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[1] zusammenzufassen und von außen aufzurufen – einfach um Redundanz durch doppelte Funktionen zu vermeiden und letztendlich Platz zu sparen.

[1] *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`-Namespaces 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)[2], kurz vor der Ausführung in nativen Assembler-Code kompiliert werden (*JIT* steht für ***J**ust **i**n **T**ime*, 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, finanztechnische Programmbibliotheken in Visual Basic zu formulieren und diese von Ihrer eigentlichen Windows-Anwendung aufzurufen, die Sie in C# entwickelt haben.

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

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

Die BCL ist *Teil* der Common Language Runtime, und sie enthält alle Basisobjekte, die Sie während Ihres Entwicklungsalltags 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.

[2] 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. FreeBDS und – natürlich – Windows.[3]

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 Datentypenkonzepten 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

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

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.[4] 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 vorkompiliert, 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 vorkompiliert, 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.

[4] O.k., na gut, sagen wir: In den seltensten Fällen.

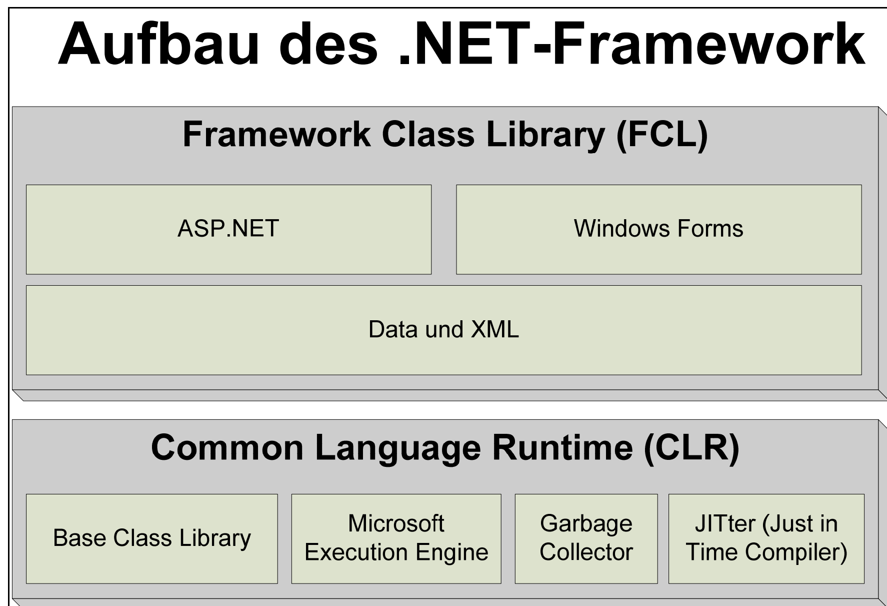

Abbildung 7.1: Diese Grafik stellt den Aufbau des .NET Framework bildlich dar

Erzwungene Typsicherheit und Deklarationszwang von Variablen

Visual Basic bietet die Möglichkeit, Programme zu entwickeln, die weder typsicher sind noch einen Variablendeklarationszwang 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\D - OOP\Kap07\OptionSamples*. Öffnen Sie dort die entsprechende Projektmappen-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
        locDateVaraible = 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:

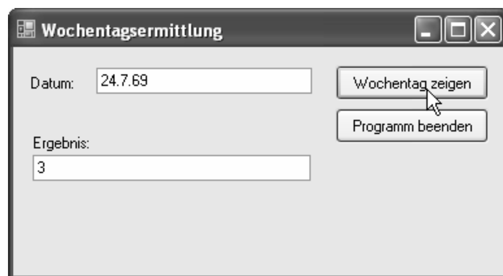

Abbildung 7.2: Dieses Programm »scheint« nur zu funktionieren – in Wahrheit verbergen sich in den paar Zeilen Code zwei dicke Fehler!

