

Klaus Löffelmann

# **Visual Basic 2005 – Das Entwicklerbuch**

**Microsoft®**  
*Press*

Klaus Löffelmann: Visual Basic 2005 – Das Entwicklerbuch  
Microsoft Press Deutschland, Konrad-Zuse-Str. 1, 85716 Unterschleißheim  
Copyright © 2006 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  
08 07 06

ISBN 3-86063-537-9

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

Satz: Silja Brands, Klaus Löffelmann, ActiveDevelop, Lippstadt (<http://ActiveDevelop.de>)  
Fachlektorat: Ruprecht Dröge, Ratingen (<http://BeConstructed.de>)  
Testing: Jürgen Heckhuis, Lippstadt  
Umschlaggestaltung: Hommer Design GmbH, Haar ([www.HommerDesign.com](http://www.HommerDesign.com))  
Layout und Gesamtherstellung: Kösel, Krugzell ([www.KoeselBuch.de](http://www.KoeselBuch.de))

# Inhaltsverzeichnis

|   |            |
|---|------------|
| <b>Am Anfang war ...</b>  | <b>XIX</b> |
| http://activedevelop.de – Ein wenig Werbung in eigener Sache  | XX         |
| Danksagungen  | XX         |
| <b>Teil A – Einführung</b>  | <b>1</b>   |
| <b>1 Einführung</b>   | <b>3</b>   |
| Welche Softwarevoraussetzungen benötigen Sie?   | 3          |
| Die Visual Basic 2005 Express Edition auf der beiliegenden Buch-CD  | 4          |
| Wissenswertes zur Installation von Visual Studio 2005   | 4          |
| Deinstalltion von Beta 2, CTP oder Release-Kandidaten   | 5          |
| Diese Versionen von Visual Studio 2005 gibt es  | 6          |
| Der Umgang mit Web-Links in diesem Buch – <a href="http://links.entwicklerbuch.net">http://links.entwicklerbuch.net</a> | 9          |
| Die Begleitdateien zum Buch   | 10         |
| Nützliches zu Visual Basic 2005 – <a href="http://vb2005.de">http://vb2005.de</a>                                       | 10         |
| <b>Teil B – Die Visual Studio-Entwicklungsumgebung</b>  | <b>11</b>  |
| <b>2 Ein Flug über die Weiten der Visual Studio-IDE</b>   | <b>13</b>  |
| Die Startseite – der erste Ausgangspunkt für Ihre Entwicklungen   | 13         |
| Der Visual Studio-Nachrichten-Channel   | 14         |
| Anpassen der Liste der zuletzt bearbeiteten Projekte  | 15         |
| Die IDE auf einen Blick   | 16         |
| Genereller Umgang mit Fenstern in der Entwicklungsumgebung  | 18         |
| Wechseln zwischen aktiven Dokumentfenstern mit der Tastatur   | 20         |
| Die wichtigsten Toolfenster   | 20         |
| Der Projektmappen-Explorer  | 21         |
| Das Eigenschaftenfenster  | 25         |
| Die Fehlerliste   | 27         |
| Die Aufgabenliste   | 29         |
| Das Ausgabefenster  | 30         |
| Die dynamische Hilfe  | 32         |
| Die Klassenansicht  | 33         |
| Codeeditor und Designer   | 34         |
| Die wichtigsten Tastenkombinationen auf einen Blick   | 34         |
| <b>3 Formular-Designer und Codeeditor enthüllt</b>  | <b>37</b>  |
| Das Fallbeispiel – Der DVD-Hüllen-Generator »Covers«  | 38         |
| Das »Pflichtenheft« von Covers  | 39         |
| Erstellen eines neuen Projektes   | 41         |

|   |     |
|---|-----|
| Gestalten von Formularen mit dem Windows Forms-Designer .....   | 43  |
| Positionieren von Steuerelementen .....   | 43  |
| Wem »gehört« eigentlich der Formular-Designer? .....  | 45  |
| Häufige Arbeiten an Steuerelementen mit Smarttags erledigen .....   | 47  |
| Dynamische Anordnung von Steuerelementen zur Laufzeit .....   | 48  |
| Was ist ein Property Extender? .....  | 56  |
| Automatisches Scrollen von Steuerelementen in Containern .....  | 58  |
| Selektieren von Steuerelementen, die Sie mit der Maus nicht erreichen .....                                     | 60  |
| Festlegen der Tabulatorreihenfolge (Aktivierreihenfolge) von Steuerelementen .....                              | 60  |
| Über die Eigenschaften Name, Text und Caption .....   | 62  |
| Einrichten von Bestätigungs- und Abbrechen-Funktionalitäten für Schaltflächen in Formularen .....               | 64  |
| Hinzufügen neuer Formulare zu einem Projekt .....   | 65  |
| Wie geht's weiter? .....  | 67  |
| Namensgebungskonventionen für Steuerelemente in diesem Buch .....   | 67  |
| Funktionen zum Layouten von Steuerelementen im Designer .....   | 68  |
| Tastaturkürzel für die Platzierung von Steuerelementen .....  | 70  |
| Der Codeeditor .....  | 71  |
| Die Wahl der richtigen Schriftart für ermüdfreies Arbeiten .....  | 71  |
| Viele Wege führen zum Codeeditor .....  | 72  |
| IntelliSense – Ihr stärkstes Zugpferd im Coding-Stall .....   | 73  |
| Automatische Vervollständigung von Struktur-Schlüsselworten und Codeeinrückung..                                | 75  |
| Fehlererkennung im Codeeditor .....   | 76  |
| Smarttags im Editor von Visual Basic .....  | 79  |
| Autokorrektur für intelligentes Kompilieren .....   | 79  |
| Erzwungene Typsicherheit (Option Strict) projektweit einstellen .....   | 80  |
| Erzwungene Typsicherheit für alle folgenden neuen Projekte .....  | 80  |
| XML-Dokumentationskommentare für IntelliSense bei eigenen Objekten und Klassen .....                            | 80  |
| Hinzufügen neuer Codedateien zum Projekt .....  | 84  |
| Code umgestalten (Refactoring) .....  | 86  |
| Die Bibliothek der Codeausschnitte (Code Snippets Library) .....  | 89  |
| Einstellen des Speicherns von Anwendungseinstellungen mit dem Settings-Designer...                              | 93  |
| Über die Konfigurationseinstellungen »Debug« und »Release« sowie die Geschwindigkeiten der Codeausführung ..... | 100 |
| Weitere Funktionen des Codeeditors .....  | 101 |
| Aufbau des Codefensters .....   | 101 |
| Automatischen Zeilenumbruch aktivieren/deaktivieren .....   | 101 |
| Navigieren zu vorherigen Bearbeitungspositionen im Code .....   | 102 |
| Rechteckige Textmarkierung .....  | 103 |
| Gliederungsansicht .....  | 103 |
| Suchen und Ersetzen, Suche in Dateien .....   | 104 |
| Suchen in Dateien .....   | 105 |
| Inkrementelles Suchen .....   | 106 |
| Gehe zu Zeilennummern .....   | 106 |
| Lesezeichen .....   | 107 |

|   |            |
|---|------------|
| <b>4 Tipps &amp; Tricks für das angenehme Entwickeln zuhause und unterwegs . . . . .</b>                          | <b>109</b> |
| Der Einsatz mehrerer Monitore . . . . .   | 109        |
| Zwei Grafikkarten in einem Rechner? . . . . .   | 110        |
| Und die Bildschirmschirmdarstellung auf Notebooks? . . . . .  | 111        |
| Zurücksetzen der Fenstereinstellungen . . . . .   | 112        |
| Sichern, Wiederherstellen oder Zurücksetzen aller Visual Studio-Einstellungen . . . . .                           | 112        |
| Sichern der Visual Studio Einstellungen . . . . .   | 112        |
| Wiederherstellen von Visual Studio-Einstellungen . . . . .  | 114        |
| Zurücksetzen von Visual Studio in den Originalzustand . . . . .   | 115        |
| Wieviel Arbeitsspeicher darf's denn sein? . . . . .   | 116        |
| Testen Ihrer Software unterwegs und zuhause – Microsoft Virtual PC und Microsoft Virtual Server . . . . .         | 117        |
| Microsoft Virtual PC . . . . .  | 118        |
| Virtual Server 2005 . . . . .   | 120        |
| Hilfe zur Selbsthilfe . . . . .   | 122        |
| Erweitern Sie die Codeausschnittsbibliothek um eigene Codeausschnitte . . . . .                                   | 127        |
| Erstellen einer Code Snippets-XML-Vorlage . . . . .   | 129        |
| Hinzufügen einer neuen Snippet-Vorlage zur Snippet-Bibliothek<br>(Codeausschnittsbibliothek) . . . . .            | 131        |
| Verwenden des neuen Codeausschnittes . . . . .  | 133        |
| Parametrisieren von Codeausschnitten . . . . .  | 133        |
| <b>Teil C – Der Umstieg auf Visual Basic 2005 . . . . .</b>   | <b>141</b> |
| <b>5 Der Umstieg von Visual Basic 6.0 . . . . .</b>   | <b>143</b> |
| Unterschiede in der Variablenbehandlung . . . . .   | 144        |
| Veränderungen bei primitiven Integer-Variablen – die Größen von Integer und Long und der neue Typ Short . . . . . | 144        |
| Anekdoten aus der Praxis . . . . .  | 145        |
| Typen, die es nicht mehr gibt . . . . .   | 145        |
| ... und die primitiven Typen, die es jetzt gibt . . . . .   | 146        |
| Deklaration von Variablen und Variablentypzeichen . . . . .   | 148        |
| Typsicherheit und Typliterale zur Typdefinition von Konstanten . . . . .  | 150        |
| Deklaration und Definition von Variablen »in einem Rutsch« . . . . .  | 155        |
| Vorsicht: New und New können zweierlei in VB6 und VB.NET sein! . . . . .  | 155        |
| Überläufe bei Fließkommazahlen und nicht definierte Zahlenwerte . . . . .   | 156        |
| Alles ist ein Objekt oder »let Set be« . . . . .  | 158        |
| Direktdeklaration von Variablen in For-Schleifen . . . . .  | 159        |
| Unterschiede bei verwendbaren Variablentypen für For/Each in VB6 und VB.NET . . . . .                             | 161        |
| Gültigkeitsbereiche von Variablen . . . . .   | 161        |
| Globale bzw. öffentliche (public) Variablen . . . . .   | 161        |
| Variablen mit Gültigkeit auf Modul, Klassen oder Formularebene . . . . .  | 162        |
| Gültigkeitsbereiche von lokalen Variablen . . . . .   | 162        |
| Arrays . . . . .  | 164        |
| Die Operatoren += und -= und ihre Verwandten . . . . .  | 165        |
| Die Bitverschiebeoperatoren << und >> . . . . .   | 166        |
| Fehlerbehandlung . . . . .  | 167        |
| Elegantes Fehlerabfangen mit Try/Catch/Finally . . . . .  | 169        |

|  |            |
|--|------------|
| Kurzschlussauswertungen mit OrElse und AndAlso . . . . .   | 174        |
| Variablen und Argumente auch an Subs in Klammern! . . . . .  | 175        |
| Namespaces und Assemblies . . . . .  | 176        |
| Assemblies . . . . .   | 176        |
| Namespaces . . . . .   | 178        |
| So bestimmen Sie Assemblynamen und Namespace für Ihre eigenen Projekte . . . . .                               | 181        |
| Verschiedene Namespaces in einer Assembly . . . . .  | 182        |
| <b>6 Der Umstieg von Visual Basic.NET 2002 und 2003 . . . . .</b>  | <b>185</b> |
| Neue Sprachelemente in Visual Basic 2005 . . . . .   | 185        |
| Continue in Schleifen . . . . .  | 185        |
| Gezieltes Freigeben von Objekten mit Using . . . . .   | 186        |
| Zugriff auf den Framework-System-Namespace mit Global . . . . .  | 188        |
| Über mehrere Codedateien aufgeteilter Klassencode – Partial Class . . . . .                                    | 190        |
| Übersicht über weitere Neuerungen in Visual Basic 2005 . . . . .   | 190        |
| <b>Teil D – OOP – Objektorientiertes Programmieren . . . . .</b>   | <b>193</b> |
| <b>7 Vorüberlegungen zur objektorientierten Programmierung . . . . .</b>                                       | <b>195</b> |
| Über Assemblies, Namespaces, CLR, CLI, BCL, JITter und andere .NET-Terminologien . . . . .                     | 196        |
| Was ist eine Assembly? . . . . .   | 196        |
| Was ist ein Namespace? . . . . .   | 197        |
| Was versteckt sich hinter CLR (Common Language Runtime) und CLI<br>(Common Language Infrastructure)? . . . . . | 198        |
| Was ist die FCL (Framework Class Library) und was die BCL<br>(die Base Class Library)? . . . . .               | 198        |
| Was ist das CTS (Common Type System)? . . . . .  | 199        |
| Was ist MS-IML (Microsoft-Intermediate Language) und wozu dient der JITter? . . . . .                          | 199        |
| Erzwungene Typsicherheit und Deklarationszwang von Variablen . . . . .   | 201        |
| Namensgebung von Variablen . . . . .   | 204        |
| Und welche Sprache ist die beste? . . . . .  | 206        |
| Prozedurale Programmierung versus OOP . . . . .  | 206        |
| Prozedurale Programmierung ade? . . . . .  | 207        |
| <b>8 Auf zum Klassentreffen! . . . . .</b>   | <b>215</b> |
| Und ab in den Sandkasten . . . . .   | 216        |
| Konsolenanwendung in VB.NET . . . . .  | 216        |
| Das Klassenprinzip am einfachsten Beispiel . . . . .   | 218        |
| Das Klassenprinzip am eigenen Beispiel . . . . .   | 220        |
| Statische und nicht-statische Methoden und Variablen . . . . .   | 221        |
| Nicht verwirren lassen: Static und Shared in VB . . . . .  | 223        |
| Smarttags im Editor von Visual Basic . . . . .   | 225        |
| Kleiner Exkurs – womit startet ein Programm? . . . . .   | 227        |
| Mit Sub New bestimmen, was beim Instanziieren passiert – der Klassenkonstruktor . . . . .                      | 227        |
| Überflüssige Funktionen mit dem Obsolete-Attribut markieren . . . . .  | 230        |
| Überladen von Funktionen und Konstruktoren . . . . .   | 231        |
| Methodenüberladung und optionale Parameter . . . . .   | 233        |
| Gegenseitiges Aufrufen von überladenen Methoden . . . . .  | 234        |

|   |            |
|---|------------|
| Gegenseitiges Aufrufen von überladenen Konstruktoren .....  | 236        |
| Hat jede Klasse einen Konstruktor?.....   | 237        |
| Zusätzliche Werkzeuge für .NET .....  | 238        |
| Statische Konstruktoren und Variablen .....   | 241        |
| Eigenschaften .....   | 246        |
| Zuweisen von Eigenschaften .....  | 247        |
| Ermitteln von Eigenschaften .....   | 247        |
| Nur-Lesen und Nur-Schreiben-Eigenschaften.....  | 249        |
| Eigenschaften mit Parametern.....   | 250        |
| Überladen von Eigenschaften .....   | 251        |
| Statische Eigenschaften .....   | 252        |
| Default-Eigenschaften (Standardeigenschaften) .....   | 252        |
| Öffentliche Variablen oder Eigenschaften – eine Glaubensfrage? .....                                  | 255        |
| Zugriffsmodifizierer von Klassen, Prozeduren, Eigenschaften und Variablen.....                        | 256        |
| Zugriffsmodifizierer bei Klassen .....  | 257        |
| Zugriffsmodifizierer bei Prozeduren (Subs, Functions, Properties).....                                | 257        |
| Zugriffsmodifizierer bei Variablen .....  | 258        |
| Unterschiedliche Zugriffsmodifizierer für Eigenschaften-Accesors.....                                 | 259        |
| <b>9 Klassenvererbung und Polymorphie .....</b>   | <b>261</b> |
| Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance) .....                                  | 261        |
| Initialisierung von Member-Variablen bei Klassen ohne Standardkonstruktoren .....                     | 270        |
| Überschreiben von Methoden und Eigenschaften.....   | 271        |
| Überschreiben vorhandener Methoden und Eigenschaften von Framework-Klassen ..                         | 274        |
| Das Speichern von Objekten im Arbeitsspeicher – und die daraus resultierende Vorsicht mit ihnen ..... | 275        |
| Polymorphie.....  | 279        |
| Zahlen mit ToString formatiert in Zeichenketten umwandeln .....                                       | 287        |
| Polymorphie und der Gebrauch von Me, MyClass und MyBase .....   | 293        |
| Abstrakte Klassen und virtuelle Prozeduren .....  | 294        |
| Eine Klasse mit MustInherit als abstrakt deklarieren.....   | 295        |
| Eine Methode oder Eigenschaft einer abstrakten Klasse mit MustOverride als virtuell deklarieren ..... | 296        |
| Schnittstellen (Interfaces).....  | 297        |
| Unterstützung bei abstrakten Klassen und Schnittstellen durch den Editor.....                         | 304        |
| Schnittstellen, die Schnittstellen implementieren .....   | 309        |
| Einbinden mehrere Schnittstellen in eine Klasse .....   | 310        |
| Die Methoden und Eigenschaften von Object.....  | 311        |
| Polymorphie am Beispiel von ToString und der ListBox .....  | 312        |
| Prüfen auf Gleichheit von Objekten mit Object.Equals oder dem Is/IsNot-Operator ..                    | 315        |
| Equals, Is und IsNot im praktischen Entwicklungseinsatz.....  | 317        |
| Übersicht über die Eigenschaften und Methoden von Object .....  | 318        |
| Shadowing (Überschatten) von Klassenprozeduren .....  | 318        |
| Shadows als Unterbrecher der Klassenhierarchie .....  | 320        |
| Sonderform »Modul« in Visual Basic .....  | 324        |
| Singleton-Klassen und Klassen, die sich selbst instanzieren .....                                     | 324        |

|   |            |
|---|------------|
| <b>10 Über Structure und den Unterschied zwischen Referenz- und Wertetypen</b> .....                                | <b>327</b> |
| Der Unterschied zwischen Referenz- und Wertetyp .....   | 327        |
| Erstellen von Wertetypen mit Structure am praktischen Beispiel .....  | 329        |
| Unterschiedliche Verhaltensweisen von Werte- und Referenztypen .....  | 334        |
| Verhalten der Parameterübergabe mit ByVal und ByRef steuern .....   | 336        |
| Konstruktoren und Standardinstanzierungen von Wertetypen .....  | 336        |
| Gezieltes Zuweisen von Speicherbereichen für Struktur-Member mit den StructLayout- und FieldOffset-Attributen ..... | 338        |
| Performance-Unterschiede zwischen Werte- und Referenztypen .....  | 341        |
| Wieso kann durch Vererbung aus einem Object-Referenztyp ein Wertetyp werden? ..                                     | 342        |
| <b>11 Typumwandlungen (Type Casting) und Boxing von Datentypen</b> .....  | <b>343</b> |
| Konvertieren von primitiven Typen .....   | 344        |
| Konvertieren von und in Zeichenketten (Strings) .....   | 345        |
| Konvertieren von Strings mit den Parse- und ParseExact-Methoden ..  | 346        |
| Konvertieren in Strings mit der ToString-Methode .....  | 346        |
| Abfangen von fehlschlagenden Typkonvertierungen mit TryParse oder Ausnahmebehandlern .....                          | 347        |
| Casten von Referenztypen mit DirectCast .....   | 348        |
| Boxing von Wertetypen und primitiven Typen .....  | 349        |
| Zufallszahlen mit der Random-Klasse .....   | 350        |
| Was DirectCast nicht kann .....   | 352        |
| Boxen beim Implementieren von Schnittstellen in Strukturen .....  | 352        |
| <b>12 Beerdigen von Objekten – Dispose, Finalize und der Garbage Collector</b> .....                                | <b>355</b> |
| Der Garbage Collector – die Müllabfuhr in .NET .....  | 357        |
| Generationen .....  | 358        |
| Finalize .....  | 359        |
| Wann Finalize nicht stattfindet .....   | 361        |
| Dispose .....   | 364        |
| Unterstützung durch den Visual Basic-Editor beim Einfügen eines Disposable-Patterns .....                           | 374        |
| <b>13 Operatoren für benutzerdefinierte Typen</b> .....   | <b>377</b> |
| Einführung in Operatorenprozeduren .....  | 378        |
| Vorbereitung einer Struktur oder Klasse für Operatorenprozeduren .....  | 379        |
| Implementierung von Rechenoperatoren .....  | 383        |
| Überladen von Operatorenprozeduren .....  | 384        |
| Implementierung von Vergleichsoperatoren .....  | 385        |
| Implementierung von Typkonvertierungsoperatoren mit Operator CType .....  | 386        |
| Implementieren von Wahr- und Falsch-Auswertungsoperatoren .....   | 387        |
| Problembehandlungen bei Operatorenprozeduren .....  | 389        |
| Aufgepasst bei der Verwendung von Referenztypen .....   | 389        |
| Mehrdeutigkeiten bei der Auflösung von Signaturen .....   | 390        |
| Übersicht der implementierbaren Operatoren .....  | 391        |
| <b>14 Generische Klassen und Strukturen (Generics)</b> .....  | <b>393</b> |
| Verwenden einer Codebasis für verschiedene Typen .....  | 393        |

|   |            |
|---|------------|
| Lösungsansätze . . . . .  | 394        |
| Typengeneralisierung durch den Einsatz generischer Datentypen . . . . .           | 397        |
| Beschränkungen (Constraints) . . . . .  | 400        |
| Beschränkungen für generische Typen auf eine bestimmte Basisklasse . . . . .      | 400        |
| Beschränkungen auf Klassen, die bestimmte Schnittstellen implementieren . . . . . | 405        |
| Beschränkungen auf Klassen, die über einen Standardkonstruktor verfügen . . . . . | 408        |
| Beschränkungen auf Wertetypen . . . . .   | 409        |
| Kombinieren von Beschränkungen und Bestimmen mehrerer Typparameter . . . . .      | 409        |
| Vererben von generischen Typen . . . . .  | 410        |
| <b>15 Ereignisse und Delegaten . . . . .</b>                                      | <b>413</b> |
| Konsumieren von Ereignissen mit WithEvents und Handles . . . . .                  | 415        |
| Auslösen von Ereignissen . . . . .  | 418        |
| Der Umweg über Onxxx . . . . .  | 419        |
| Zur-Verfügung-Stellen von Ereignisparametern . . . . .                            | 420        |
| Die Quelle des Ereignisses: Sender . . . . .                                      | 420        |
| Nähtere Informationen zum Ereignis: EventArgs . . . . .                           | 422        |
| Dynamisches Anbinden von Ereignissen mit AddHandler . . . . .                     | 423        |
| Delegaten . . . . .   | 432        |
| <b>Teil E – Die wichtigsten Datentypen des Frameworks . . . . .</b>               | <b>437</b> |
| <b>16 Primitive Datentypen . . . . .</b>  | <b>439</b> |
| Einführung . . . . .  | 440        |
| .NET-Äquivalente primitiver Datentypen . . . . .                                  | 440        |
| Numerische Datentypen . . . . .   | 442        |
| Numerische Datentypen deklarieren und definieren . . . . .                        | 442        |
| Delegation numerischer Berechnungen an den Prozessor . . . . .                    | 442        |
| Hinweis zur CLS-Konformität . . . . .   | 444        |
| Die numerischen Datentypen auf einen Blick . . . . .                              | 445        |
| Rundungsfehler bei der Verwendung von Single und Double . . . . .                 | 451        |
| Besondere Funktionen, die für alle numerischen Datentypen gelten . . . . .        | 454        |
| Spezielle Funktionen der Fließkommatypen . . . . .                                | 457        |
| Spezielle Funktionen des Wertetyps Decimal . . . . .                              | 459        |
| Der Datentyp Char . . . . .   | 459        |
| Der Datentyp String . . . . .   | 460        |
| Strings – gestern und heute . . . . .   | 461        |
| Strings deklarieren und definieren . . . . .                                      | 461        |
| Der String-Konstruktor als Ersatz von String\$ . . . . .                          | 462        |
| Einem String Zeichenketten mit Sonderzeichen zuweisen . . . . .                   | 463        |
| Speicherbedarf von Strings . . . . .  | 464        |
| Strings sind unveränderlich . . . . .   | 464        |
| Speicheroptimierung von Strings durch das Framework . . . . .                     | 464        |
| Ermitteln der String-Länge . . . . .  | 465        |
| Ermitteln von Teilen eines Strings oder eines einzelnen Zeichens . . . . .        | 466        |
| Angleichen von String-Längen . . . . .  | 466        |
| Suchen und Ersetzen . . . . .   | 467        |
| Algorithmisches Auflösen eines Strings in Teile . . . . .                         | 470        |

|  |            |
|--|------------|
| Ein String-Schmankerl zum Schluss . . . . .  | 472        |
| Iterieren durch einen String . . . . .   | 475        |
| StringBuilder vs. String – wenn es auf Geschwindigkeit ankommt . . . . .                             | 476        |
| Wrapper-Klassen für Betriebssystemaufrufe . . . . .  | 480        |
| Der Datentyp Boolean . . . . .   | 482        |
| Konvertieren von und in numerische Datentypen. . . . .   | 483        |
| Konvertierung von und in Strings . . . . .   | 483        |
| Vergleichsoperatoren, die boolesche Ergebnisse zurückliefern . . . . .                               | 484        |
| Anweisungen, die bedingten Programmcode ausführen . . . . .  | 485        |
| Der Datentyp Date . . . . .  | 488        |
| Rechnen mit Zeiten und Datumswerten – TimeSpan . . . . .   | 489        |
| Bibliothek mit brauchbaren Datumsrechenfunktionen . . . . .  | 490        |
| Zeichenketten in Datumswerte wandeln . . . . .   | 493        |
| <b>17 Kulturabhängiges Formatieren von Zahlen- und Datumswerten . . . . .</b>                        | <b>497</b> |
| Allgemeines über Format Provider in .NET . . . . .   | 497        |
| Kulturabhängige Formatierungen mit CultureInfo . . . . .   | 498        |
| Vermeiden von kulturabhängigen Programmfehlern . . . . .   | 500        |
| Formatierung durch Formatzeichenfolgen . . . . .   | 501        |
| Formatierung von numerischen Ausdrücken durch Formatzeichenfolgen . . . . .                          | 501        |
| Formatierung von numerischen Ausdrücken durch vereinfachte Formatzeichenfolgen . . . . .             | 504        |
| Formatierung von Datums- und Zeitausdrücken durch Formatzeichenfolgen . . . . .                      | 505        |
| Formatierung von Zeitausdrücken durch vereinfachte Formatzeichenfolgen . . . . .                     | 510        |
| Gezielte Formatierungen mit Format Providern . . . . .   | 511        |
| Gezielte Formatierungen von Zahlenwerten mit NumberFormatInfo . . . . .                              | 511        |
| Gezielte Formatierungen von Zeitwerten mit DateTimeFormatInfo . . . . .                              | 512        |
| Kombinierte Formatierungen . . . . .   | 513        |
| Ausrichtungen in kombinierten Formatierungen . . . . .   | 514        |
| Angeben von Formatzeichenfolgen in den Indexkomponenten . . . . .                                    | 516        |
| So helfen Ihnen benutzerdefinierte Format Provider, Ihre Programme zu internationalisieren . . . . . | 517        |
| Benutzerdefinierte Format Provider durch IFormatProvider und ICustomFormatter . . . . .              | 527        |
| Automatisch formatierbare Objekte durch Einbinden von IFormattable . . . . .                         | 530        |
| <b>18 Enums (Aufzählungen) . . . . .</b>   | <b>533</b> |
| Bestimmung der Werte der Aufzählungselemente . . . . .   | 534        |
| Bestimmung der Typen der Aufzählungselemente . . . . .   | 535        |
| Ermitteln des Elementtyps zur Laufzeit . . . . .   | 535        |
| Konvertieren von Enums . . . . .   | 536        |
| In Zahlenwerte umwandeln und aus Werten definieren . . . . .   | 536        |
| In Strings umwandeln und aus Strings definieren . . . . .  | 536        |
| Flags-Enum (Flags-Aufzählungen) . . . . .  | 537        |
| Abfrage von Flags-Aufzählungen . . . . .   | 538        |
| <b>19 Arrays und Auflistungen (Collections) . . . . .</b>  | <b>539</b> |
| Grundsätzliches zu Arrays . . . . .  | 540        |
| Arrays als Parameter und Funktionsergebnis . . . . .   | 541        |

|  |            |
|--|------------|
| Änderung der Array-Dimensionen zur Laufzeit .....  | 542        |
| Wertevorbelegung von Array-Elementen im Code .....   | 547        |
| Mehrdimensionale und verschachtelte Arrays.....  | 547        |
| Die wichtigsten Eigenschaften und Methoden von Arrays .....  | 548        |
| Implementierung von Sort und BinarySearch für eigene Klassen .....                                 | 551        |
| Enumeratoren .....   | 558        |
| Benutzerdefinierte Enumeratoren mit IEnumerable .....  | 559        |
| Grundsätzliches zu Auflistungen .....  | 562        |
| Die wichtigen Auflistungen der Base Class Library.....   | 565        |
| ArrayList – universelle Ablage für Objekte .....   | 565        |
| Typsichere Auflistungen auf Basis von CollectionBase .....   | 568        |
| Hashtables – für das Nachschlagen von Objekten.....  | 571        |
| Anwenden von Hashtables .....  | 572        |
| Verwenden eigener Klassen als Schlüssel (Key).....   | 581        |
| Enummerieren von Datenelementen in einer Hashtable .....   | 585        |
| Typsichere Hashtable .....   | 585        |
| Queue – Warteschlangen im FIFO-Prinzip.....  | 587        |
| Stack – Stapelverarbeitung im LIFO-Prinzip.....  | 589        |
| SortedList – Elemente ständig sortiert halten .....  | 590        |
| Sortierung einer SortedList nach beliebigen Datenkriterien .....                                   | 591        |
| <b>20 Arbeiten mit generischen Typen und generischen Auflistungen .....</b>                        | <b>595</b> |
| Wertetypen, die Nothing speichern können – Nullable(Of ).....                                      | 595        |
| Besonderheiten bei Nullable(Of ) beim Boxen.....   | 597        |
| Generische Auflistungen (Generic Collections) .....  | 599        |
| KeyedCollection – Schlüssel/Wörterbuch-Auflistung mit zusätzlichem Index- Abrufen .....            | 602        |
| Elementverkettungen mit LinkedList(Of ).....   | 605        |
| Auflistungen und Aktionen (Actions), Aussageprüfer (Predicates) und Vergleiche (Comparisons) ..... | 607        |
| ForEach und die generische Action-Klasse .....   | 608        |
| Sort und die generische Comparison-Klasse .....  | 609        |
| Find und die generische Predicate-Klasse .....   | 610        |
| <b>21 Reguläre Ausdrücke (Regular Expressions) .....</b>   | <b>613</b> |
| RegExperimente mit dem RegExplorer .....   | 614        |
| Erste Gehversuche mit Regulären Ausdrücken .....   | 616        |
| Einfache Suchvorgänge .....  | 616        |
| Einfache Suche nach Sonderzeichen .....  | 617        |
| Komplexere Suche mit speziellen Steuerzeichen .....  | 618        |
| Verwendung von Quantifizierern .....   | 620        |
| Gruppen .....  | 622        |
| Suchen und Ersetzen .....  | 624        |
| Captures .....   | 625        |
| Optionen bei der Suche .....   | 627        |
| Steuerzeichen zu Gruppdefinitionen .....   | 629        |
| Programmieren von Regulären Ausdrücken .....   | 630        |
| Ergebnisse im Match-Objekt .....   | 630        |

|  |            |
|--|------------|
| Die Matches-Auflistung .....   | 631        |
| Abrufen von Captures und Gruppen eines Match-Objektes .....  | 632        |
| Regex am Beispiel: Berechnen beliebiger mathematischer Ausdrücke.....  | 634        |
| Der Formelparser .....   | 635        |
| Die Klasse ADFormularParser .....  | 636        |
| Vererben der Klasse ADFormularParser, um eigene Funktionen hinzuzufügen .....  | 654        |
| <b>22 Serialisierung von Objekten</b> .....  | <b>657</b> |
| Einführung in Serialisierungstechniken .....   | 658        |
| Serialisieren mit SoapFormatter und BinaryFormatter .....  | 660        |
| Flaches und tiefes Klonen von Objekten.....  | 666        |
| Universelle DeepClone-Methode .....  | 669        |
| Serialisieren von Objekten mit Zirkelverweisen .....   | 671        |
| Serialisierung von Objekten unterschiedlicher Versionen beim Einsatz von<br>BinaryFormatter oder SoapFormatter-Klassen ..... | 674        |
| XML-Serialisierung .....   | 674        |
| Prüfen der Versionsunabhängigkeit der XML-Datei .....  | 679        |
| Serialisierungsfehler bei KeyedCollection .....  | 680        |
| Workaround .....   | 684        |
| <b>23 Attribute und Reflection</b> .....   | <b>687</b> |
| Genereller Umgang mit Attributen .....   | 688        |
| Einsatz von Attributen am Beispiel von ObsoleteAttribute .....   | 688        |
| Die speziell in Visual Basic verwendeten Attribute .....   | 690        |
| Einführung in Reflection .....   | 691        |
| Die Type-Klasse als Ausgangspunkt für alle Typenuntersuchungen .....   | 692        |
| Klassenanalysefunktionen, die ein Type-Objekt bereitstellt. ....   | 694        |
| Objekthierarchie von MemberInfo und Casten in den spezifischen Info-Typ .....  | 696        |
| Ermitteln von Eigenschaftswerten über PropertyInfo zur Laufzeit .....  | 696        |
| Erstellung benutzerdefinierter Attribute und deren Erkennen zur Laufzeit .....   | 697        |
| Ermitteln von benutzerdefinierten Attributen zur Laufzeit .....  | 700        |
| <b>Teil F – Vereinfachungen in Visual Basic 2005</b> .....   | <b>703</b> |
| <b>24 Eine philosophische Betrachtung der Vereinfachungen in Visual Basic 2005</b> .....                                     | <b>705</b> |
| Die VB2005-Vereinfachungen am Beispiel DotNetCopy .....  | 706        |
| DotNetCopy mit /Autostart und /Silent-Optionen .....   | 709        |
| Exkurs: Die Klassen FileInfo- und DirectoryInfo zur Pflege von Verzeichnissen und<br>Dateien .....                           | 709        |
| Die prinzipielle Funktionsweise von DotNetCopy .....   | 711        |
| <b>25 Der My-Namespace</b> .....   | <b>713</b> |
| Formulare ohne Instanzierung aufrufen .....  | 715        |
| Auslesen der Befehlszeilenargumente mit My.Application.CommandLineArgs .....   | 716        |
| Gezieltes Zugreifen auf Ressourcen mit My.Resources .....  | 718        |
| Anlegen und Verwalten von Ressource-Elementen .....  | 718        |
| Abrufen von Ressourcen mit My.Resources .....  | 719        |

|   |            |
|---|------------|
| Internationalisieren von Anwendungen mithilfe von Ressource-Dateien und dem My-Namespace . . . . .                                | 721        |
| Vereinfachtes Durchführen von Dateioperationen mit My.Computer.FileSystem . . . . .   | 724        |
| Verwenden von Anwendungseinstellungen mit My.Settings . . . . .   | 727        |
| <b>26 Das Anwendungsframework . . . . .</b>   | <b>731</b> |
| Die Optionen des Anwendungsframeworks . . . . .   | 733        |
| Windows XP Look and Feel für eigene Windows-Anwendungen – Visuelle XP-Stile aktivieren . . . . .                                  | 733        |
| Verhindern, dass Ihre Anwendung mehrfach gestartet wird – Einzelinstanzanwendung erstellen . . . . .                              | 733        |
| MySettings-Einstellungen automatisch sichern – Eigene Einstellungen beim Herunterfahren speichern . . . . .                       | 733        |
| Bestimmen, welcher Benutzer-Authentifizierungsmodus verwendet wird . . . . .  | 733        |
| Festlegen, wann eine Anwendung »zu Ende« ist – Modus für das Herunterfahren . . . . .   | 734        |
| Einen Splash-Dialog beim Starten von komplexen Anwendungen anzeigen – Begrüßungsdialog . . . . .                                  | 734        |
| Eine Codedatei implementieren, die Anwendungsereignisse (Starten, Beenden, Netzwerkzustand, generelle Fehler) behandelt . . . . . | 734        |
| <b>Teil G – Entwickeln von SmartClient-Anwendungen . . . . .</b>  | <b>739</b> |
| <b>27 Programmieren mit Windows Forms . . . . .</b>   | <b>741</b> |
| Strukturieren von Daten in Windows Forms-Anwendungen . . . . .  | 743        |
| Formulare zur Abfrage von Daten verwenden – Aufruf von Formularen aus Formularen . . . . .  | 754        |
| Der Umgang mit modalen Formularen . . . . .   | 755        |
| Überprüfung auf richtige Benutzereingaben in Formularen . . . . .   | 766        |
| Anekdoten aus der Praxis . . . . .  | 767        |
| Anwendungen über die Tastatur bedienbar machen . . . . .  | 771        |
| Definition von Schnellzugriffstasten per »&«-Zeichen . . . . .  | 771        |
| Accept- und Cancel-Schaltflächen in Formularen . . . . .  | 774        |
| Über das »richtige« Schließen von Formularen . . . . .  | 775        |
| Unsichtbarmachen eines Formulars mit Hide . . . . .   | 775        |
| Schließen des Formulars mit Close . . . . .   | 775        |
| Entsorgen des Formulars mit Dispose . . . . .   | 776        |
| Grundsätzliches zum Darstellen von Daten aus Auflistungsklassen in Steuerelementen . . . . .                                      | 777        |
| Darstellen von Daten aus Auflistungen im ListView-Steuerelement . . . . .   | 777        |
| Spaltenköpfe . . . . .  | 778        |
| Hinzufügen von Daten . . . . .  | 778        |
| Beschleunigen des Hinzufügens von Elementen . . . . .   | 779        |
| Automatisches Anpassen der Spaltenbreiten nach dem Hinzufügen aller Elemente . . . . .  | 779        |
| Zuordnen von ListViewItems und Objekten, die sie repräsentieren . . . . .   | 780        |
| Feststellen, dass ein Element der Liste selektiert wurde . . . . .  | 781        |
| Selektieren von Elementen in einem ListView-Steuerelement . . . . .   | 782        |
| Verwalten von Daten aus Auflistungen mit dem DataGridView-Steuerelement . . . . .   | 782        |
| Über Datenbindung, Bindungsquellen, die BindungSource-Komponente und warum Currency nicht unbedingt Währung heißt . . . . .       | 784        |
| Erstellen einer gebundenen DataGridView mit dem Visual Basic-Designer . . . . .   | 786        |

|  |            |
|--|------------|
| Sortieren und Benennen der Spalten eines DataGridView-Steuerelements . . . . .                               | 791        |
| Programmtechnisches Abrufen der aktuellen Zeile und aktuellen Spalte . . . . .                               | 796        |
| Formatieren von Zellen . . . . .   | 797        |
| Problemlösung für das Parsen eines Zellennwerts mit einem komplexen Anzeigeformat . . . . .                  | 797        |
| Verhindern des Editierens bestimmter Spalten aber Gestatten ihrer Neueingabe . . . . .                       | 799        |
| Überprüfen der Richtigkeit von Eingaben auf Zellen- und Zeilenebene . . . . .                                | 800        |
| Ändern des Standardpositionierungsverhaltens des Zellencursors nach dem Editieren einer Zelle . . . . .      | 803        |
| Entwickeln von MDI-Anwendungen . . . . .   | 805        |
| Vererben von Formularen . . . . .  | 813        |
| ControlCollection vs. Steuerelemente-Array aus VB6 . . . . .   | 814        |
| Ein erster Blick auf den Designer-Code eines Formulars . . . . .   | 817        |
| Modifizierer von Steuerelementen in geerbten Formularen . . . . .  | 820        |
| <b>28 Im Motorraum von Formularen und Steuerelementen . . . . .</b>  | <b>821</b> |
| Über das Vererben von Form und die Geheimnisse des Designer-Codes . . . . .                                  | 821        |
| Geburt und Tod eines Formulars – New und Dispose . . . . .   | 823        |
| Ereignisbehandlung von Formularen und Steuerelementen . . . . .  | 827        |
| Vom Programmstart mithilfe des Anwendungsframeworks über eine Benutzeraktion zur Ereignisauslösung . . . . . | 828        |
| Implementieren neuer Steuerelementereignisse auf Basis von Warteschlangenauswertungen . . . . .              | 833        |
| Wer oder was löst welche Formular- bzw. Steuerelementereignisse wann aus? . . . . .                          | 834        |
| Kategorie Erstellen und Zerstören des Formulars . . . . .  | 836        |
| Kategorie Mausereignisse des Formulars . . . . .   | 839        |
| Kategorie Tastaturereignisse des Formulars . . . . .   | 841        |
| Kategorie Position und Größe des Formulars . . . . .   | 842        |
| Kategorie Anordnen der Komponenten und Neuzeichnen des Formulars . . . . .                                   | 843        |
| Kategorie Fokussierung des Formulars . . . . .   | 845        |
| Kategorie Tastaturvorverarbeitungsnachrichten des Formulars . . . . .  | 846        |
| Kategorie Erstellen/Zerstören des Controls (des Steuerelements) . . . . .                                    | 847        |
| Kategorie Mausereignisse des Controls . . . . .  | 848        |
| Kategorie Tastaturereignisse des Controls . . . . .  | 850        |
| Kategorie Größe und Position des Controls . . . . .  | 851        |
| Kategorie Neuzeichnen des Controls und Anordnen untergeordneter Komponenten . . . . .                        | 853        |
| Kategorie Tasturnachrichtenvorverarbeitung des Controls . . . . .  | 855        |
| Die Steuerungsroutinen des Beispielprogramms . . . . .   | 856        |
| Anmerkungen zum Beispielprogramm . . . . .   | 861        |
| <b>29 GDI+ zum Zeichnen von Formular- und Steuerelementinhalten verwenden . . . . .</b>                      | <b>863</b> |
| Einführung in GDI+ . . . . .   | 864        |
| Linien, Flächen, Pens und Brushes . . . . .  | 867        |
| Angabe von Koordinaten . . . . .   | 868        |
| Wieso Integer- und Fließkommaangaben für Positionen und Ausmaße? . . . . .                                   | 869        |
| Wie viel Platz habe ich zum Zeichnen? . . . . .  | 869        |
| Das gute, alte Testbild und GDI+ im Einsatz sehen! . . . . .   | 869        |
| Exaktes Einpassen von Text mit GDI+-Skalierungsfunktionen . . . . .  | 876        |

|   |            |
|---|------------|
| Flimmerfreie, fehlerfreie und schnelle Darstellungen von GDI+-Zeichnungen.....      | 878        |
| Zeichnen ohne Flimmern .....  | 879        |
| Programmtechnisches Bestimmen der Formulargröße .....                               | 882        |
| Was Sie beim Zeichnen von breiten Linienzügen beachten sollten .....                | 884        |
| <b>30 Entwickeln von Steuerelementen.....</b>                                       | <b>893</b> |
| Neue Steuerelemente auf Basis vorhandener Steuerelemente implementieren .....       | 894        |
| Der Weg eines Steuerelements vom Klassencode in die Toolbox.....                    | 896        |
| Implementierung von Funktionslogik und Eigenschaften .....                          | 898        |
| Steuern von Eigenschaften im Eigenschaftenfenster .....                             | 900        |
| Entwickeln von konstituierenden Steuerelementen .....                               | 903        |
| Anlegen eines Projekts für die Erstellung einer eigenen Steuerelement-Assembly..... | 904        |
| Initialisieren des Steuerelements .....   | 908        |
| Methoden und Ereignisse delegieren.....   | 911        |
| Implementieren der Funktionslogik .....   | 912        |
| Implementierung der Eigenschaften .....   | 914        |
| Erstellen von Steuerelementen von Grund auf .....                                   | 916        |
| Ein Label, das endlich alles kann .....   | 916        |
| Vorüberlegungen und Grundlagenerarbeitung .....                                     | 917        |
| Klasseninitialisierungen und Einrichten der Windows-Darstellungsstile               |            |
| des Steuerelements .....  | 918        |
| Zeichnen des Steuerelements .....   | 920        |
| Der Unterschied zwischen Refresh, Invalidate und Update .....                       | 923        |
| Größenbeeinflussung durch andere Eigenschaften .....                                | 924        |
| Implementierung der Blink-Funktionalität .....                                      | 926        |
| Designercode-Generierung und Zurücksetzen von werteerbenden Eigenschaften mit       |            |
| ShoudSerializeXXX und ResetXXX .....  | 929        |
| Designer-Reglementierungen .....  | 931        |
| <b>31 Mehreres zur gleichen Zeit erledigen – Threading in .NET .....</b>            | <b>933</b> |
| Threads durch ein Thread-Objekt initiieren .....                                    | 936        |
| Starten von Threads .....   | 938        |
| Grundsätzliches über Threads .....  | 939        |
| Synchronisieren von Threads .....   | 939        |
| Synchronisieren der Codeausführung mit SyncLock .....                               | 939        |
| Mehr Flexibilität in kritischen Abschnitten mit der Monitor-Klasse .....            | 943        |
| Synchronisieren von beschränkten Ressourcen mit Mutex .....                         | 946        |
| Weitere Synchronisierungsmechanismen .....  | 949        |
| Verwenden von Steuerelementen in Threads .....                                      | 953        |
| Managen von Threads .....   | 954        |
| Starten eines Threads mit Start .....   | 954        |
| Vorübergehendes Aussetzen eines Threads mit Sleep – Statusänderungen im             |            |
| Framework bei Suspend und Resume .....  | 954        |
| Abbrechen und Beenden eines Threads .....   | 955        |
| Datenaustausch zwischen Threads durch Kapseln von Threads in Klassen .....          | 959        |
| Der Einsatz von Thread-Klassen in der Praxis .....                                  | 962        |
| Verwenden des Thread-Pools .....  | 970        |
| Thread-sichere Formulare in Klassen kapseln .....                                   | 975        |

|  |             |
|--|-------------|
| Threads durch den Background-Worker initiieren . . . . .   | 978         |
| Threads durch asynchrone Aufrufe von Delegaten initiieren . . . . .  | 981         |
| <b>32 SQL Server 2005 und ADO.NET . . . . .</b>  | <b>983</b>  |
| SQL Server 2005 Express . . . . .  | 984         |
| Einsatz von SQL Server Express in eigenen Anwendungen . . . . .  | 985         |
| Installation von SQL Server 2005 Express unter Windows XP im »gemischten Modus« . . . . .                              | 986         |
| Konfiguration der Netzwerkeinstellungen und Einrichten der Firewall unter Windows XP SP2 . . . . .                     | 991         |
| Grundsätzliches zu Datenbanken . . . . .   | 992         |
| Aufbau der Beispieldatenbank . . . . .   | 992         |
| Klärung grundsätzlicher Begriffe . . . . .   | 994         |
| Einsehen von Daten mit dem Server-Explorer . . . . .   | 995         |
| Anekdoten aus der Praxis . . . . .   | 997         |
| Programmieren mit ADO.NET . . . . .  | 998         |
| Verbindungen zur Datenbank mit der Connection-Klasse herstellen und Befehle mit der Command-Klasse ausführen . . . . . | 998         |
| Datenverbindungen herstellen und Resultsets mit der DataReader-Klasse auslesen . . . . .                               | 999         |
| Unverbundene Daten mit dem DataTable-Objekt verwalten . . . . .  | 1001        |
| Einige SELECT-Beispiele: . . . . .   | 1006        |
| Ändern und Ergänzen von Daten in Datentabellen . . . . .   | 1009        |
| Was machte das »alte« ADO? . . . . .   | 1011        |
| Verwenden von DataAdapter, DataTable und Command-Objekten zur Übermittlung von Aktualisierungen . . . . .              | 1012        |
| Für »Mal-Eben-Abfragen« – die CommandBuilder-Klasse . . . . .  | 1019        |
| DataSets und typisierte DataSets . . . . .   | 1021        |
| Erstellen eines typisierten DataSets mit der Visual Studio IDE . . . . .   | 1022        |
| Grundsätzliches zu typisierten DataSets . . . . .  | 1025        |
| Erweitern des TableAdapters um zusätzliche Abfragefunktionen, denen Parameter übergeben werden können . . . . .        | 1027        |
| Erstellen von datenbankgebundenen Formularen in Windows Forms-Anwendungen .  | 1033        |
| Und so geht es weiter . . . . .  | 1037        |
| <b>Stichwortverzeichnis . . . . .</b>  | <b>1039</b> |

# Am Anfang war ...

... eine erste Beta-Version, wie immer keine Zeit, aber dennoch nach dem ersten Blick soviel Ablenkung durch Begeisterung, dass andere Termine fast platzten. Warum? Ich sah mich mit der ersten »stabilen« Beta-Version von Visual Studio 2005 konfrontiert und war davon so aus dem Häuschen, dass ich mich mehrere Tage damit beschäftigte. Das ist jetzt schon gut eineinhalb Jahre her, aber ich kann mich noch sehr genau daran erinnern, dass ich mit einem solchen Enthusiasmus, der bei mir damals aufkam, nach dem Umstieg von Visual Basic 6 auf Visual Basic .NET nicht mehr gerechnet hatte. Denn schon damals, also vor ca. 3 Jahren, war ich ebenfalls enorm begeistert – die Leser, die das Vorwort des Vorläufers dieses Buchs gelesen haben, wissen warum – und ich glaubte nicht, dass man auf ein schon geniales Entwicklungssystem, wie es Visual Studio .Net 2003 schnell für mich wurde, noch eins draufsetzen konnte. Ich wurde eines Besseren belehrt.

Diese Begeisterung hatte Konsequenzen, denn auch wenn es im Juli 2004 noch so aussah, dass die fertige Version von Visual Studio 2005 noch in weiter Ferne lag, traf ich eine mehr emotionale als rationale Entscheidung: Ich entschloss mich, ein Projekt mit einem Umgang von über 1000 Mannstunden bereits mit dieser Version zu beginnen. Und um es gleich vorweg zu nehmen: Durch das frühe Stadium von Visual Studio 2005 hatte ich Erlebnisse, die mich mehr als einmal daran zweifeln ließen, ob ich die richtige Entscheidung getroffen hatte. Bestimmte Sachen, die noch nicht richtig funktionierten, schmissen unser Team teils Wochen zurück. Andere Sachen, die funktionierten, wurden aus der finalen Version aus diesen oder jenen Gründen verbannt, und wir mussten ganze Funktionsblöcke unserer Software neu konzipieren und natürlich auch neu entwickeln. Aber glauben Sie mir: Letzten Endes bereue ich die Entscheidung auf keinen Fall. Mit dem Erscheinen von Visual Studio 2005 ist auch unser Projekt abgeschlossen. Und unsere Anwendung sieht dank Visual Studio 2005 so aus, wie moderne Anwendungsprogramme heutzutage aussehen sollten. Es verfügt über eine Benutzeroberfläche, die mit Visual Studio .NET nie in der gleichen Zeit machbar gewesen wäre. Es lief selbst in der ersten Version unglaublich stabil – unsere Tester hatten nie weniger Arbeit bei den ersten Performance- und Zuverlässigkeitstests – und die Software machte selbst bei den ersten Testinstallationen für den Produktiveinsatz bemerkenswert wenige Probleme. Ohne uns selbst loben zu wollen, liegt das sicherlich auch daran, dass ein paar wirklich erfahrene .NET-Entwickler am Werk waren. Doch ist das auch zu einem abermals so großen Teil der darunter liegenden Plattform – in diesem Fall SQL Server 2005 und .NET-Framework 2.0 – zu verdanken; eine Tatsache übrigens, die alle Beteiligten unseres Entwicklungsteams angenehm überraschte, denn schließlich arbeiteten wir bis zu letzt mit nicht vollständig getesteten Beta-Versionen von Visual Studio 2005.

Was diese 2005er Version von Visual Studio bzw. Visual Basic anbelangt: Die Jungs aus Redmond haben sich wirklich was einfallen lassen und nicht nur tonnenweise Gehirnschmalz in neue, sinnvolle Funktionen investiert, sondern auch dafür gesorgt, dass Anwendungen, die auf diesem System aufbauen, zuverlässiger agieren als je zuvor.

So ganz nebenbei haben wir uns in der Zeit natürlich auch noch eine ganze Menge an Know-how angeeignet, das inzwischen absolut praxiserprobт ist und Ihnen in diesem Buch zu Gute kommt – so meine ich zu mindest.

Ich hoffe in diesem Sinne, Ihnen mit diesem Buch viele der Erfahrungen weiter geben zu können, die wir im Laufe der letzten Monate gesammelt haben, und dieses Buch dadurch zu einem richtigen Entwicklerbuch zu machen, das nicht nur bloße Theorie transportiert, sondern Ihnen wirklich brauchbare Tipps für die tägliche Praxis liefern kann.

Auf Ihre Reaktionen, die Sie mir an *entwicklerbuch@activedevelop.de* senden können, bin ich wirklich gespannt!

*Klaus Löffelmann, Lippstadt im März 2006.*

## **http://activedevelop.de – Ein wenig Werbung in eigener Sache**

Man bekommt nicht oft die Möglichkeit, für sein eigenes Gewerbe ohne größeren finanziellen Aufwand werben zu können. Umso mehr mag ich natürlich die Gelegenheit am Schopf packen, um an dieser Stelle ein wenig Öffentlichkeitsarbeit für mein eigenes Dienstleistungsunternehmen zu betreiben. So denn: Falls Sie Bedarf an

- Beratungsdienstleistungen in Sachen .NET 1.1, 2.0 und SQL Server 2005,
- Schulungen (auch inhouse) mit den Schwerpunkten »Umstieg von VB6«, SQL Server 2005 und ADO.NET 2.0, C# 2005 und Visual Basic 2005,
- Dokumentationserstellung, Übersetzungen und Softwarelokalisierungen oder
- Software Development Outsourcing

haben, informieren Sie sich einfach auf unserer Website unter *http://activedevelop.de*, und setzen Sie sich mit uns in Verbindung.

## **Danksagungen**

Das Schreiben dieses Buches war mal wieder ein hartes Stück an Arbeit und wäre ohne entsprechende Unterstützung von Freunden, Familie und den Mitarbeitern von Microsoft und Microsoft Press nicht möglich gewesen. Deswegen möchte ich zunächst **Thomas Braun-Wiesholler**, meinem Lektor, für die Unterstützung in Form von Software, Betriebssystemen, Links, Büchern und vor allen Dingen unzähligen Betaversionen von Visual Studio 2005 bedanken. Unzähligen? Doch nicht: Ich hab sie alle auf einer Spindel gesammelt, und, lieber Thomas, du hast seit der Beta 1 von Visual Studio 2005 im April 2004 nicht weniger als 56 DVDs mit den verschiedensten Versionen von Visual Studio, dem Foundation-Server, der MSDN und Yukon mit heldenhaftem Einsatz organisiert, gebrannt und mir meistens mit der teureren Samstagszustellung geschickt. Und außerdem, das sei hier in aller Deutlichkeit gesagt: Mir macht es echt Spaß, mit dir zusammenzuarbeiten!

Ein ganz dickes Dankeschön an dieser Stelle auch dem Profi aller Profis in Sachen SQL Server **Ruprecht Dröge**, der das Fachlektorat dieses Buches übernommen hat, und mit dem es richtig, richtig lustig ist, zusammenzuarbeiten.

Um die Überprüfung der Lauffähigkeit aller Beispielprogramme und die Kontrolle der richtigen Pfadangaben im Buch hat sich darüber hinaus mein Bürogemeinschaftskollege und guter Freund **Jürgen Heckhuis** gekümmert – auch ihm sei dafür ausdrücklich gedankt!

Dank gilt auch meinen Eltern Gabi und Arnold sowie meinen Freunden, die mich wie beim letzten Buch durch regelmäßiges Besuchen und mich ab und zu entführen gezeigt haben, dass es auch noch »eine Welt da draußen« gibt. Dazu zählen insbesondere Claudia, Christian, Gaby, Miri, Momo und Uta. Und natürlich Uwe Thiemann und mein irischer Freund Gareth Clarke: *Go n'éirí leat, go n' éirí an bóthar leat is céad míle beannachta. Sin sin, níl aon scéal eile agam.*

Und lieber Hartmut Woerrlein. Du bist ein guter Freund und darüber hinaus der Mensch, den ich in dieser verrückten Branche, in der wir arbeiten, mit über 18 Jahren schon am längsten kenne. Und Du hast in Deinem ComputerBILD-Lexikon einen Vorwortkrieg angezettelt, auf den ich mich nur zu gerne einlasse. Du willst es so. Ich habe Dich in meinem letzten Buch also nicht erwähnt und Dir schon gar nichts gewidmet? Wie konnte ich, ich Rüpel!? Das will ich hiermit wieder gut machen. Ich widme Dir deswegen als kleine Stichelei das Stichwortverzeichnis dieses Buches! ;-)

### **Eine kleine Anekdote in Sachen Motivation**

Vor ca. 20 Jahren habe ich eine gute Freundin kennen gelernt. Diese Freundin habe ich bis vor 7 Jahren als einen Menschen gekannt, die mit ihrem Leben scheinbar zufrieden war. Als Zahnärzthelferin hat sie ihren beruflichen Werdegang begonnen, und keiner aus unserem Freundeskreis hat es jemals auch nur in Erwägung gezogen, dass sie sich in Sachen Job und Karriere auch nur irgendwann einmal in ihrem Leben irgendetwas anderes vorstellen würde.

Zwischendurch haben wir uns für einen gewissen Zeitraum aus den Augen verloren. Und so freute ich mich sehr, als ich sie nach einiger Zeit zufällig wieder traf. Doch was folgte, verschlug mir die Sprache: Sie erzählte mir von Software Requirements, Pflichtenheften und Projektplanungen. Sie verwickelte mich in ein Gespräch über objektorientiertes Programmieren, Klassen, Java, .NET und Entwicklungsumgebungen – doch das Gespräch war vergleichsweise einseitig. Sie beobachte mich mit einem triumphierenden aber ganz bestimmt verdienten Grinsen auf dem Gesicht, wie ich vor ungläubigem Staunen eigentlich nur stillschweigend ihren Worten zu folgen versuchte.

Es stellte sich heraus: Sie hatte hart gearbeitet und tatsächlich eine Ausbildung hingelegt, für die sie sich jeden Tag aufs Neue hatte motivieren müssen. Und leicht fiel ihr das beileibe nicht. Aber sie hat sich durchgebissen, es geschafft und schließlich einen nicht schlecht dotierten Job in »ihrer« neuen Branche gefunden, der ihr Spaß macht, und in dem sie endlich zu zeigen in der Lage war und ist, was sie wirklich kann.

Sich zu motivieren und sich täglich wieder den Herausforderungen zu stellen, gehört gerade in der Branche, in der wir uns bewegen, zu den Eigenschaften, die enorm wichtig sind. Mindestens genau so wichtig, wie das Beherrischen des Handwerkzeugs. Mich hat das, was sie geleistet und erreicht hat, enorm beeindruckt, und ich finde es deswegen auch so erwähnenswert.

Diese Freundin hat es an Weihnachten 2005 leider gesundheitsmäßig schwer mit etwas getroffen, das man *niemals*, und schon gar nicht mit 34 Jahren bekommen sollte. Aber Silke: Du hast schon sooft gezeigt, was man durch Selbstmotivation, Durchhalten und die Welt stets positiv Sehen alles erreichen kann.

Und deswegen schaffst Du das dieses Mal auch.



# **Teil A**

## **Einführung**

---

### **3 Einführung**

---

Eine kleine Orientierungshilfe ergibt meiner Meinung nach direkt am Anfang dieses Buches Sinn. Welche Voraussetzungen brauchen Sie, um aus diesem Buch den größtmöglichen Nutzen zu ziehen? Welche Versionen von Visual Basic und Visual Studio gibt es, und wie unterscheiden sich diese genau? Wie kommen Sie an die Beispiele zu diesem Buch? Wo gibt es weitere Infos und Quellen, die Ihnen neben diesem Buch bei der Entwicklung von Nutzen sein können? Auf diese Frage gibt Ihnen der erste Teil dieses Buches Antworten.



# 1 Einführung

---

- 
- 3 Welche Softwarevoraussetzungen benötigen Sie?**
  - 4 Wissenwertes zur Installation von Visual Studio 2005**
  - 6 Diese Versionen von Visual Studio 2005 gibt es**
  - 9 Der Umgang mit Web-Links in diesem Buch – <http://links.entwicklerbuch.net>**
  - 10 Die Begleitdateien zum Buch**
  - 10 Nützliches zu Visual Basic 2005 – <http://vb2005.de>**
- 

Um es ohne Umschweife zu sagen: Dieses Buch richtet sich an Softwareentwickler, die bereits Programmiererfahrungen mit Basic-Derivaten oder anderen Programmiersprachen haben; es ist also kein Einsteigerbuch und wird Ihnen die Sprache Basic an sich nicht beibringen.

Dieses Buch richtet sich an Programmierer, die sich vor allen Dingen die Eigenarten von Visual Basic 2005 und dem .NET-Framework aneignen und schnelle Ergebnisse in .NET erzielen wollen. Dieses Buch wird Ihnen also beispielsweise nicht erklären, was Variablen oder Datentypen sind; es wird Ihnen aber die Datentypen des Frameworks erläutern und damit die primitiven Datentypen von Visual Basic 2005 erklären. Es wird auch nicht ausführen, was eine Bitmap ist; Sie werden aber Beispiele finden, die zeigen, wie Bitmaps in .NET verwendet werden.

Dieses Buch wird Ihnen allerdings erklären, was Klassen und Schnittstellen sind, denn ohne die läuft in .NET gar nichts mehr. In meiner langjährigen Erfahrung als Softwareentwickler habe ich immer wieder festgestellt, dass ganze Teams noch immer die prozedurale Programmierung der objektorientierten Programmierung vorziehen. Ich werde mich darüber hinaus bemühen, Sie nach Möglichkeit nicht mit theoretischen Ausführungen über bestimmte Themen zu langweilen und Ihnen dafür lieber viele brauchbare Praxisbeispiele präsentieren.

## Welche Softwarevoraussetzungen benötigen Sie?

Um *alle* Beispiele in diesem Buch nachzuvollziehen, benötigen Sie eine aktuelle Version von Visual Studio 2005 in der Professional Edition. Die dem Buch beiliegende Visual Basic 2005 Express Edition reicht aber für das Nachvollziehen von 99 % der Beispiele dieses Buches aus – lediglich einige Beispiele im ADO.NET-Kapitel (► Kapitel 32) erfordern mindestens die Standard Edition von Visual Studio 2005.

## Die Visual Basic 2005 Express Edition auf der beiliegenden Buch-CD

Insgesamt zwei Begleit-CDs finden Sie im diesem Buch. Auf einer befindet sich die zur Drucklegung dieses Buches frei verfügbare Visual Basic 2005 Express Edition, mit der Sie, wie schon gesagt, 99 % Beispiele dieses Buches nachvollziehen können. Alternativ können Sie auch die 180-Tage-Version von Visual Studio 2005 bei Microsoft anfordern, mit der Sie alle Beispiele des Buches nachvollziehen können. Auf der Seite <http://links.entwicklerbuch.net> finden Sie weitere Hinweise zu dieser Version und ihrer Bezugsquellen. Sie gehen damit kein Risiko oder zusätzlichen Installationsaufwand ein, denn Sie können von dieser Version jederzeit auf die lizenzierte Vollversion von Visual Studio umsteigen.<sup>1</sup>

Wenn es um das Nachvollziehen reinen Programmcodes geht (und darum geht es ja in 90 % der Fälle), wäre übrigens sogar der Visual Basic-Compiler ohne die integrierte Entwicklungsumgebung ausreichend, der Bestandteil des frei verfügbaren *Framework.SDK* ist.

Allerdings möchte ich an dieser Stelle eine deutliche Empfehlung aussprechen: Für professionelle Softwareentwicklung sollten Sie auf lange Zeit gesehen den Einsatz von Visual Studio 2005 mindestens in der Ausführung *Professional Edition* 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 2005 größere Entwicklungen durchführen wollten.

---

**TIPP:** Welche Versionen von Visual Studio es gibt, wie sich diese unterscheiden und welche daher für Sie geeignet sind, erfahren Sie im Abschnitt »Diese Versionen von Visual Studio 2005 gibt es« ab Seite 6.

---

## Wissenswertes zur Installation von Visual Studio 2005

Eigentlich müssen Sie nichts Großartiges beachten, wenn Sie Visual Studio auf Ihrem Computer zum Laufen bringen wollen.<sup>2</sup> Die folgende Tabelle zeigt Ihnen die ideale Hardwarevoraussetzung, die ein reibungsloses Arbeiten mit Visual Studio 2005 ermöglicht.

---

**WICHTIG:** Orientieren Sie sich bei dieser Liste lieber nicht an der angegebenen Mindestkonfiguration nach Microsoft-Spezifikation, die eher humoristischen Anforderungen genügen, 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 hohem Kaffeekonsums führt. Ein paar Euro in ein wenig Hardware investiert, können Ihre Turn-Around-Zeiten beim Entwickeln drastisch verbessern!

---

<sup>1</sup> Starten Sie dazu die Systemsteuerung, und öffnen Sie *Software*. Suchen Sie *Visual Studio* in der Liste, und klicken Sie auf *Ändern/Entfernen*. Im Dialog des Visual Studio-Wartungsmodus, der nach einem Warten (heißt es deswegen *Wartungsmodus?*) erscheint, haben Sie nach Klick auf *Weiter* die Möglichkeit, einen neuen Produkt-Key für eine voll lizenzierte Version einzugeben.

<sup>2</sup> ... und Sie zuvor noch keine Beta-Version, eine CTP-Version oder einen Release-Kandidaten von Visual Studio 2005 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)   |
|---|-------------------------------------|---|
| Prozessor   | 600 MHz                             | Pentium P4 2,8 Ghz; Pentium Mobile mit 1,6 Ghz oder Athlon 64 3000 XP.  |
| Ram   | 128 MByte                           | 384 MByte unter Windows 2000 bzw. 512 MByte unter Windows XP  |
| Festplattengröße (Installation der MSDN vorausgesetzt – freier Speicher vor Installation) | 2,8 GByte                           | 2,8 GByte   |
| 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! <sup>3</sup> |

**Tabelle 1.1:** Die Minimal- und Idealvoraussetzungen an die Hardware für ein reibungsloses Arbeiten mit Visual Studio 2005

**HINWEIS:** Mit Visual Studio 2005 erstellte Programme laufen *nicht* auf Windows 95 und Windows NT, und unter Windows 98 oder Windows Millenium steht Ihnen nicht der komplette Funktionsumfang des Framework 2.0 zur Verfügung (Framework 2.0-Programme laufen aber grundsätzlich unter diesen beiden älteren Betriebssystemen entgegen vieler Meinungen schon). Sie benötigen allerdings *mindestens* Windows 2000 mit Service Pack 4, Windows XP Home, Windows XP Professional, Windows 2000 Server (Service Pack 4), die XP-Media-Center-Edition, Windows 2003 Server (natürlich auch R2) oder eine der Windows Vista-Versionen, um die Entwicklungsumgebung (Express Edition, Visual Studio) installieren zu können.

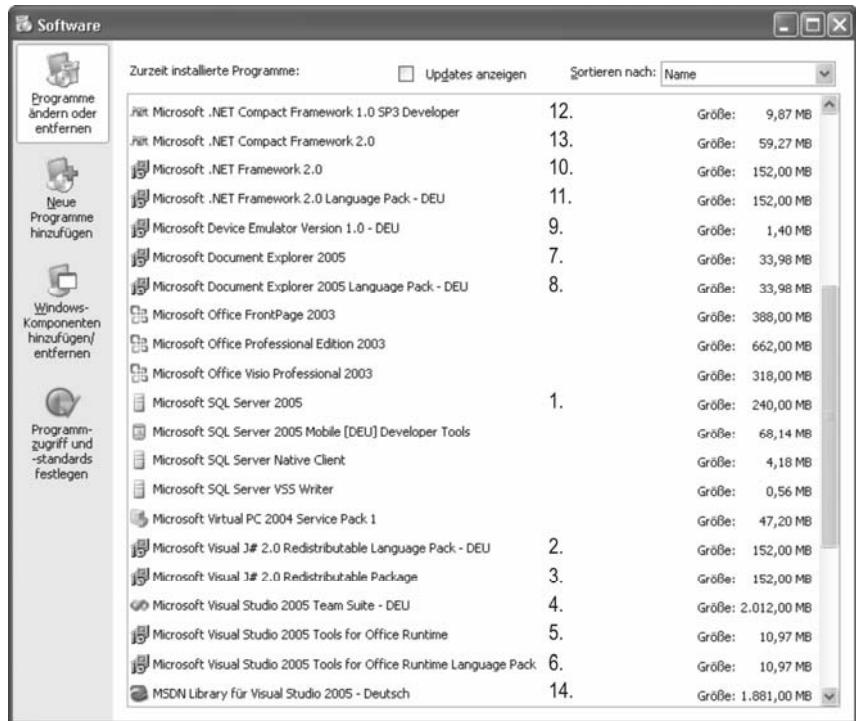
## Deinstallation von Beta 2, CTP oder Release-Kandidaten

Erfahrungsgemäß arbeiten Beta-2-Versionen oder Release-Kandidaten schon erstaunlich stabil, so dass es oft versäumt wird, die »alten« Versionen gegen die RTM<sup>4</sup>-Versionen auszutauschen. Wenn Sie eine der Vorläufer-Versionen von Visual Studio 2005 noch von Ihrem Computer verbannen, um Sie durch die fertige Version zu ersetzen, halten Sie sich unbedingt an eine bestimmte Deinstallationsreihenfolge der verschiedenen Komponenten, damit eine erneute Installation – ein reines Update ohne weitere Handgriffe wird nicht funktionieren – nicht fehlschlägt.

