug gezeichnet werden soll – oder genauer: in welcher Farbe oder welcher Stärke das geschehen soll. Wie die Anwendung des Pen-Objektes generell funktioniert, konnten Sie bereits im vergangenen Beispiel sehen. Der Pen-Konstruktor verfügt über mehrere Überladungen. Neben Farbe und Liniendicke haben Sie die Möglichkeit, ein Pen-Objekt auch aus einem Brush-Objekt zu erstellen und so dickere Linien oder Umrisse bestimmter geometrischer Figuren mit einer spezifischen Füllung zu versehen.

Brush-Objekte in GDI+ sind vielseitig. Sie dienen der Bestimmung der Eigenschaften von Flächenfüllungen. Das Brush-Objekt selbst ist, anders als ein Pen, eine abstrakte Basisklasse, es kann also nicht direkt instanziert und verwendet werden. Stattdessen gibt es in GDI+ fünf verschiedene Brush-Ableitungen, die unterschiedliche Aufgaben beim Erstellen von Füllfarben bzw. -mustern für Flächen erfüllen:

- **SolidBrush:** Definiert einen simplen, einfarbigen Pinsel. Flächen, die Sie mit einem SolidBrush füllen, werden mit der durch den Brush bestimmten Farbe ausgefüllt.
- **HatchBrush:** Definiert einen rechteckigen Pinsel mit einer Schraffurart; Vorder- und Hintergrundfarbe der Schraffur sind dabei getrennt einstellbar.
- **TextureBrush:** Definiert einen Pinsel aus einer Bitmap. In welcher Form die Bitmap als Zeichenvorlage für eine Füllung verwendet wird, stellen Sie über den Konstruktor ein. Interessantes hierzu finden Sie in der VS.NET-Onlinehilfe unter dem `ImageAttributes`-Objekt und der `WrapMode`-Enum.
- **LinearGradientBrush:** Definiert ein Brush-Objekt mit einem linearen Farbverlauf.
- **PathGradientBrush:** Definiert ein Brush-Objekt mit einem Farbverlauf, der durch einen beliebigen Kurvenverlauf (`GraphicsPath`) bestimmt wird. Ein solcher Pinsel entsteht entweder aus einem `GraphicsPath`-Objekt oder einem Points- bzw. `PointF`-Array, das die Eckpunkte des Kurvenverlaufs bestimmt.

Grundsätzlich gilt: Wenn Sie eine Linie, einen Linienzug, einen Polygonumriss oder einen Kreisumriss zeichnen möchten, verwenden Sie eine der `DrawXXX`-Methoden des `Graphics`-Objektes. Sie benötigen zum Zeichnen in diesem Fall ein `Pen`-Objekt, das die Eigenschaften des verwendeten Stiftes bestimmt.

Möchten Sie eine Fläche füllen, verwenden Sie eine der `FillXXX`-Methoden des `Graphics`-Objektes. Sie benötigen zum Zeichnen dann eines der von Brush abgeleiteten Objekte, das die Eigenschaften der Füllung bestimmt.

Angabe von Koordinaten

Die meisten Zeichenfunktionen, die Ihnen das `Graphics`-Objekt zur Verfügung stellt, arbeiten mit der Angabe von Koordinaten, die Ihnen entweder als Single- oder als Integer-Werte übergeben werden. Der Koordinatenursprung liegt in der linken, oberen Ecke des Bildschirms. Neben der Bestimmung von Koordinaten durch einzelne Single- oder Integer-Werte hält das Framework einige Strukturen bereit, um Sie bei der Angabe von Positionen, Umrissen oder Größen zu unterstützen. Die wichtigsten sind:

- **Point:** Die Point-Struktur dient zur Angabe einer Koordinate. Sie übergeben ihr im Konstruktor den X- und Y-Wert als Integerwert.

- **PointF:** Die PointF-Struktur dient zur Angabe einer Koordinate mit Fließkommawerten. Sie übergeben ihr im Konstruktor den X- und Y-Wert jeweils als Wert vom Typ Single.
- **Size:** Die Size-Struktur dient zur Angabe des Ausmaßes eines rechteckigen Objektes. Sie übergeben ihr im Konstruktor die Breite und Höhe als Integer-Wert.
- **SizeF:** Die SizeF-Struktur dient zur Angabe des Ausmaßes eines rechteckigen Objektes mit Fließkommawerten. Sie übergeben ihr im Konstruktor die Breite und Höhe jeweils als Wert vom Typ Single.
- **Rectangle:** Diese Struktur dient zur Angabe von Position und Ausmaßen eines Rechtecks. Ein Rectangle definiert sich durch einen Startpunkt, der als Koordinate angegeben wird, sowie durch die Höhe und die Breite.
- **RectangleF:** Es gilt das zu Rectangle Gesagte, nur dass die Werte als Fließkommawert vom Typ Single angegeben werden.

Wieso Integer- und Fließkommaangaben für Positionen und Ausmaße?

Das Graphics-Objekt ist nicht auf Pixel als feste Maßeinheit für die Positions- bzw. Größenangaben festgelegt. Vielmehr erlauben die PageUnit-Eigenschaft, die PageScale-Eigenschaft sowie verschiedene Transformationsmethoden eine individuelle Skalierung des Koordinatensystems. Integerwerte sind dann natürlich nicht mehr ausreichend, wenn Sie beispielsweise *Inches* (englische Bezeichnung für die Maßeinheit Zoll, entspricht 2,54 cm) als Maßeinheit bestimmt haben. Auf dem Bildschirm liegen zwischen zwei Pixel, die einen Inch auseinander liegen, natürlich viele weitere Pixel. Die dazwischen liegenden Punkte ließen sich nicht erreichen, könnte man keine gebrochenen Werte für Koordinaten angeben.

Beide Verfahren haben Vor- und Nachteile: Wenn Sie sich für Pixel als Maßeinheit entscheiden (das ist die Standardeinstellung, die durch PageUnit bestimmt wird), können Sie Integer-Werte verwenden; interne Skalierungsumrechnungen entfallen dabei, allerdings können Sie damit nur bestimmte Anwendungen realisieren, wie beispielsweise die Begrenzungen eigener Steuerelemente zeichnen, deren Ausmaße ohnehin in Pixel angegeben werden.

Entscheiden Sie sich für eine andere Maßeinheit, wie beispielsweise Millimeter oder Inch, müssen Sie Fließkommawerte verwenden, um alle Pixel im Koordinatensystem ansteuern zu können. Intern finden natürlich wieder Umrechnungen auf die eigentlichen, physischen Pixelkoordinaten statt, und das kostet ein wenig Rechenzeit. Allerdings ist die Anwendung von Maßeinheiten, die Sie aus der »echten« Welt kennen, für die Umsetzung vieler Anwendungen leichter und flexibler.

Wie viel Platz habe ich zum Zeichnen?

Um die zur Verfügung stehenden Ausmaße eines Formulars (oder einer von Control abgeleiteten Klasse) zu ermitteln, gibt es zwei Möglichkeiten.

- Wenn Ihnen das Graphics-Objekt bekannt ist, verwenden Sie die Nur-Lesen-Eigenschaft VisibleClipbounds des Grafikobjektes. Der Vorteil: Wenn Sie mit der PageUnit-Eigenschaft eine andere Maßeinheit für das Koordinatensystem eingestellt haben, liefert VisibleClipbounds die Ausmaße in den Einheiten zurück, die durch PageUnit eingestellt sind.
- Mit ClientArea des Formulars oder der verwendeten Control-Klasse (oder deren Ableitung) ermitteln Sie den sichtbaren Zeichenbereich in Pixel. Sie benötigen dazu kein Graphics-Objekt.

Das gute, alte Testbild und GDI+ im Einsatz sehen!

Nach so viel grauer Theorie sind Sie sicherlich gespannt, GDI+-Funktionen in der Praxis zu sehen. Seit der Einführung von Kabel- und Satellitenfernsehen Mitte der 80er Jahre können wir uns über eine Unterversorgung mit farbigen Bildern nicht mehr beklagen. Allerdings: Ein bestimmtes Programm ist fast gänzlich von unseren Mattscheiben verschwunden – gemeint ist das gute, alte Testbild. Die Älteren unter Ihnen werden sich sicherlich noch erinnern können, wie es uns erging, wenn wir morgens nicht zur Schule wollten, weil dort entweder eine nicht zu packende Klassenarbeit oder der Kerl aus der 4. Klasse auf uns wartete. Diesem hatten wir tags zuvor sein Pausenbrot »frisiert«, gleichzeitig aber nicht bedacht, dass wir zwar der Held des Tages aber auch ein potentieller Kandidat für die Notaufnahme am nächsten Tag waren. Also hieß es: Fieberthermometer in den Tee, Seifenwasser in die Augen (zwei Minuten mit aufgerissenen Lidern ohne zu blinzeln reichten meist auch aus, um die notwendige Augenrötung herbeizuführen) und nach einigen, Oscar-verdächtigen Jammereinlagen ging's ab aufs elterliche Sofa, wo die Flimmerkiste – gerade in Farbe – schon auf ein gut sortiertes Programm wartete: Telekolleg im Dritten, Testbild im Zweiten und Testbild im Ersten.

Und dann hieß es: Sehnsüchtig auf die Sesamstraße um halb zehn warten. Das Testbild, geben Sie es zu, hassten Sie damals sicherlich, wie ich es tat. Und jetzt? Jetzt verbinden wir alle wahrscheinlich kindliche Unbeschwertheit mit exakt dieser Grafik, und wäre es nicht schön, solche Momente nochmals erleben zu können?

Aber: Warum jammern? Wir haben Rechenpower, wir haben Visual Basic und wir haben GDI+! Holen wir es uns zurück!

BEGLEITDATEIEN

Sie finden das folgende Projekt unter:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 39\\VBTestbild01

Es trägt den Projektnamen *VBTestbild*, und wenn Sie meiner Generation (so ungefähr 1966–1972) angehören, werden Sie es noch kennen und lieben!

Wenn Sie dieses Programm starten, sehen Sie das Testbild, etwa wie in Abbildung 39.2 gezeigt. Sie können dieses Bild vergrößern und verkleinern, und Sie werden feststellen, dass sich sein Inhalt immer an die aktuelle Größe anpasst. Mal ganz abgesehen davon, dass es heftig flimmert (jedenfalls in dieser Version), ist das der Beweis, dass es sich bei der Grafik nicht um eine simple, im Internet geklauten Bitmap handelt, sondern jeder einzelne Strich, jede einzelne Fläche und jedes Polygon farbig und bunt zur Laufzeit gezeichnet wird. Und glauben Sie mir: Salopp gesagt, war das Kreieren dieses Programms eine tierische Fuckelei, aber ich finde, der Aufwand hat sich gelohnt.

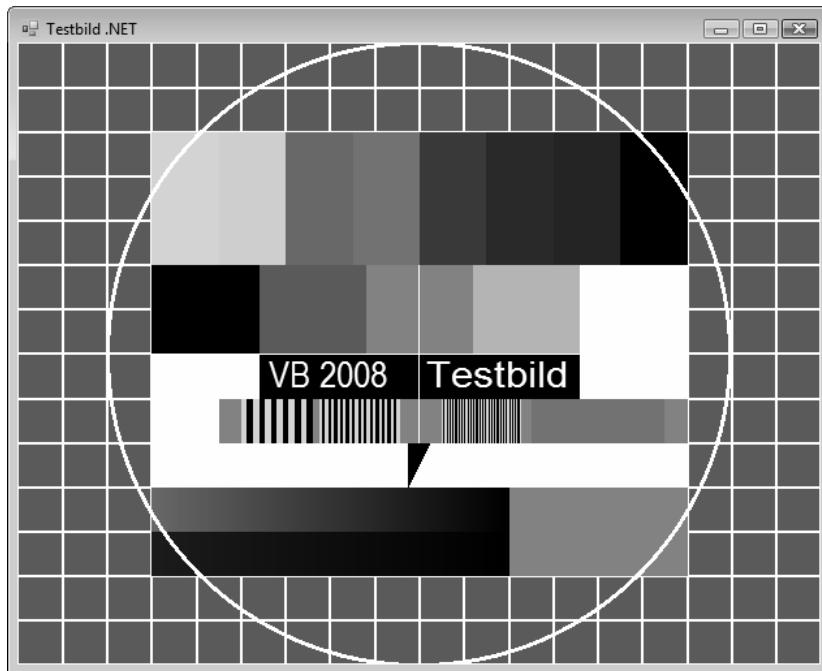


Abbildung 39.2 Für Erinnerungen an die kindliche Unbeschwertheit – das Testbild

Gelohnt, im Übrigen, nicht nur wegen des Ergebnisses, sondern auch, weil das alte Testbild der Öffentlichen-Rechtlichen (ob man damals schon so weit gedacht hat?) viele unterschiedliche Elemente enthielt, die den Einsatz von vielen verschiedenen GDI+-Funktionen erforderlich macht – Sie werden das gleich sehen.

Einige Worte zur Funktionsweise vorweg: Ich habe das Formular so gestaltet, dass es eher als Komponente denn als Formular fungiert. Aus diesem Grund gibt es einige Member-Variablen, zu denen sowohl äquivalente, als auch öffentliche Eigenschaften existieren. Das gibt uns zum Einen die Möglichkeit, das komplette Formular später abzuleiten und bestimmte Verhaltensweisen durch gezieltes Überschreiben von Eigenschaftenprozeduren zu beeinflussen. Zum Anderen können wir aus dem Formular auch ein Steuerelement machen, dessen Verhalten sich durch die öffentlichen Eigenschaften gezielt von außen steuern lässt. Aus Platzgründen finden Sie die meisten dieser Eigenschaften in der vorliegenden Version nicht abgedruckt, da sie die Inhalte der geschützten Member-Variablen, die das Zeichnen eigentlich steuern, nur nach oben durchreichen.

Das Codelisting:

```
Imports System.Drawing.Drawing2D  
  
Public Class frmMain  
    Inherits System.Windows.Forms.Form
```

Einige Funktionen des Formulars zum Zeichnen werden aus dem Namespace Drawing2D benötigt; deswegen die entsprechende Imports-Anweisung am Anfang des Codes. Die Formularklasse wird aus Form abgeleitet.

```
'Legt fest, ob das Gitter gezeichnet werden soll oder nicht.
Private myDrawGrid As Boolean
'Bestimmt, wie das Gitter gezeichnet werden soll.
Private myGridStyle As GridStyle
'Bestimmt die Stiftbreite für das Zeichnen des Gitters/Rasters.
Private myGridLineWidth As Integer
'Bestimmt die Stiftfarbe für das Zeichnen des Gitters/Rasters.
Private myGridColor As Color
'Bestimmt die Hintergrundfarbe.
Private myBackground As Color
'Bestimmt die Farben der oberen Balkenreihe.
Private myBarColors As Color()
'Bestimmt die Grauschattierungen der darunterliegenden Balkenreihe.
Private myBarShades As Color()
'Bestimmt die Grauschattierungen der Balkenreihe, in der sich die Beschriftung befindet.
Private myBarTitleShades As Color()
'Definiert die Anzahl der Gitter-/Rasterspalten.
Private myGridCols As Integer
'Definiert die Anzahl der Gitter-/Rasterzeilen.
Private myGridRows As Integer
'Definiert, in Rasterzeilen gerechnet, den Abstand des "Bildes" von oben.
Private myUpperOffset As Integer
'Definiert den Zeichenmodus.
Private mySmoothingMode As SmoothingMode
```

Die Member-Variablen der Klasse folgen anschließend. Sie bestimmen, in welcher Weise Gitter/Raster und eigentliches Bild gemalt werden sollen. Die Variablen selbst werden im Konstruktor des Programms initialisiert:

```
Public Sub New()
    MyBase.New()

    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()

    ' Initialisierungen nach dem Aufruf InitializeComponent() hinzufügen.
    DrawGrid = True
    GridStyle = GridStyle.Lines
    GridLineWidth = 2
    GridColor = Color.White

    '18 Spalten und 14 Zeilen
    GridCols = 18
    GridRows = 14

    'Bild beginnt in der 3. Reihe.
    UpperOffset = 2

    'Hintergrund ca. 70% grau
    Background = Color.FromArgb(90, 90, 90)
```

```
'Farben für die Farbbalken
BarColors = New Color() {Color.Pink, Color.GreenYellow, Color.Magenta, Color.Olive, _
                         Color.DarkMagenta, Color.DarkRed, Color.Indigo, Color.Black}
'Farben für die darunterliegenden Grauflächen
BarShades = New Color() {Color.FromArgb(0, 0, 0), _
                         Color.FromArgb(90, 90, 90),
                         Color.FromArgb(130, 130, 130), -
                         Color.FromArgb(180, 180, 180), -
                         Color.FromArgb(255, 255, 255)}
'Farben für die darunterliegenden Grauflächen der Titelzeile
BarTitleShades = New Color() {Color.White, -
                               Color.Black, -
                               Color.Black, -
                               Color.Black, -
                               Color.White}

'Fenstertitel
Text = "Testbild .NET"
End Sub
```

Einige Anmerkungen zur Angabe von Farben bei der Verwendung des *Graphics*-Objektes: Die *Color*-Struktur bietet eine elegante und einfache Möglichkeit, Farben zu definieren. Sie verfügt über zahlreiche, statische Funktionen, um Farbwerte anhand von Farbnamen zu ermitteln. Die Farb- und Graubalken, die Sie im Testbild sehen, haben zwar feste Ausmaße, aber Sie können die Anzahl der Balken, die sich in dieser Größenvorgabe befinden, variieren, indem Sie den entsprechenden Arrays Elemente hinzufügen oder aus ihnen entfernen.

```
'Wird aufgerufen, wenn das Bild aus irgendeinem Grund neu gezeichnet werden muss.
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    paintBackground(e.Graphics)
    paintContent(e.Graphics)
End Sub
```

Diese Routine wird automatisch aufgerufen, wenn ein Neuzeichnen des Bildes erforderlich wird. Sie zeichnet zunächst den Hintergrund des Bildes und anschließend das eigentliche Testbild. Diese Routinen sehen folgendermaßen aus:

```
'Hier wird der eigentliche Inhalt gezeichnet.
Private Sub paintContent(ByVal g As Graphics)

    Dim locPen As Pen
    Dim locBrush As Brush
    Dim locBarWidth As Single
    Dim locX As Single
    Dim locRectF As RectangleF

    g.SmoothingMode = mySmoothingMode

    'Raster malen im Bedarfsfall
    If DrawGrid Then
        paintGrid(g)
    End If
```

```
'Figuren malen:

'obere Balkenreihe mit Farbblöcken...
locRectF = New RectangleF(GridSize.Width * 3, GridSize.Height * myUpperOffset, _
    GridSize.Width * 12, GridSize.Height * 3)
locBarWidth = locRectF.Width / BarColors.Length / 1
For i As Integer = 0 To BarColors.Length - 1
    locBrush = New SolidBrush(myBarColors(i))
    locRectF.Width = locBarWidth
    g.FillRectangle(locBrush, locRectF)
    locRectF.X += locBarWidth
Next

'darunter liegende Balkenreihe mit Grauschattierungen...
locRectF = New RectangleF(GridSize.Width * 3, GridSize.Height * (UpperOffset + 3), _
    GridSize.Width * 12, GridSize.Height * 2)
locBarWidth = locRectF.Width / BarShades.Length / 1
For i As Integer = 0 To BarShades.Length - 1
    locBrush = New SolidBrush(myBarShades(i))
    locRectF.Width = locBarWidth
    g.FillRectangle(locBrush, locRectF)
    locRectF.X += locBarWidth
Next

'darunter liegende Balkenreihe mit Titel...
locRectF = New RectangleF(GridSize.Width * 3, GridSize.Height * (UpperOffset + 5), _
    GridSize.Width * 12, GridSize.Height)
locBarWidth = locRectF.Width / BarTitleShades.Length / 1
For i As Integer = 0 To BarTitleShades.Length - 1
    locBrush = New SolidBrush(myBarTitleShades(i))
    locRectF.Width = locBarWidth
    g.FillRectangle(locBrush, locRectF)
    locRectF.X += locBarWidth
Next

'darunter komplette zwei Reihen zunächst weiß...
locBrush = New SolidBrush(Color.White)
g.FillRectangle(locBrush, GridSize.Width * 3, GridSize.Height * (UpperOffset + 6), _
    GridSize.Width * 12, GridSize.Height * 2)

'jeweils zwei Farbverlaufsstreifen...
locRectF = New RectangleF(GridSize.Width * 3, GridSize.Height * (UpperOffset + 8), _
    GridSize.Width * 8, GridSize.Height)
locBrush = New LinearGradientBrush(locRectF, Color.Magenta, Color.Black, _
    LinearGradientMode.Horizontal)
g.FillRectangle(locBrush, locRectF)

'der zweite Farbverlaufsstreifen...
locRectF = New RectangleF(GridSize.Width * 3, GridSize.Height * (UpperOffset + 9), _
    GridSize.Width * 8, GridSize.Height)
locBrush = New LinearGradientBrush(locRectF, Color.Blue, Color.Black, _
    LinearGradientMode.Horizontal)
g.FillRectangle(locBrush, locRectF)
```

```
'grauer Kasten neben die Farbverläufe...
locBrush = New SolidBrush(Color.FromArgb(130, 130, 130))
g.FillRectangle(locBrush, GridSize.Width * 11, GridSize.Height * (UpperOffset + 8), _
    GridSize.Width * 4, GridSize.Height * 2)

'Geriffele malen...
'Zuerst grauer Kasten als Unterlage
g.FillRectangle(locBrush, GridSize.Width * 4.5F, GridSize.Height * (UpperOffset + 6), _
    GridSize.Width * 10.5F, GridSize.Height)

'dann das linke, größere Geriffele malen...
locRectF = New RectangleF(GridSize.Width * 5, GridSize.Height * (UpperOffset + 6), _
    GridSize.Width * 1.5F, GridSize.Height)

drawAreaCorrugated(g, locRectF, Color.Black, Color.LightGray, 2, GridSize.Width)

'das kleinere Geriffele rechts daneben malen...
locRectF.X = GridSize.Width * 6.75F
locRectF.Width = GridSize.Width * 1.75F
drawAreaCorrugated(g, locRectF, Color.Black, Color.LightGray, 1, GridSize.Width)

'das noch kleinere Geriffele rechts daneben malen...
locRectF.X = GridSize.Width * 9.5F
locRectF.Width = GridSize.Width * 1.75F
drawAreaCorrugated(g, locRectF, Color.Black, Color.LightGray, 0.5F, GridSize.Width)

'das Olivenfarbene daneben...
locRectF.X = GridSize.Width * 11.5F
locRectF.Width = GridSize.Width * 3
locBrush = New SolidBrush(Color.Olive)
g.FillRectangle(locBrush, locRectF)

'Zielkreuz Kreis in die Mitte...
locPen = New Pen(Color.White, 3)
g.DrawEllipse(locPen, ClientSize.Width \ 2 - ClientSize.Height \ 2, 0, ClientSize.Height, _
    ClientSize.Height)
locPen.Width = 1
g.DrawLine(locPen, GridSize.Width * 9, GridSize.Height * (UpperOffset + 3), _
    GridSize.Width * 9, GridSize.Height * (UpperOffset + 7))
g.DrawLine(locPen, GridSize.Width * 5, GridSize.Height * (UpperOffset + 5), _
    GridSize.Width * 13, GridSize.Height * (UpperOffset + 5))

'das kleine Dreieck in den weißen Zwischenraum...
Dim locPoints(2) As PointF
locPoints(0) = New PointF(GridSize.Width * 8.75F, GridSize.Height * (UpperOffset + 7))
locPoints(1) = New PointF(locPoints(0).X, GridSize.Height * (UpperOffset + 8))
locPoints(2) = New PointF(GridSize.Width * 9.25F, locPoints(0).Y)
g.FillPolygon(New SolidBrush(Color.Black), locPoints)

'Texte hineinschreiben.
drawStringInFrame(g, New SolidBrush(Color.White), "VB.NET", "Arial", _
    New RectangleF(GridSize.Width * 5.5F, GridSize.Height * (UpperOffset + 5), _
        GridSize.Width * 3, GridSize.Height))
```

```

        drawStringInFrame(g, New SolidBrush(Color.White), "Testbild", "Arial",
                           New RectangleF(GridSize.Width * 9, GridSize.Height * (UpperOffset + 5),
                                          GridSize.Width * 3.5F, GridSize.Height))
    End Sub

```

Diese Prozedur bildet den Kern des Programms – sie sorgt dafür, dass das Bild auch tatsächlich auf dem Bildschirm erscheint. Anhand dieser Routine können Sie sehen, wie Füllungen und Linienfiguren mit dem GDI+ gezeichnet werden, und Sie können ebenfalls erkennen, wie einfach die Handhabung von Grafikfunktionen mit dem GDI+ im Grunde genommen ist.

Komplexe Grafikfunktionen, wie beispielsweise das Zeichnen des »geriffelten« Bereiches oder das Platzieren des Textes in der richtigen Größe, sind in verschiedene Unteroutinen ausgelagert, die Sie im Folgenden beschrieben finden:

```

'Geriffelten Bereich malen.
Private Sub drawAreaCorrugated(ByVal g As Graphics, ByVal rectF As RectangleF, _
                               ByVal col1 As Color, ByVal col2 As Color,
                               ByVal relCellStepWidth As Single, ByVal relCellWidth As Single)

    Dim locPColl As New SolidBrush(col1)
    Dim locPCol2 As New SolidBrush(col2)
    Dim locCurrentBrush As Brush
    Dim locAltFlag As Boolean = False
    Dim locStep As Single = relCellWidth / (1 / relCellStepWidth * 15)
    For x As Single = rectF.X To rectF.Right Step locStep
        locCurrentBrush = DirectCast(IIf(locAltFlag, locPColl, locPCol2), SolidBrush)
        g.FillRectangle(locCurrentBrush, x, rectF.Y, locStep, rectF.Height)
        locAltFlag = Not locAltFlag
    Next
End Sub

'Zeichnet den Text so, dass er genau in ein Rechteck passt.
Private Sub drawStringInFrame(ByVal g As Graphics, ByVal brush As Brush, ByVal text As String, _
                               ByVal fontName As String, ByVal rectF As RectangleF)

    'Skalierungseinstellungen zum Wiederherstellen speichern.
    Dim locGState As GraphicsState = g.Save
    'Font mit 12 Pt. Höhe aus Fontnamen anlegen.
    Dim locFont As New Font(fontName, 12, FontStyle.Regular)
    'Ausmaße des Strings messen.
    Dim locSize As SizeF = g.MeasureString(text, locFont)
    'Faktoren für die Skalierung ermitteln, sodass der String...
    Dim locScaleV As Single = rectF.Height / locSize.Height
    '...genau in das angegebene Rechteck passt.
    Dim locScaleH As Single = rectF.Width / locSize.Width
    'Koordinatensystem verschieben
    g.TranslateTransform(rectF.X, rectF.Y)
    'KoordinatenSystem neu skalieren
    g.ScaleTransform(locScaleH, locScaleV)

```

```
'String im skalierten Koordinatensystem ausgeben.  
g.DrawString(text, locFont, brush, 0, 0)  
'Alte Skalierungs- und Transformationseinstellungen wiederherstellen.  
g.Restore(locGState)  
  
End Sub
```

Diese letzte Prozedur demonstriert den Einsatz von Text- und Skalierungsfunktionen. Für deren genaues Verständnis ist allerdings ein klein wenig mehr Hintergrundwissen erforderlich, das Ihnen der folgende kurze Exkurs liefert.

Exaktes Einpassen von Text mit GDI+-Skalierungsfunktionen

Um eine Zeichenkette in ein Graphics-Objekt auszugeben, benötigen Sie neben einem Font-Objekt, das den zu verwendenden Zeichensatz bestimmt, auch ein Brush-Objekt, das die Pinseleigenschaften darstellt und damit Farbe und Füllung definiert, mit denen der Text gezeichnet wird. Für die Größenbestimmung der auszugebenden Zeichenkette ist normalerweise ausschließlich die Größe des Zeichensatzes verantwortlich; GDI+ bietet Ihnen standardmäßig keine Möglichkeit, einen auszugebenden Text automatisch auf eine bestimmte Höhe oder Breite zu skalieren. Wenn Sie einen Text mit der `DrawString`-Funktion in ein Graphics-Objekt hineinschreiben, hat er damit genau die Größe, die sich durch das angegebene Font-Objekt und die Breite der im auszugebenden Text enthaltenen Buchstaben ergibt.

Allerdings bietet das GDI+ eine Reihe von Skalierungsfunktionen, mit denen das Koordinatensystem in beide Richtungen gedehnt oder gestaucht werden kann.

In Zusammenarbeit mit der `MeasureString`-Funktion, die die Ausmaße eines Textes ermitteln kann, *ohne* ihn dabei wirklich auszugeben, erlaubt das die genaue Dehnung/Stauchung des Koordinatensystems, sodass der Text – egal mit welchen Fonteinstellungen gezeichnet – exakt in die Ausmaße eines bestimmbaren rechteckigen Bereichs eingepasst werden kann. Die Verfahrensweise dabei ist relativ einfach:

Der Text wird zunächst mit `MeasureString` vermessen; dabei werden seine Höhe und seine Breite in Pixel³ ermittelt. Die Skalierungsfaktoren für den Text ergeben sich jetzt aus einer simplen Division der Ausmaße des Zielrechtecks durch die Ausmaße des den Text tatsächlich umschließenden Rechtecks.

Damit die Skalierung für das Graphics-Objekt später wieder auf seine Ursprungseinstellung zurückgestellt werden kann, kann die aktuelle Skalierungseinstellung mit der `Save`-Methode in einem so genannten `GraphicsState`-Objekt gespeichert werden. Erst jetzt werden die errechneten Skalierungsfaktoren mithilfe der Methode `TranslateTransform` für das aktuelle `Graphics`-Objekt bestimmt. Da der zuvor gemessene Text derselbe ist, der nun mit `DrawString` in das `Graphics`-Objekt ausgegeben wird, und die Skalierung durch Anwenden von `TranslateTransform` umgestellt wurde, passt er exakt in das Zielrechteck.

Damit alle weiteren Zeichenfunktionen unskaliert stattfinden können, wird die ursprüngliche Skalierung durch die `Restore`-Methode zu guter Letzt wieder in den Ausgangszustand zurückversetzt.

³ Vorausgesetzt, Sie haben die verwendete Maßeinheit für das aktuelle `Graphics`-Objekt nicht zuvor mit seiner `PageUnit`-Eigenschaft auf einen anderen Wert gesetzt.

Die einzige Zeichenroutine, die jetzt noch fehlt, ist die zum Zeichnen des Gitters. Im folgende Codelisting werden Sie feststellen: Die `paintGrid`-Methode ist so ausgelegt, dass sie wahlweise ein Raster oder ein Gitter zeichnen kann. Ein Rasterpunkt besteht dabei allerdings nicht aus einem einzelnen Pixel, sondern aus einem kleinen Rechteck. Das GDI+ stellt keine Funktion zur Verfügung, mit der ein einzelner Punkt gezeichnet werden kann – das ist der Hintergrund. Alternativ könnten Sie auch eine Linie mit gleichem Anfangs- und Endpunkt zeichnen, das Ergebnis wäre ähnlich:

```
'Zeichnet das Raster oder das Gitter.
Private Sub paintGrid(ByVal g As Graphics)
    Dim locPen As New Pen(GridColor, GridLineWidth)
    Dim locBrush As New SolidBrush(GridColor)

    If GridStyle = GridStyle.Dots Then
        For x As Single = 0 To ClientSize.Width Step GridSize.Width
            For y As Single = 0 To ClientSize.Height Step GridSize.Height
                g.FillRectangle(locBrush, x, y, GridLineWidth, GridLineWidth)
            Next
        Next
    Else
        For x As Single = 0 To ClientSize.Width Step GridSize.Width
            g.DrawLine(locPen, x, 0, x, ClientSize.Height)
        Next
        For y As Single = 0 To ClientSize.Height Step GridSize.Height
            g.DrawLine(locPen, 0, y, ClientSize.Width, y)
        Next
    End If
End Sub
```

Übrigens: Wenn Sie das Formular vergrößern oder verkleinern, löst das nicht notwendigerweise ein Paint-Ereignis aus. Das Paint-Ereignis wird nur beim Vergrößern des Formulars ausgelöst, und Sie können mit dem `Graphics`-Objekt, das jetzt übergeben wird, nur den Bereich erneuern, der durch das Vergrößern auch wirklich neu gezeichnet werden müsste. Das ergibt Sinn bei Inhalten, die statisch sind – also nicht durch die Größe des Formulars beeinflusst werden. Bei einer Tabellenkalkulation beispielsweise ist der dargestellte Inhalt eines Fensters nicht von seiner Größe abhängig; nur der dargestellte Ausschnitt verändert sich mit dem Vergrößern des Fensters.

In unserem Beispiel ist das anders. Wenn das Formular vergrößert oder verkleinert wird, muss der *komplette* Formularinhalt neu gezeichnet werden. Aus diesem Grund schaut die Prozedur, die dieses Ereignis verarbeitet, folgendermaßen aus:

```
'Wird aufgerufen, wenn das Formular in seiner Größe verändert wird.
Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
    Me.Invalidate()
End Sub
```

`Invalidate` bewirkt, dass der gesamte Bereich auf den es angewendet wird, hier `Me` – also das Formular selbst –, als »ungültig« gekennzeichnet wird. Dabei wird dann das Paint-Ereignis ausgelöst. Der Formularinhalt wird also komplett neu gezeichnet.

Und jetzt, nachdem Sie wissen, wie das Programm funktioniert: Stürzen Sie sich hinein in den Quellcode und experimentieren Sie mit den Einstellungen, die im Konstruktor des Formulars vorgenommen werden. Sie werden sehen, dass Sie durch Ausprobieren der verschiedenen Einstellungen der Objekte, die Graphics anbietet, am besten den Umgang mit dem GDI+ lernen können!

Flimmerfreie, fehlerfreie und schnelle Darstellungen von GDI+-Zeichnungen

Das Beispielprogramm aus dem vorherigen Abschnitt hat die Leistungsfähigkeit des GDI+ eindrucksvoll demonstriert. Allerdings hat es ebenso gezeigt, dass noch einige Handgriffe notwendig sind, um die Darstellung wirklich zu perfektionieren, denn:

- Die Bilddarstellung flimmert heftig, wenn der Anwender das Testbild vergrößert oder verkleinert.
- Das Formular sollte sich nur proportional vergrößern lassen (Festes Seitenverhältnis in X- und Y-Richtung), damit man feststellen kann, ob der Monitor einen Kreis auch wirklich rund darstellt. Das Seitenverhältnis sollte sich durch eine Eigenschaft einstellen lassen.
- Der Bildaufbau ist gerade beim Vergrößern und Verkleinern des Formulars vergleichsweise langsam.
- Und falls Sie ganz genau hingeschaut haben, werden Sie bemerkt haben, dass die Liniенstärke bei den äußereren Linien und bei den Berührungs punkten des Kreises mit den Formularrändern berücksichtigt werden sollte.

Diese Liste von Unzulänglichkeiten ist typisch für grafische Inhalte, die in Fenstern unter Windows dargestellt werden. Selbst ein Programm wie der Windows-Explorer, von dem man meinen könnte, es sei nach Jahren wirklich ausgereift, flimmert beim Vergrößern oder Verkleinern munter vor sich hin.⁴ Die nächsten Abschnitte sollen Ihnen helfen, die richtige Vorgehensweise zu finden, um Ihren Formularen und (später auch) Steuerelementen ein professionelles Aussehen zu verleihen.

Zeichnen ohne Flimmen

Bei dem ersten Problem hilft das Framework mit einem Prinzip, nach dem man schon seit Jahren und nicht erst seit Windows verfährt, um Flimmen bei der Darstellung von bewegten Bildern zu vermeiden. Die Technik ist so simpel wie genial: Anstatt ein Bild zu löschen und es verändert neu zu zeichnen, komplettiert man es zunächst in einer unsichtbaren Bitmap im Speicher. Wenn das Bild komplett gezeichnet wurde, kopiert man es als Ganzes in den sichtbaren Anzeigebereich. So ist das eigentliche Entstehen des Bildes unsichtbar und damit flimmerfrei. Darüber hinaus muss das Bild für das Neuzeichnen nicht gelöscht werden (was ein Großteil des Flimmers verursacht), denn wenn eine entsprechende Instanz das komplett fertige Bild in das zu überschreibende, sichtbare Bild kopiert, wird sowieso jeder einzelne Pixel des alten Bildes gelöscht. Ein Löschen des Hintergrunds wäre also total überflüssig.

⁴ Achten Sie mal beim Vergrößern oder Verkleinern auf das TreeView-Steuerelement des Explorers, das die Laufwerke darstellt.

Wie schon angedeutet, müssen Sie diese Funktionalität nicht selber implementieren, da sie das Framework schon fix und fertig bereithält. Sie müssen dem Framework lediglich mitteilen, dass Sie die Sonderbehandlung des »Hintergrund-Löschen-Ereignisses«, das von Windows ausgelöst wird, nicht wünschen, sondern die gerade beschriebene Methode – sie nennt sich auf neudeutsch *Double Buffering* (Doppelpufferung) – anwenden möchten.

BEGLEITDATEIEN

Sie finden das modifizierte Testbild-Projekt im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 39\\VBTestbild02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Sie erreichen das mit zwei simplen Zeilen, die Sie im Konstruktor hinterlegen:

```
Public Sub New()
    MyBase.New()

    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()

    Me.SetStyle(ControlStyles.DoubleBuffer, True)
    Me.SetStyle(ControlStyles.AllPaintingInWmPaint, True)
```

Wenn Sie das Programm nach dieser Modifizierung starten, ist es vorbei mit dem Flimmern. Sie können das Formular nach Belieben vergrößern oder verkleinern, ohne beim Neuaufbau zuschauen zu müssen.

HINWEIS Neu seit Visual Basic 2005 bzw. seit dem Framework 2.0 ist eine Eigenschaft der Basisklasse Control namens DoubleBuffered. Das Setzen dieser Eigenschaft bewirkt letzten Endes dasselbe wie Me.SetStyle(ControlStyles.DoubleBuffer, True) zusammen mit Me.SetStyle(ControlStyles.AllPaintingInWmPaint, True). Der neu eingeführte Stil ControlStyles.OptimizedDoubleBuffer verspricht mehr, als er hält: Das Durchführen des internen Double Buffering funktioniert auf die gleiche Weise wie beim Einstellen des Double Buffering auf die ursprünglichen Art über die beiden SetStyle-Methoden. Es ist also anzunehmen, dass das OptimizedBuffering-Flag nur aus Abwärtskompatibilitätsgründen eingeführt wurde, denn nur auf dieses Flag wird Framework-intern durch die DoubleBuffered-Eigenschaft zurückgegriffen.

Eigenschaften von Formularen und Control-Ableitungen mit SetStyle definieren

SetStyle dient übrigens nicht nur dazu, das *Double Buffering* ein- und auszuschalten. Auch andere Verhaltensweisen einer von Control abgeleiteten Klasse (dazu gehört auch ein Formular) lassen sich mit dieser Methode steuern.

Die grundsätzliche Verwendung von SetStyle funktioniert folgendermaßen:

Me.SetStyle(ControlStyles.Style1 or ControlStyle2 or ..., True|False)

Als ersten Parameter übergeben Sie `SetStyle` eine Kombination aus Elementen der `ControlStyles`-Enum. Der zweite Parameter bestimmt, ob die entsprechenden Flags ein- bzw. ausgeschaltet werden sollen. Welche Flags dabei welche Aufgaben übernehmen, zeigt die folgende Tabelle. Bitte verwenden Sie in diesem Falle nicht die Visual Studio-Online-Hilfe, da sie die Aufgaben der einzelnen Flags recht schwammig, teilweise sogar missverständlich erklärt.

Member-Name	Wert	Beschreibung
ContainerControl	1	Die auf Control basierende Komponente ist ein <i>Container</i> -Control. Dieses Flag ist automatisch gesetzt, wenn Sie eine Klasse aus <i>ScrollableControl</i> ableiten.
UserPaint	2	Definiert, dass die Ereignisse <i>OnPaint</i> und <i>OnBackgroundPaint</i> von <i>WndProc</i> ausgelöst werden. Dieses Flag ist standardmäßig gesetzt. Wenn Sie dieses Flag löschen, müssen Sie <i>WndProc</i> überschreiben und die entsprechenden Nachrichten auswerten, um die notwendigen Maßnahmen zum Zeichnen des Fensterinhaltes zu ergreifen.
Opaque	4	Wenn Sie dieses Flag setzen, wird <i>OnPaintBackground</i> nicht ausgelöst und der Hintergrund damit nicht gezeichnet.
ResizeRedraw	16	Bestimmt, dass <i>OnResize</i> , das bei Größenveränderung des Controls/Formulars automatisch aufgerufen wird, ein <i>Invalidate</i> auslöst und damit automatisch das Neuzeichnen des Client-Bereichs erzwingt.
FixedWidth	32	Formulare lassen sich durch die <i>AutoScale</i> -Eigenschaft so einrichten, dass sie sich automatisch an eine neue Schriftart anpassen, die ihnen durch die <i>Font</i> -Eigenschaft zugewiesen wird. Dadurch wird das Formular selbst und auch alle in ihm enthaltenen Steuerelemente entsprechend der Größe des verwendeten Fonts vergrößert oder verkleinert. ⁵ Möchten Sie verhindern, dass die Skalierung der Formularbreite stattfindet, setzen Sie dieses Flag. Mit diesem Flag können Sie <i>nicht</i> festlegen, dass ein Formular oder Control in seiner Breite nicht verändert werden darf.
FixedHeight	64	Es gilt das für <i>FixedWidth</i> Gesagte, nur für die Höhe des Formulars.
StandardClick	256	Dieses Flag ist standardmäßig gesetzt. Wenn Sie möchten, dass das Control oder das Formular kein <i>Click</i> -Ereignis auslösen soll, löschen Sie dieses Flag. WICHTIG: Wenn Sie dieses Flag löschen, löst das Control/Formular auch kein <i>DoubleClick</i> -Ereignis mehr aus.
Selectable	512	Dieses Flag ist für Formulare und Controls standardmäßig gesetzt. Es bestimmt, ob entsprechende Ereignisse von Windows überhaupt zur Fokussierung des Controls/Formulars führen können. Beachten Sie, dass <i>Container Controls</i> (<i>ScrollableControl</i> , <i>ContainerControl</i>) den Fokus nicht erhalten können, wenn sie andere Controls beinhalten. Das gilt unabhängig von der Einstellung dieses Flags. Beachten Sie auch, dass von <i>Control</i> abgeleitete Klasseninstanzen standardmäßig die <i>Fokusbenachrichtigung durch Mausklickaktivierung nicht erhalten</i> , wenn das folgende <i>UserMouse</i> -Flag nicht gesetzt ist.
UserMouse	1024	Dieses Flag muss gesetzt werden, damit eine von Control abgeleitete Instanz Fokussierungsnachrichten erhalten kann, wenn diese durch Mausklick ausgelöst wurden. Bitte beachten Sie, dass entgegen den Angaben in der Online-Hilfe das Setzen dieses Flags nicht dazu führt, dass Sie Mausereignisse von Grund auf neu implementieren müssen! <i>OnMouseXXX</i> , <i>OnClick</i> und <i>OnDoubleClick</i> werden nach wie vor ausgeführt!

⁵ Beachten Sie, dass dieses Verhalten nur beim Erstellen des Formulars funktioniert, also wenn der Font noch im Konstruktor verändert wird.

Member-Name	Wert	Beschreibung
SupportsTransparentBackColor	2048	Direkte Ableitungen von <i>Control</i> unterstützen normalerweise keine transparenten Hintergrundfarben. Mithilfe dieses Flags können Sie die Unterstützung explizit einschalten. Anschließend akzeptiert die <i>Control</i> -Ableitung zum Simulieren von Transparenz eine Hintergrundfarbe mit einer Alpha ⁶ -Komponente, die kleiner als 255 ist. Für Formulare ist dieses Flag standardmäßig gesetzt; Sie können die Farbe, die die Transparenz bestimmt, mit der <i>TransparencyKey</i> -Eigenschaft definieren.
StandardDoubleClick	4096	Dieses Flag ist standardmäßig gesetzt. Wenn Sie möchten, dass das Control oder das Formular kein <i>DoubleClick</i> -Ereignis auslösen sollen, löschen Sie dieses Flag. Auch wenn Sie das <i>StandardClick</i> -Flag löschen, erhält das Control/Formular kein <i>DoubleClick</i> -Ereignis mehr.
AllPaintingInWmPaint	8192	Normalerweise löst Windows für Controls und Formulare eine <i>WM_ERASEBKND</i> -Nachricht bei der Notwendigkeit des Neuzeichnens eines Client-Bereichs aus. Die Behandlung dieser Nachricht führt dazu, dass der Client-Bereich gelöscht – also mit einer bestimmten Farbe komplett ausgefüllt wird. In vielen Fällen führt das zu unerwünschten Flimmereffekten beim Neuzeichnen des Client-Bereichs. Setzen Sie dieses Flag, ignoriert das Steuerelement die <i>WM_ERASEBKND</i> -Fenstermeldung, um das Flimmen zu verringern.
CacheText	16384	Setzen Sie dieses Flag, bewahrt das Steuerelement eine Kopie des Textes auf, sodass dieser nicht jedes Mal, wenn er benötigt wird, aus dem Windows-Fenster abgerufen werden muss. Dieses Flag ist standardmäßig nicht gesetzt. Dieses Verhalten verbessert die Leistung, erschwert jedoch die Textsynkronisierung.
EnableNotifyMessage	32768	Setzen Sie dieses Flag, wird <i>OnNotifyMessage</i> für jede Meldung aufgerufen, die aus der Nachrichtenwarteschlange an <i>WndProc</i> des Controls bzw. Formulars gesendet wird. Dieses Flag ist standardmäßig nicht gesetzt.
DoubleBuffer	65536	Schaltet das <i>DoubleBuffering</i> ein. Parallel dazu sollten Sie <i>AllPaintingInWmPaint</i> ebenfalls setzen, damit das <i>Double Buffering</i> in Kraft treten kann.
OptimizedDoubleBuffer	131072	Wenn Sie diese Eigenschaft auf True festlegen, müssen Sie auch <i>AllPaintingInWmPaint</i> auf True festlegen, um das Double Buffering beim Zeichnen zu aktivieren. Hinweis: Dieses Flag wird von der im Framework 2.0 neu vorhandenen <i>DoubleBuffered</i> -Eigenschaft verwendet und ist vermutlich nur aus Gründen der Abwärtskompatibilität vorhanden, da beim Zeichnen eines Steuerelements in seiner internen <i>WmPaint</i> -Routine bei gesetztem <i>OptimizedDoubleBuffer</i> -Flag nichts anderes passiert als beim Setzen des »einfachen« <i>DoubleBuffer</i> -Flags.
UseTextForAccessibility	262144	Gibt an, dass der Wert der Text-Eigenschaft bei Steuerelementen, sofern festgelegt, den Namen und die Tastenkombination für aktive Eingabehilfen des Steuerelements bestimmt.

Tabelle 39.1 Die Einstellungen der *ControlStyles*-Enum

Mit dem Wissen um die Funktionsweisen dieser Flags können wir mit einem kleinen Handgriff das Programm weiter optimieren. Durch das Setzen des Flags *ResizeRedraw* kümmert sich schon die Basisklasse um den Aufruf von *Invalidate* beim Vergrößern oder Verkleinern des Formulars. Durch Einfügen einer weiteren Zeile

```
Me.SetStyle(ControlStyles.ResizeRedraw, True)
```

⁶ Die Alpha-Komponente bei einer Farbangabe bestimmt die »Durchscheinstärke«. 255 entspricht »voll deckend«, 0 entspricht »voll durchscheinend«.

im Konstruktor des Programms wird damit die Behandlung des kompletten Resize-Ereignisses überflüssig. Es ist in dieser Version des Programms auch nicht mehr vorhanden.

Was das Thema Geschwindigkeit anbelangt: Wenn Sie das Programm in dieser Version starten, bemerken Sie, dass es gar nicht so langsam läuft, wie es ursprünglich den Anschein hatte. Durch das Eliminieren des Flimmerns erkennen Sie die wahre Geschwindigkeit, mit der man auch auf langsameren Maschinen durchaus leben kann.

Programmtechnisches Bestimmen der Formulargröße

Es gibt zahlreiche Anwendungen, bei denen es notwendig ist, ein Formular oder ein Steuerelement zur Laufzeit auf eine bestimmte Breite oder eine bestimmte Höhe zu beschränken. In unserem Beispiel ergibt es beispielsweise Sinn, eine Eigenschaft einzuführen, die das Seitenverhältnis reglementiert. Nur wenn das Seitenverhältnis des Formulars auch dem Seitenverhältnis des verwendeten Monitors entspricht, können Sie beurteilen, ob der Monitor einen Kreis auch wirklich als Kreis darstellt.

Mit der SetBoundsCore-Methode eines Formulars oder eines Control-Derivats bestimmen Sie dessen Ausmaße. Allerdings können Sie die Größeneinstellungen nicht im Resize-Ereignis vornehmen, da das Neupositionieren des Fensters schon vor dem Auslösen des Ereignisses geschieht. Das Ergebnis wäre ein heftiges Flimmern. Was wir bräuchten, wären weitere Ereignisse, mit denen wir erkennen könnten, wann ein bestimmter Vorgang wie das Verschieben oder das Vergrößern bzw. Verkleinern eines Fensters beginnt und wann er abgeschlossen ist.

Seit der Version 2.0 des Frameworks gibt es zwei Ereignisse, die Sie dabei unterstützen. Diese Ereignisse nennen sich ResizeBegin und ResizeEnd. Wenn das Vergrößern eines Formulars abgeschlossen wurde, wird das ResizeEnd-Ereignis ausgelöst, und wir können das Formular hier auf das richtige Seitenverhältnis einstellen.

Leider verhält sich diese Sache nicht ganz so einfach. Denn das ResizeEnd-Ereignis wird auch dann ausgelöst, wenn Sie das Formular nur verschoben haben. Warum das so ist, ist mir ehrlich gesagt schleierhaft – im *Product Feedback Center* von Visual Studio gibt es aber bereits einen Vorschlag, das Verhalten dieses Ereignisses zu überarbeiten.

Um dem Programm abzugewöhnen, das Formular auch beim Verschieben anzupassen, behelfen wir uns deswegen mit einem kleinen Trick, der deutlich wird, wenn Sie sich die Implementierung der entsprechenden Codezeilen anschauen.

BEGLEITDATEIEN

Sie finden das modifizierte Testbild-Projekt im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 39\\VBTestbild03

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
'Wird zu Beginn einer Größenänderung aufgerufen
Protected Overrides Sub OnResizeBegin(ByVal e As System.EventArgs)
    MyBase.OnResizeBegin(e)
    'Das Flag zurücksetzen, das festhält, dass das Formular verschoben wurde
    myJustMoved = False
End Sub
```

```
'Wird aufgerufen, wenn eine Größenänderung abgeschlossen wurde
Protected Overrides Sub OnResizeEnd(ByVal e As System.EventArgs)
    'Nach Abschluss des Resize-Vorgangs Seitenverhältnis berücksichtigen,
    'aber nur, wenn dieses Ereignis nicht durch das Verschieben des Formulars
    'ausgelöst wurde.
    If Not myJustMoved Then
        'Neue Größe des Formulars im richtigen Seitenverhältnis:
        MyBase.SetBoundsCore(Location.X, Location.Y, _
            CInt(Size.Height * XYRatio), Size.Height, BoundsSpecified.Width)
    End If
    MyBase.OnResizeEnd(e)
End Sub

'Wird beim Verschieben des Formulars aufgerufen
Protected Overrides Sub OnMove(ByVal e As System.EventArgs)
    MyBase.OnMove(e)
    'Das Formular wurde verschoben; ResizeEnd wird am Ende auch
    'ausgelöst, darf aber nicht berücksichtigt werden!
    myJustMoved = True
End Sub
```

Mit dieser Änderung können wir die Möglichkeit zur Einstellung des Seitenverhältnisses nun als Eigenschaft in die neue Programmversion einbauen und im nunmehr vorhandenen ResizeEnd-Ereignis dafür sorgen, dass das Formular nach Abschluss des Größenveränderungsvorgangs richtig positioniert wird:

```
Public Property XYRatio() As Single
    Get
        Return myXYRatio
    End Get
    Set(ByVal Value As Single)
        myXYRatio = Value
    End Set
End Property
```

Diese Eigenschaft wird im Konstruktor auf ein Seitenverhältnis von 4:3 gesetzt – wie es den meisten Monitoren entspricht:⁷

```
Public Sub New()
    .
    .
    .
    'Seitenverhältnis definieren
    XYRatio = 4 / 3

    'für 19"-TFTs mit 1280x1024:
    'XYRatio = 5 / 4
    .
    .
    .
```

⁷ Ausnahmen bilden 17“, 19“ und 20“-TFT-Displays mit einer Auflösung von 1280 x 1024 Pixel, die ein Seitenverhältnis von 5:4 aufweisen.

Was Sie beim Zeichnen von breiten Linienzügen beachten sollten

Wenn Sie das Testbild in Abbildung 39.2 genauer betrachten, werden Sie feststellen, dass die äußereren Linien nicht richtig zu sehen sind. Das gilt für die Linien des Rasters genau so wie für die Berührungs punkte des Kreises. Die Gründe dafür soll das folgende kleine Projekt demonstrieren.

BEGLEITDATEIEN

Sie finden dieses Projekt für dieses Beispiel im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 39\\PenWidthDemo01

Öffnen Sie dort die entsprechende Projektmappe (SLN-Datei).

Wenn Sie das Programm starten, sehen Sie im Formular eine Grafik, die in etwa der Abbildung 39.3 entspricht.

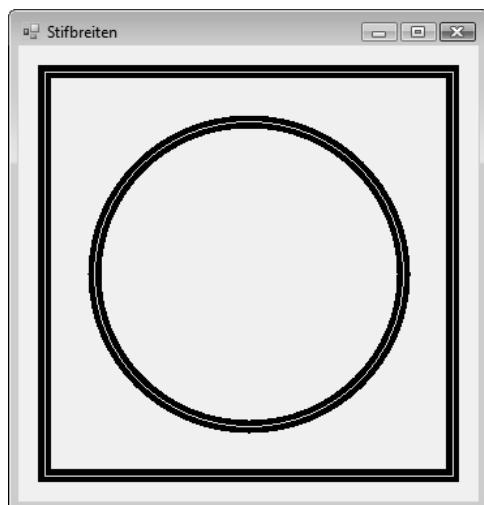


Abbildung 39.3 Wenn Sie Stiftstärken größer als ein Pixel verwenden, dann müssen Sie davon ausgehen, dass die weiteren Pixel einer Linie gleichmäßig um den eigentlich Pixel herum verteilt sind. In dieser Grafik haben die gelben und schwarzen Figuren die gleichen Ausmaße, aber andere Stiftstärken.

Die Abbildung zeigt es deutlich: Die Pixel des breiteren Stiftes der schwarzen Figuren verteilen sich um die ein Pixel breiten Linien des gelben Stiftes, und zwar auf allen Seiten zu genau gleichen Teilen. Wenn Sie also beispielsweise einen Rahmen voll sichtbar in den Client-Bereich eines Formulars zeichnen wollen, dann müssen Sie im Falle des Rechtecks die Hälfte der Liniendicke in die Koordinaten mit einrechnen. Zur Verdeutlichung: Der Code, der diese Figuren ins Formular malt, sieht folgendermaßen aus:

```
'Zeichnet jeweils ein Rechteck in einer dicken, schwarzen und einer dünnen, gelben Umrandung.  
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)  
    Dim locSchwarzerStift As New Pen(Color.Black, 10)  
    Dim locGelberStift As New Pen(Color.Yellow, 1)  
    Dim locFürRechteck As New Rectangle(20, 20, _  
                                         ClientSize.Width - 40, ClientSize.Height - 40)  
    Dim locOffsetDrittel As New Size(ClientSize.Width \ 6, ClientSize.Height \ 6)  
    Dim locFürKreis As New Rectangle(locOffsetDrittel.Width, _
```

```

    locOffsetDrittel.Height,
    ClientSize.Width - 2 * locOffsetDrittel.Width,
    ClientSize.Height - 2 * locOffsetDrittel.Height)
e.Graphics.DrawRectangle(locSchwarzerStift,
    locFürRechteck)
e.Graphics.DrawRectangle(locGelberStift,
    locFürRechteck)
e.Graphics.DrawEllipse(locSchwarzerStift,
    locFürKreis)
e.Graphics.DrawEllipse(locGelberStift,
    locFürKreis)
End Sub

```

Noch problematischer wird es, wenn Sie Linienzüge aufbauen wollen – beispielsweise, wenn Sie ein zu einer Seite offenes Dreieck aus drei verschiedenen Linien zusammensetzen müssen; beim Zeichnen von breiten Linienzügen gibt es nämlich unschöne Überschneidungen, wenn diese aus einzelnen Linien aufgebaut sind, wie das folgende Beispiel zeigt:

BEGLEITDATEIEN Sie finden dieses Projekt im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel 39\\PenWidthDemo02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

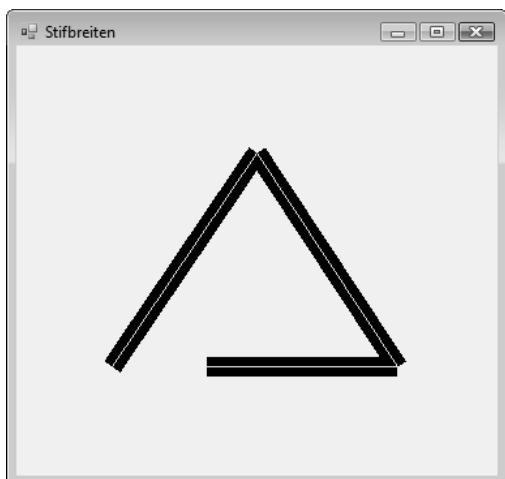


Abbildung 39.4 Bei unabhängigen Linien, die Linienzüge bilden sollen, ist das Verhalten bei großen Stiftstärken umso störender

In diesem Beispiel sollen drei Linien zu einer Figur – in diesem Falle zu einem auf einer Seite offenen Dreieck – verbunden werden; leider klappt das nur ansatzweise: Während es bei den inneren, ein Pixel breiten Linien keine Probleme gibt, sind die äußeren Linien als nicht wirklich miteinander verbunden erkennbar. Der Grund: Die jeweils nächste Linie weiß nichts davon, dass eine weitere Linie folgt, und dass die offenen Zwischenräume mit der Stiftfarbe (oder dem Muster, falls der Stift aus einem Pinsel entstanden ist) aufgefüllt werden sollten.

Geschlossene Figuren mit Polygon und GraphicsPath

Um dieses Manko zu beheben, bietet das GDI+ das so genannte `GraphicsPath`-Objekt an. Das `GraphicsPath`-Objekt erlaubt das Erstellen von Linienzügen, die wirklich miteinander verbunden sind, und seine Verwendung ist denkbar einfach. Zu jeder herkömmlichen Malmethode des GDI+ gibt es ein Äquivalent, mit dem Sie einem `GraphicsPath` einen Linienzug hinzufügen können.

Angenommen, Sie haben mit der Anweisung

```
Dim locLinienverbund As New GraphicsPath
```

ein neues `GraphicsPath`-Objekt erstellt. In diesem Fall verwenden Sie zum Zeichnen einer Linie nicht die `DrawLine`-Methode, die Sie direkt auf das `Graphics`-Objekt anwenden, sondern die `AddLine`-Methode, die Sie auf das `GraphicsPath`-Objekt beziehen. Für andere Malmethoden gibt es ähnliche `AddXXX`-Äquivalente. Auf diese Weise erstellen Sie den Linienzug zunächst, ohne ihn konkret zu zeichnen.

Haben Sie den Linienzug komplett aufgebaut, entscheiden Sie sich, ob Sie die Anweisung

```
locLinienverbund.CloseFigure()
```

anwenden, die den letzten Punkt des Linienzugs automatisch mit dem Anfangspunkt des Linienzugs verbindet (für unser Beispiel nicht erwünscht).

Das eigentliche Zeichnen des Linienzugs passiert erst jetzt mit der Anweisung

```
g.DrawPath(locPen, locLinienverbund)
```

So können Sie die Linienzüge schließen, die wirklich geschlossen werden sollen. Ein Doppelklick in das Formular demonstriert diese Vorgehensweise. Das Ergebnis sehen Sie in Abbildung 39.5:



Abbildung 39.5 Mit dem `GraphicsPath`-Objekt erstellen Sie Linienzüge, bei denen die einzelnen Komponenten wirklich miteinander verbunden sind

Der Code, der beide Figuren in das Formular zaubert, sieht dabei auszugsweise folgendermaßen aus:

```
'Zeichnet jeweils ein Rechteck in einer dicken, schwarzen und einer
'dünnen, gelben Umrandung.
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    'Nur um Tipparbeit zu sparen,
    Dim g As Graphics = e.Graphics
    'Hier werden die Eckpunkte des offenen Dreiecks gespeichert,
    Dim locDreiecksPunkte(3) As Point
    'Diese Variable dient nur dem Sparen der Tipparbeit,
    Dim locCB As Size = Me.ClientSize
    'Hier werden die Eckpunkte des Dreiecks definiert,
    Dim c As Integer
    'Zählvariable für die Schleife zum Zeichnen der Linien
    locDreiecksPunkte(0) = New Point(locCB.Width \ 5, (locCB.Height \ 4) * 3)
    locDreiecksPunkte(1) = New Point(locCB.Width \ 2, locCB.Height \ 4)
    locDreiecksPunkte(2) = New Point((locCB.Width \ 5) * 4, (locCB.Height \ 4) * 3)
    locDreiecksPunkte(3) = New Point((locCB.Width \ 5) * 2, (locCB.Height \ 4) * 3)

    'Zwei mögliche Malverfahren. Das ändert sich bei jedem Doppelklick:
    If Not myDoppelklickTrigger Then
        'Einzelne Linien malen.
        Dim locPen As New Pen(Color.Black, 15)
        'Alle Eckpunkte per Linie miteinander verbinden.
        For c = 0 To locDreiecksPunkte.Length - 2
            g.DrawLine(locPen, locDreiecksPunkte(c).X, locDreiecksPunkte(c).Y,
                      locDreiecksPunkte(c + 1).X, locDreiecksPunkte(c + 1).Y)
        Next

        'Das Gleiche nochmal in gelb und dünn.
        locPen = New Pen(Color.Yellow, 1)
        For c = 0 To locDreiecksPunkte.Length - 2
            g.DrawLine(locPen, locDreiecksPunkte(c).X, locDreiecksPunkte(c).Y,
                      locDreiecksPunkte(c + 1).X, locDreiecksPunkte(c + 1).Y)
        Next

    Else
        '...oder als Linienzug mithilfe des GraphicPaths-Objektes:
        Dim locPen As New Pen(Color.Black, 15)
        Dim locLinienverbund As New GraphicsPath
        For c = 0 To locDreiecksPunkte.Length - 2
            locLinienverbund.AddLine(locDreiecksPunkte(c).X, locDreiecksPunkte(c).Y,
                                   locDreiecksPunkte(c + 1).X, locDreiecksPunkte(c + 1).Y)
        Next

        'Mit dieser Anweisung würden Sie den Endpunkt des Linienzuges
        'mit dem Startpunkt verbinden und so die Figur schließen.
        'locFigur.CloseFigure()
        e.Graphics.DrawPath(locPen, locLinienverbund)
        locPen = New Pen(Color.Yellow, 1)
        e.Graphics.DrawPath(locPen, locLinienverbund)
        e.Graphics.DrawPath(locPen, locLinienverbund)
    End If

End Sub
```

```
Protected Overrides Sub OnDoubleClick(ByVal e As System.EventArgs)
    myDoppelklickTrigger = Not myDoppelklickTrigger
    Invalidate()
End Sub
```

HINWEIS Wenn Sie ein GraphicsPath-Objekt erstellen, dann ist dieses nicht auf nur eine Figur beschränkt. Mithilfe der Methode StartFigure können Sie innerhalb desselben GraphicPaths einen neuen Linienzug beginnen, ohne dass der alte zunächst geschlossen wird. CloseFigure hingegen schließt den aktuellen Linienzug (verknüpft also den letzten mit dem ersten Punkt). Einige Methoden, wie beispielsweise AddEllipse, fügen dem GraphicsPath einen geschlossenen Linienzug direkt hinzu.

Abschließende Änderungen am Testbild

Mit diesem Wissen lassen Sie uns nun noch den letzten Feinschliff am Testbild-Programm vornehmen. Zwei Dinge sind hier noch zu tun.

- Bei der Berechnung des Kreises muss die verwendete Stiftgröße so einkalkuliert werden, dass der komplette Kreiszug zu sehen ist.
- Beim Zeichnen des Rasters gilt dasselbe. Die entsprechenden Änderungen finden Sie in den folgenden Codezeilen wieder. Die geänderten Codezeilen befinden sich zum besseren Vergleich auskommentiert im Codelisting.

BEGLEITDATEIEN Sie finden das modifizierte Testbild-Projekt im Verzeichnis:

```
...\VB 2008 Entwicklerbuch\G - WinForms\Kapitel 39\VBTestbild03
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Platzierung des Kreises:

```
'Zielkreuz und Kreis in die Mitte
locPen = New Pen(Color.White, GridLineWidth)

'Alte Version:
'g.DrawEllipse(locPen, ClientSize.Width \ 2 - ClientSize.Height \ 2, _
'               0, _
'               ClientSize.Height, -
'               ClientSize.Height)

'Liniенstärke bei der Umfangberechnung des Kreises mit einbeziehen.
locRectF = New RectangleF((ClientSize.Width / 2.0! - ClientSize.Height / 2.0!) + GridLineWidth / 
2, -
                           GridLineWidth / 2.0!, -
                           ClientSize.Height - GridLineWidth / 2, -
                           ClientSize.Height - (GridLineWidth))
g.DrawEllipse(locPen, locRectF)
```

Berechnung der Rastergröße:

```

Public ReadOnly Property GridSize() As SizeF
    Get
        'So sah es vorher aus:
        'Return New SizeF(CSng(ClientSize.Width / GridCols),
        '                  CSng(ClientSize.Height / GridRows)) ->

        'Bei der Berechnung der Zellenausmaße die Linienbreite
        'mit in die Verhältnisrechnung einbeziehen.
        Return New SizeF(CSng((ClientSize.Width - GridLineWidth) / GridCols),
                          CSng((ClientSize.Height - GridLineWidth) / GridRows))
    End Get
End Property

```

Zeichnen des Gitters:

```

Private Sub paintGrid(ByVal g As Graphics)

    Dim locPen As New Pen(GridColor, GridLineWidth)
    Dim locBrush As New SolidBrush(GridColor)

    If GridStyle = GridStyle.Dots Then
        For x As Single = 0 To ClientSize.Width Step GridSize.Width
            For y As Single = 0 To ClientSize.Height Step GridSize.Height
                g.FillRectangle(locBrush, x, y, GridLineWidth, GridLineWidth)
            Next
        Next
    Else:
        'Alte Version
        'For x As Single = 0 To ClientSize.Width Step GridSize.Width
        '    g.DrawLine(locPen, x, 0, x, ClientSize.Height)
        'Next

        'For y As Single = 0 To ClientSize.Height Step GridSize.Height
        '    g.DrawLine(locPen, 0, y, ClientSize.Width, y)
        'Next

        'Aufgepasst: Dieses Konstrukt funktioniert nicht, wegen Rundungsfehlern!
        'Dim locStart, locEnd As Single
        'locStart = GridLineWidth / 2
        'locEnd = ClientSize.Width

        'For x As Single = locStart To locEnd Step GridSize.Width
        '    Problem: locEnd würde niemals genau gleich x sein, wg. Rundungsfehler,
        '    die letzte Linie damit nie an die richtige Stelle gemalt.
        '    If x = locStart Or x = locEnd Then
        '        g.DrawLine(locPen, x, GridLineWidth / 2, x, ClientSize.Height)
        '    Else
        '        g.DrawLine(locPen, x - GridLineWidth / 2, GridLineWidth / 2, x - GridLineWidth / 2,
        '                  ClientSize.Height)
        '    End If
    End If
'Next

```

```
'locStart = GridLineWidth / 2
'locEnd = ClientSize.Height
'For y As Single = locStart To locEnd Step GridSize.Height
'    If y = locStart Or y = locEnd Then
'        g.DrawLine(locPen, GridLineWidth / 2, y, ClientSize.Width, y)
'    Else
'        g.DrawLine(locPen, GridLineWidth / 2, y - GridLineWidth / 2, _
'                   ClientSize.Width, y - GridLineWidth / 2)
'    End If
'Next
```

Zu diesem, auskommentierten Teil des Listings vielleicht noch ein paar Anmerkungen, damit Sie diese typischen Fehler beim Arbeiten mit Grafikkoordinaten, die Sie mit Single-Werten bestimmen, in Projekten von vornherein vermeiden können. Hier startete der Programmierer den Versuch, das Zeichnen der jeweils ersten und letzten Linie des Gitters durch Koordinatenvergleich zu entdecken – doch dieser Vorgang ist bei vielen Single-Werten grundsätzlich zum Scheitern verurteilt: Beim Vergleichen der aktuell verarbeiteten Koordinate mit dem bekannten Wert der jeweils letzten Koordinate durch den Gleichheitsoperator wird der Ausdruck

```
x = locEnd
```

bzw.

```
y = locEnd
```

niemals wahr. Zwar entspricht locEnd dem Wert x bzw. y *fast*, durch die Umwandlung vom Dezimal- in das Binärsystem und die sich daraus kumulierenden Rundungsfehler aber eben nur *fast*. Das bedeutet:

HINWEIS Vermeiden Sie es grundsätzlich, Programmzustände auf Grund von Fließkommavergleichen festzustellen. Sie können mit Double- bzw. Single-Werten gefahrlos rechnen und die errechneten Ergebnisse zur Bestimmung von Koordinaten verwenden. Durch kumulierte Rundungsfehler schlagen Vergleiche aber in den meisten Fällen fehl, und Ihr Programm arbeitet nicht wie erwartet. Im hier gezeigten Beispiel könnte locEnd beispielsweise den Wert 477,5 innehaben; x ist durch kumulierte Rundungsfehler im entscheidenden Moment aber nicht 477,5, sondern 477,500001 – und das ist zwar *fast* 477,5 aber nicht ausreichend, um im Vergleichsausdruck True zurückzuliefern und die Sonderbehandlung für die als zuletzt gemalt erkannte Linie einzuleiten.

```
Dim locStart, locEnd As Single
locStart = GridLineWidth / 2
locEnd = ClientSize.Width - GridSize.Width

'Erste Linie Sonderfall:
g.DrawLine(locPen, GridLineWidth / 2, 0, GridLineWidth / 2, ClientSize.Height)

'Mittlere Linien malen.
For x As Single = locStart To locEnd Step GridSize.Width
    g.DrawLine(locPen, x - GridLineWidth / 2, GridLineWidth / 2, _
               x - GridLineWidth / 2, ClientSize.Height)
Next
```

```
'Letzte Linie Sonderfall.  
g.DrawLine(locPen, ClientSize.Width - GridLineWidth / 2, 0,  
          ClientSize.Width - GridLineWidth / 2, ClientSize.Height)  
  
'Horizontale Linien:  
locStart = GridLineWidth / 2  
locEnd = ClientSize.Height - GridSize.Height  
  
'Erste Linie Sonderfall:  
g.DrawLine(locPen, 0, GridLineWidth / 2, _  
          ClientSize.Width, GridLineWidth / 2)  
  
For y As Single = locStart To locEnd Step GridSize.Height  
    g.DrawLine(locPen, GridLineWidth / 2, y - GridLineWidth / 2, _  
              ClientSize.Width, y - GridLineWidth / 2)  
Next  
  
'Letzte Linie Sonderfall:  
g.DrawLine(locPen, 0, ClientSize.Height - GridLineWidth / 2,  
          ClientSize.Width - GridLineWidth / 2, ClientSize.Height - GridLineWidth / 2)  
  
End If  
  
End Sub
```

Stattdessen machen Sie es besser, wie in dieser Version der *Gitter-Mal*-Prozedur gezeigt. Die Behandlung der Sonderfälle ist absolut codiert und ihre Ausführung obliegt keiner bedingten Programmverzweigung.

Kapitel 40

Entwickeln von Steuerelementen für Windows Forms

In diesem Kapitel:

Neue Steuerelemente auf Basis vorhandener Steuerelemente implementieren	1171
Entwickeln von konstituierenden Steuerelementen	1180
Erstellen von Steuerelementen von Grund auf	1192

Die konsequente Einhaltung der objektorientierten Programmierung prädestiniert Entwickler geradezu zur Entwicklung von wieder verwendbaren Komponenten wie beispielsweise Funktionsbibliotheken oder Benutzersteuerelementen.

Hier und da konnten Sie im Verlauf der letzten Kapitel schon mehrfach sehen, wie einfach in Visual Basic 2008 das Erstellen neuer Benutzersteuerelemente vonstatten geht; denken Sie dabei exemplarisch an Kapitel 38 und die »Entwicklung« eines Button-Elements, das auch ein Rechts-Klick-Ereignis zur Verfügung stellt.

Sobald Sie Windows Forms-Anwendungen entwickeln, arbeiten Sie automatisch mit visualisierten Komponenten oder einfach ausgedrückt: mit Steuerelementen. Steuerelemente sind in Sachen RAD¹ ein wichtiges Werkzeug, um Anwendungen mit grafischem Frontend tatsächlich schnell fertig stellen zu können. Aus diesem Grund bietet es sich für wiederkehrende Aufgaben an, eigene Steuerelemente zu entwickeln, und wie das funktioniert, mag Ihnen dieses Kapitel zeigen.

Die Anwendung von Steuerelementen selbst beschränkt sich dabei nicht nur auf eine simple Erweiterung von Bedienungselementen einer Programmoberfläche. Sie können ganze Geschäftslogiken in Steuerelementen zusammenfassen und diese in mehreren Ihrer Anwendungen wieder verwenden. In vielen Anwendungen müssen beispielsweise Kontaktdaten bestimmter Personen erfasst und verwaltet werden. Sie könnten also beispielsweise ein Kontakte-Steuerelement entwickeln, das dem Anwender die Verwaltung und das Neuanlegen von Adressen ermöglicht. Dieses Steuerelement entwickeln Sie nur ein einziges Mal, und Sie verwenden es wieder, indem Sie es wie jedes andere Steuerelement in ein Formular einer neuen Anwendung einfügen, in der Sie diese Funktionalität benötigen.

Um neue Steuerelemente zu entwickeln, stehen Ihnen grundsätzlich drei Wege zur Verfügung:

- Sie erweitern ein vorhandenes Steuerelement. Möchten Sie beispielsweise das TextBox-Steuerelement um zusätzliche Funktionalitäten ergänzen, leiten Sie es in eine neue Klasse ab und implementieren lediglich die Erweiterungen.
- Sie erstellen ein neues Steuerelement auf Basis mehrerer schon vorhandener Steuerelemente, fassen also die Funktionalität verschiedener vorhandener Steuerelemente zu einem neuen Steuerelement zusammen. Solche Steuerelemente werden auch als *konstituierende Steuerelemente* bezeichnet. Um Steuerelemente dieser Art zu entwerfen, greifen Sie entweder auf Visual Studio selbst zurück und nutzen dessen Designer-Unterstützung, um neue Steuerelemente genau so einfach wie Formulare zu erstellen.² Oder Sie leiten Ihre neue Steuerelementklasse von ContainerControl ab und implementieren die Funktionalität ausschließlich über Code.
- Sie implementieren ein Steuerelement komplett neu von Grund auf. In diesem Fall leiten Sie Ihre neue Steuerelementklasse von der Control-Klasse ab. Sie müssen sich dann aber auch um das Zeichnen seiner sichtbaren Komponenten kümmern – feste Trittsicherheit im Umgang mit den Funktionen des GDI+ ist dabei eine wichtige Voraussetzung.

¹ Rapid Application Development, etwa: zügige Anwendungsentwicklung.

² Steuerelemente dieser Art werden in Visual Studio Benutzersteuerelemente genannt.

Neue Steuerelemente auf Basis vorhandener Steuerelemente implementieren

Die einfachste Weise, ein neues Steuerelement zu erstellen, ist es auf Basis eines bereits vorhandenen Steuerelements zu implementieren. Durch simples Vererben einer vorhandenen Steuerelementklasse schaffen Sie bereits ein neues Steuerelement, das über die gesamte Funktionalität seines Ahnen verfügt. Ihre Aufgabe beschränkt sich anschließend darauf, durch Ergänzen von Funktionen oder Überschreiben von schon vorhandenen Prozeduren die Funktionalität des ursprünglichen Steuerelements zu verändern oder zu ergänzen.

Noch in Visual Basic 2003.NET war es ein vergleichsweise großer Aufwand, ein Steuerelement so zu erstellen, dass Sie es in der Toolbox wieder finden konnten. Sie mussten dazu Ihr Steuerelement in einer eigenen Assembly erstellen, und einige weitere Handgriffe waren nötig, um es letzten Endes in der Toolbox als entsprechendes Symbol anzuzeigen.

In Visual Studio 2005 und 2008 ist das so einfach wie das Erstellen einer neuen Klasse geworden. Steuerelemente können zwar nach wie vor in eigenen Assemblies liegen, aber das ist kein Muss mehr, um es in der Toolbox sehen zu können.



Abbildung 40.1 Die Einfärbung des jeweils fokussierten Eingabefelds hilft dem Anwender die Übersicht zu bewahren, ist für den Entwickler aber unangenehme Fließbandarbeit

Das folgende Beispiel soll das verdeutlichen. Das Szenario: Sie möchten eine Eingabemöglichkeit für Formulare schaffen, bei der das aktuelle (das »fokussierte«) Eingabefeld, der besseren Übersichtlichkeit halber, gelb unterlegt wird. Damit wird es für den Anwender gerade in großen Formularen einfacher, über eine längere Zeit ermüdfrei zu arbeiten.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel40\\ColoredTextBoxForm

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Normalerweise müssten Sie wie folgt vorgehen, um eine solche Funktionalität in einem Formular zu implementieren:

- Sobald ein TextBox-Steuerelement im Formular den Eingabefokus erhält, müssten Sie seine Hintergrundfarbe in eine auffällige aber dennoch für die Eingabe nicht störende Farbe (beispielsweise gelb) ändern. Das TextBox-Steuerelement stellt dafür das GetFocus-Ereignis zur Verfügung, das Sie durch eine geeignete Ereignisbehandlungsroutine behandeln und dort mit der Backcolor-Eigenschaft des Steuerelements die Hintergrundfarbe ändern.

- Wenn das TextBox-Steuerelement den Fokus wieder verliert, würden Sie seine Hintergrundfarbe mit dem ursprünglichen Farbwert wiederherstellen.
- Diese Implementierung müsste für jedes Eingabefeld erfolgen.

Der Code eines entsprechenden Formulars sähe dann folgendermaßen aus:

```
Public Class Form1

    Private myOriginalBackcolor As Color

    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
        Me.Close()
    End Sub

    Private Sub txtMatchcode_GotFocus(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles txtMatchcode.GotFocus
        myOriginalBackcolor = txtMatchcode.BackColor
        txtMatchcode.BackColor = Color.Yellow
    End Sub

    Private Sub txtMatchcode_LostFocus(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles txtMatchcode.LostFocus
        txtMatchcode.BackColor = myOriginalBackcolor
    End Sub

    Private Sub txtNachname_GotFocus(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles txtNachname.GotFocus
        myOriginalBackcolor = txtNachname.BackColor
        txtNachname.BackColor = Color.Yellow
    End Sub

    Private Sub txtNachname_LostFocus(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles txtNachname.LostFocus
        txtNachname.BackColor = myOriginalBackcolor
    End Sub

    Private Sub txtVorname_GotFocus(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles txtVorname.GotFocus
        myOriginalBackcolor = txtVorname.BackColor
        txtVorname.BackColor = Color.Yellow
    End Sub

    Private Sub txtVorname_LostFocus(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles txtVorname.LostFocus
        txtVorname.BackColor = myOriginalBackcolor
    End Sub
```

```
Private Sub txtOrt_GotFocus(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles txtOrt.GotFocus
    myOriginalBackColor = txtOrt.BackColor
    txtOrt.BackColor = Color.Yellow
End Sub

Private Sub txtOrt_LostFocus(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles txtOrt.LostFocus
    txtOrt.BackColor = myOriginalBackColor
End Sub

End Class
```

Und das kann es doch wohl nicht sein, oder? – Der Aufwand rechtfertigt hier eigentlich nicht das Ergebnis.³ Ungleich besser wäre es, gäbe es ein TextBox-Steuerelement – nennen wir es FocusColoredTextBox – das eine FocusColor-Eigenschaft sowie eine OnFocusColor-Eigenschaft anbietet. Die FocusColor-Eigenschaft würde dann die Farbe bestimmen, mit der der Hintergrund des Eingabebereichs beim Erhalten des Fokus eingefärbt würde, und die OnFocusColor-Eigenschaft vom Typ Boolean trüfe die Aussage, ob die Einfärbung des Hintergrunds überhaupt stattfände.