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

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

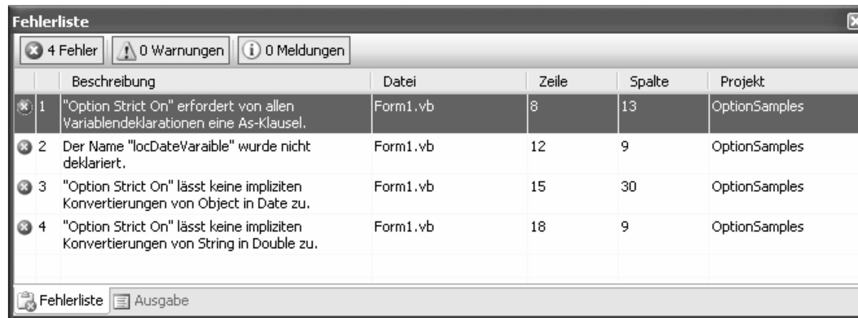

Abbildung 7.3: Bescheid wissen über Fehler heißt, Debugging zu vermeiden. Nach diesen Fehlern hätten Sie ohne *Option Explicit* und *Option Strict* wahrscheinlich eine ganze Weile gesucht!

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:

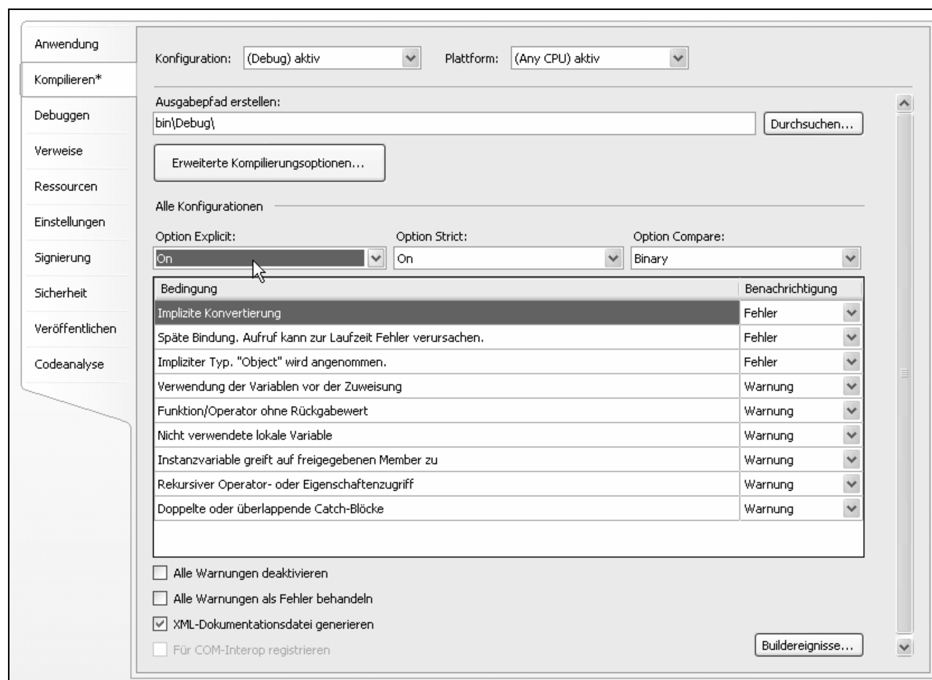

Abbildung 7.4: Auf dieser Registerkarte der Eigenschafteneinstellungen definieren Sie das *Option*-Verhalten global für das gesamte Projekt

- 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).