Die folgende Grafik, die dem Dialog entspricht, den Sie sehen, wenn Sie aus der Systemsteuerung Software aufrufen, zeigt Ihnen, in welcher Reihenfolge Sie welche Komponenten deinstallieren sollten.

<sup>3</sup> Kein Budget für einen zweiten Bildschirm, aber Sie haben einen Laptop? Nutzen Sie Ihren Laptop als Zweitbildschirm. Mehr dazu gibt unter dem IntelliLink A0103.

<sup>4</sup> RTM (sprich: »Ahr Tie Ämm« = *Release to manufacturing*, etwa: »zur Produktion freigegeben«).



**Abbildung 1.1:** Sollten Sie noch eine Vorläuferversion von Visual Studio 2005 auf Ihrem Computer installiert haben, deinstallieren Sie bitte die entsprechenden Komponenten in dieser Reihenfolge

**WICHTIG:** Bei der Deinstallation von SQL Server Express können, je nach verwendeter Version, verschiedene Komponenten »hängenbleiben«. Stellen Sie sicher, dass die unter Microsoft SQL Server 2005 aufgezeigten Komponenten alle deinstalliert wurden. Sollte das nicht der Fall sein, deinstallieren Sie die in Abbildung 1.1 zwischen mit 1. und 2. gekennzeichneten Komponenten alle nacheinander manuell.

## Diese Versionen von Visual Studio 2005 gibt es

Das Produktpotfolio der Visual Studio 2005-Familie ist einigermaßen umfassend und variiert in der Leistungsfähigkeit stark im Vergleich zur Visual Studio 2003-Familie. Ein Blick in die folgende Tabelle lohnt sich vor einer Kaufentscheidung also auf alle Fälle!

| Feature                         | Express-Produkte | Visual Studio Standard Edition | Visual Studio Professional Edition | Visual Studio Tools for Office | Visual Studio Team System |
|---------------------------------|------------------|--------------------------------|------------------------------------|--------------------------------|---------------------------|
| IntelliSense                    | Ja               | Ja                             | Ja                                 | Ja                             | Ja                        |
| Codeeditor                      | Ja               | Ja                             | Ja                                 | Ja                             | Ja                        |
| Codeausschnitte (Code Snippets) | Ja               | Ja                             | Ja                                 | Ja                             | Ja                        |

| Feature   | Express-Produkte  | Visual Studio Standard Edition  | Visual Studio Professional Edition | Visual Studio Tools for Office                            | Visual Studio Team System                                 |
|---|---|---|------------------------------------|---|---|
| Enthaltene Programmiersprachen                      | Bei VB, VC#, VC++ und VJ# handelt es sich um einzelne Programmiersprachen.<br>Visual Web Developer beinhaltet VC#, und VB | Alle  | Alle                               | VB und C#   | Alle  |
| Office-Entwicklungsunterstützung                    | Nein  | Nein  | Nein                               | Unterstützung für Excel 2003, Word 2003 und InfoPath 2003 | Unterstützung für Excel 2003, Word 2003 und InfoPath 2003 |
| Benutzeroberfläche                                  | Abgespeckt  | Abgespeckt  | Vollständig                        | Vollständig   | Vollständig   |
| Windows Forms-Designer                              | VB, VC#, VC++, VJ#  | Ja  | Ja                                 | Ja  | Ja  |
| Web Forms-Designer                                  | Visual Web Developer  | Ja  | Ja                                 | Ja  | Ja  |
| Unterstützung für mobile Geräte                     | Nein  | Ja  | Ja                                 | Nein  | Ja  |
| Datenbank-Design-Tools<br>»Datenverbindungs-Knoten« | Lokal   | Zu allen  | Zu allen                           | Zu allen  | Zu allen  |
| Datenzugriff  | VB, VC#, VC++, VJ#: lokal<br><br>Visual Web Developer: lokal und entfernt   | Zu allen  | Zu allen                           | Zu allen  | Zu allen  |
| Dokumentation                                       | 10 MByte "Erste Schritte"; Starter Kits zielen auf Anfänger ab; 200 MByte optional MSDN Express                           | MSDN (Microsoft Developer Network – eine umfangreiche Online-Dokumentation) | MSDN                               | MSDN  | MSDN  |
| Klassendesigner / Objekttest-Bench                  | Nein  | Ja  | Ja                                 | Ja  | Ja  |
| XML-Unterstützung                                   | Nur XML   | XML/XSLT  | XML/XSLT                           | XML/XSLT  | XML/XSLT  |
| Tools zur Veröffentlichung von Anwendungen          | Click Once (im Gegensatz zu einigen MSDN-Angaben!)  | Click Once  | Umfassend                          | Umfassend   | Umfassend   |

| Feature   | Express-Produkte  | Visual Studio Standard Edition                                      | Visual Studio Professional Edition                            | Visual Studio Tools for Office                                | Visual Studio Team System   |
|---|---|---|---|---|---|
| Erweiterbarkeit mit der Visual Studio Extensibility | Lässt nur externe Tools zum Menü hinzufügen. Verwenden Sie Produkte von Drittherstellern. | Erweiterungen können eingebunden, aber nicht selber erstellt werden | Erstellen und einbinden (vollständige Unterstützung)          | Erstellen und einbinden (vollständige Unterstützung)          | Erstellen und einbinden (vollständige Unterstützung)  |
| Report-Funktionalität                               | SQL Server Reporting Services   | SQL Server Reporting Services                                       | SQL Server Reporting Services / Crystal Reports               | SQL Server Reporting Services / Crystal Reports               | SQL Server Reporting Services / Crystal Reports   |
| Quellcodeverwaltung                                 | Keine   | MSSCCI-compatible (Visual SourceSafe wird separat vertrieben)       | MSSCCI-compatible (Visual SourceSafe wird separat vertrieben) | MSSCCI-compatible (Visual SourceSafe wird separat vertrieben) | MSSCCI-compatible (beinhaltet Visual SourceSafe, Visual Studio Team Foundation wird separat vertrieben) |
| Debugging   | Lokal   | Lokal   | Lokal/entfernt  | Lokal/entfernt  | Lokal/entfernt  |
| 64-bit Compiler-unterstützung                       | Nein  | Nein  | Ja  | Ja  | Ja  |
| Server Explorer-"Server"-Zweig                      | Nein  | Nein  | Alle  | Alle  | Alle  |
| SQL Server 2005-Integration                         | Nein  | Nein  | Ja  | Ja  | Ja  |
| Code Profiling                                      | Nein  | Nein  | Nein  | Nein  | Ja  |
| Statische Analysen                                  | Nein  | Nein  | Nein  | Nein  | Ja  |
| Unit testing  | Nein  | Nein  | Nein  | Nein  | Ja  |
| Code Coverage                                       | Nein  | Nein  | Nein  | Nein  | Ja  |
| Projektmanagement                                   | Nein  | Nein  | Nein  | Nein  | Ja  |
| Testfall-Management                                 | Nein  | Nein  | Nein  | Nein  | Ja  |
| Größe   | 80 MByte (Express + SQL Express + .NET Framework Redist)                                  | Eine CD   | Mehrere CDs   | Mehrere CDs   | Mehrere CDs   |



| Feature                     | Express-Produkte | Visual Studio Standard Edition | Visual Studio Professional Edition | Visual Studio Tools for Office  | Visual Studio Team System   |
|-----------------------------|------------------|--------------------------------|------------------------------------|---|---|
| Zusätzliche Tools enthalten | Nein             | Nein                           | SQL Server 2005 Developer Edition  | Windows 2003 Server Developer Edition; SQL Server 2005 Developer Edition; Microsoft Office Access 2003 Developer Extensions; Access 2003 Laufzeitlizenz | SQL Server 2005 Developer Edition (nur in den Client-Produkten enthalten) |

## Der Umgang mit Web-Links in diesem Buch – <http://links.entwicklerbuch.net>

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 (oder, als immer funktionierendes Backup, durch den Anhang dieses Buches) auflösen lassen. Diese Web-Seite lautet:

<http://links.entwicklerbuch.net>

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 *Intelli-Link* (toller Begriff, was?) finden. Lautet eine Passage also:

»Mehr zum Thema Silk-Dämpfung finden Sie unter dem IntelliLink **A0101** – Sie können dort einiges Interessantes über die Entstehung von Materie und dunkler Materie erfahren.«

In diesem Fall geben Sie den URL <http://links.entwicklerbuch.net> 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; Sie benötigen zur Wiedergabe den Real-Player, den Sie unter dem Intelli-Link **A0102** finden).

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 [entwicklerbuch@activedevelop.de](mailto:entwicklerbuch@activedevelop.de) darüber informieren.

## Die Begleitdateien zum Buch

Neben der CD, die Visual Basic 2005 Express Edition beinhaltet, finden Sie im Buch eine weitere CD, die neben den PDF-Dateien des Vorläufers dieses Buches »Visual Basic .NET – Das Entwicklerbuch« auch sämtliche Begleitdateien beinhaltet.

---

**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 2005 – <http://vb2005.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. Die Seite <http://vb2005.de> sollten Sie deswegen ebenso regelmäßig besuchen, wie die Entwicklerbuch-Web Site <http://entwicklerbuch.net>. Vielleicht finden Sie dort das eine oder andere nützliche Werkzeug oder zusätzliche Codebeispiel, mit dem Sie sich Ihre tägliche Entwicklungsarbeit erleichtern können.

# Teil B

## Die Visual Studio-Entwicklungsumgebung

---

- 
- 13      Ein Flug über die Weiten der Visual Studio-IDE**
  - 37      Codeeditor und Formular-Designer enthüllt**
  - 109     Tipps & Tricks für das angenehme Entwickeln zuhause und unterwegs**
- 

Bei einem Buch dieses Umfangs ist es vergleichsweise schwierig, eine Struktur zu schaffen, die auf die Bedürfnisse jedes Lesers ideal einzugehen in der Lage ist. Redundanzen bei Erklärungen lassen sich dabei nicht vermeiden – eigentlich sind sie an einigen Stellen sogar recht sinnvoll. Gerade dieses erste Kapitel, das die Entwicklungsumgebung – die so genannte *IDE (Integrated Development Environment*, etwa: *integrierte Entwicklungsumgebung*) – vorstellt, und Ihnen Tipps und Tricks gibt, die Ihnen das Entwickeln im Büro und auch unterwegs so einfach wie möglich machen sollen, ist davon besonders betroffen: Es nimmt einige Features vorweg, auf die im Laufe des Buches natürlich noch detaillierter eingegangen wird. Und gerade für Leser, die schon einige Erfahrungen beim Entwickeln von .NET mitbringen, ist die Vorgehensweise dieses Kapitels wahrscheinlich dennoch die beste, um sich in der neuen Umgebung schnellstmöglich zurechtzufinden.



# **2 Ein Flug über die Weiten der Visual Studio-IDE**

---

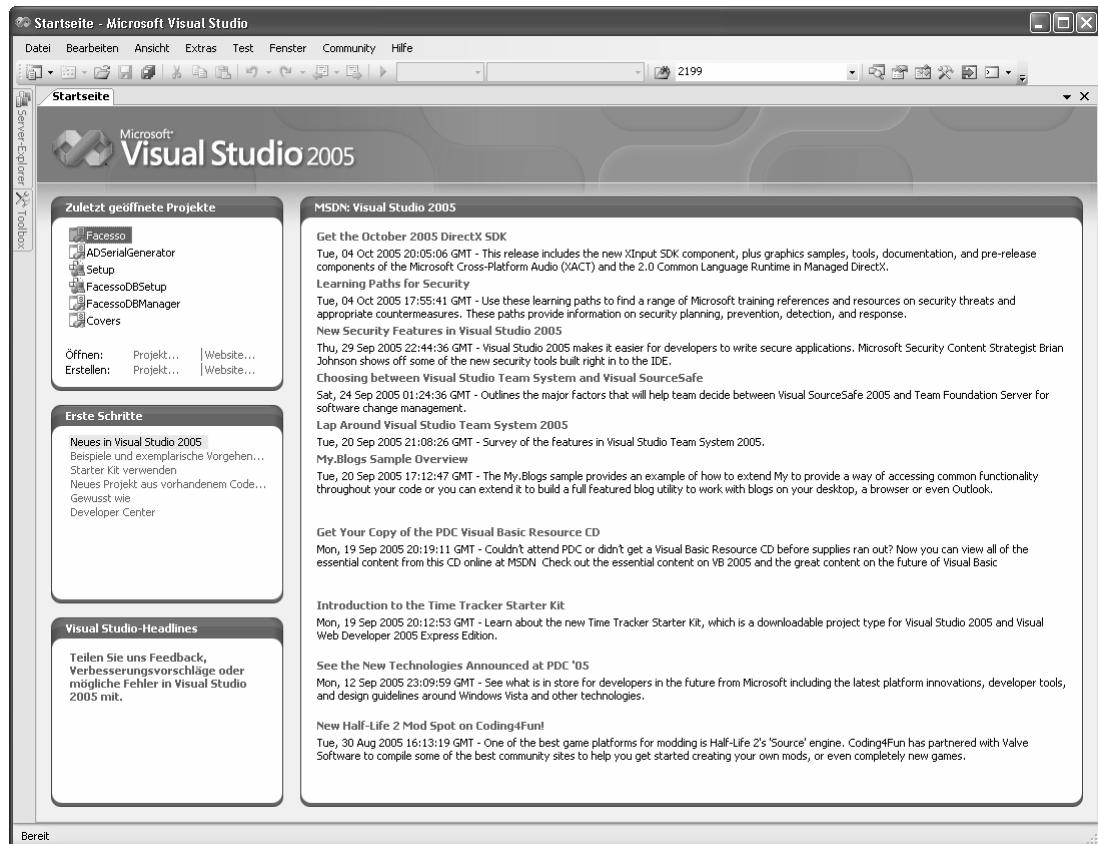
- 
- 13 Die Startseite – der erste Ausgangspunkt für Ihre Entwicklungen**
  - 16 Die IDE auf einen Blick**
  - 20 Die wichtigsten Toolfenster**
  - 34 Die wichtigsten Tastenkombinationen auf einen Blick**
- 

Die IDE – die integrierte Entwicklungsumgebung – ist das »Modul« in Visual Studio, mit dem Sie ganz sicher die meiste Zeit verbringen werden. Je besser Sie die wichtigsten Funktionen und Eigenarten dieses unglaublich mächtigen Werkzeugs kennen, desto mehr Zeit werden Sie bei der täglichen Entwicklung Ihrer Softwareprojekte sparen.

Eben weil die Visual Studio-IDE gespickt ist mit Funktionen, Toolfenstern, Werkzeugen und anderen Hilfsmitteln 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.

## **Die Startseite – der erste Ausgangspunkt für Ihre Entwicklungen**

Jedes Mal, wenn Sie Visual Studio 2005 starten, begrüßt Sie die IDE mit der Startseite, etwa wie in Abbildung 2.1 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 2005, SQL Server 2005 und das Framework 2.0.



**Abbildung 2.1:** Nach dem Start von Visual Studio 2005 begrüßt Sie die Startseite, auf der Sie die zuletzt bearbeiteten Projekte öffnen können, und die Ihnen aktuelle Infos rund um Visual Studio liefert

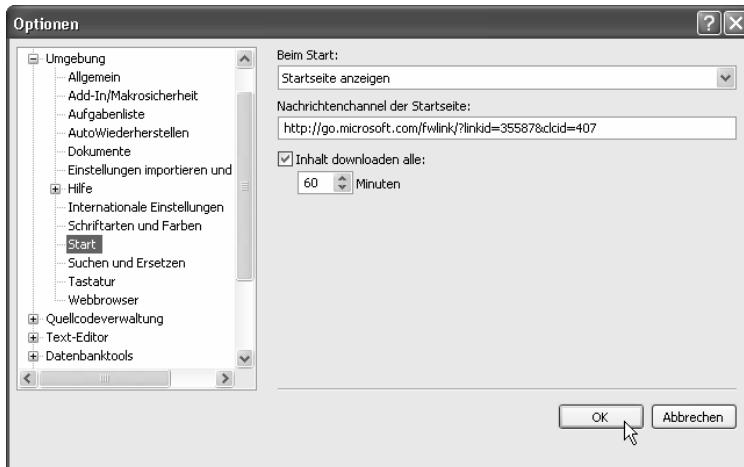
In der linken, oberen Ecke der Startseite finden Sie die letzten Projekte, die Sie bearbeitet haben. Ein einfacher Mausklick genügt, um das entsprechende Projekt zu öffnen. Sie können auch direkt von hier aus neue Projekte anlegen oder Projekte öffnen, die sich nicht in der Liste der zuletzt verwendeten 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 2005, SQL Server 2005 und dem Framework 2.0 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*.<sup>1</sup>



**Abbildung 2.2:** 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\8.0\ProjectMRUList* die Liste der Projektdateien einsehen 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. **Und ganz 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 löschenen Eintrag zu übertragen und dann den letzten Eintrag der Liste zu löschen.

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.

<sup>1</sup> Im Bedarfsfall müssen Sie bei einigen Visual Studio-Versionen erst im Dialog das Kontrollkästchen *Alle Einstellungen anzeigen*, um Zugriff auf alle Optionen nehmen zu können.

Die Liste der zuletzt geöffneten Dateien können Sie übrigens ebenfalls bearbeiten. Verfahren Sie dazu wie oben beschrieben, verwenden Sie jedoch statt des genannten Schlüssels den Schlüssel `HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\8.0\FileMRUList`.

## 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, und denken Sie darüber nach, welche der IDE-Elemente Sie am häufigsten benötigen, und bringen Sie diese gemäß der Erklärungen der nächsten Abschnitte in den Vordergrund.

---

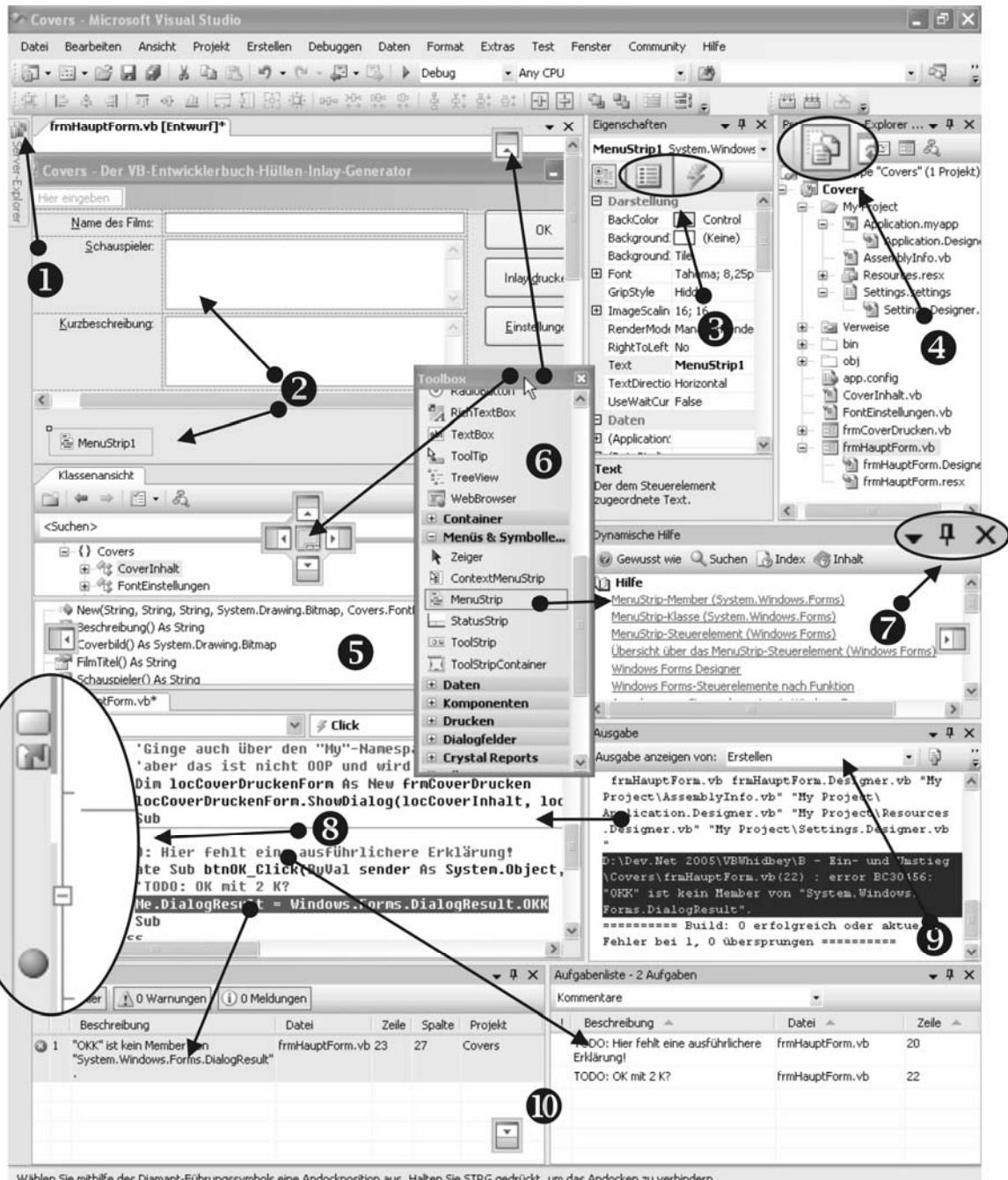
**TIPP:** Um sich mit allen Elementen in Visual Studio .NET vertraut machen zu können, benötigen Sie ein Projekt zum »Spielen«. Da Sie das in den folgenden Beispielen verwendete Projekt noch nie verwendet haben, steht es natürlich noch nicht in der Liste.

---

- 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 2005 - Entwicklerbuch\B - IDE\02 - Ereignisse`.
- 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 3 liefert mehr Informationen dazu).

Die meisten Grafikkarten, die heutzutage in modernen Computern oder Notebooks verbaut werden, erlauben ohnehin den Anschluss eines zweiten Monitors und die entsprechende Erweiterung des Desktop 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 zu viele zu kleine Toolfenster stark nachlassen! Außerdem verlieren Sie durch das ständige Navigieren zwischen den Elementen zu viel Zeit.



Wählen Sie mithilfe des Diamant-Führungssymbols eine Andockposition aus. Halten Sie STRG gedrückt, um das Andocken zu verhindern.

**Abbildung 2.3:** Die wichtigsten IDE-Elemente auf einen Blick. In den nachfolgenden Abschnitten finden Sie beschrieben, zu welchem Zweck sie dienen und wie sie untereinander interagieren und sich beeinflussen können

# Genereller Umgang mit Fenstern in der Entwicklungsumgebung

Dass die Benutzeroberfläche der Entwicklungsumgebung unwahrscheinlich 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 2.3 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 ③ 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 MDI<sup>2</sup>-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 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 ⑥) in Abbildung 2.3.

- 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 Positions hilfen 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 Positions hilfen, die an den vier Seiten der IDE zu finden sind. In der Abbildung erkennen Sie eine dieser Positions hilfen, 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 Positions hilfe 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).

## 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 Umgebungsbereich verlassen, klicken Sie auf das Heftzwecken-Symbol, wie Sie es in Abbildung 2.3 mit ⑦ bezeichnet erkennen können.

---

<sup>2</sup> 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.

- 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*.

## Wechseln zwischen aktiven Dokumentfenstern mit der Tastatur

Übrigens: Genau so wie Sie in Windows selbst mit der Tastenkombination **Alt+Tabulator** zwischen den verschiedenen laufenden Anwendungen wechseln können, haben Sie die Möglichkeit, in Visual Studio mit **Strg+Tabulator** zwischen den aktiven Dokumenten zu wechseln. Visual Studio blendet dabei als Orientierungshilfe ein Fenster ein, wie Sie es auch in der folgenden Abbildung sehen können.

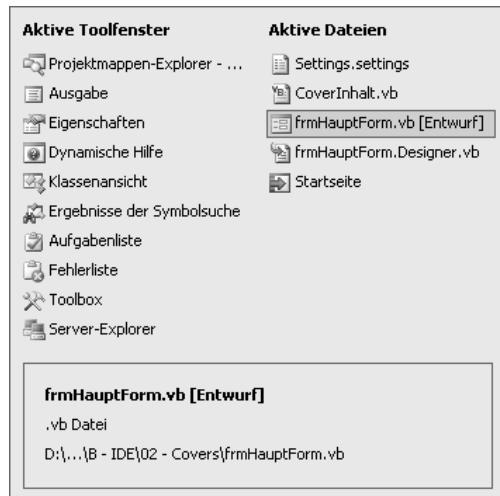


Abbildung 2.4: Mit **Strg+Tabulator** können Sie zwischen den geöffneten Dokumentfenstern wechseln

## 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 Sie mit Abstand am häufigsten benötigen, finden Sie in den folgenden Abschnitten beschrieben.

# Der Projektmappen-Explorer

Mithilfe des Projektmappen-Explorers navigieren Sie in Ihrem Projekt (in Abbildung 2.3 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.

## Projektmappen und Projekte

Eine Projektmappe kann verschiedene Projekte enthalten. Dabei ist es egal, ob diese Projekte sich gegenseitig benötigen und aufrufen oder ob sie von einander 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, Plattformeigenarten 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.
- ... das Projekt/die Projektmappe umzubenennen.
- ... die Eigenschaften für die Projektmappe abzurufen.

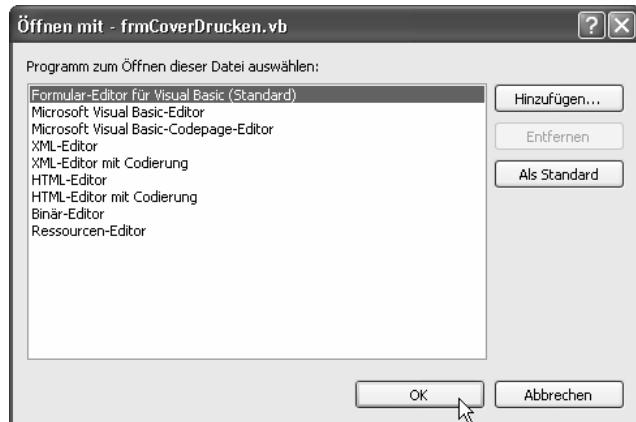
## Projektdateitypen

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

| Symbol | Typ                       | Aufgabe  |
|--------|---------------------------|--|
|        | Projektmappe (Solution)   | Beinhaltet die Zusammenstellung der Projektmappendateien 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 <code>System.Windows.Forms</code> 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 <code>AssemblyInfo</code> dar.  |

**Tabelle 2.1:** 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 2.5:** Mit diesem Dialog, den Sie durch *Öffnen mit* 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 2.3 unter ④ mit dem Pfeil gekennzeichneten Symbol können Sie alle Dateien eines Projektes ein- und ausblenden.

## Weitere Funktionen des Projektmappen-Explorers

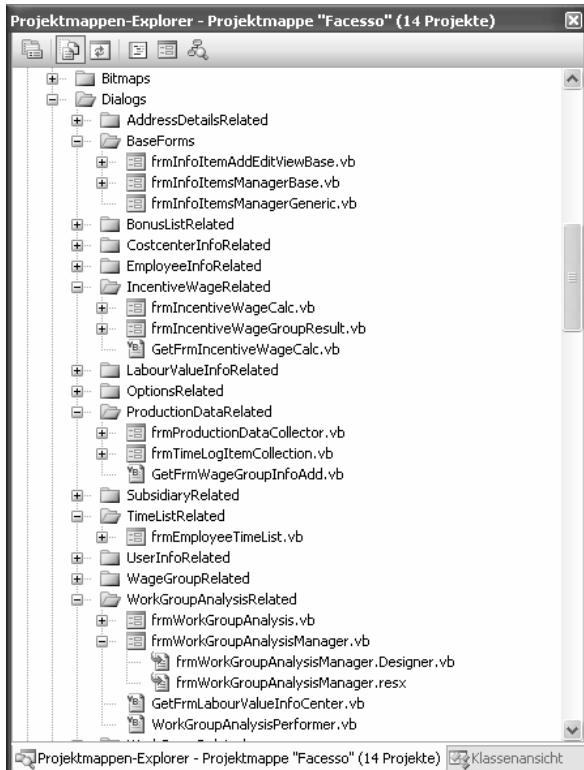
Die folgende Tabelle erklärt kurz die weiteren Funktionen, die sich durch die Symbole des Projektmappen-Explorers aufrufen lassen:

| Symbol | Aufgabe  |
|--------|--|
|        | <p>Stellt die Eigenschaftenseite einer Projektmappe oder eines Projektes dar.</p> <p><b>HINWEIS:</b> Anders als 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><b>TIPP:</b> 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 Entwicklerteam ist dieser Bug bekannt – offensichtlich aus Kompatibilitätsgründen zur Visual Studio Extensibility kann er aber nicht ohne weiteres behoben werden. Der IntelliLink <i>B0201</i> 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 Formulklassle eingebettet. Die vollständige Anzeige erlaubt ebenfalls einen Blick in die von einem Projekt referenzierten Assemblies (die Sie benötigen, wenn Sie bestimmte Funktionalitäten des Framework oder eines anderen Projektes benötigen), die Sie anderenfalls nicht sehen würden.</p>  |
|        | <p>Aktualisiert die Projektdateienliste im Projektmappenexplorer.</p>  |
|        | <p>Ruft den Codeeditor auf und stellt die zuvor im Projektmappen-Explorer ausgewählte Codedatei dort dar.</p>  |
|        | <p>Ruft den Formular-Designer auf und stellt die zuvor im Projektmappen-Explorer ausgewählte Formulardatei dort dar.</p>   |
|        | <p>Falls Sie eine Klassendatei im Projektmappen-Explorer ausgewählt haben, gelangen Sie mit dieser Funktion zum visuellen Klassendesigner.</p>   |

**Tabelle 2.2:** 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 2.6:** 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 6 erklärt mehr zum Thema partielle Klassen und dem Modifizierer Partial.

---

Abbildung 2.6 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 mit Drag&Drop in Verbindung mit der rechten Maustaste und dem Kontextmenü 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 »erwischen«. Das gilt unter Umständen für andere Projektdateien wie beispielsweise typisierte Data-Sets ebenfalls.

---

## Das Eigenschaftenfenster

Mithilfe des Eigenschaftenfensters – in Abbildung 2.3 unter ③ 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

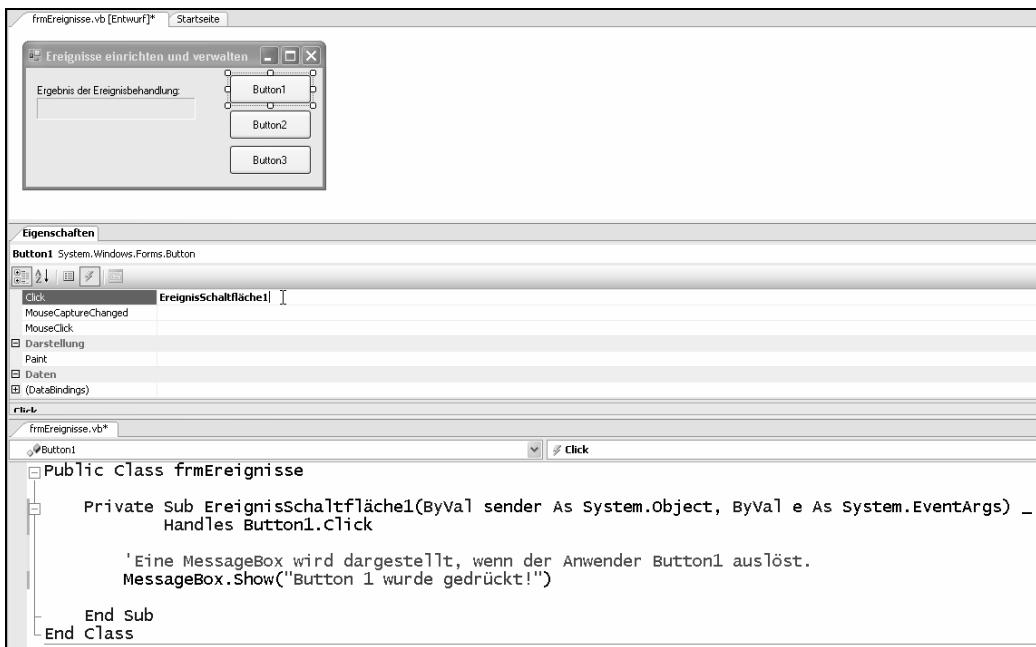
Das Eigenschaftenfenster verwenden Sie ebenfalls, um die einem mit dem Designer editierbaren Objekt zugeordneten Ereignisse visuell zu bearbeiten. Sie schalten mit den Symbolen, die durch den von Punkt ③ abgehenden Pfeil in Abbildung 2.3 markiert sind, zwischen den Ereignissen und den Eigenschaften eines Objektes um.

Ereignisbehandlungsroutine 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 Ereignisbehandlungsroutine 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 Ereignisbehandlungsroutine 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.
- 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 an-

schließend **Eingabe**. Die IDE fügt anschließend automatisch einen entsprechenden Stub in die Codedatei mit dem von Ihnen vorgegebenen Namen ein (siehe Abbildung 2.7).



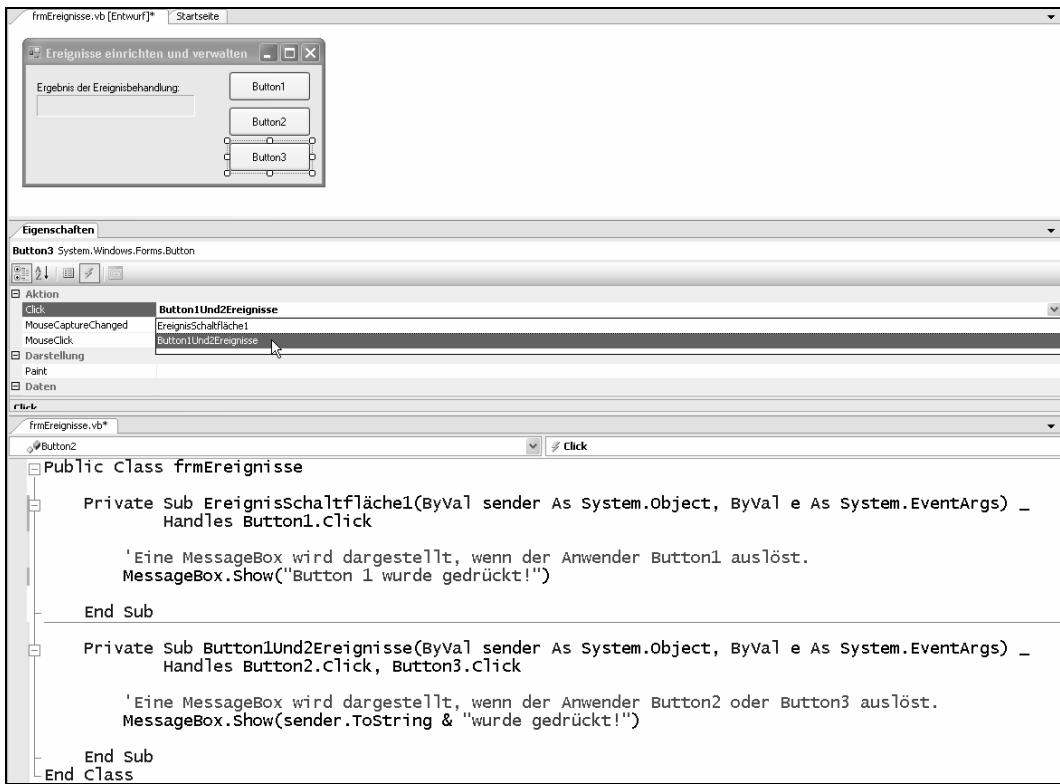
**Abbildung 2.7:** Der Anwender markiert die Schaltfläche mit dem Namen *Button1* (oberes Drittel) und gibt den Namen der Ereignisbehandlungsroutine (mittleres Drittel) ein. Die Behandlungsroutine trägt dann diesen Namen (*EreignisSchaltfläche1*) und behandelt (*Handles*) das entsprechende Ereignis (*Click*) des Objektes (*Button1*).

---

**TIPP:** Falls Sie sich wundern, wieso das Eigenschaftenfenster in Abbildung 2.7 als Dokument erscheint. Im Abschnitt »Toolfenster« auf Seite 18 erfahren Sie, wie Sie Toolfenster als Dokument einer Dokumentenregisterkartengruppe hinzufügen können.

---

- Zum Zuweisen der gleichen Ereignisbehandlungsroutinen 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 2.11).



**Abbildung 2.8:** Der Anwender markiert die Schaltfläche mit dem Namen *Button3* (oberes Drittel) und wählt den Namen der Ereignisbehandlungsroutine (mittleres Drittel) aus. Die schon 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.

Sie sehen die Fehlerliste in Abbildung 2.3 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

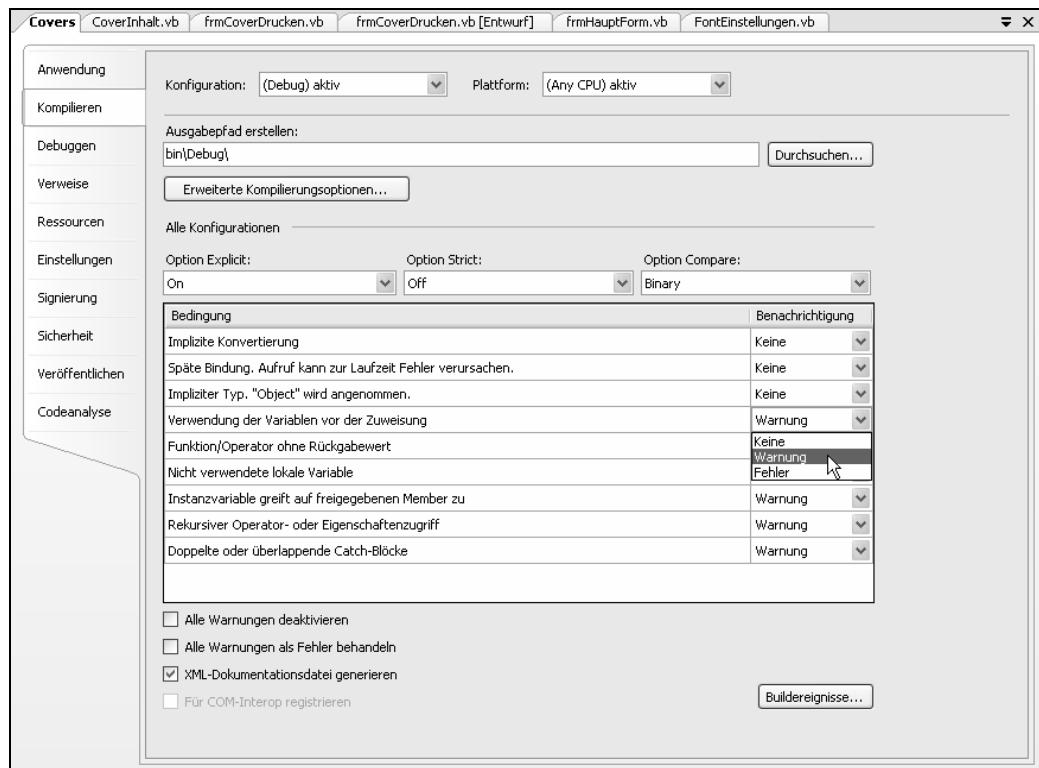
Es gibt übrigens drei verschiedene Meldungstypen in der Fehlerliste, deren Anzeige Sie nach Belieben konfigurieren können:

- **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.

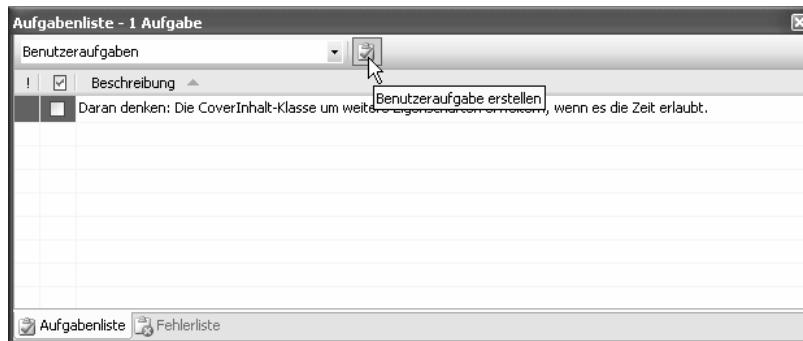


**Abbildung 2.9:** Mit dem Register *Kompilieren* stellen Sie das Fehlermeldungsverhalten eines Projektes ein

Um die Eigenschaften eines Projektes zu öffnen, rufen Sie das Kontextmenüs des entsprechenden Projektes mit der rechten Maustaste auf und wählen anschließend *Eigenschaften*. Sie sehen anschließend einen Dialog, wie er etwa auch in Abbildung 2.9 zu sehen ist. Dort können Sie dann die verschiedenen Meldungstypen individuell konfigurieren.

## Die Aufgabenliste

Die Aufgabenliste verhält sich in Visual Basic 2005 ähnlich wie die Fehlerliste, nur mit dem Unterschied, dass dort Punkte erscheinen, die Sie selber eintragen.

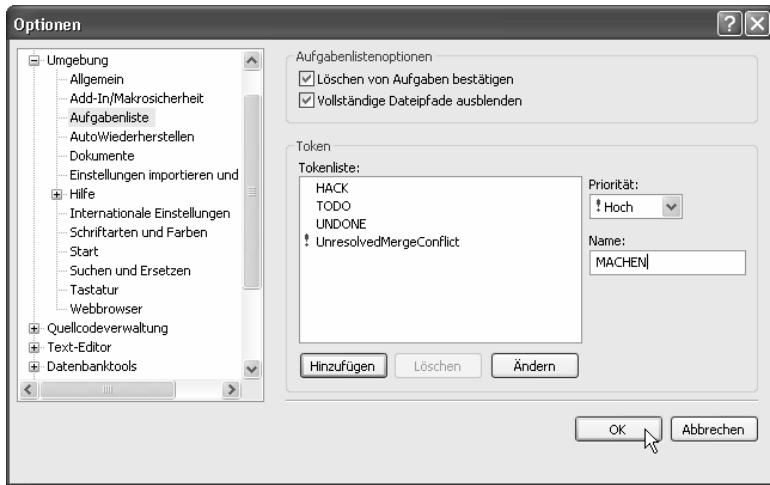


**Abbildung 2.10:** Die Aufgabenliste zeigt in Abhängigkeit der Auswahl in der Aufklappliste entweder benutzerdefinierte, manuell hinzugefügte Aufgaben oder durch Kommentare im Code eingefügte

Meldungen, die in der Aufgabenliste erscheinen, können durch zwei verschiedene Methoden dort hineingelangen.

- Durch das manuelle Hinzufügen einer Aufgabe zur Aufgabenliste. Dazu wählen Sie aus der Aufklappliste den Punkt Benutzerobergaben, und klicken Sie anschließend auf das rechts daneben stehende Symbol, etwa wie in Abbildung 2.10 zu sehen.
- Durch Einfügen von Kommentaren innerhalb des Listings. In Abbildung 2.3 sehen Sie unter Punkt ⑧ einen Pfeil, der vom Codefenster in das Aufgabenfenster reicht. Sobald Sie einen Kommentar im Codelisting, 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 Aufklappliste 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 Umgebung/Aufgabenliste konfigurieren Sie vorhandene Schlüsselwörter (so genannte »Tokens«) oder richten weitere ein. Abbildung 2.11 zeigt, wie es geht. Neben der Einrichtung der Schlüsselworte können Sie auch deren Priorität festlegen. Dazu wählen Sie aus der Aufklappliste 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 2.11:** 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 2005 erfüllt zwei Funktionen. Zum einen gibt es während des Kompilierens 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 Ausgabentypen im Ausgabefenster dargestellt werden, bestimmen Sie mit der Aufklappliste *Ausgabe anzeigen von*.

### Ausgabe anzeigen von Erstellen

Abbildung 2.12 zeigt Ihnen beispielhaft die Ausgabe eines Projekt-Builds, in dem (mit Absicht natürlich) zwei Compiler-Fehler aufgetreten sind.

The screenshot shows the 'Ausgabe' (Output) window from Visual Studio. The title bar says 'Ausgabe' and 'Ausgabe anzeigen von: Erstellen'. The main area displays the build log:

```

-----
Erstellen gestartet: Projekt: Covers, Konfiguration: Debug Any CPU -----
Der Buildvorgang wurde um 25.10.2005 14:33:20 gestartet.
CoreResGen-Ziel:
Es gibt keine Ressourcen, die im Hinblick auf ihre Quelldateien veraltet sind.
Die Ressourcengenerierung wird übersprungen.
CoreCompile-Ziel:
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Wbc.exe /noconfig /imports:
Microsoft.VisualBasic,System,System.Collections,System.Collections.Generic,System.
Data,System.Drawing,System.Diagnostics,System.Windows.Forms /nowarn:42016,41999,42017
,42018,42019,42032,42036,42020,42021 /rootnamespace:Covers /doc:obj\Debug\
Covers.xml /define:""CONFIG=""Debug""",DEBUG=-1,TRACE=-1,_MyType=""WindowsForms"",
PLATFORM=""AnyCPU"" /reference:C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.
Data.dll,C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Deployment.dll,C:\
WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.dll,C:\WINDOWS\Microsoft.NET\
Framework\v2.0.50727\System.Drawing.dll,C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727
\System.Windows.Forms.dll,C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Xml.
dll /main:Covers.My.Application /debug /debug:full /out:obj\Debug\Covers.exe /
resource:obj\Debug\Covers.frmCoverDrucken.resources /resource:obj\Debug\Covers.
frmHauptForm.resources /resource:obj\Debug\Covers.Resources.resources /target:winexe
CoverInhalt.vb FontEinstellungen.vb frmCoverDrucken.Designer.vb frmCoverDrucken.vb
frmHauptForm.vb frmHauptForm.Designer.vb "My Project\AssemblyInfo.vb" "My Project\
Application.Designer.vb" "My Project\Resources.Designer.vb" "My Project\Settings.
Designer.vb"
D:\Dev.Net 2005\VBWhidbey\B - Ein- und Umstieg\Covers\frmCoverDrucken.vb(120) : error
BC30451: Der Name "locOffset_X" wurde nicht deklariert.
D:\Dev.Net 2005\VBWhidbey\B - Ein- und Umstieg\Covers\frmCoverDrucken.vb(121) : error
BC30451: Der Name "locOffset_Y" wurde nicht deklariert.

Fehler beim Buildvorgang.

Vergangene Zeit 00:00:00.07
===== Build: 0 erfolgreich oder aktuell, Fehler bei 1, 0 Übersprungen =====

```

**Abbildung 2.12:** Dieses Beispiel zeigt die Build-Protokollierung (*Ausgabe von Erstellen*) eines Visual Basic-Projektes mit normaler Ausführlichkeit

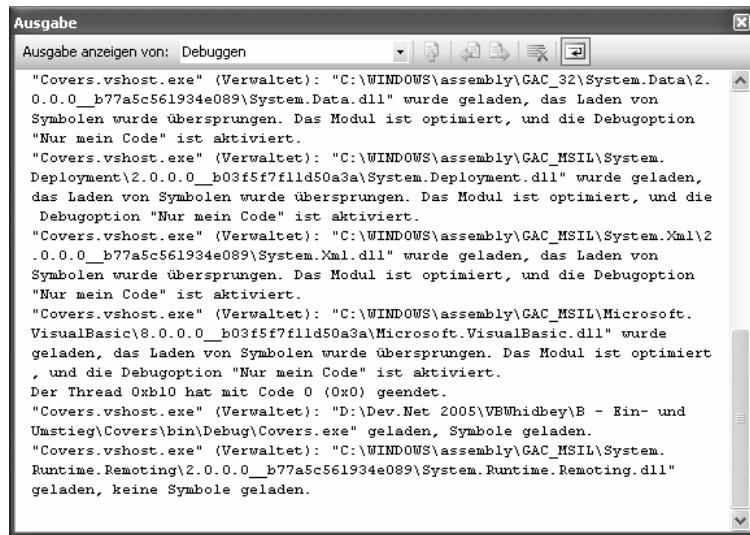
Mit den Symbolen neben der Aufklappliste am oberen Fensterrand haben Sie die Möglichkeit, bestimmte Funktionalitäten abzurufen, die Ihnen das Ausgabefenster zur Verfügung stellt. Die folgende Tabelle zeigt, welche Funktionen es gibt:

| 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 entsprechenden Quellcodetextstelle. Abbildung 2.3 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.   |

**Tabelle 2.3:** 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 2.13 zeigt eine typische Ausgabe einer Windows Forms-Anwendung nach dem Start.



The screenshot shows the 'Ausgabe' (Output) window in Visual Studio. The title bar says 'Ausgabe'. The dropdown menu 'Ausgabe anzeigen von:' is set to 'Debuggen'. The window contains several lines of text output from the application 'Covers.vshost.exe'. The output includes messages about assembly loading, symbol resolution, and thread termination. It also shows the path to the application's executable ('D:\Dev.Net 2005\VBWhidbey\B - Ein- und Umstieg\Covers\bin\Debug\Covers.exe').

```
"Covers.vshost.exe" (Verwaltet): "C:\WINDOWS\assembly\GAC_32\System.Data\2.0.0.0__b77a5c561934e089\System.Data.dll" wurde geladen, das Laden von Symbolen wurde übersprungen. Das Modul ist optimiert, und die Debugoption "Nur mein Code" ist aktiviert.  
"Covers.vshost.exe" (Verwaltet): "C:\WINDOWS\assembly\GAC_MSIL\System.Deployment\2.0.0.0__b03f5f7f11d50a3a\System.Deployment.dll" wurde geladen, das Laden von Symbolen wurde übersprungen. Das Modul ist optimiert, und die Debugoption "Nur mein Code" ist aktiviert.  
"Covers.vshost.exe" (Verwaltet): "C:\WINDOWS\assembly\GAC_MSIL\System.Xml\2.0.0.0__b77a5c561934e089\System.Xml.dll" wurde geladen, das Laden von Symbolen wurde übersprungen. Das Modul ist optimiert, und die Debugoption "Nur mein Code" ist aktiviert.  
"Covers.vshost.exe" (Verwaltet): "C:\WINDOWS\assembly\GAC_MSIL\Microsoft.VisualBasic\8.0.0.0__b03f5f7f11d50a3a\Microsoft.VisualBasic.dll" wurde geladen, das Laden von Symbolen wurde übersprungen. Das Modul ist optimiert, und die Debugoption "Nur mein Code" ist aktiviert.  
Der Thread 0xb10 hat mit Code 0 (0x0) geendet.  
"Covers.vshost.exe" (Verwaltet): "D:\Dev.Net 2005\VBWhidbey\B - Ein- und Umstieg\Covers\bin\Debug\Covers.exe" geladen, Symbole geladen.  
"Covers.vshost.exe" (Verwaltet): "C:\WINDOWS\assembly\GAC_MSIL\System.Runtime.Remoting\2.0.0.0__b77a5c561934e089\System.Runtime.Remoting.dll" geladen, keine Symbole geladen.
```

**Abbildung 2.13:** Dieses Beispiel Meldungen im Ausgabefenster (*Ausgabe 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 2.3 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.

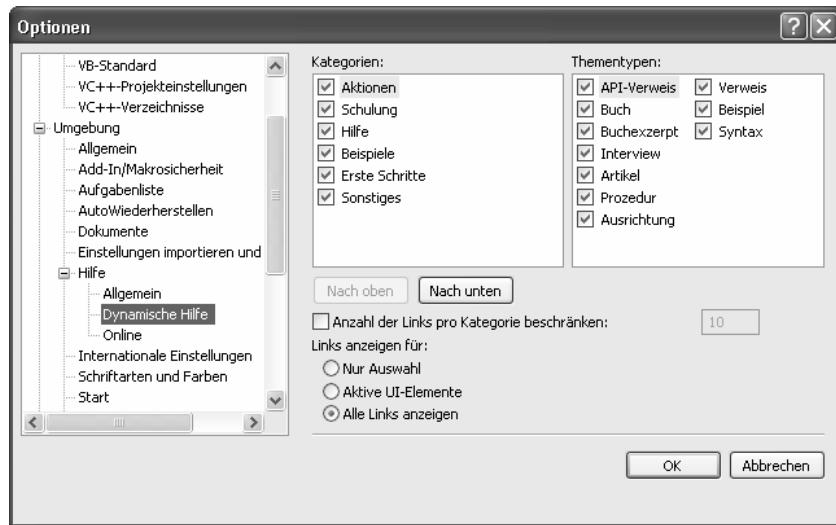
---

### 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 2.14:** 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 2.3 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 2.3 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.

## Die wichtigsten Tastenkombinationen auf einen Blick

| Kurzbeschreibung                 | Tastenkomb.                      | Beschreibung  | Befehlsname                         |
|----------------------------------|----------------------------------|---|-------------------------------------|
| Cursor zum nächsten Element      | <b>F8</b>                        | Verschiebt den Cursor zum nächsten Element – beispielsweise einer Aufgabe oder einem Fehler im Aufgabenfenster.                                   | Bearbeiten.GehezunächsterInstanz    |
| Cursor zum vorherigen Element    | <b>Umschalt+F8</b>               | Verschiebt den Cursor zum vorherigen Element (Fehler, Aufgabe im Aufgabenfenster).  | Bearbeiten._GehezuvorherigerInstanz |
| Cursor zur Definition            | <b>Umschalt+F12</b>              | Verschiebt den Cursor zur Definition des Elementes, das sich derzeit unterhalb des Cursors befindet.  | Bearbeiten.GehezuVerweis            |
| Rückgängig                       | <b>Strg+Z oder Alt+Rücktaste</b> | Macht die letzte Aktion rückgängig.   | Bearbeiten.Rückgängig               |
| Rückgängig rückgängig machen     | <b>Strg+Y</b>                    | Stellt die zuvor rückgängig gemachte Aktion wieder her.   | Bearbeiten.Wiederholen              |
| Alles speichern                  | <b>Strg+Umschalt+S</b>           | Speichert alle Dateien, an denen seit dem letzten Speichern Änderungen vorgenommen wurden.  | Datei.AllesSpeichern                |
| Zur letzten Änderung springen    | <b>Strg + –</b>                  | Setzt den Cursor auf die Position im Code, an der Sie die letzte Änderung vorgenommen haben. Sie können nicht nur einen Schritt zurücknavigieren. | Ansicht.Rückwärtsnavigieren         |
| Zur vorherigen Änderung springen | <b>Strg+Umschalt + –</b>         | Nachdem Sie rückwärts navigiert haben (siehe vorherigen Eintrag), erreichen Sie damit wieder die ursprüngliche Position.                          | Ansicht.Vorwärtsnavigieren          |
| Suchdialog öffnen                | <b>Strg+F</b>                    | Öffnet den Suchen-Dialog.   | Bearbeiten.Suchen                   |
| Inkrementelles Suchen            | <b>Strg+I</b>                    | Startet das inkrementelle Suchen.   | Bearbeiten._InkrementelleSuche      |
| Zur Zeile mit Nummer springen    | <b>Strg+G</b>                    | Öffnet den Dialog »Gehezu Zeilennummer« und erlaubt den Sprung zur Zeile mit angegebener Nummer.  | Bearbeiten.GeheZu                   |



| Kurzbeschreibung  | Tastenkomb.           | Beschreibung   | Befehlsname                                   |
|---|-----------------------|--|---|
| Automatischen Zeilenumbruch ein- und ausschalten          | <b>Strg+R, Strg+R</b> | Schaltet den automatischen Zeilenumbruch ein und aus. Drücken Sie beide Tastenkombinationen dazu nacheinander.   | Bearbeiten. _ Zeilenumbruchumschalten         |
| Lesezeichen einfügen/entfernen                            | <b>Strg+K, Strg+K</b> | Schaltet ein Lesezeichen in einer Zeile ein bzw. wieder aus, wenn bereits eines gesetzt war.   | Bearbeiten. _ Lesezeichenumschalten           |
| Zum nächsten Lesezeichen                                  | <b>Strg+K, Strg+N</b> | Setzt den Cursor in die Zeile, in der sich das nächste Lesezeichen befindet.   | Bearbeiten. _ NächstesLesezeichen             |
| Zum vorherigen Lesezeichen                                | <b>Strg+K, Strg+P</b> | Setzt den Cursor in die Zeile, in der sich das vorherige Lesezeichen befindet.   | Bearbeiten. _ VorherigesLesezeichen           |
| Aufgabeneintrag für aktuelle Codezeile hinzufügen/löschen | <b>Strg+K, Strg+H</b> | 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 2.4:** Die wichtigsten Tastaturkommandos. Bitte beachten Sie, dass die Kommandos zur Tastaturanpassung in VB-Manier mit dem Unterstrich getrennt sind und eigentlich eine Zeile bilden



# **3 Formular-Designer und Codeeditor enthüllt**

---

- 38 Das Fallbeispiel – Der DVD-Hüllen-Generator »Covers«**
  - 43 Gestalten von Formularen mit dem Windows Forms-Designer**
  - 71 Der Codeeditor**
  - 79 Smarttags im Editor von Visual Basic**
  - 80 Erzwungene Typsicherheit (Option Strict) projektweit einstellen**
  - 101 Weitere Funktionen des Codeeditors**
- 

Wenn es darum geht, Windows Forms-Anwendungen, die auch unter dem Modenamen »Smartclients«<sup>1</sup> gehandelt werden, zu entwickeln – und darauf legt dieses Buch erklärterweise 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 auf schnellste Weise die Feinheiten und Möglichkeiten der Werkzeuge kennen, und Sie sind viel besser für die Praxis gerüstet.

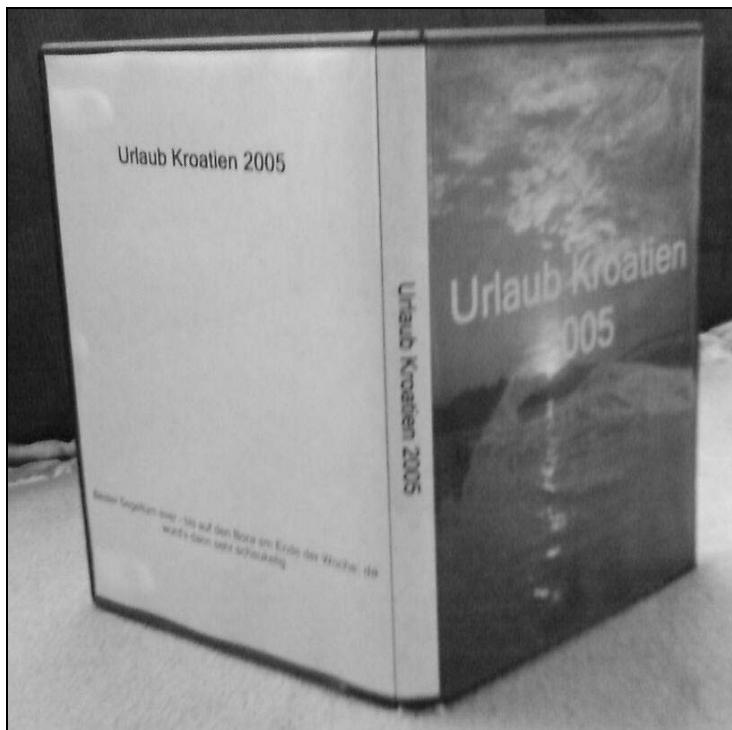
Aus diesem Grund habe ich mich dazu entschieden, die Vorstellung der Möglichkeiten von Formular-Designer und Codeeditor an einem konkreten Beispiel durchzuexerzieren. Das Beispiel geht sogar noch ein wenig über die thematischen Kapitelgrenzen hinaus und liefert Ihnen schon jetzt die eine oder andere 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 viel Repertoire zu verhelfen, dass Sie ohne das Buch zunächst komplett auswendig lernen zu müssen, schon Ihre ersten kleinen Projekte realisieren können.

---

<sup>1</sup> Die ► Kapitel ab Teil G beschäftigen sich mit diesem Thema im Detail.

# 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, untermtitelt 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.



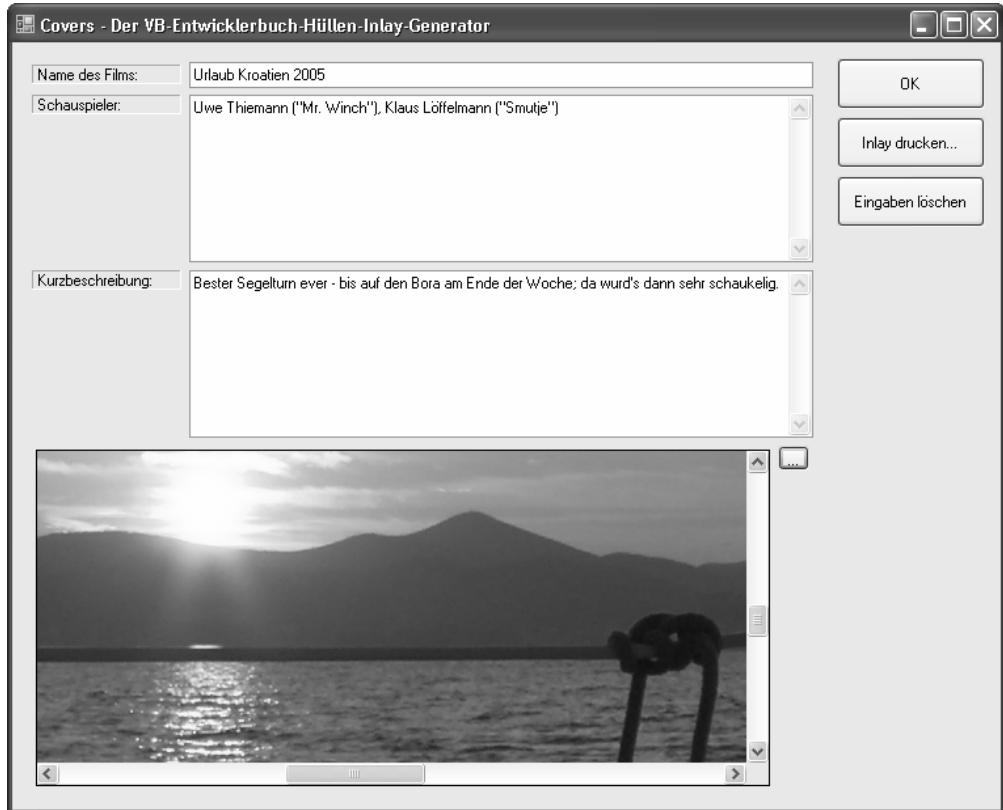
**Abbildung 3.1:** Ein ausgedrucktes Inlay schneiden Sie einfach entlang der Markierungslinien aus und stecken es in das Plastikcover – hier ein Beispiel

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.

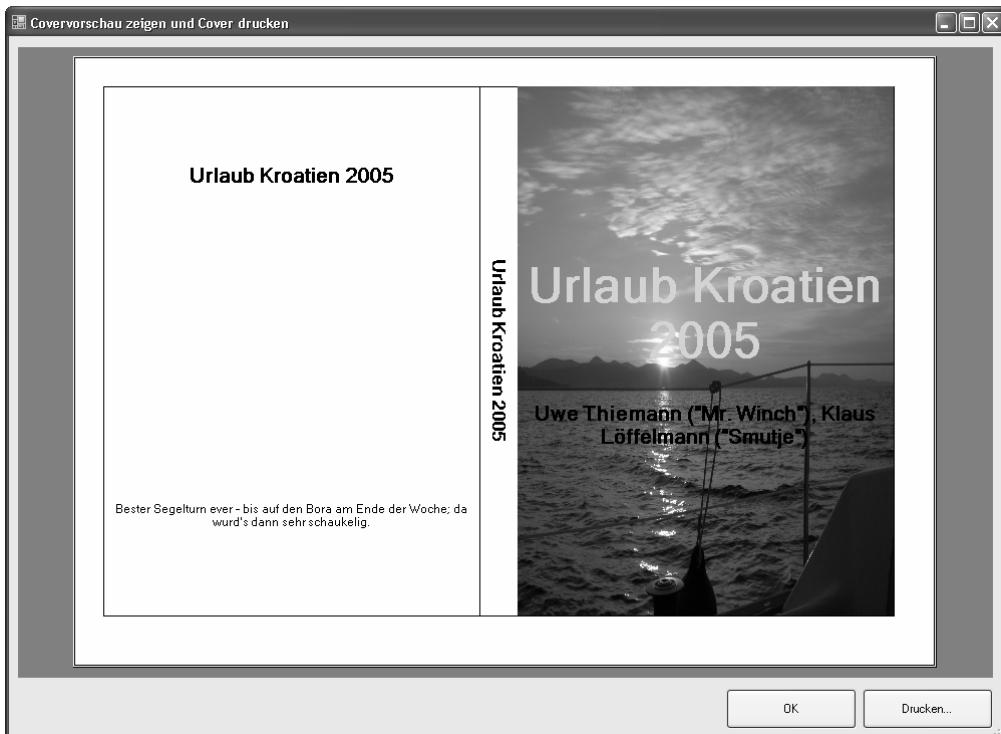
## Das »Pflichtenheft« von Covers

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 3.1 zeigt, wie so was in etwa im Ergebnis ausschauen kann; die beiden folgenden Abbildungen vermitteln Ihnen einen Eindruck von der Bedienung des Programms.



**Abbildung 3.2:** Dazu soll »Covers« in der Lage sein: Aus einer simplen Erfassung der Filmdaten druckt es buchstäblich auf Knopfdruck ...



**Abbildung 3.3:** ... 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 2005 und Visual Basic 2005 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 2005 - Entwicklerbuch\B - IDE\03 - 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 die *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 3.4 hilft Ihnen bei der Orientierung.



**Abbildung 3.4:** 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 3.5 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 *Projektmappenname* den Namen der Projektmappe ein.

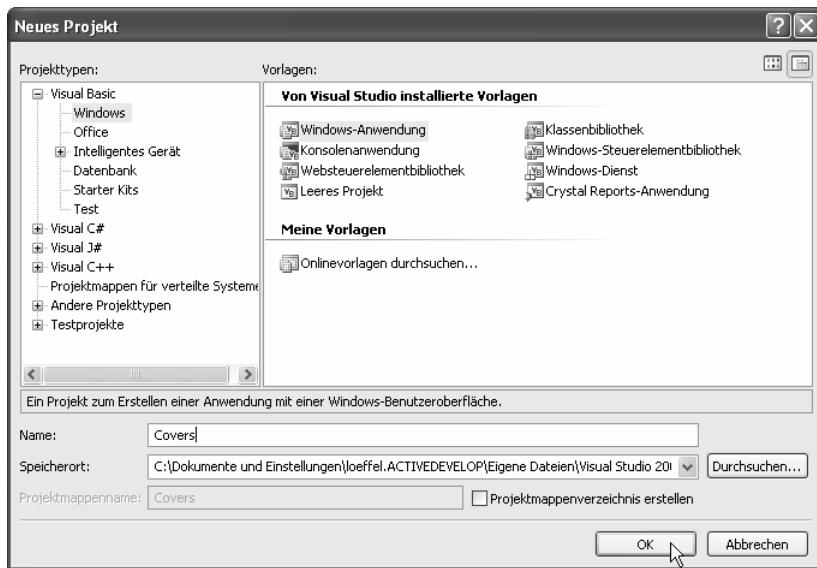
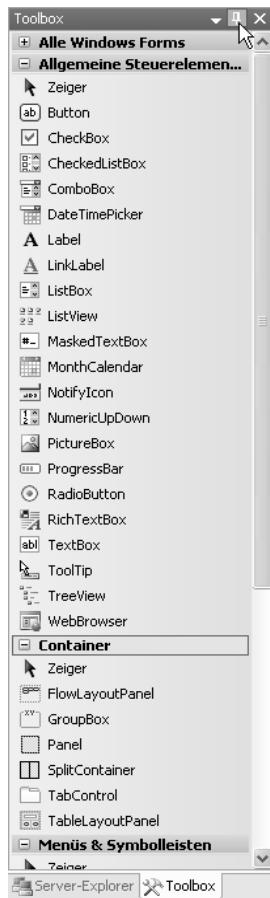


Abbildung 3.5: Mit diesem Dialog richten Sie ein neue Windows-Anwendung ein

# 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.

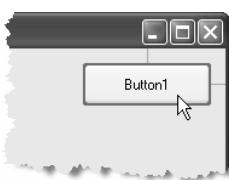
Fahren Sie mit dem Mauszeiger auf das Toolbox-Symbol, worauf sich die Toolbox in den Vordergrund schiebt, und klicken Sie auf das Pin-Symbol im Fenstertitel (links im Bild zu sehen). Dadurch bleibt die Toolbox, wo sie ist, und fährt nicht wieder automatisch in den Hintergrund, wenn der Mauszeiger sie 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 3.2 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 3.2 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 anpassen können.
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 3.6), die die Schaltfläche auf Abstand halten.



**Abbildung 3.6:** 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 grundsätzliche 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 eingehalten werden soll, wenn ein beinhaltendes Steuerelement am Rand eines beinhaltenden Steuerelements angeordnet wird.

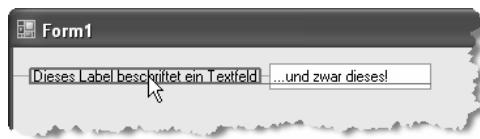
---

**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.<sup>2</sup>

---

**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 3.7 zu sehen.

---



**Abbildung 3.7:** Gerade beim Beschriften von `Textbox`-Steuerelementen mit `Label`-Steuerelementen helfen Ihnen Ausrichtungslinien, da sich die Steuerelemente auch an den Textbasislinien einschnappen

---

<sup>2</sup> 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 [B0301](#).

**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 2005 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 2005 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 Programmteil 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 Fachterminus 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 die 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 »zurechtzuppeeln« 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 bestimmt

te Funktionen an, die Sie im Abschnitt »Funktionen zum Layouten von Steuerelementen im Designer« ab Seite 68 beschrieben finden.