Der Weg eines Steuerelements vom Klassencode in die Toolbox

Das Entwickeln und der anschließende Umgang mit selbst definierten oder weiteren Steuerelementen ist nie so einfach gewesen wie in Visual Basic ab Version 2005. Sobald Sie Ihrem Projekt eine Klassendatei hinzufügen, in der sich die Definition einer von Control (oder einer auf Control basierenden) abgeleiteten Komponente befindet, dann sehen Sie die neue Komponente direkt nach dem ersten Kompilieren in der Toolbox.

Probieren Sie es aus:

- Erstellen Sie eine neue Windows Forms-Anwendung.
- Fügen Sie dem Projekt eine neue Klassendatei hinzu, indem Sie aus dem Kontextmenü des Projektmappen-Explorers den Menüpunkt *Hinzufügen* und *Klasse* wählen.

³ Aufmerksamen Lesern ist es sicherlich nicht entgangen, dass man diesen Code schon ein wenig kompakter formulieren könnte, würde man das Ein- und Ausfärben in nur zwei Ereignisbehandlungsroutinen platzieren, die an *alle* TextBox-Steuerelemente gebunden wären – *Handles* erlaubt ja schließlich die Verdrahtung mit mehreren Ereignisquellen. Aber auch dieses Verfahren würde nicht der OOP und dem Kapseln von Funktionalitäten in geschlossenen Komponenten entsprechen.



Abbildung 40.2 Um ein neues Steuerelement zu erstellen, fügen Sie Ihrem Projekt zunächst eine ganz normale Klassendatei hinzu, ...

- Geben Sie im Dialog, der jetzt erscheint, einen neuen Klassencode-Dateinamen für die Codedatei Ihres zukünftigen Steuerelements an – beispielsweise *FocusedColoredTextBox*.
- Doppelklicken Sie auf die neu erstellte Klassencode-Datei im Projektexplorer, um sie im Codeeditor zu öffnen. Ändern Sie den Klassencode zum Beispiel folgendermaßen ab:

```
FocusedColoredTextBox.vb*
FocusedColoredTextBox
Public Class FocusedColoredTextBox
    Inherits TextBox
End Class
```

Abbildung 40.3 ... ergänzen die Klassendefinition mit *Inherits*, um die Ableitung von einem vorhandenen Steuerelement (mindestens von Control), ...

- Doppelklicken Sie auf die Formulardatei *Form1.vb* im Projektexplorer, um sie im Designer darzustellen.
- Falls die Toolbox nicht permanent angezeigt wird, holen Sie diese in den Vordergrund, und klicken Sie auf das kleine Reißzweckensymbol an der rechten oberen Ecke des Fensters, um sie permanent darzustellen.
- Wählen Sie aus dem Menü *Erstellen* den Menüpunkt *Projektmappe neu erstellen* (bzw. *[Projektname] neu erstellen*), und beobachten Sie, was in der Toolbox passiert.

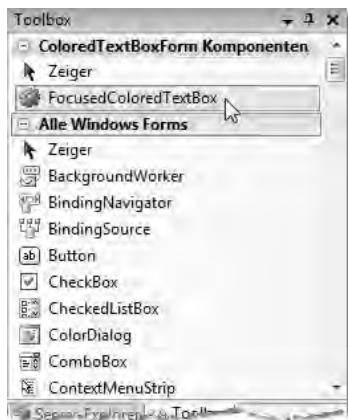


Abbildung 40.4 ... und sofort nach dem ersten Erstellen Ihres Projektes befindet sich das neue Steuerelement auch schon in der Toolbox zur sofortigen Verwendung in Ihren Formularen

Wie in Abbildung 40.4 zu sehen, befindet sich das neue Steuerelement zur sofortigen Verwendung in der Toolbox.

Sie haben also jetzt ein neues Steuerelement namens FocusColoredTextBox. Da der Klassencode außer der Klassenableitung noch keine weitere Funktionalität aufweist, kann dieses neue Steuerelement exakt das, was ein TextBox-Steuerelement auch kann – aber ich finde, selbst das ist schon beeindruckend, oder nicht?

Probieren Sie es aus: Ziehen Sie das neue Steuerelement im Formular auf, und schauen Sie sich im Eigenschaftenfenster an, welche Eigenschaften dieses Steuerelement zu bieten hat.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel40\\FocusColoredTextBoxDemo01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Implementierung von Funktionslogik und Eigenschaften

Die Implementierung der Funktionslogik ist keine aufwändige Angelegenheit, wie Sie anhand des folgenden Listings sehen können, bei dem die XML-Dokumentierungszeilen alleine fast zur Hälfte des Gesamtcores beitragen.

```
Public Class FocusedColoredTextBox
    Inherits TextBox

    Private myFocusColor As Color          ' Fokus-Einfärbfarbe (intern)
    Private myOriginalBackColor As Color   ' Zwischenspeicher der ursprünglichen Farbe
    Private myOnFocusColor As Boolean      ' Flag, dass Autofärben bei Fokussierung bestimmt (intern)

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse
    ''' </summary>
    ''' <remarks></remarks>
    Sub New()
```

```

'NIE VERGESSEN, gerade bei Steuerelementen!
'Den Basiskonstruktor aufrufen!
 MyBase.New()

'Voreingestellter Wert ist gelb.
myFocusColor = Color.Yellow

'Voreingestellt ist: Bei Fokus wird eingefärbt.
myOnFocusColor = True
End Sub

'Überschrieben: Färbt im Bedarfsfall mit FocusColor ein,
'wenn das Steuerelement den Fokus erhält.
Protected Overrides Sub OnGotFocus(ByVal e As System.EventArgs)
    MyBase.OnGotFocus(e)
    If OnFocusColor Then
        myOriginalBackcolor = Me.BackColor
        Me.BackColor = FocusColor
    End If
End Sub

'Überschrieben: Stellt die Ursprungshintergrundfarbe im
'Bedarfsfall wieder her, wenn das Steuerelement den Fokus verliert.
Protected Overrides Sub OnLostFocus(ByVal e As System.EventArgs)
    MyBase.OnLostFocus(e)
    If OnFocusColor Then
        Me.BackColor = myOriginalBackcolor
    End If
End Sub

''' <summary>
''' Bestimmt oder ermittelt die Farbe, die das Steuerelement automatisch erhält,
''' wenn es fokussiert wird, und wenn die OnFocusColor-Eigenschaft gesetzt wurde.
''' </summary>
''' <value>Die Farbe, die beim Fokussieren verwendet werden soll.</value>
''' <returns></returns>
''' <remarks></remarks>
Public Property FocusColor() As Color
    Get
        Return myFocusColor
    End Get
    Set(ByVal value As Color)
        myFocusColor = value
    End Set
End Property

''' <summary>
''' Bestimmt oder ermittelt, ob das Steuerelement bei Erhalt des Fokus
''' automatisch mit der in FocusColor eingestellten Farbe eingefärbt werden soll.
''' </summary>
''' <value>True, wenn die Autoeinfärbung aktiviert werden soll.</value>
''' <returns></returns>
''' <remarks></remarks>
Public Property OnFocusColor() As Boolean
    Get
        Return myOnFocusColor
    End Get
    Set(ByVal value As Boolean)
        myOnFocusColor = value
    End Set
End Property
End Class

```

Wenn Sie das Beispiel kompilieren und starten, können Sie die neuen Steuerelemente, auf denen das Formular des Beispiels basiert, direkt ausprobieren.

Sobald Sie das Programm beenden und eine der vorhandenen Instanzen des neuen Steuerelements selektieren, können Sie die beiden neuen Eigenschaften im Eigenschaftenfenster entdecken:

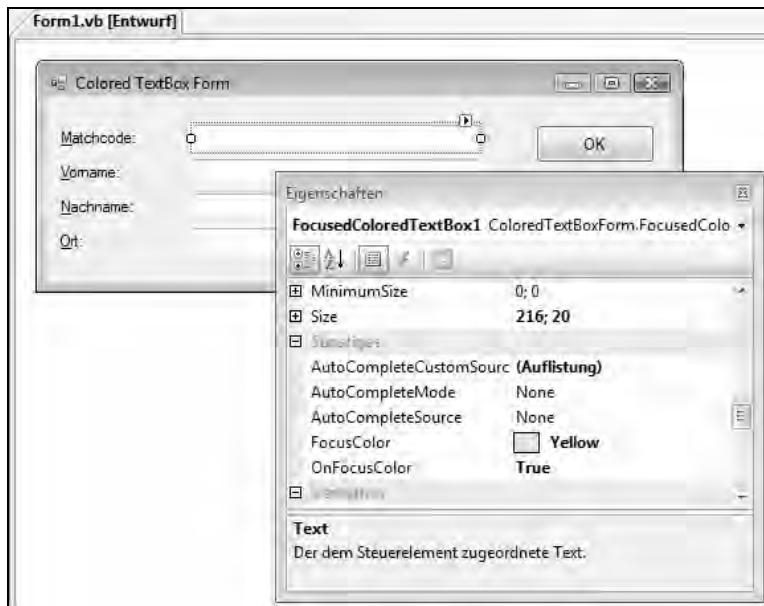


Abbildung 40.5 Die neuen Eigenschaften der FocusedColoredTextBox sind im Eigenschaftenfenster zu finden – allerdings unter der Kategorie Sonstiges, und die Werte der Eigenschaften sind auf eine Weise markiert, die implizieren, dass sie verändert wurden

Sie sehen dabei allerdings ebenfalls, dass sie in einer Kategorie platziert wurden (»Sonstiges«), in die sie eigentlich gar nicht hingehören. Darüber hinaus werden die Werte der Eigenschaften auch so dargestellt, als wären sie abweichend von den Standardeigenschaften eingestellt worden – denn nur in diesem Fall werden die Werte von Eigenschaften im Fenster fett hervorgehoben.

Der nächste Abschnitt zeigt, wie Sie diese Verhaltensweisen normalisieren.

Steuern von Eigenschaften im Eigenschaftenfenster

Zur Implementierung einer neuen Eigenschaft, die im Eigenschaftenfenster zur Entwurfszeit angezeigt werden soll, genügt es, wenn Sie eine Eigenschaft im Sinne von Klassen implementieren, also mit entsprechenden Property-Prozeduren.

Das .NET Framework kennt allerdings gerade für die Steuerung von Eigenschaften im Eigenschaftenfenster eine Reihe von Attributen, die Sie in der folgenden Tabelle beschrieben finden.

Attribut	Beschreibung
Browsable	Bestimmt, ob die Eigenschaft, der dieses Attribut zugeordnet ist, zur Entwurfszeit im Eigenschaftenfenster sichtbar ist oder nicht.
Description	Definiert den Beschreibungstext, der unterhalb des Eigenschaftenfensters zur Entwurfszeit angezeigt wird, wenn die Eigenschaft ausgewählt ist.

Attribute	Beschreibung
DefaultValue	<p>Bestimmt, welchen Typ und Wert der Standardwert der Eigenschaft aufweist. Wenn der Standardwert der Eigenschaft zur Entwurfszeit definiert ist, wird kein Serialisierungscode vom Designer erzeugt. Aus diesem Grund muss die Steuerelementklasse dafür Sorge tragen, dass während der Initialisierung des Steuerelements (am besten schon im Konstruktor) der entsprechende Wert gesetzt ist. Im Eigenschaftenfenster wird eine Eigenschaft, die ihren Standardwert besitzt, in Normalschrift angezeigt; veränderte Werte werden in Fettschrift angezeigt.</p> <p>Wichtig: Falls es – durch komplexere Datentypen oder Auswertungslogiken – nicht möglich ist, einen Standardwert mit diesem Attribut zu bestimmen, implementieren Sie eine Funktion mit dem Namen der Eigenschaft und dem Präfix »ShouldSerialize...«, die True zurückgibt, wenn die Eigenschaft im aktuellen Zustand nicht den Standardwert zurückliefert. Lautet eine Eigenschaft also beispielsweise SampleProperty, so sollte die entsprechende Funktion, die den Code-Serialisierer für diese Eigenschaft steuert, ShouldSerializeSampleProperty lauten und True zurückliefern, wenn die Wertdefinition für diese Eigenschaft <i>nicht</i> dem Standardwert entspricht.</p> <p>Um in diesem Fall die <i>Zurücksetzen</i>-Funktionalität des Eigenschaftenfensters zur Verfügung zu stellen, implementieren Sie eine weitere Funktion mit dem Präfix »Reset...« (also beispielsweise ResetSampleProperty); diese Funktion muss den Standardwert der Eigenschaft wieder herstellen.</p>
Category	Legt fest, unter welcher Kategorie im Eigenschaftenfenster die Eigenschaft eingesortiert werden soll. Wenn Sie dieses Attribut nicht definieren, wird die Eigenschaft automatisch unter einer neuen Kategorie namens <i>Sonstiges</i> eingeordnet.

Tabelle 40.1 Diese Attribute sind wichtig für die Steuerung der Anzeige im Eigenschaftenfenster

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel40\\FocusColoredTextBoxDemo02

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Der folgende Codeausschnitt zeigt beispielhaft, wie diese Attribute für Eigenschaften zur Steuerung ihres Verhaltens im Eigenschaftenfenster zum Einsatz kommen:

```

''' <summary>
''' Bestimmt oder ermittelt die Farbe, die das Steuerelement automatisch erhält,
''' wenn es fokussiert wird, und wenn die OnFocusColor-Eigenschaft gesetzt wurde.
''' </summary>
''' <value>Die Farbe, die beim Fokussieren verwendet werden soll.</value>
''' <returns></returns>
''' <remarks></remarks>
<Category("Darstellung"), _
DefaultValue(GetType(Color), "Yellow"), _
Description("Bestimmt oder ermittelt die Farbe, die das Steuerelement automatisch erhält, " & _
"wenn es fokussiert wird, und wenn die OnFocusColor-Eigenschaft gesetzt wurde."), _
Browsable(True)>_
Public Property FocusColor() As Color
    Get
        Return myFocusColor
    End Get
    Set(ByVal value As Color)
        myFocusColor = value
    End Set
End Property
''' <summary>
''' Bestimmt oder ermittelt, ob das Steuerelement bei Erhalt des Fokus

```

```

    /// automatisch mit der in FocusColor eingestellten Farbe eingefärbt werden soll.
    /// </summary>
    /// <value>True, wenn die Autoeinfärbung aktiviert werden soll.</value>
    /// <returns></returns>
    /// <remarks></remarks>
<Category("Verhalten"), _
DefaultValue(True), _
Description("Bestimmt oder ermittelt, ob das Steuerelement beim Erhalten des Fokus " &
"automatisch mit der in FocusColor eingestellten Farbe eingefärbt werden soll."), _
Browsable(True)>
Public Property OnFocusColor() As Boolean
    Get
        Return myOnFocusColor
    End Get
    Set(ByVal value As Boolean)
        myOnFocusColor = value
    End Set
End Property

```

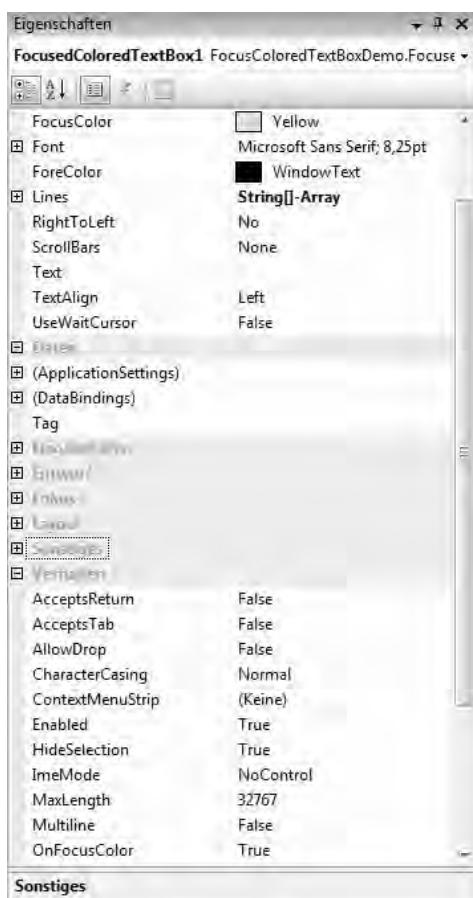


Abbildung 40.6 Die neuen Eigenschaften gliedern sich durch den Einsatz entsprechender Attribute so sauber in die vorhandenen ein, dass ein Unterschied nicht mehr erkennbar ist. Sowohl die Anzeige der Standardwerte funktioniert bei beiden neuen Eigenschaften (siehe FocusColor, ganz oben, sowie OnFocusColor, ganz unten) als auch das Einblenden der helfenden Erklärung der Eigenschaften, die im unteren Teil des Eigenschaftenfensters zu sehen ist. Und auch die Platzierung der Eigenschaften in den richtigen Kategorien funktioniert überdies.

HINWEIS Die `DefaultValueAttribut`-Klasse verfügt über nicht weniger als 11 Überladungen. Folgende Regel sollte deswegen bei ihrer Anwendung gelten: Für primitive Datentypen geben Sie den Standardwert einfach als einziges Argument in Klammern an (siehe `OnFocusColor` im vorherigen Listingauszug). Komplexere Strukturen wie beispielsweise `Color`, `Size`, `Location` und andere Typen benötigen so genannte Eigenschaften-Deskriptoren-Klassen, die in der Lage sind, eine Instanz des Wertetyps aus einer Zeichenkette zu bilden. Alle wichtigen im .NET Framework vorhandenen Wertetypen verfügen über solcherlei Eigenschaften-Deskriptoren. In diesem Fall reicht es allerdings nicht aus, einen einzelnen Standardwert der `DefaultValueAttribut`-Klasse zu übergeben; stattdessen geben Sie zwei Parameter an: Den Typ der Struktur als auch seine Wertrepräsentation als Zeichenkette, so, wie es im vorherigen Listingauszug bei der `FocusColor`-Eigenschaft gezeigt wird.

TIPP Gerade bei Steuerelementen, die Sie immer wieder verwenden wollen, ergibt es Sinn, eine eigene Assembly als Komponentenbibliothek zu erstellen. Abschnitt »Anlegen eines Projekts für die Erstellung einer eigenen Steuerelement-Assembly« ab Seite 1181 zeigt, wie das geht, wenn auch mit anderen Steuerelementtypen. Die gezeigten Schritte für das Erstellen einer Assembly können Sie aber genauso gut auf die in diesem Kapitel besprochenen Steuerelementtypen anwenden, die von anderen Steuerelementen abgeleitet sind.

Entwickeln von konstituierenden Steuerelementen

Bei konstituierenden Steuerelementen handelt es sich, wie eingangs bereits erwähnt, um solche, die aus mehreren vorhandenen ein neues mit erweiterter Funktionalität bilden. Konstituierende Steuerelemente sollten Sie dann einsetzen, wenn Sie Business-Logik in Form von UI an mehreren Stellen Ihrer Anwendung benötigen – beispielsweise eine Artikelauswahlliste oder eine Zeiterfassungs-Komponente. Wenn Sie mit Visual Studio konstituierende Steuerelemente entwickeln, haben Sie generell zwei Möglichkeiten:

- Sie gehen so vor, wie Sie es im vorangegangenen Abschnitt kennen gelernt haben, leiten Ihr Steuerelement allerdings von `ContainerControl`⁴ ab. Um das Steuerelement auf mehreren vorhandenen Steuerelementklassen basieren zu lassen, fügen Sie der `Controls`-Auflistung im Konstruktor neue Instanzen der vorhandenen Steuerelementklassen hinzu. Die Positionierung der Controls definieren Sie durch das Setzen der `Location`-, `Size`-, `Dock`- bzw. `Anchor`-Eigenschaften im Code.
- Sie nehmen den Visual Studio-Designer zu Hilfe und definieren das neue Steuerelement wie ein Formular. Der Vorteil: Sie sparen wertvolle Zeit (nämlich die des Codeschreibens zum Instanziieren und Positionieren der Basissteuerelemente) und können sich auf die eigentliche Funktionalitätsimplementierung des Steuerelements konzentrieren.

Im folgenden Beispiel verwenden wir die letzte der beiden angesprochenen Vorgehensweisen, um ein Steuerelement zu erstellen. In diesem Beispiel handelt es sich um die Kombination eines `TextBox`- und eines `ListBox`-Steuerelements. Die Liste kann wie eine herkömmliche `ListBox` mit entsprechenden Einträgen ausgestattet werden. Wenn der Anwender in der `TextBox` einen Text eingibt, versucht das Steuerelement auf Grund der vorhandenen Listeneinträge den eingegebenen Text zu kompletieren (siehe Abbildung 40.7).

⁴ Theoretisch könnten Sie ein konstituierendes Steuerelement auch von `Control` ableiten, da es ebenfalls über die erforderliche `ControlCollection` (über seine `Controls`-Eigenschaft) verfügt. Allerdings hat `ContainerControl` einige weitere Vorzüge – so kann es beispielsweise für eingebundene Steuerelemente, die nicht in den Anzeigebereich passen, automatisch eine Rollbalkenfunktionalität zur Verfügung stellen, mit der der Anwender den sichtbaren Ausschnitt darstellen lassen kann (`AutoScroll`-Eigenschaft).

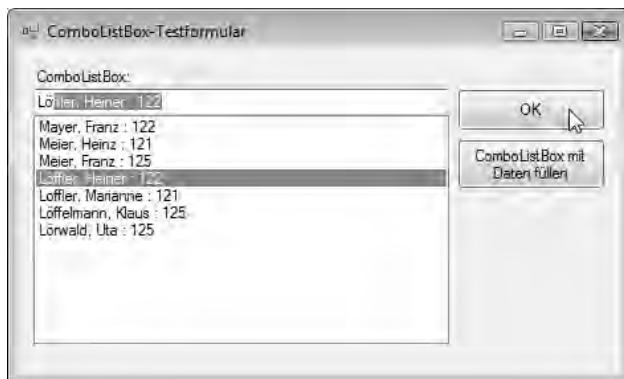


Abbildung 40.7 Mit der ComboBox vereinen Sie die Funktionalität von TextBox und ListBox

Der Aufwand für das Erstellen eines konstituierenden Steuerelements ist ein wenig intensiver, weil .NET die so genannte Mehrfachvererbung nicht zulässt: Eine Klasse kann nicht aus zwei Klassen erben. Aus diesem Grund werden die bestehenden Steuerelemente als Klassen-Member angelegt, und das Offenlegen entsprechender Eigenschaften, Ereignisse und Methoden kann nur durch »Delegierung« (zu den eingebundenen Klassen) erfolgen. Das heißt im Klartext: Im Gegensatz zur echten Vererbung müssen Elemente komplett neu implementiert werden, die sich bei Anwendung der richtigen Vererbung in der neuen Klasse ergeben würden.

HINWEIS Im Gegensatz zum vorherigen Beispiel, bei dem das Steuerelement Bestandteil eines einzelnen Projektes war, zeigt dieses Beispiel einen etwas aufwändigeren aber dafür sinnvolleren Weg bei der Implementierung des Steuerelements. Das Steuerelement wird nämlich hier in einer eigenen Assembly abgelegt, und wenn Sie die neue Komponente erst einmal entwickelt und ausgiebig getestet haben, können Sie sie durch einfaches Zurückgreifen auf das erstellte Projekt ohne Probleme immer wieder verwenden.

Natürlich finden Sie auch das folgende Beispiel in den Begleitdateien zum Buch. Dennoch ergeht an dieser Stelle wieder der Hinweis, dass es durchaus Sinn ergibt, die Erstellung des folgenden Projektes »manuell« nachzuvollziehen, um für spätere, eigene Entwicklungen vorbereitet zu sein.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel40\\ComboBoxTest

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Anlegen eines Projekts für die Erstellung einer eigenen Steuerelement-Assembly

Die folgende Schritt-für-Schritt-Anleitung zeigt, wie Sie vorgehen müssen, um ein Steuerelement in einem Projekt unterzubringen, aus dem nach dem Austesten und Erstellen eine eigene Assembly hervorgeht. Wenn Sie entscheiden, dass das Steuerelement einen finalen Versionsstand erreicht hat, und keine Modifikationen

mehr notwendig sind, dann können Sie diese Assembly zukünftig durch das Einrichten eines simplen Verweises in Sekundenschnelle anderen Projekten hinzufügen; die Quellcode-Dateien des Steuerelements benötigen Sie dann in zukünftigen Projekten nicht einmal mehr.

- Um eine neue Projektmappe zu erstellen, die sowohl das Projekt für die Steuerelement-Assembly als auch ein notwendiges Testprojekt enthält, legen Sie zunächst erst einmal eine simple Windows-Anwendung an. Wählen Sie dazu aus dem Menü *Datei* die Menüpunkte *Neu/Projekt*.
- Wählen Sie *Visual Basic/Windows* im Bereich *Projekttypen* und als Vorlage *Windows-Anwendung*.
- Bestimmen Sie das Projektmappen-Verzeichnis und geben Sie für das Windows-Anwendungsprojekt einen aussagekräftigen Namen ein – etwa *ComboBoxTest*.
- Klicken Sie auf *Projektmappen-Verzeichnis erstellen*, damit eine weitere Verzeichnisebene eingerichtet wird, in der sich später *beide* Projekte in separaten Verzeichnissen befinden.
- Klicken Sie auf *OK*, um sowohl Projektmappe als auch Projekt zu erstellen.

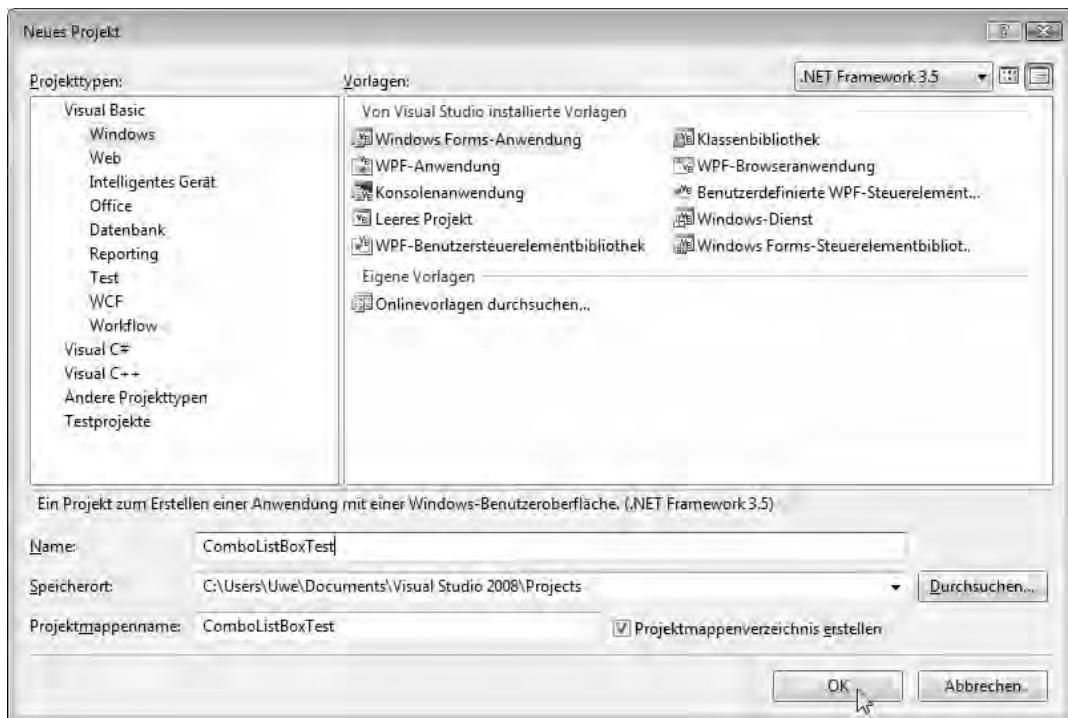


Abbildung 40.8 Legen Sie mit diesem Dialog zunächst die Projektmappe und das Projekt zum Testen des Steuerelements an

- Klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf die Projektmappe *ComboBoxTest* (nicht auf das Projekt!), und wählen Sie den Menübefehl *Hinzufügen/Neues Projekt*.
- Wählen Sie *Visual Basic/Windows* im Bereich *Projekttypen* und als Vorlage *Windows-Steuerelementbibliothek*.
- Geben Sie für das Windows-Anwendungsprojekt einen aussagekräftigen Namen ein – etwa *ADComboBox*.

- Klicken Sie auf **OK**, um das Projekt neu zu erstellen und der Projektmappe hinzuzufügen.
- Klicken Sie im Projektmappen-Explorer mit der rechten Maustaste über der Klassencodedatei *UserControl1.vb* des Projekts *ADComboBox*, und benennen Sie sie in *ADComboBox.vb* um.

Einrichten von Namespace, Assembly-Namen und Projektverweise

Im folgenden Schritt bestimmen Sie den Namespace sowie den Namen der Assembly-DLL, unter der das Steuerelement abrufbar wird.

- Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf den Projektnamen *ADComboBox*, und wählen Sie im sich jetzt öffnenden Kontextmenü den Menüpunkt *Eigenschaften*.

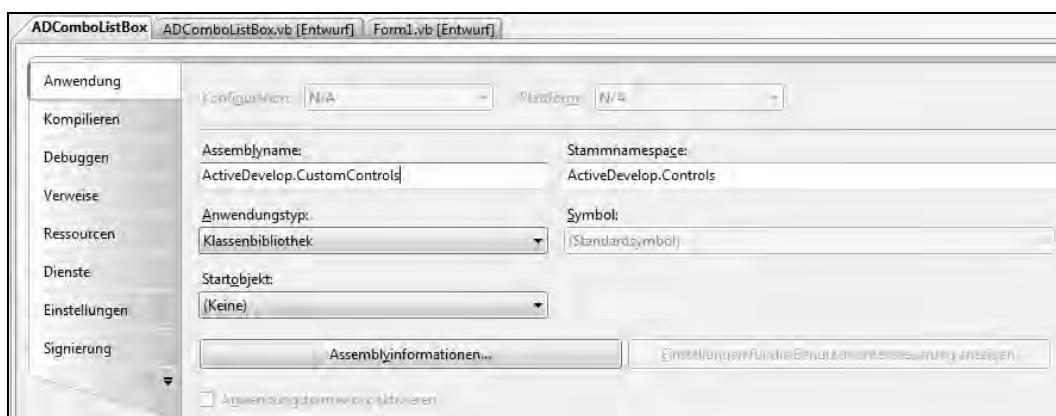


Abbildung 40.9 Der Assemblyname ist auch der Name der später aus dem Projektkomplilit hervorgehenden DLL; über den Stammnamespacenamen greifen Sie auf die Klasse(n) selbst zu

- Unter *Assemblyname* bestimmen Sie den Namen der Assembly, die später die Klasse Ihres Steuerelements beinhaltet soll. Diesen Namen wird auch die *DLL*-Datei tragen, die die Assembly mit Ihrem Steuerelement enthält. Für dieses Beispiel habe ich eine Kombination aus Firmennamen und »Custom-Controls« verwendet – denn diese Assembly soll eventuell später einmal dazu gedacht sein, weitere Benutzersteuerelemente zu beinhalten. Der Name der DLL lautet also nach dem Kompilieren *ActiveDevelop.CustomControls.dll*.
- Unter *Stammnamespace* bestimmen Sie, unter welchem Namespace die Steuerelementklasse (oder später einmal die Klassen, falls mehrere Klassen dem Projekt hinzugefügt werden sollen) abrufbar ist (bzw. sind). Für dieses Beispiel habe ich eine Kombination aus Firmennamen und *Controls* gewählt. Dazu ein wenig mehr Hintergrundinformationen: Genau so, wie Sie beispielsweise die Zeile

Imports System.IO

vor eine Klassencodedatei einfügen, um auf die I/O-Funktionen des Frameworks zugreifen zu können, benötigen Sie später, im *ComboBoxTest*-Programm, die folgende Anweisung

Imports ActiveDevelop.Controls

um auf die Klasse zugreifen zu können, die das *ADComboBox*-Steuerelement darstellt.

- Die Imports-Anweisung alleine reicht allerdings nicht. Damit die eigentlichen Funktionsnamen, die Sie erst über die Einbindung des Namespaces ActiveDevelop.Controls für den Compiler erkennbar machen, auch später tatsächlich im IML-Code gefunden und aufgelöst werden können, müssen Sie zusätzlich einen Verweis auf die eigentliche Assembly (das Projekt) einrichten. Dazu öffnen Sie das Kontextmenü des Projektes *ComboBoxTest* im Projektmappen-Explorer und wählen dort den Eintrag *Verweis hinzufügen*.
- Wechseln Sie im Dialog, der jetzt erscheint (siehe Abbildung 40.10), auf die Registerkarte *Projekte*, doppelklicken Sie auf den Eintrag *ADComboBox*, und klicken Sie anschließend auf *OK*.

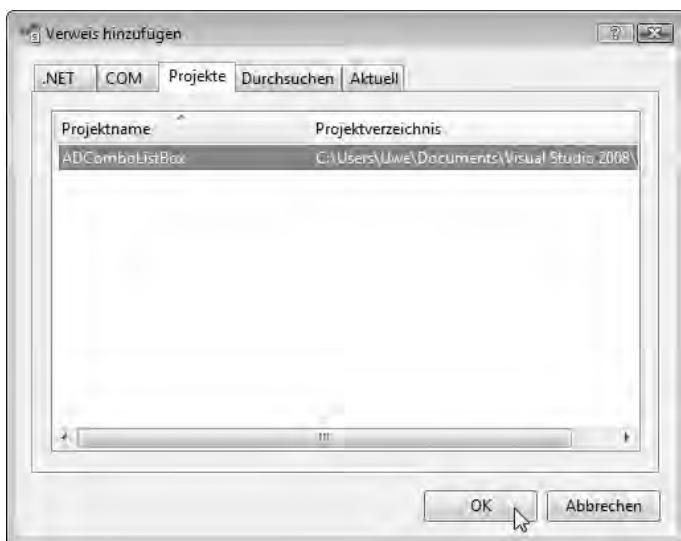


Abbildung 40.10 Wenn Sie mit Klassen anderer Projekte Ihrer Projektmappe arbeiten wollen, müssen Sie ebenfalls Verweise auf die benötigten Projekte hinzufügen

Anordnen von Steuerelementen im benutzerdefinierten Steuerelement

Das Steuerelement, das wir in diesem Beispiel entwickeln, besteht aus zwei im .NET Framework schon vorhandenen Steuerelementen: Einer TextBox und einer ListBox. Sie können diese beiden Steuerelemente im Benutzersteuerelement genau wie in einem Formular anordnen und mit den entsprechenden Eigenschaften ausstatten.

- Öffnen Sie dazu den Designer zur Klassendatei *ADComboBox.vb* aus dem Projekt *ADComboBox* per Doppelklick auf den entsprechenden Eintrag im Projektmappen-Explorer.
- Ziehen Sie jetzt jeweils ein TextBox - und ein ListBox -Steuerelement im Steuerelementcontainer auf, etwa wie in Abbildung 40.11 zu sehen (bitte in dieser Reihenfolge, da es sonst Probleme bei der späteren Einstellung der Dock-Eigenschaft geben kann).

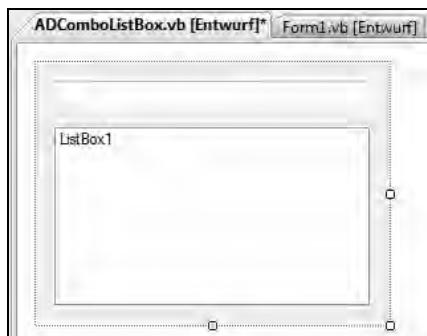


Abbildung 40.11 In einem Benutzersteuerelement können Sie Steuerelemente genau wie in einem Formular einfügen

- Setzen Sie die Dock-Eigenschaft der TextBox auf Top und die Dock-Eigenschaft der ListBox auf Fill. Die Dock-Eigenschaft bewirkt, dass sich die Ränder eines Steuerelements an die entsprechenden Innenseiten ihres Containers fügen. Wird der Container vergrößert oder verkleinert, passen sich die gedockten Steuerelemente innerhalb des Containers an. So gesehen funktioniert die Dock-Eigenschaft fast wie die Anchor-Eigenschaft, mit dem Unterschied, dass gedockte Steuerelemente immer am äußersten Rand des sie umgebenden Containers verankert werden.

TIPP Die so genannte Z-Reihenfolge der Steuerelemente ist wichtig, wenn Sie mit umfangreichen, gedockten Steuerelementen arbeiten. Die Z-Reihenfolge bestimmt, welches Steuerelement über oder unter einem anderen liegt. Durch das Öffnen des Kontextmenüs eines Steuerelements im Designer können Sie dessen Position in der Z-Reihenfolge mit den Funktionen *In den Hintergrund* oder *In den Vordergrund* verändern. Auf diese Weise ändern Sie auch deren Docking-Verhalten untereinander.⁵

- Benennen Sie die Steuerelemente um, damit sie den bisherigen Standards der Namensgebung von Klassen-Membern entsprechen: Nennen Sie das ListBox-Steuerelement `myListBox` und das TextBox-Steuerelement `myTextBox`.

HINWEIS Sobald Sie Änderungen an den Eigenschaften eines Steuerelements im Designer vorgenommen haben und das Steuerelementprojekt bereits erstellt wurde, zeigt Visual Studio eine Warnmeldung in der Aufgabenliste und weist sie mit dieser darauf hin, dass Sie das Projekt neu erstellen müssen, damit sich die Änderungen auf alle Instanzen auswirken, die das Steuerelement einbinden.

Die Grundeinstellungen für das Steuerelementprojekt sind damit abgeschlossen. Sie können jetzt das Projekt das erste Mal erstellen lassen und so die Assembly für das Steuerelement generieren.

WICHTIG Wenn Sie konstituierende Benutzersteuerelemente in der Visual Studio-Umgebung erstellen, wird ein entsprechendes Toolbox-Symbol automatisch eingefügt. Sie können auf das neue Steuerelement ohne weitere Maßnahmen direkt zugreifen. Das Steuerelementprojekt muss dazu allerdings mindestens einmal erstellt worden sein.

⁵ Das kann in einigen Fällen in eine ziemliche Fummelie ausarten. Sie sollten sich daher überlegen, ob Sie mehrere Steuerelemente in solchen Fällen nicht in weiteren Containern zusammenfassen können und nur die Container docken. In vielen Fällen könnte auch die Verwendung der Anchor-Eigenschaft die bessere Alternative darstellen.

- Das neue Steuerelement ist nun zum Einsatz bereit, und Sie können es im vorerst letzten Schritt testweise im Formular des *ADComboBoxTest*-Projektes platzieren.

BEGLEITDATEIEN Um die Implementierung beschreibungstechnisch abzukürzen, möchte ich für die nächsten Erklärungen auf das bereits fertige Projekt verweisen, das Sie, wie schon eingangs erwähnt, im Verzeichnis

... \VB 2008 Entwicklerbuch\ G - WinForms\Kapitel40\ComboBoxTest\ ComboBoxTest.sln

finden.

Initialisieren des Steuerelements

Im Gegensatz zum vorherigen Beispiel, bei dem das neue Steuerelement aus einem vorhandenen erbte, müssen Sie bei diesem Beispielprojekt viele Eigenschaften erneut implementieren. Das gilt nicht nur für die Eigenschaften, sondern auch für die wichtigsten Ereignisse. Aus diesem Grund ist der Deklarationsteil sowie der Konstruktor dieses Beispiels ein wenig umfangreicher, wie der folgende Codeausschnitt zeigt:

```
Imports System.ComponentModel

Public Class ADComboBox
    Inherits System.Windows.Forms.UserControl

    Private mySenderIsThis As Boolean
    Private myAutoComplete As Boolean
    Private myAutoSelect As Boolean

    Public Event SelectedIndexChanged(ByVal sender As Object, ByVal e As EventArgs)
    Public Event SelectedValueChanged(ByVal sender As Object, ByVal e As EventArgs)

    Public Sub New()
        MyBase.New()

        ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
        InitializeComponent()

        ' Eigenschaften vorselektieren.
        myAutoComplete = True
        myAutoSelect = True

        ' Die Listbox soll nicht mit der Tabulatortaste fokussiert werden dürfen!
        myListBox.TabStop = False
    End Sub
```

Die beiden Steuerelemente, mit denen unser Benutzersteuerelement arbeitet, sind nichts anderes als Instanzen einer bestimmten Klasse, die vom Konstruktor der Steuerelementklasse erzeugt werden (zwar nicht direkt, sondern durch `InitializeComponent`, aber immer noch im Rahmen des Konstruktorkodes). Wollten Sie ein konstituierendes Steuerelement ohne Designer-Unterstützung erstellen, würden Sie im Prinzip genau so vorgehen, wie es der Designer in diesem Beispiel für Sie gemacht hat – den entsprechenden Code finden Sie in der Codedatei *ADComboBox.Designer.vb*, wenn Sie für das Projekt *ADComboBox* im Projektmappen-Explorer alle Dateien anzeigen lassen, und er ist im Folgenden zu sehen:

```
<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
Partial Class ADComboBox
    Inherits System.Windows.Forms.UserControl

    'UserControl1 überschreibt den Löschtvorgang, um die Komponentenliste zu bereinigen.
    <System.Diagnostics.DebuggerNonUserCode()>
    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing AndAlso components IsNot Nothing Then
            components.Dispose()
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Wird vom Windows Form-Designer benötigt.
    Private components As System.ComponentModel.IContainer

    'Hinweis: Die folgende Prozedur ist für den Windows Form-Designer erforderlich.
    'Das Bearbeiten ist mit dem Windows Form-Designer möglich.
    'Das Bearbeiten mit dem Codeeditor ist nicht möglich.
    <System.Diagnostics.DebuggerStepThrough()>
    Private Sub InitializeComponent()
        Me.myTextBox = New System.Windows.Forms.TextBox
        Me myListBox = New System.Windows.Forms.ListBox
        Me.SuspendLayout()
        '
        'myTextBox
        '
        Me.myTextBox.Dock = System.Windows.Forms.DockStyle.Top
        Me.myTextBox.Location = New System.Drawing.Point(0, 0)
        Me.myTextBox.Name = "myTextBox"
        Me.myTextBox.Size = New System.Drawing.Size(185, 20)
        Me.myTextBox.TabIndex = 0
        '
        'myListBox
        '
        Me myListBox.Dock = System.Windows.Forms.DockStyle.Fill
        Me myListBox.FormattingEnabled = True
        Me myListBox.Location = New System.Drawing.Point(0, 20)
        Me myListBox.Name = "myListBox"
        Me myListBox.Size = New System.Drawing.Size(185, 95)
        Me myListBox.TabIndex = 1
        '
        'ADComboBox
        '
        Me.AutoScaleDimensions = New System.Drawing.SizeF(6.0!, 13.0!)
        Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font
        Me.Controls.Add(Me myListBox)
        Me.Controls.Add(Me myTextBox)
        Me.Name = "ADComboBox"
        Me.Size = New System.Drawing.Size(185, 121)
        Me.ResumeLayout(False)
        Me.PerformLayout()
        Me.PerformLayout()

    End Sub
    Friend WithEvents myTextBox As System.Windows.Forms.TextBox
    Friend WithEvents myListBox As System.Windows.Forms.ListBox

End Class
```

Es gilt dabei dasselbe wie für Formulare: Sobald Sie eine von Control abgeleitete Klasseninstanz der Controls-Auflistung hinzugefügt haben, erscheint es im Steuerelement (gültige Positionseinstellungen vorausgesetzt).

Ebenfalls im Gegensatz zum vorherigen Beispiel müssen wir in unserem Code zumindest die beiden Ereignisse SelectedIndexChanged und SelectedValueChanged neu definieren, denn: Sie werden von myListBox ausgelöst und können bestenfalls durch das Benutzersteuerelement eingebunden werden. Da es keine Vererbung der ListBox gibt, ist mit der Ereigniskette beim Benutzersteuerelement aber auch bereits Schluss. Würden wir die Ereignisse nicht einbinden, unser Steuerelement darüber hinaus selbst mit diesen Ereignissen ausstatten und diese zu gegebener Zeit wieder auslösen, hätte der Entwickler, der unser Steuerelement verwendet, keine Chance, vom Ereigniseintritt zu erfahren – daher die neue Definition.

TIPP Wenn Sie komplexere konstituierende Steuerelemente erstellen, und die Steuerelemente, auf denen es basiert, mit Werten vorbelegen müssen, ist der Konstruktor nach *InitializeComponent* die richtige Stelle, um das zu erreichen.

Methoden und Ereignisse delegieren

Was für Ereignisse und Eigenschaften notwendig ist, gilt für Methoden gleichermaßen. Es ist anzunehmen, dass der Entwickler, der Ihr Steuerelement verwendet, bestimmte Funktionen der TextBox und der ListBox benötigt. Ihnen bleibt auch dann nichts weiter übrig, als die entsprechenden Methoden zu implementieren und die Parameter zum entsprechenden Steuerelement hinab zu delegieren, wie im Beispiel:

```
'Diese Funktionen nach unten durchrouten, weil diese Instanz...
Public Function FindString(ByVal s As String) As Integer
    Return myListBox.FindString(s)
End Function

'...aber auch andere Instanzen sie häufig...
Public Function FindString(ByVal s As String, ByVal startIndex As Integer) As Integer
    Return myListBox.FindString(s, startIndex)
End Function

'...benötigen.
Public Function FindStringExact(ByVal s As String) As Integer
    Return myListBox.FindStringExact(s)
End Function

Private Sub myTextBox_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles myTextBox.TextChanged
    'Löst, durch die Veränderung der Text-Eigenschaft, OnTextChanged des UserControls aus
    Me.Text = myTextBox.Text
    Console.WriteLine(Me.Text)
End Sub

'Für unsere Zwecke müssen wir nur dieses Ereignis nach oben routen, aber...
Private Sub myTextBox_KeyDown(ByVal sender As Object, ByVal e As System.Windows.Forms.KeyEventArgs) _
    Handles myTextBox.KeyDown
    Me.OnKeyDown(e)
End Sub

'andere Anwendungen benötigen vielleicht auch dieses...
Private Sub myTextBox_KeyPress(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyPressEventArgs) _
```

```

Handles myTextBox.KeyPress
Me.OnKeyPress(e)
End Sub

'...und dieses Ereignis.
Private Sub myTextBox_KeyUp(ByVal sender As Object, ByVal e As System.Windows.Forms.KeyEventArgs) _
    Handles myTextBox.KeyUp
    Me.OnKeyUp(e)
End Sub

'Der Entwickler, der unser Steuerelement verwendet, muss mitbekommen, wenn sich die Auswahl ändert.
'Aus diesem Grund müssen wir diese beiden Ereignisse nach oben durchreichen.
Private Sub myListBox_SelectedIndexChanged(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles myListBox.SelectedIndexChanged
    OnSelectedIndexChanged(e)
End Sub

```

HINWEIS Rein theoretisch könnten Sie die Steuerelemente, aus denen das Benutzersteuerelement besteht, natürlich auch als *Public* deklarieren – doch davon ist dringend abzuraten! Der Entwickler könnte in diesem Fall Eigenschaften der basierenden Steuerelemente verändern, und das Benutzersteuerelement würde nichts von den Veränderungen mitbekommen – die Funktionalität könnte dadurch stark beeinträchtigt werden – bis hin zum völligen Versagen!

Implementieren der Funktionslogik

Die Implementierung der Funktionslogik ist der des vorherigen Beispiels sehr ähnlich. Der einzige Unterschied: Durch die Natur des Steuerelements ist die Liste ständig sichtbar, aber beide Steuerelemente sind in keiner Weise miteinander verbunden. Die Vorselektierung des Eintrags in der *ListBox*, der zur bisherigen Eingabe in die *TextBox* passt, muss deswegen programmiert werden. Dieses Verhalten lässt sich übrigens mit der *AutoSelect*-Eigenschaft des Benutzersteuerelements steuern. Bis auf diesen Unterschied werden Sie in der Implementierung überwiegend Übereinstimmungen zum vorherigen Beispiel bemerken:

```

Protected Overridable Sub OnSelectedIndexChanged(ByVal e As EventArgs)
    'Den Listeneintrag in die TextBox kopieren,
    'dabei ungewollte Rekursion verhindern,
    mySenderIsThis = True
    Me.Text = myListBox.SelectedItem.ToString
    mySenderIsThis = False
    'Ereignis auslösen, damit auch andere Instanzen vom Ereignis erfahren.
    RaiseEvent SelectedIndexChanged(Me, e)
End Sub

Private Sub myListBox_SelectedIndexChanged(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles myListBox.SelectedIndexChanged
    OnSelectedIndexChanged(e)
End Sub

Protected Overridable Sub OnSelectedValueChanged(ByVal e As EventArgs)
    RaiseEvent SelectedValueChanged(Me, e)
End Sub

'Wird aufgerufen, *weil* der Handler myTextBox_TextChanged der TextBox die Text-Eigenschaft

```

```

'des UserControls verändert.
'Wird damit aufgerufen, *sobald* irgendeine Veränderung im Eingabefeld vorgenommen wird
'(ganz gleich, ob programmtechnisch oder durch den Anwender).
Protected Overrides Sub OnTextChanged(ByVal e As System.EventArgs)
    'Dieses Flag wird benötigt, damit eine Eingabebereichsveränderung, die
    'programmtechnisch durch das Steuerelement selbst vorgenommen wird, das Ereignis
    'nicht erneut auslöst, und das Programm in einer Endlosschleife hängen bleibt.
    Console.WriteLine("OnTextChanged")
    If Not mySenderIsThis Then
        'Verhindern, dass das Ereignis durch eigene Manipulation erneut ausgelöst wird.
        mySenderIsThis = True
        'Auto-Vervollständigen durchführen
        PerfAutoCompletion()
        mySenderIsThis = False
    End If
    'Basis-Funktion aufrufen nicht vergessen!
    MyBase.OnTextChanged(e)
End Sub

Private Sub PerfAutoCompletion()

    'Zwischenspeichern, damit die Originaleingabe erhalten bleibt.
    Dim locTemp As String = Me.Text
    'Index des Eintrags finden, der der bisherigen Texteingabe entspricht.
    Dim locIndex As Integer = Me.FindString(locTemp)
    'Nur wenn AutoComplete eingeschaltet ist, überhaupt etwas machen.
    If AutoComplete Then
        If locIndex > -1 Then
            'Eintrag gefunden, zum einfacheren Handling in Variable kopieren.
            Dim locFoundEntry As String = myListBox.Items(locIndex).ToString
            If locIndex > -1 Then
                'Eintrag ins Eingabefeld kopieren.
                myTextBox.Text = locFoundEntry
                'Den noch nicht eingegebenen Teil markieren, damit
                'er einfach überschrieben werden kann.
                myTextBox.Select(locTemp.Length, locFoundEntry.Length - locTemp.Length)
            End If
        End If
    End If

    'AutoSelect-Eigenschaft im Bedarfsfall umsetzen:
    If AutoSelect Then
        'Passenden Eintrag gefunden, ...
        If locIndex > -1 Then
            '...dann selektieren.
            myListBox.SelectedIndex = locIndex
        End If
    End If
End Sub

'Wird ausgelöst, wenn eine Taste in der TextBox gedrückt wird.
Protected Overrides Sub OnKeyDown(ByVal e As System.Windows.Forms.KeyEventArgs)
    Console.WriteLine("OnKeyDown")
    'Cursor-nach-unten, dann...
    If e.KeyCode = Keys.Down Then

```

```
'Testen ob die Listbox nicht schon fokussiert ist, und ob bereits Einträge vorhanden sind.
If Not myListBox.Focused AndAlso myListBox.Items.Count > 0 Then
    '...ListBox fokussieren,
    myListBox.Focus()
    'Wenn noch kein Eintrag in der Listbox ausgewählt war,
    If myListBox.SelectedIndex = -1 Then
        'ersten Eintrag selektieren.
        myListBox.SelectedIndex = 0
    End If
    e.Handled = True
End If
Else
    'Wenn Delete oder Backspace gedrückt wird, dann keine Modifizierungen vornehmen.
    mySenderIsThis = (e.KeyCode = Keys.Delete) Or (e.KeyCode = Keys.Back)
End If
 MyBase.OnKeyDown(e)
End Sub
```

Implementierung der Eigenschaften

Bei der Implementierung der neuen Eigenschaften verfahren wir fast genauso wie im vorherigen Beispiel – doch da wir es mit einem konstituierenden Steuerelement zu tun haben, ist es damit wieder nicht getan. Einige Eigenschaften müssen wieder nach oben hoch gereicht werden – in diesem Beispiel habe mich auf die Items-Eigenschaft der ListBox und die Text-Eigenschaft der TextBox beschränkt:

```
'Hier kommen die Eigenschaften:
<Description("Bestimmt oder ermittelt, ob die Auto-Ergänzen-Funktion verwendet wird."), _
Category("Verhalten"),
DefaultValue(GetType(Boolean), "True"), _
Browsable(True)>
Public Property AutoComplete() As Boolean
    Get
        Return myAutoComplete
    End Get
    Set(ByVal Value As Boolean)
        myAutoComplete = Value
    End Set
End Property

<Description("Bestimmt oder ermittelt, ob mit dem Text übereinstimmende Einträge automatisch selektiert werden."), _
Category("Verhalten"),
DefaultValue(GetType(Boolean), "True"), _
Browsable(True)>
Public Property AutoSelect() As Boolean
    Get
        Return myAutoSelect
    End Get
    Set(ByVal Value As Boolean)
        myAutoSelect = Value
    End Set
End Property

'Die Texteigenschaft wird vom Designer für das UserControl und deren Ableitungen unterdrückt.
'Mit Browsable(True) machen wir es im Eigenschaftenfenster wieder darstellbar.
<Browsable(True)> _
```

```

Public Overrides Property Text() As String
    Get
        Return MyBase.Text
    End Get
    Set(ByVal Value As String)
        'Neuen Text der TextBox zuordnen.
        myTextBox.Text = Value
        ''WICHTIG: Wenn diese Anweisung fehlt, wird OnTextChanged nicht aufgerufen,
        ''wenn die Texteigenschaft neu zugewiesen wird!
        MyBase.Text = Value
    End Set
End Property

<Description("Die Elemente im Listenfeld."), _
Category("Daten"), _
Browsable(True)>
Public ReadOnly Property Items() As ListBox.ObjectCollection
    Get
        Return myListBox.Items
    End Get
End Property

End Class

```

Das Aufrufen der Basisroutine der Text-Eigenschaft ist übrigens in diesem Beispiel besonders wichtig, da andernfalls das OnTextChanged-Ereignis nicht ausgelöst wird. Die folgende Ereigniskette würde dann nämlich unterbrochen, und weder Autokomplettieren noch Vorselektieren in der Liste würden funktionieren:

- Der Anwender gibt ein Zeichen in der Textbox ein.
- Das TextChange-Ereignis wird ausgelöst, das von `myTextBox_TextChanged` behandelt wird.
- Diese Ereignisbehandlungsroutine setzt die Text-Eigenschaft des Benutzersteuerelements mit dem (nun geänderten) Inhalt der TextBox – der Set-Part der Text-Eigenschaftsprozedur wird ausgeführt. Da der Text, der hier `myTextBox` wieder zugewiesen wird, nicht nur der gleiche, sondern derselbe ist, wird kein erneutes TextChange-Ereignis ausgelöst. Die Gefahr einer unfreiwilligen Rekursion besteht hier also nicht.
- Durch `MyBase.Text` (vorletzte Zeile im fett markierten Code) ändert sich die Text-Eigenschaft des Benutzersteuerelements, was zur Ausführung von `OnTextChanged` des Benutzersteuerelements führt. Hier kann jetzt die eigentliche Funktionslogik des Steuerelements ausgeführt werden.

Erstellen von Steuerelementen von Grund auf

Wenn Sie Steuerelemente erstellen wollen, die nicht auf Funktionalitäten von einem oder mehreren bereits vorhandenen Steuerelementen basieren sollen, dann verfahren Sie prinzipiell so, wie es der Abschnitt »Neue Steuerelemente auf Basis vorhandener Steuerelemente implementieren« ab Seite 1171 erklärt. Der einzige Unterschied: Ihre neue Steuerelementklasse basiert auf der Control-Klasse. Anstatt also die Klasse mit

```

Public Class ADComboBox
    Inherits ComboBox

```

zu definieren, verwenden Sie die Anweisungen

```
Public Class GanzNeuesControl
    Inherits Control
```

Was das Anlegen einer Projektmappe für ein Steuerelement anbelangt, das Sie von Grund auf neu erstellen wollen, bleibt alles andere genau gleich – deswegen möchte ich die erforderlichen Schritte aus Platzgründen hier nicht wiederholen. Um wirklich professionelle Steuerelemente entwickeln zu können, ist beim eigentlichen Ausformulieren des Codes allerdings viel mehr Handarbeit erforderlich, als bei Steuerelementen, die auf anderen basieren. Haben Sie jedoch erst einmal einen Blick hinter die Kulissen gewagt und verstanden, welche Techniken angewendet werden müssen, um stabile Komponenten dieser Art zu entwerfen, werden Sie diesen Vorzug des Frameworks nicht mehr missen wollen. Mit dieser Methode, wieder verwendbare Komponenten entwerfen zu können, sparen Sie auf Dauer nicht nur eine Menge Arbeit, Sie werden den enormen Komfort, den Sie sich mit dem anfänglich zusätzlichen Aufwand erkaufen müssen, schnell schätzen lernen.

Ein Label, das endlich alles kann

O.k., »alles« ist vielleicht ein wenig übertrieben, und man mag kaum glauben, dass ein Steuerelement, das auf den ersten Blick nichts weiter macht, als Elemente einer Benutzerumgebung zu beschriften, tatsächlich verbesserungswürdig ist. Doch ich bin mir sicher, dass Sie Ihre Meinung ändern, wenn Sie einen Blick auf das folgende Beispiel geworfen haben.

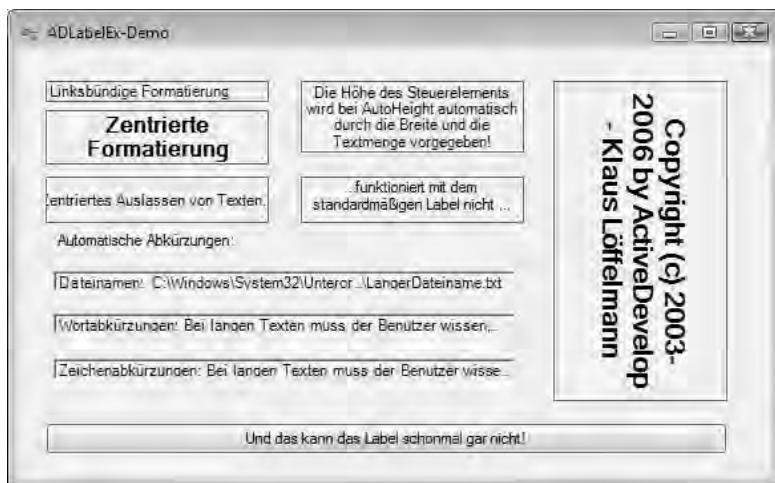


Abbildung 40.12 Dieses Demo-programm demonstriert die Leistungsstärke des Steuerelementes ADLabelEx

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\\VB 2008 Entwicklerbuch\\G - WinForms\\Kapitel40\\LabelExDemo
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie dieses Programm starten, sehen Sie einen Dialog, etwa wie in Abbildung 40.12 zu sehen. Schon in diesem Dialog werden Sie die Verbesserungen im Vergleich zum herkömmlichen Label-Steuerelement bemerken:

- `ADLabelEx` ermöglicht es, Beschriftungen mit um 90 Grad gedrehtem Text durchzuführen.
- Mit bestimmten Eigenschaften können automatische Abkürzungen von Text realisiert werden – demonstriert durch die `ADLabelEx`-Elemente im Rahmen des Formulars. Das Standard-Label vom .NET Framework kann das dank seiner `AutoEllipses`-Eigenschaft zwar auch, versagt aber bei der Platzierung des Textes, wenn dieser in der Mitte eines Steuerelements angeordnet werden soll. Außerdem beschränkt es sich auf eine Auslassungsform. `ADLabelEx` kennt die `TextTrimming`-Eigenschaft, mit der Sie das Auslassungsverfahren einstellen können.
- Auch hier zu sehen: `ADLabelEx` verfügt über eine `AutoHeight`-Eigenschaft, mit der sich das Label in der Höhe automatisch an den Textumfang anpasst.
- Zu sehen, wenn Sie auf die Schaltfläche klicken: Mit der `Flash`-Eigenschaft können Sie jede `ADLabelEx`-Instanz zum Blinken bringen. Diese Möglichkeit kann die Benutzerfreundlichkeit von Anwendungen extrem steigern, gerade wenn es um die Darstellung wichtiger Hinweise und Warnungen oder die Kennzeichnung von Eingabefehlern in Formularen geht.

Neugierig geworden? Die folgenden Abschnitte zeigen, wie die verschiedenen Funktionen in der Steuerelementklasse `ADLabelEx` implementiert wurden.

Vorüberlegungen und Grundlagenerarbeitung

Die Grundidee für dieses Steuerelement ist eigentlich aus einer realen Notwendigkeit heraus entstanden. Bei der Gestaltung von Formularen mit umfangreichen Eingabefeldern fiel mir auf, dass sich rechtsbündig formatierte Texte im Framework 1.1 nicht wirklich bündig untereinander platzieren ließen. Je länger der Text eines Labels wurde, desto weiter entfernte er sich vom rechten Rand. Da rechtsbündig vor dem Eingabefeld platzierte Labels für die Benutzerfreundlichkeit aber am förderlichsten sind, galt es, eine Lösung zu finden – und diese bestand in der Entwicklung dieses Steuerelements. Dieses Manko wurde mit dem Framework 2.0 (und insbesondere der `UseCompatibleTextRendering`-Eigenschaft) zwar behoben, aber viele weitere Features rechtfertigen dieses Demo-Steuerelement natürlich immer noch.

Textausgabe mit `DrawString` und dem GDI+

`DrawString` des GDI+ stellt eigentlich schon alle Funktionen zur Verfügung, die man zur Realisierung eines Label-Steuerelements benötigt. Mit `DrawString` können Sie bestimmen, wie Texte umgebrochen oder abgekürzt werden sollen, wenn sie nicht in einen definierten Bereich passen. `DrawString` erlaubt darüber hinaus mit dem `StringFormat`-Objekt das Einstellen individueller Ausrichtungen (linksbündig, oben; mittig; rechtsbündig unten), wenn der String in einem rechteckigen Bereich ausgegeben werden soll. Allerdings produziert `DrawString` unter bestimmten Umständen auch Darstellungsfehler, gerade bei zentrierter und rechtsbündiger Formatierung. Der Grund dafür ist Auflösungsunabhängigkeit bei der Verwendung von `DrawString`:

Die Buchstaben eines Zeichensatzes sind vergleichsweise kleine Objekte, gemessen an der relativ geringen Auflösung, die ein Bildschirm darzustellen in der Lage ist. GDI+ kann deswegen gerade bei kleinen Fonts auf nur relativ wenig Pixel zurückgreifen, um einen Buchstaben auf dem Bildschirm anzuzeigen. Die einzelnen Buchstaben sind aus diesem Grund nicht so lang, wie sie es rein rechnerisch eigentlich sein sollten. Dennoch werden sie in vielen Fällen direkt nebeneinander platziert, sodass sich eine Lücke am Ende des Strings bildet. Und das hat dann genau diesen Effekt: Die theoretisch berechnete Stringlänge weicht von der

Länge des tatsächlich dargestellten Textes ab; und dieser Fehler wird umso deutlicher, wenn der auszugebende Text aus vielen Buchstaben besteht. Allerdings – und das ist merkwürdig – zeigt sich dieses Verhalten nur dann, wenn `DrawString` einzeilige Texte auf dem Bildschirm ausgibt. Bei mehrzeiligen Texten ist dieses Verhalten nicht zu beobachten. Und genau das ist auch des Problems Lösung: Aus diesem Grund hängt die `OnPaint`-Funktion des Steuerelements, die den Text in den sichtbaren Bereich des Steuerelements malt, einfach ein Carriage Return (ASCII 13) hinter den eigentlich auszugebenden Text, und schon erscheinen rechtsbündige Texte im vorgegebenen Rahmen wirklich am rechten Rand ausgerichtet.

Verwenden von Timer-Objekten zum Auslösen von Ereignissen in Intervallen

Auch was den Blinkmechanismus anbelangt, sind einige Vorüberlegungen anzustellen, denn: Natürlich muss das Blinken in irgendeiner Form getriggert werden, und dazu bietet sich das `Timer`-Objekt an. Mit einem `Timer`-Objekt haben Sie die Möglichkeit, ein Intervall zu bestimmen (mit seiner `Interval`-Eigenschaft), nach deren Ablauf ein Ereignis (das `Tick`-Ereignis) ausgelöst wird. Anstatt nun für jede in einem Programm vorhandene `ADLabelEx`-Instanz ein eigenes `Timer`-Objekt zu instanziieren – was Verschwendug von Rechenzeit und Resourcen bedeuten würde – besteht natürlich die ungleich bessere Möglichkeit, nur ein einziges `Timer`-Objekt zu verwenden, das *alle* verwendeten `Label` triggert. Das hat den zusätzlichen Vorteil, dass Texte, wenn sie blinken, synchronisiert blinken. Mit diesen Vorüberlegungen im Hinterkopf können wir uns nun an das eigentliche Codieren des Steuerelements heranwagen:

Klasseninitialisierungen und Einrichten der Windows-Darstellungsstile des Steuerelements

Damit das Steuerelement ordnungsgemäß funktionieren kann, müssen bei seiner Instanziierung zwei grundlegende Dinge gewährleistet sein:

- Klassen-Member-Variablen müssen initialisiert werden.
- Der grundsätzliche Windows-Stil für das Steuerelement muss definiert werden.

Der zweite Punkt bedarf hier einer genaueren Erklärung: Wie Sie in Kapitel 38 schon erfahren konnten, basieren sämtliche Bedienungselemente von Benutzeroberflächen unter dem Windows-Betriebssystem auf einem nativen Element, das sich (sehr ungünstig für Erklärungen) *Window* nennt. Ein *Window* im Betriebssystem-Sinn kann die Basis für eine Schaltfläche, einen Tooltip, ein »echtes« Dokumentenfenster, ein .NET-Formular und natürlich auch für ein Steuerelement sein. *Window* ist dabei aber niemals gleich *Window*: Es gibt welche mit und ohne Rollbalken, solche mit Rahmen oder eingelassenem Rahmen im 3D-Look, Fenster, die grundsätzlich über anderen Fenstern liegen usw.

Wenn ein auf `Control` basierendes Objekt in .NET erstellt wird, hat es durch Überschreiben der `CreateParams`-Eigenschaft die Möglichkeit, diese grundsätzlichen Stile für das zugrunde liegende *Window* zu bestimmen. Auf diese Art wird beispielsweise für jedes Steuerelement dessen `BorderStyle`-Eigenschaft umgesetzt; das Zeichnen des entsprechenden Rahmens wird hierbei – entgegen vielen Vermutungen – nicht durch GDI+-Funktionen, sondern durch das Windows-Betriebssystem selbst realisiert.

Der Klassencode, der zum Einen die Member-Variablen mit sinnvollen Werten vorbelegt und zum Anderen dafür sorgt, dass das basierende »Windows-Window« korrekt eingerichtet wird, sieht daher folgendermaßen aus:

```
<Designer("ActiveDev.ADLabelExDesigner")> _
Public Class ADLabelEx
    Inherits Control

    Private Const WS_BORDER As Integer = &H800000
    Private Const WS_EX_CLIENTEDGE As Integer = &H200
    Private Const myFlashInterval As Integer = 400

    Private Shared myFlashTimer As Timer

    Private myBorderStyle As BorderStyle
    Private myTextAlign As ContentAlignment
    Private myUseMnemonic As Boolean
    Private myDirectionVertical As Boolean

    Private myAutoHeight As Boolean
    Private myRequestedHeight As Integer
    Private myTextWrap As Boolean
    Private myTextTrimming As StringTrimming

    Sub New()
        MyBase.new()
        'Eigenschaften initialisieren.
        myBorderStyle = BorderStyle.None
        myTextAlign = ContentAlignment.TopLeft
        myUseMnemonic = True
        myTextWrap = True
        myTextTrimming = StringTrimming.None
        myFlashBackColor = Color.Empty
        myFlashForeColor = Color.Empty

        'Windows-Stile setzen.
        SetStyle(ControlStyles.AllPaintingInWmPaint, True)
        SetStyle(ControlStyles.ResizeRedraw, True)
        SetStyle(ControlStyles.DoubleBuffer, True)

        'Initialwert für die Höhe merken.
        myRequestedHeight = Me.Height

        'Flash-Ereignishandler einrichten.
        AddHandler FlashOn, AddressOf FlashOnHandler
        AddHandler FlashOff, AddressOf FlashOffHandler
    End Sub

    'Definiert die Parameter für das Anlegen des "Windows-Window".
    Protected Overrides ReadOnly Property CreateParams() As CreateParams

        Get
            Dim params1 As CreateParams
            Dim style1 As BorderStyle
            params1 = MyBase.CreateParams

            'Möglichweise eingeschaltete BorderStyles ausschalten.
            params1.ExStyle = (params1.ExStyle And Not WS_EX_CLIENTEDGE)
            params1.Style = (params1.Style And Not WS_BORDER)
        End Get
    End Property
End Class
```

```
'Herausfinden, welcher Borderstyle eingeschaltet werden soll.
style1 = Me.myBorderstyle
Select Case style1 - 1

    Case 0
        'Simpler Rand
        params1.Style = (params1.Style Or WS_BORDER)

    Case 1
        'Drei-D-Rand
        params1.ExStyle = (params1.ExStyle Or WS_EX_CLIENTEDGE)
End Select
Return params1
End Get
End Property
```

HINWEIS CreateParams ist eine Eigenschaft, die das geschützte .NET-Zuhause schon fast verlässt, und mit deren Benutzung Sie sich nahezu inmitten des Windows-Betriebssystem befinden. Durch das CreateParams-Objekt können Sie großen Einfluss auf das Grundaussehen eines Steuerelements oder eines Formulars nehmen. Wenn Sie mehr aus dieser Eigenschaft herausholen wollen, sollten Sie sich mit der Programmierung des Windows-Betriebssystems, insbesondere der nativen Fenstersteuerung, einigermaßen vertraut machen. Die Visual Studio-Hilfe zum Suchbegriff CreateWindowEx (eine native Windows-Funktion) verrät Ihnen mehr zu diesem Thema.

Übrigens: Wenn Sie wollen, dass das dem Steuerelement zugrunde liegende »Windows-Window« neu erstellt werden soll, verwenden Sie die Methode UpdateStyles, die eine erneute Abfrage von CreateParams veranlasst.

Zeichnen des Steuerelements

Das Ausformulieren des Codes zum Zeichnen des Steuerelements bereitet bei dieser Klasse eigentlich die geringsten Schwierigkeiten, da sämtliche umzusetzenden Eigenschaften mit entsprechenden Funktionen des GDI+ realisiert werden können, wie das folgende Codelisting zeigt. Etwas aufwändig ist das Finden der Parameter für das StringFormat-Objekt, das bei DrawString die Textausrichtung steuert – letzten Endes ist das aber nur eine reine Fleißarbeit.

Das eigentliche Zeichnen des Textes geschieht, wie bei allen Steuerelementen und Formularen, in der überschriebenen OnPaint-Methode. Sie wird entweder dann aufgerufen, wenn ein anderes *Window*-Objekt das Steuerelement verdeckt hatte und es anschließend wieder sichtbar wurde, oder wenn eine Instanz von außen ein Neuzeichnen des Inhalts erforderlich machte – beispielsweise, wenn sich eine Eigenschaft geändert hat, die die grafische Gestaltung des Steuerelementinhalts in irgendeiner Form beeinflusst.

```
*****  
*** Alles für das Zeichnen ***  
*****  
  
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)  
    'DrawString arbeitet mit RectangleF, ClientRectangle mit Rectangle;  
    'deswegen die Werte ins andere "Format" konvertieren.
```

```

Dim locRectf As New RectangleF(0, 0, ClientSize.Width, ClientSize.Height)
'StringFormat-Objekt für die Ausgabe des Strings erzeugen
Dim locSf As StringFormat = CreateStringFormat()

'Diese "Version" malen, wenn nicht geblinkt wird, oder gerade die Aus-Phase stattfindet.
If Not Flash Or Not myFlashState Then
    'Bei der Ausgabe des Strings "CR" anhängen, damit wird bei rechtsbündiger und
    'zentrierter Formatierung die richtige Stringlänge berücksichtigt.
    e.Graphics.DrawString(Text + vbCr, Font, New SolidBrush(ForeColor), locRectf, locSf)
Else
    'Sonst die An-Phase zeichnen, mit FlashBackColor und FlashForeColor.
    e.Graphics.Clear(FlashBackColor)
    e.Graphics.DrawString(Text + vbCr, Font, New SolidBrush(FlashForeColor), locRectf, locSf)
End If
'Keinen Speicher verschwenden!
locSf.Dispose()
End Sub

'Bastelt aus der Einstellung für ContentAlignment das StringFormat-Objekt zusammen,
'über das diese Eigenschaft bei der Ausgabe mit DrawString umgesetzt wird.
Protected Overrides Function StringFormatForAlignment(ByVal textAlign As ContentAlignment) _
As StringFormat

    Dim locStringFormat As New StringFormat
    If (textAlign And ContentAlignment.BottomLeft) = ContentAlignment.BottomLeft Or _
        (textAlign And ContentAlignment.MiddleLeft) = ContentAlignment.MiddleLeft Or _
        (textAlign And ContentAlignment.TopLeft) = ContentAlignment.TopLeft Then
        locStringFormat.Alignment = StringAlignment.Near
    ElseIf (textAlign And ContentAlignment.BottomRight) = ContentAlignment.BottomRight Or _
        (textAlign And ContentAlignment.MiddleRight) = ContentAlignment.MiddleRight Or _
        (textAlign And ContentAlignment.TopRight) = ContentAlignment.TopRight Then
        locStringFormat.Alignment = StringAlignment.Far
    ElseIf (textAlign And ContentAlignment.BottomCenter) = ContentAlignment.BottomCenter Or _
        (textAlign And ContentAlignment.MiddleCenter) = ContentAlignment.MiddleCenter Or _
        (textAlign And ContentAlignment.TopCenter) = ContentAlignment.TopCenter Then
        locStringFormat.Alignment = StringAlignment.Center
    End If

    If (textAlign And ContentAlignment.TopLeft) = ContentAlignment.TopLeft Or
        (textAlign And ContentAlignment.TopRight) = ContentAlignment.TopRight Or
        (textAlign And ContentAlignment.TopCenter) = ContentAlignment.TopCenter Then
        locStringFormat.LineAlignment = StringAlignment.Near
    ElseIf (textAlign And ContentAlignment.MiddleLeft) = ContentAlignment.MiddleLeft Or
        (textAlign And ContentAlignment.MiddleRight) = ContentAlignment.MiddleRight Or
        (textAlign And ContentAlignment.MiddleCenter) = ContentAlignment.MiddleCenter Then
        locStringFormat.LineAlignment = StringAlignment.Center
    ElseIf (textAlign And ContentAlignment.BottomLeft) = ContentAlignment.BottomLeft Or
        (textAlign And ContentAlignment.BottomCenter) = ContentAlignment.BottomCenter Or
        (textAlign And ContentAlignment.BottomRight) = ContentAlignment.BottomRight Then
        locStringFormat.LineAlignment = StringAlignment.Far
    End If
    Return locStringFormat
End Function

'Baut das StringFormat-Objekt zusammen und berücksichtigt nicht nur ContentAlignment,

```

```
'sondern auch andere Eigenschaften des AdLabelEx-Steuerelements.  
Protected Overrides Function CreateStringFormat() As StringFormat  
  
    Dim locStringFormat As StringFormat  
    'Grundsätzliche Einstellungen aufgrund des ContentAlignment holen.  
    locStringFormat = StringFormatForAlignment(Me.TextAlign)  
    'RightToLeft-Einstellung für Arabische Sprachen berücksichtigen  
    If Me.RightToLeft = RightToLeft.Yes Then  
        locStringFormat.FormatFlags = locStringFormat.FormatFlags Or _  
            StringFormatFlags.DirectionRightToLeft  
    End If  
  
    'Zugriffstastenanzeige berücksichtigen.  
    If Not Me.UseMnemonic Then  
        locStringFormat.HotkeyPrefix = System.Drawing.Text.HotkeyPrefix.None  
    Else  
        If (Me.ShowKeyboardCues) Then  
            locStringFormat.HotkeyPrefix = System.Drawing.Text.HotkeyPrefix.Show  
        Else  
            locStringFormat.HotkeyPrefix = System.Drawing.Text.HotkeyPrefix.Hide  
        End If  
    End If  
  
    'If Me.AutoSize Then  
    '    locStringFormat.FormatFlags = locStringFormat.FormatFlags Or _  
    '        StringFormatFlags.MeasureTrailingSpaces  
    'End If  
  
    'Möglichst genaue Formatierung.  
    locStringFormat.FormatFlags = locStringFormat.FormatFlags Or StringFormatFlags.FitBlackBox  
    'LineLimit wird nicht berücksichtigt, wenn der Text nicht in den Rahmen passt  
    locStringFormat.FormatFlags = locStringFormat.FormatFlags And Not StringFormatFlags.LineLimit  
  
    'Text um 90 Grad im Uhrzeigersinn drehen?  
    If DirectionVertical Then  
        locStringFormat.FormatFlags = locStringFormat.FormatFlags Or  
        StringFormatFlags.DirectionVertical  
    End If  
  
    'Textwrapping eingeschaltet?  
    If Not TextWrap Then  
        locStringFormat.FormatFlags = locStringFormat.FormatFlags Or StringFormatFlags.NoWrap  
    End If  
  
    'Das Trimming definieren.  
    locStringFormat.Trimming = TextTrimming  
  
    'Wert zurückgeben.  
    Return locStringFormat  
End Function
```

Die fett hervorgehobenen Codezeilen im oben gezeigten Listing zeichnen übrigens den Text in den Clientbereich des Steuerelements. Dabei muss natürlich auch ein möglicherweise eingeschaltetes Blinken (Flash = True) berücksichtigt werden. Wie das Blinken des Steuerelements implementiert ist, erfahren Sie in einem späteren Abschnitt.

TIPP Das Verändern bestimmter Eigenschaften führt oft dazu, dass sich das Aussehen eines Steuerelements ändert und es daher neu gezeichnet werden soll. Wenn Sie wollen, dass sich ein Steuerelement neu zeichnet, stehen Ihnen dazu prinzipiell drei verschiedene Vorgehensweisen zur Verfügung, nämlich mit Refresh, Invalidate und Update. Der folgende Abschnitt macht die Unterschiede in der Anwendung deutlich:

Der Unterschied zwischen Refresh, Invalidate und Update

Aus vielen Diskussionen im Usenet wird klar, dass es Unklarheiten über die richtige Anwendung und Funktionsweise dieser drei Funktionen gibt, da sie alle drei Ähnliches bewirken. Fakt ist, dass sie tatsächlich viel miteinander zu tun haben, sich teilweise auch gegenseitig aufrufen. Invalidate verwenden Sie, wenn Sie einen bestimmten oder den ganzen Bereich des Clientbereiches eines Fensters oder Steuerelements neu zeichnen lassen wollen. Invalidate löst dann das Aufrufen der OnPaint-Methode aus, die für das Neuzeichnen des Clientbereichs Sorge tragen muss. Allerdings macht sie das nicht sofort, sondern erst, wenn die Nachrichtenwarteschlange die nächste Gelegenheit bekommt, ausstehende Nachrichten zu verarbeiten. Das ist aber erst dann der Fall, wenn Ihre Anwendung in den so genannten *Idle*⁶-Modus wechselt. In einigen Fällen kann es aber notwendig sein, dass das Steuerelement (oder Formular) seinen Clientbereich sofort neu zeichnen soll. In diesem Fall ist ein anschließendes Update notwendig, das die OnPaint-Methode und das dazugehörige Ereignis auslöst.

Refresh wiederum fasst beide Methoden unter einem Dach zusammen und macht noch ein bisschen mehr: Es bewirkt mit einem Invalidate(True), dass auch alle untergeordneten Steuerelemente neu gezeichnet werden und erzwingt mit einem anschließenden Update darüber hinaus das unmittelbare Neuzeichnen.