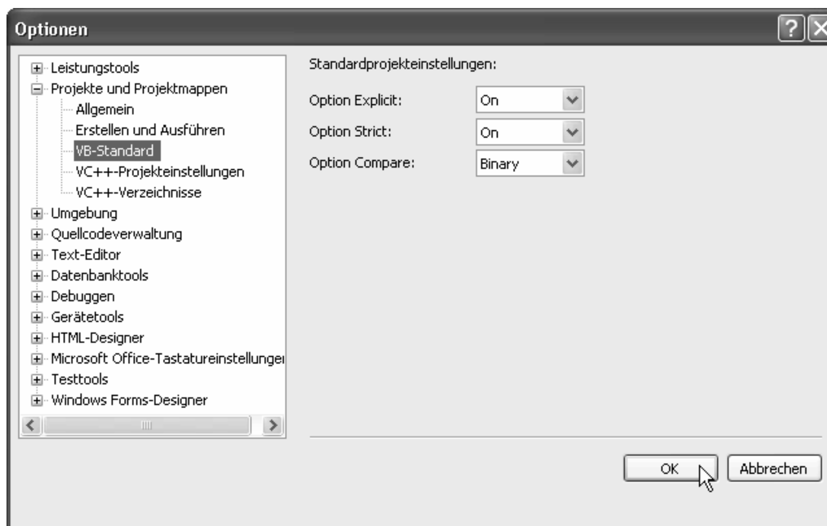

Abbildung 7.5: Mit diesem Dialog bestimmen Sie das *Option*-Verhalten für zukünftige Projekte

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.

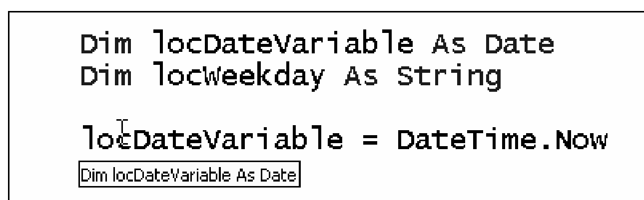

Abbildung 7.6: Ein einfaches »Daraufzeigen« mit der Maus reicht aus, um dem Codeeditor den Typ einer Variablen zu entlocken

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.[5]

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

        //  Klassenvariablen
        //
        internal int m_currentThread = InternalGetCurrentThread();
        internal int m_MaxCapacity = 0;
        internal String m_StringValue = null;
        // Statische Konstanten
        //
        internal const int DefaultCapacity = 16;
.
.
.
// Hängt ein Zeichen an das Ende dieses StringBuilder-Objektes an.
// Die Kapazität wird im Bedarfsfall angepasst.
        public StringBuilder Append(char value, int repeatCount) {
            if (repeatCount==0) {
                return this;
            }
            if (repeatCount<0) {
                throw new ArgumentOutOfRangeException("repeatCount",
                                    Environment.GetResourceString("ArgumentOutOfRange_NegativeCount"));
            }


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

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

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

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

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

[5] Die komplette CLR-Implementierung der CLI 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 Benutzersteuerelemente zu entwickeln, würde ich Englisch selbst dann vorziehen. Es ergib 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 Programmpart 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.

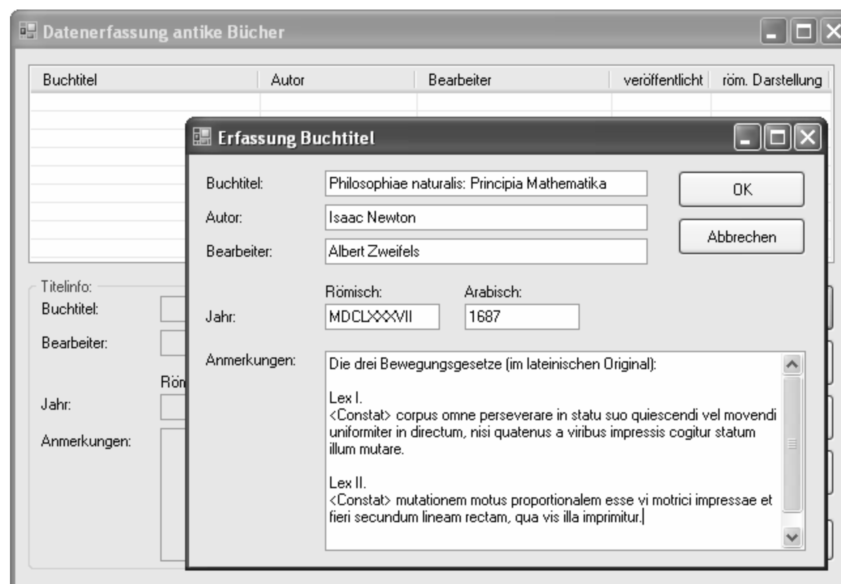

Abbildung 7.7: Das Erfassungsprogramm für antiquarische Buchtitel in Aktion

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 Projektmappen-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
```

```
        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
'auch 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 Dialogeinträge 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.