### Selektieren mehrerer Steuerelemente und Bestimmen des Referenzsteuerelements

Um mehrere Steuerelemente im Formular gleichzeitig zu selektieren, halten Sie entweder die **Umschalt**-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 3.4 ab Seite 68 beschrieben finden, ist das, welches als erstes selektiert wurde, wenn Sie die Steuerelemente mit gedrückter **Umschalt**-Taste selektieren. Möchten Sie das Referenzsteuerelement nach der Selektion der Steuerelemente ändern, klicken Sie es einfach an (dabei halten Sie die **Umschalt**-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.

---

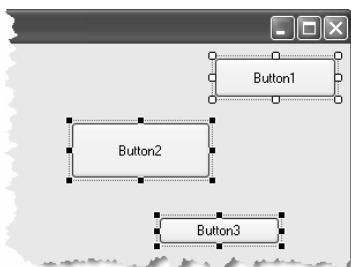
**TIPP:** Das Referenzsteuerelement erkennen Sie übrigens daran, dass es mit weißen Anfasspunkten versehen ist, während alle anderen markierten Steuerelemente über schwarze Anfasspunkte verfügen.

---

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 auf einen freien Bereich neben einer schon eingeblendetem Symbolleiste, und wählen Sie aus dem Kontextmenü, das jetzt erscheint, den Eintrag *Layout*.

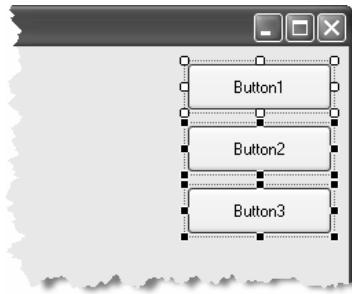
2. Wählen Sie aus der Symbolleiste die Funktion *Größe angleichen* – die Tabelle im Abschnitt »Funktionen zum Layouten von Steuerelementen im Designer« ab Seite 68 hilft Ihnen, das richtige Symbol dafür zu finden.



**Abbildung 3.8:** 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.

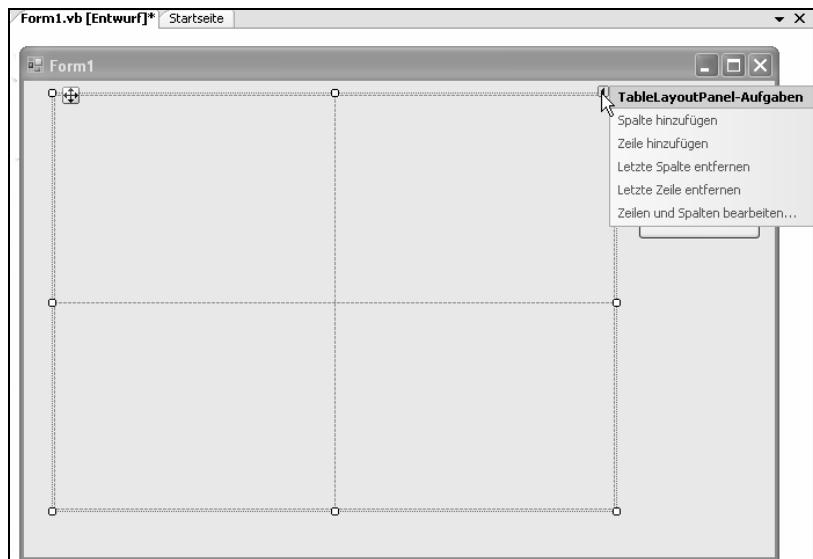


**Abbildung 3.9:** ... und hier nach Anwenden der Funktionen *Größe angelichen*, *Links ausrichten*, *Vertikalen Abstand angleichen* sowie *Vertikalen Abstand verkleinern*

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 3.10 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 3.10:** 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 3.10 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 2005 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 2005 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 unterbrochen. 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).

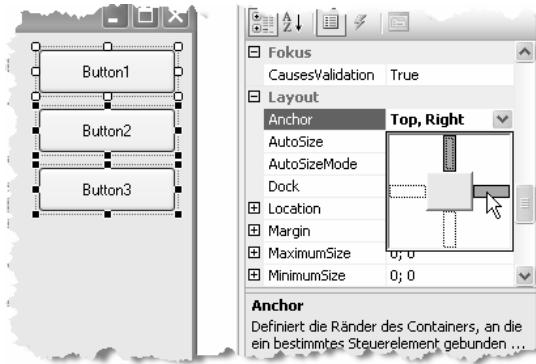
Für unser Beispiel werden wir im Folgenden die drei Schaltflächen so neu verankern, dass sie hinter dem rechten Formularrand mitwandern, wenn der Anwender das Formular in X-Richtung vergrößert oder verkleinert. Das TableLayoutPanel, das später die Eingabefelder, Beschriftungen und das Bild enthält, wird 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 3.11.

---

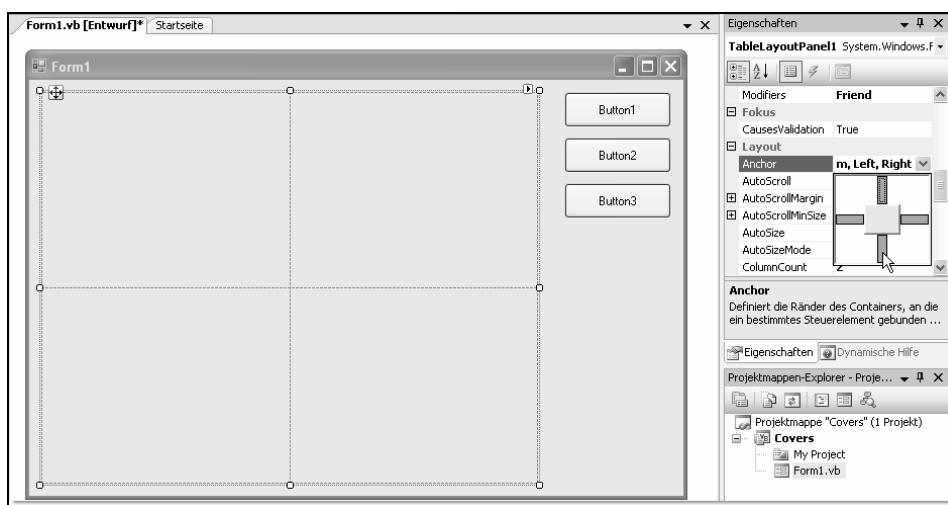
**TIPP:** Sie müssen wie hier im Beispiel nicht Eigenschaften verschiedener Steuerelemente, die Sie mit gleichen Werten belegen möchten, nacheinander setzen. Selektieren Sie stattdessen die Steuerelemente, für die der gleiche Wert einer Eigenschaft gelten soll, und weisen Sie mit dem Eigenschaftenfenster 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 3.11:** 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 TableLayoutPanel so, dass es mit den drei betroffenen Seiten möglichst nah an den Formularrändern liegt.
4. Setzen Sie die Anchor-Eigenschaft des TableLayoutPanel so, dass es an alle vier Seiten gebunden ist.



**Abbildung 3.12:** 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

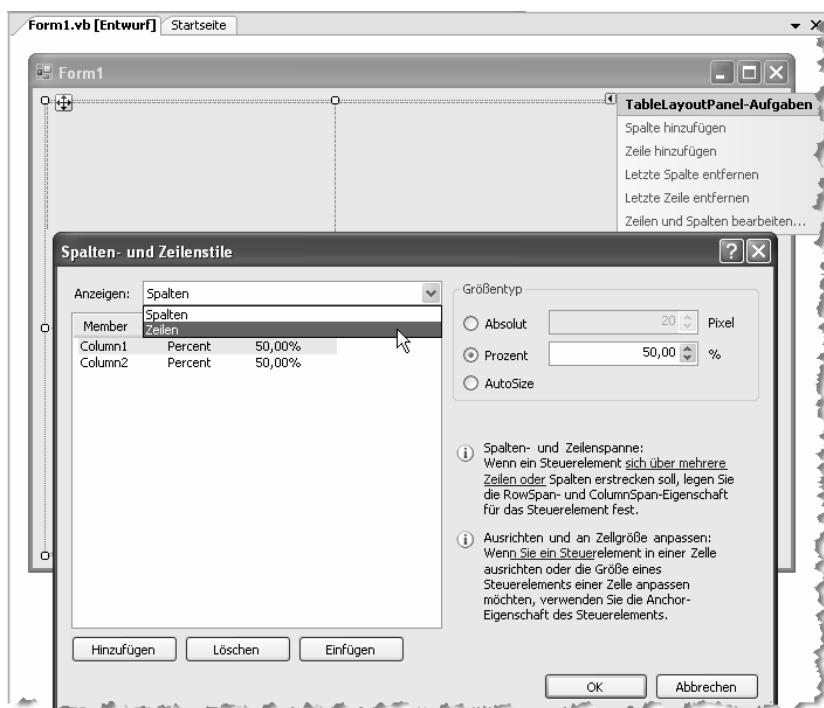
Wenn Sie das TableLayoutPanel in Abbildung 3.12 betrachten, sehen Sie, dass es aus insgesamt vier Zellen besteht. Diese Zellen dienen, wie beispielsweise das Panel, als Container für jeweils ein weiteres Steuerelement. Das Besondere: Wenn sich das TableLayoutPanel an sich vergrößert bzw. verklei-

nert, 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.

### 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 3.13.
2. Wählen Sie aus der Aufklappliste *Anzeigen* das Element *Zeilen*.



**Abbildung 3.13:** 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 3.14:** In diesem Dialog konfigurieren Sie die Größenverhältnisse der einzelnen Zellen

4. 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 3.14. 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 **Eingabe**, sondern klicken Sie, falls Sie weitere Größentypen oder Werte ändern möchten, einfach auf die nächste Zeile in der Liste, deren Einstellungen Sie ändern möchten. **Eingabe** 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.

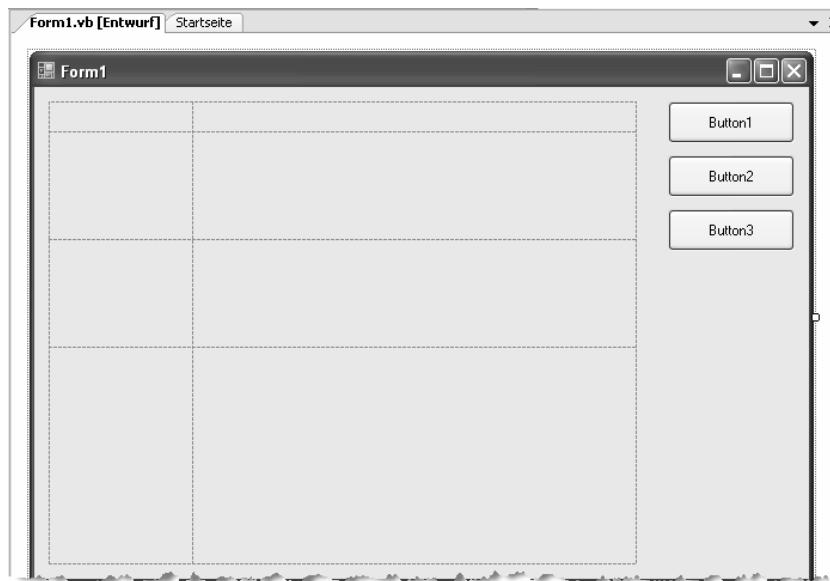
---

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 *Column1* (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 gleichlauenden 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 **Eingabe** 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 3.15 ausschauen.

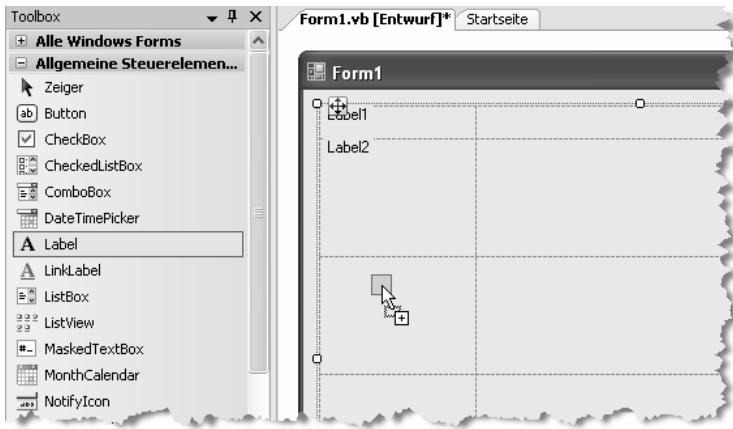


**Abbildung 3.15:** So sollte das vorläufige Ergebnis ausschauen, nachdem Sie die Zeilen- und Spaltenparameter der Zellen bestimmt haben

### Anordnen von Steuerelementen in den Zellen eines TableLayoutPanel

Ein Blick auf das Ergebnis in Abbildung 3.2 offenbart, was als nächstes auf uns zukommt: die Anordnung der TextBox- und Label-Steuerelemente für die Eingabe der Daten sowie deren Beschriftung.

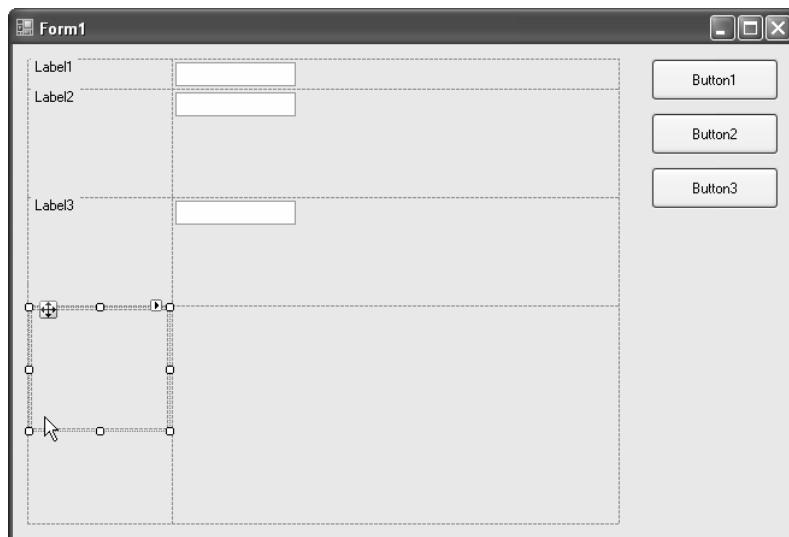
1. Zum Platzieren der Steuerelemente fügen Sie sie per Drag&Drop aus der Toolbox in die Zellen des TableLayoutPanel ein. Abbildung 3.16 hilft Ihnen bei diesem Vorgang.



**Abbildung 3.16:** Ziehen Sie zum Platzieren der Steuerelemente diese einfach per Drag&Drop in die jeweiligen Zellen des TableLayoutPanel

2. Nach diesem Verfahren ziehen Sie drei Label-Elemente in die oberen drei linken Zellen und drei TextBox-Elemente in die oberen drei rechten Zellen.
3. 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 3.17 gestalten.



**Abbildung 3.17:** 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-Panel 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 3.17 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 die PictureBox, 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 3.18.

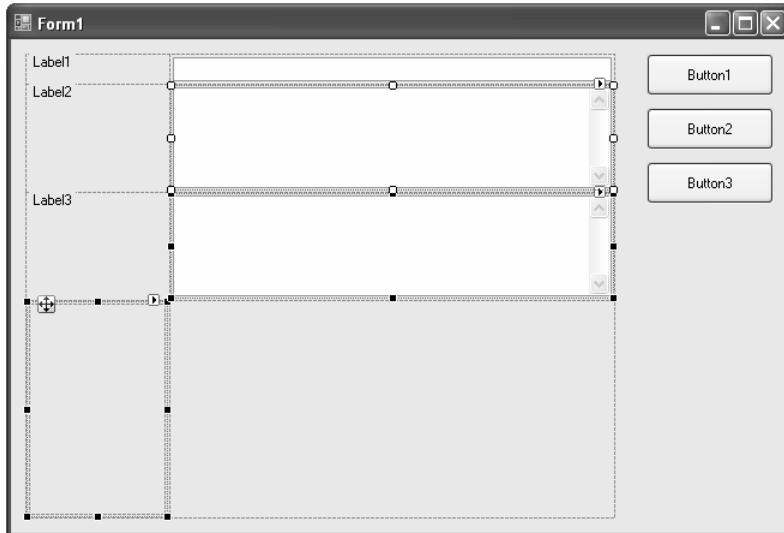


Abbildung 3.18: Nach dem Verankern aller Steuerelemente sollte das Formular wie hier ausschauen

### 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*) ausgelegt 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.

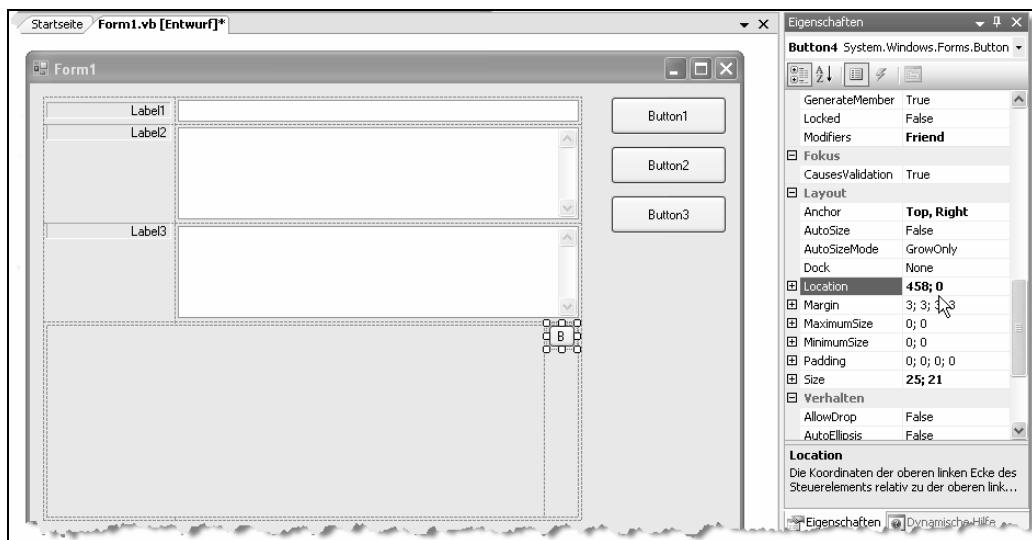
Eine solche Komponente schaut sich dabei an, welche Steuerelemente, die in ihrer Reichweite liegen, für die Erweiterung durch neue Eigenschaften, die im Kontext Sinn machen, in Frage kommen und stattet diese Steuerelemente mit neuen Eigenschaften aus. Diese Eigenschaften existieren aber dann nur für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

Steuerelemente außerhalb der Reichweite des Properties 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 3.2) 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, dass der Y-Wert nicht negativ sondern am besten 0 ist. Damit liegt die Schaltfläche ganz oben an.



**Abbildung 3.19:** 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.

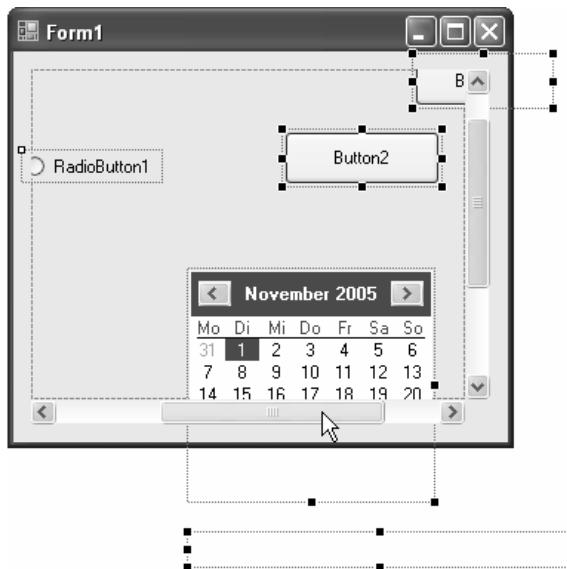
5. Vergrößern Sie das zweite Panel, das Sie gerade eingefügt haben, so, dass es möglichst ohne Abstand an den äußeren Rändern des umgebenden Panel anliegt. Die Eigenschaften Location und Size können Ihnen beim Feintuning helfen.
6. Setzen Sie die Anchor-Eigenschaft der Schaltfläche auf *Top, Right*.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

Sie sollten anschließend ein Ergebnis etwa wie das in Abbildung 3.19 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 3.2 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.



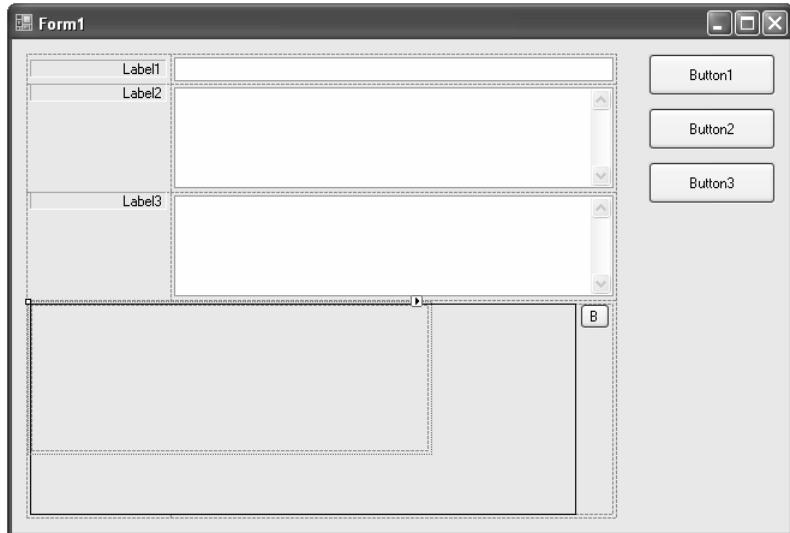
**Abbildung 3.20:** In diesem Beispiel befinden sich alle Steuerelemente in einem Panel1, das als Container fungiert. Da nicht alle Steuerelemente in den sichtbaren Ausschnitt passen und seine *AutoScroll*-Eigenschaft gesetzt ist, lässt sich der dargestellte Ausschnitt zur Entwurfs- und Laufzeit mit automatisch zur Verfügung gestellten Rollbalken einstellen.

Allerdings stellen Container-Steuerelemente eine Funktionalität zur Verfügung, mit der wir sehr nah an das rankommen, 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 3.20 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 lautet *SizeMode*, 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,<sup>3</sup> Sie brauchten sogar noch *nicht einmal eine einzige Zeile* Code dazu zu schreiben. Gehen wir's also an:

---

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



**Abbildung 3.21:** 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 3.21 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 **Tabulator**, 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.

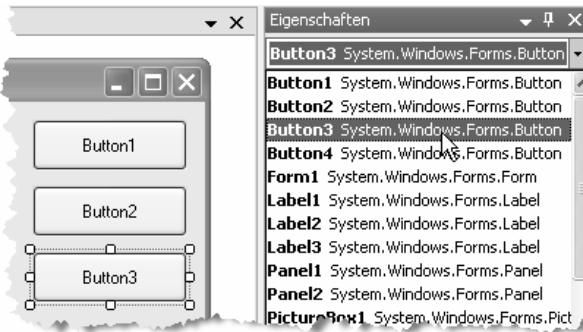
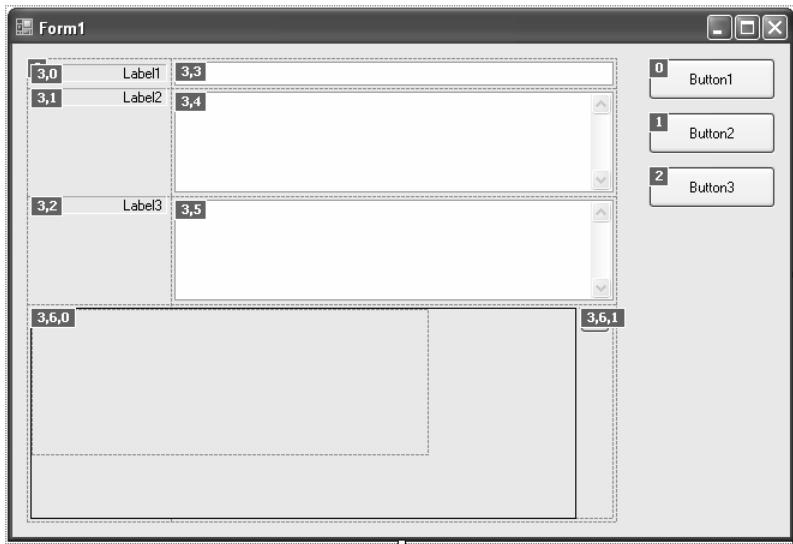


Abbildung 3.22: Mithilfe des Eigenschaftenfensters können Sie jedes Steuerelement im Designer selektieren

## 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 Tabulator zum jeweils nächsten Element springen will.



**Abbildung 3.23:** Nach dem Aufrufen von *Ansicht/Aktivierreihenfolge* können Sie die Reihenfolge festlegen, mit der Steuerelemente zur Laufzeit per Tabulator zu erreichen sind. Das Formular muss dazu selektiert sein.

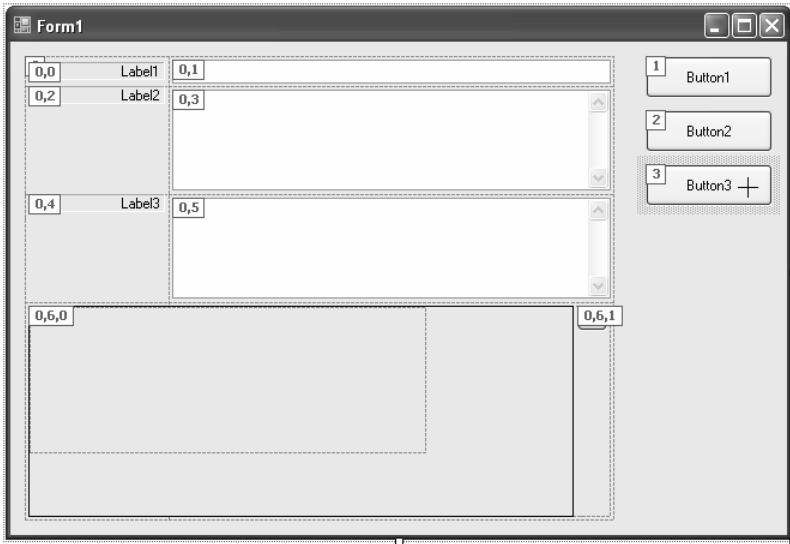
Mit Visual Studio 2005 ist das ruckzuck erledigt. Um beim Beispiel zu bleiben:

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 3.23 zu sehen.
3. Klicken Sie nacheinander die Steuerelemente in der Reihenfolge an, wie Sie durch Tabulator 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 3.23 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 3.24 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 3.24 sehen Sie das bei *Button3*.

---



**Abbildung 3.24:** Die Ordinalnummern aller Steuerelemente, die Sie bereits bestimmt haben, werden mit einem hellen Hintergrund eingefärbt. Drücken Sie **Esc**, wenn Sie mit der Zuweisung fertig sind.

- Nachdem Sie die letzte Schaltfläche angeklickt haben, drücken Sie **Esc**, um den Aktivierreihenfolgenmodus zu verlassen.

---

**TIPP:** Die Aktivierreihenfolge wird maßgeblich durch die TabIndex-Eigenschaft eines Steuerelements festgelegt. Sie können die Aktivierreihenfolge deswegen auch ändern, indem Sie die TabIndex-Eigenschaften der Steuerelemente manuell anpassen.

---

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, doch ist die Text-Eigenschaft bei diesen dennoch 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 ...

---

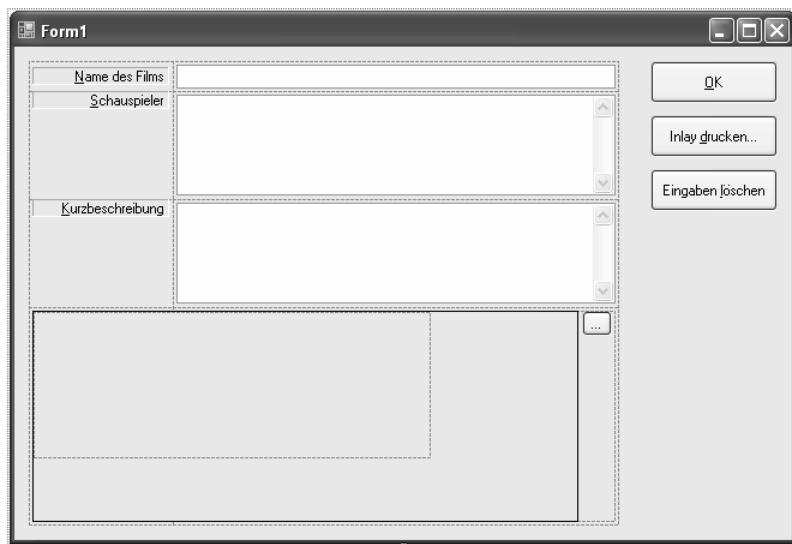
**TIPP:** Wenn Sie Text- und Name-Eigenschaften zuweisen, sollten Sie das nicht wechselweise sondern nacheinander machen. Bestimmen Sie also erst alle Name-Eigenschaften und dann alle Text-Eigenschaften. Der Grund: Wenn Sie das erste Steuerelement angeklickt und dann beispielsweise die Text-Eigenschaft 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 Eigenschaftenfensters 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 Texteigenschaft 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, in dem er die Schnellzugriffstaste in Verbindung mit **Alt** betätigt.

Auf diese Weise bestimmen Sie zunächst die Text-Eigenschaften aller Label- und Button- (Schaltflächen-) Steuerelemente. Orientieren Sie sich dabei am Abbildung 3.25 und folgender Tabelle.



**Abbildung 3.25:** 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 3.1:** 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 67 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: **Eingabe** (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 **Eingabe** 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.

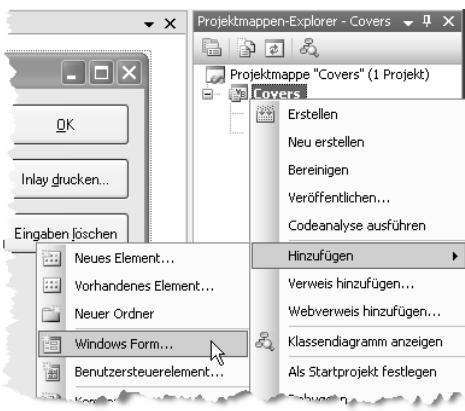
---

**WICHTIG:** Was im Formular passiert (das Setzen oder Abfragen von Eingabefeldern, das Auswerten von Benutzeraktionen) sollte einzig und allein der Formularklasse vorbehalten sein! Vermeiden Sie es, Inhalte von Benutzersteuer-elementen 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*.
2. Wählen Sie *Hinzufügen* und *Windows Form*.



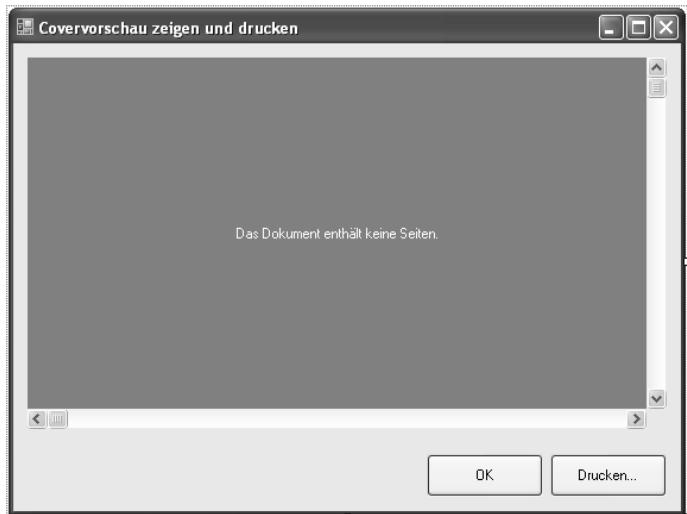
**Abbildung 3.26:** Mithilfe des Eigenschaftenfensters können Sie jedes Steuerelement im Designer selektieren

3. Im Dialog, der jetzt erscheint, belassen Sie den vorgegebenen Namen zunächst mit *Form2.vb* so, wie er ist.
4. Klicken Sie auf *Hinzufügen*.
5. 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 3.27.
6. 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.
7. Setzen Sie die Anchor-Eigenschaft dieses Steuerelements auf *Top, Bottom, Left, Right*.
8. Fügen Sie zwei Schaltflächen in das Formular nebeneinander ein. Setzen Sie die Anchor-Eigenschaften beider Schaltflächen auf *Bottom, Right*.
9. 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 3.2:** Name- und Text-Eigenschaften für die Steuerelemente im zweiten (Druck-) Formular

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



**Abbildung 3.27:** 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 daran, 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 71 fort.

## Namensgebungskonventionen für Steuerelemente in diesem Buch

Microsoft erklärt ausdrücklich, dass die Namensgebung von Objektvariablen keinerlei Hinweise darauf enthalten sollen, welchen Typ sie darstellen. Diverse Stil- und Sicherheitsüberprüfungswerzeuge 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.

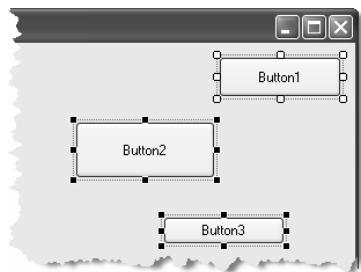
Für die Beispiele in diesem Buch habe ich mich dennoch dazu entschlossen, zumindest bei der Benennung von Objektvariablen die Steuerelemente referenzieren, darauf zu verzichten. Ich finde, dass das das Lesen von Listings auf Papier erleichtert und zum besseren Verständnis beiträgt. IntelliSense steht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

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

**Tabelle 3.3:** Ein Vorschlag für Namenskonventionen bei der Namensgebung von Steuerelementen

## Funktionen zum Layouten von Steuerelementen im Designer

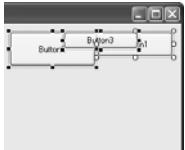
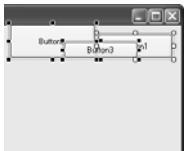
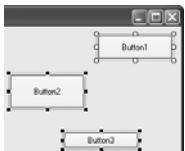
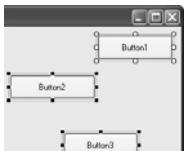
Die folgende Tabelle zeigt Ihnen die Funktionen, die sich hinter den Symbolen der Werkzeugleiste *Layout* befinden. Die folgende Grafik zeigt die Ausgangspositionierung der drei Steuerelemente im Formular. In der Tabelle finden Sie in der rechten Spalte das Ergebnis der Anordnung, nach dem Sie auf das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



**Abbildung 3.28:** Die Ausgangspositionierung der Schaltflächen für die folgende Tabelle. Achten Sie darauf, dass sich bestimmte Funktionen an der oberen rechten Schaltfläche orientieren.

<sup>4</sup> 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.

<sup>5</sup> Von **OptionButton**, der Name der Optionsschaltfläche unter VB6.

| Symbol | Funktion          | Ergebnis nach Anwendung auf Steuerelemente in Abbildung                             |
|--------|-------------------|---|
|        | Links ausrichten  |    |
|        | Zentrieren        |    |
|        | Rechts ausrichten |    |
|        | Oben ausrichten   |    |
|        | Mittig ausrichten |    |
|        | Unten ausrichten  |   |
|        | Breite angleichen |  |
|        | Höhe angleichen   |  |



| Symbol | Funktion                          | Ergebnis nach Anwendung auf Steuerelemente in Abbildung |
|--------|-----------------------------------|---|
|        | 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 3.4: Symbole, mit der Sie erweiterte Funktionalitäten des Ausgabefensters steuern

## Tastaturkürzel für die Platzierung von Steuerelementen

| Taste                         | Aufgabe  |
|-------------------------------|--|
| <b>Pfeiltasten</b>            | Verschiebt das ausgewählte Steuerelement pixelweise in die den Pfeiltasten entsprechende Richtung.                               |
| <b>Strg + Pfeiltasten</b>     | Verschiebt das ausgewählte Steuerelement zur ersten, in der den Pfeiltasten entsprechenden Richtung liegenden Ausrichtungslinie. |
| <b>Umschalt + Pfeiltasten</b> | Vergrößert oder verkleinert das ausgewählte Steuerelement pixelweise entsprechend der verwendeten Pfeiltaste. ►                  |

| Taste                                | Aufgabe   |
|--------------------------------------|---|
| <b>Strg + Umschalt + Pfeiltasten</b> | Vergrößert oder verkleinert das Steuerelement so, dass es – abhängig von der verwendeten Pfeiltaste – mit der entsprechenden Seite an der jeweils nächsten Ausrichtungslinie anliegt. |
| <b>Alt</b>                           | Schaltet die Ausrichtungslinienfunktion so lange aus, wie die Taste beim Verschieben von Steuerelementen mithilfe der Maus gedrückt bleibt.   |

**Tabelle 3.5:** Symbole, mit der 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:

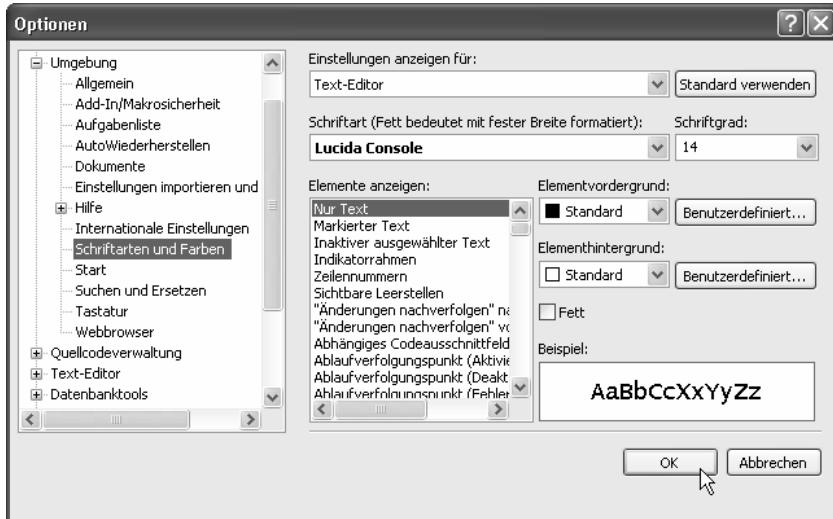
### Die Wahl der richtigen Schriftart für ermüdfreies Arbeiten

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

1. Andere befreundete Programmierer die Original-Einstellung der Schriftart des Editors genau so stört wie mich, und
2. welche Schriftart die meisten Entwickler, die mit Visual Studio 2003 .NET oder Visual Studio 2005 arbeiten, bevorzugen.

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

- Bis zu einer Auflösung von 1024 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 1280 x 1024 Pixel oder bei 1024 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 3.29:** 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 *Schriftarten 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 Lucidia 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-Explorer 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 Formulkarklasse, 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 spart Ihnen beim Entwickeln die meiste Zeit. Warum? IntelliSense liefert Ihnen alle denkbaren Informationen über Objekte und Sprachelemente, die Sie gerade in Bearbeitung 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 im Klassenkapitel 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 3.30).

---

**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 3.30 zu erkennen.

---

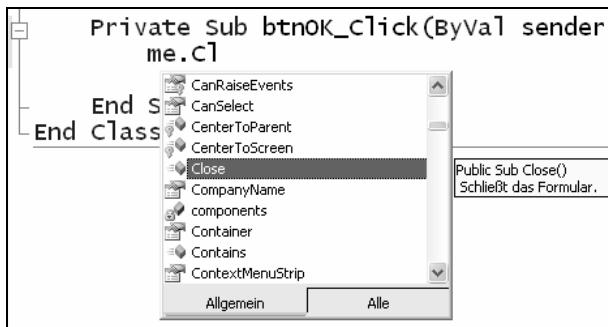
5. Sie brauchen den Methodennamen nun gar nicht weiter einzugeben. Sobald der richtige Methodename markiert ist, drücken Sie einfach **Eingabe** – der Codeeditor fügt die restlichen Buchstaben des Methodennamens dann automatisch ein und stellt den Cursor in die nächste Zeile.

---

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

---

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 3.30:** 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 3.30 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

Für das nächste IntelliSense-Feature werden wir die Funktion »Eingaben löschen« implementieren.

1. Wechseln Sie mit **Strg+Tab** erneut in die Designerdarstellung des Formulars *Form1*.
2. Doppelklicken Sie auf die Schaltfläche *Eingaben löschen*, um den Funktionsrumpf für die Ereignisbehandlungsroutine dieser Schaltfläche einzufügen.
3. 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 **Eingabe**.

4. 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 3.31:** Bei Methoden, die über Parameter verfügen, zeigt IntelliSense alle Parameter mit samt Erklärungen der Parameter an. Der jeweils aktuelle Parameter, den Sie gerade eingeben, 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:

1. Ergänzen Sie die Zeile (die ja noch nicht vollständig eingegeben wurde), um

"Sind Sie sicher?", "Eingaben löschen?", \_

und drücken anschließend **Eingabe**.

Nach dem Zeilenumbruch ist die Parameterinfo verschwunden. Um die Parameterinfo auch in der neuen Zeile wieder darzustellen, drücken Sie einfach **Strg+Umschalt+Leertaste**.

2. Geben sie nun den restlichen Teil der logischen Befehlszeile ein.

```
MessageBoxButtons.YesNo, _
MessageBoxIcon.Question, MessageBoxDefaultButton.Button2)
```

## Automatische Vervollständigung von Struktur-Schlüsselworten und Codeeinrückung

1. Für die Komplettierung der Methode zum Löschen der Eingabefelder geben Sie bitte die folgende Zeile an:

```
If locDr = Windows.Forms.DialogResult.Yes Then
```

Sobald Sie **Eingabe** 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-Zeiten<sup>6</sup> 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.<sup>7</sup>

### 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 3.32 zu sehen.

```
If locDr = Windows.Forms.DialogResult.OK Then  
    txtKurzbeschreibung.Text = "..."  
    txtNameDesFilms.Text = "..."  
    txtSchauspieler.Text = "..."  
    picCoverbild.Image = Nothing  
    myBilddateiname = "..."  
End Sub
```

**Abbildung 3.32:** Fehler, die der Hintergrundcompiler feststellt, werden direkt im Editor noch vor dem eigentlichen Kompilierungs-vorgang beim Erstellen des Projektes gekennzeichnet

---

<sup>6</sup> 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.

<sup>7</sup> Hintergrundcompiler gibt es zwar auch in diesen Sprachen, aber die sind lange nicht so konsequent wie in Visual Basic .NET bzw. Visual Basic 2005, 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 bzw. 2005 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
```

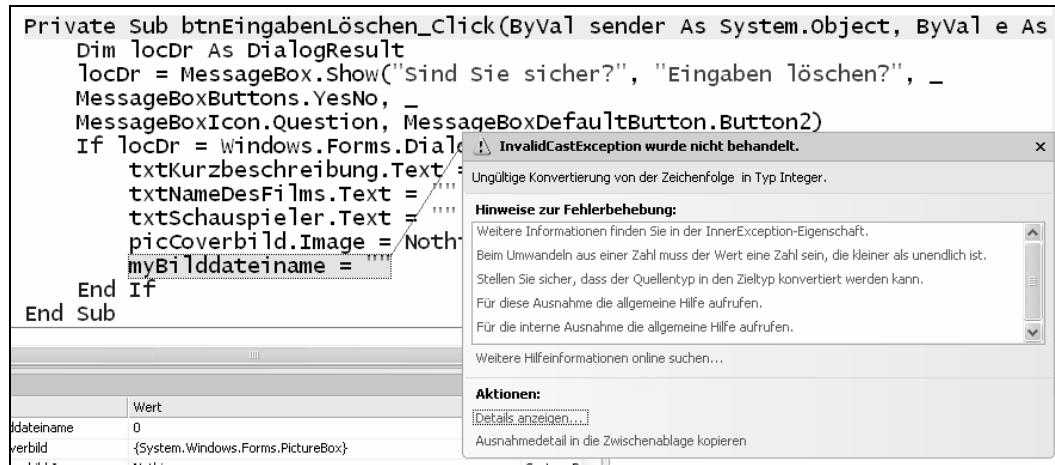
```
    Dim myBilddateiname As Integer
```

Sie sehen, dass der Fehler nun nicht mehr durch Unterschlängelung markiert ist.<sup>8</sup>

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 löschen, bricht das Programm mit folgender Meldung ab (Abbildung 3.33):

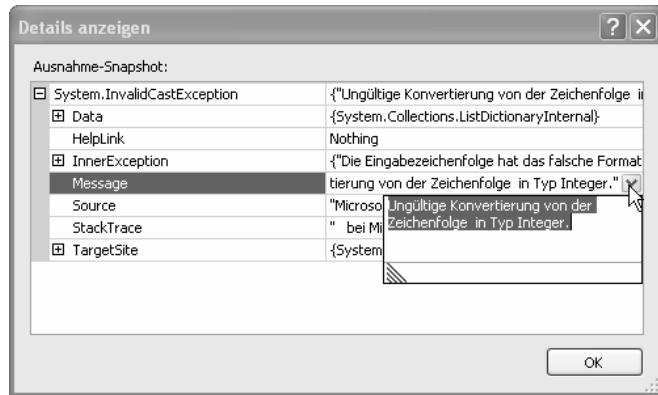


**Abbildung 3.33:** Tritt während der Anwendungsausführung im Debug-Modus ein Fehler (eine so genannte »Ausnahme« – engl.: *Exception*) auf, ruft Visual Studio den Editor auf, unterlegt die betroffene Stelle gelb und zeigt einen entsprechenden Hinweis an

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

<sup>8</sup> 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! ;-)

`myBilddateiname` passiert,<sup>9</sup> 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 3.34:** Mithilfe des Ausnahmedetail-Dialogs erfahren Sie Genaueres über die Umstände, die zur Ausnahme führten

Falls Sie in einem solchen Fall genauere Information zum Ausnahmenumstand benötigen, klicken Sie unten im Dialog unter *Aktionen* auf *Details anzeigen...*. Der Editor zeigt Ihnen anschließend einen weiteren Dialog (siehe Abbildung 3.34), 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 unsere 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.

Sobald Sie die Anweisung

`Option Strict On`

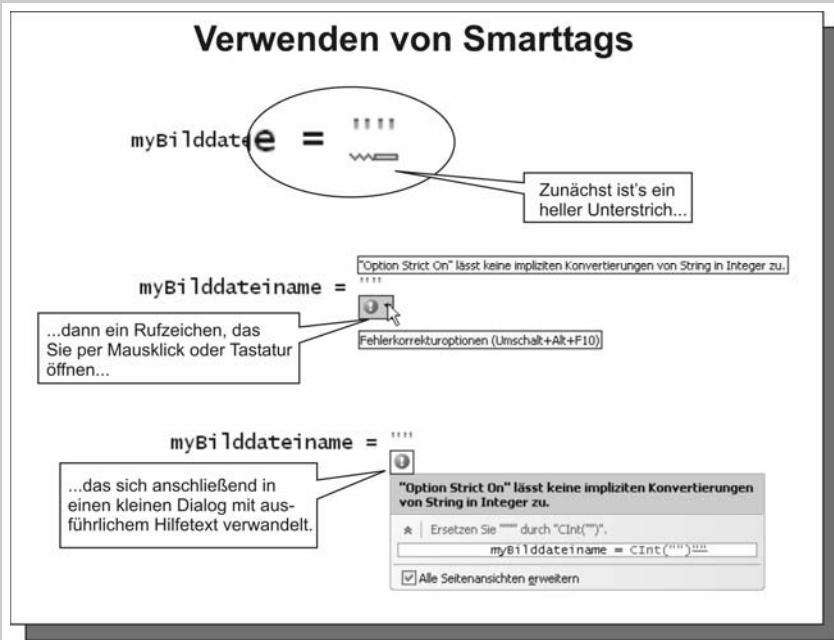
---

<sup>9</sup> O.K., o.k. – mir, nicht Ihnen.

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 vorne herein 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 Code datei 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.

In ► Kapitel 6 finden Sie im Abschnitt »Erzwungene Typsicherheit und Deklarationszwang von Variablen« ein weiteres Beispiel für Option Explicit und Option Strict.

## 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.

- 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 Is Nothing AndAlso CoverbildDateiname <> "" Then
        locImage = Image.FromFile(CoverbildDateiname)
        Return locImage
    End If
    Return Nothing
End Function
```

- Wechseln Sie mit **F7** zum Entwurfsmodus (zum Designer-Dokumentenfenster). Übrigens: Mit **F7** wechseln Sie zwischen Designer- und Codedarstellung eines Formulars.
- Doppelklicken Sie auf die Auslassungsschaltfläche neben der *PictureBox*, um den Codeeditor zu öffnen und die Ereignisbehandlungsroutine für diese Schaltfläche zu bearbeiten.
- Fügen Sie in die Stub<sup>10</sup> (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
```

- 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 3.35 an. Fällt Ihnen was auf?

|   |  |
|---|--|
| myBilddateiname = locOfd.FileName<br>picCoverbild.Image = CoverbildAusDateinamen()<br>End With<br>End Sub | CoverbildAusDateinamen ( <i>CoverbildDateiname As String</i> ) As System.Drawing.Image |
|---|--|

**Abbildung 3.35:** Auch bei selbst geschriebenen Methoden funktioniert IntelliSense – natürlich fehlen dabei zunächst noch die Erklärungen

---

<sup>10</sup> 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 2005 ist das anders, wie Sie gleich sehen werden:

6. Bevor Sie nämlich die Zeile nun komplettieren, positionieren Sie die Schreibmarke oberhalb der Zeile

```
Function CoverbildAusDateinamen(ByVal CoverbildDateiname As String) As Image
```

7. Tippen Sie zwei Hochkomma (**Umschalt+#+**).

8. 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 3.36:** Der Editor fügt ein XML-Skelett über der Methode ein, in der Sie ihre Dokumentation nur zu ergänzen brauchen

9. Für die folgenden Schritte orientieren Sie sich auch an Abbildung 3.37. Geben Sie zwischen <summery> und </summery> 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 3.37:** 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.

---

10. 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.

---

11. Zwischen `<returns>` und `</returns>` geben Sie die Bedeutung des Rückgabewertes an.
12. Optional geben Sie zwischen `<remarks>` und `</remarks>` eine Bemerkung ein.
13. 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+Leertaste**. 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 3.38).



**Abbildung 3.38:** Wenn Sie Ihre Methoden- und Eigenschaftenprozeduren mit entsprechenden XML-Tags versehen haben, zeigt IntelliSense sowohl eine Funktionsbeschreibung ...

14. Drücken Sie **Strg+Eingabe**, um den Funktionsnamen zu vervollständigen.
15. Tippen Sie die Klammer, wird IntelliSense wieder aktiv (siehe Abbildung 3.39). Sie sehen, dass IntelliSense nun nicht nur den Parameternamen und den Parametertyp, sondern auch die Parameterdokumentation zeigt.



**Abbildung 3.39:** ... als auch entsprechende Parametererklärungen an

16. 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 Bildauschnittseinstellung 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 getätigter 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.

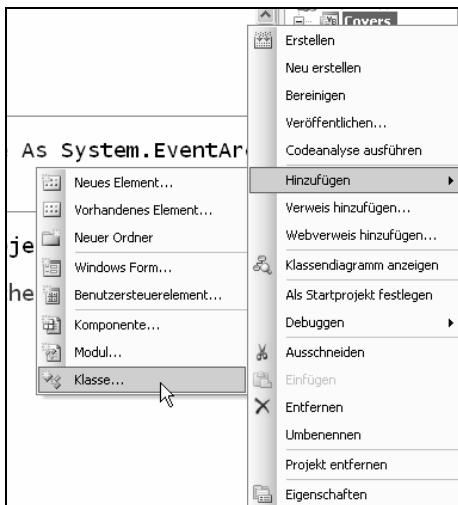
Auf diese Weise sind die beiden Aufgaben – Datenerfassung und Drucken – sauber voneinander getrennt. Der Hauptdialog sorgt in eigener Regie für die Datenerfassung, der Druckdialog selbstständig für das Drucken der Daten. Der Hauptdialog muss so nicht auf den Druckdialog direkt zugreifen, und dort irgendwelche 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 3.40:** So fügen Sie eine neue Klassendatei zum Projekt hinzu

1. 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.
2. Wählen Sie *Hinzufügen | Klasse*.
3. Im jetzt erscheinenden Dialog geben Sie den Namen der Klassendatei (ohne Dateiendung) ein – für unser Beispiel **CoverInhalt**.
4. Klicken Sie auf **OK** oder drücken Sie **Eingabe**.

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.

---

**BEGLEITDATEIEN:** 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 2005 - Entwicklerbuch\B - IDE\03 - Covers\Druckroutine für Covers.txt**.

---

1. Öffnen Sie die Textdatei **\B - Ein- und Umstieg\Druckroutine für Covers.txt** mit dem Notepad von Windows.
2. Drücken Sie **Strg+A** (alles markieren), **Strg+C** (in die Zwischenablage kopieren).
3. 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+Tabulator** 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# 2005 wurde leider in viel größerem Umfang mit Refactoring-Werkzeugen bedacht als Visual Basic 2005. Im Grunde genommen gibt es in Visual Basic 2005 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 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.

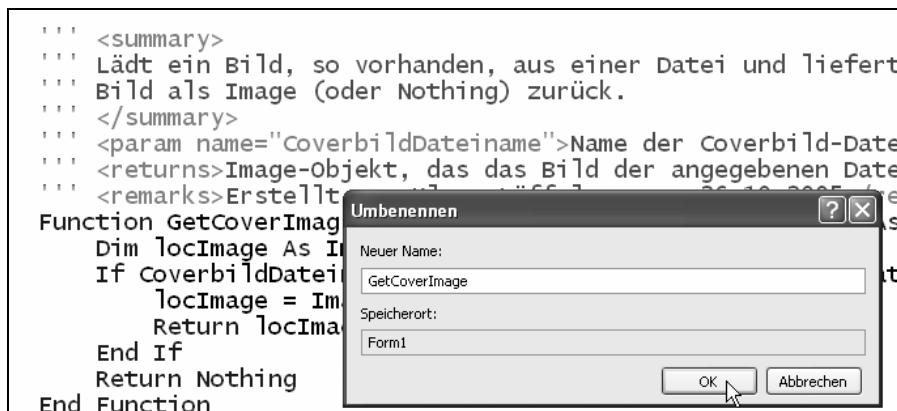
Am schlimmsten ist es bei solchen Aktionen, wenn sich Namen, die Sie beispielsweise Methoden gegeben haben, als falsch oder nicht ausreichend aussagekräftig entpuppen. Gerade wenn Sie im Team arbeiten, sind Sie auf dieses Problem sicherlich schon das ein oder andere Mal gestoßen. Und jeder, der eine 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 2005 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 3.41:** Mit dem Umbenennen beispielsweise einer Methode (einer Funktion) ndern 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* angeschaut haben.

---

**HINWEIS:** Bei umfangreichen Projekten mit vielen Formulardateien oder Klassendateien kann sich das Refactoring von Klassen- und Formulkarklassen durch das Umbenennen ihrer Codedatei 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 3.42 zu sehen, setzen Sie die *EnableRefactoringOnRename*-Eigenschaft auf *False*.<sup>11</sup>

---

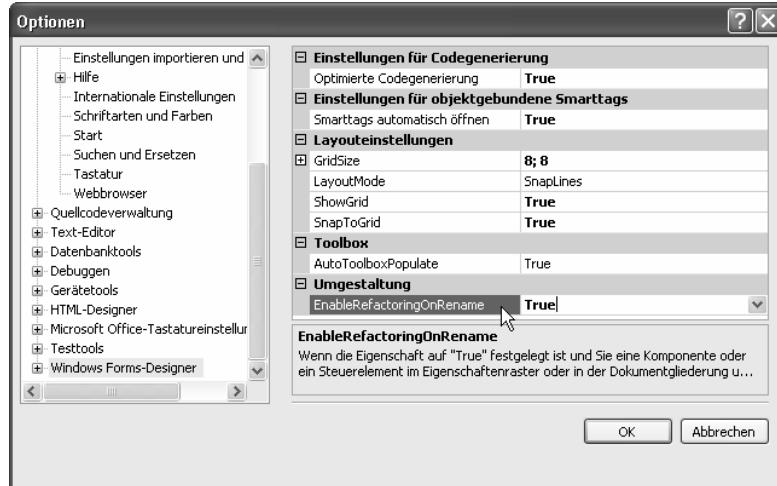


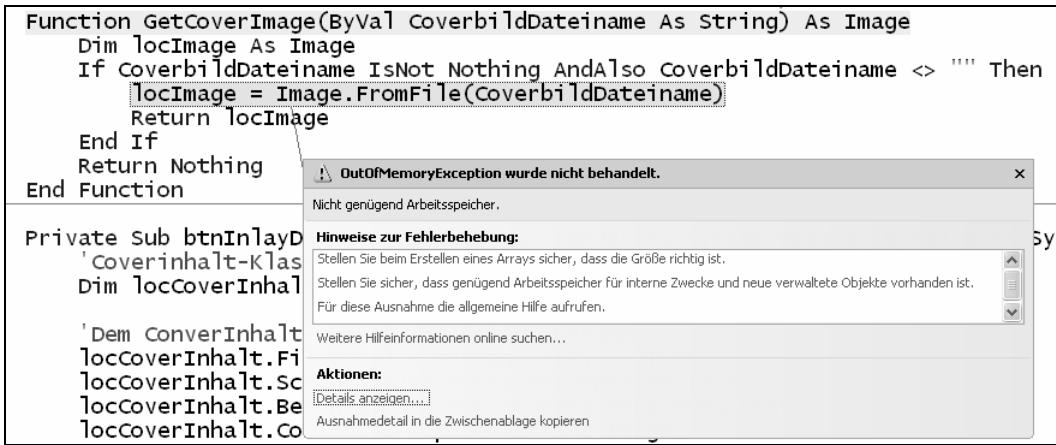
Abbildung 3.42: Hier schalten Sie das Refactoring durch Umbenennen von Codedateien aus

## 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:

---

<sup>11</sup> Die in VS 2005 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 B0302 als Testversion oder kostenpflichtige Vollversion beziehen.



**Abbildung 3.43:** 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 2005 verfügt über eine Codeausschnittsbibliothek, die für jeden Geschmack etwas Passendes bereitstellt. Zum Beispiel, wie man einen Coderumpf zum Abfangen 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 3.44:** 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 Auffangen 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 5.

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 in dem 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 einfügen und Tabulator 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 **Tabulator** 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", "Nachrichtentext")
            Dim emailClient As New SmtpClient("E-Mail-Servername")
            emailClient.Send(message)
        End Try
    End If
    Return Nothing
End Function
```

**Abbildung 3.45:** Der eingefügte Codeausschnitt ist hier zu sehen. Mit Tabulator springen Sie bequem von Parameter zu Parameter und ändern diese in einem Rutsch.

Sie brauchen nun nur mit Tabulator 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,<sup>12</sup> 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 Serialisierung<sup>13</sup> ihrer Einstellungsdaten kümmern.

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 Zeichen-

---

<sup>12</sup> 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.

<sup>13</sup> 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.

ketten 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 2.0 bzw. Visual Studio 2005 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, dass die Inhalte dieser Variablen, wenn Sie es wünschen, beim Programmende gesichert und beim nächsten Programmstart automatisch wieder gestartet werden.

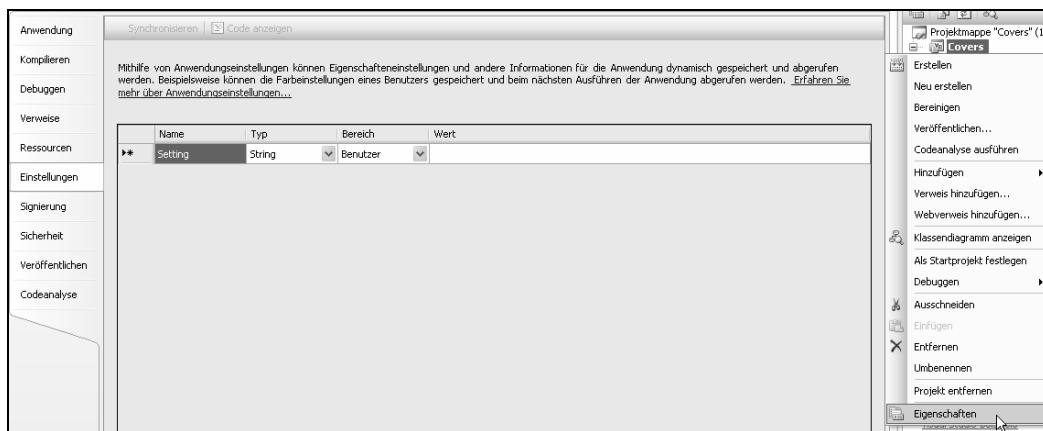
## Einrichten von Settings-Variablen

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 Projektexplorer den Menüpunkt *Eigenschaften*.
- Wählen Sie die Registerkarte *Einstellungen*, indem Sie auf die entsprechende Registerkarte 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-Variable nennen wir daher *LetzterFilmTitel*, und sie soll vom Typ *String* sein, da sie Zeichenketten speichert.

- Klicken Sie, wie in Abbildung 3.46 zu sehen, in die ersten Namens-Zelle der Settings-Tabelle, und geben Sie als Variablenname *LetzterFilmname* ein. Drücken Sie **Tabulator**.

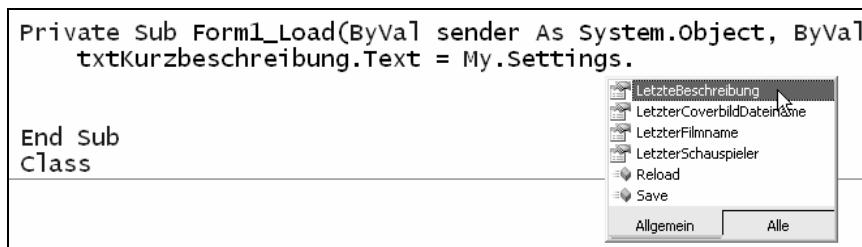


**Abbildung 3.46:** Den Settings-Designer (Einstellungs-Designer) finden Sie in den Projekteigenschaften, die Sie aus dem Kontextmenü des Projektes erreichen

4. In der nächsten Spalte würden Sie in der Aufklapplistene den gewünschten VariablenTyp auswählen, was Sie in diesem Fall nicht machen müssen, da String für Texteingaben bereits der passende ist. Drücken Sie daher einfach **Tabulator**.
5. 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.
6. 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.
7. Drücken Sie **Tabulator**, um in die nächste Zeile zu gelangen, die vom Settings-Designer automatisch angelegt wird.
8. Auf diese Weise legen Sie weitere Variablen (alle vom Typ String und dem Bereich *Benutzer*) namens *LetzterSchauspieler*, *LetzteBeschreibung* und *LetzterCoverbildDateiname* an.
9. Klicken Sie auf das *Ausgewählte Elemente Speichern*-Symbol, um die Änderungen zu übernehmen.

### Verwenden von Settings-Variablen im Code

1. Wechseln Sie nun mit **Strg+Tabulator** 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 3.47 zu sehen, alle Settings-Variablen aufgelistet, die Sie gerade eingerichtet haben.



**Abbildung 3.47:** 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 Gesamt-codeblock 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.LetzterFilname
    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 Feldinhalt) 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**,<sup>14</sup> und die entsprechende Ereignisbehandlungsroutine werden wir im Folgenden implementieren:

1. Wählen Sie mit **Strg+Tabulator** 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.
4. In der Rubrik *Verhalten* finden Sie das **FormClosing**-Ereignis; auf diesen Eintrag doppelklicken Sie nun.

---

<sup>14</sup> Mehr zum Thema »Schließen von Formularen« erfahren Sie in ► Kapitel 27.

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
```

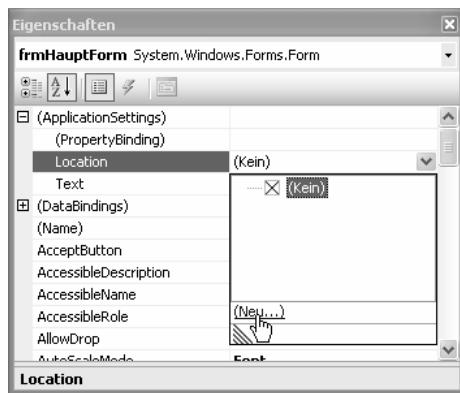
### Verknüpfen von Settings-Werten mit Formular- oder Steuerelementeigenschaften

Settings-Werte können übrigens auch dazu verwendet werden, Eigenschaften von Formularen und/oder deren Steuerelementen zu speichern. Dabei werden die entsprechenden Eigenschaftenwerte 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.

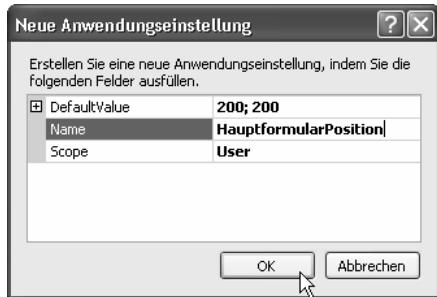
Doch diesen Vorgang können Sie auch automatisieren – Sie binden die entsprechende Eigenschaft einfach an einen Settings-Wert, und das funktioniert folgendermaßen:

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 3.48 zu sehen.



**Abbildung 3.48:** An dieser Stelle steuern Sie das Binden von Formulareigenschaften an die Anwendungseinstellungen (Application Settings)

- In der Liste, die sich nun öffnet, klicken Sie auf (*Neu...*).
- 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 richtigem 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 3.49:** An dieser Stelle steuern Sie das Binden von Formulareigenschaften an die Anwendungseinstellungen (Application Settings)

- 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_ohvclojvckpztoiykwlhs3ba2acq2v4i\1.0.0.0
```

Warum?

- Mein Anmeldename in unserem Active Directory-Netzwerk lautet *loeffel*. Unsere Domäne *Active-Develop* (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 können<sup>15</sup> 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="LetzterFilmname" 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. Anrie 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>
```

---

<sup>15</sup> Dabei handelt es sich übrigens um das vieldiskutierte Edit & Continue-Feature, das es seit Visual Basic .NET 2002 nicht mehr und erst mit Visual Basic 2005 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 zu miss verstehen. 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-2.0-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

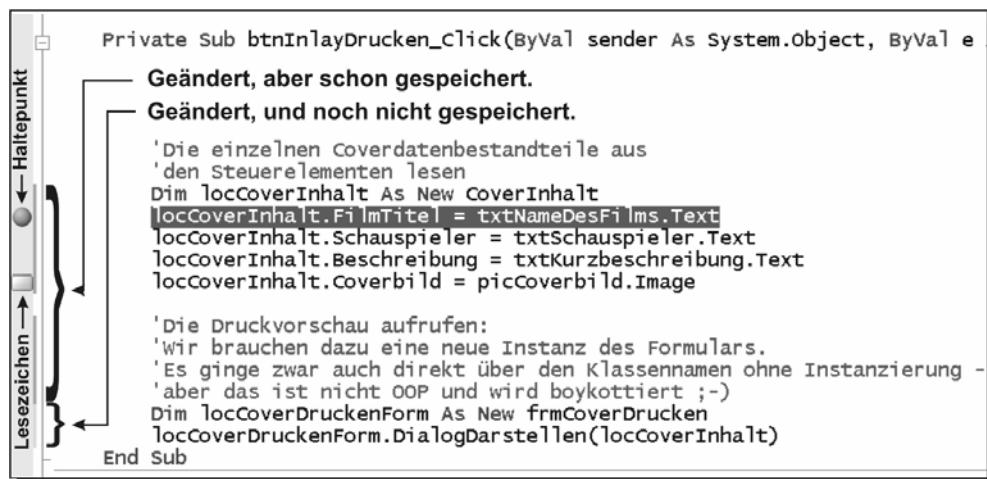
Nun hat Ihnen das Durchexerzieren des Beispiels schon viele der Funktionen des Editors nahe gebracht – nur leider nicht alle. Einige, von denen ich meine, dass Sie Ihnen beim Entwickeln von Projekten ebenfalls gute Dienste leisten können, finden Sie in den folgenden Abschnitten beschrieben.

### 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.

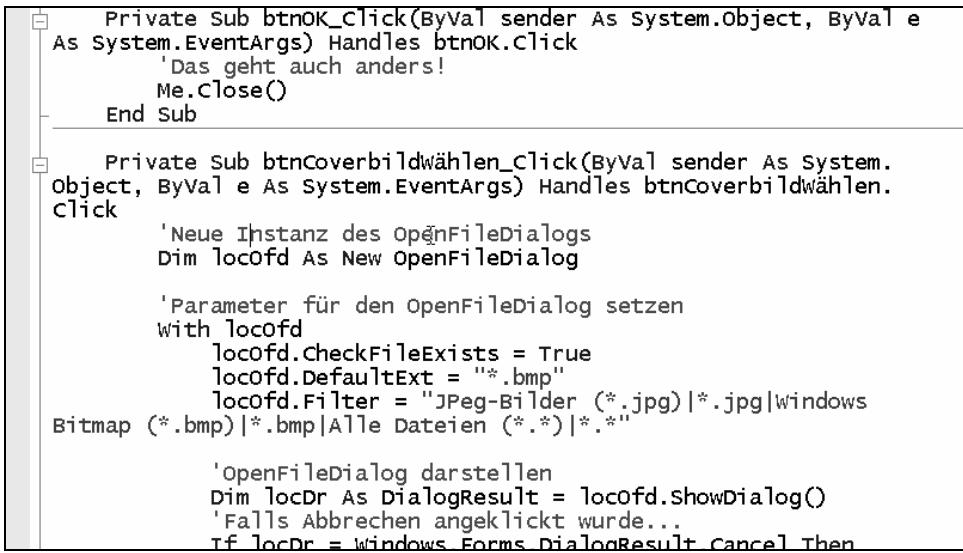
Sie können im Indikatorrand ebenfalls erkennen, welche Änderungen Sie seit dem Öffnen einer Codedatei am Code durchgeführt haben, und welche dieser Änderungen wiederum schon gespeichert wurden. Diese Codezustandsanzeige wird durch entsprechende Balken im Indikatorrand hervorgehoben. Abbildung 3.50 verdeutlicht die Funktionsweise des Indikatorrands.



**Abbildung 3.50:** 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 3.51).