Größenbeeinflussung durch andere Eigenschaften

Bestimmte Steuerelemente können nicht nur durch die Size-Eigenschaft, sondern auch durch andere Eigenschaften größenmäßig beeinflusst werden. Sie haben das bestimmt schon selbst bei der TextBox erlebt: Wenn Sie eine TextBox in einem Formular platzieren, so kann sie in ihrer Höhe nicht verändert werden – standardmäßig ist sie nämlich für den einzeiligen Betrieb vorgesehen. Die TextBox merkt sich die Höhe, in der sie hätte platziert werden sollen, aber sehr wohl: Sobald Sie nämlich ihre MultiLine-Eigenschaft auf True setzen, »springt« sie förmlich auseinander, und zwar nimmt sie dann genau die Größe an, die sie schon beim ersten Aufziehen im Formular hätte haben sollen.

Wenn die Größe eines Steuerelements oder Formulars verändert werden soll, ist die Methode SetBoundsCore der Control-Basisklasse für diese Aufgabe zuständig. Sie sorgt dafür, dass die entsprechenden Windows-Nachrichten an das Betriebssystem gesendet werden, um das Steuerelement auf die richtige Größe zu bringen. Um diesen Vorgang zu reglementieren (also beispielsweise zu verhindern, dass eine bestimmte Höhe überschritten wird), kann eine vererbte Klasse diese Methode überschreiben und veränderte Parameter für die Steuerelementausmaße an die Basismethode übergeben.

⁶ Auf Deutsch etwa »brachliegend«, »unbeschäftigt«, »untätig« (interessanterweise auch »faul«, »nutzlos«, »müsig«). Kleiner Tipp: Wenn Sie per Ereignis benachrichtigt werden wollen, dass Ihre Applikation in den *Idle*-Modus gewechselt ist, binden Sie via AddHandler das Application.Idle-Ereignis ein. Hinweis: Sie können auch per Anweisung dafür sorgen, dass die Anwendung die Nachrichtenwarteschlange abarbeitet, sie also bewusst für einen kurzen Moment in den *Idle*-Modus bringen: Der unter den VB6 Entwicklern so beliebte Befehl DoEvents steht auch in .NET als Methode des Application-Objekts zur Verfügung.

Um zu gewährleisten, dass ein ursprünglich zugewiesener Wert für die Höhe oder die Breite erhalten bleibt (die TextBox beispielsweise beim Setzen von Multiline auf die ursprüngliche Höhe wieder zurückspringt), implementiert jedes Steuerelement, das dieses Verhalten an den Tag legt, eine oder zwei Variablen,⁷ die die Ausgangsausmaße für die jeweilige Dimension zwischenspeichern.

Diese Zwischenspeicher dürfen aber nur dann neu gesetzt werden, wenn eine Eigenschaft die entsprechende Dimension gezielt überschrieben hat. Wurde beispielsweise nur die Height-Eigenschaft des Steuerelements verändert, darf sich das nicht auf den zwischengespeicherten Wert für die Breite (requestedWidth) auswirken. Aus diesem Grund wird SetBoundsCore neben Parametern, die die kompletten neuen (oder eben alten) Ausmaße des Steuerelements enthalten, ein weiterer Parameter namens specified übergeben, der bestimmt, welcher Ausmaßparameter (X, Y, Breite, Höhe) gezielt verändert wurde.

Unser Beispielsteuerelement implementiert dieses Verhalten übrigens auch. Wenn die AutoHeight-Eigenschaft des Steuerelements auf True gesetzt wird, passt es sich höhenmäßig an die Ausmaße des Textes an, vergrößert bzw. verkleinert sich automatisch. Das funktioniert auch dann, wenn Sie anschließend die Breite des Steuerelements verschieben. Setzen Sie die AutoHeight-Eigenschaft anschließend wieder zurück, nimmt das Steuerelement seine ursprüngliche Größe an.

Sobald die AutoHeight-Eigenschaft gesetzt wurde, lässt sich die Höhe des Steuerelements nicht mehr verändern. Für dieses Verhalten ist aber nicht die eigentliche Steuerelementklasse verantwortlich, sondern eine weitere, die Sie im Projektmappen-Explorer unter ADLabelExDesigner finden. Im Abschnitt »Designer-Reglementierungen« ab Seite 1207 finden Sie mehr zu diesem Thema.

Das folgende Codelisting zeigt, wie die Größensteuerung des Steuerelements implementiert wurde.

```
'Die neue Höhe einstellen. Diese Methode wird aufgerufen, wenn sich eine Eigenschaft  
'geändert hat, die die Höhe des Steuerelements beeinflusst, und wenn AutoHeight eingeschaltet ist.  
Private Sub AdjustHeight()
```

```
    Dim locRequestedHeightTemp As Integer  
    locRequestedHeightTemp = myRequestedHeight  
    Try  
        If AutoHeight Then  
            MyBase.Size = New Size(Me.Size.Width, PreferredHeight)  
        Else  
            MyBase.Size = New Size(Me.Size.Width, locRequestedHeightTemp)  
        End If  
    Finally  
        myRequestedHeight = locRequestedHeightTemp  
    End Try  
End Sub
```

```
'Ermittelt die Höhe des Textes bei einer bestimmten Breite. Diese Funktion  
'wird von AdjustHeight für die automatische Höhenanpassung des Steuerelements verwendet.  
Public Overridable ReadOnly Property PreferredHeight() As Integer
```

```
    Get  
        Dim locHeightToReturn As Integer  
        If Me.Text = "" Then  
            locHeightToReturn = Me.FontHeight  
        Else
```

⁷ Es hat sich eingebürgert, diese Variablen *requestedHeight* und *requestedWidth* zu nennen.

```

Dim locG As Graphics
Dim locSf As StringFormat
Dim locSizeF As SizeF
locG = Graphics.FromHwnd(Me.Handle) ' Graphics-Objekt erzeugen.
locSf = CreateStringFormat()           ' Gleiches StringFormat wie beim Ausgeben.
'Texthöhe automatisch ermittelt. Das erreichen Sie, wenn Sie für die Höhe 0 übergeben.
locSizeF = locG.MeasureString(Text, Font, New SizeF(ClientSize.Width, 0), locSf)
'Immer nach unten abrunden!
locHeightToReturn = CInt(Math.Ceiling(locSizeF.Height))
End If
'Falls es einen Borderstyle gibt, 2 Pixel draufrechnen, damit es nicht
'zu gequetscht wird.
If BorderStyle <> BorderStyle.None Then
    locHeightToReturn += 2
End If
Return locHeightToReturn
End Get
End Property

'Setzt alle Ausmaße des Steuerelements oder nur bestimmte Größenkomponenten,
'die von Specified bestimmt werden.
Protected Overrides Sub SetBoundsCore(ByVal x As Integer, ByVal y As Integer, ByVal width As
Integer, _
        ByVal height As Integer, ByVal specified As
System.Windows.Forms.BoundsSpecified)

Dim locRect As New Rectangle

'Falls AutoHeight eingeschaltet ist...
If AutoHeight Then
    '...und die Breite bestimmt werden soll...
    If (specified And BoundsSpecified.Width) = BoundsSpecified.Width Then
        '...dann die neue Breite im Steuerelement setzen...
        MyBase.SetBoundsCore(x, y, width, height, specified)
        '...jetzt muss aber auch die Höhe neu errechnet werden...
        AdjustHeight()
        '...und wenn die zwischengespeicherte Höhe gesetzt war...
        If myRequestedHeight > 0 Then
            'dann bricht der Vorgang hier ab. Andernfalls wurde myRequestedHeight nicht
            'initialisiert, und zwar dadurch, dass Height noch 0 war, als das
            'Steuerelement erstellt wurde. Erst die erste Zuweisung der Size-Eigenschaft
            'bestimmt die Höhe, die aber selbst mit SetBoundsCore gesetzt wird. Aus diesem
            'Grund kann myRequestedHeight beim ersten Durchlauf keinen anderen Wert als
            '0 haben und muss entsprechend initialisiert werden.
            Return
        End If
    End If
End If

'Aktuelle Ausmaße zwischenspeichern.
locRect = Me.Bounds
If (specified And BoundsSpecified.Height) = BoundsSpecified.Height Then
    'myRequestedHeight wird neu definiert, wenn die Höhe (zum Beispiel durch Size)
    'explizit zugewiesen wird. Am vom SetBoundsCore "verlangten" Height
    'ändert sich nur dann was...
    myRequestedHeight = height
End If

```

```
'...wenn AutoHeight eingeschaltet ist. Dann wird die Höhe des Steuerelements auf die  
'gemessene Höhe des Textes festgeschrieben.  
If (Me.AutoHeight AndAlso (locRect.Height <> height)) Then  
    height = Me.PreferredHeight  
End If  
  
'Basis aufrufen  
MyBase.SetBoundsCore(x, y, width, height, specified)  
End Sub
```

Implementierung der Blink-Funktionalität

Wie in der Einführung schon erwähnt, kam es bei der Implementierung der Blink-Funktionalität darauf an, möglichst wenige Ressourcen zu verwenden und das Blinken der einzelnen ADLabelEx-Instanzen untereinander zu synchronisieren. Aus diesem Grund gibt es nur einen einzigen Timer, der für das Triggern des Blinkens zuständig ist. Und da es nur einen Timer gibt, ist es fast schon selbstverständlich, dass er als statische Instanz im Steuerelement implementiert wurde.

Der Timer selbst wird erst dann instanziert, wenn das erste ADLabelEx seine Flash-Eigenschaft auf True setzt und damit beginnen will zu blinken. Eine statische Zählvariable sorgt dafür, dass der Timer weiß, wie viele Instanzen von ihm Gebrauch machen. Wenn die Zählvariable wieder den Wert 0 erreicht hat, wird die Dispose-Methode des Timers ausgeführt, um ihn ordnungsgemäß und rückstandsfrei zu entsorgen.

Damit jede ADLabelEx-Instanz über das Eintreten des Tick-Ereignisses im Bedarfsfall informiert wird, meldet sie ihre Ereignisbehandlungsroutine FlashOnHandler und FlashOffHandler schon in ihrem Konstruktor mit AddHandler an. Diese Ereignisse werden durch die statische Ereignisbehandlungsroutine FlashTimeHandler für alle ADLabelEx-Instanzen ausgelöst. Damit dieser zentrale Ereignisverteiler selbst vom Ablauf des Timers durch das Eintreten seines Tick-Ereignisses erfährt, sorgt die Routine StartFlashHandlerOnDemand für das Zuweisen der Ereignisbehandlungsroutine ebenfalls durch eine AddHandler-Anweisung. Der typische Weg der Ereigniskette ist also folgender:

- Eine ADLabelEx-Instanz wird instanziert, und sie bindet die Ereignisbehandlungsroutine mit AddHandler schon im Konstruktor ein.
- Ihre Flash-Eigenschaft wird auf True gesetzt, und das bewirkt den Aufruf von StartFlashHandlerOnDemand, das das Timer-Objekt instanziert, startet und die zentrale Ereignisbehandlungsroutine FlashTimeHandler einrichtet.
- Der Timer läuft irgendwann ab und löst sein Tick-Ereignis aus. Das wiederum löst den Aufruf von FlashTimeHandler aus, und diese Routine löst, in Abhängigkeit vom Phasen-Flag myFlashState, entweder das FlashOn- oder das FlashOff-Ereignis aus.
- Da jede ADLabelEx-Instanz diese Ereignisse im Konstruktor registriert hat, werden die Ereignisbehandlungsroutine FlashOnHandler und FlashOffHandler jeder Instanz aufgerufen.
- Wenn die Flash-Eigenschaft einer Instanz nicht gesetzt ist, erfolgt ein sofortiger Rücksprung aus der Behandlungsroutine – es passiert gar nichts. Andernfalls lösen beide Routinen mit Invalidate ein Neuzeichnen des Steuerelementeinhals mit der dem jeweiligen Phasenzustand entsprechenden Farbeinstellung aus. In der Aus-Phase werden FlashBackColor und FlashForeColor für das Zeichnen verwendet, in der An-Phase BackColor und ForeColor. Durch geschicktes Wählen der Farben entsteht der Eindruck eines rhythmischen Blinkens.

- Wenn eine Steuerelementinstanz entsorgt wird und ihre Flash-Eigenschaft zuvor verwendet wurde, meldet sie sich in ihrer Dispose-Methode durch Aufruf der Prozedur StopFlashHandlerOnDemand wieder ab. Der *Instanzzähler* wird heruntergezählt, und der Timer wird erst dann ordnungsgemäß entsorgt, wenn die letzte ADLabelEx-Instanz ihn nicht mehr benötigt.

Der Code, der diese Funktionalität implementiert, sieht folgendermaßen aus:

```

'*****
'* Flash-Handling
'*****
'*****

Private Shared myFlashState As Boolean
Private Shared myFlashTimerUseCounter As Integer
Private myFlashTimerUsed As Boolean
Private myFlash As Boolean
Private myFlashBackColor As Color
Private myFlashForeColor As Color
Public Shared Event FlashOn(ByVal sender As Object, ByVal e As EventArgs)
Public Shared Event FlashOff(ByVal sender As Object, ByVal e As EventArgs)

'Es gibt einen einzigen Timer für alle blinkenden ADLabelEx-Instanzen. Alles andere
'wäre Verschwendung von Ressourcen.
'Und auch der eine Timer wird erst dann angeworfen, wenn das erste ADLabelEX
'blinken will.
Private Shared Sub StartFlashHandlerOnDemand()
    If myFlashTimerUseCounter = 0 Then
        myFlashTimer = New Timer
        myFlashTimer.Interval = myFlashInterval
        myFlashTimer.Start()
        'Hier wird die Ereignisbehandlungsroutine eingebunden, die beim
        'Ablauen von myFlashInterval-Millisekunden (also alle 300) aufgerufen wird.
        AddHandler myFlashTimer.Tick, AddressOf FlashTimeHandler
    End If
    'Damit die Steuerelement-Klasse weiß, wie viele Instanzen blinken,
    'gibt es einen Zähler...
    myFlashTimerUseCounter += 1
End Sub

Private Shared Sub StopFlashHandlerOnDemand()
    myFlashTimerUseCounter -= 1
    '...damit das Timer-Objekt ordnungsgemäß entladen werden kann,
    'wenn es nicht mehr benötigt wird.
    If myFlashTimerUseCounter = 0 Then
        myFlashTimer.Stop()
        myFlashTimer.Dispose()
    End If
End Sub

'Dispose wird benötigt, damit der letzte das Licht (den Timer) ausmachen kann!
Protected Overrides Sub Dispose(ByVal disposing As Boolean)
    If disposing Then
        If myFlashTimerUsed Then
            RemoveHandler FlashOn, AddressOf FlashOnHandler
            RemoveHandler FlashOff, AddressOf FlashOffHandler
            StopFlashHandlerOnDemand()
        End If
    End If
End Sub

```

```
End If
 MyBase.Dispose(disposing)
End Sub
'Dieser private Ereignishandler löst zwei neue Ereignisse aus, die öffentlich empfangen werden
'können. Es gibt jeweils für beginnende An- und Aus-Phase ein Ereignis.
Private Shared Sub FlashTimeHandler(ByVal sender As Object, ByVal e As EventArgs)
    myFlashState = Not myFlashState
    If myFlashState Then
        RaiseEvent FlashOn("ADLabelEx.FlashHandler", e.Empty)
    Else
        RaiseEvent FlashOff("ADLabelEx.FlashHandler", e.Empty)
    End If
End Sub

Protected Overridable Sub FlashOnHandler(ByVal sender As Object, ByVal e As EventArgs)
    'Da die Ereignis-Handler schon im Konstruktor eingebunden werden (war einfacher ;-)
    'treten die Ereignisse auch auf, wenn ein anderes ADLabelEx blinken will.
    'Deswegen muss diese Instanz testen, ob sie blinken darf.
    If Not myFlash Then Return
    'Alles weitere regelt OnPaint...
    Me.Invalidate()
End Sub

'Dasselbe in blau/schwarz.
Protected Overridable Sub FlashOffHandler(ByVal sender As Object, ByVal e As EventArgs)
    If Not myFlash Then Return
    Me.Invalidate()
End Sub
```

Designercode-Generierung und Zurücksetzen von werteerbenden Eigenschaften mit ShouldSerializeXXX und ResetXXX

Besondere Beachtung verdienen die beiden Eigenschaften `FlashBackColor` und `FlashForeColor`, da es sich bei ihnen um so genannte »werteerbende Eigenschaften« handelt. Solange Sie ihnen keine speziellen Werte zuweisen, liefern sie den Wert der `BackColor`-Eigenschaft zurück. Ändern Sie den Wert der `BackColor`-Eigenschaft, ändern sich auch die Werte von `FlashBackColor` und `FlashForeColor`. Der fett markierte Bereich zeigt, wie diese Implementierung vonstatten geht.

Nun gibt es keinen fest definierten Standardwert für diese beiden Eigenschaften, weil sie vom Wert einer anderen abhängig sind. Damit diese beiden Eigenschaften nicht unnötigerweise serialisiert werden (d.h., entsprechender Initialisierungscode in `InitializeComponent` für die Instanz erzeugt wird, die das `ADLabelEx` einbindet), versagt das schon bekannte `DefaultValue`-Attribut an dieser Stelle. In solchen Fällen implementieren Sie eine Funktion, die genauso lautet wie die eigentliche Eigenschaft und zusätzlich das Präfix `ShouldSerialize` trägt. Durch Reflection ermittelt der Designer vor der Codegenerierung das Vorhandensein einer solchen Funktion und ruft sie auf, um herauszufinden, ob die Serialisierung der Eigenschaft notwendig ist. Die Funktion bestimmt also anhand eines bestimmten Algorithmus und nicht auf Grund eines konstanten Wertes, ob die Code-Serialisierung notwendig ist.

Damit die `Reset`-Funktion des Eigenschaftenfensters für eine derartige Eigenschaft funktionieren kann, implementieren Sie eine weitere Funktion, die den Namen der Eigenschaft und das Präfix `Reset` trägt. Diese

Funktion stellt dann im Bedarfsfall (der Anwender wählt aus dem Kontextmenü der Eigenschaft im Eigenschaftenfenster *Zurücksetzen*) den gültigen Ausgangswert wieder her.

```
<DefaultValue(GetType(Boolean), "False"), _
Category("Darstellung"),
Description("Bestimmt, ob der Label-Text blinkend angezeigt werden soll."), _
Browsable(True)>
Public Property Flash() As Boolean
    Get
        Return myFlash
    End Get
    Set(ByVal Value As Boolean)
        myFlash = Value
        'Im Entwurfsmodus wird nicht geblinkt!
        If Not DesignMode Then
            If Value Then
                'Erst das erste Setzen initialisiert den Blink-Timer,
                'aber nur beim ersten Mal!
                If Not myFlashTimerUsed Then
                    StartFlashHandlerOnDemand()
                End If
                myFlashTimerUsed = True
            Else
                'Alten Zustand wiederherstellen
                Invalidate()
            End If
        End If
    End Set
End Property
<Category("Darstellung"), _
Description("Bestimmt die Hintergrundfarbe beim Blinken, wenn sich das Steuerelement in der An-Phase befindet."), _
Browsable(True)>
Public Property FlashBackColor() As Color
    Get
        'Hier läuft's anders mit den Standardwerten. Wenn keine Farbe definiert ist,
        '"erbt" diese Eigenschaft von BackColor. Dadurch muss nur BackColor verändert
        'werden, um auch FlashBackColor zu verändern. Allerdings gibt es damit keinen
        'festen Standardwert...
        If myFlashBackColor.Equals(Color.Empty) Then
            Return BackColor
        Else
            Return myFlashBackColor
        End If
    End Get
    Set(ByVal Value As Color)
        If Value.Equals(BackColor) Then
            myFlashBackColor = Color.Empty
        Else
            myFlashBackColor = Value
        End If
    End Set
End Property
```

```
'...deswegen muss mit einer Funktion ermittelt werden, ob der aktuelle Wert der Standardwert ist.  
'Nur wenn er es nicht ist, wird serialisiert (Code für die Eigenschaft in der sie einbindenden  
'Instanz erzeugt).  
Public Function ShouldSerializeFlashBackColor() As Boolean  
    Return Not myFlashBackColor.Equals(Color.Empty)  
End Function  
  
'Damit wird die Reset-Funktion für diese Eigenschaft im Eigenschaftenfenster  
'(Kontext-Menü über der Eigenschaft) aktiviert.  
Public Sub ResetFlashBackColor()  
    myFlashBackColor = Color.Empty  
End Sub  
  
<Category("Darstellung"), _  
Description("Bestimmt die Vordergrundfarbe beim Blinken, wenn sich das Steuerelement in der An-Phase  
befindet."),  
Browsable(True)>  
Public Property FlashForeColor() As Color  
    Get  
        If myFlashForeColor.Equals(Color.Empty) Then  
            Return BackColor  
        Else  
            Return myFlashForeColor  
        End If  
    End Get  
  
    Set(ByVal Value As Color)  
        If Value.Equals(BackColor) Then  
            myFlashForeColor = Color.Empty  
        Else  
            myFlashForeColor = Value  
        End If  
    End Set  
End Property  
  
Public Function ShouldSerializeFlashForeColor() As Boolean  
    Return Not myFlashForeColor.Equals(Color.Empty)  
End Function  
  
Public Sub ResetFlashForeColor()  
    myFlashForeColor = Color.Empty  
End Sub  
End Class
```

Designer-Reglementierungen

Wie Sie vielleicht wissen, verfügt jedes Steuerelement über seinen eigenen Designer. Visual Studio .NET hat also keine Funktionalität zum interaktiven Verändern der einzelnen Steuerelemente implementiert, sondern fungiert nur als so genannter *Designer Host*. Für das Zur-Verfügung-Stellen eines Designers, mit dem der Anwender zum Entwurfsmodus das Steuerelement beispielsweise in seinen Ausmaßen verändern kann, ist also jedes Steuerelement selbst verantwortlich.

Natürlich brauchen Sie nicht für jedes Steuerelement, das Sie neu erstellen, eine vollständige Designer-Logik von Grund auf zu entwerfen. Im günstigsten Falle müssen Sie überhaupt nichts machen. Wenn Sie keinen speziellen Designer für Ihr Steuerelement definiert haben, verwendet jeder Designer-Host automatisch den Standard-Control-Designer, mit dem Ihr Steuerelement zur Entwurfszeit bearbeitet werden kann. Soll Ihr Steuerelement jedoch zur Entwurfszeit design-technisch in irgendeiner Form von der Norm abweichen, müssen Sie selbst Hand anlegen.

In unserem Fall ist das notwendig, da das Steuerelement nicht in der Höhe verändert werden darf, wenn seine `AutoHeight`-Eigenschaft gesetzt ist. Der folgende Code zeigt, wie eine solche Implementierung vonstatten geht.

HINWEIS Damit ein individueller Designer und nicht der *Standard Control Designer* vom Designer Host verwendet wird, müssen Sie Ihre Steuerelementklasse mit einem Attribut namens *Designer* ausstatten. Dieses Attribut muss über der Klassendefinition der eigentlichen Steuerelementklasse platziert werden, etwa folgendermaßen für dieses Beispiel:

```
<Designer("ActiveDevelop.Controls.ADLabelExDesigner")> _
Public Class ADLabelEx
    Inherits Control
    .
    .

```

Wenn Sie einen eigenen Designer implementieren, erben Sie ihn aus der Klasse `ControlDesigner`. Die Funktionen, die Sie verändern möchten, überschreiben Sie in dieser Klasse. Ebenfalls wichtig: Damit Sie auf die Designerfunktionalität des Frameworks zurückgreifen können, müssen Sie einen Verweis auf die Assembly `System.Design.dll` in Ihr Projekt einfügen. Das Importieren des Namespaces `System.Windows.Forms.Design` ist darüber hinaus erforderlich.

```
Imports System.Windows.Forms
Imports System.Drawing
Imports System.ComponentModel
Imports System.Windows.Forms.Design

'WICHTIG: Wenn Sie einen ControlDesigner einfügen,
'müssen Sie den System.Windows.Forms.Design-Namespace einbinden,
'und System.Design.dll als Verweis dem Projekt hinzufügen!
Public Class ADLabelExDesigner
    Inherits ControlDesigner

    'Muss überschrieben werden, damit bei einem Steuerelement mit fixer Größe in
    'einer vertikalen Richtung tatsächlich nur eine vertikale Größenänderung möglich wird.
    'Die vertikalen Anfasspunkte sind dann ausgeblendet
    Public Overrides ReadOnly Property SelectionRules() As System.Windows.Forms.Design.SelectionRules
        Get
            Dim locThisComponent As Object
            Dim locSelectionRules As SelectionRules
            locThisComponent = Me.Component
            Try
                'In Abhängigkeit von ConsiderFixedSize (die sich beispielsweise durch Multiline ändert)
                If _

```

```
Convert.ToBoolean(TypeDescriptor.GetProperties(locThisComponent).Item("AutoHeight").GetValue(locThisComponent)) _  
    Then  
        'Nur vertikale Größenveränderungen...  
        locSelectionRules = SelectionRules.Moveable Or SelectionRules.Visible Or _  
                            SelectionRules.LeftSizeable Or SelectionRules.RightSizeable  
    Else  
        '...oder komplette Größenveränderungen ermöglichen  
        locSelectionRules = SelectionRules.Moveable Or SelectionRules.Visible Or _  
                            SelectionRules.AllSizeable  
    End If  
    Return locSelectionRules  
Catch ex As Exception  
    Debug.WriteLine("Designermessage:" & ex.Message)  
    Return MyBase.SelectionRules  
End Try  
End Get  
End Property  
End Class
```


Teil H

Entwickeln von WPF-Anwendungen

In diesem Teil:

Steuerelemente	1213
Layout	1253
Grafische Grundelemente	1283

Kapitel 41

Wichtige Steuerelemente zum Aufbau von WPF-Anwendungen

In diesem Kapitel:

Einführung	1214
Weitergeleitete Ereignisse (Routed Events)	1215
Weitergeleitete Befehle (Routed Commands)	1224
Eigenschaften der Abhängigkeiten	1228
Eingaben	1231
Schaltflächen	1233
Bildlaufleisten und Schieberegler	1235
Steuerelemente für die Texteingabe	1239
Das Label-Element	1243
Menüs	1244
Werkzeugleisten (Toolbars)	1249
Zusammenfassung	1252

Mit Windows Presentation Foundation wollen wir die Benutzerschnittstellen unserer Applikationen deklarieren. Das funktioniert natürlich nicht ohne Steuerelemente wie Schaltflächen, Eingabefelder, Bildlaufleisten, usw. Diese Elemente stehen Ihnen auch in WPF zur Verfügung. Jedes WPF-Element enthält im .NET Framework 3.0 eine entsprechende Klasse, die das Element mit allen Eigenschaften, Methoden und Ereignissen abbildet.

In diesem Buch beschäftigen sich zwei Kapitel mit den Steuerelementen von WPF. In diesem Kapitel lernen Sie die Standardsteuerelemente, wie z. B. TextBox- oder Button-Elemente, kennen. In einem zweiten Kapitel werden die Elemente behandelt, die sich mit dem Anordnen von Steuerelementen (Layout) beschäftigen.

BEGLEITDATEIEN

Die Begleitdateien zu folgenden Beispielen finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\H – WPF\\

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Einführung

Steuerelemente sind kleine Einheiten der Benutzerschnittstelle einer Applikation. Normalerweise bestehen sie selbst wieder aus einer meist einfachen Benutzerschnittstelle (Aussehen des Steuerelements) und einer implementierten Logik (Verhalten). Häufig bilden die Steuerelemente eine Abstraktionsschicht in einem Framework für Benutzerschnittstellen. Dies ist in WPF allerdings nicht so, da die Steuerelemente hier nicht ihr Aussehen »implementieren«. Das Aussehen der WPF-Steuerelemente wird über Templates (Vorlagen) realisiert. Diese Templates oder Vorlagen können jederzeit ausgetauscht werden. Dadurch wird zwar das Aussehen eines Steuerelements modifiziert, sein Verhalten bleibt jedoch unverändert. Dieses Verhalten wird in Abbildung 41.1 dargestellt.

Das Steuerelement besteht weiterhin aus der Logik, die sein Verhalten festlegt. Hierzu gehören die Methoden, die Eigenschaften und Ereignisse, die ausgelöst werden können. Ein Anwender interagiert mit dem Steuerelement über diese Logik. Die Darstellung des Steuerelements ist in Abbildung 41.1 allerdings losgelöst von der Logik als Template (Vorlage) implementiert. Über diese Vorlage wird dem Anwender das Steuerelement mit seinem Inhalt dargestellt. Auch bei den WPF-Steuerelementen ist also Logik und Design getrennt. Eine solche Trennung ist bei Steuerelementen meistens nicht vollständig zu erreichen. Trotzdem macht diese Aufteilung Sinn, wie wir in diesem Kapitel sehen werden.

Der Vorteil dieser Vorgehensweise in WPF liegt eigentlich auf der Hand. Wenn Sie aus einer normalen, rechteckigen Schaltfläche mit grauem Hintergrund eine elliptische Schaltfläche mit einem Bild und einem Farbgradienten machen wollen, so ist das möglich, ohne ein neues Steuerelement zu programmieren. Das Verhalten (Logik) der Schaltfläche soll erhalten bleiben. Was geändert werden muss, ist nur das Aussehen, welches über eine Vorlage abgebildet wird und separat geändert werden kann.

Alle WPF-Steuerelemente können Ereignisse auslösen, auf die ein Softwareentwickler in Ereignismethoden reagieren kann. In dieser Hinsicht unterscheidet sich WPF nicht von anderen Frameworks für Benutzerschnittstellen. Steuerelemente können natürlich auch Methoden enthalten, die ganz bestimmte Arbeiten erledigen. So können Sie z.B. mit einer SelectAll-Methode alle Elemente in einem ListBox-Steuerelement auswählen.

Die Ereignisse (*Events*), Befehle (*Commands*) und Eigenschaften (*Properties*) von WPF-Steuerelementen wollen wir in den folgenden Abschnitten noch etwas genauer betrachten.

HINWEIS Beachten Sie bitte auch das im Einführungskapitel 6 zu Ereignissen Gesagte, das sich auf die spezielle Eigenart von Visual Basic-Anwendungen bezieht, Ereignisprozeduren mit dem `Handles`-Schlüsselwort mit Ereignissen bestimmter Objekte zu verknüpfen.

TIPP Einige wichtige Steuerelemente fehlen zurzeit noch, um wirklich professionelle Business-Anwendungen unter der WPF erstellen zu können; am schmerzlichsten werden dabei Kalender- und DataGridView-Steuerelemente vermisst. Abhilfe schafft hier das Vorab-Release des WPF-Toolkits (IntelliLink [G3602](#), [G3603](#)). Um viel Zeit beim Aufbauen von Formularen unter WPF zu sparen, sind auch die XAML-Powertoys unbedingt einen Blick wert. Sie finden sie unter dem IntelliLink [G3601](#).

Weitergeleitete Ereignisse (Routed Events)

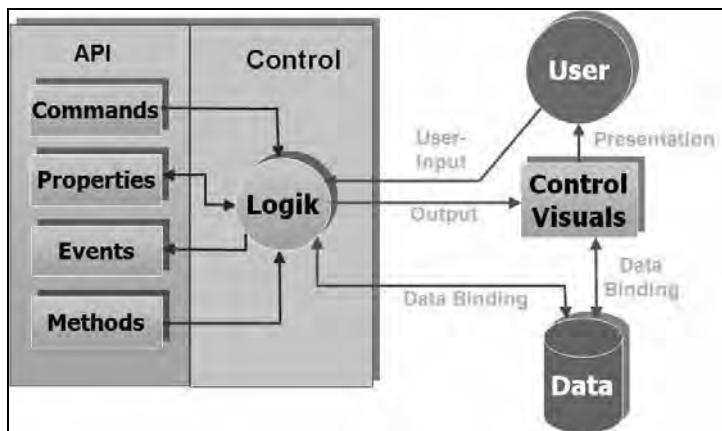


Abbildung 41.1 Aufbau der Steuerelemente in WPF

Ein sehr wichtiges Konzept für Steuerelemente unter Windows Presentation Foundation sind die *Routed Events*. Routed Events sind Ereignisse, die an verschiedene WPF-Elemente in der Hierarchie weitergeleitet werden können. Hierzu wollen wir uns einmal vorstellen, dass z.B. eine WPF-Schaltfläche aus mehreren Elementen bestehen kann: Auf der Schaltfläche kann sich ein Grid-Element mit zwei Zellen befinden. Eine Zelle enthält ein Canvas-Element mit mehreren Grafik-Objekten. Die zweite Zelle kann ein Border- und ein TextBlock-Element beinhalten. Alle Objekte in diesem Szenario (auch die grafischen Elemente) können nun z.B. ein MouseDown-Ereignis auslösen. Abbildung 41.2 zeigt ein entsprechendes Beispiel.

In WPF gibt es drei Ereignistypen:

- Direkte Ereignisse (*Direct Events* etwa: *direkte Ereignisse*)
- In der Hierarchie nach oben laufende Ereignisse (*Bubbling Events* etwa: *aufsprudelnde Ereignisse*)
- In der Hierarchie nach unten laufende Ereignisse (*Tunneling Events* etwa: *tunnelnde Ereignisse*)

Die drei Ereignistypen verhalten sich unterschiedlich und müssen jeweils an der richtigen Stelle angewendet werden.

Die Direct Events sind mit den normalen CLR-Ereignissen identisch. Ein solches Event wird nur in dem Element verarbeitet, in dem es erzeugt wurde. Denken Sie bei diesen Ereignissen z.B. an ein MouseLeave- oder MouseEnter-Ereignis. Diese Ereignisse sind eigentlich nur für das jeweilige Element interessant, welches vom Mauszeiger betreten oder verlassen wird. Andere WPF-Elemente in der logischen Hierarchie werden sich nicht für diese Ereignisse interessieren.

Bubbling Events und Tunneling Events laufen nun im Gegensatz zu den Direct Events durch die logische Hierarchie und werden somit an andere WPF-Elemente weitergegeben.

Ein Bubbling Event beginnt seine Reise durch die logische Hierarchie beim auslösenden Element und wird dann nach oben bis zum Wurzelement weitergeleitet. In welchem Element Sie das Ereignis nun bearbeiten, liegt ganz bei Ihnen. Sie können das Ereignis auch in mehreren WPF-Elementen verarbeiten. In diesem Fall implementieren Sie die Ereignismethode mehrfach unter verschiedenen Namen. Ein typisches Bubbling Event ist z.B. das MouseDown-Ereignis.

Ein Tunneling Event startet in der logischen Hierarchie oben beim Wurzelement und fällt dann nach unten bis zum auslösenden Element durch. Auch hier können Sie das Ereignis an beliebiger Stelle in der Hierarchie bearbeiten.

Wenn ein WPF-Element ein Tunneling Event und ein Bubbling Event auslöst, so wird zunächst immer das von oben beginnende Ereignis ausgelöst. Wir sprechen hier von einem *Vorschau-Ereignis* (Preview-Event). Ein ausgelöste Vorschau-Ereignis können Sie sehr gut benutzen, um das Ereignis selbst zu blockieren, sodass es nicht durch die logische Hierarchie hindurch bis zum auslösenden Element wandert. Das Vorschau-Ereignis zum MouseDown-Ereignis heißt dann PreviewMouseDown. Vorschau-Ereignisse haben also immer die Vorsilbe Preview.

Die Elementhierarchie im hier verwendeten Beispiel (Abbildung 41.2) sieht vereinfacht folgendermaßen aus:

- Window
- Grid
- Canvas
- Ellipse
- Rectangle
- Border
- TextBlock



Abbildung 41.2 Ein Fenster mit einer Elementhierarchie

In dieser Hierarchie können Sie sehen, dass die beiden grafischen Elemente Rectangle und Ellipse auf einer Ebene liegen. Trotzdem liegen sie in der Darstellung übereinander, da das Rechteck im XAML-Code erst nach der Ellipse deklariert wurde. Wir wollen nun sehen, welche Ereignisse in dieser Hierarchie ausgelöst werden. Dazu implementieren wir im Beispiel die diversen Ereignismethoden für das Bubbling und das Tunneling MouseDown-Ereignis.

```
<Window x:Class="Hierarchie.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hierarchie"
    Height="170" Width="250"
    MouseDown="OnWindow" PreviewMouseDown="OnWindowPrev"
    >
    <Grid ShowGridLines="True" MouseDown="OnGrid" PreviewMouseDown="OnGridPrev">
        <!-- Spalten für Grid-Element definieren -->
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <!-- Element für Grafik in der linken Grid-Zelle -->
        <Canvas Grid.Column="0" Width="100" Height="100"
            MouseDown="OnCanvas" PreviewMouseDown="OnCanvasPrev">
            <Ellipse Canvas.Left="5" Canvas.Top="5" Stroke="Black" Fill="Red" Width="70" Height="50"
                MouseDown="OnEllipse" PreviewMouseDown="OnEllipsePrev" />
            <Rectangle Canvas.Left="10" Canvas.Top="30" Stroke="Black" Fill="Yellow" Width="50" Height="50"
                MouseDown="OnRectangle" PreviewMouseDown="OnRectanglePrev"/>
        </Canvas>

        <!-- Rand- mit Text-Element in der rechten Grid-Zelle -->
        <Border Grid.Column="1" Width="100" Height="100" BorderThickness="2"
            BorderBrush="Black" MouseDown="OnBorder" PreviewMouseDown="OnBorderPrev">
            <TextBlock FontSize="15" TextWrapping="Wrap" TextAlignment="Center"
                MouseDown="OnTextBlock" PreviewMouseDown="OnTextBlockPrev"
                HorizontalAlignment="Center" VerticalAlignment="Center">
                Ein Border-Element mit Text...
            </TextBlock>
        </Border>
    </Grid>
</Window>
```

Listing 41.1 Die logische Hierarchie für das Beispiel in Abbildung 41.2

Da wir noch nicht so viele XAML-Hierarchien erzeugt haben, wollen wir hier noch einmal etwas genauer auf das Listing 41.1 eingehen. Das Wurzelement ist hier ein Window-Element mit dem Klassennamen Window1. Für dieses Element wurden sowohl das MouseDown-Ereignis als auch das PreviewMouseDown-Ereignis implementiert. Das Bubbling Event wird in der Methode OnWindow und das dazugehörige Tunneling Event in der Methode OnWindowPrev bearbeitet. Den VB-Code zur XAML-Deklaration finden Sie in Listing 41.2.

Dort werden auch alle Ereignismethoden, welche dieses Beispiel benötigt, implementiert. In den einzelnen Ereignismethoden wird immer ein Informationstext an eine String-Variablen angehängt. Hier werden keine MessageBox-Ausgaben benutzt, da die Erzeugung eines neuen Fensters den Durchlauf des MouseDown-Ereignisses beeinflusst. Um den Ereignisdurchlauf ohne irgendwelche Wechselwirkungen mit anderen WPF-Elementen zu gewährleisten, werden die in der String-Variablen gesammelten Informationen im Finalizer der Hauptfensterklasse mithilfe eines StreamWriter-Objekts in eine Datei geschrieben, die wir dann nach dem Schließen der Applikation mit dem Notepad von Windows anschauen können.

HINWEIS Wenn Sie für die Ausgabe der Informationen in den Ereignismethoden eine MessageBox benutzen, werden normalerweise die Ereignismethoden der Bubbling Events nicht aufgerufen.

Zurück zum XAML-Code (Listing 41.1). Innerhalb des Hauptfensters wird zunächst ein Grid-Element deklariert, für das auch das MouseDown- und das PreviewMouseDown-Ereignis implementiert werden. In diesem Fall werden die Methoden OnGrid bzw. OnGridPrev angesprungen. Über die Grid.ColumnDefinition-Eigenschaft werden dann zwei Spalten für das Grid definiert. Nun geht es weiter in der logischen Hierarchie mit den WPF-Elementen in den beiden Grid-Zellen. In der linken Zelle (Grid.Column="0") wird ein Canvas-Element deklariert, welches die beiden Grafikelemente Rectangle und Ellipse enthält. Für alle Elemente werden die beiden Ereignisse MouseDown und PreviewMouseDown implementiert. Schließlich wird die rechte Grid-Zelle zunächst mit einem Rand (Border) versehen, welcher dann ein TextBlock-Element mit den jeweiligen Ereignis-Aufrufen enthält.

```
Imports System
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Input
Imports System.IO

Namespace Hierarchie
    Partial Public Class Window1
        Inherits System.Windows.Window
        Dim strTest As String = ""

        Public Sub New()
            InitializeComponent()
        End Sub

        Protected Overrides Sub Finalize()
            MyBase.Finalize()
            ' Im Finalizer der Windows1-Klasse werden die gesammelten
            ' Daten in eine Datei gespeichert.
            Dim sw As StreamWriter = New StreamWriter("C:\test.txt")
            sw.WriteLine(strTest)
            sw.Flush()
            sw.Close()
        End Sub

        ' Nun kommen alle Ereignismethoden
        Private Sub OnGrid(ByVal sender As Object,
                           ByVal e As RoutedEventArgs)
            strTest = strTest + "Grid" & vbCrLf
        End Sub
    End Class
End Namespace
```

```
Private Sub OnGridPrev(ByVal sender As Object, _
                      ByVal e As RoutedEventArgs)
    strTest = strTest + "Grid - Preview" & vbCrLf
End Sub

Private Sub OnWindow(ByVal sender As Object, _
                     ByVal e As RoutedEventArgs)
    strTest = strTest + "Window" & vbCrLf
End Sub

Private Sub OnWindowPrev(ByVal sender As Object, _
                       ByVal e As RoutedEventArgs)
    strTest = strTest + "Window - Preview" & vbCrLf
End Sub

Private Sub OnCanvas(ByVal sender As Object, _
                    ByVal e As RoutedEventArgs)
    strTest = strTest + "Canvas" & vbCrLf
End Sub

Private Sub OnCanvasPrev(ByVal sender As Object, _
                       ByVal e As RoutedEventArgs)
    strTest = strTest + "Canvas - Preview" & vbCrLf
End Sub

Private Sub OnTextBlock(ByVal sender As Object, _
                      ByVal e As RoutedEventArgs)
    strTest = strTest + "Textblock" & vbCrLf
End Sub

Private Sub OnTextBlockPrev(ByVal sender As Object, _
                          ByVal e As RoutedEventArgs)
    strTest = strTest + "Textblock - Preview" & vbCrLf
End Sub

Private Sub OnBorder(ByVal sender As Object, _
                    ByVal e As RoutedEventArgs)
    strTest = strTest + "Border" & vbCrLf
End Sub

Private Sub OnBorderPrev(ByVal sender As Object, _
                       ByVal e As RoutedEventArgs)
    strTest = strTest + "Border - Preview" & vbCrLf
End Sub

Private Sub OnEllipse(ByVal sender As Object, _
                     ByVal e As RoutedEventArgs)
    strTest = strTest + "Ellipse" & vbCrLf
End Sub

Private Sub OnEllipsePrev(ByVal sender As Object, _
                        ByVal e As RoutedEventArgs)
    strTest = strTest + "Ellipse - Preview" & vbCrLf
End Sub
```

```

Private Sub OnRectangle(ByVal sender As Object,
                      ByVal e As RoutedEventArgs)
    strTest = strTest + "Rectangle" & vbCrLf
End Sub

Private Sub OnRectanglePrev(ByVal sender As Object,
                           ByVal e As RoutedEventArgs)
    strTest = strTest + "Rectangle - Preview" & vbCrLf
End Sub
End Class
End Namespace

```

Listing 41.2 Der Code zum Beispiel in Abbildung 41.2

Die gesamte logische Hierarchie des Beispiels mit einem MouseDown- und einem PreviewMouseDown-Ereignis können Sie in Abbildung 41.3 verfolgen.

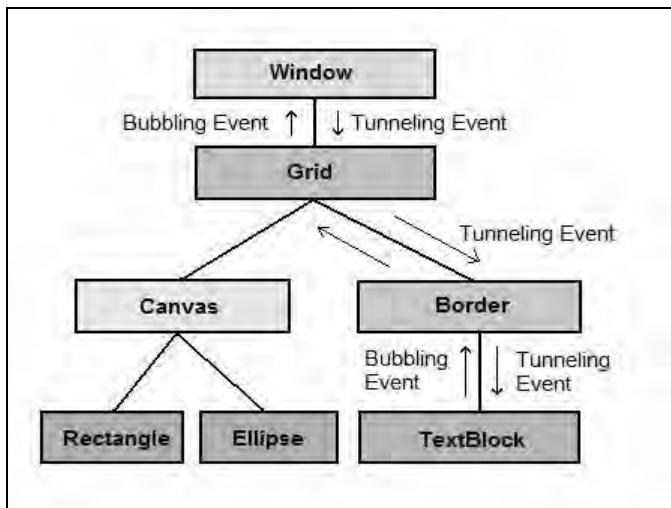


Abbildung 41.3 Ereignisse in der logischen Hierarchie

Nun können wir das Beispielprogramm endlich aufrufen (Abbildung 41.2). Wenn wir mit der Maus auf den Text in der rechten Grid-Zelle klicken und danach die Applikation beenden, finden wir in der Datei »C:\Test.txt« folgende Informationen:

```

Window – Preview
Grid – Preview
Border – Preview
TextBlock – Preview
TextBlock
Border
Grid
Window

```

Sie können sehen, dass zuerst das »Tunneling Event« ausgelöst wird. Der Durchlauf beginnt im Wurzelement der Hierarchie, also im Hauptfenster der Applikation. Dann läuft das Preview-Ereignis durch den logischen Baum bis zum TextBlock-Element, welches das Ereignis ausgelöst hat. Nun läuft das »Bubbling Event« wieder den Weg zurück bis hin zum Wurzelement. In jeder Ereignismethode könnten Sie das Ereignis bearbeiten. Sie müssen aber die Weitergabe des Ereignisses nicht programmieren, denn dies erledigt WPF für Sie. Das Ereignis läuft natürlich auch dann durch die logische Hierarchie, wenn einzelne Ereignismethoden nicht implementiert werden.

Im zweiten Beispiel wollen wir nun auf das gelbe Rechteck in der linken Grid-Zelle klicken und zwar so, dass wir in dem Bereich klicken, in dem auch darunter die rote Ellipse liegt (Abbildung 41.4). Das Ergebnis in der Datei »C:\Test.txt« lautet nun:

```
Window - Preview
Grid - Preview
Canvas - Preview
Rectangle - Preview
Rectangle
Canvas
Grid
Window
```



Abbildung 41.4 Der zweite Versuch mit den Ereignissen

Hier wird das »Tunneling Event« zuerst vom Hauptfenster aus bis zum grafischen Element Rectangle durchlaufen. Wie Sie sehen, wird die Ereignismethode OnEllipsePrev des Ellipse-Elements nicht aufgerufen, da diese Ellipse in der logischen Hierarchie auf der gleichen Ebene wie das Rechteck liegt. Beide grafischen Objekte liegen im gleichen Canvas-Element. Auch hier wird dann das »Bubbling Event« durchlaufen, welches schließlich im Hauptfenster ankommt.

Welche Ereignismethoden in der Hierarchie aufgerufen werden, kann letztendlich von Ihnen gesteuert werden. Jedes weitergeleitete Ereignis erhält beim Aufruf der Ereignismethode ein Objekt vom Typ RoutedEventArgs oder ein Objekt, welches von dieser Klasse abgeleitet ist:

```
Private Sub OnCanvasPrev(ByVal sender As Object,
                         ByVal e As RoutedEventArgs)
    strTest = strTest + "Canvas - Preview" & vbCrLf
End Sub
```

Sie können die Eigenschaft `Handled` im Parameter `e` benutzen, um den weiteren Durchlauf eines Ereignisses durch die logische Hierarchie zu unterbrechen:

```
Private Sub OnCanvasPrev(ByVal sender As Object,_
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Canvas - Preview" & vbCrLf
    e.Handled = True
End Sub
```

Als Standard wird in einer Ereignismethode für die Eigenschaft `Handled` der Wert `False` zurückgegeben. In diesem Fall wird das Ereignis weitergeleitet, wie wir es in den vorangegangenen Beispielen gesehen haben. Setzen Sie nun den Wert von `Handled` auf `True`, ist das Ereignis sozusagen »erledigt« und die Ereignisweiterleitung wird an dieser Stelle unterbrochen. Als Beispiel wollen wir die Eigenschaft `Handled` im Ereignis `PreviewMouseDown` des `Canvas`-Elements auf `True` setzen. Nun werden die in der Hierarchie darunter liegenden `Preview`-Ereignismethoden und alle Methoden des Bubbling Events nicht aufgerufen, wie Sie an der ausgegebenen Test-Datei sehen können:

```
Window – Preview
Grid – Preview
Canvas – Preview
```

Direkte Ereignisse (Direct Events) werden nur für das Element verarbeitet, in dem sie auch ausgelöst wurden. In die Kategorie der direkten Ereignisse zählen z.B. das `MouseEnter`- oder das `MouseLeave`-Ereignis. Diese Ereignisse sind im Normalfall nur für das auslösende Element von Interesse.

Wenn Sie sich die Eigenschaften der Klasse `RoutedEventArgs` anschauen, werden Sie dort jedoch einige wichtige Informationen vermissen. Da wir ein Mausereignis verarbeitet haben, wollen wir meistens auch die Klickposition auswerten. Die Lösung des Problems ist sehr einfach. Wir müssen als zweiten Parameter in der Ereignismethode ein Element vom Typ `MouseEventArgs` annehmen. Listing 41.3 zeigt ein einfaches Fenster, welches wieder die beiden Ereignisse `MouseDown` und `PreviewMouseDown` implementiert (Listing 41.4). Auch hier benutzen wir im Code zunächst einen String in den Ereignismethoden, um die anfallenden Informationen zu sammeln. Da wir nun wissen, in welcher Reihenfolge die Ereignisse auftreten, können wir im »Bubbling Event« eine `MessageBox` mit dem Ergebnis anzeigen. Zusätzlich benötigen wir im Beispiel ein einfaches Rechteck, welches in einem `Canvas`-Element platziert wird.

Im `MouseEventArgs`-Objekt `e` gibt es allerdings immer noch keine X- und Y-Eigenschaften für die Position der Maus. Hier gibt es stattdessen eine interessante Vereinfachung für die Positionsabfrage. Wir benutzen die Methode `GetPosition` aus dem `MouseEventArgs`-Objekt. Beim Aufruf von `GetPosition` müssen Sie als einzigen Parameter ein Objekt in der Benutzerschnittstelle angeben, zu dem die Mauskoordinaten dann relativ umgerechnet werden sollen. Die Ausgabe der Koordinaten bezieht sich nun auf die obere linke Ecke des Hauptfensters. Im »Bubbling Event« beziehen wir uns in der Methode `GetPosition` auf das Rechteck mit dem Objektnamen `rect`. Der Unterschied der beiden Aufrufe ist im Ausgabetext in der `MessageBox` leicht erkennbar.

```
<Window x:Class="EventArgs.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="EventArgs" Height="200" Width="250"
    MouseDown="OnWindow"
```

```
PreviewMouseDown="OnWindowPrev"
>
<Grid>
    <Canvas>
        <Rectangle Canvas.Left="30" Canvas.Top="30" Width="150" Height="100"
            Fill="Red" Stroke="Black" Name="rect"/>
    </Canvas>
</Grid>
</Window>
```

Listing 41.3 Benutzung der MouseEventArgs-Klasse

```
Imports System
Imports System.Windows
Imports System.Windows.Input

Namespace EventArgs
    Partial Public Class Window1
        Inherits System.Windows.Window
        Private strTest As String = String.Empty

        Public Sub New()
            InitializeComponent()
        End Sub

        Public Sub OnWindow(ByVal sender As Object, _
                           ByVal e As MouseEventArgs)
            Dim pt As New Point
            pt = e.GetPosition(Me)
            strTest = strTest + "Bubble X: " _
                & pt.X.ToString & _
                "Y: " & pt.Y.ToString & vbCrLf
            MessageBox.Show(strTest)
        End Sub

        Public Sub OnWindowPrev(ByVal sender As Object, _
                               ByVal e As MouseEventArgs)
            Dim pt As New Point
            pt = e.GetPosition(rect)
            strTest = strTest + "Tunnel X: " _
                & pt.X.ToString & _
                "Y: " & pt.Y.ToString & vbCrLf
        End Sub
    End Class
End Namespace
```

Listing 41.4 Benutzung der Klasse MouseEventArgs

Abschließend sei noch gesagt, dass nicht alle weitergeleiteten Ereignisse bis zum obersten Wurzel-Element der Anwendung laufen. So ist für das MouseDown-Ereignis z.B. ein Button-Element eine Schranke, die das Ereignis nicht ohne weiteres überwinden kann. Dadurch laufen bestimmte Ereignisse nicht sinnloserweise durch die gesamte Objekthierarchie.

Weitergeleitete Befehle (Routed Commands)

Ein anderes wichtiges Konzept in WPF sind die weitergeleiteten Befehle (Routed Commands). Hier geht es um das Problem, dass eine Methode mit mehreren Elementen aus der Benutzerschnittstelle verbunden werden soll. Denken Sie z. B. an den Fall, dass Sie einen Menüpunkt und eine Schaltfläche in der Symbolleiste der Anwendung mit einer einzigen Ereignismethode verbinden wollen. Dies ist im Grunde genommen einfach zu realisieren, indem wir das Klickereignis der beiden Elemente mit der gleichen Methode im Code verbinden.

Nun wollen wir noch einen Schritt weiter gehen. Es ist häufig erforderlich, die beiden Elemente der Benutzerschnittstelle (Menüpunkt und Schaltfläche in der Symbolleiste) in bestimmten Situationen ein- oder auszuschalten. Normalerweise erledigen wir die Einstellungen für das Menü erst dann, wenn das Popup-Menü heruntergeklappt werden soll. Dafür gibt es auch die entsprechenden Ereignisse, die wir benutzen können. Nun gibt es aber ein Problem, denn die Schaltfläche in der Symbolleiste muss sofort bearbeitet werden, wenn sich der Status für die Benutzerschnittstelle ändert. Bauen wir also die Aktivierung und Deaktivierung der Schaltfläche in die gleiche Methode ein, welche die Logik für den Menüpunkt enthält, so wird die Schaltfläche zu spät bearbeitet. Wir müssen also eine eigene Behandlungsmethode für die Elemente in der Symbolleiste programmieren. Das ist natürlich keine gute Lösung.

Hier kommen nun die Routed Commands ins Spiel. Sie können mit diesem WPF-Konzept verschiedene Elemente der Benutzerschnittstelle mit Ereignismethoden verbinden, die dann auch entsprechend zeitnah aufgerufen werden.

Wir wollen das in einem Beispiel vertiefen. Im Hauptfenster der Applikation gibt es ein Menü in einem DockPanel-Element mit zwei Menüpunkten *Neu* und *Beenden*. Zusätzlich haben wir mitten im Fenster eine Schaltfläche *Neu*, die an die gleiche Ereignismethode gebunden werden soll wie der Menüpunkt. Der dazugehörige XAML-Code ist in Listing 41.5 zu sehen.

```
<Window x:Class="Commands.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Commands" Height="200" Width="300"
    >
<DockPanel>
    <Menu DockPanel.Dock="Top">
        <MenuItem Header="Datei">
            <MenuItem Command="New" Header="Neu..."/>
            <Separator />
            <MenuItem Click="OnBeenden" Header="Beenden" />
        </MenuItem>
    </Menu>
    <Button Command="New" Width="100" Height="25">Neu...</Button>
</DockPanel>
</Window>
```

Listing 41.5 Benutzerschnittstelle für das Beispiel mit Routed Commands

Was hier auffällt, ist die Eigenschaft `Command`, die im ersten `MenuItem`- und im `Button`-Element verwendet wird. Für den zweiten Menüpunkt *Beenden* wird dagegen eine ganz normale Ereignismethode für das Klickereignis implementiert.

Hinter der Eigenschaft `Command` steckt nun ein so genannter »Routed Command«. Wir wollen darum zunächst den VB-Code in Listing 12.6 anschauen.

```
Imports System
Imports System.Windows
Imports System.Windows.Input

Namespace Commands
    Partial Public Class Window1
        Inherits System.Windows.Window
        Private strDokument As String = "Das ist ein Test-Text!"
        Public Sub New()
            InitializeComponent()

            ' CommandBinding-Objekt anlegen und Ereignismethoden zuweisen
            Dim commandNew As New CommandBinding(ApplicationCommands.[New])
            AddHandler commandNew.Executed, AddressOf NeuHandler
            AddHandler commandNew.CanExecute, AddressOf NeuAusfuehrenHandler

            'Neues CommandBinding hinzufügen
            Me.CommandBindings.Add(commandNew)
        End Sub

        Private Sub NeuHandler(ByVal sender As Object,
                               ByVal e As ExecutedRoutedEventArgs)
            Dim strHelp As String = "Wollen Sie wirklich alles löschen?"
            Dim res As New MessageBoxResult
            res = MessageBox.Show(strHelp, "Test", MessageBoxButton.YesNo)
            If res = MessageBoxResult.Yes Then
                strDokument = String.Empty
            End If
        End Sub

        Private Sub NeuAusfuehrenHandler(ByVal sender As Object,
                                         ByVal e As CanExecuteRoutedEventArgs)
            ' Darf der Befehl ausgeführt werden?
            e.CanExecute = strDokument.Length > 0
        End Sub

        Private Sub OnBeenden(ByVal sender As Object,
                             ByVal e As RoutedEventArgs)
            Me.Close()
        End Sub
    End Class
End Namespace
```

Listing 41.6 CommandBinding im Einsatz

Im Fensterobjekt wird eine String-Variable deklariert, in welcher sich ein kurzer Text befindet. Dieser Text soll hier im Beispiel unser Dokument darstellen. Das heißt, dass der *Neu*-Befehl als Menüpunkt und als Schaltfläche nur dann aktiviert sein soll, wenn sich in der String-Variablen auch tatsächlich ein Text befindet. Ist die String-Variable leer, darf der *Neu*-Befehl nicht ausgeführt werden. Hierzu benutzen wir ein Objekt der CommandBinding-Klasse, welches wir im Konstruktor des Hauptfensters anlegen. Bei der Instanzierung des Objekts wird im Konstruktor als Parameter der Wert ApplicationCommands. [New] übergeben. In WPF gibt es verschiedene, vordefinierte CommandBinding-Klassen, die Sie in Ihren Applikationen sofort benutzen können.

WICHTIG Es gibt folgende Grundklassen für Routed Commands in WPF:

- ApplicationCommands
New, Open, Save, SaveAs, Print, PrintPreview, Copy, Cut, Paste, Replace, SelectAll,...
- ComponentCommands
MoveDown, MoveUp, MoveLeft, MoveRight, ScrollPageDown, ScrollPageUp,...
- EditingCommands
AlignCenter, AlignJustify, Backspace, Delete, EnterLineBreak, MoveDownByLine,...
- MediaCommands
BoostBass, ChannelUp, ChannelDown, FastForward, MuteVolume, NextTrack, Play, Stop,...

Die obige Liste enthält nicht alle Befehle der jeweiligen *Commands*-Klassen. Die Klasse ApplicationCommands enthält als statische Eigenschaften die diversen Befehle, die normalerweise im Menü einer Applikation zu finden sind. In der Klasse ComponentCommands finden Sie Befehle, die mit Komponenten zu tun haben. EditingCommands enthält die Befehle, die mit dem Ändern von Dokumenten zu tun haben. Schließlich gibt es noch die Klasse MediaCommands, die Befehle enthält, die mit Mediendateien arbeiten.

Die vier *Commands*-Klassen enthalten nicht den Code, um die verschiedenen Befehle auszuführen (z. B. einen Text zu kopieren oder eine Datei zu speichern), sondern nur statische Eigenschaften, die es Ihnen ermöglichen, mehrere Elemente der Benutzerschnittstelle unter die Kontrolle einer Ereignismethode zu stellen.

Wenn Ihnen die vorgegebenen Befehle in den vier *Commands*-Klassen nicht ausreichen, können Sie eigene Befehlsklassen erzeugen.

Nach der Erzeugung des CommandBinding-Objekts in unserem Beispiel können Sie zwei Ereignismethoden definieren, die einerseits den Code aufrufen, der ausgeführt werden soll, wenn der Menüpunkt oder die Schaltfläche angeklickt werden (Ereignis: Executed) und andererseits die Steuerung für die Aktivierung und Deaktivierung der Elemente der Benutzerschnittstelle übernehmen (Ereignis: CanExecute). Diese Ereignisse, die hier ausgelöst werden, sind übrigens ganz normale Routed Events, die die logische Hierarchie der Benutzerschnittstelle durchlaufen.

Schließlich fügen wird das erzeugte CommandBindings-Objekt mit der Methode Add in die Liste der CommandBindings des Elternelements ein. Nun »weiß« unser Hauptfenster von den Bindungen der Schaltfläche und des Menüpunktes und kann diese bei Bedarf durch Aufruf der Ereignismethoden ausführen.

Die Ereignismethoden selbst erhalten ganz normalen Code. In der Methode NeuHandler wird der Anwender gefragt, ob er das Dokument (der string strDokument) löschen will. Wenn ja, wird die String-Variable geleert.

In der Methode NeuAusfuehrenHandler muss die Eigenschaft CanExecute des CanExecuteRoutedEventArgs-Objekts, das als Parameter übergeben wird, auf True gesetzt werden, wenn der Befehl ausgeführt werden darf. Ansonsten wird hier False zurückgegeben. Im Beispiel prüfen wir hierzu, ob die Länge des Textes in strDokument größer als Null ist.

Wenn wir die Applikation nun starten (Abbildung 41.5), ist der *Neu*-Befehl sowohl als Menüpunkt, sowie als Schaltfläche aktiviert, da in der String-Variable ein Text steht.



Abbildung 41.5 Nach dem Start des CommandBindings-Beispiels

Klicken Sie nun auf die Schaltfläche *Neu*. Die Abfrage, ob Sie löschen wollen, beantworten Sie mit »Ja«. Sobald das MessageBox-Fenster geschlossen wird, ändert sich auch der Status in der Benutzerschnittstelle. Die Schaltfläche und der Menüpunkt werden deaktiviert und der Befehl *Neu* kann nicht mehr ausgeführt werden.

Die Ereignismethode NeuHandler wird natürlich nur dann aufgerufen, wenn der Befehl selbst über das Menü oder die Schaltfläche ausgelöst wird. Es stellt sich jedoch die Frage, wie häufig die Methode NeuAusfuehrenHandler aufgerufen wird. Um dies zu prüfen, können wir in diese Ereignismethode zusätzlichen Code einbauen, der jedes Mal einen kleinen Text im Ausgabe-Fenster von Visual Studio ausgibt. Wir benutzen in diesem Fall kein MessageBox-Element für die Datenausgabe, um den Fluss der Ereignisse nicht zu ändern. Zuerst werden wir feststellen, dass die Ereignismethode nicht einfach regelmäßig aufgerufen wird (Polling). Die Ereignismethoden für die Aktivierung und Deaktivierung der Elemente werden immer dann aufgerufen, wenn »irgendwie etwas Wichtiges« in Ihrer Benutzerschnittstelle passiert. So wird die Methode NeuAusfuehrenHandler z.B. aufgerufen, wenn Sie in das Hauptfenster klicken, wenn Sie Popup-Menüs herunterklappen, wenn Sie auf die Schaltfläche *Neu* klicken oder wenn die Applikation gestartet wird. Die Ereignismethode wird nicht aufgerufen, wenn Sie z.B. den Mauszeiger durch das Fenster oder über die Schaltfläche bewegen, ohne mit der Maus zu klicken.

Um alle Aktivierungsmethoden von Hand zu starten, müssen Sie die statische Methode InvalidateRequerySuggested aus der Klasse CommandManager aufrufen:

```
CommandManager.InvalidateRequerySuggested()
```

Ein solcher »außerordentlicher« Aufruf kann dann wichtig sein, wenn ein separater Thread bestimmte Rechenarbeiten erledigt hat, welche den Status der Benutzerschnittstelle beeinflussen. Beim Beenden eines Threads wird nicht automatisch die Methode NeuAusfuehrenHandler aufgerufen.

Das bedeutet natürlich, dass der Code in den Ereignismethoden für die Aktivierung und Deaktivierung der Steuerelemente möglichst kurz sein sollte, damit er sehr schnell ausgeführt werden kann. Sie müssen bedenken, dass es in der Applikation mehrere CommandBindings im Hauptfenster geben kann. In den oben skizzierten Fällen werden dann natürlich alle Ereignismethoden aufgerufen, die am entsprechenden Elternelement gebunden sind.

Weitergeleitete Ereignisse unterstützen die Trennung von Logik und Design. Der Softwareentwickler implementiert Befehle mit ganz bestimmten Namen. Der Designer kann diese Befehle mit den Steuerelementen in der Benutzerschnittstelle verbinden, ohne dass er einen Namen für eine spezielle Ereignismethode vergeben muss. Er benutzt einfach den Namen des gewünschten Befehls:

```
<MenuItem Command="New" Header="Neu..."/>
<Button Command="Copy">Kopieren</Button>
```

Eigenschaften der Abhängigkeiten

Kommen wir nun zu den Eigenschaften der Abhängigkeiten (Dependency Properties) bei den Steuerelementen von Windows Presentation Foundation.

Eigenschaften werden in der logischen Hierarchie von WPF weiter vererbt. Wird z.B. in einem StackPanel-Element eine bestimmte Schriftartgröße (Eigenschaft FontSize) gesetzt, so wird diese an alle Elemente weitergegeben, die in diesem Layout-Element enthalten sind. Dabei wird auch das Layout der Elemente neu berechnet, denn durch eine größere oder kleinere Schriftart für die Kindelemente im StackPanel können sich natürlich auch die Abmessungen und Positionen ändern (sofern es das Elternelement erlaubt). Das folgende Beispiel soll die Möglichkeiten, die sich durch die Abhängigkeitseigenschaften ergeben, demonstrieren (Listing 41.7 und Listing 41.8).

```
<Window x:Class="Depend.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Depend" Height="200" Width="650"
    >
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Button Margin="5" Tag="8" Grid.Row="0" Grid.Column="0" Click="WindowOnClick">Window: 8pt</Button>
    <Button Margin="5" Tag="16" Grid.Row="0" Grid.Column="1" Click="WindowOnClick">Window: 16pt</Button>
    <Button Margin="5" Tag="32" Grid.Row="0" Grid.Column="2" Click="WindowOnClick">Window: 32pt</Button>
```

```
<Button Margin="5" Tag="8" Grid.Row="1" Grid.Column="0" Click="ButtonOnClick">Button: 8pt</Button>
<Button Margin="5" Tag="16" Grid.Row="1" Grid.Column="1" Click="ButtonOnClick">Button: 16pt</Button>
<Button Margin="5" Tag="32" Grid.Row="1" Grid.Column="2" Click="ButtonOnClick">Button: 32pt</Button>
</Grid>
</Window>
```

Listing 41.7 Test der Eigenschaften von Abhängigkeiten

```
Imports System
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Input

Namespace Depend
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Public Sub WindowOnClick(ByVal sender As Object,
                               ByVal e As RoutedEventArgs)
            'FontSize-Eigenschaften des Fensters ändern
            Dim btn As New Button
            btn = CType(e.Source, Button)
            Me.FontSize = Convert.ToDouble(btn.Tag)
        End Sub

        Private Sub ButtonOnClick(ByVal sender As Object,
                               ByVal e As RoutedEventArgs)
            'FontSize-Eigenschaft einer Schaltfläche ändern
            Dim btn As New Button
            btn = CType(e.Source, Button)
            btn.FontSize = Convert.ToDouble(btn.Tag)
        End Sub
    End Class
End Namespace
```

Listing 41.8 Die Schriftartgröße wird als Eigenschaft einer Abhängigkeit geändert

In diesem Beispiel werden sechs Schaltflächen dargestellt, welche die Schriftgrößen im Fenster oder auf den Schaltflächen selbst ändern können. Der Code für die Änderung der Schriftgrößen wurde in den Klick-Ereignismethoden `WindowOnClick` und `ButtonOnClick` untergebracht. Dort wird zunächst mithilfe des Methodenparameters `sender` die Schaltfläche ermittelt, auf die geklickt wurde. In der Eigenschaft `Tag` der einzelnen Schaltflächen ist die einzustellende Schriftgröße hinterlegt, die dann konvertiert und entweder für das gesamte Fenster oder nur für die jeweilige Schaltfläche gesetzt wird. Nach dem Start des Programms haben alle Schaltflächen die gleiche Schriftgröße (Abbildung 41.6).



Abbildung 41.6 Start des Testprogramms

Die Schaltflächen erben in diesem Beispiel die Schriftgröße des Elternelements, also des umgebenden Fensters. Klicken Sie nun auf die Schaltfläche *Button: 8pt*. Es wird nur die Schriftgröße dieser einen Schaltfläche geändert (Abbildung 41.7).



Abbildung 41.7 Schaltfläche mit kleiner Schriftgröße

Im nächsten Test klicken wir nun auf die Schaltfläche *Window: 16pt*. Nun sollte für das gesamte Fenster die Schriftgröße 16pt verwendet werden, außer für die Schaltfläche, die wir eben im ersten Test auf die kleine Schriftgröße (8pt) gesetzt haben (Abbildung 41.8):



Abbildung 41.8 Das Fenster mit 16pt-Schriftgröße

Die eingestellte Schriftgröße wurde bei diesem Versuch durch die gesamte logische Hierarchie der Benutzeroberfläche weiter vererbt. Da für die Schaltfläche *Button: 8pt* bereits explizit eine andere Schriftgröße gesetzt wurde, ist an dieser Stelle die Vererbung unterbrochen worden. Klicken Sie nun noch auf die Schaltfläche *Button: 32pt* und Sie werden sehen, dass nur die Schriftart dieser Schaltfläche vergrößert wird (Abbildung 41.9).



Abbildung 41.9 Der letzte Test

Ein weiterer Klick auf die Schaltfläche *Window: 8pt* würde nun durch die Vererbung der Eigenschaften alle Schaltflächen in der oberen Reihe und die Schaltfläche *Button: 16pt* in der unteren Reihe des Fensters ändern.

Da im letzten Beispiel ein Grid-Element benutzt wurde, um die Schaltfläche zu positionieren, hat sich die Größe der Kindelemente bei der Änderung der Schriftgröße nicht geändert. Sie können natürlich andere Szenarien definieren, bei denen die Größe der Kindelemente durch eine Layout-Berechnung von WPF neu ermittelt wird.

Wenn Sie die Vererbung einer Eigenschaft durch das explizite Setzen dieser Eigenschaft »unterbrochen« haben, wird der vorgegebene Wert für die Eigenschaft dieses Steuerelements benutzt und natürlich von dort an andere Steuerelemente, die sich in der Hierarchie darunter befinden, weitergegeben. Sie können das explizite Setzen der Eigenschaft wieder aufheben, indem Sie die Methode *ClearValue* anwenden:

```
Me.ClearValue(Window.FontSizeProperty)
```

Mit der obigen Codezeile können Sie die Eigenschaft *FontSize* wieder durch die gesamte Hierarchie, einschließlich des Steuerelements *Me*, vererben.

Eingaben

Eingaben können mit der Maus, über die Tastatur und mit dem Stift gemacht werden. Die Mauseingaben werden immer zunächst zu dem Element der Benutzerschnittstelle geleitet, das sich genau unter dem Mauszeiger befindet. Handelt es sich um ein Routed Event, läuft das Ereignis nun durch die logische Hierarchie und wird ggf. in den verschiedenen Ereignismethoden abgearbeitet. Ein direktes Ereignis kann nur Element-spezifisch verarbeitet werden.

Maus-Ereignis	Weiterleitung	Aktion
GotMouseCapture	Bubbling	Das Element hält den Mausfokus fest
LostMouseCapture	Bubbling	Das Element hat den Mausfokus verloren
MouseEnter	Direkt	Der Mauszeiger wird in das Element hinein bewegt
MouseLeave	Direkt	Der Mauszeiger wird aus dem Element hinaus bewegt
MouseDown, PreviewMouseDown	Bubbling, Tunneling	Eine Maustaste wurde gedrückt
MouseUp, PreviewMouseUp	Bubbling, Tunneling	Eine Maustaste wurde losgelassen

Maus-Ereignis	Weiterleitung	Aktion
MouseMove, PreviewMouseMove	Bubbling, Tunneling	Die Maus wurde bewegt
MouseWheel, PreviewMouseWheel	Bubbling, Tunneling	Das Mausrad wurde gedreht
MouseLeftButtonDown, PreviewMouseLeftButtonDown	Bubbling, Tunneling	Die linke Maustaste wurde gedrückt
MouseLeftButtonUp, PreviewMouseLeftButtonUp	Bubbling, Tunneling	Die linke Maustaste wurde losgelassen
MouseRightButtonDown, PreviewMouseRightButtonDown	Bubbling, Tunneling	Die rechte Maustaste wurde gedrückt
MouseRightButtonUp, PreviewMouseRightButtonUp	Bubbling, Tunneling	Die rechte Maustaste wurde losgelassen
QueryCursor	Bubbling	Ein Element fragt nach dem momentanen Mauscursor

Tabelle 41.1 Maus-Ereignisse

Beim Programmieren mit der Maus gibt es noch eine interessante Eigenschaft, die aus der Klasse `UIElement` kommt, von der jedes WPF-Steuerelement abgeleitet ist. Die Eigenschaft heißt `IsMouseOver` und liefert einen booleschen Wert zurück. Die Eigenschaft `IsMouseOver` gibt immer dann den Wert `True` zurück, wenn sich der Mauszeiger über dem Steuerelement oder einem seiner sichtbaren Kindelemente befindet.

Bitte beachten Sie, dass das schon oft verwendete `Click`-Ereignis in Tabelle 41.1 der Maus-Ereignisse nicht enthalten ist. Ein `Click`-Ereignis kann nicht nur durch die Maus ausgelöst werden, sondern auch durch eine Tastatur. Außerdem können sich hinter einem `Click`-Ereignis mehrere Maus-Ereignisse verbergen (`MouseUp`, `MouseDown`). Darum finden Sie in der Klasse `Control` die beiden zusätzlichen Ereignisse `MouseDoubleClick` und `PreviewMouseDoubleClick`. Die Klasse `ButtonBase` implementiert dann das `Click`-Ereignis. `ButtonBase` ist die Basisklasse der Klassen `Button`, `RadioButton` und `CheckBox`.

Auch für die Tastatur stehen diverse Ereignisse zur Verfügung (Tabelle 41.2). Hier spielt das Konzept des Eingabefokus eine entscheidende Rolle. Nur das Steuerelement enthält Eingaben von der Tastatur, welches den Eingabefokus hat. Der Fokus kann durch Anklicken mit der Maus, durch die -Taste oder durch Navigieren mit den Cursor-Tasten zu einem bestimmten Steuerelement gebracht werden.

Tastatur-Ereignis	Weiterleitung	Aktion
GotFocus, PreviewGotFocus	Bubbling, Tunneling	Ein Element erhält den Eingabefokus
LostFocus, PreviewLostFocus	Bubbling, Tunneling	Ein Element verliert den Eingabefokus
KeyDown, PreviewKeyDown	Bubbling, Tunneling	Eine Taste wird gedrückt
KeyUp, PreviewKeyUp	Bubbling, Tunneling	Eine Taste wird losgelassen
TextInput, PreviewTextInput	Bubbling, Tunneling	Ein Element empfängt eine Texteingabe

Tabelle 41.2 Tastatur-Ereignisse

In Tabelle 41.3 können Sie schließlich die Ereignisse für den Stift des Tablett-PCs sehen.

Stift-Ereignis	Weiterleitung	Aktion
GotStylusCapture	Bubbling	Das Element hält den Stiftfokus fest
LostStylusCapture	Bubbling	Das Element verliert den Stiftfokus
StylusDown, PreviewStylusDown	Bubbling, Tunneling	Der Stift berührt den Bildschirm über einem Element
StylusUp, PreviewStylusUp	Bubbling, Tunneling	Der Stift wird vom Bildschirm über einem Element hoch gehoben
StylusEnter	Direkt	Der Stift wird in das Element hinein bewegt
StylusLeave	Direkt	Der Stift wird aus dem Element hinaus bewegt
StylusInRange, PreviewStylusInRange	Bubbling, Tunneling	Der Stift befindet sich dicht über dem Bildschirm
StylusOutOfRange, PreviewStylusOutOfRange	Bubbling, Tunneling	Der Stift befindet sich außerhalb des Bildschirmerfassungsabstands
StylusMove, PreviewStylusMove	Bubbling, Tunneling	Der Stift wird über das Element bewegt
StylusInAirMove, PreviewStylusInAirMove	Bubbling, Tunneling	Der Stift wird über das Element bewegt, berührt dabei aber nicht den Bildschirm
StylusSystemGesture, PreviewStylusSystemGesture	Bubbling, Tunneling	Es wird eine Stift-Geste erkannt
TextInput, PreviewTextInput	Bubbling, Tunneling	Das Element empfängt eine Texteingabe

Tabelle 41.3 Stift-Ereignisse

Schaltflächen

Schaltflächen sind einfache Steuerelemente, die angeklickt werden können. Hierbei unterscheiden wir verschiedene Typen: Neben der Standard-Schaltfläche (Button) gibt es in WPF auch noch die Steuerelemente Kontrollkästchen (CheckBox) und Auswahlkästchen (RadioButton). Die drei Klassen sind von ButtonBase abgeleitet und stellen das Click-Ereignis zur Verfügung. Die Anwendung dieser Elemente ist sehr einfach:

```
<Button Click="OnButtonClicked" Name="btn">Button</Button>
<CheckBox Click="OnCheckboxClicked" Name="check">CheckBox</CheckBox>
<RadioButton Click="OnRadioButtonClicked" Name="radio">RadioButton</RadioButton>
```

Die Methode für das Click-Ereignis der Schaltfläche kann folgendermaßen implementiert werden:

```
Private Sub OnButtonClicked(ByVal sender As Object, ByVal e As RoutedEventArgs)
    MessageBox.Show("Schaltfläche geklickt!")
End Sub
```

Entsprechende Methoden können auch für die beiden anderen Steuerelemente erstellt werden. Wenn Sie aus dem Code auf die Steuerelemente zugreifen wollen, müssen diese mit einem Namen versehen werden.

Die drei Schaltflächenarten können nicht nur einen einfachen Text enthalten, sondern sie können beliebige andere Elemente, auch Grafik, darstellen. Im folgenden Listing 41.9 werden die drei Elemente mit einem erweiterten Inhalt dargestellt.

```
<Window x:Class="ButtonTest.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Test mit Schaltflächen" Height="200" Width="300"
    >
    <Grid>
        <StackPanel Width="150">

            <Button>
                <TextBlock>
                    <Ellipse Margin="0,0,5,0" Width="20" Height="10" Fill="Red" Stroke="Black" />
                    Button
                    <Ellipse Margin="5,0,0,0" Width="20" Height="10" Fill="Green" Stroke="Black" />
                </TextBlock>
            </Button>

            <CheckBox>
                <TextBlock Foreground="White" FontSize="18">
                    <TextBlock.Background>
                        <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
                            <GradientStop Color="Blue" Offset="0" />
                            <GradientStop Color="Red" Offset="1" />
                        </LinearGradientBrush>
                    </TextBlock.Background>
                    CheckBox
                </TextBlock>
            </CheckBox>

            <RadioButton>
                <TextBlock FontSize="20" FontFamily="Algerian">
                    RadioButton
                </TextBlock>
            </RadioButton>

        </StackPanel>
    </Grid>
</Window>
```

Listing 41.9 Steuerelemente mit erweitertem Inhalt



Abbildung 41.10 Steuerelemente mit erweitertem Inhalt

In Abbildung 41.10 sehen Sie das Ergebnis unserer Programmierung. Einige der oben benutzten WPF-Elemente werden Sie in den folgenden Kapiteln noch näher kennen lernen. Das Beispiel soll Ihnen hier nur zeigen, dass es mit WPF sehr einfach ist, auch komplexe grafische Effekte mit den vorhandenen Steuerelementen zu nutzen.

Für alle Schaltflächenarten wird in Listing 41.9 ein `TextBlock`-Element verwendet, um eine weitere Gestaltung und Formatierung durchzuführen. Die normale Schaltfläche wird mit grafischen Grundelementen (Ellipsen) erweitert. Für das `CheckBox`-Element wird ein neuer Hintergrund mit einem Farbverlauf (`LinearGradientBrush`) definiert und für das `RadioButton`-Element wurde eine andere Schriftart ausgewählt.

Bildlaufleisten und Schiebereglер

In WPF finden Sie Steuerelemente, mit denen Sie einen Wert aus einem vorgegebenen Bereich auswählen können. Zu diesen Steuerelementen gehören die Klassen `Slider` und `ScrollBar`. Die Benutzung der `Slider`-Elemente ist in Listing 41.10 dargestellt.

```
<Window x:Class="Schieber.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Schieberegler" Height="280" Width="300"
>
<Grid>
    <Canvas>
        <Slider Name="sldVert" Orientation="Vertical" Margin="15,15,15,15"
            Minimum="0" Maximum="100" Height="200" Value="0" ValueChanged="OnValueVert" />
        <Slider Name="sldHorz" Orientation="Horizontal" Margin="60,15,15,15"
            Minimum="0" Maximum="100" Width="200" Value="0" ValueChanged="OnValueHorz" />

        <Label Name="lblHorz" FontSize="15" Margin="120,30,15,15" />
        <Label Name="lblVert" FontSize="15" Margin="30,100,15,15" />
    </Canvas>
</Grid>
</Window>
```

Listing 41.10 Slider-Elemente

```
Imports System
Imports System.Windows

Namespace Schieber
    Partial Public Class Window1
```

```

Inherits System.Windows.Window

Public Sub New()
    InitializeComponent()
End Sub

Private Sub OnValueHorz(ByVal sender As Object, ByVal e As RoutedEventArgs)
    lblHorz.Content = sldHorz.Value.ToString()
End Sub

Private Sub OnValueVert(ByVal sender As Object, ByVal e As RoutedEventArgs)
    lblVert.Content = sldVert.Value.ToString()
End Sub
End Class
End Namespace

```

Listing 41.11 Ereignismethoden für die Schieberegler

Im Beispiel werden zwei Schieberegler deklariert, die in horizontaler und vertikaler Richtung verlaufen (Listing 41.10). Dies wird über die Eigenschaft `Orientation` eingestellt. Das hier verwendete `Canvas`-Element dient nur zur Positionierung aller Elemente und muss für unser Beispiel nicht weiter betrachtet werden. Für die Schieberegler werden jeweils ein `Minimum`- und ein `Maximum`-Wert festgelegt. Die Ausgaben der `Value`-Eigenschaft bewegen sich genau in diesem vorgegebenen Bereich. In unserem Beispiel wurden zusätzlich zwei `Label`-Elemente definiert, welche die Eigenschaft `Value` der beiden Schieberegler ausgeben.

Nun wollen wir zwei Ereignismethoden im Code implementieren (Listing 41.11). Das Ereignis `ValueChanged` der beiden Schieberegler wird auf die Methoden `OnValueHorz` und `OnValueVert` geleitet. Dort wird die Ausgabe der `Value`-Eigenschaft der Schieberegler in einen String konvertiert und im dazugehörigen `Label`-Element dargestellt. Die `Value`-Eigenschaft des `Slider`-Elements liefert übrigens einen Wert vom Typ `Double` zurück, wie Sie es beim Ausführen des Beispielprogramms sofort erkennen können (Abbildung 41.11).



Abbildung 41.11 Zwei Schieberegler im Einsatz

Mit den Bildlaufleisten (`ScrollBar`) können Sie ganz ähnlich wie mit den Schieberegbern (`Slider`) arbeiten. Bildlaufleisten bieten zusätzlich die Eigenschaften `LargeChange` und `SmallChange` an, mit denen Sie das Blättern durch ein größeres Dokument steuern können.

Ein sehr interessantes Steuerelement von WPF ist das `ScrollViewer`-Element, mit dem es sehr leicht möglich ist, einen Inhalt mit Bildlaufleisten darzustellen, wenn dieser nicht in das Elternelement passt (Listing 41.12). Der darzustellende Inhalt wird als Kindelement im `ScrollViewer`-Element deklariert. Ist das Kind-element größer als der verfügbare Platz, werden die benötigten Bildlaufleisten dargestellt (Abbildung 41.12) und können benutzt werden, ohne dass eine weitere Programmierung erforderlich ist.

```
<Window x:Class="Scroller.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ScrollViewer" Height="300" Width="300"
>
<Grid>
    <ScrollViewer HorizontalScrollBarVisibility="Auto" VerticalScrollBarVisibility="Auto">
        <Ellipse Width="1000" Height="1000" Fill="Red" Stroke="Black" StrokeThickness="5" />
    </ScrollViewer>
</Grid>
</Window>
```

Listing 41.12 Viel Inhalt mit den `ScrollViewer`-Element

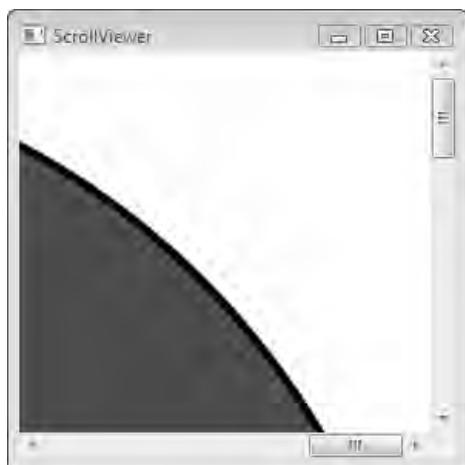


Abbildung 41.12 Benutzung des `ScrollViewer`-Elements

Das `ScrollViewer`-Element kann in beliebigen Layout-Elementen eingesetzt werden. So können Sie z.B. die Zelle eines `Grid`-Elements sehr einfach mit Bildlaufleisen versehen. Auch die Ausgabe von Texten kann von einem `ScrollViewer`-Element unterstützt werden. Hierbei ist es meistens sinnvoll, die horizontale Bildlaufleiste auszuschalten und nur die senkrechte Leiste zu benutzen. Listing 41.13 zeigt ein Beispiel mit einem `Grid`-Element, bestehend aus zwei Zeilen und zwei Spalten, in dem eine Ellipse, ein Bild und ein Text in den einzelnen Zellen ausgegeben werden. Für die Ausgabe des Textes in der unteren `Grid`-Zeile wird die horizontale Bildlaufleiste ausgeschaltet und der Zeilenumbruch wird für das `TextBlock`-Element aktiviert.

Versuchen Sie nach dem Start des Beispielprogramms das Fenster zu vergrößern. Sobald der Inhalt vollständig in einer `Grid`-Zelle dargestellt werden kann, verschwinden die Bildlaufleisten (Abbildung 41.13).

```
<Window x:Class="Scroller2.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ScrollViewer" Height="300" Width="300"
>
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>

    <ScrollViewer Grid.Column="0" Grid.Row="0" HorizontalScrollBarVisibility="Auto"
        VerticalScrollBarVisibility="Auto">
        <Ellipse Width="200" Height="200" Fill="Red" Stroke="Black" StrokeThickness="3" />
    </ScrollViewer>

    <ScrollViewer Grid.Column="1" Grid.Row="0" HorizontalScrollBarVisibility="Auto"
        VerticalScrollBarVisibility="Auto">
        <Image Source="IC5146.Jpg" />
    </ScrollViewer>

    <ScrollViewer Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="2" VerticalScrollBarVisibility="Auto"
        HorizontalScrollBarVisibility="Disabled" >
        <TextBlock TextWrapping="Wrap" FontSize="20">
            Hier werden mehrere ScrollViewer-Elemente von Windows Presentation Foundation eingesetzt.
            Es ist nun sehr einfach möglich, große Elemente mit Bildlaufleisten darzustellen.
        </TextBlock>
    </ScrollViewer>
</Grid>
</Window>
```

Listing 41.13 ScrollViewer-Elemente in einem Grid



Abbildung 41.13 Mehrere ScrollViewer-Elemente

Steuerelemente für die Texteingabe

Das einfachste Steuerelement für die Texteingabe ist das TextBox-Element, mit dem Sie normalerweise eine Zeile Text eingeben können. Durch Setzen der Eigenschaft AcceptsReturn auf True ist es jedoch möglich, mehrere Textzeilen einzugeben. Den eingegebenen Text können Sie über die Eigenschaft Text abfragen oder vorgeben.

Eine weitere Variante bei der Texteingabe stellt das PasswordBox-Element dar. Die Texte in diesem Steuerelement werden nicht lesbar dargestellt. Außerdem wird dieses Element über die Eigenschaft Password abgefragt.

HINWEIS Denken Sie daran, dass Kennwörter, die in einem normalen String-Objekt gespeichert werden, solange im Speicher stehen, bis der Garbage Collector den Speicher löscht bzw. überschreibt. Kennwörter sollten darum in SecureString-Objekten abgelegt werden. Wenn diese nicht mehr benötigt werden, kann der Speicher mit dem Kennwort sofort gelöscht werden. Außerdem wird das Kennwort im Speicher verschlüsselt abgelegt, sodass es nicht mit einem Debugger oder aus der Swap-Datei ausgelesen werden kann.

Die Anwendung des TextBox-Elements wird in Listing 41.14 gezeigt. Wenn Sie die Anwendung starten, können Sie die vorgegebenen Texte modifizieren (Abbildung 41.14). Bei der Eingabe eines Kennworts (ganz unten) wird mit der Eigenschaft PasswordChar das Zeichen »#« gesetzt. Dies ist das Zeichen, welches als Platzhalter für die Kennwortzeichen im PasswordBox-Element ausgegeben wird.

```
<Window x:Class="TextBoxen.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Textboxen" Height="300" Width="300"
    >
    <Grid>
        <StackPanel Orientation="Vertical">
            // Einfache TextBox
            <Label Margin="5">Einfache TextBox:</Label>
            <TextBox Name="text1" Margin="5">Einfache TextBox</TextBox>

            // Mehrzeilige TextBox
            <Label Margin="5">TextBox mit mehreren Zeilen:</Label>
            <TextBox Name="text2" AcceptsReturn="True" Margin="5" Height="70">Mehrere Zeilen</TextBox>

            // TextBox für ein Kennwort
            <Label Margin="5">Kennwort-Eingabe:</Label>
            <PasswordBox Name="text3" PasswordChar="#" Margin="5" />
        </StackPanel>
    </Grid>
</Window>
```

Listing 41.14 Verschiedene Texteingaben



Abbildung 41.14 TextBox-Elemente und PasswordBox-Element

Ein TextBox-Element erlaubt nur die einfache Eingabe von Buchstaben. Wesentlich flexibler ist dagegen das RichTextBox-Element. Hier können Texte und grafische Elemente gemischt werden (Listing 41.15).

```
<Window x:Class="RichText.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="RichTextBox" Height="140" Width="500"
>
<Grid>
    <RichTextBox>
        <FlowDocument>
            <Paragraph>
                <Run xml:space="preserve" FontSize="20">Ein RichTextBox-Element </Run>
                <Ellipse Width="50" Height="20" Fill="Red" Stroke="Black" />
                <Run xml:space="preserve" FontSize="20"> kann neben einem Text auch </Run>
                <Run FontWeight="Bold" FontStyle="Italic" FontSize="20">grafische Elemente</Run>
                <Run xml:space="preserve" FontSize="20"> enthalten.</Run>
                <Polyline Stroke="Green" StrokeThickness="3" Points="0,0 20,20 40,0 60,20 80,0 100,20" />
            </Paragraph>
        </FlowDocument>
    </RichTextBox>
</Grid>
</Window>
```

Listing 41.15 Text und Grafik in einem RichTextBox-Element

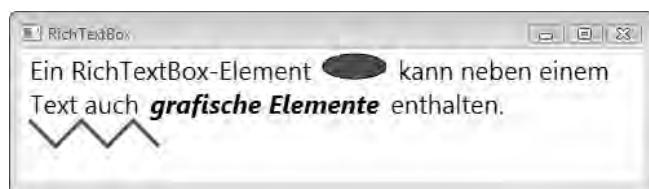


Abbildung 41.15 Das RichTextBox-Element mit Text und Grafik

Innerhalb des RichTextBox-Elementes haben wir zunächst ein FlowDocument-Element deklariert, welches dann ggf. mehrere Paragraph-Elemente enthalten kann. Jedes Paragraph-Element stellt einen kompletten Abschnitt des Dokuments dar und kann beliebige andere Elemente enthalten. Dazu gehören Texte, grafische Elemente und auch normale Steuerelemente. Texte werden in Run-Elementen vorgegeben, die sehr vielfältige Formatierungsanweisungen enthalten können. Um Zeichnungen innerhalb der Texte darzustellen, werden an den entsprechenden Stellen einfach die normalen grafischen WPF-Elemente benutzt (Listing 41.15).

Sie können die einzelnen Elemente, die in einem RichTextBox-Element enthalten sind, auch zur Laufzeit modifizieren. Hierzu müssen Sie die Elemente, die verändert werden sollen, mit einem Namen versehen. Danach können Sie aus dem Code ganz normal auf die Elemente zugreifen. Listing 41.16 zeigt den XAML-Teil eines Beispiels.

```
<Window x:Class="RichText2.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Änderbares Rechteck" Height="190" Width="400"
>
<Grid>
    <StackPanel Orientation="Vertical">
        // Eingabe von Breite und Höhe des Rechtecks
        <StackPanel Orientation="Horizontal">
            <Label FontSize="20">Breite:</Label>
            <TextBox Name="textBreite" FontSize="20" Width="100" TextChanged="OnTextChanged">50</TextBox>
            <Label FontSize="20">Höhe:</Label>
            <TextBox Name="textHoehe" FontSize="20" Width="100" TextChanged="OnTextChanged">20</TextBox>
        </StackPanel>

        // Text mit einem Rechteck
        <RichTextBox Margin="10" Height="100">
            <FlowDocument>
                <Paragraph>
                    <Run xml:space="preserve" FontSize="20">Hier ist ein </Run>
                    <Rectangle Name="rect" Width="50" Height="20" Fill="Red" Stroke="Black"
                        StrokeThickness="2" />
                    <Run FontSize="20">, welches in Breite und Höhe geändert werden kann.</Run>
                </Paragraph>
            </FlowDocument>
        </RichTextBox>
    </StackPanel>
</Grid>
</Window>
```

Listing 41.16 Ein Text mit einem variablen Rechteck

Zunächst erzeugen wir zwei TextBox-Elemente mit den Namen »textBreite« und »textHoehe« im oberen Bereich eines StackPanel-Elements. Das TextChanged-Ereignis dieser Eingabeelemente wird an die Methode OnTextChanged gebunden (Listing 41.17). In unteren Teil deklarieren wir ein RichTextBox-Element, welches einen Text und das zu steuernde Rechteck mit dem Namen »rect« enthält.

```

Imports System
Imports System.Windows

Namespace RichText2
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnTextChanged(ByVal sender As Object, _
                                  ByVal e As RoutedEventArgs)

            ' Prüfen, ob alle Elemente vorhanden sind
            If textBreite Is Nothing Or _
                textHoehe Is Nothing Or _
                rect Is Nothing Then Return

            Dim dBreite As Double
            Dim DHoehe As Double
            Dim bTest As Boolean

            ' Fehlerhafte Eingaben werden mit TryParse erkannt
            bTest = Double.TryParse(textBreite.Text, dBreite)
            bTest = Double.TryParse(textHoehe.Text, DHoehe)

            ' Alle Werte sind OK: Rechteck ändern
            If bTest And dBreite > 0.0 And DHoehe > 0.0 Then
                rect.Width = dBreite
                rect.Height = DHoehe
            End If

        End Sub
    End Class
End Namespace

```

Listing 41.17 Steuerung des Rechtecks im *RichTextBox*-Element

Wenn wir die Ereignismethode programmieren, müssen wir allerdings ein bisschen aufpassen. Zuerst wird in der logischen Hierarchie die *TextBox* für die Breite deklariert und dementsprechend wird sie zur Laufzeit des Programms auch als erste erzeugt und dargestellt. Nun wird bei der Initialisierung dieser *TextBox* im XAML-Code die Zahl »50« zugewiesen und dadurch das *TextChanged*-Ereignis ausgelöst. In der Ereignismethode müssen wir also berücksichtigen, dass beim ersten Aufruf die zweite *TextBox* (Höhe) und das Rechteck im *RichTextBox*-Element noch gar nicht existieren. Darum werden in der Ereignismethode die einzelnen Objekte auf *nothing* getestet.

Außerdem müssen wir damit rechnen, dass ein Anwender unseres Programms fehlerhafte Eingaben tätigt. Dazu gehören negative Zahlen oder auch Buchstaben. Aus diesem Grund wird die Methode *TryParse* aus dem Typ *Double* benutzt, um die Eingaben zu prüfen, bevor das Rechteck manipuliert wird (Abbildung 41.16).

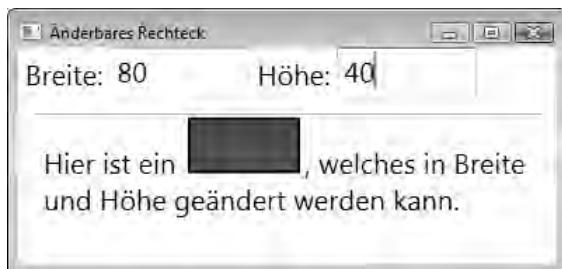


Abbildung 41.16 Das Rechteck wurde geändert

Das Label-Element

Das Label-Element haben Sie in den vorangegangenen Beispielen schon kennen gelernt. Es dient in erster Linie dazu, vor anderen Steuerelementen einen Text auszugeben, wie in Listing 41.16 bereits gezeigt wurde. Im ersten Moment sieht es so aus, als ob das Label-Element überflüssig wäre und durch eine einfache TextBox simuliert werden könnte. Bei einem Label kommt jedoch als neue Möglichkeit die Verarbeitung des Fokus hinzu.

Sie sollten ihre Benutzerschnittstellen immer so aufbauen, dass der Anwender auch mit der Tastatur die verschiedenen Steuerelemente in einem Dialogfenster ansteuern kann. Hierbei spielt das Label-Element eine große Rolle. Sie können für jedes Label eine Kurztaste (Access Key) definieren, die der Anwender in Kombination mit der Alt-Taste zur Ansteuerung des Label-Elements verwenden kann. Da nun das Label selbst jedoch nicht den Eingabefokus erhalten kann, wird dieser an das im Label gebundene Element weitergegeben.

In Listing 41.18 wird statt des normalen Texts im Label ein AccessText-Element benutzt. Der Buchstabe hinter dem Unterstrichungszeichen ist die Kurztaste für das Label-Element. Sie können also den Eingabefokus der TextBox für die Breite zuordnen, indem Sie die Tastenkombination **Alt** **B** eingeben. Das Ziel-Element, dem der Eingabefokus zugewiesen werden soll, wird in der Target-Eigenschaft des Label-Elements angegeben. Hier müssen Sie folgende Syntax benutzen:

```
Target="{Binding ElementName=textBreite}"
```

Als ElementName wird das Steuerelement angegeben, welches den Eingabefokus enthalten soll.

HINWEIS Die Kurztasten werden in der Benutzerschnittstelle erst dann durch die Unterstrichungszeichen gekennzeichnet, wenn Sie die Alt-Taste drücken (Abbildung 41.17).

Wie Sie in Listing 41.18 sehen können, ist das Element AccessText nicht unbedingt erforderlich, um die Kurztaste zu definieren. Das AccessText-Element ist dann sehr nützlich, wenn Sie die Textausgabe formatieren oder die vielfältigen Möglichkeiten mit Animationen benutzen wollen.

```
<Window x:Class="Labels.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Label" Height="100" Width="450"
>
```

```

<Grid>
    <StackPanel Orientation="Horizontal" Height="40">
        <Label FontSize="20" Target="{Binding ElementName=textBreite}">
            <AccessText> Breite:</AccessText>
        </Label>
        <TextBox Name="textBreite" FontSize="20" Width="100" ToolTip="Breitenangabe">50</TextBox>

        <Label FontSize="20" Target="{Binding ElementName=textHoehe}">
            Höhe:
        </Label>
        <TextBox Name="textHoehe" FontSize="20" Width="100" ToolTip="Höhenangabe">20</TextBox>

        <Button FontSize="20" Margin="5,0,0,0" Click="OnAusgabe" ToolTip="Ausgabe der Daten">
            <AccessText> Ausgabe...</AccessText>
        </Button>
    </StackPanel>
</Grid>
</Window>

```

Listing 41.18 Label-Elemente und Schaltflächen mit Kurztasten



Abbildung 41.17 Die dargestellten Kurztasten in der Benutzeroberfläche

Natürlich können Sie die Reihenfolge der Steuerelemente für das Drücken der **Esc**-Taste über die Eigenschaft `TabIndex` festlegen, welche in jedem Steuerelement vorhanden ist. Außerdem haben wir in Listing 41.18 die Eigenschaft `ToolTip` verwendet, um kleine Hilfstexte auf den Steuerelementen darzustellen, wenn Sie mit der Maus darüber verweilen.

Menüs

Viele Windows-Applikationen werden durch hierarchisch angeordnete Menüs gesteuert. Wir unterscheiden zwischen *Popup-Menüs*, die am oberen Rand eines Fensters angeordnet sind (Hauptmenü) und den *Kontext-Menüs*, die dann erscheinen, wenn Sie mit der linken Maustaste auf ein Steuerelement klicken. Ein Menü enthält im Allgemeinen mehrere `MenuItem`-Elemente.

Das folgende Beispiel (Listing 41.19) zeigt den Code für ein einfaches Hauptmenü eines noch einfacheren Textverarbeitungsprogramms. In diesem Menü wurden alle gängigen Befehle einer Anwendung dargestellt, jedoch hier nicht immer implementiert.

```

<Window x:Class="TestMenue.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Ein Menü" Height="300" Width="400"
    >

```

```
<Window.Resources>
    // Bild für Befehl "Neu"
    <Path x:Key="imgNeu" Fill="White" Stroke="Black" StrokeThickness="1"
        Data="M 3,3 L 3,15 14,15 14,3 Z M 10,3 L 14,7" />

    // Bild für Befehl "Beenden"
    <Path x:Key="imgBeenden" Fill="Red" Stroke="Black" StrokeThickness="1"
        Data="M 7,13 L 2,7 5,7 5,0 9,0 9,7 12,7 z" />
</Window.Resources>

<DockPanel>
    <Menu DockPanel.Dock="Top">
        <MenuItem Header="_Datei">
            <MenuItem Header="_Neu..." Command="New" InputGestureText="Strg+N" Icon="{StaticResource imgNeu}" />
            <Separator />
            <MenuItem Header="_Öffnen..." />
            <MenuItem Header="_Speichern" />
            <MenuItem Header="Speichern _unter..." />
            <Separator />
            <MenuItem Header="_Beenden" Click="OnBeenden" Icon="{StaticResource imgBeenden}" />
        </MenuItem>

        <MenuItem Header="Bearbeiten">
            <MenuItem Header="_Ausschneiden" Command="Cut" InputGestureText="Strg+X" />
            <MenuItem Header="_Kopieren" Command="Copy" InputGestureText="Strg+C" />
            <MenuItem Header="_Einfügen" Command="Paste" InputGestureText="Strg+V" />
            <Separator />
            <MenuItem Header="_Alles markieren" Command="SelectAll" InputGestureText="Strg+A" />
        </MenuItem>

        <MenuItem Header="Hilfe">
            <MenuItem Header="_Hilfe..." />
            <Separator />
            <MenuItem Header="_Über..." />
        </MenuItem>
    </Menu>
    <TextBox Name="text" AcceptsReturn="True" TextWrapping="Wrap" VerticalScrollBarVisibility="Auto" />
</DockPanel>
</Window>
```

Listing 41.19 Ein einfaches Hauptmenü

Damit das Hauptmenü an der oberen Fensterkante anliegt, benutzen wir als Layout-Element ein `DockPanel` und setzen die Eigenschaft `DockPanel.Dock` auf den Wert `Top`. Nun können wir `MenuItem`-Elemente in das Hauptmenü einfügen. Jeder Menüpunkt wird über mehrere wichtige Eigenschaften bestimmt. Zunächst wird die Eigenschaft `Header` benutzt, um den Text für den Menüpunkt festzulegen. In diesem Text können Sie wiederum ein Unterstrichzeichen benutzen, um die Kurztaste (in Kombination mit der `Alt`-Taste) für den Menüpunkt festzulegen.

Da ein Menüpunkt normalerweise einen Befehl in der Anwendung auslöst, müssen wir eine Verbindung mit den entsprechenden Ereignismethoden herstellen. Die kann entweder über die Eigenschaft `Command` mit einem `CommandBinding`-Objekt oder durch direktes Binden des `Click`-Ereignisses an die Ereignismethode realisiert werden. Im Beispiel wurden beide Methoden benutzt (Listing 41.20). Die benötigten `CommandBinding`-Objekte werden im Konstruktor-Code des Fensters implementiert. Die einzelnen Ereignismethoden, die an die Menübefehle gebunden sind, führen sehr einfachen Code aus und müssen nicht weiter erläutert werden.

Wenn Sie den Text für die Kurztaste des Menübefehls ändern wollen, benutzen Sie hierzu die Eigenschaft `InputGestureText`.

Horizontale Trennelemente zwischen zwei Menüpunkten werden durch ein `Separator`-Element dargestellt.

Oft möchten wir im linken Bereich des Menüpunktes ein kleines Bild (Icon) anzeigen, welches dann auch auf den Schaltflächen der Werkzeugleiste benutzt werden kann. Hier müssen wir etwas vorgreifen, denn wir benutzen »Ressourcen«, um diese Bilder zu definieren. Im Block `Window.Resources` deklarieren wir zwei Path-Elemente, denen wir über die Eigenschaft `Key` einen Namen zuweisen.

Nun können wir die definierten Grafiken über die Eigenschaft `Icon` des `MenuItem`-Elements zuweisen. Die Bilder sollten die Größe 17x17 Einheiten nicht überschreiten, ansonsten wird der Menüpunkt sehr hoch bzw. breit.

HINWEIS Sie können natürlich auch Pixel-Grafiken (BMP, JPG, o.ä.) über die `Icon`-Eigenschaft in einem Menüpunkt darstellen. Verwenden Sie dann ein `Image`-Element für die Eigenschaft `MenuItem.Icon`. Ggf. müssen Sie auch hier die Größe der Grafik mit einem `Viewbox`-Element anpassen.

Nach dem Hauptmenü deklarieren wir noch ein `TextBox`-Element mit dem Namen »text«, das die Textverarbeitungsmöglichkeit realisieren soll. Die Befehle *Einfügen*, *Kopieren*, *Ausschneiden* und *Alles markieren* werden auf diese `TextBox` angewendet, um eine möglichst einfache Implementierung zu erhalten.

Starten Sie nun die Anwendung, und »spielen« Sie etwas mit den Befehlen in den Menüs *Datei* und *Bearbeiten*. Beobachten Sie, wie die Menüpunkte durch die implementierten `CommandBinding`-Elemente ein- und ausgeschaltet werden. Benutzen Sie auch die möglichen Kurztasten, die wir für das Menü definiert haben. Die laufende Anwendung ist in Abbildung 41.18 zu sehen.

```
Imports System
Imports System.Windows
Imports System.Windows.Input

Namespace TestMenue
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
            text.Focus()

            ' Befehl: Neu
            Dim cbNeu As CommandBinding
            cbNeu = New CommandBinding(ApplicationCommands.[New])

```

```
AddHandler cbNeu.Executed, AddressOf OnNeu
AddHandler cbNeu.CanExecute, AddressOf OnNeuCanExecute
Me.CommandBindings.Add(cbNeu)

' Befehl: Ausschneiden
Dim cbAusschneiden As CommandBinding
cbAusschneiden = New CommandBinding(ApplicationCommands.Cut)
AddHandler cbAusschneiden.Executed, AddressOf OnAusschneiden
AddHandler cbAusschneiden.CanExecute, AddressOf OnAusschneidenCanExecute
Me.CommandBindings.Add(cbAusschneiden)

' Befehl: Einfügen
Dim cbEinfuegen As CommandBinding
cbEinfuegen = New CommandBinding(ApplicationCommands.Paste)
AddHandler cbEinfuegen.Executed, AddressOf OnEinfuegen
AddHandler cbEinfuegen.CanExecute, AddressOf OnEinfuegenCanExecute
Me.CommandBindings.Add(cbEinfuegen)

' Befehl: Kopieren
Dim cbKopieren As CommandBinding
cbKopieren = New CommandBinding(ApplicationCommands.Copy)
AddHandler cbKopieren.Executed, AddressOf OnKopieren
AddHandler cbKopieren.CanExecute, AddressOf OnAusschneidenCanExecute
Me.CommandBindings.Add(cbKopieren)
End Sub

Private Sub OnNeu(ByVal Sender As Object, ByVal e As RoutedEventArgs)
    ' Text löschen
    text.Clear()
End Sub

Private Sub OnNeuCanExecute(ByVal sender As Object, _
                           ByVal e As CanExecuteRoutedEventArgs)
    ' Neu-Befehl nur ausführen, wenn TextBox einen Text enthält
    e.CanExecute = (text.Text.Length > 0)
End Sub

Private Sub OnBeenden(ByVal sender As Object, _
                      ByVal e As RoutedEventArgs)
    'Fenster schließen
    Me.Close()
End Sub

Private Sub OnAusschneiden(ByVal sender As Object, _
                           ByVal e As RoutedEventArgs)
    ' Befehl: Ausschneiden
    text.Cut()
End Sub

Private Sub OnAusschneidenCanExecute(ByVal sender As Object, _
                                    ByVal e As CanExecuteRoutedEventArgs)
    e.CanExecute = (text.Text.Length > 0)
End Sub
```

```

Private Sub OnEinfuegen(ByVal sender As Object,
                      ByVal e As RoutedEventArgs)
    ' Befehl: Einfügen
    text.Paste()
End Sub

Private Sub OnEinfuegenCanExecute(ByVal sender As Object,
                                  ByVal e As CanExecuteRoutedEventArgs)
    ' Einfügen-Befehl nur ausführen, wenn Text in der Zwischenablage ist
    e.CanExecute = Clipboard.ContainsText
End Sub

Private Sub OnKopieren(ByVal sender As Object,
                      ByVal e As RoutedEventArgs)
    ' Befehl: Kopieren
    text.Copy()
End Sub

Private Sub OnAllesMarkieren(ByVal sender As Object,
                            ByVal e As RoutedEventArgs)
    'Befehl: Alles markieren
    text.SelectAll()
End Sub
End Class
End Namespace

```

Listing 41.20 Implementierung der Befehle im Hauptmenü



Abbildung 41.18 Hauptmenü mit Bildern und Kurztasten

Die Benutzung von kontextsensitiven Menüs funktioniert ganz ähnlich wie die Deklaration eines Hauptmenüs. Die gewünschten Menüpunkte werden in dem Steuerelement als `ContextMenu`-Element deklariert, in dem sie benutzt werden sollen. Listing 41.21 zeigt die Erweiterung des letzten Beispiels. Durch Anklicken der TextBox mit der rechten Maustaste erscheint das Menü mit den beiden Befehlen *Einfügen* und *Neu*.

```

...
Private Sub OnSuchenStart(ByVal sender As Object,
                         ByVal e As RoutedEventArgs)
    Dim iPos As Integer
    iPos = text.SelectionStart + text.SelectionLength
    Dim strText As String
    strText = text.Text
    iPos = strText.IndexOf(textSuchen.Text, iPos)
    If iPos >= 0 Then
        text.Select(iPos, textSuchen.Text.Length)
    Else
        text.Select(0, 0)
    End If
End Sub
...

```

Listing 41.21 Erweiterter XAML-Code für ein kontextsensitives Menü im TextBox-Element

Der implementierte Code bleibt unverändert, da die Menüpunkte aus dem Kontextmenü über CommandBinding-Elemente an die entsprechenden Ereignismethoden gebunden sind. Auch die Aktivierung und Deaktivierung der Menüpunkte funktioniert durch diese Bindung wie erwartet. Natürlich können Sie auch in einem Kontextmenü kleine Bildchen auf der linken Seite darstellen, indem Sie die Icon-Eigenschaft entsprechend setzen.

Werkzeugeisten (Toolbars)

Ein ebenfalls häufig verwendetes Element in Benutzerschnittstellen von Windows-Anwendungen ist die Werkzeugeiste (Toolbar). Die WPF-Werkzeugeiste kann nicht nur Schaltflächen, sondern auch andere Steuerelemente, wie TextBox- oder ComboBox-Elemente, enthalten. Wir wollen das letzte Beispiel mit dem Menü nun mit einer Werkzeugeiste erweitern. Dazu implementieren wir den folgenden XAML-Code zur Definition von Menu- und TextBox-Element aus dem vorherigen Beispiel (Listing 41.22):

```

...
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        <Button Command="New" Content="{StaticResource imgNeu}" />
        <Separator />
        <Label> Suchen:</Label>
        <TextBox Name="textSuchen" Width="100" />
        <Button Click="OnSuchenStart">
            <AccessText>_Start</AccessText>
        </Button>
        <Separator />
        <Button Click="OnBeenden" Content="{StaticResource imgBeenden}" />
    </ToolBar>
</ToolBarTray>
...

```

Listing 41.22 Die Anwendung mit einer Werkzeugeiste

Wir beginnen mit einem `ToolBarTray`-Element, welches das Layout der gesamten Werkzeugeiste kontrolliert. Damit die Werkzeugeiste direkt unter dem Hauptmenü erscheint, setzen wir die Eigenschaft `DockPanel.Dock` wieder auf den Wert `Top`. Im Layout-Element platzieren wir jetzt ein `ToolBar`-Element, in dem dann die verschiedenen Steuerelemente der Werkzeugeiste positioniert werden können.

Zunächst benötigen wir eine Schaltfläche für den *Neu*-Befehl und verwenden das bereits definierte Bildchen aus den vorhandenen Windows-Ressourcen. Danach kommt eine senkrechte Trennlinie (Separator).

Wir wollen den Text in unserer `TextBox` nach einem bestimmten Text durchsuchen können. Dazu erzeugen wir im `ToolBar`-Element nun erst ein `Label`-Element mit dem Text *Suchen*, dann eine `TextBox` mit einer festen Breite und schließlich eine Schaltfläche mit der Aufschrift *Start*. Diese Schaltfläche ist mit der Ereignismethode `OnSuchenStart` (Listing 41.23) verbunden. Nach einer weiteren Trennlinie wird noch eine Schaltfläche deklariert, mit der das Programm beendet werden kann.

Auch für die Werkzeugeiste können wir die bereits existierenden `CommandBinding`-Elemente oder Ereignismethoden verwenden.

Wir müssen also nur die Ereignismethode `OnSuchenStart` implementieren (Listing 41.23). Damit wir ein mehrfaches Auftreten des eingegebenen Suchtextes ermitteln können, starten wir die Suche am Ende des selektierten Bereichs (Variable `iPos`) im `TextBox`-Element mit dem Namen »`text`«. Wenn kein Text selektiert ist, entspricht dies der Cursor-Position in der `TextBox`. Eine Referenz auf den Text wird aus dieser `TextBox` in eine `String`-Variable übernommen und mit der Methode `IndexOf` nach dem Suchtext aus der `TextBox` der Werkzeugeiste durchsucht. Das Ergebnis der Suchmethode `IndexOf` wird wieder als Startposition für die nächste Suche verwendet. Somit kann der Suchtext auch mehrfach vorkommen. Die einzelnen Positionen werden dann nacheinander selektiert. Wenn der Suchtext nicht gefunden wird, wird der Cursor in `TextBox` »`text`« an Position 0 gesetzt (Abbildung 41.19).

```
...
Private Sub OnSuchenStart(ByVal sender As Object,
                         ByVal e As RoutedEventArgs)
    Dim iPos As Integer
    iPos = text.SelectionStart + text.SelectionLength
    Dim strText As String
    strText = text.Text
    iPos = strText.IndexOf(textSuchen.Text, iPos)
    If iPos >= 0 Then
        text.Select(iPos, textSuchen.Text.Length)
    Else
        text.Select(0, 0)
    End If
End Sub
...
```

Listing 41.23 Der Zusatzcode für den Suchen-Befehl

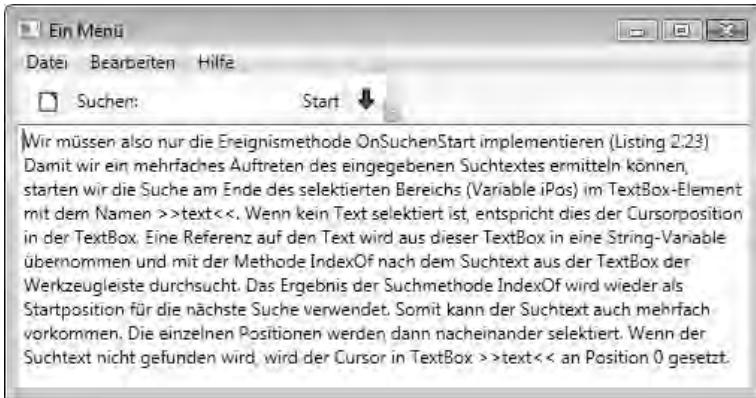


Abbildung 41.19 Der Text-Editor mit einer Werkzeugleiste

Mehrere Werkzeugeisten können nacheinander in einem `ToolBarTray`-Element deklariert werden (Abbildung 41.20):

```
<ToolBarTray DockPanel.Dock="Top">
  <ToolBar>
    <!-- Die erste Toolbar -->
  </ToolBar>
  <!-- Die zweite Toolbar -->
  <ToolBar>
    <Button Command="Cut">Ausschneiden</Button>
    <Button Command="Copy">Kopieren</Button>
    <Button Command="Paste">Einfügen</Button>
  </ToolBar>
</ToolBarTray>
```

Die einzelnen Werkzeugeisten können Sie nebeneinander oder untereinander anordnen. Eine Verschiebung an den rechten, linken oder unteren Rand des Fensters ist in dieser Anwendung jedoch nicht möglich.



Abbildung 41.20 Zwei Werkzeugeisten in der Anwendung

Zusammenfassung

Steuerelemente sind in jedem Framework für Benutzerschnittstellen vorhanden, so auch in Windows Presentation Foundation. Auch hier werden Steuerelemente über Klassen implementiert, die über Eigenschaften und Methoden verfügen. Diese nutzt der Programmierer, um die Steuerelemente zu kontrollieren.

Wichtig für Steuerelemente sind die Konzepte *Routed Events* (weitergeleitete Ereignisse) und *Routed Commands* (weitergeleitete Befehle). Diese Konzepte erleichtern die Programmierung von hierarchischen Benutzerschnittstellen erheblich. Auch die Weitergabe von Eigenschaften stellt eine große Erleichterung für den Softwareentwickler dar.

Denken Sie auch daran, dass WPF-Steuerelemente eine Trennung von Design und Logik durchführen, sodass das Aussehen von Steuerelementen sehr leicht geändert werden kann. Diese Möglichkeit betrachten wir in einem späteren Kapitel noch genauer.

Die einzelnen Steuerelemente von WPF wurden hier nicht im Detail durchgesprochen. Das würde den Rahmen dieses Buchs leider sprengen.

Kapitel 42

Layout

In diesem Kapitel:

Das StackPanel	1254
Das DockPanel	1256
Das Grid	1260
Das GridSplitter-Element	1266
Das UniformGrid	1269
Das Canvas-Element	1270
Das Viewbox-Element	1271
Text-Layout	1273
Das WrapPanel	1277
Standard-Layout-Eigenschaften	1278
Zusammenfassung	1281

Layout ist ein wichtiges und sehr umfangreiches Feature von Windows Presentation Foundation (WPF). Ohne die Möglichkeiten des Layouts wäre es nur sehr schwer möglich, Applikationen mit Dialogfeldern oder Fenstern, die in der Größe änderbar sind, zu programmieren. In herkömmlichen Windows-Frameworks (z.B. MFC, Windows Forms) werden die einzelnen Steuerelemente eines Dialogfeldes oft mit fest vorgegebenen Positionen und Größen angelegt. Dieser Weg wird in WPF selten verwendet, da es die Möglichkeit gibt, alle Steuerelemente eines Fensters mithilfe von Layout-Steuerelementen so zu verwalten, dass eine variable Position und Größe ohne wesentlichen Programmieraufwand erzielbar sind.

In Windows Presentation Foundation wird das Layout der Steuerelemente in einem Fenster über so genannte Panels gesteuert. Je nach Art des verwendeten Panels werden die enthaltenen Steuerelemente angeordnet. Die Position des jeweiligen Elements wird durch das Panel bestimmt, welches das Element enthält. Einige Panel-Elemente verwalten auch die Größe ihrer Kindelemente. In WPF stehen diverse Steuerelemente zur Verfügung, die das Layout beeinflussen (Tabelle 42.1):

Panel	Anwendung
Canvas	Einfaches Element für die genaue Positionierung von Kindelementen.
StackPanel	Ordnet die Kindelemente vertikal oder horizontal an.
DockPanel	Ordnet die Kindelemente am Rand des Elternelements an.
TabPanel	Ordnet die Kindelemente als einzelne umschaltbare Seiten an.
WrapPanel	Ordnet die Kindelemente in einer Zeile an. Wenn der Platz in der Zeile nicht reicht, findet ein Zeilenumbruch statt.
Viewbox	Darstellung von Grafiken.
Grid	Ordnet die Kindelemente in einer Tabelle mit Zeilen und Spalten an.

Tabelle 42.1 Die vorhandenen Panel-Typen

Wir wollen die Möglichkeiten dieser Layout-Steuerelemente nun im Einzelnen betrachten.

BEGLEITDATEIEN

Die Begleitdateien zu folgenden Beispielen finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\H - WPF

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Das StackPanel

Das Panel, welches am einfachsten zu benutzen ist, heißt StackPanel. Mit einem StackPanel können Sie die Kindelemente entweder nebeneinander oder untereinander anordnen. Bei einer vertikalen Anordnung der Elemente im StackPanel werden die Standardhöhen der Kindelemente für die Positionierung verwendet. Dies bedeutet, dass die Höhe eines Elements aus der Höhe des Inhalts bestimmt wird, es sei denn, Sie geben explizit einen Wert für die Höhe des Elements an. In einem horizontalen StackPanel werden die Standardbreiten der Kindelemente angewendet. Somit ergibt sich die Breite aus der Breite des Inhaltselements. Natürlich können Sie auch hier die Breite selbst angeben. In diesem Fall werden die Eigenschaften innerhalb

der Hierarchie nicht weitergegeben. Die Kindelemente, die in einem Layout-Panel angelegt werden, müssen nicht vom gleichen Typ sein. Sie können also unterschiedliche Steuerelemente im Panel beliebig mischen. Im folgenden Beispiel werden verschiedene Steuerelemente (TextBlock, Button, CheckBox,...) untereinander angeordnet.

```
<Window x:Class="StackPanel1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Vertikales StackPanel" Height="300" Width="300"
    >
<StackPanel>
    <TextBlock>Hier steht ein Text!</TextBlock>
    <Button>Die erste Schaltfläche</Button>
    <CheckBox>Wählen Sie hier aus</CheckBox>
    <Button>Klicken Sie hier!</Button>
    <TextBlock>Auch hier steht was...</TextBlock>
    <TextBox>Eingabe hier!</TextBox>
</StackPanel>
</Window>
```

Listing 42.1 Ein StackPanel mit vertikaler Anordnung der Elemente

Das StackPanel in Listing 42.1 ordnet die Steuerelemente folgendermaßen an:

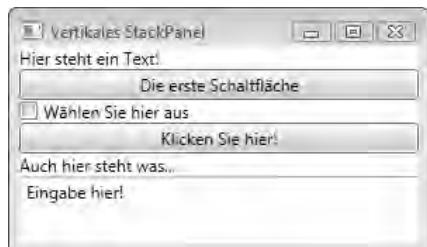


Abbildung 42.1 Vertikale Anordnung im StackPanel

Im Beispiel aus Listing 42.1 müssen wir beachten, dass das StackPanel die Breite und Höhe des Anwendungsfensters annimmt. Weiterhin nehmen die Kindelemente im StackPanel wiederum genau diese Breite an. Für die jeweilige Höhe der einzelnen Elemente wird die Standardhöhe verwendet, die sich aus der Höhe des Inhalts eines Kindelementes ergibt. Die Standardhöhe kann für die verschiedenen Elemente durchaus einen unterschiedlichen Wert annehmen.

Wird nun das Fenster in der Breite verändert, dann passt sich das StackPanel an die neue Breite an. Die Kindelemente des StackPanel passen sich dann ebenfalls an diese Breite an. Verkleinern Sie dagegen die Höhe des Fensters, so werden die Kindelemente des StackPanel ebenfalls verkleinert und irgendwann einfach abgeschnitten bzw. nicht mehr dargestellt.

Mit einer kleinen Änderung im Listing 42.1 können Sie das StackPanel dazu bewegen, seine Kindelemente in horizontaler Reihenfolge anzuordnen:

```
<!-- Wie in Listing 1.1 -->
<StackPanel Orientation="Horizontal">
<!-- Weiter wie in Listing 1.1 -->
```

Listing 42.2 Horizontale Anordnung im StackPanel

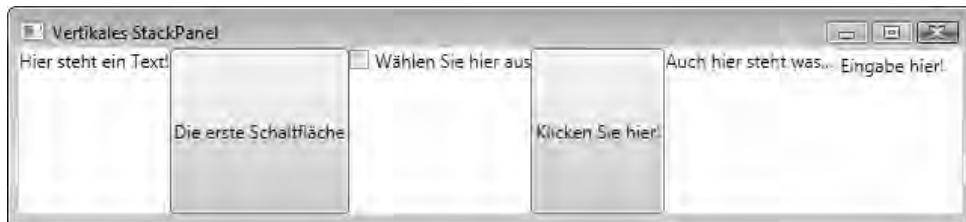


Abbildung 42.2 Ein horizontales StackPanel

In Abbildung 42.2 ist deutlich zu sehen, dass die Höhen der Kindelemente nun durch die Höhe des StackPanel (welches das Fenster vollständig ausfüllt) bestimmt werden. Die Breite der einzelnen Elemente ist durch ihre Standardbreite gegeben. Sie können gut erkennen, dass sich diese Breite aus der Breite des Inhalts des Elements ergibt. So ist die erste Schaltfläche ganz links etwas breiter als die zweite Schaltfläche, da die Texte unterschiedliche Breiten haben. Wird die Breite des Fensters und damit des StackPanel verkleinert, werden die Elemente rechts nach und nach abgeschnitten und nicht mehr dargestellt. Dieses Verhalten können Sie jedoch durch eine explizite Angabe von Breiten bzw. Höhen für die Kindelemente beeinflussen.

Das DockPanel

Ein DockPanel-Element ermöglicht es, das Gesamt-Layout eines Teils der Benutzerschnittstelle zu definieren. Sie können angeben, an welchem Rand (oben, unten, rechts oder links) ein Kindelement angeordnet werden soll. Werden mehrere Elemente am gleichen Rand angelegt, so werden diese Elemente neben- oder untereinander, ähnlich wie in einem StackPanel, angeordnet.

HINWEIS Wenn nicht anders angegeben, werden beim Andocken an das Elternelement die jeweiligen Standardbreiten und -höhen verwendet. Sie können natürlich explizit Breiten und Höhen für die Kindelemente angeben, die dann auch entsprechend verwendet werden.

```
<Window x:Class="DockPanel1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DockPanel #1" Height="300" Width="300"
    >
<DockPanel>
    <Button DockPanel.Dock="Top">Oben</Button>
    <Button DockPanel.Dock="Bottom">Unten</Button>
    <Button DockPanel.Dock="Left">Links</Button>
    <Button DockPanel.Dock="Right">Rechts</Button>
</DockPanel>
</Window>
```

Listing 42.3 Einfaches Docking mit dem DockPanel

Wenn Sie das Beispiel aus Listing 42.3 ausführen, werden Sie sehen, dass die Schaltfläche, die zuletzt erzeugt wird (Schaltfläche mit dem Inhalt *Rechts*), den Bereich des DockPanel vollständig auffüllt.

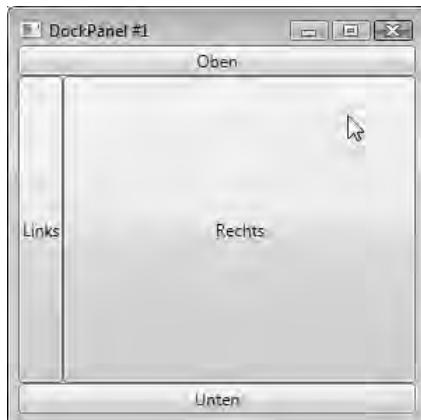


Abbildung 42.3 Das DockPanel aus Listing 42.3

Für jede Schaltfläche wird angegeben, an welchem Rand des DockPanel sie angedockt werden soll. Hierzu wird die spezielle Eigenschaft Dock aus dem DockPanel-Element verwendet:

```
DockPanel.Dock="Right"
```

Im Beispiel aus Listing 42.1 werden die Schaltflächen, die oben und unten angedockt werden, in der Breite des DockPanel dargestellt. Die beiden Schaltflächen rechts und links werden in die verbleibende Höhe eingepasst. Die linke Schaltfläche hat die Standardbreite und die rechte Schaltfläche füllt schließlich den verbleibenden Rest des DockPanel-Elements auf.

HINWEIS Wenn Sie die Größe des Fensters aus Listing 42.1 ändern, bleibt die Anordnung der Schaltflächen im DockPanel erhalten. Nur die jeweiligen Breiten und Höhen, die nicht fest vorgegeben wurden, werden entsprechend der Fenstergröße neu berechnet und dargestellt.

Wenn Sie das Auffüllen durch das letzte Kindelement verhindern, müssen Sie das Attribut LastChildFill auf False setzen (Listing 42.4). Die letzte Schaltfläche, die im DockPanel dargestellt wird, erhält dann einfach die Standardbreite. Ein Teil des DockPanel-Elements bleibt dann frei.

```
<Window x:Class="DockPanel12.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DockPanel #2" Height="300" Width="300"
    >
<DockPanel LastChildFill="False">
    <Button DockPanel.Dock="Top">Oben</Button>
    <Button DockPanel.Dock="Bottom">Unten</Button>
    <Button DockPanel.Dock="Left">Links</Button>
    <Button DockPanel.Dock="Right">Rechts</Button>
</DockPanel>
</Window>
```

Listing 42.4 DockPanel ohne »Auffüllen«

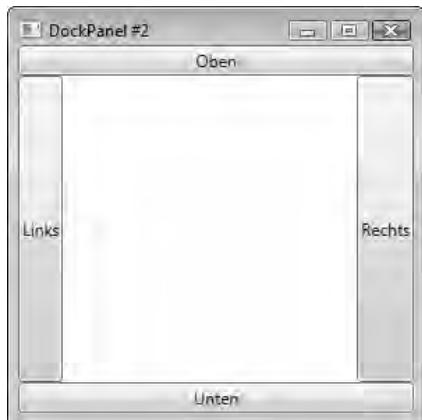


Abbildung 42.4 Ein nicht vollständig gefülltes DockPanel

Im nächsten Beispiel soll die Schaltfläche *Links* nun nicht mehr die gesamte Höhe des DockPanel-Elements ausfüllen. In diesem Fall (Listing 42.5) müssen wir die Höhe für diese Schaltfläche explizit angeben. Die Schaltfläche wird dann in der Mitte des zur Verfügung stehenden Bereiches dargestellt.

```
<Window x:Class="DockPanel13.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DockPanel #3" Height="300" Width="300"
    >
    <DockPanel LastChildFill="False">
        <Button DockPanel.Dock="Top">Oben</Button>
        <Button DockPanel.Dock="Bottom">Unten</Button>
        <Button DockPanel.Dock="Left" Height="22">Links</Button>
        <Button DockPanel.Dock="Right">Rechts</Button>
    </DockPanel>
</Window>
```

Listing 42.5 Eine Schaltfläche mit fest definierter Höhe im DockPanel

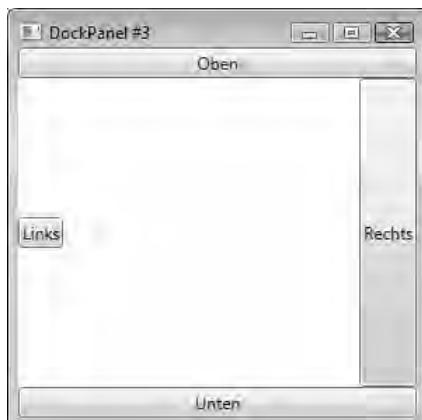


Abbildung 42.5 Die Schaltfläche steht in der Mitte des Bereiches, der im DockPanel zur Verfügung steht

Um die Position der Schaltfläche zu modifizieren, können wir die Attribute `HorizontalAlignment` und `VerticalAlignment` entsprechend einsetzen. Im folgenden Code-Beispiel soll die Schaltfläche mit dem Text *Links* nun weiter unten, aber direkt oberhalb der Schaltfläche mit dem Text *Unten* angeordnet werden (Listing 42.6 und Abbildung 42.6).

```
<Window x:Class="DockPanel14.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DockPanel #4" Height="300" Width="300"
>
<DockPanel LastChildFill="False">
    <Button DockPanel.Dock="Top">Oben</Button>
    <Button DockPanel.Dock="Bottom">Unten</Button>
    <Button DockPanel.Dock="Left" Height="22" VerticalAlignment="Bottom">Links</Button>
    <Button DockPanel.Dock="Right">Rechts</Button>
</DockPanel>
</Window>
```

Listing 42.6 Die Schaltfläche ist jetzt unten angeordnet

Sie erkennen sicherlich schon, wie leistungsfähig selbst einfache Layout-Elemente sind, wenn die Darstellung der Kindelemente über die verschiedenen Attribute angepasst wird.

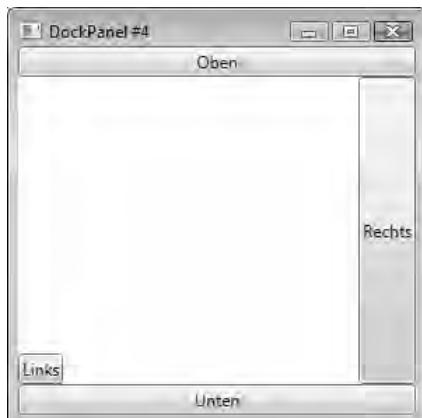


Abbildung 42.6 Die kleine Schaltfläche ist unten links angeordnet

Die Darstellung der Kindelemente im DockPanel ist natürlich von der Reihenfolge abhängig, in der die Elemente erzeugt werden. Gehen Sie vom Beispiel in Listing 42.4 aus. Nun sollen zuerst die beiden Schaltflächen auf der rechten und der linken Seite erzeugt werden. Wir erhalten dann den in Abbildung 42.5 gezeigten neuen Aufbau des Fensters.

```
<Window x:Class="DockPanel5.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DockPanel #5" Height="300" Width="300"
>
<DockPanel LastChildFill="False">
    <Button DockPanel.Dock="Left">Links</Button>
```

```

<Button DockPanel.Dock="Right">Rechts</Button>
<Button DockPanel.Dock="Top">Oben</Button>
<Button DockPanel.Dock="Bottom">Unten</Button>
</DockPanel>
</Window>

```

Listing 42.7 Die Reihenfolge der Definition ist entscheidend für den Aufbau

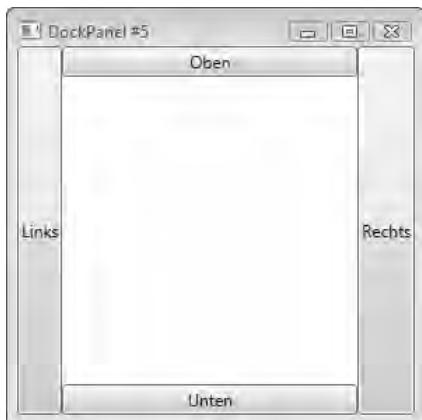


Abbildung 42.7 Zuerst werden die beiden Schaltflächen rechts und links erzeugt

HINWEIS Die Elemente in einem DockPanel überlappen sich niemals. Wenn einzelne Elemente in der geforderten Größe nicht dargestellt werden können, wird dieser Teil einfach abgeschnitten.

Die Kindelemente im DockPanel entscheiden teilweise selbst, in welcher Größe sie sich darstellen müssen. Wenn eine Schaltfläche oben oder unten im DockPanel platziert wird, so entscheidet das DockPanel über die Breite der Schaltfläche oder Sie müssen die Breite des Kindelements extra angeben. Das Kindelement, im Beispiel die Schaltfläche, entscheidet aber ggf. selbst, wie hoch das Element sein soll. Wird die Schaltfläche rechts oder links angedockt, so verhält es sich umgekehrt. Die Höhe wird vom DockPanel bestimmt und die Breite wird vom Kindelement vorgegeben. Hierbei spielt dann der Text auf der Schaltfläche eine entscheidende Rolle. Wenn möglich, wird die Breite so gesetzt, dass der gesamte Text auf der Schaltfläche sichtbar ist. Wenn dies nicht möglich ist, wird ein Teil des Kindelements abgeschnitten.

Das Grid

Das Grid ist ein sehr mächtiges Layout-Steuerelement. Es bietet die Möglichkeit, die Kindelemente in einer oder mehreren Zeilen und Spalten anzurichten. Sie werden sehen, dass die Höhe der Zeilen und die Breite der Spalten im Grid sehr leicht konfigurierbar sind.

Das einfachste Grid-Element enthält nur eine Zelle, also eine Zeile und eine Spalte. In dieser Grid-Zelle kann ein Kindelement platziert werden. Dieses Element wird standardmäßig in der Mitte der Zelle dargestellt (Listing 42.8).

```
<Grid>
  <Button Width="100" Height="30">Test</Button>
</Grid>
```

Listing 42.8 Das einfachste Grid mit einer Zelle

Um nun mehrere Spalten und Zeilen im Grid darzustellen, müssen diese in den Bereichen `Grid.ColumnDefinitions` bzw. in den `Grid.RowDefinitions` angelegt werden. Im Beispiel aus Listing 42.9 wird ein Grid mit zwei Spalten und drei Zeilen erzeugt. In den einzelnen Zellen werden `TextBlock`-Elemente für die Ausgabe von Texten benutzt. Beachten Sie hierbei, dass die `TextBlock`-Elemente links oben mit der Ausgabe von Texten beginnen. Diese Standardeinstellung können Sie über die Attribute `HorizontalAlignment` und `VerticalAlignment` des `TextBlock`-Elements ändern. In diesem Beispiel wollen wir mithilfe des `Grid`-Attributs `ShowGridLines="True"` die Trennlinien zwischen den Zellen im Grid darstellen (Abbildung 42.8). Für jedes `TextBlock`-Element wird über die Attribute `Grid.Column` und `Grid.Row` (aus der `Grid`-Klasse) festgelegt, in welcher Zeile und Spalte es dargestellt werden soll.

```
<Window x:Class="Grid.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Grid #1"
  Height="300" Width="300">

  <Grid ShowGridLines="True">
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>

    <TextBlock Grid.Column="0" Grid.Row="0">Name:</TextBlock>
    <TextBlock Grid.Column="1" Grid.Row="0">Vorname:</TextBlock>
    <TextBlock Grid.Column="0" Grid.Row="1">Heckhuis</TextBlock>
    <TextBlock Grid.Column="1" Grid.Row="1">Jürgen</TextBlock>
    <TextBlock Grid.Column="0" Grid.Row="2">Löffelmann</TextBlock>
    <TextBlock Grid.Column="1" Grid.Row="2">Klaus</TextBlock>
  </Grid>
</Window>
```

Listing 42.9 Ein Grid mit mehreren Spalten und Zeilen

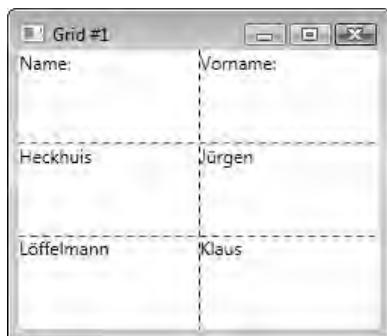


Abbildung 42.8 Ein Grid mit mehreren Spalten und Zeilen

Beachten Sie bitte auch das Verhalten des Grid-Elements beim Ändern der Größe des Fensters. Das Grid füllt in diesem Beispiel immer das gesamte Fenster aus. Zeilen und Spalten werden entsprechend in der Höhe und der Breite angepasst, so dass die Höhen- und Breitenverhältnisse im Grid-Element erhalten bleiben.

Nun ist es im Normalfall nicht damit getan, einfach nur einige Zeilen und Spalten zu definieren. Im nächsten Beispiel werden darum die Breiten und Höhen der Spalten und Zeilen im Grid-Element konfiguriert. Dies wird ebenfalls in den jeweiligen RowDefinition- und ColumnDefinition-Elementen durchgeführt.

Es gibt drei Möglichkeiten, die Höhe und Breite der Zellen zu definieren. Um Spalten (und Zeilen) mit einer festen Breite (Höhe) zu erzeugen, wird im ColumnDefinition-Element (RowDefinition) einfach die gewünschte Pixelanzahl angegeben:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="35" />
</Grid.ColumnDefinitions>
```

Wird in einer ColumnDefinition der Wert Auto angegeben, so wird die Breite der Spalte aus der maximalen Breite der Kindelemente bestimmt, die in der Spalte enthalten sind:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="Auto" />
</Grid.ColumnDefinitions>
```

Die dritte Variante der Breitenangabe in einer ColumnDefinition erstellt Spalten, deren Breite sich am vorhandenen Platz im Elternfenster orientiert. In diesem Fall wird als Parameter ein »*« verwendet:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>
```

Im letzten Beispiel wird ein Grid mit einer Spalte erzeugt, welche die gesamte Breite des Elternelements bekommt. Wird das Elternelement in der Breite verändert, so wird die Spaltenbreite entsprechend angepasst. Das ist aber noch nicht alles. Betrachten Sie folgendes Beispiel:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="*" />
  <ColumnDefinition Width="2*" />
  <ColumnDefinition Width="4*" />
  <ColumnDefinition Width="0.5*" />
<Grid.ColumnDefinitions>
```

Hier erhalten Sie ein Grid mit vier Spalten, die sich an die Breite des Elternelements anpassen. Das besondere ist jedoch, dass die Breiten der einzelnen Spalten in dem Verhältnis angelegt werden, die in den Width-Attributen angegeben sind. Die Summe aller Breiten ist »7.5*« und entspricht der Gesamtbreite des Grid-Elements. Wenn die Gesamtbreite im Grid nun zum Beispiel 300 Pixel beträgt, dann haben die einzelnen Spalten folgende Breiten:

Spalte-Nr.	Breitenangabe	Breite in Pixel
1	*	40
2	2*	80
3	4*	160
4	0.5*	20
Summe:	7.5*	300

Wird die Breite dieses Grid-Elements geändert, z.B. durch eine Änderung der Fenstergröße, so bleibt das Breitenverhältnis der einzelnen Grid-Spalten jedoch erhalten.

Alles was hier über die Definition von Spaltenbreiten geschrieben wurde, gilt ebenfalls für die Definition von Zeilenhöhen im Grid. Das entsprechende Definitionselement heißt dann RowDefinition. Auch hier ist es möglich, feste, automatisch angepasste und im Verhältnis angepasste Höhen für eine Zeile anzugeben. In Listing 42.10 wird ein vollständiges Beispiel mit einem konfigurierten Grid gezeigt. Interessant ist hier die Breitenermittlung für die erste Spalte ganz links. Für diese Spalte wird die Einstellung Auto verwendet. Somit wird die Breite des größten Elements dieser Spalte benutzt. Hier ist das die Breite der Schaltfläche in der zweiten Zeile.

```
<Window x:Class="Grid2.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Grid #2" Height="300" Width="300"
  >
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
    <RowDefinition Height="2*" />
    <RowDefinition Height="50" />
    <RowDefinition />
  </Grid.RowDefinitions>
```

```

<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="0.5*" />
    <ColumnDefinition Width="50" />
    <ColumnDefinition />
</Grid.ColumnDefinitions>

<Button Grid.Column="0" Grid.Row="0">Test</Button>
<Button Grid.Column="0" Grid.Row="1">Ein großer Button</Button>
<TextBlock Grid.Column="0" Grid.Row="2">Auto</TextBlock>
<TextBlock Grid.Column="1" Grid.Row="2">*</TextBlock>
<TextBlock Grid.Column="2" Grid.Row="2">0.5*</TextBlock>
<TextBlock Grid.Column="3" Grid.Row="2">Fest 50</TextBlock>
<TextBlock Grid.Column="4" Grid.Row="2">
    Keine<LineBreak />Angabe
</TextBlock>
</Grid>
</Window>

```

Listing 42.10 Ein Grid-Element mit diversen Breiten- und Höhendefinitionen

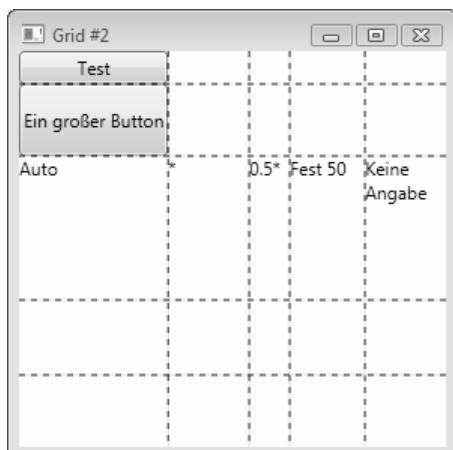


Abbildung 42.9 Das Grid aus Listing 42.10

Oft ist es erforderlich, dass ein Element im Grid mehrere Zeilen oder Spalten belegt. Hierfür stehen in der Grid-Klasse die beiden Eigenschaften `Grid.RowSpan` und `Grid.ColumnSpan` zur Verfügung. Mit diesen beiden Eigenschaften ist es sehr leicht möglich, ein Grid-Element zur beliebigen Positionierung von anderen Elementen zu benutzen und über mehrere Zellen zu verteilen. Die linken Ränder der Grid-Spalten haben dann in etwa das Verhalten von definierten Tabulatorpositionen.

Das Beispiel in Listing 42.11 zeigt die Positionierung von mehreren `TextBlock`-Elementen in einem Grid. Für die einzelnen `TextBlock`-Elemente werden an verschiedenen Stellen die Attribute `Grid.ColumnSpan` und `Grid.RowSpan` verwendet, damit die längeren Textinhalte auch vollständig ausgegeben werden. Abbildung 42.10 zeigt die Ausgabe von Listing 42.11 in einem Fenster. Die Attribute `Grid.RowSpan` und `Grid.ColumnSpan` können natürlich bei beliebigen Kindelementen angewendet werden.

HINWEIS Beachten Sie, dass bei einem TextBlock-Element der Text nur dann über mehrere Zeilen (Grid.RowSpan) verteilt wird, wenn das Attribut TextWrapping="Wrap" benutzt wird.

```
<Window x:Class="Grid3.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Grid #3" Height="300" Width="300"
    >
    <Grid ShowGridLines="true">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="30" />
            <RowDefinition Height="30" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="20" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <TextBlock Grid.Column="0" Grid.Row="0">Name:</TextBlock>
        <TextBlock Grid.Column="0" Grid.Row="1">Vorname:</TextBlock>
        <TextBlock Grid.Column="0" Grid.Row="2">PLZ und Ort:</TextBlock>
        <TextBlock Grid.Column="0" Grid.Row="3">Strasse und Nr.:</TextBlock>
        <TextBlock Grid.Column="0" Grid.Row="4">Telefon:</TextBlock>
        <TextBlock Grid.Column="0" Grid.Row="5">Angaben:</TextBlock>

        <TextBlock Grid.Column="1" Grid.Row="0" Grid.ColumnSpan="5">Heckhuis</TextBlock>
        <TextBlock Grid.Column="2" Grid.Row="1" Grid.ColumnSpan="5">Jürgen</TextBlock>
        <TextBlock Grid.Column="3" Grid.Row="2" Grid.ColumnSpan="5">12345 Wpfstadt</TextBlock>
        <TextBlock Grid.Column="4" Grid.Row="3" Grid.ColumnSpan="5">Avalonstrasse 22</TextBlock>
        <TextBlock Grid.Column="5" Grid.Row="4">01234-567890</TextBlock>
        <TextBlock Grid.Column="1" Grid.Row="5" Grid.ColumnSpan="4" Grid.RowSpan="3" TextWrapping="Wrap">
            Hier können weitere Angaben zur Person eingetragen werden. Diese Angaben sind natürlich freiwillig.
        </TextBlock>
    </Grid>
</Window>
```

Listing 42.11 Ein Grid zur Positionierung von Texten, die mehrere Spalten und Zeilen überspannen

WICHTIG Bedenken Sie, dass die vorgestellten Layout-Elemente in einer beliebigen Hierarchie verwendet werden können. Es ist möglich, in der Zelle eines komplexen Grid-Elementes ein DockPanel oder StackPanel zu platzieren. Ebenso kann man in eine Grid-Zelle ein weiteres Grid einfügen und konfigurieren.

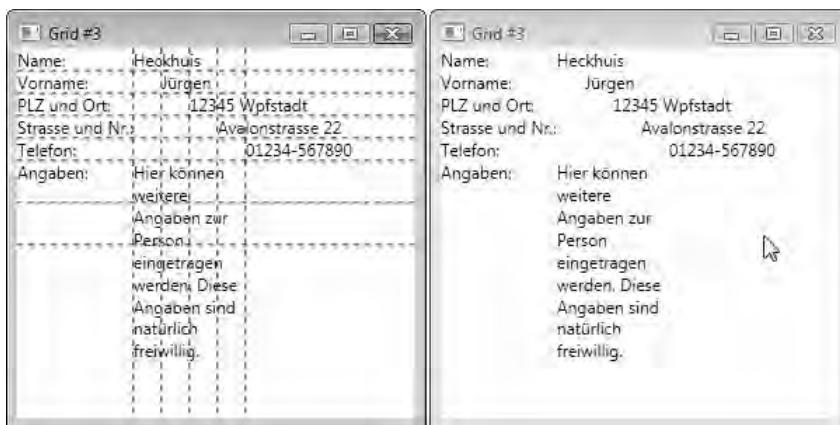


Abbildung 42.10 Das Grid aus Listing 42.11 – mit und ohne Trennlinien

Das GridSplitter-Element

Beim Einsatz eines Grid-Elementes taucht natürlich sofort die Frage auf, ob die Breite der Spalten bzw. die Höhe der Zeilen zur Laufzeit einfach änderbar sind. Um das zu ermöglichen, können Sie ein oder mehrere GridSplitter-Elemente einsetzen. Eine einfache Anwendung des GridSplitter-Elements zeigt Listing 42.12:

```
<Window x:Class="GridSplitter.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="GridSplitter" Height="120" Width="300"
>
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Button Grid.Column="0">Button Nr. 1</Button>
    <Button Grid.Column="1">Button Nr. 2</Button>
    <GridSplitter Grid.Column="1" Width="5" Background="Blue" HorizontalAlignment="Left"/>
</Grid>
</Window>
```

Listing 42.12 Grid mit einem GridSplitter-Element

Der Bereich des GridSplitter-Elements ist in Abbildung 42.11 gut erkennbar in blau sichtbar. Wenn Sie die Maus über dieses Element bewegen, ändert sich auch der Mauszeiger entsprechend. Bei einer Verschiebung des GridSplitter-Elements wird in diesem Fall eine Schaltfläche größer und die andere kleiner. Im Beispiel haben wir ein vertikales Trennelement, welches nach der Grid-Spalte mit dem Index »0« eingefügt werden soll.

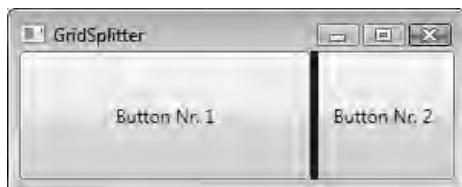


Abbildung 42.11 Das Grid mit nach rechts verschobenem GridSplitter-Element

Welcher Wert für die Eigenschaft `Grid.Column` angegeben werden muss, hängt allerdings auch noch von der Eigenschaft `HorizontalAlignment` des GridSplitter-Elements ab. Der Standardwert für diese Eigenschaft ist `Right`. In diesem Fall wird `Grid.Column` auf den Wert 0 gesetzt, wie in Listing 42.12 gezeigt wurde. Das GridSplitter-Element ist dann ein Teil der linken Grid-Zelle. Wird die Eigenschaft `HorizontalAlignment` auf den Wert `Left` eingestellt, dann ist das Trennelement ein Teil der linken Grid-Zelle und die Eigenschaft `Grid.Column` muss auf den Wert 1 gesetzt werden, damit die Trennung zwischen den beiden Schaltflächen erfolgt.

```
<GridSplitter Grid.Column="1" Width="5" Background="Blue" HorizontalAlignment="Left"/>
```

HINWEIS Beachten Sie bitte, dass ein GridSplitter-Element das Element in der Zelle (hier die Schaltfläche) teilweise überdeckt. Das ist besser erkennbar, wenn man die Breite des Trennelements vergrößert. Die Ereignisbearbeitung für die Schaltflächen wird aber nur dann ausgelöst, wenn Sie in den Bereich klicken, der als Schaltfläche auch tatsächlich sichtbar ist.

Das nächste Beispiel (Listing 42.13) zeigt den Einsatz mehrerer GridSplitter-Elemente, um alle Zeilen und Spalten eines Grid-Steuerelements in der Breite und in der Höhe zu modifizieren.

```
<Window x:Class="GridSplitter2.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="GridSplitter #2" Height="250" Width="300"
>
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
</Grid>
```

```

<Button Grid.Column="0" Grid.Row="0">1 - 1</Button>
<Button Grid.Column="0" Grid.Row="1">1 - 2</Button>
<Button Grid.Column="0" Grid.Row="2">1 - 3</Button>
<Button Grid.Column="1" Grid.Row="0">2 - 1</Button>
<Button Grid.Column="1" Grid.Row="1">2 - 2</Button>
<Button Grid.Column="1" Grid.Row="2">2 - 3</Button>
<Button Grid.Column="2" Grid.Row="0">3 - 1</Button>
<Button Grid.Column="2" Grid.Row="1">3 - 2</Button>
<Button Grid.Column="2" Grid.Row="2">3 - 3</Button>
<GridSplitter Width="6" Grid.Column="0" Grid.Row="0" Grid.RowSpan="3" Background="Blue"
    ResizeDirection="Columns" />
<GridSplitter Width="6" Grid.Column="1" Grid.Row="0" Grid.RowSpan="3" Background="Blue"
    ResizeDirection="Columns" />
<GridSplitter Height="6" Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="3" Background="Red"
    ResizeDirection="Rows" HorizontalAlignment="Stretch" VerticalAlignment="Bottom" />
<GridSplitter Height="6" Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="3" Background="Red"
    ResizeDirection="Rows" HorizontalAlignment="Stretch" VerticalAlignment="Bottom" />
</Grid>
</Window>

```

Listing 42.13 Ein Grid mit horizontalen und vertikalen GridSplitter-Elementen

Nach der Definition von Spalten und Zeilen für das Grid-Element wird für jede Zelle eine Schaltfläche angelegt. Danach werden zuerst zwei vertikale und dann zwei horizontale GridSplitter-Elemente definiert. Für die vertikalen Trennelemente wird die Eigenschaft `ResizeDirection` auf den Wert `Columns` gesetzt. Dies ist die Standardeinstellung für diese Eigenschaft. Die horizontalen Trennelemente verwenden den Wert `Rows` für diese Eigenschaft. Hier müssen außerdem die Eigenschaften `HorizontalAlignment` und `VerticalAlignment` angepasst werden. Für die vertikalen Trennelemente können dagegen die Standardeinstellungen benutzt werden. Da die Trennelemente über das gesamte Grid hinweg sichtbar und anwendbar sein sollen, müssen die Eigenschaften `Grid.RowSpan` und `Grid.ColumnSpan` entsprechend auf den Wert 3 gesetzt werden. Das Resultat zeigt Abbildung 42.12. Sie können nun Breite und Höhe aller Zellen im Grid mit der Maus beeinflussen.

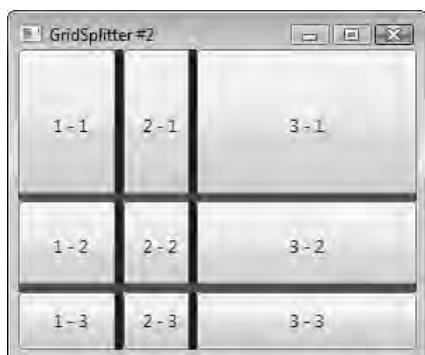


Abbildung 42.12 Vertikale und horizontale GridSplitter-Elemente im Einsatz

Das UniformGrid

Im `UniformGrid`-Element haben alle Spalten die gleiche Breite und alle Zeilen die gleiche Höhe. Die Kindelemente werden der Reihe nach in die Grid-Zellen eingefügt. Es beginnt mit Zeile 1, Spalte 1. Danach wird das Element in Zeile 1, Spalte 2 eingefügt. Das geht so weiter, bis die erste Zeile komplett mit den Kindelementen belegt ist. Nun wird mit der zweiten Zeile im `UniformGrid` fortgefahren. Im `UniformGrid`-Element gibt es keine Eigenschaften, die eine direkte Adressierung einer Zelle (wie beim normalen `Grid`-Element) ermöglichen.

```
<Window x:Class="UniformGrid.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="UniformGrid" Height="300" Width="300"
>
<UniformGrid Columns="4" Rows="3">
    <Button>1, 1</Button>
    <Button>2, 1</Button>
    <Button>3, 1</Button>
    <Button>4, 1</Button>
    <Button>1, 2</Button>
    <Button>2, 2</Button>
    <Button>3, 2</Button>
    <Button>4, 2</Button>
    <Button>1, 3</Button>
    <Button>2, 3</Button>
    <Button>3, 3</Button>
    <Button>4, 3</Button>
</UniformGrid>
</Window>
```

Listing 42.14 Ein `UniformGrid` mit vier Spalten und drei Zeilen

Listing 42.14 zeigt ein `UniformGrid`-Element mit vier Spalten und drei Zeilen aus Abbildung 42.13. Fügt man im Beispiel aus Listing 42.14 weitere Kindelemente in das Grid ein, so werden diese nicht dargestellt, da alle Zeilen und Spalten schon belegt sind. Das `UniformGrid` verhält sich jedoch anders, wenn eine feste Breite und Höhe für das Element vorgegeben wird. In diesem Fall werden auch die Kindelemente dargestellt, welche die vorgegebene Spalten- bzw. Zeilenanzahl überschreiten.



Abbildung 42.13 Ein UniformGrid mit gleicher Zeilenhöhe und Spaltenbreite für alle Zellen

Das Canvas-Element

Viele Layout-Anforderungen können mit den bereits vorgestellten Panels erfüllt werden. Was nun noch fehlt, ist ein Element, welches eine präzise Positionsangabe für seine Kindelemente ermöglicht. Oft ist es erforderlich, einzelne Grafikelemente genau an einer bestimmten Position auszugeben. Die Position der Grafiken soll nicht durch das automatische Layout beeinflusst werden. In diesem Fall müssen Sie ein Canvas-Element benutzen.

Eigentlich verhält sich das Canvas-Element sehr einfach. Es ermöglicht die genaue Positionierung von Elementen in Abhängigkeit von den Ecken des Elements. Ein Canvas-Element funktioniert etwa so, wie man früher mit Visual Basic 6 oder den Microsoft Foundation Classes (MFC) die Steuerelemente in einem Dialogfeld fest positioniert hat. Bei der Positionierung im Canvas müssen wir also die Koordinaten relativ zu den Ecken des Canvas-Elements angeben. Hierzu gibt es vier »angehängte« Eigenschaften (attached properties) in der Canvas-Klasse: Top, Left, Bottom und Right. Das folgende Beispiel (Listing 42.15) zeigt die Anwendung des Canvas bei der Positionierung von vier Schaltflächen.

```
<Window x:Class="Canvas.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Canvas" Height="300" Width="300"
    >
    <Canvas>
        <Button Canvas.Left="20" Canvas.Top="20">Button 1</Button>
        <Button Canvas.Right="20" Canvas.Top="20">Button 2</Button>
        <Button Canvas.Left="20" Canvas.Bottom="20">Button 3</Button>
        <Button Canvas.Left="200" Canvas.Top="200">Button 4</Button>
    </Canvas>
</Window>
```

Listing 42.15 Vier Schaltflächen im Canvas-Element

Beachten Sie bei diesem Beispiel die Bewegung der Schaltflächen beim Verändern der Fenstergröße. Je nachdem, an welche Eigenschaft im Canvas die Position des Kindelements gebunden wird, bleibt die Position relativ zur linken, oberen Fensterecke gleich (*Button 1* und *Button 4*) oder sie verschiebt sich (*Button 2* und *Button 3*).