```
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 (*.*)|*.*"

        'OpenFileDialog darstellen
        Dim locDr As DialogResult = locOfd.ShowDialog()
        'Falls Abbrechen angeklickt wurde...
        If locDr = Windows.Forms.DialogResult.Cancel Then

```

**Abbildung 3.51:** Mit dem automatischen Zeilenumbruch werden auch lange Zeilen auf einen Blick erkennbar

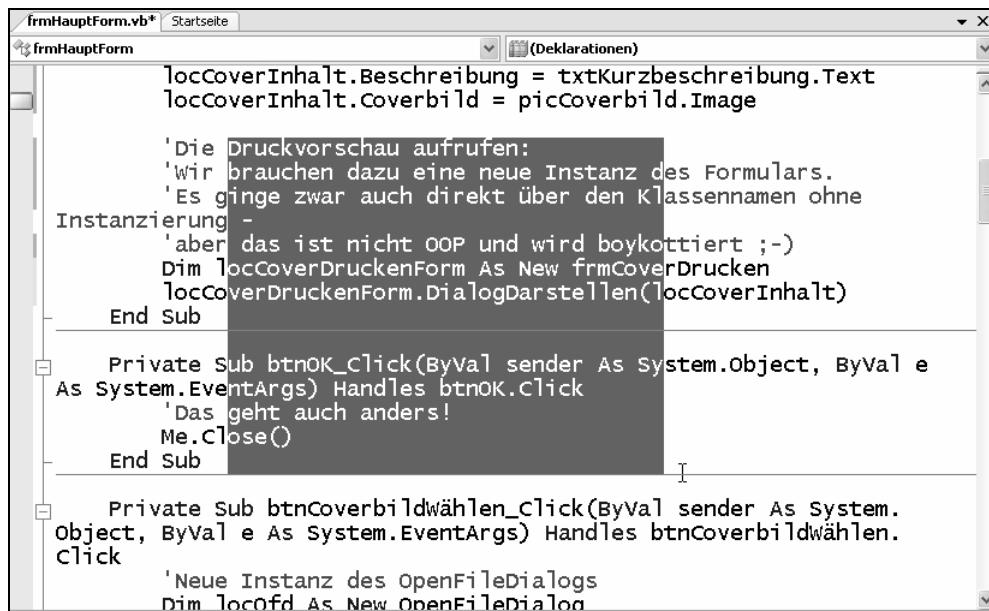
Doch aufgepasst: Eine derart umbrochene Zeile entspricht nicht dem Codezeilenumbruch von Visual Basic mit dem »\_«-Zeichen am Zeilenende. Die Gefahr ist groß, eine durch den Editor umbrochene Zeile mit Tabulatoren oder Leerzeichen bündig zu formatieren – Sie würden dadurch aber Leerzeilen in die eigentliche 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 + Umschalt + -**, 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 3.52 zu sehen.



The screenshot shows the Visual Studio .NET code editor with the file `frmHauptForm.vb` open. A rectangular selection is made around the following VB.NET code:

```
    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

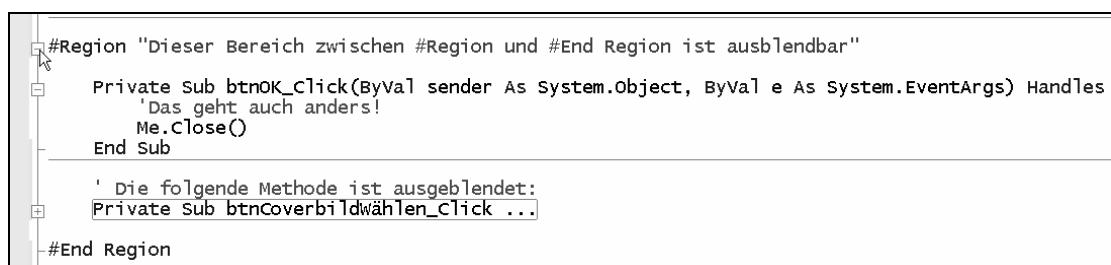
    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
        'Das geht auch anders!
        Me.Close()
    End Sub

    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
    End Sub
```

Abbildung 3.52: 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 3.53) können Sie den Code ausblenden.



The screenshot shows the Visual Studio .NET code editor with the file `frmHauptForm.vb` open. Several regions are collapsed, indicated by a small triangle icon next to the region header. One region is expanded, showing the following VB.NET code:

```
#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 ...
```

Abbildung 3.53: 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 3.53 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.

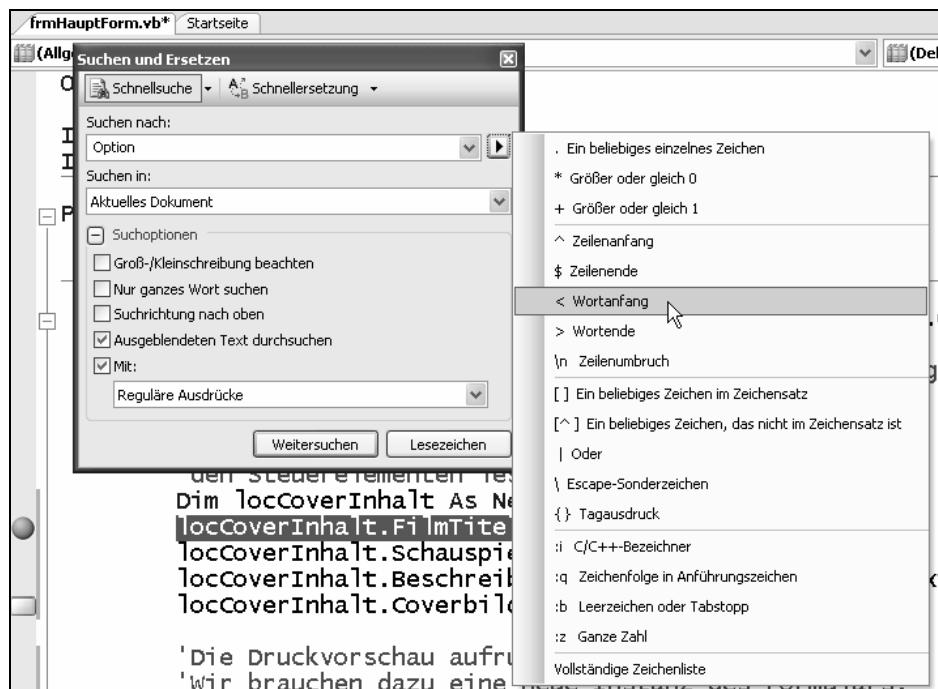
Mit der Schnellsuchfunktion finden Sie in der aktuellen Datei bzw. in allen geöffneten Dateien einen beliebigen Suchtext. Diese Funktion arbeitet bei Bedarf auch mit regulären Ausdrücken; Sie können den zu durchsuchenden Bereich genauer spezifizieren; Sie können nach Sonderzeichen suchen und haben sogar die Möglichkeit, alle Stellen, an denen ihr Suchbegriff vorkommt, durch Lesezeichen zu markieren (siehe auch Abbildung 3.54).

---

**HINWEIS:** Reguläre Ausdrücke sind leistungsfähige Werkzeuge – allerdings braucht man ein wenig Einarbeitungszeit, um sie nutzen zu können. Im ► Kapitel 21 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.



**Abbildung 3.54:** 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 3.55:** 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.

| File                                                                                 | Line | Code Snippet                   |
|--------------------------------------------------------------------------------------|------|--------------------------------|
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\CoverInhalt.vb(1):      |      | Public Class CoverInhalt       |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\CoverInhalt.vb(3):      |      | Public FilmTitel As String     |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\CoverInhalt.vb(4):      |      | Public Schauspieler As String  |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\CoverInhalt.vb(5):      |      | Public Beschreibung As String  |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\CoverInhalt.vb(6):      |      | Public Coverbild As Image      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\CoverInhalt.vb(8):      |      | End Class                      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(5):  |      | Public Class frmCoverDrucken   |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(7):  |      | Private WithEvents myPrintDocu |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(8):  |      | Private myInhalt As CoverInhal |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(10): |      | Public Function DialogDarstel  |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(12): |      | 'Die übergebenen Werte de      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(15): |      | 'Das folgende Objekt brau      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(21): |      | 'drucken, da das Cover so      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(25): |      | 'Dazu den Preview-Steuere      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(26): |      | 'Sobald die Zuweisung erf      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(36): |      | Private Sub btnDruckAufStanda  |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(38): |      | 'Den Anwender den Drucker      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(39): |      | Dim locPR As New PrintDia      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(40): |      | Dim locDR As DialogResult      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(53): |      | Private Sub myPrintDocument_P  |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(55): |      | Dim g As Graphics = e.Gra      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(59): |      | 'TODO: Man könnte die Fon      |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(60): |      | Dim locFontHaupttitel As       |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(61): |      | Dim locFontUntertitel As       |
| D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\B - IDE\02 - Covers\frmCoverDrucken.vb(62): |      | Dim locFontRückenstein As      |

**Abbildung 3.56:** ... 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 Einfügemarke 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 Einfügemarke bewegen wollen.



**Abbildung 3.57:** 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 Einfügemarkie 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.



# **4 Tipps & Tricks für das angenehme Entwickeln zuhause und unterwegs**

---

- 109 Der Einsatz mehrerer Monitore**
  - 112 Zurücksetzen der Fenstereinstellungen**
  - 112 Sichern, Wiederherstellen oder Zurücksetzen aller Visual Studio-Einstellungen**
  - 116 Wieviel Arbeitsspeicher darf's denn sein?**
  - 117 Testen Ihrer Software unterwegs und zuhause – Microsoft Virtual PC und Microsoft Virtual Server**
  - 122 Hilfe zur Selbsthilfe**
  - 127 Erweitern Sie die Codeausschnittsbibliothek um eigene Codeausschnitte**
- 

So Sie das letzte Kapitel durchgearbeitet haben, wissen Sie, wie sehr Ihnen Visual Studio 2005 die tägliche Entwicklungsarbeit erleichtern kann. Doch es kann nicht alles. Sie können mit ein wenig Investition die Ergonomie Ihres Arbeitsplatzes enorm verbessern und Ihr eigenes Wohlbefinden und damit nicht zuletzt auch Ihren täglichen Output steigern.

Auch einiges Wissenswertes zur IDE, was im vorherigen Kapitel nicht berücksichtigt wurde, trägt dazu bei.

## **Der Einsatz mehrerer Monitore**

Die wohl beste Errungenschaft von Windows 98 – die Älteren unter Ihnen 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 Desktopoperweiterung, 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  $1280 \times 1024$  Punkten Auflösungsfähigkeit zu erwerben, und

hatte man auch die entsprechende Grafikkarte zur Verfügung, deren RAMDAC diese Auflösung mit mehr als flimmernden 60 Herz darstellen konnte, gab's unter VB6 keine Probleme.

Heute ist das anders.  $1024 \times 768$  ist meiner Meinung für ein wirklich produktives Arbeiten mit Visual Studio 2005 ohnehin zu wenig – ich denke, darüber brauchen wir uns gar nicht zu unterhalten.  $1280 \times 1024$  ist in meinen Augen buchstäblich die Obergrenze,  $1600 \times 1200$  macht erst bei deutlich mehr als 20 Zoll TFT Sinn, will man nicht nur 10 Zentimeter entfernt vor dem Monitor kleben.

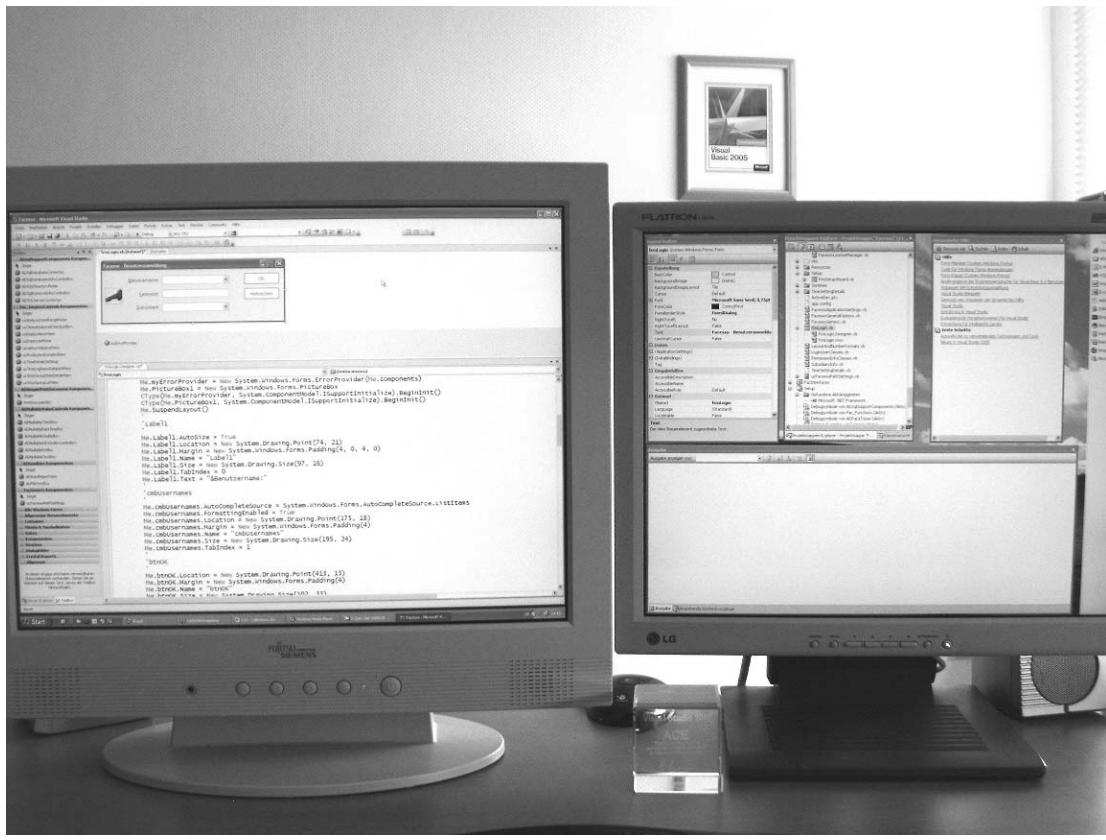
Aber es geht auch besser, flexibler und billiger – mit mehreren Monitoren an einem Rechner. Wenn Ihr Arbeitsplatzrechner nicht deutlich älter als 3 Jahre ist, könnten Sie nämlich das Glück haben, dass Ihre Grafikkarte bereits über zwei Anschlüsse für Monitore verfügt. Und dann beschaffen Sie sich für wenig Geld einfach einen zweiten, z.B. 15"-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 4.1:** Mit dem Dialog *Eigenschaften von Anzeige* erweitern Sie auf der entsprechenden Registerkarte den Windows-Desktop auf mehrere Monitore

## Zwei Grafikkarten in einem Rechner?

Das geht auch, für den Fall, dass Ihre primäre Grafikkarte nur über einen Monitoreausgang verfügt. Allerdings sollten Sie darauf achten, nicht neuere Modelle gleicher Hersteller zu mischen, da diese oft über Treiber mit gleichem Namen verfügen, die sich deswegen dann ins Gehege kommen. Und: Eine Karte die Sie dazu kaufen, sollte dann auch eine PCI-Karte sein, weil in den meisten Rechnern nur ein AGP-Slot bzw. ein PCI-Express-Slot zu finden ist. PCI-Karten bekommen Sie selbst heutzutage noch bei jedem größerem EDV-Zubehör-Versandhändler.



**Abbildung 4.2:** Die Visual Studio-Oberfläche verteilt auf zwei Monitoren – das lässt richtig Platz für kreative Ergüsse und spart viel Zeit!

## Und die Bildschirmsdarstellung 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 1600 x 1200er Auflösung auf einem 15"-Display nicht zu denken – 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 eine Wide-Screen-Auflösung mit 1650 x 1050 Pixel empfehlen.

Modernere Notebooks bieten aber auch wie neuere Grafikkarten die Möglichkeit, externe Monitore anzuschließen. Sollten Sie also ein größeres Projekt bei einem Kunden vor Ort erledigen müssen, böte sich ein zweites preiswertes TFT-Display mehr als an. Und da 15-Zoll TFTs auch nicht allzu viel wiegen, sind sie für längere »Draußen«-Einsätze allemal geeignet.

---

**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.

---

---

Abgesehen davon, dass »Shared Graphics«-Systeme sich aus dem immer knappen Arbeitsspeicher Ihres Rechners bedienen (aus 512 MByte werden so ruckzuck 486 MByte Speicher, 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.

---

## Zurücksetzen der Fenstereinstellungen

In diesem Zusammenhang: Wenn Sie sich, ob mit einem, zwei oder drei Monitoren, durch Einblenden, Verschieben, Andocken und Entdocken von Toolfenstern ins Fensterchaos gestürzt haben – es gibt immer wieder einen einfachen und schnellen Weg zurück zur ursprünglichen Fensteranordnung: Wählen Sie einfach aus dem Menü *Fenster* den Befehl *Fensterlayout zurücksetzen*, und Sie finden anschließend sämtliche Fenstereinstellungen so vor, wie Sie sie auch direkt nach der Installation von Visual Studio vorgefunden haben.

## Sichern, Wiederherstellen oder Zurücksetzen aller Visual Studio-Einstellungen

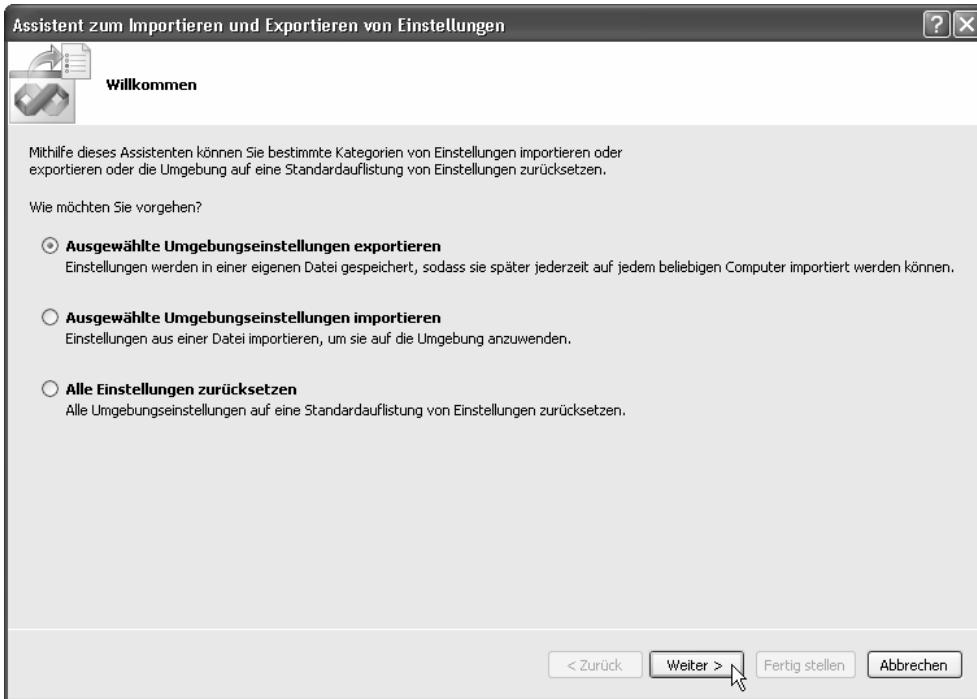
Nachdem Sie sich zum ersten Mal Ihre persönliche IDE-Umgebung so eingerichtet haben, wie Sie sie wirklich haben wollten, wissen Sie, wie viel Aufwand das ist. Visual Studio bietet Ihnen eine einfache Möglichkeit, alle Einstellungen von Visual Studio, die Sie einmal vorgenommen haben, in einer Datei zu speichern, sodass Sie sie beispielsweise auch auf andere Computer übertragen können oder einen Entwicklungsrechner, den Sie neu installieren mussten, schnell wieder in seinen VS-Ausgangszustand zurückversetzen können.

Sie können alle Visual Studio-Einstellungen aber auch in den Ausgangszustand zurückversetzen, die das Programm direkt nach der Installation aufwies.

## Sichern der Visual Studio Einstellungen

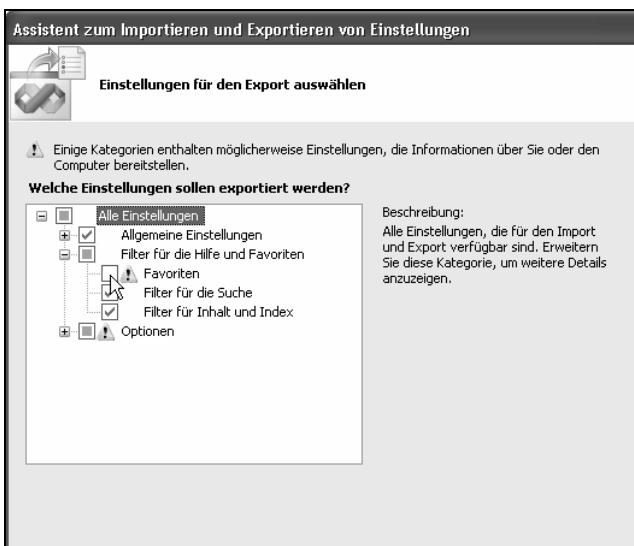
Um die Visual Studio-Einstellungen zu sichern, damit Sie sie auf andere Rechner übertragen oder für die Wiederherstellung desselben Rechners verwenden können, verfahren Sie folgendermaßen:

- Wählen Sie aus dem Menü *Extras* den Menüpunkt *Einstellungen importieren und exportieren*.
- Visual Studio zeigt Ihnen nun einen Assistenten, etwa wie in Abbildung 4.3 zu sehen.



**Abbildung 4.3:** 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 exportiere*, und klicken Sie auf *Weiter*.



**Abbildung 4.4:** Wählen Sie, welche Einstellungen Sie exportieren möchten. Die mit dem Ausrufezeichen versehenen könnten übrigens persönliche Informationen enthalten.

- Wählen Sie die Einstellungen, die Sie exportieren möchten, und klicken Sie anschließend auf *Weiter*.

---

**HINWEIS:** Die mit dem Ausrufezeichen 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.

---

**TIPP:** Wenn Sie alle Einstellungen exportieren wollen, wählen Sie *Alle Einstellungen*. Das könnte u.U. persönliche Einstellungen beinhalten.

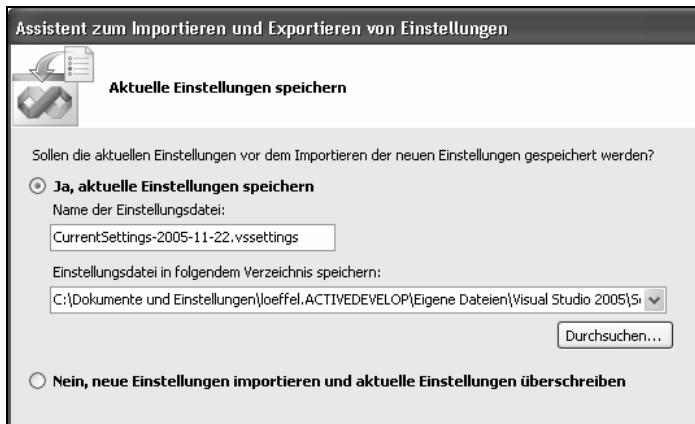
---

- 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 4.3 zu sehen, und dort wählen Sie *Ausgewählte Umgebungseinstellungen importieren*.



**Abbildung 4.5:** 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 4.5 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 zu-

vor gespeicherte Einstellungsdatei für das Importieren auszuwählen, klicken Sie auf die Schaltfläche *Durchsuchen*, die Sie auch in Abbildung 4.6 sehen können.



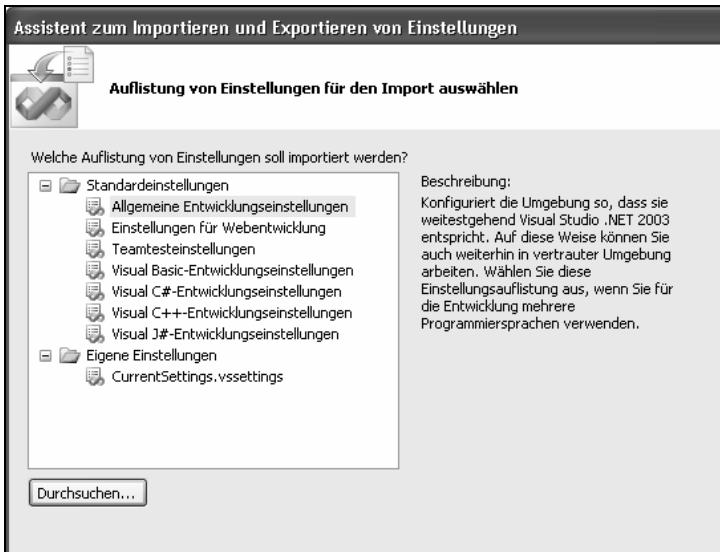
**Abbildung 4.6:** Entscheiden Sie sich an dieser Stelle entweder für eine Reihe standardmäßig vorhandener Einstellungsvorgaben oder für eine Einstellungsdatei, die zuvor exportierte Einstellungsdaten enthält

- Klicken Sie anschließend auf *Fertigstellen*, um den Importvorgang der Einstellungen zu starten. Dieser Vorgang kann eine Weile dauern.

## Zurücksetzen von Visual Studio in den Originalzustand

Sie können alle Visual Studio Einstellungen in den Ausgangszustand zurückversetzen, den es nach der Installation aufwies. Dazu verfahren Sie wie folgt:

- Wählen Sie aus dem Menü *Extras* den Menüpunkt *Einstellungen importieren und exportieren*.
- Visual Studio zeigt Ihnen nun einen Assistenten, etwa wie in Abbildung 4.3 zu sehen.
- Wählen Sie die Option *Alle Einstellungen zurücksetzen*, und klicken Sie auf *Weiter*.
- Sie können vor dem Zurücksetzen aller Einstellungen die aktuellen Einstellungen sichern – dazu wählen Sie im Dialog, den Sie auch in Abbildung 4.5 sehen, den ersten Punkt und geben den Dateinamen und den Speicherort ein. Wählen Sie anderenfalls die zweite Option.
- Klicken Sie auf *Weiter*.
- Entscheiden Sie sich im nächsten Assistentenschritt für eine Reihe standardmäßig vorhandener Einstellungsvorgaben, wie Sie sie auch vor dem ersten Start von Visual Studio ausgewählt haben (siehe Abbildung 4.7).
- Klicken Sie anschließend auf *Fertigstellen*.



**Abbildung 4.7:** Entscheiden Sie sich an dieser Stelle für eine Reihe standardmäßig vorhandener Einstellungsvorgaben, wie Sie sie auch vor dem ersten Start von Visual Studio ausgewählt haben

## Wieviel Arbeitsspeicher darf's denn sein?

Um es ganz klar zu sagen: Die Speichervoraussetzungen, die Microsoft empfiehlt, reichen meiner Meinung nach mit 192 MByte auch auf Windows 2000-Systemen bei weitem nicht aus.<sup>1</sup> Es sind im wahren Wortsinn tatsächlich Mindestanforderungen.

512 MByte sind meiner Meinung nach nötig aber auch ausreichend, wenn Sie ausschließlich mit Visual Studio 2005 arbeiten. Falls Sie Datenbankanwendungen entwickeln möchten und dazu auf SQL Express zurückgreifen, das auf dem gleichen Computer installiert sein soll, sollten Sie am besten 768 MByte oder gleich 1 GByte Hauptspeicher für ein zügiges Arbeiten in Betracht ziehen. Ich arbeite mit 2 GB und denke manchmal immer noch: Mehr wäre auch nicht schlimm.

Wenn Sie sogar Testinstallationen mithilfe von Virtual PC oder ähnlichen Tools auf demselben Rechner durchführen möchten (siehe auch »Testen Ihrer Software unterwegs und zuhause – Microsoft Virtual PC und Microsoft Virtual Server« ab Seite 117), dann sollten sogar 2 GByte im Rechner sein. Und obwohl Speicher vergleichsweise preiswert geworden ist: Mehr als 4 GByte machen unter der 32-bit-Version von Windows XP übrigens keinen Sinn, da Windows XP mit herkömmlichen Mitteln nicht mehr als 3 GByte pro Prozess<sup>2</sup> zur Verfügung stellen kann, und selbst die bekommen Sie nur

<sup>1</sup> Im Übrigen finde ich, dass selbst 256 MByte auf Windows-XP-Systemen ohnehin nur dann ausreichen, wenn Sie nichts anderes außer Windows-XP installieren. Aber dann müssten Sie Ihre Berichte mit WordPad und Ihre Kalkulationen mit dem Taschenrechner durchführen ...

<sup>2</sup> 1 GByte wird dann als Kernel-Speicher – also für das gemeinsam benutzte Betriebssystem – verwendet; und auch von dem kann u.U. nicht alles verwendet werden, da sich beispielsweise der eigene Grafikspeicher der Grafikkarte ebenfalls in diesem Bereich befindet.

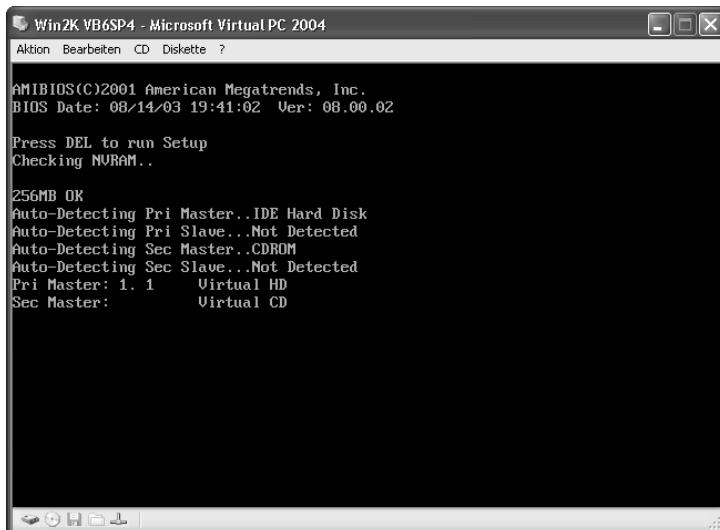
dann, wenn Sie die *boot.ini*-Datei von Windows XP modifizieren.<sup>3</sup> Außerdem können die meisten handelsüblichen Motherboards für 32-Bit-Windows-Systeme ebenfalls nicht mehr als 4 GByte Hauptspeicher verwalten.

## Testen Ihrer Software unterwegs und zuhause – Microsoft Virtual PC und Microsoft Virtual Server

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, 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 XP entwickelt haben, läuft Sie auch unter Windows 2000? Wie sieht es aus mit Vista oder Windows 2003?
- 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!



**Abbildung 4.8:** 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

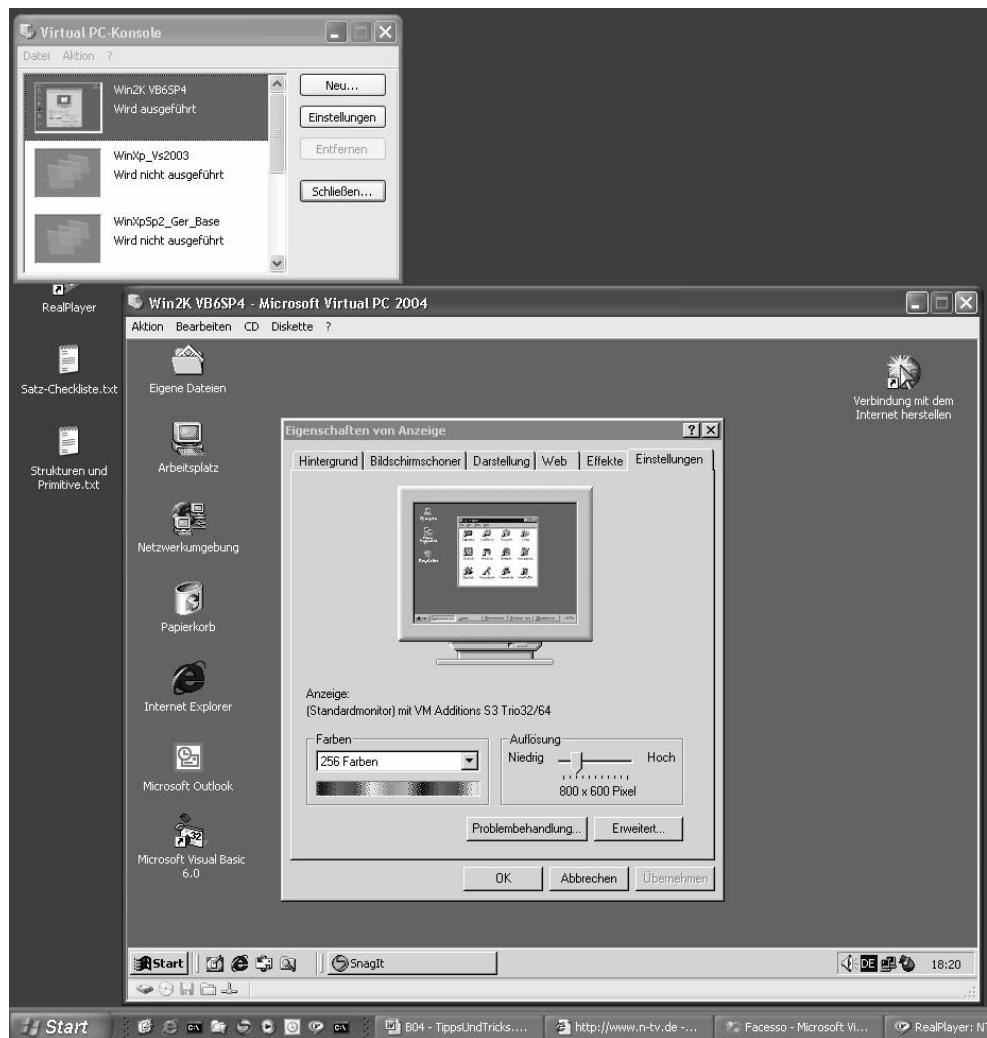
---

<sup>3</sup> Googeln Sie dazu am besten mal nach +“/3GB Switch“ +“Windows XP“, um die vielen verfügbaren Hinweise zum Thema zu bekommen.

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 4.9:** ... um Sekunden später ein voll funktionsfähiges und durchaus schnell laufendes Windows 2000 zu präsentieren, mit Windows XP 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 4.8 und Abbildung 4.9 demonstrieren dieses Konzept, 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 Softwarelizenzen benötigen, als handelte es sich bei diesen um »echte« PCs. Microsoft greift Ihnen aber gerade als Softwareentwickler mit dem MSDN-Abo<sup>4</sup> 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.

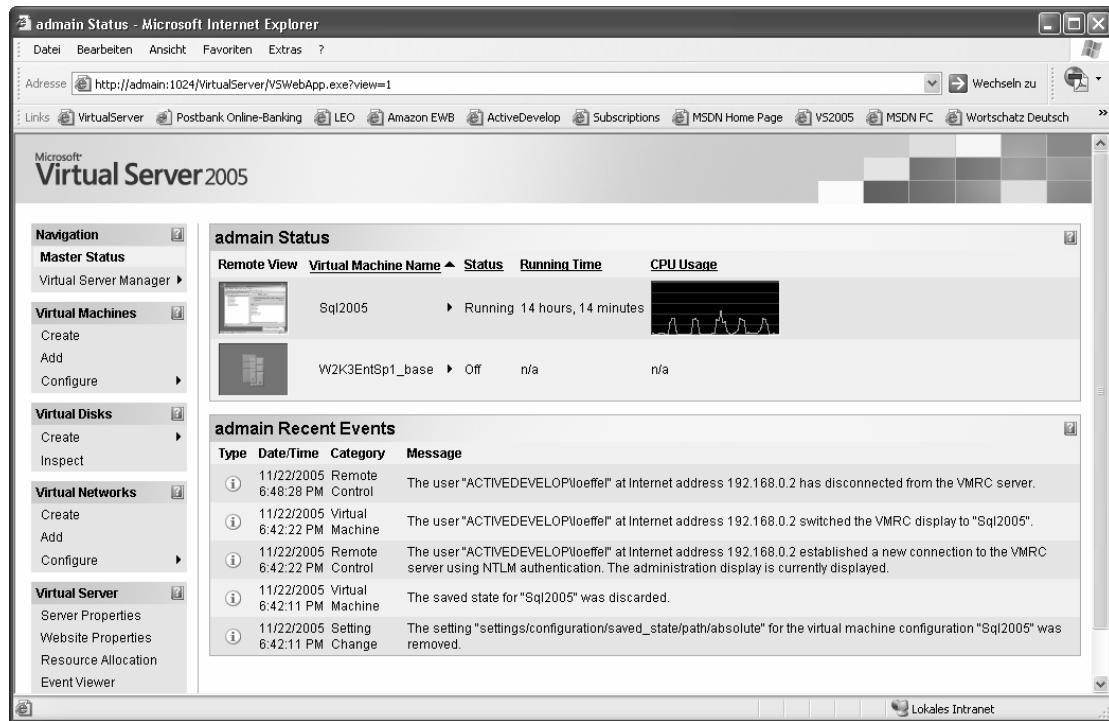
---

Wie in Abbildung 4.9 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. Eine kostenlose Testversion finden Sie schnell auf der Microsoft Webseite.

Ein etwas anderes Konzept verfolgt Virtual Server 2005 von Microsoft.

---

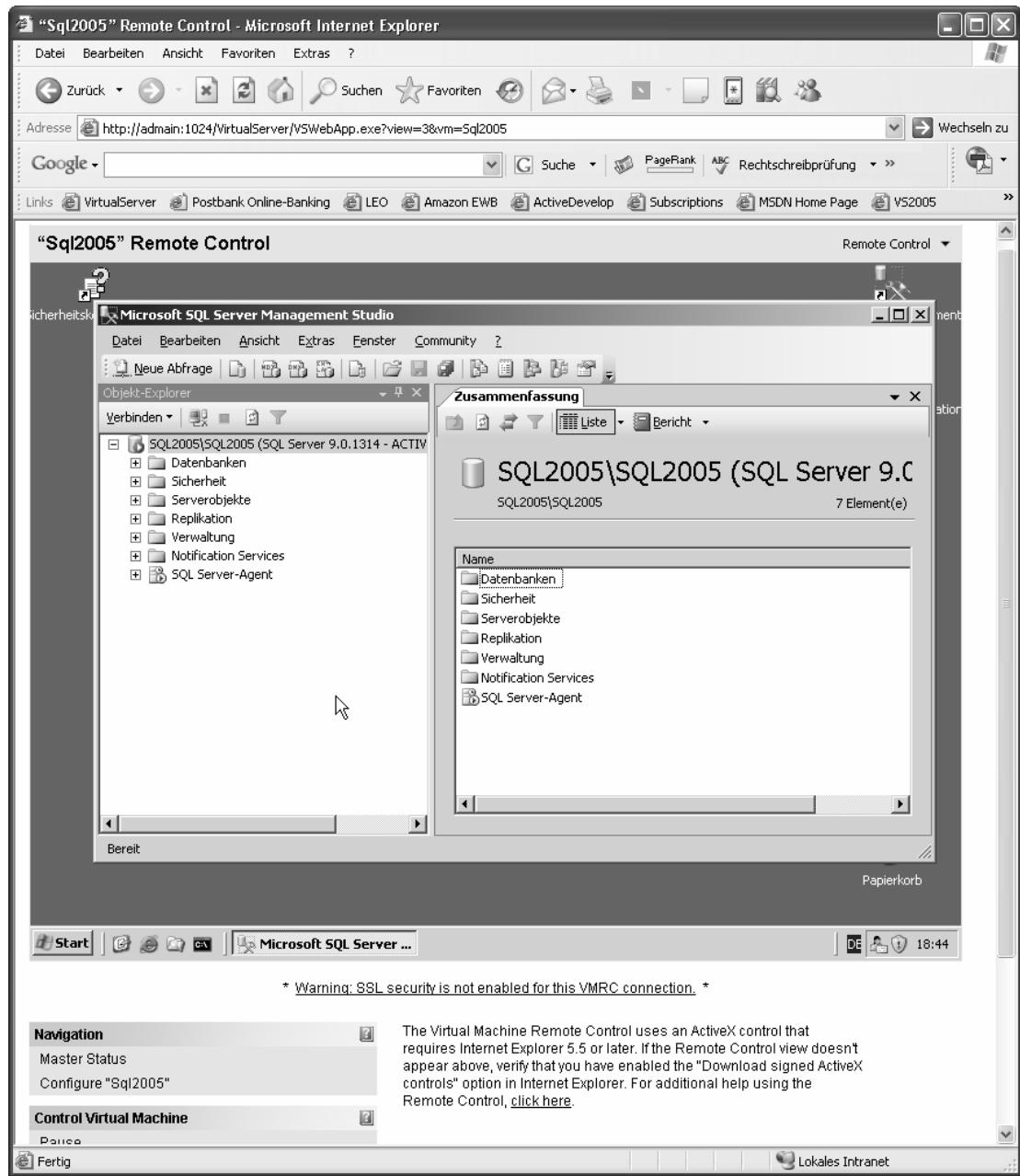
<sup>4</sup> Hinweise zu den verschiedenen MSDN-Abos gibt es unter dem IntelliLink *B0401*.



**Abbildung 4.10:** Alle auf einem Server installierten virtuellen Server werden von jedem beliebigen Client des Netzwerkes aus auf denkbar einfachste Weise administriert: über den Internet-Explorer ...

## Virtual Server 2005

Wie Virtual PC dient auch Virtual Server 2005 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.



**Abbildung 4.11:** ... 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. 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.

## 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, wissen wo es steht!

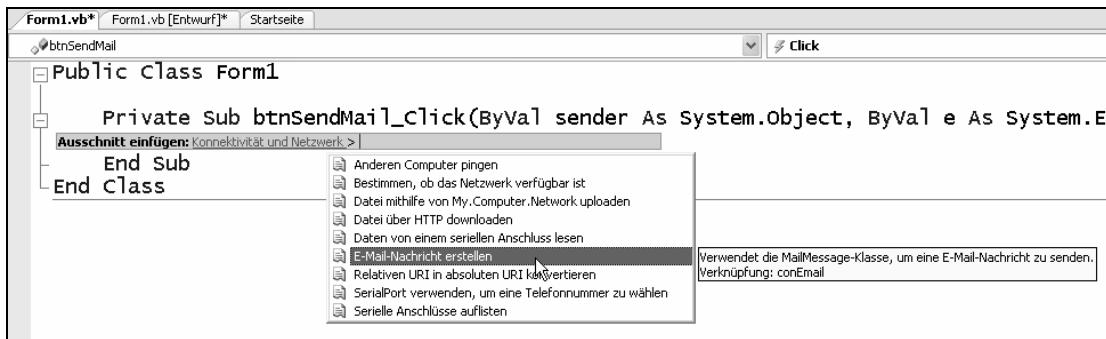
Bei rund über 8.000 verschiedenen Klassen können Sie unmöglich, auch mit jahrelanger Erfahrung, die genaue Funktionsweise jeder dieser Klassen gelernt haben und direkt aus dem Stegreif anwenden. Was Sie allerdings lernen und perfektionieren können, ist so schnell wie möglich an die gewünschten Informationen zu kommen.

Die Kombination der Hilfsmittel *Codeausschnittsbibliothek*, *Vervollständigungsliste von IntelliSense*, *dynamische Hilfe* und *Autokorrektur für automatisches Kompilieren* 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 im letzten Kapitel 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 im letzten Kapitel 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 4.12:** Über das Editor-Kontextmenü erreichen Sie zunächst die Codeausschnittbibliothek, in der Sie zunächst 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 dieser 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 4.13:** 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 Instanzierungsanweisung für das `SmtpClient`-Objekt `emailClient` fahren, dann stellt die dynamische Hilfe im gleichen Moment einige Querverweisvorschläge zur Verfügung, wie in Abbildung 4.13 zu sehen. Und dort sehen wir als Hilfetopic beispielsweise den Dokumentationsverweis zur `SmtpClient`-Klasse.

**SmtpClient-Klasse**

Siehe auch [Beispiel](#) [Member](#)

Alle Ebenen reduzieren Sprachfilter: Visual Basic

**MailAddress** Stellt die E-Mail-Adresse des Absenders und des Empfängers dar.

**MailMessage** Stellt eine E-Mail-Nachricht dar.

Um mit **SmtpClient** eine E-Mail-Nachricht zu erstellen und zu senden, müssen Sie die folgenden Informationen angeben:

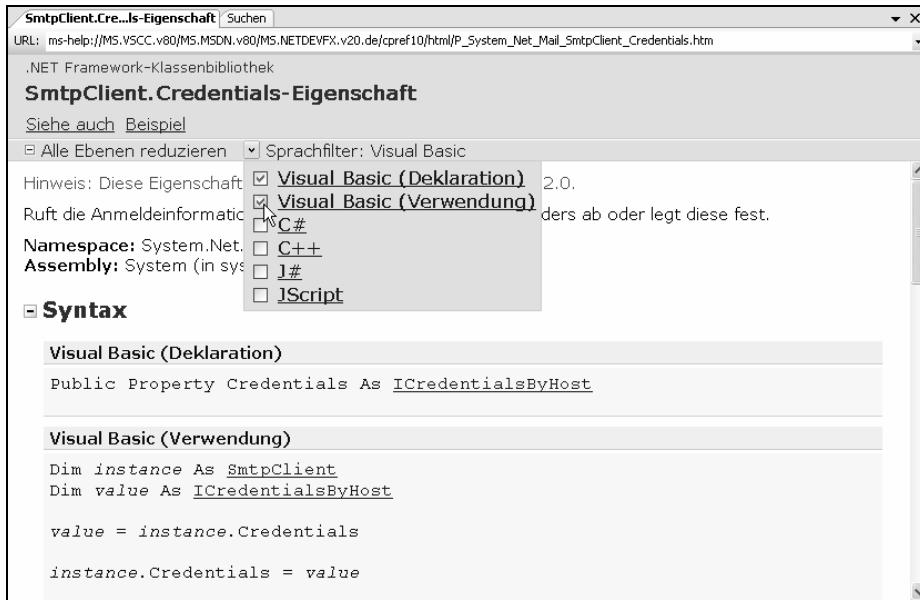
- Der SMTP-Hostserver, den Sie zum Senden von E-Mail-Nachrichten verwenden. Siehe [Host](#) und [PortProperties](#).
- Anmeldeinformationen für die Authentifizierung, sofern für den SMTP-Server erforderlich. Siehe die [Credentials-Eigenschaft](#).
- Die E-Mail-Adresse des Absenders. Siehe die [Send](#)-Methode und die [SendAsync](#)-Methode, die einen `from`-Parameter verwenden. Siehe auch die [MailMessage.From](#)-Eigenschaft.
- Die E-Mail-Adressen der Empfänger. Siehe die [Send](#)-Methode und die [SendAsync](#)-Methode, die einen `recipient`-Parameter verwenden. Siehe auch die [MailMessage.To](#)-Eigenschaft.
- Der Nachrichteninhalt. Siehe die [Send](#)-Methode und die [SendAsync](#)-Methode, die einen `body`-Parameter verwenden. Siehe auch die [MailMessage.Body](#)-Eigenschaft.

Um eine Anlage in eine E-Mail-Nachricht einzufügen, erstellen Sie zunächst die Anlage mit der **Attachment**-Klasse, und fügen Sie sie dann mit der [MailMessage.Attachments](#)-Eigenschaft zur Nachricht hinzu. Je nach dem von den Empfängern verwendeten E-Mail-Reader und dem Dateityp der

**Abbildung 4.14:** 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 4.14 sieht man es exemplarisch. Es findet sich tatsächlich ein Hinweis auf »Anmeldeinformationen für die Authentifizierung«, und ein weiterer Klick auf den in diesem Zu-

sammenhang stehenden Link bringt uns noch einen Schritt in der Recherche-Kette weiter: Der Link zur Dokumentation der Credentials-Eigenschaft.



**Abbildung 4.15:** 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 4.15 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.

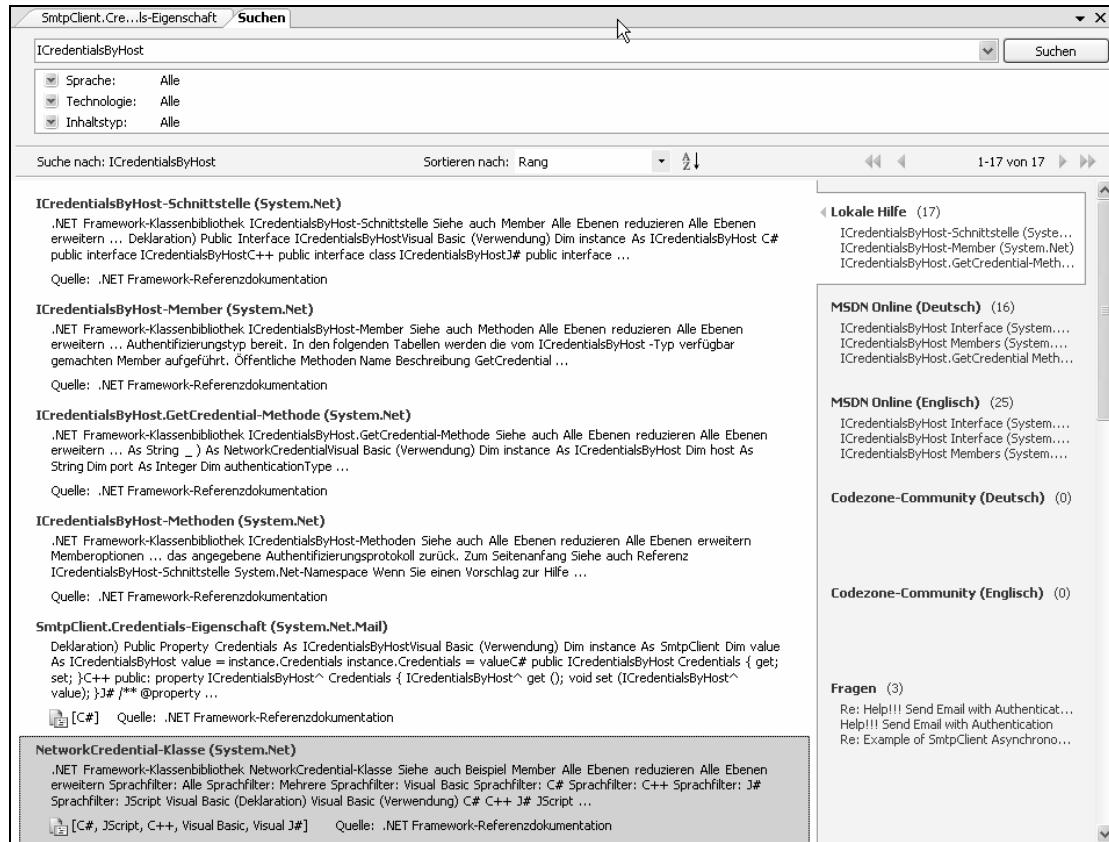
Gleichzeitig ist es auch hilfreich, sich mit möglichst allen Techniken im objektorientiert arbeitenden Visual Basic auszukennen. So werden Sie später wissen, nachdem Sie den OOP-Teil dieses Buches durchgearbeitet haben, dass die so genannten Interfaces (Schnittstellen) Vorschriften für erst eigentlich 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 Buches 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 2005 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:



**Abbildung 4.16:** 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 Anmelddaten werden im Klartext übertragen!
        'Dieses Verfahren also nur bei Nicht-AD-Clients im Intranet anwenden!
        Dim myCred As New NetworkCredential("k_loeffel", "passwort", "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

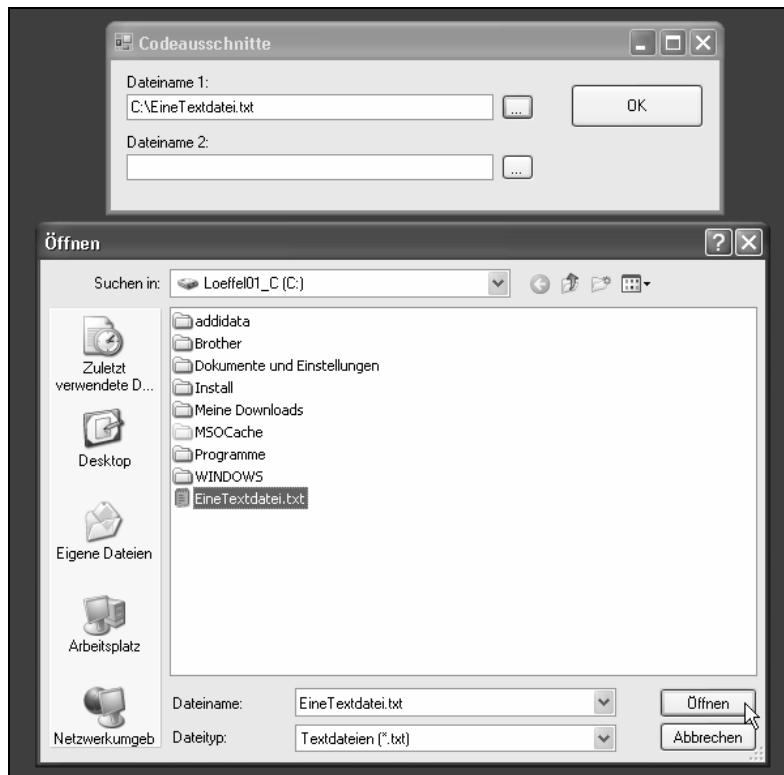
Es gibt bei der Entwicklung von Projekten immer wiederkehrende Dinge, die oftmals in pure Fließbandarbeit ausarten. Was mich persönlich beispielsweise am meisten nervt, ist das Programmieren von Dateiauswahl-Dialogen. Der Benutzer muss auf eine Schaltfläche klicken, daraufhin öffnet sich ein Datei-Öffnen-Dialog, 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 VB2005 demonstriert, finden Sie unter .\VB 2005 - Entwicklerbuch\B - IDE\04 – TippsUndTricks\Codeausschnitte unter dem Projektnamen *Codeausschnitte.sln*.

---

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 4.17:** Ein Vorgang, den Sie in Ihrem Leben sicherlich schon zigmals programmieren mussten: das Zur-Verfügung-Stellen von Dateiauswahl-Dialogen

In Visual Basic 2005 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 btnDateiAuswählen.Click

    'Eine neue OpenFileDialog-Klasse instanzieren
    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-Variablen nimmt das Dialogergebnis (OK, Abbrechen) entgegen
        Dim dialogErgebnis As DialogResult = .ShowDialog
    End With

```

```

'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 VB2005 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 – so was 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/2005/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 2005 - 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. Wenn Sie mit dieser Aussage im Moment noch nichts anfangen können, empfehle ich die Lektüre des nächsten Kapitels, das sich gerade für VB6-Umsteiger eignet, um sich die Basics zu Assemblies und Assembly-Verweisen anzueignen. 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>
```

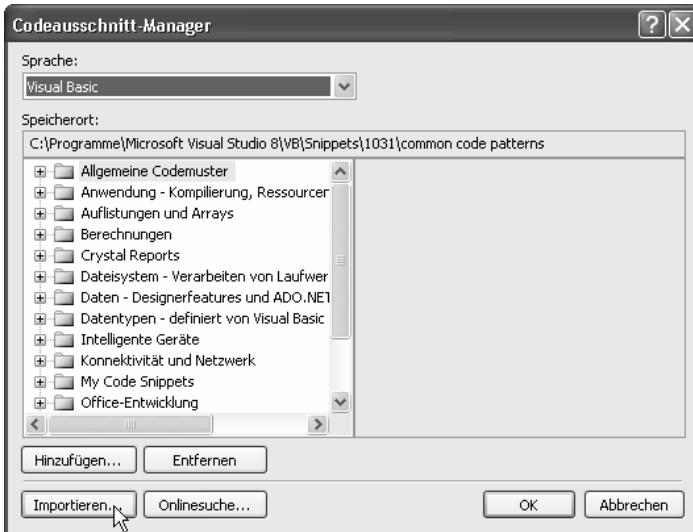
Und damit ist die erste einfache Code Snippet-Vorlage bereits fertig gestellt. Speichern Sie diese jetzt noch unter einen anderen Namen ab (*Datei | Vorlage.Snippet speichern unter...*) – beispielsweise unter *DateiÖffnenDialog.Snippet*.

Als nächstes müssen wir Visual Studio das neue Snippet noch beibringen. Wie das geschieht, zeigt der folgende Abschnitt.

## Hinzufügen einer neuen Snippet-Vorlage zur Snippet-Bibliothek (Codeausschnittsbibliothek)

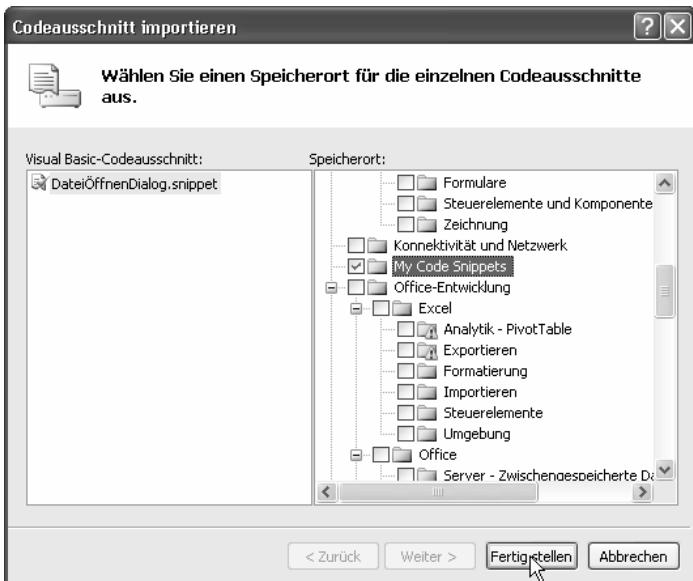
Für diese Zwecke kennt Visual Studio den so genannten Codeausschnitt-Manager, den Sie über das *Extras*-Menü öffnen können.

- Klicken Sie auf den entsprechenden Menüpunkt, erscheint ein Dialog, wie Sie ihn auch in Abbildung 4.18 sehen können.



**Abbildung 4.18:** 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 4.19:** 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 2. 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*.

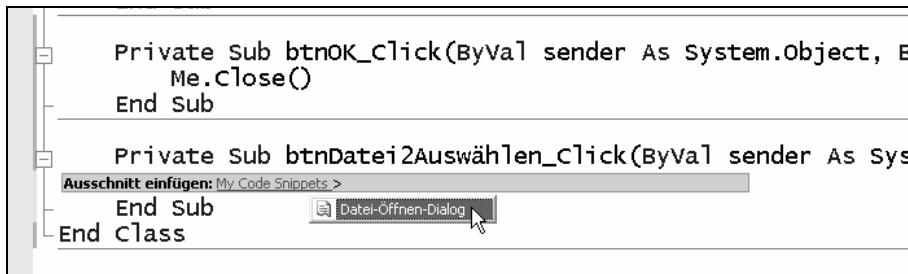


Abbildung 4.20: Der neue Codeausschnitt steht jetzt zum Einfügen in der Codeausschnittsbibliothek zur Verfügung

- 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 uns 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 **Tabulator** 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 **Tabulator**-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 **Tabulator** 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:

```

Dim dateiöffnenDialog As New OpenFileDialog
With dateiöffnenDialog
    .CheckFileExists = True
    .CheckPathExists = True
    .DefaultExt = "*.bmp" ← Ein Ändern einer literalen Ersetzung
    .Filter = "Textdateien" & " (" & "*.txt" & ")" | "*" & ".txt" & "|Alle
End With

Dim dateiöffnenDialog As New OpenFileDialog
... bewirkt automatisch die Änderung an
With dateiöffnenDialog
    .CheckFileExists = True      Stellen, an denen die gleiche literale
    .CheckPathExists = True      Ersetzung in der Vorlage verwendet wurde.
    .DefaultExt = "*.bmp" ←
    .Filter = "Textdateien" & " (" & "* bmp" & ")" | "*" & ".bmp" & "|Alle
Dim dialogErgebnis As DialogResult = .ShowDialog
If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
    Exit Sub
End If
End With

```

**Abbildung 4.21:** 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/2005/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 (*.*)|*.*"
        End With
      ]]>
    </Code>
  </Snippet>
</CodeSnippets>
```

```

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 Fettschrift 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 Beispiellisting zu sehen, mit dem <Literal>-Tag ein. Anschließend folgt die Definition der literalen Ersetzung durch den <ID>-Tag, das dem Kind einen Namen gibt – in diesem Beispiel *Standardfilter*. Anstelle also später, im Code, direkt vordefinierten Text zu verwenden, setzen Sie später 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 Tabulator-taste erreichen lässt. Unsere XML-Datei erfährt also einen weiteren »Umbau«:

```

<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets
  xmlns="http://schemas.microsoft.com/VisualStudio/2005/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 verwendende 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 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 verwendende 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 verwendet.

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 Beispieldialog zu sehen. Wiederum genau so 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:

```

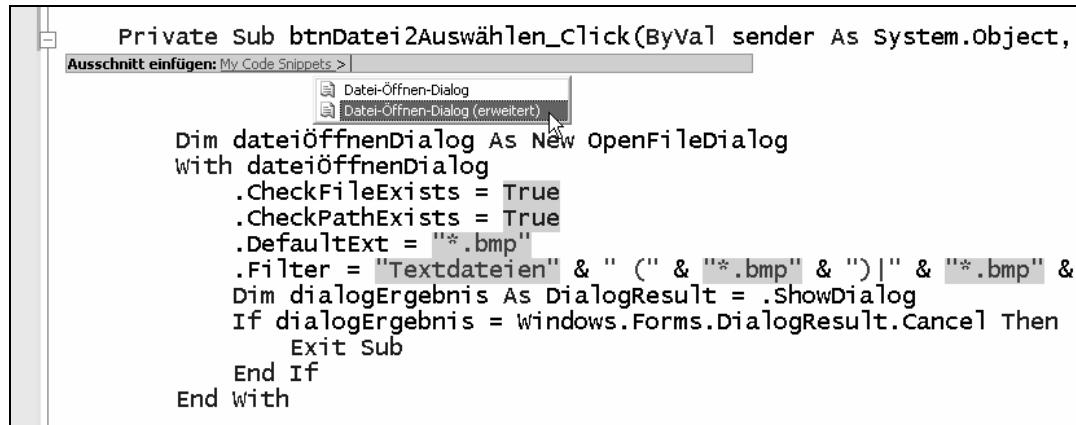
.
.
.

<!-- 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 4.22:** 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 *DateiÖffnenDialog-Ex.snippet*. Falls nicht, finden Sie die Vorlage unter dem gleichen Namen im Projektverzeichnis des Beispieldokumentes.

Und dann wiederholen Sie das Spielchen aus Abschnitt »Hinzufügen einer neuen Snippet-Vorlage zur Snippet-Bibliothek (Codeausschnittsbibliothek)« ab Seite 131.

Das Einfügen des neuen Codeausschnittes und der eingefügte Ausschnitt selbst sollten anschließend so ausschauen, wie es Abbildung 4.22 zeigt.

## **Teil C**

# **Der Umstieg auf Visual Basic 2005 ...**

---

**143            Der Umstieg von Visual Basic 6.0**

**185            Der Umstieg von Visual Basic.NET 2002 oder 2003**

---

Mit einiger Überraschung musste ich im Verlauf der letzten Monate feststellen, dass auch große Firmen mit einigermaßen großen Entwicklungsabteilungen tatsächlich auf das Erscheinen von Visual Basic 2005 gewartet haben, um den »Transit« in die .NET-Welt zu vollziehen. Als ich das Umsteigerkapitel zum Vorläufer dieses Buches schrieb, war ich der Meinung, ein Kapitel mit nicht so langem »Haltbarkeitsdatum« zu kreieren. Ich hatte Recht, aber anders, als ich es dachte.

Beide Kapitel in diesem Buchteil sind nämlich weitgehend neu. Sie konzentrieren sich mit kleinen Beispielen auf das Wesentliche, was Sie als Allererstes wissen müssen, um nicht auf Überraschungen bei Ihren ersten Projekten mit VB2005 zu stoßen. Und sie bauen aufeinander auf, was nicht zuletzt der Grund dafür war, das VB6-Umsteigerkapitel nicht 1:1 aus dem Vorgängerwerk dieses Titels zu verwenden.

Kommen Sie also aus der VB6-Welt, dann lesen Sie am besten beide Kapitel. Haben Sie sich jedoch zuvor schon in Visual Basic .NET (Versionen 2002 oder 2003) eingearbeitet, können Sie das nächste Kapitel vielleicht überblättern und direkt einen Blick in das übernächste werfen – wobei ein Blick in das erste Kapitel dieses Teils mit Sicherheit auch nicht schaden kann, um das eine oder andere aufzufrischen oder sogar was Neues zu entdecken.



# 5 Der Umstieg von Visual Basic 6.0

---

- 
- 144 Unterschiede in der Variablenbehandlung**
  - 158 Alles ist ein Objekt oder »let Set be«**
  - 159 Direktdeklaration von Variablen in For-Schleifen**
  - 161 Gültigkeitsbereiche von Variablen**
  - 165 Die Operatoren += und -= und ihre Verwandten**
  - 167 Fehlerbehandlung**
  - 174 Kurzschlussauswertungen mit OrElse und AndAlso**
  - 175 Variablen und Argumente auch an Subs in Klammern!**
  - 176 Namespaces und Assemblies**
- 

Um es vorweg zu nehmen: 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.

Doch eines möchte ich ohne Umschweife sagen: Wenn Sie planen, eine größere Entwicklung, die Sie ursprünglich unter VB6 vorgenommen haben, auf .NET 2.0 zu portieren, tun Sie sich, wenn es Budget und Zeit erlauben, selber den Gefallen, und überlegen Sie sich eine Strategie, komplett von vorn zu beginnen, und lassen Sie den Upgrade-Wizard einfach außen vor. Er ist sicherlich mit Überlegung entwickelt und für das eine oder andere sinnvoll einsetzbar. Doch kann er leider keine Entwicklungs-konzepte anpassen, und so wird er eine in VB6 nach prozeduralen Vorgehensweisen entwickelte Lösung bestimmt nicht in ein objektorientiertes Modell umwandeln können. Doch genau das sollte Ihr Ziel bei einer Migration gerade bei größeren Softwareentwicklungen sein.

Sollte das aus zeit- und geldabhängigen Gründen vorläufig unmöglich erscheinen, überlegen Sie sich am besten eine Strategie, wenigstens einzelne Module nach und nach zu ersetzen, beide Welten für eine Weile nebeneinander existieren zu lassen und diese neuen Module später zu einem vollständig migrierten Gesamtpaket zu verschnüren.

Und über etwas Weiteres sollten Sie sich im Klaren sein: Visual Basic 2005 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 mit dem .NET-Framework 2.0 rund 8.000 Klassen zur Lösung der unterschiedlichsten Aufgaben zur Verfügung

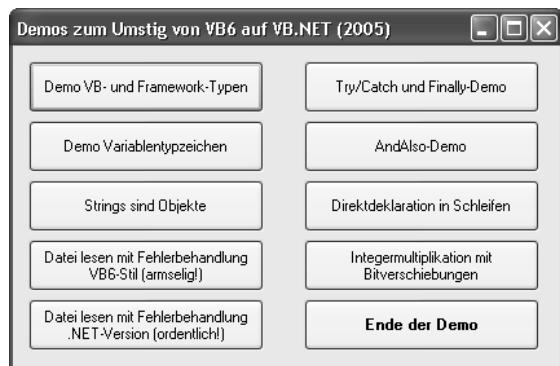
stehen, möchte ich VB2005 nicht nur als einfache Weiterentwicklung von VB6 bezeichnen. Wenn Sie mit der Vorstellung an das Erlernen von VB2005 herangehen, dass Sie im Grunde genommen eine komplett neue Sprache lernen, kommt das 1. nicht nur der Realität recht nahe, sondern Sie werden 2. auch schnelle 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 VB6 und VB2005 gibt, die aber dann doch gar keine ist, soll dieses Kapitel klären. Nicht mehr und nicht weniger. Die eigentlichen Neuerungen werden Sie als VB6-Programmier dann im nächsten Buchteil erfahren.

---

**BEGLEITDATEIEN:** Übrigens: Die abgedruckten Codeausschnitte dieses Buchs vereinigt ein Projekt, das Sie im Pfad .\VB 2005 - Entwicklerbuch\C - Ein- und Umstieg\VonVb6Zu2005 unter dem Projektmarkennamen *VonVb6Zu2005.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+O** anzeigen. Kleinere Codeschnipsel, VB6-Code oder Code, der zu Demonstrationszwecken mit Absicht Fehler enthält, sind dort natürlich nicht berücksichtigt.

---



**Abbildung 5.1:** 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« Variablentypen 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 VB2005 nicht mehr gibt und viele neue hinzugekommen sind, haben sich einige Variablentypen auch verändert.

## **Integer- und Short-Variablen**

Integer-Variablen verfügen nunmehr über einen 32-Bit-breiten vorzeichenbehafteten Darstellungsbereich, anders als in VB6, wo sie mit 16-Bit auskommen mussten.

Damit stellen sie in VB2005 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 VB2005 durch den Datentyp Short dargestellt – Short in VB2005 entspricht also Integer in VB6.

## **Long-Variablen**

Das was Long-Variablen in VB6 waren sind sie jetzt in Form von Integer in VB2005. Long gibt es aber auch in VB2005, nur basiert er hier auf 64 und nicht auf 32 Bit. Sein 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.