Hier drängt sich sofort ein Vergleich zur *Anchor*-Eigenschaft bei WindowsForms-Steuerelementen auf. Das *Canvas*-Element verhält sich jedoch etwas anders: Wenn man z. B. *Canvas.Left* und *Canvas.Right* an die Schaltfläche bindet, so wird nicht – wie in WindowsForms – das Steuerelement beim Aufziehen des Fensters verbreitert, sondern es wird eine der beiden Eigenschaften einfach ignoriert und die Größe der Schaltfläche bleibt unverändert. Um das *Anchor*-Verhalten nachzubauen, müssen Sie zusätzlich ein *Grid*-Element verwenden.



Abbildung 42.14 Vier Schaltflächen im Canvas-Element

Das *Canvas*-Element dient in erster Linie zur exakten Positionierung von Grafiken. Wenn ein Kindelement im *Canvas* zu groß ist, so wird der überstehende Teil normalerweise nicht abgeschnitten. Dies geschieht erst, wenn die Eigenschaft *ClipToBounds* des *Canvas* auf *True* gesetzt wird.

Das Viewbox-Element

Um Grafiken in der Größe anzupassen, gibt es das *Viewbox*-Element. Es ist möglich, eine beliebige Grafik automatisch an die Größe der *Viewbox* anzupassen. Dabei können verschiedene Varianten gewählt werden. Einerseits kann das Seitenverhältnis der Grafik beibehalten werden oder die Grafik so skaliert werden, dass sie das gesamte Elternelement ausfüllt. Dieses Verhalten der *Viewbox* wird mit der Eigenschaft *Stretch* gesteuert.

```

<Window x:Class="ViewBox1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Viewbox" Height="300" Width="300"
>
<Viewbox Stretch="None">
    <!-- Stretch="Fill" oder Stretch="Uniform" oder Stretch="UniformToFill" -->
    <Canvas Width="30" Height="30">
        <Ellipse Canvas.Left="1" Canvas.Top="1"
            Width="14" Height="14"
            Fill="Red" Stroke="Black" />
        <Ellipse Canvas.Left="1" Canvas.Bottom="1"
            Width="14" Height="14"
            Fill="Green" Stroke="Black" />
        <Ellipse Canvas.Right="1" Canvas.Top="1"
            Width="14" Height="14"
            Fill="Blue" Stroke="Black" />
        <Ellipse Canvas.Right="1" Canvas.Bottom="1"
            Width="14" Height="14"
            Fill="Yellow" Stroke="Black" />
    </Canvas>
    </Viewbox>
</Window>

```

Listing 42.16 Benutzung der Viewbox

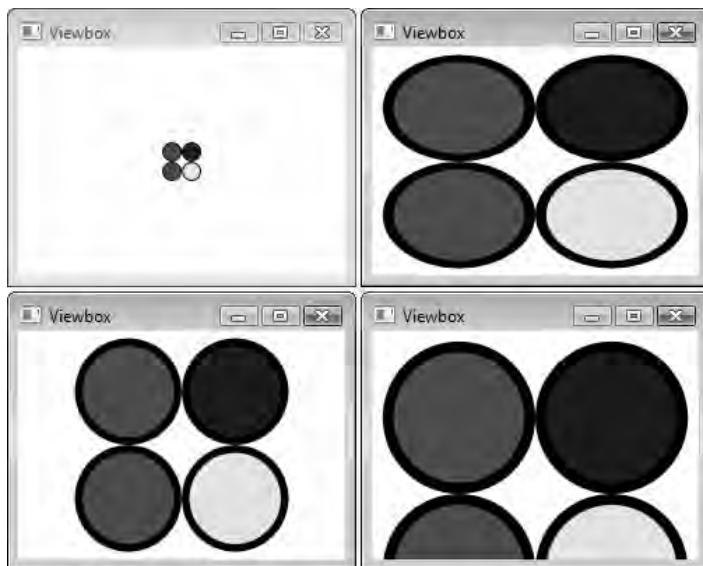


Abbildung 42.15 Grafikausgabe mit der Viewbox

Abbildung 42.15 zeigt die verschiedenen Möglichkeiten mit dem Viewbox-Element. Für das Fenster oben links wurde das Stretch-Attribut auf None gesetzt. In diesem Fall wird die Grafik in der Originalgröße ausgegeben. Im Fenster rechts oben wurde das Attribut Stretch auf Fill eingestellt. In der unteren Reihe wurde links der Wert Uniform und für das rechte Fenster der Wert UniformToFill verwendet. Beim Wert Uniform wird

das Seitenverhältnis der Grafik beibehalten und diese wird vollständig ausgegeben. Wird die Größe des Fensters geändert, dann wird, je nach eingestelltem Stretch-Attribut, die Ausgabe der Grafik angepasst.

In einer Viewbox kann man nicht nur ein Canvas-Element skalieren, sondern auch alle anderen WPF-Elemente. Die Qualität beim Vergrößern oder Verkleinern von beliebigen Elementen ist dabei sehr gut, da alles als Vektorgrafik dargestellt wird (Abbildung 42.16). Bei der Skalierung von Pixel-Grafiken können jedoch Qualitätsverluste auftreten, wenn die Vergrößerungswerte sehr groß werden oder die Grafik eine geringe Auflösung hat.



Abbildung 42.16 Ein Button-Element in der Viewbox, links: Stretch="None", rechts: Stretch="Fill"

Text-Layout

Beim Layout von Texten müssen verschiedene Eigenschaften berücksichtigt werden. Einerseits muss der Umbruch der Texte korrekt sein. Andererseits gibt es viele Parameter bei einer Schriftart, welche die Größe und die Darstellung eines Textes stark beeinflussen. Das einfachste Element, welches Texte darstellen kann, ist das TextBlock-Element.

Um einfache, kurze Texte darzustellen, kann das TextBlock-Element folgendermaßen benutzt werden:

```
<TextBlock>Hallo, Welt!</TextBlock>
```

Der Text wird im oben gezeigten Beispiel ohne weitere Formatierung dargestellt. Dennoch bietet das TextBlock-Element viele Formatierungsmöglichkeiten an. Zunächst kann man eine Schriftart für die Textdarstellung bestimmen. Dabei ist es natürlich auch möglich, die Größe der Schrift und den Schriftschnitt (fett, kursiv,...) festzulegen:

```
<TextBlock FontFamily="Courier New" FontSize="24" FontWeight="Bold">Guten Morgen!</TextBlock>
```

Eine weitere wichtige Darstellungsart des TextBlock-Elements erlaubt den Umbruch von längeren Textzeilen. Es gibt zwei Varianten des Zeilenumbruchs. Mit dem Wert `Wrap` für das Attribut `TextWrapping` erreicht man einen »vollständigen« Umbruch. D.h., wenn ein Wort nicht mehr in die Zeile passt, wird an einer Buchstabengrenze umgebrochen. Wird dem Attribut dagegen der Wert `WrapWithOverflow` zugewiesen, so wird ein nicht in die Zeile passendes Wort einfach abgeschnitten. Die überstehenden Buchstaben werden nicht dargestellt. Die beiden Varianten werden im Listing 42.17 vorgestellt.

```
<StackPanel>
  <TextBlock FontSize="24" FontStyle="Italic" FontFamily="Courier New" TextWrapping="WrapWithOverflow">
    Guten Morgen, alle miteinander hier im Raum!
  </TextBlock>
```

```
<TextBlock FontSize="24" FontWeight="Bold" FontFamily="Arial" TextWrapping="Wrap">
    Auf Wiedersehen, meine Damen und Herren!
</TextBlock>
</StackPanel>
```

Listing 42.17 Das TextBlock-Element mit TextWrapping-Möglichkeiten



Abbildung 42.17 TextBlock mit TextWrapping-Möglichkeiten

In Abbildung 42.17 können Sie die Wirkungsweise der verschiedenen Zeilenumbruchsvarianten sehen. Das erste TextBlock-Element schneidet das Wort »miteinander« einfach ab und stellt den Rest des Wortes nicht dar (TextWrapping="WrapWithOverflow"). Im zweiten TextBlock wird das längere Wort »Wiedersehen« einfach im Wort selbst umgebrochen (TextWrapping="Wrap"). Auch wenn ein Zeilenumbruch aktiviert ist, versucht ein TextBlock-Element den Text zunächst in einer Zeile auszugeben.

Ein TextBlock-Element kann jedoch nicht nur Texte umbrechen, sondern auch andere Steuerelemente, wie z.B. Schaltflächen oder Eingabefelder. In Listing 42.18 werden mehrere Elemente mithilfe eines TextBlocks zeilenweise angeordnet. Die einzelnen Elemente im TextBlock werden im Prinzip genauso angeordnet, als wären es normale Wörter.

```
<TextBlock FontSize="24" TextWrapping="Wrap">
    <Button>Test</Button>
    <Button>Noch ein Test</Button>
    <CheckBox>Ein neues Control</CheckBox>
    <TextBox>Hier Eingabe</TextBox>
</TextBlock>
```

Listing 42.18 Steuerelemente im TextBlock-Element mit Zeilenumbruch

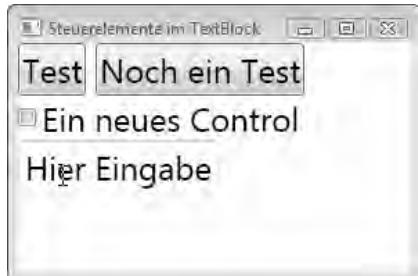


Abbildung 42.18 Steuerelemente im TextBlock mit Zeilenumbruch

Innerhalb eines Textes können Sie im TextBlock-Element verschiedene Effekte in beliebigen Kombinationen anwenden. Dazu zählen fette und kursive Schrift, sowie unterstrichener Text und ein Text als Hyperlink. Die verschiedenen Möglichkeiten können Sie im Listing 42.19 sehen.

```
<TextBlock FontSize="20" FontFamily="Arial" TextWrapping="Wrap">
    <Italic>Ein schöner Text</Italic>
    <Bold>in schöner Schrift</Bold>
    <Underline>mit einem Strich darunter.</Underline>
    <LineBreak /><LineBreak />
    <Hyperlink>Klicken Sie bitte hier!</Hyperlink>
</TextBlock>
```

Listing 42.19 Verschiedene Effekte im TextBlock-Element

An dieser Stelle soll auch erwähnt werden, dass mehrfache Leerzeichen hintereinander vom XAML-Compiler entfernt werden. Leerzeichen werden im Allgemeinen für die Formatierung des XAML-Codes benutzt. Um dies zu zeigen, ändern wir die dritte Zeile in Listing 42.19 durch Einfügen vieler Leerzeichen:

```
<Bold> in schöner Schrift </Bold>
```

Trotz dieser Modifikation im XAML-Code wird sich die Ausgabe dieser Zeile nicht ändern, wenn das Programm ausgeführt wird. Die »überzähligen« Leerzeichen werden vom XAML-Compiler einfach entfernt. Natürlich gibt es viele Fälle, in denen der Programmierer dies verhindern möchte. Hierzu muss das `xml:space`-Attribut auf den Wert `preserve` gesetzt werden, wie in Listing 42.20 gezeigt wird.

```
<TextBlock FontSize="20" FontFamily="Arial" TextWrapping="Wrap">
    <Italic>Ein schöner Text</Italic>
    <Bold xml:space="preserve"> in schöner Schrift </Bold>
    <Underline>mit einem Strich darunter.</Underline>
    <LineBreak /><LineBreak />
    <Hyperlink xml:space="preserve">Klicken Sie bitte hier!</Hyperlink>
</TextBlock>
```

Listing 42.20 Das Attribut `xml:space` wird benutzt

In Listing 42.20 kann man sehen, dass das Attribut `xml:space` für jedes Element im TextBlock erneut gesetzt werden muss. Wenden Sie dieses Attribut im TextBlock-Element selbst an, werden auch die Leerzeichen, die an den Zeilenanfängen zum Formatieren des XAML-Codes stehen, mit in die Ausgabe einbezogen.

HINWEIS Beachten Sie, dass bei Anwendung des Attributs `xml:space="preserve"` auch Zeilenumbrüche im XAML-Code in der Ausgabe umgesetzt werden. Um einen Zeilenumbruch ohne das Attribut `xml:space="preserve"` zu erzielen, können Sie das Element `<LineBreak />` beliebig oft in den Text einsetzen.



Abbildung 42.19 Mehrere Leerzeichen mit dem Attribut `xml:space`

Im `TextBlock`-Element ist es ebenfalls möglich, die Anordnung des Textes mit dem Attribut `TextAlignment` zu beeinflussen. Es sind vier Varianten vorhanden: `Left`, `Right`, `Center` und `Justify`. Listing 42.21 zeigt die Auswirkungen beim `TextAlignment`. Bei der Darstellung des Textes wird mit der gewählten Anordnung auch der Zeilenumbruch (`TextWrapping`) im `TextBlock`-Element berücksichtigt.

```
<Window x:Class="TextBlock4.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TextBlock mit TextAlignment" Height="300" Width="300">
</Window>
<StackPanel Orientation="Vertical">
    <StackPanel Orientation="Horizontal">
        <Border BorderBrush="Blue" BorderThickness="1">
            <TextBlock Width="140" TextAlignment="Left" TextWrapping="Wrap">
                Das ist ein Test mit dem TextAlignment-Attribut, welches den Text <Bold>links</Bold>
                anordnet.
            </TextBlock>
        </Border>
        <Border BorderBrush="Blue" BorderThickness="1">
            <TextBlock Width="140" TextAlignment="Right" TextWrapping="Wrap">
                Das ist ein Test mit dem TextAlignment-Attribut, welches den Text <Bold>rechts</Bold>
                anordnet.
            </TextBlock>
        </Border>
    </StackPanel>
    <StackPanel Orientation="Horizontal">
        <Border BorderBrush="Blue" BorderThickness="1">
            <TextBlock Width="140" TextAlignment="Center" TextWrapping="Wrap">
                Das ist ein Test mit dem TextAlignment-Attribut, welches den Text <Bold>zentriert</Bold>
                anordnet.
            </TextBlock>
        </Border>
        <Border BorderBrush="Blue" BorderThickness="1">
            <TextBlock Width="140" TextAlignment="Justify" TextWrapping="Wrap">
                Das ist ein Test mit dem TextAlignment- Attribut, welches den Text im <Bold>Blocksatz</Bold>
                anordnet.
            </TextBlock>
        </Border>
    </StackPanel>
</StackPanel>
```

```
</TextBlock>
</Border>
</StackPanel>
</StackPanel>
</Window>
```

Listing 42.21 Die Benutzung des Attributs `TextAlignment`

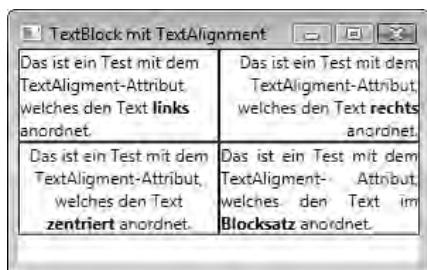


Abbildung 42.20 Die Ausgabe mit verschiedenen Werten für das `TextAlignment`

Das WrapPanel

Ein weiteres Panel, welches WPF-Elemente in einer ganz bestimmten Art und Weise anordnet, ist das `WrapPanel`. In diesem Steuerelement werden die Kindelemente zunächst horizontal nebeneinander angeordnet, bis die maximal erlaubte Breite erreicht wird. Dann findet, wie bei Texten, ein Zeilenumbruch statt. Das bedeutet, dass alle weiteren Elemente in der nächsten Zeile ausgegeben werden.

Im Listing 42.22 wird ein `WrapPanel` mit mehreren nebeneinander angeordneten Schaltflächen vorgestellt. Wird das linke Fenster in Abbildung 42.21 verkleinert, reicht der vorhandene Platz für die Schaltflächen nicht mehr aus. Nach und nach werden nun die Schaltflächen in weiteren Steuerelementzeilen umgebrochen und ausgegeben. Alle Kindelemente bleiben sichtbar, bis die Breite oder Höhe des Fensters generell für die vollständige Darstellung des Inhalts zu klein wird.

```
<WrapPanel>
  <Button>Button 1</Button>
  <Button>Hier klicken</Button>
  <Button>Test 3</Button>
</WrapPanel>
```

Listing 42.22 `WrapPanel` in horizontaler Ausrichtung



Abbildung 42.21 Das WrapPanel wird Schritt für Schritt in der Breite verkleinert

Auch das WrapPanel erlaubt zwei Orientierungen. Wird die Eigenschaft Orientation auf Horizontal gesetzt, so bekommen Sie das oben gezeigte Verhalten. Wird dagegen Orientation auf Vertical eingestellt, so werden die einzelnen Kindelemente untereinander angeordnet, bis der untere Rand des Elternelements erreicht ist. Dann werden die weiteren Elemente in einer neuen Spalte rechts daneben ausgegeben.

Standard-Layout-Eigenschaften

Alle Steuerelemente in Windows Presentation Foundation haben einige Standardeigenschaften, d.h., Eigenschaften, die Sie mit jedem Steuerelement anwenden können. Hierbei handelt es sich um grundlegende Eigenschaften, welche die einzelnen Elemente aus der Basisklasse FrameworkElement erben.

Width- und Height-Eigenschaft

Die beiden Eigenschaften `Width` und `Height` geben eine Breite und eine Höhe für ein Steuerelement vor. Verwenden Sie eines oder beide dieser Eigenschaften, so fixieren Sie die Größe dieses Elements. Dies hat verschiedene Auswirkungen. Es ist z.B. schwieriger, eine Benutzerschnittstelle mit vielen in Breite und Höhe fixierten Elementen zu lokalisieren, da die verschiedenen Landessprachen oft unterschiedliche Textbreiten benötigen. Außerdem ist es schwieriger, Dialogfelder zu erstellen, die in der Größe veränderbar sind. Wenn es geht, sollten Sie also die Definition von festen Breiten und Höhen vermeiden.

Bei der Benutzung der Eigenschaften `Width` und `Height` wird das Layout-System von WPF jedoch immer versuchen, Ihre Wünsche auszuführen. Wenn ein Element nicht vollständig darstellbar ist, z.B. weil das Fenster oder der Bildschirm zu klein ist, wird es an der entsprechenden Stelle einfach abgeschnitten.

Eine vorgegebene Breite oder Höhe eines Elements wird ggf. an die Kindelemente weitergegeben.

MinWidth-, MaxWidth-, MinHeight- und MaxHeight-Eigenschaft

Mit diesen vier Eigenschaften können Sie die minimal erlaubte Breite und Höhe und die maximal erlaubte Breite und Höhe für ein Steuerelement definieren. Die Benutzung dieser Eigenschaften macht eigentlich nur dann einen Sinn, wenn Sie kein fest definiertes Layout mit `Width` und `Height` benutzen.

Mit den Min- und Max-Eigenschaften für Breite und Höhe ist es möglich, die Größenänderungen, welche das Layout-System von WPF vornehmen kann, einzuschränken. Dies ist eine einfache Möglichkeit, die Größe von Steuerelementen zu kontrollieren.

Wenn Sie für eine der Eigenschaften einen »unmöglichen« Wert angeben, z.B. `MinHeight="1000000"`, so wird das Element zwar dargestellt, aber an der passenden Stelle abgeschnitten. Das Programm wird in solchen Fällen nicht mit einer Fehlermeldung abgebrochen. Daran ändert auch die zusätzliche Definition einer sehr kleinen maximalen Höhe nichts (z.B. `MaxHeight="50"`). In diesem Fall wird die zu kleine maximale Höhe ignoriert und die `MinHeight`-Eigenschaft für die Darstellung des Elements benutzt.

HorizontalAlignment- und VerticalAlignment-Eigenschaft

Die Eigenschaft `TextAlignment` haben Sie schon beim `TextBlock`-Element kennen gelernt. Ganz ähnlich funktionieren die Eigenschaften `HorizontalAlignment` und `VerticalAlignment`. Hierbei wird aber nicht der Text rechts-, linksbündig oder zentriert angeordnet, sondern die Steuerelemente werden entsprechend rechts, links oder mittig angeordnet. Es ist also sehr einfach, eine Schaltfläche ohne viele komplizierte Berechnungen mitten in einem Elternelement zu platzieren.

Die vier Möglichkeiten bei der Eigenschaft `HorizontalAlignment` sind `Left`, `Right`, `Center` und `Stretch`. Die Eigenschaft `VerticalAlignment` erlaubt `Top`, `Bottom`, `Center` und `Stretch`.

In einem `StackPanel` oder `WrapPanel` mit der Einstellung `Orientation="Vertical"` werden alle Kindelemente von oben nach unten angelegt. Das erste Element wird am oberen Rand des Elternelementes ausgegeben. Nun können Sie für das Panel zusätzlich die Eigenschaft `VerticalAlignment="Bottom"` setzen. Die Kindelemente werden jetzt so angeordnet, dass das letzte Element mit dem unteren Rand des Elternelementes abschließt.

HINWEIS Wenn die Eigenschaften `VerticalAlignment="Bottom"` bzw. `HorizontalAlignment="Right"` gesetzt werden, bleibt so die Reihenfolge der Elemente erhalten. Das letzte Kindelement schließt nur mit dem unteren bzw. rechten Rand ab.

Die Auswahl `Stretch` ist für beide Eigenschaften der Standardwert. In diesem Fall versucht das Element, den gesamten Platz auszunutzen, der zur Verfügung steht.

Margin-Eigenschaft

Die Eigenschaft `Margin` bestimmt die Breite des Randes, der um das Element herum frei gelassen werden soll. Diese Eigenschaft kann benutzt werden, wenn ein Element sein Elternelement nicht vollständig ausfüllen soll. `Margin`-Werte können in unterschiedlicher Weise angegeben werden. Wenn Sie eine Zahl angeben (z.B. `Margin="10"`), so wird dieser Wert für alle vier Ränder (oben, unten, rechts und links) des Kindelements herangezogen. Mit der Angabe von zwei Zahlen für die Eigenschaft (z.B. `Margin="10 20"`) definieren Sie mit der ersten Zahl die Breite des rechten und linken Randes. Die zweite Zahl in der `Margin`-Eigenschaft gibt an, wie breit die Ränder oben und unten sein sollen. Wenn Sie vier Werte angeben, können Sie vier unterschiedliche Werte für alle Ränder definieren. Die Reihenfolge der Zahlen ist dann: Linker, oberer, rechter und unterer Rand.

Das Beispiel in Listing 42.23 zeigt die Benutzung der verschiedenen `Margin`-Arten. Dort wird ein `Grid` mit vier Spalten definiert. Die Zellen werden durch gestrichelte Linien dargestellt (`ShowGridLines="True"`).

HINWEIS Die Angabe von drei Zahlen in der Margin-Eigenschaft ist nicht erlaubt und führt zum Abbruch des Programms.

```
<Window x:Class="ShowMargin.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Margin" Height="300" Width="500"
    >
    <Grid ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <Button Grid.Column="0">Ohne Margin</Button>
        <Button Grid.Column="1" Margin="15">Margin 1</Button>
        <Button Grid.Column="2" Margin="10,30">Margin 2</Button>
        <Button Grid.Column="3" Margin="10,20,30,40">Margin 4</Button>
    </Grid>
</Window>
```

Listing 42.23 Verwendung der Eigenschaft Margin

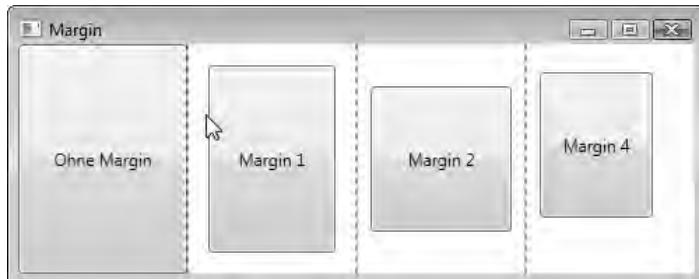


Abbildung 42.22 Verwendung unterschiedlicher Margin-Definitionen

Wird das Fenster mit dem in Listing 42.23 gezeigten Programm in der Größe geändert, so werden die Breiten der Ränder der einzelnen Schaltflächen unverändert bleiben. Die Größen der Schaltflächen werden aber entsprechend angepasst. Verkleinern Sie das Fenster nun immer weiter, so wird irgendwann ein Punkt erreicht werden, an dem der definierte Rand und die Standardgröße des Kindelements nicht mehr in den vorhandenen Bereich passen. Dann wird das jeweilige Kindelement (hier die Schaltflächen) nicht mehr vollständig dargestellt. Es kann sogar passieren, dass ein Element gar nicht mehr dargestellt wird, d.h., die Darstellung des Randes hat immer Vorrang vor der Ausgabe des Kindelementes.

Padding-Eigenschaft

Die Eigenschaft Padding beeinflusst den Abstand zwischen dem Rand eines Steuerelements und den darin enthaltenen Daten (Content). In Abbildung 42.23 wird der Unterschied zwischen den beiden Eigenschaften Margin und Padding dargestellt.

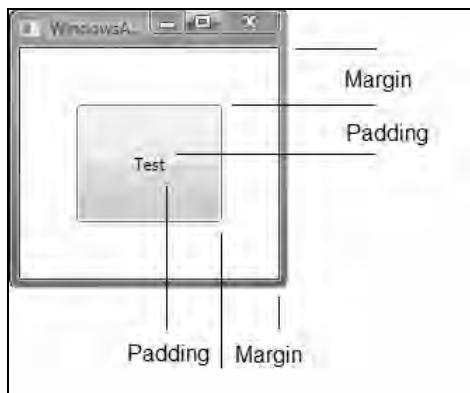


Abbildung 42.23 Margin und Padding

Die Padding-Eigenschaft finden Sie nur in solchen WPF-Elementen, die einen Inhalt darstellen können. Wenn Sie die Werte für das Padding zu groß wählen, ist es möglich, dass der Inhalt des Elements gar nicht oder nur teilweise dargestellt wird. Auch bei der Padding-Eigenschaft können Sie eine Zahl, zwei oder vier Zahlen als Parameter angeben. Das Verhalten der Padding-Parameter ist identisch mit dem der Margin-Angaben.

Zusammenfassung

In diesem Kapitel haben Sie die verschiedenen Layout-Panels von WPF kennen gelernt. Mit WPF werden die Positionen und Größen der einzelnen Elemente in einem Fenster normalerweise nicht mehr statisch festgelegt, sondern es werden Panels verwendet, die eine dynamische Anordnung der Elemente beim Vergrößern oder Verkleinern des Fensters ermöglichen. WPF enthält viele leistungsfähige Panels: StackPanel, DockPanel und WrapPanel. Weiterhin steht das Grid-Element für ein sehr flexibles Layout zur Verfügung.

In den Layout-Steuerelementen gibt es bestimmte Standardeigenschaften, die es ermöglichen, die Art des Layouts zu steuern.

Außerdem gibt es ein WPF-Element, welches das genaue Positionieren von Elementen auf absoluter Basis ermöglicht. In diesem Canvas-Element können einzelne Grafikelemente exakt arrangiert werden.

Für das Layout von Texten wurde in WPF natürlich auch gesorgt. Das TextBlock-Element lässt viele Formattierungsmöglichkeiten zu und stellt auch einen Zeilenumbruch zur Verfügung.

Sie können alle Layout-Elemente hierarchisch ineinander verschachteln, so dass auch komplexe Anordnungen von Steuerelementen in einem Fenster mit änderbarer Größe automatisch angeordnet werden können.

Kapitel 43

Grafische Grundelemente

In diesem Kapitel:

Grundlagen	1284
Die Grafik-Auflösung	1291
Die grafischen Grundelemente	1293
Zusammenfassung	1307

Ohne die grafischen Grundelemente wie Linien, Rechtecke, Ellipsen u.v.m. geht in einem fensterorientierten System eigentlich gar nichts. WPF hat einiges in diesem Bereich zu bieten, denn das *P* in WPF steht ja bekanntlich für *Presentation*. In diesem Kapitel möchte ich Ihnen die Möglichkeiten vorstellen, welche Sie mit diesen Grafikelementen haben. In der zweiten Hälfte des Kapitels werden wir uns mit den komplexen Grafikelementen beschäftigen, dazu gehören die Pfade, Polygone und Bezierkurven.

BEGLEITDATEIEN

Die Begleitdateien zu folgenden Beispielen finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\H - WPF

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Grundlagen

Grafische Grundelemente können mit WPF überall in der Benutzerschnittstelle verwendet werden. Eine Trennung von Steuerelementen und grafischen Elementen ist hier nicht vorhanden. Grafische Elemente, wie Linien und Ellipsen, sind genau so in einer Benutzerschnittstelle einsetzbar, wie Schaltflächen oder Textboxen. Auch das Programmierparadigma ist identisch, wie das Beispiel in Listing 43.1 zeigt.

```
<Window x:Class="TextKomplex.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Text mit Grafik" Height="150" Width="650"
>
<DockPanel>
    <StackPanel Height="30" DockPanel.Dock="Top" Orientation="Horizontal">
        <TextBlock FontSize="20">Mit WPF</TextBlock>
        <Ellipse Width="20" Fill="Red" />
        <TextBlock VerticalAlignment="Center">kann man Grafik</TextBlock>
        <Rectangle Width="30" Fill="Blue" />
        <TextBlock VerticalAlignment="Center">einfach mischen. Und einen</TextBlock>
        <Button Width="120">
            <StackPanel Orientation="Horizontal">
                <Ellipse Width="30" Height="10" Stroke="Blue" Fill="Green" />
                <TextBlock>Hallo</TextBlock>
                <Ellipse Width="30" Height="10" Stroke="Blue" Fill="Red" />
            </StackPanel>
        </Button>
        <TextBlock VerticalAlignment="Center">kann man auch einfügen!</TextBlock>
        <Ellipse Width="30" Fill="Yellow" Stroke="Black" />
    </StackPanel>
</DockPanel>
</Window>
```

Listing 43.1 Texte und grafische Elemente

Im Beispiel aus Listing 43.1 wird in einem DockPanel ein StackPanel mit horizontaler Orientierung platziert. Im StackPanel können nun sowohl Texte (TextBlock) als auch grafische Elemente (Ellipse, Rectangle) in beliebiger Anordnung dargestellt werden. Auch Steuerelemente, wie hier eine Schaltfläche (Button), können

eingesetzt werden. Die Schaltfläche selbst kann dann wieder aus grafischen Elementen und Texten bestehen. Das Ergebnis kann in Abbildung 43.1 betrachtet werden.

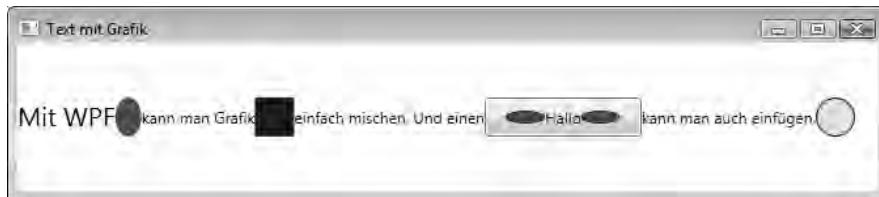


Abbildung 43.1 Texte und Grafik gemischt

Die grafischen Ausgaben können allerdings nicht nur in XAML definiert, sondern auch durch normalen Programmcode erzeugt werden. Hierbei werden Sie jedoch schnell feststellen, dass eine hierarchische Darstellung der einzelnen Elemente in XAML meistens viel übersichtlicher und einfacher zu lesen ist. In Listing 43.2 wird das Fenster aus Abbildung 43.1 nur durch VB-Code erzeugt.

```
Imports System
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Media
Imports System.Windows.Shapes

Namespace TextKomplex
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
            InitMyLayout()
        End Sub

        Private Sub InitMyLayout()
            Dim dock As New DockPanel

            Dim stack As New StackPanel
            stack.Orientation = Orientation.Horizontal
            stack.Height = 30

            Dim t1 As New TextBlock
            t1.FontSize = 20
            t1.Text = "Mit WPF"
            stack.Children.Add(t1)

            Dim e1 As New Ellipse
            e1.Width = 20
            e1.Fill = Brushes.Red
            stack.Children.Add(e1)

            Dim t2 As New TextBlock
            t2.Text = "kann man Grafik"
            t2.VerticalAlignment = Windows.VerticalAlignment.Center
            stack.Children.Add(t2)
        End Sub
    End Class
End Namespace
```

```
Dim r1 As New Rectangle
r1.Width = 30
r1.Fill = Brushes.Blue
stack.Children.Add(r1)

Dim t3 As New TextBlock
t3.Text = "einfach mischen. Und einen"
t3.VerticalAlignment = Windows.VerticalAlignment.Center
stack.Children.Add(t3)

Dim btn As New Button
btn.Width = 120

Dim pa As New StackPanel
pa.Orientation = Orientation.Horizontal

Dim ebtn1 As New Ellipse
ebtn1.Width = 30
ebtn1.Height = 10
ebtn1.Fill = Brushes.Green
ebtn1.Stroke = Brushes.Blue
pa.Children.Add(ebtn1)

Dim tbtn As New TextBlock
tbtn.Text = "Hallo"
pa.Children.Add(tbtn)

Dim ebtn2 As New Ellipse
ebtn2.Width = 30
ebtn2.Height = 10
ebtn2.Fill = Brushes.Red
ebtn2.Stroke = Brushes.Blue
pa.Children.Add(ebtn2)
btn.Content = pa

stack.Children.Add(btn)

Dim t4 As New TextBlock
t4.Text = "kann man auch einfügen."
t4.VerticalAlignment = Windows.VerticalAlignment.Center
stack.Children.Add(t4)

Dim e2 As New Ellipse
e2.Width = 30
e2.Fill = Brushes.Yellow
e2.Stroke = Brushes.Black
stack.Children.Add(e2)

dock.Children.Add(stack)
Me.Content = dock
End Sub
End Class
End Namespace
```

Listing 43.2 Definition von Hierarchien mit VB-Code ist oft aufwändig

Die Darstellung von grafischen Elementen auf der Schaltfläche im obigen Beispiel ist ebenfalls sehr einfach. Hierbei sollten Sie beachten, dass WPF sehr flexibel ist, was die Hierarchie der einzelnen Elemente anbelangt. So ist es z.B. ohne weiteres möglich, ein Grid-Element auf einer Schaltfläche zu platzieren, um dann die einzelnen Zellen zur Darstellung von grafischen Elementen auszunutzen. Ein grafisches Element kann auch mehrere Zellen im Grid belegen. Listing 43.3 zeigt ein Beispiel.

```
<Window x:Class="ButtonKomplex.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Eine Super-Schaltfläche" Height="300" Width="400" Background="LightGray"
>
<Button Width="250" Height="150">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition Width="3*" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition Height="3*" />
            <RowDefinition />
        </Grid.RowDefinitions>

        <Rectangle Name="rect1" Grid.Column="0" Grid.Row="0" Fill="Red" Stroke="Black" />
        <Rectangle Name="rect2" Grid.Column="2" Grid.Row="2" Fill="Red" Stroke="Black" />

        <TextBlock Name="text1" Grid.Column ="2" Grid.Row="0"
            FontSize="14" VerticalAlignment="Top" Text="Klicken!" />

        <TextBlock Name="text2" Grid.Column ="0" Grid.Row="2"
            FontSize="14" VerticalAlignment="Bottom" Text="Klicken!" />

        <Image Grid.Column="1" Grid.Row="1"
            Source="D:\Avalon-Buch\Buch-Dateien\Kap04\Astro.jpg" />

        <Ellipse Name="ellip" Grid.Column="0" Grid.Row="0"
            Grid.ColumnSpan="3" Grid.RowSpan="3"
            Width="230" Height="130" Stroke="Blue" StrokeThickness="5" />
    </Grid>
</Button>
</Window>
```

Listing 43.3 Eine Schaltfläche mit grafischen Elementen

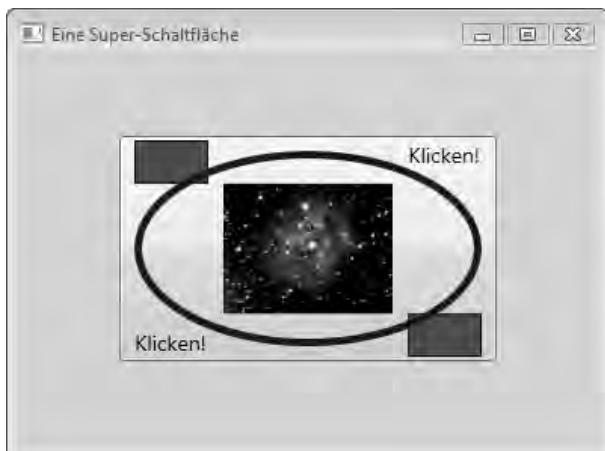


Abbildung 43.2 Die grafische Schaltfläche aus Listing 43.3

Die Schaltfläche in Abbildung 43.2 wird nicht über eine Bitmap, also eine pixelorientierte Grafik realisiert, sondern besteht weiter aus den einzelnen vektororientierten, grafischen Grundelementen, die z.B. auch auf Ereignisse reagieren können.

Bei vielen Benutzeroberflächen-Technologien müssen Steuerelemente von Grund auf neu programmiert werden, wenn sich die Darstellung ändert. Oft ist das Verhalten beim Neuzeichnen dieser Steuerelemente nicht effizient und sie werden öfter gezeichnet, als eigentlich notwendig ist. Dies führt manchmal zu »flackernden« Benutzerschnittstellen, mit denen das Arbeiten nicht sehr angenehm ist.

WPF geht hier einen anderen Weg. Die grafischen Elemente werden ebenso wie die Steuerelemente in einer baumähnlichen Struktur verwaltet. Grafiken können dadurch genauso wie z.B. eine Schaltfläche durch Änderung ihrer Eigenschaften modifiziert werden. Der Programmierer muss sich jedoch nicht um das korrekte Neuzeichnen des Fensters oder des Fensterausschnitts kümmern.

Das Beispiel aus Listing 43.3 soll so erweitert werden, dass die Schaltfläche das Click-Ereignis bedient:

```
<Button Width="250" Height="150" Click="OnClicked">
```

Die Elemente, die aus dem VB-Code heraus verändert werden sollen, sind in Listing 43.3 bereits mit den Objektnamen versehen. Es muss also nur noch die Methode für das Click-Ereignis implementiert werden (Listing 43.4). Beim Anklicken der Schaltfläche ändert sich sofort das Aussehen derselben. Es wird keine Methode zum Neuzeichnen der Grafiken aufgerufen.

```
Imports System
Imports System.Windows

Namespace ButtonKomplex
    Partial Public Class Window1
        Inherits System.Windows.Window

        Public Sub New()
            InitializeComponent()
        End Sub
    End Class
End Namespace
```

```
Private Sub OnClicked(ByVal sender As Object, ByVal e As RoutedEventArgs)
    rect1.Fill = Brushes.Yellow
    rect2.Fill = Brushes.Yellow
    text2.FontStyle = FontStyles.Italic
    ellip.Stroke = Brushes.Red
    ellip.StrokeThickness = 8

    Dim d As New DoubleCollection
    d.Add(4)
    d.Add(1)
    d.Add(3)
    d.Add(3)
    ellip.StrokeDashArray = d
End Sub
End Class
End Namespace
```

Listing 43.4 Veränderung der Objekte aus VB-Code

Im nächsten Beispiel soll geprüft werden, wie die Behandlung der Ereignisse solcher Grafikelemente funktioniert. In einem Fenster werden mit XAML mehrere Rechtecke (Rectangle) erzeugt. Jedes Rechteck beinhaltet eine Methode für die Verarbeitung des Click-Ereignisses. Beim Anklicken wird das jeweilige Rechteck um 10 Einheiten vergrößert. Hierbei stellen sich nun drei interessante Fragen:

- Vergrößert sich der Bereich für das Click-Ereignis entsprechend der Vergrößerung des Rechtecks?
- Wie kann man das angeklickte Rechteck identifizieren?
- Was passiert, wenn Rechtecke angeklickt werden, die so groß sind, dass sie übereinander liegen?

Listing 43.5 zeigt den entsprechenden VB- und XAML-Code und Abbildung 43.3 zeigt eine mögliche Ausgabe im Fenster.

```
<!-- XAML: Window1.xaml -->
<Window x:Class="TestEreignis.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Test Ereignisse" Height="300" Width="500"
        >
<Canvas>
    <Rectangle Canvas.Left="10" Canvas.Top="30" Fill="Blue"
               Width="40" Height="40" MouseLeftButtonDown="OnClicked" />
    <Rectangle Canvas.Left="110" Canvas.Top="30" Fill="Red"
               Width="40" Height="40" MouseLeftButtonDown="OnClicked" />
    <Rectangle Canvas.Left="210" Canvas.Top="30" Fill="Black"
               Width="40" Height="40" MouseLeftButtonDown="OnClicked" />
    <Rectangle Canvas.Left="310" Canvas.Top="30" Fill="Green"
               Width="40" Height="40" MouseLeftButtonDown="OnClicked" />
    <Rectangle Canvas.Left="410" Canvas.Top="30" Fill="Yellow"
               Width="40" Height="40" MouseLeftButtonDown="OnClicked" />
</Canvas>
</Window>
```

```

Imports System
Imports System.Windows

Namespace TestEreignis
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnClicked(ByVal sender As Object, ByVal e As RoutedEventArgs)
            Dim r As Rectangle = CType(sender, Rectangle)
            r.Width += 10
            r.Height += 10
        End Sub
    End Class
End Namespace

```

Listing 43.5 Mehrere Grafikelemente mit einer Ereignismethode

In der Ereignismethode können Sie (wie z.B. aus Windows Forms bekannt) einfach das Objekt mit dem Namen `sender` in ein `Rectangle`-Objekt konvertieren. Danach können Sie, wie gewohnt, auf die Eigenschaften des Elements zugreifen und diese ändern. Im Beispiel wird nicht explizit eine Neuzeichnungsmethode aufgerufen. WPF zeichnet das Element nach der Änderung der Eigenschaften `Width` und `Height` neu. Mit der Vergrößerung des Rechtecks erweitert sich auch der Bereich für das `MouseLeftButtonDown`-Ereignis ohne weiteren Programmieraufwand.

Wenn die einzelnen Rechtecke größer werden und sich überlagern, so erfolgt die Darstellung in der Reihenfolge der Definition aus dem XAML-Code. Zuerst wird also das blaue Rechteck, dann das rote und zuletzt das gelbe Rechteck gezeichnet.

Beim Experimentieren mit dem Beispiel können Sie feststellen, dass bei überlagerten Rechtecken nur das `Click`-Ereignis für das jeweils ganz oben liegende Rechteck ausgelöst wird. Das `Click`-Ereignis wird nicht an die eventuell (optisch) darunter liegenden Rechtecke weitergeleitet, da alle Rechtecke in der Element-Hierarchie (logischer Baum) in der gleichen Ebene liegen.

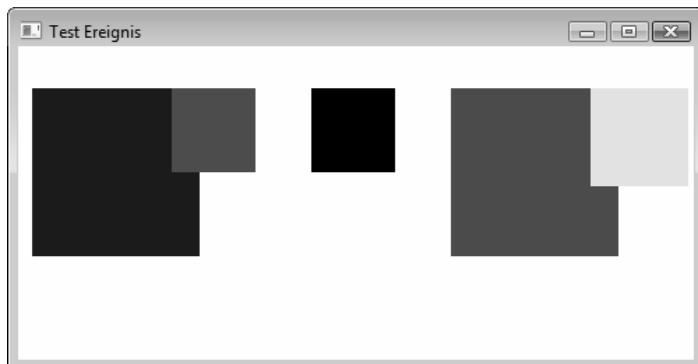


Abbildung 43.3 Alle Rechtecke werden in einer Ereignismethode behandelt

Die Grafik-Auflösung

Unsere Rechner sind in den letzten Jahren nicht nur schneller geworden, auch die Grafikkarten sind im Laufe der Zeit immer weiter verbessert worden. Mein erster Computer, ein Radio Shack TRS-80 Level II, hatte einen Z80-Prozessor mit 0,9 MHz Taktfrequenz. Die Grafikauflösung auf einem Schwarz/Weiß-Monitor betrug sagenhafte 128x48 »Klötz« (Nein, da fehlt keine Null hinten!). Aber auch bei diesen Auflösungen konnten verschiedene Programme schon 3D-Darstellungen von einfachen Objekten auf den Bildschirm zaubern. Eigentlich kaum vorstellbar.

In der heutigen Zeit sind die Grafikkarten wesentlich leistungsfähiger. Aber nicht jede Software nutzt alle Möglichkeiten der Grafikhardware aus. Grafiken, die in beliebiger Weise vergrößer- oder verkleinerbar sind, kann man mit Windows schon seit langer Zeit programmieren. Die Grundlagen hierfür finden Sie in der GDI.DLL von Windows. Diese Grafik-Bibliothek ist im Jahr 2001 durch GDI+ noch wesentlich erweitert worden. Trotzdem ist es nicht möglich, mit vertretbarem Aufwand und minimalen Einschränkungen eine vollständig skalierbare Benutzerschnittstelle (hier ist auch die Größe der Steuerelemente gemeint) für Windows-Anwendungen zu erstellen.

Wie bereits weiter oben erwähnt, gibt es diese Trennung von Grafik- und Benutzerschnittstellenelementen in WPF nicht mehr. Eine in der Größe änderbare Benutzerschnittstelle sollte also möglich sein. Wie sieht es aber mit der Darstellungsqualität aus? Wenn Rastergrafiken (Bitmaps) vergrößert werden, erscheinen sehr schnell die unansehnlichen »Klötz« auf dem Bildschirm, die das Betrachten der Grafiken eher zu einem unangenehmen Erlebnis machen. Aber auch hier hat sich in WPF Einiges getan! Alle Elemente der Benutzerschnittstelle werden aus den vektororientierten Grafikelementen erstellt. Es werden keine Rastergrafiken verwendet.

Im folgenden Beispiel (Listing 43.6) können Sie eine Schaltfläche sehen, die mit einer LayoutTransform in unterschiedliche Größen transformiert wurde. Die Schaltfläche bleibt hierbei immer exakt die gleiche, nur die Größe wird durch *Zoomen* geändert. Die hierfür verwendete Operation ist eine LayoutTransform. Diese Art der Transformation werden wir im Laufe dieses Kapitels noch genauer kennen lernen. Im Moment genügt es, wenn Sie wissen, dass diese Transformation die gesamte Benutzerschnittstelle beeinflussen kann. Dabei muss die Transformation nicht für jedes Element neu angegeben werden. Im Beispiel wird die LayoutTransformation nur für die Schaltfläche definiert. Alle Elemente, die in der Hierarchie unter der Schaltfläche liegen (*Ellipse*, *TextBlock*, *Rectangle* und *Line*) werden automatisch entsprechend skaliert.

```
<Window x:Class="ButtonGross.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Schaltfläche transformieren" Height="300" Width="600">
</Window>
<StackPanel>
    <Button Height="50" Width="250" Margin="10">
        <Button.LayoutTransform>
            <ScaleTransform ScaleX="1" ScaleY="1" />
        </Button.LayoutTransform>

        <StackPanel Orientation="Horizontal">
            <Ellipse Width="50" Height="25" Fill="Red" Stroke="Black" />
            <TextBlock FontSize="20">Klicken!</TextBlock>
        </StackPanel>
    </Button>
</StackPanel>
```

```

<Rectangle Width="50" Height="25" Stroke="Blue" RadiusX="8" RadiusY="8" />
<Line X1="0" Y1="0" X2="50" Y2="25" Stroke="Green" StrokeThickness="3" />
</StackPanel>
</Button>
</StackPanel>
</Window>

```

Listing 43.6 Eine Schaltfläche mit einer LayoutTransformation

In Abbildung 43.4 werden mehrere der Schaltflächen dargestellt. Hierbei wurden die Skalierungsfaktoren (`ScaleX` und `ScaleY`) für die `LayoutTransform` auf die Werte 1, 2, 3 und 5 eingestellt. Die Schaltfläche wird Schritt für Schritt größer. Trotzdem bleibt die Qualität der Grafik sehr hoch. Das Bild wird nicht unscharf und enthält keine »Klötzte«. Alle Grafikelemente, die Schaltfläche und der Text werden nicht als Rastergrafik vergrößert, sondern mit dem jeweiligen Zoomfaktor neu gezeichnet. Dabei werden auch die Liniendicken der einzelnen Grafikelemente entsprechend vergrößert, sodass die gesamte Darstellung immer noch »ausgewogen« aussieht.

Genauso, wie die Schaltfläche in Abbildung 43.4 skaliert wurde, können Sie jedes Element der Benutzeroberfläche vergrößern oder verkleinern. Aber damit noch nicht genug. Ebenso einfach können Sie ein Steuerelement oder ein Grafikelement drehen, kippen oder nur an eine andere Stelle verschieben. Auch diese Operationen können mit einer `LayoutTransform` durchgeführt werden.

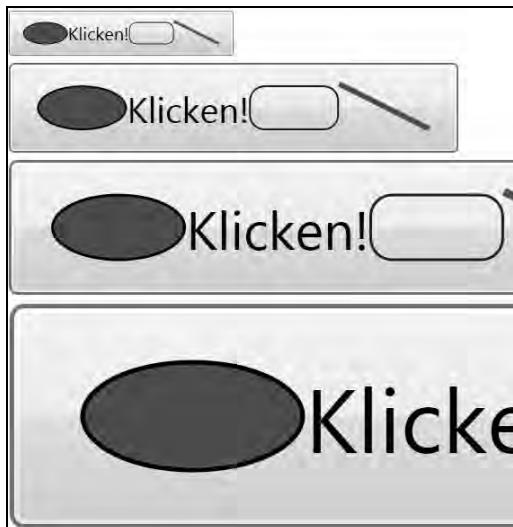


Abbildung 43.4 Die vergrößerte Schaltfläche

Da Sie alle Elemente in einer beliebigen Größe darstellen können, gibt es im Grunde genommen keine feste Beziehung zwischen den Koordinaten der Grafikobjekte und den physikalischen Pixeln des Bildschirms. Ganz abgesehen davon können alle Koordinaten als Fließkommazahlen angegeben werden. In WPF werden die Koordinaten in so genannten *geräteunabhängigen Pixeln* angegeben. Ein geräteunabhängiger Pixel entspricht 1/96 Zoll. Anders ausgedrückt entsprechen 96 Pixel einer Länge von 25,4 mm auf dem Bildschirm. Die Anzahl der tatsächlichen, physikalischen Pixel, die hierzu benötigt werden, hängt von der Auflösung des Bildschirms ab.

Der »krumme« Wert von 96 DPI (Dots per Inch) pro Zoll kommt daher, dass in Windows die Auflösung des Standard-Displays 96 DPI ist. Für das Standard-Display gilt somit: 1 logischer Pixel = 1 echter Pixel! Für andere Wiedergabegeräte werden alle Koordinaten entsprechend umgerechnet.

Die grafischen Grundelemente

Beginnen wir mit dem Zusammenspiel der grafischen Grundelemente (Shapes), der Stifte (Pens) und der Pinsel (Brushes). Es gibt diverse Klassen, welche die grafischen Grundelemente zur Verfügung stellen. Diese Klassen beinhalten verschiedene Eigenschaften, um das Aussehen zu beeinflussen.

- Rectangle
- Ellipse
- Line
- Polyline
- Polygon
- Path

Für diese grafischen Objekte kann man Füllfarben in Form von Pinseln (Brush) auswählen. Der äußere Rand kann durch einen Stift (Pen) definiert werden. Der einfachste Pinsel ist ein SolidColorBrush, der nur aus einer vorgegebenen Farbe besteht. Andere Pinsel, die wir später noch kennen lernen werden, können dagegen komplexe Farbverläufe darstellen. Listing 43.7 zeigt, wie Sie Ellipsen, Rechtecke und abgerundete Rechtecke zeichnen können. Die Füllfarbe wird mit der Eigenschaft Fill bestimmt, die Randfarbe mit der Eigenschaft Stroke. Die Dicke der Randlinie wird mit der Eigenschaft StrokeThickness angegeben. Schließlich können Sie beliebige gestrichelte Linien erzeugen. Hierzu geben Sie mit der Eigenschaft StrokeDashArray an, wie viele Einheiten des Striches gezeichnet und wie viele Einheiten nicht gezeichnet werden sollen. Sie können hier auch mehrere Zahlen angeben, sodass Sie sehr komplizierte Strichmuster erzeugen können (Abbildung 43.5).

```
<Window x:Class="Shapes1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Grafische Grundelemente" Height="350" Width="300">
<
<Canvas>
    <Ellipse Canvas.Left="10" Canvas.Top="10" Width="80" Height="60" Fill="Red" />
    <Rectangle Canvas.Left="100" Canvas.Top="10" Width="80" Height="60" Fill="Blue" />
    <Rectangle Canvas.Left="190" Canvas.Top="10" Width="80" Height="60"
        RadiusX="15" RadiusY="15" Fill="Green" />

    <Ellipse Canvas.Left="10" Canvas.Top="80" Width="80" Height="60" Fill="Red" Stroke="Black" />
    <Rectangle Canvas.Left="100" Canvas.Top="80" Width="80" Height="60" Fill="Blue" Stroke="Red" />
    <Rectangle Canvas.Left="190" Canvas.Top="80" Width="80" Height="60"
        RadiusX="15" RadiusY="15" Fill="Green" Stroke="Aquamarine" />

    <Ellipse Canvas.Left="10" Canvas.Top="150" Width="80" Height="60" Fill="Red" Stroke="Black"
        StrokeThickness="3" />
    <Rectangle Canvas.Left="100" Canvas.Top="150" Width="80" Height="60" Fill="Blue" Stroke="Red"
        StrokeThickness="7" />
    <Rectangle Canvas.Left="190" Canvas.Top="150" Width="80" Height="60" RadiusX="15" RadiusY="15"
```

```

        Fill="Green" Stroke="Aquamarine" StrokeThickness="10" />

<Ellipse Canvas.Left="10" Canvas.Top="220" Width="80" Height="60" Fill="Red" Stroke="Black"
    StrokeThickness="3" StrokeDashArray="2 2" />
<Rectangle Canvas.Left="100" Canvas.Top="220" Width="80" Height="60" Fill="Blue" Stroke="Red"
    StrokeThickness="7" StrokeDashArray="2 3 4 2" />
<Rectangle Canvas.Left="190" Canvas.Top="220" Width="80" Height="60" RadiusX="15" RadiusY="15"
    Fill="Green" Stroke="Aquamarine" StrokeThickness="10" StrokeDashArray="1 1"/>
</Canvas>
</Window>

```

Listing 43.7 Verschiedene Ellipsen und Rechtecke



Abbildung 43.5 Verschiedene Ellipsen und Rechtecke

Rechteck und Ellipse

Die grafischen Grundelemente Rectangle, Ellipse, Line, Polygon, Polyline und Path sind von der Basisklasse Shape abgeleitet. Die meisten Eigenschaften der grafischen Grundelemente kommen aus dieser Basisklasse.

Wie Sie bereits gelernt haben, wird der Rand eines grafischen Elements über die Eigenschaft Stroke der Shape-Klasse angegeben, welche vom Typ Brush ist. Intern wird an dieser Stelle ein Pen-Objekt benutzt, wie man das eigentlich auch erwarten würde. Damit die XAML-Syntax aber nicht zu kompliziert wird, werden die Eigenschaften des Pen-Objektes über eigene Eigenschaften im Shape-Objekt abgebildet.

Rectangle und Ellipse haben wir schon in einem Beispiel kennen gelernt (Listing 43.7). Dort wurde die Positionierung der grafischen Elemente mit einem Canvas-Objekt durchgeführt. Die grafischen Elemente können aber auch genauso in einen DockPanel, StackPanel, WrapPanel oder einem Grid (Listing 43.8) positioniert werden.

```
<Window x:Class="Shapes2.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Grafik im Grid" Height="200" Width="300"
>
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="2*" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="2*" />
        <RowDefinition />
    </Grid.RowDefinitions>

    <Ellipse Grid.Column="0" Grid.Row="0" Fill="Blue" />
    <Rectangle Grid.Column="1" Grid.Row="0" Fill="Red" />
    <Rectangle Grid.Column="0" Grid.Row="1" Fill="Green" />
    <Canvas Grid.Column="1" Grid.Row="1">
        <Ellipse Canvas.Left="10" Canvas.Top="10" Width="70" Height="35"
            Fill="White" Stroke="Black" StrokeThickness="3" />
    </Canvas>
</Grid>
</Window>
```

Listing 43.8 Die Positionierung von grafischen Elementen mit einem Grid



Abbildung 43.6 Grafische Elemente im Grid

In Abbildung 43.6 können Sie das Ergebnis des Programms aus Listing 43.8 sehen. In diesem Grid ist die erste Zeile doppelt so hoch wie die zweite Zeile. Dies gilt auch für die Spaltenbreiten. Die blaue Ellipse und die beiden Rechtecke füllen jeweils die gesamte Zelle im Grid aus, da explizit keine Größen (Width, Height) angegeben wurden. Die kleine weiße Ellipse wurde mithilfe eines Canvas-Elementes positioniert. Das Canvas-Element füllt in diesem Fall die gesamte Grid-Zelle aus. Im Canvas selbst kann nun die Ellipse beliebig angeordnet werden.

Einfache Transformationen

Da wir gerade mit Grafiken zu tun haben, stellt sich natürlich auch die Frage, ob es möglich ist, ein schräg liegendes Rechteck oder eine »schräge« Ellipse zu zeichnen. Bisher wurden die Ellipsen und Rechtecke immer so dargestellt, dass die beiden Hauptachsen des Grafikobjektes parallel zu den Achsen des Koordinatensystems lagen. Das soll nun geändert werden.

Die meisten Elemente in WPF enthalten zwei wichtige Transformationen, die als Eigenschaften in den Klassen vorhanden sind: `LayoutTransform` und `RenderTransform`. Die `LayoutTransform`-Eigenschaft wird in erster Linie benutzt, um Teile der Benutzerschnittstelle zu vergrößern oder zu verkleinern. Bei der Anwendung einer `LayoutTransform` werden die einzelnen Elemente in der Benutzerschnittstelle durch das Layout-System von WPF ggf. neu arrangiert. Diese Transformation werden wir später noch genauer kennen lernen.

Die Eigenschaft `RenderTransform` beeinflusst dagegen nur den Inhalt eines Steuerelements. Diese Transformation können wir benutzen, um unsere grafischen Objekte zu drehen. Abbildung 43.7 zeigt das Ergebnis aus Listing 43.9.

```
<Window x:Class="Transform1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Gedrehte Objekte" Height="350" Width="400"
>
<Canvas>
    <Rectangle Canvas.Left="100" Canvas.Top="50" Width="200" Height="50" Fill="Red" />

    <Ellipse Canvas.Left="100" Canvas.Top="50" Width="200" Height="50" Fill="Green">
        <Ellipse.RenderTransform>
            <RotateTransform Angle="30" />
        </Ellipse.RenderTransform>
    </Ellipse>

    <Rectangle Canvas.Left="100" Canvas.Top="50" Width="200" Height="50" Fill="Blue">
        <Rectangle.RenderTransform>
            <RotateTransform Angle="60" />
        </Rectangle.RenderTransform>
    </Rectangle>

    <Ellipse Canvas.Left="100" Canvas.Top="50" Width="200" Height="50" Fill="Violet">
        <Ellipse.RenderTransform>
            <RotateTransform Angle="90" />
        </Ellipse.RenderTransform>
    </Ellipse>
</Canvas>
</Window>
```

Listing 43.9 Gedrehte Rechtecke und Ellipsen

Bei der Ausführung einer Rotation ist nicht nur der Rotationswinkel von Bedeutung, sondern auch der Punkt, um den die einzelnen Objekte gedreht werden sollen. Im Beispiel aus Listing 43.9 ist das der obere, linke Eckpunkt der Rechtecke (bzw. Ellipsen). Sie können jedoch einen beliebigen Drehpunkt für die Rotation mit den Eigenschaften `CenterX` und `CenterY` im `RotateTransform`-Objekt angeben (Listing 43.10).



Abbildung 43.7 Gedrehte Rechtecke und Ellipsen

Die Transformationen im XAML-Code können folgendermaßen angepasst werden:

```
<Rectangle.RenderTransform>
  <RotateTransform Angle="60" CenterX="100" CenterY="25" />
</Rectangle.RenderTransform>
```

Nun liegt der Drehpunkt genau in der Mitte des ersten, roten Rechtecks und die ausgegebene Grafik wird in Abbildung 43.8 gezeigt.

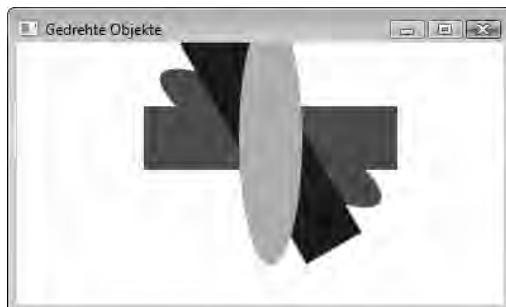


Abbildung 43.8 Der Drehpunkt liegt in der Mitte des blauen Rechtecks

Das Thema »Transformationen« ist hiermit aber noch nicht abgeschlossen. In den Eigenschaften `LayoutTransform` und `RenderTransform` stecken noch viel mehr Möglichkeiten, die dann an den entsprechenden Stellen vorgestellt werden.

Die Linie

Das einfachste grafische Element ist eine Linie (`Line`). Für eine Linie wird ein Anfangspunkt (x_1, y_1) und ein Endpunkt (x_2, y_2) definiert. Eine Füllfarbe ist natürlich nicht erforderlich. Mit der `Stroke`-Eigenschaft wird die Farbe der Linie festgelegt. Die Linienstärke wird mit der Eigenschaft `Thickness` gesetzt und die Strichelung mit `DashArray`. Bei Linienobjekten können Sie außerdem das Ende der Linien beeinflussen (Listing 43.10). Diese Möglichkeit ist besonders bei dicken Linien wichtig. In Abbildung 43.9 können Sie erkennen, dass das Linienende über den Anfangs- und Endpunkt der Linie hinausragt.

```
<Window x:Class="Linecap.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Linienenden definieren" Height="160" Width="300"
>
<Canvas>
    <Line X1="30" Y1="30" X2="250" Y2="30" Stroke="Black" StrokeThickness="20" />
    <Line X1="30" Y1="60" X2="250" Y2="60" Stroke="Black" StrokeThickness="20"
        StrokeStartLineCap="Round" StrokeEndLineCap="Round" />
    <Line X1="30" Y1="90" X2="250" Y2="90" Stroke="Black" StrokeThickness="20"
        StrokeStartLineCap="Triangle" StrokeEndLineCap="Triangle" />
</Canvas>
</Window>
```

Listing 43.10 Linienende definieren

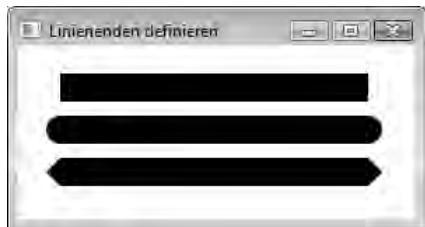


Abbildung 43.9 Verschiedene Linienenden

Die Polylinie

Mit dem Element **Polyline** können Sie mehrere Linien, die miteinander verbunden sind, zeichnen. Statt für jedes Liniensegment den Anfangs- und den Endpunkt anzugeben, können Sie hier einfach die einzelnen »Eckpunkte« der Gesamtlinie als Zahlenfeld mit der Eigenschaft **Points** definieren. Der Aufwand ist somit wesentlich geringer, wie auch das Beispiel »Fieberkurve« in Listing 43.11 und Abbildung 43.10 zeigt. Alle anderen, bereits bekannten Eigenschaften können Sie ganz normal benutzen.

```
<Window x:Class="Fieberkurve.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Fieberkurve" Height="180" Width="250"
>
<Canvas>
    <Polyline Stroke="Red" StrokeThickness="3"
        Points="10,110 20,50 30,20 60,25 75,20 100,120 115,50 135,60 165,65 200,90" />
</Canvas>
</Window>
```

Listing 43.11 Eine Linie aus vielen Teilstücken



Abbildung 43.10 Ein Polyline-Element als »Fieberkurve«

Das Polygon-Element ist dem Polyline-Element sehr ähnlich. Der einzige Unterschied ist, dass ein Polygon-Element immer geschlossen wird, d.h., der Anfangspunkt der ersten Linie wird mit dem Endpunkt der letzten Linie verbunden. Sie können das letzte Beispiel nehmen, und den XAML-Code geringfügig ändern (Listing 43.12). Das Ergebnis zeigt Abbildung 43.11.

```
<Canvas>
    <Polygon Stroke="Red" StrokeThickness="3"
        Points="10,110 20,50 30,20 60,25 75,20 100,120 115,50 135,60 165,65 200,90" />
</Canvas>
```

Listing 43.12 Eine geschlossene Kurve mit Polygon

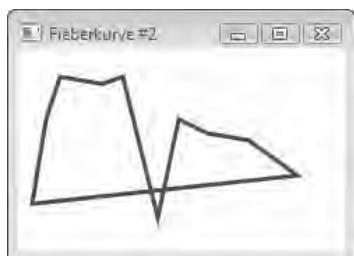


Abbildung 43.11 Geschlossene Kurve mit Polygon

Das automatische Schließen der Kurve vereinfacht viele Zeichenoperationen, da wir uns den Anfangspunkt der gesamten Linie nicht mehr merken müssen. Nun gibt es bei Polygonen noch eine weitere Besonderheit. Da Polygone immer geschlossen werden, kann man sie mit einer Füllfarbe ausfüllen. Aber das ist nicht ganz so einfach, wie es sich im ersten Moment anhört. Wenn wir den Polygon in Abbildung 43.11 betrachten, stellen wir fest, dass es gleich mehrere Flächenteile gibt, die ausgefüllt werden müssen.

Im ersten Versuch wollen wir das letzte Beispiel nur um eine `Fill`-Eigenschaft erweitern:

```
<Canvas>
    <Polygon Stroke="Red" StrokeThickness="3" Fill="Blue"
        Points="10,110 20,50 30,20 60,25 75,20 100,120 115,50 135,60 165,65 200,90" />
</Canvas>
```

Listing 43.13 Der Polygon soll gefüllt werden



Abbildung 43.12 Polygon mit Fill-Eigenschaft

Die Ausgabe von Listing 43.13 zeigt Abbildung 43.12: Alle Teilflächen des Polygons sind ausgefüllt worden. Für das Füllen von Polygonen gibt es zwei Möglichkeiten, die über die Eigenschaft `FillRule` eingestellt werden. Der Standardwert für diese Eigenschaft ist `EvenOdd`. Das bedeutet, dass nur solche Flächen gefüllt werden, von denen aus eine ungerade Anzahl von Linien bis nach außen überquert werden müssen. Schauen Sie sich einfach das folgende Beispiel in Listing 43.14 an:

```
<Window x:Class="FillRule1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Füllen" Height="175" Width="150"
    >
<Canvas>
    <Polygon Stroke="Red" StrokeThickness="6" Fill="Blue" FillRule="EvenOdd"
        Points="20,70 20,20 120,20 120,120 20,120 20,70 40,70 40,40 100,40 100,100 40,100 40,70" />
</Canvas>
</Window>
```

Listing 43.14 Die Füllregel EvenOdd

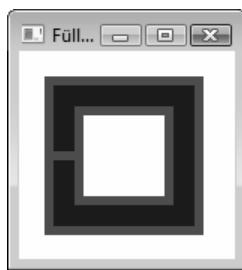


Abbildung 43.13 Die Füllregel EvenOdd

In einem Polygon-Element wurden in Abbildung 43.13 zwei Rechtecke mit der Füllregel `EvenOdd` ineinander gezeichnet. Dabei wurde das innere Rechteck nicht mit der Füllfarbe blau ausgefüllt. Wenn wir nun eine Hilfslinie aus dem inneren Rechteck nach außen (bis in die Unendlichkeit) legen (Abbildung 43.14, Linie 1), dann werden zwei Linien des Polygons überquert. Da »Zwei« bekanntermaßen eine gerade Zahl ist, wird die innere Fläche nicht gefüllt. Wenn Sie dagegen aus dem Bereich zwischen dem inneren und dem äußeren Rechteck eine Linie bis in die Unendlichkeit zeichnen (Abbildung 43.14, Linie 2 oder 3), werden eine oder drei Polygon-Linien überschritten. Wir haben eine ungerade Zahl, d.h., die Fläche wird mit blauer Farbe ausgefüllt.

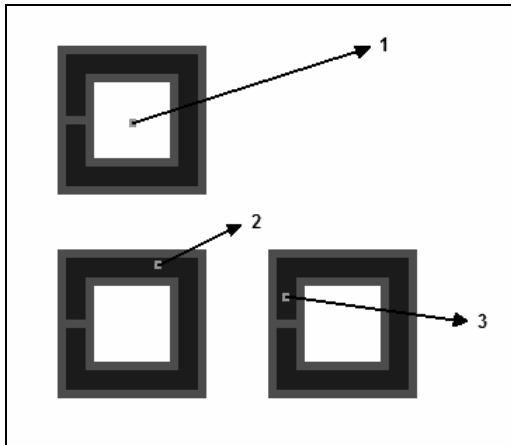


Abbildung 43.14 Die Füllregel EvenOdd

Die zweite Füllregel NonZero ist etwas schwerer verständlich. Für diese Regel benutzen wir ein etwas anderes Beispiel, welches in Listing 43.15 dargestellt ist.

```
<Window x:Class="FillRule2.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Füllen" Height="200" Width="350"
>
<Canvas>
    <Polygon Stroke="Red" StrokeThickness="4" Fill="Blue" FillRule="EvenOdd"
        Points="20,50 20,100 70,100 70,20 130,20 130,70 50,70 50,130 100,130 100,50" />
    <Polygon Stroke="Red" StrokeThickness="4" Fill="Blue" FillRule="NonZero"
        Points="170,50 170,100 220,100 220,20 280,20 280,70 200,70 200,130 250,130 250,50" />
</Canvas>
</Window>
```

Listing 43.15 Die Füllregeln EvenOdd und NonZero

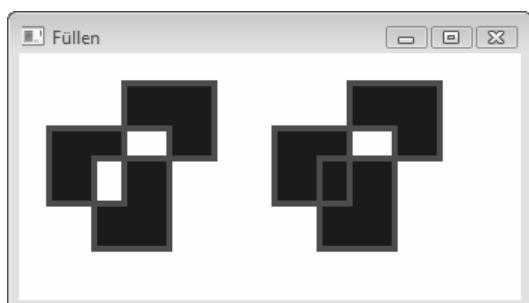


Abbildung 43.15 Die Füllregeln EvenOdd und NonZero

In Abbildung 43.15 sehen Sie auf der linken Seite einen Polygon mit der Füllregel EvenOdd, rechts dagegen ist der gleiche Polygon mit der Regel NonZero dargestellt. Das Ergebnis mit EvenOdd ist klar, wenn wir wieder die Hilfslinien gedacht nach außen ziehen.

Die Füllregel `NonZero` liefert bei einer ungeraden Anzahl von Linienüberquerungen das gleiche Ergebnis beim Füllen wie die Regel `EvenOdd`. Wenn jedoch eine gerade Anzahl von Linienüberquerungen notwendig ist, wird die Teilfläche nur dann gefüllt, wenn die Anzahl der Linien, die in eine bestimmte Richtung relativ zur Grafik verlaufen, ungleich der Anzahl der Linien in die andere Richtung sind. Abbildung 43.16 stellt dieses Verhalten etwas genauer dar. Die Zeichenrichtung für die entscheidenden Linien ist dort durch Pfeile dargestellt.

Die Hilfslinie Nr. 1 in Abbildung 43.16 überquert zwei Linien, welche in die gleiche Richtung führen. Da nun in die gegensätzliche Richtung gar keine Linie verläuft, wird der Teilbereich mit der angegebenen Füllfarbe ausgefüllt. Bei der Hilfslinie Nr. 2 werden zwei Linien überquert, von denen eine nach rechts und die andere Linie nach links verläuft. Damit ist die Anzahl der Linien, die jeweils in eine Richtung verläuft, gleich und die Teilfläche wird nicht gefüllt.

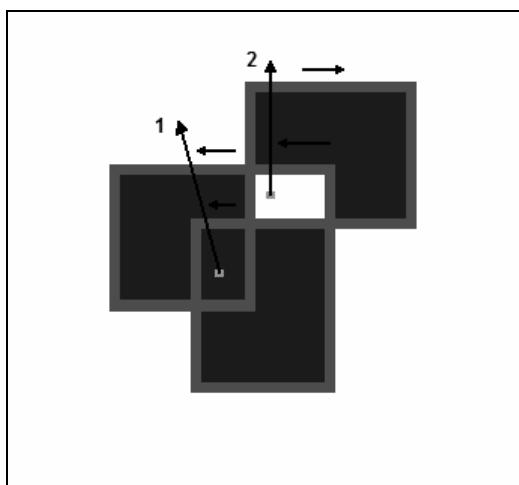


Abbildung 43.16 Die Füllregel `NonZero`

Das Path-Element

Ein weiteres grafisches Grundelement ist das Path-Element. Es ist das leistungsfähigste 2D-Grafikobjekt in WPF. Mithilfe des Path-Elementes können Sie sehr komplexe Figuren aufbauen und darstellen. Das Path-Element enthält die Eigenschaft `Data`, über die Sie ein oder mehrere grafische Grundelemente definieren können. Die Elemente, die Sie dort benutzen können, heißen `RectangleGeometry`, `EllipseGeometry`, `LineGeometry`, usw. Mehrere Grafikelemente können in einem Path mit einer `GeometryGroup` zusammengefasst werden. Listing 43.16 zeigt ein Beispiel.

```
<Window x:Class="Path1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Ein grafischer Pfad" Height="300" Width="300"
>
<Grid>
    <Path Stroke="Red" StrokeThickness="3" Fill="Yellow" >
        <Path.Data>
```

```
<GeometryGroup>
    <RectangleGeometry Rect="0, 0, 150, 100" />
    <EllipseGeometry Center="75, 50" RadiusX="75" RadiusY="50" />
    <LineGeometry StartPoint="0, 0" EndPoint="150,100" />
    <LineGeometry StartPoint="0, 100" EndPoint="150, 0" />
</GeometryGroup>
</Path.Data>
</Path>
</Grid>
</Window>
```

Listing 43.16 Mit einem Path-Objekt komplexere Grafiken definieren



Abbildung 43.17 Das Path-Element mit Ellipse, Rechteck und Linien

Für das Path-Element steht ebenfalls die Eigenschaft `Fill` zur Verfügung, um eine Füllfarbe für die gesamte Figur zu definieren. Im Element `GeometryGroup` dagegen befindet sich die Eigenschaft `FillRule`, die wir aus den vorherigen Beispielen kennen und die sich auch genau so verhält. In Abbildung 43.17 wird die innere Ellipse nicht gelb ausgefüllt, weil die Eigenschaft `FillRule` standardmäßig auf `EvenOdd` gesetzt ist und die Anzahl der Linien, die von innen nach außen durchquert werden müssen, gerade ist.

Innerhalb eines Path-Elements können Sie sehr komplexe Figuren definieren. In Listing 43.17 wird eine PathGeometry definiert. Diese kann mehrere Figuren beinhalten, die bei Bedarf auch geschlossen werden können. Jede Figur wiederum besteht aus einem Startpunkt und mehreren Segmenten, welche dann die gewünschte Figur aufbauen. Die Segmente (Tabelle 43.1) können Linien, Bögen oder auch komplexe Kurven sein. Diese einzelnen Segmente werden fortlaufend aneinander gezeichnet. In der Eigenschaft PathFigure.Segments können Sie beliebig viele, unterschiedliche Segmente für eine Figur definieren.

```

        <LineSegment Point="200,70" />
        <LineSegment Point="250,100" />
        <LineSegment Point="250,150" />
        <ArcSegment Point="200,200" Size="50,50" SweepDirection="Clockwise"/>
    </PathSegmentCollection>
</PathFigure.Segments>
</PathFigure>
</PathFigureCollection>
</PathGeometry.Figures>
</PathGeometry>
</Path.Data>
</Path>
</Canvas>
</Window>

```

Listing 43.17 Eine komplexe Figur mit einem Path-Element

Segmenttyp	Verhalten
LineSegment	Einfache gerade Linie
PolyLineSegment	Mehrere gerade Linien
ArcSegment	Bogensegment
BezierSegment	Kubische Bezierkurve
QuadraticBezierSegment	Quadratische Bezierkurve
PolyBezierSegment	Mehrere kubische Bezierkurven
PolyQuadraticBezierSegment	Mehrere quadratische Bezierkurven

Tabelle 43.1 Die möglichen Segmenttypen in einem Path-Element

Hit-Testing mit dem Path-Element

Wenn wir komplexe Grafiken mit dem Path-Element erzeugen, ergibt sich schnell die Frage, ob wir einfach feststellen können, wann in eine geschlossene Kurve hineingeklickt wurde. Eine einfache Demo kann den Sachverhalt klären. Wir erweitern hierzu das Beispiel aus Listing 43.17 und implementieren das MouseDown-Ereignis für das Path-Element:

```
<Path Fill="Blue" Stroke="Black" StrokeThickness="3" MouseDown="OnMouseDownPath">
```

Die Ereignismethode wird in VB implementiert und gibt einfach nur eine MessageBox aus, wenn das Path-Element mit dem Mausklick getroffen wurde.

```

Imports System
Imports System.Windows
Imports System.Windows.Input

Namespace Path4
    Partial Public Class Window1

```

```
Inherits System.Windows.Window

Public Sub New()
    InitializeComponent()
End Sub

Private Sub OnMouseDownPath(ByVal sender As Object, ByVal e As MouseEventArgs)
    MessageBox.Show("Im Path-Element!")
End Sub
End Class
End Namespace
```

Wenn Sie das Beispielprogramm laufen lassen, merken Sie sofort, dass nur Klicks innerhalb des Path-Elements die Meldung erscheinen lassen. Dadurch ist das Hit-Testing in diesem Fall sehr einfach zu implementieren.

Genauso einfach funktioniert das übrigens auch mit Polygon-Elementen. Hier werden beim Hit-Testing im Polygon die Einstellungen der FillRule-Eigenschaft berücksichtigt. Wenn ein Bereich des Polygons mit der gewählten Füllfarbe gezeichnet wird, dann wird auch das MouseDown-Ereignis in diesem Bereich ausgelöst (siehe Abbildung 43.15). In den Bereichen, die in Abbildung 43.15 nicht in blau eingefärbt sind, wird auch das MouseDown-Ereignis nicht ausgelöst. In den Polygon-Elementen wird nur das Mausereignis implementiert, das dann im Code verarbeitet werden kann:

```
<Polygon Stroke="Red" StrokeThickness="4" Fill="Blue" FillRule="EvenOdd" MouseDown="OnMouseDown"
        Points="20,50 20,100 70,100 70,20 130,20 130,70 50,70 50,130 100,130 100,50" />
<Polygon Stroke="Red" StrokeThickness="4" Fill="Blue" FillRule="NonZero" MouseDown="OnMouseDown"
        Points="170,50 170,100 220,100 220,20 280,20 280,70 200,70 200,130 250,130 250,50" />
```

Es gibt jedoch auch komplizierte Fälle beim Hit-Testing, welche die Implementierung von etwas mehr Code erfordern. In den oben gezeigten Beispielen werden nur die Mausklickpositionen für den Hit-Test herangezogen. Wir wollen ein Beispiel nun so abändern, dass ein kreisförmiger Bereich um die Klickposition herum als Testbereich verwendet wird.

Der XAML-Code mit einem Path-Element für das erweiterte Hit-Testing wird in Listing 43.18 gezeigt. Das MouseDown-Ereignis wird in diesem Beispiel im Hauptfenster Window1 »eingehängt«, damit wir das Ereignis auch dann auswerten können, wenn nicht direkt in das Path-Element geklickt wurde. Im Code (Listing 43.19) wird in der Mausereignis-Methode zunächst die Position des Klicks ermittelt. Dann wird ein EllipseGeometry-Objekt angelegt, in dem ein Kreis von 20 Einheiten Durchmesser definiert wird. Nun kann die statische Methode HitTest aus der VisualTreeHelper-Klasse benutzt werden, um im Trefferfall eine Rückrufmethode mit dem Namen HitResult aufzurufen. Dort wird das Hit-Testergebnis ausgewertet. Mit der Eigenschaft IntersectionDetail aus der Klasse GeometryHitTestResult können wir dann ermitteln, ob der angelegte Kreis vollständig, teilweise oder gar nicht im definierten Path-Element liegt. Nachdem ein Treffer erzielt wurde, können Sie die Suche fortführen (HitTestResultBehavior.Continue) oder abbrechen (HitTestResultBehavior.Stop). Eine Steuerung der Suche über die Aufzählung HitTestResultBehavior ist besonders dann interessant, wenn mehrere grafische Objekte übereinander liegen und ermittelt werden soll, ob ein ganz bestimmtes Grafikobjekt angeklickt wurde. Bei der Benutzung von Continue wird die Rückrufmethode für das nächste getroffene Objekt erneut aufgerufen. Dort findet dann die weitere Verarbeitung der Treffer-Informationen statt. Eine Verwendung von Stop beendet die laufende Trefferanalyse und die Rückrufmethode wird nicht mehr aufgerufen, auch wenn es noch weitere Trefferobjekte geben sollte.

```
<Window x:Class="HitTesting.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hit-Testing" Height="300" Width="300" MouseDown="OnMouseDown"
>
<Grid>
    <Path Name="path" Stroke="Black" StrokeThickness="3" Fill="Red">
        <Path.Data>M 50,50 h 130 l 50,50 v 100 h -100 v -50 h 50 Z</Path.Data>
    </Path>
</Grid>
</Window>
```

Listing 43.18 Erweitertes Hit-Testing

```
Imports System
Imports System.Windows
Imports System.Windows.Input
Imports System.Windows.Media

Namespace HitTesting
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnMouseDown(ByVal sender As Object,
                               ByVal e As MouseEventArgs)
            ' Mausposition ermitteln
            Dim pt As Point = e.GetPosition(DirectCast(sender, UIElement))
            ' Fläche für Trefferbereich festlegen
            Dim newHitTestArea As EllipseGeometry = New EllipseGeometry(pt, 20.0, 20.0)
            ' Rückrufprozedur für einen Treffer einrichten
            VisualTreeHelper.HitTest(path, Nothing,
                                     New HitTestResultCallback(AddressOf HitResult),
                                     New GeometryHitTestParameters(newHitTestArea))
        End Sub

        Public Function HitResult(ByVal result As HitTestResult) As HitTestResultBehavior

            ' Ermitteln des Hit-Test-Ergebnisses
            Select Case DirectCast(result, GeometryHitTestResult).IntersectionDetail
                Case IntersectionDetail.FullyInside
                    MessageBox.Show("Vollständig innen!")
                    Return HitTestResultBehavior.Continue
                Case IntersectionDetail.FullyContains
                    MessageBox.Show("Voll getroffen!")
                    Return HitTestResultBehavior.Stop
                Case IntersectionDetail.Intersects
                    MessageBox.Show("Teilweise getroffen!")
                    Return HitTestResultBehavior.Continue
            End Select
            Return HitTestResultBehavior.Stop
        End Function
    End Class
End Namespace
```

Listing 43.19 Das erweiterte Hit-Testing

Zusammenfassung

Im vorliegenden Kapitel haben Sie erfahren, wie vielfältig die verschiedenen Grafikelemente von Windows Presentation Foundation eingesetzt werden können. Entscheidend bei der Darstellung einer Benutzeroberfläche ist die einfache Kombination von Steuerelementen und grafischen Elementen.

Viele interessante Vereinfachungen werden durch Anwendung der Path-Klasse ermöglicht. Die Definition und Zusammenfassung von grafischen Figuren wird mit einem Path-Element stark vereinfacht.

Entscheidende Design-Möglichkeiten werden aber durch überall anwendbare Transformationen geschaffen. Es ist ohne großen Programmieraufwand mit WPF möglich, einerseits moderne Layouts für Spielprogramme zu definieren, aber andererseits auch eine vollständig in der Größe einstellbare Benutzeroberfläche zu erstellen, die einen hohen Komfort beim Arbeiten mit dieser Applikation ermöglicht.

Teil I

Anwendungsausführung parallelisieren

In diesem Teil:

- Einführung in die Technik des Threading
- Threading-Techniken

1311

1315

Kapitel 44

Einführung in die Technik des Threading

Dass Gleichzeitigkeit eigentlich eine Illusion ist, hat Einstein mit seiner Relativitätstheorie bewiesen.¹ Wenn auch aus anderen Gründen, besteht dennoch der Eindruck, dass ein normaler Computer Dinge wirklich gleichzeitig erledigen könnte. Auch wenn er im Hintergrund Robbie Williams spielt, eine seiner Festplatten defragmentiert und mich gleichzeitig diese Zeilen mit Word schreiben lässt, so zerlegt er diese drei Sachen in klitzekleine Aufgaben und arbeitet sie im Grunde genommen nacheinander ab. Aus der Geschwindigkeit, mit der er diese kleinen Dinge hintereinander macht, entsteht dann der Eindruck, er mache sie wirklich gleichzeitig.

Ausnahmen davon bilden Multiprozessor- oder Multicore-Systeme, die bestimmte Aufgaben tatsächlich gleichzeitig erledigen können. Anmerkung am Rande: Die so genannten *Hyperthreading*-Prozessoren von Intel nehmen dabei eine Art Zwitterstellung ein – sie nutzen Pausen, die der Prozessor von Zeit zu Zeit einlegen muss, wenn er beispielsweise auf Daten aus dem Hauptspeicher wartet, um schon mal Teile anderer Threads zu verarbeiten, sodass dieser Prozessor auf unterster Ebene betrachtet Gleichzeitigkeit ziemlich perfekt vortäuscht. In der Tat führt das zu einer Leistungssteigerung von durchschnittlich 10 % bis zu ca. maximal und unter günstigsten Umständen 20 %, doch im Grunde genommen arbeiten auch Hyperthreading-Prozessoren die zu erledigenden Threads auch nur nacheinander ab. Echte Gleichzeitigkeit ist allein Multiprozessor- bzw. Multicore-Systemen vorbehalten.

Doch gerade die letzten Systemtypen machen genau in diesem Moment, in dem die Zeilen entstehen, Schluss mit dem Taktfrequenzrennen der verschiedenen Prozessorhersteller. Da die Grenze des physikalisch Möglichen quasi erreicht ist, können Prozessoren nicht mehr wesentlich schneller werden: man kann Ihnen nur beibringen, mehrere Dinge wirklich gleichzeitig zu machen, und genau dahin geht der Trend. Dummerweise bedeutet es nicht automatisch, dass ein Prozessor, der im Grunde genommen zwei oder vier Prozessoren in sich vereint, auch automatisch das zwei- bzw. vierfache an Leistung bringt. Nur, wenn eine Software auch in der Lage ist, die beiden Prozessorkerne auch wirklich zu nutzen, indem sie eine oder mehrere Aufgaben (so genannte *Tasks*) in verschiedene Threads verlagert, die dann wiederum unabhängig auf mehreren Prozessorkernen laufen können, können Sie als Anwender (und als Entwickler zeitkritischer Systeme) auch wirklich von dieser neuen Technologie profitieren.

Sie sehen also, dass dieses Kapitel eines der wirklich wichtigen Themen der kommenden Zeit behandelt.

Unter dem Namen »Multitasking« hat wohl jeder, der sich nur ein wenig mit Computern beschäftigt, diese Fähigkeit schon einmal kennen gelernt. Multitasking ist die Kombination der englischen Wörter »multi« – für »viel« – und »task« – für »Aufgabe« – und bedeutet im Deutschen das, was Frauen beneidenswerter und normalerweise eher können als Männer: nämlich mehrere Dinge zur gleichen Zeit erledigen.

Ein weiterer, ähnlicher Begriff, stammt ebenfalls aus dem Englischen, aber er ist nicht ganz so bekannt wie »Multitasking«. Gemeint ist »Multithreading«², wieder abgeleitet von »multi« – für »viel« – nur im zweiten Teil des Wortes diesmal von »thread« – für »Faden«. Man könnte eine Multithreading-fähige Anwendung also als »mehrädiges« Programm bezeichnen, wollte man den Ausdruck unbedingt übersetzen.

¹ Siehe auch die Folge von Alpha Centauri (vom 10.6.2001), zu erreichen über den IntelliLink **I4401**.

² Ausgesprochen etwa »Maltibrädding«, und wenn Sie es ganz perfekt machen wollen, lispen Sie das scharfe S.

Und was bedeutet dieser Ausdruck jetzt genau? Dazu folgender Hintergrund: wenn Sie eine Windows-Applikation starten, dann besteht sie aus mindestens einem so genannten Thread. In diesem Zusammenhang ist »Thread« eine abstrakte Bezeichnung für einen bestimmten Programmverlauf. Ein Thread startet, löst eine bestimmte Aufgabe und wird beendet, wenn diese Aufgabe erledigt ist. Wenn diese Aufgabe sehr rechenintensiv ist, dann bedeutet das in der Regel für das Programm, dass es nicht weiter bedienbar ist. Der aktuelle Thread beansprucht die gesamte Rechenleistung eines Prozessorkerns (und wenn der Computer nur über einen Prozessorkern verfügt, seine komplette Rechenleistung), sodass für die Behandlung der Bedienungselemente nichts mehr übrig bleibt, wie das folgende kleine Beispielprogramm eindrucksvoll zeigt:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\ I - Threading\\Kapitel44\\SingleThreading

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Programm macht nichts anderes, als eine Zählvariable bis auf einen bestimmten Wert hinaufzuzählen und den aktuellen Wert im Label-Steuerelement anzuzeigen.

```
Public Class Form1

    Private Sub btnZählenStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnZählenStarten.Click
        For z As Integer = 0 To 100000
            lblAusgabe.Text = z.ToString
            lblAusgabe.Refresh()
        Next
    End Sub
End Class
```

Wenn Sie dieses Programm gestartet haben, klicken Sie auf die Schaltfläche *Zählen starten*.

Ein Blick in den Task-Manager offenbart, dass es fast alles an Prozessorleistung verschlingt. Da es darüber hinaus keine Anstalten macht, das Formular die Warteschleife verarbeiten zu lassen, lässt es sich, während es läuft, auch nicht bedienen – Sie können das Programmfenster nicht verschieben oder schließen; obwohl die Zählung läuft, scheint es zu hängen.

Sie sehen: In dem Moment, in dem Sie Ihr Programm eine umfangreiche Verarbeitung von Daten durchführen lassen müssen, sollten Sie auf eine andere Technik ausweichen, damit andere Funktionen des Programms nach wie vor zur Verfügung stehen können. Und hier kommt die Thread-Programmierung ins Spiel. Threads sind Programmteile, die unabhängig voneinander und quasi gleichzeitig operieren können. So könnte beispielsweise die eigentliche Zählschleife des Programms in einem eigenen Thread laufen. Sie würde in diesem Fall parallel zum eigentlichen Programm ausgeführt werden. Das Programm könnte sich dann nicht nur um seine Nachrichtenwarteschlange kümmern und das Programm so bedienbar halten, sondern sich zusätzlich um die Ausführung weiterer Aufgaben kümmern.

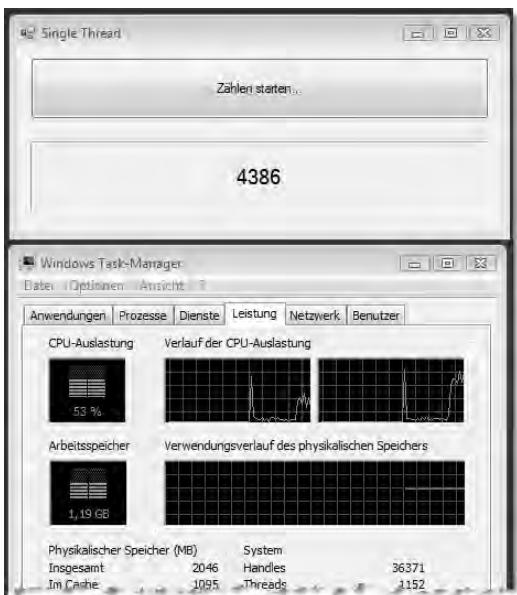


Abbildung 44.1 Sobald Sie das Programm starten, verschlingt es die komplette Prozessorleistung eines Prozessorkerns. Darüber hinaus lässt die Schleife keine Nachrichtenauswertungen zu, sodass sich das Programm während des Zählens nicht anderweitig bedienen lässt.

Beispiele, wann Threads »angesagt« sind, gibt es viele:

- Ihr Programm muss eine umfangreiche Auswertung von Daten zusammenstellen und drucken. Diese Aufgabe könnte in einem eigenen Thread im Hintergrund passieren, ohne dass es den weiteren Arbeitsablauf des Programms stören würde.
- Ihr Programm muss umfangreiche Datensicherungen durchführen. Ein Thread könnte Momentaufnahmen dieser Dateien erstellen und sie anschließend sozusagen im Hintergrund auf eine andere Ressource kopieren.
- Ihr Programm muss Zustände bestimmter Hardwarekomponenten aufzeichnen – beispielsweise Produktionsdaten von Maschinen abrufen. Auch diese Funktion könnte im Hintergrund ablaufen; Auswertungsfunktionen könnten dennoch zu jeder Zeit parallel laufen und vom Anwender verwendet werden, wenn die Produktionsdatenerfassung als Thread im Hintergrund läuft.
- Bei komplizierten Berechnungen oder Auswertungen könnten zwei oder mehrere Threads diese Aufgabe übernehmen. In diesem Fall würde auf Hyperthreading- oder Multiprozessorsystemen eine Auslastung mehrerer Prozessoren durch *jeweils* einen eigenen Thread die Verarbeitungsgeschwindigkeit der gesamten Aufgabe deutlich erhöhen.

Anwendungen gibt es also viele, um Threads einzusetzen. Allerdings gibt es bei Threads auch ein paar Dinge, die beachtet werden müssen, denn: Sie dürfen sich nicht gegenseitig ins Gehege kommen. Doch dazu später mehr.

Betrachten wir zunächst die Grundlagen, also wie wir das Framework überhaupt dazu bewegen können, dass bestimmte Teile einer Anwendung als eigener Thread ausgeführt werden können.

HINWEIS Sollten Sie einfach nur die Anforderung haben, eine bestimmte Aufgabe in einem anderen Thread erledigen zu lassen, und möchten Sie dazu nicht tiefer in die Materie einsteigen, empfehle ich Ihnen den Einsatz der `BackgroundWorker`-Komponente, die Sie im entsprechenden Abschnitt des nächsten Kapitels beschrieben finden.

Kapitel 45

Threading-Techniken

In diesem Kapitel:

Gleichzeitige Codeausführung mithilfe eines Thread-Objektes	1316
Synchronisieren von Threads	1319
Verwenden von Steuerelementen in Threads	1331
Managen von Threads	1333
Datenaustausch zwischen Threads durch Kapseln von Threads in Klassen	1338
Verwenden des Thread-Pools	1349
Thread-sichere Formulare in Klassen kapseln	1354
Threads durch den Background-Worker initiieren	1357
Threads durch asynchrone Aufrufe von Delegaten initiieren	1360

Gleichzeitige Codeausführung mithilfe eines Thread-Objektes

Das folgende Beispiel zeigt am einfachsten Beispiel, wie Sie eine Prozedur Ihrer Klasse mithilfe der Thread-Klasse als Thread parallel zum aktuellen Programm ablaufen lassen können. Alle weiteren Techniken des Threading wird dieses Kapitel übrigens anhand dieser Klasse besprechen.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\I - Threading\\Kapitel45\\SimpleThread01

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

WICHTIG In den Beispielen dieses Kapitels finden Sie des Öfteren eine Klasse, die die Ausgabe von Zeilen in einem speziellen Dialog ermöglicht. Diese Klasse nennt sich `ADThreadSafeInfoBox`, und sie stellt die Methoden `TSWrite` und `TSWriteLine` zur Verfügung. Sie ist für die Verwendung in Windows Forms-Anwendungen gedacht, und um sie zu verwenden, müssen Sie sie nicht instanzieren, sondern können ihre Funktionen wie die der `Console`-Klasse direkt verwenden, da sie statisch sind. Ihre Verwendung ist notwendig, da die Aktualisierung von Steuerelementen in Formularen aus Thread-Prozeduren eine besondere Vorgehensweise erforderlich macht. Auf dieses Thema werde ich jedoch am Ende dieses Kapitels genauer eingehen. Für den Moment arbeiten Sie mit der Klasse einfach so, als wäre sie fest im Framework vorhanden.

Wenn Sie dieses Programm starten, sehen Sie einen simplen Dialog, der lediglich aus zwei Schaltflächen besteht. Sobald Sie die Schaltfläche *Thread starten* anklicken, öffnet sich ein weiteres Fenster, in dem ein Wert von 0 bis 50 hoch gezählt wird. Soweit ist das noch nichts Besonderes. Allerdings können Sie die Schaltfläche ein weiteres Mal anklicken, um einen weiteren Thread zu starten. Auch der zweite Thread führt die Zählung durch, und das Ausgabefenster zeigt dabei die Ergebnisse beider Zahlenfolgen an – etwa wie in Abbildung 44.1 zu sehen:

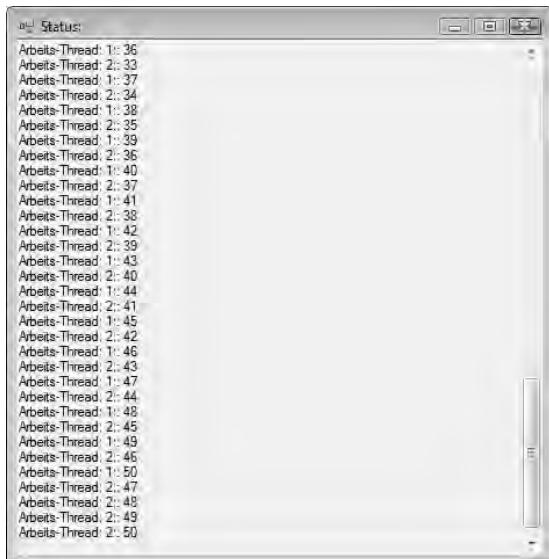


Abbildung 44.1 Zwei Threads laufen parallel und teilen sich das Ausgabefenster, um Ergebnisse ihres Schaffens anzuzeigen

Soweit, so gut. Nun lassen Sie uns als nächstes den Code betrachten, der dieses Ergebnis im Ausgabefenster Zustand bringt:

```
Imports System.Threading

Public Class frmMain

    'Member-Variable, damit die Threads durchnummiert werden können.
    'Dient nur zur späteren Unterscheidung des laufenden Threads, wenn
    'er Ergebnisse im Ausgabefenster darstellt.
    Private myArbeitsThreadNr As Integer = 1

    'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
    'Hochzählen und die Werteausgabe übernimmt
    Private Sub UmfangreicheBerechnung()
        For c As Integer = 0 To 50
            'Dient zur Ausgabe des Wertes. TSWriteLine ist eine statische
            'Prozedur, die für die Darstellung des Fensters selbst sorgt,
            'sobald sie das erste Mal verwendet wird.
            ADThreadSafeInfoBox.TSWriteLine(Thread.CurrentThread.Name + ":: " + c.ToString)
            'Aktuellen Thread um 100ms verzögern, damit die ganze
            'Geschichte nicht zu schnell vorbei ist.
            Thread.Sleep(100)
        Next
    End Sub

    Private Sub btnThreadStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnThreadStarten.Click
        'Dieses Objekt kapselt den eigentlichen Thread
        Dim locThread As Thread
        'Dieses Objekt benötigen Sie, um die Prozedur zu bestimmen,
        'die den Thread ausführt.
        Dim locThreadStart As ThreadStart

        'Threadausführende Prozedur bestimmen
        locThreadStart = New ThreadStart(AddressOf UmfangreicheBerechnung)
        'ThreadStart-Objekt dem Thread-Objekt übergeben
        locThread = New Thread(locThreadStart)
        'Thread-Namen bestimmen
        locThread.Name = "Arbeits-Thread: " + myArbeitsThreadNr.ToString
        'Thread starten
        locThread.Start()
        'Zähler, damit die Threads durch ihre Namen unterschieden werden können
        myArbeitsThreadNr += 1
    End Sub

    Private Sub btnBeenden_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
        btnBeenden.Click
        'Einfach so geht's normalerweise nicht
        Me.Close()
    End Sub
End Class
```

Starten von Threads

Wie aus dem Listing des vorherigen Beispielprogramms ersichtlich, benötigen Sie zwei Objekte, um einen Thread tatsächlich zum Laufen zu bringen: das Thread- und das ThreadStart-Objekt. Das ThreadStart-Objekt dient lediglich dazu, die Adresse der Prozedur aufzunehmen, die als Thread ausgeführt werden soll. Sie können es als Delegaten mit besonderen Eigenschaften betrachten, der darauf ausgerichtet ist, mit dem eigentlichen Thread-Objekt zusammenzuarbeiten. Nachdem Sie das ThreadStart-Objekt instanziert und ihm dabei die Thread-Prozedur mit AddressOf zugespielt haben, übergeben Sie die generierte Instanz dem Thread-Konstruktor. Haben Sie auch dieses Objekt erzeugt, starten Sie den Thread mit der Start-Methode.

Grundsätzliches über Threads

Jedes Thread-Objekt, das Sie auf die beschriebene Weise erzeugt haben, speichert spezifische Informationen über den eigentlichen Thread. Das ist notwendig, da Windows auf Basis des so genannten *Preemptive Multitasking*¹ arbeitet, bei dem Prozessorzeit den verschiedenen Threads durch das Betriebssystem zugewiesen wird.² Wenn das Betriebssystem bestimmt, dass es Zeit für die Ausführung eines bestimmten Threads wird, müssen beispielsweise der komplette Zustand der CPU-Register für den laufenden Thread gesichert und der ursprüngliche Zustand der Register für den als nächstes auszuführenden Thread wiederhergestellt werden. Diese Daten werden unter anderem in einem bestimmten Speicherbereich gesichert, den das Thread-Objekt kapselt. Sie werden für gewöhnlich auch als *Thread Context* bezeichnet.

Thread-Prozeduren sind im Grunde genommen nichts Besonderes. Eine Thread-Prozedur ist grundsätzlich – wie jede andere Prozedur auch – ein Teil einer Klasse oder eines Moduls (das ja im Grunde genommen auch nichts weiter als eine Klasse mit nur statischen Funktionen ist). Die lokalen Variablen, die die Thread-Prozedur verwendet, sind für den jeweils ausgeführten Thread unterschiedlich. Anders ausgedrückt, können lokale Variablen eines Threads also nicht denen eines anderen ins Gehege kommen. Anders ist das beim Zugriff auf Klassen-Member: Für jeden Thread gilt dieselbe Instanz einer Member-Variablen, und dabei können sich besondere Probleme ergeben:

Stellen Sie sich vor, ein Thread greift auf eine Member-Variable der Klasse zu, um sie beispielsweise neu zu berechnen und anschließend auszugeben. Genau in dem Moment, in dem der erste Thread sie berechnet hat, aber noch bevor er dazu gekommen ist, das berechnete Ergebnis tatsächlich auf dem Bildschirm auszugeben, hat ein zweiter Thread die Berechnung mit dem Member abgeschlossen. In der Member-Variablen steht nun ein aus Sicht des ersten Threads völlig falsches Ergebnis, und das ausgegebene Ergebnis des ersten Threads ist ebenso falsch.

¹ Etwa »bevorrechtigt«, »präventiv«.

² Im Gegensatz dazu gibt es das so genannte *Cooperative Multitasking*, bei dem ein Thread selber bestimmt, wie viel Prozessorzeit er benötigt. Dadurch kann ein Thread, der in einer nicht enden wollenden Operation fest hängt, die Stabilität des gesamten Systems gefährden.

Synchronisieren von Threads

Damit Zugriffs- bzw. Synchronisationsprobleme verhindert werden können, gibt es eine ganze Reihe von Techniken, die in diesem Abschnitt besprochen werden sollen.

Die einfachste Methode erlaubt das automatische Synchronisieren eines Codeblocks auf Grund einer verwendeten Objektvariablen. Der folgende Abschnitt erläutert das Problem und dessen Lösung anhand eines konkreten Beispiels.

Synchronisieren der Codeausführung mit SyncLock

Das nächste Beispielprogramm lehnt sich an das des vorherigen Beispiels an – es ist nur ein wenig »eloquenter«. Lediglich um zu zeigen, inwieweit nicht synchronisierte Vorgänge beim Threading richtig daneben gehen können, modifiziert es die Ausgaberoutine der Thread-Routine auf folgende Weise:

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\I - Threading\\Kapitel45\\SimpleThread02 (SyncLock)

Öffnen Sie dort die entsprechende Projektmappe (SLN-Datei).

```
'Member-Variablen, mit der demonstrativ Mist gebaut wird...
Private myThreadString As String

'Dies ist der eigentliche Arbeits-Thread (auch "Worker Thread" genannt),
'der das Hochzählen und die Werteausgabe übernimmt.
Private Sub UmfangreicheBerechnung()

    Dim strTemp As String

    For c As Integer = 0 To 50
        strTemp = Thread.CurrentThread.Name + ":: " + c.ToString

        'Nehmen Sie die Auskommentierung von SyncLock zurück,
        'um den "Fehler" des Programms zu beheben.
        'SyncLock Me
        myThreadString = ""

        For z As Integer = 0 To strTemp.Length - 1
            myThreadString += strTemp.Substring(z, 1)
            Thread.Sleep(5)
        Next

        ADThreadSafeInfoBox.TSWriteLine(myThreadString)
        'End SyncLock

    Next
End Sub
```

Hier passiert folgendes: myThreadString ist eine Member-Variable der Klasse, und sie wird für jedes Zeichen, das im Ausgabefenster für einen Zählungseintrag erscheinen soll, Zeichen für Zeichen zusammengesetzt.

Die `TWriteLine`-Methode gibt diesen String anschließend aus, wenn das Zusammenbasteln des Strings für einen Eintrag erledigt ist.

Wenn Sie das Programm starten, und die Schaltfläche nur ein einziges Mal anklicken, sodass auch nur ein einziger Thread läuft, bleibt alles beim Alten. Doch wehe Sie starten, schon während der erste Threads läuft, auch nur einen weiteren, dann nämlich ist Chaos angesagt, wie auch in Abbildung 44.2 zu sehen.

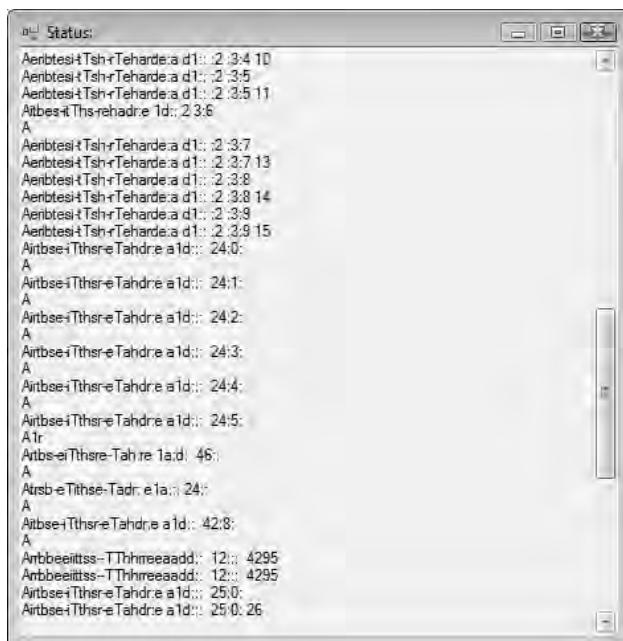


Abbildung 44.2 Zwei Threads im Kampf um eine Member-Variable – und das daraus resultierende und nicht wirklich zufriedenstellende Ergebnis

Das Problem ist, nicht genau voraussehen zu können, wann eine Thread-Prozedur zum Zug kommt – denn beim *Preemptive Multitasking* bestimmt diesen Zeitpunkt das Betriebssystem. Im hier vorliegenden Fall »meint« das Betriebssystem, dass ein anderer Thread am besten zum Zuge kommen kann, wenn der eine Thread gerade zu warten beginnt (was beide durch die `Sleep`-Methode in regelmäßigen Abständen machen).

Dieses Problem können Sie durch das Blockieren von Codeabschnitten in Abhängigkeit von verwendeten Objekten aus der Welt schaffen. Die `SyncLock`-Anweisung ist hier der Schlüssel zur Lösung. Wenn Sie den Code folgendermaßen umbauen, kann sich das Ergebnis wieder sehen lassen:

```
'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
'Hochzählen und die Werteausgabe übernimmt.
Private Sub UmfangreicheBerechnung()
    Dim strTemp As String

    For c As Integer = 0 To 50
        strTemp = Thread.CurrentThread.Name + ":: " + c.ToString
        SyncLock Me
            myThreadString = ""
            For z As Integer = 0 To strTemp.Length - 1
                myThreadString += strTemp.Substring(z, 1)
            Thread.Sleep(5)
```

```
    Next  
    'ADThreadSafeInfoBox.TSWriteLine(myThreadString)  
    Console.WriteLine(myThreadString)  
End SyncLock  
  
Next  
End Sub
```

SyncLock verwendet ein Objekt, um den nachfolgenden Code für alle anderen Threads zu schützen, die einen Verweis auf dasselbe Objekt halten. Dieses Objekt sollte daher für die Dauer des Zugriffs nicht verändert werden, da der Schutz sonst nicht oder nicht mehr zuverlässig funktioniert.

WICHTIG Im Gegensatz zu vielen vorherrschenden Meinungen schützt SyncLock nicht das Objekt selbst vor Veränderungen, sondern den Code gegen den gleichzeitigen Zugriff von einem anderen Thread. Natürlich kann jeder andere Thread das Objekt verändern und so die ganze Bemühung zur Synchronisation des Programms an dieser Stelle zunichte machen. Außerdem sollte am Rande erwähnt werden: Da SyncLock intern einen Try/Catch-Block verwendet und damit auch im Falle einer Ausnahme der Schutz des Blocks wieder aufgehoben werden kann, dürfen Sie nicht mit der Goto-Anweisung in einen SyncLock-Block springen.

Was passiert also im Detail bei der Ausführung dieser Routine? Nun, sobald der erste Thread den Block erreicht, sperrt er den Zugriff auf den Code für jeden weiteren Thread durch Zuhilfenahme des Objektes. Für dieses Objekt müssen folgende Bedingungen gelten:

- Es muss sich um ein Objekt handeln, das auf eine gültige Adresse im Managed Heap verweist – darf also nicht Nothing sein.
- Es muss eine Member-Variable sein, also jedem Thread den Zugriff darauf ermöglichen.
- Es muss zwingend ein Referenztyp sein (logisch, denn Wertetypen liegen nicht auf dem Managed Heap).
- Es darf durch die geschützte Routine nicht verändert werden.

Im Beispiel wird das einfach durch die Verwendung der eigenen Instanz Me erreicht. Me erfüllt diese Bedingungen stets. Bei statischen Routinen, bei denen Me natürlich nicht anwendbar ist, können Sie das Typ-Objekt der verwendeten Klasse verwenden, um es als Hilfe zum Schutz einer Routine zu verwenden, etwa:

```
.  
. .  
SyncLock (GetType(Klassenname))  
  
    'Hier steht der Code der zu schützenden Routine  
  
End SyncLock  
. .
```

Wenn nun ein zweiter Thread den geschützten Programmteil erreicht, dann wird er so lange in die Knie gezwungen, bis der Thread, der den Programmteil bereits durchläuft, End SyncLock erreicht hat.

In unserem Beispiel kann also der Member-String ohne Sorge zu Ende aufgebaut und anschließend verarbeitet werden. Dass ein anderer Thread den Member-String an dieser Stelle in irgendeiner Form verändert, ist ausgeschlossen, weil er auf alle Fälle zu warten hat.

HINWEIS Einen Programmteil, der besonderen Synchronisationsschutz benötigt bzw. erfährt, nennt man *kritischer Abschnitt* oder neudeutsch: *Critical Section*.

Mehr Flexibilität in kritischen Abschnitten mit der Monitor-Klasse

Die SyncLock-Funktion hat einen großen Nachteil: Sie ist unerbittlich. Wenn ein anderer Thread bereits einen kritischen Abschnitt durchläuft, dann warten alle anderen Threads am Anfang des kritischen Abschnitts – ob sie wollen oder nicht. Sie können dagegen nichts tun. Die Monitor-Klasse bietet an dieser Stelle die größere Flexibilität, da sie ein paar zusätzliche Methoden im Angebot hat, mit der ein Thread auch nachschauen kann, ob er warten müsste, wenn er einen kritischen Bereich beträte. Ein weiteres Beispiel soll das verdeutlichen:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

... \VB 2008 Entwicklerbuch\I - Threading\Kapitel45\SimpleThread03 (Monitor)

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
'Hochzählen und die Werteausgabe übernimmt.
Private Sub UmfangreicheBerechnung()

    Dim strTemp As String

    For c As Integer = 0 To 50
        strTemp = Thread.CurrentThread.Name + ":: " + c.ToString
        'Nehmen Sie die Auskommentierung von SyncLock zurück,
        'um den "Fehler" des Programms zu beheben.
        If Not Monitor.TryEnter(myLock, 1) Then
            ADThreadSafeInfoBox.TSWriteLine(Thread.CurrentThread.Name + " meldet: Gerade besetzt!")
            Thread.Sleep(50)
        Else
            myThreadString = ""
            For z As Integer = 0 To strTemp.Length - 1
                myThreadString += strTemp.Substring(z, 1)
                Thread.Sleep(5)
            Next
            ADThreadSafeInfoBox.TSWriteLine(myThreadString)
            Monitor.Exit(myLock)
        End If
    Next
End Sub
```

In dieser Version des Arbeits-Threads erfolgt die Synchronisation durch die Monitor-Klasse. Allerdings ist der Thread, der auf seinen Vorgänger warten muss, ein wenig ungeduldig. Bekommt er nicht innerhalb von einer Millisekunde Zugriff auf den kritischen Abschnitt, verarbeitet er den Code für die entsprechende Werterhöhung nicht, sondern gibt nur lapidar die Meldung *Threadname meldet: Gerade besetzt!* aus.

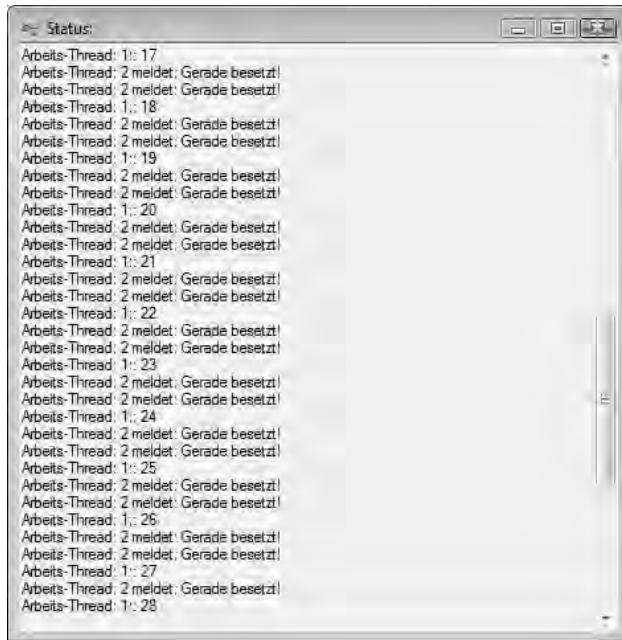


Abbildung 44.3 Ein Thread, der auf den kritischen Abschnitt nicht zugreifen kann, wartet maximal eine Millisekunde, bevor er mit einer lapidaren Meldung den Wartevorgang abbricht

Möglich wird das durch die Verwendung der TryEnter-Methode der Monitor-Klasse – die im Übrigen statisch ist; Sie brauchen die Monitor-Klasse also nicht zu instanzieren (Sie könnten es auch gar nicht). Diese Methode überprüft, ob der Zugriff auf einen kritischen Abschnitt, der durch ein angebautes Objekt genau wie bei SyncLock gesteuert wird, freigegeben ist. Ist er freigegeben, liefert sie True als Funktionsergebnis zurück. Ist er nicht freigegeben, ergibt sie False.

Auf diese Weise gibt es beim Ablauf des Programms Effekte, wie Sie sie auch in Abbildung 44.3 beobachten können.

Doch die Monitor-Klasse kann noch mehr, wie das folgende Beispiel zeigt.

Angenommen Sie möchten, dass im bislang gezeigten Beispielprogramm die durchgeföhrten Operationen in Fünferportionen durchgeführt werden. In diesem Fall benötigen Sie drei weitere Funktionen oder besser: Funktionalitäten:

- Sie benötigen eine Methode, die einen Thread in einen Wartezustand versetzt, wenn er sich innerhalb eines kritischen Bereichs befindet.
- Sie benötigen eine Methode, die einen Thread, der sich in einen Wartezustand versetzt hat, wieder aufwecken kann. Gibt es mehrere wartende Threads, sollte die Funktion in der Lage sein, nicht nur alle, sondern auch nur einen (beliebigen) Thread wieder zum Leben zu erwecken.

Diese Möglichkeiten bzw. Methoden gibt es. Sie heißen `Monitor.Wait`, `Monitor.PulseAll` und `Monitor.Pulse`. Für unser Vorhaben brauchen wir darüber hinaus eine Technik, die es uns ermöglicht, zwischen einem und mehreren laufenden Threads zu unterscheiden. Denn solange nur ein einzelner Thread läuft, darf der sich natürlich nicht in den Schlafmodus versetzen, da es keinen anderen gibt, der ihn wieder wach rütteln könnte. Doch diese Technik zu implementieren ist simpel: Eine einfache, statische Thread-Zählvariable vollführt diesen Trick. Wird ein neuer Thread gestartet, wird der Zähler zu Beginn der Thread-Prozedur hoch gezählt; beendet er seine Arbeitsroutine, zählt er ihn wieder runter. Er versetzt sich nur dann in Schlaf, wenn noch mindestens zwei Threads laufen. Getreu dem Motto »Der Letzte macht das Licht aus« muss jeder Thread vor dem Verlassen seiner Arbeitsprozedur noch überprüfen, ob es weitere Threads gibt und den jeweils letzten prophylaktisch aus seinem Dornrösenschlaf erwecken.

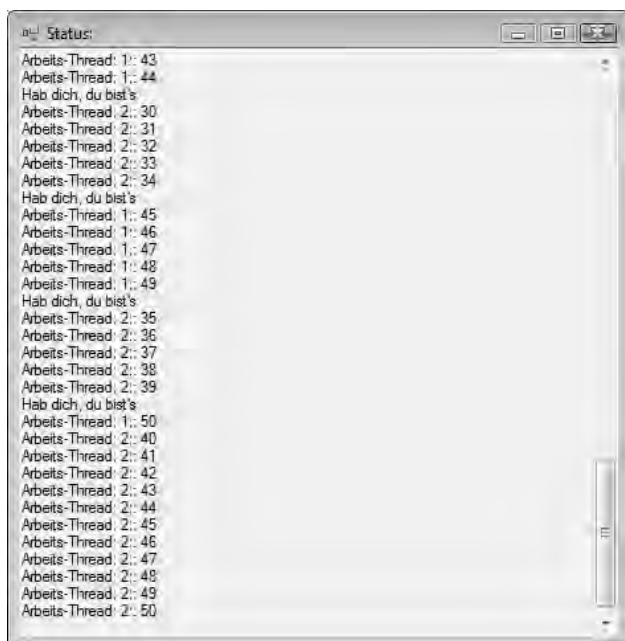


Abbildung 44.4 Ein Thread, der auf den kritischen Abschnitt nicht zugreifen kann, wartet maximal eine Millisekunde, bevor er mit einer lapidaren Meldung den Wartevorgang abbricht

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\I - Threading\\Kapitel45\\SimpleThread04 (Monitor Wait Pulse)

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie dieses Programm starten und mindestens zwei Mal auf die Schaltfläche zum Starten des Threads klicken, sehen Sie nach einer Weile ein Ergebnis, wie es in etwa dem in Abbildung 44.4 gezeigten entspricht.

Das entsprechende Listing des Arbeits-Threads sieht folgendermaßen aus:

```

'Die brauchen wir, um festzustellen, wie viele Threads unterwegs sind.
Private myThreadCount As Integer

'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das

```

```
'Hochzählen und die Werteausgabe übernimmt.
Private Sub UmfangreicheBerechnung()

    Dim strTemp As String
    Dim stepCount As Integer

    'Neuer Thread: Zählen!
    myThreadCount += 1

    'Hier beginnt der kritische Abschnitt.
    Monitor.Enter(Me)
    For c As Integer = 0 To 50
        strTemp = Thread.CurrentThread.Name + ":: " + c.ToString
        'Der Thread wartet maximal eine Sekunde; bekommt er in dieser
        'Zeit keinen Zugriff auf den Code, steigt er aus.
        'Zugriff wurde gewährt - jetzt nimmt der Arbeits-Thread
        'erst seine eigentliche Arbeit auf.
        myThreadString = ""
        For z As Integer = 0 To strTemp.Length - 1
            myThreadString += strTemp.Substring(z, 1)
            Thread.Sleep(1)
        Next
        ADThreadSafeInfoBox.TSWriteLine(myThreadString)
        'Der Thread darf sich nur dann schlafen legen, wenn mindestens
        'ein weiterer Thread unterwegs ist, der ihn wieder wecken kann.
        If myThreadCount > 1 Then
            stepCount += 1
            If stepCount = 5 Then
                stepCount = 0
                'Ablösung naht!
                Monitor.Pulse(Me)
                ADThreadSafeInfoBox.TSWriteLine("Hab dich, du bist's")
                'Abgelöster geht schlafen.
                Monitor.Wait(Me)
            End If
        End If
    Next
    If myThreadCount > 1 Then
        'Alle anderen schlafen zu dieser Zeit.
        'Also mindestens einen wecken, bevor dieser Thread geht.
        'Passiert das nicht, schlafen die anderen bis zum nächsten Stromausfall...
        Monitor.Pulse(Me)
    End If
    'Hier ist der kritische Abschnitt wieder vorbei.
    Monitor.Exit(Me)
    'Thread-Zähler vermindern.
    myThreadCount -= 1
End Sub
```

Synchronisieren von beschränkten Ressourcen mit Mutex

Wenn Threads lediglich theoretische Aufgaben lösen müssen, ist es fast egal, wie viele Threads gleichzeitig laufen (natürlich sollten dabei Sie berücksichtigen, dass Sie grundsätzlich nur so viele Threads wie gerade nötig verwenden sollten, da das Umschalten zwischen den Threads selbst natürlich auch nicht wenig an Rechenleistung verschlingt). Doch wenn bestimmte Anwendungen Komponenten der nur begrenzt vorhandenen Hardware benötigen, dann müssen diese Komponenten zwischen den verschiedenen Threads ideal aufgeteilt werden.

An dieser Stelle kommt die `Mutex`-Klasse ins Spiel. Ihr Vorteil: Sie funktioniert zwar prinzipiell wie die `Monitor`-Klasse, doch sie ist instanziierbar. Das bedeutet: Sie können durch `Mutex`-Instanzen, deren Anzahl Sie von der Verfügbarkeit benötigter Hardwarekomponenten abhängig machen, einen Verteiler organisieren, der sich um die Zuteilung der jeweils nächsten freien Hardware-Komponente kümmert.

Das folgende Beispiel demonstriert den Umgang mit `Mutex`-Klassen. Damit dieses Beispiel die `Mutex`-Klasse auch hardwareunabhängig auf jedem Rechner demonstrieren kann, ist es ein wenig anders aufgebaut als die Beispiele, die Sie bisher kennen gelernt haben: Drei Ausgabefelder innerhalb des Formulars dienen zur Emulation von drei Hardwarekomponenten; Ziel des Programms ist es, die laufenden Threads, die inhaltlich nichts anderes machen als das vorherige Beispielprogramm, so auf die drei Komponenten aufzuteilen, dass sie optimal genutzt werden.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\I - Threading\\Kapitel45\\SimpleThread05 (MutexDemo)

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie das Programm starten, sucht es sich nach dem Klick auf die Schaltfläche *Thread starten* die erste freie Hardware-Ressource (also ein Ausgabetextfeld). Ein weiterer Klick auf diese Schaltfläche startet einen weiteren Thread, der das nächste freie Ausgabefenster verwendet. Wenn Sie den vierten Thread starten, während alle vorherigen Threads noch laufen, wartet dieser auf das nächste freie Fenster. Sobald einer der Threads beendet ist, übernimmt er dessen Ausgabefenster. Die Arbeits-Thread-Routine ist so ausgelegt, dass eine einzelne Operation innerhalb des Threads unterschiedlich lange dauert. Nur so kann eine realistische Emulation einer simulierten Hardwarekomponente realisiert werden. Zu diesem Zweck gibt es ein im Konstruktor des Programms initialisiertes `Random`-Objekt, das den jeweils nächsten zufälligen Wert für die `Sleep`-Methode eines Arbeits-Threads generiert.

HINWEIS Das Programm gibt die Bildschirmmeldungen nicht direkt mit einer bestimmten Anweisung aus – so wie Sie es von den bisherigen Beispielen gewohnt waren. Stattdessen verwendet es drei `TextBox`-Steuerelemente für die Textausgabe.

HINWEIS Bitte beachten Sie, dass die Veränderung der Eigenschaften eines Steuerelements ausschließlich aus dem Thread erfolgen darf, in dem das Steuerelement erstellt wurde. Aus diesem Grund werden Sie die Vorgehensweise zur Aktualisierung von Eigenschaften in diesem Beispiel ein wenig befremdlich finden. Ich werde im Abschnitt »Verwenden von Steuerelementen in Threads« ab Seite 1331 auf dieses Problem genauer eingehen.

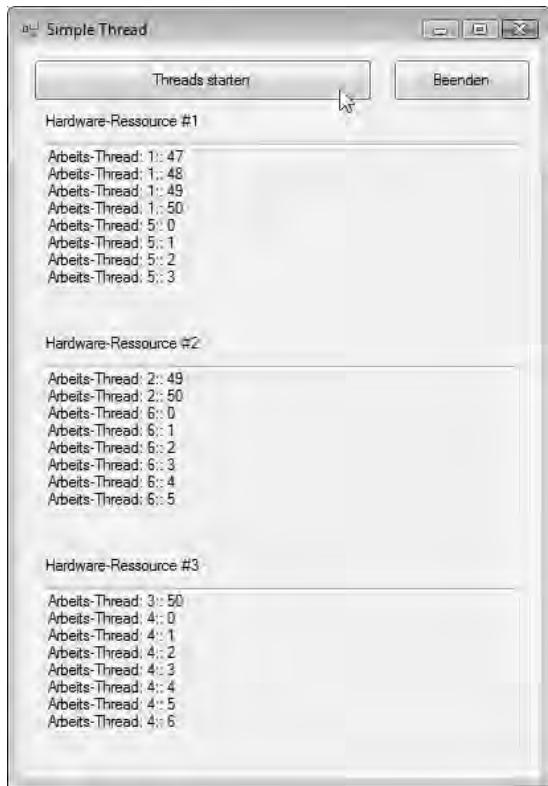


Abbildung 44.5 Drei Ausgabebereiche simulieren drei Hardware-Ressourcen. Über ein Array von Mutex-Objekten wird erkannt, welche als Nächstes zur Verfügung steht; der nächste anstehende Thread läuft dann im nächsten freien Fenster.

Das Codelisting:

```
'Speicher für Mutex-Objekte
Private myMutexes() As Mutex

'Speicher für TextBox-Controls
Private myTxtBoxes() As TextBox

'Zufallsgenerator für die künstlichen Wartezeiten im
'Arbeitsthread. Damit kann ein Thread unterschiedlich lange dauern.
Private myRandom As Random

'Delegat für das Aufrufen von Invoke, da Controls nicht
'thread-sicher sind.
Delegate Sub AddTBTextTSActuallyDelegate(ByVal tb As TextBox, ByVal txt As String)

Public Sub New()
    MyBase.New()

    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()
    'Mutexes definieren.
    myMutexes = New Mutex() {New Mutex, New Mutex, New Mutex}
    'Textbox-Array zuweisen.
```

```

myTxtBoxes = New TextBox() {txtHardware1, txtHardware2, txtHardware3}
'Zufallsgenerator initialisieren.
myRandom = New Random(DateTime.Now.Millisecond)

End Sub

'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
'Hochzählen und die Werteausgabe übernimmt.
Private Sub UmfangreicheBerechnung()

    Dim locMutexIndex As Integer
    Dim locTextBox As TextBox

    'Hier beginnt der kritische Abschnitt.
    'Warten, bis eine TextBox "frei" wird.
    locMutexIndex = Mutex.WaitAny(myMutexes)
    'Textbox, die dem freien Mutex entspricht, finden.
    locTextBox = myTxtBoxes(locMutexIndex)
    For c As Integer = 0 To 50
        SyncLock Me
            'Text in die TextBox des Threads ausgeben.
            AddTBTextTS(locTextBox, Thread.CurrentThread.Name + ":: " + c.ToString + vbCrLf)
        End SyncLock
        'Eine zufällige Weile lang warten.
        Thread.Sleep(myRandom.Next(50, 400))
    Next
    'Hier ist der kritische Abschnitt wieder vorbei.
    'Verwendete TextBox (Mutex) wieder freigeben.
    myMutexes(locMutexIndex).ReleaseMutex()
End Sub

```

Ein paar zusätzliche Erklärungen zu diesem Programm: Ein Mutex-Array wird als Klassen-Member zur Verwaltung der zur Verfügung stehenden Ressourcen im Konstruktor des Programms erstellt. Mit der statischen Funktion WaitAny, der ein Mutex-Array übergeben wird, wartet ein Thread solange, bis eines der Mutex-Objekte im Array frei wird. Wird es frei, liefert WaitAny den Index auf das jetzt freie Mutex-Objekt zurück, ändert dessen Zustand aber gleichzeitig in »blockiert«. Der zurück gelieferte Index wird anschließend verwendet, um die korrelierende TextBox zu finden, die diesem Mutex (nur über den Indexwert) quasi zugeordnet ist. Der Thread verwendet anschließend diese TextBox (wird über locTextBox referenziert), um die Ausgabe durchzuführen. WaitAny wartet übrigens nicht nur auf einen freien Mutex, sondern blockiert ihn auch wieder für den Thread, der ihn angefordert hat. Der Mutex bleibt solange im »Blockiert-Zustand«, bis der Thread den Mutex mit der ReleaseMutex-Methode wieder freigibt.

Weitere Synchronisationsmechanismen

Neben den bereits beschriebenen Synchronisationsmechanismen kennt das .NET Framework weitere Techniken, die in den folgenden Abschnitten kurz angerissen werden sollen:

Synchronization- und MethodImpl-Attribut

Das Synchronization-Attribut erlaubt die Erklärung einer ganzen Klasse zum kritischen Abschnitt. Wenn Sie eine Klasse mit diesem Attribut versehen, kann nur ein einziger Thread zur gleichen Zeit auf die Klasseninstanz zugreifen. Die Anwendung dieses Attributes funktioniert prinzipiell folgendermaßen:

```
<System.Runtime.Remoting.Contexts.Synchronization()> _
Public Class KomplettSynchronisiert
    Sub MacheIrgendwas()
        'Hier die Anweisungen
        'des Arbeitsthreads.
    End Sub
    Sub MacheIrgendwasAnderes()
        'Hier die Anweisungen
        'des Arbeitsthreads.
    End Sub
End Class
```

Wenn die Synchronisation einer ganzen Klasse zu viel des Guten wäre, steht Ihnen mit `MethodImpl` und dessen Parameter `MethodImplOptions.Synchronized` ein weiteres Attribut zur Verfügung, das nur eine einzelne Prozedur einer Klasse zum kritischen Abschnitt erklärt. Seine Anwendung funktioniert wie folgt:

```
Public Class TeilweiseSynchronisiert

<System.Runtime.CompilerServices.MethodImpl(Runtime.CompilerServices.MethodImplOptions.Synchronized)> _
    Sub MacheIrgendwasSynchronisiertes()
        'Hier die Anweisungen
        'des Arbeitsthreads.
    End Sub

    Sub MacheIrgendwasAnderes()
        'Hier die Anweisungen
        'des Arbeitsthreads.
    End Sub
End Class
```

Die Interlocked-Klasse

Mit Hilfe der `Interlocked`-Klasse können Sie steuern, wie viele Threads einen kritischen Abschnitt betreten dürfen, bevor weitere Threads ausgesperrt werden. Prinzipiell funktioniert sie also wie die `Monitor`-Klasse und dessen Methode `Enter`, nur dass Sie einen Parameter (eine Member-Variable der Klasse) angeben können, über die gesteuert wird, wie viele Threads den kritischen Abschnitt betreten dürfen.

```
Public Class InterlockedTest

    Dim myThreadZählerFürInterlocked As Integer

    Sub Arbeitsthread()

        'Maximal 3 Threads kommen hier gleichzeitig hinein,
        'dann ist Schluss!
        If Interlocked.Increment(myThreadZählerFürInterlocked) <= 3 Then
            'Hier die Anweisungen
            'den Arbeitsthreads.
            Interlocked.Decrement(myThreadZählerFürInterlocked)
        End If

    End Sub
End Class
```

Die Klasse selbst bietet übrigens ausschließlich statische Funktionen an; sie muss also nicht instanziert werden (tatsächlich ist sie eine abstrakte Klasse (`Noninheritable`), und kann auch gar nicht instanziert werden).

Die ReaderWriterLock-Klasse

Wenn mehrere Threads auf eine Datei zugreifen müssen, dann ist das so lange unkritisch, wie diese Threads aus der Datei lediglich lesen. Beim Schreiben sieht das anders aus: Sobald eine Datei geschrieben wird, darf kein anderer Thread zusätzlich in die Datei schreiben. Auch das Lesen aus einer Datei, in die gerade von einem anderen Thread geschrieben wird, könnte für die Ergebnisse einer Prozedur fatale Folgen haben.

Um dieses Problem zu lösen, gibt es die `ReaderWriterLock`-Klasse. Ihre wichtigsten Funktionen sind `AcquireReaderLock`, `ReleaseReaderLock`, `AcquireWriterLock` und `ReleaseWriterLock`.

Mehrere Threads, die sich über diese Klasse und deren Methoden synchronisieren wollen, müssen Zugriff auf dasselbe `ReaderWriterLock`-Objekt haben – eine Instanz dieser Klasse sollte also mindestens ein Klassen-Member sein oder als statische, öffentliche Eigenschaft programmweit verfügbar sein.³

Sowohl der Lesebereich als auch der Schreibbereich eines Threads werden nun als kritische Abschnitte definiert – etwa wie folgt:

```
Public Class ReaderWriteLockTest

    Dim myReaderWriterLock As New ReaderWriterLock

    Sub DateiLeseThread()
        'Hier kommt das Programm nur rein,
        'wenn nicht geschrieben wird, ein Schreibvorgang also
        'nicht zuvor durch ein myReaderWriterLock.AcquireWriterLock
        'eingeleitet wurde!
        '1000 Millisekunden wird auf den Lock gewartet.
        myReaderWriterLock.AcquireReaderLock(1000)
        'Lese-Thread nur ausführen, wenn Lock erteilt wurde.
        If myReaderWriterLock.IsReaderLockHeld Then
            'Hier die Anweisungen
            'des Arbeitsthreads
            'der aus einer Datei liest.
            myReaderWriterLock.ReleaseReaderLock()
        Else
            'TimeOut, in die Datei konnte nicht rechtzeitig geschrieben werden.
        End If
    End Sub

    Sub DateiSchreibThread()
        'Hier kommt ein neuer Thread nicht eher rein,
        'als bis ein anderer Lese-Thread zu Ende gelesen wurde
        'oder ein anderer Schreib-Thread den Schreibvorgang
    End Sub

```

³ Es könnte ja durchaus vorkommen, dass verschiedene Klassen und deren gleichzeitig laufende Arbeits-Threads einer Anwendung auf die gleiche Datei zugreifen müssen. In diesem Fall definieren Sie eine Klasse, die im statischen Konstruktor (`Shared New`) einen statischen Member vom Typ `ReaderWriterLock` instanziert und diesen über eine öffentliche statische Eigenschaft (`Public Shared Property ...`) allen anderen Klassen zur Verfügung stellt.

```
'komplettiert hat.  
'1000 Millisekunden wird auf den Lock gewartet.  
myReaderWriterLock.AcquireWriterLock(1000)  
'Schreib-Thread nur ausführen, wenn Lock erteilt wurde.  
If myReaderWriterLock.IsWriterLockHeld Then  
    'Hier die Anweisungen  
    'des Arbeitsthreads  
    'der in die Datei schreibt.  
    myReaderWriterLock.ReleaseWriterLock()  
Else  
    'TimeOut, in die Datei konnte nicht rechtzeitig geschrieben werden.  
End If  
  
End Sub  
End Class
```

Schon die Kommentare im Beispielprogramm machen die Zusammenhänge der Operationen deutlich:

- Wenn ein Thread die Leseroutine betritt, erhält er einen Leseschutz. Dieser Leseschutz schützt diesen Thread davor, dass ein neuer Schreibvorgang beginnt, *bevor* das Lesen abgeschlossen ist (und der Thread das durch ReleaseReaderLock angezeigt hat).
- Betritt ein Thread die Schreibroutine, wartet sein AcquireWriterLock solange, bis alle ausstehenden Lesevorgänge abgearbeitet sind. Neue Lese-Anforderungen werden nun solange zurückgestellt, wie AcquireWriterLock auf seine Schutzzuteilung wartet und diesen Schutz nach Beenden des Schreibens wieder freigegeben hat.

Synchronisieren von voneinander abhängigen Threads mit ManualResetEvent und AutoResetEvent

Eines vorweg: Lassen Sie sich von den Begriffen Event in den Namen dieser beiden Klassen nicht verwirren. Beide Begriffe bezeichnen instanzierbare Klassen und haben deshalb mit Ereignissen (*Events*), wie Sie sie bisher kennen gelernt haben, nicht das Geringste zu tun.

Sie verwenden beide Objekte, wenn bestimmte Threads voneinander abhängig sind. Sie haben beispielsweise einen Thread, der bestimmte Ergebnisse produziert, und weitere, die diese Ergebnisse anschließend weiterverarbeiten sollen. In diesem Fall verwenden Sie diese Objekte zur Synchronisation untereinander.

Beide Klassen unterscheiden sich lediglich in einem Punkt: Die AutoResetEvent-Klasse setzt ihren Zustand automatisch sofort zurück, sobald ein durch WaitOne geblockter Thread wieder gestartet wurde. ManualResetEvent macht eine Signaländerungsanzeige durch die Set-Eigenschaft erforderlich. Ein Beispiel für die AutoResetEvent-Klasse finden Sie übrigens im Abschnitt »Abbrechen und Beenden eines Threads« auf Seite 1334.

Verwenden von Steuerelementen in Threads

Wenn Sie Steuerelemente der Hauptanwendung von einem Thread aus verwenden wollen, müssen Sie Folgendes beachten: Steuerelemente sind nicht Thread-sicher, das heißt, sie dürfen nur aus dem Thread heraus verwendet werden, der sie erstellt hat. In der Regel ist das der Haupt-Thread der Anwendung, also der Thread, der die Benutzeroberfläche Ihrer Anwendung regelt. Aus diesem Grund nennt man diesen Thread auch den *UI-Thread* (UI als Abkürzung für *User Interface*, oder zu Deutsch: *Benutzeroberfläche*).

Damit ein anderer Thread dennoch das Steuerelement eines Formulars verwenden kann, muss er sich eines Tricks bedienen: Er muss dem Steuerelement mitteilen, dass es eine Prozedur aus seiner Klasse aufrufen soll. Diese Prozedur verändert dann die Eigenschaften des Steuerelements. Und da das Steuerelement diese Prozedur – wenn auch indirekt – selbst aufgerufen hat, wurde sie auch vom richtigen Thread (nämlich dem UI-Thread) aus aufgerufen.

Das Steuerelement selbst stellt dafür eine der wenigen Thread-sicheren Prozeduren zur Verfügung, die es besitzt. Ihr Name: `Invoke`. Ihr wird als Parameter ein Delegat übergeben, der als Zeiger auf die eigentliche Prozedur dient, die aus dem Thread aufgerufen werden soll. Im Mutex-Beispiel ist diese Vorgehensweise schon zur Anwendung gekommen, und aus diesem Grund werden wir dieses Programm noch einmal unter diesem Aspekt unter die Lupe nehmen.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\I - Threading\\Kapitel45\\SimpleThread05 (MutexDemo)

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Dieses Programm besteht aus drei `TextBox`-Steuerelementen, deren `Text`-Eigenschaft verwendet wird, um die Ausgaben des Programms anzuzeigen. Da die `TextBox`-Steuerelemente ausschließlich von der Thread-Prozedur verwendet werden (also in jedem Fall nicht vom UI-Thread), muss die Prozedur, die die Steuerelement-Eigenschaft setzt, zwingend über `Invoke` erfolgen.

```
'.
'.
' Delegat für das Aufrufen von Invoke, da Controls nicht Thread-sicher sind.
Delegate Sub AddTBTextTSActuallyDelegate(ByVal tb As TextBox, ByVal txt As String)
'.

'.
'Dient zum Setzen einer Eigenschaft auf einer TextBox indirekt über Invoke.
Private Sub AddTBTextTS(ByVal tb As TextBox, ByVal txt As String)
    Dim locDel As New AddTBTextTSActuallyDelegate(AddressOf AddTBTextTSActually)

    Me.Invoke(locDel, New Object() {tb, txt})

End Sub

Private Sub AddTBTextTSActually(ByVal tb As TextBox, ByVal txt As String)
    tb.Text += txt
    tb.SelectionStart = tb.Text.Length - 1
    tb.ScrollToCaret()
End Sub
```

Bitte lassen Sie sich in diesem Beispiel nicht von der Tatsache irritieren, dass hier ein `TextBox`-Steuerelement selbst als Parameter mit übergeben wird. Diese Vorgehensweise ist für das Beispiel deshalb notwendig, da die zu verwendende `TextBox` vom nächsten freien Mutex-Objekt im Thread abhängig ist. Da das Steuerelement hier als Parameter selbst mit übergeben wird, ist es eigentlich egal, über welches Steuerelement im Formular die `Invoke`-Methode aufgerufen wird. In diesem Beispiel ist es das Formular selbst, dessen `Invoke` verwendet wird. Genauso gut könnte die entsprechende Zeile, die `Invoke` ausführt, auch folgendermaßen ausschauen:

```
tb.Invoke(locDel, New Object() { tb, txt})
```

Voraussetzung dafür ist allerdings, dass tb aus dem UI-Thread stammt.

Letzten Endes kommt es also nur darauf an, dass ein Steuerelement des UI-Threads den Aufruf des Delegaten durchführt. Um das sicherzustellen, sollten Sie, von Ausnahmen wie in diesem Fall einmal abgesehen, die Invoke-Methode des Steuerelements, auf das sich die Änderungen auch beziehen, für einen indirekten Delegatenauftrag verwenden.

Managen von Threads

Thread-Objekte verfügen über einige Methoden, mit denen sie sich sowohl selbst steuern als auch von außen steuern lassen können. Zwei dieser Methoden haben Sie bereits kennen gelernt: die Start- und die Sleep-Methode. Die nächsten Abschnitte nehmen diese und weitere Methoden der Thread-Klasse ein wenig genauer unter die Lupe.

Starten eines Threads mit Start

Sie starten einen Thread mit seiner Start-Methode. Voraussetzung dafür ist, dass Sie den Thread zuvor mit einem ThreadStart-Objekt instanziert haben, das bei seiner Instanzierung wiederum die Adresse der Prozedur erhalten hat, die den eigentlichen Thread darstellt.

Vorübergehendes Aussetzen eines Threads mit Sleep – Statusänderungen im Framework bei Suspend und Resume

Wenn ein Thread für eine gewisse Zeit unterbrochen werden soll, verwenden Sie die Sleep-Methode des Thread-Objektes. Sleep übernimmt als Parameter entweder einen Integer-Wert, der die abzuwartende Zeitspanne in Millisekunden bestimmt oder einen TimeSpan-Wert, der ebenfalls die abzuwartende Dauer bestimmt, nach deren Ablauf der Thread wieder aktiv wird. Mit der Sleep-Methode kann sich ein Thread dadurch selbst »aussetzen« und wieder »aufwecken«.

Die Methoden Suspend und Resume sind im .NET 2.0-Framework übrigens als veraltet markiert und sollten laut Microsoft nicht mehr verwendet werden. Stattdessen sollten Sie andere Synchronisierungstechniken verwenden.

Wie Sie einen Update Ihres Source-Codes vornehmen, zeigt der Vergleich der beiden Beispiele in den Verzeichnissen *ObsoleteThreadMethodsDemo* und *ThreadPoolThreads* im Verzeichnis der Beispielprojekte dieses Kapitels. Sie entsprechen beide dem Prinzip des Beispiels, das in Abschnitt »Verwenden des Thread-Pools« ab Seite 1349 besprochen wird. Im ersten Verzeichnis finden Sie jedoch die veraltete Version, im zweiten die entsprechend angepasste.

Abbrechen und Beenden eines Threads

Abort nennt sich die Methode, mit der Sie die Möglichkeit haben, einen Thread abzubrechen. Allerdings sollten Sie von dieser Funktion nur in Ausnahmefällen Gebrauch machen, denn: Abort wird nicht unmittelbar ausgeführt. Framework-intern wird eine Ausnahme vom Typ `ThreadAbortException` ausgelöst. Diese Ausnahme ist allerdings etwas Besonderes, da Sie sie nicht abfangen können. Falls der Thread jedoch in einem Try/Catch/Finally-Codeblock ausgeführt wird, garantiert das Framework, dass der Finally-Block bei Abort auf jeden Fall abgearbeitet wird. Ein Thread kann dabei mithilfe seiner `ThreadState`-Eigenschaft feststellen, dass er gerade abgebrochen wird, und das Abbrechen sogar mit `ResetAbort` verhindern.

Allerdings ist das Abbrechen eines Threads keine elegante Vorgehensweise – eher die Brechstangenmethode. Ein Thread sollte ausschließlich beendet werden, indem er seine Arbeitsprozedur verlässt. Auf der anderen Seite muss ein Thread auf jeden Fall dafür sorgen, dass er beendet wird, wenn das Hauptprogramm beendet wird, das ihn beherbergt.

In den vorangegangenen Beispielen ist das bislang nicht der Fall gewesen. Wenn Sie das Hauptfenster schließen, wird zwar das Hauptprogramm beendet – Threads, die zu dieser Zeit noch aktiv sind, laufen aber unbehelligt weiter. Das heißt, für das letzte Beispiel gilt »unbehelligt« nicht wirklich. Tatsächlich steigt das Programm mit einer Fehlermeldung aus, wenn Sie es schließen, während andere Threads noch munter am Werkeln sind. Das liegt daran, dass der Thread seine Ausgabe in eine `TextBox` vornimmt, die es zu diesem Zeitpunkt schon gar nicht mehr gibt.

Die erste Idee, die einem in den Sinn kommen könnte, um dieses Problem zu lösen, wäre ein klassenweites Flag, das dann gesetzt wird, wenn das Formular geschlossen wird. Dazu müsste man lediglich die `OnClosing`-Routine des Formulars überschreiben. Bevor ein Thread eine Ausgabe im Formular vornimmt, könnte er dieses Flag überprüfen. Wäre es gesetzt, beendete der Thread sich mit einem simplen `Exit Sub` auf unproblematische Art und Weise.

Allerdings ist diese Problemlösung – jedenfalls was Windows Forms-Anwendungen anbelangt – nicht konsequent zu Ende gedacht, denn: Wird dieses »Thread-Abbrechen-Flag« in dem Moment auf `True` gesetzt, in dem sich der Thread gerade zwischen der Flag-Abfrage und der Ausgabe des Textes ins Steuerelement befindet, hat die ganze Aktion überhaupt nichts gebracht. Auch in diesem Fall rauscht der Thread unbremst in die Ausgaberroutine für die `TextBox`, die es auch in diesem Fall nicht mehr gibt. Abermals ist eine Ausnahme die Folge. Was also tun?

Eine Synchronisierungstechnik ist an dieser Stelle wieder indiziert. `OnClosing` muss so lange warten, bis ein Thread den kritischen Bereich des Schreibens in die `TextBox` beendet hat. Dazu muss der Thread diesen kritischen Bereich aber erst einmal definieren und dafür bietet sich in diesem Fall das `AutoRaiseEvent`-Objekt an (das `ManualRaiseEvent`-Objekt täte es übrigens genau so gut für unser Vorhaben).

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\I - Threading\\Kapitel45\\SimpleThread06 (MutexDemo proper)

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie dieses Beispiel starten, können Sie das Formular schließen, egal wie viele Threads noch unterwegs sind. Der veränderte Code für das Beispielprogramm sieht folgendermaßen aus:

```
'Member-Variable, damit die Threads durchnummieriert werden können.  
'Dient nur zur späteren Unterscheidung des laufenden Threads, wenn  
'er Ergebnisse im Ausgabefenster darstellt.  
Private myArbeitsThreadNr As Integer = 1  
  
'Speicher für Mutex-Objekte  
Private myMutexes() As Mutex  
  
'Speicher für TextBox-Controls  
Private myTxtBoxes() As TextBox  
  
'Zufallsgenerator für die künstlichen Wartezeiten im  
'Arbeitsthread. Damit kann ein Thread unterschiedlich lange dauern.  
Private myRandom As Random  
  
'Delegat für das Aufrufen von Invoke, da Controls nicht  
'Thread-sicher sind.  
Delegate Sub AddTBTTextTSActuallyDelegate(ByVal tb As TextBox, ByVal txt As String)  
  
'Flag, das bestimmt, wann alle Threads zu gehen haben  
Private myAbortAllThreads As Boolean  
  
'Zusätzliches Synchronisierungsobjekt für "Alle Threads beenden!"  
Private myAutoResetEvent As AutoResetEvent  
  
Public Sub New()  
    MyBase.New()  
  
    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.  
    InitializeComponent()  
    'Mutexes definieren  
    myMutexes = New Mutex() {New Mutex, New Mutex, New Mutex}  
    'Textbox-Array zuweisen  
    myTxtBoxes = New TextBox() {txtHardware1, txtHardware2, txtHardware3}  
    'Zufallsgenerator initialisieren  
    myRandom = New Random(DateTime.Now.Millisecond)  
    'Zusätzliches Synchronisierungsobjekt für "Alle Threads beenden!" initialisieren  
    myAutoResetEvent = New AutoResetEvent(True)  
  
End Sub  
  
'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das  
'Hochzählen und die Werteausgabe übernimmt  
Private Sub UmfangreicheBerechnung()  
    Dim locMutexIndex As Integer  
    Dim locTextBox As TextBox  
  
    'Hier beginnt der 1. kritische Abschnitt der Mutexes  
    'Warten, bis eine TextBox "frei" wird  
    locMutexIndex = Mutex.WaitAny(myMutexes)  
    'Thread beenden, wenn das "Alle-Threads-Beenden-Flag" während  
    'des Wartens signalisiert wurde  
    If myAbortAllThreads Then  
        'Verwendeten Textbox-Mutex wieder freigeben,  
        'sonst knallt es, wenn die anderen Threads in diesem  
        'Abschnitt beendet werden sollen!
```

```

    myMutexes(locMutexIndex).ReleaseMutex()
    Exit Sub
End If
'Textbox, die dem freien Mutex entspricht, finden
locTextBox = myTxtBoxes(locMutexIndex)
'Hier beginnt der 2. kritische Abschnitt für OnClosing
myAutoResetEvent.Reset()
For c As Integer = 0 To 20
    SyncLock Me
        'Falls abgebrochen werden soll,
        If myAbortAllThreads Then
            'OnClosing benachrichtigen
            myAutoResetEvent.Set()
            'Verwendeten TextBox-Mutex wieder freigeben.
            'Grund: Siehe oben.
            myMutexes(locMutexIndex).ReleaseMutex()
            Exit Sub
        End If
        'Text in die TextBox des Threads ausgeben
        AddTBTextTS(locTextBox, Thread.CurrentThread.Name + ":: " + c.ToString + vbNewLine)
        Console.WriteLine(Thread.CurrentThread.Name + ":: " + c.ToString + vbNewLine)
    End SyncLock
    'Eine zufällige Weile lang warten
    Thread.Sleep(myRandom.Next(50, 400))
Next
'2. kritische Abschnitt beendet (OnClosing)
myAutoResetEvent.Set()
'Hier ist der 1. kritische Abschitt wieder vorbei.
'Verwendete TextBox (Mutex) wieder freigeben
myMutexes(locMutexIndex).ReleaseMutex()
End Sub

Protected Overrides Sub OnClosing(ByVal e As System.ComponentModel.CancelEventArgs)
    'WICHTIG: Ein einfaches Warten mit WaitOne reicht in diesem
    'Fall nicht aus, da es sich um den Hauptthread handelt.
    'Dann aber würde WaitOne die Nachrichtenwarteschlange blockieren,
    'und ohne die funktioniert Invoke nicht. Deswegen wird hier ein TimeOut
    'angegeben, der Nachrichtenwarteschlange die Möglichkeit gegeben, einmal
    '"Luft zu holen", und dann weiter gewartet.
    myAbortAllThreads = True
    'DoEvents beim Warten in Ein-Millisekunden-Abständen auslösen
    Do While Not myAutoResetEvent.WaitOne(1, True)
        Application.DoEvents()
    Loop
End Sub

'Dient zum Setzen einer Eigenschaft auf einer TextBox indirekt über Invoke
Private Sub AddTBTextTS(ByVal tb As TextBox, ByVal txt As String)
    Dim locDel As New AddTBTextTSActuallyDelegate(AddressOf AddTBTextTSActually)
    Me.Invoke(locDel, New Object() {tb, txt})
End Sub

```

```
Private Sub AddTBTTextTSActually(ByVal tb As TextBox, ByVal txt As String)
    tb.Text += txt
    tb.SelectionStart = tb.Text.Length - 1
    tb.ScrollToCaret()
End Sub

Private Sub btnThreadStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnThreadStarten.Click
    'Dieses Objekt kapselt den eigentlichen Thread
    Dim locThread As Thread
    'Dieses Objekt benötigen Sie, um die Prozedur zu bestimmen,
    'die den Thread ausführt.
    Dim locThreadStart As ThreadStart

    'Threadausführende Prozedur bestimmen
    locThreadStart = New ThreadStart(AddressOf UmfangreicheBerechnung)
    'ThreadStart-Objekt dem Thread-Objekt übergeben
    locThread = New Thread(locThreadStart)
    'Thread-Namen bestimmen
    locThread.Name = "Arbeits-Thread: " + myArbeitsThreadNr.ToString
    'Thread starten
    locThread.Start()
    'Zähler, damit die Threads durch ihren Namen unterschieden werden können
    myArbeitsThreadNr += 1

End Sub

Private Sub btnBeenden_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnBeenden.Click
    'Einfach so geht's normalerweise nicht
    Me.Close()
End Sub
```

Eine kleine Anmerkung vielleicht noch an dieser Stelle zur überschriebenen `OnClosing`-Prozedur: Wie aus den Kommentaren im Codelisting schon ersichtlich, gilt es noch ein letztes Problem aus der Welt zu schaffen. Ein einfaches `WaitOne` würde den UI-Thread einfrieren. Dummerweise benötigt `Invoke` eines Steuerelements eine aktive Nachrichtenwarteschlange, um zu funktionieren. Aus diesem Grund müssen wir an dieser Stelle wieder einen Trick anwenden, damit sich das Programm beim Warten auf die letzte Textausgabe nicht aufhält. Als Argument übergeben wir einen `Timeout`-Wert von einer Millisekunde, nachdem `WaitOne` zunächst den Wartevorgang auf das Abschließen des kritischen Abschnittes eines Threads unterbricht. Ein anschließendes `DoEvents` erlaubt das Ausführen eines möglicherweise noch ausstehenden `Invoke` im Arbeits-Thread. `DoEvents` wird in einer Schleife abgehandelt, die so lange ausgeführt wird, bis `WaitOne` zurückmeldet, dass es nicht durch `TimeOut`, sondern tatsächlich durch die Signalisierung des Arbeits-Threads beendet wurde. In diesem Fall »weiß« `OnClosing`, dass der ausstehende Thread beendet wurde, die `TextBox` zur Ausgabe aber ergo nicht mehr gebraucht wird und das Formular zu diesem Zeitpunkt sicher geschlossen werden kann.

HINWEIS Da die Verarbeitung dieser Warteschleife natürlich auch Zeit kostet, sollten Sie bei sehr zeitkritischen Operationen von dieser Methode absehen. Die Prozessorleistung, die für die Warteschleife benötigt wird, fehlt anderen Threads natürlich. Eine Zwischenlösung: Sie können auf Kosten der Reaktionszeit natürlich den `TimeOut`-Parameter für `WaitOne` erhöhen, um Leistung zu sparen. Denn während `WaitOne` »ausgeführt« wird, können andere Threads bedient werden.

Datenaustausch zwischen Threads durch Kapseln von Threads in Klassen

Sie haben im Laufe der letzten Kapitel bereits feststellen können, dass Sie beim Einrichten eines Threads keine Parameter an die eigentliche Thread-Prozedur übergeben können. In der Praxis sind Threads, denen keine Daten zum Verarbeiten übergeben werden können, aber eher nutzlos. Doch so groß ist das Problem nicht, denn es gibt einen ganz einfachen Trick, mit dem Sie es meistern können: Sie kapseln die eigentliche Thread-Routine einfach in einer Klasse. Die notwendigen Parameter, die die Thread-Routine benötigt, können Sie dann einfach im Konstruktor der Klasse übergeben.

Da der Thread asynchron läuft, also nur sozusagen »angeschmissen« wird und dann alleine weiterarbeitet, muss die Thread-Klasse natürlich auch in der Lage sein, ein »ich habe fertig« an die ihn einbindende Instanz mitteilen zu können. Doch auch dieses Problem ist vergleichsweise einfach in den Griff zu bekommen: Wenn der Arbeits-Thread innerhalb der Klasse seine Aufgabe abgeschlossen hat, löst er einfach ein Ereignis aus. Die Ergebnisse, die durch ihn zustande gekommen sind, können anschließend durch Eigenschaften der Öffentlichkeit zugänglich gemacht werden.

Das folgende Codelisting zeigt, wie beispielsweise eine Klasse aussehen kann, die ein beliebiges Array in einem Thread sortieren kann.

```
Public Class SortArrayThreaded
    Implements IDisposable

    Public Event SortCompleted(ByVal sender As Object, ByVal e As SortCompletedEventArgs)
    Private myArrayToSort As ArrayList
    Private myAutoResetEvent As AutoResetEvent
    Private myTerminateThread As Boolean
    Private myThreadStart As ThreadStart
    Private myThread As Thread
    Private mySortingComplete As Boolean

    'Konstruktor übernimmt die zu sortierende Liste. Der ThreadName hat nur
    'dokumentarischen Charakter.
    Sub New(ByVal ArrayToSort As ArrayList, ByVal ThreadName As String)
        myArrayToSort = ArrayToSort
        myThread = New Thread(New ThreadStart(AddressOf SortArray))
        myThread.Name = ThreadName
        myAutoResetEvent = New AutoResetEvent(True)
    End Sub
```

Der eigentliche Thread kann beginnen, wenn die die Klasse einbindende Instanz nach der Instanziierung die folgende Methode aufruft. Damit der Thread mit anderen Aufgaben synchronisiert werden kann, stellt er ein AutoResetEvent-Objekt zur Verfügung, dessen Signalisierung an dieser Stelle zurückgesetzt wird. Ein anderer Thread, der darauf wartet, dass die Sortierung beendet wird, kann nun nicht nur durch das Ereignis, sondern auch über die WaitOne-Methode (oder, wenn mehrere Threads zu synchronisieren sind, auch über WaitAny oder WaitAll) über das Ende der Sortierung informiert werden.

```
'Thread starten und signalisieren, dass blockiert.  
Public Sub StartSortingAsynchron()  
    myAutoResetEvent.Reset()  
    myThread.Start()  
End Sub  
  
Private Sub SortArray()  
  
    Dim locSortCompletionStatus As SortCompletionStatus  
  
    If Not myArrayToSort Is Nothing Then  
        locSortCompletionStatus = Me.ShellSort()  
    End If  
    myAutoResetEvent.Set()  
    'Dieses Ereignis könnte man einbinden, falls der Sort-Thread über das  
    'Sortierende benachrichten soll.  
    RaiseEvent SortCompleted(Me, New SortCompletedEventArgs(locSortCompletionStatus))  
End Sub
```

Kleine Anmerkung am Rande: Die folgende Prozedur stellt den eigentlichen Arbeits-Thread der Klasse dar. Natürlich gibt es eine Reihe von Sortierungsmöglichkeiten von Elementen, die im Framework eingebaut sind. Aber behalten Sie im Hinterkopf, dass es hier in erster Linie um die Demonstration der grundsätzlichen Threading-Möglichkeiten geht.