## **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.

## **Typen, die es nicht mehr gibt ...**

Von einigen Variablentypen hat sich VB2005 (wie VB.NET 2002, 2003 auch schon) 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.<sup>1</sup> 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 Buches.

| Typename                                                                                      | .NET-Typname  | Aufgabe                                                                | Wertebereich                     |
|-----------------------------------------------------------------------------------------------|---------------|------------------------------------------------------------------------|----------------------------------|
| <b>Byte</b>                                                                                   | System.Byte   | Speichert vorzeichenlose Integer-Werte mit 8 Bit Breite                | 0 bis 255                        |
| <b>SByte</b>                                                                                  | System.SByte  | Speichert vorzeichenbehaftete Integer-Werte mit 8 Bit Breite           | -127 bis 128                     |
| <b>Short</b>                                                                                  | System.Int16  | Speichert vorzeichenbehaftete Integer-Werte mit 16 Bit (2 Byte) Breite | -32.768 bis 32.767               |
| <b>UShort</b>                                                                                 | System.UInt16 | Speichert vorzeichenlose Integer-Werte mit 16 Bit (2 Byte) Breite      | 0 bis 65.535                     |
| <b>Integer</b>                                                                                | System.Int32  | Speichert vorzeichenbehaftete Integer-Werte mit 32 Bit (4 Byte) Breite | -2.147.483.648 bis 2.147.483.647 |
| <b>HINWEIS:</b> Auf 32-Bit-Systemen ist dies der am schnellsten verarbeitete Integer-Datentyp |               |                                                                        |                                  |
| <b>UInteger</b>                                                                               | System.UInt32 | Speichert vorzeichenlose Integer-Werte mit 32 Bit (4 Byte) Breite      | 0 bis 4.294.967.295              |

---

<sup>1</sup> 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-Typname    | Aufgabe                                                                                                                                                                                                                                                   | Wertebereich                                                                                                                                                                                                                                                |
|----------|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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.<br><b>HINWEIS:</b> 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.<br><b>HINWEIS:</b> 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 ein 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                                                                                                                                                                                                                           | Variabile Länge. 0 bis ca. 2 Milliarden Unicode-Zeichen                                                                                                                                                                                                     |

**Tabelle 5.1:** Die primitiven Datentypen in .NET 2.0 bzw. VB2005

**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 = #12/24/2005 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 Variablenotypzeichen

Anders als in VB6 können Variablen gleichen Typs in einem Rutsch deklariert werden.

Darüber hinaus gibt es in VB2005, wie in VB6, Variablenotypzeichen. 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 btnVariablenotypzeichen_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles btnVariablenotypzeichen.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 Variablenarten in Klartext ausgeben:
        Debug.Print("locInt1 ist: " & locInt1.GetType.ToString)
        Debug.Print("locInt2 ist: " & locInt2.GetType.ToString)
    End Sub

```

```

        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 Variablen-deklaration automatisch den zu verwendenden Datentyp festgelegt haben (also beispielsweise DefInt A-Z, um alle Variablen, 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 VariablenTypen explizit angeben. Typzeichen sind meiner Meinung nach Relikte aus alter Zeit und eigentlich überflüssig. Im Übrigen gibt es sie bis auf eine kleine Ausnahme<sup>2</sup>, in keiner anderen »Erwachsenenprogrammiersprache« (wie C, C++, C# oder Java), und wir wollen ja schließlich alle, das 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 darüber sofort Auskunft.

---

'Und das hier auch:  
Dim a%, b&, c!, d!, e#, f\$  
f\$ = "Muss w**Dim c As Decimal** sein!"

**Abbildung 5.2:** 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 einklammt.

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 VariablenTypzeichen für einen bestimmten Typ der Tabelle geben, finden Sie diese ebenfalls in der Tabelle angegeben.

| Typename      | Typzeichen | Typliteral | Beispiel                      |
|---------------|------------|------------|-------------------------------|
| <b>Byte</b>   | –          | –          | Dim var As Byte = 128         |
| <b>SByte</b>  | –          | –          | Dim var As SByte = -5         |
| <b>Short</b>  | –          | S          | Dim var As Short = -32700S    |
| <b>UShort</b> | –          | US         | Dim var As UShort = 65000US ► |

---

<sup>2</sup> In C# 2005 gibt es die Möglichkeit, einen primitiven Datentyp auf sein nullbares Pendant durch ein Typzeichen festzulegen.

| Type name | Type Zeichen | Type literal                                              | Beispiel                                                                                                                  |
|-----------|--------------|-----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Integer   | %            | I                                                         | Dim var% = -123I oder<br>Dim var As Integer = -123I                                                                       |
| UInteger  | -            | UI                                                        | Dim var As UInteger = 123UI                                                                                               |
| Long      | &            | L                                                         | Dim var& = -123123L oder<br>Dim var As Long = -123123L                                                                    |
| ULong     | -            | UL                                                        | Dim var As ULong = 123123UL                                                                                               |
| Single    | !            | F                                                         | Dim var! = 123.4F oder<br>Dim var As Single = 123.4F                                                                      |
| Double    | #            | R                                                         | Dim var# = 123.456789R oder<br>Dim var As Double = 123.456789R                                                            |
| Decimal   | @            | D                                                         | Dim var@ = 123.456789123D oder<br>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<br>#dd/mm/yyyy hh:mm:ss am/pm# | Dim var As Date = #12/24/2005 04:30:15<br>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<br>Dim var As String = "Zeichenkette"                                                     |

**Tabelle 5.2:** Typliterale und Variablenotypzeichen der primitiven Datentypen in Visual Basic 2005

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 Zei-

chenkette behandelt wird, und eben da es sich 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 Nachkomma- 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:

```
Dim einAndererString As String  
einAndererString = 1.23
```

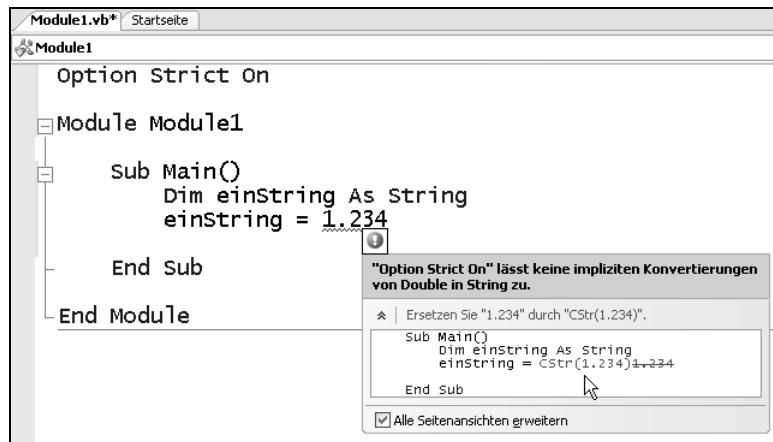


Abbildung 5.3: Typsicherheit unter .NET verlangt, dass implizit nur gleiche oder sichere Typen einander zugewiesen werden können

Die Autokorrektur für intelligentes Kompilieren 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«?

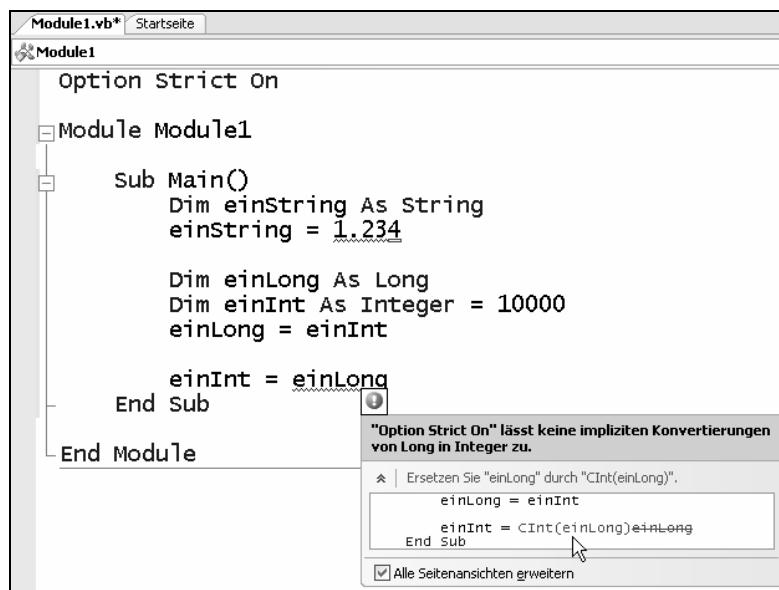
Bei den Strings haben wir es bereits schon im oben stehenden VB6-Beispiel erlebt. Diese implizite Konvertierung war nicht sicher. Auf einem amerikanischen System hätte man ein anderes Resultat als auf einem deutschen System gehabt. Die implizite Konvertierung ist also nicht sicher, weil es offensichtlich 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.

Andersherum sieht es schon wieder anders aus, wie die folgende Abbildung zeigt:



**Abbildung 5.4:** 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« ein Char-Typ machen, und das erreichen Sie durch das Hintenanstellen eines Typliterals, wie die folgende Abbildung zeigt:

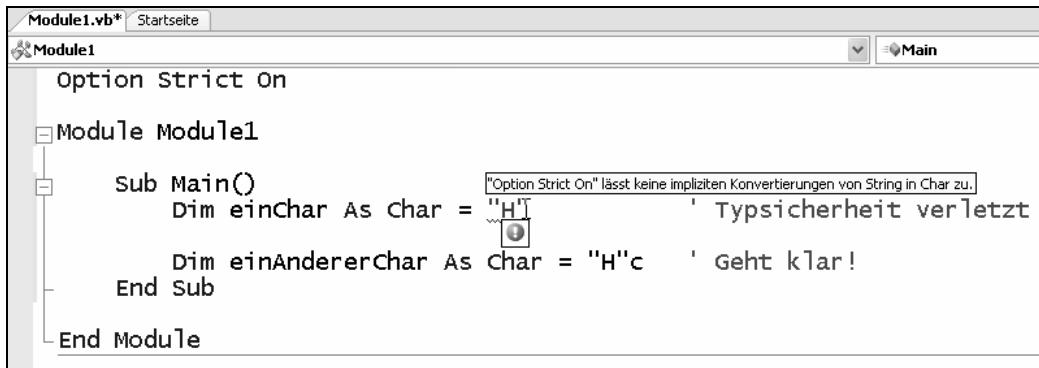


Abbildung 5.5: Mit Typliteralen zwingen Sie Konstanten in einen bestimmten Typ

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 = 1.231231231231D

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 bemerkt: Anders als in VB6 können Variablen in VB2005 (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 VB2005 auch für Instanzierungen<sup>3</sup> 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 nächste Abschnitt eingeht.

---

## 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 2002, 2003 und 2005) 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

---

<sup>3</sup> 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.

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 ändern:

#### 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. Deklarieren und instanzieren Sie eine Objektvariable in VB6 in einem Rutsch, wird sie niemals Nothing werden können!

---

**WICHTIG:** Der VB6-Compiler sorgt beim Deklarieren und Instanziieren von Objektvariablen in einer Zeile immer dafür, dass diese mit New (»unter der Haube« sozusagen) neu instanziert werden, sollten sie einmal Nothing geworden sein.  
**Das ist in VB.NET nicht so.**

---

## 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 so genannte statische<sup>4</sup> 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 folgende 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 um Methoden und Eigenschaften – etwa, wie hier zu sehen, über eine `Substring`-Methode.

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.

---

<sup>4</sup> *Statisch* deswegen, weil Sie keine definierte Variable brauchen, um die Funktion bzw. Eigenschaft verwenden zu können, sondern Ihnen diese direkt über den Typnamen immer (statisch) zur Verfügung steht.

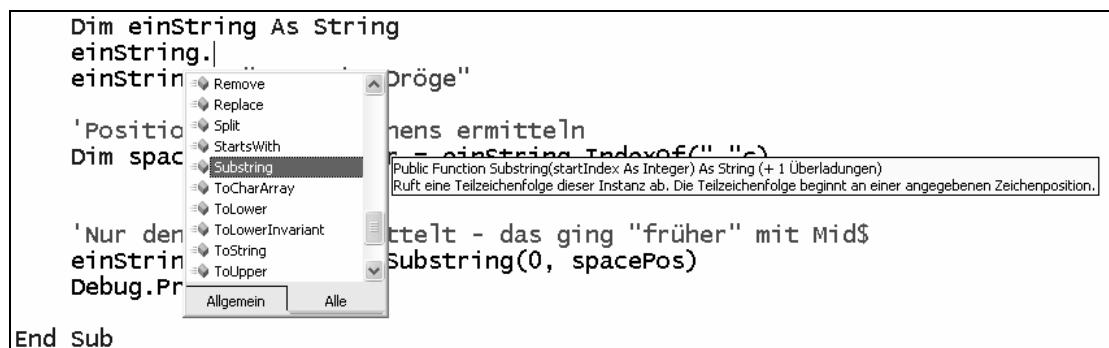
Reine Grunddatentypen wie in Visual Basic 6.0 gibt es im Framework.NET nur noch per Definition. In .NET heißen diese dann auch »primitive Datentypen«, und wie alles in .NET handelt es sich im Grunde genommen auch bei ihnen um Objekte. Und da man, hätte man das alte Konzept zur Verarbeitung von Objekten 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.

---

**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 5.6:** Wenn Sie wissen wollen, welche Methoden und Eigenschaften ein primitiver Variabtentyp zur Verfügung stellt – deklarieren Sie ihn, und benutzen Sie IntelliSense für eine Funktionsübersicht

## 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 161.

---

### 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
```

## Unterschiede bei verwendbaren Variabtentypen 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 entstandenen 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 Objekt 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.

## 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 Codefiles, 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.
- Blockvariablen, die innerhalb eines If-Blocks, einer For/Next- oder Do/Loop- oder While/Wend-Schleife gültig sind.

## 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 komplettete 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 Buches durchgearbeitet 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 Ihrer Klasse zugreifen:

### Achtung! VB6-Code!

```
Private einIntergerMember As Integer
```

```
Private Sub Command1_Click()
    'Von hieraus ist dranzukommen
    einIntergerMember = 10
End Sub
```

```
Private Sub Command2_Click()
    'Von hieraus auch
    einIntergerMember = 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.

## Gültigungsbereiche 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 Gültigungsbereich von Variablen, die in diesem Block definiert sind. Ein Beispiel zeigt Abbildung 5.7.

```

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 5.7:** 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 Fettschrift 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

```

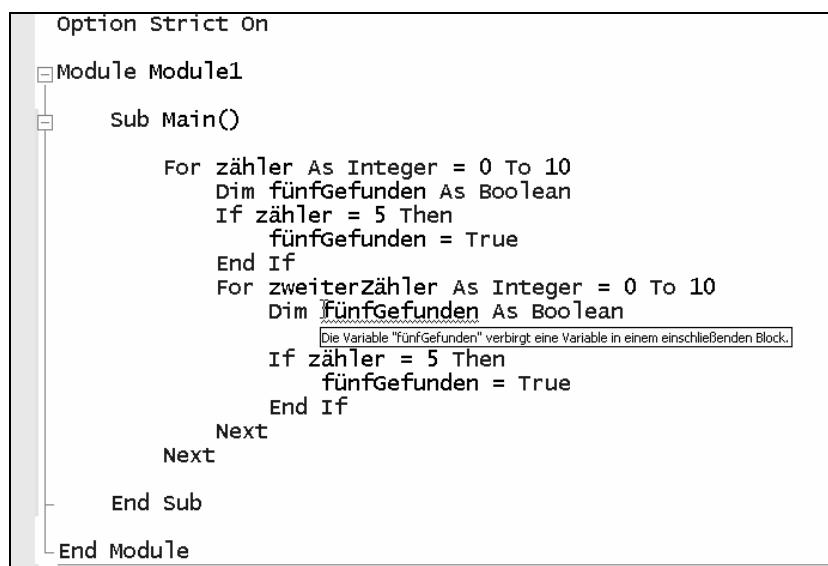
```

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 Microsoft Visual Studio code editor window. The code is as follows:

```

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 over the declaration of `FünfGefunden` in the inner loop, stating: "Die Variable "FünfGefunden" verbirgt eine Variable in einem einschließenden Block." (The variable "FünfGefunden" hides a variable in an enclosing block.)

**Abbildung 5.8:** 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 5.8 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%
```

Das geht in den .NET-Basic-Derivaten nicht mehr. Arrays in VB.NET sind grundsätzlich 0-basierend, und können grundsätzlich nicht mehr mit `To` definiert werden. Anders als in anderen .NET-Programmiersprachen geben Sie aber in Visual Basic .NET nicht die Anzahl der zu dimensionierenden Elemente sondern die höchste Array-Grenze an. Die Anweisung:

```
Dim einArray(10) As Integer
```

definiert also nicht 10 Elemente, sondern die Elemente 0–10 – und das sind 11 Elemente.

Der besseren Lesbarkeit halber erlaubt deswegen Visual Basic 2005 auch 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.

## Die Operatoren `+=` und `-=` und ihre Verwandten

Mit `+=` und `-=` ist Visual Basic um Operatoren für numerische Berechnungen und bei `&=` auch für Stringverkettungen reicher geworden – das folgende Beispiel verdeutlicht ihre Verwendung.

```
If Not IsRangeOkProper(txtValue.Text) Then
    MsgBox("Wertebereich wurde überschritten!" & vbCr & "(Nur im Integerbereich von 0 bis 32768)" & vbCr _
        & "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 – Fließkom-madivision).       |
| 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 String <i>varString</i> die Zeichenkette »Klaus« anhängen.   |

**Tabelle 5.3:** Kurzformen von Operatoren in Visual Basic

**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 Ihnen 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 rechts 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 Unfassbarere 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 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 **Eingabe** 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 5.9:** 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 5.10:** 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 schlicht weg 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 vorneherein:

---

```

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.
    MessageBox.Show("Datei wurde nicht gefunden!" & vbCrLf & _
        vbCrLf & "Der Ausnahmetext lautete:" & vbCrLf & ex.Message, _
        "Ausnahme", MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try

```

**Abbildung 5.11:** 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 Auffangen 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 Lesenroutine 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 schießen 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 in umgewandelt werden, weil sie leer war!")
        ' 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 Erfolgfall 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, in dem 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:

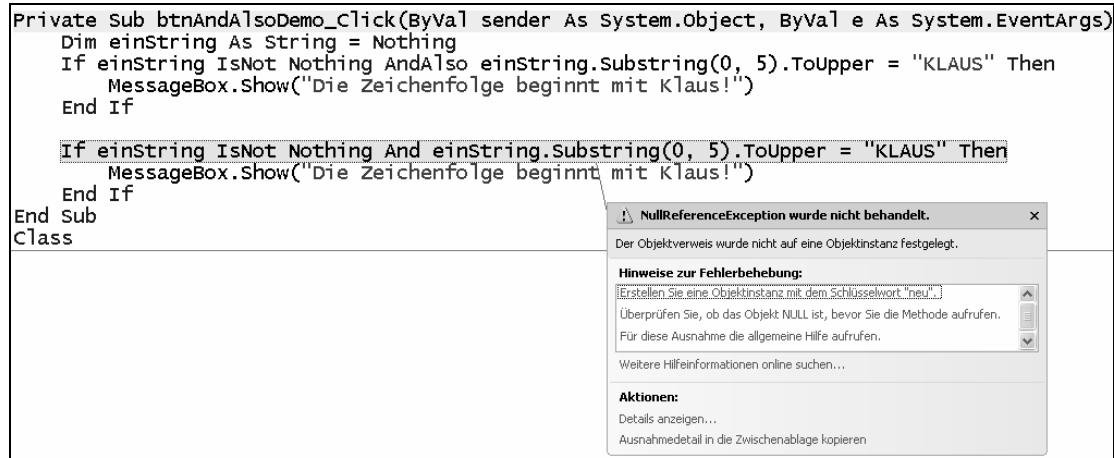
```
Private Sub btnAndAlsoDemo_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _  
    Handles btnAndAlsoDemo.Click  
    Dim einString As String = "Klaus ist das Wort, mit dem diese Zeile beginnt"  
    If einString IsNot Nothing AndAlso einString.Substring(0, 5).ToUpper = "KLAUS" 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 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 5.12:** *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. 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 paradoxalement 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:** Laden Sie zu diesem Zweck das Projekt *AssemblyDemo.sln*, das Sie im Pfad *.|VB 2005 - Entwicklerbuch|C - Ein- und Umstieg|* finden.

---

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.

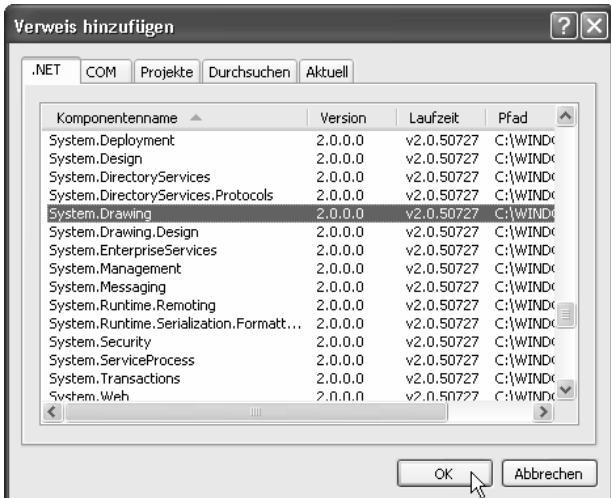


**Abbildung 5.13:** Irgendetwas hindert das Formular daran, dass es korrekt angezeigt werden kann

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 des 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 5.14 sehen. In diesem Dialog suchen Sie anschließend die .NET-Framework-Assembly *System.Drawing.dll*, klicken Sie an und klicken anschließend auf *OK*.



**Abbildung 5.14:** 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 nach einer Grobzählung nicht weniger als rund 8.000 verschiedene Klassen im .NET-Framework. 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,<sup>5</sup> 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 vorangegangen Beispiel (was sich ja nun öffnen lassen sollte), und doppelklicken Sie im Designer auf die Schaltfläche.

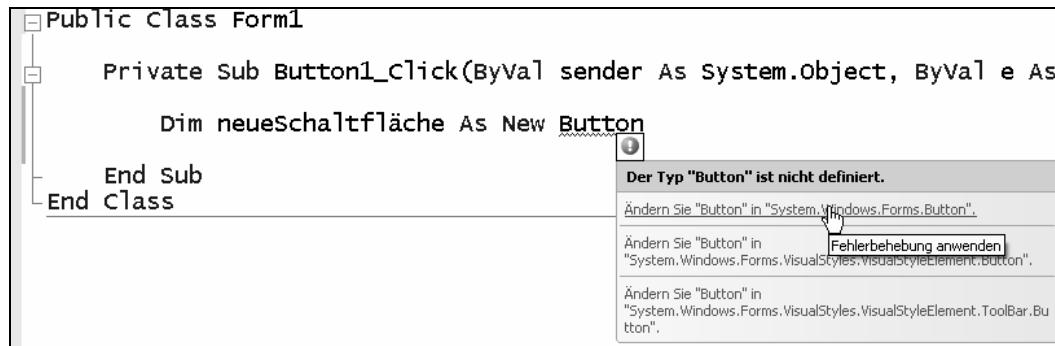
Wir möchten nun Code entwickeln, der zur Laufzeit eine zweite Schaltfläche im Formular platziert.

---

<sup>5</sup> Abbildung 5.14 gibt Ihnen auch einen Eindruck, wie viele verschiedene Assemblies es gibt, denn die Liste ist ja schon eindrucksvoll lang.

Dazu deklarieren wir eine Objektvariable vom Typ `Button` und fügen diesen 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:



**Abbildung 5.15:** Prinzipiell ist diese Zeile nicht falsch, doch findet der Compiler die `Button`-Klasse aus irgendwelchen Gründen nicht

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.Forms.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.
```

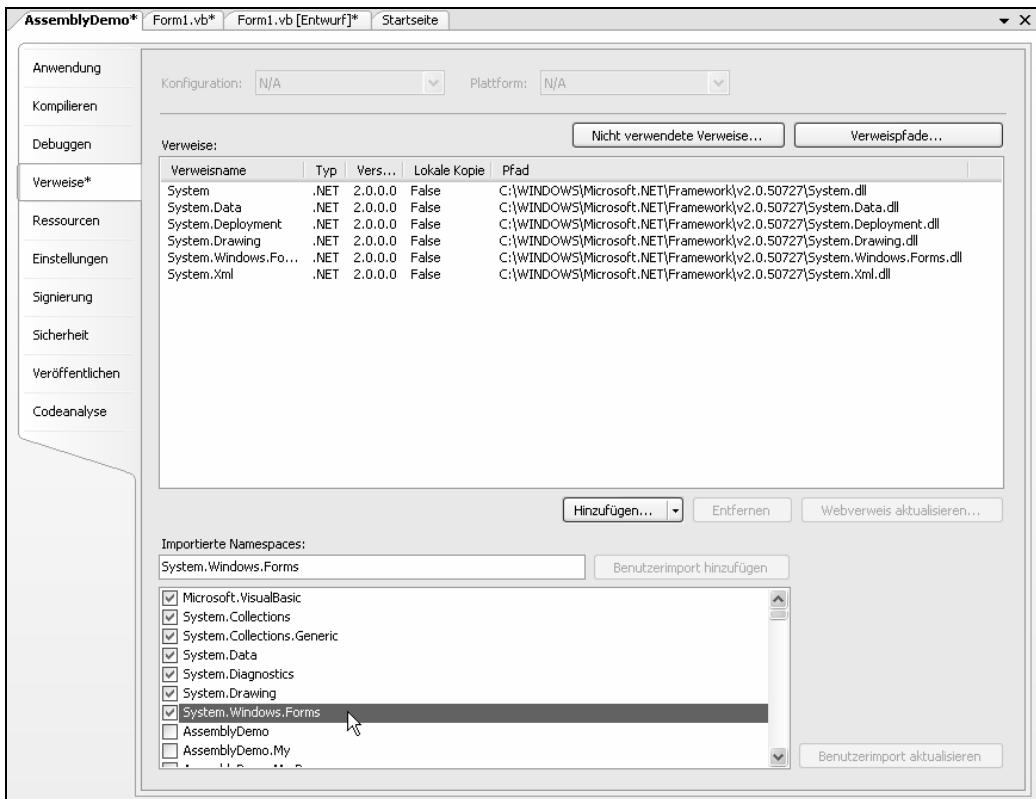
- Oder: Sie setzen eine Imports-Anweisung an den Anfang der Codedatei, die den Namensbereich für diese Codedatei einbindet. Dann können Sie innerhalb der Codedatei auf alle Klassen des importierten Namensbereiches zugreifen, ohne ständig den vollqualifizierten Namen eingeben zu müssen. Beispiel:

```
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 5.16:** 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.Windows.Forms`-Namespace Bescheid weiß.

Übrigens: Abbildung 5.15 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 zu

Hause hat. Deswegen kann Ihnen die Autokorrektur für intelligentes Kompilieren 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 instanzieren
    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:
        .ContentAlignment = 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 Steuerelement 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 machen werden, sind aber möglicherweise nicht alle erforderlichen Assembly-Referenzen oder Namespace-Importe von vornherein vorhanden. Aber jetzt wissen Sie ja, wie Sie sich in solchen Fällen helfen können.

---

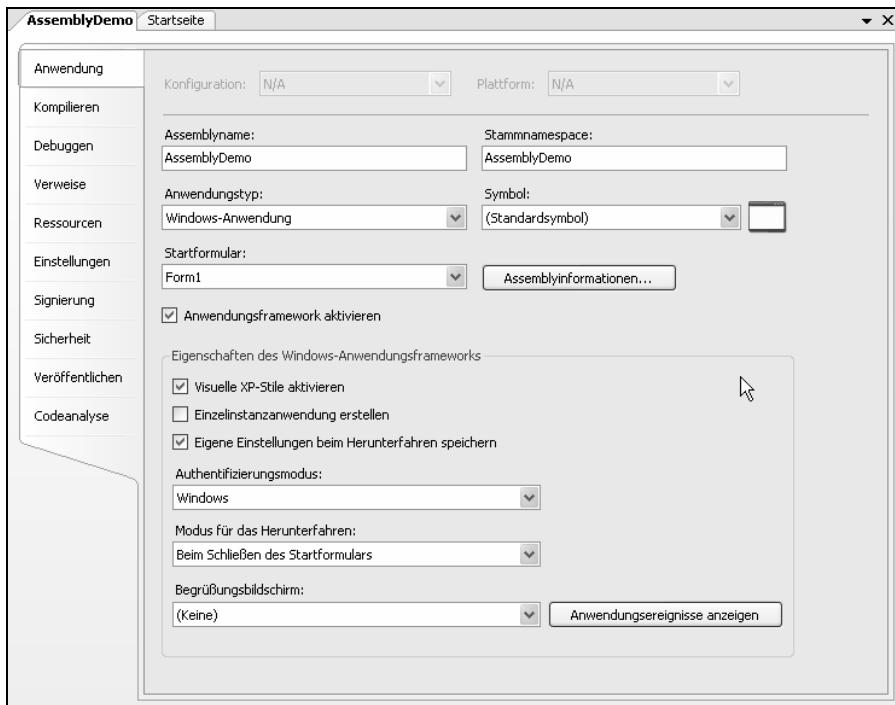
Um es übrigens nochmals zu wiederholen. Dass die *System.Windows.Forms.dll*-Assembly die Button-Klasse in einem Namespace definiert, der genau so heißt, wie die Assembly selbst, ist in diesem Fall zwar so, muss aber nicht so sein. So definiert dieselbe Assembly etwa auch eine *ResXResourceSet*-Klasse (es soll gar nicht interessieren, wozu die da ist), die sich aber im Namespace *System.Ressources* befindet.

## So bestimmen Sie Assemblynamen 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 5.17:** 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

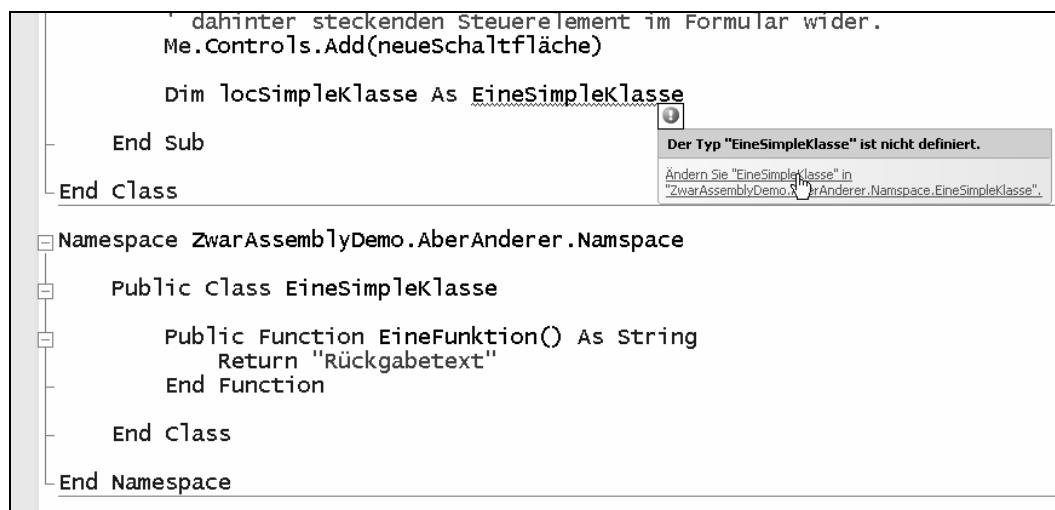
```

```
Namespace ZwarAssemblyDemo.AberAnderer.Namspace
```

```
    Public Class EineSimpleKlasse
        Public Function EineFunktion() As String
            Return "Rückgabetext"
        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 Code-datei des Projektes) zu verwenden.



**Abbildung 5.18:** 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



# 6 Der Umstieg von Visual Basic.NET 2002 und 2003

---

- 
- 185    **Neue Sprachelemente in Visual Basic 2005**
  - 190    **Übersicht über weitere Neuerungen in Visual Basic 2005**
- 

## Neue Sprachelemente in Visual Basic 2005

Zwar hat sich nicht viel, aber immerhin ein wenig an der Syntax der Basic-Sprache in der VB2005-Version geändert. Das betrifft einerseits eine Modifikation in der Behandlung von Schleifen, andererseits die gezielte Steuerung der Freigabe des Objektspeicherbedarfs sowie einige andere Punkte.

### 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
    SkipLoop:
```

**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.

---

## 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 Beispiellisting 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`-Teil zu einer Ausnahme kam oder nicht.

---

**HINWEIS:** Zwei geschachtelte `Try`-Blöcke sind hier übrigens notwendig, 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

```

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 12.

---

**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. 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.

---

## 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 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 begin-

nen. 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

## Über mehrere Codedateien aufgeteilter Klassencode – Partial Class

Sie können die Definition einer Klasse oder Struktur mithilfe des `Partial`-Schlüsselworts auf mehrere Deklarationen aufteilen. Das bedeutet: Sie können beliebig viele Teildeklarationen einer Klasse in beliebig vielen unterschiedlichen Quelldateien verwenden. Alle Deklarationen müssen jedoch in der gleichen Assembly und dem gleichen Namespace enthalten sein.

---

**HINWEIS:** Visual Basic verwendet partielle Klassendefinitionen, um in jeweils eigenen Quelldateien generierten Code von 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 Steuerelementen nicht ändern.

---

Normalerweise wird die Entwicklung einer einzelnen Klasse oder Struktur nicht auf zwei oder mehr Deklarationen aufgeteilt. 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 21 beim Beispiel des Formel Parsers.

## Übersicht über weitere Neuerungen in Visual Basic 2005

Weitere Neuerungen, die in Visual Basic 2005 implementiert wurden, würden den Rahmen an dieser Stelle sprengen und auch zu unnötigen Redundanzen innerhalb dieses Buches führen. Aus diesem Grund finden Sie im Folgenden eine Tabelle, die die wichtigsten Neuerungen zusammenfasst und Ihnen einen Verweis innerhalb dieses Buches nennt, der zu einem entsprechenden Kapitel oder Abschnitt führt, der diese Neuerungen genauer unter die Lupe nimmt.

| Thema                                                          | Kurzbeschreibung                                                                                                                                  | Mehr Info siehe                                                                                                                                      |
|----------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| Eigenschaften von Klassen und Strukturen                       | Eigenschaften können so definiert werden, dass der Lesezugriff mit anderen Zugriffsmodifizierern erfolgt als der Schreibzugriff.                  | ► Kapitel 8 »Unterschiedliche Zugriffsmodifizierer für Eigenschaften-Accesors«.                                                                      |
| Aufbau und Umgang mit Formularen bei Windows Forms-Anwendungen | Anders als bei Visual Basic .NET, wird der Designer-Code und die eigentliche Angabe der Vererbung von Form in einer separaten Codedatei abgelegt. | ► Kapitel 28 hält genauere Informationen zu diesem Thema bereit.<br>► Kapitel 3 zeigt den Umgang mit den Neuheiten im Formular-Designer am Beispiel. |
| Operatorenprozeduren                                           | Klassen bzw. Strukturen können eigene Operatoren implementieren. Dadurch wird der Umgang mit ihnen wesentlich vereinfacht.                        | ► Kapitel 13 informiert Sie über dieses Thema genauer.                                                                                               |

| Thema                                           | Kurzbeschreibung                                                                                                                                                                                                                                                               | Mehr Info siehe                                                                                                                                                                                                             |
|-------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Generische Datentypen<br>(Generics)             | Generische Datentypen dienen zur allgemein gültigen Formulierung typsicherer Codes. Sie vereinen damit die Flexibilität von Object mit Typsicherheit.                                                                                                                          | ► Kapitel 14 hält Genaues zu diesem Thema bereit.                                                                                                                                                                           |
| Primitive Datentypen                            | Mit Visual Basic 2005 gibt es eine Reihe neuer primitiver Datentypen.                                                                                                                                                                                                          | Einige dieser Datentypen wurden bereits in ► Kapitel 5 angesprochen. Genaues über die neuen Typen erfahren Sie in ► Kapitel 16.                                                                                             |
| Primitive Typen, die »nullbar« sind (Nullables) | Dank des Konzeptes der generischen Datentypen gibt es neu in .NET 2.0 die so genannten »Nullables«. Dabei handelt es sich um primitive Datentypen, die über ihre Wertetypen hinaus auch Nothing speichern können, was beispielsweise der Datenbankprogrammierung zugute kommt. | Mehr zu diesem Thema finden Sie in ► Kapitel 20.                                                                                                                                                                            |
| Generische Auflistungsklassen                   | Generische Auflistungsklassen dienen zur einfachen Typisierung von Auflistungen, stellen neue Funktionalitäten für Auflistungen zur Verfügung, sorgen für robusteren und teils auch schnelleren Code.                                                                          | Zu diesem Thema finden Sie in ► Kapitel 20 Genaueres.                                                                                                                                                                       |
| My-Namespace                                    | Mit dem My-Namespace haben es gerade VB6-Entwickler einfacher, sich in der .NET-Welt zurecht zu finden. Der My-Namespace stellt quasi einen Schnellzugriff für Entwickler auf häufig verwendete Funktionen dar, stellt teilweise sogar neue Funktionalitäten bereit.           | In ► Kapitel 25 finden Sie detaillierte Beschreibungen und eine umfangreiche Beispielanwendung, die den Umgang mit My verdeutlicht.                                                                                         |
| Anwendungsframework                             | Das Anwendungsframework dient in VB2005 für das einfache Handling von Windows Forms-Anwendungen aus Entwicklersicht.                                                                                                                                                           | ► Kapitel 26 klärt Details zum Anwendungsframework.                                                                                                                                                                         |
| BackgroundWorker-Komponente                     | In Sachen Threading hat sich nicht nur durch die Entwicklung neuer Mehrkern-Prozessoren einiges getan. Die BackgroundWorker-Komponente bietet den einfachsten und komplikationslosesten Einstieg in das Multithreading.                                                        | ► Kapitel 31 beschreibt den Umgang mit der neuen Komponente. <b>TIPP:</b> Sie sollten sich dennoch das komplette Kapitel zu Gemüte führen, auch wenn es zunächst für die Anwendung dieser Komponente nicht nötig erscheint. |
| ADO.NET 2.0; SQL Server 2005                    | Viele neue Features haben Einzug in ADO.NET gehalten, und nicht zuletzt durch SQL Server 2005 Express werden Visual Basic-Anwendungen, die auf SQL Server basieren, ältere Access-basierende Anwendungen zunehmend ablösen.                                                    | ► Kapitel 32 beschreibt nicht nur die Konfiguration von SQL Server 2005 Express, sondern führt auch in die Programmierung der ADO.NET 2.0-Klassen ein.                                                                      |

**Tabelle 6.1:** Neues in Visual Basic 2005 – hier finden Sie die Verweise auf die Kapitel im Buch



# Teil D

## OOP – Objektorientiertes Programmieren

- 
- 195 Vorüberlegungen zur objektorientierten Programmierung**
  - 215 Auf zum Klassentreffen!**
  - 261 Klassenvererbung und Polymorphie**
  - 327 Über Structure und den Unterschied zwischen Referenz- und Wertetypen**
- 

Objekte sind das A und O in .NET – falls Sie schon Erfahrungen in .NET gesammelt haben, wissen Sie das längst. Gehören Sie zu denjenigen, die .NET erst im letzten Teil näher kennen gelernt haben, haben Sie auf alle Fälle schon eine Ahnung davon bekommen.

Objekte entstehen, wenn man sie aus Strukturen oder Klassen instanziert. Doch was genau sind Klassen eigentlich? Dieser Teil soll Ihnen zeigen, wie das Konzept der objektorientierten Programmierung funktioniert, was man unter Strukturen und Klassen zu verstehen hat, welche Unterschiede es gibt, und wie man ihren Einsatz mit der Verwendung so genannter Schnittstellen perfektionieren kann.

Darüber hinaus wird auf ein neues Programmierkonzept von Visual Studio 2005 eingegangen – den so genannten *Generics*. Mit ihrer Hilfe lassen sich viele Aufgaben im Bereich der OOP lösen, die in der Vorgängerversion noch ziemlichen Aufwand verursacht hätten oder schlicht gar nicht möglich gewesen wären.

Schließlich gibt es in Visual Basic 2005 auch die Möglichkeit, wie schon von Anfang an in C#, eigene Typen, die aus Strukturen oder Klassen hervorgehen, durch so genannte Operatoren-Prozeduren viel flexibler zu machen. Auch diesem Thema ist ein eigenes Kapitel in diesem Buchteil gewidmet.



# 7 Vorüberlegungen zur objektorientierten Programmierung

---

- 196 Über Assemblies, Namespaces, CLR, CLI, BCL, JITter und andere .NET-Terminologien
  - 201 Erzwungene Typsicherheit und Deklarationszwang von Variablen
  - 204 Namensgebung von Variablen
  - 206 Und welche Sprache ist die beste?
  - 206 Prozedurale Programmierung versus OOP
- 

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. 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 auch unter .NET weiterhin in ihrem alten Stil weiterprogrammieren. Doch ganz ehrlich: Das ist wie Porsche fahren mit angezogener Handbremse. Visual Basic .NET ist gerade durch die OOP-Fähigkeiten endlich erwachsen geworden, und dieses Kapitel zeigt Ihnen – nachdem es einen kleinen Abstecher durch den Terminologien-Parcours von .NET gemacht hat – anhand eines Fallbeispiels, wo die grundsätzlichen Probleme bei der prozeduralen Programmierung liegen.

Auch wenn Sie meinen, Klassen grundsätzlich zu beherrschen und in Ihren eigenen Projekten schon längst verwenden: Riskieren Sie es doch dennoch, sich die folgenden Kapitel zu Gemüte zu führen. Zum Thema Klassen und OOP gehört weit mehr, als nur das Wissen um ihre bloße Instanzierung und die Anwendung von Polymorphie, denn: Wie sieht es mit Schnittstellen aus? Kennen Sie die Unterschiede zwischen Verweis- und Wertetypen? Können Sie erklären, wieso ein Objekt ein Referenztyp ist, alle primitiven Typen zwar 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 neuen Objekttypen in Visual Studio 2005, mit denen Sie – richtige Anwendung vorausgesetzt – viele Dinge des täglichen Entwicklerlebens enorm vereinfachen und extrem 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). Denken Sie daran: 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 möchte ich Ihnen empfehlen, dass Sie sich ein wenig Zeit nehmen und die folgenden Abschnitte am besten hintereinander durcharbeiten. Das Verstehen des ganzen Zusammenhangs von Klassen, Schnittstellen und allem was sonst noch dazu gehört wird Ihnen dann sicherlich viel leichter fallen – und Sie werden sich im Handumdrehen zum »Klassenprimus« mausern!

---

## Über Assemblies, Namespaces, CLR, CLI, BCL, JITter und andere .NET-Terminologien

Dieses Kapitel ist das erste Kapitel, das sich im Buch intensiv mit Techniken der Programmierung unter .NET beschäftigt, und daher möchte ich – bevor es in medias res geht – an dieser Stelle zunächst für Klarheit bei den .NET-Terminologien sorgen, aber auch einige Zeilen über meine persönliche Vorgehensweise beim Benennen von Variablen, Prozeduren und Objekten verlieren.

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 meine Erfahrung hat gezeigt, dass es auch eine ganze Menge Entwickler gibt, die bestenfalls über Halbwissen verfügen, von denen man entweder nur Halbwahrheiten oder sogar völlig falsche Informationen bekommt.

Die folgenden Abschnitte sollen deshalb die wichtigsten .NET-Fachbegriffe 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 DLLs<sup>1</sup> zusammenzufassen und von außen aufzurufen – einfach um Redundanz durch doppelte Funktionen zu vermeiden und letztendlich Platz zu sparen.

---

<sup>1</sup> *Dynamic Link Libraries*, etwa: *dynamisch verbindbare Bibliotheken*.

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.

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 Buches nur »lose im Raum« stehen, litt 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, von einander völlig unabhängig.

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 per Definition so, dass Namespace und Assembly gleiche Namen tragen. Das muss aber, wie schon gesagt, nicht so sein: Andere Objekte des gleichen Namespaces 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 geschachtelter 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 6.

---

## 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 sie hat weitere, genau so 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)<sup>2</sup>, kurz vor der Ausführung in nativen Assembler-Code kompiliert werden (*JIT* steht für *Just in Time*, etwa: *genau rechtzeitig*).

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 noch referenziert wird und entsorgt es, wenn feststeht, dass 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 Entwicklungsgeschäfts ständig 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.

---

<sup>2</sup> Einige Microsoftler verwenden die Abkürzung *MSIL*, einige *MIL* und einige nur *IL*.

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.<sup>3</sup>

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 ...).

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 der einen .NET-Sprache entwickelt worden sind, sich in einer anderen .NET-Sprache verwenden lassen. 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 MS-IML (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 *Intermediate Language* oder

---

<sup>3</sup> 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.

kurz: *IL*). »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#.

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.<sup>4</sup> Der JITter ist so optimiert, dass er nur jeweils die Methoden einer Klasse kompiliert, die als nächstes benötigt werden. Und das bisschen 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 ca. 22 MByte großen Framework selbst auf einem flotten Rechner zwar immer noch schnell vonstatten 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 *kompiliert* 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.

---

<sup>4</sup> O.k., na gut, sagen wir: In den seltensten Fällen.

# Aufbau des .NET-Framework

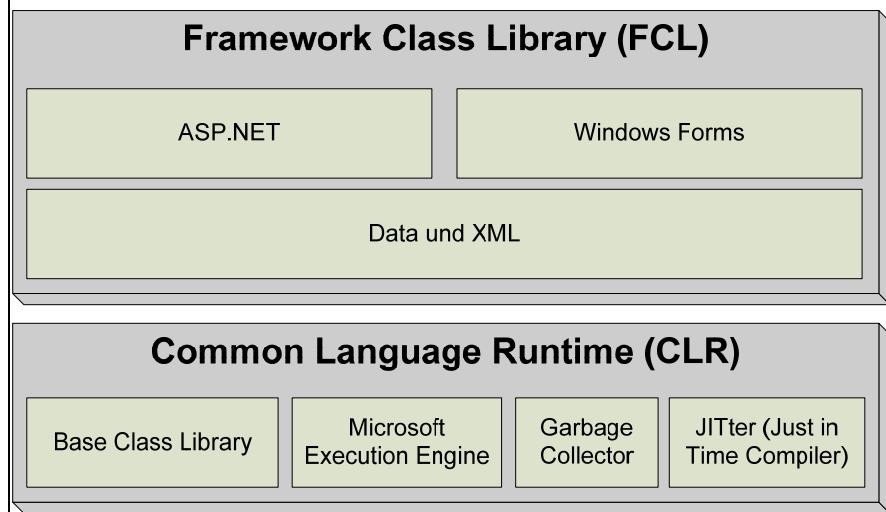


Abbildung 7.1: Diese Grafik stellt den Aufbau des .NET Framework bildlich dar

## 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 .NET-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 23 informieren.

Ein Beispiel soll dieses Problem verdeutlichen.

---

**BEGLEITDATEIEN:** Sie finden die Codedateien zu diesem Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\Design - OOP\Kap07\OptionSamples`. Öffnen Sie dort die entsprechende Projektmappe-Datei (`.sln`).

---

```

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

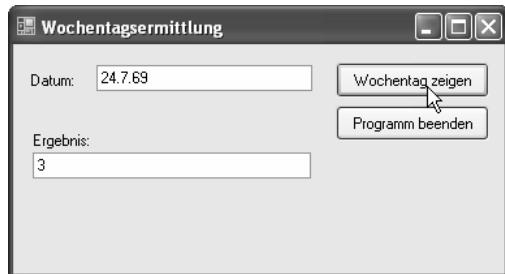
        'Wochentagsnummer ausgeben
        txtWeekday.Text = locWeekday

    End Sub

```

End Class

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

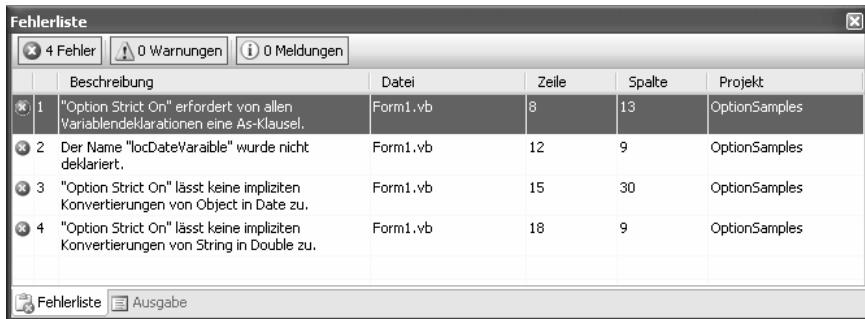


**Abbildung 7.2:** Dieses Programm »scheint« nur zu funktionieren – in Wahrheit verborgen sich in den paar Zeilen Code zwei dicke Fehler!

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 `locDateVariaible` 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 produzieren kann. Wenn Sie sich durch `Option Strict On` selbst dazu zwingen, dass Sie

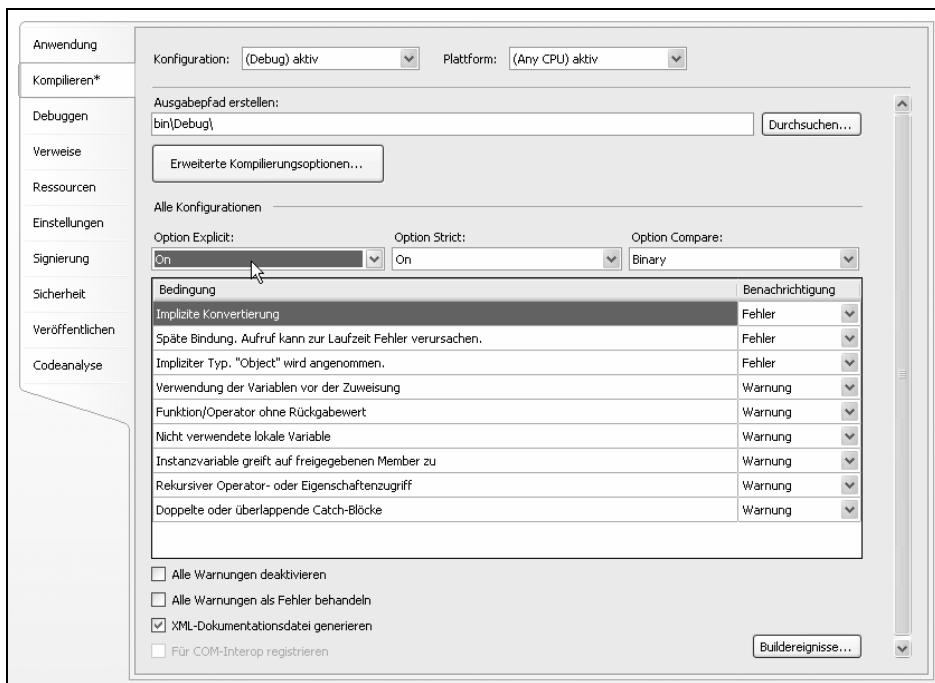
beispielsweise keiner String-Variablen einen Wert vom Typ Integer zuweisen können, vermeiden Sie solche Fehler im Vorfeld.



**Abbildung 7.3:** 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 7.3:

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 7.4:** 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 7.5).

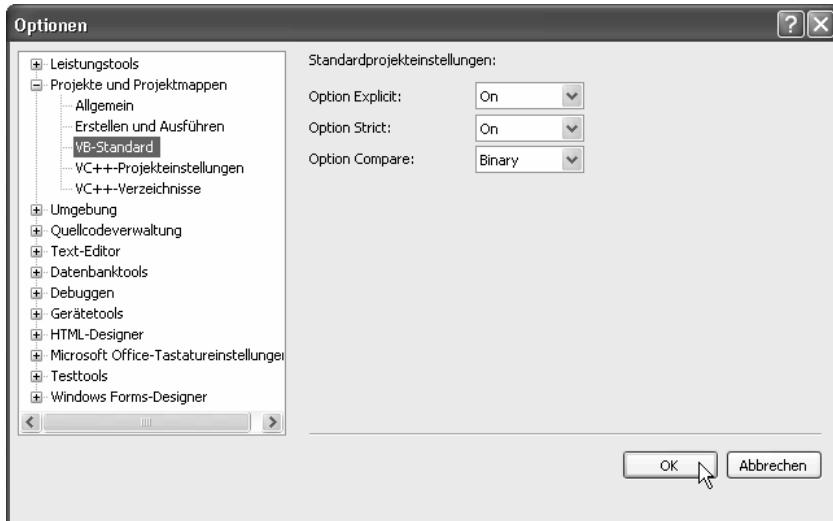


Abbildung 7.5: Mit diesem Dialog bestimmen Sie das *Option*-Verhalten für zukünftige Projekte

## 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 7.6: 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 Buchstaben 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.<sup>5</sup>

```
// 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;
    }
}
```

---

<sup>5</sup> Die komplette CLR-Implementierung der CLI erhalten Sie über den IntelliLink D0701. Nur die Version 1.0 (ohne beispielsweise die Implementierung von Generics) war zum Zeitpunkt der Drucklegung verfügbar.

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). Die genauen Konventionen dafür finden Sie in ► Kapitel 3 (»Namensgebungskonventionen für Steuerelemente in diesem Buch«).

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 (echte, 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 Benutzesteuerelemente 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.

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.

## Prozedurale Programmierung versus OOP

»Diesen ganzen OOP-Quatsch brauche ich nicht. – Ich programmiere unter VB.NET genau wie unter VB6, das ist am einfachsten und geht am schnellsten.« Sie glauben gar nicht, wie oft mir diese Aussage in den vergangenen vier Jahren schon begegnet ist. Und sie zu widerlegen, ist mein größtes Anliegen; denn wenn Sie das OOP-Konzept in Visual Basic .NET verstanden und verinnerlicht haben, glauben Sie mir, erst dann können Sie alle Möglichkeiten des Frameworks richtig nutzen.

Ein Fallbeispiel soll deswegen die grundsätzliche Problematik verdeutlichen. Dabei handelt es sich um ein Programm, dass zwar in Visual Basic .NET verfasst, aber im typischen VB6-Stil gehalten ist. Der Quell-Code würde sich mit nur marginalen Änderungen unter VB6 in 10 Minuten zum Laufen bringen lassen. Und darum geht es:

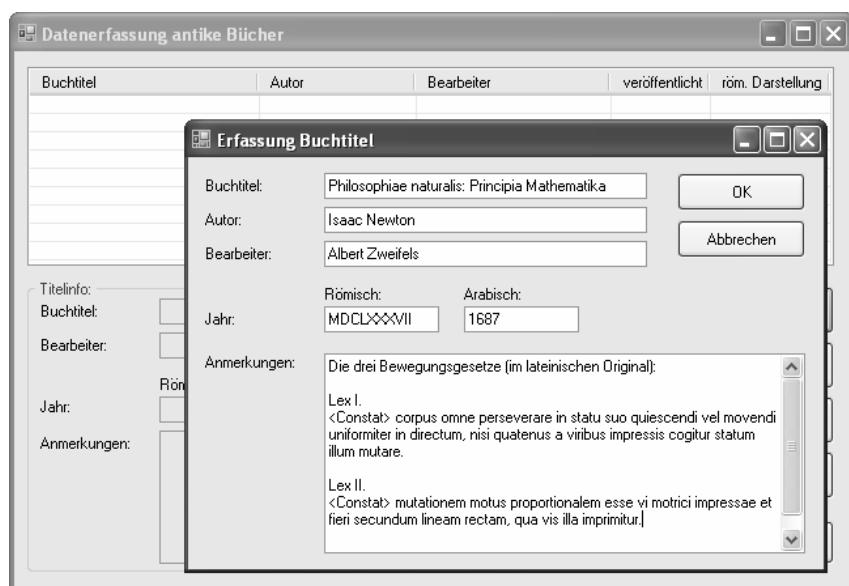
## Prozedurale Programmierung ade?

Stellen Sie sich vor, Sie arbeiten in einem Softwarehaus als Entwickler, und Sie werden mit der Aufgabe betraut, einen Programmteil zu schaffen, mit dem Daten von antiquarischen Büchern erfasst werden können. In der Eingabemaske sollen

- der Autor des Buches
- der Titel des Buches
- der Bearbeiter (also der, der die Buchdaten erfasst)
- eine Bemerkung
- und das Erscheinungsjahr des Buches erfasst werden.

Bei letzterem Punkt gibt es eine Besonderheit: Viele Bücher haben die Angabe Ihrer Erstveröffentlichung nur in römischen Zahlen aufgedruckt – um Umrechnungsfehler zu vermeiden, soll Ihr Programm part in der Lage sein, sowohl arabische als auch römische Zahlen zu verarbeiten. Sie programmieren seit Ihrem Umstieg auf VB.NET immer noch im Visual Basic-6.0-Stil, haben von Klassenprogrammierung eigentlich noch nie etwas gehört, und schon bei der Planung des Programms stoßen Sie auf die ersten Schwierigkeiten, denn:

Ihr Ziel ist es, anderen Programmierern eine möglichst flexible Benutzung Ihres Formulars zu ermöglichen. Es soll es mit nur einer Zeile Code aufzurufen sein, dem Anwender die Datenerfassung erlauben, und dem Programmteil, das Ihr Formular aufruft, die Daten zurückliefern.



**Abbildung 7.7:** Das Erfassungsprogramm für antiquarische Buchtitel in Aktion

Doch die Anforderung schieben Sie zunächst beiseite. Sie kümmern sich als erstes um die interne Funktionalität und schaffen eine Funktion, die Sie in das Modul `mdlRomanNumeralsLib.vb` packen, die eine normale Zahl – einen Integerwert beispielsweise – in einen String mit dem römischen Nu-

merale verwandelt. Der entsprechende Code dafür ist kompakt und einfach zu verstehen, etwa wie im folgenden Listing zu sehen.

---

**BEGLEITDATEIEN:** Sie finden die Codedateien zu diesem Beispiel im Verzeichnis *.|VB 2005 - Entwicklerbuch|D - OOP\Kap07\AncientBooks*. Öffnen Sie dort die entsprechende Projektmappe-Datei (*.sln*).

---

Module mdlRomanNumeralsLib

```
Public Function RomanNumeral(ByVal ArabicNumber As Integer) As String

    Dim locCount As Integer = 0
    Dim locDigitValue As Integer = 0
    Dim locRoman As String = ""
    Dim locDigits As String = ""

    'Diese römischen Basis-Numeralia gibt es:
    locDigits = "IVXLCDM"

    'Der maximal darstellbare Bereich - eine Null gibt es nicht.
    If ArabicNumber < 1 Or ArabicNumber > 3999 Then
        RomanNumeral = "#N/A#"
        Exit Function
    End If

    locCount = 1
    Do While ArabicNumber > 0
        locDigitValue = ArabicNumber Mod 10
        Select Case locDigitValue

            'Ziffern 1 bis 3 werden einfach hintereinander geschrieben (I, II, III).
            Case 1 To 3
                locRoman = String$(locDigitValue, Mid$(locDigits, locCount, 1)) & locRoman

            'Die 4. Ziffer ist der "Einer-Wert" vor dem nächsten "fünfer-Wert" (IV).
            Case 4
                locRoman = Mid$(locDigits, locCount, 2) & locRoman

            'Die 5. Ziffer hat ein eigenes Numerale (V).
            Case 5
                locRoman = Mid$(locDigits, locCount + 1, 1) & locRoman

            'Kombination aus "Fünfer-Werten" und "Einer-Werten" (VI, VII, VIII):
            Case 6 To 8
                locRoman = Mid$(locDigits, locCount + 1, 1) &
                           String$(locDigitValue - 5, Mid$(locDigits, locCount, 1)) & locRoman

            'Kombination aus "Einer-Wert" und "Zehner-Wert" (IX):
            Case 9
                locRoman = Mid$(locDigits, locCount, 1) & Mid$(locDigits, locCount + 2, 1) & locRoman

        End Select
        locCount = locCount + 2
    Loop
    RomanNumeral = locRoman
```

```

ArabicNumber = ArabicNumber \ 10
Loop
RomanNumeral = locRoman
End Function

So weit, so einfach. Anschließend entwickeln Sie das Gegenstück zu dieser Funktion, die nämlich einen String mit einem römischen Numerale entgegennimmt und zurück in einen wirklichen (im Sinne des Computers) Wert wandelt. Auch diese Routine ist für Sie vergleichsweise einfach in die Tat umzusetzen:

Public Function ValueFromRomanNumeral(ByVal RomanNumeral As String) As Integer

On Error GoTo vfrn_error

RomanNumeral = UCASE(RomanNumeral)

Static Table(0 To 6, 0 To 1) As String

Dim locCount As Integer
Dim locChar As Char
Dim retValue As Integer
Dim z1 As Integer, z2 As Integer

If RomanNumeral = "" Then
    ValueFromRomanNumeral = 0
    Exit Function
End If

'Tabelle zum Nachschlagen
Table(0, 0) = "I" : Table(0, 1) = "1"
Table(1, 0) = "V" : Table(1, 1) = "5"
Table(2, 0) = "X" : Table(2, 1) = "10"
Table(3, 0) = "L" : Table(3, 1) = "50"
Table(4, 0) = "C" : Table(4, 1) = "100"
Table(5, 0) = "D" : Table(5, 1) = "500"
Table(6, 0) = "M" : Table(6, 1) = "1000"

locCount = 1

Do While locCount <= Len(RomanNumeral)
    locChar = Convert.ToChar(Mid(RomanNumeral, locCount, 1))

    If locCount < Len(RomanNumeral) Then
        For z1 = 0 To 6
            If Table(z1, 0) = locChar Then Exit For
        Next z1
        For z2 = 0 To 6
            If Table(z2, 0) = Mid(RomanNumeral, locCount + 1, 1) Then
                Exit For
            End If
        Next z2
        If Val(Table(z1, 1)) < Val(Table(z2, 1)) Then
            'Stringfragment entfernen

```

```

        RomanNumeral = Microsoft.VisualBasic.Left(RomanNumeral, locCount - 1) + _
                      Mid(RomanNumeral, locCount + 2)
        retValue = retValue + (CInt(Table(z2, 1)) - CInt(Table(z1, 1)))
    Else
        For z2 = 0 To 6
            If Table(z2, 0) = locChar Then Exit For
        Next z2
        retValue = retValue + CInt(Table(z2, 1))
        locCount = locCount + 1
    End If
Else
    For z2 = 0 To 6
        If Table(z2, 0) = locChar Then Exit For
    Next z2
    retValue = retValue + CInt(Table(z2, 1))
    locCount = locCount + 1
End If
Loop

ValueFromRomanNumeral = retValue
Exit Function

vfrn_error:
    ValueFromRomanNumeral = 0
    Exit Function
End Function

```

Doch jetzt kommt das Entscheidende: Wie behandeln Sie die Datenermittlung im Formular und wie liefern Sie alle Daten zurück an das Programm, das Ihr Formular aufgerufen hat? Sie schaffen zunächst das Formular samt Steuerung in der Formular-Klassendatei *frmRomanNumerals.vb*, das sich folgendermaßen gestaltet:

```

'Typische Vorgehensweise bei der Prozeduralen Programmierung
'á la Visual Basic 6.0. Bis auf den Unterschied "Class" und "Form"
'(und einige ganz unwesentliche Kleinigkeiten) wäre dieses Programm
'aus unter Visual Basic 6.0 lauffähig und dafür typisch.
Public Class frmRomanNumerals
    'Diese Flags dienen dazu, zu verhindern, dass eine Änderung
    'einer TextBox die Änderung der anderen nach sich zieht,
    'die wiederum die Änderung der ersten verursacht.
    'Durch vorheriges Setzen der entsprechenden Flags wird diese
    'Ereigniskette verhindert.
    Dim myDontPerformRoman As Boolean = False
    Dim myDontPerformArabic As Boolean = False

    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
        Me.DialogResult = Windows.Forms.DialogResult.OK
        'Die folgende Anweisung müsste übrigens nicht sein,
        'da ein Ändern von DialogResult in OK innerhalb eines Button Click-Events
        'automatisch zu Form.Hide führt.
        Me.Hide()
    End Sub

```

```

Private Sub btnCancel_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnCancel.Click
    Me.DialogResult = Windows.Forms.DialogResult.Cancel
    Me.Hide()
End Sub

Private Sub txtRomanYear_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles txtRomanYear.TextChanged

    'Umwandeln, wenn sich der Text ändert
    If Not myDontPerformRoman Then
        myDontPerformArabic = True
        txtArabicYear.Text = CStr(ValueFromRomanNumeral(txtRomanYear.Text))
        txtArabicYear.SelectAll()
        myDontPerformArabic = False
    End If

End Sub

Private Sub txtArabicYear_TextChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles txtArabicYear.TextChanged

    'Umwandeln, wenn sich der Text ändert
    If Not myDontPerformArabic Then
        myDontPerformRoman = True
        If txtArabicYear.Text <> "" Then
            txtRomanYear.Text = CStr(RomanNumeral(CInt(txtArabicYear.Text)))
            txtRomanYear.SelectAll()
        Else
            txtRomanYear.Text = ""
        End If
        myDontPerformRoman = False
    End If

End Sub

```

Bis hier hin ist Ihre Formularsteuerung ganz passabel und Sie sind mit dem Ergebnis ganz zufrieden. Bis jetzt. Denn ohne die objektorientierte Programmierung bleiben Ihnen nicht viele Möglichkeiten, das nächste Problem zu lösen: den Datenfluss zwischen Ihrem Modul und dem aufrufenden Programm. Sie entscheiden sich für das Einfachste und lösen dieses Problem, indem Sie die Dialogeingänge in Variablen zurückgeben, die Ihnen durch das aufrufende Programm per Referenz übergeben werden. Dass Ihnen diese Lösung in Zukunft noch richtig Stress bereiten wird, ist Ihnen zu diesem Zeitpunkt noch gar nicht bewusst. Ihre Funktion sieht so aus:

```

'Wichtig: Variablen sind als ByRef deklariert, damit sie Ergebnisse zurückliefern können.
Public Function EditOrNewBookData(ByRef Titel As String, ByRef Author As String, _
    ByRef Editor As String, ByRef YearPublished As String, ByRef Notes As String) As DialogResult

    'Formular darstellen
    'bleibt bis zum nächsten Hide stehen,
    'da modaler Dialog

```

```

txtEditor.Text = Editor
txtAuthor.Text = Author
txtBookTitel.Text = Titel
txtArabicYear.Text = YearPublished
txtNotes.Text = Notes
Me.ShowDialog()

'Überprüfen ob Abbrechen gedrückt wurde,
If Me.DialogResult = Windows.Forms.DialogResult.Cancel Then
    'Variablen zurückliefern
    Editor = ""
    Titel = ""
    YearPublished = ""
Else
    'sonst die Inhalte des Dialoges zurückliefern
    Editor = txtEditor.Text
    Author = txtAuthor.Text
    Titel = txtBookTitel.Text
    YearPublished = txtArabicYear.Text
    Notes = txtNotes.Text
End If

'Anzeigen, dass Dialog Daten "hatte"
EditOrNewBookData = Me.DialogResult

End Function
End Class

```

Wenn Sie, wie hier im Listing zu sehen, die Variable ändern, die die Prozedur entgegennimmt, ändert sich die Variable auch im Programmteil, der Ihre Funktion aufruft (vorausgesetzt, Ihnen sind die Parameter als Variablen übergeben worden, und diese sind jeweils auch nicht eingeklammert gewesen).

Ihren Kollegen teilen Sie mit, wie dieses Formular zu verwenden ist, und diese implementieren den Aufruf nach den von Ihnen vorgegebenen Konventionen unter anderem im Formular *frmMain.vb* auf folgende Weise (hier beispielhaft für die Methode, die ausgelöst wird, wenn der Anwender die Schaltfläche *Titel hinzufügen* im Hauptformular anklickt – der Aufruf Ihrer Funktion ist fett hervorgehoben):

```

'Wird ausgelöst, wenn der Anwender auf die Schaltfläche
'[Titel hinzufügen] klickt.
Private Sub btnAddTitel_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnAddTitle.Click

    Dim locTitel As String
    Dim locAuthor As String
    Dim locEditor As String
    Dim locYearPublished As String
    Dim locNotes As String

    locTitel = ""
    locAuthor = ""
    locEditor = ""

```

```

locYearPublished = ""
locNotes = ""

If frmRomanNumerals.EditOrNewBookData(locTitel, locAuthor, locEditor, locYearPublished, locNotes) =
    = Windows.Forms.DialogResult.OK Then
    With lvwBookItems
        Dim locListViewItem As ListViewItem
        locListViewItem = .Items.Add(locTitel)
        locListViewItem.SubItems.Add(locAuthor)
        locListViewItem.SubItems.Add(locEditor)
        locListViewItem.SubItems.Add(locYearPublished)
        locListViewItem.SubItems.Add(RomanNumeral(CInt(locYearPublished)))
        'Die Anmerkungen werden nicht in der Liste dargestellt, müssen aber
        'irgendwo gespeichert werden. Deswegen wandern sie in die Tag-Eigenschaft
        'eines Eintrags der ListView, die jedes beliebige Objekt speichern kann.
        locListViewItem.Tag = locNotes
    End With
End If

```

End Sub

Eine weitere Woche später kommt es aber knüppeldick. Ihr Projektleiter verrät Ihnen, dass sich die Planung geändert hat. Sie sollen die Funktionalität des Programms so überarbeiten, dass Ihr Programm das Erfassungsdatum ebenfalls zurückliefert. Sie machen sich sofort an die Arbeit, ändern das Formular und ihre Funktion gemäß der neuen Anforderungen ab,