```
'Sortiert eine ArrayList, die IComparable-Member enthält.  
'Null-Werte oder inkompatible Typen werden nicht geprüft!  
Private Function ShellSort() As SortCompletionStatus  
  
    Dim locOutCount, locInCount As Integer  
    Dim locDelta As Integer  
    Dim locElement As IComparable  
  
    locDelta = 1  
  
    'Größten Wert der Distanzfolge ermitteln.  
    Do  
        locDelta = 3 * locDelta + 1  
    Loop Until locDelta > myArrayToSort.Count  
  
    Do  
        'War eins zu groß, also wieder teilen.  
        locDelta \= 3  
  
        'Shellsorts Kernalgorithmus  
        For locOutCount = locDelta To myArrayToSort.Count - 1  
            locElement = CType(myArrayToSort(locOutCount), IComparable)  
            locInCount = locOutCount  
            Do While CType(myArrayToSort(locInCount - locDelta), IComparable).CompareTo(locElement)  
                = 1  
                    If myTerminateThread Then  
                        Return SortCompletionStatus.Aborted  
                    End If  
                    myArrayToSort(locInCount) = myArrayToSort(locInCount - locDelta)  
                    locInCount = locInCount - locDelta  
            Loop  
        Next locOutCount  
    Loop
```

```

        If (locInCount <= locDelta) Then Exit Do
        Loop
        myArrayToSort(locInCount) = locElement
    Next
Loop Until locDelta = 0
Return SortCompletionStatus.Completed

End Function

```

WICHTIG Damit ein laufender Thread auf elegante Art und Weise beendet werden kann (und nicht mit brachialer Gewalt durch Abort), gibt es eine Member-Variable namens myTerminateThread, deren Setzen auf True bewirkt, dass der Sortier-Algorithmus seine Arbeit abbricht, die Prozedur verlässt und damit den Arbeits-Thread sauber zu Ende führt. Um ein Programm nicht durch laufende Sortier-Threads zu blocken, wenn es für die Klasseninstanz Zeit wird, vom Garbage Collector entsorgt zu werden, implementiert die Klasse IDisposable und damit Dispose, das diesen Member setzt. Der Garbage Collector ist damit spätestens derjenige, der den Thread beendet. Eigentlich ist das aber nur das Netz unter dem Trapez. Sie sollten in jedem Fall selbst durch geschickte Programmierung darauf achten, dass kein noch laufender Thread existiert, wenn Sie das umgebende Programm beenden. Rufen Sie im Zweifelsfall lieber selbst die Dispose-Methode Ihrer Thread-Klasse auf, damit der Thread in jedem Fall ordnungsgemäß beendet wird.

```

'Dispose beendet einen vielleicht noch laufenden Sortier-Thread.
Public Sub Dispose() Implements System.IDisposable.Dispose
    myTerminateThread = True
End Sub

'Stellt die benötigten Eigenschaften zur Verfügung.
Public ReadOnly Property AutoResetEvent() As AutoResetEvent
    Get
        Return myAutoResetEvent
    End Get
End Property

Public ReadOnly Property ArrayList() As ArrayList
    Get
        Return myArrayList
    End Get
End Property
End Class

'Die beiden möglichen Ergebnisse, wenn der Sortier-Thread beendet wurde.
'Falls er durch Dispose abgebrochen wurde, liefert er Aborted zurück.
Public Enum SortCompletionStatus
    Completed
    Aborted
End Enum

```

Damit das Ereignis, das ausgelöst wird, wenn der Thread die Sortierung beendet hat, auch ordnungsgemäß über die durchgeführten Vorgänge informieren werden kann, gibt es zusätzlich zur eigentlichen Klasse noch ein paar Hilfselemente für diesen Zweck:

```
'Dient nur der Ereignisübergabe, wenn Sie das SortCompleted-Ereignis
'nach Abbruch oder Abschluss des Sortierens auslösen wollen.
Public Class SortCompletedEventArgs
    Inherits EventArgs

    Private mySortCompletionStatus As SortCompletionStatus
    Sub New(ByVal scs As SortCompletionStatus)
        mySortCompletionStatus = scs
    End Sub

    Public Property SortCompletionStatus() As SortCompletionStatus
        Get
            Return mySortCompletionStatus
        End Get
        Set(ByVal Value As SortCompletionStatus)
            mySortCompletionStatus = Value
        End Set
    End Property
End Class
```

Soviel zum eigentlichen Arbeitstier dieses Beispiels. Nun ist es an der Zeit herauszufinden, ob es auch wirklich das leistet, was wir von ihm erwarten.

Der Einsatz von Thread-Klassen in der Praxis

Sie finden in den Begleitdateien ein Beispiel, dass die im vorherigen Abschnitt vorgestellte Klasse demonstriert. Und es macht nicht nur das. Es zeigt auch die Geschwindigkeitsverhältnisse von Programmen, wenn Sie mit einem oder mehreren Threads bestimmte Probleme lösen.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2008 Entwicklerbuch\I - Threading\Kapitel45\ThreadsInPraxis (MultiThreadingSorting)
```

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Wenn Sie das Programm starten, zeigt es Ihnen einen Dialog, wie Sie ihn in etwa auch in Abbildung 44.6 sehen können. Bevor Sie den Test starten, entscheiden Sie sich durch Anklicken des entsprechenden Feldes, ob Sie für die Sortierung der Daten Single- oder Multithreading verwenden wollen. Mit einem weiteren Klick auf *Benchmark* starten Sie die Threading-Operationen.



Abbildung 44.6 Solche Unterschiede erreichen Sie nur auf einem Multiprozessor- oder MultiCore-System. Wichtig: Beim Einsatz von Hyperthreading auf einem Prozessor kann das Sortieren unter Umständen sogar länger dauern, als beim ausgeschalteten Hyperthreading.

Das folgende Codelisting zeigt, wie das Programm funktioniert. Längere Kommentare im Listing finden Sie im Folgenden der leichteren Lesbarkeit wegen in Normalschrift gesetzt und dort, wo es sinnvoll erscheint, etwas ausführlicher beschrieben. Sie sind im Listing aber ebenfalls vorhanden.

Einige Worte zur generellen Funktionsweise vorweg: Das Programm arbeitet mit maximal vier Threads, aber mindestens drei Threads, wenn die Sortierung vorgenommen wird. Der UI-Thread – also der Thread, der das Formular verwaltet – startet automatisch mit dem Programm. Wenn der Anwender den Benchmark startet, beginnt ein zweiter Thread, der die zu sortierenden Elemente erzeugt und dann entweder einen weiteren Thread oder zwei weitere Threads startet, um diese Elemente zu sortieren. Mit dieser Vorgehensweise wird erreicht, dass der UI-Thread ungestört weiter operieren kann, ohne durch den eigentlichen Arbeits-Thread angehalten zu werden.

```
'Dieser Namensbereich enthält die benötigten Threading-Objekte.
Imports System.Threading
Imports System.IO

Public Class frmMain
    Inherits System.Windows.Forms.Form

#Region " Vom Windows Form Designer generierter Code "
    'Aus Platzgründen ausgelassen
#End Region

    Private Const cAmountOfElements As Integer = 75000
    Private Const cCharsPerString As Integer = 100
    Private Const cShowResults As Boolean = False
```

Die folgenden Delegaten werden benötigt, um bestimmte Eigenschaften der Steuerelemente des Formulars thread-sicher verändern zu können. Das gilt für das Setzen der Text-Eigenschaft des TextBox-Steuerelements, das für die Darstellung der Kommentare zuständig ist, auf der einen und für das Wiedereinschalten der Schaltflächen auf der anderen Seite.

```
'Delegaten für das Aufrufen von Invoke, da Controls nicht
'Thread-sicher sind.
'Für die Textbox zur Ausgabe
Private Delegate Sub AddTBTextTSAcuallyDelegate(ByVal txt As String)
'Für das Einschalten der Buttons, wenn der Vorgang beendet ist.
Private Delegate Sub TSEnableControlsDelegate()

'Flag, das bestimmt, wann alle Threads zu gehen haben.
Private myAbortAllThreads As Boolean

'Ausgabe-Textbox. Statisch deswegen, damit jeder Thread
'zu jeder Zeit darauf zugreifen kann.
Private Shared myAusgabeTextBox As TextBox

'Synchronisation für die Ausgabe-Textbox
Private Shared myAutoResetEvent As AutoResetEvent

'Merkt sich Thread-sicher, ob die Sortierung mit einem oder
'zwei Threads durchgeführt werden soll.
Private myUseTwoThread As Boolean

'Ereignis, um dem UI-Thread mitteilen zu können,
'dass der Hauptarbeits-Thread abgeschlossen ist.
Private Event MainThreadFinished()

Public Sub New()
    MyBase.New()

    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()

    'Synchronisation für die Ausgabe-Textbox
    myAutoResetEvent = New AutoResetEvent(True)

End Sub
```

Wichtig bei der Verwendung der folgenden Funktion ist die Synchronisation mit allen Programmprozeduren, die Sie verwenden, denn: Sollte das Formular genau in dem Moment geschlossen werden, in dem ein Thread genau diese Routine erreicht hat, dann würde Invoke auf ein Steuerelement zugreifen, das zu dieser Zeit nicht mehr existierte. Eine Ausnahme wäre die Folge. Aus diesem Grund wird das Schließen des Formulars mit der Ausgabe in der TextBox über ein AutoResetEvent-Objekt synchronisiert. OnClosing fängt das Schließen-Ereignis ab und wartet, bis eine mögliche, gerade laufende Ausgabe in der TextBox abgeschlossen ist.

```
'Dient zum Setzen einer Eigenschaft auf der TextBox indirekt über Invoke.
Public Shared Sub AddTBText(ByVal txt As String)
    'Formular nicht mehr oder noch nicht da...
    If AusgabeTextBox Is Nothing Then
        '...und tschö!
        Exit Sub
    End If
    'Threads synchronisieren: Hier beginnt kritischer Abschnitt.
    myAutoResetEvent.Reset()
    Dim locDel As New AddTBTextTSActuallyDelegate(AddressOf AddTBTextTSActually)
    AusgabeTextBox.Invoke(locDel, New Object() {txt})
    myAutoResetEvent.Set()
End Sub

'Diese Routine wird indirekt über Invoke aufgerufen, und ist damit UI-Thread.
Private Shared Sub AddTBTextTSActually(ByVal txt As String)
    AusgabeTextBox.Text += txt
    AusgabeTextBox.SelectionStart = AusgabeTextBox.Text.Length - 1
    AusgabeTextBox.ScrollToCaret()
End Sub

'Liefert die TextBox als Eigenschaft.
Private Shared ReadOnly Property AusgabeTextBox() As TextBox
    Get
        Return myAusgabeTextBox
    End Get
End Property
```

Die folgende Routine kümmert sich um die notwendigen Schritte, wenn der Anwender die Schaltfläche *Benchmark starten* angeklickt hat. Sie startet dann den Arbeits-Thread (den Haupt-Thread), der zunächst alle zu sortierenden Elemente erzeugt und sie dann anschließend durch die Sortier-Threads ordnet.

```
Private Sub btnThreadStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnBenchmarkStarten.Click
    'Dieses Objekt kapselt den eigentlichen Thread.
    Dim locThread As Thread
    'Dieses Objekt benötigen Sie, um die Prozedur zu bestimmen,
    'die den Hauptthread ausführt.
    Dim locThreadStart As ThreadStart

    'Ausgabetextbox bestimmen.
    myAusgabeTextBox = Me.txtAusgabe
```

Damit der Haupt-Thread kein zweites Mal gestartet werden kann, schaltet die Routine die Schaltflächen an dieser Stelle aus. Wenn der komplette Testparcours abgeschlossen ist, löst der Haupt-Thread ein Ereignis aus, das vom Formular eingebunden wird. Die Behandlungsprozedur dieses Ereignisses schaltet die Schaltflächen dann indirekt über Invoke wieder ein. Diese Vorgehensweise dient nur der Demonstration: Genauso gut könnte der Haupt-Thread selbst die Schaltflächen wieder einschalten, wenn alle Operationen abgeschlossen sind.

```

btnBenchmarkStarten.Enabled = False
btnBeenden.Enabled = False

'Festhalten, ob die Sortierung in einem oder zwei Threads erfolgen soll.
myUseTwoThread = Me.chkMultithreading.Checked

'Thread ausführende Prozedur bestimmen.
locThreadStart = New ThreadStart(AddressOf StartMainThread)
'ThreadStart-Objekt dem Thread-Objekt übergeben
locThread = New Thread(locThreadStart)
'Thread-Namen bestimmen.
locThread.Name = "MainThread"
'Haupt-Thread starten.
locThread.Start()

'Der UI-Thread (die Nachrichtenwarteschlange) läuft jetzt leer nebenbei.
'Auf diese Weise kann der komplette Testparcours ruhig 100% Prozessorleistung
'verschlingen; da er in einem Extra-Thread läuft, bleibt das Programm
'dennoch bedienbar.
End Sub

```

Dies ist die eigentliche Haupt-Thread-Routine (aber nicht der UI-Thread!), die das Array mit den zu sortierenden Elementen generiert und dann selbst wiederum entweder ein oder zwei Arbeits-Threads aufruft, die die eigentliche Sortierung durchführen.

```

Private Sub StartMainThread()

    Dim locStrings(cAmountOfElements - 1) As String
    Dim locGauge As New HighSpeedTimeGauge
    Dim locAutoResetEvents(1) As AutoResetEvent

    Dim locRandom As New Random(DateTime.Now.Millisecond)
    Dim locChars(cCharsPerString) As Char

    'Damit die Controls nach Abschluss des Sortierens auf dem Formular
    'wieder eingeschaltet werden können, muss der Haupt-Thread das Ende
    'der Sortierung mitbekommen. Deswegen: Ereignis
    AddHandler MainThreadFinished, AddressOf MainThreadFinishedHandler

    'String-Array erzeugen; hatten wir schon zig Mal...
    AddTBText("Erzeugen von " + cAmountOfElements.ToString + " Strings..." + vbNewLine)
    locGauge.Start()
    For locOutCount As Integer = 0 To cAmountOfElements - 1
        For locInCount As Integer = 0 To cCharsPerString - 1
            If myAbortAllThreads Then
                RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler
                Exit Sub
            End If
            Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)
            If locIntTemp > 26 Then
                locIntTemp += 97 - 26
            Else
                locIntTemp += 65
            End If
        Next
    Next

```

```

    locChars(locInCount) = Convert.ToChar(locIntTemp)
    Next
    locStrings(locOutCount) = New String(locChars)
    Next
    locGauge.Stop()
    AddTBText("...in " + locGauge.DurationInMilliSeconds.ToString + " Millisekunden" + vbNewLine)
    AddTBText(vbNewLine)
    locGauge.Reset()

```

Die Member-Variablen myUseTwoThread wird durch den Ereignis-Handler des CheckBox-Steuerelements gesetzt. Sie bestimmt, ob der Haupt-Thread das Sortieren mit zwei oder nur einem Thread starten soll.

```

If Not myUseTwoThread Then
    'Messung starten - hier wird mit nur einem Thread sortiert.
    locGauge.Start()
    Dim locFirstSortThread As New SortArrayThreaded(New ArrayList(locStrings), "1. SortThread")

    AddTBText("Sortieren von " + cAmountOfElements.ToString + " Strings mit einem Thread..." + _
              vbNewLine)
    locAutoResetEvents(0) = locFirstSortThread.AutoResetEvent
    locFirstSortThread.StartSortingAsynchron()

```

Die folgende Do-While-Schleife wartet, bis der Sortier-Thread abgeschlossen ist. Der Sortier-Thread, der durch die SortArrayThreaded-Klasse gekapselt wird, stellt eine AutoResetEvents-Eigenschaft bereit, die signalisiert wird, wenn der Sortierungsvorgang abgeschlossen ist.

TIPP Wenn Sie eigene Klassen implementieren, die Aufgaben kapseln, welche in Threads ausgeführt werden, sollten Sie ebenfalls dafür sorgen, dass der Abschluss einer Aufgabe nicht nur durch ein Ereignis, sondern auch mit einem Synchronisations-Objekt wie beispielsweise der Mutex- oder der AutoResetEvent-Klasse signalisiert wird.

```

Do While Not locAutoResetEvents(0).WaitOne(1, True)
    'Mit Timeout-Wert warten, damit ein Eingreifen (Abbrechen) im Sortier-Thread möglich
    'bleibt, wenn das komplette Programm beendet werden soll.
    If myAbortAllThreads = True Then
        'Die Dispose-Methode der SortArrayThread-Klasse bricht einen vielleicht laufenden
        'Sortiert-Thread ab.
        locFirstSortThread.Dispose()
        RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler
        Exit Sub
    End If
Loop
'Messung beenden.
locGauge.Stop()
AddTBText("...in " + locGauge.DurationInMilliSeconds.ToString + " Millisekunden" + _
          vbNewLine)
AddTBText(vbNewLine)
'ArrayList zurück-casten.
locStrings = CType(locFirstSortThread.ArrayList.ToArray(GetType(String)), String())
Else
    'Es soll mit zwei Threads sortiert werden.
    locGauge.Start()
    'Zwei ArrayLists erstellen, die jeweils den oberen und den unteren Teil der

```

```
'zu sortierenden Gesamtliste beinhalten.  
Dim locFirstAL As New ArrayList(locStrings.Length \ 2 + 1)  
Dim locSecondAL As New ArrayList(locStrings.Length \ 2 + 1)  
Dim locMitte As Integer = locStrings.Length \ 2  
  
'Elemente über beide ArrayLists verteilen  
For c As Integer = 0 To locMitte - 1  
    locFirstAL.Add(locStrings(c))  
    locSecondAL.Add(locStrings(c + locMitte))  
Next  
  
'Zwei SortArrayThreaded-Instanzen erzeugen und ihnen die Teillisten übergeben.  
Dim locFirstSortThread As New SortArrayThreaded(locFirstAL, "1. SortThread")  
Dim locSecondSortThread As New SortArrayThreaded(locSecondAL, "2. SortThread")  
  
AddTBText("Sortieren von " + cAmountOfElements.ToString + " Strings mit zwei Threads..." + _  
vbNewLine)  
'Beide AutoResetEvent-Objekte der beiden Sortier-Threads in ein Array packen,  
'damit auf den Abschluss beider Threads gewartet werden kann.  
locAutoResetEvents(0) = locFirstSortThread.AutoResetEvent  
locAutoResetEvents(1) = locSecondSortThread.AutoResetEvent  
'Los geht's!  
locFirstSortThread.StartSortingAsynchron()  
locSecondSortThread.StartSortingAsynchron()  
'Warten, bis beide Sortier-Threads beendet sind; dabei die Erkennung eines  
'möglichen Programmabbruchs offen halten.  
Do While Not AutoResetEvent.WaitAll(locAutoResetEvents, 1, True)  
    If myAbortAllThreads = True Then  
        locFirstSortThread.Dispose()  
        locSecondSortThread.Dispose()  
        RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler  
        Exit Sub  
    End If  
Loop  
  
'Beide sortierten String-Arrays wieder zusammenführen.  
Dim locIndexOnSecond As Integer  
Dim locTempArray As New ArrayList(cAmountOfElements)  
For locIndexOnFirst As Integer = 0 To locFirstAL.Count - 1  
    Do  
        If locIndexOnSecond < locSecondAL.Count Then  
            If CType(locFirstAL(locIndexOnFirst), IComparable).CompareTo( _  
                CType(locSecondAL(locIndexOnSecond), IComparable)) < 0 Then  
                locTempArray.Add(locFirstAL(locIndexOnFirst))  
                Exit Do  
            Else  
                locTempArray.Add(locSecondAL(locIndexOnSecond))  
                locIndexOnSecond += 1  
            End If  
        Else  
            locTempArray.Add(locFirstAL(locIndexOnFirst))  
            Exit Do  
        End If  
    Loop  
'Möglichen Abbruch auch an dieser Stelle berücksichtigen.
```

```

        If myAbortAllThreads = True Then
            locFirstSortThread.Dispose()
            locSecondSortThread.Dispose()
            RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler
            Exit Sub
        End If
    Next
    'Falls es Reste aus dem zweiten Array gibt, diese auch noch kopieren.
    If locIndexOnSecond < (locSecondAL.Count - 1) Then
        For c As Integer = locIndexOnSecond To locSecondAL.Count - 1
            locTempArray.Add(locSecondAL(c))
        Next
    End If
    locStrings = CType(locTempArray.ToArray(GetType(String)), String())

    locGauge.Stop()
    AddTBText("...in " + locGauge.DurationInMilliseconds.ToString() + " Millisekunden" +
vbNewLine)
    AddTBText(vbNewLine)
End If

'Falls das entsprechende Flag gesetzt ist, die Ergebnisse in die Textbox schreiben.
If cShowResults Then
    For Each s As String In locStrings
        If myAbortAllThreads Then
            RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler
            Exit Sub
        End If
        AddTBText(s + vbNewLine)
        AddTBText(vbNewLine)
    Next
End If

'Ereignis auslösen, das die Schaltflächen wieder einschaltet.
RaiseEvent MainThreadFinished()
'Ereignishandler wieder entfernen.
RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler

End Sub

```

WICHTIG Wenn ein Thread einen Ereignishandler über RaiseEvent aufruft, gehört dieser Ereignishandler immer zum Ereignis auslösenden Thread, ganz egal, welcher Klasse der Ereignis-Handler zugeordnet ist. Deshalb gilt auch für eine Routine, die durch ein Thread-Ereignis aufgerufen wurde: Steuerelemente, die hier manipuliert werden, können nur über Invoke manipuliert werden, damit sie innerhalb des UI-Threads verändert werden.

HINWEIS Diese Tatsache macht die Thread-Programmierung von Windows-Benutzeroberflächen vergleichsweise aufwändig. Abhilfe schafft hier die BackgroundWorker-Komponente, die im Abschnitt »Threads durch den Background-Worker initialisieren« ab Seite 1357 besprochen wird.

```

Private Sub MainThreadFinishedHandler()
    Console.WriteLine("MainThread beendet: " + Thread.CurrentThread.Name)
    Me.Invoke(New TSEnableControlsDelegate(AddressOf TSEnableControlsActually))

```

```
End Sub

Private Sub TSEnableControlsActually()
    btnBeenden.Enabled = True
    btnBenchmarkStarten.Enabled = True
End Sub

Private Sub btnBeenden_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnBeenden.Click
    'Nur schließen. OnClosing synchronisiert.
    Me.Close()
End Sub

Protected Overrides Sub OnClosing(ByVal e As System.ComponentModel.CancelEventArgs)
    'Falls eine TextBox-Ausgabe läuft, warten, bis diese abgeschlossen ist.
    'Sonst erfolgt die Ausgabe womöglich gerade zu der Zeit, wenn der Anwender
    'das Formular geschlossen hat. Dann wird die TextBox allerdings entladen,
    'ihr Handle zerstört, und die Ausgabe löst eine Ausnahme aus, weil sie das
    'zerstörte Handle der TextBox verwenden will.
    Do While Not myAutoResetEvent.WaitOne(1, True)
        Application.DoEvents()
    Loop
    'Ausgabe sicher --> jetzt erst abbrechen.
    myAbortAllThreads = True
End Sub

End Class
```

TIPP Denkbar für die Lösung dieses Beispiels ist auch noch eine andere Vorgehensweise, die obendrein noch effektiver ist: Der Haupt-Thread könnte auf die AutoResetEvent-Objekte zum Warten auf das Beenden des Sortiervorgangs komplett verzichten und sich stattdessen selbst mit Suspend aussetzen, nachdem er das Sortieren angeworfen hat. Die Sortierroutinen könnten ihrerseits durch das Auslösen des »Fertig!«-Ereignisses einen Ereignis-Handler aufrufen, der den Haupt-Thread wieder in Gang und schließlich zum Abschluss bringt. Das Beispiel des folgenden Abschnittes, dessen Hauptaufgabe eigentlich die Demonstration des Thread-Pools ist, verdeutlicht diese Vorgehensweise nebenbei.

Verwenden des Thread-Pools

In den bisherigen Beispielen haben Sie Threads als Instrument kennen gelernt, die nur dann einmalig hervorgeholt werden, wenn man sie braucht. In Anwendungen, die unter Praxisbedingungen arbeiten, sieht das oft anders aus: Sicherlich lässt sich voraussagen, welche Threads in Anwendungen häufiger und welche weniger häufig benötigt werden; doch in jedem Fall macht es die Praxis notwendig – im Gegensatz zu den bislang gezeigten Beispielen – dass Thread-Objekte ständig erzeugt, verwendet, ausgesetzt und wieder zerstört werden; unter Umständen passiert das hunderte Male in einer komplexen Applikation.

Das Erstellen eines Threads gehört für das Betriebssystem mit zu den aufwändigeren Dingen, die es zu erfüllen hat. Schon das Erstellen eines Threads kostet, gerade wenn es für ständig sich wiederholende, kleinere Aufgaben zigfach wiederholt werden muss, wertvolle Ressourcen. Aus diesem Grund bedient man sich eines Tricks, wenn innerhalb einer Anwendung viele kleine Threads eingesetzt werden sollen. Threads werden nicht ständig neu, sondern innerhalb eines so genannten Thread-Pools definiert. Wenn eine Anwendung einen Thread aus dem Thread-Pool anfordert, dann wird dieser beim ersten Mal zwar auch neu

erstellt, doch anschließend nicht sofort wieder gelöscht, wenn die Anwendung ihn nicht mehr benötigt. Stattdessen bleibt seine Grunddefinition im Pool erhalten, und das Thread-Objekt muss nicht komplett neu erstellt werden, wenn die Anwendung es für eine andere Aufgabe benötigt. Sie muss lediglich die Adresse der neuen Arbeits-Thread-Prozedur angeben; ansonsten kann das Thread-Objekt des Thread-Pools weiterverwendet werden.

Das Anlegen eines Threads aus dem Thread-Pool gestaltet sich eigentlich noch simpler als das Anlegen eines Threads mit der herkömmlichen Thread-Klasse. Sie übergeben der statischen Methode `QueueUserWorkItem` lediglich die Adresse der Prozedur, die als Thread ausgeführt werden soll. Im Gegensatz zum normalen Thread können Sie der Thread-Routine sogar eine Variable vom Typ `Object` als Parameter mitgeben (optional, Sie müssen also nicht).

HINWEIS Threads, die über den Thread-Pool erstellt werden, laufen als so genannte Hintergrund-Threads. Diese Threads haben eine besondere Eigenschaft: Wenn der Haupt- bzw. UI-Thread der Applikation endet, werden auch die Hintergrund-Threads automatisch beendet.

Zusammenfassend gesagt, haben also Thread-Pool-Threads folgende Vorteile:

- Sie benötigen keinen zusätzlichen Programmieraufwand, um sie vorzeitig beenden zu können, da sie als Hintergrund-Thread eingerichtet werden.
- Sie benötigen kein zusätzliches Delegate-Objekt (Startobjekt), um gestartet zu werden.
- Sie werden in einem Rutsch definiert und gestartet.
- Ihre benötigten Ressourcen lassen sich schneller reservieren, da intern die Thread-Instanz nicht gelöscht sondern nur »ruhiggestellt« wird und damit ohne Aufwand für andere Zwecke wiederverwendet werden kann.
- Der Arbeits-Thread-Routine kann einen Parameter vom Typ `Object` empfangen.

Allerdings gibt es auch Nachteile: Maximal stehen pro verfügbarem Prozessor 25 Threads im Thread-Pool zur Verfügung. Wenn Sie einen neuen Thread darüber hinaus anfordern, wartet die dafür zuständige Methode `QueueUserWorkItem` solange, bis ein zurzeit noch reserviertes Thread-Objekt des Pools wieder verfügbar ist.

BEGLEITDATEIEN Das folgende Beispiel demonstriert das Sortieren mit Threads aus dem Thread-Pool. Sie finden das Projekt dieses Beispiels im Verzeichnis

...\\VB 2008 Entwicklerbuch\\I - Threading\\Kapitel45\\ThreadPoolThreads

Da es weitestgehend dem vorherigen Beispiel entspricht, zeigt das folgende Codelisting nur die geänderten Stellen.

Betrachten wir zunächst das Codelisting des Sortier-Threads, der nunmehr auf einem Threadpool-Thread beruht (die eigentliche Sortierroutine ist in dieser Version aus Platzgründen ausgelassen).

```
Public Class SortArrayThreaded

    Public Event SortCompleted(ByVal sender As Object, ByVal e As SortCompletedEventArgs)
    Private myArrayToSort As ArrayList
    Private myTerminateThread As Boolean
```

```
Private mySortingComplete As Boolean
Private myThreadName As String

'Konstruktor übernimmt die zu sortierende Liste. Der ThreadName hat nur
'dokumentarischen Charakter.
Sub New(ByVal ArrayToSort As ArrayList, ByVal ThreadName As String)
    myArrayToSort = ArrayToSort
    myThreadName = ThreadName
End Sub

'Thread starten
Public Sub StartSortingAsynchron()
    ThreadPool.QueueUserWorkItem(AddressOf SortArray)
End Sub

Private Sub SortArray(ByVal state As Object)
    .
    'Wie gehabt.
    .
End Function

Public ReadOnly Property ArrayList() As ArrayList
    Get
        Return myArrayToSort
    End Get
End Property
End Class
```

Damit eine den Thread in Gang setzende Instanz weiß, welcher Thread beendet wurde, wenn mehrere Sortierungen gestartet werden, sind die Ereignisparameter in dieser Version leicht erweitert worden und geben nunmehr auch den Thread-Namen zurück:

```
'Dient nur der Ereignisübergabe, wenn Sie das SortCompleted-Ereignis
'nach Abbruch oder Abschluss des Sortierens auslösen wollen.
Public Class SortCompletedEventArgs
    Inherits EventArgs

    Private mySortCompletionStatus As SortCompletionStatus
    Private myThreadName As String

    Sub New(ByVal scs As SortCompletionStatus, ByVal ThreadName As String)
        mySortCompletionStatus = scs
    End Sub

    Public Property ThreadName() As String
        Get
            Return myThreadName
        End Get
        Set(ByVal Value As String)
            myThreadName = Value
        End Set
    End Property
```

```

Public Property SortCompletionStatus() As SortCompletionStatus
    Get
        Return mySortCompletionStatus
    End Get
    Set(ByVal Value As SortCompletionStatus)
        mySortCompletionStatus = Value
    End Set
End Property

End Class

```

Die wesentlichen Unterschiede im eigentlichen Arbeits-Thread beschränken sich auf das Warten des Sortierungsabschlusses. Während im vorherigen Beispiel eine Warteschleife einiges an Rechenzeit gekostet hat, wird der Arbeits-Thread hier komplett ausgesetzt. Erst die Ereignisbehandlungsroutine des SortCompleted-Ereignisses setzt den Arbeits-Thread wieder in Gang, der seine Aufgabe dann zu Ende führen kann:

```

'Dies ist die eigentliche Haupt-Thread-Routine (aber nicht der UI-Thread!), die das Testarray
'generiert und dann selbst wiederum entweder ein oder zwei Arbeits-Threads aufruft,
'die die eigentliche Sortierung durchführen.
Private Sub StartMainThread()

    Dim locStrings(cAmountOfElements - 1) As String
    Dim locGauge As New HighSpeedTimeGauge

    Dim locRandom As New Random(DateTime.Now.Millisecond)
    Dim locChars(cCharsPerString) As Char

    'Damit die Controls nach Abschluss des Sortierens auf dem Formular
    'wieder eingeschaltet werden können, muss der Haupt-Thread das Ende
    'der Sortierung mitbekommen. Deswegen: Ereignis
    AddHandler MainThreadFinished, AddressOf MainThreadFinishedHandler

    'String-Array erzeugen; hatten wir schon zig Mal...
    AddTBText("Erzeugen von " + cAmountOfElements.ToString + " Strings..." + vbNewLine)
    locGauge.Start()
    For locOutCount As Integer = 0 To cAmountOfElements - 1
        For locInCount As Integer = 0 To cCharsPerString - 1
            Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)
            If locIntTemp > 26 Then
                locIntTemp += 97 - 26
            Else
                locIntTemp += 65
            End If
            locChars(locInCount) = Convert.ToChar(locIntTemp)
        Next
        locStrings(locOutCount) = New String(locChars)
    Next
    locGauge.Stop()
    AddTBText("...in " + locGauge.DurationInMilliSeconds.ToString + " Millisekunden" + vbNewLine)
    AddTBText(vbNewLine)
    locGauge.Reset()

    If Not myUseTwoThread Then
        'Messung starten - hier wird mit nur einem Thread sortiert.
    End If
End Sub

```

```
    locGauge.Start()
    Dim locFirstSortThread As New SortArrayThreaded(New ArrayList(locStrings), "1. SortThread")

    AddTBText("Sortieren von " + cAmountOfElements.ToString() + " Strings mit einem Thread..." _
              + vbNewLine)

    'Handler hinzufügen, der das Ende des Sortierens mitbekommt und dann wiederum
    'mySortCompletion hochzählt, damit die Warteschleife des Hauptthreads beendet werden kann
    AddHandler locFirstSortThread.SortCompleted, AddressOf SortCompletionHandler

    'los geht's
    locFirstSortThread.StartSortingAsynchron()

    'Warten, bis der Sortier-Thread abgeschlossen ist.
    'mySortCompletion wird durch das SortCompletion-Ereignis hochgezählt
    'wenn es 1 erreicht hat, schmeißt der Ereignishandler diesen Thread wieder an.
    mySyncUIResetEvent.WaitOne()

    'Handler wieder entfernen
    RemoveHandler locFirstSortThread.SortCompleted, AddressOf SortCompletionHandler

    'Messung beenden
    locGauge.Stop()
    AddTBText("...in " + locGauge.DurationInMilliseconds.ToString() + " Millisekunden" +
vbNewLine)
    AddTBText(vbNewLine)
    'Array-List zurück-casten.
    locStrings = CType(locFirstSortThread.ArrayList.ToArray(GetType(String)), String())
    Else
        'Es soll mit zwei Threads sortiert werden
        .
        .
        .
    End Sub
```

Der Thread hält hier, nachdem er die Ereignisbehandlungsroutine eingerichtet und den Sortiervorgang gestartet hat, komplett an (siehe fett markierte Zeilen im Listing). Durch die folgende Ereignisroutine wird er wieder zum Leben erweckt, wenn der Sortiervorgang abgeschlossen wurde:

```
Private Sub SortCompletionHandler(ByVal sender As Object, ByVal e As SortCompletedEventArgs)
    mySortCompletion += 1
    If myUseTwoThread Then
        If mySortCompletion = 2 Then
            mySyncUIResetEvent.Set()
        End If
    Else
        If mySortCompletion = 1 Then
            mySyncUIResetEvent.Set()
        End If
    End If
End Sub
```

HINWEIS Diese Version des Beispiels verzichtet der Einfachheit halber auf die Möglichkeit, den Thread vorzeitig abzubrechen.

Thread-sichere Formulare in Klassen kapseln

Steuerelemente können von Threads aus nur durch deren Invoke Thread-sicher bearbeitet werden – die vergangenen Abschnitte haben das in aller Ausführlichkeit gezeigt. Wenn Sie Ihre Formulare Thread-sicher gestalten wollen, dann ist es das Beste, wenn Sie nach Möglichkeit die dazugehörigen Threads in den Formular-Klassen kapseln – etwa, wie es der Abschnitt »Datenaustausch zwischen Threads durch Kapseln von Threads in Klassen« ab Seite 1338 gezeigt hat. Auch dabei gilt natürlich, dass Sie die Steuerelemente eines solchen Formulars aus seinen Arbeits-Threads heraus nur über Invoke verarbeiten dürfen.

Für einige Anwendungen kann es jedoch sinnvoll sein, dass ein Thread selbst ein Formular erstellt und es unter seine Verwaltung stellt. Denken Sie an die Beispiele, die Sie schon kennen gelernt haben: Einige der ersten Testprogramme haben auf eine ADThreadSafeInfoBox-Klasse zurückgegriffen, die die Thread-sichere Ausgabe von Text in einem Formular erlaubte, obwohl an keiner Stelle die Instanziierung einer auf *Form* basierten Klasse erfolgte.

Wenn ein Thread völlig unabhängig vom dem ihn einbindenden Programm ein Formular erstellen will, hat er nur eine Chance: Er muss für das Formular (und alle weiteren, die dieses Formular aufruft), eine eigene Warteschlange implementieren.

Wenn Sie also eine komplette Klasse schaffen wollen, die einen Arbeits-Thread beinhaltet, der selbst auf ein Formular zugreifen muss, dann sind folgende Schritte nötig:

- Die den Thread startende Prozedur muss zunächst einen neuen Thread erstellen, der zum zusätzlichen UI-Thread wird.
- Dieser neue Thread instanziert dann das Hauptformular des neuen UI-Threads und bindet das Ereignis Application.ThreadExit ein.
- Mit Application.Run kann er nun im neu geschaffenen Thread eine Nachrichtenwarteschlange erstellen und die Instanz des Formulars an diese binden. Der Thread ist dann automatisch beendet, wenn der Anwender (oder eine andere Instanz) das gebundene Formular auf irgendeine Weise schließt.
- Für den Fall, dass die umgebende Applikation zuvor geschlossen wurde, löst das Application-Objekt das ThreadExit-Ereignis aus. Die Klasse muss in der Ereignisbehandlungsroutine dieses Ereignisses ein paar Aufräumarbeiten durchführen: Durch das eigentliche Programmende ist der zweite UI-Thread nämlich noch nicht ebenfalls automatisch abgeschlossen. Ein explizites Application.ExitThread wird an dieser Stelle noch notwendig, um den zweiten UI-Thread spätestens jetzt zu beenden.

Das folgende Beispiel geht sogar noch einen Schritt weiter, um absolute Unabhängigkeit zu erlangen: Es erstellt den zweiten UI-Thread im statischen Konstruktor der Klasse. Wenn eine andere Instanz die statische Funktion TSWrite (oder TSWriteLine) das erste Mal aufruft, um eine Ausgabe in das Statusfenster durchzuführen, sorgt der statische Konstruktor dafür, dass der neue UI-Thread erstellt wird. Die Ausgabe erfolgt dann anschließend in das nun vorhandene Fenster (durch ein einfaches TextBox-Steuerelement simuliert). Damit der Konstruktor dem neuen UI-Thread ein wenig Zeit gibt, in Gang zu kommen und das Formular zu instanziieren, und damit verhindert wird, dass TSWrite[Line] eine Ausgabe in ein Steuerelement durchführt, das noch nicht existiert, erfolgt die notwendige Synchronisierung mit einem ManualResetEvent-Objekt.

```
Imports System.Threading  
  
Public Class ADThreadSafeInfoBox
```

```
Private Shared myText As String
Private Shared myInstance As ADThreadSafeInfoBox
Private Shared myThread As Thread
Private Shared myManualResetEvent As ManualResetEvent

Private Delegate Sub TSWriteActuallyDelegate()
Private Delegate Sub ThisInstanceCloseDelegate()

Private Shared myThisInstanceCloseDelegate As ThisInstanceCloseDelegate

Private Shared Sub ThisInstanceCloseActually()
    myInstance.Dispose()
End Sub

'Statischer Konstruktor erstellt neuen UI-Thread
Shared Sub New()
    myText = ""
    myManualResetEvent = New ManualResetEvent(False)
    CreateInstanceThroughThread()
End Sub

'Teil des Konstruktors; könnte theoretisch auch von
'anderswo in Gang gesetzt werden. Diese Routine erstellt
'den neuen UI-Thread und startet ihn...
Private Shared Sub CreateInstanceThroughThread()
    myThread = New Thread(New ThreadStart(AddressOf CreateInstanceThroughThreadActually))
    myThread.Name = "2. UI-Thread"
    myThread.Start()
End Sub

'...und der neue UI-Thread erstellt jetzt das Formular
'und bindet es an eine neue Nachrichtenwarteschlange.
Private Shared Sub CreateInstanceThroughThreadActually()
    'Instanz des Formulars erstellen
    myInstance = New ADThreadSafeInfoBox
    'Wir müssen auf jeden Fall wissen, wann die Hauptapplikation (der Haupt-UI-Thread) geht.
    myThisInstanceCloseDelegate = New ThisInstanceCloseDelegate(AddressOf ThisInstanceCloseActually)

    AddHandler Application.ThreadExit, AddressOf ThreadExitHandler
    'Und wir müssen wissen, ab wann das Formular wirklich existiert;
    'vorher dürfen keine Ausgaben ins Formular erfolgen
    AddHandler myInstance.HandleCreated, AddressOf HandleCreatedHandler
    'und ab wann nicht mehr. Für dieses Beispiel ist dieses Ereignis nicht soooo wichtig.
    AddHandler myInstance.HandleDestroyed, AddressOf HandleDestroyedHandler
    'Hier wird die Warteschlange gestartet
    Application.Run(myInstance)
End Sub

'Dieser Ereignis-Handler wird aufgerufen, wenn das Hauptprogramm beendet wird.
'Der zweite UI-Thread wird damit beendet.
Private Shared Sub ThreadExitHandler(ByVal sender As Object, ByVal e As EventArgs)
    Console.WriteLine("ThreadExit")
    'Keine TextBox mehr vorhanden:
    ' Synchronisationsvoraussetzung für TSWrite schaffen
    myManualResetEvent.Reset()
    Try
```

```
myInstance.Invoke(myThisInstanceCloseDelegate)
Catch ex As Exception
End Try
End Sub

'TSWrite signalisieren, dass die Ausgabe in die TextBox jetzt sicher ist,
'da das Formular-Handle erstellt wurde.
Private Shared Sub HandleCreatedHandler(ByVal sender As Object, ByVal e As EventArgs)
    Console.WriteLine("HandleCreated")
    myManualResetEvent.Set()
End Sub

'Vielleicht später mal wichtig; hier nur zur Demo.
Private Shared Sub HandleDestroyedHandler(ByVal sender As Object, ByVal e As EventArgs)
    'Nur für Testzwecke
    Console.WriteLine("HandleDestroyed")
End Sub

'Nutzt TSWrite; siehe dort.
Public Shared Sub TSWriteLine(ByVal Message As String)
    SyncLock (GetType(ADThreadSafeInfoBox))
        Message += vbNewLine
        TSWrite(Message)
    End SyncLock
End Sub

'Ausgabe ohne neue Zeile
Public Shared Sub TSWrite(ByVal Message As String)
    SyncLock (GetType(ADThreadSafeInfoBox))
        'Synchronisierung: Wenn nach 50 Millisekunden
        'keine TextBox vorhanden ist --> Befehl ignorieren
        If Not myManualResetEvent.WaitOne(50, True) Then
            Exit Sub
        End If
        myText += Message
        Try
            myInstance.Invoke(New TSWriteActuallyDelegate(AddressOf TSWriteActually))
        Catch ex As Exception
        End Try
    End SyncLock
End Sub
'Thread-sichere Ausgabe in die TextBox mit Invoke
Private Shared Sub TSWriteActually()
    myInstance.txtOutput.Text = myText
    myInstance.txtOutput.SelectionStart = myText.Length - 1
    myInstance.txtOutput.ScrollToCaret()
End Sub
End Class
```

Threads durch den Background-Worker initialisieren

Neu im .NET Framework 2.0 ist die so genannte `BackgroundWorker`-Komponente, die die meiner Meinung nach einfachste Methode darstellt, eine rechenintensive Methode in einem anderen Thread laufen zu lassen.

Falls Sie dieses Kapitel aufmerksam studiert haben, werden Sie feststellen, dass der Einsatz dieser Komponente zwar nicht so viel Flexibilität bietet, wie die Programmierung von Threads über das `Thread`-Objekt; aber durch eine besondere intern angewandte Technik hat sie einen unschlagbaren Vorteil: Sie kann Statusmeldungen über Ereignisse zur Verfügung stellen, die im Ursprungs-Thread (in der Regel also dem UI-Thread) laufen; der Aufwand, sich `Invoke` bedienen zu müssen, um beispielsweise ein `Label`-Steuerelement zu aktualisieren, fällt also weg.

Die Vorgehensweise, um mithilfe der `BackgroundWorker`-Komponente eine Prozedur als neuen Thread laufen zu lassen, ist außergewöhnlich aber simpel: Ihr Thread läuft als Ereignisbehandlungsprozedur.

- Sie triggern die `BackgroundWorker`-Komponente mit der Methode `RunWorkerAsync`. Die wiederum sorgt dafür, dass das `DoWork`-Ereignis ausgelöst wird, und ihre implementierte Behandlungsroutine, die dieses Ereignis behandelt, *ist* die Prozedur, die auf einem anderen Thread läuft.
- Möchten Sie, dass Fortschrittsinformationen aus ihrer Thread-Routine heraus gesendet werden, sorgen Sie vor Beginn des Threads, dass die `WorkerReportsProgress`-Eigenschaft auf `True` gesetzt ist, denn nur dann unterstützt die `BackgroundWorker`-Komponente das Berichten des Fortschrittsstatus. Auch die Kommunikation des Fortschritts funktioniert über Ereignisse. Wenn die entsprechenden Vorbereitungen getroffen wurden, lösen Sie zu gegebener Zeit innerhalb Ihres Threads (also dem `DoWork`-Ereignisbehandler) das `ProgressChanged`-Ereignis aus, das Sie mit der `ReportProgress`-Methode initialisieren.

WICHTIG Dieses Ereignis läuft dann, anders als zu erwarten wäre, auf dem Thread, der die `BackgroundWorker`-Komponente kapselt (in der Regel also dem UI-Thread), und nicht auf dem Thread, der das `ProgressChanged`-Ereignis ausgelöst hat. UI-Komponenten können deswegen direkt in diesem Ereignis angesprochen werden!

- Wenn der `DoWork`-Thread beendet wurde, löst die `BackgroundWorker`-Komponente automatisch das `RunWorkerCompleted`-Ereignis aus. Eine Prozedur, die dieses Ereignis behandelt, läuft auch wieder auf dem ursprünglichen Thread (also in der Regel dem UI-Thread) und nicht auf dem Thread, der durch `DoWork` initiiert wurde.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\I - Threading\\Kapitel45\\BackgroundWorkerDemo

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

Das Beispiel dient zur Berechnung von Primzahlen in einem eigenen Thread. Wenn Sie es starten, sehen Sie einen Dialog, wie Sie ihn auch in Abbildung 44.7 sehen können.

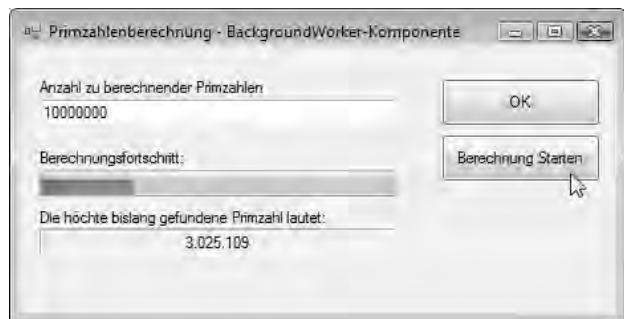


Abbildung 44.7 Mit dem BackgroundWorker können Sie auf einfachste Weise neue Threads starten, und bekommen Fortschritts-Informationen auf dem ursprünglichen UI-Thread!

Der entsprechende Code des Programms sieht folgendermaßen aus:

```

Public Class frmMain

    Private myPrimzahlen As List(Of Long)

    Private Sub btnBerechnungStarten_Click(ByVal sender As System.Object,
                                         ByVal e As System.EventArgs) Handles btnBerechnungStarten.Click

        'Festlegen, dass der BackgroundWorker über den Fortschritt berichten können soll.
        PrimzahlenBackgroundWorker.WorkerReportsProgress = True

        'Fortschrittsanzeige konfigurieren.
        pbBerechnungsfortschritt.Minimum = 0
        pbBerechnungsfortschritt.Maximum = 1000
        pbBerechnungsfortschritt.Value = 0
        lblGefundenText.Text = "Die höchste bislang gefundene Primzahl lautet:"

        'Ereignis auslösen lassen, damit der Thread
        'in der DoWork-Ereignisbehandlungsroutine gestartet werden kann.
        'Die Obergrenze wird aus dem Textfeld als Argument übergeben.
        Me.PrimzahlenBackgroundWorker.RunWorkerAsync(txtAnzahlPrimzahlen.Text)
    End Sub

    Private Sub PrimzahlenBackgroundWorker_DoWork(ByVal sender As Object,
                                                ByVal e As System.ComponentModel.DoWorkEventArgs) _
                                                Handles PrimzahlenBackgroundWorker.DoWork

        'Das Argument (die Obergrenze) aus den Ereignisparametern wieder herauslesen.
        Dim locMax As Integer = CInt(e.Argument)

        'Ein paar Variablen für die Fortschrittsprozentanzeige einrichten.
        '(OK: Es sind Promille).
        Dim locProgressFaktor As Double = 1000 / locMax
        Dim locProgressAlt As Integer = 0
        Dim locProgressAktuell As Integer

        'Fertig, wenn Obergrenze kleiner 3 ist.
        If (locMax < 3) Then Return

        'Neues Array definieren, das die Primzahlen enthält.
        '(Wir verwenden als Algorithmus das Sieb des Eratosthenes.)
    End Sub

```

```
myPrimzahlen = New List(Of Long)

For z As Integer = 2 To locMax
    If IstPrimzahl(myPrimzahlen, z) Then
        myPrimzahlen.Add(z)

        'Progressinformation senden. Das darf, da der Thread durch SendMessage
        '"gewechselt" wird, nicht zu häufig passieren, sonst hängt der
        'UI-Thread, der die BackgroundWorker-Komponente anfangs initiiert hat.
        locProgressAktuell = CInt(z * locProgressFaktor)
        If locProgressAktuell > locProgressAlt Then
            locProgressAlt = locProgressAktuell
            'Das ProgressChange-Ereignis auslösen, um über den
            'Fortschritt zu informieren.
            PrimzahlenBackgroundWorker.ReportProgress(locProgressAktuell)
        End If
    End If
Next
End Sub

Private Function IstPrimzahl(ByVal Primzahlen As List(Of Long), ByVal Number As Long) As Boolean
    For Each locTestZahl As Long In Primzahlen
        If (Number Mod locTestZahl = 0) Then Return False
        If (locTestZahl >= Math.Sqrt(Number)) Then Exit For
    Next
    Return True
End Function

'Wird aufgerufen, wenn eine BackgroundWorker-Komponente mit
'ReportProgress über den Fortschritt informiert. Dieser Ereignisbehandler
'läuft dennoch auf dem UI-Thread und nicht auf dem DoWork-Thread. Er kann daher
'ohne Probleme Steuerelemente direkt manipulieren.
Private Sub PrimzahlenBackgroundWorker_ProgressChanged(ByVal sender As Object, _
    ByVal e As System.ComponentModel.ProgressChangedEventArgs)
    Handles PrimzahlenBackgroundWorker.ProgressChanged
    pbBerechnungsfortschritt.Value = e.ProgressPercentage
    lblHöchstePrimzahl.Text = myPrimzahlen(myPrimzahlen.Count - 1).ToString("#,##0")
End Sub

'Wird aufgerufen, wenn der DoWork-Thread beendet wurde.
'Auch dieser Ereignisbeandler läuft auf dem UI-Thread.
Private Sub PrimzahlenBackgroundWorker_RunWorkerCompleted(ByVal sender As Object, _
    ByVal e As System.ComponentModel.RunWorkerCompletedEventArgs)
    Handles PrimzahlenBackgroundWorker.RunWorkerCompleted
    lblGefundenText.Text = "Die höchste gefundene Primzahl im Bereich lautet:"
    lblHöchstePrimzahl.Text = myPrimzahlen(myPrimzahlen.Count - 1).ToString("#,##0")
End Sub
End Class
```

HINWEIS Achten Sie darauf, das ProgressChanged-Ereignis nicht zu häufig auszulösen, da Sie den UI-Thread unter Umständen durch das Aktualisieren von Steuerelementen überlasten könnten. Der DoWork-Thread läuft dabei zwar regelmäßig weiter, der UI-Thread reagiert dann aber quasi nicht mehr.

Threads durch asynchrone Aufrufe von Delegaten initiieren

Delegaten haben Sie in Kapitel 14 kennen gelernt, und für genauere Informationen zu diesem Thema lesen Sie bitte dort nach. Kurz zusammengefasst: Sie dienen zum indirekten Aufruf von Methoden über in bestimmten Objektvariablen abgelegten Adressen dieser Methoden.

Mit der Methode `BeginInvoke` haben Sie die Möglichkeit, eine Prozedur, die durch einen Delegaten dargestellt wird, asynchron, also auf einem eigenen Thread laufen zu lassen. Sie sollten `EndInvoke` aufrufen, wenn die Arbeit erledigt ist. Mit `BeginInvoke` haben Sie die Möglichkeit, einen zweiten Delegaten anzugeben, der aufgerufen wird, wenn die Arbeit abgeschlossen wurde.

Das folgende Beispiel, das ebenfalls Primzahlen berechnet, demonstriert den Einsatz des asynchronen Ausführens von Prozeduren über Delegaten.

BEGLEITDATEIEN

Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

...\\VB 2008 Entwicklerbuch\\ I - Threading\\Kapitel45\\AsyncDelegates

Öffnen Sie dort die entsprechende Projektmappe (.SLN-Datei).

```
Imports System.Collections.Generic
Imports System.ComponentModel

Public Class Form1

    Private Delegate Sub PrimzahlenErstellenDelegate(ByVal Max As Integer)
    Private Delegate Sub ErgebnisSetzenDelegate(ByVal ErgebnisText As String)

    Private myPrimzahlen As List(Of Long)
    Private myPZDelegate As PrimzahlenErstellenDelegate
    Private myErgebnisSetzenDelegate As ErgebnisSetzenDelegate
    Private myAsyncOperation As AsyncOperation

    Sub New()

        ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
        InitializeComponent()

        ' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.
        myErgebnisSetzenDelegate = New ErgebnisSetzenDelegate(AddressOf ErgebnisSetzen)
    End Sub

    Private Sub btnBerechnungStarten_Click(ByVal sender As System.Object,
                                           ByVal e As System.EventArgs) Handles btnBerechnungStarten.Click
        'Delegate definieren.
        myPZDelegate = New PrimzahlenErstellenDelegate(AddressOf PrimzahlenErstellen)

        'Delegate ansynchron aufrufen.
        myPZDelegate.BeginInvoke(Integer.Parse(txtAnzahlPrimzahlen.Text),
                               AddressOf BerechnungAbgeschlossenCallback, Nothing)
    End Sub
```

```
Private Sub PrimzahlenErstellen(ByVal Max As Integer)

    If (Max < 3) Then Return
    myPrimzahlen = New List(Of Long)

    For z As Integer = 2 To Max
        If IstPrimzahl(myPrimzahlen, z) Then
            myPrimzahlen.Add(z)
            'Direkte Manipulation des Steuerelements ist nicht möglich,
            'da dieser Thread nicht dem UI-Thread entspricht.
            lblHöchstePrimzahl.Invoke(myErgebnisSetzenDelegate,
                myPrimzahlen(myPrimzahlen.Count - 1).ToString("#,#0"))
        End If
    Next
End Sub

Private Function IstPrimzahl(ByVal Primzahlen As List(Of Long), ByVal Number As Long) As Boolean

    For Each locTestZahl As Long In Primzahlen
        If (Number Mod locTestZahl = 0) Then Return False
        If (locTestZahl >= Math.Sqrt(Number)) Then Exit For
    Next
    Return True
End Function

'Delegatroutine für das Aktualisieren der Benutzeroberfläche.
'Notwendig, weil die Aktualisierung auf dem Thread des Delegaten ausgeführt wird.
Private Sub ErgebnisSetzen(ByVal ErgebnisText As String)
    lblHöchstePrimzahl.Text = ErgebnisText
End Sub

'Der Rückrufroutine des Delegaten, der aufgerufen wird, wenn seine Aufgabe beendet ist.
Private Sub BerechnungAbgeschlossenCallback(ByVal ar As System.IAsyncResult)
    lblHöchstePrimzahl.Invoke(myErgebnisSetzenDelegate,
        myPrimzahlen(myPrimzahlen.Count - 1).ToString("#,#0"))
    myPZDelegate.EndInvoke(ar)
End Sub

Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
    Me.Close()
End Sub
End Class
```


Stichwortverzeichnis

#End Region 166
#Region 166
&= -Operator 340
*= -Operator 340
.cctor 425
.ctor 425, 455
.NET
 primitive Datentypen 208
 Quellcode 378
.NET Framework
 DateTime *siehe* Date-Datentyp
 Int16 *siehe* Short-Datentyp
 Int32 *siehe* Integer-Datentyp
 Int64 *siehe* Long-Datentyp
 UInt16 *siehe* UShort-Datentyp
 UInt32 *siehe* UInteger-
 Datentyp
 UInt64 *siehe* ULong-Datentyp
 Version Targeting 364
 verwendete Version einstellen
 364
.NET-Anwendungen
 Installations-Voraussetzungen 8
.NET-Framework
 Assembly 47
 Auflistungen 642
 Base Class Library 52
 BCL 52
 Class Library 52
 CLI 51
 CLR 51
 Common Language
 Infrastructure 51
 Common Language Runtime 51
 Common Type System 53, 55
 CTS 53, 55
 Eigenschaften überschreiben
 462
 FCL 52
 Framework, Aufbau 54
 IML 53
 Intermediate Language 53

JITter 53
Klassen erweitern 462
Managed Heap 396, 463
Methoden überschreiben 462
Namespace 47
Polymorphie *siehe*
 Polymorphie
Terminologien 46
Typsicherheit 55
was es ist 46
.NET-Version einstellen 8
/= 340
:IML
 Klassen, Umsetzung 455
+= Operator 340
<<-Operator 341
= 340
-= -Operator 340
>>-Operator 341

A

Abfangen von Fehlern 342
Abfragen
 auf mehreren Cores gleichzeitig
 881
 SQL und LINQ 881
Abstrakte Klassen 485
Abstrakte Objekte (Klassen) 31
Accessor *siehe* Eigenschaften,
 Accessor
 unterschiedliche für
 Lesen/Schreiben
 (Eigenschaften) 436
Accessor: 434
AccessText 1243
activedevelop.de 10
Add 230
AddHandler 600
AddressOf
 Anwendungsbeispiel 806
ADLabelEx-
 Benutzersteuerelement 1193
ADO.NET Entity Framework 981
AdventureWorks-
 Beispieldatenbank 920
AdventureWorks-Datenbank 928
Aggregatfunktionen 898
Aggregieren und gruppieren 899
Aktionen (List (Of)) 692
Aktivierreihenfolge 123
Aktivierungsreihenfolge 1050
Allgemeine
 Entwicklungseinstellungen 15, 63
All-Methode 955
Alpha Centauri 11
AndNot 348
Anonyme Typen 374
Anwendungen
 Mehrfachstart verhindern 851
Anwendungen in .NET
 Installations-Voraussetzungen 8
Anwendungseinstellungen *siehe*
 Settings
 automatisch speichern 851
 Bereiche 848
 veränderliche 848
Anwendungsereignisse 852
 NetworkAvailabilityChanged 854
 ShutDown 853
 Startup 853
 StartUpNextInstance 854
 UnhandledException 854
Anwendungsframework 850
 Aktivieren 850
 Authentifizierungsmodus für
 Benutzer 851
 Ereignisse *siehe*
 Anwendungsereignisse
 Herunterfahren, Modus 852
 Mehrfachstart verhindern 851
 Splash-Dialoge 852

- Anwendungsframework (*Fortsetzung*)
 Windows XP-Stile verwenden 851
- Anwendungsschichten 1020
 Anwendungslogik 1020
 Business Logic 1020
 Datenschicht 1020
 Geschäftslogik 1020
- Any-Methode 955
- Any-Zieltyp 54
- ApplicationCommands 1226
- Application-Objekt 187
- Argumente *siehe* Parameter
- ArrayList-Auflistung 642
 AddRange-Methode 642
 Arrays, umwandeln in 642
 Clear-Methode 642
 Count-Eigenschaft 642
 RemoveAt-Methode 642
 Remove-Methode 642
- Arrays 620
 ArrayList, aus 642
 BinarySearch-Methode 627
 Dimensionen ändern 621
 durchsuchen 627
 Eigenschaften, wichtige 625
 Enumeratoren 635
 Grundsätzliches 620
 konservieren, nach ReDim 623
 Konstanten 623
 Länge ermitteln 625
 Length-Methode 625
 mehrdimensional 624
 Methoden, wichtige 625
 Mutex-Objekte (Threading) 1328
 Preserve-Anweisung 623
 ReDim-Anweisung 621
 Reihenfolge ändern, Elemente 626
 Reverse-Methode 626
 Sortieren 625
 Sortierung, benutzerdefiniert 627
 Sort-Methoden 625
 suchen, Elemente, benutzerdefiniert 627
 Unterschiede zu VB6 339
 verschachtelt 624
 Wertevorbelegung 623
- ASCII umwandeln in Char-Datentyp 231
- AscW-Methode 231
- Assemblies 350
 mit mehreren Namespace-Definitionen 357
 Namen bestimmen für Projekte 356
 Verweise auf, hinzufügen 352
- Assembly
 Erklärung 47
 in Projekt einbinden 48
- Assembly (CTS) 434–436
- Assembly-Eigenschaft 735
- AssemblyQualifiedName-Eigenschaft 735
- Attached Properties (XAML) 198
- Attribute 728
 AttributeUsage 731
 benutzerdefinierte ermitteln 741
 COMClass 731
 DebuggerDisplay 378
 DllImport 731
 einfaches Beispiel 729
 Flags 616
 genereller Umgang 728
 MarshalAs 731
 Obsolete 729
 Serializable 731
 StructLayout 532
 Synchronization, zur (Threading) 1328
 VBFixedArray 731
 VBFixedString 731
 WebMethod 731
 zur Laufzeit ermitteln 737
- Attributes-Eigenschaft 735
- AttributeUsage-Attribut 731
- Aufbau von LINQ-Abfragen 881
- Aufgabe durch Kommentar 82
- Aufgabenliste 81
 Aufgabe hinzufügen 82
 Codekommentar 82
 Codetoken einrichten 82
 Navigieren im Code, mit 83
 Punkt hinzufügen 169
 Tipps 83
- Auflistungen
 ArrayList 642
 Captures (Reguläre Ausdrücke) 798
- Darstellen in Steuerelementen 1053
- Enumeratoren 635
- For/Each für benutzerdefinierte 635
- generische *siehe* Generische Auflistungen
- Groups (Reguläre Ausdrücke) 798
- Grundsätzliches 638
- Hashtable 648
- IList-Schnittstelle 646
- Items (ListBox) 504
- Key *siehe* Schlüssel, Abruf durch 797
- Load-Faktor 639
- Matches (Reguläre Ausdrücke) 797
- Queue 662
- Schlüssel, Abruf durch 648
- SortedList 664
- Stack 663
- typsichere *siehe* Generische Auflistungen vorhandene 642
- Auflistungen in Relation setzen 892, 894, 896, 897
- Aufrufliste 1108
- Aufzählungen 612
- Ausblenden von Projektdateien 74
- Ausdrücke 27
 boolesche 29
- Ausdrücke, komplexe 28
- Ausdrücke, reguläre *siehe* Reguläre Ausdrücke
- Ausgabefenster 83
 Debugger-Meldungen anzeigen 85
 löschen 84
 Navigieren im Code, mit 84
 Quelle einstellen 84
 Zeilenumbruch einstellen 84
- Ausnahmen 342
 Behandeln mit Try/Catch/Finally 344
 Codeausführung sicherstellen 347
- globale Behandlung 854
- OutOfRangeException 216
- typabhängig behandeln 345

Ausrichtungen 761
Aussageprüfer (List (Of)) 695
Auswahlrand 163
Autokorrektur für intelligentes Kompilieren 141
Automatischer Zeilenumbruch 163
Automatisches Vervollständigen *siehe* IntelliSense, Vervollständigungsliste
AutoResetEvent-Klasse (Threading) 1331
AutoScroll-Eigenschaft 121

B

Background Compiler 79, 138
Base Class Library 52
BaseType-Eigenschaft 735
BASIC 14
BCL 330
Bearbeitungshilfen *siehe* IntelliSense
Bedingte Programmausführung 38
Befehle 1215
Befehlszeilenargumente auslesen 835
BeginUpdate-Methode (ListView) 1055
Begleitdateien auf Buch-DVD 11
PDF-Dateien Visual Basic .NET Entwicklerbuch 11
Softwarevoraussetzungen 5
Updates und Errata 10
Begrüßungsdialoge *siehe* Splash-Dialoge
Beispieldatenbanken (SQL Server) 928
Benutzer Authentifizierungsmodus 851
Benutzereingaben überprüfen *siehe* Validieren
Benutzerkontensteuerung 368
Benutzersteurelemente 1170 ADLabelEx 1193 als Container 1180 Anordnen von Steuerelementen 1184 aus vorhandenen 1171 Basis-Window 1195

Beschriftung 1194
Blinken 1203
Borderline-Eigenschaft 1195
Browsable-Attribut 1177
Category-Attribut 1178
Codegenerierung steuern 1205
ContainerControl 1180 Control, Ableiten von 1170 CreateParams-Eigenschaft 1195 DefaultValue-Attribut 1178 Delegieren von Funktionen 1181 Description-Attribut 1177 Designer 1184 Designerreglementierungen 1207 Docken mit Dock 1185 DrawString-Methode 1194 Eigenschaften als Code serialisieren 1205 Eigenschaften, werteerbende 1205 Eigenschaftenfenster steuern 1177 Ereignisse delegieren 1188 erstellen von Grund auf 1192 FocusColoredTextBox, Beispiel 1171 Größe festschreiben 1207 Größenbeeinflussung durch Eigenschaften 1200 Grundfenster erstellen 1195 Grundlagen 1170 Initialisieren 1186 Konstituierende 1180 Methoden delegieren 1188 Neuziehen auslösen 1200 ohne Vorlage 1192 Projektmappe für eigene Assembly 1181 ShouldSerializeXXX-Methode 1205 Standardwerte für Eigenschaften 1205 Stil 1195 Toolbox, einfügen in 1173 unveränderliche Größe 1207 Windows-Stil 1195 zeichnen, nativ 1197 Z-Reihenfolge 1185 Beschränkungen 674

Betriebssystemaufrufe 252
Bezierkurve Kubische 1304 Quadratische 1304 Bildlaufleisten 1235 Binäre Suche (Arrays) 627 BinaryFormatter (Serialisierung) 701 BinarySearch 524 BinarySearch-Methode (Arrays) 627 BindingSource *siehe* Bindungsquellen BindingSource-Klasse 1061 Bindungsquellen 1060 Block-Gültigkeitsbereiche 42, 337 Bogensegment 1304 Boolean-Datentyp 254 numerische Äquivalente 254 String, wandeln in 255 Vergleiche 39 Boolesche Ausdrücke 29 Borderline-Eigenschaft 1195 Bottom 1270 Boxing Details zum 545 Schnittstellen, Besonderheiten 548 Unboxing 545 Wertetypen 542 Breite Linienzüge zeichnen 1161 Brush-Objekt 1144 Bubbling Events 1215 Bucket (Hashtable) 661 Build 73 Business Logic 1020 ByRef 527 Byte-Datentyp 214 ByVal 527

C

CallBack-Funktion 1106 CanExecuteRoutedEventArgs 1227 Canvas 1270 Bottom 1270 Left 1270 Right 1270 Top 1270

Captures-Auflistung (Reguläre Ausdrücke) 791, 798
 Case-Anweisung 41
 Casting 536
 Catch 345
 CByte 214
 CDbl 219
 CDec 220
 CellTemplate-Eigenschaft (DataGridView) 1071
 CellValidating-Ereignis 1077
 Center 1276
 Char-Datentyp 230
 Werte umwandeln in 231
 CheckBox 1234
 CheckBox-Steuerelement
 ThreeState-Eigenschaft,
 Besonderheiten bei 372
 Child Window 1081
 ChrW-Methode 231
 CIL-Analyse 362
 CInt 216, 537
 ClearValue 1231
 CLI 51
 ClientArea-Eigenschaft 1145
 ClipToBounds 1271
 CLng 217
 Cloning *siehe* Klonen
 Close-Methode
 Formulare 1051
 Closing-Ereignis 1036
 CLR 51
 Execution Engine 52
 Garbage Collector 52
 mscoree.dll 52
 CLS
 Compliance 213
 Code
 Aufgabe aus Kommentar 82
 aus- und einblenden 165
 Ausführung bei Fehlern
 sicherstellen 347
 automatisch durch Designer 1099
 bearbeiten *siehe* Codeeditor
 Bearbeiten, Hilfe beim *siehe* IntelliSense
 Editor reagiert träge 86
 Fehlerarten im Editor 80
 für Kulturen 746
 gleicher für mehrere Ereignisse 79

Gültigkeitsbereiche durch
 Blöcke 42, 337
 inkrementelles Suchen 168
 Klassencode über Dateien
 aufteilen 447
 Lesezeichen 169
 mehrere Suchergebnisse 169
 Meldungen 80
 rechteckig markieren 164
 Refactoring 148
 Region 166
 reguläre Ausdrücke 166
 Rumpf für Ereignisse einfügen 77
 Sprung zu Zeilennummer 168
 statt Formular öffnen 74
 Suchen und Ersetzen 166
 Token für Aufgaben 82
 Umgestalten (Refactoring) 148
 Vererben von Klassencode 450
 Warnungen 80
 Warnungen konfigurieren 80
 Wiederverwandlung 450
 Wiederverwenden 480
 Zugriff auf Settings-Variablen 157
 Code Snippets
 Assembly-Verweise
 berücksichtigen 292
 Bibliothek, hinzufügen zur 293
 Codeausschnittsbibliothek 293
 Datei-Öffnen-Programmlogik,
 als 288
 Eigene erstellen 288
 Literale Ersetzungen 296
 Neue Ausschnitte verwenden
 294
 Objektersetzungen 299
 Parametrieren 295
 Verwenden 151
 XML-Vorlagen 290
 Codeausschnitte *siehe* Code
 Snippets
 Codebereich 163
 Codedateien
 verteilte, für eine Klasse 447
 Codedateien organisieren 75
 Codedateien, durchsuchen 166
 Codeditor
 Navigieren mit Aufgabenliste,
 Fehlerliste 83
 Navigieren mit Ausgabefenster 84
 Codeeditor 133
 Abstrakte Klassen,
 Unterstützung für 500
 Anzeigen von 134
 aufrufen aus Projektmappen-
 Explorer 75
 Auswahlrand 163
 Autokorrektur für intelligentes
 Kompilieren 141
 Automatischer Zeilenumbruch
 163
 automatisches Codeeinrücken
 137
 automatisches Vervollständigen
 siehe IntelliSense,
 Vervollständigungsliste
 Background Compiler 138
 Code aus- und einblenden 165
 Codeausschnitte *siehe* Code
 Snippets
 Codebereich 163
 Dokumentationskommentare,
 benutzerdefiniert 142
 Ereignisbehandlungsroutinen,
 Unterstützung bei 591
 Fehler, Unterstützung beim
 Korrigieren von 439
 Fehlerbehebung bei
 Laufzeitfehlern 139
 Fehlererkennung 138
 Gliederungsansicht 165
 Indikatorrand 163
 Intellisense 428
 IntelliSense *siehe* IntelliSense
 Interfaces, Unterstützung für 495
 Kennen lernen am Beispiel 100
 Klassen, Unterstützung für
 abstrakte 500
 Laufzeitfehler beheben 139
 Lesezeichen 166
 navigieren im Code 164
 Neue Codedateien anlegen 146
 Prozeduren-Beschreibungen,
 benutzerdefiniert 142
 rechteckige Textmarkierung
 164
 Refactoring 148
 Region 166
 Schnittstellen, Unterstützung
 für 495

Codeeditor (*Fortsetzung*)
Smarttags 141
Smarttags, Verwendung von 439
sonstige Funktionen 163
Suchen und Ersetzen 166
Tipps für ermüdfreies
Arbeiten 133
Umgestalten von Code
(Refactoring) 148
Variablenzuweisung mit
Konstanten 320
wird langsam 86
XML-Kommentare,
benutzerdefiniert 142
Codefenster
Lesezeichen 169
CollectionBase-Auflistung
Add-Methode
(Implementierung) 645
Elementzugriff über List 646
IList 646
Item-Eigenschaft
(Implementierung) 645
List 646
Collections *siehe* Auflistungen
Column 1261
ColumnDefinition 1261, 1262
Columns-Eigenschaft
(DataGridView) 1067
Columns-Eigenschaft (ListView)
1054
ColumnsHeaderCollection-
Auflistung (ListView) 1054
ColumnSpan 1264
ColumnType-Eigenschaft
(DataGridView) 1070
COMClass-Attribut 731
CommandBinding 1226, 1246
ApplicationCommands 1226
ComponentCommands 1226
EditingCommands 1226
MediaCommands 1226
Common Language Infrastructure
52
Common Language Runtime 51
Common Type System 53, 55
Comparer-Klasse 630
benutzerdefiniert 632
ComponentCommands 1226
Conceptual Model 982

Concurrency-Checks
LINQ to Entities 1010
LINQ to SQL 968
Constraints 674
Container für Steuerelemente 106
ContainerControl 106
ContextMenu 1248
Continue-Anweisung 334
ControlCollection 1103
ControlDesigner 108
ControlNativeWindow 1107
Convenience-Patterns 572
Convert.ToByte 214
Convert.ToDecimal 220
Convert.ToDouble 219
Convert.ToInt16 215, 216
Convert.ToInt32 216
Convert.ToInt64 217
Convert.ToSByte 215
Convert.ToSingle 219
Convert.ToInt32 217
Convert.ToInt64 218
Convert-Klasse 537
Cooperative Multitasking 1318
Covers
Begleitdateien 103
Covers – kennen lernen der
Entwicklungsumgebung, mit 100
Covers: 101
CreateParams-Eigenschaft 1195
CSByte 215
CShort 215, 216
CSng 219
CTS 53, 55
CType 537
für eigene Klassen/Strukturen 581
CUInt 217
CULng 218
CultureInfo 745
CultureInfo-Klasse 227
Currency (Zeitnähe bei
Bindungen) 1060
CurrencyManager-Klasse 1061
CurrentCellChanged-Ereignis
(DataGridView) 1072
CurrentCell-Eigenschaft
(DataGridView) 1072
CustomFormatProvider 764

D

DataContext
Logging 951
DataContext (LINQ to SQL) 948
DataGridViewCellCollection-
Auflistung 1070
DataGridViewCell-Objekt 1072
DataGridViewColumnCollection-
Auflistung 1067
DataGridView-Steuerelement 1059
Aktuelle Zelle ermitteln 1072
Bearbeitungsmodus, prüfen auf
1077
CellTemplate-Eigenschaft 1071
CellValidating-Ereignis 1077
Columns-Eigenschaft 1067
ColumnType-Eigenschaft 1070
Curorsprungverhalten ändern
nach Eingabetaste 1079
CurrentCellChanged-Ereignis
1072
CurrentCell-Eigenschaft 1072
DataGridViewCellCollection-
Auflistung 1070
DataGridViewCell-Objekt
1072
DataGridViewColumnCollec-
tion-Auflistung 1067
Designer-Unterstützung 1062
Editieren von Spalten
verhindern 1076
Eingaben verarbeiten 1073
Formatieren von Inhalten 1073
Häufige Aufgaben (Designer)
1068
IsInEditMode-Eigenschaft 1077
IsNewRow-Eigenschaft 1076
Neue Zeile, testen auf 1076
Nur-Lesen-Spalten 1076
Parse von Eingaben 1073
ReadOnly-Spalte 1076
Spalten konfigurieren 1067, 1069
Spalten programmtechnisch
ansprechen 1067
Spaltenkopfnamen bestimmen
1070
Spaltentypen bestimmen 1070
Unterscheiden bearbeiten/
neu eingeben 1076

DataGridView-Steuerelement (*Fortsetzung*)
 Validieren von Eingaben 1077
 Verhindern falscher Eingaben 1077
 Vorgehensweisen, grundsätzliche 1060
 Vorlagen 1071
 Zeilensvalidierungen 1077
 Zelle, aktuelle ermitteln 1072
 Zellen formatieren 1073
 Zellencursorverhalten ändern 1079
 Zellenspeicherung 1072
 Zellensvalidierungen 1077
 Zellenvorlagen 1071
 Zellenwechsel mit Eingabetaste 1079
 Zelleselektionsänderung feststellen 1072
 Date-Datentyp 255, 539, formatieren, für Textausgabe 744
 Formatzeichenfolgen 748
 ParseExact-Methode 260
 Parse-Methode 260
 rechnen mit 256
 String, umwandeln in 260
 Dateinamen standardisieren 243
 Dateioperationen (My-Namespace) 843
 Dateioperationen programmieren 828
 Dateitypen von Projekten 74
 Daten bearbeiten, Anwendungsbeispiel 1039
 Binden an Komponenten 1060
 Darstellen in Steuerelementen 1053
 Eingeben 1045
 erfassen, Anwendungsbeispiel 1039
 Listendarstellung 1053
 Strukturieren in Anwendungen 1020
 Datenbindung 1060
 Anwendungsbeispiel 1066
 BindingSource-Klasse 1061
 Blättern in Datenquelle 1061
 CurrencyManager-Klasse 1061
 komplexe 1060

Objekte, an 1064
 PropertyManager-Klasse 1061
 Datenerfassung, mit Formularen 1032
 Datenrückgabe aus Formularen 1036
 Datensätze aktualisieren (LINQ to SQL) 948
 Datenschicht 1020
 Datentypen .NET-Synonyme 263
 Berechnungen durch Prozessor 210
 Boolean *siehe* Boolean-Datentyp
 Byte *siehe* Byte-Datentyp
 Char *siehe* Char-Datentyp
 CLS-Compliance 213
 Date *siehe* Date-Datentyp
 Decimal *siehe* Decimal-Datentyp
 deklarieren und definieren 209
 Double *siehe* Double-Datentyp
 Einführung in 208
 Ermitteln des Typs 733
 generisch, benutzerdefiniert *siehe* Generische Datentypen
 Integer *siehe* Integer-Datentyp
 Long *siehe* Long-Datentyp
 Nullable 369
 null-bar (Wertetypen) 605
 numerische 208
 numerische in String wandeln 224
 numerische, Rundungsfehler 221
 numerische, Überblick über 213
 primitive *siehe* Primitive-Datentypen
 Rundungsfehler 221
 SByte *siehe* SByte-Datentyp
 Short *siehe* Short-Datentyp
 Single *siehe* Single-Datentyp
 String *siehe* String-Datentyp
 TimeSpan 256
 ToString 226
 Typermittlung 733
 Typparameter 672
 UInteger *siehe* UInteger-Datentyp
 ULong *siehe* ULong-Datentyp
 umwandeln 224
 Umwandlungsversuch 229
 UShort *siehe* UShort-Datentyp
 Datentypengrößen 315
 DateTime *siehe* Date-Datentyp
 DateFormatInfo-Klasse 759
 DateTimeStyle 261
 DateTimeStyles.AdjustToUniversal 261
 DateTimeStyles.AllowInnerWhite 261
 DateTimeStyles.AllowLeadingWhite 261
 DateTimeStyles.AllowTrailingWhite 261
 DateTimeStyles.AllowWhiteSpaces 262
 DateTimeStyles.NoCurrentDateDefault 262
 DateTimeStyles.None 262
 Datum berechnen 256
 Darstellungsstile 261
 formatieren 744
 Umwandlungsoptionen 261
 Debuggen CustomDebugDisplays 377
 In den .NET-Quellcode 378
 Werteinhalte eigener Typen darstellen 377
 DebuggerDisplayAttribute 378
 Debug-Klasse 560
 Debug-Konfigurationseinstellungen 162
 Decimal 316
 Decimal-Datentyp 219
 Add 230
 Floor-Funktion 230
 formatieren, für Textausgabe 744
 Formatzeichenfolgen 748
 Negate-Funktion 230
 Remainder-Funktion 230
 Round-Funktion 230
 spezielle Funktionen 230
 Truncate-Funktion 230
 Deklaration, Variablen 25
 Delegaten Asynchron aufrufen 1360
 Beispieldwendung 805
 Kommunikation in Formularen mit 1040
 Predicates 867
 Thread, als 1360
 DeleteOnSubmit (LINQ to SQL) 967
 Dependency Properties 1228
 DependencyObject 196
 Deserialisierung 701