```

'Wichtig: Variablen sind als ByRef deklariert, damit sie Ergebnisse zurückliefern können.
Public Function EditOrNewBookData(ByRef Titel As String, ByRef Author As String, ByRef Collected as Date, _
    ByRef Editor As String, ByRef YearPublished As String, ByRef Notes As String) As DialogResult

```

und bekommen am selben Nachmittag von Ihren Kollegen richtig Ärger, weil keiner ihrer Programmteile mehr funktioniert. Klar, denn Sie haben tief in die Konventionen des Projektes eingegriffen und Standards, die Sie selbst gesetzt haben, verändert. Ihre Routine erwartet nunmehr sechs Parameter, doch allen Programmierern haben Sie die Routine mit fünf Parametern erklärt.

Dieses Szenario ist vielleicht ein wenig gestelzt, aber von Firmen, die ich regelmäßig berate, weiß ich, dass es immer noch eine Vielzahl von Programmier- und Entwicklerteams gibt, die auf diese Weise arbeiten. Da VB6 weder echte Vererbung noch Methodenüberladung und Polymorphie auch nur durch Hintertürchen in Form von Schnittstellen erlaubte, bin ich der letzte, der hier den ersten Stein schmeißen will, und es Teams ankreidet, bisher auf diese Weise entwickelt zu haben.

Doch die Zeiten haben sich geändert. Wenn Sie in .NET erfolgreich programmieren wollen, dann kommen Sie an Objekten nicht mehr vorbei. Jedes Programm, das Sie in Zukunft schreiben, verfügt über mindestens eine Klasse. In Visual Basic .NET sieht man das vielleicht nicht sofort, aber es ist definitiv so. Wieso das so ist, erfahren Sie in den folgenden Kapiteln. Schreiten wir also zur nächsten Ebene.



# 8 Auf zum Klassentreffen!

---

- 216 **Und ab in den Sandkasten**
  - 216 **Konsolenanwendung in VB.NET**
  - 218 **Das Klassenprinzip am einfachsten Beispiel**
  - 220 **Das Klassenprinzip am eigenen Beispiel**
  - 221 **Statische und nicht-statische Methoden und Variablen**
  - 223 **Nicht verwirren lassen: Static und Shared in VB**
  - 225 **Smarttags im Editor von Visual Basic**
  - 227 **Kleiner Exkurs – womit startet ein Programm?**
  - 227 **Mit Sub New bestimmen, was beim Instanzieren passiert – der Klassenkonstruktor**
  - 230 **Überflüssige Funktionen mit dem Obsolete-Attribut markieren**
  - 231 **Überladen von Funktionen und Konstruktoren**
  - 238 **Zusätzliche Werkzeuge für .NET**
  - 241 **Statische Konstruktoren und Variablen**
  - 246 **Eigenschaften**
  - 256 **Zugriffsmodifizierer von Klassen, Prozeduren, Eigenschaften und Variablen**
- 

Stellen Sie sich vor, Sie könnten dem Dilemma des Eingangsbeispiels aus dem letzten Kapitel entgehen, indem Sie eine Einheit verwenden könnten, die in sich völlig geschlossen ist, aber dennoch völlig erweiterbar bleibt. Und zu verstehen, wie und wieso das mit Klassen funktionieren kann, lassen Sie mich versuchen, das Konzept von Klassen mithilfe einer Analogie zu beschreiben.

# Und ab in den Sandkasten

Versetzen Sie sich in Ihre Kindheit zurück. Und zwar so weit zurück, dass Sie im Sandkasten sitzen und mit Förmchen und im Matsch 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 es selbst ist noch kein Objekt, sondern nur eine Vorlage. Wenn Sie aus einer Klasse (einem Förmchen) einen Sandkuchen (ein Objekt) machen wollen, dann müssen Sie die Klasse in ein Objekt instanzieren. Um bei der Analogie zu bleiben: Sie instanzieren einen Sandkuchen aus einem Förmchen, indem Sie nassen Sand in die Form hineingeben, das ganze Ding umdrehen und die Form abziehen.

Klassen in der objektorientierten Programmierung beinhalten Daten *und* Programmcode. Was bei dieser Analogie sind aber die Daten und was der Programmcode? Die Daten sind das, was Sie in das Förmchen hinein geben. Das kann nasser Sand, nasse Muttererde, Lehm, Ton oder Ähnliches sein. Je nachdem, wie Sie die Daten (den Inhalt, das Substrat) verändern, variieren ihre Ergebnisse in Form der instanzierten Objekte. Ihr instanziertes Objekt wird heller oder dunkler, wird fest, wenn es von der Sonne getrocknet wird, oder es zerbröselt nach ein paar Minuten. Doch alle Objekte, die Sie aus Ihrem Förmchen generieren, halten sich an bestimmte Rahmenbedingungen. Diese Rahmenbedingungen könnten Sie mit dem Programmcode einer Klasse vergleichen. Die Silhouette des Förmchens gibt buchstäblich die Rahmenbedingungen vor. Wenn das Förmchen in Sternform »programmiert« wurde, ergeben sich aus ihm auch sternförmige Sandkuchen, ist es in Form eines Schmetterlings »programmiert«, bekommen Sie eben Sandkuchenobjekte in Zitronenfalterform heraus.

Der Programmcode innerhalb einer Klasse regelt, welche Daten die Objekte, die aus ihr hervorgehen, später speichern können und reglementiert auch den Zugriff auf die Daten.

Mit diesem Wissen können Sie Programmkonzepte von Grund auf ändern und mit .NET Klassen schaffen, die auf der einen Seite die Daten für Problemlösungen speichern und auf der anderen Seite Code zur Verfügung stellt, um die Daten zu verwalten und den Zugriff auf diese zu reglementieren.

---

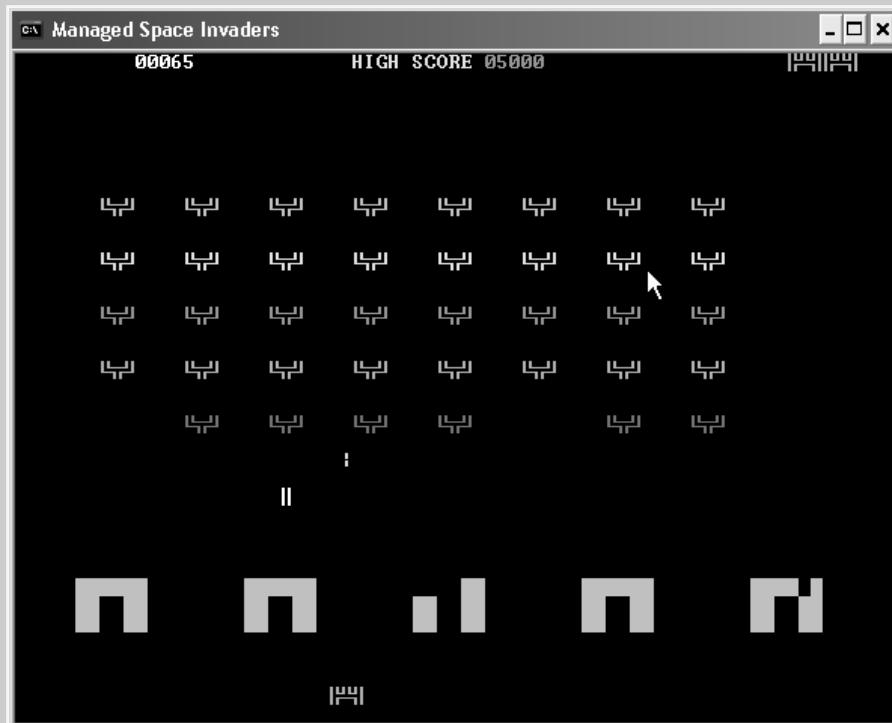
**HINWEIS:** Bei den Beispielprogrammen der folgenden Abschnitte handelt es sich nicht um Windows-, sondern um so genannte Konsolenanwendungen – um Anwendungen also, die sich ausschließlich unter der Windows-Eingabeaufforderung verwenden lassen.

---

## Konsolenanwendung in VB.NET

Im Gegensatz zu Visual Basic 6.0 können Sie in Visual Basic .NET auch ohne größeren Aufwand Konsolenanwendungen entwerfen. Das sind Programme, die über keine grafische Oberfläche verfügen, sondern nur unter der Windows-Eingabeaufforderung laufen und mit dem Anwender über reine Textein- und -ausgabe kommunizieren. Konsolenanwendungen sind für Programme sehr gut geeignet, die bei der reinen Stapelverarbeitung eingesetzt werden sollen und wenig oder überhaupt keine Kommunikation mit dem Anwender erfordern. Aber gerade auch beim Debuggen – zum Beispiel, um ohne großen Aufwand neue Typen (Klassen, Strukturen) zu testen – leisten sie sehr gute Dienste.

Möchten Sie selber eine neue Kommandozeilenanwendung erstellen, wählen Sie in der Visual Studio-IDE aus dem Menü *Datei* den Menüpunkt *Neu/Projekt* und im Dialog, den Visual Studio anschließend zeigt, aus dem Zweig *Visual Basic* und dem Bereich *Windows* die Vorlage *Konsolenanwendung*.



**Abbildung 8.1:** Kid George vom BCL-Team demonstriert die Möglichkeiten von Konsolenanwendungen eindrucksvoll mit seiner Implementierung von Space Invaders!<sup>1</sup>

Seit Visual Studio 2005, bzw. dem Framework 2.0 gibt es übrigens einige Erweiterungen in der Kommandozeilenunterstützung aus .NET heraus. So haben Sie direkt über die `Console`-Klasse Einfluss auf die Farbgebung, auf die direkte Positionierung der jeweils nächsten Ausgabe und Sie können sogar ganze Bildschirmbereiche mit einfachen Befehlen verschieben, und damit Scrolling in jede Richtung oder sogar einfache bewegte Bildschirmgrafiken realisieren. Ein Blick in die Hilfe zum `Console`-Objekt lässt Sie hier Interessantes erfahren, und auch die Space Invaders-Demo, die Sie über den IntelliLink D0802 herunterladen können, ist schön anzuschauen – auch wenn der Quellcode in C# und nicht in Visual Basic vorliegt.

---

<sup>1</sup> Wenn Sie den Quellcode dieser Anwendungen herunterladen, müssen Sie – Stand 3.1.2006 – leider erst drei kleine Fehler beseitigen, da die Anwendung noch mit einer Beta-Version von Visual Studio entwickelt wurde. Die Eigenschaft `BackSpace` korrigieren Sie in `BackPressed` und `SpaceBar` an zwei Stellen in `Spacebar`. Die Fehlerliste wird Ihnen beim Finden der Fehler helfen. Bedenken Sie, dass C# die Groß-/Kleinschreibung unterscheidet!

# Das Klassenprinzip am einfachsten Beispiel

Bevor wir uns an die Entwicklung einer eigenen Klasse machen, betrachten wir zunächst den Umgang mit zwei Wertetypen in .NET genauer, die Sie in eigenen Programmen vielleicht schon selbst verwendet haben: Die Datentypen `Decimal` und `Date`.

Der `Decimal`-Datentyp wird verwendet, um gebrochene Werte mit höchster Präzision zu speichern. Er findet Einsatz gerade in finanzmathematischen Anwendungen, weil er durch besondere Rechenverfahren zahlensystembedingte Rundungsfehler ausschließt.

Mit dem `Date`-Datentyp speichern Sie Datumswerte. Sie können mit seiner Hilfe aber auch spezielle Funktionen verwenden, die beispielsweise herausfinden, welche Wochentagsnummer ein bestimmtes Datum hat (1 für Montag, 2 für Dienstag, etc.).

Diese beiden Datentypen, die zu den so genannten *primitiven Datentypen* (da fest im CTS verankert) zählen, bieten sich für die Demonstration einfacher Klassen<sup>2</sup> an, weil man an ihnen am einfachsten erkennt und versteht, was mit der Aussage »Klassen kapseln Daten *und* Programmcode« gemeint ist.

Zu diesem Zweck schauen wir uns ein einfaches Beispielprogramm an.

---

**BEGLEITDATEIEN:** Sie finden die Codedateien zu diesem Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\Design\OOP\Kap08\PrimDataTypes`. Öffnen Sie dort die entsprechende Projektmappen-Datei (`.sln`).

---

Schauen wir uns das Programmchen, bevor wir es starten, zunächst einmal an:

Module Module1

```
Sub Main()
    '*****
    'Klassen bzw. Strukturen speichern Daten...
    '*****

    'Einen Decimal-Datentyp deklarieren und ihm den Wert 11.100,34 zuweisen
    Dim locDecimal As New Decimal(11100.34)
    Console.WriteLine("locDecimal enthält den Wert: " & locDecimal)

    'Einen Date-Datentyp deklarieren und ihm den Wert "10.8.2005" zuweisen
    Dim locDate As New Date(2005, 8, 10)

    '*****
    '...stellen aber auch Programmcode zur Verfügung
    '*****
```

---

<sup>2</sup> Obwohl es sich bei ihnen streng genommen nicht um Klassen sondern um Strukturen handelt. Doch für die ersten Gehversuche ignorieren wir diesen Unterschied zunächst, da er für das Verständnis, um das es zunächst geht, nicht relevant ist, und Klassen und Strukturen diesbezüglich auf gleichen Prinzipien basieren; mehr über die Unterschiede zwischen den beiden Entitätstypen *Klassen* und *Strukturen* erfahren Sie später in diesem Kapitel.

```

Dim locAusgabe As String
'ToString wandelt unter Angabe des Formats den Wert
'von locDecimal in eine Zeichenkette um
locAusgabe = locDecimal.ToString("#,##0.0000 $")
Console.WriteLine("Formatierte Zahlenausgabe: " & locAusgabe)

'DayOfWeek bestimmt den Wochentag des angegebenen Datums.
Dim locWochentag As Integer
locWochentag = locDate.DayOfWeek
Console.WriteLine("Der " & locDate & " war der " & locWochentag & ". Tag der Woche!")

'Auf Tastendruck warten
Console.ReadKey()
End Sub

End Module

```

Wie Sie im Listing erkennen können, besteht dieses Programm aus zwei kleinen Blöcken. Der erste Block deklariert wie angekündigt die zwei Datentypen, weist ihnen Werte zu und gibt diese schließlich als Beweis für ordnungsgemäßes Funktionieren auf dem Bildschirm aus.

Der zweite Block verwendet die beiden Datentypen ein zweites Mal – doch dieses Mal werden Funktionen verwendet, die direkt irgendwelche Operationen mit den Daten durchführen, die die beiden Typen speichern (im oben stehenden Listing fett gedruckt): Die `Tostring`-Funktion des `Decimal`-Wertes wandelt den Wert, den die `Decimal`-Variable speichert, in eine druckbare Zeichenkette um und formatiert die Zahlendarstellung so, dass Tausenderpunkte gezeigt und genau vier Nachkommastellen mit ausgegeben werden – dazu dient die Formatzeichenfolge »#,##0.0000 \$«. Jeder »Lattenzaun« dient dabei für eine mögliche auszugebende Zahl, die berücksichtigt wird, wenn die Zahl ausreichend groß ist. Jede »0« dient für eine Ziffer, die in jedem Fall ausgegeben wird. Ist die Zahl nicht groß bzw. nicht klein genug, um die entsprechenden Ziffern »zu produzieren«, werden an den entsprechenden Stellen Nullen ausgegeben.

Im zweiten Beispiel sorgt die `DayOfWeek`-Funktion des `Date`-Wertes, dass die Wochentagsnummer des Datums errechnet wird, das gegenwärtig in der `Date`-Variablen `locDate` gespeichert wird.

Die Ausgabe sieht dementsprechend folgendermaßen aus:

```

locDecimal enthält den Wert: 11100,34
locDate enthält den Wert: 10.08.2005

Formatierte Zahlenausgabe: 11.100,3400 $
Der 10.08.2005 war der 3. Tag der Woche!

```

Was dieses simple Beispiel verdeutlicht: Selbst einfache Datentypen wie `Decimal`, `Date` aber auch `Integer`, `Double` und andere dienen nicht nur zur Speicherung von Daten. Sie enthalten auch Programmcode. Und das Kapseln dieses Programmcodes im Datentyp (der Struktur oder der Klasse) ist das eigentliche Geniale daran.

Um Daten zu manipulieren, brauchen Sie nicht irgendwelche Funktionen aufzurufen, die sich sonst irgendwo in Ihrem Programm befinden. Sie bedienen sich einfach der Funktionen, die Sie direkt mit in dem »Bereich« Ihres Programms speichern, der auch zur Speicherung der eigentlichen Daten dient. Und dazu dienen eben die Klassen (bzw. Strukturen – doch die sind unter Berücksichtigung unseres momentanen Wissenstandes noch dasselbe).

Bis hierher haben Sie gesehen, was es heißt, in der CLR vorhandene Klassen bzw. Strukturen zu verwenden, die es ermöglichen, Daten und Programmfunktionalität unter einem Dach zu kapseln. Sie konnten nachvollziehen, dass gerade Code umfangreicher Projekte auf diese Weise sehr viel einfacher wieder zu verwenden ist, da er – etwa wie ein Kontextmenü des Elements einer Windows-Anwendung – quasi an dem Objekt »hängt«, das er verarbeiten bzw. manipulieren soll. Sie können sich auch sicherlich vorstellen, dass dieses Prinzip angewendet in Ihren eigenen Programmen sehr dazu beiträgt, dass diese aufgeräumter, klarer strukturiert und auch sehr viel leichter lesbar werden.

## Das Klassenprinzip am eigenen Beispiel

Wie es nun aussieht, eigene Datentypen in Form von eigenen Klassen (und später auch in Form von eigenen Strukturen zu entwerfen), soll das Beispiel in diesem Abschnitt demonstrieren.

Ziel unseres ersten Klassengehversuchs soll es sein, die Problemstellung aus dem letzten Kapitel (Sie erinnern sich an die antiquarischen Bücher) ein wenig OOP-näher zu verpacken. Auch dazu finden Sie in den Begleitdateien ein vorbereitetes Programmbeispiel.

---

**BEGLEITDATEIEN:** Sie finden die Codedateien zu diesem Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\Design - OOP\Kap08\RomNum_Net01`. Öffnen Sie dort die entsprechende Projektmappen-Datei (`.sln`).

---

Die Klasse, mit der wir die Problemlösung beginnen, soll zunächst nur die Aufgabe haben, einen Wert vom Typ `RomanNumerals` zu speichern. Dabei ist es natürlich klar, dass wir das Framework nicht dazu bringen werden, römische Numeralia in nativer Form zu speichern. Vielmehr kann unsere Klasse den eigentlichen Wert nur durch einen der bereits vorhandenen primitiven Datentypen speichern und on top Konvertierungsfunktionen zur Verfügung stellen, die sich um die entsprechenden Umwandlungen von römischen Numeralia, die in Zeichenkettenform vorliegen, in numerische Werte und umgekehrt kümmern. Oder anders gesagt: Der Plan ist es, einen eigenen Datentypen zu schaffen, der in etwa wie ein schon vorhandener Datentyp funktioniert – beispielsweise wie ein `Integer` (der ganze Zahlen in einem bestimmten Zahlenbereich), ein `String` (der Zeichenketten) oder `Double` (der gebrochene Werte speichert). Unser neuer Typ speichert ebenfalls einen Wert, etwa wie der Datentyp `Integer`. Doch gleichzeitig stellt dieser neue Typ auch Funktionen bereit, um römischen Numeralia in numerische Werte, oder gespeicherte Werte wieder zurück in römische Numeralia umzuwandeln.

Nicht weniger als fünf Funktionen braucht unsere Klasse dazu. Wieso fünf? Aus folgendem Grund: Wir werden zwei Funktionen haben, die unabhängig vom Wert verwendbar sind, den die Klasse speichert. Sie stellen die eigentlichen »Malocher« in der Klasse dar, und die eine wandelt römische Numeralia, die als `String` vorliegen, in `Integer`-Werte um; die andere macht das Umgekehrte. Und dann haben wir zwei Funktionen, die das Gleiche mit dem Wert machen, den die Klasse selbst speichert.

Schauen wir uns die Beschreibung der Funktionen an:

- `RomanNumeralFromValue` wandelt die angegebene arabische Zahl (ganz normaler `Integer`) in einen `String` um, der dann das entsprechende römische Numerale enthält. Das war in der prozeduralen Beispielversion des letzten Kapitels schon so.
- `ValueFromRomanNumeral` wandelt den angegebenen `String`, der das römische Numerale enthält, wieder zurück in einen `Integer`-wert – die arabische Zahl. Auch das war schon so in der prozeduralen Version.

- FromRomanNumeral weist den angegebenen String, der ein römisches Numerale enthält, dem »Klassenelement« zu. Das ist neu, denn in der an den VB6-Stil angelehnten Version dieses Beispiels gab es nichts, wohinein man das römische Numerale hätte speichern können.
- ToRomanNumeral hat gar keinen Parameter. Es wandelt das »Klassenelement« zurück in einen String, der das römische Numerale darstellt.

Es gibt im Übrigen noch eine fünfte Funktion in dieser Klasse, nämlich:

- FromInt verhält sich prinzipiell wie FromRomanNumeral, nimmt allerdings kein römisches Numerale sondern einen normalen Integer-Wert als Parameter entgegen, um die Klasseninstanz mit einem Wert zu füllen.

Sie können also wahlweise nur die »festen« Funktionen der Klasse verwenden, um Elemente hin- und herzukonvertieren, ohne irgendetwas zu speichern. Sie können aus der Klasse aber auch ein »Klassenelement machen« – korrekt muss es heißen: Sie instanzieren die Klasse und erhalten ein Objekt – um dann im RomanNumeral-Objekt selbst den Wert für die römische Darstellung zu speichern. Sie brauchen in diesem Fall keine spezielle Variable mehr (wie im vorherigen Beispiel), die den »unterliegenden« Wert speichert.

Der folgende Abschnitt arbeitet die Unterschiede zwischen den beiden Funktionstypen heraus.

## Statische und nicht-statische Methoden und Variablen

Im Prinzip unterscheidet sich der neue Datentyp RomanNumerals nicht von »normalen«, so genannten primitiven Datentypen: Ein Integer speichert beispielsweise ganze Zahlen in einem bestimmten Wertebereich. Wenn Sie den Inhalt einer Integervariablen mit Console.WriteLine ausgeben, wird er in arabischen Ziffern dargestellt. Der Datentyp des Beispielprogramms speichert ganze Werte im Bereich von 1-3999; er liefert die Zahlen im Bedarfsfall aber in »römischer Schreibweise« aus.

Schauen Sie sich als nächstes den Inhalt der Klasse RomanNumerals an (doppelklicken Sie dazu im Projektmappen-Explorer auf *RomanNumerals.vb*):

```
Public Class RomanNumerals

    Private myUnderlyingValue As Integer = 1000

    Public Function ToRomanNumeral() As String

        'Statische Funktion aufrufen
        Return RomanNumeralFromValue(myUnderlyingValue)

    End Function

    Public Sub FromNumeral(ByVal RomanNumeral As String)

        'Statische Funktion aufrufen
        myUnderlyingValue = ValueFromRomanNumeral(RomanNumeral)

    End Sub

```

```

Public Sub FromInt(ByVal ArabicInt As Integer)

    'einfach der Instanzvariable zuweisem
    myUnderlyingValue = ArabicInt

End Sub

Public Shared Function RomanNumeralFromValue(ByVal ArabicNumber As Integer) As String

    .
    .
    .

End Function

Public Shared Function ValueFromRomanNumeral(ByVal RomanNumeral As String) As Integer

    .
    .
    .

End Function

End Class

```

Den Inhalt der beiden Funktionen habe ich aus Platzgründen nicht gezeigt, da er in diesem Zusammenhang weniger interessiert und vom eigentlichen Thema nur ablenkt. Später innerhalb dieses Kapitels werde ich für die Demonstration anderer Konzepte nochmals auf die Beschreibung dieser Funktionen zurückkommen.

Betrachten Sie zunächst die Variable `myUnderlyingValue` (im Listing fett markiert). Auf diese Variable kann (fast) vom gesamten Klassencode aus zugegriffen werden, da sie direkt unter dem Klassenkopf definiert wurde. Variablen, die so deklariert werden, dass sie in der ganzen Klasse gültig sind, werden als Klassenvariablen oder als *Member-Variablen* bezeichnet. Man sagt auch: Der Gültigkeitsbereich der Member-Variablen umfasst die gesamte Klasse.

Die drei nächsten Funktionen sind Instanzmethoden der Klasse. Sie lassen sich nur verwenden, wenn die Klasse zuvor *instanziert* wurde. Mit `FromNumeral` können Sie eine Instanz dieser Klasse mit einem Wert füllen und damit die Klasse z. B. folgendermaßen verwenden:

```
Dim Numerale as New RomanNumerals
Numerale.FromNumeral("IV")
```

Um noch mal zurück zur Sandkastenanalogie zu kommen: Sie nehmen das Förmchen `RomanNumerals` und füllen es mit *neuem (New)* Sand. Den geformten Sandklotz, der dabei herauskommt, nennen Sie `Numerale`. Sie können jetzt ein Stöckchen nehmen, und »IV« in den Sandklotz kratzen. (Sie könnten im Übrigen aber niemals die »IV« in das Förmchen – in die Vorlage `RomanNumerals` kratzen – deswegen müssen Sie eine Klasse in ein Objekt instanzieren, bevor Sie seinen Wert verändern können.)

Mit dieser Vorgehensweise haben Sie den Wert 4 der Klasseninstanz von RomanNumerals zugewiesen. Wenn Sie den Wert abrufen wollen, verwenden Sie die Funktion ToRomanNumeral,

```
Dim strVar as String=Numerale.ToRomanNumeral()
```

die den Wert der Klasseninstanz als String zurückliefert.

Wenn Sie von den so genannten *statischen* (den »festen«) Funktionen Gebrauch machen möchten, rufen Sie sie mit dem Klassennamen selbst als Referenz auf. Etwa:

```
Dim strVar as String=RomanNumerals.RomanNumeralFromValue ("IX")
```

Sie haben jetzt im Gegensatz zum vorherigen Beispiel den Inhalt der Klasseninstanz nicht berührt. Statische Methoden (Funktionen) werden als solche bezeichnet, da sie *immer* zur Verfügung stehen – sie müssen die Klasse nicht in ein Objekt instanzieren, um sie verwenden zu können. Natürlich brauchen auch statische Funktionen Daten (Sandkuchen), die sie verarbeiten können – aber in unserem Beispiel liefern Sie der Funktion den Sandkuchen, in den sie die »IV« kratzen soll, gleich mit.

---

**WICHTIG:** In Visual Basic werden statische Funktionen oder Variablen mit dem Schlüsselwort Shared definiert – was leider ein wenig verwirrend ist, denn das Schlüsselwort Static gibt es ebenfalls.

---

## Nicht verwirren lassen: Static und Shared in VB

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. Diese Verwendung von Static gibt es übrigens bei keiner anderen der mit Visual Studio ausgelieferten .NET-Programmiersprachen. 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).

Um Beispiele für statische Methoden im Framework zu finden, brauchen Sie gar nicht lange zu suchen – Sie finden sie schon bei den primitiven Typen. Ein Beispiel:

Mit

```
Dim intTemp as Integer=5  
Dim strTemp as String=intTemp.ToString()
```

wandeln Sie den Wert 5 in eine Zeichenkette um und weisen sie der String-Variablen strTemp zu. Sie haben hier eine nicht-statische Methode der Klasseninstanz (des aus der Klasse hervorgegangenen Objektes, in diesem Fall intTemp) verwendet.

Mit

```
Dim intTemp As Integer  
intTemp = Integer.Parse("1234")
```

hingegen verwenden Sie die statische Funktion der »Klasse«<sup>3</sup> Integer, die versucht, eine Zeichenkette in eine Integer-Variable umzuwandeln. Inhalt von möglicherweise existierenden Instanzen (also wieder von intTemp, das aus Integer entstand) wird dabei nicht verändert oder auch nur in irgendeiner Form verwendet. Sie nutzen Parse nur als eigenständige Funktion (und geben ihr einen eigenen Sandkuchen zur Verarbeitung). Der Sandkuchen, der durch den Wert von intTemp selbst dargestellt wird, trocknet weiterhin unangetastet in der Sonne.

Nicht-statische Member-Variablen sind logischerweise nicht aus statischen Methoden zu erreichen. Platzieren Sie beispielsweise die Zeile

```
myUnderlyingValue=5
```

in der statischen Funktion RomanNumeralFromValue, dann zeigt Ihnen Visual Basic sofort einen Fehler an, etwa wie in Abbildung 8.2 zu sehen.

```
Public Shared Function RomanNumeralFromValue(ByVal Arab  
    Dim locCount As Integer = 0  
    Dim locDigitValue As Integer = 0  
    Dim locRoman As String = ""  
    Dim locDigits As String = ""  
  
    myUnderlyingValue = 5  
    Auf einen Instanzmember einer Klasse kann nicht ohne explizite Instanz einer Klasse von einer/m freigegebenen  
    Methode/Member aus verwiesen werden.  
    locDigits = "IVXLCDM"
```

**Abbildung 8.2:** Was dieser Fehlerhinweis wirklich meint: In statischen Funktionen können Sie nicht auf Member-Variablen einer Instanz zugreifen

Ganz klar: Die statische Funktion RomanNumeralsFromValue kann den Klassen-Member auch gar nicht verwenden, denn wie sollte sich das auswirken? Sie können die Klasse RomanNumerals in 100 verschiedene Objekte instanziert haben, und jedes Objekt hat einen anderen Wert für myUnderlyingValue. Da RomanNumeralFromValue eine statische Klassenfunktion ist, könnte man, falls ein Verändern von myUnderlyingValue erlaubt wäre, vielleicht noch annehmen, dass sich dabei der Wert *aller* zu diesem Zeitpunkt instanziierten Klassen ändert, aber das ergibt letzten Endes überhaupt keinen Sinn. Es wäre so, als hätten Sie 100 Sandkuchen aus einem Förmchen erzeugt, und würden erwarten, dass sich die schon zum Trocknen ausgelegten Sandkuchen veränderten, wenn Sie das Förmchen selbst vielleicht mithilfe eines Feuerzeugs heiß machten und umformten.

---

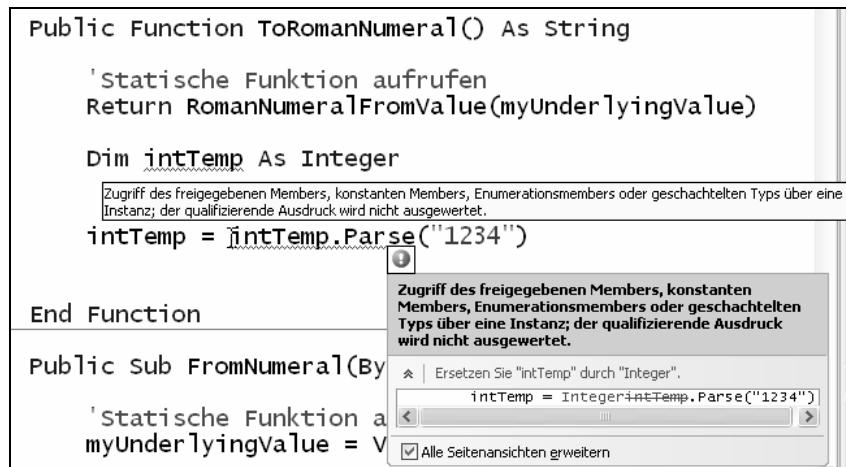
<sup>3</sup> In Anführungszeichen deswegen geschrieben, da es sich bei Integer nicht wirklich um eine Klasse, sondern um eine Struktur handelt. Doch dazu später mehr.

Eine kleine Anmerkung zu einer Tatsache, die leicht verwirrt: Um die statischen Funktionen einer Klasse oder Struktur aufzurufen, können Sie übrigens nicht nur den Weg über den Klassen- bzw. Struktturnamen, sondern auch den über die Instanz der Klasse nehmen. Die Zeilen

```
Dim intTemp As Integer  
intTemp = intTemp.Parse("1234")
```

bewirken exakt dasselbe, wie der Aufruf von `Integer.Parse`, also wie der Weg über den Klassennamen selbst – allerdings ist das kein toller Stil, der auch dem Compiler seit Visual Studio 2005 Anlass zum Meckern gibt (siehe Abbildung 8.3).

**TIPP:** Wie in Abbildung 8.3 zu sehen, gibt es – vielleicht kennen Sie das schon von Office 2003 – im Visual Studio-2005-Editor nach einer Bearbeitung eines Objektes bzw. Objektnamens so genannte Smarttags; so Sie sich ► Kapitel 3 zu Gemüte geführt haben, wissen Sie darüber längst Bescheid. Der folgende graue Kasten fasst den Umgang mit Smarttags nochmals kurz zusammen.



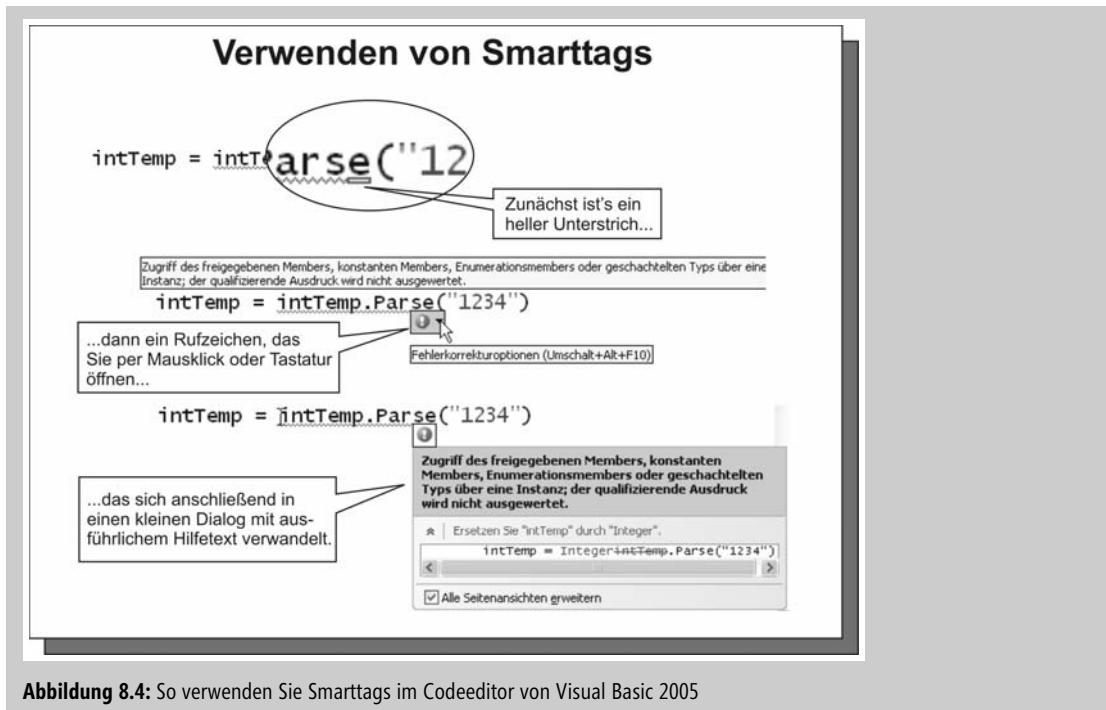
**Abbildung 8.3:** Ist zwar gestattet, aber nicht schön: Aufrufen von statischen Methoden über eine Objektvariable. Verbesserungsvorschläge gibt es durch den Smarttag-Klick (auf's Rufzeichen)

## 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 Smart Tag ein Dialog, der Ihnen einen Korrekturvorschlag für den erkannten »Fehler« unterbreitet.

Die folgende Abbildung verdeutlicht den Zusammenhang. ►



**Abbildung 8.4:** So verwenden Sie Smarttags im Codeeditor von Visual Basic 2005

Schauen Sie sich zum Abschluss das Hauptprogramm und dessen Verhalten in Aktion an. Wenn Sie das Programm starten, werden Sie aufgefordert, eine Zahl einzugeben. Das Programm nimmt die Zahl entgegen und zeigt Ihnen das entsprechende römische Numerale auf dem Bildschirm an. Die Klasse, die das Hauptprogramm enthält, befindet sich übrigens in der Klassendatei *Main.vb*, die Sie per Doppelklick im Projekt-Explorer zum Bearbeiten öffnen können.

```
Public Class Main
    Shared Sub Main()
        Dim locRomanNumeral As New RomanNumerals

        'Text ausgeben; Anwender zur Eingabe auffordern.
        Console.WriteLine("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")

        'Zahl als Text einlesen, mit der statischen Methode Parse in Integer
        'umwandeln und das Ergebnis der Klasseninstanz zuweisen.
        locRomanNumeral.FromInt(Integer.Parse(Console.ReadLine()))

        'Das römische Literale ausgeben, das in der Klasseninstanz gespeichert ist...
        Console.WriteLine("Entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)

        'Dies nur noch, damit nicht alles sofort wieder verschwindet...
        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden...")
        Console.ReadLine()
    End Sub
End Class
```

Dieses Programm veranschaulicht noch mal das gerade Besprochene. Es legt mit New eine neue Instanz der Klasse RomanNumerals mit dem Objekt locRomanNumeral an. Es benutzt eine statische Funktion des Integer-Typs, um die von der Tastatur durch ReadLine eingelesene Zeichenkette in einen Integer-Wert umzuwandeln. Und es verwendet eine Instanzfunktion von locRomanNumeral (also der Klasse RomanNumerals), um den Wert des RomanNumeral-Objektes als römisches Numerale auszugeben.

## Kleiner Exkurs – womit startet ein Programm?

Wie Sie sicherlich unschwer bemerkt haben, startet das Programm mit der Prozedur *Sub Main*. Wenn Sie hingegen eine Windows-Applikation entwickeln, startet sie standardmäßig mit der Darstellung des als erstes dem Projekt hinzugefügten Formulars. Natürlich können Sie bestimmen, wie Ihr Programm starten soll. Und auch wenn Sie eine Windows-Applikation entwickeln, muss diese nicht notwendigerweise mit einem Formular »beginnen«.

Sie legen das Startobjekt der Anwendung fest, indem Sie den Eigenschaftendialog des Projektes anzeigen lassen. Diesen bringen Sie auf den Bildschirm, indem Sie mit der rechten Maustaste auf den Projektnamen im Projekt-Explorer klicken und im Menü, das sich jetzt öffnet, *Eigenschaften* auswählen. Im aufklappbaren Listenfeld *Startobjekt* wählen Sie aus, wie Ihr Programm starten soll.

Darüber hinaus haben Sie für Windows-Anwendungen durch das so genannte Anwendungsframework auch die Möglichkeit, in den Startprozess einzugreifen. Bevor in einer Windows-Anwendung also das erste Formular angezeigt wird, können Sie bereits eine Ereignisbehandlungsroutine implementieren, in der Sie bestimmte notwendige Vorbereitungen erledigen. Mehr zum Anwendungsframework für Windows Forms-Anwendungen erfahren Sie in ► Kapitel 26.

## Mit Sub New bestimmen, was beim Instanziieren passiert – der Klassenkonstruktor

Sicherlich haben Sie bemerkt, dass die Klasse aus dem ersten Beispiel ein wenig unhandlich war. Sie musste erst instanziert werden, und anschließend konnten Sie dem aus ihr hervorgehenden Objekt durch die Benutzung einer Methode einen Wert zuweisen. Von Klassen, die Sie aus dem Framework möglicherweise bereits verwendet haben, wissen Sie, dass die Instanzierung 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.

Bisher instanzieren Sie die Klasse mit einer Codezeile und weisen der Klasseninstanz einen Wert mit einer weiteren Codezeile zu:

```
Dim locRomanNumeral As New RomanNumerals  
locRomanNumeral.FromInt(Integer.Parse(Console.ReadLine()))
```

Einfacher wäre es, wenn Sie schon beim New in der ersten Zeile bestimmen könnten, welchen Wert die Objektinstanz annehmen soll, etwa mit:

```
Dim locRomanNumeral As New RomanNumerals(Integer.Parse(Console.ReadLine()))
```

Ab Visual Basic .NET ist das kein Problem. Im Gegensatz zu Visual Basic 6.0 kennt Visual Basic .NET die Sub New für Klassen. Eine solche Funktion nennt man den *Klassenkonstruktor*. Sie formulieren einen Klassenkonstruktor genau wie jede andere Prozedur, und damit ist seine Implementierung denkbar einfach.

---

**BEGLEITDATEIEN:** Sie finden die Codedateien zum folgenden Beispiel im Verzeichnis .\VB 2005 - Entwicklerbuch\Didaktik\OOP\Kap08\RomNum\_Net02. Öffnen Sie dort die entsprechende Projektmappe-Datei (.sln).

---

Lassen Sie uns die veränderte Version der Klasse betrachten:

```
Public Class RomanNumerals

    Private myUnderlyingValue As Integer

    Public Function ToRomanNumeral() As String

        'Statische Funktion aufrufen.
        Return Me.RomanNumeralFromValue(myUnderlyingValue)

    End Function

    Public Sub New(ByVal RomanNumeral As String)

        'Statische Funktion aufrufen.
        myUnderlyingValue = ValueFromRomanNumeral(RomanNumeral)

    End Sub

    Public Sub New(ByVal ArabicInt As Integer)

        'Einfach der Instanzvariablen zuweisen.
        myUnderlyingValue = ArabicInt

    End Sub

    Public Shared Function RomanNumeralFromValue(ByVal ArabicNumber As Integer) As String
        ...'Zuviel Programmcode; dieser Part interessiert momentan nicht.
        .
        .
        .
    End Function

    Public Shared Function ValueFromRomanNumeral(ByVal RomanNumeral As String) As Integer
        .
        .
        .
    End Function

End Class
```

Whoops, was ist das? Nicht nur, dass in dieser Version zwei Funktionen fehlen – Sub New ist hier wie angekündigt vorhanden und das sogar doppelt!

Warum? Die beiden veralteten Funktionen `FromNumeral` und `FromInt` dienten bislang lediglich der Initialisierung einer Klasseninstanz. Mit der überarbeiteten Version können Sie eine Klasseninstanz schon im Konstruktor definieren, und damit sind die beiden Funktionen hinfällig.

Wenn Sie möchten, dass Code im Konstruktor einer Ihrer Klassen ausgeführt wird, in dem Sie dann beispielsweise Member-Variablen initialisieren können, dann implementieren Sie eine Sub `New`. Der einzige Unterschied zu normalen Funktionen besteht darin, dass der Prozedurenname aus einem Schlüsselwort besteht, 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 schon das instanzierte Objekt sozusagen als »Funktionsergebnis« von `New` zurückliefert.<sup>4</sup>

Zurück zum Beispielprogramm: Um die Klasseninstanz zu deklarieren und ihr einen Wert zuzuweisen, benötigen Sie mit dieser Version des Beispielprogramms nur noch eine einzige Zeile. Sub `Main` des Beispielprogramms dieser Version sieht folgendermaßen aus:

```
Public Class Main

    Shared Sub Main()

        'Text ausgeben; Anwender zur Eingabe auffordern.
        Console.WriteLine("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")

        'Instanz der Klasse RomanNumerals bilden UND
        'Zahl als Text einlesen, mit der statischen Methode Parse in Integer
        'umwandeln und das Ergebnis der Klasseninstanz zuweisen.
        Dim locRomanNumeral As New RomanNumerals(Integer.Parse(Console.ReadLine))

        'Das römische Literale ausgeben, das in der Klasseninstanz gespeichert ist.
        Console.WriteLine("Entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)

        'Dies nur noch, damit nicht alles sofort wieder verschwindet.
        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden...")
        Console.ReadLine()

    End Sub

End Class
```

Was dieses Beispiel anbelangt, stellt sich nur noch die Frage, worin der Sinn von zwei Konstruktoren in der Beispielklasse besteht. Doch lassen Sie mich aus gegebenem Anlass zuvor einen kleinen Ausritt in die Abwärtskompatibilität von Code Libraries nehmen, bevor wir zum eigentlichen Thema zurückkommen:

---

<sup>4</sup> Wobei diese Erklärung rein technisch natürlich falsch ist, und lediglich als Eselsbrücke dienen soll.

# Überflüssige Funktionen mit dem Obsolete-Attribut markieren

Machen Sie das hier im Beispiel gezeigte, bitte AUF KEINEN FALL in Ihren eigenen Klassen. Löschen Sie keine Funktionen oder sonstige Elemente bei Assemblies, die Sie schon im Einsatz haben. Sie riskieren dadurch, dass Anwendungen, die bereits mit einer älteren Version einer solchen Klasse arbeiten, nicht mehr funktionieren.

Stattdessen können Sie eine Technik des Frameworks verwenden, die es erlaubt, Elemente einer Klasse oder die Klasse selbst zu markieren – so genannte *Attribute*. Attribute selbst basieren auf der Attribute-Klasse des Frameworks, die allerdings selbst, mit Ausnahme bestimmter Informationen die sie speichern, über keinerlei Funktionen verfügen.

Sie dienen nur dazu, das markierte Element näher zu beschreiben, und damit verhalten sie sich, wie beispielsweise verschiedene Schriften in einem Text. Zeichnen Sie in Ihren Texten bestimmte Passagen mit verschiedenen *Attributen* aus, dann machen Sie den Leser auf gewisse Dinge **aufmerksam** (so wie hier mit dem Attribut Fett- oder Kursivschrift). Dem gleichen Zweck dienen Attribute zur Markierung von Klassen und Klassenelementen.

Stattdessen können Sie ein Element, das nicht mehr benutzt werden soll, mit dem *Obsolete*-Attribut kennzeichnen. Das *Obsolete*-Attribut vor einem Funktionsnamen bewirkt, dass der Compiler, wenn er die Verwendung eines derart gekennzeichneten Elements entdeckt, eine Warnung oder einen Fehler ausgibt.

OK, zugegeben, ich hab das in diesem Beispiel auch nicht gemacht. Die beiden Funktionen sind noch vorhanden – sie waren nur durch eine #Region-Direktive ausgeblendet – Sie konnten das natürlich nur dann bemerken, wenn Sie sich den Quellcode nicht nur hier im Buch, sondern auch in der Visual Studio-IDE angeschaut haben.

Hier sind die beiden versteckten Funktionen – die folgenden Zeilen demonstrieren Ihnen dabei gleich die korrekte Verwendung des *Obsolete*-Attributs:

```
#Region "Obsolete Funktionen"
    <Obsolete("Verwenden Sie statt dieser Funktion den Klassenkonstruktor")>
    Public Sub FromNumeral(ByVal RomanNumeral As String)

        'Statische Funktion aufrufen
        myUnderlyingValue = ValueFromRomanNumeral(RomanNumeral)

    End Sub

    <Obsolete("Verwenden Sie statt dieser Funktion den Klassenkonstruktor")>
    Public Sub FromInt(ByVal ArabicInt As Integer)

        'einfach der Instanzvariable zuweisen
        myUnderlyingValue = ArabicInt

    End Sub
#End Region
```



Wenn Sie eine der Funktionen nun spaßeshalber im Beispielprogramm verwenden, macht Sie die IDE in der Aufgabenliste darauf aufmerksam.

```
IlocRomanNumeral.FromNumeral("XI")
```

```
"Public Sub FromNumeral(RomanNumeral As String)" ist veraltet: "Verwenden Sie statt dieser Funktion den Klassenkonstruktor"
```

**Abbildung 8.5:** Verwenden Sie eine als obsolet gekennzeichnete Funktion, werden Sie vom Compiler darauf hingewiesen, dies besser nicht zu tun

Falls Sie möchten, dass die Verwendung einer Funktion, die als obsolet gekennzeichnet ist, den Kompilierungsvorgang sogar fehlschlagen lässt, geben Sie als zweiten Parameter des Obsolete-Attributs einfach True an. Das Programm lässt sich dann solange nicht mehr starten, bis der Entwickler die entsprechenden Maßnahmen ergriffen und seine Anwendung ohne die Verwendung der »alten« Funktion umgeschrieben hat.

## Überladen von Funktionen und Konstruktoren

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 Typreihenfolge 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_anderer_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 Prozeduren 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 Beispiel kommen Sie spätestens beim Anwenden der Klasse RomanNumerals in den Genuss des Komforts von überladenen Funktionen. Sie müssen nicht wissen, welche Funktion beispielsweise für welche Teilaufgabe zuständig ist; sie können einfach die (eine) Funktion verwenden, und IntelliSense<sup>5</sup> 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:

The screenshot shows a code editor window with the following code:

```
Shared Sub Main()
    'Text ausgeben; Anwender zur Eingabe auffordern
    Console.WriteLine("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")

    'Zahl als Text einlesen, mit der statischen Methode Parse in Integer
    'umwandeln und das Ergebnis der Klasseninstanz zuweisen
    Dim locRomanNumeral As New RomanNumerals(Integer.Parse(Console.ReadLine))

```

Intellisense is displayed below the last line of code, showing the option "New (RomanNumeral As String)". A status bar at the bottom indicates "1 von 2 New (RomanNumeral As String)".

**Abbildung 8.6:** IntelliSense hilft Ihnen bei der Auswahl der richtigen Signatur bei überladenen Funktionen – hier ist die Reihenfolge der beiden Programmzeilen übrigens noch nicht stimmig!

Im Gegensatz zu optionalen Parametern können Sie also Methoden implementieren, deren Versionen völlig unterschiedliche Dinge tun. Natürlich haben beide Methodenversionen 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 Numerals), die beiden Methoden sind auch visuell sauber voneinander getrennt.

---

**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. Das heißt im Klartext:

---

<sup>5</sup> 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 8.6 zeigt IntelliSense in Aktion.

```

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 von einander 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:

- 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++<sup>6</sup> unterstützen optionale Parameter.
- Die Verwendung von optionalen Parametern macht Ihren Code schwerer lesbar und kaum wieder verwendbar. 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.

---

<sup>6</sup> Managed C++ nennt sich die .NET-Version von C++, die wie alle anderen .NET-Programmiersprachen verwalteten (managed) Code erzeugt.

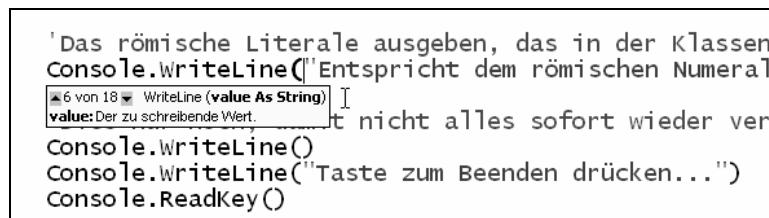
- Sollten Sie ohne strikte Typbindung arbeiten (*Option Strict Off*) und gleichzeitig überladene Funktionen und optionale Parameter für die gleichen Methoden verwenden, bedeutet das einen unglaublichen Leistungsverzicht, da die Laufzeitbibliothek von Visual Basic unter Umständen erst herausfinden muss, 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 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* schreiben und anschließend die geöffnete Klammer eintippen, zeigt Ihnen IntelliSense die Signaturen der 18 verschiedenen Überladungen an.



**Abbildung 8.7:** Die *WriteLine*-Methode hat nicht weniger als 18 Überladungen vorzuweisen!

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 aufrufen – vom Sonderfall *Sub New* einmal abgesehen – ohne Einschränkungen.

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. 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

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 umzustellen:

```

Public Sub New(ByVal RomanNumeral As String)

    'Statische Funktion aufrufen.
    Dim temp As Integer = ValueFromRomanNumeral(RomanNumeral)
    New(temp)

End Sub

Public Sub New(ByVal ArabicInt As Integer)

    'Einfach der Instanzvariablen zuweisen.
    myUnderlyingValue = ArabicInt

End Sub

```

Visual Basic quittiert diesen Aufruf mit einem simplen Syntaxfehler<sup>7</sup> und lässt den Aufruf ganz einfach nicht zu. Mit einem kleinen Trick lässt sich der Fehler zwar nicht abstellen, uns aber dem Ziel ein klein wenig näher kommen. Denn tauschen Sie die zweite Zeile in der ersten Sub New gegen die folgende

```
Me.New(temp)
```

aus, dann lautet die Fehlermeldung in der Fehlerliste:

Ein Aufruf an einen Konstruktor ist nur als erste Anweisung in einem Instanzenkonstruktor gültig.

Aha. Schauen wir einmal, was passiert, wenn wir die Zeilen zu einer zusammenfassen, etwa mit

```
Me.New(ValueFromRomanNumeral(RomanNumeral))
```

sodass sie dadurch in der ersten Zeile steht. Sie werden sehen, jetzt funktioniert es. Natürlich haben wir in diesem Fall nichts gewonnen – weder Programmierarbeit gespart noch die Ausführungs geschwindigkeit erhöht. In anderen Fällen, bei komplexeren Klassen, kann das aber ganz anderes aus schauen.

Wichtig ist 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 in Ihrer Klasse und das New der Klasse, von der Sie Ihre Klasse abgeleitet haben. Da Sie hier im

<sup>7</sup> 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!

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` angäben, 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.

## Hat jede Klasse einen Konstruktor?

Oh ja. Zwar gab es im ersten Beispiel dieses Kapitels eine Version, die nicht einmal über einen Standardkonstruktor<sup>8</sup> im Quellcode verfügte, aber für einen solchen Fall verlangt es das CTS,<sup>9</sup> dass der Compiler automatisch den Code generiert, um einen leeren Standardkonstruktor der Klasse hinzuzufügen. In diesem Standardkonstruktor wird auch der Code platziert, der bei der Initialisierung von Member-Variablen ausgeführt wird. Ein Beispiel: Lassen Sie uns noch einmal zu der ersten Version des Beispielprogramms zurückkehren, in der es noch keine Konstruktoren gab. Um die einzige Member-Variable mit einem Standardwert vorzuinitialisieren, könnten Sie die Zeile wie folgt abändern:

```
Public Class RomanNumerals  
  
    Private myUnderlyingValue As Integer = 1000  
  
    .  
    .  
    .
```

Die Frage: Wann wird diese Initialisierung eigentlich ausgeführt? Zur Beantwortung dieser Frage muss ich ein komplexes Themengebiet teilweise vorwegnehmen, denn Sie müssen wissen, wie Programme unter .NET letzten Endes ablaufen.

Wie im letzten Kapitel 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*.<sup>10</sup> 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.

<sup>8</sup> Als Standardkonstruktor bezeichnet man einen parameterlosen Konstruktor, für den Fall VB also eine Sub `New`, der keine Parameter übergeben werden.

<sup>9</sup> Zur Wiederholung: Das Common Type System ist ein System, das strikte Typbindung und Codekonsistenz erfordert und damit die Common Language Runtime (CLR) zur Coderobustheit zwingt.

<sup>10</sup> Zur Auffrischung: JIT ist die Abkürzung von »Just in time«, auf deutsch etwa »genau rechtzeitig«.

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

Das Framework-SDK (*Software Development Kit*, etwa: *Software-Entwicklungs-Baukasten*), das selbstverständlich Bestandteil von Visual Studio .NET ist, beinhaltet ein paar zusätzliche Tools, die Sie nicht in der Programmgruppe von Visual Studio finden. Einige dieser zusätzlichen Werkzeuge lassen sich zudem nur als Konsolenanwendung verwenden.

Eine Übersicht über die zusätzlichen Werkzeuge (und darüber, wo sie sich oft mit Erfolg vor Ihnen verstecken) finden Sie, indem Sie aus dem Startmenü *Alle Programme* auswählen und in der Programmgruppe *Microsoft .NET Framework SDK 2.0* im Unterpunkt *Tools* das HTML-Dokument *Tools* anklicken.

In der vorliegenden Framework .NET-SDK-Version 2.0 finden Sie die Tools im Verzeichnis `%windir%\Microsoft.NET\Framework\v2.0.xxxx` - die letzten »X« geben dabei die Build-Nummer an<sup>11</sup>; »%windir%« bezeichnet das Basisverzeichnis von Windows.

Andere Werkzeuge, die Bestandteil von Visual Studio .NET sind, wie beispielsweise der Intermediate Language Disassembler (*ILDASM*), den wir als nächstes verwenden werden, sind ebenfalls nicht direkt von der Visual Studio-IDE aus zu erreichen (warum eigentlich nicht?). Sie finden ihn für Visual Studio 2005 im Verzeichnis `C:\Programme\Microsoft Visual Studio 8\SDK\v2.0\Bin` unter dem Namen *ILDASM.Exe* (natürlich nur dann, wenn Sie den vorgeschlagenen Pfad bei der VS-Installation übernommen haben). Mein Vorschlag: Legen Sie eine Verknüpfung zu diesem Programm auf dem Desktop Ihres Computers an, weil Sie ihn später noch des Öfteren gebrauchen werden.

Wichtig dabei ist zu wissen: Wenn Sie die Befehlszeile *Visual Studio 2005 .NET Command Prompt* verwenden, die Sie in der Programmgruppe *Microsoft Visual Studio 2005* des Startmenüs und dort in der Untergruppe *Visual Studio Tools* finden, müssen Sie sich um das Finden der richtigen Pfade keine Gedanken machen, da die *Path*-Variable dieser Befehlszeile entsprechend eingerichtet ist. Das alleinige Eingeben der Programmnamen führt dann in den meisten Fällen zum Ziel.

Nach diesem kurzen Exkurs in die Verzeichnistiefen des SDKs lassen Sie uns zurück zu unserem Vorhaben kommen. Es ist wichtig für das Nachvollziehen des folgenden Szenarios, dass Sie das erste Beispielprogramm (*RomNum\_Net01*) nochmals geladen und erstellt haben, damit die Binärdateien (die eigentlich ausführbaren Programmdateien, die den IL-Code enthalten) vorhanden sind.

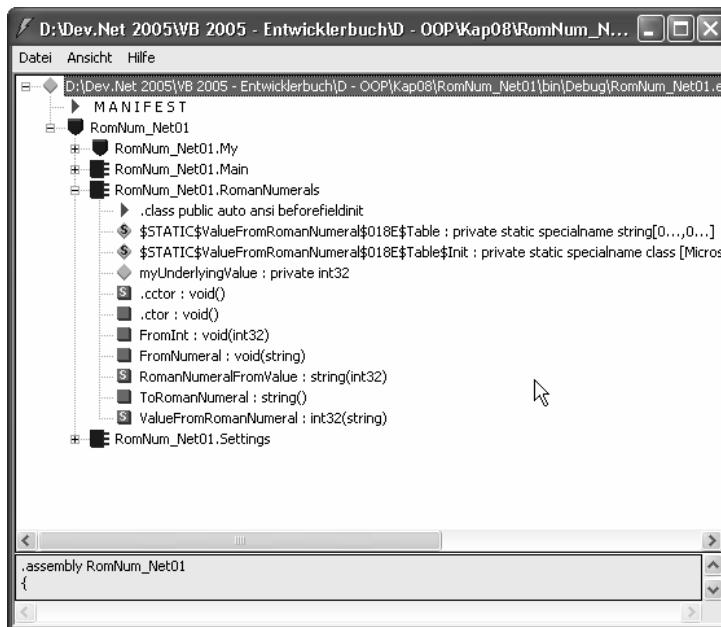
- Falls Sie sich nicht sicher sind, laden Sie das Projekt, und wählen Sie *Projektmappe neu erstellen* aus dem Menü *Erstellen*.
- Starten Sie anschließend den Intermediate-Language-Disassembler *ILDASM* (siehe vorheriger grauer Kasten).

---

<sup>11</sup> Die letzte Build-Version des RTMs (*Release to Manufacturing* – das ist die Version, die dann ausgeliefert wurde und in die Presswerke kam) trug die Versionsnummer 50727. Nach einem Service-Pack kann sich diese Version anpassen.

- Wählen Sie *Öffnen* aus dem Menü *Datei* und im Dialog, der jetzt erscheint, die .EXE-Datei des Beispielprogramms. Sie finden die Programmdatei *RomNum\_Net01.exe* im Projektverzeichnis und dort im Unterverzeichnis *\bin\Debug\*.

**HINWEIS:** Sie werden im Debug-Verzeichnis zwei .EXE-Dateien finden – eine davon endet mit *vhost.exe*. Diese verwenden Sie bitte *nicht*. Bei ihr handelt es sich um ein kleines Hilfsprogramm, dass von der Visual Studio IDE benötigt wird, um 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 8.8:** Die Metadaten der EXE-Datei erlauben die Ansicht der Programmstruktur im IL-Disassembler

- Im Fenster, das anschließend erscheint, öffnen Sie den Zweig *RomNum\_Net01* und den sich darunter befindlichen Zweig *RomNum\_Net01.RomanNumerals* ebenfalls.
- Aus dem Menü *Ansicht* wählen Sie die Option *Quellcodezeilen anzeigen*. Sie sollten anschließend ein Bild vor Augen haben, das etwa dem in Abbildung 8.8 entspricht.

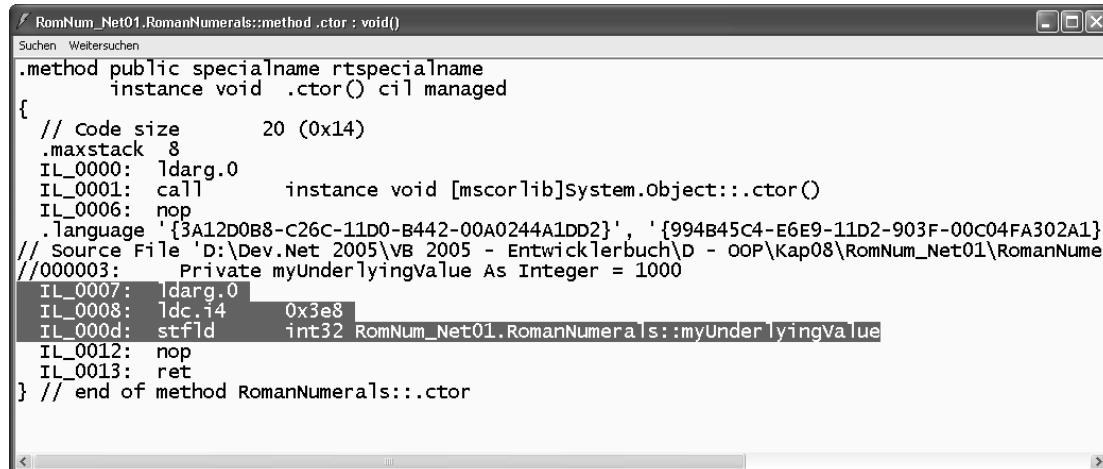
In dieser Abbildung sehen Sie die Struktur des Programms. Sie finden Elemente wieder, die Sie selber bestimmt haben – beispielsweise die Funktionsnamen. Sie erkennen an den vorangestellten Symbolen, welche Strukturen statisch sind und welche nicht-statisch. Und Sie erkennen ebenfalls zwei Methoden (jeweils eine statische und eine nicht-statische), für die Sie nicht der unmittelbare Urheber waren – die Methoden *.ctor* und *.cctor*.

---

**HINWEIS:** An dieser Stelle können Sie übrigens sehr schön erkennen, was für ein Aufwand intern betrieben werden muss, um statische Variablen im VB6-Sinne zu ermöglichen. Zuständig dafür sind die beiden Einträge »\$STATIC\$ ...«, die das Array als klassenglobal-statisch (im CTS-Sinne) deklarieren, die Verwendung aber auf ValueFromRomanNumeral limitieren.

---

Fürs erste beschäftigen wir uns mit dem Standardkonstruktor der Klasse, mit .ctor. Obwohl Sie in der Klasse keinen VB-Code für eine Sub New ohne Parameter platziert hatten, ist er dennoch vorhanden. Ein Doppelklick offenbart, wozu er in unserem Beispiel dient (siehe Abbildung 8.9).



The screenshot shows the IL-Disassembler window with the assembly code for the constructor. The code is as follows:

```
RomNum_Net01.RomanNumerals::method .ctor : void()
Suchen Weitersuchen
.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}'
    // Source File 'D:\Dev.Net 2005\VB 2005 - Entwicklerbuch\Ch - OOP\Kap08\RromNum_Net01\RomanNumerals.cs'
    // Line 13
    IL_0003: Private myUnderlyingValue As Integer = 1000
    IL_0007: ldarg.0
    IL_0008: ldc.i4    0x3e8
    IL_000d: stfld    int32 RomNum_Net01.RomanNumerals::myUnderlyingValue
    IL_0012: nop
    IL_0013: ret
} // end of method RomanNumerals::ctor
```

**Abbildung 8.9:** Der disassemblierte IML-Code des Standardkonstruktors

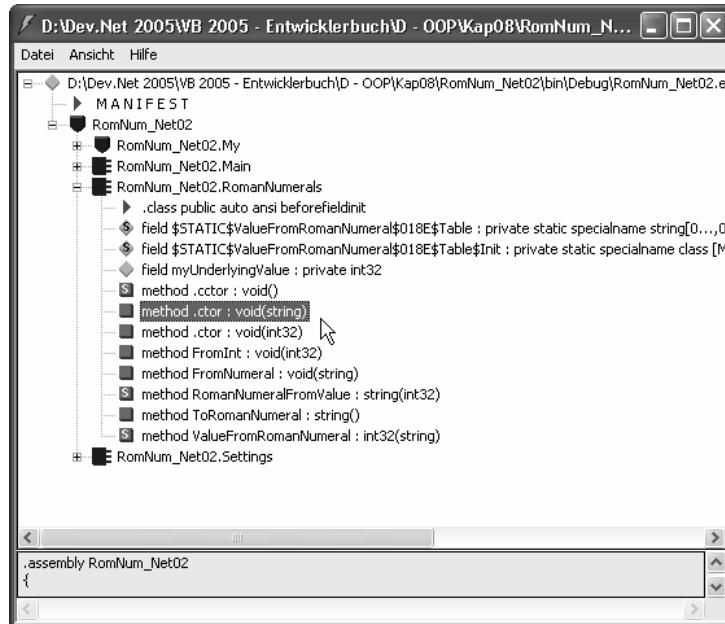
Auch ohne genau zu wissen, welche IL-Anweisung für welche Aufgabe zuständig ist, wird eines doch sofort deutlich: Im ersten Teil der Methode wird der Standardkonstruktor der Basisklasse aufgerufen. Anschließend wird die Variable myUnderlyingValue im zweiten Part der Methode mit dem Wert 1000 (hexadezimal 3E8) initialisiert.

Dieses Beispiel zeigt, 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,
- dass innerhalb des Standardkonstruktors die 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 Sie dies bei deren Deklarierung angegeben haben.

Schauen wir uns zum Vergleich den Konstruktor des zweiten Beispiels an, denn hier haben Sie mit Sub New eine Konstruktorlogik selbst implementiert.

Wählen Sie dazu im IL-Disassembler *Öffnen* aus dem Menü *Datei* und im Dialog, der jetzt erscheint, die .EXE-Datei des zweiten Beispielprogramms (auch die sollte neu erstellt sein, damit nicht nur alle Änderungen, sondern die erforderlichen Unterverzeichnisse überhaupt vorhanden sind). Sie finden die Programmdatei *RomNum\_Net02.exe* im entsprechenden Projektverzeichnis und dort wieder im Unterverzeichnis *\bin\Debug\*.



**Abbildung 8.10:** Die Strukturansicht der zweiten Version des Beispielprogramms

In Abbildung 8.10 können Sie erkennen, dass es keinen Standardkonstruktor, sondern nur die beiden Konstruktoren gibt, die als Sub New auch in der Klasse definiert waren.

Behalten Sie diese Tatsache bitte im Hinterkopf, denn sie wird uns zum Zeitpunkt der Auseinandersetzung mit der Klassenvererbung noch ein wenig beschäftigen.

**HINWEIS:** Schließen Sie Ihre Dateien, nachdem Sie sie im IML-Disassembler betrachtet haben; wenn Sie das Schließen der Dateien vergessen und anschließend eine neue Version kompilieren möchten, gibt Ihnen der Compiler eine Fehlermeldung aus, da die Dateien vom Disassembler gesperrt wurden und darum nicht durch neue Versionen ersetzt werden können.

## Statische Konstruktoren und Variablen

Ich finde, es ist nun an der Zeit, sich ein wenig mit den Prozeduren zu beschäftigen, die den eigentlichen Job in unserem Beispielprogramm erledigen. Insbesondere die Funktion ValueFromRomanNumeral bedarf einer genaueren Betrachtung – hier gibt es nämlich etwas, das unnötig Rechenzeit verschleudert.

```
Public Shared Function ValueFromRomanNumeral(ByVal RomanNumeral As String) As Integer

    Static Table(6, 1) As String
    Dim locCount As Integer
    Dim locChar As Char
    Dim retValue As Integer
    Dim z1 As Integer, z2 As Integer
```

```

If RomanNumeral = "" Then
    Return 0
End If

'Tabelle zum Nachschlagen
'das kann man auch eleganter machen,
'aber für diese Version soll es reichen.
Table(0, 0) = "I" : Table(0, 1) = "1"
Table(1, 0) = "V" : Table(1, 1) = "5"
Table(2, 0) = "X" : Table(2, 1) = "10"
Table(3, 0) = "L" : Table(3, 1) = "50"
Table(4, 0) = "C" : Table(4, 1) = "100"
Table(5, 0) = "D" : Table(5, 1) = "500"
Table(6, 0) = "M" : Table(6, 1) = "1000"

locCount = 0

Do While locCount < Len(RomanNumeral)

    locChar = RomanNumeral.Chars(locCount)

    'VB6-Stil: If locCount < Len(RomanNumeral) - 1 Then
    If locCount < RomanNumeral.Length - 1 Then
        For z1 = 0 To 6
            If Table(z1, 0) = locChar Then Exit For
        Next z1
        For z2 = 0 To 6
            'VB6-Stil If Table(z2, 0) = Mid$(RomanNumeral, locCount + 1, 1) Then
            If Table(z2, 0) = RomanNumeral.Substring(locCount + 1, 1) Then
                Exit For
            End If
        Next z2
        If CInt(Table(z1, 1)) < CInt(Table(z2, 1)) Then

            'Stringfragment entfernen
            'VB6-Stil: RomanNumeral = Left$(RomanNumeral, locCount - 1) + Mid$(RomanNumeral,
            'locCount + 2)
            RomanNumeral = RomanNumeral.Substring(0, locCount - 1) +
                RomanNumeral.Substring(locCount + 2)
            retValue = retValue + (CInt(Table(z2, 1)) - CInt(Table(z1, 1)))
        Else
            For z2 = 0 To 6
                If Table(z2, 0) = locChar Then Exit For
            Next z2
            'VB6-Stil: retValue = retValue + Convert.ToInt32(Val(Table(z2, 1)))
            retValue += CInt(Table(z2, 1))
            locCount += 1
        End If
    Else
        For z2 = 0 To 6
            If Table(z2, 0) = locChar Then Exit For
        Next z2
        retValue += CInt(Table(z2, 1))
    End If
End Do

```

```

    locCount += 1
End If
Loop

'VB6-Stil: ValueFromRomanNumeral = retValue : Exit Function
Return retValue

End Function

```

Wie Sie aus dem Listing erkennen können (fett hervorgehoben), bedient sich die Routine einer Tabelle, um die Zuordnung von römischem Numerale und eigentlichem Wert vorzunehmen. Genau diese Tabelle ist aber Stein des Anstoßes, denn: Sie wird jedes Mal neu aufgebaut, wenn die Funktion aufgerufen wird. Da sie ohnehin nicht viel Speicher benötigt, wäre es ungleich besser, sie ein einziges Mal anzulegen und sie dann bis zum Ende des Programms im Speicher zu belassen.