- Designer
aufrufen aus Projektmappen-Explorer 75
automatische Codegenerierung 1099
Benutzersteuerelemente
entwerfen 1184
Codegenerierung 1093
ControlDesigner 108
DataGridView-Unterstützung 1062
für Formulare, Funktionsweise 107
für WPF 194
Funktionen von
 Steuerelementen 107
Größe von Steuerelementen 108
Guidelines 106
Häufige Aufgaben 1068
Kennen lernen am Beispiel 100
Layout-Funktionen für
 Formulare 131
Layout-Modus für Formulare 107
Position von Steuerelementen 108
Raster in Formularen 107
Referenzsteuerelement beim Anordnen 108
Smarttags für Steuerelemente 109
Split View 194
Steuerelemente dynamisch
 Anordnen 110
Steuerelemente positionieren 105
Steuerelemente selektieren, unsichtbare 1084
Steuerelemente, Aufgaben, häufige 109
Steuerelemente, Randabstände 106
Tastenkürzel 133
Designer Host 108
Designer, Windows Presentation Foundation 93
Designer-Code 1093
Designer-Komponenten 1046
Device Independant Pixel 1293
Dialoge *siehe* Modale Formulare
Dialogergebnisse 1033
- DialogResult-Eigenschaft 1033
Dim 209
DIP 1293
Direct Events 1215
DirectoryInfo-Klasse 828
DirectX 179
Dispose 550
Dispose 560
Dispose 1115
Dispose 1124
Dispose-Methode
 Formulare 1051
DllImport-Attribut 731
Dock-Eigenschaft 1185
DockPanel 1256
Dokumentationskommentare,
 benutzerdefiniert 142
Dokumentfenster 69
Dokumentklassen 1081
Doppelpufferung 1156
DotNetCopy 824
 /AutoStart-Option 827
 /Silent-Option 827
 Bedienung 824
 Funktionsweise, Abstrakt 829
 Kopierverhalten einstellen 826
 Optionen 827
Double Buffering 1156
Double-Datentyp 219
 formatieren, für Textausgabe 744
 Formatzeichenfolgen 748
 Parse-Methode 224
 Vergleichen 224
DrawString-Methode 1153
Dual Core-Prozessoren 1312
Duotrigimalsystems 521
Durchschnittsberechnung 898
Dynamische Abfragen (LINQ to SQL) 974
Dynamische Hilfe 85
 Inhalt anpassen 86
Dynamische Objekte 55
- E
Eager-Loading 955, 996
Editieren, Code *siehe* Codeeditor
EditingCommands 1226
Editor *siehe auch* Codeeditor
EDM 980
- Eigenschaften 1215
 Accessor 410, 455
Accessor, unterschiedliche für
 Lesen/Schreiben 436
angehängte (XAML) 198
auslesen 410
Default 413
definieren 409
definieren (Wertezuweisung) 410
der Abhängigkeiten 1228
ermitteln 410
Get-Accessor 410
Leselogik definieren 410
Let-Anweisung 413
Margin 106
MustOverride 486
Padding 106
Parameter überladen 433
Parametern, mit 412
schreiben, Werte 410
Schreiblogik definieren 410
Set-Accessor 410
Set-Anweisung 413
Standardeigenschaften 413
statische 440
überladen 433
Überschatten 510
überschreiben 460
Überschreiben (Polymorphie)
 siehe Polymorphie
Überschreibungzwang 486
Variablen, Vergleich zu
 öffentlichen 414
virtuelle 485
Werte auslesen 410
Werte, zuweisen an 410
Wertezuweisung 410
Eigenschaften eines Projektes 75
Eigenschaftenfenster 77
 Ereignisse 77
 Selektieren von
 Steuerelementen 123
Einblenden von Projektdateien 74
Eingabefehler anzeigen
 Tabellenzeilen, in 1077
 Tabellenzellen, in 1077
Eingaben 1231
Eingaben überprüfen *siehe*
 Validieren
Ellipse 1284, 1293
ElseIf-Anweisung 38

Elternfenster 1081
Empty (EventArgs) 598
End Select-Anweisung 41
Endlosschleife *siehe*
 Endlosschleife
EndUpdate-Methode (ListView)
 1055
Entitätsmodell 980
Entity Data Model
 abfragen 990
 Conceptual Model 982
 CSDL-Datei (Conceptual Model) 982
 Designer 987
 Entitätscontainernamen ändern 988
 Erstellen 984
 Funktionsweise 981
 Konzeptionelles Modell 982
 Mapping 982
 MSL-Datei (Mapping) 982
 Physikalisches Speichermodell 982
 POCO 1004
 Praxisbeispiel 983
 SSDL-Datei (Storage Model) 982
 Storage Model 982
Entity Framework 981
 Ändern, Daten 1004
 Concurrency-Checks 1010
 Datenänderungen übermitteln 1004
 Einfügen, Datensätze 1006
 kaskadiertes Löschen 1008
 Löschen, Datensätze 1008
 SaveChanges 1004
 Schreibkonfliktprüfung 1010
 SQL Profiler verwenden 993
Entity SQL (eSQL) 993
Entwicklerbuch
 IntelliLinks 10
Entwicklungsumgebung
 alle Projektdateien anzeigen 74
 auf einen Blick 67
 Aufgabenliste *siehe*
 Aufgabenliste
 Ausgabefenster *siehe*
 Ausgabefenster
 beste Monitorauflösung 69
 Dokumentfenster 69

Dynamische Hilfe *siehe*
 Dynamische Hilfe
Eigenschaftenfenster *siehe*
 Eigenschaftenfenster
Einführung 62
Einsatz mehrere Monitore 268
Einstellungen sichern 270
Fehlerliste *siehe* Fehlerliste
Fensterlayout zurücksetzen 67
Hilfe 283
Klassenansicht *siehe*
 Klassenansicht
Konfigurationseinstellungen
 162
Nachrichten-Channel 65
Registerkartengruppen 69
Startseite 65
Tastenkombinationen 87
Toolfenster *siehe* Toolfenster
Umgang mit Fenstern 69
Enum
 Praxisbeispiel, Einsatz im 479
 Zahlen-Datentyp, konvertieren aus 615
Enumeratoren 635
 benutzerdefiniert 636
Enums 612
 direkte Wertebestimmung 613
 Dubletten 613
 Elementtyp ermitteln 614
 Flags 615
 Flags, Abfrage von 617
 Kombinationen, Abfrage von 617
 Kombinieren von Werten 615
 konvertieren 614
 String-Datentyp, konvertieren in 615
Equals-Methode 506, 510, 658
Ereignisbehandlungsroutinen
 erstellen 79
Ereignisparameter 596, 598
 EventArgs 598
 EventArgs.Empty 598
 leere 598
 Sender 597
Ereignisse 590, 1104, 1214
 AddHandler 600
 Anwendungen, für *siehe*
 Anwendungsereignisse
 Auslösen 594
Auslösen, durch Überschreiben von Onxxx 595
Behandeln (Handles) 591
Behandlungscode erstellen 78
Benutzerdefinierte, Beispiel 591
Code, gleicher für mehrere Ereignisse 79
dynamische Einbinden 600
Ereignisbehandlungsroutinen 591
Feuern (ugs.) 594
Formulare, von *siehe*
 Formular- und Steuerelementereignisse
Handles 591
Maus 1232
Objekte zum Konsumieren deklarieren 591
Onxxx überschreiben 595
PaintEventArgs-Klasse 1141
Parameter *siehe*
 Ereignisparameter
RaiseEvent 595
RemoveHandler 600
Rumpf für Behandlungscode einfügen 77
Steuerelementen, von *siehe*
 Formular- und Steuerelementereignisse
Stift 1233
Tastatur 1232
Verdrahten 591
verwalten durch Eigenschaftenfenster 77
WithEvents 591
Erl (VB6) 342
ErrorProvider-Komponente 1045
Erweiterungsmethode
 GroupBy 875
 kombinieren von mehreren 876
 OrderBy 872
 Select 868
 vereinfachte Anwendung 878
 Where 867
Erzwungene Typsicherheit 142
EventInfo-Klasse 736
Events *siehe* Ereignisse
 fire (col.) *siehe* Ereignisse, Auslösen
 raise *siehe* Ereignisse, Auslösen

Exception
 OutOfRangeException 215
Exceptions 344
 Codeausführung sicherstellen
 347
 typabhängig behandeln 345
Execution Engine 52
Express Edition
 SQL Server 2008 10
 Visual Basic 2008 2008 5
Extended Application Markup
 Language, *siehe* XAML 187

F

False 254
Family (CTS) 435, 436
FamilyOrAssembly (CTS) 435,
 436
Faustregeln für LINQ-Abfragen
 889
Fehler 342
 Abfangen mit Try/Catch 344
 Behandeln 342
 Codeausführung sicherstellen 347
 kulturspezifisch, bei
 Konvertierungen 747
 typabhängig behandeln 345
Fehler abfangen
 global 854
Fehlerliste 79
 Background Compiler 79
 Meldungstypen 80
 Navigieren im Code, mit 83
 Tipps 83
Fehlertypen im Codeeditor 80
Fehlerüberprüfung im
 Hintergrund 79
Feigenbaum, Lisa 881
Fenster *siehe* Formulare
Feuern, Ereignisse (ugs.) 594
FieldInfo-Klasse 736
FIFO-Prinzip 662
FileInfo-Klasse 828
FileStream 566
Fill 1299
FillRule 1300, 1305
Finalize 550, 556
Finalize-Methode 510
Finally 347

Flächen 1144
Flächen füllen 1144
flaches Klonen 706
Flags-Attribut (Enums) 616
Flags-Enum (Flags-Aufzählungen)
 615
Flimmerfreie Darstellung
 (Formulare, Steuerelemente) 1155
FlowLayoutPanel 111
Fokussierung, Reihenfolge *siehe*
 Aktivierungsreihenfolge
fomatieren
 Zahlen und Datumswerte 744
FontFamily 1273
FontSize 1273
FontWeight 1273
For/Each-Anweisung
 Auflistungen, für
 benutzerdefinierte 635
For/Next-Schleifen
 Fließkommazahlen 224
Format Provider 226
 Ausrichtungen 761
 benutzerdefiniert 764
 CultureInfo 745
 DateTimeFormatInfo-Klasse
 759
 Einführung 744
 Fehler vermeiden,
 kulturgebunden 747
 Formatierung, gezielte 758
 Formatzeichenfolgen 748
ICustomFormatter-Schnittstelle
 774
IFormatProvider-Schnittstelle 774
Länderkürzel 746
 NumberFormatInfo-Klasse 758
formatieren
 Format Provider, mit 758
 IFormattable-Schnittstelle 777
 kombiniert 760
 kulturgebunden 745
Formatieren, Zahlenwerte 477
Formatvorlagen 751
Formatzeichenfolgen 591, 748
 Formatvorlagen 751
 vereinfacht 757
 Zeitformatierung, für 757
Formel Parser 805
Formeln berechnen 800
Formular
 Code statt Designer öffnen 74
Formular- und
 Steuerelementereignisse
 auslösende Instanz 1110
 Click 1116, 1125
 Closed 1115
 Closing 1115
 CreateControl 1114, 1124
 Deactivate 1115
 DoubleClick 1116, 1125
 Enter 1121, 1130
 GotFocus 1122, 1131
 HandleCreated 1114, 1124
 HandleDestroyed 1124
 Invalidated 1120, 1129
 KeyDown 1118, 1127
 KeyPress 1118, 1127
 KeyUp 1118, 1127
 Layout 1120, 1130
 Leave 1122, 1131
 Load 1114
 LocationChanged 1119, 1128
 LostFocus 1122, 1131
 MouseDown 1116, 1125
 MouseEnter 1117, 1125
 MouseHover 1117, 1126
 MouseLeave 1117, 1126
 MouseMove 1117, 1126
 MouseUp 1117, 1125
 MouseWheel 1118, 1126
 Move 1119, 1128
 OnHandleDestroyed 1115
 Paint 1121, 1130
 PaintBackground 1121, 1130
 Resize 1119, 1128
 SetVisibleCore 1114
 SizeChanged 1120, 1128
 WndProc 1109, 1123
Formulare 186, 197, 1018
 Abbrechen (Tastatur) 1051
 AcceptButton 127
 Aktivierungsreihenfolge von
 Steuerelementen 123
 Aktivierungsreihenfolge 1050
Anwendungsframework, Hilfe
 durch 850
automatisches Scrollen 121
AutoScroll-Eigenschaft 121
CancelButton 127
ClientArea-Eigenschaft 1145

- Formulare (*Fortsetzung*)
 Close-Methode 1051
 Closing-Ereignis 1036
 ControlCollection 1103
 Datenerfassung mit 1032
 Designer-Code 1093
 Dispose-Methode 1051
 Eigenschaften in Settings speichern 159
 Eingaben überprüfen *siehe* Validieren, Formulareingaben entsorgen 1051
 Ereignisse *siehe* Formular- und Steuerelementereignisse erstellen 1100
 Fensterausschnitt, sichtbarer 1145
 Fensterinnenbereich 1145
 flimmerfreie Darstellung 1155
 gestalten *siehe* Designer Größe anpassen, programmtechnisch 1159
 Größe von Steuerelementen 108
 Grundverhalten 1156
 Hide-Methode 1051
 Hinzufügen zu Projekten 127
 Inhalte zeichnen in OnPaint 1141
 InitializeComponent 1101
 Instanzieren, ohne 833
 IsMdiContainer-Eigenschaft 1081
 Klassennamen-Anpassung beim Umbenennen 150
 Kommunikation zwischen 1040
 Layout-Funktionen 131
 Layout-Logik aussetzen 1102
 Layout-Modus 107
 MDI-Anwendungen *siehe* MDI-Anwendungen
 MDI-Container, bestimmen als 1081
 modal darstellen 1034
 modale *siehe* Modale Formulare
 Neuzeichnen auslösen 1200
 OnPaint-Methode 1141
 Rasterung im Designer 107
 Ressourcen freigeben 1033
 Resumelayout 1102
 Schließen verhindern 1036
 Schließen, richtiges 1051
 Schnellzugriffstasten 127
 ShowDialog 1034
- Splash-Dialoge 852
 Steuerelemente positionieren *siehe* Designer, Steuerelemente positionieren SuspendLayout 1102, 1106
 Tabulatorreihenfolge von Steuerelementen 123
 Tastenkürzel für Layout von Steuerelementen 133
 unsichtbar machen 1052
 Using 1034
 Vererben 1089
 WMClose-Methode 1052
 WndProc-Methode 1052
 Zeichnen von Inhalten 1141
 zerstören 1100
 Z-Reihenfolge 1185
 Zugriffsmodifizierer 1095
- Formulare: 108
 Framework
 Aufbau 54
 Betriebssystemvoraussetzungen 9
 Quellcode 378
 Framework Class Library 52
 Framework Targeting 8
 Framework Version Targeting 364
 Friend 434, 435, 436
 From-Klausel 881
 FullName-Eigenschaft 735
 FullRowSelect-Eigenschaft (ListView) 1054
 Funktionen 25
 optionale Parameter 429
 Signatur 428
 überladen 427
 Funktionen auswerten 800
 Funktionszeiger 806
- G**
- Garbage Collector 52, 552
 Generationen 553
 GC 552
 GDI 1140
 Funktionsprinzip 177
 GDI+ 1140
 Breite Linienzüge 1161
 Brush-Objekt 1144
 ClientArea-Eigenschaft 1145
 Demo 1146
- Doppelpufferung 1156
 Double Buffering 1156
 DrawString-Methode 1153
 DrawXXX-Methoden 1144
 Fensterausschnitt, sichtbarer 1145
 Fensterinnenbereich 1145
 FillXXX 1144
 Flächen füllen 1144
 GraphicsPath-Objekt 1163
 HatchBrush-Objekt 1144
 Inch 1145
 Koordinaten 1144
 Koordinatengenauigkeit 1145
 LinearGradientBrush-Objekt 1144
 Linienzüge zeichnen 1144
 Linienzüge, benutzerdefiniert 1163
 Millimeter 1145
 PageUnit-Eigenschaft 1145
 PathGradientBrush-Objekt 1144
 Pen-Objekt 1144
 Pixel 1145
 PointF-Struktur 1145
 Point-Struktur 1144
 Polygone 1163
 RectangleF-Struktur 1145
 Rectangle-Struktur 1145
 Seiteneinteilung 1145
 Seitenskalierung 1145
 SizeF-Struktur 1145
 Size-Struktur 1145
 SolidBrush-Objekt 1144
 Text einpassen 1153
 TextureBrush-Objekt 1144
 VisibleClipBounds-Eigenschaft 1145
 Geerbte Formulare 1089
 Gehe zu 168
 Generationen 553
 Generics
 Typrückschluss 869
 Generische Auflistungen 684
 Collection(Of) 685
 Dictionary(Of) 686
 KeyedCollection(Of) *siehe* KeyedCollection(Of)-Auflistung
 LinkedList(Of) \t 689
 List(Of) 686
 Queue(Of) 686

Generische Auflistungen (*Fortsetzung*)
 ReadOnlyCollection(Of) 686
 SortedDictionary(Of) 686
 SortedList(Of) 686
 Stack(Of) 686
Generische Datentypen 668
 Beschränken auf Wertetypen 683
 Beschränkungen 674
 Beschränkungen kombinieren
 683
 Constraints 674
 JITter, Umsetzung durch 672
 Konstruktorbeschränkungen 683
 Notwendigkeit, Beispiel 669
 Schnittstellenbeschränkungen
 679
 Typparameter 672
Geometry 1302
GeometryGroup 1302
Geschäftslogik 1020
Gestalten von Formularen *siehe*
 Designer
Get-Accessor (Eigenschaften) 410
GetCustomAttributes-Methode 735
GetEvent-Methode 735
GetEvents-Methode 735
GetField-Methode 735
GetFields-Methode 735
GetHashCode-Methode 658
GetHashCode-Methode 510
GetMember-Methode 735
GetMembers-Methode 735
GetPosition 1222
GetProperties-Methode 735
GetProperty-Methode 735
GetType
 Methode, als 733
GetType-Methode 510
Gleichzeitigkeit 1312
Gliederungsansicht 165
Global-Anweisung 358
Grafik-Auflösung 1291
Grafiken
 volatile 1141
Graphics Device Interface *siehe*
 GDI
Graphics-Klasse 1141
GraphicsPath-Objekt 1163
Grid 1260
 Column 1264

ColumnDefinition 1264
ColumnDefinitions 1264
ColumnSpan 1264
Row 1264
RowDefinition 1264
RowDefinitions 1264
RowSpan 1264
ShowGridLines 1264
GridLines-Eigenschaft (ListView)
 1054
GridSplitter 1266
Group By-Methode 894
GroupBy-Erweiterungsmethode 875
Groups-Auflistung (Reguläre
 Ausdrücke) 798
Gruppen (Reguläre Ausdrücke) 788
Gruppieren in LINQ-Abfragen 894
Gruppieren und aggregieren 899
Gültigkeitsbereiche 335
 Block-Gültigkeitsbereiche 42,337
 Formular-Ebene 336
 Klassen-Ebene 336
 lokale Variable 42,337
 Member-Variablen 336
 Modul-Ebene 336
 öffentliche Variablen 335
 public 335
Guidelines 106

H

Haltepunkte setzen 163
Handles-Anweisung 591
Hashtable-Auflistung 648
 Aufzählungen in 661
 Bucket 661
 DictionaryEntry 661
 enumerieren 661
 For/Each für 661
 Funktionsweise 655
 Load-Faktor 655
 Schlüssel-Klassen,
 benutzerdefiniert 658
 Verarbeitungsgeschwindigkeit
 651
 Zugriffszeiten 654
HatchBrush-Objekt 1144
Height 1278'
Hide-Methode 1051
HideSelection-Eigenschaft
 (ListView) 1054
Hilfe 283
Hintergrunderkennung von
 Fehlern 138
Hintergrundkompilierung 79
Hit-Testing 1304
HitTestResultBehavior 1305
HorizontalAlignment 1259, 1267,
 1279
http
 //www.activedevelop.de 10
HyperThreading 1312
Hyper-V 282

I

IComparable-Schnittstelle 630
IComparer-Schnittstelle 632
ICustomFormatter-Schnittstelle 774
IDE
 Einführung 62
 Navigieren in 72
IDisposable 565
IEnumerable-Schnittstelle 636
If/Then/Else-Anweisung 38
If-Operator 368
IFormatProvider-Schnittstelle 774
IFormattable-Schnittstelle 777
IList-Schnittstelle 646
IML 53,423
 Code untersuchen 423
 Codeanalyse 425
 Umsetzung in nativen Code 423
IML-Analyse 362
Impedance Mismatch 913
Implementierungsvorschriften
 siehe Schnittstellen
Implements 491
Implizite Konvertierung 491
Imports 227,354
Imports-Anweisung 50
Inch 1145
Indikatorrand 163
Infinity 228
Infodialog (beim Starten einer
 Anwendung) *siehe* Splash-
 Dialoge
Informationsbeschaffung 283
Inheritance 450

Inherits 478
 INI-Dateien, Ersatz für 846
 InitializeComponent 1101
 inkrementelles Suchen 168
 InsertOnSubmit (LINQ to SQL) 963
 Instanziieren
 Wertetypen 528
 Instanziieren von Klassen,
 Erklärung 394
 Instanziieren von Objekten 420
 Instanzierung
 Objektspeicher 396, 463
 Speicher für Objekte 396, 463
 Instanziieren von Klassen 31
 Instr-Funktion 238
 InstrRev-Funktion 238
 Int16 *siehe* Short-Datentyp
 Int32 *siehe* Integer-Datentyp
 Int64 *siehe* Long-Datentyp
 Integer 315, 537
 Integer-Datentyp 216
 Enum, konvertieren in 615
 formatieren, für Textausgabe 744
 Formatzeichenfolgen 748
 Parse-Methode 224
 IntelliSense 428
 Abstrakte Klassen,
 Unterstützung für 500
 automatisches Codeeinrücken
 137
 automatisches Vervollständigen
 135
 Auto-Vervollständigungsliste 135
 Codeeditor, Fehlererkennung 138
 Codeeinrückungen 137
 Dokumentationskommentare,
 benutzerdefiniert 142
 Filtern beim Tippen 96
 Interfaces, Unterstützung für
 492, 495
 Klassen, Unterstützung für
 abstrakte 500
 Kurzbeschreibungen 135
 Methoden, Signaturen von 428
 Parameterinfo 136
 Parameterinfo, mehrzeilige
 Befehlszeilen 137
 Schnittstellen, Unterstützung
 für 492, 495
 Signaturanzeige von Methoden
 428

Überschreiben von Methoden,
 Hilfe bei 463
 Verbesserungen 95
 Vervollsätnidungsliste 135
 Vervollständigen von
 Schlüsselworten 137
 Vervollständigungsliste filtern 136
 Interfaces *siehe* Schnittstellen
 Interlocked-Klasse (Threading) 1329
 Intermediate Language *siehe*
 IML
 Internationalisieren von
 Anwendungen 763
 Interop Forms Toolkit 311
 Invalidate-Methode
 (Steuerelemente) 1200
 InvalidateQuerySuggested 1227
 Invariant Culture 227
 IQueryable 975
 IsEditMode-Eigenschaft
 (DataGridView) 1077
 IsMdiContainer-Eigenschaft 1081
 IsMouseOver 1232
 IsNewRow-Eigenschaft
 (DataGridView) 1076
 IsNot-Operator 506
 IsNot-Operator, Praktische Tipps
 509
 Is-Operator 506
 Is-Operator, Praktische Tipps 509
 Items-Auflistung (ListBox) 504
 Items-Auflistung (ListView) 1055
 ItemSelectionChanged-Ereignis
 (ListView) 1057

J

JIT-Compiler 189
 JITter 51, 53, 423
 Generische Datentypen,
 Umsetzung von 672
 Joker 780
 Just in time-Kompilierung 51
 Justify 1276

K

Kaskadiertes Löschen (LINQ to
 Entities) 1008
 Key (Auflistungen) *siehe* Schlüssel
 (Auflistungen)

KeyedCollection(Of)-Auflistung
 Serialisierungsprobleme 721
 Workaround bei
 Serialisierungsproblemen 725
 Kindfenster 1081
 Klassen
 .cctor 425
 .ctor 425, 455
 Abstrakt, als solche definieren 485
 abstrakte 485
 Analysieren 734
 anonyme 374
 Basisklasse erreichen 433
 BindingSource 1061
 CLS-Compliance 213
 Code über mehrere
 Codedateien aufteilen 447
 Convert 537
 CurrencyManager 1061
 DirectCast 541
 Dispose 550, 560
 Ereignisse *siehe* Ereignisse
 Ereignisse verknüpfen 591
 Ereignisse, Instanziieren zum
 Konsumieren von 591
 Ereignisse, Parameter für 598
 Ermitteln des Typs 733
 EventArgs 598
 explizite Konvertierung,
 implementieren 581
 Finalize 550
 Formel Parser 805
 generisch, benutzerdefiniert
 siehe Generische Datentypen
 geteilte (Partial) 447
 Hierarchie unterbrechen 512
 IML, Umsetzung 455
 Implementierungsvorschriften
 siehe Schnittstellen
 Implements 491
 implizite Konvertierung,
 implementieren 581
 Instanzen, prüfen auf
 Gleichheit 506
 Instanziieren 394, 420
 instanziieren 31
 Interfaces *siehe* Schnittstellen
 Kommunikation zwischen
 siehe Ereignisse
 Konstruktor 420
 Konstruktoren überladen 427

- Klassen (*Fortsetzung*)
Konstruktoren, parametrisierte 457
Konstruktorzwang 422
Late-Binding 55
Managed Heap 396, 463
Me 433, 483
Member über Schnittstellen erreichen 647
Methoden überschreiben 459
Modul 440, 516
MustInherit 485
MyBase 433, 483
MyClass 483
New 433
Object 503
Operatoren, benutzerdefinierte *siehe* Operatorenprozeduren
Overridable 460
Overrides 459
partielle (Designercode) 1095
Polymorphie 450 *siehe* Polymorphie
Prinzip als Analogie 394
private Member, indirekt zugreifen auf 647
PropertyManager 1061
Random 543
Schnittstellen *siehe* Schnittstellen
Schnittstellen einbinden, mehrere 502
Schnittstellen implementieren 491
serialisieren 698
Singleton 516
Speicher für Objekte 396, 463
Speichern von Inhalten 698
Standardkonstruktor 422, 456
Sub New 420, 427
Type 733
Typermittlung 733
Vererbung 450
Wiederverwendung durch Vererbung 450
Wrapper (Betriebssystemaufrufe) 252
Klassenansicht 87
Klassendesigner 75
Klonen 706
Kombinierte Formatierungen 760
Kompilieren
 Just in time 51
 Komplizierte Abfragen 1003
Komplexe Ausdrücke 28
Komponenten 1046
 BackgroundWorker 1357
 ErrorProvider 1045
 Komponentenfach 1046
Konfigurationsdateien 846
Konsolenanwendungen 16, 416
Konstanten 320
Konstituierende Steuerelemente 1181
Konstruktoren
 .cctor 425
 .ctor 425
 aus Konstruktoren aufrufen 432
 automatisch durch VB 456
Erzwingen in Generischen Datentypen 683
Notwendigkeit 422
parametrisierte 457
Standardkonstruktor 456
Strukturen, bei 528
Syntaxfehler 433
überladen 427
überladene aufrufen 432
Überschreiben (Polymorphie) *siehe* Polymorphie
Konvertieren
 Arrays zu ArrayList 642
 Boolean in String 255
 Byte-Datentyp, in 214
 Date in String 260
 Datum in Zeichenkette 260
 Decimal-Datentyp, in 220
 DirectCast 541
 Double-Datentyp, in 219
 Enums 614
 Erweitern von Typen 582
 Fehler bei numerischen Typen 221
 Fehler vermeiden, kulturgebunden 747
 Formatvorlagen 751
 Integer-Datentyp, in 214
 kulturgebundene Fehler vermeiden 747
 Long-Datentyp, in 217
Parse 540
ParseExact 540
SByte-Datentyp, in 215
Short-Datentyp, in 215
Single-Datentyp, in 219
String in Boolean 255
String in Date 260
UInteger-Datentyp, in 217
ULong-Datentyp, in 218
UShort-Datentyp, in 216
Verkleinern von Typen 582
versuchen 229
versuchen (TryParse) 540
Wert in Zeichenkette 224
Zeichenkette in Datum 260
Zeichenkette in Datum (aus Formatvorlagen) 260
Zeichenkette in Wert 224
Konvertierung, von Datentypen 536
Konvertierungen
 explizite, implementieren
 benutzerdefiniert 581
 implizite, implementieren
 benutzerdefiniert 581
Koordinaten 1144, 1292
Skalierung 1145
Koordinatengenauigkeit 1145
Kubische Bezierkurve 1304
Kultur, nicht bestimmt 227
Kulturabhängige Formate 226
Kulturinformationen 745
Kulturkürzel 746
Kurzschlussauswertung 348
Kurzschlussauswertungen 348

L

- Label 1243
Ladenverhalten (Entity Framework) 996
Ladestrategien (LINQ to SQL) 955
Lamda-Ausdrücke
 in LINQ-Abfragen 865
Länderkürzel 746
Language integrated Query *siehe* LINQ
LastChildFill 1257

- Late-Binding 55
 Layer *siehe* Anwendungsschichten
 LayoutTransform 1291, 1296
 Lazy-Loading 955, 996
 Left 1270, 1276
 Left-Funktion 237
 Leistungsmesser 252
 Len-Funktion 237
 Length-Methode (Arrays) 625
 Lesezeichen 166
 Lesezeichen (Programmcode) 169
 Let-Anweisung
 (Variablenzuweisung) 413
 LIFO-Prinzip 663
 Like-Operator 40
 Line 1293, 1297
 LinearGradientBrush-Objekt 1144
 LineBreak 1276
 Linie 1297
 Linien 1144
 Linienzüge zeichnen 1144
 LinkedList(Of)-Auflistung 689
 AddAddFirst-Methode 690
 AddAfter-Methode 690
 AddBefore-Methode 690
 AddLast-Methode 690
 FindLast-Methode 690
 Find-Methode 690
 First-Eigenschaft 690
 Last-Eigenschaft 690
 RemoveFirst -Methode 690
 RemoveLast -Methode 690
 Remove-Methode 690
 Links zu Aktualisierungen 10
 LINQ
 Abfragen gezielt ausführen 890
 Abfragen kombinieren 876, 887
 Aggregatfunktionen 898
 All-Methode 955
 Anonyme Typen 869
 Any-Methode 955
 Aufbau einer Abfrage 881
 Auflistungen gruppieren 894
 Auflistungen in Relation setzen
 892, 894, 896, 897
 Auflistungen mit From
 verbinden 892
 Auflistungen verbinden 891
 DataContext 948
 Einführung 860
 Elemente von Abfrage trennen
 890
 Ergebnisse mehrere
 Aggregatfunktionen 899
 Faustregeln für Abfragen 889
 From-Klausel 881
 generelle Funktionsweise 862
 Group By-Methode 894
 GroupBy-
 Erweiterungsmethode 875
 Gruppieren 894
 Join, Explizit 894
 Join, Implizit 892
 Kaskadierte Abfragen 887, 890,
 952
 Kombinieren von
 Gruppierungen und
 Aggregate 899
 Lambda-Ausdrücke 865
 O/RM 860
 OrderBy-Erweiterungsmethode
 872
 parallel LINQ 881
 Prädikate 867
 Relation von Auflistungen 892,
 894, 897
 Relation von gruppierten
 Auflistungen 896
 Select-Erweiterungsmethode 868
 SQL Server-Kommunikation 951
 SQL-Daten aktualisieren 948
 SQL-Klartext 951
 Sub-Selects 952
 to DataSets 940
 to Objects 880
 to SQL 912
 ToArray-Methode 890
 ToDictionary-Methode 891
 ToList-Methode 890
 ToLookup-Methode 891
 Verzögerte Ausführung 885,
 887
 Where-Erweiterungsmethode
 867
 LINQ to Entities 980
 abfragen, EDM 990
 Ändern, Daten 1004
 Conceptual Model 982
 Concurrency-Checks 1010
 Designer, EDM 987
 Eager-Loading 996
 EDM 980
 EDM erstellen 984
 Einfügen, Datensätze 1006
 Entitätsmodell 980
 Entity SQL 993
 Impedance Mismatch 913
 kaskadiertes Löschen 1008
 Kompilierte Abfragen 1003
 Konzeptionelles Modell 982
 Lazy-Loading 996
 Löschen, Datensätze 1008
 Mapping 913
 Objektkontext 990
 Physikalisches Speichermodell
 982
 POCO 1004
 Praxisbeispiel 983
 SaveChanges 1004
 Schreibkonfliktprüfung 1010
 SQL Profiler verwenden 993
 Storage Model 982
 Technologieentscheidungen
 935
 T-SQL-Umsetzung 994
 Verwalterklasse 990
 vs. LINQ to SQL 935
 LINQ to Objects
 Einführung 880
 Skalierbarkeit 880
 LINQ to SQL
 Abfrage vom Kontext trennen
 960
 Abfragebäume 974
 Aktualisierungslogik 948
 Änderungen übertragen 961
 Beispiel, erstes 941
 Concurrency-Checks 968
 DataContext 948
 DataContext-Klasse 946
 Daten ändern 961
 Datenbankplattform 914
 DeleteOnSubmit 967
 dynamische Abfragen 974
 Eager-Loading 955
 Einfügen, Daten 963
 Einfügen, Daten in verknüpften
 Tabellen 964
 Ergebnisliste, unabhängig 960
 Impedance Mismatch 913

- LINQ to SQL (*Fortsetzung*)
 InsertOnSubmit 963
 IQueryable 975
 JOIN-Abfragen 950
 Kaskadierte Abfragen 952
 Ladestrategien 955
 Lazy-Loading 955
 Löschen, Daten 967
 Mapping 913
 O/R-Designer 940
 O/RM 912
 Objektnamen-Pluralisierung 947
 Schreibkonfliktprüfung 968
 SubmitChanges 961
 Technologieentscheidungen 935
 TransactionScope 971
 Transaktionen 971
 Transaktionssteuerung,
 DataContext 972
 T-SQL direkt ausführen 973
 T-SQL-Klartext 951
 Verwalterklasse 946
 vs. DataSets 939
 vs. LINQ to Entities 935
 Zukunft, über 936
- List(Of)-Auflistung
 Aktionen (Actions) 692
 Aussageprüfer (Predicates) 695
 For/Each-Ersatz 692
 Sortierungssteuerung-Ersatz 694
 Suchsteuerung 695
 Vergleicher (Comparisons) 694
- ListBoxItem 509
- ListBox-Steuerelement 503
 ausgewählter Eintrag 509
 Eintrag entfernen 509
 Elemente identifizieren 509
 Items-Auflistung 504
 Remove 509
 SelectedItem 509
 selektiertes Element 509
- Listen 621
- Listendarstellung 1053
- ListViewItem-Objekt
 Selected-Eigenschaft 1058
 Tag-Eigenschaft 1056
- ListViewItem-Objekt (ListView)
 1055
- ListView-Steuerelement 1054
 BeginUpdate-Methode 1055
 Beschleunigen 1055
- Columns-Eigenschaft 1054
ColumnsHeaderCollection-
 Auflistung 1054
Daten für Spalten 1055
Daten hinzufügen 1055
Detaillierte Listendarstellung
 1054
 Einträge selektieren 1058
EndUpdate-Methode 1055
FullRowSelect-Eigenschaft 1054
Gitterlinien anzeigen 1054
GridLines-Eigenschaft 1054
HideSelection-Eigenschaft 1054
Items-Auflistung 1055
ItemSelectionChanged-Ereignis
 1057
Korrelation Objekt/Eintrag 1056
ListViewItem-Objekt 1055
Mehrere Einträge selektieren 1057
MultiSelect-Eigenschaft 1057
Objekinstanzen zuordnen 1056
SelectedIndexChanged-
 Ereignis 1057
Selektierte Elemente immer
 anzeigen 1054
Selektion feststellen 1057
Spaltenkopfbreite 1054
Spaltenkopfbreiten anpassen,
 automatisch 1056
Spaltenköpfe 1054
SubItems-Auflistung 1055
View-Eigenschaft 1054
Zeile hinzufügen 1055
Zeile selektieren, komplette 1054
- Load-Faktor 639, 655
- Local Type Inference 366
- LogicalTreeHelper
 FindLogicalNode 196
 GetChildren 196
 GetParent 196
- Lokale Typrückschlüsse 366
- Lokalisieren *siehe auch*
 Ressourcen
- Lokalisieren von Anwendungen 763
- Long 315, 537
- Long-Datentyp 217
 Enum, konvertieren in 615
 formatieren, für Textausgabe 744
 Formatzeichenfolgen 748
 Parse-Methode 224
- Look and Feel von Windows XP 851
- LSet-Funktion 238
- LTrim-Funktion 241
- ## M
- Main-Methode 23
- Managed Heap 396, 463
- Management Studio (SQL Server)
 915
- ManualResetEvent-Klasse
 (Threading) 1331
- Mapping (Entity Data Model) 982
- Margin 1279
- Margin-Eigenschaft 106
- MarshalAs-Attribute 731
- Matches-Auflistung (Reguläre
 Ausdrücke) 797
- Match-Klasse (Reguläre
 Ausdrücke) 796
- Maus-Ereignisse 1232
- MaxHeight 1278
- Maximalwerte 227
- Maximum ermitteln 898
- .MaxValue 227
- MaxWidth 1278
- MDI *siehe* Multi Document
 Interface
- MDI-Anwendungen 1081
 Aufteilung von
 Funktionalitäten 1082
 Child Window 1081
 Dokumentklassen 1081
 Elternfenster 1081
 Formular als Container 1081
 Funktionalitäten aufteilen in
 Parent/Child 1082
 Kindfenster 1081
 Menüs verschmelzen 1083
 Parent Window 1081
- Me 433, 483
- MediaCommands 1226
- Meldungsfelder *siehe* Modale
 Dialoge
- MemberInfo-Klasse 736
- Member-Variablen
 Speicherfolge in Strukturen
 532
- MemberwiseClone-Methode 510
- Menü 1244

- Menüs *siehe* ToolStrip-Steuerelement
 MenuItem 1244
 ToolStrip-Steuerelement
 MDI-Anwendungen, konfigurieren 1082
 Schnellzugriffstasten 1048
 ShortcutKeyDisplayString-Eigenschaft 1048
 ShortcutKeyes-Eigenschaft 1048
 Tastenkürzel 1048
 Message Queue *siehe* Nachrichtenwarteschlange
 Message-Klasse 1106
 MethodBase-Klasse 736
 Methode
 mit Rückgabewerten 25
 Methoden
 Basisklassenmethoden
 überschreiben 459
 Ersetzen 460
 MustOverride 486
 optionale Parameter 429
 Signatur 428
 überladen 427
 Überschatten 510
 Überschreiben (Polymorphie) *siehe* Polymorphie
 überschreiben in Basisklasse 459
 Überschreibungzwang 486
 virtuelle 485
 Zeiger auf 806
 MethodImpl-Attribut 1328
 Microsoft Developer Network 5
 Microsoft Virtual PC 278
 Microsoft Virtual Server 281
 Mid-Funktion 237
 Millimeter 1145
 MinHeight 1278
 Minimalwerte 227
 Minimum ermitteln 898
 MinValue 227
 MinWidth 1278
 Modale Formulare 1032,
 Abbrechen (Tastatur) 1051
 Aktivierungsreihenfolge 1050
 Anwendungsbeispiel 1038
 Bestätigen (Tastatur) 1051
 Closing-Ereignis 1036
 Darstellen 1034
 Daten erfassen/bearbeiten 1039
 Datenaustausch in 1033
 Dateneingabe 1045
 Datenrückgabe 1036
 Dialogergebnis 1033
 DialogResult-Eigenschaft 1033
 Instanzieren, ohne 833
 Kommunikation zwischen 1040
 MessageBox 1032
 Muster für 1033
 Nachrichtenwarteschlange 1032
 Ressourcen freigeben 1033
 Schließen verhindern 1036
 ShowDialog 1034
 Umgang mit 1032
 Using 1034
 Validierung *siehe* Validieren,
 Formulareingaben
 Modifizierer *siehe*
 Zugriffsmodifizierer
 Narrowing 582
 Widening 582
 WithEvents 591
 Module 440, 516
 Monitor-Klasse (Threading) 1322
 MouseEventArgs 1222
 MRU-Liste der Projekte 67
 mscoree.dll 52
 MSDN-Abonnement 5
 MSIL-Code 189
 Müllabfuhr 552
 Multi Core-Systeme 1312
 Multi Document Interface 1081
 Multiprozessor-Systeme 1312
 MultiSelect-Eigenschaft
 (ListView) 1057
 Multitasking
 Cooperative Multitasking 1318
 preemptive 1318, 1320
 MustInherit 485
 MustOverride 486
 Mutex
 Einsatz bei Singleton-Klassen 518
 Mutex-Klasse 1326
 Array 1328
 Mutual Exclusion *siehe* Mutex
 My.Application 832, 835
 My.Computer 832
 Dateioperationen 843
 FileSystem 843'
 My.Forms 832
 My.Resources 832
 My.Settings 832
 Anwendungseinstellungen 846
 My.User 832
 My.WebServices 832
 MyBase 433, 483
 MyClass 483
 My-Namespace
 Anwendungsbeispiel 824
 Anwendungseinstellungen 846
 Dateioperationen 843
 Formulare, ohne Instanzen 833
 Hintergrund 832
 Ressourcen 837
 Übersetzen in andere Sprachen 840
 MyProject-Klasse 835
- ## N
- Nachrichten-Channel 65
 einstellen 66
 Nachrichtenfelder *siehe* Modale Dialoge
 Nachrichtenwarteschlange
 Ereignisse auslösen 1109
 Modale Formulare 1032
 Schließen von Formularen 1052
 Namespace
 Erklärung 47
 verwenden 48
 Namespaces 353
 Bestimmen für Projekte 356
 Global 358
 Importieren 354
 Importieren, global für Projekt 354
 verschiedene in einer Assembly 357
 Zugriff auf System-Namespace (Global) 358
 Narrowing-Modifizierer 582
 NativeWindow 1106
 Navigieren 164
 Navigieren, im Code mit
 Ausgabefenster 84
 Navigieren, im Code mit
 Fehlerliste, Aufgabenliste 83

NetworkAvailabilityChanged - Ereignis (Anwendungsframework) 854
Neue Codedateien anlegen 146
Neues Projekt erstellen 16
New 433
New, Sub
Konstruktoren aufrufen, weitere 432
MyBase 433
Überladen 427
New, Sub in Klassen 420
Nullable(Of 605
Nullable-Datentypen 369
Boxing, Besonderheiten 372
null-bare Wertetypen 605
NumberFormatInfo-Klasse 758

O

O/R-Designer 940
Entity Data Model 987
Pluralisierung 947
O/RM
Funktionsweise 860
Object 316, 503
Eigenschaften 510
Equals 506, 510
Equals, Praktische Tipps 509
Finalize 510
GetHashCode 510
GetType 510, 733
MemberwiseClone 510
Methoden 510
ReferenceEquals 510
Sender bei Ereignissen, Träger der Quelle 597
ToString 510
Object Relational Mapper 912
Objekte 30
ableiten 31
abstrakte 31
Analysieren 734
Binden an Steuerelemente 1060
BindingSource-Klasse 1061
CurrencyManager 1061
DeepCopy 710
Deklarieren mit Ereignissen 591
dynamische 55

entsorgen 550
Ermitteln des Typs 733
finalisieren 550
freigeben, automatisch 330
Gleichheit, prüfen auf 506
Instanzieren 420
instanziieren 31
Klonen 706
Konstruktoren überladen 427
Late-Binding 55
Managed Heap 396, 463
Methoden von Object 503
Polymorphie *siehe* Polymorphie
PropertyManager 1061
serialisieren 698
Speichern von Inhalten 698
Speicherung 396, 463
Type 733
Typermittlung 733
Typsicherheit 55
vergleichen 630
XML, speichern als 715
Zeiger 396, 463
Objektkontext
LINQ to Entities 990
Obsolete-Attribut 729
On Error GoTo, Ersatz für 343
OnClick-Methode 1116, 1125
OnClosed-Methode 1115
OnClosing-Methode 1036, 1115
OnCreateControl-Methode 1114, 1124
OnDeactivate-Methode 1115
OnDoubleClick-Methode 1116, 1125
OnEnter-Methode 1121, 1130
OnGotFocus-Methode 1122, 1131
OnHandleCreated-Methode 1114, 1124
OnHandleDestroyed-Methode 1124
OnInvalidated-Methode 1120, 1129
OnKeyDown-Methode 1118, 1127
OnKeyPress-Methode 1118, 1127
OnKeyUp-Methode 1118, 1127
OnLayout-Methode 1120, 1130
OnLeave-Methode 1122, 1131
OnLoad-Methode 1114
OnLocationChanged-Methode 1119, 1128
OnLostFocus-Methode 1122, 1131
OnMouseDown-Methode 1116, 1125
OnMouseEnter-Methode 1117, 1125
OnMouseHover-Methode 1117, 1126
OnMouseLeave-Methode 1117, 1126
OnMouseMove-Methode 1117, 1126
OnMouseUp-Methode 1117, 1125
OnMouseWheel-Methode 1118, 1126
OnMove-Methode 1119, 1128
OnPaintBackground-Methode 1121, 1130
OnPaint-Methode 1121, 1130
OnResize-Methode 1119, 1128
OnSizeChanged-Methode 1120, 1128
Onxxx-Methoden *siehe* Ereignisse, Onxxx überschreiben
OOP
wichtiger Tipp zu 830
Operatoren
AndAlso 348
benutzerdefinierte *siehe* Operatorenprozeduren
Bitverschiebungen 341
Boolesche 39
GetType 733
Kurzschreibweise 340
OrElse 348
Type Of 733
Überladen von
Operatorenprozeduren 579
Vergleiche, für 39
Operatorenprozeduren 572
Einführung 572
Klassen vorbereiten für 574
Problembehandlung 584
Rechenoperatoren, implementieren 578
Signaturenmehrdeutigkeiten 586
Strukturen vorbereiten für 574
Typkonvertierungen, implementieren 581
Überladen 579
Übersicht möglicher Operatoren 586

Operatorenprozeduren (*Fortsetzung*)
 Vergleichsoperatoren,
 implementieren 580
 Wahr/Falsch-Auswerter,
 implementieren 583
 Option
 Explicit/Strict/Infer/Compare
 generelles Verhalten 58
 Option Strict 142, 430
 optionale Parameter 429
 OrderBy-Erweiterungsmethode 872
 OrElse 348
 Orientation 1255, 1278
 OutOfRangeException 215, 216
 Overridable-Anweisung 460
 Overrides-Anweisung 459

P

Padding 1280
 Padding-Eigenschaft 106
 Page 197
 PageUnit-Eigenschaft 1145
 PaintEventArgs-Klasse 1141
 Panel 1254
 DockPanel 1254
 StackPanel 1254
 TabPanel 1254
 WrapPanel 1254
 Parallel LINQ 881
 Parameter
 Befehlszeile, auslesen 835
 ByRef 527
 ByVal 527
 Eigenschaften, für 412
 Ereignisse, für *siehe auch*
 Ereignisparameter
 ermitteln, in Formularen 1033
 EventArgs 598
 Konstruktoren, in 457
 Operatorenprozeduren
 überladen 579
 PaintEventArgs-Klasse 1141
 überladen (Eigenschaften) 433
 Parent Window 1081
 Parse 524, 539, 540
 ParseExact 539, 540
 ParseExact-Methode (Date-Datentyp) 260
 Parse-Methode (Date-Datentyp) 260

Parse-Methode (numerische Werte) 224
 Partial 447
 PasswordBox 1239
 Path 1293, 1302
 PathGeometry 1303
 PathGradientBrush-Objekt 1144
 Pen-Objekt 1144
 Performace-Counter 252
 PictureBox-Steuerelement, Ausschnitte einstellen in 121
 Pinsel 1144
 Pixel 1145
 Pluralisierung 947
 POCO 1004
 PointF-Struktur 1145
 Point-Struktur 1144
 Polygon 1293
 Polygone 1163
 Polyline 1293, 1298
 Polylinie 1298
 Polymorphie 450, 466
 am Beispiel 473
 Beispiel als Analogie 467
 Beispiel, Praxisbeispiel 470
 Beispiel, ToString 503
 Einführung 466
 ListBox 503
 Me 483
 MyBase 483
 MyClass 483
 ToString 503
 Wiederverwenden von Code 480
 PowerShell 919
 Prädikate 867
 Preemptive Multitasking 1318
 PreProcessMessage 1109
 Preserve-Anweisung 623
 Primitive Datentypen 208
 Boolean 317
 Byte 220, 316
 Char 317
 Currency (VB6) 316
 Date 317
 Decimal 221, 317
 Decimal, Unterschied zu VB6 316
 Deklaration 318
 Double 221, 317
 Integer 221, 317
 Integer, Unterschied zu VB6 315
 Konstanten 320
 Long 221, 317
 Long, Unterschied zu VB6 315
 Object 317
 Object, Unterschied zu VB6 316
 Operatoren 340
 SByte 220, 316
 Short 220, 316
 Short, Unterschied zu VB6 315
 Single 221, 317
 String 317
 Typliterale 320
 Typsicherheit 320
 UInteger 221, 317
 ULong 221, 317
 UShort 220, 317
 Variablendeclaration in For-Schleifen 333
 Variablenverwendung in For/Each-Schleifen 334
 Variant (VB6) 316
 Werteneicht darstellbare Werte 327
 Zahlenüberläufe 327
 Private 434, 435
 ProcessCmdKey-Methode 1122, 1131
 ProcessDialogChar-Methode 1123, 1132
 ProcessDialogKey-Methode 1123, 1132
 ProcessKeyPreview-Methode 1123
 Programmcode bearbeiten *siehe* Codeeditor
 Programme
 internationalisieren 763
 lokalisieren 763
 Programmfehler
 kulturspezifisch 747
 Programmieren
 Regular Expressions 796
 Programmstrukturen 32
 Projekt
 Assembly-Verweis 48
 neues erstellen 16
 starten 18
 Projekt/Projektmappe öffnen 67
 Projekte 73
 alle Dateien anzeigen 74
 als Startprojekt 73
 Assemblyname, resultierender 356

Projekte (*Fortsetzung*)
Dateianzeige aktualisieren 75
Dateitypen 74
Eigenschaften anzeigen 75
Formular hinzufügen 127
Konfigurationseinstellungen 162
Liste der letzten 67
Migrieren aus VB2005 63
MRU-Liste 67
MyProject 835
Namespace zuweisen 356
Neu anlegen 103
versteckte Dateien anzeigen 74
verwalten 73
Projektmappe 73
ausführen 73
Build 73
Eigenschaften abrufen 73
Eigenschaften festlegen 73
erstellen (kompilieren) 73
neu erstellen 73
neu erstellen (kompilieren) 73
speichern 73
umbenennen 73
verwalten 73
zur Quellcodewaltung
hinzufügen 73
Projektmappen
Any-Zieltyp 54
Konfigurationseinstellungen 162
öffnen 67
Projektmappen-Explorer 73
Aktualisieren der Ansicht 75
alle Projektdateien anzeigen 74
Codedateien organisieren 75
Codeeditor für Projektdatei
aufrufen 75
Dateioperationen 76
Designer für Projektdatei
aufrufen 75
Funktionen der Werkzeugeiste 75
Klassendesigner aufrufen 75
Neue Codedateien anlegen 146
Symbole 75
Property Extender 119
PropertyInfo-Klasse 736, 737
PropertyManager-Klasse 1061
Protected 435, 436
Protected Friend 435, 436
Public 434, 435

Pulldown-Menüs *siehe*
MenuStrip-Steuerelement
Pull-Methode 663
Push-Methode 663

Q

Quadratische Bezierkurve 1304
Quantifizierer (Reguläre
Ausdrücke) 786
Quellcode, .NET-Framework 3.5
378
Queue-Auflistung 662

R

RadioButton 1234
RaiseEvent-Anweisung 595
Random-Klasse 543
Rastergrafiken 1291
ReaderWriterLock-Klasse
(Threading) 1330
Recherchieren 283
Rectangle 1284, 1293
RectangleF-Struktur 1145
Rectangle-Struktur 1145
ReDim-Anweisung 621
Refactoring 148
Begriffsdefinition 1022
ReferenceEquals-Methode 510
Referenztypen
Performance-Unterschied zu
Wertetypen 533
Reflection 728
Assembly-Klasse 735
AssemblyQualifiedName-
Eigenschaft 735
Attribute ermitteln 737
Attribute ermitteln,
benutzerdefiniert 741
Attributes-Eigenschaft 735
BaseType-Eigenschaft 735
Eigenschafteninformationen 737
Einführung 732
FullName-Eigenschaft 735
GetCustomAttributes-Methode
735
GetEvent-Methode 735
GetEvents-Methode 735
GetField-Methode 735
GetFields-Methode 735
GetMember-Methode 735
GetMembers-Methode 735
GetProperties-Methode 735
GetProperty-Methode 735
Reflector 362
Refresh-Methode
(Steuerelemente) 1200
Region 166
Designer-Code 1099
Regionen (geographisch) 746
Registerkartengruppen 69
Regular Expressions *siehe*
Reguläre Ausdrücke
Reguläre Ausdrücke 780
austesten 780
Beispiel 800
Captures-Auflistung 791, 798
Einführung 782
Groups-Auflistung 798
Gruppen 788
Gruppensteuerzeichen 795
Joker 780
Matches-Auflistung 797
Match-Klasse 796
Namensbereich 780
Programmieren mit 796
Quantifizierer 786
Sonderzeichen, suchen nach 783
Suche, komplex 784
Suchen und Ersetzen 790
Suchoptionen 794
Suchvorgänge, einfache 782
Wildcard 780
Rekursion *siehe* Rekursion
Relation von Auflistungen 892,
894, 897
Relation von gruppierten
Auflistungen 896
Release-
Konfigurationseinstellungen 162
RemoveHandler 600
Remove-Methode (ListBox) 509
RenderTransform 1296
Replace-Funktion 238
ResizeDirection 1268

Ressourcen
 Abrufen 838
 Anlegen und Verwalten 837
 Internationalisieren 840
 Ressourcen aufteilen (Threading)
 1326
 ResumeLayout 1102
 Reverse-Methode 626
 RichBoxText 1240
 Right 1270, 1276
 Right-Funktion 237
 RotateTransform 1296
 Routed Commands 1224
 RoutedEventArgs 1221
 Row 1261
 RowDefinition 1262
 RowDefinitions 1261
 RowSpan 1264
 RSet-Funktion 238
 RTrim-Funktion 241
 Rundungsfehler 221

S

SByte-Datentyp 214
 Schaltflächen 1233
 Schieberegler 1235
 Schleifen 32
 Continue 37, 334
 Do/Loop 36
 Exit 37
 For/Each 35
 For/Next 32
 vorzeitig verlassen 37
 vorzeitig wiederholen 37
 While/End 36
 Schleifenabbruchbedingungen 37
 Schlüssel
 benutzerdefinierter 658
 Equals-Methode 658
 GetHashCode-Methode 658
 Unveränderlichkeit 661
 Schlüssel (Auflistungen) 648
 Schnellzugriffstasten 125, 1048
 Schnittstellen 488
 Anwendungsbeispiel 489
 Benennungskonventionen 489
 Beschränkungen, auf
 (Generische Datentypen) 679
 Boxing, Besonderheiten 548

explizite Member-Definition 647
 Implementieren 491
 Implementieren in Klassen 491
 Implementieren von
 Schnittstellen 500
 private Member öffentlich
 machen 647
 Schnittstellen, IntelliSense-
 Unterstützung 492
 Schreibkonfliktprüfung
 LINQ to SQL 968, 1010
 ScrollBar 1235
 Scroll-Funktionalität für
 Formulare 121
 ScrollViewer 1237
 Seiteneinteilung (beim Zeichnen)
 1145
 Seitenkalierung (beim Zeichnen)
 1145
 Select-Anweisung 41
 Selected-Eigenschaft
 (ListViewItem-Objekt) 1058
 SelectedIndexChanged-Ereignis
 (ListView) 1057
 SelectedItem-Eigenschaft
 (ListBox) 509
 Select-Erweiterungsmethode 868
 anonyme Typen 869
 Serialisierung 698
 BinaryFormatter 701
 Probleme bei
 KeyedCollection(Of) 721
 SOAPFormatter 701
 Versionen 715
 Versionsunabhängigkeit 720
 Workaround bei
 KeyedCollection-Auflistung
 725
 XML 715
 Zirkelverweise 712
 Serializable-Attribute 731
 Set 330
 Set-Accessor (Eigenschaften) 410
 Set-Anweisung
 (Objektzuweisung) 413
 SetBoundsCore 1129
 SetClientSize 1129
 SetStyle-Methode 1156
 Settings 155, 846
 im Code verwenden 157
 Speicherort 160
 Variablen einrichten 156
 Verknüpfen von Formularen,
 mit 159
 Verknüpfen von
 Steuerelementen, mit 159
 SetVisibleCore 1114
 Shadows 510
 Shape-Klasse 1294
 Short 315
 ShortcutKeyDisplayString-
 Eigenschaft 1048
 ShortcutKeyes-Eigenschaft 1048
 Short-Datentyp 215
 Enum, konvertieren in 615
 Parse-Methode 224
 ShowDialog-Methode 1034
 ShowGridLines 1261
 ShutDown-Ereignis
 (Anwendungsframework) 853
 Signatures 428
 Single-Datentyp 218
 formatieren, für Textausgabe 744
 Formatzeichenfolgen 748
 Parse-Methode 224
 Vergleichen 224
 Singleton-Klassen 516
 SizeF-Struktur 1145
 Size-Struktur 1145
 Sleep 1333
 Slider 1235
 SmartClients
 Anwendungsframework 850
 Anwendungsschichten 1020
 Beenden der Anwendung,
 Benachrichtigung 853
 Daten strukturieren in 1019
 Dokumenttypen 1018
 Einführung 1018
 Ereignisse, globale *siehe*
 Anwendungsereignisse
 Fehlerbehandlung, globale 854
 Haupt-Thread 1331
 Herunterfahren, Modus 852
 Internationalisieren 840
 lokalisieren 837
 MDI-Anwendungen *siehe*
 MDI-Anwendungen
 Mehrfachstart verhindern 851
 Netzwerkzustandsänderung,
 Benachrichtigung 854

- SmartClients (*Fortsetzung*)
Schichten für Anwendungen
siehe Anwendungsschichten
- Splash-Dialoge 852
- Sprachen, andere 837
- SQL Server 2008 Express 916
- Starten der Anwendung,
Benachrichtigung 853
- Tastaturbedienung 1047
- Thread, erster 1331
- übersetzen in andere Sprachen
837
- UI-Thread 1331
- Unterstützung durch VB2005 850
- Weiteres Starten der
Anwendung,
Benachrichtigung 854
- Smarttags
Codeeditor, für 141
Codeeditor, Verwendung im 439
Steuerelemente, für 109
- SoapFormatter (Serialisierung) 701
- Softwarevoraussetzungen 5
- SolidBrush-Objekt 1144
- SolidColorBrush 1293
- Solution-Explorer 73
- SortedList-Auflistung 664
- Sortieren (Arrays) 625
- Sort-Methode (Arrays) 625
- Sourcecode-Analyse 362
- Speicher verwaltung 396, 463
- Splash-Dialoge 852
- SplitContainer 111
- Split-Funktion 242
- Sprachintegrierte Abfragen *siehe*
LINQ
- SQL 914
- SQL generieren (LINQ to SQL) 951
- SQL Server (Microsoft) 914
- SQL Server 2008 9
- SQL Server 2008 Express 915
AdventureWorks 928
Advanced Services 916
Anfügen, Datenbanken 930
Ausliefern, mit eigenen
Anwendungen 915
Authentifizierungsmodus 925
Beispieldatenbanken 928
Deployment 915
Dienstkonten 924
Einschränkungen 915
- Einsetzen in eigenen
Anwendungen 916
- Firewall 922
- Installation 919
- Management Studio 915
- Netzwerkkonfiguration 943
- PowerShell 919
- Reporting Services 916
- Trennen, Datenbanken 935
- Verwaltungstools 922
- Volltextsuche 916
with Tools 919
- Stack-Auflistung 663
Abladen (Pull-Methode) 663
Aufladen (Push-Methode) 663
- StackPanel 1254
- Standardeigenschaften 413
- Standardkonstruktor 422, 456
- Standardprojekteinstellungen 58
- Standardschaltflächen 127
- Start Page 65
- Starten einer VB-Anwendung 18
- Startobjekt 73
- Startprojekt 73
- Startseite 65
Neues Projekt anlegen 65
Projekt öffnen 65
überflüssige Projekteinträge
entfernen 66
- Startup-Ereignis
(Anwendungsframework) 853
- StartUpNextInstance-Ereignis
(Anwendungsframework) 854
- Steuerelemente 1214
Aktivierreihenfolge 123
als Container 106
Anchor-Eigenschaft 111
Anordnen in Tabellen 111
Anordnen in TableLayoutPanel
117
Anordnen in veränderbaren
Bereichen 111
Anordnen mit Umbruchlogik 111
Arrays 1090
Aufgaben, häufige 109
benutzerdefiniert *siehe*
Benutzersteuerelemente
- Binden an Daten 1060
- Caption-Eigenschaft 125
- ControlCollection 1103
- Darstellen von Daten in 1053
- DataGridView *siehe*
DataGridView-
Steuerelement
- Dock-Eigenschaft 111
- dynamisch Anordnen 110
- Eigenschaften in Settings
speichern 159
- Ereignisse *siehe* Formular- und
Steuerelementereignisse
- erweitern vorhandener
Eigenschaften 119
- flimmerfreie Darstellung 1155
- FlowLayoutPanel 111
- Fokusierungsreihenfolge *siehe*
Aktivierungsreihenfolge
- Größe anpassen 108
- Größe anpassen,
programmtechnisch 1159
- Grundverhalten 1156
- Layout-Funktionen 131
- ListBox *siehe* ListBox-
Steuerelement
- ListView *siehe* ListView-
Steuerelement
- Menüs *siehe* ToolStrip-
Steuerelement
- Name-Eigenschaft 125
- Namenskonventionen im Buch
130
- Neuzeichnen auslösen 1200
- OnPaint-Methode 1141
- PictureBox 121
- Positionieren, genaues 108
- Property Extender 119
- proportionales Vergrößern 113
- Pulldown-Menüs *siehe*
MenuStrip-Steuerelement
- Referenzsteuerelement beim
Anordnen 108
- Schnellzugriffstasten 125
- Scrollen in
Formularen/Containern 121
- Seiten docken 111
- Selektieren von unsichtbaren
1084
- selektieren, erweitertes 122
- SetStyle-Methode 1156
- Smarttags 109
- SplitContainer 111
- Tab Order *siehe*
Aktivierungsreihenfolge

Steuerelemente (*Fortsetzung*)
 TableLayoutPanel 110, 111, 113
 Tabulatorreihenfolge 123
 Tastenkürzel für Layout 133
 Text statt Caption 125
 Text-Eigenschaft 125
 Threading in 1331
 thread-sicher 1331
 verankern 111
 Verankern in TableLayoutPanel 118
 Zeichnen von Inhalten 1141
 Z-Reihenfolge 1185
 zur Laufzeit nicht sichtbare 1046
 Steuerelemente positionieren 105
 Stifte 1144
 Stift-Ereignisse 1233
 Storage Model 982
 Stretch 1272
 Strg (Tastatur) 1048
 strikte Typbindung 430
 String\$-Funktion 233
 String.Intern-Methode 236
 Stringbuilder-Klasse 247
 String-Datentyp 231
 Ähnlichkeit, prüfen auf 40
 Boolean, aus 255
 Carriage Return 234
 Date, umwandeln in 260
 deklarieren und definieren 232
 Enum, konvertieren aus 615
 Funktionen, neue in .NET 232
 Geschwindigkeit beim Zusammenbauen 247
 IndexOfAny-Methode 238
 Intern-Methode 236
 Iterieren durch Zeichen 247
 Konstruktor 233
 Länge ermitteln 237
 Längen angeleichen 238
 Length-Methode 237
 Like-Operator 40
 numerische Werte wandeln, in 224
 PadLeft-Methode 238
 PadRight-Methode 238
 Remove-Methode 238
 Replace-Methode 238
 Sonderzeichen 234
 Speicherbedarf 235
 Speicheroptimierung 236

Split-Methode 242
 String.IndexOf-Methode 238
 StringBuilder, Vergleich 247
 Stringpool 236
 SubString-Methode 237
 Suchen und Ersetzen 238
 Teile ermitteln 237
 TrimEnd-Methode 241
 Trimen 241
 Trim-Methode 241
 TrimStart-Methode 241
 umfangreiches Beispiel 243
 Unveränderlichkeit 235
 vbBack 234
 vbCr 234
 vbCrLf 234
 vbLf 234
 vbnewline 234
 vbNullChar 234
 vbNullString 234
 vbTab 234
 Verketten, Performance-Tipps 233
 Wagenrücklauf 234
 Zeilenvorschub 234
 zerlegen in Teile 242
 Strings
 Konvertieren 538
 Parsen 539
 Stroke 1293
 StrokeDashArray 1293
 StrokeThickness 1293
 StructLayout-Attribut 532
 Structure 520
 Structured Query Language *siehe* SQL
 Strukturen 520
 CLS-Compliance 213
 Ereignisse *siehe* Ereignisse
 Ereignisse verknüpfen 591
 Ereignisse, Instanziieren zum Konsumieren von 591
 explizite Konvertierung, implementieren 581
 generisch, benutzerdefiniert *siehe* Generische Datentypen
 implizite Konvertierung, implementieren 581
 Kommunikation zwischen *siehe* Ereignisse
 Konstruktoren 528

Operatoren, benutzerdefinierte *siehe* Operatorenprozeduren
 Stubs erstellen 78
 Sub Main 23
 Sub New 420
 Konstruktoren aufrufen, weitere 432
 MyBase 433
 Überladen 427
 SubItems-Auflistung (ListView) 1055
 SubmitChanges (LINQ to SQL) 961
 Subs und Functions 25
 Sub-Selects 952
 Suchen und Ersetzen 166
 Reguläre Ausdrücke 790
 Suchergebnisse
 durch Lesezeichen markieren 169
 Summenbildung in Abfragen 898
 Suspend 1333
 SuspendLayout 1102
 Symbole
 Projektmappen-Explorer 75
 Synchronization-Attribute
 (Threading) 1328
 SyncLock-Methode 1319
 Syntaxfehler 433
 Syntax-Tooltipps 95
 System.Boolean (Datentyp) 254
 System.Byte (Datentyp) 214
 System.Char (Datentyp) 230
 System.DateTime (Datentyp) 255
 System.Decimal (Datentyp) 219
 System.Double (Datentyp) 219
 System.Globalization 227
 System.Int16 (Datentyp) 215
 System.Int32 (Datentyp) 216
 System.Int64 (Datentyp) 217
 System.SByte (Datentyp) 214
 System.Single (Datentyp) 218
 System.UInt16 (Datentyp) 215
 System.UInt32 (Datentyp) 216
 System.UInt64 (Datentyp) 218

T

Tab Order *siehe* Aktivierungsreihenfolge
 TabIndex 1244

- TableLayoutPanel 110, 111, 113
 Prozentuale und fixe
 Zellengrößen 116
Spalten einstellen 113
Steuerelemente anordnen, in 117
Steuerelemente verankern 118
Zeilen einstellen 113
Zellen verbinden 119
TabPanel 1254
Tabulatorreihenfolge *siehe*
 Aktivierungsreihenfolge
Tag-Eigenschaft (ListViewItem-
Objekt) 1056
Tag-Eigenschaft (Steuerelemente)
 1090
Task-Manager 1313
Tastaturbedienung 1047
Tasturbefehle 87
Tastatur-Ereignisse 1232
Tastenkombinationen 87
Template 1214
Testbild (Demo für GDI+) 1146
Testumgebung für Software 277
TextAlignment 1276, 1279
 Center 1276
 Justify 1276
 Left 1276
 Right 1276
TextBlock 1273
TextBox 1239
Texte (Grafik)
 ausmessen 1153
 einpassen 1153
 zeichnen 1153
Texteingabe 1239
TextureBrush-Objekt 1144
TextWrapping 1273
Then-Anweisung 38
Threading 1312
Thread-Pool 1349
Threads
 Abbrechen 1334
 aus Pool 1349
 Auslastung 1313
 Aussetzen 1333
 AutoResetEvent-Klasse 1331
 BackgroundWorker-
 Komponente 1357
 Beenden 1334
 Dateizugriffe synchronisieren
 1330
Dataaustausch 1338
Delegaten, durch 1360
einfachste Vorgehensweise 1357
Fortschrittsanzeigen realisieren
 1357
Grundlagen 1316
Grundsätzliches 1318
HyperThreading 1312
Interlocked-Klasse 1329
Managen 1333
ManualResetEvent-Klasse 1331
MethodImpl-Attribut 1328
Monitor-Klasse 1322
Multiprozessor-Systeme 1312
Mutex-Klasse 1326
Notwendigkeit für 1314
Praxiseinsatz 1341
Prozessorauslastung 1313
ReaderWriterLock-Klasse 1330
Ressourcen aufteilen zwischen
 1326
Sleep 1333
Starten 1318, 1333
Steuerelemente, in 1331
Synchronisieren 1319
Synchronisieren, mehrere
 gegenseitig 1331
Synchronisierungstechniken 1328
Synchronization-Attribut 1328
SyncLock 1319
UI-Thread 1331
vorhandene verwenden 1349
wechseln zwischen 1357
tiefes Klonen 706
Tiers *siehe* Anwendungsschichten
Timer-Klasse 1195
TimeSpan-Struktur 256
Toolbars 1249
ToolBarTray 1250
Toolfenster
 andocken 70
 Anordnung 71
 arbeiten mit 71
 Aufgabenliste *siehe*
 Aufgabenliste
 Ausgabefenster *siehe*
 Ausgabefenster
 automatisch im Hintergrund 71
 Dynamische Hilfe *siehe*
 Dynamische Hilfe
Eigenschaften *siehe*
 Eigenschaftenfenster
Einführung 69
Fehlerliste *siehe* Fehlerliste
Klassenansicht *siehe*
 Klassenansicht
Layout in der
 Entwicklungsumgebung 71
Projektmappen-Explorer 73
umschalten per Tastatur 72
wichtige 72
zur Laufzeit/Entwurfszeit 71
zusätzliche öffnen 71
Top 1270
ToString 226, 539
 ListBox-Eintrag, Überschreiben
 für 503
 Object, von 510
 Polymorphie 503
ToString-Methode 477
Trace-Listener 560
Transaktionen
 Gültigkeitsbereich 971
 Transaktionssteuerung für
 DataContext 972
Transaktionen (LINQ to SQL) 971
Transformationen 1296
Trim-Funktion 241
True 254
Try/Catch/Finally 344
TryParse 540
TryParse-Anweisung 229
Tunneling Events 1215
Type
 Assembly-Eigenschaft 735
 AssemblyQualifiedName-
 Eigenschaft 735
 Attributes-Eigenschaft 735
 BaseType-Eigenschaft 735
 EventInfo-Klasse 736
 FieldInfo-Klasse 736
 FullName-Eigenschaft 735
 GetCustomAttributes-Methode
 735
 GetEvent-Methode 735
 GetEvents-Methode 735
 GetField-Methode 735
 GetFields-Methode 735
 GetMember-Methode 735
 GetMembers-Methode 735
 GetProperties-Methode 735

Type (*Fortsetzung*)
 GetProperty-Methode 735
 MemberInfo-Klasse 736
 MethodBase-Klasse 736
 PropertyInfo-Klasse 736
 Type Of-Operator 733
 Type-Klasse 733
 Typen 733
 anonyme 374
 casten 536
 Ereignisse, 591
 generisch, benutzerdefiniert
siehe Generische Datentypen
 null-bar 605
 Parameter für 672
 umwandeln 536
 Typkonvertierung
 DirectCast 541
 Typliterale 320
 Typparameter 672
 Typrückschluss
 generische Typparameter 869
 Typsicherheit 55, 320, 430
 Typsicherheit erzwingen 142
 Typumwandlungen 536

U

Überladen 427
 Überschatten von Prozeduren 510
 Überschreiben 459
 Überschreiben von Onxxx-
 Methoden *siehe* Ereignisse,
 Onxxx überschreiben
 UIElement 204
 UInt16 *siehe* UShort-Datentyp
 UInt32 *siehe* UInteger-Datentyp
 UInt64 *siehe* ULong-Datentyp
 UInteger-Datentyp 216
 UI-Thread 1331
 ULong-Datentyp 218
 Umgestalten von Code
(Refactoring) 148
 Umwandeln, von Datentypen 536
 Umwandlungsversuch 229
 Unboxing 545
 Unendlich 228
 UnhandledException-Ereignis
(Anwendungsframework) 854
 Unicode-Zeichen 231

UniformGrid 1269
 Update-Methode
(Steuerelemente) 1200
 User Controls *siehe*
 Benutzersteuerelemente
 UShort-Datentyp 215
 Using-Anweisung 330, 1034

V

Validieren
 Benutzereingaben
(DataGridView) 1076
 Eingaben in Zellen 1077
 Formulareingaben 1043
 Variablen
 Arrays, Unterschiede zu VB6
 339
 definieren 27
 Deklaration 25, 318
 Deklaration in For-Schleifen
 333
 Deklaration und Definition
 325
 Eigenschaften, Vergleich zu
 öffentlichen 414
 Eliminierung von Set 330
 globale,
Initialisierungszeitpunkt 458
 Gültigkeitsbereiche *siehe*
 Gültigkeitsbereiche
 Initialisierungszeitpunkt bei
 globalen 458
 Managed Heap 396, 463
 Member-Variablen
(Gültigkeitsbereich) 336
 Namensgebung 58
 öffentliche, Vergleich zu
 Eigenschaften 414
 Operatoren 340
 Option Strict 430
 Referenztypen 520
 Speicherfolge in Strukturen 532
 strikte Typbindung 430
 Structure 520
 Typliterale 320
 typsicher 430
 Typsicherheit 55, 320
 Typzeichen 318
 Übergabe an Subs 350
 Verweise 396, 463
 Verwendung in For/Each-
 Schleifen 334
 Werteverlust im
 Gültigkeitsbereich 438
 Zeiger 396, 463
 Zuweisung durch Konstante 320
 VB6-Migration
 Aufwandsabschätzung 310
 Grundproblem 306
 grundsätzliche
 Vorgehensweisen 308
 Interop Forms Toolkit 311
 Lauffähigkeit unter Vista,
 Windows 7 307
 Virtualisierung als temp.
 Lösung 308
 Webcast zum Thema 311
 VBFixedArray-Attribut 731
 VBFixedString-Attribut 731
 vektororientierte Grafikelemente
 1291
 Vereinfachter
 Eigenschaftenzugriff 42
 Vererbung 450
 Vergleichen
 Double- oder Single-Datentyp
 224
 Vergleichen von Objekten 630
 Vergleicher (List (Of)) 694
 Vergleichsoperatoren 39
 Verhindern falscher Eingaben
(DataGridView) 1077
 Versionsunabhängigkeit
(Serialisierung) 720
 Versteckte Projektdateien
 anzeigen 74
 Versuchte Typkonvertierung 229
 VerticalAlignment 1259, 1268, 1279
 Vervollständigungsliste *siehe*
 IntelliSense,
 Vervollständigungsliste
 Verwaltung von Projekten 73
 Verweis hinzufügen (Projekt) 48
 Verweisvariablen 396, 463
 Verzögerte Ausführung, LINQ-
 Abfragen 885, 887
 Vielgestaltungkeit *siehe*
 Polymorphie
 Viewbox 1271
 View-Eigenschaft (ListView) 1054

- Virtualisierungssysteme
Hyper-V 282
Microsoft Virtual PC 278
Virtual Server 281
Virtuelle Prozeduren 485
VisibleClipBounds-Eigenschaft 1145
Visual Basic
Anatomie einer Anwendung 20
Anwendungsprojekt starten 18
Main-Methode 23
Standardprojekteinstellungen 58
Visual Basic 2005
Anwendungsframework 850
Visual Basic 2008
Betriebssystemvoraussetzungen 9
Installation 7
Umsteigen von VB6 308
Upgrade-Assistent 308
Visual Basic 6.0
Programme upgraden 308
Visual Basic 6.0-Unterschiede
Arrays 339
Currency 316
Datentypen, weggefallene 316
Datentypengrößen 315
Decimal 316
Eigenschaften, Default 413
Eigenschaften, Standard 413
Fehlerbehandlung 342
Gültigkeitsbereiche von
Variablen *siehe*
Gültigkeitsbereiche
Instanzierung von Objekten 326
Integer 315
Kurzschlussauswertungen
(AndAlso, OrElse) 348
Let 329
Long 315
New 326
Object 316
Objekte, Umgang mit 329
On Error GoTo, Ersatz für 343
Operatoren 340
Parameterübergabe an
Sub/Function 350
Set 329
Short 315
Variablen in For/Each-
Schleifen, verwendbare 334
Variablen in For-Schleifen,
Deklaration 333
- Variablen, Gültigkeitsbereiche
von *siehe*
Gültigkeitsbereiche
Variablenbehandlung 315
Variablen Deklaration und -
definition 325
Variant 316
Zahlenüberläufe 327
Visual Basic Team Blog 881
Visual Studio
Allgemeine
Entwicklungseinstellungen 63
Arbeitsspeicherbedarf 276
der erste Start 14
erster Start 62
Parallelinstallation/Versionen 9
Umgebungseinstellungen von
Vorgänger 63
Umgebungskonfiguration 15
Versionen 5
Visual Studio 2008
Betriebssystemvoraussetzungen 9
Einstellungen
sichern/wiederherstellen 270
Installation 7
Mindestanforderungen 7
Visualisieren von Daten 1053
VisualTreeHelper 1305
Visuelle Hierarchie 197
Visuelle XP-Stile 851
Vorschriften zur
Klassenimplementierung *siehe*
Schnittstellen
- W**
- Warnungen konfigurieren 80
Was ist .NET? 46
Web-Links zu Aktualisierungen 10
WebMethod-Attribut 731
Werkzeugeisten 1249
Werte
Zeichen umwandeln in 231
Wertetypen
Beschränkung, auf (Generische
Datentypen) 683
Konstruktoren, bei 528
null-bar 605
Performance-Unterschied zu
Referenztypen 533
- Where-Erweiterungsmethode 867
Widening-Modifizierer 582
Width 1278
Wildcard 780
Window *siehe* Formulare
WindowClass 1106
Windows Presentaion Foundation
siehe WPF 172
Windows Presentation
Foundation *siehe* WPF
Windows XP-Stile verwenden 851
With/End With 42
WithEvents-Anweisung 591
WMClose-Methode 1052
WndProc-Methode 1052, 1106,
1123
Wolfenstein 179
WPF 172
Aufbau von Bildinformationen
179
Beachtenswertes 173
Beispiel für Funktionsprinzip 181
Bildlaufleiste 202
Business-Anwendungen in
1016
Button 188
Demo, eindrucksvolle 185
DependencyObject 196
Designer 93, 194
DirectX 179
Einführung 172
Fähigkeiten gegenüber GDI 183
InitializeComponent 189
Loaded-Ereignis 191
Logischen Baum 195
Neue Konzepte 173
Rendering 179
Scrollbar 202
Trennung von Design und
Code 185
Vererbung von Eigenschaften 195
Visuelle Hierarchie 197
vs. GDI 174
XAML 187
WPF vs. WinForms -
Entscheidungshilfen 1016
Wrap 1274
WrapPanel 1277
Wrapper-Klassen 252
WrapWithOverflow 1274
Write 560

X

XAML 187, 197
 attached Properties 198
 Eigenschaften des Elternobjektes 198
 Eigenschaften, angehängte 198
 Page 197
 Überblick 197
 Window 197
 XAMLPad 201
 XamlReader 205
 xml
 space 1275
 XML 715
 Abfragen mit LINQ 907
 Einführung 902
 Erstellen mit LINQ 905
 Intellisense Unterstützung 908
 Namespace 909
 XAttribute 906
 XDocument 907
 XElement 906
 XML-Ausdrücke
 XML Literale 904

XML-Kommentare,
 benutzerdefiniert 142

Z

Zahlen
 formatieren 744
 Zeichenketten wandeln, in 224
 Zahlenkonvertierungsfehler 221
 Zahlsysteme 221
 Zahlenwerte
 formatieren 477
 Text, umwandeln in 477
 Zeichen
 Carriage Return 234
 Datentyp für 230
 Sonderzeichen 234
 Unicode 231
 verkettete 231
 Wagenrücklauf 234
 Werte umwandeln in 231
 Zeilenvorschub 234
 Zeichenketten *siehe* String-Datentyp

Zeichenparameter *siehe*
 PaintEventArgs-Klasse
 Zeigervariablen 396, 463
 Zeitdifferenzen 256
 Zeitnähe der Datenbindung 1060
 Zirkelverweise 712
 Z-Reihenfolge 1185
 Zufallszahlen 543
 Zugriffsebenen *siehe*
 Zugriffsmodifizierer
 Zugriffsmodifizierer 434
 Ändern durch Schnittstellen 647
 Assembly (CTS) 434–436
 Family (CTS) 435, 436
 FamilyOrAssembly (CTS) 435, 436
 Friend 434, 435, 436
 Private 434, 435
 Protected 435, 436
 Protected Friend 435, 436
 Public 434, 435