Dazu bietet sich ein statisches (also ein als Shared deklariertes) Array an, das idealerweise im statischen Konstruktor der Klasse definiert wird. Der Umbau erfordert nur wenige Handgriffe.

---

**BEGLEITDATEIEN:** Sie finden die Codedateien zu diesem Beispiel im Verzeichnis *.\VB 2005 - Entwicklerbuch\Disk\OOP\Kap08\RomNum\_Net03*. Öffnen Sie dort die entsprechende Projektmappen-Datei (*.sln*).

---

```

Public Class RomanNumerals

    Private myUnderlyingValue As Integer = 1000
    Private Shared Table(6, 1) As String

    Shared Sub New()

        'Tabelle zum Nachschlagen
        'Diesmal statisch (Shared) deklariert - das spart schon einmal Zeit.
        Table(0, 0) = "I" : Table(0, 1) = "1"
        Table(1, 0) = "V" : Table(1, 1) = "5"
        Table(2, 0) = "X" : Table(2, 1) = "10"
        Table(3, 0) = "L" : Table(3, 1) = "50"
        Table(4, 0) = "C" : Table(4, 1) = "100"
        Table(5, 0) = "D" : Table(5, 1) = "500"
        Table(6, 0) = "M" : Table(6, 1) = "1000"
    End Sub

```

Neben der Member-Variablen wird direkt unter der Klassendefinition auch das Array deklariert – und zwar als statisches (statisch auch im Sinne des CTS). Zusätzlich zu den schon vorhandenen Konstruktoren gibt es jetzt auch noch einen mit dem Zugriffsmodifizierer Shared, der bewirkt, dass aus dem Konstruktor ein statischer Konstruktor wird.

Die Frage, die sich aus dieser Modifizierung ergibt, lautet: *Was ruft wann* den statischen Konstruktor der Klasse auf? Der Zeitpunkt des Instanzierens der Klasse reicht ja bei weitem nicht aus, weil die statischen Funktionen `RomanNumeralFromValue` und `ValueFromRomanNumeral` nicht funktionierten, wenn nicht zuvor mindestens eine Klasseninstanz erstellt würde.

Um das Verhalten zu verdeutlichen, müssen wir die Debug-Fähigkeiten von Visual Studio bemühen. Visual Studio erlaubt das Setzen von Haltepunkten innerhalb des Programmcodes. Trifft das Programm bei seinem Ablauf auf einen solchen Haltepunkt, wird die Programmausführung unterbrochen, und die Visual Studio-IDE wechselt in den Debug-Modus, in dem Codezeilen Schritt für Schritt ausgeführt werden können. Genau das ist der Plan für die nächsten Schritte.

In der Klasse Main fügen wir dazu eine weitere Instanzierungsanweisung ein, um die Unterschiede beim Verhalten der Konstruktoren im Vergleich zueinander beobachten zu können (beide Stellen sind im Listing fett markiert). Außerdem entzerrern wir die Codezeile, die die Zeichenkette von der Tastatur einliest, damit wir es beim Debuggen einfacher haben. Es ergibt sich folgendes neues Listing für das Hauptprogramm.

```
Public Class Main

    Shared Sub Main()

        'Text ausgeben; Anwender zur Eingabe auffordern
        Console.WriteLine("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")

        'Zahl als Text einlesen, mit der statischen Methode Parse
        'in Integer umwandeln...
        Dim locInt As Integer = Integer.Parse(Console.ReadLine)

        'und das Ergebnis der Klasseninstanz zuweisen.
        ''An dieser Stelle befindet sich der Haltepunkt!
        Dim locRomanNumeral As New RomanNumerals(locInt)

        'Das römische Literale ausgeben, das in der Klasseninstanz gespeichert ist
        Console.WriteLine("Entspricht dem römischen Numerale " & locRomanNumeral.ToString())

        ''Nur zum Testen.
        Dim locOtherRomanNumeral As New RomanNumerals("XXXIV")  ' <<<

        'Dies nur noch, damit nicht alles sofort wieder verschwindet
        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden...")
        Console.ReadLine()

    End Sub

End Class
```

Um die benötigten Haltepunkt zu setzen (einen im Hauptprogramm, einen weiteren im statischen Konstruktor der Klasse RomanNumerals), fahren Sie zunächst mit dem Cursor auf folgende Zeile (fett markiert) der Moduls *Main.vb*.

```
'und das Ergebnis der Klasseninstanz zuweisen.
''An dieser Stelle befindet sich der Haltepunkt!
Dim locRomanNumeral As New RomanNumerals(locInt)
```

und drücken die Taste **F9**. Vor der Zeile erscheint ein kleiner, roter Ball, der den Haltepunkt kennzeichnet. Gleichzeitig wird die Zeile rot hervorgehoben. Öffnen Sie nun im Editor die Codedatei *RomanNumerals.vb*, und platzieren Sie einen weiteren Haltepunkt auf der folgenden Codezeile:

```

Public Class RomanNumerals

    Private myUnderlyingValue As Integer = 1000
    Private Shared Table(6, 1) As String

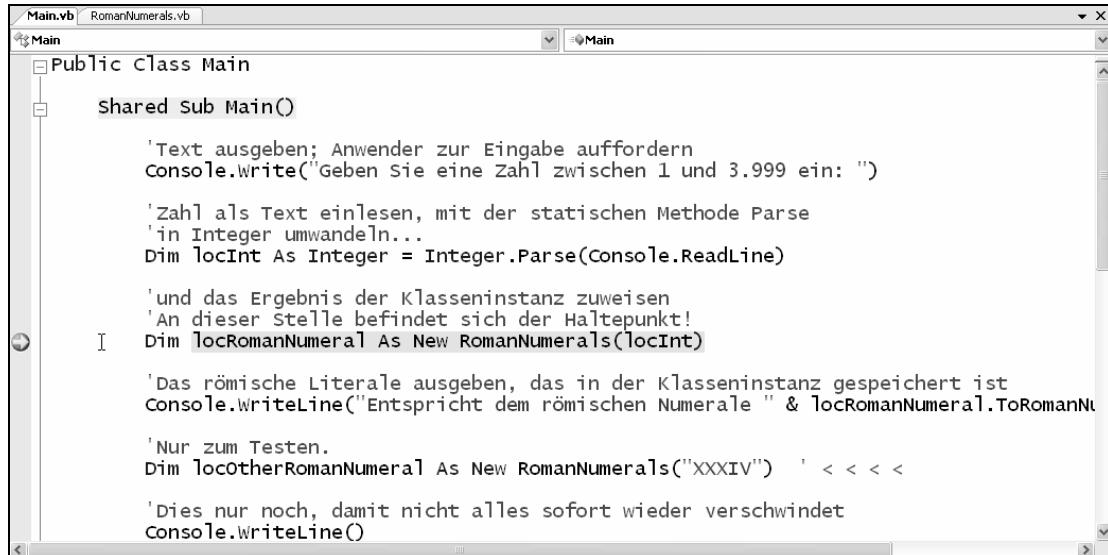
    Shared Sub New()

        'Tabelle zum Nachschlagen
        'Diesmal statisch (Shared) deklariert - das spart schon einmal Zeit.
        'Hier liegt der Haltepunkt:
        Table(0, 0) = "I" : Table(0, 1) = "1"
        Table(1, 0) = "V" : Table(1, 1) = "5"
        Table(2, 0) = "X" : Table(2, 1) = "10"
        Table(3, 0) = "L" : Table(3, 1) = "50"
        Table(4, 0) = "C" : Table(4, 1) = "100"
        Table(5, 0) = "D" : Table(5, 1) = "500"
        Table(6, 0) = "M" : Table(6, 1) = "1000"

    End Sub

```

Starten Sie nun das Programm, indem Sie *Starten* aus dem Menü *Debuggen* anklicken. Sie sehen kurze Zeit später ein Bild, etwa wie in Abbildung 8.11 zu sehen.



**Abbildung 8.11:** Ein Haltepunkt bringt das Programm in den Einzelschrittmodus

Ihr Projekt befindet sich jetzt im so genannten Einzelschrittmodus, mit dem Sie die Codezeilen Schritt für Schritt ausführen und sofort nachvollziehen können, welchen »Weg« das Programm

nimmt und was mit jedem Schritt passiert. Drücken Sie im Folgenden für jeden Programmschritt die Taste **F10**.<sup>12</sup> Sie werden feststellen, dass das Programm, nachdem es die Zeile

```
Dim locRomanNumeral As New RomanNumerals(locInt)
```

erreicht hat, zunächst in den statischen Konstruktor springt, dort die Arrayelemente initialisiert, bevor es anschließend erst in den (nicht-statischen) Konstruktor

```
Public Sub New(ByVal ArabicInt As Integer)
```

gelangt.

Wenn Sie nun weiter durch das Programm »steppen«, gelangen Sie irgendwann zur zweiten Klasseninstanzierung,

'Nur zum Testen.

```
Dim locOtherRomanNumeral As New RomanNumerals("XXXIV")  ' < < <
```

und Sie werden feststellen: Der statische Konstruktor der Klasse *RomanNumerals* wird dieses Mal nicht aufgerufen.

Der statische Konstruktor wird also nur ein einziges Mal aufgerufen, und zwar zu dem Zeitpunkt, zu dem die Klasse das erste Mal in Ihrer Applikation verwendet wird. Damit nutzen Sie statische Konstruktoren immer dann, wenn Sie grundlegende Dinge für eine Klasse vorzubereiten haben, die nur ein einziges Mal erledigt werden müssen.

---

**HINWEIS:** Der statische Konstruktor wird auch aufgerufen, bevor Sie das erste Mal eine *statische* Funktion der Klasse verwenden.

---

## Eigenschaften

Das Beispielprogramm ist durch die Konstruktoren ungleich flexibler und einfacher handhabbar geworden. Durch das Auflösen der vormals zwei Funktionen in Konstruktorroutinen gibt es allerdings jetzt einen Nachteil: In der aktuellen Version gibt es keine Möglichkeit, den Inhalt einer Klasseninstanz im Nachhinein zu verändern.

Es gäbe natürlich nun die Möglichkeit, die Funktionen wieder einzubauen. Funktionen sind allerdings auf eine Richtung des Datenflusses limitiert. Sie können entweder nur dazu herhalten, den Instanzwert zu verändern *oder* ihn abzufragen (OK, eine Funktion könnte ihn auch gleichzeitig verändern *und* abfragen, aber das wäre natürlich totaler Unfug, denn die Eingabe entspräche immer der Ausgabe).

Sie könnten sich in solchen Fällen dadurch helfen, dass Sie eine Funktion beispielsweise namens *GetValue* und eine weitere Funktion namens *SetValue* definieren. Moderne Programmiersprachen wie Visual Basic oder C# kennen dazu einen viel eleganteren Weg über so genannte Eigenschaftenprozeduren (Property-Prozeduren).

---

<sup>12</sup> Mit **F11** führen Sie Codezeilen schrittweise aus und springen bei einem Methodenaufruf in die entsprechende Prozedur. Mit **F10** führen Sie Codezeilen ebenfalls schrittweise aus; Prozeduren werden jedoch als Ganzes abgearbeitet, also wie eine einzige Codezeile behandelt. Da wir einen weiteren Haltepunkt in *Sub Shared...* gesetzt haben, unterbricht die Ausführung dort trotz **F10** (Methodenaufruf »als Ganzes«).

Wenn Sie schon längere Zeit mit Visual Basic (egal, ob mit .NET oder 6.0) programmiert haben, dann haben Sie Eigenschaften natürlich längst kennen gelernt. Mit Hilfe von Eigenschaften können Sie in der Regel bestimmte Zustände von Objekten abfragen *und* verändern. 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, kommt keiner mehr 'ran
```

Soweit die Funktionsweise aus der Sicht desjenigen, der das Objekt später verwendet. Die viel interessantere Frage lautet: 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 folgendermaßen definiert:

```
Property EineEigenschaft() As Datentyp
```

```
Get  
    Return New Datentyp  
End Get  
  
Set(ByVal Value As Datentyp)  
    mach_irgendwas_mit = 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*<sup>13</sup> (*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.

---

<sup>13</sup> Etwa: »Zugreifer«.

Damit haben Sie jetzt das erforderliche Rüstzeug, um eine Eigenschaftenprozedur in die nächste Version des Beispiels einzubauen.

---

**BEGLEITDATEIEN:** Sie finden die Codedateien zu diesem Beispiel im Verzeichnis *.|VB 2005 - Entwicklerbuch|D - OOP\Kap08\RomNum\_Net04*. Öffnen Sie dort die entsprechende Projektmappen-Datei (*.sln*).

---

Sie finden folgende Codezeilen im Klassencode von *RomanNumerals*:

```
Property UnderlyingValue() As Integer  
    Get  
        Return myUnderlyingValue  
    End Get  
  
    Set(ByVal Value As Integer)  
        myUnderlyingValue = Value  
    End Set  
  
End Property
```

---

**TIPP:** Die Visual Basic-Editor unterstützt Sie beim Erstellen von Eigenschaftenprozeduren. Wenn Sie die Eigenschaftendefinition der Eigenschaft in den Editor tippen und am Ende der Zeile die Eingabetaste drücken, fügt Visual Studio den vollständigen Codeblock für das Grundgerüst der Eigenschaftenprozedur für Sie ein.

---

In der *Main*-Klasse können Sie die neue Eigenschaft verwenden, etwa wie mit dem folgenden modifizierten Code von *Class Main*:

```
Public Class Main  
  
    Shared Sub Main()  
  
        'Text ausgeben; Anwender zur Eingabe auffordern  
        Console.WriteLine("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")  
  
        'Zahl als Text einlesen, mit der statischen Methode Parse  
        'in Integer umwandeln...  
        Dim locInt As Integer = Integer.Parse(Console.ReadLine)  
  
        'und das Ergebnis der Klasseninstanz zuweisen  
        'An dieser Stelle befindet sich der Haltepunkt!  
        Dim locRomanNumeral As New RomanNumerals(locInt)  
  
        'Das römische Literale ausgeben, das in der Klasseninstanz gespeichert ist  
        Console.WriteLine("Entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)  
  
        'Nur für den Abstand.  
        Console.WriteLine()  
  
        'Wert mit der neuen Eigenschaft verändern.  
        locRomanNumeral.UnderlyingValue = 200
```

```

' und neues Ergebnis ausgeben
Console.WriteLine("und 200 entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)

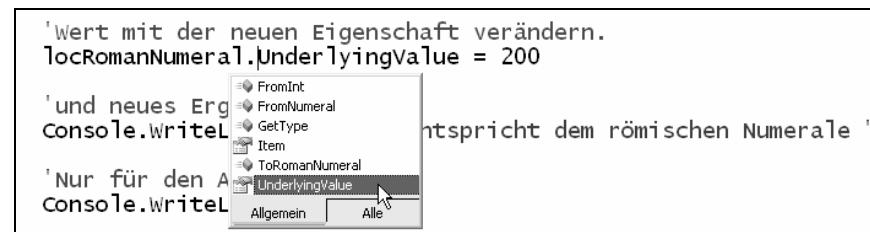
'Dies nur noch, damit nicht alles sofort wieder verschwindet.
Console.WriteLine()
Console.WriteLine("Return drücken zum Beenden...")
Console.ReadLine()

End Sub

End Class

```

Natürlich erkennt auch IntelliSense in der Visual Studio-IDE sofort, dass Sie eine Eigenschaft in der Klasse RomanNumerals eingebaut haben. Sobald Sie nach der Eingabe des Objektnamens `locRomanNumeral` den Punkt auf der Tastatur drücken, können Sie die neue Eigenschaft in der Liste erkennen (siehe Abbildung 8.12).



**Abbildung 8.12:** IntelliSense »lernt« neue Eigenschaften sofort und bietet sie Ihnen im Bedarfsfall direkt an

## Nur-Lesen und Nur-Schreiben-Eigenschaften

Sie haben die Möglichkeit, Eigenschaften auf den Datenfluss in nur eine Richtung zu begrenzen.

Falls Sie möchten, dass eine bestimmte Eigenschaft nur gelesen werden kann, ist Folgendes zu tun:

- Sie verwenden den Modifizierer `ReadOnly` vor der Property-Definition.
- Und Sie sorgen dafür, dass nur ein *Get*-Accessor in der Eigenschaftenprozedur implementiert wird.

### Beispiel für eine Nur-Lesen-Eigenschaft:

```

ReadOnly Property NurLesen() As Integer

    Get
        Return 5
    End Get

End Property

```

---

**TIPP:** Beginnen Sie direkt mit dem `ReadOnly`-Schlüsselwort, wenn Sie die Eigenschaftendefinition der Eigenschaft in den Editor tippen. Sobald Sie am Ende der Zeile die Eingabetaste drücken, fügt Visual Studio den vollständigen Codeblock für das Grundgerüst der Nur-Lesen-Eigenschaft ein und verzichtet auf den *Set*-Accessor.

---

Falls Sie möchten, dass eine bestimmte Eigenschaft nur geschrieben werden kann, ist Folgendes zu tun:

- Sie verwenden den Modifizierer `WriteOnly` vor der Property-Definition.
- Sie sorgen dafür, dass nur ein *Set*-Accessor in der Eigenschaftenprozedur implementiert wird.

### Beispiel für eine Nur-Schreiben-Eigenschaft:

```
WriteOnly Property NurSchreiben() As Integer  
  
    Set(ByVal Value As Integer)  
        MachEtwasMit(Value)  
    End Set  
  
End Property
```

---

**TIPP:** Beginnen Sie direkt mit dem `WriteOnly`-Schlüsselwort, wenn Sie die Eigenschaftendefinition der Eigenschaft in den Editor tippen. Sobald Sie am Ende der Zeile die Eingabetaste drücken, fügt Visual Studio den vollständigen Codeblock für das Grundgerüst der Nur-Schreiben-Eigenschaft ein und verzichtet auf den *Get*-Accessor.

---

## 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 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.  
locRomanNumeral.EigenschaftMitParametern(10, "Klaus") = 5  
  
'Eigenschaft mit Parametern abfragen.  
If locRomanNumeral.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). 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 C# zwar auch Eigenschaften, kann aber nur auf parameterlose Eigenschaften oder Default-Eigenschaften zugreifen (mehr dazu im ► Abschnitt »Default-Eigenschaften« auf Seite 252). Wenn Sie also Klassen im Team oder für die breite Öffentlichkeit entwickeln, sollten Sie auf Eigenschaften mit Parametern nach Möglichkeit ganz verzichten.

---

## Überladen von Eigenschaften

Eigenschaften lassen sich wie Funktionen überladen. Zum Einsatz kommt dieses Prinzip fast ausschließlich bei Default-Eigenschaften (siehe nächster Abschnitt). 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
    Set(ByVal Value As String)
        'nur durch den Rückgabetyp von der ersten unterscheidet.
    End Set
End Property

```

## 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.

## Default-Eigenschaften (Standardeigenschaften)

Default-Eigenschaften (Standardeigenschaften) haben in Visual Basic 6.0 eine erleichternde Funktion für schreibfaule Entwickler gehabt. Mit Hilfe 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.<sup>14</sup> In diesem Fall ergibt das auch Sinn, denn:

Stellen Sie sich vor, Sie entwickeln eine Array-Klasse, die verschiedene Elemente verwaltet. Gäbe es keine Default-Eigenschaft, müssten Sie ein Element auf folgende Weise abfragen:

```
'Index setzen.  
ArrayKlasse.Index = 5  
'Element abfragen.  
MachEtwasMit = ArrayKlasse.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 = ArrayKlasse.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 angeben, und die folgende Zeile wäre ausreichend:

```
MachEtwasMit = ArrayKlasse(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 InternesArray(Index)  
    End Get  
  
    Set(ByVal Value As Integer)  
        InternesArray(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.

---

Mit diesem Wissen können wir das Beispielprogramm um eine weitere Eigenschaft ergänzen – ich nenne sie BaseValues. Mit der BaseValues-Eigenschaft können Sie die Elemente der Konvertierertabelle abfragen. Sie übergeben ihr den Index als Wert zwischen 0 und 6, und die Eigenschaft liefert das entsprechende römische Numerale für eine Basiszahl (z.B. »I« oder »X«) zurück. Über den Sinn dieser Prozedur möchte ich nicht lamentieren – ich gebe zu, dass sie programmtechnisch keinen denkbaren Nutzen hat. Vielleicht benötigt aber einer Ihrer Team-Mitarbeiter diese Tabelle für das

---

<sup>14</sup> Ähnlich wie bei Überladungen, bei denen auch nur durch die Signaturen unterschieden wird, welche der mehreren vorhandenen Funktionen gemeint ist.

Implementieren einer eigenen Funktionalität, und auf diese Weise kann er ohne Probleme auf die benötigten Daten zurückgreifen.

Die entsprechenden Änderungen finden sich ebenfalls schon in der *RomanNET04*-Version, und die statische Default-Eigenschaft sieht folgendermaßen aus:

```
Default ReadOnly Property Item(ByVal Index As Integer) As String
```

```
Get  
    Return Table(Index,0)  
End Get  
  
End Property
```

Diese kleine Eigenschaft verfügt nun über alle besprochenen Komponenten. Sie kann »nur lesen«, verfügt demzufolge auch nur über einen *Get*-Accessor. Sie greift auf eine statische Variable zu, obwohl sie selbst nicht als statisch definiert ist. Und sie ist die Default-Eigenschaft der Klasse, was bedeutet, dass Sie sie direkt mit dem Instanznamen abfragen können. Der einzige Unterschied zu einer echten statischen Eigenschaft: Sie sind nicht in der Lage, auf die Elemente mit dem Klassennamen, sondern nur mit dem Instanznamen zuzugreifen – aber was soll's!

Die Verwendung dieser Eigenschaft im Hauptprogramm demonstriert ihren Umgang:

```
Public Class Main  
    Shared Sub Main()  
        'Text ausgeben; Anwender zur Eingabe auffordern.  
        Console.WriteLine("Geben Sie eine Zahl zwischen 1 und 3.999 ein: ")  
  
        'Instanz der Klasse RomanNumerals bilden UND  
        'Zahl als Text einlesen, mit der statischen Methode Parse in Integer  
        'umwandeln und das Ergebnis der Klasseninstanz zuweisen.  
        Dim locRomanNumeral As New RomanNumerals(Integer.Parse(Console.ReadLine))  
  
        'Das römische Literale ausgeben, das in der Klasseninstanz gespeichert ist.  
        Console.WriteLine("Entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)  
  
        'Nur für den Abstand.  
        Console.WriteLine()  
  
        'Wert mit der neuen Eigenschaft verändern  
        locRomanNumeral.UnderlyingValue = 200  
  
        'und neues Ergebnis ausgeben.  
        Console.WriteLine("und 200 entspricht dem römischen Numerale " & locRomanNumeral.ToRomanNumeral)  
  
        'Nur für den Abstand.  
        Console.WriteLine()  
  
        'Hier wird auf die Default-Eigenschaft zugegriffen.  
        For locCount As Integer = 0 To 6  
            Console.WriteLine("Element Nr. {0} entspricht römischen Numeral {1}", locCount, _  
                locRomanNumeral(locCount))  
        Next
```

```

'Dies nur noch, damit nicht alles sofort wieder verschwindet.
Console.WriteLine()
Console.WriteLine("Return drücken zum Beenden...")
Console.ReadLine()
End Sub
End Class

```

Wenn Sie das Programm starten, verhält es sich wie folgt:

Geben Sie eine Zahl zwischen 1 und 3.999 ein: 2557  
 Entspricht dem römischen Numerale MMDLVII

und 200 entspricht dem römischen Numerale CC

```

Element Nr. 0 entspricht römischen Numeral I
Element Nr. 1 entspricht römischen Numeral V
Element Nr. 2 entspricht römischen Numeral X
Element Nr. 3 entspricht römischen Numeral L
Element Nr. 4 entspricht römischen Numeral C
Element Nr. 5 entspricht römischen Numeral D
Element Nr. 6 entspricht römischen Numeral M

```

Return drücken zum Beenden...

## 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 EineEigenschaft
    Public DieEigenschaft As Integer
End Class

```

erfüllt zunächst nämlich den gleichen Zweck wie die folgende Klasse,

```

Class EineWeitereEigenschaft

    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, denn es geht bei der objektorientierten Programmierung um Datenkapselung. `myDieEigenschaft` ist der eigentliche Datenträger dieser Klasse, und es gilt das Innehaben seiner Verwaltung bis zum Äußersten zu verteidigen. Bei einfachen Sachen mag die erste Alternative noch die bessere, da schnellere sein. Aber wenn Ihre Programme komplexer werden, wird sich der zusätzliche Aufwand für die zweite Methode schnell bezahlt machen. Ihre Datenstruktur bleibt in jedem Fall unberührt und unabhängig, und diese Vorgehensweise garantiert, dass Sie die jederzeit die volle Kontrolle über Ihre Daten behalten. Denken Sie nämlich daran, dass Sie im Set-Accessor auch die Möglichkeit haben, auf eine Wertzuweisung mit Programmcode Einfluss zu nehmen. Sie könnten beispielsweise für die möglichen Wertzuweisungen eine Bereichsprüfung durchführen, und bei Überschreitung entweder die Maximal- oder Minimalwerte erzwingen:

```
Class EineWeitereEigenschaft
```

```
    Private myDieEigenschaft As Integer

    Public Property DieEigenschaft() As Integer
        Get
            Return myDieEigenschaft
        End Get
        Set(ByVal Value As Integer)
            If Value < 0 Then Value = 0
            If Value > 100 Then Value = 100
            myDieEigenschaft = Value
        End Set
    End Property
End Class
```

Ein weiteres wichtiges Argument ist das Ersetzen von Eigenschaften durch die so genannte Polymorphie beim Vererben von Klassen, das ich im nächsten Kapitel beschreiben werde. Eine einmal als öffentlich deklarierte Variable bleibt für alle Zeiten öffentlich. Sie können in vererbten Klassen keine zusätzliche Steuerung hinzufügen. 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. Sie brauchen dazu die ursprüngliche Klasse kein bisschen zu verändern.

## Zugriffsmodifizierer von Klassen, Prozeduren, Eigenschaften und Variablen

Bevor wir uns der Vererbung widmen, möchte ich an dieser Stelle kurz ein paar Worte über die Zugriffsmodifizierer verlieren, mit denen Sie in Visual Basic .NET 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 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          | Als Privat können Klassen nur dann definiert werden, wenn sie geschachtelt in einer anderen Klasse definiert sind. Beispiel:<br><pre>Public Class A     Private Class B         End Class     End Class</pre><br>Public Class C<br>'Zugriff verweigert, Class B ist Private!<br>Dim b as A.B<br>End Class |
| Public               | Public           | Sie können auf die Klasse uneingeschränkt von außen zugreifen, auch aus anderen Assemblies heraus.                                                                                                                                                                                                        |
| Friend               | Assembly         | Sie können innerhalb der Assembly auf die Klasse zugreifen, aber nicht aus einer anderen Assembly heraus.                                                                                                                                                                                                 |
| Protected            | Family           | Es gilt das für Private Gesagte. Zusätzlich gilt: Auch aus der Klasse abgeleitete Klassen können auf die mit <i>Protected</i> gekennzeichneten und geschachtelten »inneren« Klassen zugreifen.                                                                                                            |
| Protected Friend     | FamilyOrAssembly | Der Zugriff auf die geschachtelte Klasse ist in abgeleiteten und von Klassen der gleichen Assembly aus möglich.                                                                                                                                                                                           |

**Tabelle 8.1:** 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. ►         |

| Zugriffsmodifizierer | CTS-Bezeichnung  | Beschreibung                                                                                                              |
|----------------------|------------------|---------------------------------------------------------------------------------------------------------------------------|
| Protected Friend     | FamilyOrAssembly | Nur innerhalb der Klasse, einer abgeleiteten Klasse oder innerhalb der Assembly kann auf die Prozedur zugegriffen werden. |

**Tabelle 8.2:** 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 ProcedurenEbene können Sie eine Variable zusätzlich als `Static` deklarieren. Mehr über den `Static`-Zugriffsmodifizierer erfahren Sie im ► Abschnitt »Statische und nicht-statische Methoden und Variablen« auf Seite 221.

| 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 <code>Private</code> definiert werden, sind es aber standardmäßig.                                                                                                     |
| Public               | Public           | Von außen kann auf die Klassenvariable uneingeschränkt zugegriffen werden. Sie sollten Variablen aber bestenfalls als <code>Protected</code> 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 255. |
| Friend               | Assembly         | Sie können innerhalb der Assembly auf die Klassenvariable zugreifen, aber nicht aus einer anderen Assembly heraus. Es gilt das für <code>Public</code> 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 vererbt werden, als <code>Protected</code> 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 und nicht-statische Methoden und Variablen« auf Seite 221.                                                                                                                                                                                            |

**Tabelle 8.3:** Mögliche Zugriffsmodifizierer für Prozeduren in Visual Basic .NET

Diese Tabellen sollen Ihnen kompakt und auf einen Blick die Zugriffsmodifizierer von Variablen 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-Accesors

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)

        '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

    Public Property EineEigenschaft() As String
        Get
            Return myEineEigenschaft
        End Get
        Private Set(ByVal value As String)
            myEineEigenschaft = value
        End Set
    End Property
End Class
```

---

**BEGLEITDATEIEN:** Sie finden die Codedateien zu diesem Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\DE-OOP\Kap08\PropertiesDemo`. Öffnen Sie dort die entsprechende Projektmappe-Datei (`.sln`).

---

Dieses kleine Beispiel besteht aus zwei Einheiten – einem Modul und einer Klasse. Die Klasse `EigenschaftenTest` enthält einen parametrisierten Konstruktor sowie eine Eigenschaft, die aber Accessoren mit unterschiedlichen Zugriffsmodifizierern hat.

Innerhalb der 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 Zurverfügungstellen von Assemblies (Klassenbibliotheken) für anderer 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.

# 9 Klassenvererbung und Polymorphie

---

|     |                                                                                                        |
|-----|--------------------------------------------------------------------------------------------------------|
| 261 | <b>Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance)</b>                                  |
| 271 | <b>Überschreiben von Methoden und Eigenschaften</b>                                                    |
| 275 | <b>Das Speichern von Objekten im Arbeitsspeicher – und die daraus resultierende Vorsicht mit ihnen</b> |
| 279 | <b>Polymorphie</b>                                                                                     |
| 294 | <b>Abstrakte Klassen und virtuelle Prozeduren</b>                                                      |
| 297 | <b>Schnittstellen (Interfaces)</b>                                                                     |
| 311 | <b>Die Methoden und Eigenschaften von Object</b>                                                       |
| 318 | <b>Shadowing (Überschatten) von Klassenprozeduren</b>                                                  |
| 324 | <b>Sonderform »Modul« in Visual Basic</b>                                                              |
| 324 | <b>Singleton-Klassen und Klassen, die sich selbst instanzieren</b>                                     |

---

## **Wiederverwendbarkeit von Klassen durch Vererbung (Inheritance)**

Vererbung und die Wiederverwendbarkeit und Möglichkeit zur Erweiterung von Klassen sind der zentrale Bestandteil im Framework. Ich würde sogar soweit gehen zu sagen, dass ohne die Möglichkeit, Klassen zu vererben, .NET keinen Sinn machen würde.

Visual Basic 6.0 beherrschte die Polymorphie<sup>1</sup> – 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 – bei der eine Klasse eine andere Klasse als Member-Variable eingebunden und deren Eigenschaften durch neue Funktionen und Eigenschaftenprozeduren nach außen offen gelegt hat. In VB6 war die reine Polymorphie auf Schnittstellen beschränkt – mehr zu diesem Thema gibt's im ► Abschnitt »Schnittstellen (Interfaces)« ab Seite 297

---

<sup>1</sup> Etwa: »Vielgestaltigkeit«.

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 2005 - Entwicklerbuch\D - OOP\Kap09\Vererbung01`, aber ich würde Sie bitten, es zur Übung und zum besseren Verständnis von Grund auf mit nachzustellen.

---

- 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.

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 Wertzuweisung 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 *ErsterKlasse* 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 zu einfügen und den Punkt eintippen, sehen Sie, wie IntelliSense Ihnen die Elemente der neuen Klasse anbietet:

```

mdlMain.vb*
mdlMain
Module mdlMain
    Sub Main()
        Dim klasse1 As New ErsteKlasse
        klasse1.EinWert = 5
        Console.WriteLine(klasse1.AlsZeichenkette())

        Dim klasse2 As New ZweiteKlasse
        klasse2.

        Console.
        Console.
        Console.
        Console.

        End Sub
    End Module

```

**Abbildung 9.1:** 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 wie es im Beispiel schon kennen gelernt 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

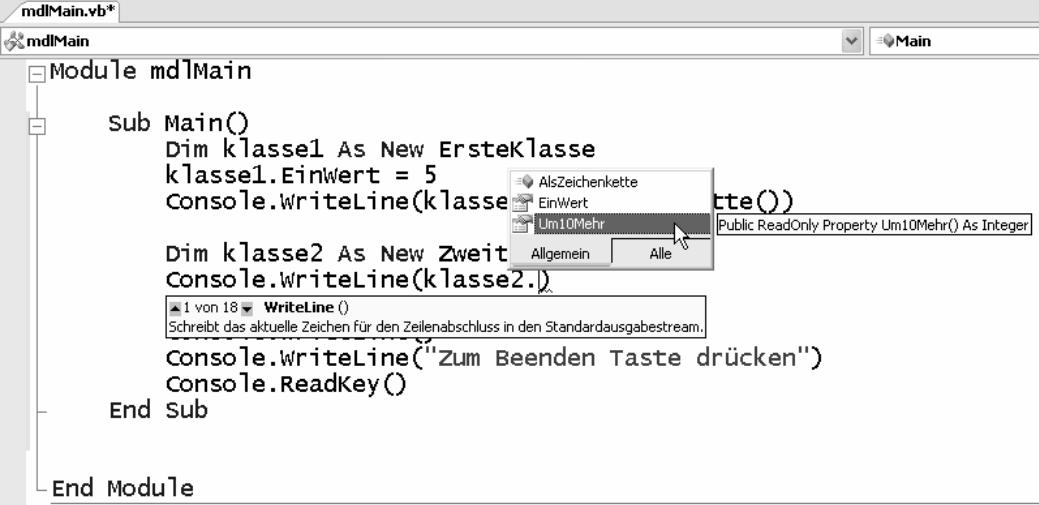
```

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!



The screenshot shows the code editor window for 'mdlMain.vb'. The cursor is at the end of the line 'Dim klasse2 As New ZweiteKlasse'. An Intellisense dropdown menu is open, listing three members: 'AlsZeichenkette', 'EinWert', and 'Um10Mehr'. The 'Um10Mehr' item is highlighted with a red box. Below the dropdown, a tooltip provides the documentation: 'Public ReadOnly Property Um10Mehr() As Integer'. At the bottom of the Intellisense list, it says '1 von 18 WriteLine()' and 'Schreibt das aktuelle Zeichen für den Zeilenabschluss in den Standardausgabestream.'

```
mdlMain.vb*
mdlMain
Module mdlMain
    Sub Main()
        Dim klasse1 As New ErsteKlasse
        klasse1.EinWert = 5
        Console.WriteLine(klasse1.EinWert)
        Dim klasse2 As New ZweiteKlasse
        Console.WriteLine(klasse2.Um10Mehr())
        Console.WriteLine("Zum Beenden Taste drücken")
        Console.ReadKey()
    End Sub
End Module
```

Abbildung 9.2: Die neue Eigenschaft ist jetzt zusammen mit den alten Elementen in *ZweiteKlasse* zu finden

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 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 Instanzierung 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 Instanzieren in Objekte benötigt.

---

Damit das erwartete Ergebnis eintritt, müssen Sie das Programm wie folgt abändern:

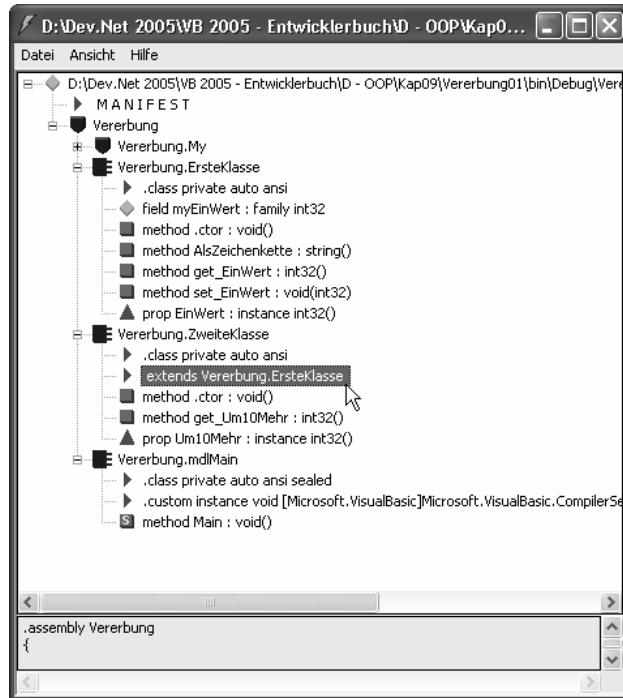
Module mdlMain

```
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 IML-Disassembler und schauen, ob uns Visual Basic wieder irgendetwas an Arbeit abgenommen hat.

- Starten Sie ILDASM, 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 9.3 zu sehen.



**Abbildung 9.3:** 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 Eigenschaftenamen 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 9.3 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 habe ich vorausschauend als Protected deklariert. Laut Tabelle des letzten Abschnittes 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.

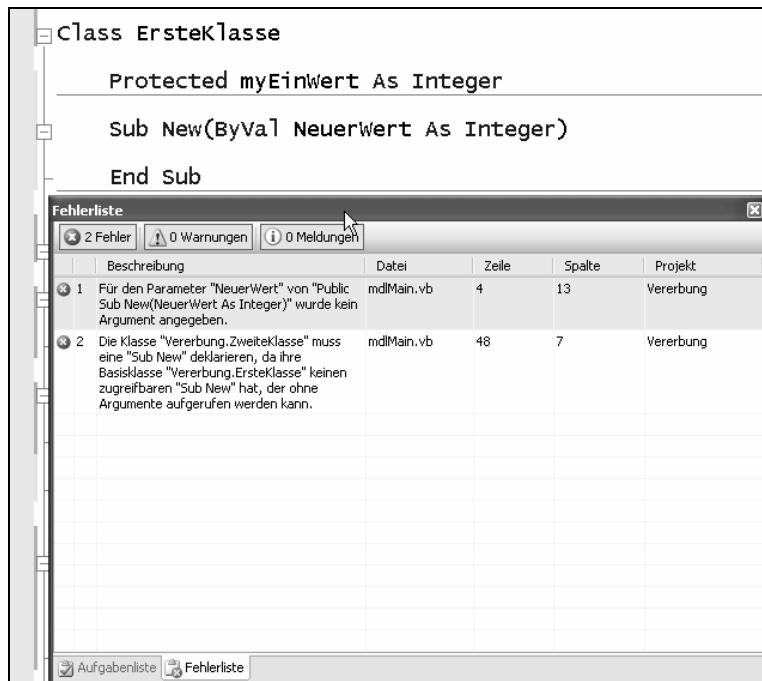
**TIPP:** Member-Variablen, die Sie in Klassen verwenden, welche später durch das Vererben wieder verwendet werden sollen, deklarieren Sie deshalb nach Möglichkeit als Protected, es sei denn, Sie wünschen ausdrücklich, dass nur die Basisklasse die Variable manipulieren darf.

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 RomanNumerals-Beispiel nämlich kennen gelernt. 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 9.4 zu sehen.



**Abbildung 9.4:** 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 »Hineinkomplizieren« eines Standardkonstruktors unter die Arme zu greifen. Kein Standardkonstruktor heißt: nehmen, was da ist. 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 instanzieren 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: Kein Parameter-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 Mem-

ber-Variablen in jedem Konstruktor einbauen (siehe Abbildung 9.4). Nur so ist gewährleistet, dass alle erforderlichen Variableninitialisierungen *in jedem Fall* durchgeführt werden.

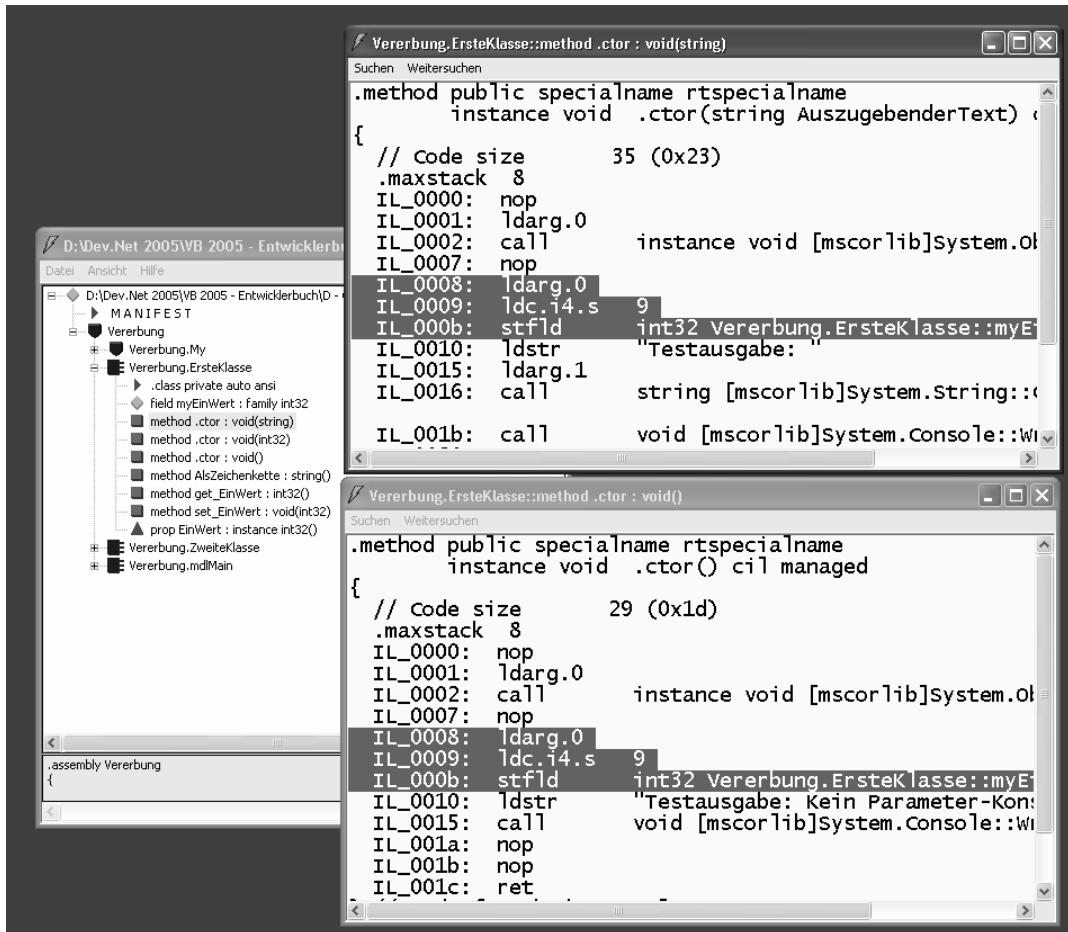


Abbildung 9.5: Initialisierungen von Member-Variablen werden im Bedarfsfall in jede Konstruktorprozedur eingebaut

## 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 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 rumröhren, 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:** Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\Chap09\Vererbung02`.

---

```
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 bestenfalls 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 9.6):

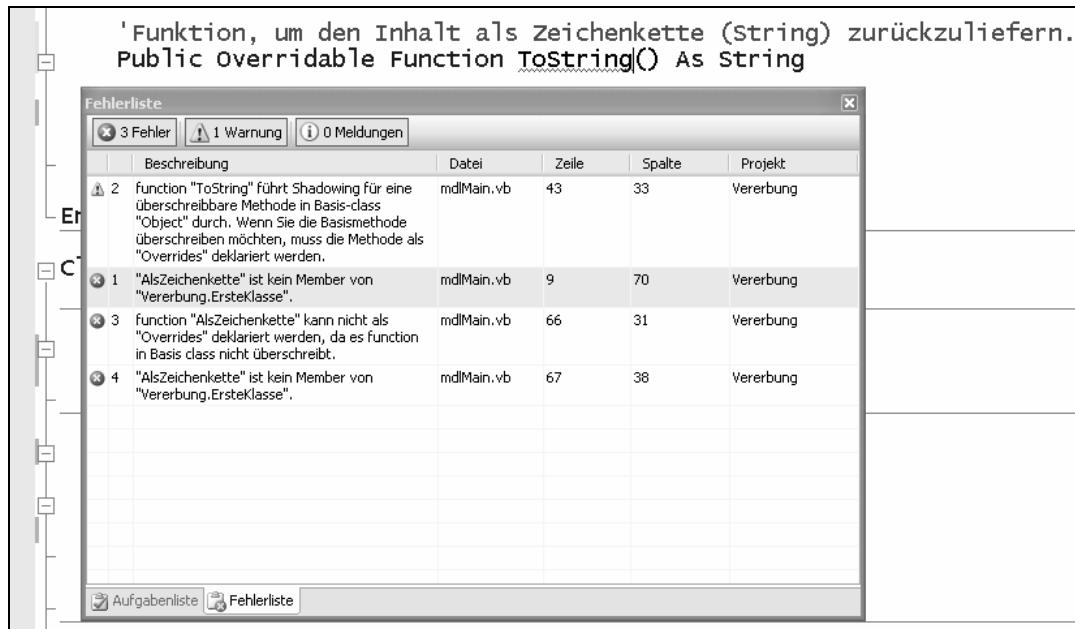
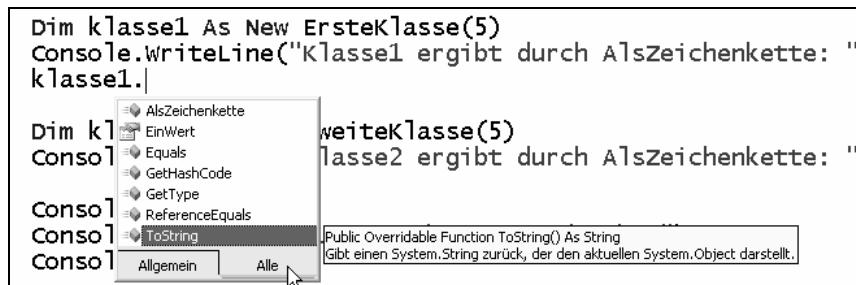


Abbildung 9.6: Versuchen Sie `ToString` zu implementieren, sehen Sie diese Fehlermeldung

Nur, welche Methode ist hier gemeint? IntelliSense listet die vorhandenen Member eines Objektes doch auf – wie in Abbildung 9.2 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 9.7 zu sehen.



**Abbildung 9.7:** 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:** Sie finden die so veränderte Version des Beispiels übrigens im Verzeichnis `.|VB 2005 - Entwicklerbuch|D - OOP\Kap09\Vererbung03`.

---

## Das Speichern von Objekten im Arbeitsspeicher – und die daraus resultierende Vorsicht mit ihnen

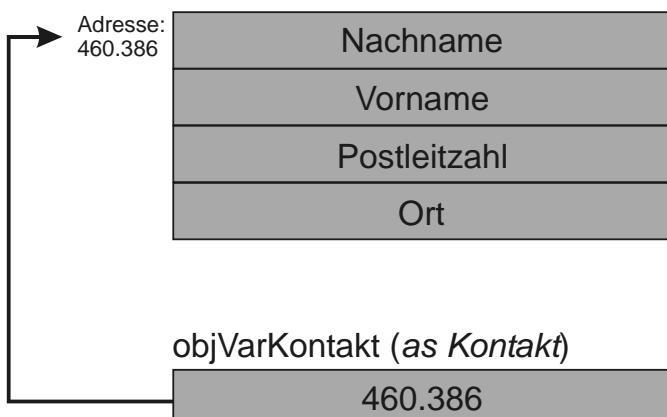
Bevor wir uns dem Allerwichtigsten dieses Kapitels nähern, wäre es zum besseren Verständnis überaus ratsam, ein paar Worte zur Speicherung von Objekten (und nahezu allen von ihnen abgeleiteten Klassen zu verlieren). Objektvariablen und Objekte sind nämlich nicht so miteinander verbunden, wie man es sich 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: Eine Objektvariable speichert im Grunde genommen nur einen Zeiger auf die eigentlichen Daten im so genannten Managed Heap – einem Speicherbereich, den das .NET-Framework verwaltet, und der sich im Arbeitsspeicher Ihres Computers befindet.

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.

Wenn Sie nun ein Objekt instanzieren – 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. Bildlich gesprochen sieht das folgendermaßen aus:

## Instanz aus Klasse: Kontakt



**Abbildung 9.8:** 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:** Sie finden den Quellcode dieses Beispiels im Verzeichnis .\VB 2005 - Entwicklerbuch\DidOOP\Kap09\Vererbung04.

---

Module mdlMain

```
Sub Main()

    'Instanzieren 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

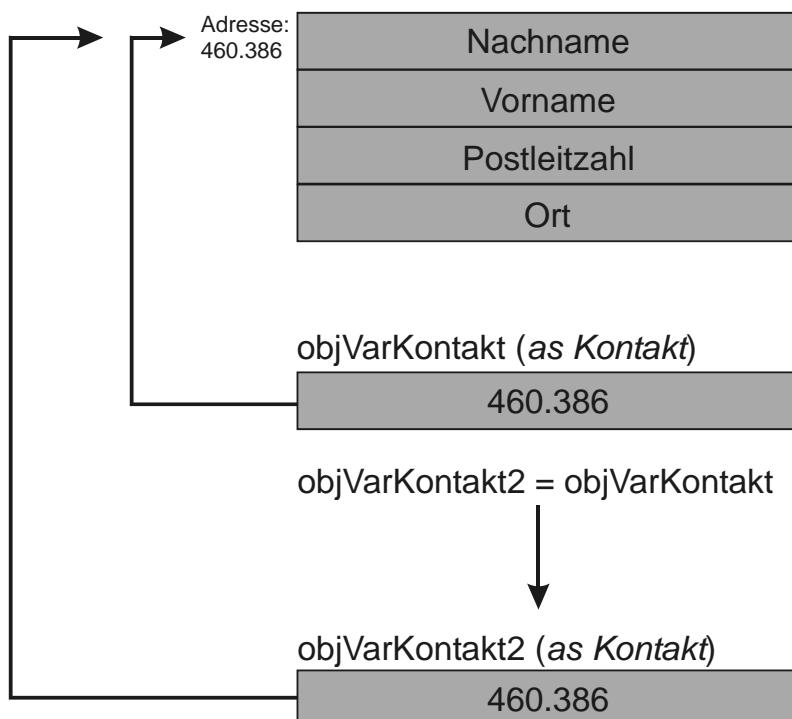
Löffelmann

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:

## Instanz aus Klasse: Kontakt



**Abbildung 9.9:** 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 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 Objektmengen auf Geschwindigkeit ankommt, und keine ganzen Speicherblöcke (mit den Daten der eigentlich Instanzen) sondern nur die Zeiger auf die Objektinstanzen selbst kopiert werden müssten.

---

**HINWEIS:** Das nächste Kapitel, ► Kapitel 10, hält weitere Informationen zu diesem Thema bereit.

---

# 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 »Polymorphie«<sup>2</sup> 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 instantiierten abgeleiteten Klasse erreichen zu können.« Das klingt sehr abstrakt, und das ist es auch.

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 aber 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 »Erledigung seines Anliegens«. Seine Methode, also die Wahl seine 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?

---

<sup>2</sup> Etwa »vielgestaltig«, »in verschiedenen Formen auftretend«.

Mitnichten! Diese Vorgehensweise ist nicht nur kein Fehler, Sie haben gerade sogar das wohl mächtigste Werkzeug der objektorientierten Programmierung kennen gelernt.

Ein etwas größeres Praxisbeispiel soll 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 Stewart
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, und das muss berücksichtigt werden. 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                                          | 18,60 Euro |  |
| 0002       | Kristall der Träume                                |            |  |
| 23,27 Euro | 1,63 Euro                                          | 23,27 Euro |  |
| 0003       | Volle Deckung - Mr. Bush: Dude where is my country |            |  |
| 12,06 Euro | 0,84 Euro                                          | 12,06 Euro |  |
| 0004       | Du bist nie allein                                 |            |  |
| 17,76 Euro | 1,24 Euro                                          | 17,76 Euro |  |
| 0005       | X-Men 2 - Special Edition                          |            |  |
| 19,82 Euro | 3,17 Euro                                          | 19,82 Euro |  |
| 0006       | Sex and the City: Season 5                         |            |  |
| 25,85 Euro | 4,14 Euro                                          | 25,85 Euro |  |

0007 Indiana Jones (Box Set 4 DVDs)  
38,78 Euro 6,21 Euro 38,78 Euro

0008 Die Akte  
12,92 Euro 2,07 Euro 12,92 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 18,60 Euro

0002 Kristall der Träume  
Autor: Barbara Wood  
23,27 Euro 1,63 Euro 23,27 Euro

0003 Volle Deckung - Mr. Bush: Dude where is my country  
Autor: Michael Moore  
12,06 Euro 0,84 Euro 12,06 Euro

0004 Du bist nie allein  
Autor: Nicholas Sparks und Ulrike Thiesmeyer  
17,76 Euro 1,24 Euro 17,76 Euro

0005 X-Men 2 - Special Edition  
Laufzeit: 128 Min.  
Hauptdarsteller: Patrik Steward  
19,82 Euro 3,17 Euro 19,82 Euro

0006 Sex and the City: Season 5  
Laufzeit: 220 Min.  
Hauptdarsteller: Sarah Jessica Parker  
25,85 Euro 4,14 Euro 25,85 Euro

0007 Indiana Jones (Box Set 4 DVDs)  
Laufzeit: 359 Min.  
Hauptdarsteller: Harrison Ford  
38,78 Euro 6,21 Euro 38,78 Euro

0008 Die Akte  
Laufzeit: 135 Min.  
Hauptdarsteller: Julia Roberts  
12,92 Euro 2,07 Euro 12,92 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 Artikelypen 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 Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap09\Inventory01* finden.

---

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 Overrides 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 Hintergrund, 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 vonstatten 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 * 1.16
    End Get

    Set(ByVal Value As Double)
        myNetPrice = Value / 1.16
    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 & 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

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 Tausendertrennzeichen ein Komma und das Nachstellentrennzeichen ein Punkt ist. Mehr zu diesem Thema finden Sie übrigens in ► Kapitel 17.

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 & vbCrLf & "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 18):

```
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 (überschriebenen) abgeleiteten Klasse aufgerufen – der formatierte Bruttopreis eines Buches berücksichtigt also korrekt 7 % Mehrwertsteuer und nicht die 16 % der Basisklasse. GrossPriceFormatted ruft also die überschriebene 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 Beschreibungstextes) 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 Polymorphie 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 & vbLf & "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 unterstützt 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:
Try
    locSr = New StreamReader(Filename, System.Text.Encoding.Default)
Catch ex As Exception
    Console.WriteLine("Fehler beim Lesen der Inventardatei!" & _
                      vbNewLine & 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!" & _
                          vbNewLine & ex.Message)
    End Try
End Do

```

```

        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 zur Verfügung zu stellen, 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 selber 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«. Siehe 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 &
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab &
            Me.GrossPriceFormatted & vbCrLf & vbCrLf
    Else
        'Langform: Die Description Eigenschaft des Objektes
        'selber wird verwendet.
        Return Me.Description & vbCrLf &
            Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab &
            Me.GrossPriceFormatted & vbCrLf & vbCrLf
    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 9.1:** 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 Buches 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 (oder die Signaturen für überladene Prozeduren) einer Prozedur 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.

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 instanzieren – 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 instanzieren, 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 instanzieren. 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 9.10 zu sehen:

The screenshot shows a portion of a Visual Studio code editor. The code is as follows:

```

Dim locShopItem As New ShopItem
Console.WriteLine("wählen Sie")
Dim locKey As Char = Console.
If locKey = "1"c Then
    locDisplayMode = PrintTyp
Else
    locDisplayMode = PrintType.Detailed
End If

```

A tooltip window is open at the top right, displaying the error message: "'New' darf nicht für eine Klasse verwendet werden, die als 'MustInherit' deklariert ist." (The keyword 'New' may not be used for a class that is declared as 'MustInherit'). Below the tooltip, there are options to 'Delete' or 'End Try'.

Abbildung 9.10: 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.

The screenshot shows a portion of a Visual Studio code editor. The code is as follows:

```

Public MustOverride Property GrossPrice() As Double
    Get
        Return myNetPrice * 1.16
    End Get
    Set(ByVal value As Double)
        myNetPrice = value / 1.16
    End Set
End Property

```

A tooltip window is open at the top right, displaying the warning: 'Die DVDItem-Klasse muss als MustInherit deklariert werden oder folgende geerbte MustOverride-Member überschreiben:'. Below the tooltip, there are options to 'Delete' or 'End Try'.

Below the code editor, a 'Fehlerliste' (Error List) window is open, showing one error:

| Beschreibung                                                                                                       | Datei      | Zeile | Spalte | Projekt   |
|--------------------------------------------------------------------------------------------------------------------|------------|-------|--------|-----------|
| Die DVDItem-Klasse muss als MustInherit deklariert werden oder folgende geerbte MustOverride-Member überschreiben: | mdlMain.vb | 261   | 7      | Inventory |

The code continues below the error:

```

    Return (GROSSPRICE - myNetPrice).ToString("#,##0.00")
End Get
End Property

```

Abbildung 9.11: 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 Fehlermeldung

Die *DVDItem*-Klasse muss als *MustInherit* deklariert werden oder folgende geerbte *MustOverride*-Member überschreiben:

```
Inventory.ShopItem : Public MustOverride Property GrossPrice() As Double.
```

an, denn es ist in der Tat richtig, dass die Klasse *DVDItem* diese Eigenschaft nicht implementiert. Stellen Sie sich vor, Videos und DVDs würden mit 15 % 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:** Das Programmbeispiel mit den Änderungen für abstrakte Klassen finden Sie übrigens im Verzeichnis *.\VB 2005 - Entwicklerbuch\Chap09\Inventory02*.

---

## 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.

Da Schnittstellen keinerlei Code enthalten, können sie, genau wie abstrakte Klassen, nicht direkt instanziert 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).
- 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.

Für den letzten Punkt gibt es dazu in der Regel 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 selber durchführen muss, sollte ihm die Komponente basierend auf den angebotenen Schnittstellen abstrakte Klassen zur Verfügung stellen, die eine gewisse Grundfunktionalität enthalten.

Ein Beispiel macht die Zusammenhänge klarer: 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.

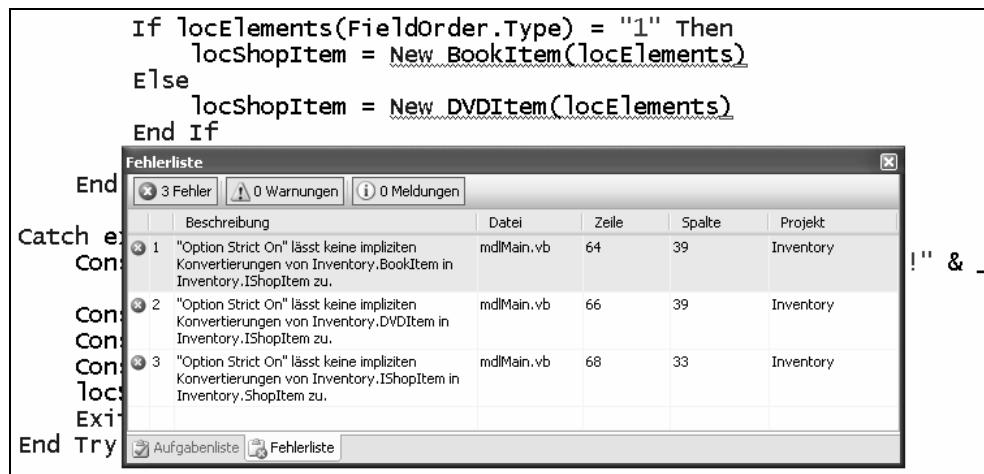
Das bislang verwendete Beispiel eignet sich ebenfalls, die Verwendung von Schnittstellen zu demonstrieren. Dazu wird eine Schnittstelle implementiert, die die Grundfunktionen vorschreibt, auf die das Hauptprogramm bislang über die Klasse *ShopItem* Zugriff hatte. Das Interface für das Beispielprogramm soll den Namen *IShopItem* bekommen (das Voranstellen des Buchstabens »I« 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 9.12):



**Abbildung 9.12:** 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 `IShopItem` (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 **Eingabe**, 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 9.13 zu sehen:



**Abbildung 9.13:** 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 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.<sup>3</sup> 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.

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 9.14):

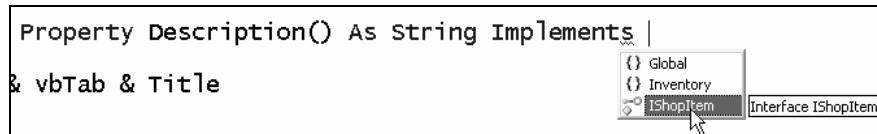


Abbildung 9.14: IntelliSense unterstützt Sie bei der Auswahl der richtigen Schnittstellenimplementierung

Um die Beispielanwendung wieder zum Laufen zu bewegen, 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 & vbCrLf & Me.VATAmountFormatted & vbCrLf & _
            Me.GrossPriceFormatted & vbCrLf & vbCrLf
```

---

<sup>3</sup> Von einer Ausnahme abgesehen, doch dazu später in ► Kapitel 19 noch mehr.

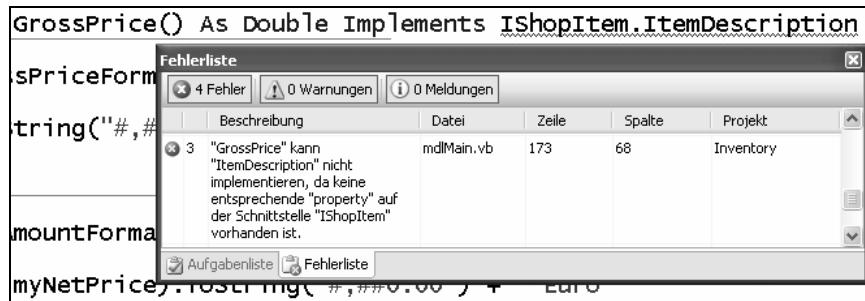
```

Else
    'Langform: Die Description Eigenschaft des Objektes
    'selber wird verwendet.
    Return Me.Description & vbCrLf &
        Me.NetPriceFormatted & vbTab & Me.VATAmountFormatted & vbTab & _
        Me.GrossPriceFormatted & vbCrLf & vbCrLf
End If
End Function

Public Overridable ReadOnly Property Description() As String Implements IShopItem.ItemDescription
Get
    Return OrderNumber & vbCrLf & 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 Klassemethode 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 9.15 zeigt.



**Abbildung 9.15:** 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 2005 - Entwicklerbuch\Chap09\Inventory03.

---

## 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 **Eingabe** bestätigen, sondern die Zeile mit einer der Cursor-Tasten verlassen.

Hätten Sie nämlich die Zeile mit Eingabe 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 **Eingabe**.

```

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 9.16:** Nachdem Sie mit Implements eine Schnittstelle eingebunden haben, fügt der Editor die kompletten Coderümpfe der benötigten Elementen 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 **Eingabe**, wird der Elementcoderumpf wieder an das Ende der Klasse angehangen. 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 **Eingabe** 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 Einfügemarkie unter die Zeile Public Class BindetITestEin hinter Implements ITest bewegen und **Eingabe** betätigen, erscheint am Ende der Klasse ein Funktionsrumpf, wie Sie ihn auch in Abbildung 9.17 erkennen können.

```
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
    End Function

    Public Function ZweiteMethode1() As String Implements ITest.ZweiteMethode
    End Function
End Class
```

**Abbildung 9.17:** 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 311).

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 9.14) zu verheiraten.

Mit `Implements` würde das auch nicht so ohne weiteres funktionieren, denn es gibt ja schließlich überhaupt keinen Methodecode, 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 Einfügemarkie wieder unterhalb der Zeile `Public Class BindetITest` hinter `Implements ITest`, und betätigen Sie **Eingabe**.

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).

```

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 Function ToString1() As String Implements ITest.ToString
            End Function
    End Class

```

**Abbildung 9.18:** 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 Eingabe 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 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. Das 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 **Eingabe** 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 mehrere Schnittstellen in eine Klasse

Mehrfachvererbung ist in Visual Basic 2005 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 2005 mehr als eine Schnittstelle in einer Klasse implementieren, und auch wenn das nicht so »toll«<sup>4</sup> wie Mehrfachvererbung ist, so können Sie sich mit der

---

<sup>4</sup> 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

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
```

Am Code der einbindenden Klasse muss in diesem Fall nur die in Fettschrift gesetzte Zeile geändert werden. An dem restlichen Code, der die Schnittstellenelemente 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 100 prozentig verlassen können.

---

dazu: »[...] Mehrfachvererbung bleibt verständlich, wenn nur eine Basisklasse wirklich Member-Funktionen implementiert und die anderen nur pure virtuelle Funktionen deklariert. [...]«

## 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 2005 - Entwicklerbuch\Design\OOP\Kap09\ToStringDemo`.

---

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
        1stkontakte.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
    End Property

```

```

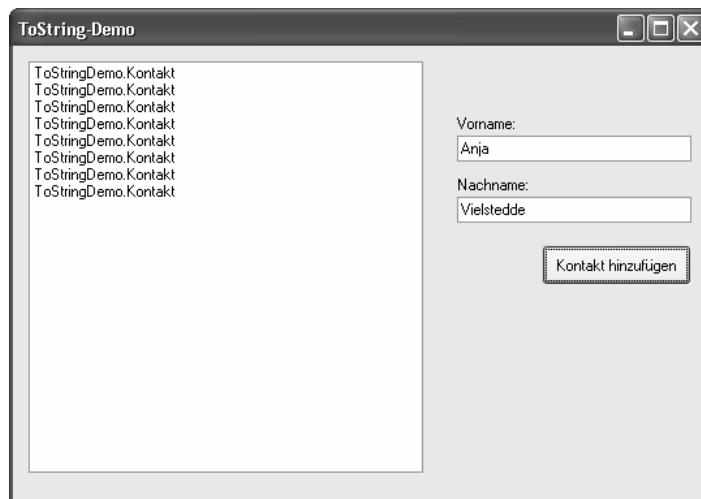
Set(ByVal value As String)
    myNachname = value
End Set
End Property
End Class

```

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 279). 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 9.19:** Das Hinzufügen eines Objektes zur *ListBox* führt zunächst nur zur Anzeige des voll qualifizierten Objekt-namens

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,<sup>5</sup> 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` wiederum die Basisklassen 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:

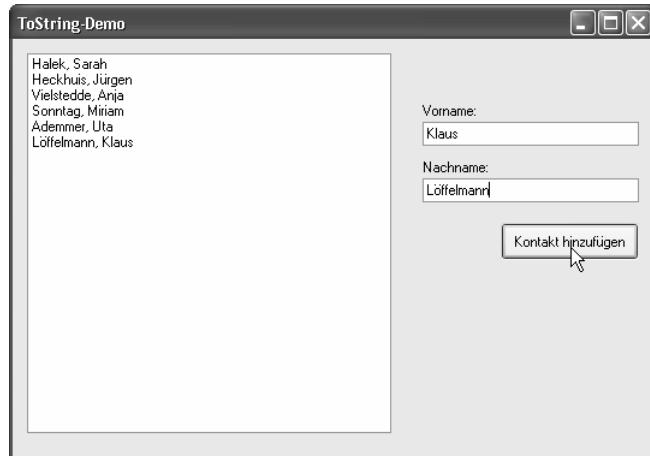
```
Public Overridable Function ToString() As String
    Return Me.GetType.ToString()
End Function
```

`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.

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, wie auch die folgende Abbildung zeigt:



**Abbildung 9.20:** Durch das Überschreiben von `ToString()` in der Klasse, deren Objektinstanz der Liste hinzugefügt wird, lässt sich der dargestellte Listentext steuern

<sup>5</sup> 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 Steuerlement handelt, das an eine andere Eigenschaft gebunden werden kann. In diesem Fall würde `GetItemText` den Text des Objektes per `ToString` ermitteln, der an das Steuerlement 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 Steuerlements übergäben (wie beispielsweise eine Schaltfläche oder das Formular selbst).

# 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). 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. Es wird dabei NICHT überprüft, ob der Inhalt der Objekte der gleiche ist. Ein Beispiel soll das verdeutlichen, und zum besseren Verständnis sollten Sie zuvor nochmals einen Blick auf Abbildung 9.9 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:** Das folgende Beispiel finden Sie unter `.\VB 2005 - Entwicklerbuch\Chap09\EqualsDemo`.

---

Module `mdlMain`

```
Sub Main()

    'Instanzieren 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 Ausgabeergebnis dieses Programms 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 Objekte 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:** Das folgende Beispiel finden Sie unter [|VB 2005 - Entwicklerbuch|D - OOP\Kap09\EqualsPraxis](#).

---

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 lockKontakt As Object
    lockKontakt = lstKontakte.SelectedItem
    If lockKontakt IsNot Nothing Then
        lstKontakte.Items.Remove(lockKontakt)
    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 mit einbeziehen. 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 23.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| (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 12.                                                                                                                                                                                                                                                                                                                                     |
| 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 3.2: 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.

```
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

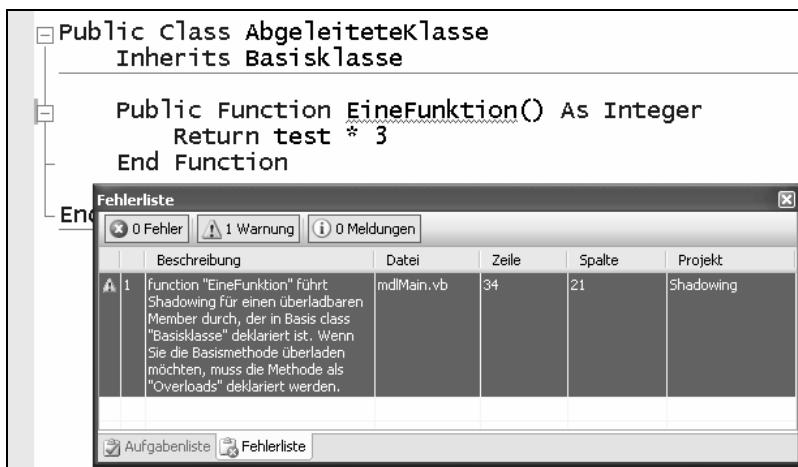
    Public Shadows Function EineFunktion() As Integer
        Return test * 3
    End Function

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 9.21 zu sehen:



**Abbildung 9.21:** In diesem Beispiel blendet eine Funktion der abgeleiteten Klasse die der Basisklasse aus

Visual Basic gibt Ihnen hier eine Fehlermeldung 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 Klasse 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:** Sie finden dieses Beispiel im Verzeichnis .\VB 2005 - Entwicklerbuch\Chap09\Shadowing02.

---

```
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)
    End Sub
End Module
```

```

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 kennen gelernt 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 ändern, wird klar, was eigentlich passiert.

---

**BEGLEITDATEIEN:** Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\Chap09\Shadowing03`.

---

```

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
    End Sub
End Module

```

```

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 instanzierbar. 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 vergangenen Kapiteln bereits erlebt haben.

## Singleton-Klassen und Klassen, die sich selbst instanzieren

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 Instanzierung 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 (wegen mir 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 kennen gelernt haben, keine »eingebauten« Klassentypen der FCL. Sie müssen sie selber entwickeln – doch das ist einfacher, als Sie denken.

---

**BEGLEITDATEIEN:** Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\Ch09\Kap09\Singleton`.

---

```

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" instanziert 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
    End Function
End Class

```

```

'Instanzieren 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 Instanzierungsvorgang 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 mehrere Programme) gleichzeitig ausgeführt werden können. Damit ein weiterer Instanzierungsversuch einer Singleton-Klasse nicht ausgerechnet dann passiert, wenn ein anderer Thread fast (aber eben noch nicht ganz) mit der Instanzierung fertig ist – beide Threads sich sozusagen »mittendrin« treffen würden und es damit doch die zu verhindernenden zwei Instanzen gäbe –, nutzt die Klasse eine Instanz von `Mutex`<sup>6</sup>, 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 (in ► Kapitel 31).

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.

---

<sup>6</sup> Mutex, abgeleitet von »mutual exclusion« etwa » gegenseitiger Ausschluss«.

# 10 Über Structure und den Unterschied zwischen Referenz- und Wertetypen

---

- 327 Der Unterschied zwischen Referenz- und Wertetyp
  - 329 Erstellen von Wertetypen mit Structure am praktischen Beispiel
  - 334 Unterschiedliche Verhaltensweisen von Werte- und Referenztypen
  - 341 Performance-Unterschiede zwischen Werte- und Referenztypen
- 

Jetzt, wo Sie sich mit der Erstellung von Klassen schon recht intensiv auseinander gesetzt haben, wird es Zeit, sich mit einer Variation von Klassen auseinander zu setzen, die ebenfalls ein sehr zentraler Bestandteil des Frameworks ist: die so genannten Wertetypen. Zum genauen Verständnis werden wir einen etwas tieferen Einblick in die Verwaltung von Daten im Framework nehmen.

Lassen Sie uns aber zunächst an der Basis beginnen, mit den Fragen: Was ist ein Referenztyp, was ist ein Wertetyp, und wie unterscheiden sich die beiden voneinander?

## Der Unterschied zwischen Referenz- und Wertetyp

Das Framework ist so konzipiert, dass es zweierlei Speicherbereiche gibt, in denen Daten für eine Anwendung gespeichert werden:

- Zum einen gibt es, wie Sie im vergangenen Kapitel schon kennen gelernt haben, den so genannten *Managed Heap* (wörtlich: *verwalteter Haufen*), auf dem die Daten von Referenztypinstanzen (also von Klassen) gespeichert werden.
- Zum anderen gibt es den (Prozessor-)Stack (etwa: *Prozessorstapel*), auf dem Wertetypen gespeichert werden.

Der Unterschied zwischen den beiden wird vor allen Dingen in ihrem Verhalten beim Kopieren oder Übergeben an Prozeduren deutlich, teilweise auch in der Geschwindigkeit: Denn während der Prozessor für das Abrufen von Daten auf den Arbeitsspeicher Ihres Computers zugreifen muss, befindet sich der Prozessorstack, wie der Name schon sagt, im Prozessor, bzw. der Zugriff erfolgt prozessorintern – zumindest nicht über den First bzw. Second Level Cache hinaus.

Was bedeutet es nun eigentlich genau, wenn Sie eine Objektvariable verwenden? Bei der Beantwortung dieser Frage möchte ich zunächst die primitiven Datentypen ausklammern, die eine Sonderbehandlung in der CLI aus Performance-Gründen genießen. Ich komme darauf später noch einmal zurück.

Wenn Sie eine Klasse instanzieren, dann referenzieren Sie sie für ihre Lebensdauer über eine Objektvariable. Die Objektvariable dient natürlich nur als Abstraktion für Entwickler und stellt das Repräsentativ für einen kleinen Speicherbereich dar, in den der »Wert« einer Variable hineingeschrieben wird. Und genau hier unterscheiden sich Wertetypen und Referenztypen entscheidend, denn: Während es für Wertetypen jeweils den Speicherbereich auf dem Prozessorstack gibt, werden streng genommen zwei Speicherbereiche für Referenztypen reserviert: Die eigentlichen Daten landen auf dem Managed Heap, und zusätzlich sichert die CLI die korrespondierende Adresse (bei einer 32-Bit-Windows-Version) in weiteren vier Bytes auf dem Stack.

Diese Tatsache wiederum hat mehrere Auswirkungen:

- Ein Referenztyp kann *null* (»nall«, englisch ausgesprochen) sein. Anders als der Wert 0 bedeutet *null*, dass kein Speicherbereich auf dem Managed Heap für das Objekt vorhanden ist. Die Objektvariable eines bestimmten Typs ist in diesem Fall zwar deklariert, es gibt allerdings keine konkreten Instanzdaten, auf die sie zeigt. Ein Wertetyp kann niemals *null* sein (»0« aber schon – jedenfalls bei numerischen Variablen), er ist grundsätzlich definiert, denn: Es ist überhaupt kein Zustand denkbar, bei dem eine Wertevervariablen ohne Daten auf dem Stack sein könnte (denn wenn es keine Daten auf dem Stack gibt, dann gibt es auch keine korrespondierende Variable dazu). In Visual Basic wird der Wert *null* übrigens durch das Schlüsselwort *Nothing* repräsentiert.
- Wenn Sie einen Wertetyp an einen anderen Wertetyp zuweisen, werden die Daten des entsprechenden Stackbereichs in den Stackbereich des anderen Wertetyps kopiert. Das Resultat: Wertetyp Nr. 2 hält ab sofort die gleichen Werte wie Wertetyp Nr. 1. Auch bei Referenztypen kopieren Sie die korrespondierenden Daten auf dem Stack, allerdings hat das ganz andere Auswirkungen: Referenztyp Nr. 2 zeigt anschließend auf Referenztyp Nr. 1, was bedeutet: Sie greifen mit beiden Objektvariablen auf dieselben Daten zu. Das »Gleiche« und das »Selbe« sind – auch wenn es wie schon im letzten Kapitel bemerkt, laut aktuellem Duden keinen Unterschied zwischen den beiden Wörtern mehr gibt – in diesem Fall in ihrer ursprünglichen Bedeutung zu nehmen. Ändern Sie nämlich Wertetyp Nr. 2, zeigt sich Wertetyp Nr. 1 davon unbeeindruckt. Anders ist das bei den Referenztypen: Ändern Sie eine Eigenschaft von Referenztyp Nr. 2, dann ändert sich auch die Eigenschaft von Referenztyp Nr. 1 – und das muss auch so sein: Denn es gibt keine zwei verschiedenen Eigenschaften; da beide Objektvariablen auf den gleichen Speicherbereich im Managed Heap zeigen, gibt es auch nur ein Objekt, dessen Eigenschaften Sie verändern können. Die Variablen bieten Ihnen nur zwei Möglichkeiten, es anzusprechen.
- Wenn Sie einen Wertetyp an eine Methode, also an eine Sub oder Function übergeben, dann legt die CLI – solange nichts anderes gesagt wird – eine Kopie des Wertetyps an. Ändern Sie die Variable innerhalb der Prozedur, so hat das keine Auswirkungen auf die Variable, die in dem Programmteil verwendet wurde, von dem aus die Prozedur angesprungen wurde. Bei einem Referenztyp verhält es sich im Gegensatz dazu ähnlich wie bei der Variablenzuweisung: Nur die Adresse des Speicherbereichs im Managed Heap wird der Prozedur übergeben; Änderung der Objekteigenschaften in der Prozedur wirken sich sehr wohl auf das Ursprungsobjekt aus (das ja dasselbe ist).
- Durch die andere Speicherverwaltung von Strukturen im Gegensatz zu Klassen sind Strukturen nicht vererbbar. Sie haben also nicht die Möglichkeit, die Elemente einer Struktur in einer anderen Struktur mit *Inherits* »zu übernehmen«.

All diese Dinge klingen in der Theorie vielleicht recht abstrakt. Deswegen möchte ich Ihnen im Folgenden ein komplettes Beispiel präsentieren, anhand dessen diese Zusammenhänge deutlich werden:

# Erstellen von Wertetypen mit Structure am praktischen Beispiel

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 Szenario: 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 zurückverwandeln. 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:** Sie finden das folgende Beispielprojekt im Verzeichnis *\VB 2005 - Entwicklerbuch\Diskettenkopie\OOP\Kap10\Structure01*.

---

So könnte Ihnen beispielsweise sogar das 32-Zahlensystem<sup>1</sup> von Nutzen sein: Sie meinen, Sie hätten das nie benutzt? Falsch: 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

---

<sup>1</sup> 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 *D1001* 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 »sexadezimal« heißen. Da die Kurzform für Hexadezimal kurz »Hex« lautet, können wir über den griechisch-lateinischen Mischmasch aber eher dankbar sein.

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 entsprechenden Numerale des gewünschten Zahlensystems 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
    End Sub
```

```

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 Ergebnisseichenkette (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 initiieren, 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
            'Aktuellen 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 Ziffernummer
            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 Ziffernummer 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
    Next
End Function
```

```

    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

$$\text{NumeraleWert} = \text{Ziffernwert} \times \text{Zahlensystembasiswert}^{\text{Ziffernposition}-1}$$

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 Ziffernposition. 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 Exponent<sup>2</sup> 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 Projektxplorer 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)

```

---

<sup>2</sup> Sie erinnern sich an die Schulzeit? – Jeder Wert potenziert mit 0 ergibt 1. Die Zahl wird dabei durch sich selbst geteilt.

```
    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:

Gegenbeispiel:  
'7U23085PJGBD9' duotrigesimal entspricht dezimal: 9153672076852735401

Return drücken zum Beenden

# **Unterschiedliche Verhaltensweisen von Werte- und Referenztypen**

Nun haben Sie Ihre erste Struktur entwickelt, aber inwiefern unterscheidet die sich nun von einer Klasse? Nun, wie in der Einführung 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 (instantiiert 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 einzige 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

```

## Konstruktoren und Standardinstanzierungen 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 Instanzierung 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 10.1).

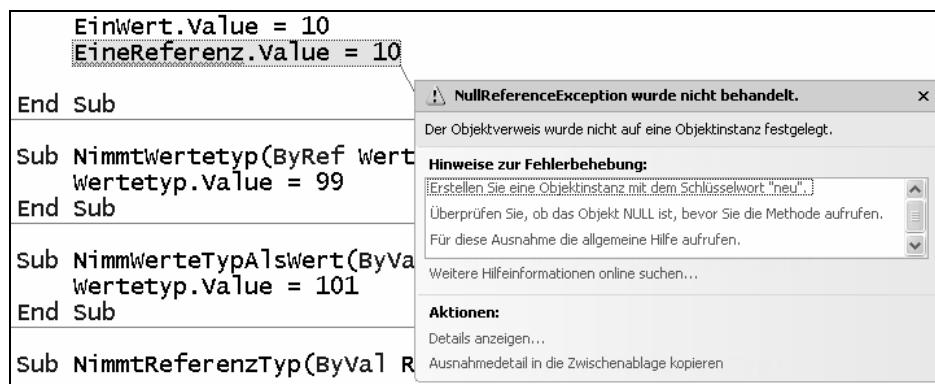


Abbildung 10.1: Versuchen Sie die Eigenschaft eines Objektes zu ändern, das nicht instanziert wurde, erhalten Sie eine Ausnahme

---

**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
    End Property

```

```

End Get
Set(ByVal Value As String)
    myTestEigenschaft = Value
End Set
End Property

End Structure

```

Strukturen dürfen in VB.NET keine Standardkonstruktoren haben, weil das Framework aus Performancegründen nicht garantieren kann, dass die Konstruktoren auch angewendet werden. Die Instanzierung 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 Instanzierung, 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 Booleans, etc.). Das geschieht aber nicht notwendigerweise über den – oder vielmehr im – Standardkonstruktor, sondern wird – wie auch immer – von der BCL übernommen. Das bedeutet, dass der Standardkonstruktor bei einer impliziten Verwendung u.U. nicht 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-Attributten

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 niedrigerwertigen.

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:** Sie finden das Projekt für dieses Beispiel im Verzeichnis *.\VB 2005 - Entwicklerbuch\Design - OOP\Kap10\StructLayout*.

---

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 angebaren Parameter, an welcher Speicherstelle im Stack 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
```

```

        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. An die Daten der Wertetypen kommen Sie in der Regel schneller heran, weil ein zusätzlicher Dereferenzierungsschritt nicht erforderlich ist. Erinnern wir uns:

- Wenn Sie mit den Daten eines Wertetyps arbeiten müssen, befinden sich die Daten auf dem Prozessorstack, 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 Genaues zu diesem Thema.

---

## **Wieso kann durch Vererbung aus einem Object-Referenztyp ein Wertetyp werden?**

Berechtigte Frage. Sie haben seit Beginn des Buches diese Tatsache schon fast als eine Art Dogma kennen gelernt, dass jedes im Framework verwendete Objekt, sei es ein primitiver Datentyp, eine Formular-Komponente, ein Datenbankobjekt, ein Thread etc. von `Object` abgeleitet ist. Dass sich Wertetypen anders verhalten, liegt an der Implementierung der Klasse `ValueType` in der CLR von .NET. Diese wird zwar ebenfalls 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.

Da `ValueType` eine abstrakte Klasse ist, können Sie sie nicht instanzieren. Sie dient also quasi nur als »Vorlage« für Wertetypen, die Sie aber nicht in eine andere Klasse ableiten, sondern eben mit `Structure` kreieren. Dafür, dass sie innerhalb der CLR einer anderen Speicherverwaltung unterliegt, 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.

---

# 11 Typumwandlungen (Type Casting) und Boxing von Datentypen

---

- 
- 344 Konvertieren von primitiven Typen
  - 345 Konvertieren von und in Zeichenketten (Strings)
  - 348 Casten von Referenztypen mit DirectCast
  - 349 Boxing von Wertetypen und primitiven Typen
  - 352 Boxen beim Implementieren von Schnittstellen in Strukturen
- 

Typen können in vielen Fällen in andere Typen umgewandelt werden; diesen Vorgang, einen Typen in einen anderen umzuwandeln, nennt man neudeutsch auch »Type Casting«<sup>1</sup>, einfach nur »Casting« oder, eingedeutscht, »Casten«. Dabei werden drei Verfahren unterschieden:

- 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*) oder »Unboxing« (»Auspakens«). Dabei wird ein Wertetyp in einen Referenztyp umgewandelt, sodass dieser anschließend auch durch eine Objektvariable einer Klasse der Klassenerbfolge referenziert werden kann.

---

**BEGLEITDATEIEN:** Sie finden alle Codeschnipsel der folgenden Beispiele in einem Projekt zusammengefasst, und zwar unter |VB 2005 - Entwicklerbuch|D - OOP|Kap11|Casting.

---

---

<sup>1</sup> Von engl. »to cast«, »auswerfen«, »abgießen« (aus einer Form). Aber auch »eine Rolle besetzen«.

# 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-Variablen 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 vom 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
```

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 Cxx 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 Cxx 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)
```

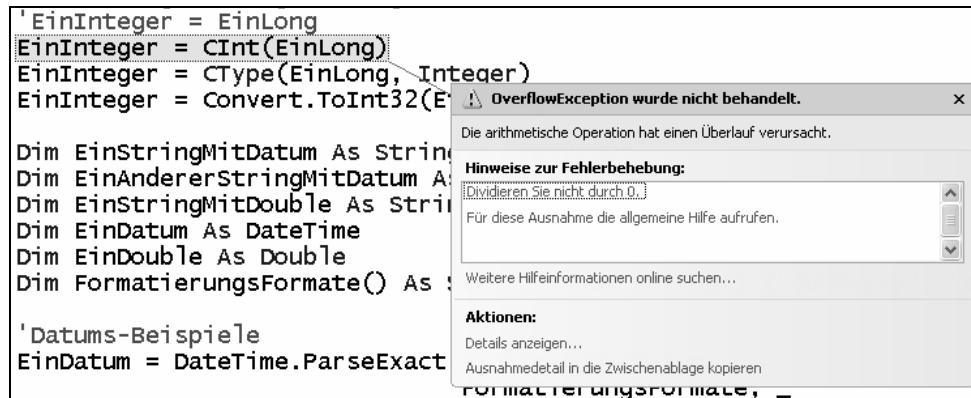
Rein theoretisch gäbe es noch folgende Möglichkeit, die Konvertierung durchzuführen: Sie entscheiden sich für Option Strict Off. In diesem Fall kümmert sich Visual Basic zur Laufzeit um die Konvertierung in den richtigen Datentyp. Da Ihnen der Codeeditor in diesem Fall aber nicht einmal eine

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 abändern, erhalten Sie eine Ausnahme (siehe Abbildung 11.1). Das liegt daran, dass Sie versuchen, einen Wert zu konvertieren, der sich schlicht und ergreifend nicht konvertieren lässt. Der zu konvertierende Wert wird mit

```
Dim EinLong As Long = Integer.MaxValue + 1L
```

auf 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 11.1:** 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. Sie 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`-Struktur<sup>2</sup>, verfügt über die statische Methode `Parse`<sup>3</sup>, 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 Datumswert 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 Zahlenumwandlung, 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 Datumswerten 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 17 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!

```
Dim EinStringMitDatum As String = "24122003"  
Dim EinAndererStringMitDatum As String = "2412"  
Dim EinStringMitDouble As String = "1.123,23 "
```

---

<sup>2</sup> Diese entspricht dem `Date`-Datentyp von Visual Basic übrigens exakt – es ist also völlig egal ob Sie `Date.Parse` oder `DateTime.Parse` schreiben.

<sup>3</sup> Vom englischen »to parse«, etwa: »analysieren«.

```

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 numerische Datentypen gibt. TryParse liefert nicht den eigentlich geprästen 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

'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 implizit nicht 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 um eine Typkonvertierung im Sinne der Anpassung von Klassenerbfolgen handelt.

```

## 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«<sup>4</sup> oder – eingedeutscht – »Boxen« bezeichnet wird.

Dazu ein kleines Beispiel: Nehmen wir an, Sie haben eine Struktur entwickelt und ihr den Namen Matrjoschka 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 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

```

---

<sup>4</sup> 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 einen zu fällenden Baum, um dessen Fallrichtung zu bestimmen und den Stamm vor dem Splittern zu bewahren...

```

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

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 instanzieren, 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 instanziert 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 locMaxValue As Integer
    End Sub

```

```

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.

## 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.

## Boxen beim Implementieren von Schnittstellen in Strukturen

Beim Boxen und dem anschließenden Entboxen von Wertetypen kann es mitunter zu Verhaltensweisen kommen, die auf den ersten Blick nicht wirklich nachzuvollziehen sind.

---

**BEGLEITDATEIEN:** Beachten Sie dazu das folgende Beispiel, das Sie im Verzeichnis *.\VB 2005 - Entwicklerbuch\Design\OOP\Kap11\Casting* finden.

---

```
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
    End Property

```

```

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 aufrufenden Programmteil wider.

Interessant wird es, wenn die Verwendung einer Schnittstellenvariablen ins Spiel kommt, die einen Wertetyp aufnimmt. Hier wird der Wertetyp nämlich von vornherein in einen Referenztyp umgewandelt; seine Daten landen in Kopie 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 wider – 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.

# 12 Beerdigen von Objekten – Dispose, Finalize und der Garbage Collector

---

357 **Der Garbage Collector – die Müllabfuhr in .NET**

359 **Finalize**

364 **Dispose**

---

Für das nächste Thema 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 was anderes – was allerdings mehr eine Fähigkeit des .NET-Framework selbst ist: den nicht benötigten Speicher nämlich 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
```

```

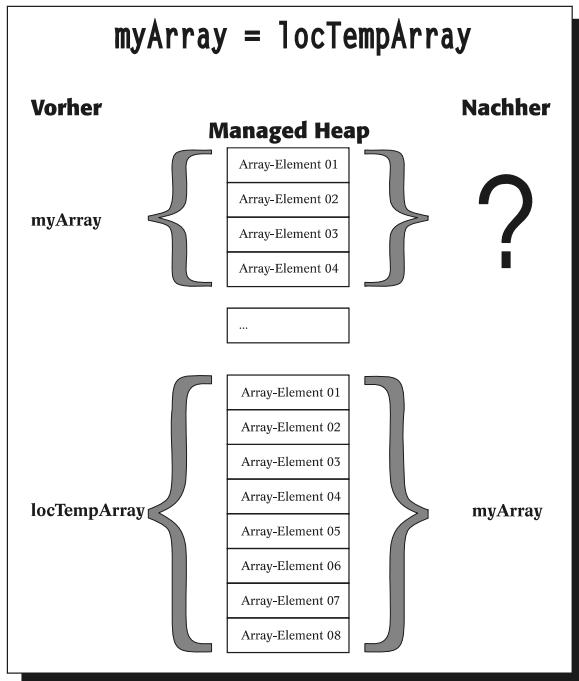
'Element im Array speichern.
myArray(myCurrentCounter) = Item

'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 stehen 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 12.1:** 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 12.1 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 Elemente, auf die von `myArray` verwiesen wurde, liegen nun unbrauchbar, da nicht mehr referenziert, irgendwo im Speicher – was passiert jetzt mit ihnen?

Erinnern wir uns, wie das bei COM 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 – nur musste der Entwickler sich um die Entsorgung des Objektes selber kümmern.

Dieses Verfahren hatte allerdings zwei Nachteile: Zum einen kostete des Prinzip des Referenzzählers wertvolle 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,<sup>1</sup> die innerhalb der AppDomain laufen, abgelegt werden. Dieser Speicherbereich wird Managed Heap genannt, mit dem wir uns schon einige Male beschäftigt haben. 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.

Der Garbage Collector läuft in einem eigenen Thread,<sup>2</sup> der die Objekte des Managed Heap dahingehend untersucht, ob sie noch in irgendeiner Form referenziert werden. Objekte, denen eine noch

---

<sup>1</sup> Streng genommen ist auch ein Programm eine Assembly.

<sup>2</sup> 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 **U**ser **I**nterface = Benutzeroberfläche). Mehr zu Threads finden Sie in ► Kapitel 31.

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.<sup>3</sup>

## 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.

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.

---

<sup>3</sup> 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.

- Der Garbage Collector ist in der AppDomain gezwungenermaßen durch die Anweisung `GC.Collect()` gestartet worden.

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 erledigt. 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 Handle<sup>4</sup> 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.

## 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.<sup>5</sup>

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 über-

---

<sup>4</sup> Eine vom Betriebssystem erteilte Kennung zur Nutzung einer bestimmten Ressource (Datei, Bildschirm, Schnittstellen, spezielle Windows-Betriebssystemobjekte, etc.).

<sup>5</sup> 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.

schreiben muss, wenn es eine eigene Funktionalität für seine »Entsorgungsvorbereitung« implementieren will.

Der Finalizer ist also im Prinzip der Destruktor von .NET Klassen.

---

**BEGLEITDATEIEN:** Beachten Sie dazu das folgende Beispiel, das Sie im Verzeichnis *.\VB 2005 - Entwicklerbuch\DOOP\Kap12\Finalize01* finden.

---

```
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
```

---

**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 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, die 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:** Das folgende Beispiel (im Verzeichnis unter .\VB 2005 - Entwicklerbuch\DE - OOP\Kap12\Finalize-NoNo01 zu finden) verdeutlicht, was Sie in echten Applikationen 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 anderes 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 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 Prozess so schnell wie möglich erledigt ist.
- Sorgen Sie dafür, dass Sie keine neuen Instanzen von irgendwelchen Objekten erstellen.

Wenn Sie diese Punkte beherzigen, tragen Sie enorm 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-Listener<sup>6</sup>.

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.

---

<sup>6</sup> 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.

# 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.

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 Collector. 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.

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 trägt<sup>7</sup>) oder durch den Garbage-Collector über `Finalize` erfolgt ist. Ihr `Dispose` muss ebenfalls dafür sorgen, dass erkannt wird, ob ein Objekt 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. Mit Hilfe dieser Klasse können Sie beliebige Objekte im SOAP-Format in einer Datei speichern (mehr zum Serialisieren von Objekten erfahren Sie im ► Kapitel 22). Umgekehrt kann die Klasse aus einer Datei die ursprünglichen Objekte wieder automatisch herstellen.

---

**BEGLEITDATEIEN:** Sie finden die Projektdateien für dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\Diskettenkopie\Kap12\Dispose`.

---

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
```

-----

---

<sup>7</sup> 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 (in dem Sie `e.Cancel` auf `True` setzen). »Schließen« Sie das Formular jedoch mit `Dispose`, was auch geht, dann »schließen« Sie es sozusagen ab. Es ist sofort zu, weg, verschwunden, entsorgt, und es löst auch keine Ereignisse mehr aus.

Eingabe der 1. Person

Nachname: Heckhuis

Vorname: Jürgen

-----  
Eingabe der 2. Person

Nachname: Thiemann

Vorname: Uwe

-----  
Eingabe der 3. Person

Nachname: Ademmer

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:«, und Sie sieht folgendermaßen aus:

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xs="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>
    <xsd:int id="ref-1">
      <m_value>4</m_value>
    </xsd:int>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xs="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:Dataset id="ref-1">
      <ns1:a1="http://schemas.microsoft.com/clr/nsassem/Dispose/Dispose%20Version%3D1.0.1431.30420%20Culture%3Dneutral%20PublicKeyToken%3Dnull">
        <myFirstName id="ref-3">Jürgen</myFirstName>
        <myLastName id="ref-4">Heckhuis</myLastName>
      </a1:Dataset>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xs="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:Dataset id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Dispose/Dispose%2C%20Version%3D1.0.1431.30420%2C%20Cultur
e%3Dneutral%2C%20PublicKeyToken%3Dnull">
<myFirstName id="ref-3">Uwe</myFirstName>
<myLastName id="ref-4">Thiemann</myLastName>
</a1:Dataset>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<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:Dataset id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Dispose/Dispose%2C%20Version%3D1.0.1431.30420%2C%20Cultur
e%3Dneutral%2C%20PublicKeyToken%3Dnull">
<myFirstName id="ref-3">Ute</myFirstName>
<myLastName id="ref-4">Ademmer</myLastName>
</a1:Dataset>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<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:Dataset id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Dispose/Dispose%2C%20Version%3D1.0.1431.30420%2C%20Cultur
e%3Dneutral%2C%20PublicKeyToken%3Dnull">
<myFirstName id="ref-3">Klaus</myFirstName>
<myLastName id="ref-4">Löffelmann</myLastName>
</a1:Dataset>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Man möchte sagen: Zwar ein ziemlicher SOAP-Overhead für diese paar Daten, aber was soll's: Es ist nur ein Demo und 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 Finali-

ze. 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 18).

```

Public Class SoapSerializer
    Implements IDisposable

```

Mit der *Implements*-Anweisung bindet die Klasse das *IDisposable*-Pattern ein. Zur Wiederholung: Bei einem Schnittstellen-Pattern wird eine Schnittstelle in erster Linie zur Standardisierung verwendet. Erst in zweiter Linie dient sie 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 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 (im ► Kapitel 22 erfahren Sie mehr darüber).

```

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 Schreiben 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 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
            'Übergründig 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)
        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 fettener 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 verstecken. 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 handelte – 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 Ausnahme 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 Implement `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 Overridable 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.



# 13 Operatoren für benutzerdefinierte Typen

---

- 378 **Einführung in Operatorenprozeduren**
  - 379 **Vorbereitung einer Struktur oder Klasse für Operatorenprozeduren**
  - 383 **Implementierung von Rechenoperatoren**
  - 385 **Implementierung von Vergleichsoperatoren**
  - 386 **Implementierung von Typkonvertierungsoperatoren mit Operator CType**
  - 387 **Implementieren von Wahr- und Falsch-Auswertungsoperatoren**
  - 389 **Problembehandlungen bei Operatorenprozeduren**
  - 391 **Übersicht der implementierbaren Operatoren**
- 

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 morgendlich 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 Trockenrühreipulver (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-/Rühreipulvergemische anschließend verdrückt.

Operatorenprozeduren, die neu sind in Visual Basic 2005, könnte man unter dem Begriff Convenience Tools (»Bequemlichkeitswerkzeuge«)<sup>1</sup> 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.

---

<sup>1</sup> Und es gibt tatsächlich den Begriff der Convenience-Patterns in der IT, gerade beim Überladen von Methoden.

# 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. 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.
```

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

```
"Verschiede|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. 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 ausschauen:

---

**BEGLEITDATEIEN:** Sie finden dieses Beispiel im Verzeichnis .\VB 2005 - Entwicklerbuch\DE - OOP\Kap13\Operatorenüberladung.

---

```
Public Structure SuperString
```

```
    Private myValue As String
```

```
    Public Sub New(ByVal Value As String)
        myValue = Value
    End Sub
```

```

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(andererString.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

```

```

    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 locSString As SuperString In locSuperStrings
    Console.WriteLine(locSString.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.

Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.

Wenn man seinen Kopf gegen eine Wand schlägt, verbraucht man 150 Kalorien.

Wenn 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 erfunden.

Der

Glückskeks

wurde

1916

von

George

Jung,

einem

amerikanischen

Nudelmacher

erfunden.

Taste drücken zum Beenden!

# Implementierung von Rechenoperatoren

Nachdem diese Vorbereitungen abgeschlossen sind, schreiten wir zur nächsten Tat: Dem eigentlichen Implementieren der statischen Operatorenprozeduren.

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 von 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 13.1 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 387.

---

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 8 im Abschnitt »Überladen von Funktionen und Konstruktoren« 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 »rauskaufen«. 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 übergebenden 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 ExperimenteFürBoolscheAusdrücke()
    '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
        Case "Y"
            Return True
        Case "YES"
            Return True
        Case "TRUE"
            Return True
        Case "CORRECT"
            Return True
        Case "SELECTED"
            Return True
        Case "PRESSED"
            Return True
    End Select
```

```

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 wäre. 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.

## 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 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.                                                                                                                  |

| Operator | Vorgesehene Funktion                                      | Bemerkung                                                                                                                                                                                                                                                                                                                                          |
|----------|-----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <=       | 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.<br><b>WICHTIG:</b> 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 387.                                                                                                                                                                              |

**Tabelle 13.1:** Operatoren in Visual Basic 2005, die für die Implementierung in eigenen Typen zur Verfügung stehen

# 14 Generische Klassen und Strukturen (Generics)

---

- 
- 393 Verwenden einer Codebasis für verschiedene Typen
  - 394 Lösungsansätze
  - 397 Typengeneralisierung durch den Einsatz generischer Datentypen
  - 400 Beschränkungen (Constraints)
  - 410 Vererben von generischen Typen
- 

## Verwenden einer Codebasis für verschiedene Typen

Wenn Sie bestimmte Klassen oder Strukturen, also wie auch immer geartete Typen, 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:** Ein Beispiel, das Sie im Verzeichnis `\VB 2005 - Entwicklerbuch\DE - OOP\Kap14\Generics01` finden, soll das verdeutlichen.

---

Dieses Beispiel verwendet die aus ► Kapitel 9 bereits 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)
```

```

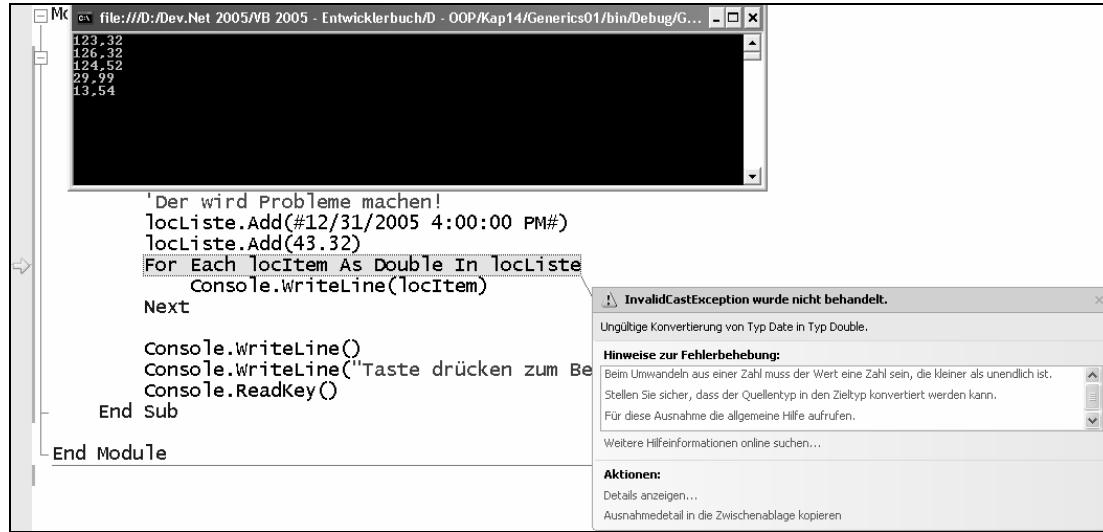
'Der wird Probleme machen!
locListe.Add(#12/31/2005 4:00:00 PM#)
locListe.Add(43.32)
For Each locItem As Double In locListe
    Console.WriteLine(locItem)
Next

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 14.1:** 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 *DynamicList*-

Double (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, 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

            '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 Overrides ReadOnly Property Count() As Integer
        Get
            Return myCurrentCounter
        End Get
    End Property
```

```

    End Get
End Property

'Erlaubt das Zuweisen
Default Public Overridable Property Item(ByVal Index As Integer) As Double
    Get
        Return myArray(Index)
    End Get

    Set(ByVal Value As Double)
        myArray(Index) = Value
    End Set
End Property

Public Function GetEnumerator() As System.Collections.IEnumerator _
    Implements System.Collections.IEnumerable.GetEnumerator
    'HINWEIS: GetEnumerator muss implementiert werden, wenn die
    'IEnumerable-Schnittstelle eingebunden wird. Die wiederum
    'ist notwendig, um For/Each zu ermöglichen.
    'Ein Array ist Enumerable - deswegen klauen wir uns
    'dessen GetEnumerator-Funktionalität.
    'Wir müssen bloß ein lokales Array erstellen,
    'das genau so viele Elemente hat, wie sich derzeitig
    'in der DynamicList-Instanz befinden, damit die noch
    'nicht genutzten Array-Elemente auch noch aufgezählt werden.
    'Aus diesem Grund verwenden wir eine lokale Kopie,
    'die exakt so groß ist, wie wir Anzahl Elemente
    'in dieser Liste haben.
    Dim locTempArray(myCurrentArraySize) As Double
    Array.Copy(myArray, locTempArray, myArray.Length)
    Return myArray.GetEnumerator
End Function
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 14.2 sehen können.

```

LocListe.Add(13.54)
'Der wird Probleme machen!
LocListe.Add(#12/31/2005 4:00:00 PM#)
LocListe.Add(43.32)
For Each locItem As Double In locListe
    Console.WriteLine(locItem)
Next

Console.WriteLine()
Console.WriteLine("Taste drücken zu")
Console.ReadKey()
End Sub

```

**Abbildung 14.2:** Bei typsichereren 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 aufwändiger. 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 Basis-Klasse 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 googeln 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 Typenpassen 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 ersetzt<sup>1</sup>.

---

**BEGLEITDATEIEN:** Das folgende Beispiel finden Sie im Verzeichnis *.\VB 2005 - Entwicklerbuch\Chap14\Generics02*.

---

In der Praxis und für unser DynamicList-Beispiel sieht das wie folgt aus:

Anstelle sich von vorneherein 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, 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)
```

```
    myArray(myCurrentCounter) = Item  
    myCurrentCounter += 1  
    If myCurrentCounter = myCurrentArraySize - 1 Then  
        myCurrentArraySize += myStep  
  
        'temporäres Array erstellen
```

---

<sup>1</sup> 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. Wenn Sie sich für die interne Funktionsweise von Generics interessieren – der IntelliLink D1401 hält Interessantes zum Thema bereit!

```

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

```

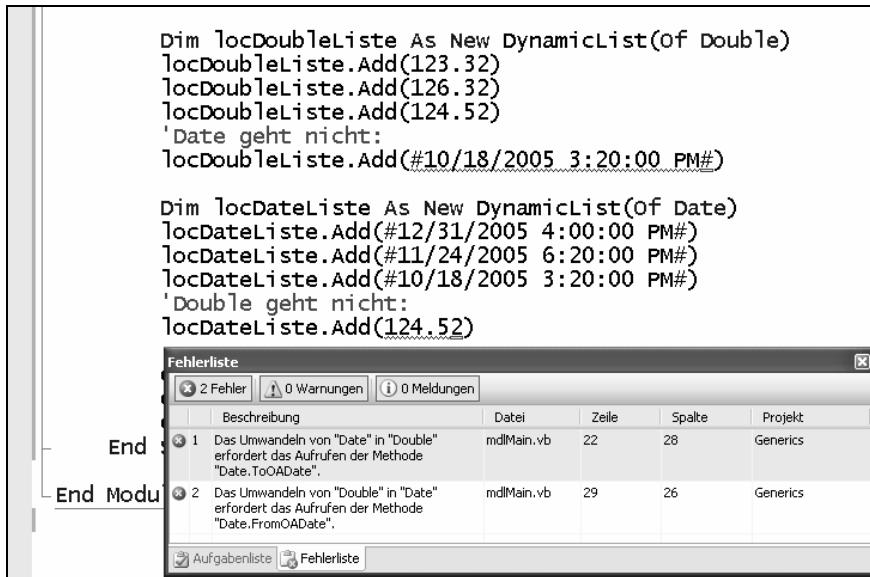
Und nun können Sie DynamicList für jeden Datentyp verwenden, den Sie möchten, und Sie müssen dabei nicht auf Typsicherheit verzichten, wie Abbildung 14.3 zeigt.

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)



**Abbildung 14.3:** 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 von 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.

### 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 speichern, sondern Sie möchten, dass diese Klasse auch deren Gesamtvolume 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

    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:

```

Module md1Main
    Sub Main()
        Dim lKörperliste As New DynamicList(Of KörperBasis)
        With lKörperliste
            .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 Gesamtvolume aller Körper beträgt:" & _
                           lKörperliste.Gesamtvolume)
    End Sub
End Module

```

**Abbildung 14.4:** 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 14.4 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 `Gesamtvolume` erschaffen, die durch alle Elemente iteriert, die sie trägt, und deren `Volumen`-Eigenschaft abfragt. Doch genau eine solche Prozedur können wir nicht implementieren, wie die folgende Grafik zeigt:

```

Public ReadOnly Property GesamtVolumen() As Double
    Get
        Dim locVolumen As Double
        For Each locItem As flexiblerDatentyp In Me
            locVolumen += locItem.
        Next
        Return locVolumen
    End Get
End Property

```

**Abbildung 14.5:** 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 zur Verfügung gestellt werden. Das sind logischerweise genau die Eigenschaften und Methoden, die `Object` zur Verfügung stellt, denn nur diese erbt, wie wir ja schon wissen, automatisch jede neue Klasse implizit.

Und genau das ist 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 innehat. Diese Klasse nennen wir `DynamicListKörper`, und die sieht folgendermaßen aus:

```

Class DynamicListKörper(Of flexibleDatentyp As KörperBasis)
    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

```

```

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:

```

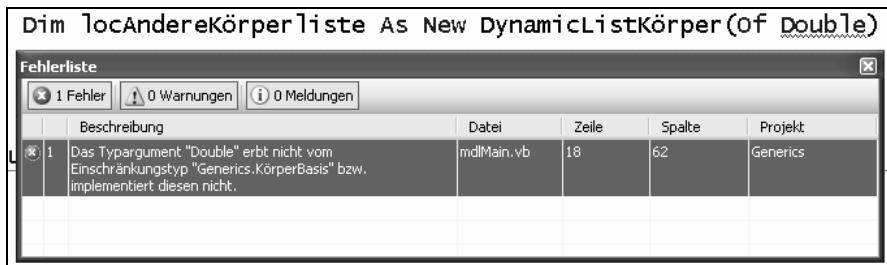
Module mdlMain

    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 14.6:** Eine beschränkte generische Klasse können Sie tatsächlich nur noch auf Basis eines Datentyps instanzieren, 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 Sie im Verzeichnis `.\VB 2005 - Entwicklerbuch\Ch - OOP\Kap14\Generics03` finden, soll das verdeutlichen.

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
            'War einen zu groß, also wieder teilen
            locDelta \= 3

            'Shellsort's Kernalgorithmus
            For locÄußererZähler = locDelta To myCurrentCounter - 1
                locItemTemp = Me.Item(locÄußererZähler)
                locInnererZähler = locÄußererZähler
                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 anderenfalls gar keine Instanzierung der DynamicListSortable möglich wäre), 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("Helige")
    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 Fettschrift 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
141
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 von ► Kapitel 19, 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 instanzieren. 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 GenerischeKlasseMitInstanzierbarenTyp(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 flexiblerDatentyp As Structure)

    Public Sub TestMethode()
        'Ist Wertetyp – keine Instanzierung durch New erforderlich!
        Dim locWerteTyp As flexiblerDatentyp

        '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 GenerischeBeschränkungskombi(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 Schüsseltyp As {Structure, IComparable}, _
    Werttyp As {New, IComparable, IDisposable})

    Public Sub TestMethode()
        Dim locWerteTyp As Schüsseltyp
        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

```

## Vererben von generischen Typen

Generische Datentypen lassen sich natürlich auch vererben. Dabei können Sie bestimmen, ob Sie den Typparameter auflösen, und die vererbte Klasse auf einen bestimmten Datentypen beschränken, oder auch die vererbte Klasse generisch anlegen und sie damit wieder flexibel für den Einsatz mit beliebigen Datentypen machen.

Eigentlich ist das, was wir im ► Abschnitt »Beschränkungen für generische Typen auf eine bestimmte Basisklasse« ab Seite 393 gemacht haben, furchtbar uneffizient. Wir haben den Quellcode mit *Kopieren/Einfügen* kopiert – und das widerspricht nun wirklich jedem Grundsatz der objektorientierten Programmierung.

Viel besser wäre es gewesen, die `DynamicList`-Klasse zu vererben, und die Ableitung dieser Klasse so zu modifizieren, dass sie den geforderten Ansprüchen gerecht geworden wäre. Das wäre nicht nur weniger Arbeit gewesen; wenn Sie im Nachhinein einen Fehler in der Basisklasse entdecken und diesen beheben, korrigieren sie ihn zwangsläufig auch in jeder Ableitung.

Möchten wir das für unsere `DynamicList` und die `KörperBasis`-Klasse machen, können wir uns prinzipiell entscheiden: Lösen wir für eine neue `DynamicList`-Klasse, die nur Objekte auf Basis von `KörperBasis` akzeptiert, die generische Eigenschaft auf, oder belassen wir sie generisch.

In diesem Fall macht es keinen Sinn, die Klasse weiterhin generisch zu »halten«, denn: Da die `DynamicList` mit der Beschränkung auf `KörperBasis`-Objekte sowieso keine anderen Objekte mehr akzeptiert, können wir in der Ableitung ihre generische Eigenschaft auch auflösen. Eine solche Implementierung würde folgendermaßen ausschauen:

```

Class Körperliste
    Inherits DynamicList(Of KörperBasis)

    Public ReadOnly Property Gesamtvolumen() As Double
        Get
            Dim locVolumen As Double
            For Each locItem As KörperBasis In Me
                locVolumen += locItem.Volumen
            Next
            Return locVolumen
        End Get
    End Property

```

```

    End Get
End Property
End Class

```

In diesem Fall lösen wir also die generische Eigenschaft der `DynamicList` auf und überführen die alte generische Klasse in eine neue nicht generische Klasse namens `KörperList`. Die Auflösung geschieht dadurch, dass der Typ, den die geerbte Klasse ab sofort verwalten soll, in der `Inherits`-Anweisung angegeben wird.

Die Alternative dazu, die, wie gesagt, für unser Beispiel nicht wirklich Sinn ergeben würde, und bei der ihre generische Eigenschaft erhalten bliebe, sähe folgendermaßen aus:

```

Class KörperlisteImmerNochGenerisch(Of flexiblerDatentyp As KörperBasis)
    Inherits DynamicList(Of flexiblerDatentyp)

    Public ReadOnly Property Gesamtvolumen() As Double
        Get
            Dim locVolumen As Double
            For Each locItem As KörperBasis In Me
                locVolumen += locItem.Volumen
            Next
            Return locVolumen
        End Get
    End Property
End Class

```

Diese Art der Vererbung macht nur dann Sinn, wenn die Beschränkung, eben nicht wie in diesem Beispiel, so restriktiv ist, dass sich die Klasse ohnehin nur noch auf einen Typen (und dessen Ableitungen) beschränkt – wenn die Ausgangsklasse also entweder mit *keiner* Beschränkung, mit *mehreren* Schnittstellen oder mit der `New`- oder `Structure`-Beschränkung arbeitet.



# 15 Ereignisse und Delegaten

---

- 
- 415 Konsumieren von Ereignissen mit WithEvents und Handles**
  - 418 Auslösen von Ereignissen**
  - 420 Zur-Freigabe-Stellen von Ereignisparametern**
  - 423 Dynamisches Anbinden von Ereignissen mit AddHandler**
  - 432 Delegaten**
- 

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 der 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, verdrachtet und damit nur im Bedarfsfall konsumiert werden. Damit werden auch Szenarien möglich, die noch in Visual Basic 6.0 ohne die Hilfe von externen DLLs nicht möglich gewesen wären.

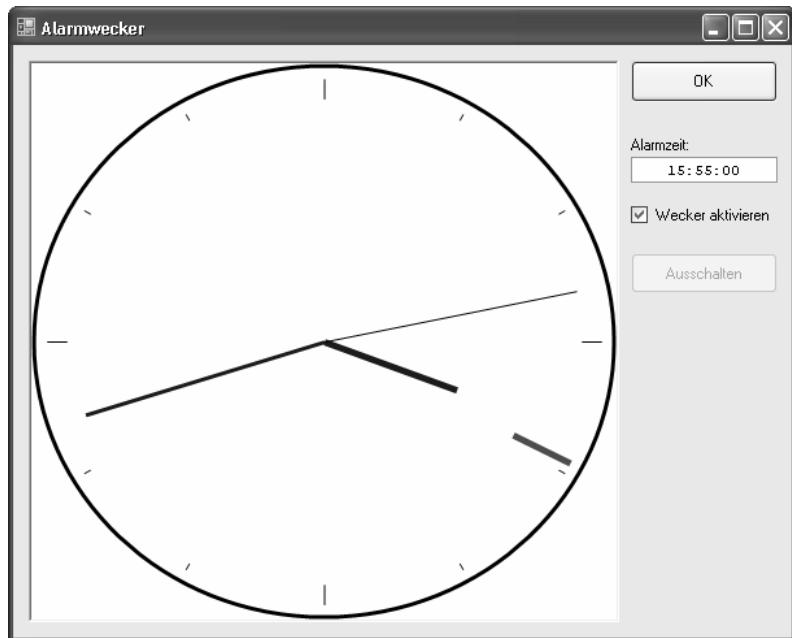
Interessant wird es schließlich mit Delegaten, die – vereinfacht ausgedrückt – so etwas wie Zeiger auf Funktionen bilden, wie man es früher nur von den C-Funktionszeigern kannte. Delegaten können sich dabei sogar polymorph verhalten, was eine enorme Flexibilität bei der Klassenprogrammierung ermöglicht.

---

**BEGLEITDATEIEN:** Der Beispielcode für die zunächst folgenden Abschnitt befindet sich in einem Projekt namens *Alarm.sln*, das Sie im Verzeichnis *\VB 2005 - Entwicklerbuch\Kap15\Alarm01\* finden.

---

Dieses Beispielprogramm, das in der ersten Version einen einfachen Wecker imitiert, werden wir im Folgenden für die Ereigniserkundigungen verwenden. Wenn Sie dieses Beispielprojekt starten, sehen Sie ein Formular, wie Sie es auch in Abbildung 15.1 erkennen können.



**Abbildung 15.1:** 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*<sup>1</sup> 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 29.

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:

---

<sup>1</sup> 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!).

- Durch das Zuweisen einer beliebigen Prozedur mit einer Member-Variable einer Klasse, die mit WithEvents deklariert wurde.
- Durch das manuelle Hinzufügen eines Ereignisbehandlers 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.

Und schließlich gibt es auch noch den Weg über so genannte Delegaten, über die im Abschnitt »Delegaten« ab Seite 432 die Rede sein wird.

---

**HINWEIS:** Wie Sie mithilfe von Editor- bzw. Designer Rümpfe von Ereignisbehandlungs routinen in Ihren Code einfügen, hat ► Kapitel 2 (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.

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 Signatur<sup>2</sup> 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.
```

---

<sup>2</sup> Zur Erinnerung: Die Signatur einer Prozedur ergibt sich aus der Abfolge der Typen, die eine Prozedur als Parameter entgegennimmt.

```

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
'aus das Hintergrundblitzen, 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
        If myAktuelleFarbe = Color.White Then
            myAktuelleFarbe = Color.Red
        Else
            myAktuelleFarbe = Color.White
        End If
        'Alarmdauerzähler 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:

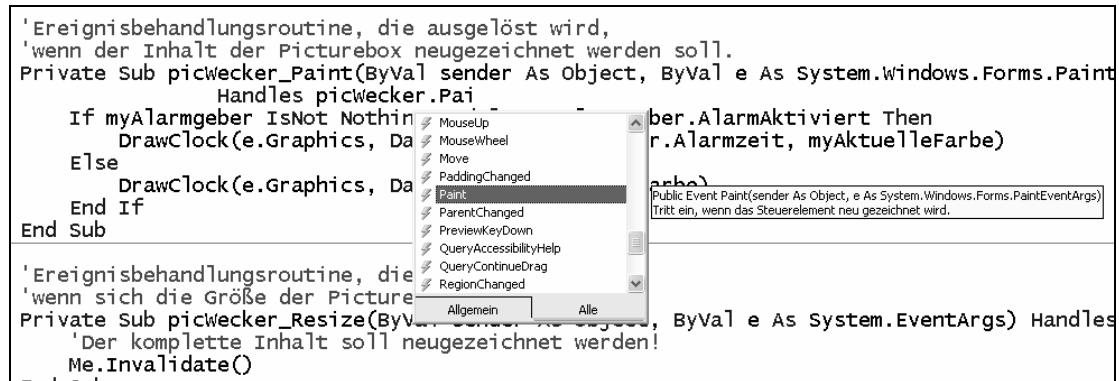
- myTimer,
- picWecker sowie
- myAlarmgeber

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 Handels erfolgen soll, wie Abbildung 15.2 zeigt.



**Abbildung 15.2:** 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 nicht auf Anhieb sehen. Der Designer, der ja auch den Formularcode generiert, nutzt nämlich die Möglichkeit von Visual Basic 2005, 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, der 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 415 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

    Friend 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 Overridable 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-Variablen 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 Weckereignis mitbekommen wollen, dann überschreiben Sie einfach die Methode OnWecken. Wenn die Ereignisbehandlungsroutine 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 Handels an bestimmte Prozeduren hängen, sondern einfach den Basiscode des Formulars überschreiben: OnResize löst nämlich seinerseits das Resize-Ereignis aus, das eintritt, wenn das Formular vergrößert oder verkleinert wird.

```

'Wird vom Basisklassenteile 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 Formulats 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.
- 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.

## 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 – denn Sie müssen ja wissen, wer für das Ereignis verantwortlich war und dementsprechend reagieren.

---

**BEGLEITDATEIEN:** Falls Sie sich den Code für dieses Beispiel anschauen möchten, finden Sie ihn im Verzeichnis .\VB 2005 - Entwicklerbuch\Ch - OOP\Kap15\Ereignistest in der Projektmappe Ereignisse.

---

Wenn Sie dieses Projekt starten, sehen Sie wie in Abbildung 15.3 ein Formular mit mehreren Schaltflächen.



**Abbildung 15.3:** 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 beschreibenden Text, sondern um eine wirkliche Referenz auf das Ereignis auslösende Objekt handelt, können Sie `sender` auch wieder zurück in seinen Ausgangstyp casten.

So ist es im Beispiel ganz sicher, dass ein Typ-Casting nicht schief gehen kann, da 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

```

## 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 grundsätzlich 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.

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 418 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.

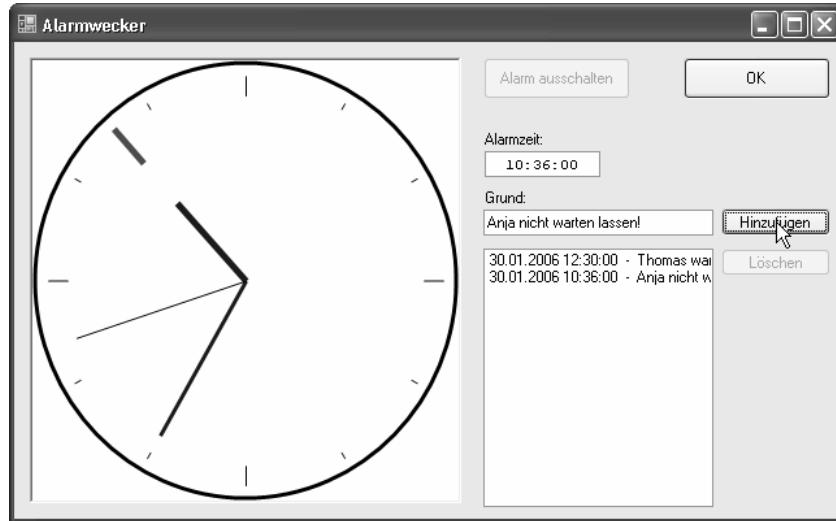
---

**BEGLEITDATEIEN:** Das folgende Beispiel, das Sie im Verzeichnis `.\VB 2005 - Entwicklerbuch\Chap15\Alarm02` finden, demonstriert ein solches Szenario.

---

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 Sie ihn auch in Abbildung 15.4 sehen können.



**Abbildung 15.4:** 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 15.5 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 15.5:** ... 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
''' </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 Termin.
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.<sup>3</sup> 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 19.

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 Fettschrift 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, die 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
```

---

<sup>3</sup> Wieso das so ist, können Sie übrigens im Blog von Krzysztof Cwalina nachlesen, den Sie unter dem IntelliLink *D1501* 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.

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. Dieser erlaubt es, 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. Mehr zum Thema Nullable erfahren Sie übrigens in ► Kapitel 20.

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

    'Alarmdauer 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 Time 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 InitializeComponents 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
    myLetzteAlarrrmeldung = e.AlarmText

    'Aus Liste löschen
    1stTermine.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 instanzieren, 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 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.

Eine andere Möglichkeit stellen so genannte Delegaten dar, von denen im folgenden Abschnitt die Rede ist.

## Delegaten

Im vorherigen Abschnitt haben Sie den Einsatz des AddressOf-Operators im Zusammenhang mit AddHandler und RemoveHandler kennen gelernt. Der AddressOf-Operator ermittelt sozusagen die Speicheradresse einer Prozedur, an der diese später nach dem Kompilieren bzw. nach der Verarbeitung durch den JITter im Arbeitsspeicher steht. Nur so wird die Verdrahtung eines Ereignisses mit einer Prozedur zur Laufzeit möglich.

Denn das, was AddressOf *Prozedurname* eigentlich ermittelt, ist etwas, was man als eine Delegaten-Konstante bezeichnen könnte. Ein Delegat speichert die Position einer Prozedur im Programm und erlaubt auch deren Zuweisung an eine Variable. Damit wird es nicht nur möglich, zur Laufzeit zu entscheiden, welche Prozeduren zu einem bestimmten Zeitpunkt aufgerufen werden sollen, man kann sogar dafür sorgen, dass Prozeduren aufgerufen werden, die Sie zur Entwurfszeit noch gar nicht kennen.

Zugegeben: Ohne Beispiel ist diese Erklärung reichlich abstrakt. Werfen Sie am besten einmal einen Blick auf den folgenden kleinen Formularcode, der in Abhängigkeit des Klickens auf eine von zwei vorhandenen Schaltflächen unterschiedliche Prozeduren innerhalb des Formular-Codes aufruft – was im Grunde genommen noch nichts Besonderes wäre, würde dieses Beispiel das Problem nicht mit der Hilfe von Delegaten lösen:

---

**BEGLEITDATEIEN:** Den Code für dieses Beispiel finden Sie im Projekt namens *Delegate.sln* im Verzeichnis *\VB 2005 - Entwicklerbuch\Chap15\Delegate*.

---

```
Public Class Form1

    Delegate Sub TestDelegate(ByVal Text As String)

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles Button1.Click, Button2.Click

        Dim locDel As TestDelegate

        If sender Is Button1 Then
            locDel = AddressOf BehandlungsroutineButton1
        Else
            locDel = AddressOf BehandlungsroutineButton2
        End If
        locDel.Invoke("Dies kam über einen Delegaten!")
    End Sub
End Class
```

```

End Sub

Private Sub BehandlungsroutineButton1(ByVal Text As String)
    MessageBox.Show("Behandlungsroutine für Button1 sagt: " & Text)
End Sub

Private Sub BehandlungsroutineButton2(ByVal Text As String)
    MessageBox.Show("Behandlungsroutine für Button2 sagt: " & Text)
End Sub
End Class

```

Interessant für dieses Beispiel sind zunächst nur die beiden ersten Schaltflächen. Wenn Sie das Programm starten, werden in Abhängigkeit von der ausgelösten Schaltfläche entweder die Methoden `BehandlungsroutineButton1` bzw. `BehandlungsroutineButton2` aufgerufen. Soweit ist das noch nichts Besonderes. Doch nun schauen Sie sich an, wie diese Behandlungsroutinen aufgerufen werden: Innerhalb der `Click`-Ereignisbehandlungsroutinen für beide Schaltflächen wird eine spezielle Variable deklariert (`locDel`), die in der Lage ist, quasi die Prozeduren an sich zu speichern, die aufgerufen werden sollen. Im Grunde genommen ist diese Erklärung aber Unsinn, denn Variablen können natürlich keine Prozeduren speichern. Was Variablen allerdings speichern können, sind die Startadressen der Prozeduren, an denen ihr Code später im Arbeitsspeicher beginnt. Diese spezielle Variable `locDel` ist im Beispiel vom Typ `TestDelegat`. Und wenn Sie nun an den Anfang des Klassencodes schauen, sehen Sie die Definition dieses Typen: `TestDelegat` wird einerseits als `Sub` (also als Methode ohne Rückgabewert) und andererseits mit einer bestimmten Signatur definiert (sie erwartet einen einzigen Parameter vom Typ `String`). Diese Signatur entspricht exakt den beiden Behandlungsroutinen, deren Adresse wir später in einer Variable von dem Typ speichern, den wir an dieser Stelle mit der `Delegate`-Anweisung definieren.

Möchten wir nun, nachdem die Delegatvariable `locDel` deklariert und ihr anschließend die Adresse einer signaturgleichen Prozedur zugewiesen wurde, diese zugewiesene Prozedur auch tatsächlich aufrufen, genügt dazu ein simpler `Invoke`-Methodenaufruf.

In diesem Beispiel, das lediglich den grundsätzlichen Umgang mit Delegaten verdeutlichen soll, ergibt diese Vorgehensweise natürlich noch nicht wirklich Sinn – die beiden Behandlungsroutinen hätten natürlich auch direkt aus dem Programm heraus ohne das Zur-Hilfe-Nehmen eines Delegaten aufgerufen werden können.

Doch Sie ahnen sicherlich bereits, was der Einsatz von Delegaten ermöglicht: Mit ihrer Hilfe haben Sie nämlich die Möglichkeit, dafür zu sorgen, dass Sie Methoden aufrufen, die Sie zur Entwurfszeit Ihrer Klasse noch gar nicht kennen. Und genau das eröffnet eine Alternative zu Ereignissen.

Sie könnten beispielsweise eine Eigenschaft in Ihrer Klasse zur Verfügung stellen, die vom Typ eines Delegaten ist. Eine Klasse, die Ihre Klasse einsetzt, könnte diese Eigenschaft dann mit einem Wert belegen, der eine Prozedur dieser Ihre Klasse einbindenden Klasse darstellt. Innerhalb Ihrer Klasse könnten Sie dann zur gegebenen Zeit dafür sorgen, dass – sofern die Delegaten-Eigenschaft definiert wurde (also nicht `Nothing` ist) – die Prozedur per `Invoke` aufgerufen wird. Welche Prozedur dabei genau dahinter steckt, wissen Sie zum Zeitpunkt, zu dem Sie Ihre Klasse entwickeln, noch gar nicht – genau so, wie es bei Ereignissen der Fall ist (Sie wissen bei der Implementierung eines Ereignisses ebenfalls nicht, »wer« dieses Ereignis Ihrer Klasse später konsumieren wird).

Mit dem Wissen können wir das schon bekannte Weckerbeispiel auf Delegaten umstellen. Sie sorgen in unserem Fall sogar dafür, dass das Programm ein wenig kompakter wird, denn: Die Notwendig-

keit, eine zweite Klasse zu entwerfen, die eine Liste mit Alarmgeber-Objekten enthält, deren Ereignisse dann wieder ein Ereignis auslösen und so »weiter nach oben delegiert« werden, fällt weg.

Stattdessen genügt es, die Alarmgeber-Klasse mit einem Delegaten auszustatten. Das Hauptprogramm kann dann die verschiedenen Alarmgeber-Instanzen direkt in der ListBox halten und braucht keine weitere Liste, die die Ereignisse nach oben hinauf reicht. Sie übergibt jeder Alarmgeber-Instanz stattdessen mit AddressOf die Prozedur, die im Fall des Erreichens der Weckzeit direkt aufgerufen werden soll. Und anstatt ein Ereignis auszulösen, überprüft die Alarmgeber-Instanz, ob der Delegat, den sie zur Verfügung stellt, definiert wurde, ruft im positiven Fall Invoke des Delegaten und damit direkt die Prozedur des Hauptprogramms auf, die dann schließlich dafür sorgt, dass der Alarm visualisiert wird.

---

**BEGLEITDATEIEN:** Das auf Delegatengebrauch umgestellte Beispiel finden Sie im Verzeichnis .\VB 2005 - Entwicklerbuch\Chap15\Alarm03, und es demonstriert ein solches Szenario.

---

Betrachten wir zunächst die relevanten Codeteile, mit denen die Alarmgeber-Klasse um den Einsatz mit Delegaten erweitert wird (nicht relevante Codeteile und Kommentare sind hier aus Platzgründen ausgelassen):

```
Public Class Alarmgeber
```

```
    Public Delegate Sub AlarmDelegate(ByVal sender As Object, ByVal e As AlarmEventArgs)

    Private WithEvents myTrigger As Timer
    Private myAlarmText As String
    Private myAlarmzeit As Date
    Private myAlarmAktiviert As Boolean
    Private mySchwellwert As Integer = 2
    Private myAlarmDelegate As AlarmDelegate
```

In dieser Klasse wird der Delegatentyp als Erstes definiert, damit einer einbindenden Klasse vorgeschrieben wird, wie die Signatur einer Prozedur auszusehen hat, die ein Alarmgeber-Objekt im »Alarmfall« mithilfe des Delegaten aufrufen soll. Anschließend definiert die Klasse den eigentlichen Delegaten als solchen Delegatentyp – dieser speichert später die eigentliche Rückrufadresse der Prozedur.

```
    ''' <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, ByVal AlarmText As String, ByVal AlarmRückrufroutine As AlarmDelegate)
        Me.Alarmzeit = Alarmzeit
        Me.AlarmText = AlarmText
        myAlarmDelegate = AlarmRückrufroutine
    End Sub
```

```

Sub New(ByVal Alarmzeit As Date, ByVal AlarmText As String, ByVal Aktiviert As Boolean, _
        ByVal AlarmRückrufroutine As AlarmDelegate)
    Me.Alarmzeit = Alarmzeit
    Me.AlarmAktiviert = Aktiviert
    Me.AlarmText = AlarmText
    myAlarmDelegate = AlarmRückrufroutine
End Sub
.
.
.
```

Eine Instanz der Alarmgeber-Klasse erhält die Rückrufprozedur ausschließlich über den Konstruktor. Unser Hauptprogramm muss also, wenn es eine Alarmgeber-Klasse instanziert, mit dem AddressOf-Operator festlegen, welche seiner Prozeduren im Alarmfall aufgerufen werden soll.

```

Protected Sub OnWecken(ByVal e As AlarmEventArgs)
    If myAlarmDelegate IsNot Nothing Then
        myAlarmDelegate.Invoke(Me, e)
    End If
    RaiseEvent Alarm(Me, e)
End Sub
.
.
.
```

End Class

Der Rest des Klassencodes bleibt, wie er ist – mit der Ausnahme der Methode OnWecken, die bislang lediglich das Alarm-Ereignis auslöste. Diese Methode kümmert sich in der neuen Version um eine weitere Sache: Sie ruft Invoke des Delegaten auf und damit die zuvor mit AddressOf übergebene Prozedur.

Das Anlegen einer Instanz der Alarmgeber-Klasse im Hauptprogramm wiederum passiert dann, wenn der Anwender einen neuen Termin erfasst. Der entsprechende Code des Hauptprogramms sieht mit den notwendigen Änderungen dann folgendermaßen aus:

```

'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 instanzieren, das wir anschließend
        'direkt zur Listbox hinzufügen. Hierbei wird auch gleichzeitig
        'die Rückruf-Prozedur angegeben, die angesprungen wird,
        'wenn der Alarm ausgelöst wurde.
        Dim locAlarm As New Alarmgeber(locAlarmzeit, txtGrund.Text, True, AddressOf myAlarmgeber_Alarm)
        lstTermine.Items.Add(locAlarm)
    End If
End Sub
.
```

```

'Den nächsten Termin ermitteln, damit er eingezeichnet wird
'(oder eben nicht, wenn es keinen mehr gibt!)
FindeNächstenTermin()

'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

```

Sie sehen hier zweierlei: zum einen, dass die zusätzliche Terminliste zur Speicherung der Alarmgeber-Instanzen (der Termine) nun nicht mehr benötigt wird. Alle Termine werden als Alarmgeber-Instanzen direkt im ListBox-Steuerelement `lstTermine` gespeichert. Zum anderen wird dem erweiterten Konstruktor nun mit `AddressOf` die Adresse der vormaligen Ereignisbehandlungsroutine des Alarm-Ereignisses übergeben. Diese Prozedur wird durch `Invoke` des Delegaten (siehe vorheriges Listing) indirekt aufgerufen, wenn das Alarmgeber-Objekt den Alarm auslöst.

Durch den Einsatz von Delegaten erreichen wir in diesem Fall einen wesentlich kompakteren Code: Die komplette Implementierung der `Collection(Of Alarmgeber)`-Liste `Terminliste` ist weggefallen und damit natürlich auch die Anforderung, zwei Listen homogen zu pflegen. Zusätzlich ist der Einsatz von Delegaten auch schneller – zwar nur für den Bruchteil von Millisekunden auf modernen Rechnern und damit für dieses Beispiel nicht sonderlich relevant – aber wenn Sie zeitkritische und häufige Ereignisaufrufe durchführen müssen, kann das schon ein ernst zu nehmender Entscheidungsfaktor für den Einsatz von Delegaten werden.

Ein weiteres Beispiel für Delegaten finden Sie übrigens in ► Kapitel 21, das einen Formelparser zum Berechnen beliebiger mathematischer Ausdrücke vorstellt, den Sie mithilfe von Delegaten um eigene Funktionen erweitern können. Und ► Kapitel 20 zeigt im Rahmen von generischen Auflistungen ebenfalls einige besondere Methoden, die sich Delegaten bedienen.

---

**HINWEIS:** In Delegaten gespeicherte Prozeduren lassen sich auf asynchron aufrufen, was bedeutet, dass sie im Sinne des Multitasking als neuer Thread initiiert werden. Da diese Vorgehensweise zum Thema Threading gehört, finden Sie es im entsprechenden ► Kapitel 31 beschrieben.

---